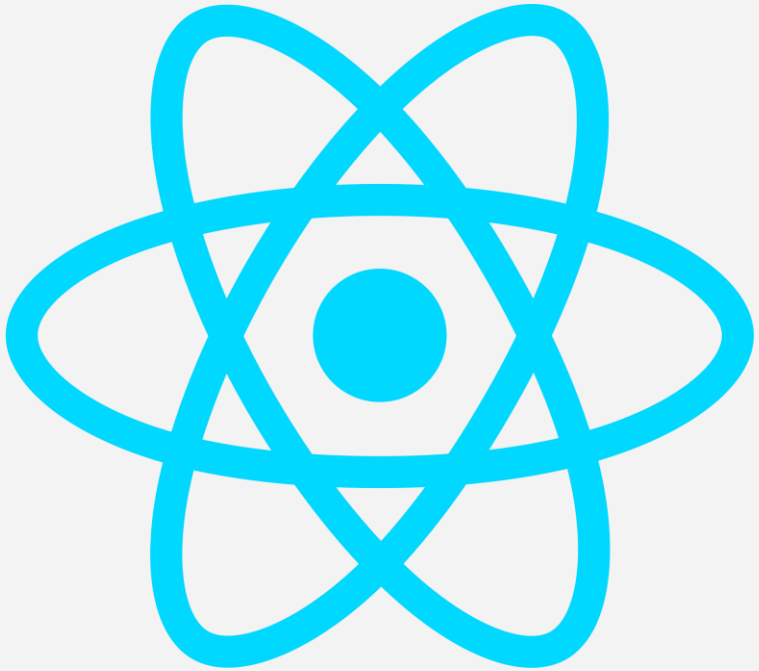


React

Une bibliothèque JavaScript qui permet de créer des interfaces utilisateur interactives et réutilisables.

About

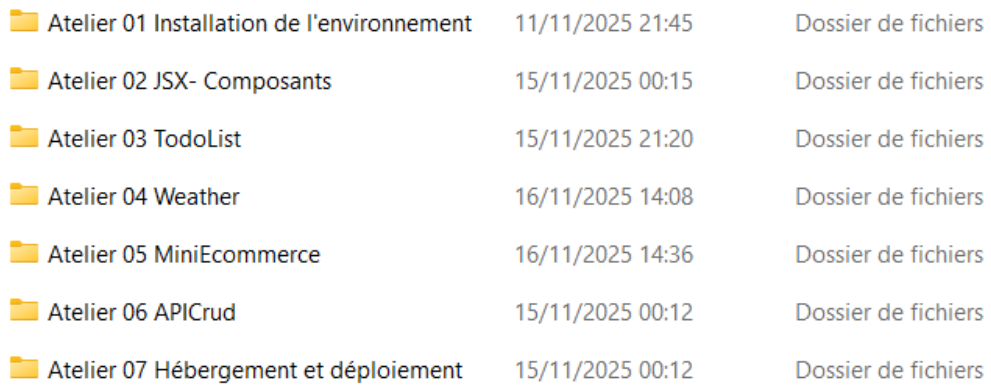
- Kamel ABBASSI, Docteur en informatique et formateur spécialisé en développement web.
- Solides compétences en PHP, Symfony, Laravel, JavaScript, ReactJS, Django, Excel et bases de données.
- Créateur de formations pratiques disponibles en ligne sur Udemy et Hotmart, destinées aux débutants et aux professionnels.
- Passionné par l'apprentissage, l'innovation et l'accompagnement des apprenants, avec contacts :
  - LinkedIn : linkedin.com/in/abbassikamel
  - Email : abbassi.kamel@gmail.com
  - Tél : 26 388 202

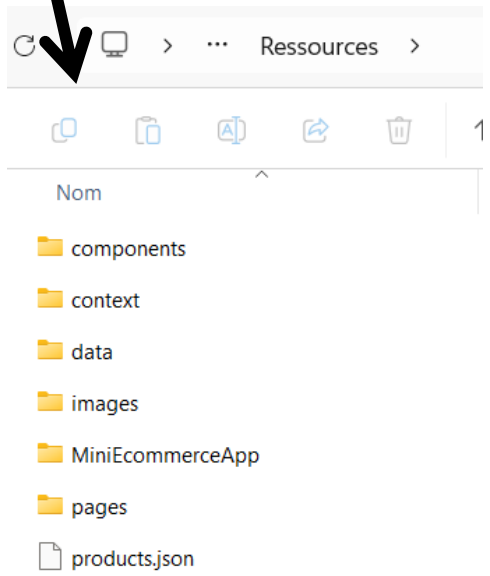
Plan

- Comprendre les bases et le fonctionnement de React.
- Construire et organiser des composants réutilisables.
- Gérer l'état avec useState et les props.
- Maîtriser les événements et les formulaires contrôlés.
- Afficher, filtrer et manipuler des listes.
- Utiliser les Hooks essentiels (useEffect, etc.).
- Consommer des API et réaliser des opérations CRUD.
- Naviguer entre pages avec React Router.
- Appliquer différentes méthodes de stylisation en React.
- Sauvegarder des données avec localStorage.
- Réaliser des mini-projets pratiques (TodoList, Weather, Ecommerce, APICrud).
- Préparer et déployer une application React (Hébergement et déploiement).

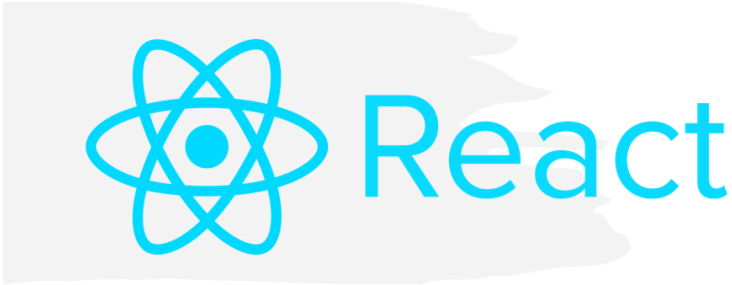
Organisation

Ateliers





Support de Cours



Une bibliothèque JavaScript qui permet de créer des interfaces utilisateur interactives et réutilisables.

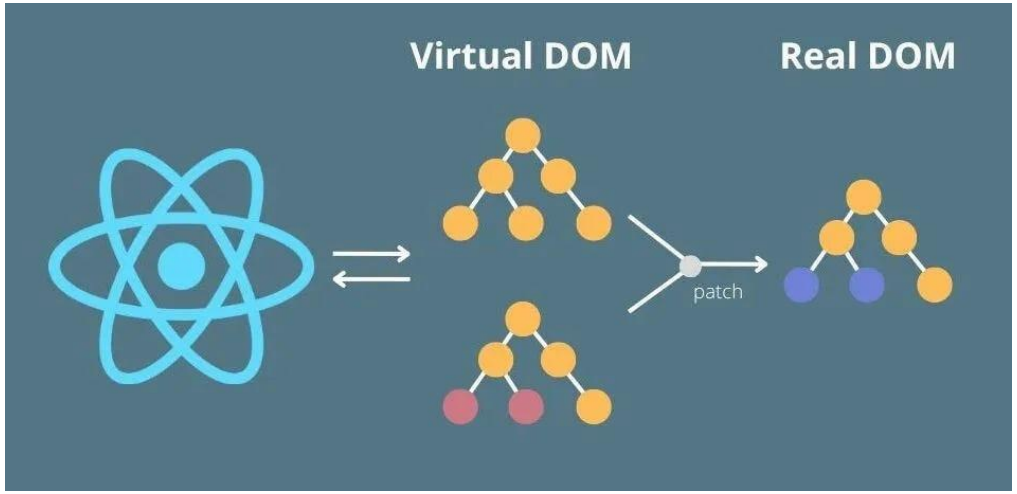
Qu'est-ce que React ?

- React, parfois appelé framework JavaScript frontal, est une bibliothèque JavaScript créée par Facebook.
- React est un outil pour créer des composants d'interface utilisateur
- React est utilisé pour créer des applications d'une seule page (**SPA**, Single Page Application)

Langage de programmation : JavaScript
Créateur : Meta
Dernière version : 19.2.0 (1er octobre 2025)
Licence : Licence MIT
Première version : 29 mai 2013 (12 ans, 170 jours)
Système d'exploitation : Multiplateforme (d)
Type : Bibliothèque JavaScript

Comment fonctionne React ?

- React crée un DOM(Document Object Model) virtuel en mémoire.
- Toutes les manipulations se font d'abord dans ce DOM virtuel, pas directement dans le navigateur.
- Ensuite, React met à jour seulement ce qui a changé dans le DOM réel.
- Cela permet des mises à jour plus rapides et efficaces.



Environnement React (1/4)

Vite est un **outil de build et de développement rapide** pour les projets JavaScript modernes (comme React, Vue ou Vanilla JS).

- Il sert à **lancer un serveur de développement** ultra-rapide.
- Il permet de **compiler et regrouper le code** pour la production.
- Comparé à d'autres outils comme **Webpack**, Vite est **beaucoup plus rapide** pour démarrer et recharger une application.

Environnement React (2/4)

- Nous avons besoin de **npm** qui est inclus avec **NODE.JS**
- **Node.js** est un environnement d'exécution JavaScript côté serveur.
  - Autrement Permet d'**exécuter du JavaScript en dehors du navigateur**

- Créer une application React on utilise:

npm create vite@latest app

```
/usr/src/app/test # npm create vite@latest app

> app@0.0.0 npx
> create-vite app

◆  Select a framework:
   ● Vanilla
   ○ Vue
   ○ React
   ○ Preact
   ○ Lit
   ○ Svelte
   ○ Solid
   ○ Qwik
   ○ Angular
   ○ Marko
   ○ Others
```

Environnement React (3/4)

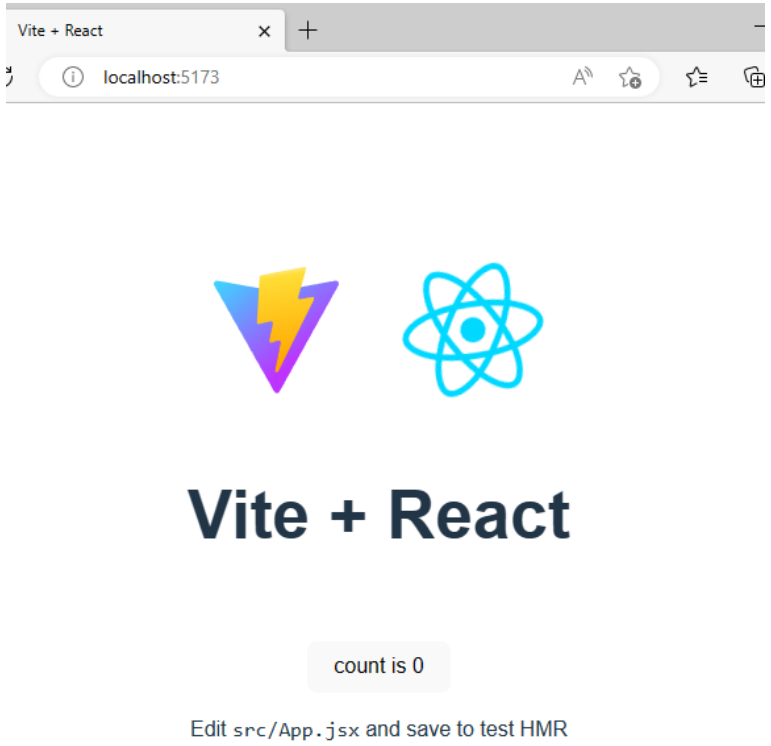
- Ensuite , suivre les étapes et rependre aux questions

```
/usr/src/app/test # npm create vite@latest app

> app@0.0.0 npx
> create-vite app

│
◇  Select a framework:
│  Preact
│
◇  Select a variant:
│  JavaScript
│
◇  Use rolldown-vite (Experimental)?:
│  No
│
◇  Install with npm and start now?
│  Yes
│
◇  Scaffolding project in /usr/src/app/test/app...
│
◇  Installing dependencies with npm...
\
```

Environnement React (4/4)

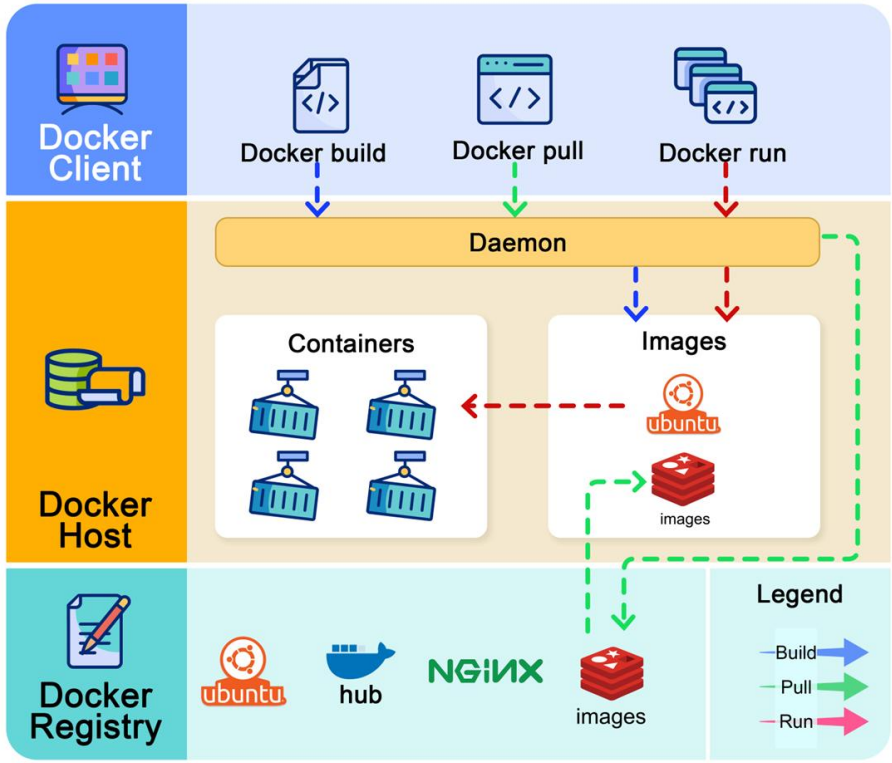
- Vous êtes maintenant prêt à exécuter votre première véritable application React !
- ***cd my-react-app***
- ***npm run dev***

- Ouvrez votre navigateur et tapez **http://localhost:5173**

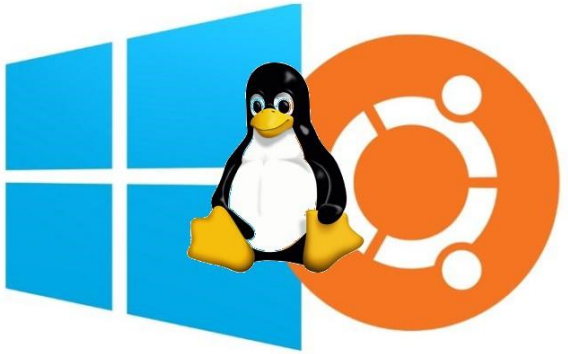


Introduction à Docker

- Docker : outil de virtualisation légère pour exécuter des applications isolées.
- Image : modèle ou "photo" d'un environnement logiciel prêt à l'emploi.
- Conteneur : instance en exécution d'une image.
- Volume : dossier partagé entre conteneur et machine locale.
- Avantages pour ReactJS :
  - Même environnement pour tous (Windows, Linux, macOS).
  - Pas de conflits de version Node.js.
  - Facile à reconstruire ou supprimer.
  - Projet portable sur toute machine avec Docker



Étapes pour installer la première app React dans Docker



- Installer **VS Code** et **Docker Desktop**.
- Activer WSL2 (Windows Subsystem for Linux 2) sur Windows si nécessaire.
- Créer la structure projet :

  C:\ReactProjects\FirstApp\app

  Dockerfile

  docker-compose.yml

Dockerfile :
- FROM node:22-alpine
- WORKDIR /usr/src/app
- EXPOSE 5173

docker-compose.yml :
- Définir service reactapp avec build, ports, volumes, working_dir et environment.

Lancer l’application React

- Construire et démarrer le conteneur :

docker-compose up -d –build

- Entrer dans le conteneur :

docker exec -it FirstApp_container sh

- Créer l’app React avec Vite :

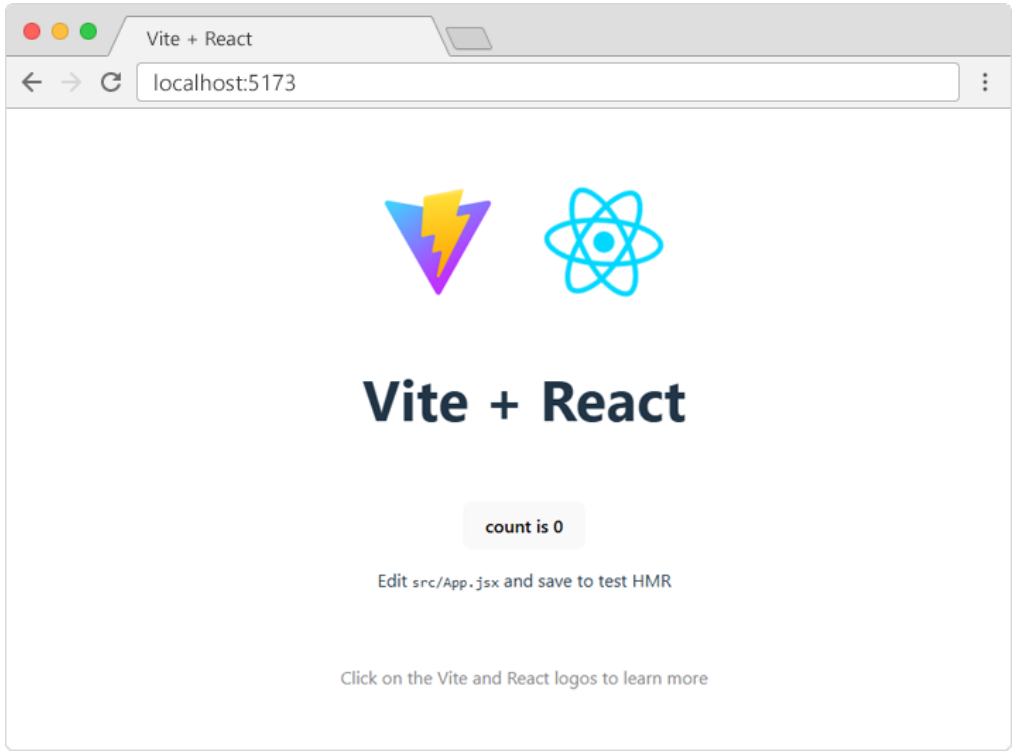
npm create vite@latest . -- --template react

- Lancer le serveur dev :

npm run dev -- --host

- Accéder à l’application :

http://localhost:5173



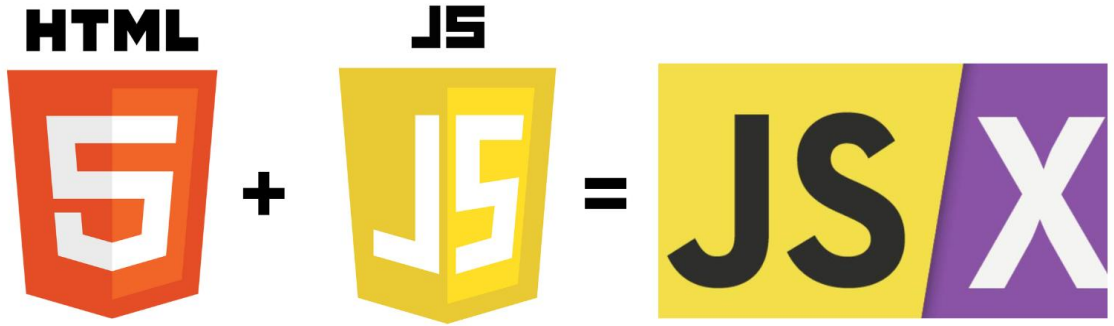
Pratique :

Dans FirstApp

Atelier 01 Installation de l'environnement

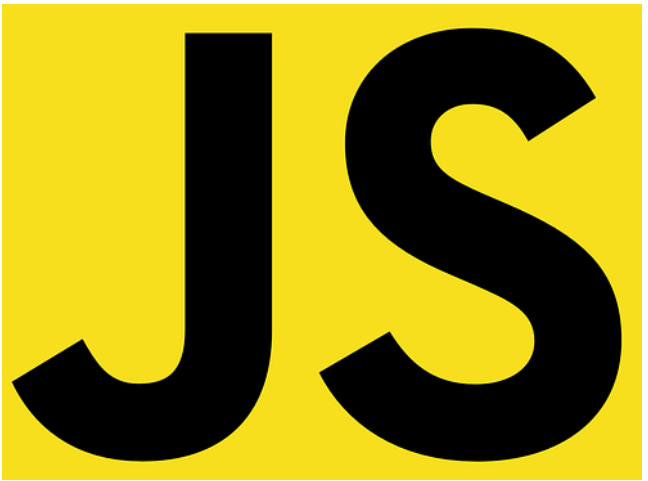
Qu'est-ce que JSX ?

- JSX = JavaScript XML
- Permet d’écrire du HTML directement dans React.
- Facilite l’ajout et la gestion du HTML dans le code React.
- Évite d’utiliser createElement() ou appendChild().
- Convertit automatiquement les balises HTML en éléments React.



Sans JSX (JavaScript pur)

```
<!DOCTYPE html>
<html>
 <body>
  <div id="root"></div>
  <script>
   // créer un élément <h1>
   const title = document.createElement("h1");
   title.textContent = "Bonjour à tous";

   // créer un paragraphe
   const paragraph = document.createElement("p");
   paragraph.textContent = "Ceci est un exemple sans JSX.";

   // ajouter les éléments dans le div root
   const root = document.getElementById("root");
   root.appendChild(title);
   root.appendChild(paragraph);
  </script>
 </body>
</html>
```



Avec React mais **sans JSX** (createElement)

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

const element = React.createElement(
  "div",
  null,
  React.createElement("h1", null, "Bonjour à tous"),
  React.createElement("p", null, "Ceci est un exemple avec createElement.")
);

const root = ReactDOM.createRoot(document.getElementById("root"));
root.render(element);
```

Avec JSX (la manière standard en React)

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

const element = (
  <div>
    <h1>Bonjour à tous</h1>
    <p>Ceci est un exemple avec JSX.</p>
  </div>
);

const root = ReactDOM.createRoot(document.getElementById("root"));
root.render(element);
```

Manipulez des données dans JSX

Insérer des expressions JavaScript dans JSX

- Les accolades **{ }** permettent d'écrire des expressions JS directement dans le JSX.
- Exemples :

```
<div>La grande réponse : { 6 * 7 } </div>     // Maths
<div> { alexa.toUpperCase() } </div>        // Chaînes
<div> { 2 > 0 ? 'Deux est plus grand que zéro' : 'Ceci n'apparaitra jamais' } </div>
// Ternaire
```

Manipulez des données dans JSX

On peut afficher des strings, nombres ou variables directement :

```
const myTitle = "Bonjour React"
<div>{ myTitle }</div>   // string
<div> { 42 } </div>        // nombre
```

Exemple avec variable dans un composant :

```
function Description() {
  const text = "Ici achetez toutes les plantes dont vous avez toujours rêvé« ;
  return <p>{ text }</p>
}
```

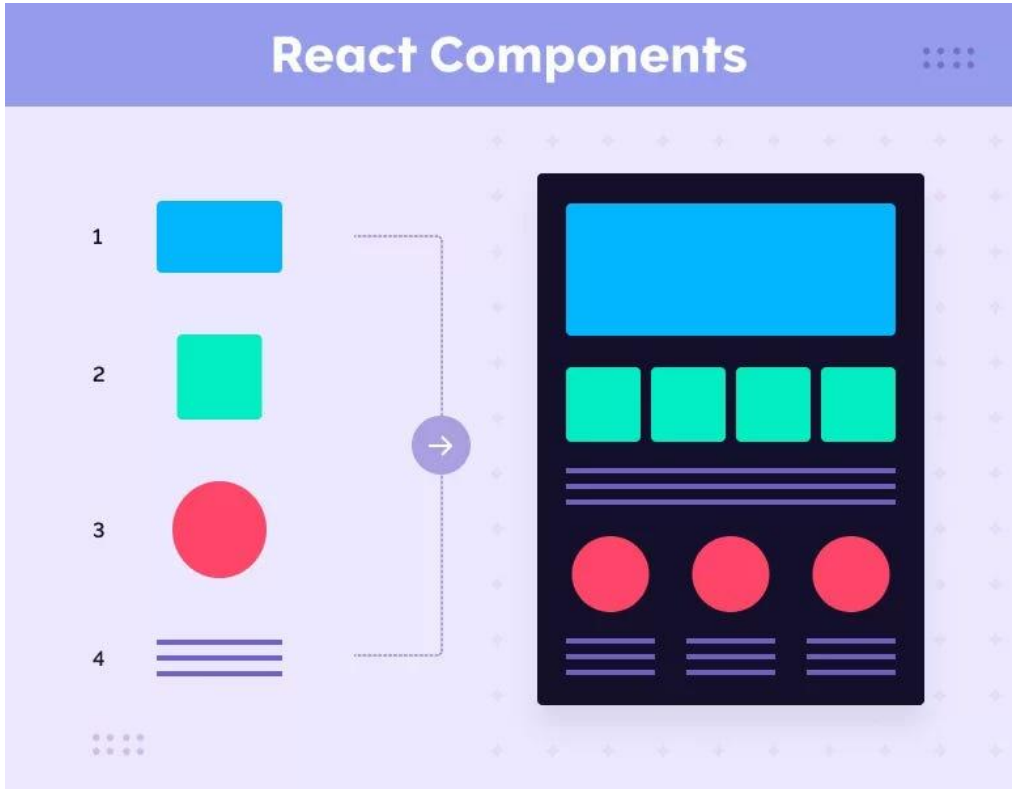
Manipulez des données dans JSX

On peut appliquer des fonctions JS et combiner des chaînes :

```
function Description() {
  const text = "Ici achetez toutes les plantes dont vous avez toujours rêvées"
  const emojis = " 💸 💸 💸"
  return <p>{ text.slice(0, 11) + emojis }</p>
}
```

React Components

- 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.
- Deux types:
  - Composants de classe
  - Composants de fonction.
- Dans cette formation, nous nous concentrerons sur les composants de fonction



Exemple: Composant de **classe**

- Le nom du composant doit :
  - commencer par une **lettre majuscule**.
  - Hérité de **React.Component**

- Le composant nécessite une méthode **render**(),
  - cette méthode renvoie HTML

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

class Car extends Component {
 render() {
  return (
    <h2>Hello, je suis votre bagnole 🚙 </h2>
  );
 }
}

export default Car;
```

Exemple: Composant de **fonction**

- Un composant Function renvoie du HTML comme un composant Class.
- Il utilise moins de code et est plus simple à écrire.
- Plus facile à comprendre et maintenir.

```
import React from "react";

function Car() {
  return (
    <h2>Hello, je suis votre bagnole 🚙 </h2>
  );
}

export default Car;
```

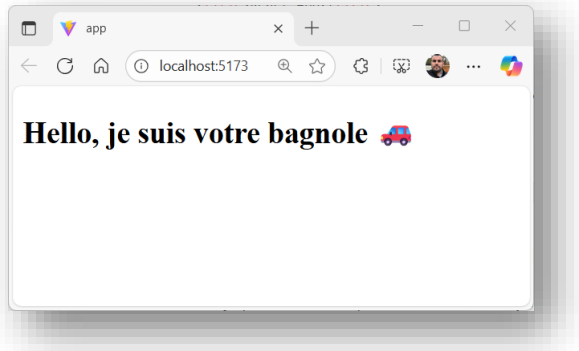
Rendu d'un composant

Un composant appelé Car, qui renvoie un élément **<h2>.**

- Pour utiliser ce composant dans votre application, utilise une syntaxe similaire à celle du HTML normal : **<Car />**

```
// app/src/main.jsx
import { StrictMode } from 'react'
import { createRoot } from 'react-dom/client'
import Car from './Car.jsx'

createRoot(document.getElementById('root')).render(
  <StrictMode>
    <Car />
  </StrictMode>,
)
```

```
{/* Fichier index.html */}

<div id="root"></div>
```



Composant dans un autre Composant

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

```
// Composant Car
function Car() {
  return <h2>Voiture</h2>;
}

// Composant Garage qui contient Car
function Garage() {
  return (
    <div>
      <h1>Mon Garage</h1>
      <Car />
    </div>
  );
}

export default Garage;
```

Pratique :

Dans FirstApp

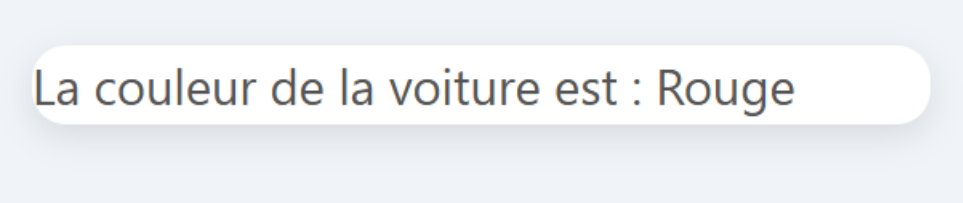
Atelier 02.01 JSX composant

Component Constructor

- Le constructor() s’exécute lors de la création d’un composant.
- Il initialise les propriétés internes, notamment le state.
- Les données internes du composant sont stockées dans state.
- Il doit appeler super() pour respecter l’héritage.
- super() permet d’exécuter le constructeur du parent et d’accéder aux fonctionnalités de React.

Component Constructor

```
import React from "react";
class Car extends React.Component {
  constructor() {
    super();
    // Définition du state
    this.state = {
      color: "Rouge"
    };
  }
  render() {
    return (
      <p>La couleur de la voiture est : {this.state.color}</p>
    );
  }
}

export default Car;
```



Le rôle de super() dans le constructor

Dans les composants classe, on écrit :

```
constructor(props) {
  super(props);
}
```

Pourquoi ?

`super(props)` appelle le constructeur de `React.Component`,
ce qui permet à notre composant **d'utiliser `this.props`** correctement.

Sans `super(props)`, React ne peut pas initialiser le composant → **erreur**.

Qu’est-ce que state ?

- **Définition**
  - state représente l’état interne d’un composant.
  - Il permet de stocker des données qui peuvent changer dans le temps.
- **Caractéristiques**
  - ✓Modifiable **à l’intérieur du composant**
  - ✓Le rendu **se met à jour automatiquement** lorsque `state` change
  - ✓Utilisé pour gérer **interactions, formulaires, compteurs, affichage dynamique…**

Qu’est-ce que state ?

```
class Car extends React.Component {
  constructor() {
    super();
    this.state = { speed: 0 };
  }

  render() {
    return <p>Vitesse actuelle : {this.state.speed} km/h</p>;
  }
}
```

Modification de l’état (State) dans un composant classe

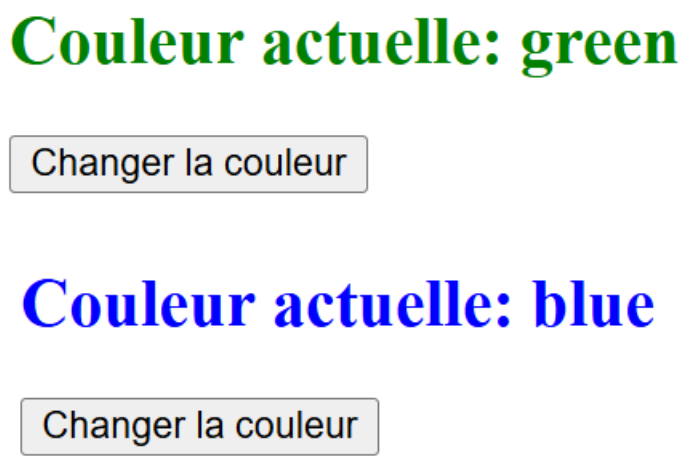
- Pour modifier une valeur dans **state**, on utilise la méthode **this.setState().**
- Quand le state change :
  - Le composant est re-rendu automatiquement.
  - L’affichage se met à jour avec les nouvelles valeurs.

Exemple : changer la couleur via un bouton

```
// Fichier : src/components/ColorChangerClass.jsx
import React from "react";

class ColorChangerClass extends React.Component {
 constructor() {
  super();
  this.state = { color: "blue" };
  this.changeColor = this.changeColor.bind(this);
 }

 changeColor() {
  this.setState({ color: this.state.color === "blue" ? "green" : "blue" });
 }

 render() {
  return (
   <div>
    <h2 style={{ color: this.state.color }}>
Couleur actuelle: {this.state.color}</h2>
    <button onClick={this.changeColor}>
Changer la couleur</button>
   </div>
  );
 }
}

export default ColorChangerClass;
```

Modification de l’état dans un composant fonctionnel

- Dans un composant fonction, on utilise **useState** à la place de **this.state** et **this.setState**.
- La logique est similaire : quand l’état change, le composant est re-rendu automatiquement.




```
// Fichier : src/components/ColorChangerFunction.jsx
import { useState } from "react";

export default function ColorChangerFunction() {
 const [color, setColor] = useState("blue");

 const changeColor = () => {
  setColor(color === "blue" ? "green" : "blue");
 };

 return (
  <div>
   <h2 style={{ color }}>Couleur actuelle: {color}</h2>
   <button onClick={changeColor}>Changer la couleur</button>
  </div>
 );
}
```

Qu’est-ce que props ?

Définition

- props = **valeurs passées au composant depuis l’extérieur** (comme des paramètres de fonction).

Caractéristiques

- **Immuables** (on ne doit jamais les modifier)
- Permettent de **réutiliser** un composant avec des données différentes
- Servent à **transmettre des informations** d’un composant parent → enfant

```
function Car(props) {
  return <p>Modèle : {props.model}</p>;
}

// Utilisation
<Car model="BMW" />
<Car model="Toyota" />
```

Différence entre **props** et **state**

Caractéristique	`props`	`state`
Modifiable ?	❌ Non (lecture seule)	✅ Oui (interne au composant)
Vient d'où ?	Du composant parent	Du composant lui-même
Sert à quoi ?	Recevoir des données	Gérer des données qui évoluent

Hook

- Un **Hook** permet d'ajouter des fonctionnalités (state, effets, contexte…) à un composant fonctionnel de manière simple et réutilisable.
- Les **Hooks** sont sortis officiellement avec **React 16.8**, publié en **février 2019**
- **useEffect** est un Hook qui permet d'exécuter du code secondaire (side effects) dans un composant fonction
- Les side effects peuvent être :
  - Requêtes API (fetch)
  - Modification du DOM (titre de la page, scroll…)
  - Timers (setInterval, setTimeout)
  - Écoute d'événements (window resize, key press…)
- Il remplace en partie les méthodes de cycle de vie des composants classes comme componentDidMount, componentDidUpdate, et componentWillUnmount.

useState

- const [state, setState] = useState(valeurInitiale);
  - state → la valeur actuelle
  - setState → fonction pour modifier cette valeur
  - valeurInitiale → la valeur par défaut (nombre, texte, booléen, objet…)

useEffect : Syntaxe de base

```
useEffect(() => {
  // code à exécuter
});
```

- Par défaut, ce code s’exécute après chaque rendu du composant.

useEffect : Avec tableau de dépendances

```
useEffect(() => {
  // code exécuté uniquement quand `variable` change
}, [variable]);
```

Explications :

- Le `useEffect` dépend de `count`.
- Chaque fois que `count` change → le titre de la page est mis à jour.
- Si `count` ne change pas, le code dans l'effet ne se réexécute pas.

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

export default function CounterTitle() {
 const [count, setCount] = useState(0);

 useEffect(() => {
  document.title = `Compteur : ${count}`;
 }, [count]); // l'effet s'exécute uniquement quand "count" change

 return (
  <div>
   <p>Compteur : {count}</p>
   <button onClick={() => setCount(count + 1)}>+1</button>
  </div>
 );
}
```

useEffect : Nettoyage (Cleanup)

- Certains effets doivent être nettoyés pour éviter des problèmes comme des fuites de mémoire, des timers qui continuent de tourner ou des écouteurs d’événements persistants.
- Pour cela, useEffect peut retourner une fonction qui s’exécute lors du démontage du composant ou avant le prochain effet.

useEffect : Nettoyage (Cleanup)

```
import { useState, useEffect } from "react";
export default function Timer() {
  const [seconds, setSeconds] = useState(0);
  useEffect(() => {
    const interval = setInterval(() => {
      setSeconds(prev => prev + 1);
    }, 1000);
    // Fonction de nettoyage : arrêt du timer
    // si le composant est démonté
    return () => clearInterval(interval);
  }, []); // [] → s'exécute une seule fois au montage

  return (
    <div>
      <p>Temps écoulé : {seconds} secondes</p>
      <button onClick={() => setSeconds(0)}>Réinitialiser</button>
    </div>
  );
}
```

Explications :

- setInterval démarre le compteur.
- return () => clearInterval(interval) → nettoie le timer si le composant est retiré du DOM.
- Tableau vide [] → l'effet s'exécute une seule fois au montage (comme componentDidMount).

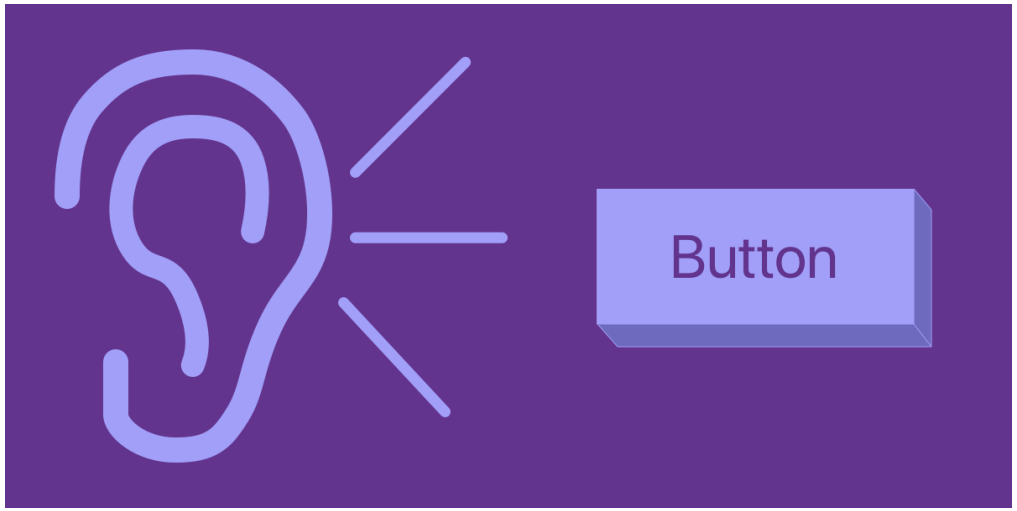
Pratique :

Dans FirstApp

Atelier 02.02 Propos, State, super

React Events - Définition

- Un événement est une action réalisée par l'utilisateur ou le navigateur (ex : clic, saisie clavier, survol).
- React permet de réagir à ces événements pour rendre l’interface interactive.
- Exemple d’événements courants :
  - click, change, submit, mouseover, keydown, etc.



React Events - Différence avec HTML

HTML classique	React
Syntaxe en **minuscule** → `onclick`	Syntaxe en **camelCase** → `onClick`
Le handler est écrit **entre guillemets** → `"action()"`	Le handler est écrit **entre accolades** → `{action}`
Peut appeler du code inline	Usage recommandé : appeler une **fonction JavaScript**

React Events - Exemple simple

```
function Button() {
  function shoot() {
    alert("Bouton cliqué !");
  }

  return (
    <button onClick={shoot}>Clique moi</button>
  );
}
```

✅ Ici, shoot est passé comme référence → pas de parenthèses !

React Events - Passage de paramètre

```
function Button() {
 function shoot(name) {
  alert(name + " a cliqué !");
 }

 return (
  <button onClick={() => shoot("Kamel")}>
   Clique
  </button>
 );
}
```

✅ Ici, On utilise une **fonction fléchée** pour transmettre des arguments !

React Events - Gestion des Inputs

```
function InputDemo() {
  function handleChange(event) {
    console.log("Valeur :", event.target.value);
  }

  return (
    <input type="text" onChange={handleChange} />
  );
}
```

✅ → **event.target.value** permet d’accéder à la valeur saisie.

Pratique :

Dans FirstApp

Atelier 02.03 Events

Rendu conditionnel : If Statement

- Permet d’exécuter du code ou d’afficher un composant uniquement si une condition est vraie.
- Utilisé dans la fonction de rendu, mais pas directement dans le JSX.

```
import React from "react";

const IfExample = () => {
  const isLoggedIn = true;

  let message;
  if (isLoggedIn) {
    message = <h1>Bienvenue sur le site !</h1>;
  } else {
    message = <h1>Veuillez vous connecter</h1>;
  }

  return <div>{message}</div>;
};

export default IfExample;
```

Rendu conditionnel : && Opérateur logique

- Permet de rendre un élément JSX uniquement si une condition est vraie.
- Syntaxe compacte pour le rendu conditionnel dans le JSX.

```
import React from "react";

const AndExample = () => {
 const showMessage = true;

 return (
  <div>
   <h1>Liste principale</h1>
   {showMessage && <p>Vous voyez ce message car showMessage est vrai !</p>}
  </div>
 );
};

export default AndExample;
```

Rendu conditionnel : Opérateur ternaire

- Permet d'afficher l'une des deux valeurs en fonction d'une condition.
- Syntaxe :

condition ? **valeur_si_vrai** : **valeur_si_faux**

```
import React from "react";

const TernaryExample = () => {
 const isLoggedIn = false;

 return (
  <div>
   {isLoggedIn ? <h1>Bienvenue !</h1> :
<h1>Veuillez vous connecter</h1>}
  </div>
 );
};

export default TernaryExample;
```

Rendu conditionnel : Listes et itération avec map()

- Permet de créer une liste d’éléments JSX à partir d’un tableau.
- Chaque élément doit avoir une clé unique pour que React puisse gérer efficacement les mises à jour.

```
import React from "react";

const fruits = ["Pomme", "Banane", "Orange"];

const ListExample = () => {
 return (
   <ul>
    {fruits.map((fruit, index) => (
      <li key={index}>{fruit}</li>
    ))}
   </ul>
 );
};

export default ListExample;
```

Le Spread Operator (opérateur de décomposition)...

- Il permet de copier les éléments d'un tableau dans un nouveau tableau.

Exemple simple :

- const nums = [1, 2, 3];
- const copy = [...nums]; // copy = [1, 2, 3]

➡️ On duplique le tableau sans le modifier.

Le Spread Operator (opérateur de décomposition)...

- Dans la ToDo List, tasks est un tableau contenant la liste des tâches.
- setTasks([**...tasks**, { id: Date.now(), text: input }]);

Décomposition :

1. `...tasks` : Copie toutes les tâches existantes dans un **nouveau tableau**
2. `{ id: Date.now(), text: input }` : Représente la **nouvelle tâche** à ajouter
3. `[...]` : Construit un nouveau tableau avec l'ancien + la nouvelle tâche

Le Spread Operator (opérateur de décomposition)...

```
tasks = [
  { id: 1, text: "Dormir 😴" },
  { id: 2, text: "Coder 💻" }
];
input = "Manger 🍽️";
```

```
setTasks([...tasks, { id: 3, text: input }]);
```

```
[
  { id: 1, text: "Dormir 😴" },
  { id: 2, text: "Coder 💻" },
  { id: 3, text: "Manger 🍽️" } // ajouté
]
```

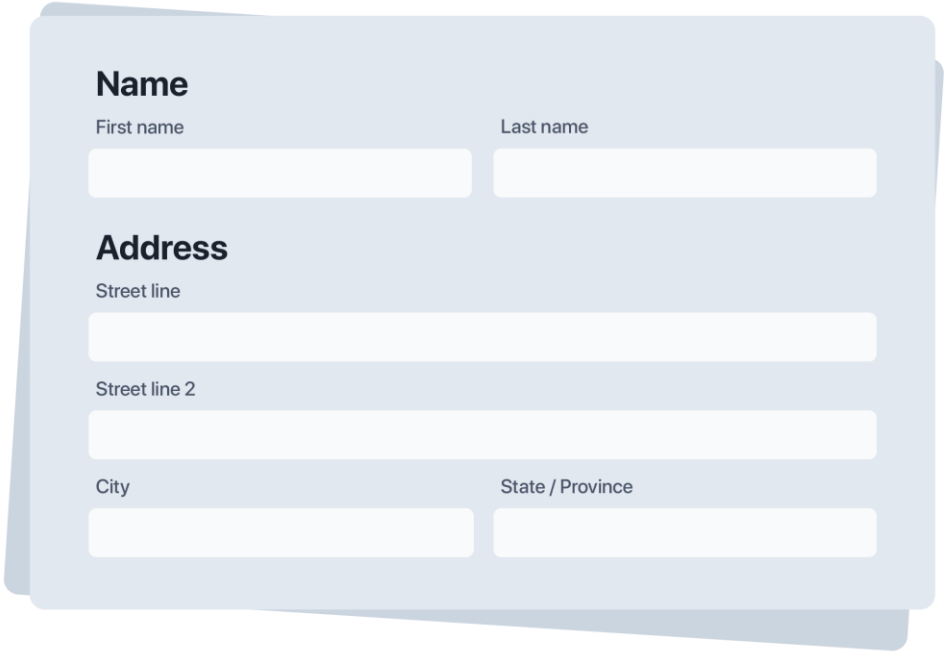
Pratique :

Dans FirstApp

Atelier 02.04 Conditions

Formulaire en React

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



Soumission par défaut des formulaires

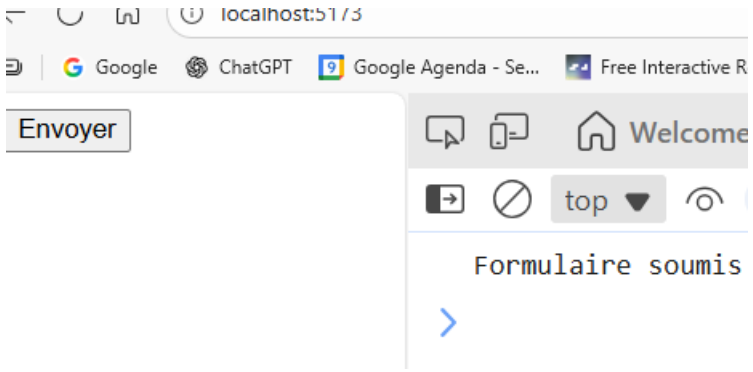
En HTML classique :

- Lorsqu'un formulaire est soumis → **la page se recharge** (comportement par défaut du navigateur).
- Dans **React**, ce comportement n'est **pas souhaité**.

➡️ Nous voulons **empêcher le rechargement** et laisser **React gérer les données**

Empêcher l'actualisation

```
export default function FormExample() {
 function handleSubmit(event) {
  event.preventDefault(); // ❗ Empêche le rechargement
  console.log("Formulaire soumis !");
  //console.log(event.target);
 }


 return (
  <form onSubmit={handleSubmit}>
   <button type="submit">Envoyer</button>
  </form>
 );
}
```



Concept : Formulaire contrôlé

HTML Classique	**React**
Le DOM contrôle les valeurs des inputs	Le **state** contrôle les valeurs
Valeurs modifiées directement dans le navigateur	Valeurs **stockées dans le state**
Pas de logique intégrée	Logique claire et centralisée

➡️ Dans React, chaque champ est connecté à un state → c'est la source unique de vérité.

Gestion du formulaire avec useState

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

function NameForm() {
 const [name, setName] = useState("");
 function handleSubmit(e) {
  e.preventDefault();
  alert("Bonjour " + name);
 }
 return (
  <form onSubmit={handleSubmit}>
   <input
    type="text"
    value={name}
    onChange={(e) => setName(e.target.value)}
   />
   <button>Valider</button>
  </form>
 );
}
export default NameForm;
```

➡️ value={name} → input contrôlé
➡️ onChange met à jour le state

Champs Multiples dans un Formulaire

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

function MultiForm() {
 const [form, setForm] = useState({});

 function handleChange(e) {
  setForm({ ...form, [e.target.name]: e.target.value });
 }

 return (
  <form>
   <input name="firstName" onChange={handleChange} />
   <input name="email" onChange={handleChange} />
   <p>{JSON.stringify(form)}</p>
  </form>
 );
}
export default MultiForm;
```

Abbassi | abbassi.kamel@gmail.com

{"firstName":"Abbassi","email":"abbassi.kamel@gmail.com"}

➡️ [e.target.name] met à jour la bonne propriété
➡️ ...form préserve les valeurs existantes

Zone de texte (textarea)

⚠️ Différence entre HTML et React :

HTML	**React**
Le texte se place **entre les balises**	La valeur se met dans **value**

```
function MessageBox() {
  const [message, setMessage] = useState("");

  return (
    <textarea
      value={message}
      onChange={(e) => setMessage(e.target.value)}
    />
  );
}
```

Liste déroulante (select)

```
function CarSelect() {
  const [car, setCar] = useState("Volvo");

  return (
    <select value={car} onChange={(e) => setCar(e.target.value)}>
      <option value="Volvo">Volvo</option>
      <option value="BMW">BMW</option>
      <option value="Audi">Audi</option>
    </select>
  );
}
```

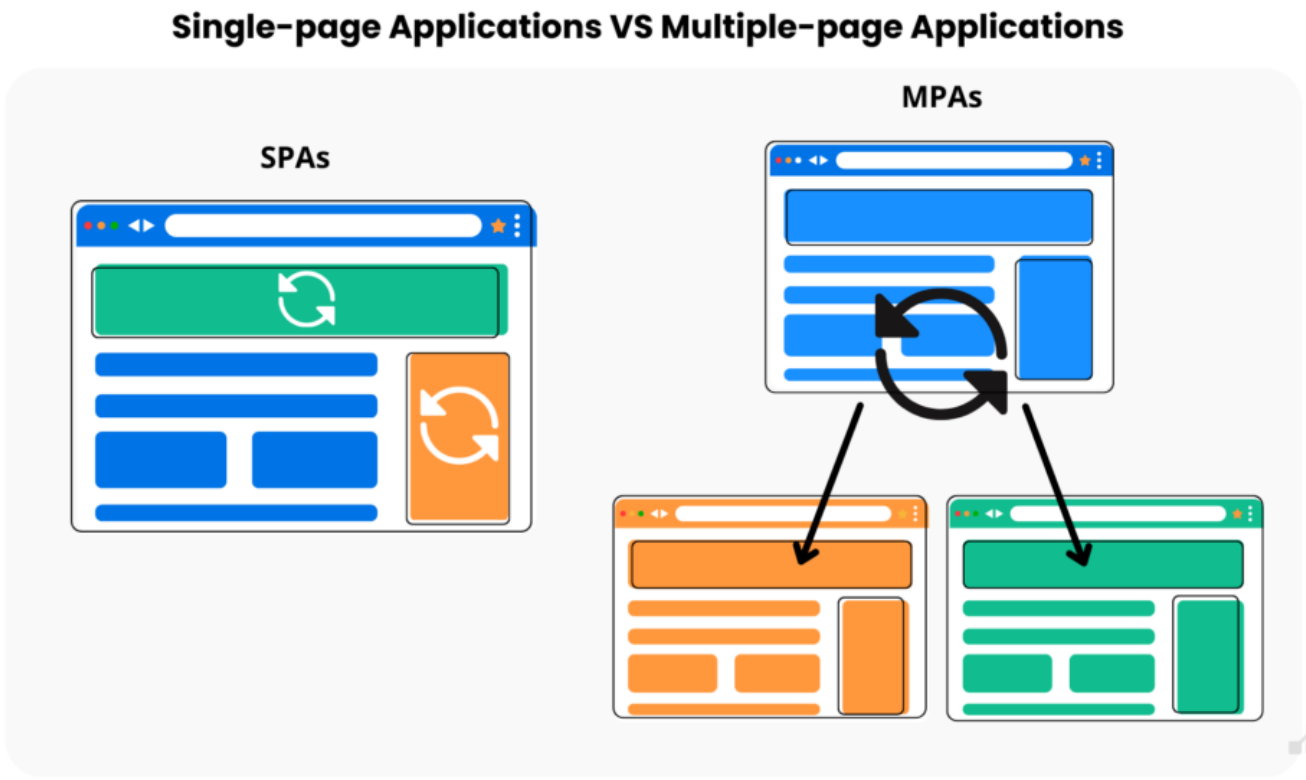
➡️ `value={car}` indique l'élément sélectionné
➡️ `onChange` met à jour le state

Pratique :

Dans FirstApp

Atelier 02.05 Formulaires

Routage et navigation SPA (Single Page Application)



Pourquoi utiliser React Router ?

- Create React App / Vite n'inclut pas de routage par défaut.
- React Router permet de créer des pages et une navigation sans rechargement complet.
- Permet de gérer :
  - L’URL du navigateur
  - Les transitions de pages
  - Les routes dynamiques
  - Les paramètres d’URL

Installation

- Ouvrir le terminal à la racine du projet et exécuter :

npm install react-router-dom

Ou bien

npm install react-router-dom@latest

- react-router-dom est la bibliothèque officielle pour le routage web en React.

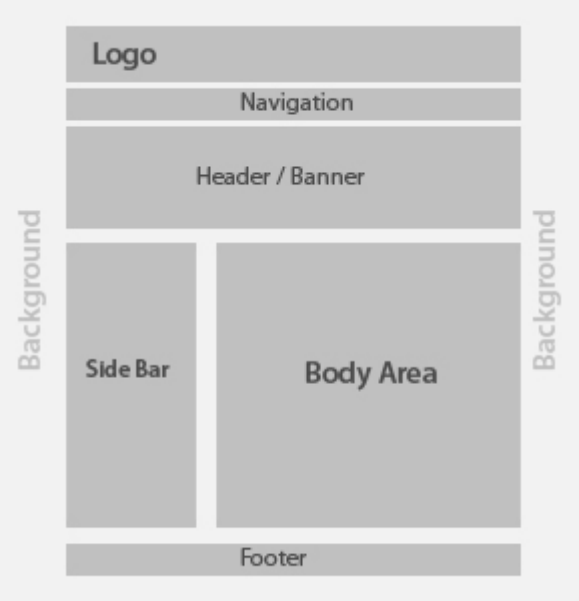
Structure de projet recommandée

src/
- App.jsx
- main.jsx
- **pages**/
  - Home.jsx
  - Blogs.jsx
  - Contact.jsx
  - NoPage.jsx (page 404)
  - Layout.jsx (gabarit + menu)

Layout.jsx (Page Layout / Template)

Rôle :

- Contient la **barre de navigation** ou le menu
- Définit la structure commune
- Affiche la page enfant grâce à **`<Outlet />`**




```
import { Outlet, Link } from "react-router-dom";

export default function Layout() {
 return (
  <>
   <nav>
    <Link to="/">Accueil</Link> |
    <Link to="/blogs">Blogs</Link> |
    <Link to="/contact">Contact</Link>
   </nav>
   <Outlet />
  </>
 );
}
```

Configuration du routage (main.jsx)

```
import React from "react";
import ReactDOM from "react-dom/client";
import { RouterProvider, createBrowserRouter }
from "react-router-dom";

import Layout from "./pages/Layout";
import Home from "./pages/Home";
import Blogs from "./pages/Blogs";
import Contact from "./pages/Contact";
import NoPage from "./pages/NoPage";
```

```
const router = createBrowserRouter([
 {
  path: "/",
  element: <Layout />,
  children: [
   { index: true, element: <Home /> },
   { path: "blogs", element: <Blogs /> },
   { path: "contact", element: <Contact /> },
   { path: "*", element: <NoPage /> },
  ],
 },
]);

ReactDOM.createRoot(document.getElementById("root")).r
ender(
 <RouterProvider router={router} />
);
```

Outlet & Link

- <Outlet />
  - Sert à afficher la page enfant active.
  - Permet d’éviter de répéter la structure (header, footer…).
- <Link to="">Navigation interne sans recharger la page
  - Remplace <a href="">.
- Exemple :
  - <Link to="/contact">Contact</Link>

Navigation programmée

Utiliser useNavigate() pour rediriger depuis un événement (ex: après l'envoi d'un formulaire) :

```
import { useNavigate } from "react-router-dom";

function Contact() {
  const navigate = useNavigate();

  const handleSubmit = () => {
    navigate("/"); // redirige vers Home
  };

  return <button onClick={handleSubmit}>Envoyer</button>;
}
```

Routage dynamique (:id)

```
<Route path="blogs/:id" element={<BlogDetail />} />
```

Récupération du paramètre :

```
import { useParams } from "react-router-dom";
const BlogDetail = () => {
  const { id } = useParams();
  return <h1>Article n° {id}</h1>;
};
```

Pratique :

Dans FirstApp

Atelier 02.06 Routage

React : Styliser vos composants

inline styles, CSS classique et modules CSS.

Pourquoi styliser React ?

- React ne propose pas de système de style natif, mais fonctionne avec CSS et objets JS.
- Styliser permet :
  - Améliorer l’apparence des composants.
  - Créer des interfaces interactives et agréables.
  - Faciliter la réutilisation des composants avec styles spécifiques.

Style en ligne (Inline Style)

- On utilise l’attribut **style** dans le JSX.
- La valeur doit être un objet JavaScript.
- Les propriétés CSS sont en **camelCase** :
  - (ex : background**C**olor au lieu de background-color).

```
function Home() {
  return (
    <div style={{ backgroundColor: 'lightblue', padding: '20px' }}>
      <h1 style={{ color: 'white' }}>Bienvenue sur Home</h1>
      <p style={{ fontSize: '18px' }}>Ceci est un texte stylisé en inline style.</p>
    </div>
  );
}
```

Objet JavaScript pour styles

- On peut créer un objet séparé pour stocker les styles.
- Réutilisable dans plusieurs composants.

```
const boxStyle = {
  backgroundColor: 'lightgreen',
  padding: '15px',
  borderRadius: '10px'
};

function Box() {
  return (
    <div style={boxStyle}>
      <p>Box avec style via objet JS.</p>
    </div>
  );
}
```

Feuilles de style CSS

- Créer un fichier .css séparé, ex : App.css.
- Importer le fichier dans le composant ou dans App.js.

```
/* App.css*/
.container {
  background-color: lightyellow;
  padding: 20px;
}

.container h1 {
  color: darkblue;
}

.container p {
  font-size: 16px;
}
```

```
import './App.css';

function Home() {
  return (
    <div className="container">
      <h1>Bienvenue sur Home</h1>
      <p>Texte avec style via CSS classique.</p>
    </div>
  );
}
```

Modules CSS

- Permettent de scoper le CSS au composant pour éviter les conflits.
- Le fichier doit avoir l'extension .module.css, ex : my-style.module.css.

```css
/* my-style.module.css*/
.title {
  color: purple;
  font-size: 24px;
}

.text {
  font-size: 18px;
  color: gray;
}
```

```jsx
import styles from './my-style.module.css';

function Home() {
  return (
    <div>
      <h1 className={styles.title}>Titre en Module CSS</h1>
      <p className={styles.text}>Texte stylisé avec module CSS</p>
    </div>
  );
}
```

Pratique :

Dans FirstApp

Atelier 02.07 Style

Hooks

- Les Hooks ont été introduits dans React 16.8 et sont pleinement utilisés en React 19.
- Ils permettent d'utiliser :
  - l’état (state)
  - le cycle de vie
  - le contexte directement dans des composants fonctionnels.
- Les composants classiques (class components) ne sont plus nécessaires.
- Objectif : rendre les composants plus simples et réutilisables.

Qu’est-ce qu’un Hook ?

- Un Hook est une fonction spéciale de React.
- Il permet de “s’accrocher” aux fonctionnalités internes de React.
- Exemple (ne pas se soucier encore des détails) :

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

function App() {
  const [count, setCount] = useState(0);
  return <button onClick={() => setCount(count + 1)}>Compteur : {count}</button>;
}
```

Règles des Hooks

- Les Hooks doivent être utilisés uniquement dans un composant fonctionnel.
- Les Hooks doivent être appelés au niveau supérieur du composant.
  - Pas dans des if, for, ou fonctions internes.
- Les Hooks ne peuvent pas être appelés en dehors d’un composant ou d’un Hook personnalisé

Le Hook useState

- useState permet de gérer l’état dans un composant.
- Syntaxe (React 19) :

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

const [color, setColor] = useState(""); // état initial = chaîne vide
```

- color = valeur actuelle de l’état
- setColor = fonction qui met à jour l’état

Exemple Simple useState

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

function ColorPicker() {
  const [color, setColor] = useState("Rouge");

  return (
    <div>
      <h2>Couleur : {color}</h2>
      <button onClick={() => setColor("Bleu")}>Changer</button>
    </div>
  );
}
```

- Le clic modifie l’état, et React ré-affiche le composant.

useState avec Objets

- Quand on change un objet, l'objet entier est remplacé.
- On utilise l'opérateur **spread** pour mettre à jour seulement une partie.

```
const [car, setCar] = useState({ brand: "Toyota", color: "Rouge" });

function changeColor() {
  setCar(prev => ({ ...prev, color: "Bleu" }));
}
```

Le Hook useEffect

- Permet de gérer les effets secondaires :
  - Appels API
  - Timers
  - Accès DOM
- Syntaxe :

useEffect(() => {

// effet ici

*}, [**dépendances**]);*

- Le Hook se déclenche lorsque les **dépendances** changent.

Exemple useEffect

Rendu initial

```
useEffect(() => {
  console.log("Composant affiché");
}, []); // tableau vide ⇒ exécution unique
```

Dépendant d’un état

```
useEffect(() => {
  console.log("Compteur modifié :", count);
}, [count]); // L’effet se relance seulement quand count change.
```

Nettoyage useEffect

- Certains effets doivent être nettoyés :timers, listeners, API socket…

```
useEffect(() => {
  const timer = setInterval(() => setCount(c => c + 1), 1000);
  return () => clearInterval(timer); // nettoyage
}, []);
```

useContext (État global)

- Évite le prop drilling (transmission manuelle des props).
- Création du contexte :

  const UserContext = createContext();
- Fournisseur :

  <UserContext.Provider value={user}>

  <App />

  </UserContext.Provider>
- Utilisation :

  const user = useContext(UserContext);

useRef

- Conserve une valeur sans re-rendu.
- Sert aussi à accéder à un élément DOM.

```
const inputRef = useRef(null);
function focusField() {
inputRef.current.focus();
}

<input ref={inputRef} />
```

useReducer (État complexe)

- Alternative à useState lorsque la logique est plus structurée.

```
function reducer(state, action) {
    switch(action.type) {
        case "increment": return { count: state.count + 1 };
}
}
const [state, dispatch] = useReducer(reducer, { count: 0 });
```

useCallback (Performance)

- Empêche une fonction d’être recréée à chaque rendu.

```
const increment = useCallback(() => setCount(c => c + 1), []);
```

Pratique :

Dans FirstApp

Atelier 02.08 Hook

Introduction à l’atelier

- Objectif :
  - créer une application TodoList en React.
- Compétences visées :
  - Gérer l’état avec useState.
  - Mettre à jour les valeurs avec onChange.
  - Filtrer une liste avec **filter**().
  - Comprendre le rendu conditionnel.

useState : rappel

const [value, setValue] = useState(initialValue);

- initialValue peut être :
  - Null
  - une valeur scalaire : nombre, string, booléenun
  - objet : { id: 1, text: "Hello" }
  - un tableau : [1, 2, 3]
  - un JSON (en réalité un objet JavaScript)
- React garde cette valeur pour le composant.T
- oute mise à jour déclenche un nouveau rendu.

onChange : comment ça marche ?

<input

value={newTask}

onChange={e => setNewTask(e.target.value)}

/>

- **Pourquoi cette syntaxe ?**
- onChange attend **une fonction**.
- e => setNewTask(e.target.value) est une **fonction fléchée**.
- Si on écrit : onChange={setNewTask(e.target.value)}

➡️ Ici setNewTask(e.target.value) s'exécute immédiatement avant même que l'utilisateur tape.
➡️ On passerait le **résultat** à onChange, pas une fonction.
➡️ React recevrait une valeur **non fonctionnelle → erreur**.

filter() : fonctionnement général

- **Syntaxe :**

const result = array.**filter**(element => condition);

- **filter**() parcourt un tableau.
- Garde uniquement les éléments pour lesquels la condition retourne true.

Exemple simple :

- const nums = [1, 2, 3, 4];
- const evens = nums.filter(n => n % 2 === 0); // [2, 4]

Pratique :

Atelier 03 TodoList

Pratique :

Atelier 04 Weather

Pratique :

Atelier 05 MiniEcommerce

Introduction aux API

Une **API** (Application Programming Interface) permet à une application de communiquer avec un serveur pour :

- Lire des données (GET)
- Ajouter (POST)
- Modifier (PUT / PATCH)
- Supprimer (DELETE)

Exemples d'APIs publiques gratuites :

- JSONPlaceholder
- ReqRes
- DummyJSON

Dans l'atelier, nous utiliserons : 👉 **DummyJSON** (API 100% gratuite)

Utiliser fetch en React

- **Exemple simple : récupérer des posts**

```
useEffect(() => {
  fetch("https://jsonplaceholder.typicode.com/posts")
      .then(res => res.json())
      .then(data => setPosts(data));
}, []);
```

Utiliser fetch en React

- **Version recommandée (async/await)**

```
useEffect(() => {
  const loadPosts = async () => {
  const res = await fetch("https://jsonplaceholder.typicode.com/posts");
  const data = await res.json();
     setPosts(data);
    };
    loadPosts();
}, []);
```

Envoyer des données : POST

```
const addPost = async () => {
const newPost = { title: "Test", body: "Contenu" };
const res = await fetch("https://jsonplaceholder.typicode.com/posts", {
method: "POST",
headers: { "Content-Type": "application/json" },
body: JSON.stringify(newPost)
});
const data = await res.json();
console.log("Nouveau post :", data);
};
```

Modifier : PUT / PATCH

```
const updatePost = async (id) => {
const res = await
  fetch(` https://jsonplaceholder.typicode.com/posts/${id}` , {
method: "PUT",
headers: { "Content-Type": "application/json" },
body: JSON.stringify({ title: "Modifié" })
});
const data = await res.json();
console.log("Maj :", data);
};
```

Supprimer : DELETE

```
const deletePost = async (id) => {
await fetch(` https://jsonplaceholder.typicode.com/posts/${id}` , {
method: "DELETE"
});
console.log("Supprimé");
};
```

LocalStorage en React

- **Permet de garder des données même après le rechargement.**

Enregistrer	**localStorage.setItem("token", "123ABC");**
Lire	**const token = localStorage.getItem("token");**
Supprimer	**localStorage.removeItem("token");**

Exemple pratique : sauvegarder un thème

```
const [theme, setTheme] = useState(localStorage.getItem("theme") || "light");
useEffect(() => {
localStorage.setItem("theme", theme);
}, [theme]);
```

Pratique :

Atelier 06 APICrud

Pratique :

Atelier 07 Hébergement et déploiement