

React.js - Développeur Web

◆ Chapitre 1 : Introduction à React

◆ Présentation de React : Pourquoi l'utiliser ?

★ Qu'est-ce que React ?

React est une **bibliothèque JavaScript** développée par **Facebook (Meta)** et utilisée pour la création d'interfaces utilisateur interactives et réactives. Il est principalement utilisé pour construire des **applications web monopages (SPA - Single Page Applications)** et des interfaces dynamiques.

◆ Présentation de React : Pourquoi l'utiliser ?

★ Qu'est-ce que React ?

React est une **bibliothèque JavaScript** développée par **Facebook (Meta)** et utilisée pour la création d'interfaces utilisateur interactives et réactives. Il est principalement utilisé pour construire des **applications web monopages (SPA - Single Page Applications)** et des interfaces dynamiques.

🔗 Pourquoi utiliser React ?

✓ Composants réutilisables

- React est basé sur une architecture **modulaire** où l'interface est découpée en **composants** indépendants et réutilisables.
- Facilite la maintenance et l'évolutivité des projets.

✓ Performance optimisée avec le Virtual DOM

- React utilise un **Virtual DOM** qui permet de mettre à jour l'interface de manière **efficace et rapide**, sans recharger la page entière.
- Seules les parties de l'UI qui ont changé sont mises à jour, améliorant ainsi les performances.

✓ Unidirectional Data Flow (Flux de données unidirectionnel)

- Contrairement à d'autres frameworks, React adopte un **flux de données unidirectionnel**, ce qui facilite la gestion de l'état et réduit les bugs.

✓ Grande communauté et écosystème riche

- Une communauté active avec des milliers de **bibliothèques et outils** disponibles (Redux, React Router, Tailwind, Material UI, etc.).

✓ Facilité d'apprentissage

- Syntaxe **JSX** intuitive qui mélange JavaScript et HTML.
- Approche déclarative qui simplifie le développement des interfaces utilisateur.

✓ Support mobile avec React Native

- React permet aussi de développer des **applications mobiles natives** avec **React Native**, en réutilisant une grande partie du code écrit pour le web.

◆ Installation et configuration de l'environnement (Node.js, npm, Vite/Create React App)

Avant de commencer à coder avec **React**, il faut configurer un environnement de développement adapté. Voici les étapes essentielles pour l'installation et la configuration.

1. Installation de Node.js et npm

Pourquoi ?

React utilise **Node.js** pour exécuter des outils comme **npm** (**N**ode **P**ackage **M**anager) et gérer les dépendances.

◆ Vérifier si Node.js est installé

Ouvre un terminal et tape la commande suivante :

```
node -v
```

Si Node.js est installé, la version s'affiche. Si ce n'est pas le cas, installe-le en suivant les étapes ci-dessous.

◆ Télécharger et installer Node.js

- Va sur <https://nodejs.org/>
- Télécharge la version **LTS (Long-Term Support)**
- Installe Node.js (npm est inclus avec Node.js)

◆ Vérifier npm

Après l'installation, vérifie que **npm** est bien installé avec :

```
npm -v
```

2. Créer un projet React avec Vite (Recommandé)

Pourquoi Vite ?

Vite est plus rapide que Create React App (CRA) car il optimise le développement avec un serveur de build ultra-rapide.

✦ Installer Vite et créer un projet React

Dans le terminal, exécute :

```
npm create vite@latest nom-du-projet --template react
```

Remplace **nom-du-projet** par le nom de ton projet.

✦ Aller dans le dossier du projet

```
cd nom-du-projet
```

✦ Installer les dépendances

```
npm install
```

✦ Démarrer le projet

```
npm run dev
```

Cela lancera un serveur local (par défaut sur `http://localhost:5173`).

3. Alternative : Créer un projet React avec Create React App (CRA)

⚠ **Create React App est plus lent** et moins optimisé que Vite, mais reste une option classique.

✦ Créer un projet avec CRA

```
npx create-react-app nom-du-projet
```

(Npx est inclus avec npm, il permet d'exécuter des paquets sans les installer globalement.)

✦ Aller dans le dossier du projet et lancer le serveur

```
cd nom-du-projet  
npm start
```

L'application s'ouvre sur `http://localhost:3000`.

◆ Concepts clés : Composants, JSX, Virtual DOM

Avant de commencer à coder en React, il est essentiel de comprendre ses concepts fondamentaux.

1. Composants (Components)

◆ Définition

- Un **composant** est un **bloc réutilisable** qui représente une partie de l'interface utilisateur (bouton, formulaire, carte, etc.).
- En React, **tout est basé sur des composants**.

◆ Types de composants

☞ Composants fonctionnels (recommandés)

- Ce sont de simples **fonctions JavaScript** qui retournent du JSX.
- Plus faciles à lire, tester et optimiser.

Exemple :

```
jsx

function Bonjour(props) {
  return <h1>Salut, {props.nom} !</h1>;
}
```

☞ Composants de classe (ancienne méthode)

- Définis avec une **classe ES6** et un `render()`.
- Utilisés avant l'introduction des **Hooks** (React 16.8).

Exemple :

```
jsx

class Bonjour extends React.Component {
  render() {
    return <h1>Salut, {this.props.nom} !</h1>;
  }
}
```

★ Bonnes pratiques

- ✓ Un composant **doit être indépendant et réutilisable**
- ✓ Toujours nommer les composants en **PascalCase** (`MonComposant.js`)

2. JSX (JavaScript XML)

◆ Définition

JSX est une **extension syntaxique** qui permet d'écrire du HTML directement dans du JavaScript.

◆ Pourquoi JSX ?

- ✓ Plus lisible et intuitif
- ✓ Permet de combiner logique et UI dans un seul fichier
- ✓ Sécurisé et optimisé après compilation

◆ Exemple de JSX

```
jsx

const titre = <h1>Bienvenue</h1>;
```

◆ JSX vs JavaScript

Sans JSX, on écrirait ceci :

```
jsx

const titre = React.createElement('h1', {}, 'Bienvenue');
```

Avec JSX, c'est **plus propre et lisible** ✓

✦ Bonnes pratiques JSX

- ✓ Un composant **doit retourner un seul élément parent**

✗ Mauvais :

```
jsx

function App() {
  return (
    <h1>Salut</h1>
    <p>Bienvenue</p>
  );
}
```

✓ Bon :

```
jsx

function App() {
  return (
    <>
      <h1>Salut</h1>
      <p>Bienvenue</p>
    </>
  );
}
```

```
);  
}
```

🔑 **Astuce** : Utiliser `<></>` (**Fragments**) pour éviter des `<div>` inutiles.

3. Virtual DOM 📄

🔑 Qu'est-ce que le DOM ?

Le **DOM (Document Object Model)** est la structure HTML interprétée par le navigateur.

🔑 Problème avec le DOM classique

- Modifier directement le DOM est **lent** ⚠️
- Chaque mise à jour **rafraîchit toute la page**, ce qui **ralentit les performances**

🔑 Solution : Virtual DOM

- **React crée une copie virtuelle du DOM en mémoire**
- Lorsqu'un changement est détecté, **React met à jour uniquement les parties modifiées**, au lieu de recharger toute la page
- Cela **améliore considérablement les performances**

🔑 Comment ça marche ?

1. React garde un **Virtual DOM** en mémoire
2. Lorsqu'un état change, React compare l'ancien et le nouveau Virtual DOM
3. Il met à jour **seulement les parties modifiées** du **vrai DOM**
4. 🔑 **Exemple illustré**

Action utilisateur	Virtual DOM met à jour	DOM réel est modifié
L'utilisateur clique sur un bouton	Virtual DOM met à jour le bouton	React met à jour uniquement ce bouton dans le DOM

★ Avantages du Virtual DOM

- ✓ **Optimisation des performances**
- ✓ **Moins de manipulations du DOM réel**
- ✓ **Expérience utilisateur fluide**

Conclusion

- ✓ **Les composants** rendent le code modulaire et réutilisable.
- ✓ **JSX** permet d'écrire du HTML directement dans JavaScript, rendant le code plus lisible.
- ✓ **Le Virtual DOM** améliore les performances en mettant à jour uniquement les parties nécessaires.

◆ Premier projet React : structure d'un projet

1. Structure d'un projet React

Après avoir créé un projet avec **Vite** (ou **Create React App**), voici la structure typique :

mon-projet-react/

```
| — node_modules/      # Dépendances installées
| — public/             # Fichiers publics (favicon, index.html...)
| — src/                # Code source de l'application
|   | — App.jsx         # Composant principal
|   | — main.jsx        # Point d'entrée de l'application
|   | — components/     # Dossier pour les composants React
|   | — assets/         # Images, styles et ressources
| — .gitignore          # Fichiers à ignorer par Git
| — package.json        # Liste des dépendances et scripts
| — vite.config.js      # Configuration de Vite
| — README.md           # Documentation du projet
```

Fichiers importants :

- `App.jsx` : Composant principal
- `main.jsx` : Monte l'application dans le DOM
- `package.json` : Contient les dépendances et scripts

2. Création d'une première application React

Nous allons créer une **application simple** qui affiche un message de bienvenue et un compteur interactif.

Étape 1 : Créer un projet React avec Vite

Dans ton terminal, exécute :

```
npm create vite@latest mon-premier-react --template react
```

```
cd mon-premier-react
```

```
npm install
```

```
npm run dev
```

Ouvre <http://localhost:5173> dans ton navigateur

```
D:\CFITECH\React\cours react 2025\exercices>npm create vite@latest test --template react
```

```
> npx
> create-vite test react

? Select a framework: » - Use arrow-keys. Return to submit.
  Vanilla
  Vue
>  React✓
  Preact
  Lit
  Svelte
  Solid
  Qwik
  Angular
  Others
```

```
> npx
> create-vite test react

✓ Select a framework: » React
? Select a variant: » - Use arrow-keys. Return to submit.
  TypeScript
  TypeScript + SWC
>  JavaScript✓
  JavaScript + SWC
  React Router v7 ↗
```



```
> npx
> create-vite test react

✓ Select a framework: » React
✓ Select a variant: » JavaScript

Scaffolding project in D:\CFITECH\React\cours react 2025\exercices\test...

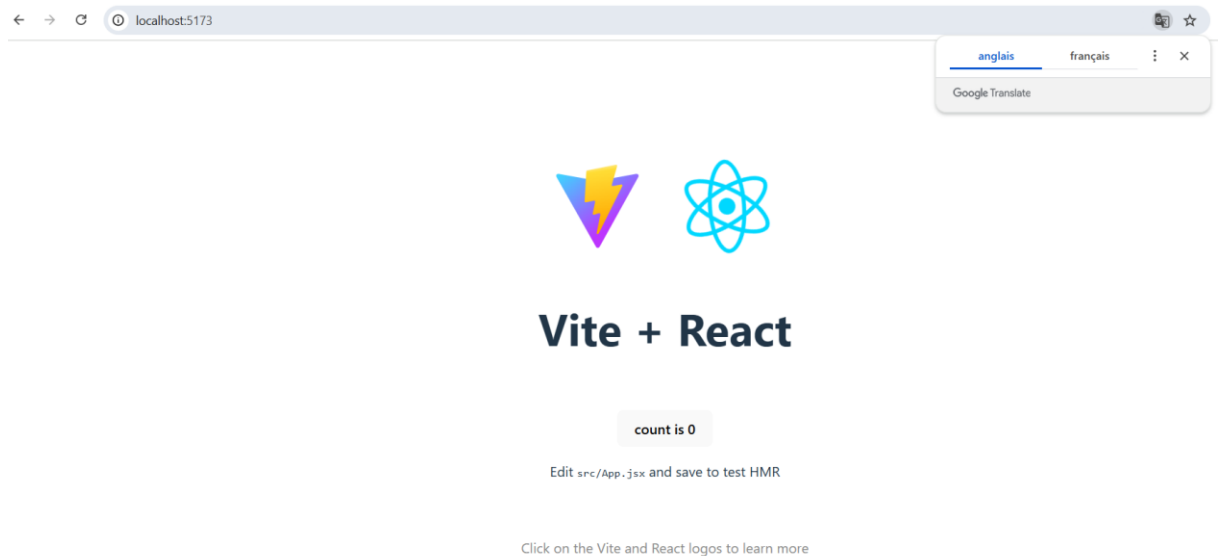
Done. Now run:

cd test ✓
npm install ✓
npm run dev ✓
```

```
C:\Windows\system32\cmd.e: X + v

VITE v6.0.11 ready in 270 ms

→ Local: http://localhost:5173/
→ Network: use --host to expose
→ press h + enter to show help
```



Étape 2 : Modifier `App.jsx` pour afficher un message de bienvenue

Dans `src/App.jsx`, remplace le code par ceci :

`jsx`

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

import './App.css';

function App() {

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

  return (

    <div className="container">

      <h1>Bienvenue sur mon premier projet React !</h1>

      <p>Ceci est une application React simple.</p>

      <h2>Compteur : {count}</h2>

      <button onClick={() => setCount(count + 1)}>➕ Augmenter</button>

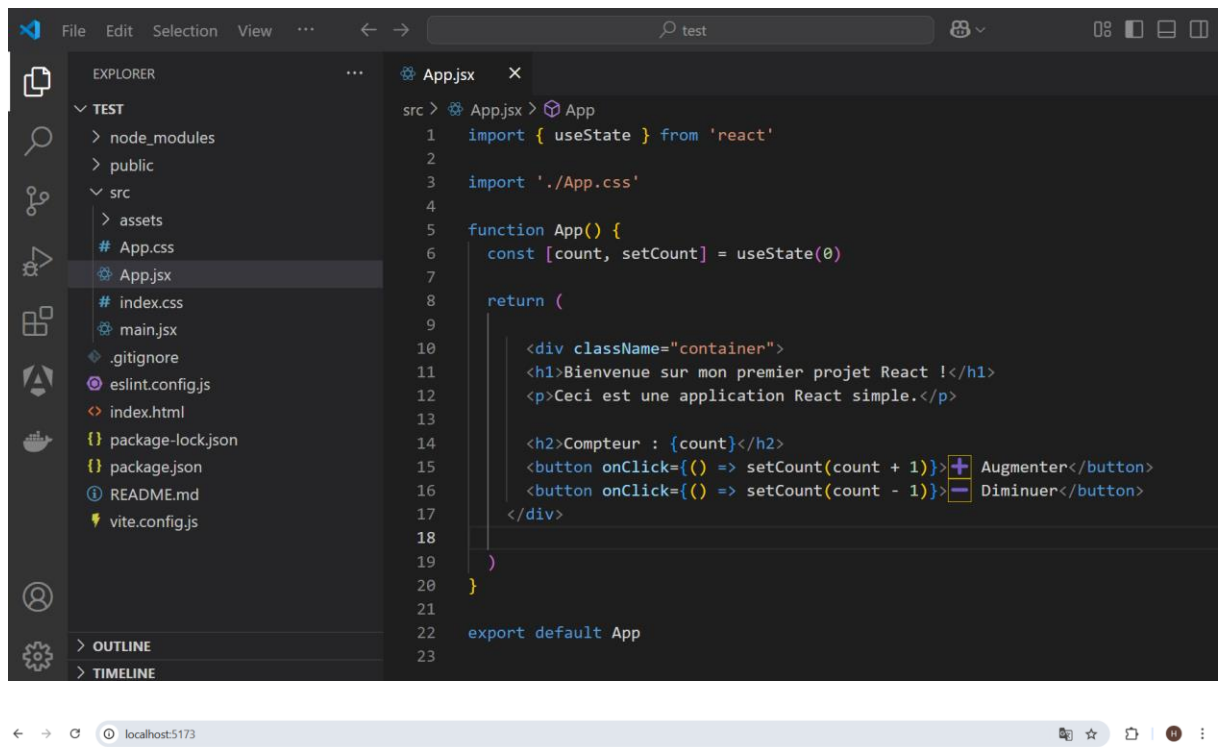
      <button onClick={() => setCount(count - 1)}>➖ Diminuer</button>

    </div>

  );

}

export default App;
```



Bienvenue sur mon premier projet React !

Ceci est une application React simple.

Compteur : -4

+ Augmenter - Diminuer

Étape 3 : Ajouter un peu de style

Dans `src/App.css`, remplace le contenu par :

```
.container {
  text-align: center;
  font-family: Arial, sans-serif;
  margin-top: 50px;
}
```

```
h1 {  
  color: #2c3e50;  
}
```

```
button {  
  margin: 10px;  
  padding: 10px 15px;  
  font-size: 16px;  
  cursor: pointer;  
  border: none;  
  border-radius: 5px;  
}
```

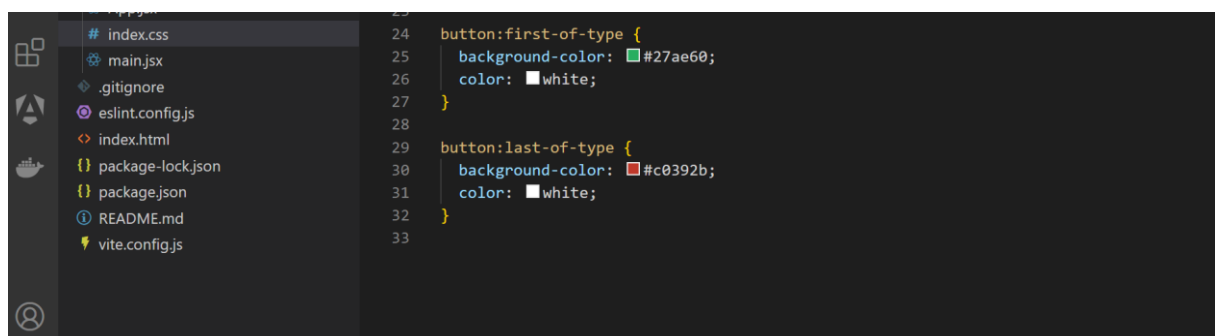
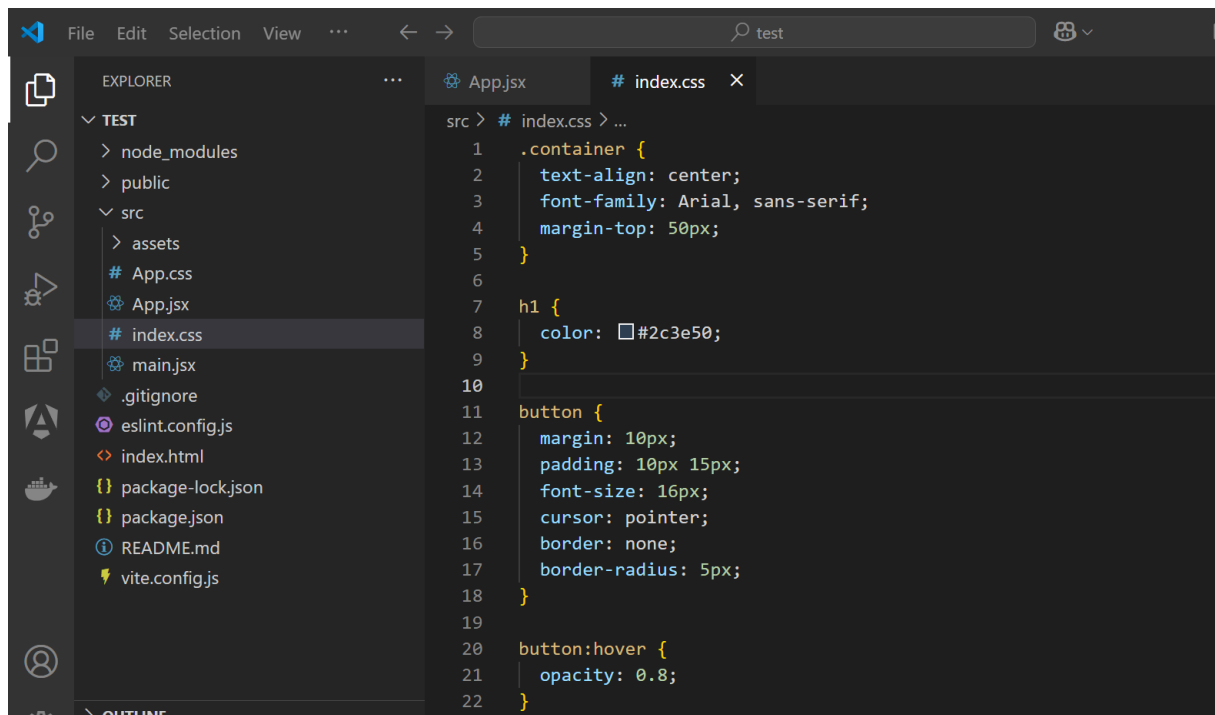
```
button:hover {  
  opacity: 0.8;  
}
```

```
button:first-of-type {  
  background-color: #27ae60;  
  color: white;  
}
```

```
button:last-of-type {  
  background-color: #c0392b;
```

```
color: white;

}
```



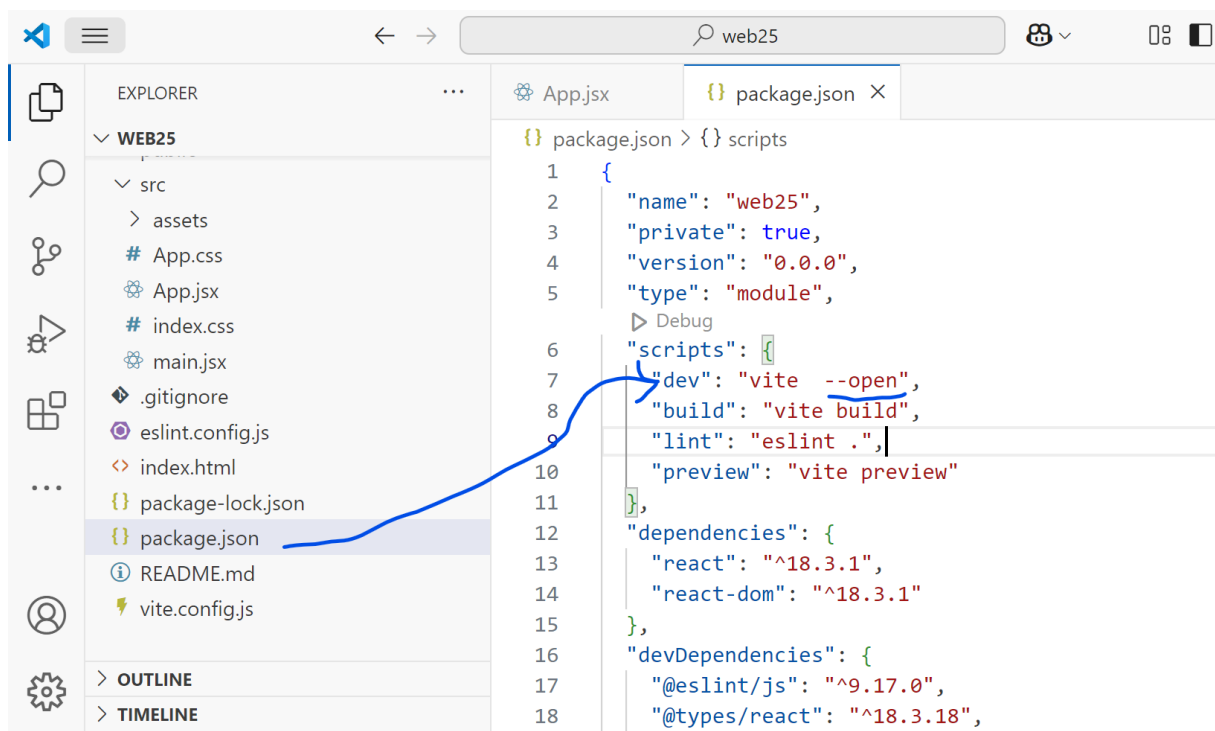
Étape 4 : Lancer l'application

Dans le terminal, tape :

```
npm run dev
```

```
D:\CFITECH\React\cours react 2025\exercices\test>npm run dev
```

Vous pouvez aussi modifier ton `package.json` pour que le script `dev` ouvre automatiquement le navigateur :



`npm run dev`

Bienvenue sur mon premier projet React !

Ceci est une application React simple.

Compteur : 0



Résumé

- ✓ On a créé un projet React avec Vite
- ✓ On a compris la structure d'un projet
- ✓ On a ajouté un premier composant avec un état (`useState`)
- ✓ On a appliqué du CSS pour améliorer l'interface

◆ Chapitre 2 : Composants et Props

Les **composants** sont la base de React. Ils permettent de **réutiliser du code** et de **structurer une application** de manière modulaire.

1. Qu'est-ce qu'un composant ?

Un **composant** en React est une **fonction** ou une **classe** qui retourne du JSX.

Il peut représenter une **petite partie de l'interface** (ex. un bouton) ou un **gros bloc** (ex. une page entière).

Il existe **deux types de composants** :

1. **Les composants fonctionnels** (recommandés)
2. **Les composants de classe** (moins utilisés depuis les Hooks)

2. Création d'un composant fonctionnel

◆ Exemple d'un composant simple

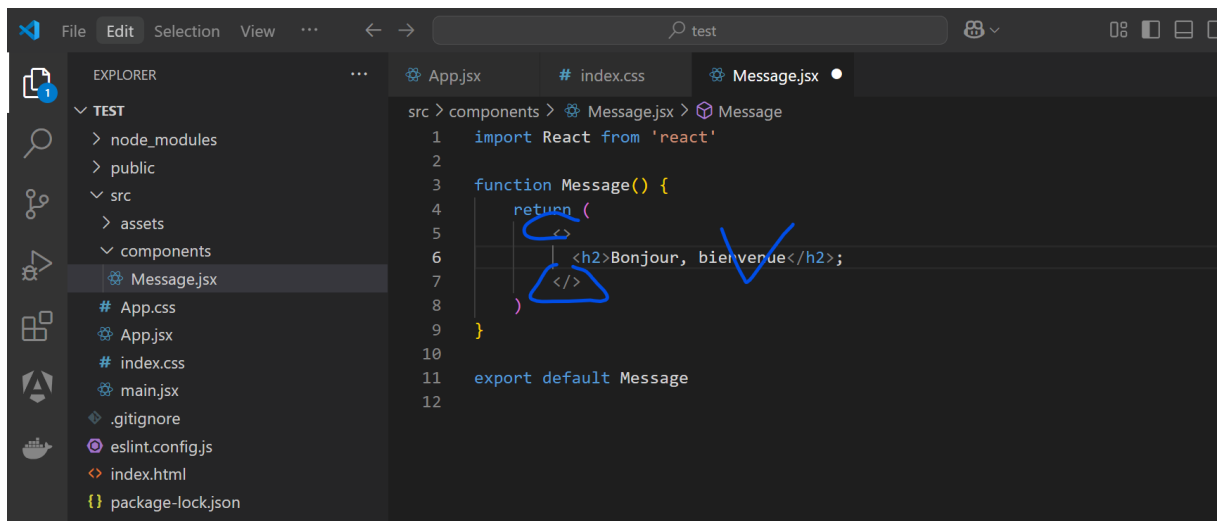
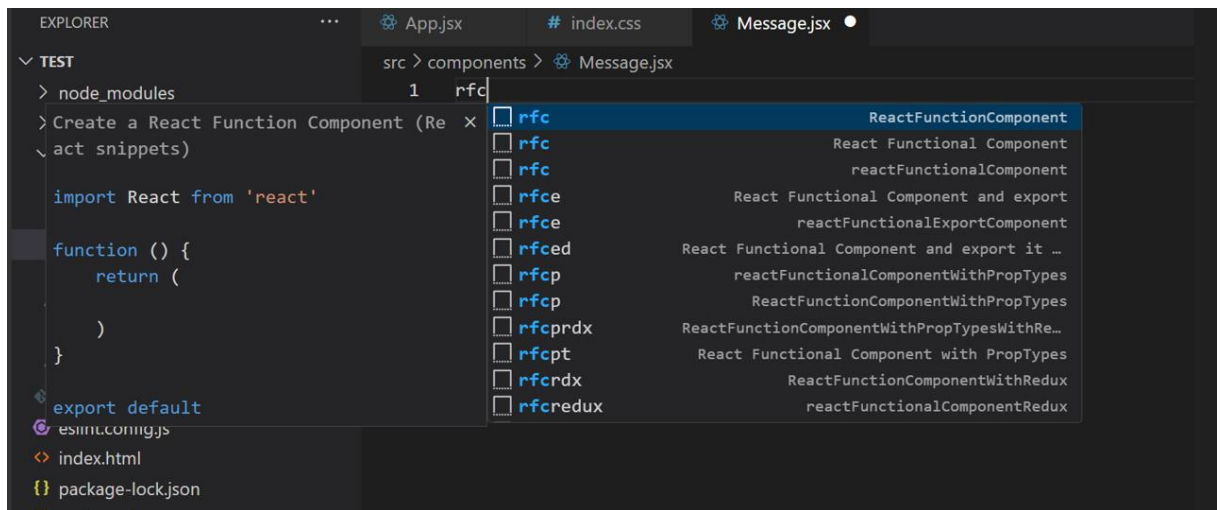
Dans le dossier `src/components/`, crée un fichier `Message.jsx` et ajoute ce code :

```
function Message() {  
  
  return <h2> Bonjour, bienvenue</h2>;  
  
}
```

```
export default Message;
```

Si tu utilises **React avec l'extension ES7+ React/Redux/React-Native snippets** sur VS Code, tu peux taper des raccourcis comme pour créer un composant:

- `rafce` → **React Arrow Function Component with Export**
- `rfce` → **React Function Component with Export**
- `rfc` → **React Function Component**
- `rafc` → **React Arrow Function Component**



◆ Utilisation du composant dans App . jsx

Dans App . jsx, importe et utilise le composant :

```
import Message from "../components/Message";
```

```
function App() {
```

```
  return (
```

```
    <div>
```

```
      <h1>Mon Application</h1>
```

```

    <Message />

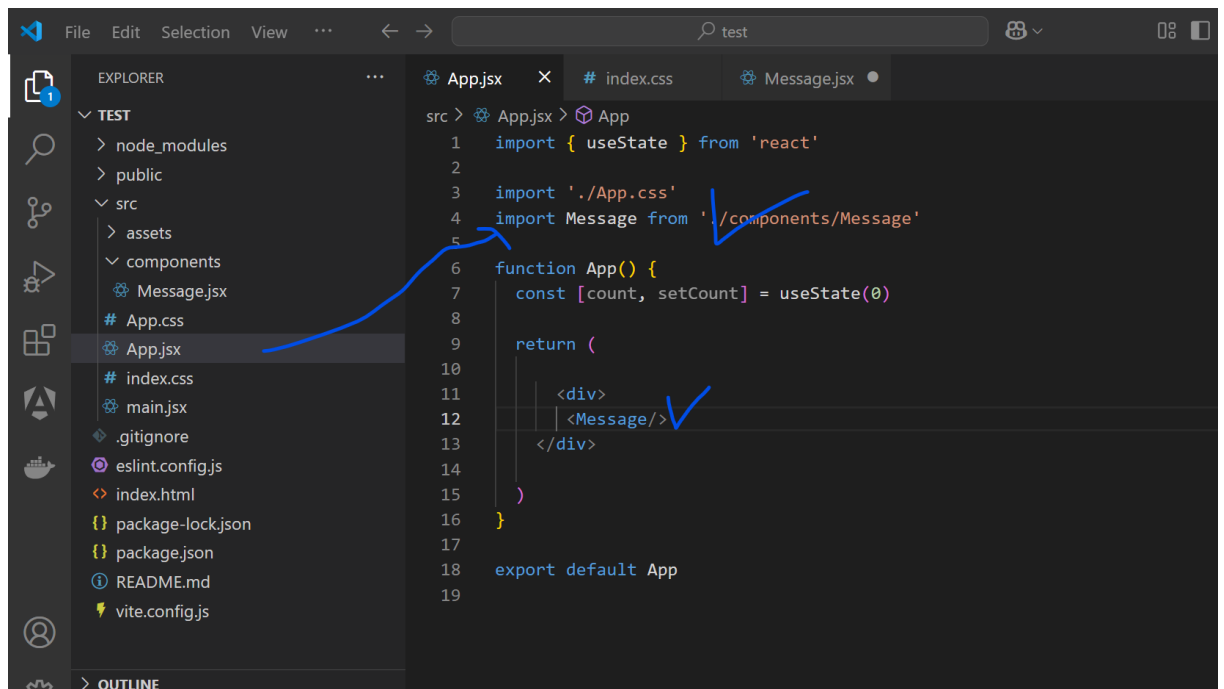
  </div>

);

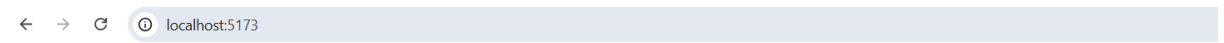
}

```

export default App;



✓ **Résultat** : L'application affiche un message grâce à un composant réutilisable.



Bonjour, bienvenue

;

3. Passage de données avec les Props

Qu'est-ce qu'une **prop** ?

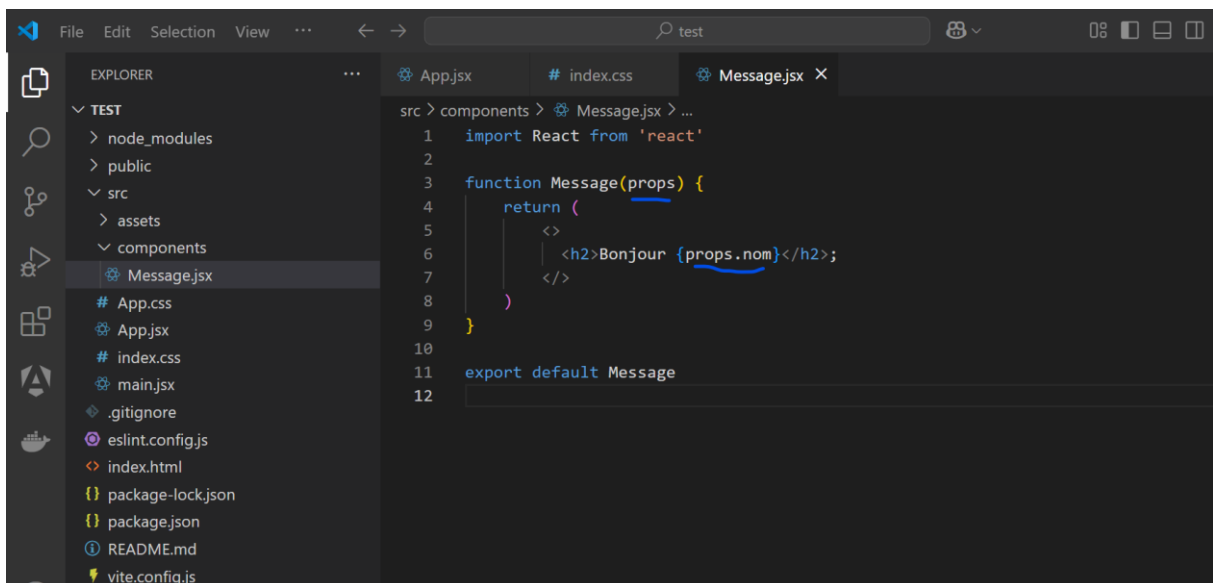
Les **props** (propriétés) permettent de **transmettre des données** d'un parent à un enfant.

◆ Exemple d'un composant avec des props

Modifions `Message.jsx` pour afficher un message personnalisé :

```
function Message(props) {  
  
  return <h2> Bonjour, {props.nom} !</h2>;  
  
}
```

```
export default Message;
```



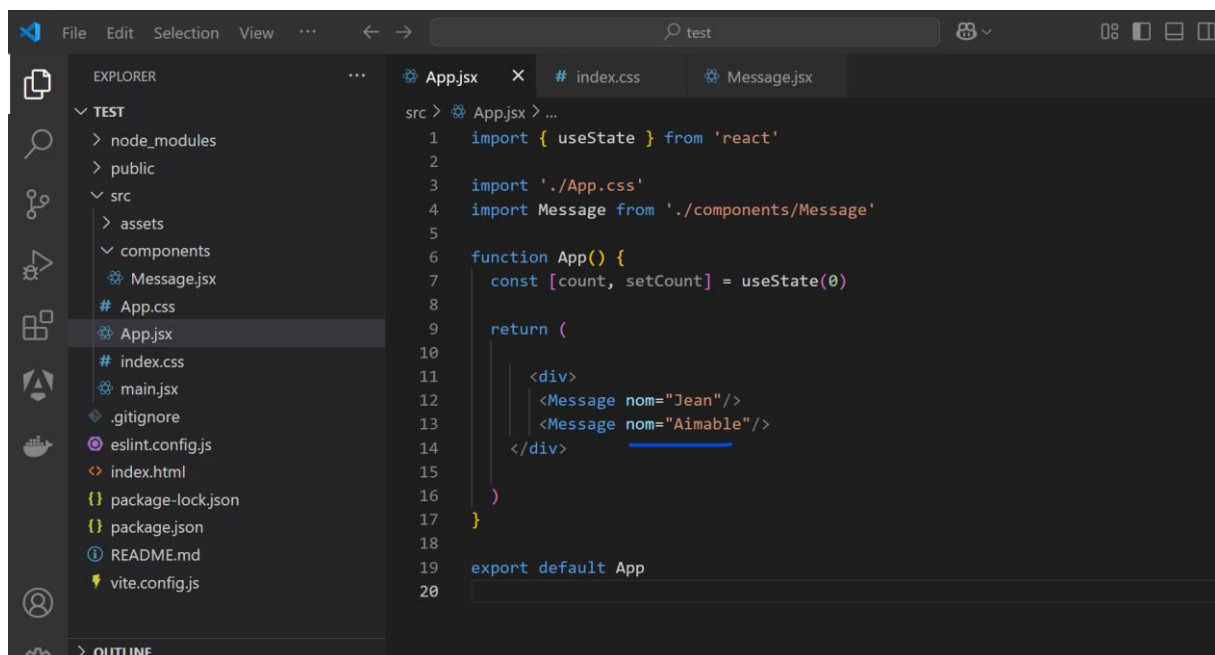
◆ Utilisation avec des valeurs dynamiques

Dans `App.jsx`, passe un nom en **prop** :

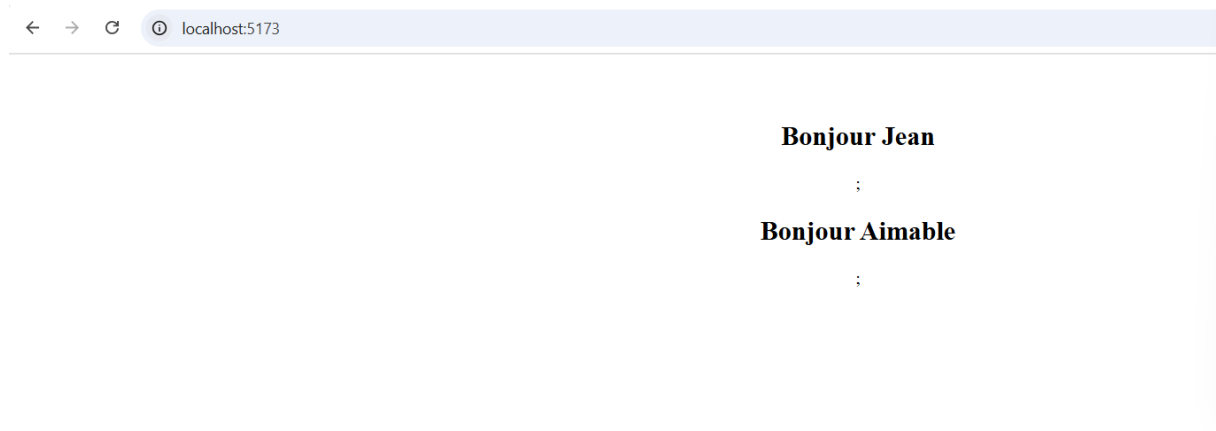
```
import Message from "../components/Message";
```

```
function App() {  
  
  return (  
  
    <div>  
  
      <h1>Mon Application</h1>  
  
      <Message nom="Jean" />  
  
      <Message nom="Aimable" />  
  
    </div>  
  
  );  
  
}
```

```
export default App;
```



✓ Résultat :



4. Composants avec plusieurs props

Un composant peut recevoir **plusieurs props**.

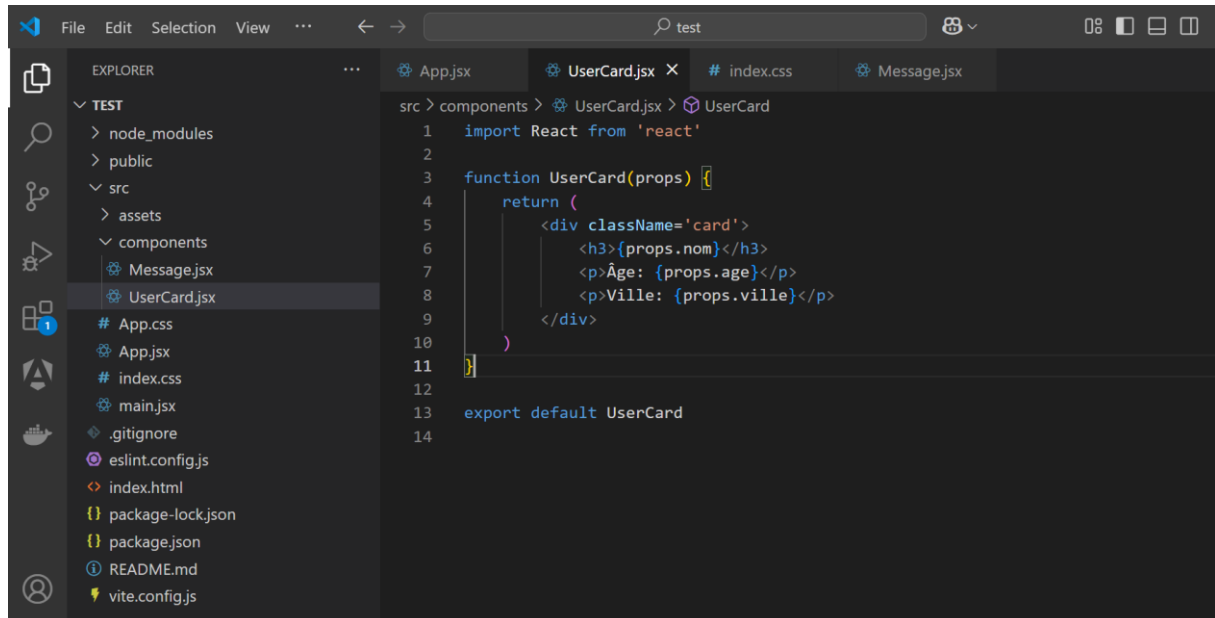
◆ Exemple : Une carte utilisateur

Dans `src/components/UserCard.jsx` :

```
function UserCard(props) {  
  return (  
    <div className="card">  
      <h3>{props.nom}</h3>  
      <p>Âge : {props.age}</p>  
      <p>Ville : {props.ville}</p>  
    </div>  
  );  
}
```

```
}
```

```
export default UserCard;
```



✦ Utilisation dans App.jsx

```
import UserCard from "../components/UserCard";
```

```
function App() {
```

```
  return (
```

```
    <div>
```

```
      <h1>Cfitech</h1>
```

```
      <UserCard nom="Michel" age={20} ville="Bruxelles" />
```

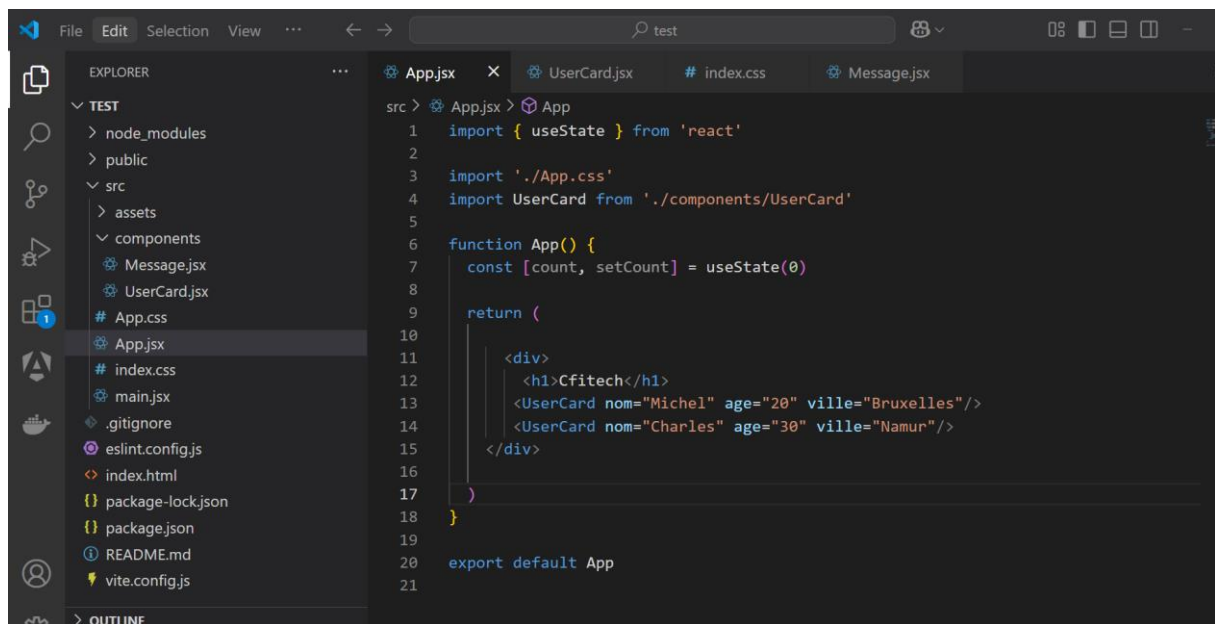
```
      <UserCard nom="Charles" age={30} ville="Namur" />
```

```
    </div>
```

```
  );
```

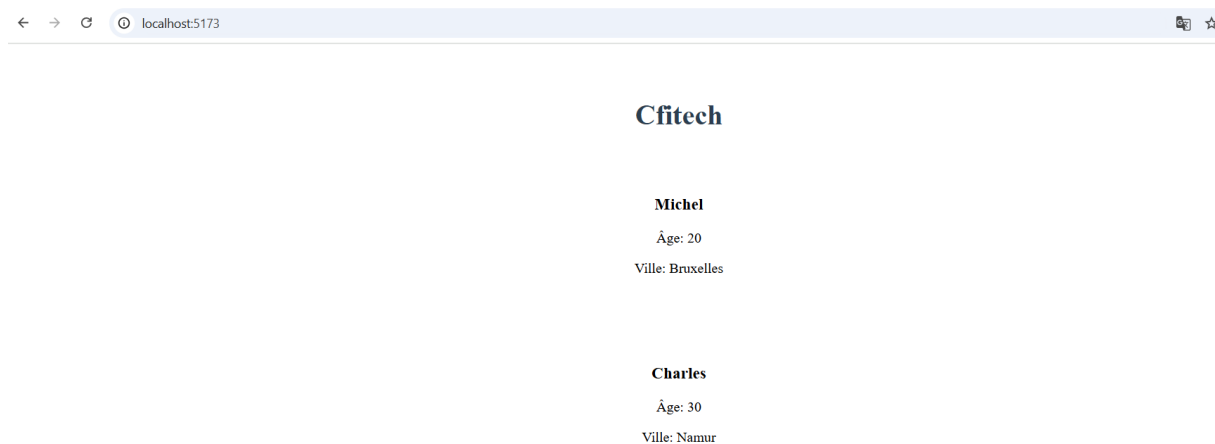
}

export default App;



```
src > App.jsx > App
1  import { useState } from 'react'
2
3  import './App.css'
4  import UserCard from './components/UserCard'
5
6  function App() {
7    const [count, setCount] = useState(0)
8
9    return (
10      <div>
11        <h1>Cfitech</h1>
12        <UserCard nom="Michel" age="20" ville="Bruxelles"/>
13        <UserCard nom="Charles" age="30" ville="Namur"/>
14      </div>
15    )
16  }
17
18  export default App
19
20
21
```

✓ Résultat :



Résumé

- ✓ Les **composants** permettent de découper l'UI en morceaux réutilisables.
- ✓ Les **props** permettent de transmettre des données entre les composants.
- ✓ Les **composants fonctionnels** sont la méthode recommandée en React.

◆ Composants Fonctionnels vs Class Components en React

Dans React, il existe **deux types de composants** :

1. Les **composants fonctionnels** (modernes, plus simples ✓)
2. Les **composants de classe** (ancienne méthode, avant les Hooks ⚠)

Depuis l'introduction des **Hooks** (React 16.8), les **composants fonctionnels sont privilégiés**

1. Composants Fonctionnels (Recommandé ✓)

Un **composant fonctionnel** est une **fonction JavaScript** qui retourne du JSX.
Il est **plus simple, plus lisible et plus performant** que les composants de classe.

Exemple : Composant Fonctionnel

```
function Message(props) {  
  
  return <h2> Bonjour, {props.nom} !</h2>;  
  
}
```

```
export default Message;
```

✓ Avantages des composants fonctionnels

- ✓ Syntaxe plus **simple et concise**
- ✓ Meilleure **lisibilité et maintenabilité**
- ✓ **Performance améliorée** (moins de code)
- ✓ Supporte les **Hooks** (`useState`, `useEffect`...)

2. Composants de Classe (Ancienne Méthode)

Avant les Hooks, on utilisait des **classes** pour créer des composants avec un état (*state*). Ils sont **plus lourds et plus complexes** que les composants fonctionnels.

Exemple : Composant de Classe

```
import React, { Component } from "react";

class Message extends Component {

  render() {

    return <h2>Bonjour, {this.props.nom} !</h2>;

  }

}

export default Message;
```

Inconvénients des composants de classe

- ✗ **Syntaxe plus lourde** (besoin d'utiliser `this.props`, `this.state`, etc.)
- ✗ **Plus difficile à lire et à comprendre**
- ✗ **Moins performant** que les composants fonctionnels

3. Gestion de l'État : `useState` vs `this.state`

Avec un composant fonctionnel (`useState`) 

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

```
function Counter() {
```

```

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

return (
  <div>
    <h2>Compteur : {count}</h2>
    <button onClick={() => setCount(count + 1)}>➕ Augmenter</button>
  </div>
);
}

export default Counter;

```

Avec un composant de classe (this.state) ⚠

```

import React, { Component } from "react";

class Counter extends Component {
  constructor(props) {
    super(props);
    this.state = { count: 0 };
  }

  increment = () => {

```

```

    this.setState({ count: this.state.count + 1 });
  };



  render() {
    return (
      <div>
        <h2>Compteur : {this.state.count}</h2>
        <button onClick={this.increment}>+ Augmenter</button>
      </div>
    );
  }
}

export default Counter;

```

✓ **Avec** useState, le code est plus simple et plus court

Conclusion : Quel type de composant utiliser ?

Critère	Composant Fonctionnel 	Composant de Classe 
Simplicité	✓ Très simple	✗ Complexe
Performance	✓ Optimisé	✗ Moins performant
Lisibilité	✓ Facile à lire	✗ Difficile à comprendre
Utilisation des Hooks	✓ Oui (<code>useState</code> , <code>useEffect</code>)	✗ Non
État et Lifecycle	✓ Plus intuitif avec Hooks	✗ <code>this.state</code> et <code>componentDidMount</code> ...

NB : En React, `props` et `useState` sont deux concepts fondamentaux qui servent des objectifs différents :

1. `props` (Propriétés)

- Les `props` sont utilisées pour **transmettre des données** d'un composant parent à un composant enfant.
- Elles sont **immuables** dans le composant enfant (le composant qui les reçoit ne peut pas les modifier directement).
- Elles permettent de rendre les composants **réutilisables** et **dynamiques**.

◆ Exemple :

```
function Enfant({ message }) {  
  
  return <h1>{message}</h1>;  
  
}
```

```
function Parent() {  
  
  return <Enfant message="Bonjour !" />;  
  
}
```

Ici, `message` est une prop passée du Parent au Enfant.

★ 2. `useState` (État local)

- `useState` est un **hook** qui permet à un composant fonctionnel de gérer son propre état interne.
- Contrairement aux `props`, l'état **peut être modifié** par le composant lui-même.
- Les changements d'état provoquent un **re-render** du composant.

◆ Exemple

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

```
function Compteur() {  
  
  const [compte, setCompte] = useState(0);  
  
  
  return (  

```

```

<div>

  <p>Valeur : {compte}</p>

  <button onClick={() => setCompte(compte + 1)}>+1</button>

</div>

);

}

```

Ici, `compte` est un état local qui change lorsqu'on clique sur le bouton.

vs Différences Clés :

Critère	props	useState
Définition	Valeurs passées d'un parent à un enfant	Valeur locale propre au composant
Modifiable ?	❌ Non (immuable)	✅ Oui (avec <code>setState</code>)
Responsabilité	Dépend du parent	Dépend du composant lui-même
Provoque un Re-render ?	❌ Non	✅ Oui, quand mis à jour
Usage principal	Partage de données entre composants	Gestion des données internes du composant

En React, le **rendering** (rendu) est le processus qui permet d'afficher ou de mettre à jour l'interface utilisateur d'un composant.

💡 Comment fonctionne le rendu en React ?

1. **Initial Render (Premier rendu)**
 - Lorsqu'un composant est monté pour la première fois, React exécute sa fonction et retourne un arbre de **JSX** (ou d'éléments React) qui sera converti en HTML et affiché dans le DOM.
2. **Re-render (Mise à jour du rendu)**
 - Un composant se re-render lorsqu'il **reçoit de nouvelles props**, qu'il **met à jour son state**, ou qu'un **parent est mis à jour**.
 - React compare l'ancien et le nouveau rendu et applique uniquement les changements nécessaires (grâce au **Virtual DOM** et au **Diffing Algorithm**).

Quand utiliser quoi ?

✓ **Utiliser `props`** lorsque vous devez **transmettre** des données à un composant enfant sans qu'il ait besoin de les modifier.

✓ **Utiliser `useState`** lorsque le composant doit **gérer et modifier** ses propres données dynamiques.

◆ Exemple combiné `props` + `useState`

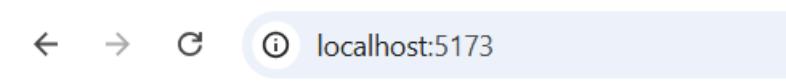
Dans cet exemple, le composant parent transmet une valeur initiale via `props`, et l'enfant peut la modifier avec `useState`.

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

```
function Enfant({ valeurInitiale }) {  
  
  const [compteur, setCompteur] = useState(valeurInitiale);  
  
  return (  
  
    <div>  
  
      <p>Compteur : {compteur}</p>  
  
      <button onClick={() => setCompteur(compteur + 1)}>+1</button>  
  
    </div>  
  
  );  
}
```

```
function Parent() {  
  
  return <Enfant valeurInitiale={5} />;  
  
}
```

```
export default Parent;
```



Compteur : 5



Explication :

1. Le **parent (Parent)** passe `valeurInitiale={5}` en **prop** au composant enfant.
2. L'**enfant (Enfant)** reçoit cette prop et l'utilise comme valeur initiale pour son `useState`.
3. Quand on clique sur le bouton, l'état local `compteur` change sans affecter `valeurInitiale` dans `Parent`

✓ Ainsi, **props** est utilisé pour passer une donnée initiale, et `useState` permet de la modifier localement !

🔗 Outils de Développement React (React DevTools)

Lorsqu'on développe une application React, il est essentiel d'avoir les **bons outils** pour **déboguer**, **analyser** et **optimiser** notre code. L'un des meilleurs outils pour ça est

React DevTools.

1.Qu'est-ce que React DevTools ?

React DevTools est une **extension pour Chrome et Firefox** qui permet de :

- ✓ **Explorer la structure des composants**
- ✓ **Voir et modifier les props et le state en direct**
- ✓ **Analyser les performances des composants**
- ✓ **Déboguer plus facilement les applications React**

2.Installation de React DevTools

Option 1 : Installer l'extension navigateur (recommandé)

🔗 **Chrome** : React Developer Tools - Chrome Web Store

🔗 **Firefox** : [React Developer Tools - Add-ons for Firefox](#)

Une fois installée, l'onglet "**Components**" et "**Profiler**" apparaîtront dans les **Outils de développement** (F12 ou Ctrl + Shift + I).

Pratique : Création d'un Mini-Projet avec des Composants Réutilisables

Objectif du projet : Une Liste de Cartes Utilisateurs

Nous allons créer une **application simple** qui affiche une liste d'utilisateurs sous forme de cartes. Chaque carte affichera le **nom**, l'**âge** et la **ville** d'un utilisateur.

Concepts abordés :

- ✓ Création de **composants réutilisables**
- ✓ Passage de **props**
- ✓ Organisation du projet **modulaire**