Hooks avancés

useContext, useReducer, useMemo et useCallback

- useContext
- useReducer
- useMemo et useCallback

useContext

useContext

Permet de lire une valeur de contexte sans passer par le prop drilling. Le contexte est un état partagé accessible par tous les composants descendants.

Création d'un contexte

```
import { createContext, useContext, useState } from 'react';

// 1. Créer le contexte
const ThemeContext = createContext('light'); // valeur par défaut

// 2. Fournir le contexte
function App() {
  const [theme, setTheme] = useState('light');

  return (
    <ThemeContext.Provider value={{ theme, setTheme }}>
      <Navbar />
      <Main />
    </ThemeContext.Provider>
  );
}

// 3. Consommer le contexte
function Navbar() {
  const { theme, setTheme } = useContext(ThemeContext);

  return (
    <nav className={theme}>
      <button onClick={() => setTheme(t => t === 'light' ? 'dark' : 'light')}>
        Basculer le thème
      </button>
    </nav>
```

```
  );
}
```

Pattern : custom hook pour le contexte

```
// Bonne pratique : encapsuler dans un hook custom
function useTheme() {
  const ctx = useContext(ThemeContext);
  if (!ctx) throw new Error("useTheme doit être utilisé dans ThemeProvider");
  return ctx;
}

// Utilisation
const { theme } = useTheme();
```

Quand utiliser le contexte ?

- Thème (dark/light)
- Langue / i18n
- Utilisateur connecté
- Toute donnée vraiment globale

⚠ Le contexte re-rend **tous les consommateurs** quand la valeur change. Pour des données très fréquemment mises à jour, préférer un gestionnaire d'état dédié (Zustand, Redux).

useReducer

useReducer

Alternative à `useState` pour des états complexes avec plusieurs sous-valeurs ou des transitions d'état interdépendantes. Inspiré du pattern Redux.

Syntaxe

```
const [state, dispatch] = useReducer(reducer, initialState);
```

Exemple : gestion d'un formulaire

```
const initialState = { nom: "", email: "", loading: false, error: null };

function reducer(state, action) {
  switch (action.type) {
    case 'SET_FIELD':
      return { ...state, [action.field]: action.value };
    case 'SUBMIT_START':
      return { ...state, loading: true, error: null };
    case 'SUBMIT_SUCCESS':
      return { ...initialState };
    case 'SUBMIT_ERROR':
      return { ...state, loading: false, error: action.error };
    default:
      return state;
  }
}

function Formulaire() {
  const [state, dispatch] = useReducer(reducer, initialState);

  const handleChange = (e) => dispatch({
    type: 'SET_FIELD', field: e.target.name, value: e.target.value
```

```
  });

  const handleSubmit = async (e) => {
    e.preventDefault();
    dispatch({ type: 'SUBMIT_START' });
    try {
      await submitForm(state);
      dispatch({ type: 'SUBMIT_SUCCESS' });
    } catch (err) {
      dispatch({ type: 'SUBMIT_ERROR', error: err.message });
    }
  };

  return (
    <form onSubmit={handleSubmit}>
      <input name="nom" value={state.nom} onChange={handleChange} />
      <input name="email" value={state.email} onChange={handleChange} />
      {state.error && <p>{state.error}</p>}
      <button disabled={state.loading}>Envoyer</button>
    </form>
  );
}
```

useReducer vs useState

useState	useReducer
État simple (string, number, bool)	État complexe (objet avec plusieurs champs)
Transitions simples	Transitions liées ou conditionnelles
Pas de logique dans le setter	Logique centralisée dans le reducer

useMemo et useCallback

useMemo et useCallback

Ces deux hooks servent à **mémoïser** des valeurs ou des fonctions pour éviter des recalculs ou recréations inutiles entre les renders.

useMemo

Mémoïse le **résultat** d'un calcul coûteux.

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

function Liste({ items, filtre }) {
  // Recalculé SEULEMENT quand items ou filtre change
  const itemsFiltres = useMemo(() => {
    return items.filter(item => item.nom.includes(filtre));
  }, [items, filtre]);

  return <ul>{itemsFiltres.map(i => <li key={i.id}>{i.nom}</li>)}</ul>;
}
```

useCallback

Mémoïse une **fonction** pour qu'elle garde la même référence entre les renders.

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

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

  // Sans useCallback : nouvelle référence à chaque render → Enfant se re-rend inutilement
  // Avec useCallback : même référence si count ne change pas
  const handleClick = useCallback(() => {
```

```
    setCount(prev => prev + 1);
  }, []); // pas de dépendance car on utilise la forme fonctionnelle

  return <Enfant onClick={handleClick} />;
}

const Enfant = React.memo(({ onClick }) => {
  console.log("Enfant rendu");
  return <button onClick={onClick}>+</button>;
});
```

Règle d'or

N'utilisez **pas** useMemo/useCallback par défaut sur tout. Ce sont des optimisations qui ont un coût. Utilisez-les seulement quand :

- Le calcul est réellement coûteux (filtres lourds, sorts, calculs mathématiques)
- La valeur/fonction est passée en prop à un composant mémoïsé (`React.memo`)
- La valeur est une dépendance d'un `useEffect`