# Performance et bonnes pratiques

Optimisations et patterns à éviter

# Optimisations

## Optimisations de performance

### React.memo — éviter les re-renders inutiles

```
// L'enfant se re-rend seulement si ses props changent
const ListItem = React.memo(function ListItem({ item, onDelete }) {
  return (
    <li>
      {item.nom}
      <button onClick={() => onDelete(item.id)}>Supprimer</button>
    </li>
  );
});
```

### Code splitting — lazy loading

```
import { lazy, Suspense } from 'react';

// Chargé seulement quand la route est visitée
const Dashboard = lazy(() => import('./pages/Dashboard'));
const Settings = lazy(() => import('./pages/Settings'));

function App() {
  return (
    <Suspense fallback={<Spinner />}>
      <Routes>
        <Route path="/dashboard" element={<Dashboard />} />
        <Route path="/settings" element={<Settings />} />
      </Routes>
    </Suspense>
  );
}
```

# Virtualisation des listes

Pour des listes de centaines/milliers d'éléments, n'afficher que ce qui est visible :

```
npm install @tanstack/react-virtual

import { useVirtualizer } from '@tanstack/react-virtual';

function GrandeListe({ items }) {
  const parentRef = useRef(null);

  const virtualizer = useVirtualizer({
    count: items.length,
    getScrollElement: () => parentRef.current,
    estimateSize: () => 50,
  });

  return (
    <div ref={parentRef} style={{ height: '400px', overflow: 'auto' }}>
      <div style={{ height: virtualizer.getTotalSize() }}>
        {virtualizer.getVirtualItems().map(vItem => (
          <div key={vItem.index} style={{ transform: `translateY(${vItem.start}px)` }}>
            {items[vItem.index].nom}
          </div>
        ))}
      </div>
    </div>
  );
}
```

# Bonnes pratiques

# Bonnes pratiques React

## Structure de fichiers

```
src/
├── components/          # Composants réutilisables
│   ├── Button/
│   │   ├── Button.jsx
│   │   └── Button.css
│   └── Modal/
├── pages/               # Composants de page (1 par route)
├── hooks/               # Custom hooks
├── contexts/            # Contextes React
├── services/            # Appels API
├── utils/               # Fonctions utilitaires
└── types/               # TypeScript types/interfaces
```

## Règles à respecter

- **Un composant = une responsabilité** — si un composant est trop gros, le découper
- **Extraire la logique** dans des custom hooks (`useForm`, `useFetch`...)
- **Remonter l'état** (lift state up) au plus proche ancêtre commun qui en a besoin
- **Toujours mettre une key** sur les éléments d'une liste — idéalement un ID, pas l'index
- **Éviter l'état dérivé** — ne pas stocker dans le state ce qui peut être calculé

## Anti-patterns courants

```
// ❌ Synchroniser deux états (état dérivé)
const [items, setItems] = useState([]);
const [count, setCount] = useState(0);
// Chaque ajout doit mettre à jour les deux

// ✅ Calculer le dérivé
const [items, setItems] = useState([]);
```

```
const count = items.length; // Calculé, pas stocké

// ❌ useEffect pour synchroniser des states
useEffect(() => setFullName(first + " " + last), [first, last]);

// ✅ Variable calculée
const fullName = first + " " + last;

// ❌ Key = index (problèmes lors de réordonnancement)
{items.map((item, i) => <Item key={i} />)}

// ✅ Key = ID stable
{items.map(item => <Item key={item.id} />)}
```