Cours – Les bases de React

Table des matières

---

0. Introduction

0.1. Qu'est-ce qu'un bundler ?

Un **bundler** est un outil qui prend tous les fichiers de ton projet front (JS, JSX, CSS, images, modules NPM...) et les transforme en quelques fichiers optimisés que le navigateur peut comprendre.

Concrètement, il :

- Lit tes `import` / `export` pour construire un graphe de dépendances
- Transforme les syntaxes modernes (JSX, TypeScript, etc.) en JS compatible navigateur
- Fusionne / découpe le code en fichiers optimisés
- Supprime le code non utilisé (tree-shaking)
- Minifie le JS / CSS pour réduire la taille
- Fournit un serveur de dev avec rechargement automatique (HMR)

Exemples de bundlers :

- **Vite** : moderne, très rapide, idéal pour React
- **Webpack** : historique, ultra configurable
- **Parcel** : zéro config

> **À retenir** : Un bundler est le traducteur/optimiseur entre ton code moderne et ce que le navigateur comprend.

0.2. Création d'un projet avec Vite

```
# Création du projet
npm create vite@latest mon-projet-react

# Choix :
# Framework : React
# Variant   : JavaScript

cd mon-projet-react

# Installation des dépendances
npm install

# Lancer le serveur de dev
npm run dev
```

0.3. Architecture d'un projet React + Vite

Une arborescence typique :

```
mon-projet-react/
  index.html
  package.json
  vite.config.js
  public/
  src/
    main.jsx
    App.jsx
    index.css
    assets/
      ...
```

`index.html`

- Seul fichier HTML
- Contient une `<div id="root"></div>` dans laquelle React va injecter l'app

`src/main.jsx`

Point d'entrée JS. Monte `<App />` dans la div `#root`.

```
// src/main.jsx
import React from "react";
import ReactDOM from "react-dom/client";
import App from "./App.jsx";
import "./index.css";

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

`src/App.jsx`

Composant racine de l'application.

```
// src/App.jsx
function App() {
  return <h1>Bonjour React □</h1>;
}

export default App;
```

`public/`

Fichiers statiques servis tels quels (favicon, images...)

`src/assets/`

Ressources front importées dans le code : images, JSON, etc.

---

1. JSX

Concepts

Le **JSX** est une syntaxe qui permet d'écrire du HTML dans le JavaScript, mais **ce n'est pas du HTML** :

- Le JSX est transformé en appels `React.createElement`
- Un composant doit retourner un seul élément parent
- On utilise des attributs en camelCase : `className`, `onClick`, `htmlFor`
- On insère des valeurs JS avec `{ ... }` (variables, expressions, opérations...)

JSX permet de raisonner en composants : HTML + logique dans la même fonction.

Exemple de code

```
// src/lessons/1.Le-JSX/Lesson1.jsx
function Lesson1JSX() {
  const name = "Victor";
  const age = 26;

  // Un seul parent, des {} pour insérer du JS
  return (
    <div>
      <h1>Bonjour {name} !</h1>
      <p>Tu as {age} ans.</p>
      <p>
        Dans 5 ans tu auras {age + 5} ans.
      </p>
    </div>
  );
}

export default Lesson1JSX;
```

---

2. Premier composant

Concepts

Un **composant React** est :

- Une fonction JavaScript dont le nom commence par une **majuscule**
- Qui retourne du **JSX**
- Et que l'on utilise comme une balise HTML custom (`<Header />`)

On l'exporte et on l'importe comme un module JS classique.

Exemple de code

```
// src/lessons/2.Premier-composant/Lesson2.jsx
function Header() {
  return <h1>Mon premier composant React</h1>;
}

function Footer() {
  return <footer>© 2025 Victor Besson</footer>;
}

function Lesson2PremierComposant() {
  return (
    <>
      <Header />
      <p>Voici le contenu principal de ma page.</p>
      <Footer />
    </>
  );
}

export default Lesson2PremierComposant;
```

---

3. Utiliser des événements

Concepts

React gère un système d'événements très proche du DOM :

- On utilise des props comme `onClick`, `onChange`, `onSubmit`, etc.
- On passe une **fonction** (et non le résultat de la fonction)
- React appelle cette fonction quand l'événement survient

On obtient un objet `event` très similaire à l'événement DOM natif.

Exemple de code

```
// src/lessons/3.Utiliser-des-événements/Lesson3.jsx
import { useState } from "react";

function Lesson3Events() {
  const [inputValue, setInputValue] = useState("");

  function handleButtonClick() {
    alert("Bouton cliqué avec valeur : " + inputValue);
  }

  function handleInputChange(event) {
    setInputValue(event.target.value);
  }

  return (
    <div>
      <input
        type="text"
        placeholder="Tape quelque chose"
        onChange={handleInputChange}
      />
      <button onClick={handleButtonClick}>Afficher la valeur</button>
    </div>
  );
}

export default Lesson3Events;
```

---

4. Le state (`useState`)

Concepts

Le **state** représente les données internes d'un composant :

- On le crée avec `useState(valeurInitiale)`
- `useState` renvoie `[valeur, setValeur]`
- Appeler `setValeur` demande à React de re-render le composant
- Pour mettre à jour en fonction de l'ancienne valeur, on utilise la **forme fonction** :

```
setCount((prev) => prev + 1);
```

Exemple de code

```jsx
// src/lessons/4.Le-state/Lesson4.jsx
import { useState } from "react";

function Lesson4State() {
  const [count, setCount] = useState(0);

  function handleIncrement() {
    setCount((prev) => prev + 1);
  }

  function handleDecrement() {
    setCount((prev) => (prev > 0 ? prev - 1 : 0));
  }

  return (
    <div>
      <p>Compteur : {count}</p>
      <button onClick={handleDecrement}>-1</button>
      <button onClick={handleIncrement}>+1</button>
    </div>
  );
}

export default Lesson4State;
```

---

5. Les props

Concepts

Les **props** sont les "paramètres" d'un composant :

- Le parent passe des props à l'enfant via des attributs JSX
- L'enfant les reçoit en argument de sa fonction (`props` ou déstructuration)
- Les props sont **en lecture seule** dans l'enfant

Différence :

- **State** : données internes au composant
- **Props** : données venant de l'extérieur (parent)

Exemple de code

```jsx
// src/lessons/5.Les-props/UserCard.jsx
function UserCard({ name, age, job }) {
  return (
    <article className="user-card">
      <h2>{name}</h2>
      <p>{age} ans</p>
      <p>{job}</p>
    </article>
  );
}

export default UserCard;
```

```jsx
// src/lessons/5.Les-props/Lesson5.jsx
import UserCard from "./UserCard";

function Lesson5Props() {
  return (
    <div>
      <UserCard name="Victor" age={26} job="Développeur web" />
      <UserCard name="Alice" age={30} job="Designer UX" />
    </div>
  );
}

export default Lesson5Props;
```

---

6. Utiliser du CSS

Concepts

On applique les mêmes concepts CSS, mais via `className` :

- **CSS global** : défini dans `index.css` ou un fichier importé
- Classes appliquées via `className="ma-classe"`
- Eventuellement : **CSS Modules** (`styles.maClasse`) pour éviter les collisions

Exemple de code

```css
/* src/lessons/6.Utiliser-du-CSS/lesson6.css */
.user-card {
  border: 1px solid #ddd;
  border-radius: 8px;
  padding: 1rem;
  margin-bottom: 0.5rem;
}

.user-card--highlight {
  background-color: #f3f4ff;
}
```

```jsx
// src/lessons/6.Utiliser-du-CSS/Lesson6.jsx
import "./lesson6.css";

function Lesson6CSS() {
  return (
    <div>
      <div className="user-card">
        <h2>Utilisateur classique</h2>
      </div>
      <div className="user-card user-card--highlight">
        <h2>Utilisateur mis en avant</h2>
      </div>
    </div>
  );
}

export default Lesson6CSS;
```

---

7. Créer une liste

Concepts

Pour afficher des listes :

- On part d'un **tableau JavaScript**
- On utilise `map()` pour le transformer en tableau de JSX
- On ajoute une prop `key` unique et stable sur chaque élément

`key` permet à React de savoir quel élément correspond à quel item du tableau lors des re-renders.

Exemple de code

```
// src/lessons/7.Créer-une-liste/Lesson7.jsx
const users = [
  { id: 1, name: "Victor", role: "Admin" },
  { id: 2, name: "Alice", role: "User" },
  { id: 3, name: "Bob", role: "User" },
];

function Lesson7List() {
  return (
    <ul>
      {users.map((user) => (
        <li key={user.id}>
          {user.name} – {user.role}
        </li>
      ))}
    </ul>
  );
}

export default Lesson7List;
```

---

8. Utiliser des images

Concepts

Deux cas :

1. **Images locales** dans `src/assets` → importées comme modules
2. **Images externes** → on donne l'URL directement

Toujours renseigner `alt` pour l'accessibilité.

Exemple de code

```
// src/lessons/8.Utiliser-des-images/Lesson8.jsx
import localAvatar from "../../assets/lessons/avatar.png"; // adapter le chemin

function Lesson8Images() {
  return (
    <div>
      <h2>Image locale</h2>
      <img src={localAvatar} alt="Avatar local" width={120} />

      <h2>Image externe</h2>
      <img
        src="https://picsum.photos/200"
        alt="Image aléatoire"
        width={120}
      />
    </div>
  );
}

export default Lesson8Images;
```

9. Rendu conditionnel avec `if`

Concepts

On n'affiche pas toujours tout : on peut conditionner le rendu :

- Pré-calculer une variable JSX avec `if`
- Retourner `null` pour ne rien afficher

Exemple de code

```
// src/lessons/9.Rendu-conditionnel-if/Lesson9.jsx
function StatusMessage({ isLoggedIn }) {
  if (!isLoggedIn) {
    return <p>Veuillez vous connecter pour continuer.</p>;
  }

  return <p>Bienvenue, vous êtes connecté </p>;
}

function Lesson9If() {
  return (
    <div>
      <h2>Utilisateur non connecté</h2>
      <StatusMessage isLoggedIn={false} />

      <h2>Utilisateur connecté</h2>
      <StatusMessage isLoggedIn={true} />
    </div>
  );
}

export default Lesson9If;
```

10. Opérateur ternaire

Concepts

Le **ternaire** permet une écriture compacte d'un if/else dans le JSX :

```
condition ? <JSXsiVrai /> : <JSXsiFaux />
```

C'est très lisible pour des conditions simples.

Exemple de code

```jsx
// src/lessons/10.Opérateur-ternaire/Lesson10.jsx
function ProductStatus({ inStock }) {
  return (
    <p>
      Statut :{" "}
      {inStock ? (
        <span style={{ color: "green" }}>En stock</span>
      ) : (
        <span style={{ color: "red" }}>Rupture de stock</span>
      )}
    </p>
  );
}

function Lesson10Ternary() {
  return (
    <div>
      <ProductStatus inStock={true} />
      <ProductStatus inStock={false} />
    </div>
  );
}

export default Lesson10Ternary;
```

---

11. Short-circuit (`&&`)

Concepts

Le **short-circuit** utilise le comportement de `&&` en JS :

- `condition && <JSX />` → si `condition` est vraie, React affiche le JSX, sinon rien

Très pratique pour afficher seulement si...

Exemple de code

```jsx
// src/lessons/11.Short-circuit/Lesson11.jsx
function Notification({ hasNotification }) {
  return (
    <div>
      <h2>Notifications</h2>
      {hasNotification && (
        <p>🔔 Vous avez une nouvelle notification importante.</p>
      )}
      {!hasNotification && <p>Aucune notification pour le moment.</p>}
    </div>
  );
}

function Lesson11ShortCircuit() {
  return (
    <>
      <Notification hasNotification={true} />
      <Notification hasNotification={false} />
    </>
  );
}

export default Lesson11ShortCircuit;
```

---

12. Toggle une classe ou du CSS

Concepts

On utilise le state pour changer l'apparence :

- Un booléen dans le state (`isDark` , `isActive` , ...)
- `className` dynamique selon cette valeur

Exemple de code

```
/* src/lessons/12.Toggle-une-classe-ou-du-CSS/lesson12.css */
.box {
  padding: 1rem;
  border-radius: 8px;
  border: 1px solid #ddd;
}

.box--dark {
  background-color: #111827;
  color: #f9fafb;
}
```

```
// src/lessons/12.Toggle-une-classe-ou-du-CSS/Lesson12.jsx
import { useState } from "react";
import "./lesson12.css";

function Lesson12ToggleCSS() {
  const [isDark, setIsDark] = useState(false);

  function handleToggle() {
    setIsDark((prev) => !prev);
  }

  return (
    <div>
      <button onClick={handleToggle}>
        Passer en mode {isDark ? "clair" : "sombre"}
      </button>

      <div className={`box ${isDark ? "box--dark" : ""}`}>
        <p>Je change de style en fonction du state.</p>
      </div>
    </div>
  );
}

export default Lesson12ToggleCSS;
```

---

13. `useRef`

Concepts

`useRef` sert à :

1. **Référencer un élément DOM** (input, div, etc.)
2. **Stocker une valeur mutable** qui ne provoque pas de re-render quand elle change (id de timer, etc.)

Différence :

- **State** → change → re-render
- **Ref** → change → pas de re-render

Exemple 1 – Donner le focus à un input

```jsx
// src/lessons/13.useRef/Lesson13Focus.jsx
import { useRef } from "react";

function Lesson13Focus() {
  const inputRef = useRef(null);

  function handleFocusClick() {
    inputRef.current?.focus();
  }

  return (
    <div>
      <input ref={inputRef} type="text" placeholder="Clique sur le bouton" />
      <button onClick={handleFocusClick}>Donner le focus</button>
    </div>
  );
}

export default Lesson13Focus;
```

Exemple 2 – Timer stocké dans une ref

```jsx
// src/lessons/13.useRef/Lesson13Timer.jsx
import { useRef, useState } from "react";

function Lesson13Timer() {
  const [tick, setTick] = useState(0);
  const intervalRef = useRef(null);

  function startTimer() {
    if (intervalRef.current) return; // déjà lancé
    intervalRef.current = setInterval(() => {
      setTick((prev) => prev + 1);
    }, 1000);
  }

  function stopTimer() {
    clearInterval(intervalRef.current);
    intervalRef.current = null;
  }

  return (
    <div>
      <p>Tick : {tick}</p>
      <button onClick={startTimer}>Démarrer</button>
      <button onClick={stopTimer}>Arrêter</button>
    </div>
  );
}

export default Lesson13Timer;
```

---

14. Controlled vs Uncontrolled Inputs

Concepts

Deux façons de gérer la valeur d'un champ :

1. **Input contrôlé** : React est la source de vérité
    - La valeur de l'input vient du state
    - Chaque frappe déclenche un `onChange` qui met à jour le state
2. **Input non contrôlé** : le DOM garde la valeur

2. **Input non contrôlé** : le DOM garde la valeur
    - On lit la valeur directement via `ref` au moment voulu

Exemple – Input contrôlé

```
// src/lessons/14.Controlled-uncontrolled-inputs/Controlled.jsx
import { useState } from "react";

function ControlledLogin() {
  const [email, setEmail] = useState("");
  const [password, setPassword] = useState("");

  function handleSubmit(e) {
    e.preventDefault();
    alert(`Email: ${email}\nMot de passe: ${password}`);
  }

  const isDisabled = email === "" || password.length < 4;

  return (
    <form onSubmit={handleSubmit}>
      <h2>Login (controlled)</h2>
      <input
        type="email"
        placeholder="Email"
        value={email}
        onChange={(e) => setEmail(e.target.value)}
      />
      <input
        type="password"
        placeholder="Mot de passe (min 4 caractères)"
        value={password}
        onChange={(e) => setPassword(e.target.value)}
      />
      <button type="submit" disabled={isDisabled}>
        Se connecter
      </button>
    </form>
  );
}

export default ControlledLogin;
```

Exemple – Input non contrôlé

```jsx
// src/lessons/14.Controlled-uncontrolled-inputs/Uncontrolled.jsx
import { useRef } from "react";

function UncontrolledLogin() {
  const emailRef = useRef(null);
  const passwordRef = useRef(null);

  function handleSubmit(e) {
    e.preventDefault();
    const email = emailRef.current.value;
    const password = passwordRef.current.value;
    alert(`Email: ${email}\nMot de passe: ${password}`);
  }

  return (
    <form onSubmit={handleSubmit}>
      <h2>Login (uncontrolled)</h2>
      <input ref={emailRef} type="email" placeholder="Email" />
      <input ref={passwordRef} type="password" placeholder="Mot de passe" />
      <button type="submit">Se connecter</button>
    </form>
  );
}

export default UncontrolledLogin;
```

Composant wrapper pour démo

```jsx
// src/lessons/14.Controlled-uncontrolled-inputs/Lesson14.jsx
import ControlledLogin from "./Controlled";
import UncontrolledLogin from "./Uncontrolled";

function Lesson14ControlledUncontrolled() {
  return (
    <div>
      <ControlledLogin />
      <hr />
      <UncontrolledLogin />
    </div>
  );
}

export default Lesson14ControlledUncontrolled;
```

Fin du cours

Tu as maintenant toutes les bases pour créer des applications React !

Pour aller plus loin, explore :

- **React Router** pour la navigation
- **useEffect** pour les effets de bord
- **Context API** ou **Redux** pour la gestion d'état globale
- **React Query** pour la gestion des données serveur