

Module 2 – React Avancé

Table des matières

- [1. Les fragments](#)
 - [2. La règle des hooks](#)
 - [3. Le hook useEffect](#)
 - [4. Cleanup function](#)
 - [5. Ajouter un événement global](#)
 - [6. Appeler une API](#)
 - [7. Utiliser un observateur \(IntersectionObserver\)](#)
 - [8. Utiliser setInterval](#)
 - [9. Sélectionner un tableau d'éléments](#)
 - [10. Comprendre props.children](#)
 - [11. Memo et useCallback](#)
 - [12. useMemo](#)
 - [13. useReducer](#)
 - [14. Hook personnalisé](#)
 - [15. CSS Modules](#)
 - [16. Utiliser TailwindCSS](#)
-

1. Les fragments

Objectif

Éviter les `<div>` inutiles dans le DOM en regroupant plusieurs éléments JSX sans créer de balise supplémentaire.

Pourquoi les fragments existent ?

En React, un composant doit retourner un **seul élément parent**.

Souvent, on ne veut **PAS** ajouter un conteneur artificiel dans le DOM (ex : pour les tableaux, le CSS, les layouts, etc.).

Fragments permettent :

- D'éviter les "div wrappers"
- D'améliorer la lisibilité du DOM
- De réduire la pollution visuelle

Équivalent court et long :

```
<> ... </>
```

```
<React.Fragment> ... </React.Fragment>
```

Exemple clair

```
function Lesson1Fragments() {
  return (
    <>
      <h1>Titre</h1>
      <p>Paragraphe 1</p>
      <p>Paragraphe 2</p>
    </>
  );
}
```

Dans le DOM, **aucune balise supplémentaire** n'apparaît.

Mini exercice

Créer un composant `Article` contenant :

- Un titre
- Deux paragraphes

Le tout **sans div inutile**.

2. La règle des hooks

Objectif

Comprendre pourquoi les hooks doivent être appelés dans le même ordre à chaque rendu.

Pourquoi cette règle ?

React associe les hooks par **position dans le code**, pas par nom.

Exemple interne (simplifié) :

```
Hook #1 → useState compteur
Hook #2 → useEffect effet
Hook #3 → useState valeur input
```

Si tu appelles un hook dans une condition :

```
if (show) {
  useState(0);
}
```

L'ordre change.

Résultat : React associe les hooks aux mauvaises valeurs → bug total.

Les 3 règles officielles

1. Toujours au niveau supérieur

Pas dans :

- Conditions
- Boucles
- Fonctions internes

2. Uniquement dans un composant React ou un hook personnalisé

Pas dans une fonction normale.

3. Toujours dans le même ordre

D'un rendu à l'autre, l'ordre doit être stable.

2. La règle des hooks

Objectif

Comprendre pourquoi les hooks doivent être appelés dans le même ordre à chaque rendu.

Pourquoi cette règle ?

React associe les hooks par **position dans le code**, pas par nom.

Exemple interne (simplifié) :

```
Hook #1 → useState compteur
Hook #2 → useEffect effet
Hook #3 → useState valeur input
```

Si tu appelles un hook dans une condition :

```
if (show) {
  useState(0);
}
```

L'ordre change.

Résultat : React associe les hooks aux mauvaises valeurs → bug total.

Les 3 règles officielles

1. Toujours au niveau supérieur

Pas dans :

- Conditions
- Boucles
- Fonctions internes

2. Uniquement dans un composant React ou un hook personnalisé

Pas dans une fonction normale.

3. Toujours dans le même ordre

D'un rendu à l'autre, l'ordre doit être stable.

Exemple simple : Afficher/masquer du texte

MAUVAIS (ce qu'on pourrait être tenté de faire)

```
function Message() {
  const [isVisible, setIsVisible] = useState(true);

  // ERREUR : hook dans une condition
  if (isVisible) {
    const [message, setMessage] = useState('Bonjour !');
  }

  return (
    <div>
      <button onClick={() => setIsVisible(!isVisible)}>
        Toggle
      </button>

      {isVisible && <p>{message}</p>}
    </div>
  );
}
```

Le problème :

Premier clic (isVisible = true) :

- Hook #1 → useState(isVisible)
- Hook #2 → useState(message) ☹

Deuxième clic (isVisible = false) :

- Hook #1 → useState(isVisible)
- Hook #2 → **Manquant !** (le if est faux)

Troisième clic (isVisible = true) :

- Hook #1 → useState(isVisible)
- Hook #2 → useState(message) ← React pense que c'est un **NOUVEAU** hook !

Résultat : Crash ou valeurs mélangées

BON (solution correcte)

```
function Message() {
  const [isVisible, setIsVisible] = useState(true);
  const [message, setMessage] = useState('Bonjour !'); // Toujours déclaré

  return (
    <div>
      <button onClick={() => setIsVisible(!isVisible)}>
        Toggle
      </button>

      {/* La condition est dans le JSX */}
      {isVisible && <p>{message}</p>}
    </div>
  );
}
```

Maintenant, à chaque rendu :

- Hook #1 → `useState(isVisible)`
- Hook #2 → `useState(message)`

L'ordre ne change **jamais** !

Règle d'or

Déclare TOUS tes hooks en haut du composant
Mets les conditions dans le JSX, pas autour des hooks

```
// BON : hooks toujours en haut
const [data, setData] = useState(null);
if (condition) {
  console.log(data); // OK d'utiliser la valeur
}

// MAUVAIS : hook dans une condition
if (condition) {
  const [data, setData] = useState(null); // Interdit !
}
```

3. Le hook useEffect

Objectif

Exécuter du code à des moments clés du cycle de vie :

- Au montage
- Lors de mises à jour
- Au démontage

Que fait useEffect exactement ?

Il permet d'exécuter un **effet après le rendu** (DOM disponible).

Il remplace les lifecycle methods :

- `componentDidMount`
- `componentDidUpdate`
- `componentWillUnmount`

Avec **une seule API**.

Structure

```
useEffect(() => {
  // effet
}, [dépendances]);
```

Dépendances :

- [] → effet exécuté **seulement au montage**
- [var] → effet exécuté **quand var change**
- Pas de tableau → à **chaque rendu** (rarement utilisé)

Exemple

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

function Lesson3UseEffect() {
  const [count, setCount] = useState(0);

  useEffect(() => {
    console.log("Compteur modifié :", count);
  }, [count]);

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

export default Lesson3UseEffect;
```

Mini exercice

Créer un composant affichant : « Le composant a été monté » dans la console.

4. Cleanup function

Objectif

Savoir nettoyer les effets pour éviter :

- Fuites mémoire
- Doublons d'écouteurs
- Intervals qui continuent de tourner

Quand React appelle le cleanup ?

- Lorsque le composant disparaît
- Juste avant de relancer l'effet (si dépendances changent)

Structure

```
useEffect(() => {
  // effet

  return () => {
    // cleanup
  };
}, []);
```

Exemple

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

function Lesson4Cleanup() {
  useEffect(() => {
    const id = setInterval(() => console.log("tick"), 1000);

    return () => clearInterval(id);
  }, []);

  return <p>Regarde la console</p>;
}

export default Lesson4Cleanup;
```

Mini exercice

Créer un intervalle qui se nettoie automatiquement au démontage.

5. Ajouter un événement global

Objectif

Écouter des événements du DOM global :

- scroll
- resize
- keydown
- mousemove

Pourquoi dans un useEffect ?

Parce que les listeners doivent être ajoutés après le rendu, et retirés au démontage.

Exemple

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

function Lesson5GlobalEvent() {
  const [scrollY, setScrollY] = useState(0);

  useEffect(() => {
    function handleScroll() {
      setScrollY(window.scrollY);
    }

    window.addEventListener("scroll", handleScroll);

    return () => window.removeEventListener("scroll", handleScroll);
  }, []);

  return <p>Position du scroll : {scrollY}px</p>;
}

export default Lesson5GlobalEvent;
```

Mini exercice

Créer un composant affichant la largeur de la fenêtre en temps réel.

6. Appeler une API

Objectif

Charger des données externes en React.

Pourquoi dans useEffect ?

Parce qu'un appel API dans le rendu provoquerait une **boucle infinie** :

```
rendu → fetch → setState → nouveau rendu → fetch → ...
```

`useEffect` empêche cette boucle.

Exemple

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

function Lesson6API() {
  const [users, setUsers] = useState([]);

  useEffect(() => {
    async function fetchUsers() {
      const res = await fetch("https://jsonplaceholder.typicode.com/users");
      const data = await res.json();
      setUsers(data);
    }
    fetchUsers();
  }, []);

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

export default Lesson6API;
```

Mini exercice

Afficher les titres de `/posts` .

7. Utiliser un observateur (IntersectionObserver)

Objectif

Détecter quand un élément entre/sort du viewport.

Très utile pour :

- Lazy loading d'images
- Animations au scroll
- Composants visibles uniquement si affichés

Exemple

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

function Lesson7Observer() {
  const ref = useRef();
  const [visible, setVisible] = useState(false);

  useEffect(() => {
    const obs = new IntersectionObserver([entry] =>
      setVisible(entry.isIntersecting)
    );

    if (ref.current) {
      obs.observe(ref.current);
    }

    return () => obs.disconnect();
  }, []);

  return (
    <div>
      <div style={{ height: "100vh" }}>Scrolle vers le bas</div>
      <div ref={ref} style={{ padding: "2rem", background: visible ? "green" : "red" }}>
        {visible ? "Visible" : "Invisible"}
      </div>
    </div>
  );
}

export default Lesson7Observer;
```

Mini exercice

Changer la couleur d'un bloc lorsqu'il apparaît.

8. Utiliser setInterval

Objectif

Effectuer une action répétitive.

Important

Toujours nettoyer l'interval pour éviter des comportements imprévisibles.

Exemple

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

function Lesson8SetInterval() {
  const [count, setCount] = useState(0);
  const intervalRef = useRef(null);

  useEffect(() => {
    intervalRef.current = setInterval(() => {
      setCount((c) => c + 1);
    }, 1000);

    return () => clearInterval(intervalRef.current);
  }, []);

  return <p>Compteur automatique : {count}</p>;
}

export default Lesson8SetInterval;
```

Mini exercice

Créer un compteur automatique avec boutons Start/Stop.

9. Sélectionner un tableau d'éléments

Objectif

Gérer plusieurs références DOM en même temps.

Exemple

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

function Lesson9SelectArray() {
  const itemsRef = useRef([]);

  function focusItem(index) {
    itemsRef.current[index]?.focus();
  }

  return (
    <div>
      <ul>
        {[ "A", "B", "C"].map((letter, i) => (
          <li key={i}>
            <input
              ref={(el) => (itemsRef.current[i] = el)}
              defaultValue={letter}
            />
          </li>
        ))}
      </ul>
      <button onClick={() => focusItem(1)}>Focus item 2</button>
    </div>
  );
}

export default Lesson9SelectArray;
```

Mini exercice

Créer un bouton "focus item 2".

10. Comprendre props.children

Objectif

Créer des composants "conteneurs".

Pourquoi utiliser children ?

Cela permet de faire des composants réutilisables comme :

- `<Card> ... </Card>`
- `<Modal> ... </Modal>`
- `<Layout> ... </Layout>`

Exemple

```
function Card({ children }) {
  return (
    <div style={{ border: "1px solid #ddd", padding: "1rem", borderRadius: "8px" }}>
      {children}
    </div>
  );
}

function Lesson10Children() {
  return (
    <Card>
      <h2>Titre</h2>
      <p>Contenu de la carte</p>
    </Card>
  );
}

export default Lesson10Children;
```

Mini exercice

Créer un composant `Alert` qui encadre son contenu.

11. Memo et useCallback

Objectif

Optimiser les performances.

Pourquoi ?

React.memo

- Empêche des re-renders inutiles

useCallback

- Mémorise une fonction pour stabiliser son identité

Exemple

```
import React, { useState, useCallback } from "react";

const Button = React.memo(function Button({ onClick }) {
  console.log("Render Button");
  return <button onClick={onClick}>Clique-moi</button>;
});

function Lesson11MemoCallback() {
  const [count, setCount] = useState(0);

  const handleClick = useCallback(() => {
    console.log("clicked");
  }, []);

  return (
    <>
      <Button onClick={handleClick} />
      <p>Count : {count}</p>
      <button onClick={() => setCount(count + 1)}>+1</button>
    </>
  );
}

export default Lesson11MemoCallback;
```

Mini exercice

Créer un composant enfant optimisé affichant un texte.

12. useMemo

Objectif

Éviter de recalculer une valeur coûteuse.

Exemple

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

const expensive = (n) => {
  console.log("Calcul coûteux...");
  for (let i = 0; i < n * 1e7; i++);
  return n * 2;
};

function Lesson12UseMemo() {
  const [count, setCount] = useState(1);
  const [other, setOther] = useState(0);

  const result = useMemo(() => expensive(count), [count]);

  return (
    <div>
      <p>Résultat : {result}</p>
      <button onClick={() => setCount(count + 1)}>+1 Count</button>
      <button onClick={() => setOther(other + 1)}>+1 Other</button>
    </div>
  );
}

export default Lesson12UseMemo;
```

Mini exercice

Mémoriser une moyenne d'un tableau via `useMemo`.

13. useReducer

Objectif

Gérer un état complexe via des actions.

Très utilisé dans :

- Formulaires complexes
- Logique multiple
- Alternatives à Redux

Exemple

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

function reducer(state, action) {
  switch (action.type) {
    case "increment":
      return { count: state.count + 1 };
    case "decrement":
      return { count: state.count - 1 };
    case "reset":
      return { count: 0 };
    default:
      return state;
  }
}

function Lesson13Reducer() {
  const [state, dispatch] = useReducer(reducer, { count: 0 });

  return (
    <div>
      <p>Compteur : {state.count}</p>
      <button onClick={() => dispatch({ type: "decrement" })}>-1</button>
      <button onClick={() => dispatch({ type: "increment" })}>+1</button>
      <button onClick={() => dispatch({ type: "reset" })}>Reset</button>
    </div>
  );
}

export default Lesson13Reducer;
```

Mini exercice

Ajouter une action `reset`.

14. Hook personnalisé

Objectif

Réutiliser une logique commune dans plusieurs composants.

Exemple

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

function useWindowWidth() {
  const [width, setWidth] = useState(window.innerWidth);

  useEffect(() => {
    const handler = () => setWidth(window.innerWidth);
    window.addEventListener("resize", handler);
    return () => window.removeEventListener("resize", handler);
  }, []);

  return width;
}

function Lesson14CustomHook() {
  const width = useWindowWidth();

  return <p>Largeur de la fenêtre : {width}px</p>;
}

export default Lesson14CustomHook;
```

Mini exercice

Créer un hook `useOnline()` qui retourne `true/false` selon la connexion internet.

15. CSS Modules

Objectif

Avoir des classes de style isolées, sans collisions.

Exemple

style.module.css

```
.title {
  color: red;
  font-size: 2rem;
}
```

Composant

```
import styles from "./style.module.css";

function Lesson15CSSModules() {
  return <h1 className={styles.title}>Titre rouge</h1>;
}

export default Lesson15CSSModules;
```

Mini exercice

Créer un badge avec style isolé.

16. Utiliser TailwindCSS

Objectif

Utiliser une librairie CSS utilitaire pour faire du design rapidement.

Installation

```
npm install -D tailwindcss postcss autoprefixer
npx tailwindcss init -p
```

tailwind.config.js

```
export default {
  content: ["/index.html", "./src/**/*.{js,jsx}"],
  theme: {
    extend: {},
  },
  plugins: [],
};
```

index.css

```
@tailwind base;
@tailwind components;
@tailwind utilities;
```

Exemple

```
function Lesson16Tailwind() {
  return (
    <div className="p-4">
      <button className="px-4 py-2 bg-blue-600 text-white rounded hover:bg-blue-700">
        Bouton Tailwind
      </button>
    </div>
  );
}

export default Lesson16Tailwind;
```

Mini exercice

Créer une carte (Card) avec :

- Titre
- Texte
- Bouton stylé

Fin du Module 2

Tu maîtrises maintenant les concepts avancés de React ! 🎉

Pour aller encore plus loin :

- React Router (navigation)
- Context API (état global)
- React Query (gestion de cache serveur)
- Testing Library (tests)