

# React

Les composants et la lutte des classes

Lors de la dernière quête, vous avez modifié le contenu du fichier **App.js**, dans le but de modifier l'affichage de la page d'accueil. Mais comment ces éléments ont-ils été affichés par React ? Et comment est construit ce fichier App.js ?

Commençons par comprendre le fonctionnement de React en utilisant nos connaissances des SPA. Dans une SPA, le navigateur ne reçoit du serveur qu'un fichier **index.html**. Et le contenu va être injecté en Javascript.

Ici le fichier **index.html** utilisé par le navigateur se trouve dans le dossier **public**. Le dossier public est donc ce qu'on appelle le point d'entrée pour votre navigateur dans votre application React.

Dans le dossier **src** se trouve le code javascript qui va permettre de manipuler le DOM et donc de modifier le contenu du HTML utilisé par le navigateur.

Le dossier **node\_modules** contient toutes les dépendances de votre application React (les packages nodes).

Les fichiers **packages.json** et **package-lock.json** permettent d'avoir la liste des dépendances et leur version. Si le dossier **node\_modules** est supprimé, vous pouvez, grâce à ces fichiers, réinstaller vos dépendances.

Le premier fichier qui interagit avec le HTML est le fichier **index.js**.

Ce fichier cible la *div* vide du fichier index.html et utilise ReactDOM pour injecter du contenu à l'intérieur (avec la fonction **render()**) en utilisant la syntaxe JSX comme nous l'avons fait « manuellement » dans la quête #2.

Jusqu'ici nous avons vu que JSX permettait de créer et injecter des éléments de type *h1*, *div* etc, dans le DOM en utilisant une syntaxe proche du HTML. Ici React semble créer un élément **<App />**. Pourtant **<App />** ne fait clairement pas partie des éléments HTML utilisables normalement.

Mais quel est donc cet élément App ?

```
const root = ReactDOM.createRoot(document.getElementById('root'));
root.render(
  <React.StrictMode>
    <App />
  </React.StrictMode>
);
```

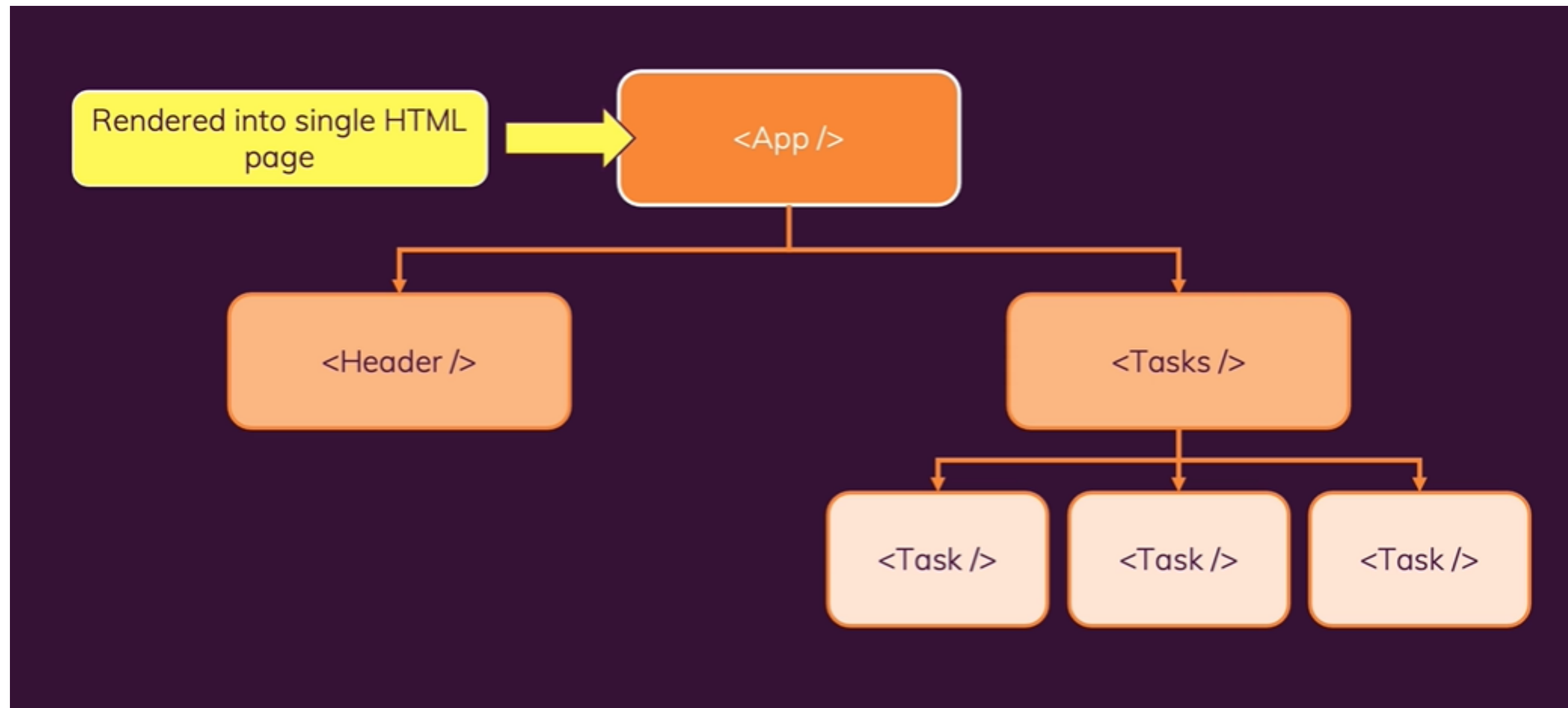
`<App />` fait en fait référence au fichier `App.js`. Ce fichier contient une fonction nommée `App()`, qui elle retourne du JSX plus « classique », permettant de créer dans le DOM des balises *div* etc.

Quand le JSX du fichier `index.js` utilise un élément `<App />` pour modifier le contenu affiché dans le navigateur, c'est donc à la fonction `App`, du fichier `App.js`, qu'elle fait référence. Cette fonction est ce que l'on appelle un **composant**. Et c'est grâce aux composants que l'on crée le contenu de nos pages.

Les pages créées avec une application React sont en effet rendus à partir de plusieurs composants que l'on peut créer : un composant pour gérer l'affichage du menu, un composant pour gérer l'affichage des articles, un composant pour gérer l'affichage du footer etc.

On utilise donc les composants comme des legos, que l'on vient imbriquer et qui, ensemble, permettront de créer le HTML complet d'une page. On a donc un **composant parent**, qui appelle un ou plusieurs **composants enfants**. Ces composants enfants peuvent créer des éléments HTML mais aussi appeler d'autres composants enfants et ainsi de suite.

Exemples de hiérarchie de composants parents / enfants :





Un composant React est donc une fonction qui retourne du JSX. Historiquement, les composants React étaient créés via des classes, donc en utilisant les principes de l'**OOP** (**object oriented programming**). Mais React tend aujourd'hui à se rapprocher des principes de la **programmation fonctionnelle**. Les classes ont été remplacées par des fonctions (plus d'infos sur la programmation fonctionnelle ici : <https://github.com/readme/featured/functional-programming>). A noter que les composants fonctionnels peuvent être créés via des expressions de fonction classique ou via des **expressions de fonctions fléchées**. Cette dernière syntaxe étant préférée.

Chaque fonction créant un composant est contenu dans un fichier portant le même nom. Pour pouvoir appeler cette fonction dans d'autres fichiers il faut l'exporter. A la fin de chaque fichier de composant, se trouve donc la ligne d'export de la fonction.

Par exemple pour le composant App :

```
export default App;
```

Dans les fichiers qui importent un composant, il faut faire l'import avec :

```
import App from './App';
```

Composant fonctionnel :



```
function Welcome() {  
  return <h1>Bonjour</h1>;  
}
```

Composant de classe :



```
class Welcome extends React.Component {  
  render() {  
    return <h1>Bonjour</h1>;  
  }  
}
```

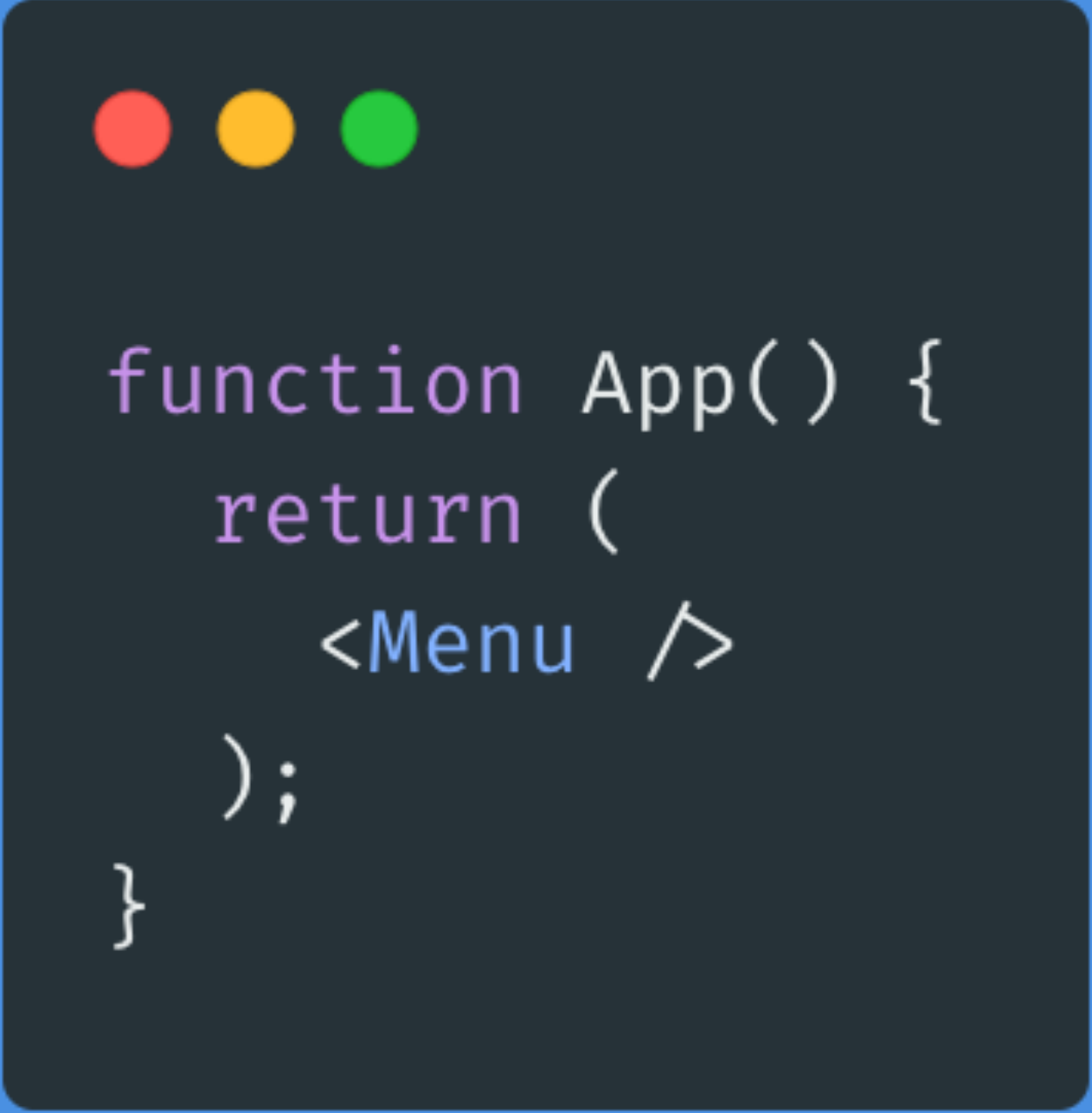
Le JSX est donc à la fois capable de créer des éléments HTML à créer dans le DOM, par exemple `<h1>`, mais aussi d'utiliser des composants `<App />`. La même syntaxe est utilisé pour les deux avec les chevrons d'ouverture / fermeture (« `<` »). Quand un composant est appelé dans le JSX, les éléments créé dans ce ce composant vont être récupérés pour être rendus dans le DOM. Et idem si le composant en question appelle d'autres composants.

Comment le JSX fait-il alors la différence entre un élément HTML à créer et un composant à rendre ?

Un élément HTML à créer en JSX sera écrit en minuscule. Un composant sera utilisé lui avec du **PascalCase**. Si le JSX rencontre un élément avec une majuscule, il va donc essayer de récupérer le contenu du composant correspondant au nom utilisé.

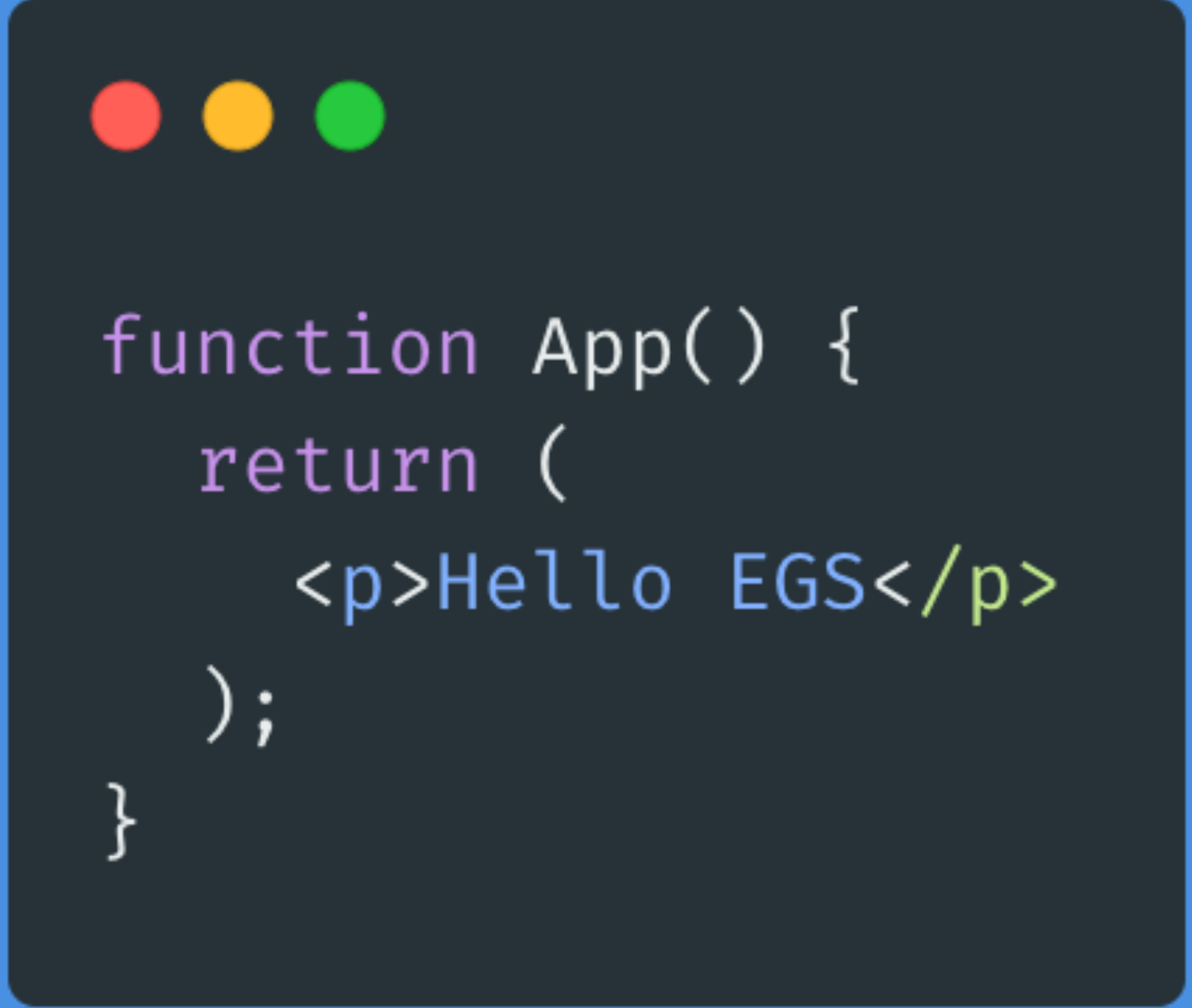


Appel du composant Menu :



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

Création d'une balise paragraphe :



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