Hooks fondamentaux

useState, useEffect et useRef

- useState
- useEffect
- useRef

useState

useState

Permet d'ajouter un état local à un composant fonctionnel.

Syntaxe

```
const [state, setState] = useState(valeurInitiale);
```

Exemples

```
// Booléen
const [isOpen, setIsOpen] = useState(false);

// Tableau
const [items, setItems] = useState([]);
const addItem = (item) => setItems(prev => [...prev, item]);
const removeItem = (id) => setItems(prev => prev.filter(i => i.id !== id));

// Objet
const [form, setForm] = useState({ email: "", password: "" });
const handleChange = (e) => {
  setForm(prev => ({ ...prev, [e.target.name]: e.target.value }));
};
```

Initialisation paresseuse (lazy initialization)

Si le calcul de la valeur initiale est coûteux, passer une fonction pour ne l'exécuter qu'une fois :

```
// □ Calculé à chaque render
const [data, setData] = useState(computeExpensiveValue());

// □ Calculé une seule fois
const [data, setData] = useState(() => computeExpensiveValue());
```

Règles importantes

- Ne jamais muter le state directement — toujours créer une nouvelle valeur
- Les mises à jour de state sont **asynchrones** (batchées)
- Utiliser la forme fonctionnelle quand le nouvel état dépend du précédent

useEffect

useEffect

Permet d'exécuter des effets de bord (appels API, timers, listeners...) après le rendu.

Syntaxe

```
useEffect(() => {
  // effet
  return () => { /* cleanup */ }; // optionnel
}, [/* dépendances */]);
```

Les 3 modes selon le tableau de dépendances

```
// 1. Sans tableau → s'exécute après CHAQUE render
useEffect(() => {
  console.log("render");
});

// 2. Tableau vide → s'exécute UNE FOIS au montage
useEffect(() => {
  console.log("monté");
}, []);

// 3. Avec dépendances → s'exécute quand une dépendance change
useEffect(() => {
  console.log("userId a changé :", userId);
}, [userId]);
```

Exemple complet : fetch de données

```
function UserProfile({ userId }) {
  const [user, setUser] = useState(null);
  const [loading, setLoading] = useState(true);
```

```
  useEffect(() => {
    setLoading(true);
    const controller = new AbortController();

    fetch(`/api/users/${userId}`, { signal: controller.signal })
      .then(res => res.json())
      .then(data => { setUser(data); setLoading(false); })
      .catch(err => { if (err.name !== 'AbortError') console.error(err); });

    return () => controller.abort();
  }, [userId]);

  if (loading) return <p>Chargement...</p>;
  return <p>{user?.nom}</p>;
}
```

Pièges courants

- **Dépendances manquantes** : toujours inclure toutes les variables utilisées dans l'effet
- **Boucle infinie** : si une dépendance est un objet/tableau créé dans le render, il change à chaque fois → utiliser `useMemo` ou le déplacer hors du composant
- **State stale** : préférer la forme fonctionnelle de setState dans un useEffect

useRef

useRef

`useRef` retourne un objet `{ current: valeur }` qui **persiste entre les renders** sans déclencher de re-render quand il change. C'est sa différence fondamentale avec `useState`.

Syntaxe

```
const ref = useRef(valeurInitiale);
// Accès : ref.current
```

Usage 1 : Accéder à un élément DOM

```
function InputAutofocus() {
  const inputRef = useRef(null);

  useEffect(() => {
    inputRef.current.focus(); // Focus au montage
  }, []);

  return <input ref={inputRef} placeholder="Tapez ici..." />;
}

// Lire la valeur sans contrôler l'input (uncontrolled)
function Form() {
  const nameRef = useRef(null);

  const handleSubmit = () => {
    console.log(nameRef.current.value);
  };

  return (
    <form onSubmit={handleSubmit}>
      <input ref={nameRef} />
```

```
      <button type="submit">Envoyer</button>
    </form>
  );
}
```

Usage 2 : Stocker une valeur sans re-render

Idéal pour stocker des valeurs qui ne doivent pas provoquer de re-render : timers, valeurs précédentes, flags...

```
// Garder trace de l'ID d'un intervalle
function Timer() {
  const [count, setCount] = useState(0);
  const intervalRef = useRef(null);

  const start = () => {
    intervalRef.current = setInterval(() => {
      setCount(prev => prev + 1);
    }, 1000);
  };

  const stop = () => clearInterval(intervalRef.current);

  return (
    <div>
      <p>{count}</p>
      <button onClick={start}>Démarrer</button>
      <button onClick={stop}>Arrêter</button>
    </div>
  );
}
```

Usage 3 : Capturer la valeur précédente d'un state

```
function usePrevious(value) {
  const ref = useRef();
  useEffect(() => {
```

```
    ref.current = value; // Met à jour APRÈS le render
  });
  return ref.current; // Retourne la valeur PRÉCÉDENTE
}

function Compteur() {
  const [count, setCount] = useState(0);
  const prevCount = usePrevious(count);

  return <p>Avant: {prevCount} | Maintenant: {count}</p>;
}
```

useRef vs useState

	useRef	**useState**
Persiste entre renders	✅	✅
Déclenche un re-render	❌	✅
Accès au DOM	✅	❌
Valeur mutable	✅ (ref.current)	❌ (setter requis)