

Introduction

Si on se fie à la définition offerte par la documentation React est "une bibliothèque JavaScript pour créer des interfaces utilisateurs". Cette définition est un peu générique et ne nous aide pas forcément à comprendre ce qu'est réellement React.

React est une librairie qui va vous permettre de représenter une interface à l'aide d'élément qui vont être capable d'évoluer en fonction des changements d'état de votre application. Pour mieux comprendre cette approche prenons un exemple concret d'interface Web basée sur la manipulation du DOM

```
class TodoList {  
  
    todos = []  
    element = null  
  
    constructor (element) {  
        this.element = element  
    }  
  
    addItem (name) {  
        this.todos.push(name)  
        const li = document.createElement('li')  
        const a = document.createElement('a')  
        a.innerText = 'Supprimer'  
        li.appendChild(a)  
        a.addEventListener('click', () => this.removeItem(name))  
        this.element.appendChild(li)  
    }  
  
    removeItem (name) {  
        this.todos = this.todos.filter(t => t !== name)  
        // Il faut supprimer l'élément dans le dom  
    }  
  
    editItem (index, name) {  
        this.todos[index] = name  
        // Il faut changer le texte dans le DOM  
    }  
}
```

On le voit ici, chaque transformation de notre état (liste de tâche) doit s'accompagner d'une transformation des éléments dans le DOM. Ces manipulations deviennent de plus en plus difficile à assurer avec la montée en complexité de notre application.

```
class TodoList extends React.Component {
```

```

constructor (props) {
  super(props)
  this.state = {todos: []}
}

addItem (name) {
  const todos = [...this.state.todos, name]
  this.setState({todos: todos})
}

removeItem (name) {
  const todos = this.todos.filter(t => t !== name)
  this.setState({todos: todos})
}

editItem (index, name) {
  const todos = [...this.state.todos]
  todos[index] = name
  this.setState({todos: todos})
}

render () {
  return <ul>
    {this.state.todos.map((name) => {
      return <li>{name} <button onClick={this.removeItem.bind(this, name)}>Supprimer</button></li>
    })}
  </ul>
}

```

React permet de séparer les choses avec 2 éléments distincts

- Le `state`, qui va permettre de stocker l'état de l'application et qui pourra être modifié suite à différentes interactions utilisateur.
- La fonction `render()`, qui rend une nouvelle version de l'interface en fonction de l'état de l'application.

On ne s'occupe plus du tout des mutations du DOM. A chaque changement de l'état de notre application, React relancera la fonction `render()` et appliquera les changements au niveau du DOM (on rentrera plus en profondeur dans ce fonctionnement tout au long de cette formation).

Pourquoi React plutôt que...

React n'est pas la seule librairie qui permet de créer des interfaces utilisateurs et on peut se poser la question concernant ce choix.

React est simple (en surface)

React est une librairie qui est "plutôt" simple en surface et qui peut être utilisé avec un nombre très limité de fonctions. Une simple fonction JavaScript peut servir de composant React et en dehors du JSX, la structure reste très proche du JavaScript classique.

L'écosystème est très développé.

React fait parti des premiers frameworks "virtual dom" et a du coup pu se développer un écosystème conséquent (on trouvera de nombreux composant déjà conçu). Il est aussi possible d'utiliser React pour autre chose que le web.

- [React Native](#) permet de créer des applications pour les mobiles, tablette
- [React Native Windows](#) permet de créer des applications bureau pour Windows et MacOS
- [Phelia](#) permet de créer des applications Slack basée sur React.

React peut être branché à n'importe quoi et n'est pas du tout spécifique à la manipulation du DOM (c'est en fait une autre librairie, `react-dom` qui lui donne ces capacités)

JSX

Pour représenter l'interface React utilise une syntaxe qui vient s'ajouter au JavaScript. Cette syntaxe est très simple à apprendre / comprendre et se rapproche de ce que l'on connaît déjà avec l'HTML.

```
return (
  <div>
    <h3>TODO</h3>
    <TodoList items={this.state.items} />
    <form onSubmit={this.handleSubmit}>
      <label htmlFor="new-todo">
        Tâche à faire
      </label>
```

```

<input
  id="new-todo"
  onChange={this.handleChange}
  value={this.state.text}
/>
<button>
  Ajouter #{this.state.items.length + 1}
</button>
</form>
</div>
);
}

```

Cette syntaxe est totalement optionnelle (on peut utiliser React sans utiliser le JSX).

Prérequis

Il est impératif d'être à l'aise avec le JavaScript et la syntaxe ES2015.

- Vous devez comprendre ce que veux dire `maFonction.bind(this, param1)`
- Vous devez savoir la différence entre `() => {}` et `function () {}`
- Vous devez être à l'aise avec les `import ... from '...'`

NB :

La **fonction bind()** crée une nouvelle **fonction liée**, qui est un objet de **fonction exotique** (un terme de l'ECMAScript 2015) qui enveloppe l'objet de **fonction original**. L'appel de la **fonction liée** entraîne généralement l'exécution de sa **fonction enveloppée**.

Fonctions fléchées

Une **expression de fonction fléchée** (*arrow function* en anglais) permet d'avoir une syntaxe plus courte que [les expressions de fonction](#) et ne possède pas ses propres valeurs pour [`this`](#), [`arguments`](#), [`super`](#), ou [`new.target`](#). Les fonctions fléchées sont souvent [**anonymes**](#) et ne sont pas destinées à être utilisées pour déclarer des méthodes.

Nos premiers pas avec React

Nous allons créer notre premier élément et le branche au DOM.

On peut passer par les liens CDN pour pouvoir travailler avec react

Liens CDN

React et ReactDOM sont tous deux accessibles depuis un CDN.

```
<script crossorigin
src="https://unpkg.com/react@18/umd/react.development.js"></script>
<script crossorigin src="https://unpkg.com/react-dom@18/umd/react-
dom.development.js"></script>
```

Ce ne pas forcement la manière standard de travailler avec react mais la manière la plus simple qui va nous permettre de commencer rapidement.

- React c'est une librairie très simple qui permet de créer les éléments.
- Ensuite ces éléments il va falloir les brancher à quelques choses, soit au DOM, soit à une application native, soit à autre chose.
Donc react DOM va nous permettre de branche les elements react à notre page web

Si quelqu'un par ex veut faire une appli native il utilise react et react native pas react dom

L'ajout de l'attribut **defer** permet de charge du JavaScript à la fin

```
<script crossorigin src="https://unpkg.com/react@18/umd/react.development.js"
defer></script>
<script crossorigin src="https://unpkg.com/react-dom@18/umd/react-
dom.development.js" defer></script>
```

Installer React



Dans la console (se placer sur le dossier ou bureau)

- ✓ node -v
- ✓ npm -v
- ✓ npx create-react-app nom-projet
- ✓ cd nom-projet
- ✓ code .

Lancer projet : npm start
Reprendre projet : npm i
Compiler projet : npm run build

Bibliothèques utiles :
npm i -s react-router react-router-dom node-sass

1^{er} télécharger nodejs

The screenshot shows the official Node.js website at nodejs.org/en. The header includes navigation links for Learn, About, Download, Blog, Docs, Contribute, and Certification. A prominent feature is the title "Run JavaScript Everywhere". Below it, a paragraph describes Node.js as a free, open-source, cross-platform JavaScript runtime environment. A large green button labeled "Download Node.js (LTS)" is visible, along with a note about long-term support for v22.13.0. To the right, there's a snippet of JavaScript code and a "Create an account" button.

← → ⌂ nodejs.org/en

NEW Security releases to be made available soon

node Learn About Download Blog Docs Contribute ↗ Certification ↗

Run JavaScript Everywhere

Node.js® is a free, open-source, cross-platform JavaScript runtime environment that lets developers create servers, web apps, command line tools and scripts.

[Download Node.js \(LTS\) ↴](#)

Downloads Node.js **v22.13.0¹** with long-term support.
Node.js can also be installed via package managers.

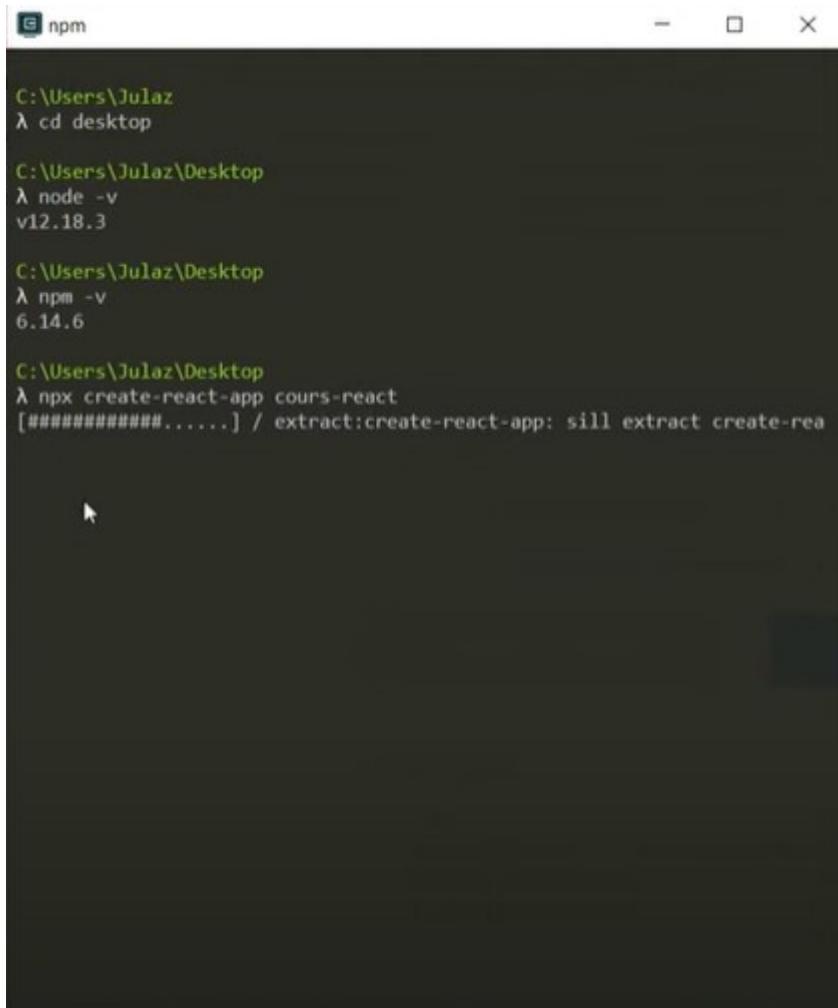
Want new features sooner? Get **Node.js v23.6.0¹** instead.

Create an account

```
1 // s
2 import
3
4 cons
5 re
6 re
7 });
8
9 // s
10 serv
11 co
12 });
13
14 // r
```

JavaScript

Une fois nodejs est bien installé.



A screenshot of a terminal window titled "npm". The window shows a command-line session with the following history:

```
C:\Users\Julaz
λ cd desktop

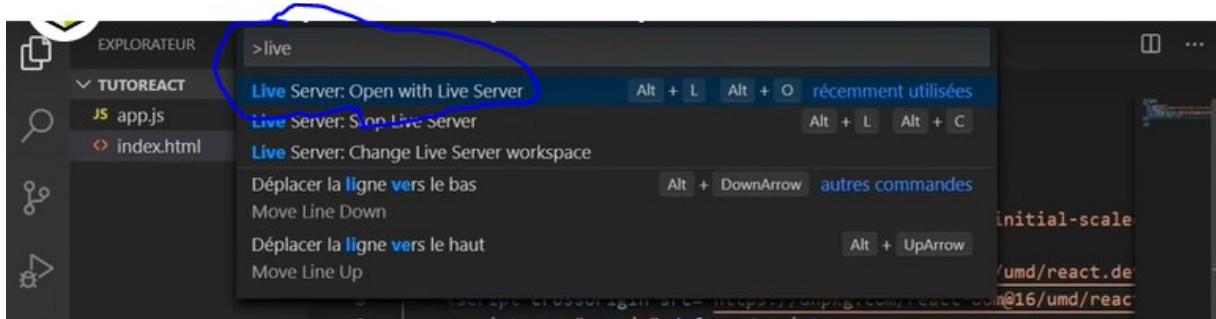
C:\Users\Julaz\Desktop
λ node -v
v12.18.3

C:\Users\Julaz\Desktop
λ npm -v
6.14.6

C:\Users\Julaz\Desktop
λ npx create-react-app cours-react
[#####.....] / extract:create-react-app: sill extract create-re
```

Si nous voulons utiliser directement le live server , qui est une extension visual studio code et cette extension va nous permettre de réactualiser les choses automatique.

Ctrl+p

A screenshot of the Visual Studio Code interface. The left sidebar shows a project structure with a folder 'EXO1' containing files 'app.js' and 'index.html'. The main area has three tabs: 'Welcome', 'index.html', and 'app.js'. The 'index.html' tab is active, displaying the following code:

```
<!DOCTYPE html>
<html lang="en">
<head>
    <meta charset="UTF-8">
    <meta name="viewport" content="width=device-width, initial-scale=1.0">
    <title>Document</title>
    <script crossorigin src="https://unpkg.com/react@18/umd/react.development.js" defer></script>
    <script crossorigin src="https://unpkg.com/react-dom@18/umd/react-dom.development.js" defer></script>
    <script src="app.js" defer></script>
</head>
<body>
</body>
</html>
```

Ici pour cette exemple est de pouvoir afficher bonjour tout le monde.

On va créer le 1er élément :

-h1 : on cible de dom

-{} : on va lui passer des expressions qui seront associées aux attributs html.

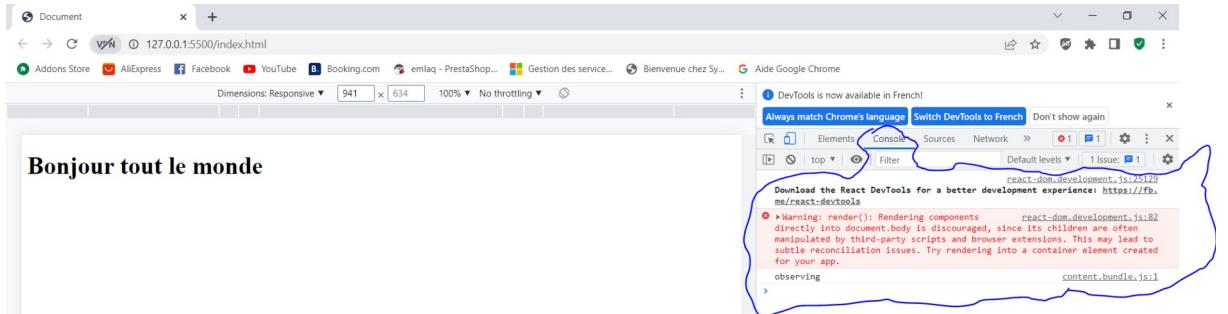
- On met les enfants, donc les enfants ça peut être d'autre élément react, ou ça peut tout simplement du texte.

The screenshot shows the Visual Studio Code interface. In the Explorer sidebar, there is a folder named "EXO1" containing "app.js" and "index.html". The "index.html" file is open in the main editor, displaying the following code:

```
<!DOCTYPE html>
<html lang="en">
<head>
<meta charset="UTF-8">
<meta name="viewport" content="width=device-width, initial-scale=1.0">
<title>Document</title>
<script crossorigin src="https://unpkg.com/react@18/umd/react.development.js" defer></script>
<script crossorigin src="https://unpkg.com/react-dom@18/umd/react-dom.development.js" defer></script>
<script src="app.js" defer></script>
</head>
<body>
</body>
</html>
```

The screenshot shows the Visual Studio Code interface. In the Explorer sidebar, there is a folder named "TUTOREACT" containing "app.js" and "index.html". The "app.js" file is open in the main editor, displaying the following code:

```
const title=React.createElement('h1',{},'Bonjour tout le monde');
ReactDOM.render(title,document.body);
```



On a aussi une petite erreur ou on nous explique que c'est déconseiller de faire ça.

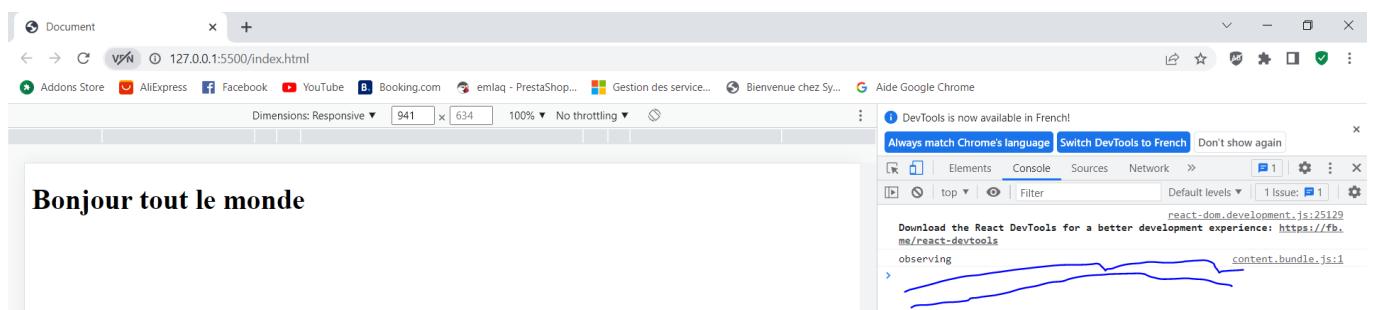
Pour éviter ce problème-là, on va tout simplement créer dans le code index.html une div qui a comme id app et on branchera notre élément react dans cette élément-là.

The screenshot shows the Visual Studio Code interface with the title bar "index.html - tutoreact - Visual Studio Code". The Explorer sidebar on the left shows files like "Get Started", "index.html", and "app.js". The main editor area displays the following HTML code:

```
<!DOCTYPE html>
<html lang="en">
  <head>
    <meta charset="UTF-8">
    <meta http-equiv="X-UA-Compatible" content="IE=edge">
    <meta name="viewport" content="width=device-width, initial-scale=1.0">
    <title>Document</title>
    <script crossorigin src="https://unpkg.com/react@16/umd/react.development.js" defer></script>
    <script crossorigin src="https://unpkg.com/react-dom@16/umd/react-dom.development.js" defer></script>
    <script src="app.js" defer></script>
  </head>
  <body>
    <div id="app"></div>
  </body>
</html>
```

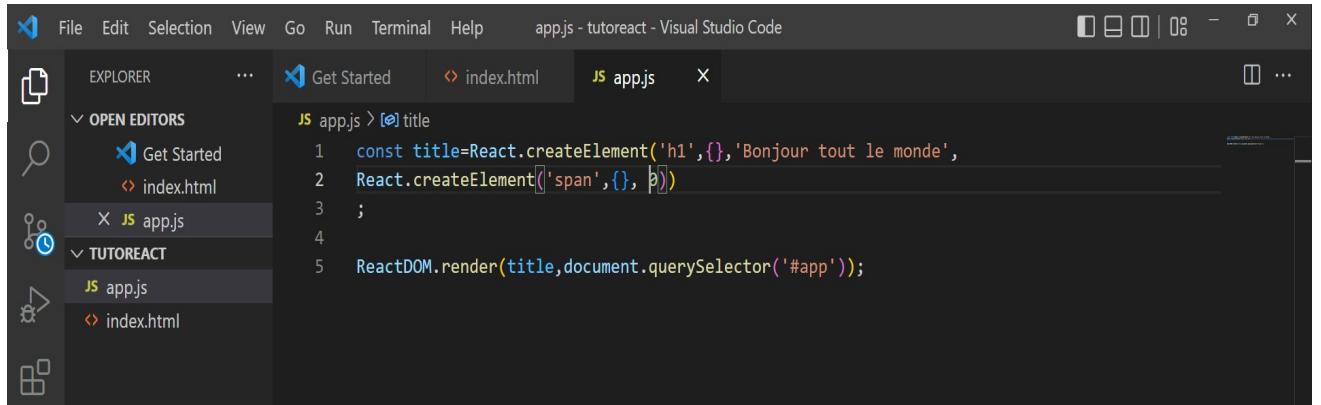
The screenshot shows the Visual Studio Code interface with the title bar "app.js - tutoreact - Visual Studio Code". The Explorer sidebar shows files like "Get Started", "index.html", and "app.js". The main editor area displays the following JavaScript code:

```
const title=React.createElement('h1',{},'Bonjour tout le monde');
ReactDOM.render(title,document.querySelector('#app'));
```

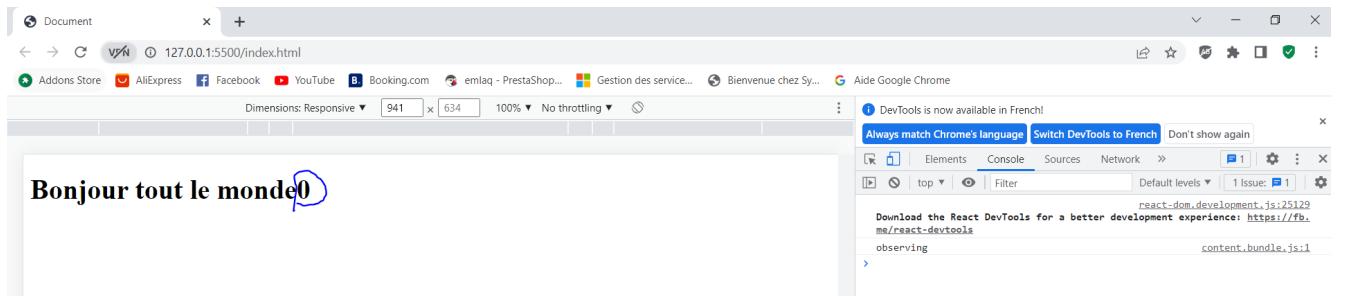


La fonction `render()` est considérée comme une fonction normale mais en réalité elle doit toujours retourner quelque chose. Lorsque le fichier composant est appelé, il appelle par défaut la méthode `render()` parce que ce composant doit afficher le balisage HTML (qu'on peut qualifier de syntaxe JSX).

Vous pouvez aussi créer d'autres éléments par ex qui sera une span et une propriété et ensuite un autre texte



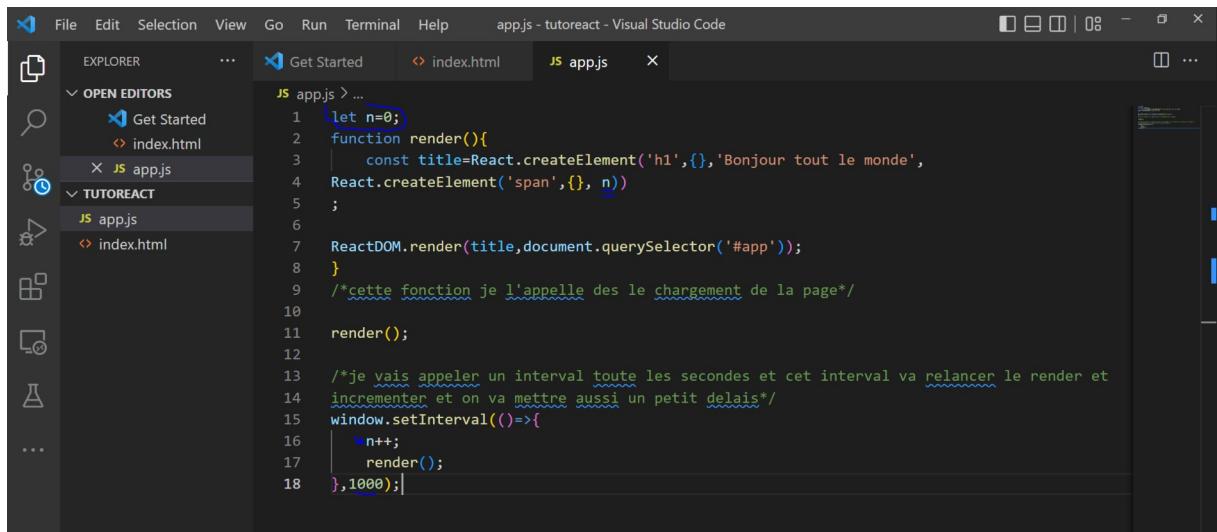
```
const title=React.createElement('h1',{},'Bonjour tout le monde', React.createElement('span',{}, 0));ReactDOM.render(title,document.querySelector('#app'));
```



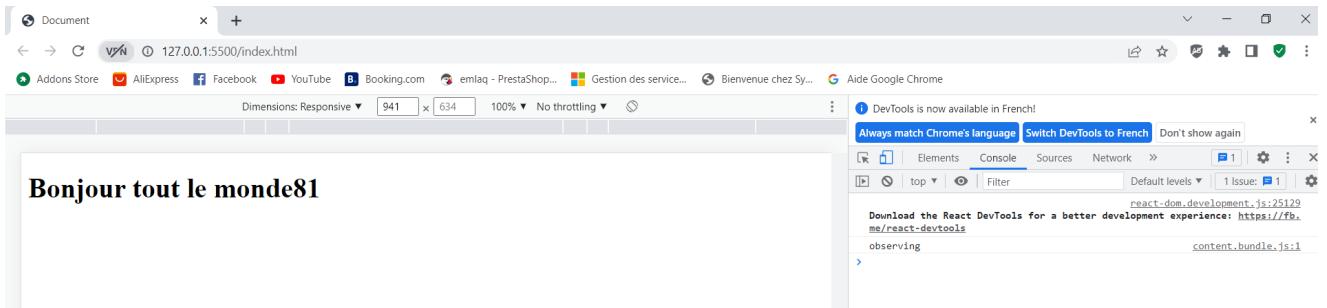
Mais j'ai envie que ça s'incrémente toutes les secondes.

Donc pour incrémenter les choses je vais lui demander de sauvegarder un numéro, incrémenter ce numéro et de réappeler cette fonction :

Je prends l'ancienne partie je la sauvegarde dans une fonction



```
let n=0;
function render(){
  const title=React.createElement('h1',{},'Bonjour tout le monde', React.createElement('span',{}, n));
  ReactDOM.render(title,document.querySelector('#app'));
}
/* cette fonction je l'appelle dès le chargement de la page */
render();
/* je vais appeler un interval toutes les secondes et cet interval va relancer le render et
incrémenter et on va mettre aussi un petit délai */
window.setInterval(()=>{
  n++;
  render();
},1000);
```



La syntaxe JSX

Qu'est-ce que JSX ?

- ➔ JSX signifie JavaScript XML.
- ➔ JSX nous permet d'écrire du HTML dans React.
- ➔ JSX facilite l'écriture et l'ajout de HTML dans React.

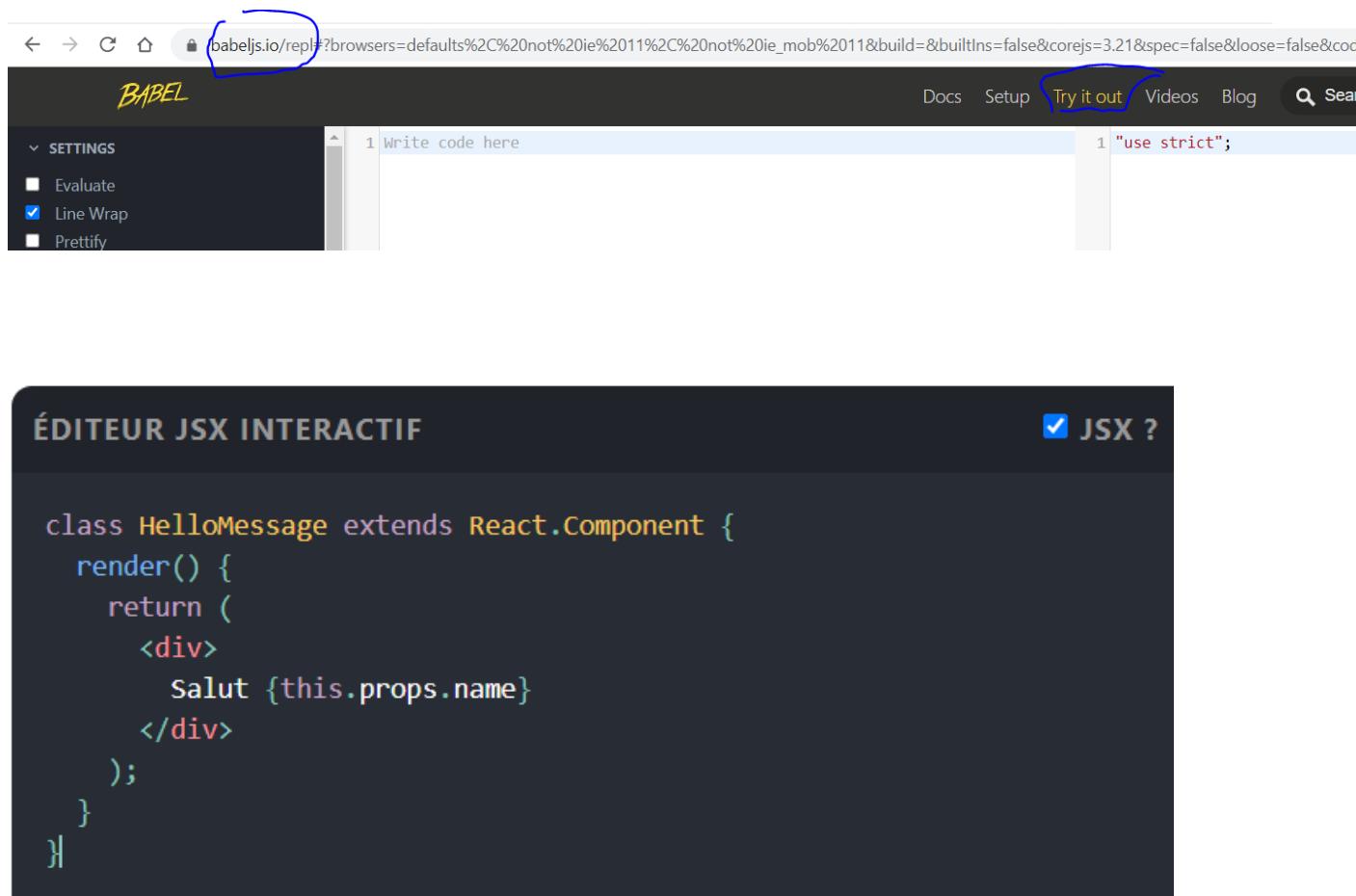
Ça nous permet d'écrire du react un tout petit peu tout simplement, et d'éviter d'avoir à utiliser la méthode `React.createElement` à chaque fois.

Le problème, le jsx est une syntaxe propre à react, et qui n'est pas forcément supporté par les navigateurs et qui ne sera probablement jamais.

Donc il nous faut un outil qui va permettre de convertir cette syntaxe là en JavaScript classique, donc cet outil là il s'appelle `babel`.

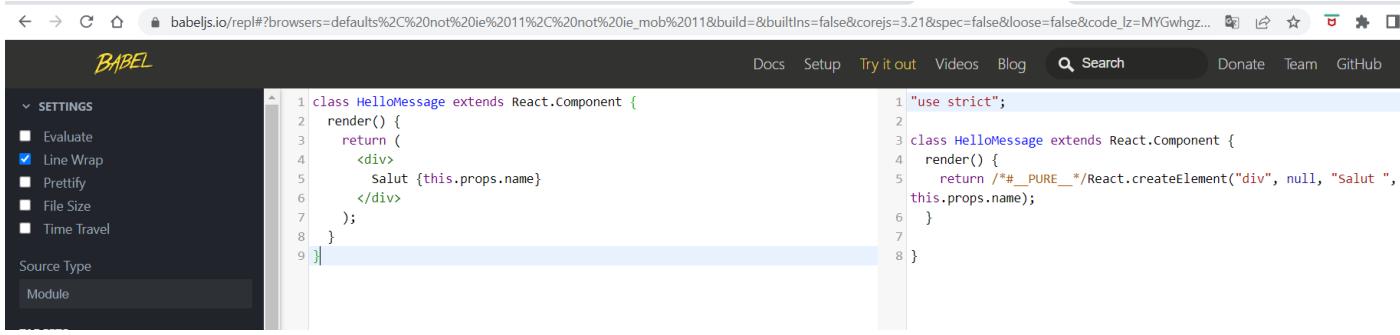
Babel cet outil plus générique, qui permet de convertir le code écrit en JavaScript moderne et qui sortira le code qui fonctionnera sur les navigateurs un tout petit peu plus anciens, et il supporte différentes fonctionnalités dont notamment la possibilité d'écrire du jsx.

Ex :



The screenshot shows the Babel REPL interface. At the top, there's a browser-like header with a URL bar containing "babeljs.io/repl/?browsers=defaults%2C%20not%20ie%2011%2C%20not%20ie_mob%2011&build=&builtIns=false&corejs=3.21&spec=false&loose=false&code=". Below the header is a dark-themed UI with "BABEL" branding. On the left, a sidebar titled "SETTINGS" has three options: "Evaluate" (unchecked), "Line Wrap" (checked), and "Prettify" (unchecked). The main area has a text input field with the placeholder "Write code here". To the right of the input field is a code editor window. The first line of code in the editor is "1 \"use strict\";" with a line number "1" to its left. Below the editor, the text "ÉDITEUR JSX INTERACTIF" is displayed, followed by a checked checkbox labeled "JSX ?".

```
class HelloMessage extends React.Component {  
  render() {  
    return (  
      <div>  
        Salut {this.props.name}  
      </div>  
    );  
  }  
}  
  
class HelloMessage extends React.Component {  
  render() {  
    return (  
      <div>  
        Salut {this.props.name}  
      </div>  
    );  
  }  
}
```



The screenshot shows the Babel REPL interface at babeljs.io/repl/#?browsers=defaults%2C%20not%20ie%2011%2C%20not%20ie_mob%2011&build=&builtIns=false&corejs=3.21&spec=false&loose=false&code_lz=MYGwhgz.... The left sidebar has 'SETTINGS' expanded, with 'Line Wrap' checked. The right pane shows two snippets of code. The first snippet is JSX:

```
1 class HelloMessage extends React.Component {  
2   render() {  
3     return (  
4       <div>  
5         salut {this.props.name}  
6       </div>  
7     );  
8   }  
9 }
```

The second snippet is the resulting JavaScript output:

```
1 "use strict";  
2  
3 class HelloMessage extends React.Component {  
4   render() {  
5     return /*#__PURE__*/React.createElement("div", null, "Salut ",  
6       this.props.name);  
7   }  
8 }
```

→ Pour essayer le jsx, on peut encore utiliser une version qui va fonctionner directement sur le navigateur

Si vous aller dans la documentation du react (reactjs.org), on va vous parler de babel, vous avez la possibilité de charge sur JavaScript qui tient vraiment de faire fonctionner babel au niveau du navigateur.

On prend le lien babel et le mettre au niveau de index.html au-dessus de mon app.js

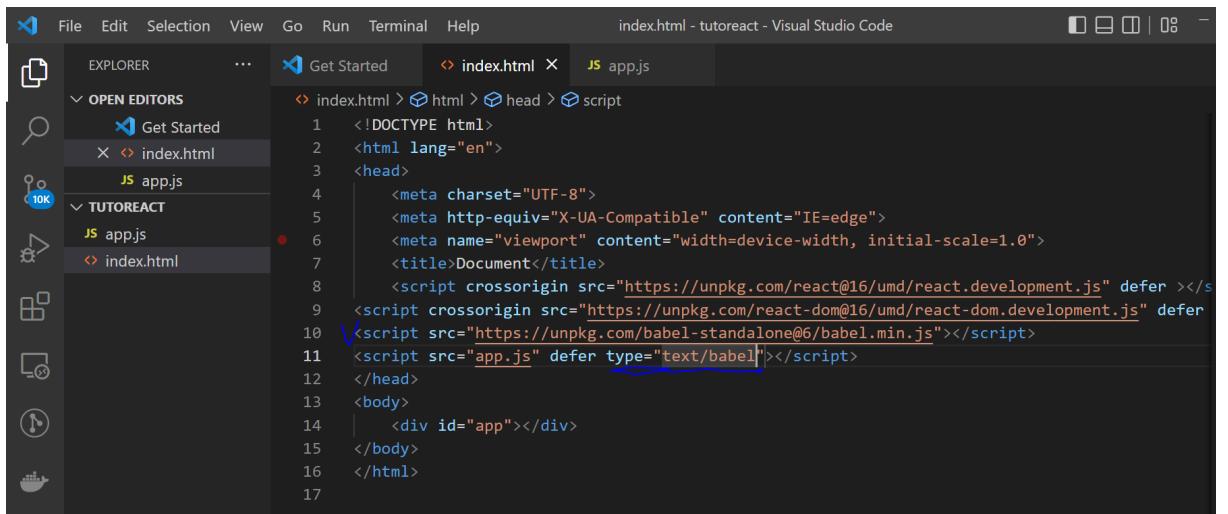
La façon la plus simple et rapide de tester JSX dans votre projet est d'ajouter la balise `<script>` ci-dessous à votre page :

Vous pouvez désormais utiliser JSX dans les fichiers chargés par n'importe quelle balise `<script>` simplement en lui ajoutant l'attribut `type="text/babel"` pour dire que ce script devra être converti avec babel. Vous pouvez tester l'exemple contenant un fichier HTML utilisant JSX.

```

<script
src="https://unpkg.com/@babel/standalone/babel.min.js"></script>

```



The screenshot shows the Visual Studio Code interface. The left sidebar (Explorer) lists files: 'Get Started', 'index.html', 'JS app.js', and 'TUTOREACT' which contains 'JS app.js' and 'index.html'. The right panel shows the content of 'index.html'. The code includes a script tag for Babel and a script tag for React. A specific line of code is highlighted: ''. The code editor has a dark theme.

```

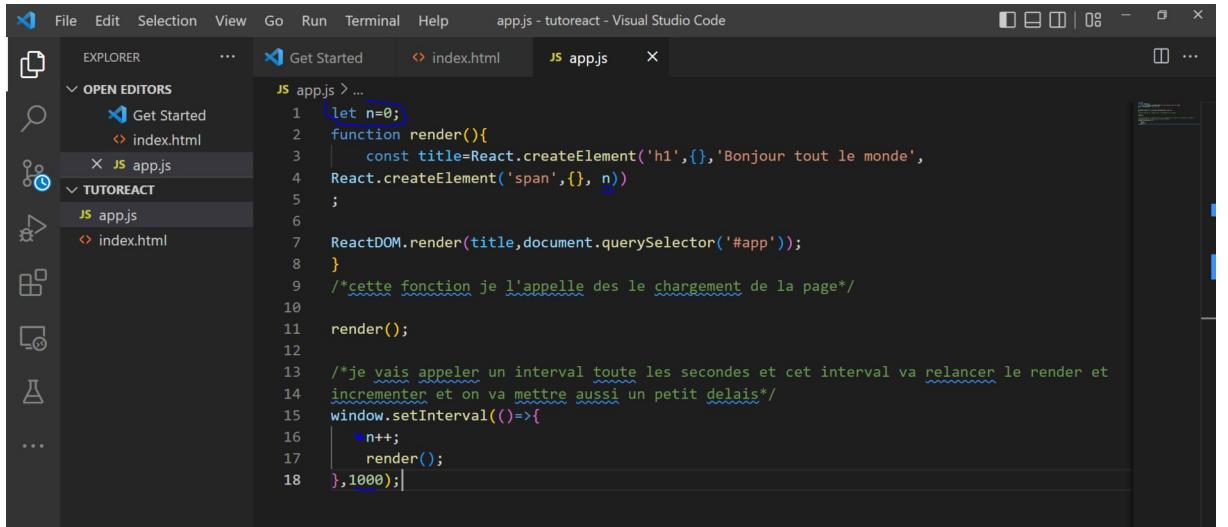
<!DOCTYPE html>
<html lang="en">
<head>
<meta charset="UTF-8">
<meta http-equiv="X-UA-Compatible" content="IE=edge">
<meta name="viewport" content="width=device-width, initial-scale=1.0">
<title>Document</title>
<script crossorigin src="https://unpkg.com/react@16/umd/react.development.js" defer ></script>
<script crossorigin src="https://unpkg.com/react-dom@16/umd/react-dom.development.js" defer ></script>
<script src="https://unpkg.com/babel-standalone@6/babel.min.js"></script>
<script src="app.js" defer type="text/babel"></script>
</head>
<body>
| <div id="app"></div>
</body>
</html>

```

A partir de là je peux commencer à utiliser la syntaxe jsx à l'intérieur de mon fichier JavaScript.

De manière générale quand un fichier utilise la syntaxe jsx, on aura la tendance à renommer le fichier avec l'extension jsx pour facilement faire la différence et aussi aider l'éditeur à comprendre qui va utiliser la syntaxe jsx.

Mais dans cet exemple ce ne pas nécessaire de faire, on peut directement commencer à travailler dessus



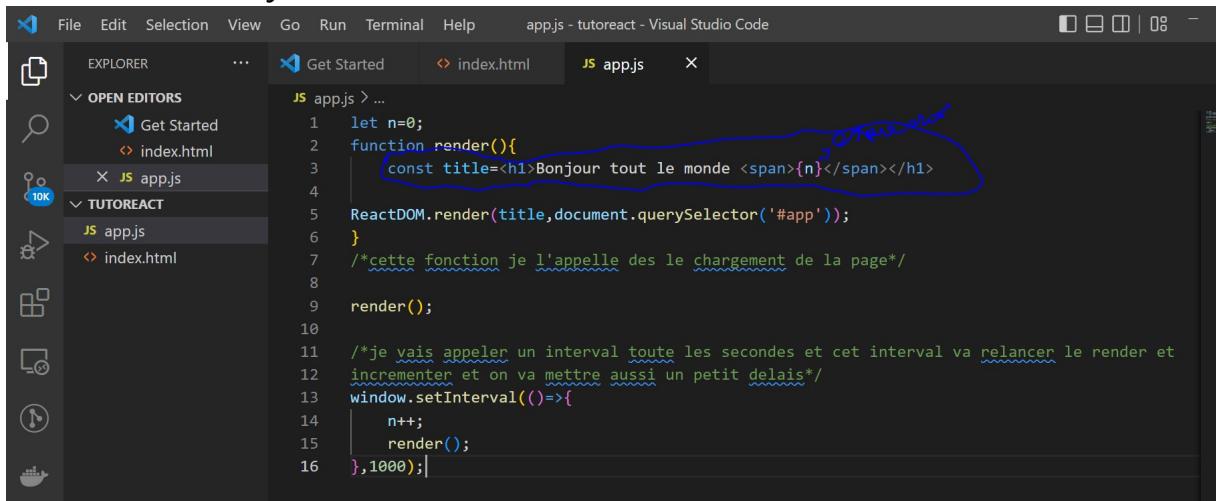
The screenshot shows the Visual Studio Code interface. The left sidebar (Explorer) lists files: 'Get Started', 'index.html', 'JS app.js', and 'TUTOREACT' which contains 'JS app.js' and 'index.html'. The right panel shows the content of 'JS app.js'. The code uses JSX syntax. A specific line of code is highlighted: '/*cette fonction je l'appelle des le chargement de la page*/'. The code editor has a dark theme.

```

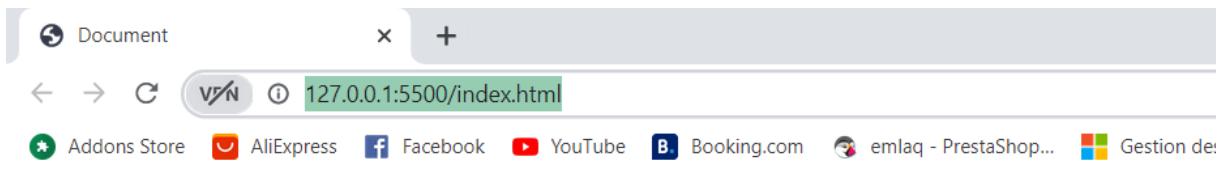
let n=0;
function render(){
  const title=React.createElement('h1',{},'Bonjour tout le monde',
  React.createElement('span',{},n))
  ;
  ReactDOM.render(title,document.querySelector('#app'));
}
/*cette fonction je l'appelle des le chargement de la page*/
render();
/*
je vais appeler un interval toute les secondes et cet interval va relancer le render et
incrementer et on va mettre aussi un petit delais*/
window.setInterval(()=>{
  n++;
  render();
},1000);

```

On le modifie en jsx :



```
JS app.js > ...
1 let n=0;
2 function render(){
3     const title=<h1>Bonjour tout le monde <span>{n}</span></h1>
4     ReactDOM.render(title,document.querySelector('#app'));
5 }
6 /*cette fonction je l'appelle des le chargement de la page*/
7
8 render();
9
10 /*je vais appeler un interval toute les secondes et cet interval va relancer le render et
11 incrementer et on va mettre aussi un petit delais*/
12 window.setInterval(()=>{
13     n++;
14     render();
15 },1000);
```



Bonjour tout le monde 316

Codage JSX

JSX nous permet d'écrire des éléments HTML en JavaScript et de les placer dans le DOM sans aucune méthode createElement() et/ou appendChild().

JSX convertit les balises HTML en éléments de react.

Vous n'êtes pas obligé d'utiliser JSX, mais JSX facilite l'écriture d'applications React.

Voici deux exemples. Le premier utilise JSX et le second non :

Un autre ex jsx :

Example 1

JSX:

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

const myElement = <h1>I Love JSX!</h1>

const root = ReactDOM.createRoot(document.getElementById('root'));
root.render(myElement);
```

localhost:3000
I Love JSX!

Example 2

Without JSX:

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

const myElement = React.createElement('h1', {}, 'I do not use JSX!');

const root = ReactDOM.createRoot(document.getElementById('root'));
root.render(myElement);
```

localhost:3000
I do not use JSX!

Expressions in JSX

Expressions dans JSX

Avec JSX, vous pouvez écrire des expressions entre accolades {}.

L'expression peut être une variable ou une propriété React, ou toute autre expression JavaScript valide. JSX exécutera l'expression et renverra le résultat :



The screenshot shows a browser window with the address bar set to "localhost:3000". The main content area displays the text "React is 10 times better". Above the browser window, there is some code:

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

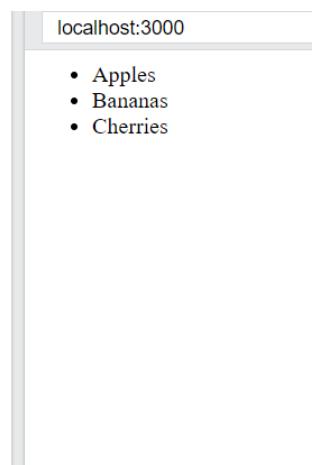
const myElement = <h1>React is {5 + 5} times better with JSX</h1>

const root = ReactDOM.createRoot(document.getElementById('root'));
root.render(myElement);
```

Inserting a Large Block of HTML

Insertion d'un gros bloc de HTML

Pour écrire du HTML sur plusieurs lignes, mettez le HTML entre parenthèses :



The screenshot shows a browser window with the address bar set to "localhost:3000". The main content area displays an unordered list: "• Apples", "• Bananas", and "• Cherries". Above the browser window, there is some code:

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

const myElement = (
  <ul>
    <li>Apples</li>
    <li>Bananas</li>
    <li>Cherries</li>
  </ul>
);

const root = ReactDOM.createRoot(document.getElementById('root'));
root.render(myElement);
```

One Top Level Element

Un élément de niveau supérieur

Le code HTML doit être encapsulé dans UN élément de niveau supérieur.

Donc, si vous aimez écrire deux paragraphes, vous devez les mettre à l'intérieur d'un élément parent, comme un élément div.

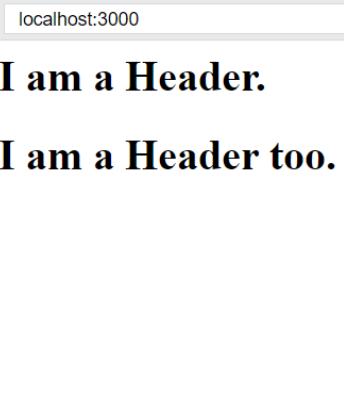
Exemple

Enveloppez deux paragraphes dans un élément DIV :

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

const myElement = (
  <div>
    <h1>I am a Header.</h1>
    <h1>I am a Header too.</h1>
  </div>
);

const root = ReactDOM.createRoot(document.getElementById('root'));
root.render(myElement);
```



JSX lancera une erreur si le HTML n'est pas correct ou s'il manque un élément parent au HTML.

Alternativement, vous pouvez utiliser un "fragment" pour envelopper plusieurs lignes. Cela évitera d'ajouter inutilement des nœuds supplémentaires au DOM.

Un fragment ressemble à une balise HTML vide : <></>.

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

const myElement = (
  <>
    <p>I am a paragraph.</p>
    <p>I am a paragraph too.</p>
  </>
);

const root = ReactDOM.createRoot(document.getElementById('root'));
root.render(myElement);
```

localhost:3000
I am a paragraph.
I am a paragraph too.

Elements Must be Closed

Les éléments doivent être fermés

JSX suit les règles XML et, par conséquent, les éléments HTML doivent être correctement fermés.

Exemple

Fermez les éléments vides avec />

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

const myElement = <input type="text" />

const root = ReactDOM.createRoot(document.getElementById('root'));
root.render(myElement);
```

localhost:3000

JSX will throw an error if the HTML is not properly closed.

Attribute class = className

Attribut classe = className

L'attribut class est un attribut très utilisé en HTML, mais comme JSX est rendu en JavaScript et que le mot-clé class est un mot réservé en JavaScript, vous n'êtes pas autorisé à l'utiliser dans JSX.

Utilisez plutôt l'attribut className.

JSX a résolu ce problème en utilisant className à la place. Lorsque JSX est rendu, il traduit les attributs className en attributs de classe.

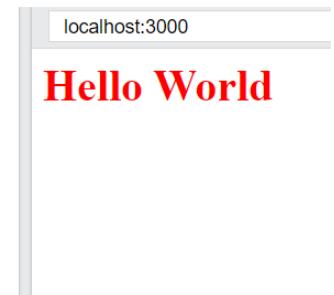
Exemple

Utilisez l'attribut className au lieu de class dans JSX :

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

const myElement = <h1 className="myclass">Hello World</h1>

const root = ReactDOM.createRoot(document.getElementById('root'));
root.render(myElement);
```



Conditions - if statements

Conditions - si les instructions

React prend en charge les instructions if, mais pas dans JSX.

Pour pouvoir utiliser des instructions conditionnelles dans JSX, vous devez placer les instructions if en dehors de JSX, ou vous pouvez utiliser une expression ternaire à la place :

Option 1:

Écrivez des instructions if en dehors du code JSX :

Exemple

Écrivez "Bonjour" si x est inférieur à 10, sinon "Au revoir":

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

const x = 5;
let text = "Goodbye";
if (x < 10) {
  text = "Hello";
}

const myElement = <h1>{text}</h1>

const root = ReactDOM.createRoot(document.getElementById('root'));
root.render(myElement);
```

localhost:3000

Hello

Option 2:

Utilisez plutôt des expressions ternaires :

Exemple

Écrivez "Bonjour" si x est inférieur à 10, sinon "Au revoir":

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

const x = 5;

const myElement = <h1>{(x) < 10 ? "Hello" : "Goodbye"}</h1>

const root = ReactDOM.createRoot(document.getElementById('root'));
root.render(myElement);
```

localhost:3000

Hello

Notez que pour intégrer une expression JavaScript dans JSX, le JavaScript doit être entouré d'accolades, {}.

Cette approche est acceptable pour se former ou réaliser des démos simples. Cependant, elle va ralentir l'affichage de votre site, elle n'est donc **pas adaptée pour la production**. Lorsque vous serez prêt·e à aller plus loin, supprimez la balise `<script>` et l'attribut `type="text/babel"` que vous venez d'ajouter. Dans la section suivante, nous verrons plutôt comment configurer le préprocesseur JSX afin de convertir automatiquement toutes les balises `<script>`.

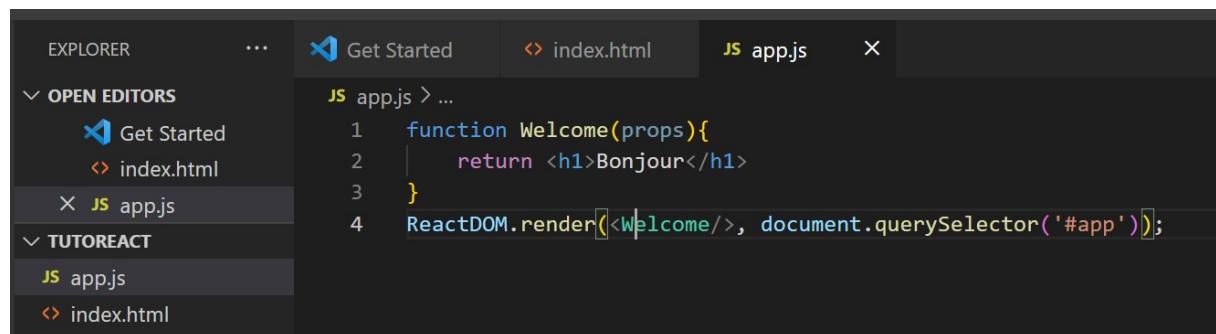
React Components

Notre premier composant

Dans ce point nous allons parler des composants, vous allez voir les composants c'est la base du fonctionnement du travail avec react et ce qui fait véritablement sa force.

Donc ce que je vous propose c'est de supprimer tout et de repartir à zero (app.js)

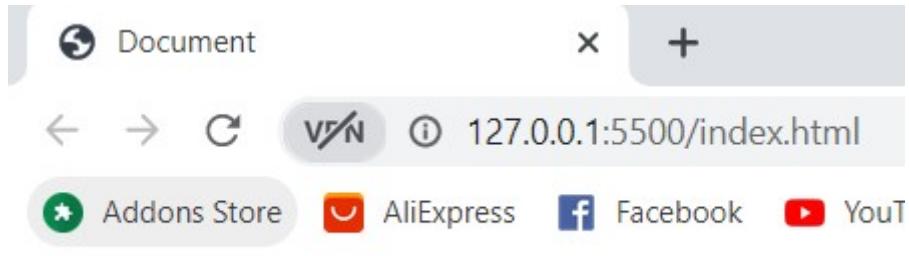
Notre idée ici c'est de créer un titre « Bonjour » suivi du nom de la personne. On crée une fonction comme un composant :



The screenshot shows a code editor interface with the following details:

- EXPLORER**: Shows files: Get Started, index.html, app.js (selected).
- OPEN EDITORS**: Shows files: Get Started, index.html, app.js (selected).
- TUTOREACT**: Shows files: app.js, index.html.
- Get Started**: Preview tab.
- index.html**: Preview tab.
- JS app.js**: Editor tab containing the following code:

```
function Welcome(props){  
  return <h1>Bonjour</h1>  
}  
ReactDOM.render(<Welcome/>, document.querySelector('#app'));
```

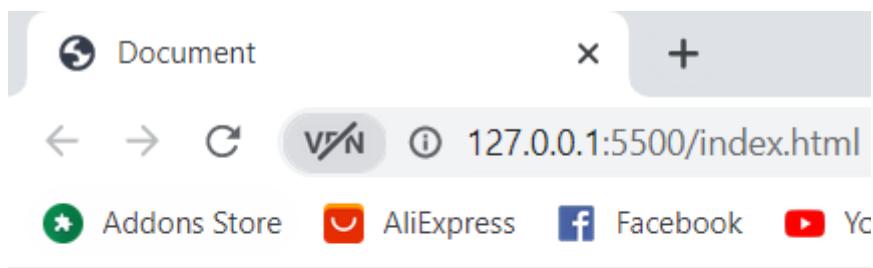


Bonjour

Création des éléments dynamique en utilisant l'argument props avec l'attribut name :

A screenshot of a code editor. On the left, there's an "EXPLORER" sidebar with "OPEN EDITORS" containing "Get Started", "index.html", and "app.js". Under "TUTORIALS", there are "app.js" and "index.html". The main panel shows a file named "app.js" with the following code:

```
JS app.js > Welcome
1 function Welcome(props){
2   return <h1>Bonjour [props.name]</h1>
3 }
4 ReactDOM.render(<Welcome name="Jean"/>, document.querySelector('#app'));
```



Bonjour Jean

Les composants sont comme des fonctions qui renvoient des éléments HTML.

Composants de react

Les composants sont des morceaux de code indépendants et réutilisables. Elles ont le même objectif que les fonctions JavaScript, mais fonctionnent de manière isolée et renvoient du HTML

Create Your First Component

Créez votre premier composant

Lors de la création d'un composant React, le nom du composant DOIT commencer par une lettre majuscule.

Composant de classe

Un composant de classe doit inclure l'instruction extend React.Component. Cette instruction crée un héritage pour React.Component et donne à votre composant l'accès aux fonctions de React.Component.

Le composant nécessite également une méthode render(), cette méthode renvoie HTML.

Exemple

Créer un composant Class appelé Car

```
class Car extends React.Component {  
  render() {  
    return <h2>Hi, I am a Car!</h2>;  
  }  
}
```

Composant de fonction

Voici le même exemple que ci-dessous, mais créé à l'aide d'un composant Function à la place.

Un composant Function renvoie également HTML et se comporte à peu près de la même manière qu'un composant Class, mais les composants Function peuvent être écrits en utilisant beaucoup moins de code, sont plus faciles à comprendre

Exemple

Créer un composant Function appelé Car

```
function Car() {  
  return <h2>Hi, I am a Car!</h2>;  
}
```

Rendering a Component

Rendu d'un composant

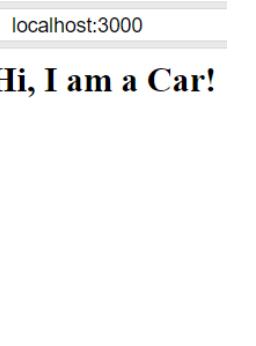
Maintenant, votre application React a un composant appelé Car, qui renvoie un élément <h2>.

Pour utiliser ce composant dans votre application, utilisez une syntaxe similaire au HTML normal : <Car />

Exemple

Affichez le composant Car dans l'élément "root":

```
import React from 'react';  
import ReactDOM from 'react-dom/client';  
  
function Car() {  
  return <h2>Hi, I am a Car!</h2>;  
}  
  
const root = ReactDOM.createRoot(document.getElementById('root'));  
root.render(<Car />);
```



Props

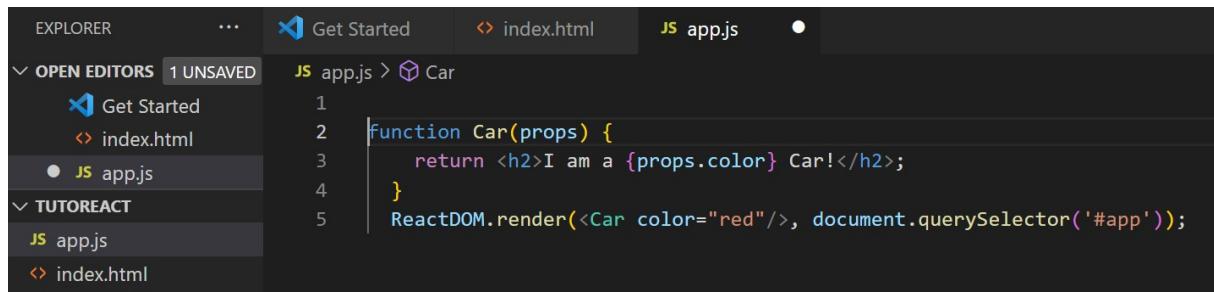
Accessoires

Les composants peuvent être passés en tant qu'accessoires, ce qui signifie propriétés.

Les accessoires(props) sont comme des arguments de fonction, et vous les envoyez dans le composant en tant qu'attributs.

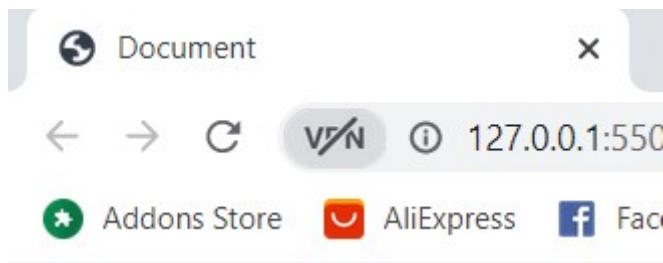
Exemple

Utilisez un attribut pour passer une couleur au composant Car et utilisez-le dans la fonction render() :



```
EXPLORER      ...  Get Started  index.html  JS app.js  ●
OPEN EDITORS  1 UNSAVED  JS app.js > Car
Get Started
index.html
● JS app.js
TUTOREACT
JS app.js
index.html
```

```
1
2  function Car(props) {
3      return <h2>I am a {props.color} Car!</h2>;
4  }
5  ReactDOM.render(<Car color="red"/>, document.querySelector('#app'));
```



I am a red Car!

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

function Car(props) {
  return <h2>I am a {props.color} Car!</h2>;
}

const root = ReactDOM.createRoot(document.getElementById('root'));
root.render(<Car color="red"/>);
```

localhost:3000

I am a red Car!

Components in Components

Composants dans Composants

Nous pouvons faire référence à des composants à l'intérieur d'autres composants :

Exemple

Utilisez le composant Car dans le composant Garage :

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

function Car() {
  return <h2>I am a Car!</h2>;
}

function Garage() {
  return (
    <>
      <h1>Who lives in my Garage?</h1>
      <Car />
    </>
  );
}

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

localhost:3000

Who lives in my Garage?

I am a Car!

Components in Files

Composants dans les fichiers

React consiste à réutiliser le code, et il est recommandé de diviser vos composants en fichiers séparés.

Pour ce faire, créez un nouveau fichier avec une extension de fichier .js et insérez-y le code :

Notez que le nom du fichier doit commencer par une majuscule.

Exemple

Voici le nouveau fichier, nous l'avons nommé "Car.js":

```
function Car() {  
  return <h2>Hi, I am a Car!</h2>;  
}  
  
export default Car;
```

Pour pouvoir utiliser le composant Car, vous devez importer le fichier dans votre application.

Exemple

Maintenant, nous importons le fichier "Car.js" dans l'application, et nous pouvons utiliser le composant Car comme s'il avait été créé ici.

```
import React from 'react';  
import ReactDOM from 'react-dom/client';  
import Car from './Car.js';  
  
const root = ReactDOM.createRoot(document.getElementById('root'));  
root.render(<Car />);
```

localhost:3000

Hi, I am a C

React Class

React Class Components

Composants de la classe React

Avant React 16.8, les composants de classe étaient le seul moyen de suivre l'état et le cycle de vie d'un composant React. Les composants fonctionnels étaient considérés comme "sans état".

Composants de react

Les composants sont des morceaux de code indépendants et réutilisables. Elles ont le même objectif que les fonctions JavaScript, mais fonctionnent de manière isolée et renvoient du HTML via une fonction render().

Les composants sont de deux types, les composants de classe et les composants de fonction. Dans ce point, vous découvrirez les composants de classe.

Créer un composant de classe

Lors de la création d'un composant React, le nom du composant doit commencer par une lettre majuscule.

Le composant doit inclure l'instruction extend React.Component, cette instruction crée un héritage à React.Component et donne à votre composant l'accès aux fonctions de React.Component.

Le composant nécessite également une méthode render(), cette méthode renvoie HTML.

Exemple

Créer un composant Class appelé Car

```
class Car extends React.Component {  
  render() {  
    return <h2>Hi, I am a Car!</h2>;  
  }  
}
```

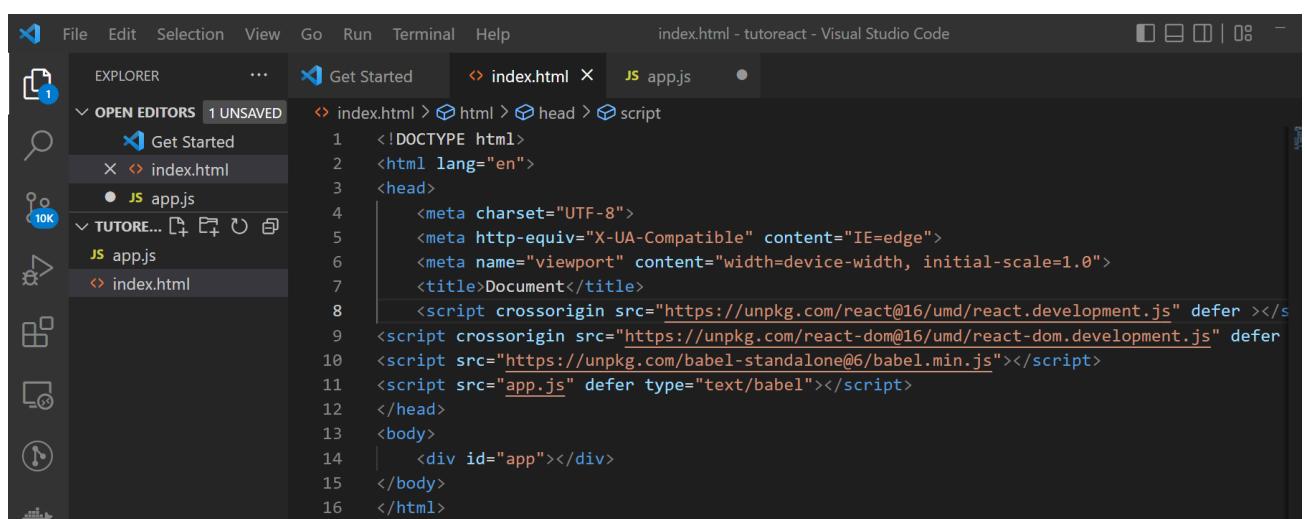
```
    }  
}
```

Maintenant, votre application React a un composant appelé Car, qui renvoie un élément <h2>.

Pour utiliser ce composant dans votre application, utilisez une syntaxe similaire au HTML normal : <Car />

Exemple

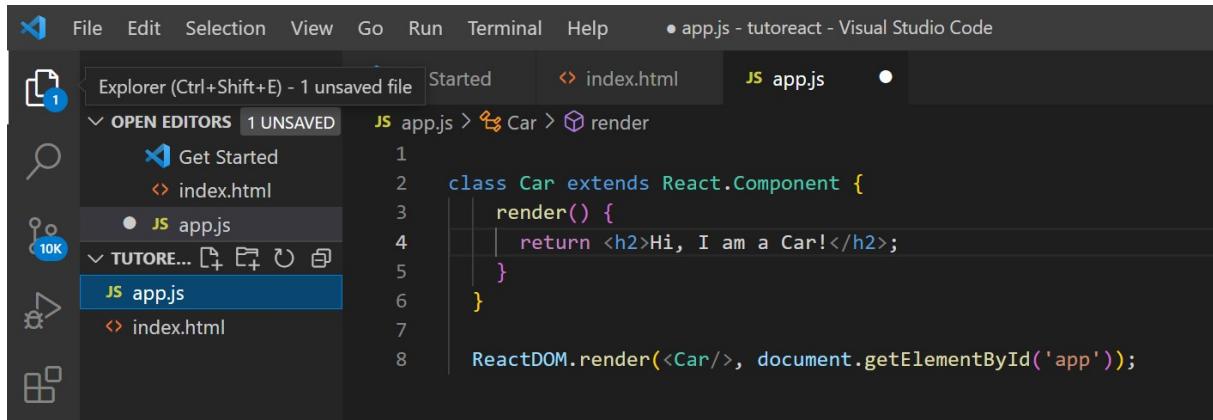
Affichez le composant Car dans l'élément "app":



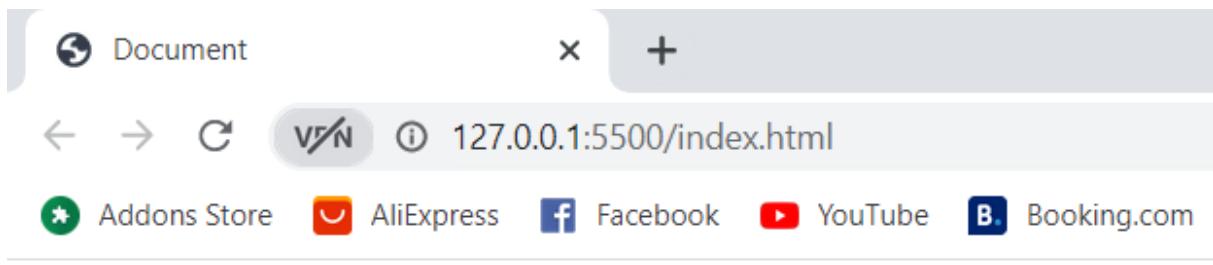
The screenshot shows the Visual Studio Code interface with the following details:

- File Bar:** File, Edit, Selection, View, Go, Run, Terminal, Help.
- Title Bar:** index.html - tutoreact - Visual Studio Code.
- Explorer Panel:** Shows 1 UNSAVED file: "Get Started". Under "OPEN EDITORS", there are "index.html" (HTML) and "app.js" (JS). Under "TUTORIELS", there are "app.js" and "index.html".
- Editor Area:** The "index.html" tab is active, displaying the following code:

```
<!DOCTYPE html>  
<html lang="en">  
  <head>  
    <meta charset="UTF-8">  
    <meta http-equiv="X-UA-Compatible" content="IE=edge">  
    <meta name="viewport" content="width=device-width, initial-scale=1.0">  
    <title>Document</title>  
    <script crossorigin src="https://unpkg.com/react@16/umd/react.development.js" defer></script>  
    <script crossorigin src="https://unpkg.com/react-dom@16/umd/react-dom.development.js" defer></script>  
    <script src="https://unpkg.com/babel-standalone@6/babel.min.js"></script>  
    <script src="app.js" defer type="text/babel"></script>  
  </head>  
  <body>  
    <div id="app"></div>  
  </body>  
</html>
```



```
File Edit Selection View Go Run Terminal Help • app.js - tutoreact - Visual Studio Code
Explorer (Ctrl+Shift+E) - 1 unsaved file Started index.html JS app.js
OPEN EDITORS 1 UNSAVED JS app.js > Car > render
Get Started index.html JS app.js
TUTOR... JS app.js index.html
1
2 class Car extends React.Component {
3     render() {
4         return <h2>Hi, I am a Car!</h2>;
5     }
6 }
7
8 ReactDOM.render(<Car/>, document.getElementById('app'));
```



Hi, I am a Car!

Ou bien

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

class Car extends React.Component {
    render() {
        return <h2>Hi, I am a Car!</h2>;
    }
}

ReactDOM.render(<Car />, document.getElementById('root'));
```

localhost:3000

Hi, I am a Car!

Component Constructor

Constructeur de composants

S'il y a une fonction `constructor()` dans votre composant, cette fonction sera appelée lorsque le composant sera lancé.

La fonction `constructor` est l'endroit où vous initiez les propriétés du composant.

Dans React, les propriétés des composants doivent être conservées dans un objet appelé `state`.

Vous en apprendrez plus sur `state` plus loin.

La fonction `constructor` est également l'endroit où vous respectez l'héritage du composant parent en incluant l'instruction `super()`, qui exécute la fonction constructeur du composant parent, et votre composant a accès à toutes les fonctions du composant parent (`React.Component`).

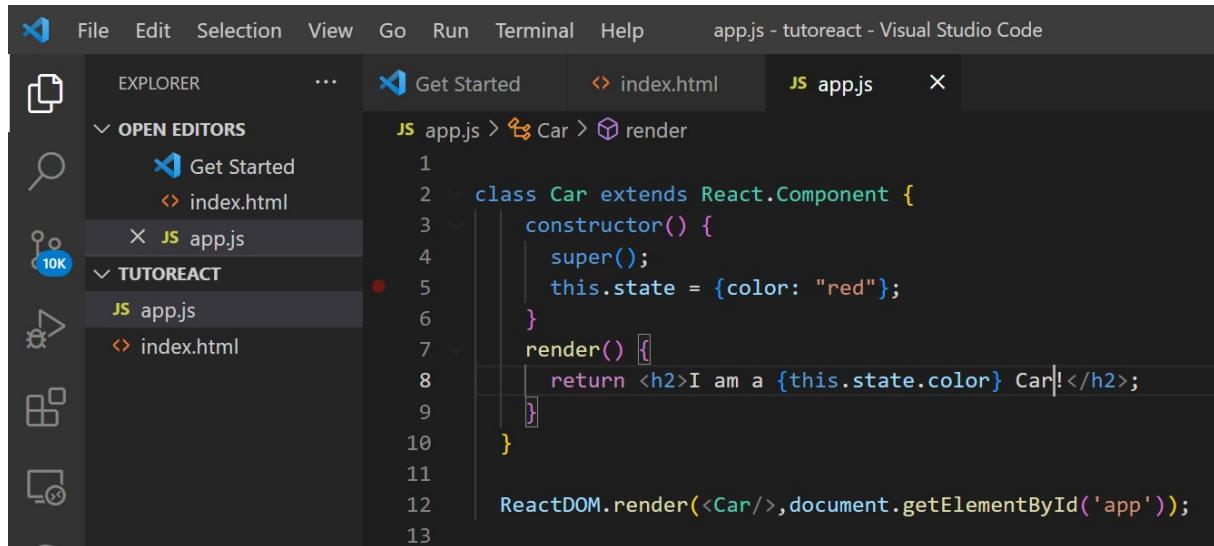
Exemple

Créez une fonction `constructor` dans le composant `Car` et ajoutez une propriété de couleur :

```
class Car extends React.Component {  
  constructor() {  
    super();  
    this.state = {color: "red"};  
  }  
  render() {  
    return <h2>I am a Car!</h2>;  
  }  
}
```

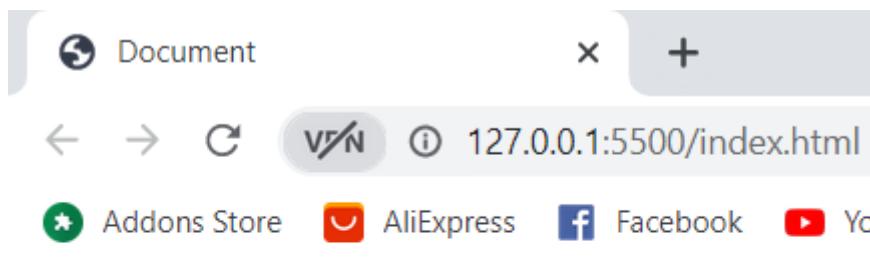
Utilisez la propriété color dans la fonction render() :

Example



The screenshot shows the Visual Studio Code interface. The title bar says "app.js - tutoreact - Visual Studio Code". The left sidebar has icons for Explorer, Search, and 10K. The "OPEN EDITORS" section lists "Get Started", "index.html", and "JS app.js". The "TUTOREACT" section lists "JS app.js" and "index.html", with a red dot next to it. The main editor area shows the following code:

```
1 class Car extends React.Component {
2     constructor() {
3         super();
4         this.state = {color: "red"};
5     }
6     render() {
7         return <h2>I am a {this.state.color} Car!</h2>;
8     }
9 }
10 ReactDOM.render(<Car/>,document.getElementById('app'));
11
12
13
```



I am a red Car!

Ou bien

```

import React from 'react';
import ReactDOM from 'react-dom/client';

class Car extends React.Component {
  constructor() {
    super();
    this.state = {color: "red"};
  }
  render() {
    return <h2>I am a {this.state.color} Car!</h2>;
  }
}

const root = ReactDOM.createRoot(document.getElementById('root'));
root.render(<Car />);

```

localhost:3000
I am a red Car!

Props

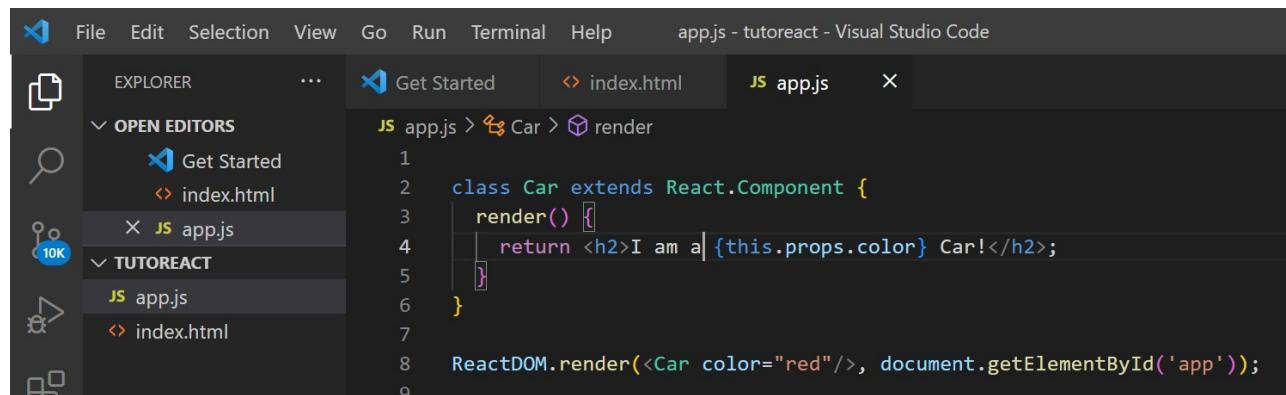
Une autre façon de gérer les propriétés des composants consiste à utiliser **props**.

Les **props** sont comme des arguments de fonction, et vous les envoyez dans le composant en tant qu'attributs.

Vous en apprendrez plus sur les **props** dans le point suivant.

Exemple

Utilisez un attribut pour passer une couleur au composant Car et utilisez-le dans la fonction render() :



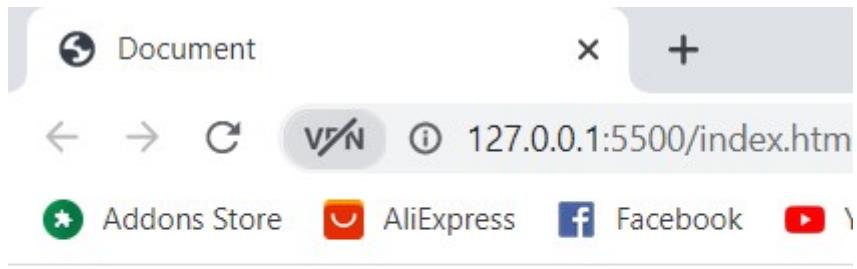
The screenshot shows the Visual Studio Code interface with the following details:

- File Bar:** File, Edit, Selection, View, Go, Run, Terminal, Help, app.js - tutoreact - Visual Studio Code
- Explorer Bar:** OPEN EDITORS: Get Started, index.html, JS app.js (highlighted), TUTOREACT: JS app.js, index.html
- Code Editor:**

```

1  class Car extends React.Component {
2    render() [
3      return <h2>I am a {this.props.color} Car!</h2>;
4    ]
5  }
6
7
8  ReactDOM.render(<Car color="red"/>, document.getElementById('app'));

```

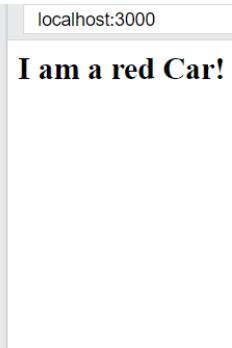


Ou bien

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

class Car extends React.Component {
  render() {
    return <h2>I am a {this.props.color} Car!</h2>;
  }
}

ReactDOM.render(<Car color="red"/>, document.getElementById('root'));
```

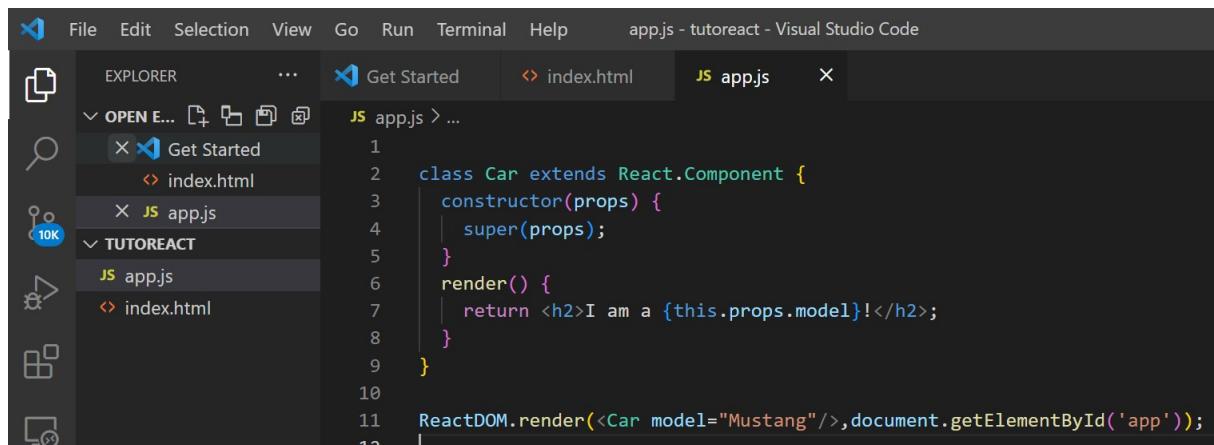


Props in the Constructor

Props dans le constructeur

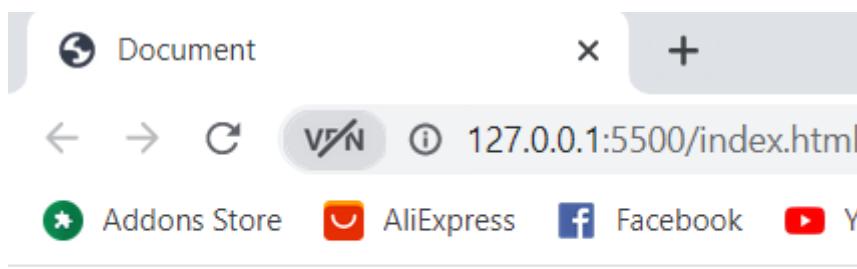
Si votre composant a une fonction de constructeur, props doivent toujours être passés au constructeur et également au React.Component via la méthode super().

Exemple



The screenshot shows the Visual Studio Code interface. The top menu bar includes File, Edit, Selection, View, Go, Run, Terminal, and Help. The title bar says "app.js - tutoreact - Visual Studio Code". The Explorer sidebar on the left shows a tree structure with "OPEN E...", "Get Started", "index.html", "JS app.js", "TUTOREACT", and "JS app.js" under it, with a "10K" badge. The main editor area displays the following code:

```
1 class Car extends React.Component {
2     constructor(props) {
3         super(props);
4     }
5     render() {
6         return <h2>I am a {this.props.model}!</h2>;
7     }
8 }
9 ReactDOM.render(<Car model="Mustang"/>,document.getElementById('app'));
```



I am a Mustang!

Ou bien

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

class Car extends React.Component {
    constructor(props) {
        super(props);
    }
    render() {
        return <h2>I am a {this.props.model}!</h2>;
    }
}

const root = ReactDOM.createRoot(document.getElementById('root'));
root.render(<Car model="Mustang"/>);
```

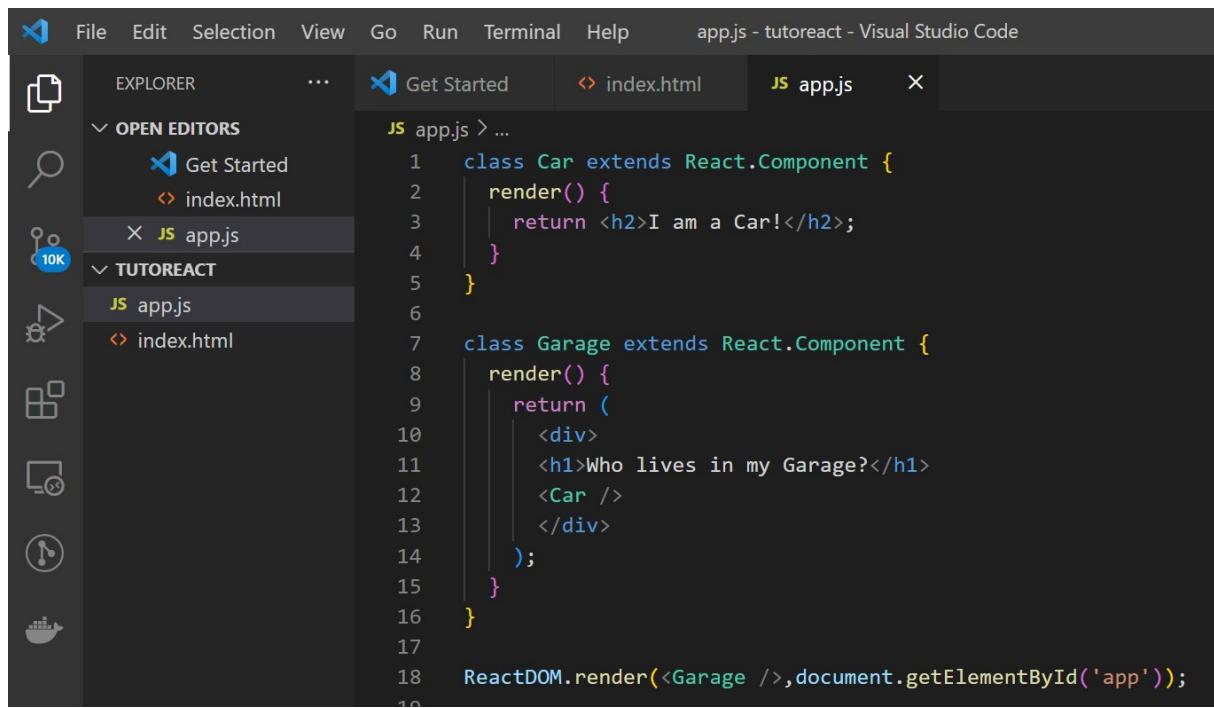
localhost:3000
I am a Mustang!

Components in Components

Nous pouvons faire référence à des composants à l'intérieur d'autres composants :

Exemple

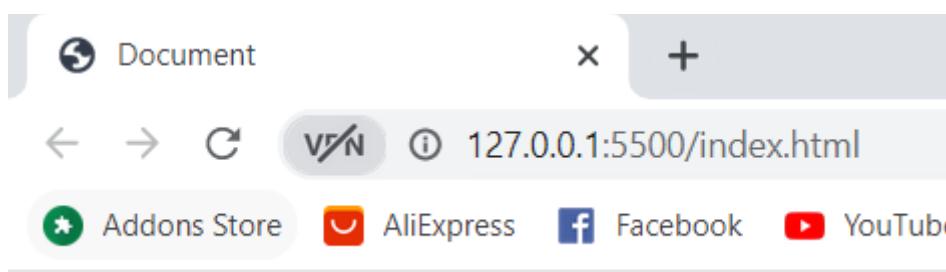
Utilisez le composant Car dans le composant Garage :



```
class Car extends React.Component {
  render() {
    return <h2>I am a Car!</h2>;
  }
}

class Garage extends React.Component {
  render() {
    return (
      <div>
        <h1>Who lives in my Garage?</h1>
        <Car />
      </div>
    );
  }
}

ReactDOM.render(<Garage />, document.getElementById('app'));
```



Who lives in my Garage?

I am a Car!

Ou bien

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

class Car extends React.Component {
  render() {
    return <h2>I am a Car!</h2>;
  }
}

class Garage extends React.Component {
  render() {
    return (
      <div>
        <h1>Who lives in my Garage?</h1>
        <Car />
      </div>
    );
  }
}

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

localhost:3000

Who lives in my Garage?

I am a Car!

React Class Component State

Les composants de la classe React ont un objet **state** intégré.

Vous avez peut-être remarqué que nous avons utilisé state plus tôt dans la section constructeur de composants.

L'objet state est l'endroit où vous stockez les valeurs de propriété qui appartiennent au composant.

Lorsque l'objet **state** change, le composant restitue.

Creating the state Object

L'objet **state** est initialisé dans le constructeur :

Exemple

Spécifiez l'objet d'état dans la méthode du constructeur :

```
class Car extends React.Component {
```

```
constructor(props) {
  super(props);
  this.state = {brand: "Ford"};
}

render() {
  return (
    <div>
      <h1>My Car</h1>
    </div>
  );
}

}
```

L'objet **state** peut contenir autant de propriétés que vous le souhaitez :

Exemple

Spécifiez toutes les propriétés dont votre composant a besoin :

```
class Car extends React.Component {
  constructor(props) {
    super(props);
    this.state = {
      brand: "Ford",
      model: "Mustang",
      color: "red",
      year: 1964
    };
}
```

```
        }

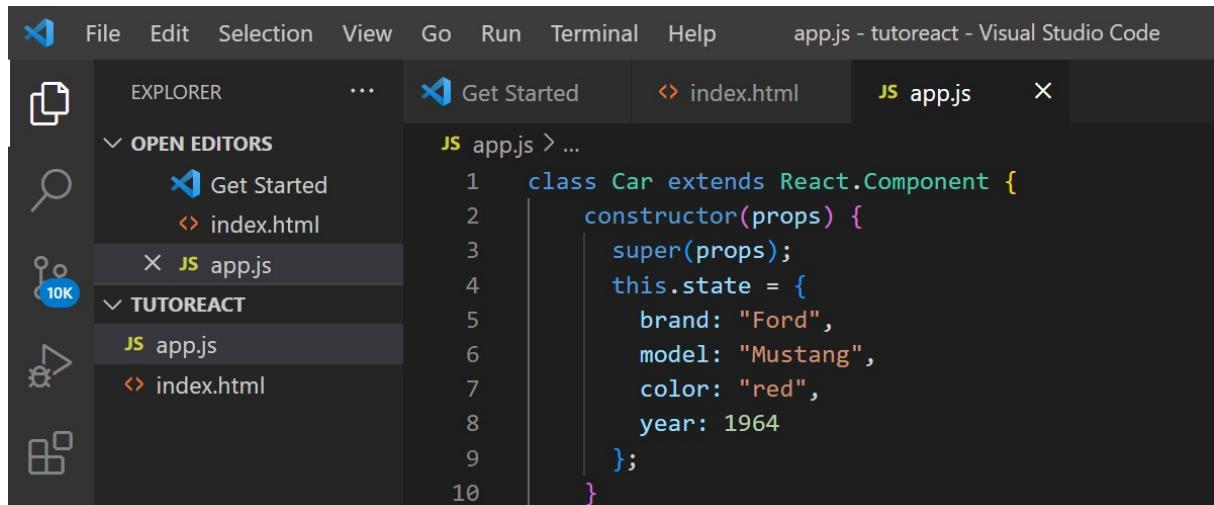
      render() {
        return (
          <div>
            <h1>My Car</h1>
          </div>
        );
      }
    }
  }
```

Using the state Object

Faites référence à l'objet **state** n'importe où dans le composant en utilisant la syntaxe **this.state.propertyname** :

Exemple:

Reportez-vous à l'objet **state** dans la méthode **render()** :



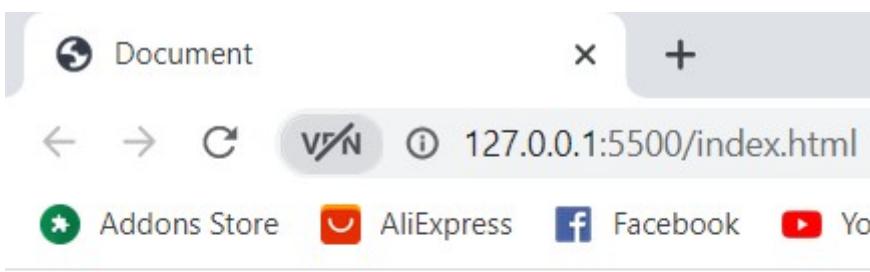
```
File Edit Selection View Go Run Terminal Help app.js - tutoreact - Visual Studio Code

EXPLORER ... Get Started index.html JS app.js X
OPEN EDITORS
  Get Started
  index.html
  X JS app.js
TUTOREACT
  JS app.js
  index.html

JS app.js > ...
1   class Car extends React.Component {
2     constructor(props) {
3       super(props);
4       this.state = {
5         brand: "Ford",
6         model: "Mustang",
7         color: "red",
8         year: 1964
9       };
10    }
```



```
11  render() {
12    return (
13      <div>
14        <h1>My {this.state.brand}</h1>
15        <p>
16          It is a {this.state.color}
17          {this.state.model}
18          from {this.state.year}.
19        </p>
20      </div>
21    );
22  }
23
24
25  ReactDOM.render(<Car />, document.getElementById('app'));
```



My Ford

It is a red Mustang from 1964.

Ou bien

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

class Car extends React.Component {
  constructor(props) {
    super(props);
    this.state = {
      brand: "Ford",
      model: "Mustang",
      color: "red",
      year: 1964
    };
  }
  render() {
    return (
      <div>
        <h1>My {this.state.brand}</h1>
        <p>
          It is a {this.state.color}
          {this.state.model}
          from {this.state.year}.
        </p>
      </div>
    );
  }
}

ReactDOM.render(<Car />, document.getElementById('root'));
```

localhost:3000

My Ford

It is a red Mustang from 1964.

```
class Car extends React.Component {
  constructor(props) {
    super(props);
    this.state = {
      brand: "Ford",
      model: "Mustang",
      color: "red",
      year: 1964
    };
  }
  render() {
    return (
      <div>
        <h1>My{this.state.brand}</h1>
        <p>it is a {this.state.color}
          {this.state.model}
          from{this.state.year}
        </p>
      </div>
    );
  }
}
```

```
}

ReactDOM.render(<Car/>, document.getElementById('app'));
```

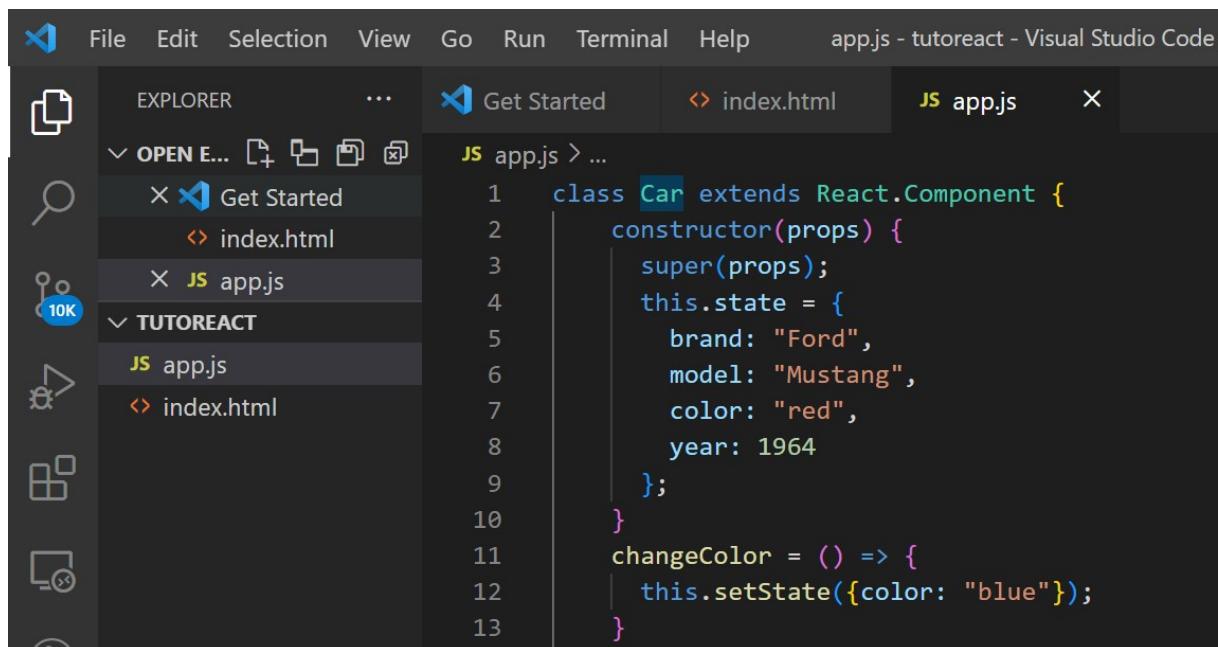
Changing the state Object

Pour modifier une valeur dans l'objet `state`, utilisez la méthode `this.setState()`.

Lorsqu'une valeur dans l'objet `state` change, le composant sera restitué, ce qui signifie que la sortie(output) changera en fonction de la ou des nouvelles valeurs.

Exemple:

Ajoutez un bouton avec un événement `onClick` qui modifiera la propriété `color` :



The screenshot shows the Visual Studio Code interface with the following details:

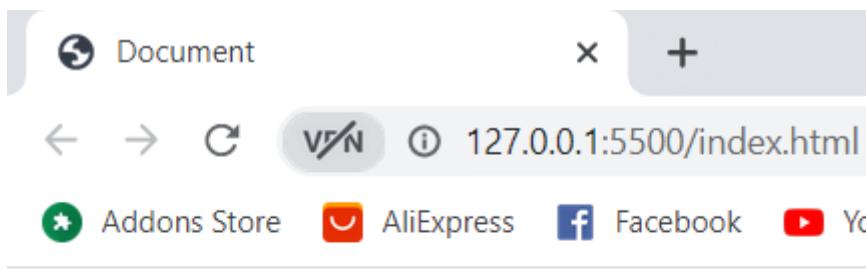
- File Bar:** File, Edit, Selection, View, Go, Run, Terminal, Help.
- Title Bar:** app.js - tutoreact - Visual Studio Code
- Explorer View:** Shows a tree structure with "OPEN E...", "Get Started", "index.html", "JS app.js" (selected), and a folder "TUTOREACT" containing "JS app.js" and "index.html".
- Code Editor:** The "app.js" file is open, showing the following code:

```
1  class Car extends React.Component {
2      constructor(props) {
3          super(props);
4          this.state = {
5              brand: "Ford",
6              model: "Mustang",
7              color: "red",
8              year: 1964
9          };
10     }
11     changeColor = () => {
12         this.setState({color: "blue"});
13     }
}
```

```

14 render() {
15   return (
16     <div>
17       <h1>My {this.state.brand}</h1>
18       <p>
19         It is a {this.state.color}
20         {this.state.model}
21         from {this.state.year}.
22       </p>
23       <button
24         type="button"
25         onClick={this.changeColor}
26       >Change color</button>
27     </div>
28   );
29 }
30
31 ReactDOM.render(<Car />, document.getElementById('app'));
32
33

```



My Ford

It is a blue Mustang from 1964.

[Change color](#)

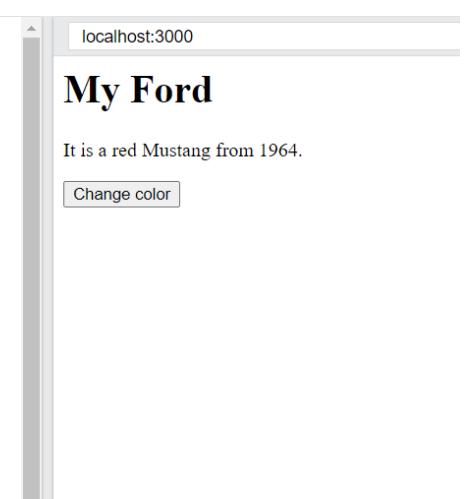
Ou bien

```

import React from 'react';
import ReactDOM from 'react-dom/client';

class Car extends React.Component {
  constructor(props) {
    super(props);
    this.state = {
      brand: "Ford",
      model: "Mustang",
      color: "red",
      year: 1964
    };
  }
  changeColor = () => {
    this.setState({color: "blue"});
  }
}

```



```
render() {
  return (
    <div>
      <h1>My {this.state.brand}</h1>
      <p>
        It is a {this.state.color}
        {this.state.model}
        from {this.state.year}.
      </p>
      <button
        type="button"
        onClick={this.changeColor}
      >Change color</button>
    </div>
  );
}

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

Utilisez toujours la méthode `setState()` pour modifier l'objet `state`, cela garantira que le composant sait qu'il a été mis à jour et appelle la méthode `render()` (et toutes les autres méthodes du cycle de vie).

Lifecycle of Components

Cycle de vie des composants

Chaque composant de React a un cycle de vie que vous pouvez surveiller et manipuler au cours de ses trois phases principales.

Les trois phases sont : le montage(**Mounting**), la mise à jour(**Updating**) et le démontage(**Unmounting**).

Mounting

Le montage signifie mettre des éléments dans le DOM.

React a quatre méthodes intégrées qui sont appelées, dans cet ordre, lors du montage d'un composant :

1. `constructor()`
2. `getDerivedStateFromProps()`
3. `render()`
4. `componentDidMount()`

La méthode `render()` est obligatoire et sera toujours appelée, les autres sont facultatives et seront appelées si vous les définissez.

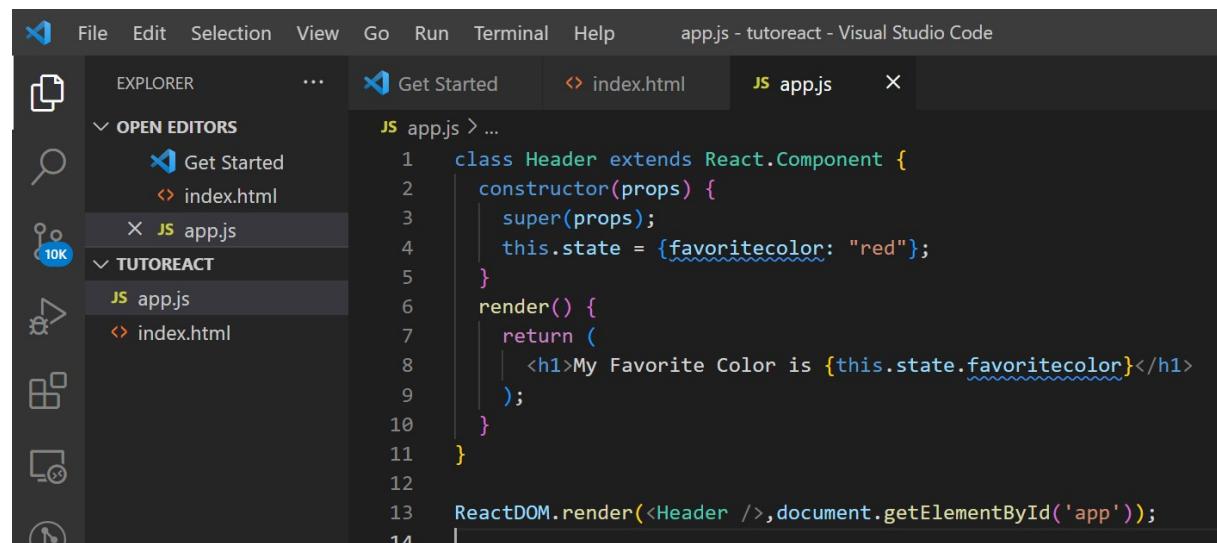
constructor

La méthode `constructor()` est appelée avant toute autre chose, lorsque le composant est lancé, et c'est l'endroit naturel pour configurer `state` initial et d'autres valeurs initiales.

La méthode `constructor()` est appelée avec les `props`, comme arguments, et vous devez toujours commencer par appeler les `super(props)` avant toute autre chose, cela lancera la méthode constructeur du parent et permettra au composant d'hériter des méthodes de son parent (React.Composant).

Exemple:

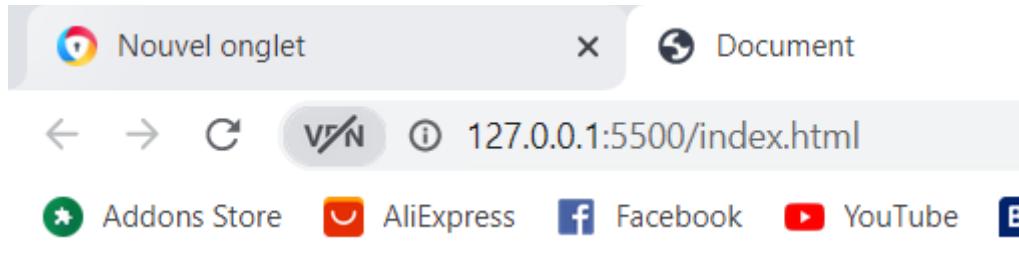
La méthode `constructor()` est appelée, par React, à chaque fois que vous créez un composant :



```
File Edit Selection View Go Run Terminal Help app.js - tutoreact - Visual Studio Code

EXPLORER ... Get Started index.html JS app.js X
OPEN EDITORS
  Get Started
  index.html
  JS app.js
TUTOREACT
  JS app.js
  index.html

JS app.js > ...
1  class Header extends React.Component {
2    constructor(props) {
3      super(props);
4      this.state = {favoritecolor: "red"};
5    }
6    render() {
7      return (
8        <h1>My Favorite Color is {this.state.favoritecolor}</h1>
9      );
10   }
11 }
12
13 ReactDOM.render(<Header />,document.getElementById('app'));
14 |
```



My Favorite Color is red

Ou bien

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

class Header extends React.Component {
  constructor(props) {
    super(props);
    this.state = {favoritecolor: "red"};
  }
  render() {
    return (
      <h1>My Favorite Color is {this.state.favoritecolor}</h1>
    );
  }
}

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



getDerivedStateFromProps

La méthode `getDerivedStateFromProps()` est appelée juste avant de rendre le ou les éléments dans le DOM.

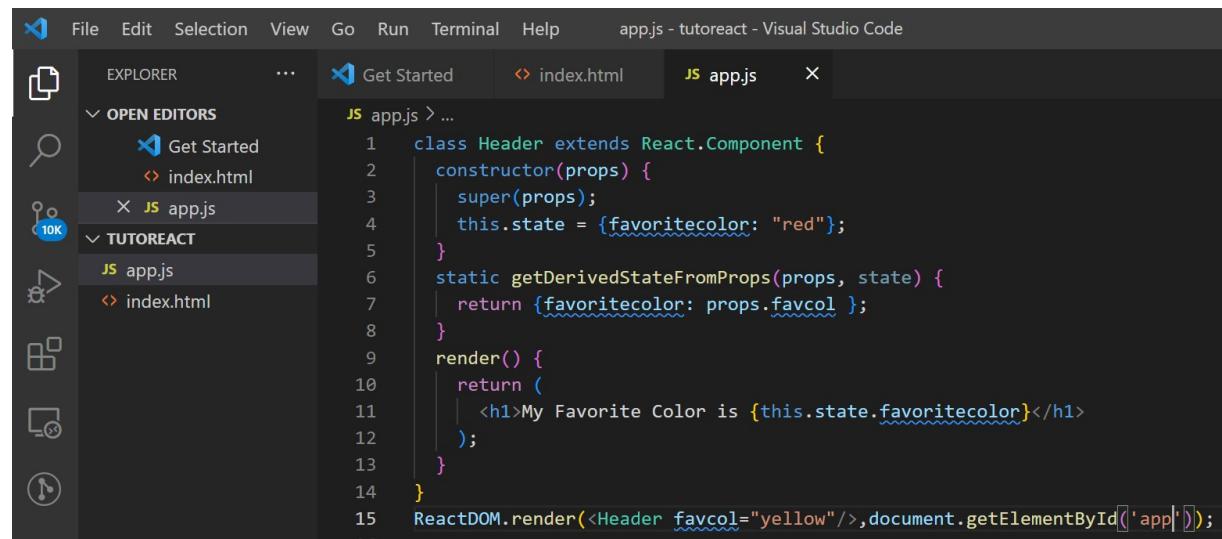
C'est l'endroit naturel pour définir l'objet `state` en fonction des `props` initiaux.

Il prend `state` comme argument et renvoie un objet avec des modifications de `state`.

L'exemple ci-dessous commence avec la couleur préférée étant "rouge", mais la méthode `getDerivedStateFromProps()` met à jour la couleur préférée en fonction de l'attribut `favcol` :

Exemple:

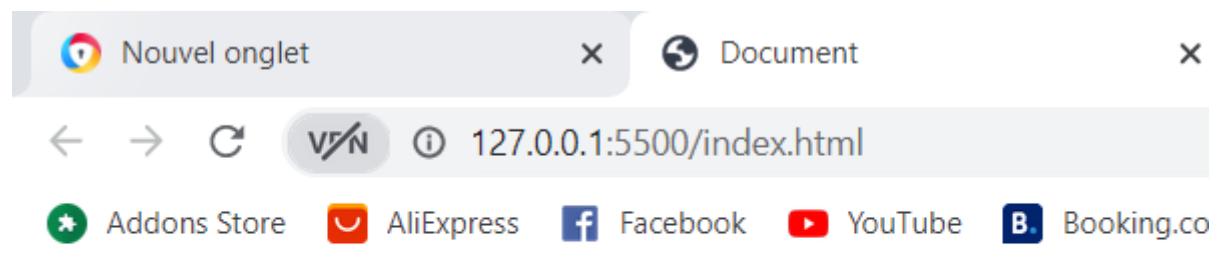
La méthode `getDerivedStateFromProps` est appelée juste avant la méthode `render` :



The screenshot shows the Visual Studio Code interface with the following details:

- File Bar:** File, Edit, Selection, View, Go, Run, Terminal, Help.
- Explorer:** Shows an open folder named "TUTOREACT" containing files "app.js" and "index.html".
- Editor:** The "app.js" file is open, displaying the following code:

```
1  class Header extends React.Component {
2    constructor(props) {
3      super(props);
4      this.state = {favoritecolor: "red"};
5    }
6    static getDerivedStateFromProps(props, state) {
7      return {favoritecolor: props.favcol};
8    }
9    render() {
10      return (
11        <h1>My Favorite Color is {this.state.favoritecolor}</h1>
12      );
13    }
14  }
15 ReactDOM.render(<Header favcol="yellow"/>,document.getElementById('app'));
```



My Favorite Color is yellow

Ou bien



The screenshot shows a browser window with the URL "localhost:3000". The page content is "My Favorite Color is yellow". On the left, there is a code editor window displaying the following React code:

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

class Header extends React.Component {
  constructor(props) {
    super(props);
    this.state = {favoritecolor: "red"};
  }
  static getDerivedStateFromProps(props, state) {
    return {favoritecolor: props.favcol};
  }
  render() {
    return (
      <h1>My Favorite Color is {this.state.favoritecolor}</h1>
    );
  }
}

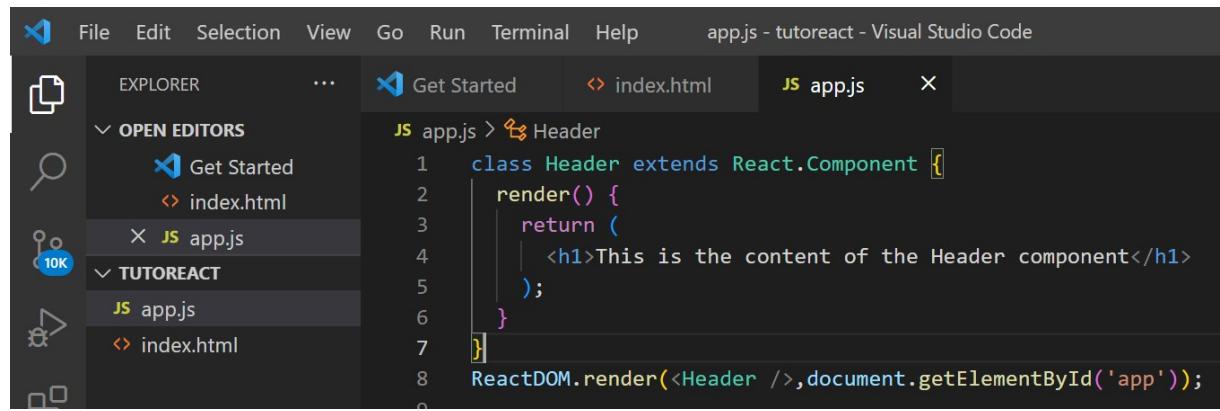
const root = ReactDOM.createRoot(document.getElementById('root'));
root.render(<Header favcol="yellow"/>);
```

render

La méthode `render()` est requise et est la méthode qui génère réellement le code HTML dans le DOM.

Exemple:

Un composant simple avec une méthode `render()` simple :



The screenshot shows the Visual Studio Code interface with the file "app.js" open. The code editor displays the following simple React component:

```
class Header extends React.Component {
  render() {
    return (
      <h1>This is the content of the Header component</h1>
    );
  }
}

ReactDOM.render(<Header />, document.getElementById('app'));
```

Ou bien

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

class Header extends React.Component {
  render() {
    return (
      <h1>This is the content of the Header component</h1>
    );
  }
}

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

localhost:3000

This is the content of the Header component

componentDidMount

La méthode `componentDidMount()` est appelée après le rendu du composant.

C'est là que vous exécutez des instructions qui nécessitent que le composant soit déjà placé dans le DOM.

Exemple:

Au début, ma couleur préférée est le rouge, mais donnez-moi une seconde, et c'est le jaune à la place :

The screenshot shows the Visual Studio Code interface with the file `app.js` open. The code defines a React component `Header` that sets its state to "red" initially and then changes it to "yellow" after a 1-second delay using `setTimeout`. The browser below shows the rendered output: "My Favorite Color is yellow".

```
class Header extends React.Component {
  constructor(props) {
    super(props);
    this.state = {favoritecolor: "red"};
  }
  componentDidMount() {
    setTimeout(() => {
      this.setState({favoritecolor: "yellow"})
    }, 1000)
  }
  render() {
    return (
      <h1>My Favorite Color is {this.state.favoritecolor}</h1>
    );
  }
}
ReactDOM.render(<Header />, document.getElementById('app'));
```

Nouvel onglet Document +

← → ⌛ ⓘ 127.0.0.1:5500/index.html

Addons Store AliExpress Facebook YouTube Booking.com

My Favorite Color is yellow

Ou bien

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

class Header extends React.Component {
  constructor(props) {
    super(props);
    this.state = {favoritecolor: "red"};
  }
  componentDidMount() {
    setTimeout(() => {
      this.setState({favoritecolor: "yellow"})
    }, 1000)
  }
  render() {
    return (
      <h1>My Favorite Color is {this.state.favoritecolor}</h1>
    );
  }
}

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

localhost:3000

My Favorite Color is yellow

Updating

La phase suivante du cycle de vie correspond à la mise à jour d'un composant.

Un composant est mis à jour chaque fois qu'il y a un changement dans **state ou props** du composant.

React a cinq méthodes intégrées qui sont appelées, dans cet ordre, lorsqu'un composant est mis à jour :

1. `getDerivedStateFromProps()`
2. `shouldComponentUpdate()`
3. `render()`
4. `getSnapshotBeforeUpdate()`
5. `componentDidUpdate()`

La méthode `render()` est obligatoire et sera toujours appelée, les autres sont facultatives et seront appelées si vous les définissez.

getDerivedStateFromProps

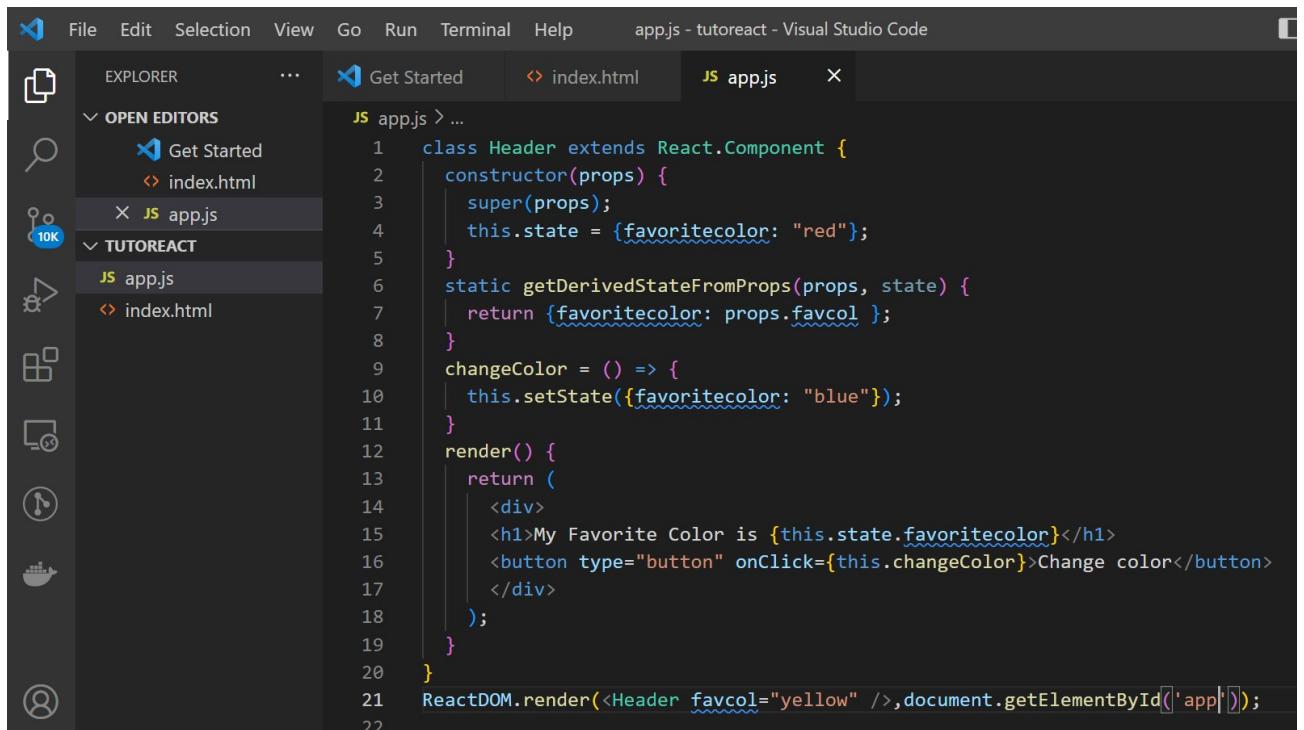
Lors des mises à jour, la méthode `getDerivedStateFromProps` est également appelée. Il s'agit de la première méthode appelée lorsqu'un composant est mis à jour.

C'est toujours l'endroit naturel pour définir l'objet `state` en fonction des `props` initiaux.

L'exemple ci-dessous a un bouton qui change la couleur préférée en bleu, mais puisque la méthode `getDerivedStateFromProps()` est appelée, qui met à jour `state` avec la couleur de l'attribut `favcol`, la couleur préférée est toujours rendue en jaune :

Exemple:

Si le composant est mis à jour, la méthode `getDerivedStateFromProps()` est appelée :



```
File Edit Selection View Go Run Terminal Help app.js - tutoreact - Visual Studio Code

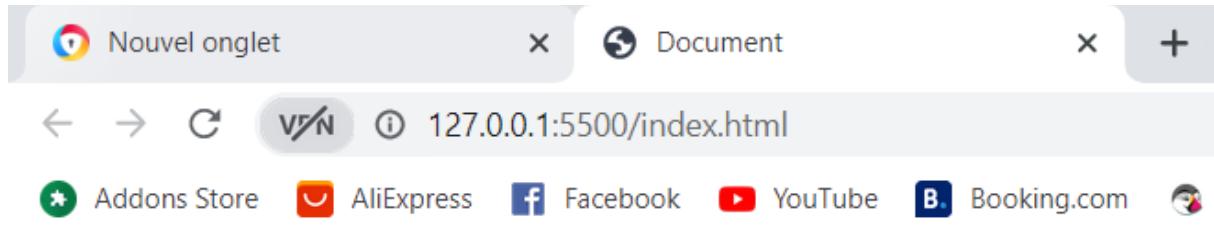
EXPLORER ... JS app.js > ...
OPEN EDITORS Get Started index.html JS app.js X
TUTORIALS 10K JS app.js index.html

JS app.js > ...
1 class Header extends React.Component {
2     constructor(props) {
3         super(props);
4         this.state = {favoritecolor: "red"};
5     }
6     static getDerivedStateFromProps(props, state) {
7         return {favoritecolor: props.favcol};
8     }
9     changeColor = () => {
10        this.setState({favoritecolor: "blue"});
11    }
12    render() {
13        return (
14            <div>
15                <h1>My Favorite Color is {this.state.favoritecolor}</h1>
16                <button type="button" onClick={this.changeColor}>Change color</button>
17            </div>
18        );
19    }
20 }
21 ReactDOM.render(<Header favcol="yellow" />, document.getElementById('app'));
22
```

/*

Cet exemple a un bouton qui change la couleur préférée en bleu, mais puisque la méthode `getDerivedStateFromProps()` est appelée, la couleur préférée est toujours rendue en jaune (parce que la méthode et à jour `state` avec la couleur de l'attribut `favcol`).

*/



My Favorite Color is yellow

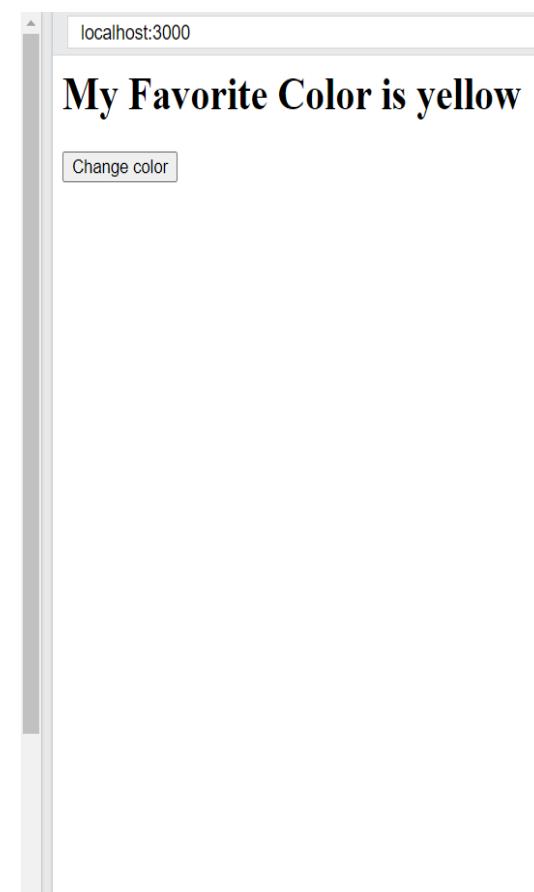
[Change color](#)

Ou bien

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

class Header extends React.Component {
  constructor(props) {
    super(props);
    this.state = {favoritecolor: "red"};
  }
  static getDerivedStateFromProps(props, state) {
    return {favoritecolor: props.favcol};
  }
  changeColor = () => {
    this.setState({favoritecolor: "blue"});
  }
  render() {
    return (
      <div>
        <h1>My Favorite Color is {this.state.favoritecolor}</h1>
        <button type="button" onClick={this.changeColor}>Change color</button>
      </div>
    );
  }
}

const root = ReactDOM.createRoot(document.getElementById('root'));
root.render(<Header favcol="yellow" />);
```



shouldComponentUpdate

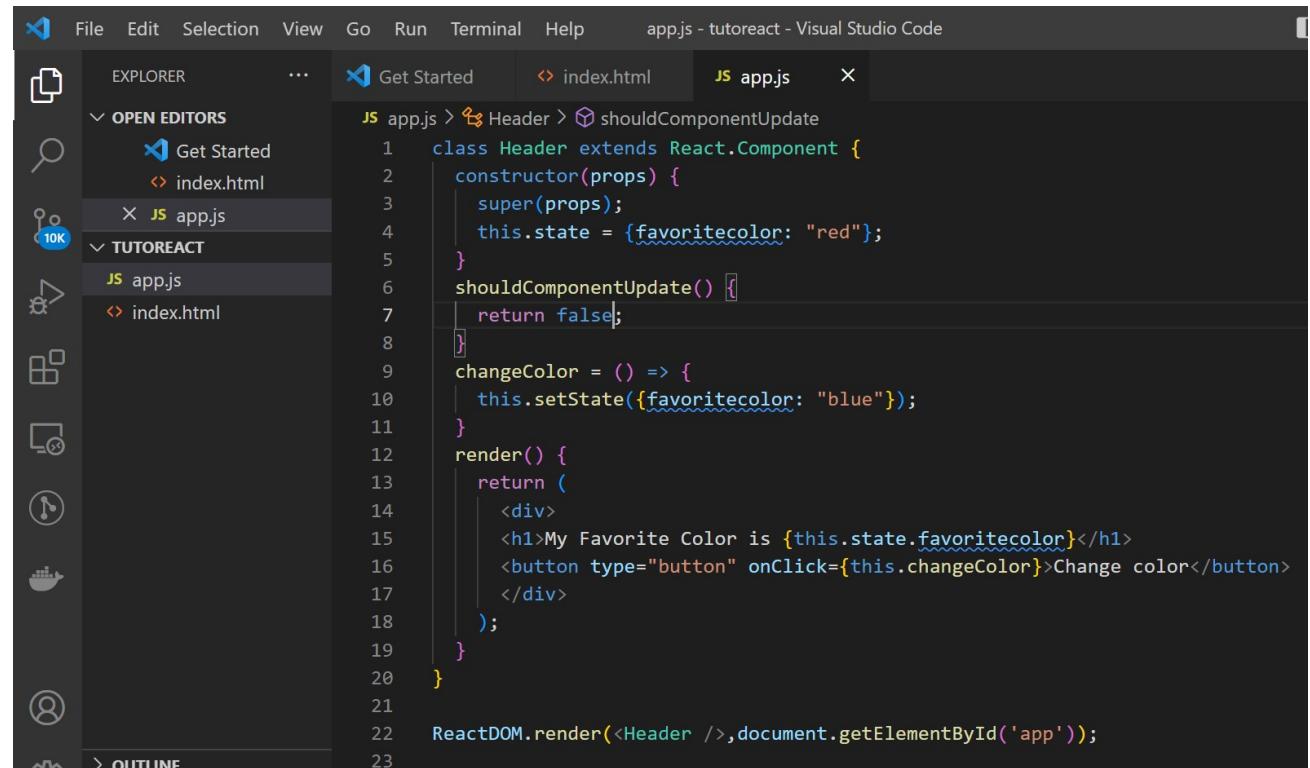
Dans la méthode `shouldComponentUpdate()`, vous pouvez renvoyer une valeur booléenne qui spécifie si React doit continuer le rendu ou non.

La valeur par défaut est true.

L'exemple ci-dessous montre ce qui se passe lorsque la méthode `shouldComponentUpdate()` renvoie false :

Exemple:

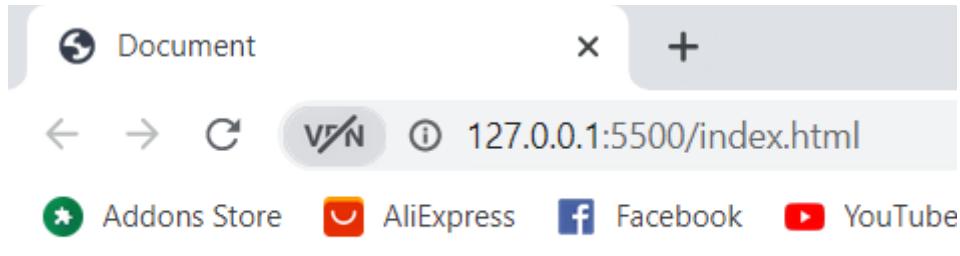
Arrêtez le rendu du composant à chaque mise à jour :



The screenshot shows the Visual Studio Code interface with the following details:

- File Bar:** File, Edit, Selection, View, Go, Run, Terminal, Help.
- Title Bar:** app.js - tutoreact - Visual Studio Code.
- Explorer Panel (Left):** Shows the project structure with "OPEN EDITORS" containing "Get Started", "index.html", and "JS app.js" (which is selected). Below it is a "TUTOREACT" folder containing "JS app.js" and "index.html".
- Editor Area (Right):** Displays the content of the "app.js" file. The code defines a class `Header` extending `React.Component`. It has a constructor, a state object with `favoritecolor: "red"`, and a `shouldComponentUpdate` method that returns `false`. The `render` method creates a `div` with an `h1` element displaying the current favorite color and a `button` to change it to blue. Finally, it uses `ReactDOM.render` to render the component.

```
class Header extends React.Component {
  constructor(props) {
    super(props);
    this.state = {favoritecolor: "red"};
  }
  shouldComponentUpdate() {
    return false;
  }
  changeColor = () => {
    this.setState({favoritecolor: "blue"});
  }
  render() {
    return (
      <div>
        <h1>My Favorite Color is {this.state.favoritecolor}</h1>
        <button type="button" onClick={this.changeColor}>Change color</button>
      </div>
    );
  }
}
ReactDOM.render(<Header />, document.getElementById('app'));
```



My Favorite Color is red

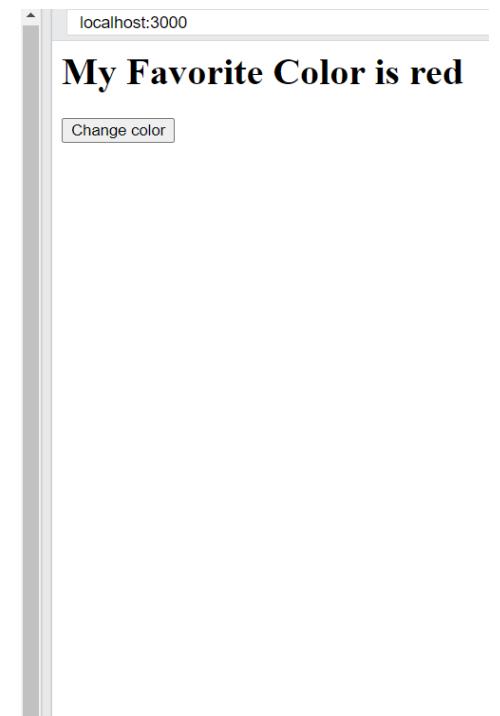
[Change color](#)

Ou bien

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

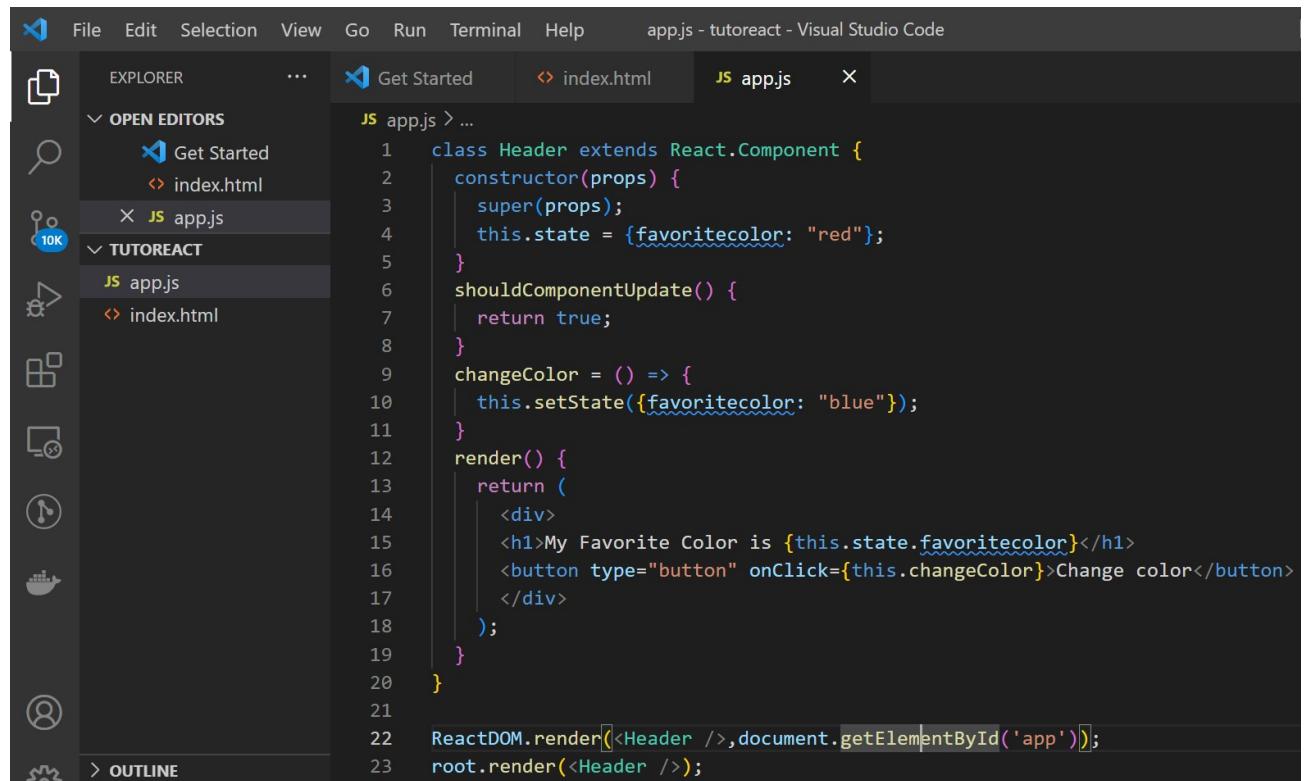
class Header extends React.Component {
  constructor(props) {
    super(props);
    this.state = {favoritecolor: "red"};
  }
  shouldComponentUpdate() {
    return false;
  }
  changeColor = () => {
    this.setState({favoritecolor: "blue"});
  }
  render() {
    return (
      <div>
        <h1>My Favorite Color is {this.state.favoritecolor}</h1>
        <button type="button" onClick={this.changeColor}>Change color</button>
      </div>
    );
  }
}

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



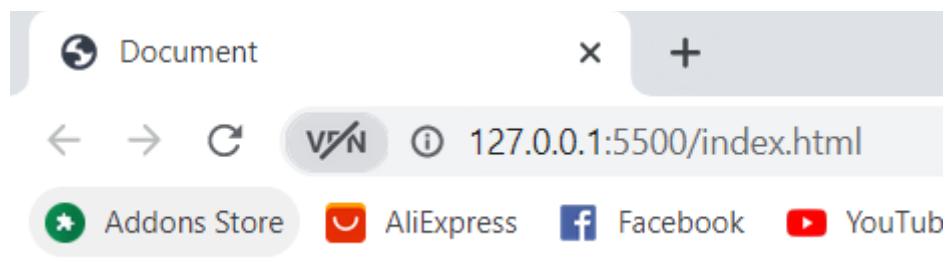
Exemple:

Même exemple que ci-dessus, mais cette fois la méthode `shouldComponentUpdate()` renvoie true :



The screenshot shows the Visual Studio Code interface. The left sidebar has icons for file operations like Open, Save, Find, and Run. The Explorer sidebar shows 'OPEN EDITORS' with 'Get Started', 'index.html', and 'JS app.js'. Under 'TUTOREACT', there are 'JS app.js' and 'index.html'. The main editor area shows the following code:

```
1 class Header extends React.Component {
2     constructor(props) {
3         super(props);
4         this.state = {favoritecolor: "red"};
5     }
6     shouldComponentUpdate() {
7         return true;
8     }
9     changeColor = () => {
10        this.setState({favoritecolor: "blue"});
11    }
12     render() {
13         return (
14             <div>
15                 <h1>My Favorite Color is {this.state.favoritecolor}</h1>
16                 <button type="button" onClick={this.changeColor}>Change color</button>
17             </div>
18         );
19     }
20 }
21 ReactDOM.render(<Header />,document.getElementById('app'));
22 root.render(<Header />);
```



My Favorite Color is blue

Change color

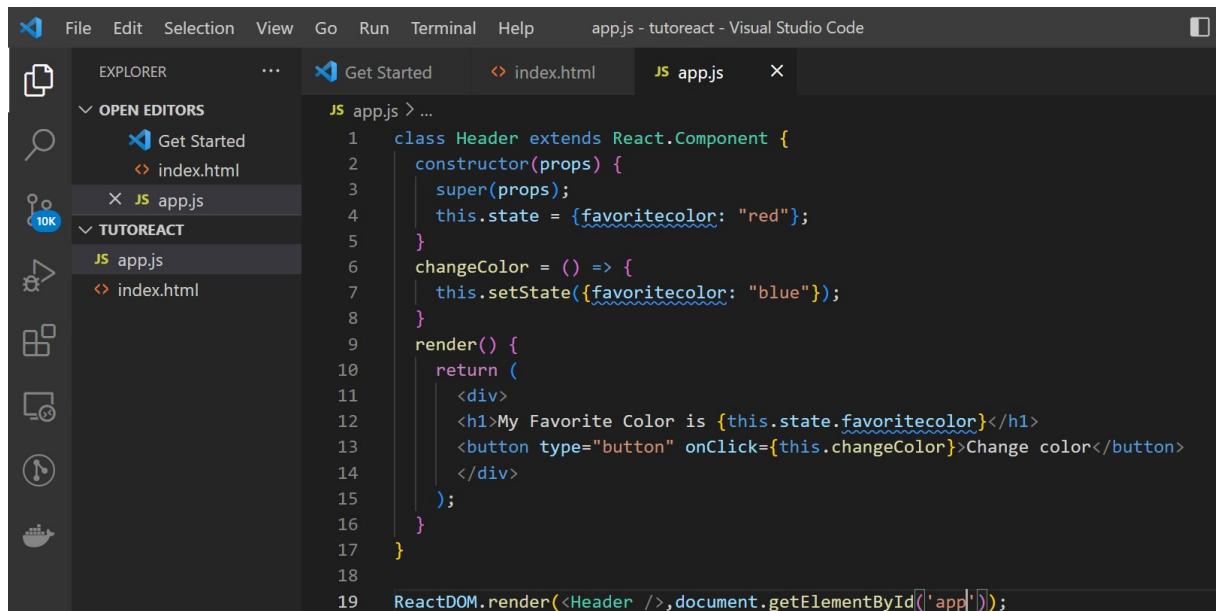
render

La méthode render() est bien sûr appelée lorsqu'un composant est mis à jour, il doit restituer le HTML au DOM, avec les nouvelles modifications.

L'exemple ci-dessous a un bouton qui change la couleur préférée en bleu :

Exemple:

Cliquez sur le bouton pour modifier l'état du composant :

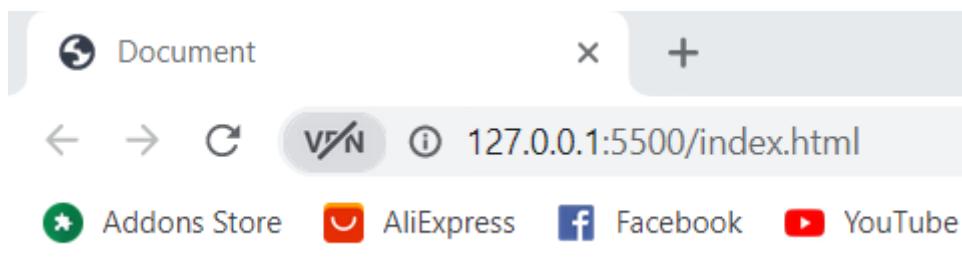


The screenshot shows the Visual Studio Code interface. The left sidebar has icons for file operations like Open, Save, Find, and Run. The Explorer sidebar shows a folder named 'TUTOREACT' containing 'app.js' and 'index.html'. The main editor area displays the following code:

```
File Edit Selection View Go Run Terminal Help app.js - tutoreact - Visual Studio Code

EXPLORER ... Get Started index.html JS app.js X

JS app.js > ...
1 class Header extends React.Component {
2   constructor(props) {
3     super(props);
4     this.state = {favoritecolor: "red"};
5   }
6   changeColor = () => {
7     this.setState({favoritecolor: "blue"});
8   }
9   render() {
10    return (
11      <div>
12        <h1>My Favorite Color is {this.state.favoritecolor}</h1>
13        <button type="button" onClick={this.changeColor}>Change color</button>
14      </div>
15    );
16  }
17
18 ReactDOM.render(<Header />, document.getElementById('app'));
19
```



My Favorite Color is red

Change color

getSnapshotBeforeUpdate

Dans la méthode `getSnapshotBeforeUpdate()`, vous avez accès aux `props` et à `state` avant la mise à jour, ce qui signifie que même après la mise à jour, vous pouvez vérifier quelles étaient les valeurs avant la mise à jour.

Si la méthode `getSnapshotBeforeUpdate()` est présente, vous devez également inclure la méthode `componentDidUpdate()`, sinon vous obtiendrez une erreur.

L'exemple ci-dessous peut sembler compliqué, mais il ne fait que ceci :

Lorsque le composant est monté, il est rendu avec la couleur préférée "rouge".

Lorsque le composant est monté, une minuterie change `state`, et au bout d'une seconde, la couleur préférée devient "jaune".

Cette action déclenche la phase de mise à jour, et puisque ce composant a une méthode `getSnapshotBeforeUpdate()`, cette méthode est exécutée, et écrit un message dans l'élément DIV1 vide.

Ensuite, la méthode `componentDidUpdate()` est exécutée et écrit un message dans l'élément DIV2 vide :

Exemple:

Utilisez la méthode `getSnapshotBeforeUpdate()` pour savoir à quoi ressemblait l'objet d'état avant la mise à jour :

The screenshot shows the Visual Studio Code interface. The left sidebar has icons for file operations like copy, paste, search, and refresh. The 'OPEN EDITORS' section lists 'Get Started', 'index.html', and 'JS app.js'. Below that is a folder named 'TUTORReact' containing 'JS app.js' and 'index.html'. The main editor area shows the following code for 'app.js':

```
1  class Header extends React.Component {  
2    constructor(props) {  
3      super(props);  
4      this.state = {favoritecolor: "red"};  
5    }  
6    componentDidMount() {  
7      setTimeout(() => {  
8        this.setState({favoritecolor: "yellow"})  
9      }, 1000)  
10  }  
11  getSnapshotBeforeUpdate(prevProps, prevState) {  
12    document.getElementById("div1").innerHTML =  
13    "Before the update, the favorite was " + prevState.favoritecolor;  
14  }  
15  componentDidUpdate() {  
16    document.getElementById("div2").innerHTML =  
17    "The updated favorite is " + this.state.favoritecolor;  
18  }  
19  render() {  
20    return (  
21      <div>  
22        <h1>My Favorite Color is {this.state.favoritecolor}</h1>  
23        <div id="div1"></div>  
24        <div id="div2"></div>  
25      </div>  
26    );  
27  }  
28  ReactDOM.render(<Header />,document.getElementById('app'));  
29  
30
```

The screenshot shows a browser window with a dark theme. The address bar shows the URL '127.0.0.1:5500/index.html'. The page content displays the text 'My Favorite Color is yellow' in large, bold, black font. Below it, two smaller lines of text are visible: 'Before the update, the favorite was red' and 'The updated favorite is yellow'.

My Favorite Color is yellow

Before the update, the favorite was red
The updated favorite is yellow

Ou bien

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

class Header extends React.Component {
  constructor(props) {
    super(props);
    this.state = {favoritecolor: "red"};
  }
  componentDidMount() {
    setTimeout(() => {
      this.setState({favoritecolor: "yellow"})
    }, 1000)
  }
  getSnapshotBeforeUpdate(prevProps, prevState) {
    document.getElementById("div1").innerHTML =
    "Before the update, the favorite was " + prevState.favoritecolor;
  }
  componentDidUpdate() {
    document.getElementById("div2").innerHTML =
    "The updated favorite is " + this.state.favoritecolor;
  }
}
```

localhost:3000

My Favorite Color is yellow

Before the update, the favorite was red
The updated favorite is yellow

```
render() {
  return (
    <div>
      <h1>My Favorite Color is {this.state.favoritecolor}</h1>
      <div id="div1"></div>
      <div id="div2"></div>
    </div>
  );
}

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

componentDidUpdate

La méthode **componentDidUpdate** est appelée après la mise à jour du composant dans le DOM.

L'exemple ci-dessous peut sembler compliqué, mais il ne fait que ceci :

Lorsque le composant est monté, il est rendu avec la couleur préférée "rouge".

Lorsque le composant a été monté, une minuterie change l'état, et la couleur devient "jaune".

Cette action déclenche la phase de mise à jour, et puisque ce composant a une méthode componentDidUpdate, cette méthode est exécutée et écrit un message dans l'élément DIV vide :

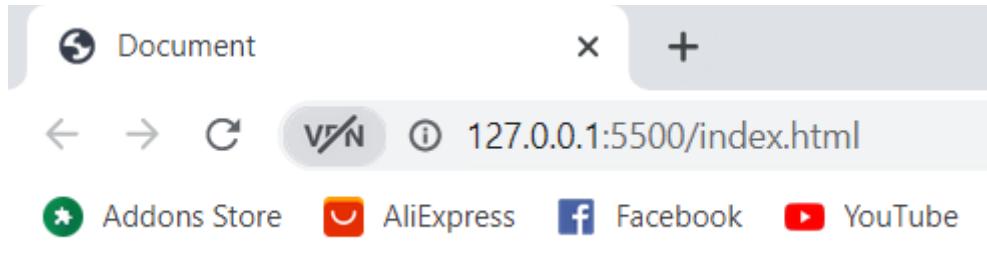
Exemple:

La méthode componentDidUpdate est appelée après le rendu de la mise à jour dans le DOM :

The screenshot shows the Visual Studio Code interface with the following details:

- File Bar:** File, Edit, Selection, View, Go, Run, Terminal, Help.
- Explorer:** Shows the file structure with "OPEN EDITORS" containing "Get Started", "index.html", and "JS app.js" (which is currently selected). It also shows a folder "TUTOREACT" containing "JS app.js" and "index.html".
- Code Editor:** Displays the content of "app.js". The code defines a class "Header" that extends React.Component. It has a constructor setting state to {favoritecolor: "red"}, a componentDidMount method setting state to {favoritecolor: "yellow"} after a 1-second timeout, and a componentDidUpdate method updating a div's innerHTML to "The updated favorite is " + state.favoritecolor;. The render method returns a div containing an h1 element with the text "My Favorite Color is " followed by the state.favoritecolor value, and a div with id="mydiv". Finally, it uses ReactDOM.render to mount the component to the DOM.

```
class Header extends React.Component {
  constructor(props) {
    super(props);
    this.state = {favoritecolor: "red"};
  }
  componentDidMount() {
    setTimeout(() => {
      this.setState({favoritecolor: "yellow"});
    }, 1000);
  }
  componentDidUpdate() {
    document.getElementById("mydiv").innerHTML =
      "The updated favorite is " + this.state.favoritecolor;
  }
  render() {
    return (
      <div>
        <h1>My Favorite Color is {this.state.favoritecolor}</h1>
        <div id="mydiv"></div>
      </div>
    );
  }
}
ReactDOM.render(<Header />, document.getElementById('app'));
```



My Favorite Color is yellow

The updated favorite is yellow

Unmounting

La phase suivante du cycle de vie est lorsqu'un composant est supprimé du DOM, ou démonté comme React aime l'appeler.

React n'a qu'une seule méthode intégrée qui est appelée lorsqu'un composant est démonté :

`componentWillUnmount()`

`componentWillUnmount`

La méthode `componentWillUnmount` est appelée lorsque le composant est sur le point d'être supprimé du DOM.

Exemple:

Cliquez sur le bouton pour supprimer l'en-tête :

The screenshot shows the Visual Studio Code interface with the following details:

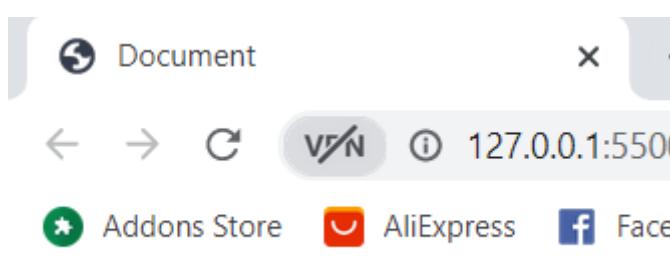
- File Bar:** File, Edit, Selection, View, Go, Run, Terminal, Help.
- Title Bar:** app.js - tutoreact - Visual Studio Code
- Explorer:** Shows "OPEN EDITORS" with "Get Started", "index.html", and "JS app.js" listed under "TUTOREACT".
- Code Editor:** Displays the content of "app.js".

```
1 class Container extends React.Component {  
2     constructor(props) {  
3         super(props);  
4         this.state = {show: true};  
5     }  
6     delHeader = () => {  
7         this.setState({show: false});  
8     }  
9     render() {  
10        let myheader;  
11        if (this.state.show) {  
12            myheader = <Child />;  
13        };  
14        return (  
15            <div>  
16                {myheader}  
17                <button type="button" onClick={this.delHeader}>Delete Header</button>  
18            </div>  
19        );  
20    }  
21 }  
22
```

The screenshot shows the Visual Studio Code interface with the following details:

- File Bar:** File, Edit, Selection, View, Go, Run, Terminal, Help.
- Title Bar:** app.js - tutoreact - Visual Studio Code
- Code Editor:** Displays the content of "Child.js".

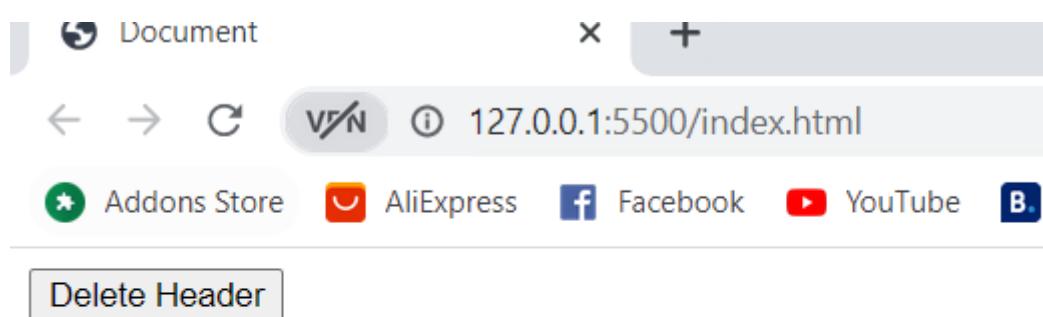
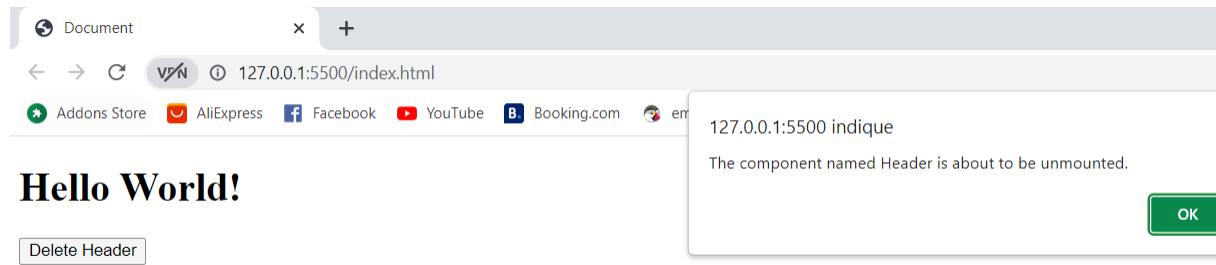
```
23 class Child extends React.Component {  
24     componentWillUnmount() {  
25         alert("The component named Header is about to be unmounted.");  
26     }  
27     render() {  
28         return (  
29             <h1>Hello World!</h1>  
30         );  
31     }  
32 }  
33  
34 ReactDOM.render(<Container />, document.getElementById('app'));  
35
```



Hello World!

Delete Header

Cliquez delete



Ou bien

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

class Container extends React.Component {
  constructor(props) {
    super(props);
    this.state = {show: true};
  }
  delHeader = () => {
    this.setState({show: false});
  }
  render() {
    let myheader;
    if (this.state.show) {
      myheader = <Child />;
    };
    return (
      <div>
        {myheader}
        <button type="button" onClick={this.delHeader}>Delete Header</button>
      </div>
    );
  }
}
```

localhost:3000

Hello World!

Delete Header

```
class Child extends React.Component {
  componentWillUnmount() {
    alert("The component named Header is about to be unmounted.");
  }
  render() {
    return (
      <h1>Hello World!</h1>
    );
  }
}

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

React Events

Tout comme les événements HTML DOM, React peut effectuer des actions basées sur des événements utilisateur.

React a les mêmes événements que HTML : click, change, mouseover etc.

Adding Events

Les événements React sont écrits en syntaxe camelCase :

onClick au lieu de onclick.

Les gestionnaires d'événements React sont écrits entre accolades :

onClick={shoot} au lieu de onClick="shoot()".

React:

```
<button onClick={shoot}>Take the Shot!</button>
```

HTML:

```
<button onclick="shoot()">Take the Shot!</button>
```

Example:

Put the shoot function inside the Football component:

The screenshot shows the Visual Studio Code interface. The Explorer sidebar on the left lists files: 'Get Started', 'index.html', 'JS app.js' (which is currently selected), and 'index.html'. The main editor area displays the following code:

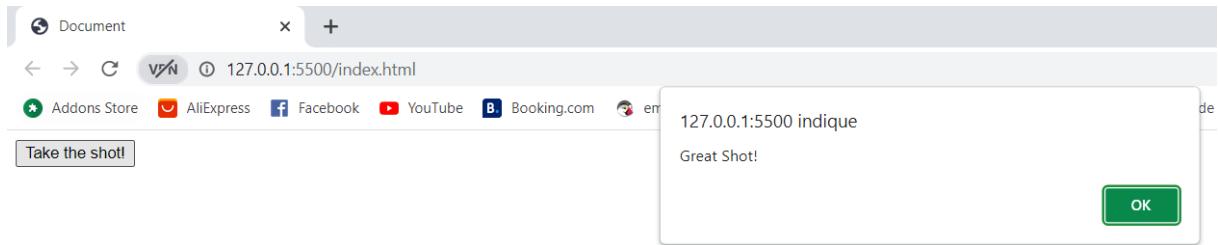
```
function Football() {
  const shoot = () => {
    alert("Great Shot!");
  }

  return (
    <button onClick={shoot}>Take the shot!</button>
  );
}

ReactDOM.render(<Football />, document.getElementById('app'));
```

Below the editor is a browser window showing the result of running the code at 127.0.0.1:5500. A button labeled "Take the shot!" is visible. When clicked, an alert box appears with the message "Great Shot!". An "OK" button is at the bottom right of the alert box.

Ou bien

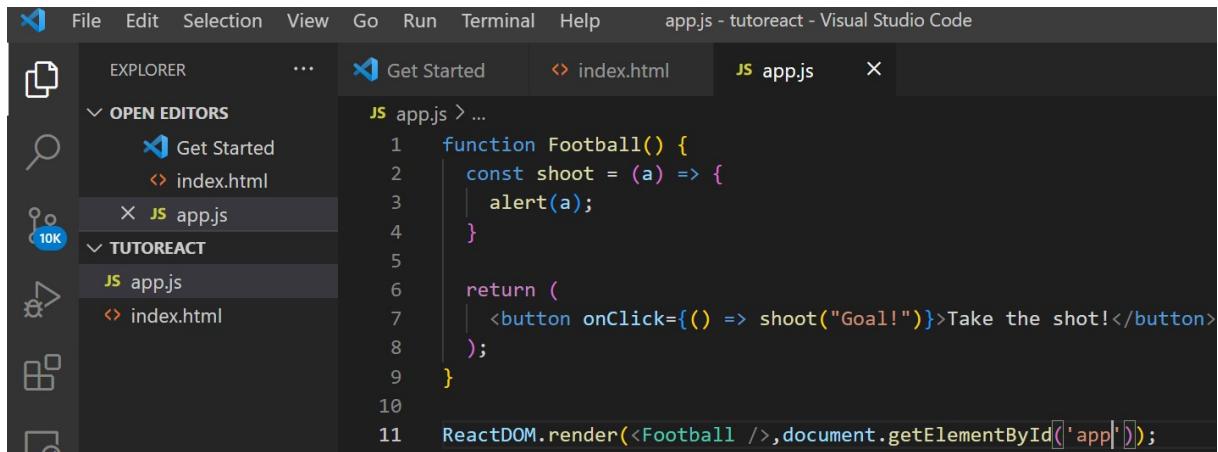


Passer des arguments

Pour passer un argument à un gestionnaire d'événements, utilisez une fonction fléchée.

Exemple:

Envoyer "Objectif !" comme paramètre de la fonction shoot, en utilisant la fonction flèche :



The screenshot shows the Visual Studio Code interface. The left sidebar has icons for file operations, search, and code navigation. The main area shows an Explorer sidebar with 'OPEN EDITORS' containing 'Get Started', 'index.html', and 'JS app.js'. Below that is a 'TUTOREACT' section with 'JS app.js' and 'index.html'. The central editor pane displays the following code:

```
1 function Football() {
2     const shoot = (a) => {
3         alert(a);
4     }
5
6     return (
7         <button onClick={() => shoot("Goal!")}>Take the shot!</button>
8     );
9 }
10 ReactDOM.render(<Football />,document.getElementById('app'));
11
```

Ou bien



The screenshot shows a browser window with the URL 'www.w3schools.com' in the address bar. A tooltip from 'www.w3schools.com' indicates the word 'Goal'. The browser's developer tools are open, showing the React component structure. The code in the component is identical to the one shown in the VS Code screenshot. A modal dialog box is displayed, containing the text 'Goal' and a blue 'OK' button. Below the modal, there is a button labeled 'Take the shot!'.

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

function Football() {
    const shoot = (a) => {
        alert(a);
    }

    return (
        <button onClick={() => shoot("Goal!")}>Take the shot!</button>
    );
}

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

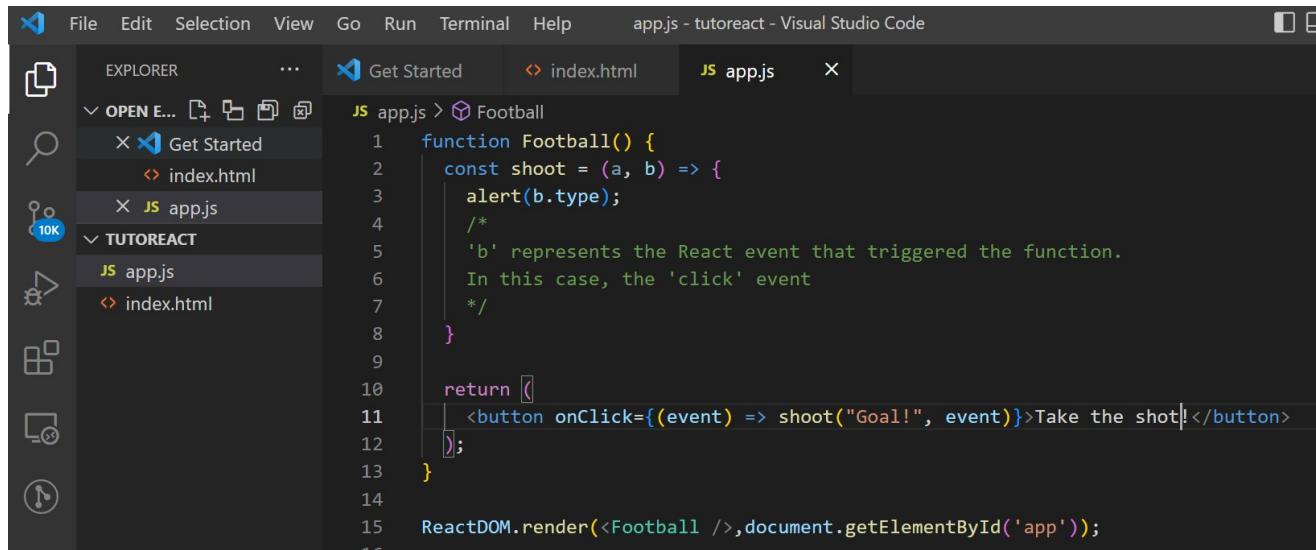
React Event Object

Les gestionnaires d'événements ont accès à l'événement React qui a déclenché la fonction.

Dans notre exemple, l'événement est l'événement "clic".

Exemple:

Fonction Flèche : Envoi manuel de l'objet événement :



```
File Edit Selection View Go Run Terminal Help app.js - tutoreact - Visual Studio Code

EXPLORER      ...
OPEN E...  Get Started  index.html  app.js  ...

JS app.js > Football
1  function Football() {
2    const shoot = (a, b) => {
3      alert(b.type);
4      /*
5       'b' represents the React event that triggered the function.
6       In this case, the 'click' event
7      */
8    }
9
10   return [
11     <button onClick={()=> shoot("Goal!", event)}>Take the shot!</button>
12   ];
13 }
14
15 ReactDOM.render(<Football />, document.getElementById('app'));
16
```

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

function Football() {
  const shoot = (a, b) => {
    alert(b.type);
    /*
    'b' represents the React event that triggered the function.
    In this case, the 'click' event
  }
}

return (
  <button onClick={(event) => shoot("Goal!", event)}>Take the shot!</button>
);
}

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

React Conditional Rendering

Dans React, vous pouvez effectuer un rendu conditionnel des composants.

Il y a plusieurs moyens de le faire.

si déclaration

Nous pouvons utiliser l'opérateur JavaScript if pour décider quel composant rendre.

Exemple:

Nous utiliserons ces deux composants :

```
function MissedGoal() {
```

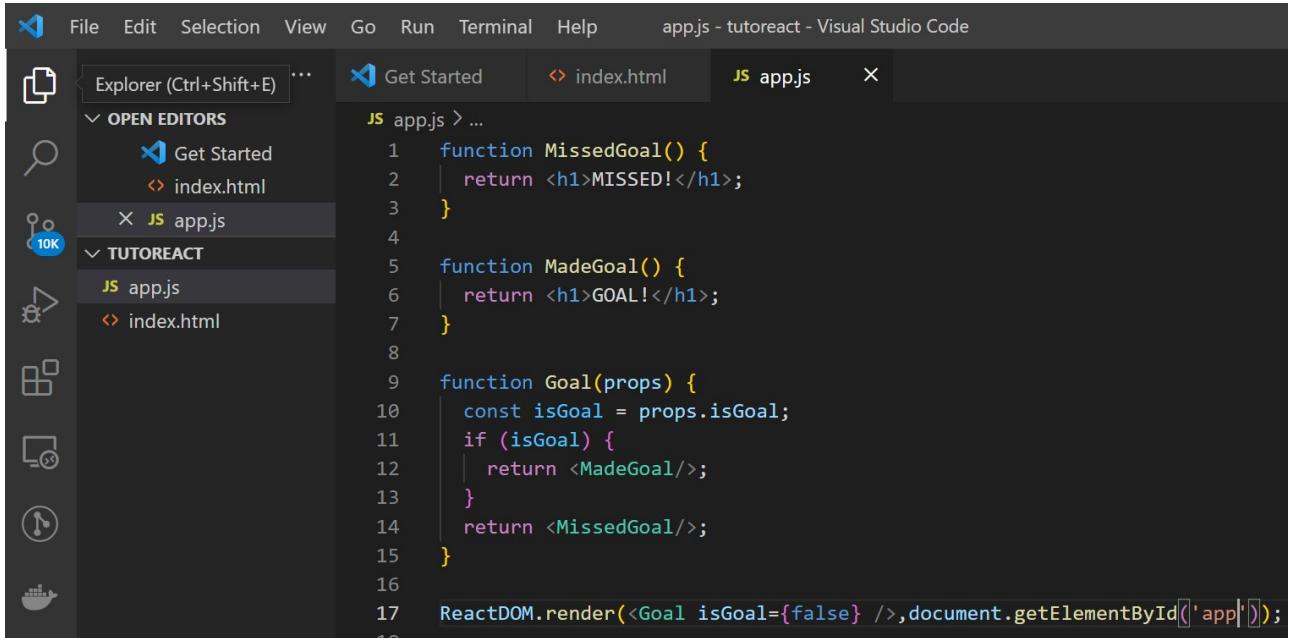
```
        return <h1>MISSED!</h1>;
    }

}
```

```
function MadeGoal() {
    return <h1>Goal!</h1>;
}
```

Exemple:

Maintenant, nous allons créer un autre composant qui choisit le composant à afficher en fonction d'une condition :



The screenshot shows the Visual Studio Code interface. The file explorer on the left lists files under 'OPEN EDITORS' (Get Started, index.html) and 'TUTOREACT' (app.js, index.html). The code editor on the right contains the following React component code:

```
function MissedGoal() {
    return <h1>MISSED!</h1>;
}

function MadeGoal() {
    return <h1>GOAL!</h1>;
}

function Goal(props) {
    const isGoal = props.isGoal;
    if (isGoal) {
        return <MadeGoal/>;
    }
    return <MissedGoal/>;
}

ReactDOM.render(<Goal isGoal={false}>, document.getElementById('app'));
```

localhost:3000

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

function MissedGoal() {
  return <h1>MISSED!</h1>;
}

function MadeGoal() {
  return <h1>GOAL!</h1>;
}

function Goal(props) {
  const isGoal = props.isGoal;
  if (isGoal) {
    return <MadeGoal/>;
  }
  return <MissedGoal/>;
}

const root = ReactDOM.createRoot(document.getElementById('root'));
root.render(<Goal isGoal={false} />);
```

MISSED!

Essayez de changer l'attribut isGoal en true :

The screenshot shows the Visual Studio Code interface with the following details:

- File Bar:** File, Edit, Selection, View, Go, Run, Terminal, Help.
- Active File:** app.js - tutoreact - Visual Studio Code
- Explorer:** Shows a tree view with 'OPEN E...', 'Get Started', 'index.html', 'JS app.js' (selected), and 'TUTOREACT' folder containing 'JS app.js' and 'index.html'.
- Code Editor:** Displays the following code in 'app.js':

```
JS app.js > ...
1 function MissedGoal() {
2   return <h1>MISSED!</h1>;
3 }
4
5 function MadeGoal() {
6   return <h1>GOAL!</h1>;
7 }
8
9 function Goal(props) {
10  const isGoal = props.isGoal;
11  if (isGoal) {
12    return <MadeGoal/>;
13  }
14  return <MissedGoal/>;
15 }
16
17 ReactDOM.render(<Goal isGoal={true} />,document.getElementById('app'));
```

The screenshot shows a browser window with the following details:

- Address Bar:** localhost:3000
- Content Area:** Displays the word "GOAL!" in large, bold, black font.
- Code Preview:** On the left, the same code from 'app.js' is displayed again, showing the original source code.

React Forms

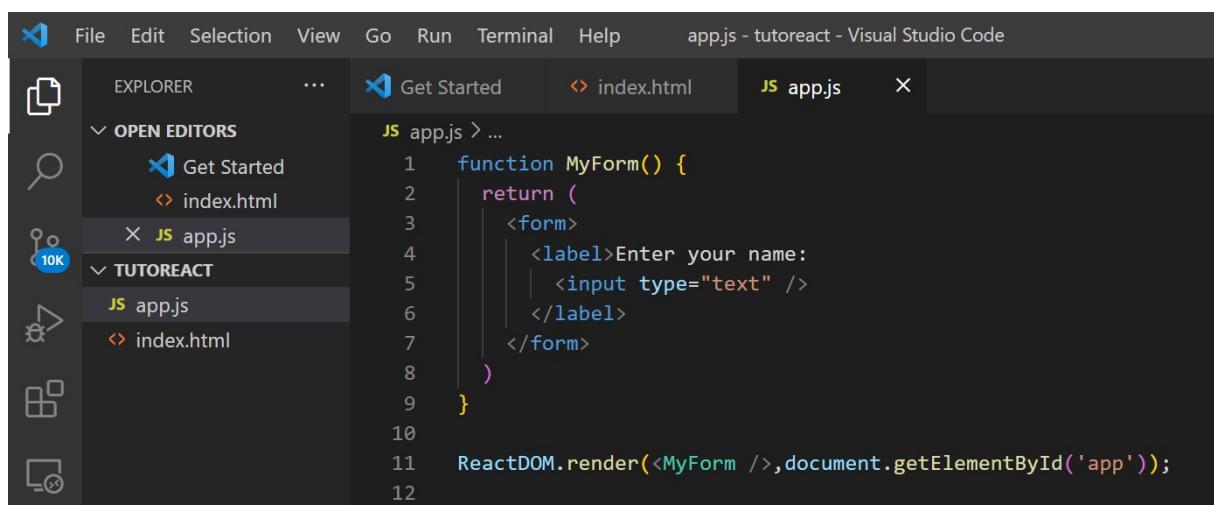
Tout comme en HTML, React utilise des formulaires pour permettre aux utilisateurs d'interagir avec la page Web.

Ajouter des formulaires dans React

Vous ajoutez un formulaire avec React comme n'importe quel autre élément :

Exemple:

Ajoutez un formulaire permettant aux utilisateurs de saisir leur nom :



The screenshot shows the Visual Studio Code interface with the following details:

- File Bar:** File, Edit, Selection, View, Go, Run, Terminal, Help.
- Title Bar:** app.js - tutoreact - Visual Studio Code
- Explorer Panel:** Shows the project structure with "OPEN EDITORS" expanded, showing "Get Started", "index.html", and "JS app.js". Below it, "TUTOREACT" is expanded, showing "JS app.js" and "index.html".
- Code Editor:** The "JS app.js" tab is active, displaying the following code:

```
function MyForm() {
  return (
    <form>
      <label>Enter your name:</label>
      <input type="text" />
    </form>
  )
}

ReactDOM.render(<MyForm />, document.getElementById('app'));
```

Ou bien

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

function MyForm() {
  return (
    <form>
      <label>Enter your name:</label>
      <input type="text" />
    </form>
  )
}

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



React Router

Creez React App n'inclut pas le routage de page.

React Router est la solution la plus populaire

Add React Router

Pour ajouter React Router dans votre application, exécutez-le dans le terminal à partir du répertoire racine de l'application :

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

Remarque : ici on utilise React Router v6.

Si vous effectuez une mise à niveau à partir de la v5, vous devrez utiliser l'indicateur @latest :

```
npm i -D react-router-dom@latest
```

Structure des dossiers

Pour créer une application sur un nouveau projet avec plusieurs routes de page, commençons d'abord par :

Installation de Node.js

Node.js est un environnement d'exécution JavaScript utilisé pour créer une pile complète applications. Node est open source et peut être installé sur Windows, macOS, Linux et autres plates-formes.

Vous devez avoir Node installé, mais vous n'avez pas besoin d'être un Node expert pour utiliser React. Si vous ne savez pas si Node.js est installé sur votre machine, vous pouvez ouvrir une fenêtre de terminal ou d'invite de commande

et tapez :

```
node -v
```

Lorsque vous exécutez cette commande, vous devriez voir un numéro de version de nœud vous est retourné, idéalement 20.9.0 ou supérieur. Si vous tapez la commande et voir un message d'erreur indiquant "Commande introuvable", Node.js n'est pas installée. Ceci est facilement résolu en installant Node.js à partir de [Node.js](#) site Internet.

Suivez simplement les étapes automatisées du programme d'installation, et lorsque vous tapez à nouveau la commande node -v, vous verrez le numéro de version.

NPM

Lorsque vous avez installé Node.js, vous avez également installé npm, le package Node gestionnaire. Dans la communauté JavaScript, les ingénieurs partagent l'open source code des projets pour éviter d'avoir à réécrire des frameworks, des bibliothèques ou fonctions d'assistance par

elles-mêmes. React lui-même est un exemple d'un outil utile bibliothèque npm. Nous utiliserons npm pour installer une variété de packages.

La plupart des projets JavaScript que vous rencontrez aujourd'hui contiendront un assortiment collection de fichiers plus un fichier package.json. Ce fichier décrit le projet et toutes ses dépendances. Si vous exécutez npm install dans le dossier qui contient le fichier package.json, npm installera tous les packages répertoriés dans le projet.

Si vous démarrez votre propre projet à partir de zéro et souhaitez inclure dépendances, exécutez simplement la commande :

```
npm init -y
```

Cela initialisera le projet et créera un fichier package.json. De là, vous pouvez installer vos propres dépendances avec npm. Pour installer un package avec npm, vous exécuterez :

```
npm install package-name
```

Pour supprimer un paquet avec npm, vous exécuterez :

```
npm remove package-name
```