





# Premier composant React

Thèmes	 <a href="#">React</a>
Description	Appréhender le système de composant de React.
Date de création	@13 mars 2025 19:33
Créée par	 Kévin Wolff

## Chapitres

 [Introduction](#)

 [Les fondamentaux](#)

 [Le JSX et sa syntaxe](#)

 [Interpolation des variables en JSX](#)


 [Expressions Javascript dans JSX](#)


 [Les conditions en JSX](#)

 [Les boucles en JSX](#)

## Sur le même sujet

 [Créer un projet Vite + React](#)

 [Le router](#)

 [Approche atomic design](#)

## Introduction

 [Documentation officielle – penser React](#)

 [Documentation officielle – tutoriel complet de React](#)

Les applications React sont construites à partir de composants, des éléments réutilisables et autonome. Un composant React est simplement une fonction JavaScript qui retourne du markup (du JSX). Il peut représenter un élément simple, comme un bouton, ou quelque chose de plus complexe, comme une carte de profil utilisateur.

## Les fondamentaux

Les composants React doivent être déclarés dans des fichiers `.jsx`. Par convention, chaque fichier contient un unique composant. Celui-ci est défini par une fonction JavaScript portant le même nom que le fichier et retournant du markup (JSX).

Cette fonction peut inclure des traitements, utiliser des bibliothèques externes, etc., mais sa seule obligation est de retourner du JSX représentant l'interface du composant. Elle peut également recevoir des propriétés (props), un concept que nous aborderons plus loin.

Exemple d'un composant utilisé pour afficher un avatar, `Avatar.jsx` :

```
export default function Avatar() {  
  return (  
      
  )  
}
```

```

    alt="Avatar de Wagle Corp"
  />
};
}

```

Vous remarquerez que le nom de la fonction commence par une majuscule, c'est une convention dans React pour distinguer les composants des éléments HTML natifs, il en va de même pour le nom du fichier.

Un des avantages de React est la composition des composants. On peut assembler plusieurs composants pour former une interface plus complexe.

Par exemple, un composant `UserProfileCard` peut regrouper plusieurs sous-composants comme `Avatar`, `UserInformation` et `UserSocialMediaLinks`. Cela permet de créer une interface modulaire et facile à maintenir.

```

import Avatar from './Avatar.jsx';
import UserInformation from './UserInformation.jsx';
import UserSocialMediaLinks from './UserSocialMediaLinks.jsx';

export default function UserProfileCard() {
  return (
    <div>
      <Avatar />
      <UserInformation />
      <UserSocialMediaLinks />
    </div>
  );
}

```

Cette approche permet de créer des interfaces utilisateur complexes en assemblant des composants d'interface simples. Elle rend le développement bien plus modulaire, avec des composants réutilisables et autonomes, garantissant une meilleure maintenabilité, uniformité et évolutivité.

Vous noterez sans doute que cela n'a pas de sens car les données de l'image d'avatar sont statiques, vous avez raison et la suite du document explore comment rendre ce composant réutilisable plus autonome.

Un composant React ne s'affiche pas automatiquement dans l'interface. Pour qu'il soit visible dans le navigateur, il doit être inclus dans l'arborescence des composants de l'application.

Concrètement, cela signifie que le composant doit être rendu quelque part dans l'arbre de React, qui commence à partir du point d'entrée `main.jsx`.

```

import { createRoot } from 'react-dom/client';
import UserProfilCard from './UserProfilCard.jsx';

createRoot(document.getElementById('root')).render(<UserProfilCard />);

```

Le bout de code ci-dessus a vocation à illustrer comment un composant est finalement intégré au HTML mais il n'est pas à reproduire. Cette approche n'aurait aucun sens.

## Les fragments React

En React, un composant doit toujours retourner un seul élément parent. Si plusieurs éléments adjacents sont retournés, cela génère une erreur. L'exemple suivant provoque une erreur car le composant retourne deux éléments `<p>`.

```
export default function Demo() {  
  return (  
    <p>Premier élément</p>  
    <p>Second élément</p>  
  );  
}
```

Cependant, il peut arriver qu'on ait besoin d'afficher plusieurs éléments sans ajouter de `<div>` supplémentaire dans le DOM, c'est ici qu'interviennent les fragments React. Un fragment React est "une balise vide" : `<></>` (n'oubliez pas de la fermer).

```
export default function Demo() {  
  return (  
    <>  
      <p>Premier élément</p>  
      <p>Second élément</p>  
    </>  
  );  
}
```

## Le JSX et sa syntaxe

Avant de découvrir les fonctionnalités propres à React il est indispensable de découvrir les particularités de la syntaxe JSX. Le JSX est une extension de Javascript utilisée par React pour décrire l'interface utilisateur. Il permet d'écrire des balises similaires au HTML directement dans du code Javascript, rendant la définition des composants plus intuitives.

Contrairement au HTML, JSX n'est pas compris directement par le navigateur. Il doit être converti en Javascript avant d'être interprété par React.

**Bien que JSX ressemble à du HTML, il y a plusieurs différences importantes :**

1. En JSX, on utilise `className` au lieu de `class`, car `class` est un mot-clé réservé.
2. En HTML, certaines balises peuvent être laissées ouvertes ( `<img>` , `<input>` ), mais en JSX toutes les balises doivent être fermées.
3. Les noms d'attributs suivent la convention camelCase en JSX : `onclick` devient `onClick` , `maxLength` devient `maxLength` , etc.

## Interpolation des variables en JSX

JSX permet d'insérer des variables Javascript dans le markup en les entourant d'accolades `{ }`. Cela permet d'afficher du contenu dynamique directement dans le rendu du composant.

```
export default function Avatar() {  
  const avatarSrc = "https://aws.image.avatar.waglecorp.png";  
  const avatarAlt = "Avatar de Wagle Corp";  
  
  return (  
    <img  
      className="avatar"  
      src={avatarSrc}  
      alt={avatarAlt}  
    />  
  );  
}
```

## Expressions Javascript dans JSX

JSX permet d'exécuter des expressions Javascript directement dans le markup grâce aux accolades `{ }`.

### Opération mathématique

```
export default function Demo() {  
  const numberA = 2;  
  const numberB = 2;  
  
  return (  
    <p>Deux + Deux = { numberA + numberB }</p>  
  );  
}
```

```
<p>Deux + Deux = 4</p>
```

### Appel de fonction

```
import { addition } from "mathsService.js";  
  
export default function Demo() {  
  return (  
    <p>Trois + Trois = { addition(3, 3) }</p>  
  );  
}
```

```
<p>Trois + Trois = 6</p>
```

## Operation ternaire

```
export default function Demo() {  
  return (  
    <p>Le chiffre 5 est { 5 > 12 ? "plus grand" : "plus petit" } que 12</p>  
  );  
}
```

<p>Le chiffre 5 est plus petit que 12</p>

## Les conditions en JSX

Dans JSX, on ne peut pas utiliser directement un `if` dans le markup, mais on peut contourner cette limite avec différentes approches : les opérateurs ternaires ( `condition ? valeur1 : valeur2` ) ou les opérateurs AND ( `&&` ).

### Opérateur ternaire

```
import { getUserAuthState } from "authService.js";  
  
export default function Demo() {  
  const userAuthState = getUserAuthState(); // True pour l'exemple.  
  
  return (  
    <p>{ userAuthState.isAuthenticated ? "Vous êtes connecté" : "Veuillez vous connecter" }</p>  
  );  
}
```

<p>Vous êtes connecté</p>

### Opérateur ternaire avec composant

```
import { getUserAuthState } from "authService.js";  
import { WelcomeScreen } from "WelcomeScreen.js";  
import { LoginScreen } from "LoginScreen.js";  
  
export default function Demo() {  
  const userAuthState = getUserAuthState(); // False pour l'exemple.  
  
  return (  
    <>{ userAuthState.isAuthenticated ? <WelcomeScreen /> : <LoginScreen /> }</>  
  );  
}
```

```
<div>  
  <form id="login-form">  
    ...
```

```
</form>
</div>
```

### Opérateur AND

```
import { getUserAuthState } from "authService.js";

export default function Demo() {
  const userAuthState = getUserAuthState(); // True pour l'exemple.

  return (
    <>
      { userAuthState.isAuth && <p>"Vous êtes connecté"</p> }
      { !userAuthState.isAuth && <p>"Veuillez vous connecter"</p> }
    </>
  );
}
```

```
<p>Vous êtes connecté</p>
```

### Opérateur AND avec composant

```
import { getUserAuthState } from "authService.js";
import { WelcomeScreen } from "WelcomeScreen.js";
import { LoginScreen } from "LoginScreen.js";

export default function Demo() {
  const userAuthState = getUserAuthState(); // False pour l'exemple.

  return (
    <>
      { userAuthState.isAuth && <WelcomeScreen /> }
      { !userAuthState.isAuth && <LoginScreen /> }
    </>
  );
}
```

```
<div>
  <form id="login-form">
    ...
  </form>
</div>
```

## Les boucles en JSX

En JSX, on utilise `map()` pour afficher une liste d'éléments dynamiques. Chaque élément généré doit avoir une `key` unique pour aider React à optimiser le rendu, sans quoi React affichera un avertissement.

```

export default function EmployeeList() {
  const employees = [
    { id: 1, lastname: "Doe", firstname: "John", city: "New York" },
    { id: 2, lastname: "Smith", firstname: "Alice", city: "Paris" },
    { id: 3, lastname: "Johnson", firstname: "David", city: "Londres" },
    { id: 4, lastname: "Dubois", firstname: "Élise", city: "Montréal" },
    { id: 5, lastname: "Lee", firstname: "Soo-Jin", city: "Séoul" }
  ];

  return (
    <ul>
      { employees.map((employee, index) =>
        <li key={`employee_${employee.id}>
          <p>{employee.firstname} - {employee.lastname}</p>
          <p>{employee.city}</p>
        </li>
      )}
    </ul>
  )
}

```