

# Introduction à React

React JS est une bibliothèque JavaScript qui permet de créer des interfaces utilisateur (UI) réactive.

Un interface réactif est un interface qui se met à jour automatiquement lors du changement de son contenu, comme le résultat d'une recherche qui s'affiche lorsque l'utilisateur tape au clavier ou encore le nombre de produits du panier qui augmente lorsque l'on clique sur + .

Il est déjà possible de créer des UI réactive en pure JS mais c'est une action qui demande beaucoup de ligne de code pour quelque chose de finalement fondamentale.

## Différence entre du JavaScript Pure et ReactJS

### JS Pure

Prenons pour exemple la mise à jour du nombre d'article d'un panier en JS pure.

*index.html*

```
<h1>Ma boutique</h1>
<div>
  <p>Adidas taille 42</p>
  <p>Nombre de produit : <span id="nb_produit"></span></p>
  <button id="add_btn">Ajouter au panier</button>
</div>
```

*script.js*

```
// Je récupère les balises
const nbProduitTag = document.querySelector("#nb_produit");
const addButton = document.querySelector("#add_btn");

// J'initialise le compteur
let compteur = 0;
// Je met à jour l'affichage du compteur dans la balise
nbProduitTag.innerText = compteur;

addButton.onclick = ()=>{
  // J'incrémente le compteur
  compteur++;
  // Je met à jour l'affichage
  nbProduitTag.innerText = compteur;
}
```

## React JS

React vous permet de créer vos propre balises HTML appelées composant .

### HTML

```
<h1>Ma boutique</h1>
<Produit />
```

script.js

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

function Produit(){
  // Je créer une variable compteur et une fonction setCompteur qui la modifie
  const [compteur,setCompteur] = useState(0); // par défaut compteur = 0

  function addProduct(){
    setCompteur(compteur+1);
  }

  // Je renvoi le HTML final de mon composant
  return (
    <div>
      <p>Adidas taille 42</p>
      <p>Nombre de produit : {compteur}</p>
      <button onClick={addProduct}>Ajouter au panier</button>
    </div>
  );
}
```

Dans le contexte d'une application complète la syntaxe React est beaucoup plus courte et à l'avantage de contenir le code HTML et JS traitant du produit au même endroit. Vous remarquerez que React met également à jour le nombre de produit de votre balise `<p>` sans jamais vous demandez d'ecrire le moindre `querySelector` , `innerText` ou `createElement` etc. Tout se met à jour grâce la fonction `setCompteur` .

***Grâce à React plus besoin de manipuler le DOM, React le fait pour vous.***

## Difficultés d'utilisation de React

React est une bibliothèque JavaScript moderne et utilise donc certaines syntaxes qui vous sont probablement étrangères.

*Je recommande la lecture de la MDN.*

### import from

La grande majorité des projets JavaScript modernes divisent leurs codes en plusieurs fichiers JS, appelées `module` , grâce à la syntaxe `import from "module"` .

Ici j'importe la fonction `useState` depuis le module JavaScript `react` .

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

J'utilise également le `destructuring assignment` qui permet d'extraire une méthode d'un objet. Le module `"react"` exporte un objet mais nous n'importons que la méthode `useState` de cet objet.

## Destructuring assignment

Plus d'info sur le `destructuring assignment` sur le guide de la MDN :

[https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Operators/Destructuring\\_assignment](https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Operators/Destructuring_assignment)

En JS je peux utiliser le `destructuring assignment` pour déclarer des variables à partir du contenu d'un array.

```
const fruits = ["pomme", "banane", "poire"];
const [apple, banana] = fruits;
```

```
console.log(apple);    // => "pomme"
console.log(fruits[0]); // => "pomme"
```

```
console.log(banana);    // => "banane"
console.log(fruits[1]);  // => "banane"
```

La fonction `useState` renvoi un array, dites vous que la syntaxe :

```
const [compteur, setCompteur] = useState(0);
```

peut également s'ecrire :

```
const stateInfos = useState(0);
const compteur = stateInfos[0];
const setCompteur = stateInfos[1];
```

Cette syntaxe est particulièrement longue, on préfère donc utiliser le `destructuring assignment`.

Vous vous habituez rapidement à ces nouvelles syntaxes et sachez quelles ne sont pas spécifiques à `React` mais au langage `JavaScript`. Apprenez les donc bien car TOUT les frameworks JS utilise, à un moment donnée, `import` ou le `destructuring assignment`.

# Pré-requis à React

## JSX, du HTML dans le JavaScript ?!

La plupart du temps en React vous allez écrire du JSX, c'est à dire un genre de HTML directement dans JavaScript.

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

function Hello(){
  const prenom = "Rémi";
  // Ceci est du JSX, du HTML dans le JavaScript.
  return <h1>Hello {prenom}</h1>;
}
```

Le JSX permet d'éviter d'écrire ceci :

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

function Hello(){
  const prenom = "Rémi";
  // Sans le JSX le code est moins clair
  return createElement("h1", {}, `Hello ${prenom}`);
}
```

Concrètement à la compilation le JSX va remplacer le code JSX par des fonctions createElement tout simplement.

Le JSX n'est pas du JavaScript il faut donc le compiler en JavaScript. Pas de panique tout ceci se configure tout seul grâce à un outil utilisé dans énormément de projet JavaScript moderne **Vite** .

## Vite

<https://vitejs.dev/>



Vite est un bundler JavaScript, concrètement c'est un programme qui va lire votre code JavaScript, JSX ou encore TypeScript et le compiler en un code optimisé pour les navigateurs.

Vite vous fournit également un serveur web local de développement donc pas besoin de WAMP, XAMPP ou autre.

Pour créer un projet react avec vite il vous faut :

1. Installer NPM (NodePackageManager)
2. Taper une commande pour créer un projet JavaScript pré configuré pour React.
3. Taper une commande pour lancer le serveur web
4. Commencez à coder 😊

Vite peut très facilement être dockerisé via une image docker de node et un port binding du port (-p).

# Créer un projet React avec Vite.

## 1. Installer NodeJS

### Windows

Téléchargez node ici : <https://nodejs.org/>

### Mac

```
brew install node
```

### Linux (Debian / Ubuntu)

```
apt install nodejs npm
```

Attention cependant apt n'utilise pas la dernière version de nodejs, pour l'avoir il rajouter le dépôt à votre machine linux avant de l'installer.

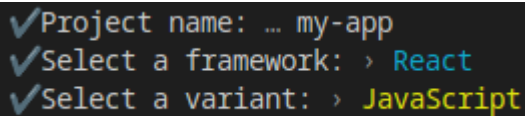
Voir les instructions d'installation ici : <https://github.com/nodesource/distributions?tab=readme-ov-file#using-ubuntu>

## 2. Créez le projet avec Vite

Dans votre dossier de travail.

```
npm create vite
```

Répondez au question de configuration comme ceci.

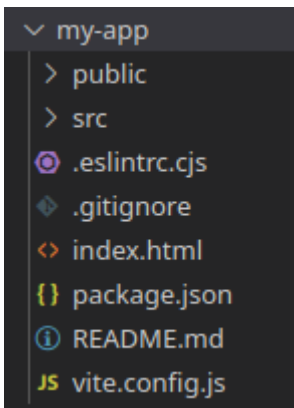


```
✓Project name: ... my-app
✓Select a framework: > React
✓Select a variant: > JavaScript
```

## 3. Ouvrez le projet dans VSCode

Vite vous à créez une application exemple dans le dossier my-app (le nom que vous avez donnez au projet).

**Ouvrez ce dossier dans VSCode.**



- **/public** , **contiendra vos images**, vidéos ou audio. Toute choses qui nécessite un accès publique, comme l'attribut `src` de la balise `<img>` par exemple, doit être dans le dossier `public` .
- **/src** , **contient tout votre code**. C'est ici que vous travaillerez la majorité du temps.
- **index.html** , **la page d'accueil du site**. C'est le point de départ de votre application.
- **.eslintrc.cjs** , est un fichier de configuration qui aide à repérer des bugs dans votre appli. Vous pouvez ignorer ce fichier.
- **.gitignore** , définit les dossiers à ignorer par git.
- **package.json** et **package-lock.json** , gère les dépendances de npm installées avec `npm install` .
- **vite.config.js** , configure `vite` pour fonctionner sans encombre avec `React` .

## 4. Lancez le serveur web localhost

Ouvrez un terminal dans votre projet VSCode et tapez :

```
npm install # Installation des dépendances du projet
npm run dev # Lancement du serveur web localhost
```

- `npm install` installe les éventuelles dépendances du projet.
- `npm run dev` **lance un serveur web** pour votre projet et vous fournit l'adresse ip du serveur, habituellement `localhost:XXXX` .

Une fois `npm install` effectuée, vous n'aurez pas besoin de la relancer la prochaine fois.

Lorsque vous voyez ce message votre projet est prêt et disponible via l'adresse localhost fournie par vite.

```
VITE v5.1.0 ready in 222 ms
→ Local: http://localhost:5174/
```



# C'est fait !

## "Ca marche pas !" En cas de soucis.

Si vous n'arrivez pas à faire fonctionner les commandes ci-dessus, voici le résumé, vous pouvez les copier-coller. 😊

Dans votre dossier de travail ouvrez un terminal et tapez :

```
npm create vite@latest my-app -- --template react
cd my-app
npm install
npm run dev
```

## Un premier pas sur React

React s'occupe exclusivement de l'UI et pour se faire il va créer un DOM virtuel qu'il copiera dans le véritable DOM de votre application quand un rafraichissement est neccessaires.

React à donc besoin d'un point d'entrée dans votre page html pour fonctionner, une balise d'origine dans laquelle toute l'arborescence de l'application React se trouvera.

## Hello World

1. Supprimer tout le contenu du dossier `/src` .
2. Dans le dossier `/src` , créez un fichier nommée `main.jsx` , ce fichier est le point d'entrée de notre application.
3. Dans le fichier `src/main.jsx` écrivez un hello world.

*/src/main.jsx*

```
console.log("Hello Main");
```

4. Dans le fichier */index.html* vérifiez que le fichier */src/main.jsx* est bien importé.

*/index.html*

```
<!doctype html>
<html lang="fr">
  <head>
    <meta charset="UTF-8" />
    <meta name="viewport" content="width=device-width, initial-scale=1.0" />
    <title>My App</title>
  </head>
  <body>
    <div id="root"></div>
    <!-- J'importe main.jsx dans le html -->
    <script type="module" src="/src/main.jsx"></script>
  </body>
</html>
```

5. Ouvrez la console du navigateur, vous devriez y voir le message "Hello Main" .

### ***Extension .jsx dans la balise <script>***

```
<script type="module" src="/src/main.jsx"></script>
```

.jsx est une extension inconnue des navigateurs web mais vite compile notre code en JavaScript à chaque sauvegarde donc tout va bien. Au final le code source sera importé sera en .js . 😊

## **Votre premier composant React.**

Tout fonctionne, il est temps de créer notre premier composant React.

Un composant est la brique élémentaire de React, vous pouvez imaginer les composants React comme des nouvelles balises HTML sur mesure que vous codez en JavaScript (plus précisément en jsx).

1. Vérifier la présence d'une balise <div> avec pour id root dans le index.html . C'est dans cette balise que React place les composants que vous codez.

/index.html

```
<!doctype html>
<html lang="fr">
  <head>
    <meta charset="UTF-8" />
    <meta name="viewport" content="width=device-width, initial-scale=1.0" />
    <title>My App</title>
  </head>
  <body>
    <!-- Une balise vide avec pour id root -->
    <!-- C'est cette balise que React remplira avec les composants que vous coderez -->
    <div id="root"></div>
    <script type="module" src="/src/main.jsx"></script>
  </body>
</html>
```

Dans le fichier `/src/main.jsx`.

2. Utilisez la méthode `ReactDOM.createRoot()` pour créer un composant React lié à la balise `div.root`.

`/src/main.jsx`

```
import React from "react";
import ReactDOM from "react-dom/client";

const root = ReactDOM.createRoot(document.querySelector("#root"));
```

3. Affichez le composant avec la méthode `render`.

`/src/main.jsx`

```
import React from "react";
import ReactDOM from "react-dom/client";

const root = ReactDOM.createRoot(document.querySelector("#root"));
/* Affichage du composant root */
root.render(<h1>Hello World</h1>);
```

Ce code affiche(render) le composant `root` dans la balise `#root`. Le contenu du composant `root` est un simple `h1 Hello World`.

# Hello World

Ce code est le point de départ de tout projet React. Il vous paraîtra plus clair plus tard mais vous n'avez pour l'instant pas d'intérêt à comprendre le processus plus que les explications fournies plus haut.

## En résumé

1. La méthode `ReactDOM.createRoot` permet de créer un composant React lié à une véritable balise HTML.
2. La méthode `render` permet d'afficher le composant.
3. La méthode `render` prend en paramètre du JSX, soit le HTML à afficher.

C'est la seule fois où nous utiliserons `createRoot` pour créer un élément React toute la suite ce sera en JSX.

## L'encapsulation des composants

Pour l'instant toute votre application est un simple `<h1>`, évidemment le code va grossir et nous voulons donc encapsuler tout notre code React dans un composant appelé `<App />`.

1. Dans le fichier `main.jsx` créez une fonction `App()` qui renvoie un `<h1>` tant que JSX.

*src/main.jsx*

```
// à la suite du code précédent...  
  
function App(){  
  return <h1> Hello World</h1>;  
}
```

Ceci est un composant fonctionnel, c'est à dire un composant React créé dans une fonction. C'est sous cette forme que la majorité de votre code React sera écrit.

2. Affichez le composant `App` dans la fonction `root.render()` .

```
import React from "react";
import ReactDOM from "react-dom/client";

const root = ReactDOM.createRoot(document.querySelector("#root"));

root.render(<App/>); // Afficher le nouveau composant App

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

Le composant nouvellement crée est affichez en tant que point de départ de notre appli dans la balise `#root`.

Pour améliorer la clareté du code, JSX nous permet d'écrire le composant `App` comme du HTML.

```
root.render(<App/>);
```

En réalité le code suivant donnerait le même résultat.

```
root.render(App()); // Même résultat mais moins explicite.
```

## Syntaxe du JSX

Le JSX permet d'écrire du pseudo HTML contenant du HTML classique et les nouveaux composants fonctionels que vous créez. Sa syntaxe est très simple.

De base JSX permet s'implement d'ecrire du bon vieux HTML.

```
function App(){
  return <p>Je suis un paragraphe</p>;
}
```

Si vous avez besoin d'ecrire des **balises** sur **plusieurs lignes** il faut mettre tout votre **JSX** dans une **balise div et entre parenthèses**.

```
function App(){
  return (
    <div>
      <h1>Titre</h1>
      
      <p>Je suis un paragraphe</p>
    </div>
  );
}
```

Un composant ne peut renvoyer au final qu'une seule et unique balise voilà pourquoi j'encapsule le tout dans une balise `<div>` .

**Attention !** Remarquez que la balise `<img />` est une balise orpheline en JSX il **OBLIGATOIRE** de finir les balises orphelines par un slash `/` .

## Executer du JavaScript dans du JSX

JSX permet d'exécuter du JavaScript dans le JSX, pour se faire il faut placer le JS entre accolades.

Ici j'affiche une variable dans mon JSX.

```
function App(){
  const name = "Massinissa";
  return <h1>Bonjour {name}</h1>;
}
```

Je peux mettre n'importe quel JavaScript entre les accolades même si, pour des raisons de clarté, on préfère effectuer le maximum de traitement JS en dehors du JSX.

```
function App(){
  const name = "Massinissa";
  const birth = new Date("12-31-1999");
  return <h1>Bonjour {name} tu as {2024 - birth.getFullYear()} ans</h1>;
}
```

On préfère extraire la logique du JSX comme ceci.

```
function App(){
  const name = "Massinissa";
  const birth = new Date("12-31-1999");
  const age = 2024 - birth.getFullYear();
  return <h1>Bonjour {name} tu as {age} ans</h1>;
}
```

## Afficher le contenu d'un array JavaScript dans du JSX

Un grand classique de la manipulation de DOM c'est l'affichage du contenu d'un array via une boucle for.

Avec JSX, on peut simplement passer un array entre accolades pour qu'il affiche tout le contenu de l'array.

Prenons un tableau d'élèves à afficher dans une liste.

```
function App(){
  const eleves = ["Mathieu", "Arnaud", "Cléo", "François"];
  return (
    <div>
      <h1>La liste des élèves</h1>
      {eleves}
    </div>
  );
}
```

## La liste des élèves

MathieuArnaudCléoFrançois

Pour chaque élève il faut créer une balise `<li>` contenant le nom. Cette action est réalisable facilement avec un `map()`.

```
function App(){
  const eleves = ["Mathieu","Arnaud","Cléo","François"];
  const elevesElements = eleves.map( eleve => <li>{eleve}</li> );
  return (
    <div>
      <h1>La liste des élèves</h1>
      <ul>
        {elevesElements}
      </ul>
    </div>
  );
}
```

## La liste des élèves

- Mathieu
- Arnaud
- Cléo
- François

### Rappel `Array.prototype.map()`

La méthode `Array.prototype.map()` permet de transformer un tableau en un nouveau tableau composés des valeurs de retour de la fonction callback passée en paramètre. Si vous devez modifier tout le contenu d'un tableau sans en changer le nombre d'éléments c'est un `map()` qu'il vous faut.

[https://developer.mozilla.org/fr/docs/Web/JavaScript/Reference/Global\\_Objects/Array/map](https://developer.mozilla.org/fr/docs/Web/JavaScript/Reference/Global_Objects/Array/map)

La constante `elevesElements` est un array de composant React, le JSX ne fait que parcourir le tableau et afficher chaque composants dans l'ordre.

*Mapper* est array JavaScript en un nouveau array de composants est une action très commune en React. En situation réel ce tableau d'élèves pourrait très bien venir d'une base de données ou d'une API par exemple.

On peut d'ailleurs supprimer la constante `elevesElements` et placer le `map()` dans le JSX directement.



```
function App(){
  const eleves = ["Mathieu","Arnaud","Cléo","François"];
  return (
    <div>
      <h1>La liste des élèves</h1>
      <ul>
        {eleves.map( eleve => <li>{eleve}</li> )}
      </ul>
    </div>
  );
}
```

## Exercice 1 - La moyenne de Maxime

Affichez la liste des notes de Maxime et la moyenne de ses notes.

**Astuce** : vous pouvez utiliser la fonction **Array.prototype.reduce()**.

```
function App(){
  const notes = [20,10,11,7,13];
  return (
    <div>
      <h1>La moyenne de Maxime </h1>
      <h2>Notes :</h2>
      <ul>

      </ul>
      <p>Moyenne générale : </p>
    </div>
  );
}
```

# La moyenne de Maxime

## Notes :

- 20 / 20
- 10 / 20
- 11 / 20
- 7 / 20
- 13 / 20

Moyenne générale : 12.2

***Attention correction juste en dessous !***

## Correction Exercice 1 - La moyenne de Maxime

```
function App(){
  const notes = [20,10,11,7,13];
  const average = notes.reduce((sum,note)=>sum+note) / notes.length;
  return (
    <div>
      <h1>La moyenne de Maxime </h1>
      <h2>Notes :</h2>
      <ul>
        {notes.map( note => <li>{note} / 20</li> )}
      </ul>
      <p>Moyenne générale : {average}</p>
    </div>
  );
}
```

`Array.prototype.reduce()` est à utiliser dans le cas où vous voulez récupérer une seule valeur à partir d'un tableau. Ici par exemple à partir du tableau de notes je veux en récupérer la somme.

Le même résultat est obtainable à partir d'une variable tampon et une boucle `for` classique sur le tableau de notes.

```

function App(){
  const notes = [20,10,11,7,13];
  // -- Equivalent de la fonction reduce ---
  let sum = 0; // la variable tampon qui contient la somme des notes
  for (let i = 0; i < notes.length; i++) {
    sum+= notes[i];
  }
  // -----
  const average = sum/notes.length;
  return (
    <div>
      <h1>La moyenne de Maxime </h1>
      <h2>Notes :</h2>
      <ul>
        {notes.map( note => <li>{note} / 20</li> )}
      </ul>
      <p>Moyenne générale : {average}</p>
    </div>
  );
}

```

Les méthodes map, reduce et filter sont à connaître sur le bout des doigts pour écrire du JavaScript plus concis.

## Exercice 2 - La liste des élèves

Affichez la liste des élèves, pour chaque élève, affichez son prénom et sa classe.

```

function App(){
  let eleves = [
    { name : "Massinissa", grade : "4eme" },
    { name : "Arnaud", grade : "3eme" },
    { name : "Cléo", grade : "3eme" },
    { name : "Louis", grade : "6eme" },
  ];
  return (
    <div>
      // Codez ici ...
    </div>
  );
}

```

**Massinissa**

4eme

**Arnaud**

3eme

**Cléo**

3eme

**Louis**

6eme

*Attention correction juste en dessous !*

## Correction Exercice 2 - La liste des élèves

```
function App(){
  let eleves = [
    { name : "Massinissa", grade : "4eme" },
    { name : "Arnaud", grade : "3eme" },
    { name : "Cléo", grade : "3eme" },
    { name : "Louis", grade : "6eme" },
  ];
  const elevesElements = eleves.map((eleve)=>{
    return (
      <div>
        <h2>{eleve.name}</h2>
        <p>{eleve.grade}</p>
      </div>
    );
  });
  return (
    <div>
      {elevesElements}
    </div>
  );
}
```

## Les composants


La conception d'une application React commence toujours par le découpage de la maquette en composants.


Une application est composée de plusieurs blocs imbriqués les uns dans les autres. Le développement d'une application avec React consiste à la création de tout ces blocs pour pouvoir, au final, assembler toute l'application.


Ces briques d'éléments graphique s'appelle des composants et pour le moment vous n'avez crée qu'un seul composant : le composant racine `<App/>` .


Les composants étant imbriqué il sont organisés en arborescence. Le composant App est le composant racine, il est obligatoire.


Prenons par exemple la maquette d'un pokedex. Selon vous, combien contient elle de composants **différents** ?


1Bulbizarre


2Herbizarre


3Florizarre


idPokemon name


idPokemon name


idPokemon name


idPokemon name

idPokemon name

idPokemon name


idPokemon name

idPokemon name

idPokemon name



Q

n°1




Bulbizarre

Types



Evolution

2Herbizarre

J'en compte 4 différents.

Le composant <Pokemon> qui est utilisé dans la liste de pokemons et pour indiquer l'évolution du pokemon.

1BulbizarrePokemon

2Herbizarre

3Florizarre

idPokemon name

idPokemon name

idPokemon name

idPokemon name

idPokemon name

idPokemon name


idPokemon name

idPokemon name

idPokemon name



idPokemon name

n°1




Bulbizarre

Types



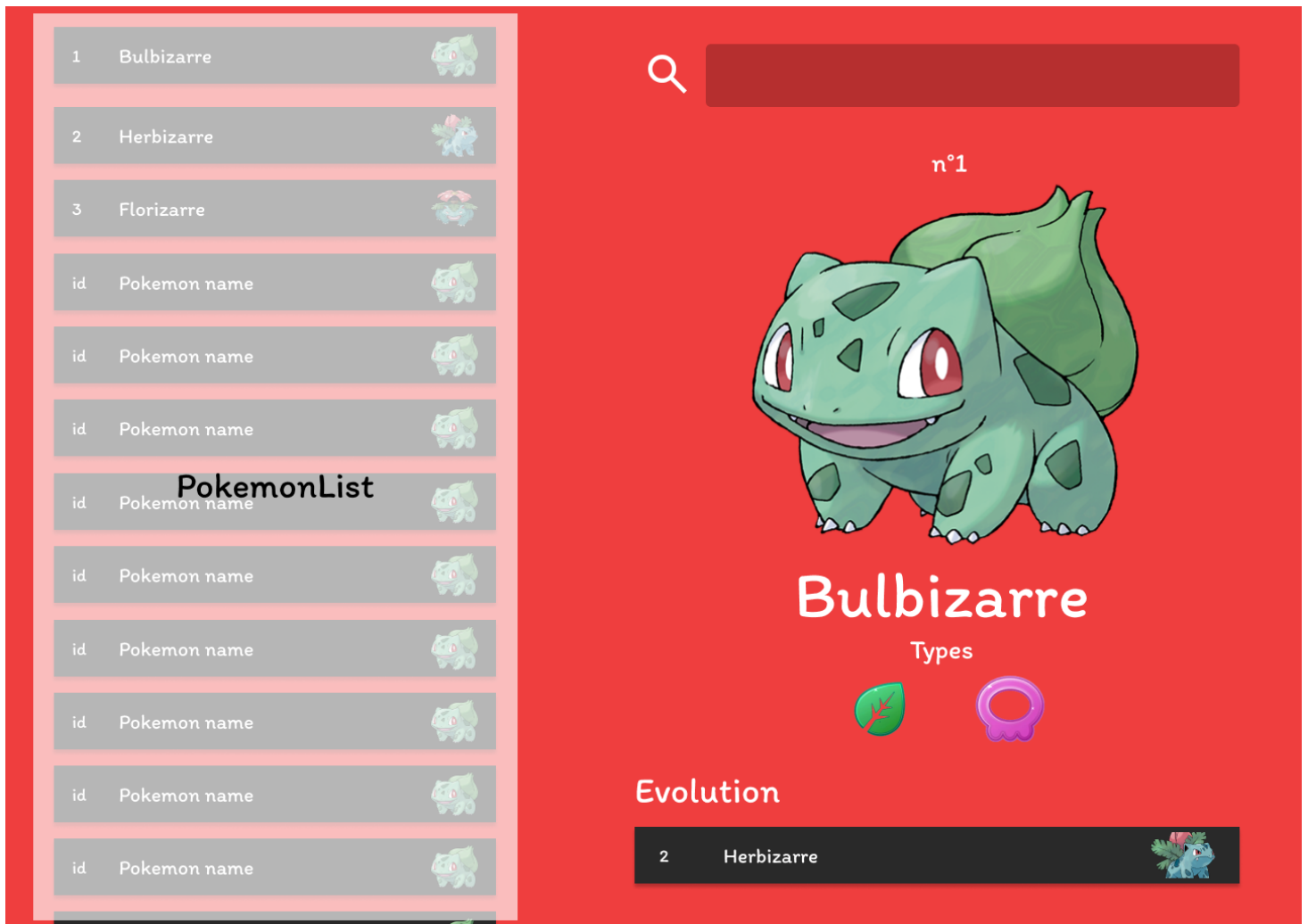
Evolution

2Herbizarre

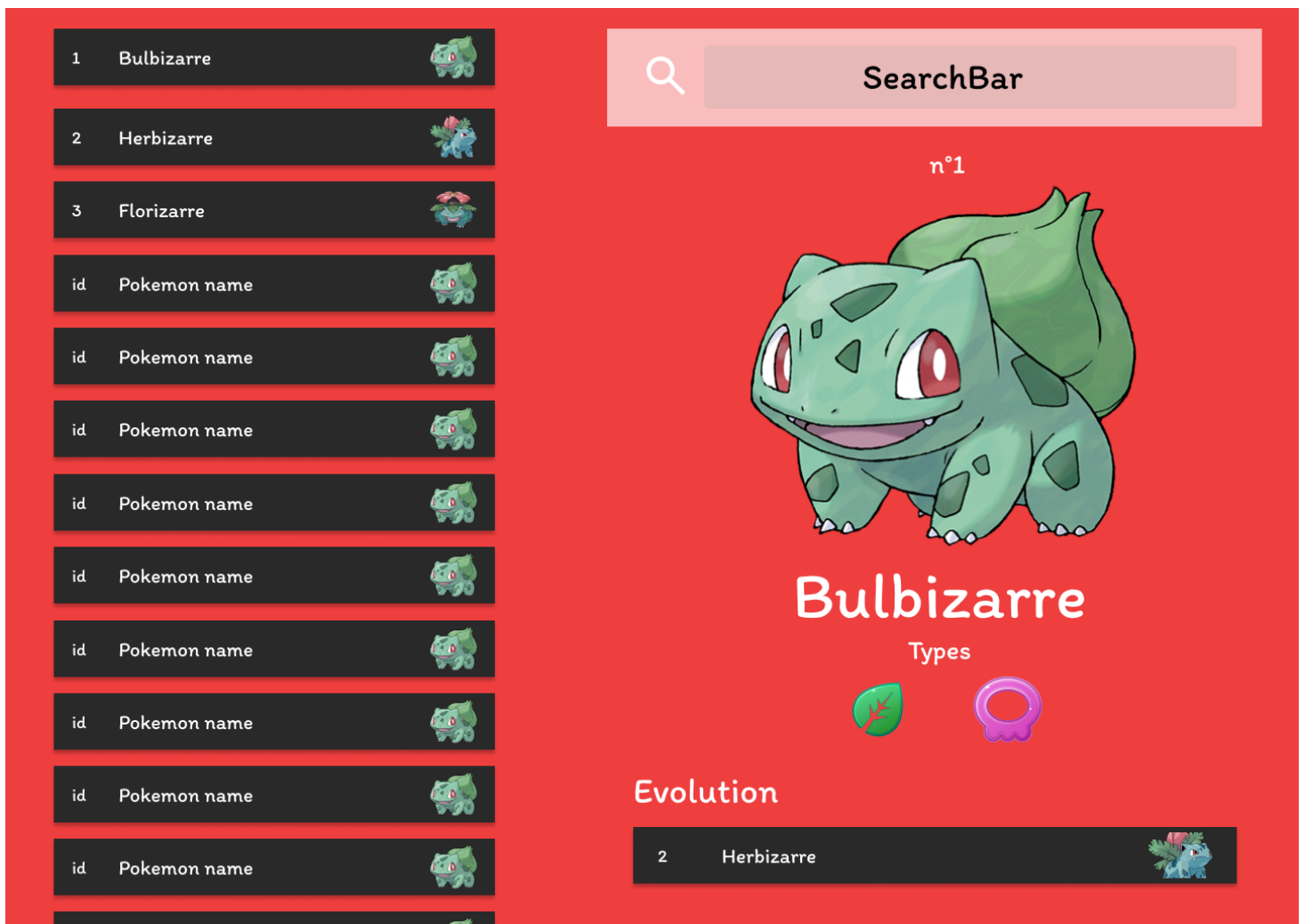


Le composant <PokemonList> qui liste tout les pokemons existant dans un menu défilable.

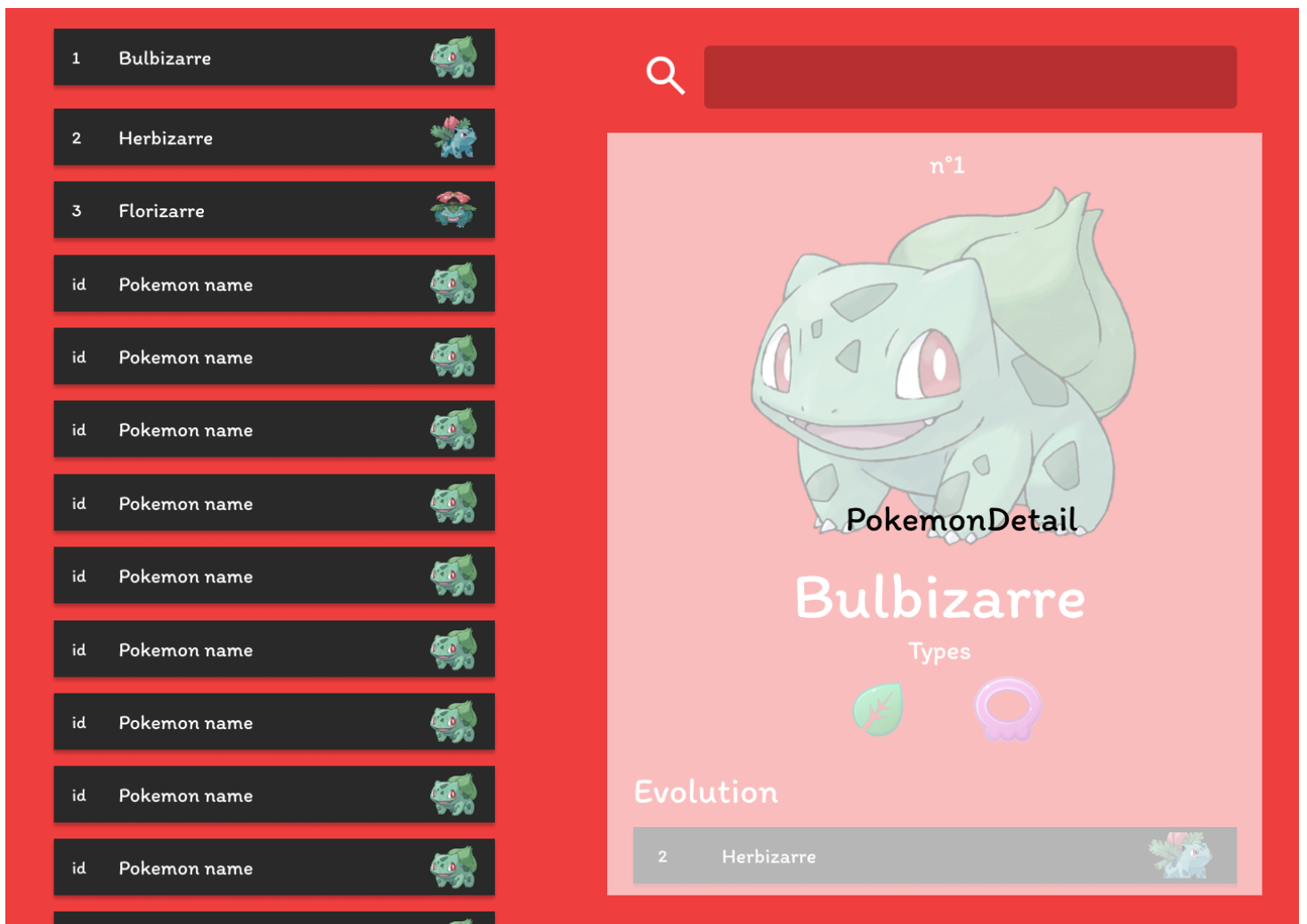




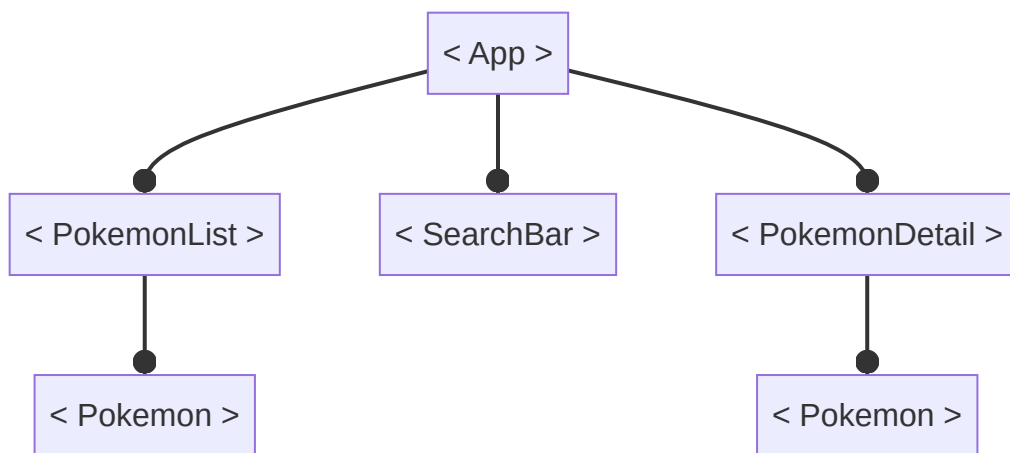
Le composant `<SearchBar>` qui permet de rechercher un pokemon.

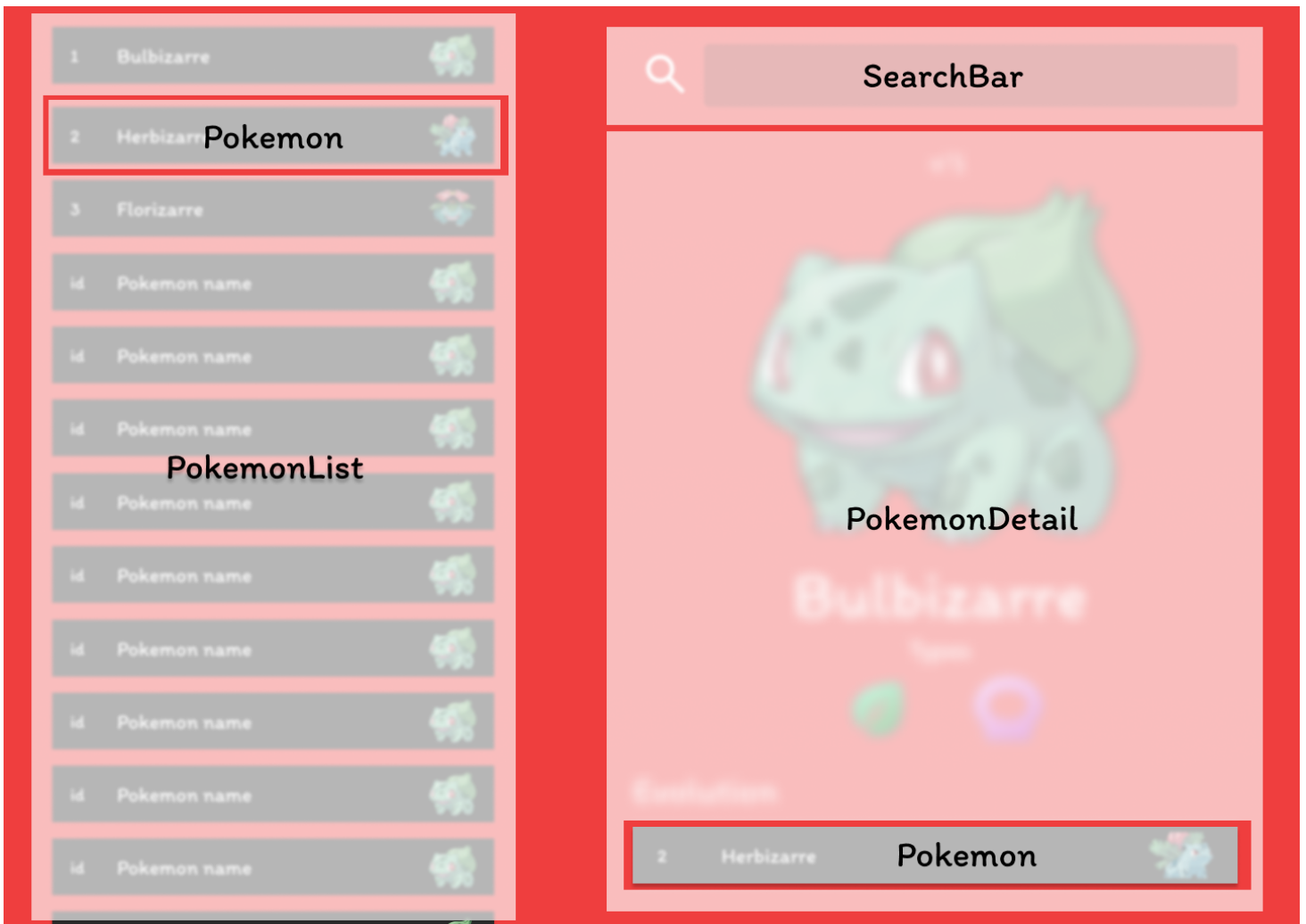


Le composant `<PokemonDetail>` qui affiche les informations d'un pokemon (nom, id, image, types et évolution) lorsque que l'utilisateur clique sur un `<Pokemon>` ou recherche un pokemon avec la `<SearchBar>` .



Une fois les différents composant identifiés ont remarque une arborescence qui se dessine.





## Créer un composant

La création d'un composant doit se faire dans un fichier à part qui porte le nom du composant.

Placez le composant racine `<App>` dans un autre fichier nommé `App.jsx`.

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

Exportez le avec le mot-clé `export`.

```
export function App(){
  return (
    <h1>Hello App</h1>
  )
}
```

Importez la fonction App dans le fichier main.jsx.

```
import React from "react";
import ReactDOM from "react-dom/client";

import {App} from "./App";      // Import du composant App

const root = ReactDOM.createRoot(document.querySelector("#root"));
root.render(<App/>);              // Appel du composant App
```

Au minimum un composant est une fonction qui renvoi du JSX.

```
function NomDuComposant(){
  /**
   * Code éventuel
   * */
  return (
    // JSX
  );
}
```

```
function Eleve()
{
  const prenom = "Massinissa";
  const nom = "CHAOUCHI";
  const age = 24;
  return (
    <div>
      <h2>{nom}</h2>
      <p>{prenom}</p>
      <p>{age} ans</p>
    </div>
  );
}
```

# Affichage conditionnel

Avec un opérateur ternaire il est possible d'afficher du JSX sous condition. C'est une opération très commune pour, par exemple, afficher le stock d'un produit ou un bouton "se connecter" si l'utilisateur n'est pas connecté.

/src/eleve.jsx

```
export function Eleve()
{
  const prenom = "Massinissa";
  const nom = "CHAOUCHI";
  const age = 24;
  return (
    <div>
      <h2>{nom}</h2>
      <p>{prenom}</p>
      <p>{age} ans</p>
      <p>{ age >= 18 ? "Majeur" : "Mineur" }</p>
    </div>
  );
}
```

# Props

Les props sont les attributs paramétrables des composants JSX que vous créez.

L'intérêt des composants est la modularité, un composant à pour objectif d'être réutilisé dans des contextes différents et se doit donc d'être paramétrable.

Les props d'un composant sont définies en paramètre de sa fonction et ont définies leurs valeurs à l'appel du composant à la manière du HTML.

```
export function Eleve({prenom, nom, age})
{
  return (
    <div>
      <h2>{nom}</h2>
      <p>{prenom}</p>
      <p>{age} ans</p>
      <p>{ age>=18 ? "Majeur" : "Mineur" }</p>
    </div>
  );
}
```

```
import {Eleve} from "./Eleve";
export function App(){
  return (
    <Eleve prenom="Massinissa" nom="CHAOUCHI" age={24}/>
    <Eleve prenom="Louis" nom="BERGER" age={25}/>
  );
}
```

Le JSX utilise une syntaxe proche de celle des attributs HTML mais en réalité le JSX passe simplement un objet en paramètre de la fonction.

Ces deux lignes produisent le même résultat.

```
<Eleve prenom="Massinissa" nom="CHAOUCHI" age={24}/>
Eleve({
  prenom : "Massinissa",
  nom : "CHAOUCHI",
  age : 24
});
```

Etant donné que la fonction prend un objet en paramètre il est faut utiliser le destructuring assingement pour accéder au props dans la fonction.

```
export function Eleve({prenom, nom, age})
```

# Event

Pour réagir à un événement React vous fournit tout un ensemble de props auquel on affecte une fonction callback.

```
export function Clicker(){
  function handleClick(){
    alert("Click !");
  }
  return <button onClick={handleClick}>Click me !</button>
}
```

Je dois placer ma fonction entre accolades pour que JSX l'interprète comme du JavaScript.

Le nom des props est analogue au nom des événements HTML, pré-fixé d'un `on`, le tout en *camelCase*.

Voir tout les événements possible : [https://www.w3schools.com/jsref/dom\\_obj\\_event.asp](https://www.w3schools.com/jsref/dom_obj_event.asp)

# State

Les states sont les données dynamique que le composant affiche. Quand vous changez la valeur d'un state, React va rafraichir l'affichage du composant.

La création d'un state se fait grâce à la fonction `useState()` du module `react`. Il faut donc l'importer dans le fichier du composant.

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

Un state est composé de deux éléments :

- une constante pour lire le state
- une fonction pour modifier le state

La plupart du temps les states seront modifiés lors d'un événement.

Par exemple ici j'incrmente un compteur quand je clique sur un bouton.



```
import { useState } from "react";
export function Counter(){
  /**
   * Je crée un nouveau state.
   * J'accède au state via la variable compteur.
   * Je modifie le state via la fonction setCompteur
   * */
  const [compteur, setCompteur] = useState(0);

  function handleClick(){
    setCompteur(compteur+1);
  }

  return (
    <button onClick={handleClick}>{compteur}</button>
  );
}
```

## Initialiser un state

La valeur de départ du state est passée en paramètre de la fonction `useState` .

```
// Le compteur est initialisé à la valeur 0.
const [compteur, setCompteur] = useState(0);
```

Cette valeur de départ peut être de n'importe quel type JavaScript.

```
const [compteur, setCompteur] = useState(0);    // number
const [product, setProduct] = useState(new Product(...));    // object
const [comments, setComments] = useState([]);    // array
```

## Modifier un state

La modification d'un *state* se fait via la fonction setter. `useState` vous passe la référence de cette fonction et vous choisissez son nom lors du *destructuring assignment*.

```
// La reference de la fonction setter est placée dans la constante setCompteur.
const [compteur, setCompteur] = useState(0);
```

Le cas le plus commun de la modification d'un state est en réaction à un événement : clic, recherche, entrées clavier.

Par exemple ici je raconte une "blague à papa" quand je clic sur un bouton.

```
import { useState } from "react";
export function DadJoke(){
  // J'initialise la blague
  const [joke, setJoke] = useState("Attends je réfléchis...");

  function tellJoke(){
    const headers = new Headers();
    headers.append("Accept", "text/plain");

    fetch("https://icanhazdadjoke.com", { headers })
      .then(res => res.text())
      .then(joke => setJoke(joke)); // Je modifie la blague
  }

  return (
    <div>
      <p>{joke}</p>
      <button onClick={tellJoke}>Dire la blague</button>
    </div>
  );
}
```

Il est interdit de modifier directement la constante `compteur` car c'est la méthode `setCompteur` qui va déclencher le rafraichissement de la page.

C'est d'ailleurs pour ça que l'on précise que `compteur` est une constante et non une variable via le mot clé `const`.

Attention également `setState` vient remplacer complètement l'ancienne valeur de state et ne fait rien du tout si la nouvelle valeur est égal à l'ancienne. Le problème c'est que la référence d'un array ou d'un objet, même après modification, ne change jamais; un objet est une référence et même en modifiant le contenu, la référence reste la même et donc pour React rien n'a changé.

La solution est de toujours fournir une copie de l'objet via l'opérateur `...` qui permet de cloner un objet dans un autre.

*/src/App.jsx*

```

export function App(){
  return (
    <div>
      <h1>Hello App</h1>
      <UserProfil name="Manu" lastName="CHAO"/>
    </div>
  );
}

```

*/src/UserProfil.jsx*

```

import { useState } from "react";
export function UserProfil({name, lastName}){
  // J'initialise la blague
  const [user, setUser] = useState({name, lastName});

  function onChangeName(event){
    const newName = event.target.value;
    setUser({...user, name : newName});
  }
  function onChangeLastName(event){
    const newLastName = event.target.value;
    setUser({...user, lastName : newLastName});
  }

  return (
    <div>
      <h2>{user.name}</h2>
      <h2>{user.lastName}</h2>
      <input onChange={onChangeName} value={user.name}/>
      <input onChange={onChangeLastName} value={user.lastName}/>
    </div>
  );
}

```

## Boucle infini avec un setState

setState déclenche un nouveau rendu du composant.

Si vous utilisez un setState sans aucune condition (if, event) une boucle infini va se produire car

`setState` va déclencher un nouveau rendu se qui va appeler `setState` se qui va déclencher un nouveau rendu, etc.

Par exemple si vous voulez effectuer un fetch dès le debut de vie de votre composant sans attendre le clic sur le bouton. L'erreur serait d'appeler `tellJoke` directement dans la fonction `DadJoke`.

```
import { useState } from "react";
export function DadJoke(){
  const [joke, setJoke] = useState("Attends je réfléchis...");

  function tellJoke(){
    const headers = new Headers();
    headers.append("Accept", "text/plain");

    fetch("https://icanhazdadjoke.com",{ headers })
      .then(res => res.text())
      .then(joke => setJoke(joke));
  }

  // ATTENTION !
  // J'appelle tellJoke pour espérer afficher une blague à l'initialisation du composant
  tellJoke();
  return (
    <div>
      <p>{joke}</p>
      <button onClick={tellJoke}>Dire la blague</button>
    </div>
  );
}
```

Un state doit toujours être modifié de façon conditionnel, le plus souvent, lors d'un événement utilisateur. Le rendu ne doit jamais appeler `setState` par défaut.

## useEffect, initialiser un composant

`useEffect` est une fonction très puissante qui permet d'exécuter du code uniquement lorsque certains states changent. **Egalement `useEffect` s'exécute toujours une fois avant le premier rendu.**

Si vous n'avez vraiment pas le choix et que l'ajout d'une prop est inévitable, la fonction `useEffect` est une solution, cependant notez bien que les développeurs de React conseillent de ne

l'utiliser quand dernier recours et que React à été pensez pour initialiser les composants grâce aux props.

`useEffect` prend deux paramètres :

- la fonction callback à exécuter,
- un tableau des *states* qui déclenche la fonction callback lors de leurs changement.

Dans notre cas nous ne souhaitons pas voir `useEffect` se déclencher en fonction d'un state mais uniquement lors du premier rendu, il faut donc passer en deuxième paramètre un tableau vide.

```
useEffect(()=>{  
    tellJoke();  
}, []);
```

Attention à ne pas oublier le tableau en second paramètre, sinon `useEffect` s'exécute sans condition et donc à l'infini.