



React

▼ Version	V1
☰ Type	Technique
📅 Date de création	@13 juin 2023 13:30
📅 Dernière modification	@14 décembre 2023 16:08

 [Tutoriel React](#)

 [Penser React](#)



Créer un composant

 [Documentation officielle chapitre "premier composant"](#)

Chaque composant React est une fonction JavaScript qui peut contenir du balisage HTML que React affiche dans le navigateur. Les composants React utilisent une extension de syntaxe appelée JSX pour représenter ce balisage.

JSX ressemble beaucoup à HTML, mais il est un peu plus strict et peut afficher des informations dynamiques.



Cette syntaxe qui combine JavaScript et HTML permet définir la structure et l'apparence des composants. Cela facilite la création de composants réutilisables et permet une meilleure gestion des données et des événements.



Dans ce chapitre nous créerons une liste des employés destinés à être affichée sur différentes pages d'un site.

▼ 1. Déclarer un composant React

Le préfixe d'exportation par défaut `export default function` est une syntaxe JavaScript standard (non spécifique à React). L'export permet de marquer la fonction principale dans un fichier afin de pouvoir l'importer ultérieurement à partir d'autres fichiers.

⚠ Les composants React sont des fonctions JavaScript classiques, mais **leurs noms doivent commencer par une majuscule**, sinon ils ne fonctionneront pas !

▼ 2. Ajouter du balisage

Pour que le composant `EmployeeList` affiche la liste des employés il faut ajouter le markup nécessaire.

Le markup HTML doit être contenu entre parenthèses et retourné par le composant React `return (<div>...</div>);`

⚠ Un composant React ne doit retourner qu'une seule balise parent.

```
// Ce code produit une erreur
// le composant retourne deux balises HTML
```

▼ Exemple déclaration composant React en Javascript ES5 (ancien mais toujours valable)

```
export default function EmployeeList() {
  // à venir
};
```

▼ Exemple déclaration composant React en Javascript ES6 (syntaxe la plus récente)

```
const EmployeeList = () => {
  // à venir
};

export default EmployeeList;
```

▼ Exemple en Javascript ES5

```
export default function EmployeeList() {
  return (
    <ul>
      <li>Emma</li>
      <li>Liam</li>
      <li>Sophia</li>
      <li>Noah</li>
    </ul>
  );
}
```

```

return (
  <p>Mon premier paragraph</p>
  <p>Mon second paragraph</p>
)

// Ce code ne produit pas d'erreur
// le composant retourne une balise HTML
return (
  <div>
    <p>Mon premier paragraph</p>
    <p>Mon second paragraph</p>
  </div>
)

```

Dans le cas où vous ne souhaitez pas utiliser une `<div>` pour contenir les deux paragraphes, React met à disposition les "Fragments". Les Fragments sont une solution pensée pour pallier le fait qu'un composant ne puisse retourner qu'une seule balise parent.

```

// Ce code ne produit pas d'erreur
return (
  <>
    <p>Mon premier paragraph</p>
    <p>Mon second paragraph</p>
  </>
)

```

```

)
};

```

▼ Exemple en Javascript ES6

```

const EmployeeList = () => {
  return (
    <ul>
      <li>Emma</li>
      <li>Liam</li>
      <li>Sophia</li>
      <li>Noah</li>
    </ul>
  )
};

export default EmployeeList;

```

▼ 3. Utiliser le composant

Imaginons que le composant `EmployeeList` précédemment créé doit être affiché sur la page d'accueil et la page de recrutement

Le composant `EmployeeList` peut être intégré dans du HTML classique, dans une balise auto-fermante. Cela aura pour résultat "d'injecter" le code présent dans le composant `EmployeeList` au sein de la page.

Nous venons donc de créer un composant réutilisable qui peut être intégré à différentes parties de notre interface utilisateur, en toute autonomie.

▼ Exemple d'utilisation dans le composant Homepage

```
export default function Homepage() {
  return (
    <h1>Bienvenue chez Magle corp</h1>
    ...
    <section>
      <h3>Présentation de l'équipe technique</h3>
      <EmployeeList />
    </section>
  )
}
```

▼ Exemple d'utilisation dans le composant RecruitmentPage

```
export default function RecruitmentPage() {
  return (
    <h1>Magle corp recrute</h1>
    ...
    <section>
      <h3>Rejoignez l'équipe technique</h3>
      <EmployeeList />
    </section>
  )
}
```

▼ Rendu final dans le DOM du navigateur

```
export default function RecruitmentPage() {  
  return (  
    <h1>Magle corp recrute</h1>  
    ...  
    <section>  
      <h3>Rejoignez l'équipe technique</h3>  
      <ul>  
        <li>Emma</li>  
        <li>Liam</li>  
        <li>Sophia</li>  
        <li>Noah</li>  
      </ul>  
    </section>  
  )  
}
```

Import et export d'un composant

 [Documentation officielle chapitre "import et export des composants"](#)

Deux possibilités sont envisageables pour l'import / export de vos composants React.

Les deux sont tout aussi valables, la méthode que vous choisissez dépend de la manière dont vous souhaitez importer les fonctions dans votre code.

▼ 1. Les exports nommés

```
// En ES5
function Demo() {
  ...
}

export { Demo }

// En ES6
const Demo = () => {
  ...
}

export { Demo }
```

Puis pour importer

```
import { Demo } from '...';
```

▼ 2. Les exports par défaut

```
// En ES5
export default function Demo() {
  ...
}

// En ES6
const Demo = () => {
  ...
}

export default Demo;
```

Puis pour importer

```
import Demo from '...';
```

Javascript dans JSX

 [Documentation officielle chapitre "Javascript dans le JSX"](#)

JSX permet d'écrire un balisage de type HTML dans un fichier JavaScript, en gardant la logique de rendu et le contenu au même endroit.

Parfois, vous voudrez ajouter un peu de logique JavaScript ou référencer une propriété dynamique à l'intérieur de ce balisage. Dans cette situation, vous pouvez utiliser des accolades dans votre JSX pour ouvrir une fenêtre vers JavaScript.

 Lorsque l'on ouvre des accolades dans le HTML il est possible d'exécuter du code Javascript, quel qu'il soit.

```
export default function Avatar() {  
  return (  
    <>  
        
      <p>Hello Maglecorp</p>  
    </>  
  );  
}
```

Nous pouvons d'ores et déjà stocker les valeurs contenues dans les attributs `src` et `alt` à l'intérieur de variables puis les utiliser dans la balise ``.

Nous pouvons aussi stocker le nom de l'utilisateur à l'intérieur d'une variable pour l'utilisateur dans la balise `<p></p>`.

```

export default function Avatar() {
  const avatarSrc = "https://aws.image.avatar.maglecorp.png";
  const avatarAlt = "Maglecorp";
  const username = "Maglecorp";


  return (
    <>
      <img
        className="avatar"
        src={avatarSrc}
        alt={avatarAlt}
      />
      <p>Hello {username}</p>
    </>
  );
}

```


Afficher une liste

 [Document officielle chapitre “afficher une liste”.](#)

Comme indiqué au chapitre “Javascript dans JSX” il est possible d'utiliser du Javascript au sein du HTML grâce aux accolades `{...}`.

Pour afficher une liste de prénom nous utiliserons donc la méthode Javascript native `map`  [document fonction](#).

La fonction `map` parcourra tout le tableau et pour chaque item un `` contenant un prénom sera affiché.

 le composant parent (React ou HTML, peu importe) rendu pour chaque item du tableau doit nécessairement avoir un attribut `key` avec une valeur unique.


```
export default function PeopleList() {
  const people = ["Alex", "Bailey", "Charlie", "Dakota", "Eli", "Finley", "Gabby", "Harper", "Indigo", "Jordan"];

  return (
    <ul>
      { people.map((person, index) => )
        <li key={index + '_' + person}>{person}</li>
      }
    </ul>
  )
}
```

▼ **Nv 2** - exemple un peu plus compliqué, cette fois-ci le tableau contient des objets avec plusieurs clés.

```
export default function PeopleList() {
  const people = [
    { nom: "Doe", prénom: "John", ville: "New York" },
    { nom: "Smith", prénom: "Alice", ville: "Paris" },
    { nom: "Johnson", prénom: "David", ville: "Londres" },
    { nom: "Dubois", prénom: "Élise", ville: "Montréal" },
    { nom: "Lee", prénom: "Soo-Jin", ville: "Séoul" },
  ];

  return (
    <ul>
      { people.map((person, index) =>
        <li key={index + '_' + person.nom}>
          <p>{person.prénom} - {person.nom}</p>
          <p>{person.ville}</p>
        </li>
      )
    }
  )
}
```

```

    })
  </ul>
)
}

```

▼ **Nv 3** - la même chose, cette fois-ci on extrait le code HTML utilisé pour un élément de la liste dans un composant propre.

```

function PersonItem({nom, prénom, ville}) {
  return (
    <li>
      <p>{prénom} - {nom}</p>
      <p>{ville}</p>
    </li>
  )
}

export default function PeopleList() {
  const people = [
    { nom: "Doe", prénom: "John", ville: "New York" },
    { nom: "Smith", prénom: "Alice", ville: "Paris" },
    { nom: "Johnson", prénom: "David", ville: "Londres" },
    { nom: "Dubois", prénom: "Élise", ville: "Montréal" },
    { nom: "Lee", prénom: "Soo-Jin", ville: "Séoul" },
  ];

  return (
    <ul>
      { people.map((person, index) =>
        <PersonItem key={index + '_' + person.nom} nom={person.nom} prénom={person.prénom} ville={person.vill
      )}
    </ul>
  )
}

```

```
)  
}
```

? Afficher conditionnellement

[Documentation officielle chapitre "rendu conditionnel"](#)

i Dans ce chapitre nous utiliserons le composant `Avatar` créé au chapitre précédent.

▼ 1. Conditionner ce que retourne le composant

Actuellement le composant `Avatar` produira une erreur si une des props qu'il attend n'est pas conforme ou manquante. Nous pouvons conditionner son affichage, si les props sont valides alors nous afficherons l'avatar, autrement nous afficherons un avatar par défaut.

```
export default function Avatar({avatarSrc, avatarAlt, username}) {  
  
  if (avatarSrc && avatarAlt) {  
    // retourne l'avatar de l'utilisateur si on a bien les props avatarSrc et avatarAlt  
    return (  
      <>  
        <img  
          className="avatar"  
          src={avatarSrc}  
          alt={avatarAlt}  
        />  
        <p>Hello {username}</p>  
      </>  
    )  
  }  
}
```

```

    );
  }

  // sinon, retourne l'avatar par défaut
  return (
    <>
      <img
        className="avatar"
        src={'https://aws.avatar_default.png'}
        alt={'avatar par défaut'}
      />
      <p>Hello {username}</p>
    </>
  );
}

```

▼ 2. Conditionner ce qu'affiche un composant

i Avant de commencer : le raccourci JSX `&&` est utilisé pour indiquer à React d'afficher le composant si la condition est remplie.

```

function DemoComponent() {
  const randomNumberA = 23;
  const randomNumberB = 14;
  const randomBoolean = true;

  return (
    <div>
      { randomNumberA > randomNumberB && <p>A plus grand que B</p> } // s'affichera si A plus grand que B
      { randomNumberA < randomNumberB && <p>A plus petit que B</p> } // s'affichera si B plus grand que A
      { randomBoolean == "false" && <p>A plus petit que B</p> } // s'affichera si randomBoolean égale false

      // il est possible de combiner plusieurs condition, tant qu'après le dernier && se trouve un composant
    </div>
  );
}

```

```

        { randomNumberA > randomNumberB && randomBoolean == "false" && <p>A plus petit que B</p> }
      </div>
    )
  }
}

```

De manière abstraite :

`{ votre condition && votre composant }` . Si la condition est remplie le composant sera affiché, autrement non.

`{ votre conditionA && votre conditionB && votre composant }` . Si les conditions sont remplies le composant sera affiché, autrement non.

Conditionner ce que retourne le composant (Cf. exemple : *1. Conditionner ce que retourne le composant*) peut s'avérer verbeux et diminuer la lisibilité du code.

Il est possible d'avoir un seul `return` et de conditionner l'affichage au sein du HTML. Reprenons l'exemple précédent et conditionnons différemment l'affichage de l'avatar à l'aide du "raccourci" JSX `&&` .

```

export default function Avatar({avatarSrc, avatarAlt, username}) {

  return (
    <>
      // affiche l'avatar utilisateur si avatarSrc et avatarAlt
      { avatarSrc && avatarAlt &&
        <img
          className="avatar"
          src={avatarSrc}
          alt={avatarAlt}
        />
      }

      // si avatarSrc et avatarAlt non renseignés affiche l'avatar par défaut
      { !avatarSrc && !avatarAlt &&

```

```

        <img
          className="avatar"
          src={'https://aws.avatar_default.png'}
          alt={'avatar par default'}
        />
      }

      // pareil avec le username
      { username &&
        <p>Hello {username}</p>
      }
    }
  }
}
);
}
}

```



Les deux méthodes sont valables, selon les cas une solution s'impose plus que l'autre, mais au final vous êtes libres de conditionner ce que retourne votre composant ou bien la manière dont il affiche les éléments.

Passer des props à un composant

 [Documentation officielle chapitre "passer des props à un composant"](#)

Les composants React utilisent des props pour communiquer entre eux. Chaque composant parent peut transmettre des informations à ses composants enfants en leur donnant des props.

Les props peuvent vous rappeler les attributs HTML, mais vous pouvez leur transmettre n'importe quelle valeur JavaScript, y compris des objets, des tableaux et des fonctions.

i Dans ce chapitre nous utiliserons le composant `Avatar` créé au chapitre précédent.

▼ 1. Passer des variables à un composant enfant

Dans le chapitre précédent nous avons créé un composant `Avatar` destiné à afficher l'avatar d'un utilisateur. Il aurait pu être appelé de la manière suivante dans, par exemple, la page de profil de l'utilisateur.

```
export default function UserProfile() {  
  return (  
    <Avatar />  
    ...  
  );  
}
```

Si on regarde de plus près le composant `Avatar` on constate qu'il n'est pas aussi dynamique qu'il devrait l'être. Actuellement il n'affiche qu'une seule image, celle renseignée dans les constantes.

Idéalement nous souhaiterions lui passer dynamiquement les valeurs de `avatarSrc`, `avatarAlt` et `username`. De cette manière le composant `Avatar` deviendrait pleinement réutilisable.

Cela est rendu possible par le système de "props" de React. Le composant parent, ici `UserProfile`, va définir des props qui seront lisibles depuis le composant enfant, ici `Avatar`.

```
export default function UserProfile() {  
  return (  
    <Avatar avatarSrc={...} avatarAlt={...} username={..} />  
  );  
}
```

```

    ...
  );
}

```

Nous venons d'indiquer au composant Avatar qu'il recevra 3 props : `avatarSrc` , `avatarAlt` et `username` . Cela étant il est nécessaire d'indiquer les valeurs que véhiculeront chacune de ces props.

Ici nous créerons 3 variables contenant les informations nécessaires à l'affiche de l'avatar de notre utilisateur, celles-ci transiteront grâce au props et seront accessibles depuis le composant enfant `Avatar` .

```

export default function UserProfile() {
  const avatarSrc = "https://aws.image.avatar.maglecorp.png";
  const avatarAlt = "Maglecorp";
  const username = "Maglecorp";

  return (
    <Avatar avatarSrc={avatarSrc} avatarAlt={avatarAlt} username={username} />
    ...
  );
}

```

Dans le composant enfant `Avatar` nous devons faire quelques ajustements pour exploiter les props mis à disposition par le composant parent `UserProfile` .

```

export default function Avatar({avatarSrc, avatarAlt, username}) {
  console.log(avatarSrc); // affiche "https://aws.image.avatar.maglecorp.png"
  console.log(avatarAlt); // affiche "Maglecorp"
  console.log(username); // affiche "Maglecorp"

  return (

```



```

    <>
      <img
        className="avatar"
        src={avatarSrc}
        alt={avatarAlt}
      />
      <p>Hello {username}</p>
    </>
  );
}

```

Dorénavant le composant `Avatar` est pleinement réutilisable !



Utiliser les props pour passer des informations (variables, fonctions, composant ...) à un composant enfant est très utile à la construction de composants réutilisables.

En définissant le composant

`Avatar` de la sorte nous en avons fait une “coquille vide” destiné à afficher l’avatar d’un utilisateur, selon les données qui lui seront passés.

▼ 2. Aller plus loin

Les props permettent de faire transiter des variables, fonctions ou encore des composants.

Le principe est le même que dans l’exemple précédent.



Exemple de composant parent passant 3 props : une variable, une fonction et un autre composant React.

```

// Le composant React passé en props
function ComposantDemo() {
  return (
    <p>composant demo</p>
  )
}

// Le composant parent
export default function UserProfile() {
  // La fonction passée en props
  function demoFunction() {
    console.log('hello demo');
  }

  // La variable passée en props
  const demoVariable = 'demo variable';

  return (
    <UnComposant propsFunction={demoFunction} propsVariable={demoVariable} propsComposant={ComposantDemo} />
    ...
  );
}

```

📖 Exemple de composant enfant recevant 3 props : une variable, une fonction et un autre composant React.

```

export default function UnComposant({propsFunction, propsVariable, propsComposant}) {
  console.log(propsVariable); // affiche 'demo variable'

  propsFunction(); // executera la fonction "demoFunction" du composant parent, affichera donc 'hello demo'
}

```

```


return (
  <>
    {propsComposant} // affichera le composant "ComposantDemo"
  </>
);
}

```



Commencer par créer le composant enfant en “dur”, lorsque vous êtes satisfait du résultat demandez-vous quelles variables, fonctions ... devraient être passées par le composant parent pour rendre votre nouveau composant pleinement réutilisable.

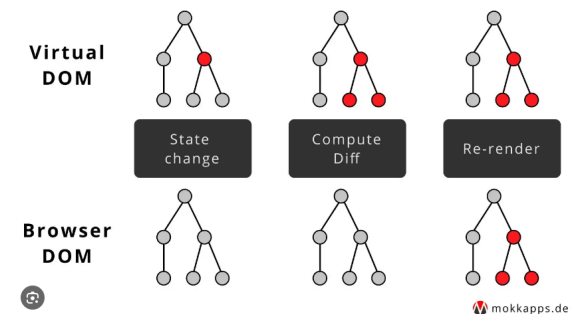
Re-rendering

 Le concept du "re-rendering" (ré-exécution) est essentiel dans le fonctionnement de React.

Le re-rendering se produit lorsque l'état d'un composant ou les propriétés (props) passées par un composant parent changent. Lorsqu'un composant est ré-exécuté, sa fonction de rendu (le `return` contenant le markup HTML) est appelée à nouveau, ce qui entraîne une mise à jour de l'interface utilisateur pour refléter les nouveaux états ou les nouvelles propriétés.

En détail


Lorsqu'un composant est monté pour la première fois, la fonction de rendu du composant est appelée, un arbre virtuel (Virtual DOM) est créé en mémoire. L'arbre virtuel représente la structure des composants et de




leur contenu, y compris les éléments DOM correspondants.

Lorsqu'un composant est ré-exécuté l'arbre virtuel est comparé à la version précédemment générée de l'arbre virtuel. React détermine les différences entre les deux arbres virtuels, c'est-à-dire les nœuds qui ont été ajoutés, supprimés ou modifiés.


Au lieu de mettre à jour directement le DOM réellement affiché dans le navigateur, React effectue uniquement les modifications nécessaires pour refléter les différences détectées. Les modifications sont appliquées de manière optimisée sur le DOM réel, réduisant ainsi les coûts de performance liés à la manipulation du DOM.

 Le re-rendering est un concept clé dans React car il permet de maintenir une interface utilisateur réactive et de garantir que les composants reflètent toujours l'état actuel de l'application.

 Il est important de prendre en compte les performances lors de la conception de votre application, en évitant les rendus inutiles ou en optimisant les composants lorsque cela est nécessaire.

Hook `useState`

 [Documentation officielle du hook `useState`](#)

 Le hook `useState()` permet de gérer l'état local dans les composants. En utilisant ce hook, vous pouvez créer et mettre à jour des variables d'état qui déclenchent le rendu de votre composant à chaque modification de leur valeur.

Pour utiliser le hook `useState()`, vous devez tout d'abord importer la fonction depuis React.

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

Pour déclarer le hook.

```
const [state, setState] = useState(initialState);
```

- `state` : variable qui contient l'état actuel.
- `setState` : fonction qui permet de mettre à jour l'état.
- `initialState` : la valeur initiale de l'état.

Exemple d'utilisation :

```
"use client"

import { useState } from 'react';

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

  const handleIncrement = () => {
    setCount(count + 1);
  };

  return (
    <div>
      <p>Count: {count}</p>
      <button onClick={handleIncrement}>Incrémenter</button>
    </div>
  );
}
```

```
);  
}
```


Dans cet exemple, nous utilisons le hook `useState()` pour créer une variable d'état appelée `count` et une fonction `setCount()` pour mettre à jour cette variable. Nous initialisons `count` à 0 en utilisant `useState(0)`.

Lorsque l'utilisateur clique sur le bouton, la fonction `increment` est appelée, ce qui met à jour la valeur de `count` en utilisant `setCount()`.

Grâce à cette mise à jour de l'état, le composant se est re-rendu (voir chapitre "re-rendering") et le nouveau nombre est affiché.

Hook useRef

 [Documentation officielle du hook useRef](#)

 Le hook `useRef()` permet de créer une référence mutable dans un composant fonctionnel. C'est à dire qu'il sert à stocker une valeur qui persiste entre les différents rendus du composant (voir chapitre re-rendering) , sans déclencher un nouveau rendu lorsque cette valeur est mise à jour.

```
import { useRef } from 'react';  
  
function MyComponent() {  
  const myRef = useRef(); // la valeur initiale d'un hook useRef est null  
  
  return (  
    <div>  
      ...  
    </div>  
  );  
}
```

```
);  
}
```

Vous pouvez maintenant utiliser

`myRef` pour stocker des valeurs qui doivent persister entre les rendus de votre composant.

Par exemple, vous pouvez l'utiliser pour stocker une référence à un élément du DOM :


```
function MyComponent() {  
  const inputRef = useRef();  
  
  const handleClick = () => {  
    // le hook useRef étant initialisé à null il est préférable de conditionner son utilisation  
    if (inputRef) {  
      console.log(inputRef.current.value);  
    }  
  };  
  
  return (  
    <div>  
      <input ref={inputRef} type="text" />  
      <button onClick={handleClick}>Focus</button>  
    </div>  
  );  
}
```

Dans cet exemple, nous créons une référence `inputRef` en utilisant `useRef()`. Nous pouvons ensuite l'attacher à l'attribut `ref` de l'élément input.

Lorsque l'utilisateur clique sur le bouton, la fonction `handleClick()` est appelée et utilise la référence `inputRef.current` pour accéder à l'élément input et afficher sa valeur dans la console.

Hook useEffect

 [Documentation officielle du hook useEffect](#)

 Le Hook `useEffect()` permet, entre autres, d'écouter les changements de valeur d'une ou plusieurs variables dans les composants fonctionnels. Il peut être déclenché dans différentes situations.

```
"use client"

import { useEffect } from 'react';

function MyComponent() {

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

  return (
    ...
  )
}
```

Pour utiliser un hook `useEffect()` la syntaxe de la déclaration est tout le temps la même :

```
useEffect(() => {
  ...
})
```



```
}, []);
```

Le tableau à la dernière ligne est destiné à accueillir le nom des variables que l'on souhaite "écouter", ce sont les dépendances du hook `useEffect()`.

💡 Si le tableau est vide le hook ne sera déclenché qu'au montage du composant (voir chapitre "re-rendering"). Cela peut-être très utile dans le cas où nous souhaiterions exécuter un traitement au montage du composant (fetch une API, accéder au local storage, etc).

Exemple d'un hook `useEffect()` écoutant la valeur d'une variable, ici `todos`, qui est un tableau contenant des tâches à réaliser :

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

On pourrait par exemple faire un `console.log()` si le tableau `todos` ne contenait plus de tâches.

```
useEffect(() => {  
  if (todos.length === 0) {  
    console.log('la journée est terminée');  
  }  
}, [todos]);
```

Dans cette exemple, à chaque fois que la valeur de la constante `todos` sera modifiée (lorsque l'on ajoute ou supprime une tâche) le hook `useEffect()` s'exécutera. De cette manière nous sommes sûrs que lorsque le tableau des tâches `todos` sera vide le log apparaîtra.

🔗 Hook useContext

[🔗 Documentation officielle du hook useContext](#)

i Le contexte est un mécanisme de React qui permet de partager des données à un arbre de composants, sans avoir à les passer explicitement de parent à enfant au travers des props. Il est très utile lorsque vous avez des données que plusieurs composants doivent utiliser, sans qu'ils soient directement reliés par une relation parent-enfant.

Dans un premier temps il faut créer le contexte, ici le contexte sera créé dans un fichier propre appelé `AppContext.jsx`.

```
"use client"

import { createContext } from "react";

// voir explication A
export const AppContext = createContext();

// voir explication B
export default function AppContextProvider({ children }) {

  // voir plus bas pour information à propre de la value
  return <AppContext.Provider value={...}>{children}</AppContext.Provider>
}
```

A On initialise le contexte, ici il s'appellera `AppContext` Cf. le nom de la variable.

B On initialise une fonction, ici `AppContextProvider`, qui retourne un le provider de notre contexte `AppContext`. C'est ce composant qui mettra à disposition les données que l'on souhaite partager.

⚠ À propos des contextes :

- Le contexte doit être placé dans l'arborescence à un niveau supérieur des composants qui en ont besoin. Cela garantit que tous les composants descendants peuvent accéder à la valeur fournie par le contexte.
- Le provider d'un contexte accepte une prop appelée `value` qui permet de spécifier la valeur du contexte que vous souhaitez fournir. Cette valeur peut être de n'importe quel type, string, objet, tableau, etc.
- Lorsque la valeur du contexte change, tous les composants descendants qui consomment ce contexte seront ré-exécutés pour refléter cette nouvelle valeur.
- Vous pouvez utiliser plusieurs contexte dans votre arborescence de composants pour fournir différents contextes à des parties spécifiques de votre application.

```
"use client"

import { createContext } from "react";

export const AppContext = createContext();

export default function AppContextProvider({ children }) {
  const valueA = 'hello world';
  const valueB = false;
  const valueC = [{ dynamos: [] }, { simploniens: [] }];

  const sayHello = () => {
    console.log('hello magle');
  }

  const contextValues = {
    valueA,
    valueB,
    valueC,
    sayHello
  }
}
```

```
    return <AppContext.Provider value={contextValues}>{children}</AppContext.Provider>
  }
```

Dans cet exemple je souhaite que le contexte `AppContext` puisse donner accès à ses données à l'ensemble de mon application, pour cela je place son provider `AppContextProvider` dans le fichier `layout.js` à la racine du dossier `app`.

```
import './globals.scss'
import AppContextProvider from "@/app/AppContext";

export const metadata = {
  title: 'Create Next App',
  description: 'Generated by create next app',
}

export default function RootLayout({ children }) {
  return (
    <html lang="en">
      <body>
        <AppContextProvider>
          {children}
        </AppContextProvider>
      </body>
    </html>
  )
}
```

Dès lors les valeurs partagées par le contexte `AppContext` peuvent être consommées par les composants enfants du provider `AppContextProvider`.

Par exemple dans le fichier `page.js` à la racine du dossier `app` (ce fichier est englobé par le layout, voir chapitre Next “routing files”), je peux consommer le context `AppContext` à l'aide de la fonction `useContext` mise à disposition par React :

```
"use client"

import { useContext } from "react";
import { AppContext } from "@app/AppContext";

export default function Home() {
  const { valueA, sayHello } = useContext(AppContext)
  console.log(valueA) // affichera "hello world" Cf. la variable valueA du contexte

  sayHello(); // affichera "hello magle" Cf. la fonction sayHello du contexte

  return (
    <p>page d'accueil</p>
  )
}
```