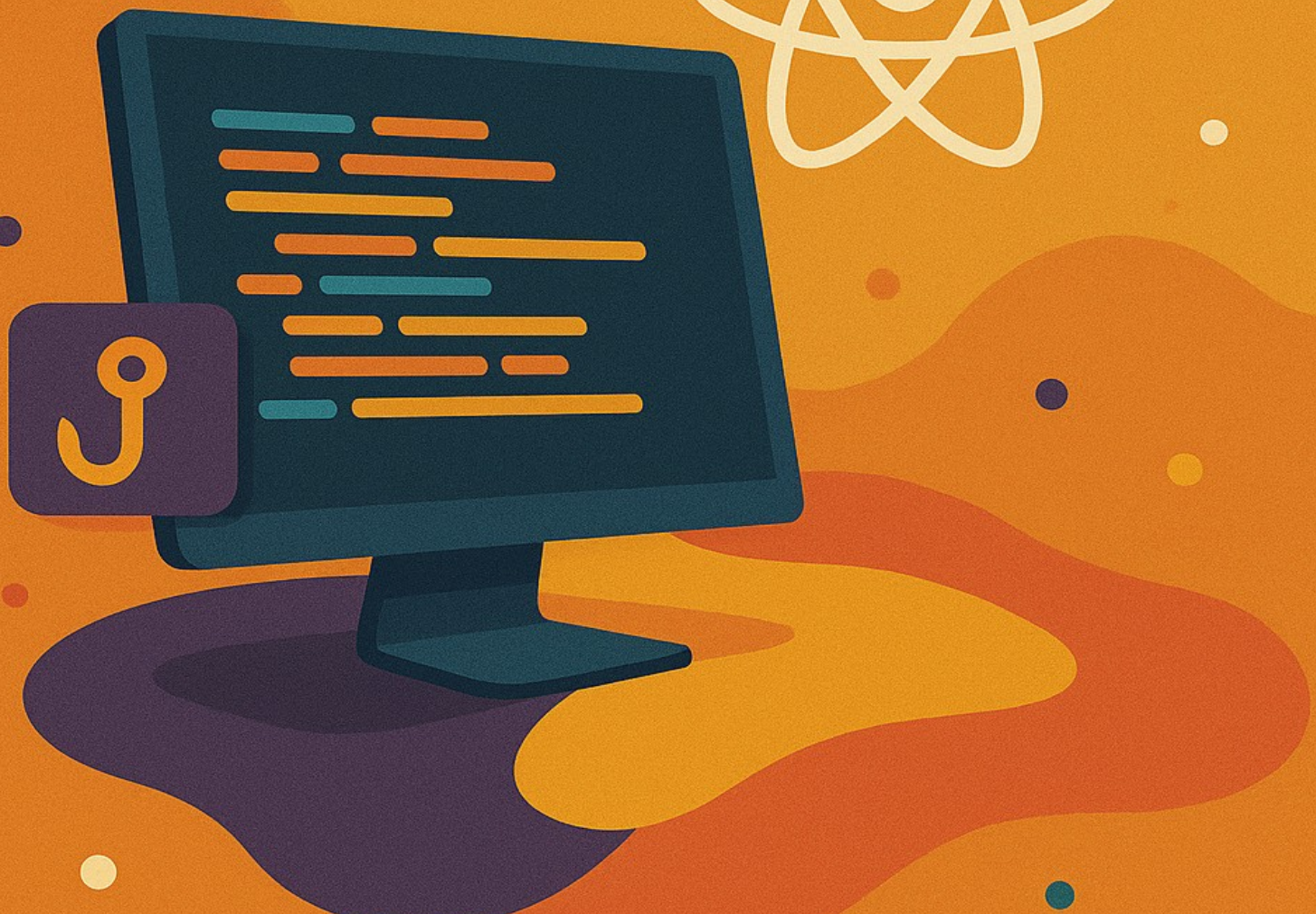
FORMATION

REACT JS HOOKS



React JS pour Débutants

Maîtriser les Hooks et les Composants

🔥 Les Hooks Avancés 🔥

Partie 2 : Niveau Intermédiaire (Pages 16-30)

📚 Table des Matières - Partie 2

6. useEffect : Les Effets de Bord

useEffect est comme un espion dans votre composant : il surveille les changements et réagit ! C'est le hook qui gère tout ce qui se passe 'à côté' du rendu normal.

> 💡 *useEffect, c'est comme mettre une alarme sur votre téléphone : vous programmez quelque chose qui se déclenchera au bon moment !*

useEffect Basique

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

function Horloge() {
  const [heure, setHeure] = useState(new Date());

  useEffect(() => {
    // Code qui s'exécute après chaque rendu
    console.log("Composant rendu !");

    // Mise à jour de l'heure chaque seconde
    const intervalle = setInterval(() => {
      setHeure(new Date());
    }, 1000);

    // Fonction de nettoyage
    return () => {
      clearInterval(intervalle);
      console.log("Nettoyage effectué !");
    };
  }, []); // Tableau vide = exécuté une seule fois

  return (
    <div>
      <h2>Il est {heure.toLocaleTimeString()}</h2>
    </div>
  );
}
```

Points clés :

- useEffect prend deux arguments : une fonction et un tableau de dépendances

- La fonction return dans useEffect = nettoyage (cleanup)
- Le tableau vide [] signifie : exécuter une seule fois au montage

Les Dépendances de useEffect

```
function Recherche() {
  const [terme, setTerme] = useState('');
  const [resultats, setResultats] = useState([]);

  // S'exécute à chaque changement de 'terme'
  useEffect(() => {
    if (terme.length > 2) {
      // Simuler une recherche
      console.log("Recherche de:", terme);

      // Appel API simulé
      fetch(`https://api.example.com/search?q=${terme}`)
        .then(res => res.json())
        .then(data => setResultats(data));
    }
  }, [terme]); // Dépendance : terme

  return (
    <div>
      <input
        value={terme}
        onChange={(e) => setTerme(e.target.value)}
        placeholder="Rechercher..."
      />
      <ul>
        {resultats.map(r => (
          <li key={r.id}>{r.nom}</li>
        ))}
      </ul>
    </div>
  );
}
```

> ⚠️ **Attention : Toujours inclure TOUTES les variables utilisées dans useEffect dans le tableau de dépendances !**

Les Trois Cas d'Usage

```
// 1. Sans dépendances : À chaque rendu
useEffect(() => {
```

```
  console.log("Exécuté à CHAQUE rendu");
});

// 2. Tableau vide : Une seule fois au montage
useEffect(() => {
  console.log("Exécuté UNE FOIS au début");
}, []);

// 3. Avec dépendances : Quand les deps changent
useEffect(() => {
  console.log("Exécuté quand count change");
}, [count]);
```

💡 Les dépendances de useEffect, c'est comme choisir ses notifications : soyez sélectif ou vous serez spammé !

Récupérer des Données avec useEffect

```
function ListeUtilisateurs() {
  const [utilisateurs, setUtilisateurs] = useState([]);
  const [chargement, setChargement] = useState(true);
  const [erreur, setErreur] = useState(null);

  useEffect(() => {
    // Fonction async dans useEffect
    const chargerUtilisateurs = async () => {
      try {
        setChargement(true);
        const response = await fetch(
          'https://jsonplaceholder.typicode.com/users'
        );
        const data = await response.json();
        setUtilisateurs(data);
        setErreur(null);
      } catch (err) {
        setErreur(err.message);
      } finally {
        setChargement(false);
      }
    };

    chargerUtilisateurs();
  }, []);

  if (chargement) return <p>Chargement...</p>;
  if (erreur) return <p>Erreur : {erreur}</p>;

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

7. Gestion d'Événements

Les événements en React sont comme des conversations : votre application écoute et répond aux actions de l'utilisateur !

Événements de Base

```
function Boutons() {
  const handleClick = () => {
    alert("Bouton cliqué !");
  };

  const handleDoubleClick = () => {
    console.log("Double-clic !");
  };

  const handleMouseEnter = () => {
    console.log("Souris entrée");
  };

  return (
    <div>
      <button onClick={handleClick}>
        Cliquer
      </button>

      <button onDoubleClick={handleDoubleClick}>
        Double-cliquer
      </button>

      <div onMouseEnter={handleMouseEnter}>
        Survolez-moi !
      </div>
    </div>
  );
}
```

💡 Les événements React sont comme des boutons d'ascenseur : appuyez une fois et attendez !

Événements avec Paramètres

```
function ListeCouleurs() {
  const [couleurActive, setCouleurActive] = useState('');

  const changerCouleur = (couleur) => {
    setCouleurActive(couleur);
    console.log("Couleur sélectionnée:", couleur);
  };

  return (
    <div>
      <p>Couleur active : {couleurActive}</p>

      {/* Méthode 1 : Arrow function inline */}
      <button onClick={() => changerCouleur('rouge')}>
        Rouge
      </button>

      {/* Méthode 2 : Bind */}
      <button onClick={changerCouleur.bind(null, 'bleu')}>
        Bleu
      </button>

      <div style={{
        width: '100px',
        height: '100px',
        backgroundColor: couleurActive
      }} />
    </div>
  );
}
```

Événements de Formulaire

```
function Formulaire() {
  const [donnees, setDonnees] = useState({
    nom: '',
    email: '',
    message: ''
  });

  const handleChange = (e) => {
    const { name, value } = e.target;
    setDonnees(prev => ({
      ...prev,
      [name]: value
    }));
  };

  const handleSubmit = (e) => {
    e.preventDefault(); // Empêche le rechargement
    console.log("Données soumises:", donnees);
    alert(`Merci ${donnees.nom} !`);
  };

  return (
    <form onSubmit={handleSubmit}>
      <input
        name="nom"
        value={donnees.nom}
        onChange={handleChange}
        placeholder="Votre nom"
      />
      <input
        name="email"
        value={donnees.email}
        onChange={handleChange}
        placeholder="Votre email"
      />
      <textarea
        name="message"
        value={donnees.message}
        onChange={handleChange}
        placeholder="Votre message"
      />
```

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

⚠️ **N'oubliez jamais e.preventDefault() dans onSubmit pour éviter le rechargement de la page !**

8. Rendu Conditionnel

Le rendu conditionnel permet d'afficher différentes choses selon les conditions. C'est comme un aiguillage de train pour vos composants !

If/Else avec return

```
function Message({ estConnecte }) {
  if (estConnecte) {
    return <h1>Bienvenue !</h1>;
  } else {
    return <h1>Veuillez vous connecter</h1>;
  }
}

// Utilisation
function App() {
  return (
    <div>
      <Message estConnecte={true} />
      <Message estConnecte={false} />
    </div>
  );
}
```

Opérateur Ternaire

```
function Statut({ estEnLigne }) {
  return (
    <div>
      <span>Statut : </span>
      <span style={{
        color: estEnLigne ? 'green' : 'red'
      }}>
        {estEnLigne ? '🟢 En ligne' : '🔴 Hors ligne'}
      </span>
    </div>
  );
}
```

💡 L'opérateur ternaire, c'est comme un interrupteur : allumé ou éteint, pas d'entre-deux !

Opérateur && (ET logique)

```
function Notifications({ messages }) {
  return (
    <div>
      <h2>Notifications</h2>

      {/* Affiche seulement si messages.length > 0 */}
      {messages.length > 0 && (
        <p>Vous avez {messages.length} nouveau(x) message(s)</p>
      )}

      {messages.length === 0 && (
        <p>Aucune notification</p>
      )}
    </div>
  );
}

// Utilisation
<Notifications messages={['Msg1', 'Msg2']} />
<Notifications messages={[]} />
```

Switch avec Conditions Multiples

```
function Badge({ type }) {
  let couleur, texte;

  switch(type) {
    case 'success':
      couleur = 'green';
      texte = '✓ Succès';
      break;
    case 'error':
      couleur = 'red';
      texte = '✗ Erreur';
      break;
    case 'warning':
      couleur = 'orange';
      texte = '⚠️ Attention';
      break;
    default:
```

```
      couleur = 'gray';
      texte = 'Info';
  }

  return (
    <span style={{ color: couleur, fontWeight: 'bold' }}>
      {texte}
    </span>
  );
}
```

9. Listes et Keys

Afficher des listes est l'une des tâches les plus courantes en React. La méthode map() est votre meilleure amie !

Liste Simple

```
function ListeFruits() {
  const fruits = ['Pomme', 'Banane', 'Orange', 'Fraise'];

  return (
    <ul>
      {fruits.map((fruit, index) => (
        <li key={index}>{fruit}</li>
      ))}
    </ul>
  );
}
```

> ⚠️ **Les keys sont OBLIGATOIRES dans les listes ! Elles aident React à identifier les éléments.**

Liste avec Objets

```
function ListeContacts() {
  const contacts = [
    { id: 1, nom: 'Alice', email: 'alice@email.com' },
    { id: 2, nom: 'Bob', email: 'bob@email.com' },
    { id: 3, nom: 'Charlie', email: 'charlie@email.com' }
  ];

  return (
    <div>
      {contacts.map(contact => (
        <div key={contact.id} className="carte-contact">
          <h3>{contact.nom}</h3>
          <p>{contact.email}</p>
        </div>
      ))}
    </div>
  );
```

```
}
```

💡 Les keys en React sont comme des plaques d'immatriculation : chaque élément doit avoir son identifiant unique !

Liste Filtrée et Triée

```
function ListeProduits() {
  const [filtre, setFiltre] = useState('');
  const [tri, setTri] = useState('nom');

  const produits = [
    { id: 1, nom: 'Laptop', prix: 1200, stock: 5 },
    { id: 2, nom: 'Souris', prix: 25, stock: 50 },
    { id: 3, nom: 'Clavier', prix: 80, stock: 30 }
  ];

  const produitsFiltres = produits
    .filter(p =>
      p.nom.toLowerCase().includes(filtre.toLowerCase())
    )
    .sort((a, b) => {
      if (tri === 'nom') return a.nom.localeCompare(b.nom);
      if (tri === 'prix') return a.prix - b.prix;
      return 0;
    });

  return (
    <div>
      <input
        value={filtre}
        onChange={(e) => setFiltre(e.target.value)}
        placeholder="Rechercher..."
      />

      <select value={tri} onChange={(e) => setTri(e.target.value)}>
        <option value="nom">Trier par nom</option>
        <option value="prix">Trier par prix</option>
      </select>

      <div>
        {produitsFiltres.map(produit => (
          <div key={produit.id}>
            <h3>{produit.nom}</h3>
            <p>Prix : {produit.prix}€</p>
            <p>Stock : {produit.stock}</p>
          </div>
        ))}
```

```
      </div>
    </div>
  );
}
```

10. Formulaires Contrôlés

Un formulaire contrôlé est un formulaire dont les valeurs sont gérées par React. C'est la méthode recommandée !

Formulaire Complet

```
function FormulaireInscription() {
  const [form, setForm] = useState({
    nom: '',
    email: '',
    age: '',
    pays: 'France',
    newsletter: false,
    genre: ''
  });

  const [erreurs, setErreurs] = useState({});

  const handleChange = (e) => {
    const { name, value, type, checked } = e.target;
    setForm(prev => ({
      ...prev,
      [name]: type === 'checkbox' ? checked : value
    }));
  };

  const valider = () => {
    const nouvellesErreurs = {};

    if (!form.nom.trim()) {
      nouvellesErreurs.nom = 'Le nom est requis';
    }

    if (!form.email.includes('@')) {
      nouvellesErreurs.email = 'Email invalide';
    }

    if (form.age < 18) {
      nouvellesErreurs.age = 'Vous devez avoir 18 ans';
    }
```

```
  setErreurs(nouvellesErreurs);
  return Object.keys(nouvellesErreurs).length === 0;
};

const handleSubmit = (e) => {
  e.preventDefault();
  if (valider()) {
    console.log('Formulaire valide:', form);
    alert('Inscription réussie !');
  }
};

return (
  <form onSubmit={handleSubmit}>
    <div>
      <label>Nom :</label>
      <input
        name="nom"
        value={form.nom}
        onChange={handleChange}
      />
      {erreurs.nom && <span style={{color: 'red'}}>
        {erreurs.nom}
      </span>}
    </div>

    <div>
      <label>Email :</label>
      <input
        name="email"
        type="email"
        value={form.email}
        onChange={handleChange}
      />
      {erreurs.email && <span style={{color: 'red'}}>
        {erreurs.email}
      </span>}
    </div>

    <div>
      <label>Âge :</label>
      <input
```

```
          name="age"
          type="number"
          value={form.age}
          onChange={handleChange}
        />
        {erreurs.age && <span style={{color: 'red'}}>
          {erreurs.age}
        </span>}
      </div>

      <div>
        <label>Pays :</label>
        <select name="pays" value={form.pays} onChange={handleChange}>
          <option value="France">France</option>
          <option value="Belgique">Belgique</option>
          <option value="Suisse">Suisse</option>
          <option value="Canada">Canada</option>
        </select>
      </div>

      <div>
        <label>Genre :</label>
        <label>
          <input
            type="radio"
            name="genre"
            value="homme"
            checked={form.genre === 'homme'}
            onChange={handleChange}
          /> Homme
        </label>
        <label>
          <input
            type="radio"
            name="genre"
            value="femme"
            checked={form.genre === 'femme'}
            onChange={handleChange}
          /> Femme
        </label>
      </div>

      <div>
        <label>
```

```
          <input
            type="checkbox"
            name="newsletter"
            checked={form.newsletter}
            onChange={handleChange}
          /> S'abonner à la newsletter
        </label>
      </div>

      <button type="submit">S'inscrire</button>
    </form>
  );
}
```

💡 *Les formulaires React sont comme des questionnaires bien organisés : chaque réponse est enregistrée en temps réel !*

11. useContext : Partager des Données

useContext résout le problème du 'prop drilling' (passer des props à travers plusieurs niveaux). C'est comme un Wi-Fi pour vos données : accessible partout !

Créer un Context

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

// 1. Créer le Context
const ThemeContext = createContext();

// 2. Créer le Provider
function ThemeProvider({ children }) {
  const [theme, setTheme] = useState('clair');

  const toggleTheme = () => {
    setTheme(prev => prev === 'clair' ? 'sombre' : 'clair');
  };

  return (
    <ThemeContext.Provider value={{ theme, toggleTheme }}>
      {children}
    </ThemeContext.Provider>
  );
}

// 3. Hook personnalisé pour utiliser le Context
function useTheme() {
  const context = useContext(ThemeContext);
  if (!context) {
    throw new Error('useTheme doit être utilisé dans ThemeProvider');
  }
  return context;
}

export { ThemeProvider, useTheme };
```

💡 useContext, c'est comme avoir un assistant personnel : plus besoin de se passer les infos de main en main !

Utiliser le Context

```
// Composant Header
function Header() {
  const { theme, toggleTheme } = useTheme();

  return (
    <header style={{
      background: theme === 'clair' ? '#fff' : '#333',
      color: theme === 'clair' ? '#000' : '#fff'
    }}>
      <h1>Mon Application</h1>
      <button onClick={toggleTheme}>
        {theme === 'clair' ? '🌙' : '☀️'}
      </button>
    </header>
  );
}

// Composant Content
function Content() {
  const { theme } = useTheme();

  return (
    <div style={{
      background: theme === 'clair' ? '#f5f5f5' : '#222',
      color: theme === 'clair' ? '#000' : '#fff',
      padding: '20px'
    }}>
      <p>Contenu de l'application</p>
    </div>
  );
}

// App avec Provider
function App() {
  return (
    <ThemeProvider>
      <Header />
      <Content />
    </ThemeProvider>
  );
}
```

Context d'Authentification

```
const AuthContext = createContext();

function AuthProvider({ children }) {
  const [user, setUser] = useState(null);

  const login = (email, password) => {
    // Simulation de connexion
    setUser({ email, nom: 'Utilisateur' });
  };

  const logout = () => {
    setUser(null);
  };

  return (
    <AuthContext.Provider value={{ user, login, logout }}>
      {children}
    </AuthContext.Provider>
  );
}

function useAuth() {
  return useContext(AuthContext);
}

// Composant protégé
function DashboardPrive() {
  const { user, logout } = useAuth();

  if (!user) {
    return <p>Accès refusé. Connectez-vous.</p>;
  }

  return (
    <div>
      <h2>Bienvenue {user.nom} !</h2>
      <button onClick={logout}>Se déconnecter</button>
    </div>
  );
}
```

12. useRef : Références et DOM

useRef crée une référence qui persiste entre les rendus sans provoquer de re-render. Parfait pour accéder au DOM ou stocker des valeurs !

Accéder aux Éléments DOM

```
import { useRef, useEffect } from 'react';

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

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

  const handleClick = () => {
    // Accès direct au DOM
    alert(`Valeur: ${inputRef.current.value}`);
    inputRef.current.style.backgroundColor = 'yellow';
  };

  return (
    <div>
      <input ref={inputRef} type="text" />
      <button onClick={handleClick}>Lire la valeur</button>
    </div>
  );
}
```

> 💡 *useRef, c'est comme avoir un post-it invisible : personne ne voit quand vous le changez, mais vous savez qu'il est là !*

Stocker des Valeurs Persistantes

```
function Chronometre() {
  const [temps, setTemps] = useState(0);
  const intervalRef = useRef(null);
  const nombreRendusRef = useRef(0);
```

```
  // Compte les rendus sans causer de re-render
  nombreRendusRef.current += 1;

  const demarrer = () => {
    if (!intervalRef.current) {
      intervalRef.current = setInterval(() => {
        setTemps(t => t + 1);
      }, 1000);
    }
  };

  const arreter = () => {
    clearInterval(intervalRef.current);
    intervalRef.current = null;
  };

  const reinitialiser = () => {
    arreter();
    setTemps(0);
  };

  return (
    <div>
      <h2>Temps : {temps}s</h2>
      <p>Rendus : {nombreRendusRef.current}</p>
      <button onClick={demarrer}>Démarrer</button>
      <button onClick={arreter}>Arrêter</button>
      <button onClick={reinitialiser}>Réinitialiser</button>
    </div>
  );
}
```

Vidéo Player avec useRef

```
function VideoPlayer({ src }) {
  const videoRef = useRef(null);
  const [estEnLecture, setEstEnLecture] = useState(false);

  const toggleLecture = () => {
    if (estEnLecture) {
      videoRef.current.pause();
    } else {
      videoRef.current.play();
    }
    setEstEnLecture(!estEnLecture);
  };

  const changerVolume = (e) => {
    videoRef.current.volume = e.target.value / 100;
  };

  return (
    <div>
      <video ref={videoRef} src={src} width="400" />
      <div>
        <button onClick={toggleLecture}>
          {estEnLecture ? '⏸️ Pause' : '▶️ Lecture'}
        </button>
        <input
          type="range"
          min="0"
          max="100"
          onChange={changerVolume}
        />
      </div>
    </div>
  );
}
```

✨ **Utilisez useRef quand vous avez besoin d'accéder directement au DOM ou de stocker une valeur sans déclencher de re-render !**

13. useMemo et useCallback

Ces hooks d'optimisation empêchent les calculs coûteux et la re-création inutile de fonctions. C'est comme mettre votre code sous stéroïdes de performance !

useMemo : Mémoriser des Calculs

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

function ListeFiltre() {
  const [recherche, setRecherche] = useState('');
  const [compteur, setCompteur] = useState(0);

  const elements = Array.from({ length: 10000 }, (_, i) => ({
    id: i,
    nom: `Element ${i}`
  }));

  // Sans useMemo : recalculé à CHAQUE rendu
  // const elementsFiltre = elements.filter(e =>
  //   e.nom.toLowerCase().includes(recherche.toLowerCase())
  // );

  // Avec useMemo : recalculé seulement si recherche change
  const elementsFiltres = useMemo(() => {
    console.log("Filtrage des éléments...");
    return elements.filter(e =>
      e.nom.toLowerCase().includes(recherche.toLowerCase())
    );
  }, [recherche]); // Dépendance

  return (
    <div>
      <input
        value={recherche}
        onChange={(e) => setRecherche(e.target.value)}
        placeholder="Rechercher..."
      />

      {/* Ce bouton ne devrait pas déclencher le filtrage */}
      <button onClick={() => setCompteur(c => c + 1)}>
```

```
        Compteur : {compteur}
      </button>

      <ul>
        {elementsFiltres.slice(0, 100).map(e => (
          <li key={e.id}>{e.nom}</li>
        ))}
      </ul>
    </div>
  );
}
```

💡 useMemo, c'est comme avoir une mémoire photographique : pourquoi recalculer ce qu'on a déjà vu ?

useCallback : Mémoriser des Fonctions

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

function TodoList() {
  const [todos, setTodos] = useState([]);
  const [input, setInput] = useState('');

  // Sans useCallback : nouvelle fonction à chaque rendu
  // const ajouterTodo = () => {
  //   setTodos([...todos, input]);
  //   setInput('');
  // };

  // Avec useCallback : même fonction si deps ne changent pas
  const ajouterTodo = useCallback(() => {
    setTodos(prev => [...prev, input]);
    setInput('');
  }, [input]);

  // Utile pour éviter re-renders d'enfants
  const supprimerTodo = useCallback((index) => {
    setTodos(prev => prev.filter((_, i) => i !== index));
  }, []);

  return (
    <div>
      <input
        value={input}
        onChange={(e) => setInput(e.target.value)}
      />
      <button onClick={ajouterTodo}>Ajouter</button>

      <TodoItem
        todos={todos}
        onSupprimer={supprimerTodo}
      />
    </div>
  );
}

// Composant mémorisé
const TodoItem = React.memo(({ todos, onSupprimer }) => {
```

```
  console.log("TodoItem rendu");
  return (
    <ul>
      {todos.map((todo, i) => (
        <li key={i}>
          {todo}
          <button onClick={() => onSupprimer(i)}>✗</button>
        </li>
      ))}
    </ul>
  );
});
```

✨ **Utilisez useMemo pour des calculs coûteux et useCallback pour des fonctions passées en props à des composants mémorisés !**

14. Custom Hooks

Les Custom Hooks vous permettent d'extraire la logique réutilisable. C'est comme créer vos propres outils magiques !

Hook de LocalStorage

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

function useLocalStorage(cle, valeurInitiale) {
  const [valeur, setValeur] = useState(() => {
    try {
      const item = window.localStorage.getItem(cle);
      return item ? JSON.parse(item) : valeurInitiale;
    } catch (error) {
      console.error(error);
      return valeurInitiale;
    }
  });

  useEffect(() => {
    try {
      window.localStorage.setItem(cle, JSON.stringify(valeur));
    } catch (error) {
      console.error(error);
    }
  }, [cle, valeur]);

  return [valeur, setValeur];
}

// Utilisation
function App() {
  const [nom, setNom] = useLocalStorage('nom', '');
  const [age, setAge] = useLocalStorage('age', 0);

  return (
    <div>
      <input
        value={nom}
        onChange={(e) => setNom(e.target.value)}
        placeholder="Nom"
```

```
      />
      <input
        type="number"
        value={age}
        onChange={(e) => setAge(Number(e.target.value))}
      />
      <p>Nom : {nom}, Âge : {age}</p>
    </div>
  );
}
```

> 💡 *Un Custom Hook, c'est comme inventer un nouveau mot : une fois créé, tout le monde peut l'utiliser !*

Hook de Fetch

```
function useFetch(url) {
  const [data, setData] = useState(null);
  const [loading, setLoading] = useState(true);
  const [error, setError] = useState(null);

  useEffect(() => {
    const fetchData = async () => {
      try {
        setLoading(true);
        const response = await fetch(url);
        const json = await response.json();
        setData(json);
        setError(null);
      } catch (err) {
        setError(err.message);
      } finally {
        setLoading(false);
      }
    };

    fetchData();
  }, [url]);

  return { data, loading, error };
}

// Utilisation
function ListeUtilisateurs() {
  const { data, loading, error } = useFetch(
    'https://jsonplaceholder.typicode.com/users'
  );

  if (loading) return <p>Chargement...</p>;
  if (error) return <p>Erreur : {error}</p>;

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

```
  );
}
```

Hook de Compteur Avancé

```
function useCompteur(valeurInitiale = 0, pas = 1) {
  const [count, setCount] = useState(valeurInitiale);

  const incrementer = () => setCount(c => c + pas);
  const decrementer = () => setCount(c => c - pas);
  const reinitialiser = () => setCount(valeurInitiale);
  const definir = (valeur) => setCount(valeur);

  return {
    count,
    incrementer,
    decrementer,
    reinitialiser,
    definir
  };
}

// Utilisation
function App() {
  const compteur1 = useCompteur(0, 1);
  const compteur2 = useCompteur(100, 10);

  return (
    <div>
      <div>
        <h3>Compteur 1 : {compteur1.count}</h3>
        <button onClick={compteur1.incrementer}>+1</button>
        <button onClick={compteur1.decrementer}>-1</button>
      </div>

      <div>
        <h3>Compteur 2 : {compteur2.count}</h3>
        <button onClick={compteur2.incrementer}>+10</button>
        <button onClick={compteur2.decrementer}>-10</button>
      </div>
    </div>
  );
```

```
}
```

15. Projet Final : Todo App Complète

Mettons tout ensemble dans une application complète avec toutes les fonctionnalités que nous avons apprises !

```
import React, { useState, useEffect, useCallback } from 'react';

// Custom Hook pour localStorage
function useLocalStorage(key, initialValue) {
  const [value, setValue] = useState(() => {
    const stored = localStorage.getItem(key);
    return stored ? JSON.parse(stored) : initialValue;
  });

  useEffect(() => {
    localStorage.setItem(key, JSON.stringify(value));
  }, [key, value]);

  return [value, setValue];
}

function TodoApp() {
  const [todos, setTodos] = useLocalStorage('todos', []);
  const [input, setInput] = useState('');
  const [filtre, setFiltre] = useState('tous');
  const [recherche, setRecherche] = useState('');

  // Ajouter une tâche
  const ajouterTodo = useCallback(() => {
    if (input.trim()) {
      const nouvelleTache = {
        id: Date.now(),
        texte: input,
        termine: false,
        date: new Date().toISOString()
      };
      setTodos(prev => [...prev, nouvelleTache]);
      setInput('');
    }
  }, [input, setTodos]);

  // Basculer l'état terminé
```

```
  const toggleTodo = useCallback((id) => {
    setTodos(prev => prev.map(todo =>
      todo.id === id ? { ...todo, termine: !todo.termine } : todo
    ));
  }, [setTodos]);

  // Supprimer une tâche
  const supprimerTodo = useCallback((id) => {
    setTodos(prev => prev.filter(todo => todo.id !== id));
  }, [setTodos]);

  // Modifier une tâche
  const modifierTodo = useCallback((id, nouveauTexte) => {
    setTodos(prev => prev.map(todo =>
      todo.id === id ? { ...todo, texte: nouveauTexte } : todo
    ));
  }, [setTodos]);

  // Filtrer les tâches
  const todosFiltres = todos
    .filter(todo => {
      if (filtre === 'actifs') return !todo.termine;
      if (filtre === 'termines') return todo.termine;
      return true;
    })
    .filter(todo =>
      todo.texte.toLowerCase().includes(recherche.toLowerCase())
    );

  // Statistiques
  const stats = {
    total: todos.length,
    actifs: todos.filter(t => !t.termine).length,
    termines: todos.filter(t => t.termine).length
  };

  return (
    <div className="todo-app">
      <h1>📝 Ma Todo App</h1>

      {/* Statistiques */}
      <div className="stats">
        <span>Total : {stats.total}</span>
```

```
        <span>Actifs : {stats.actifs}</span>
        <span>Terminés : {stats.termines}</span>
      </div>

      {/* Ajouter une tâche */}
      <div className="add-todo">
        <input
          value={input}
          onChange={(e) => setInput(e.target.value)}
          onKeyPress={(e) => e.key === 'Enter' && ajouterTodo()}
          placeholder="Nouvelle tâche..."
        />
        <button onClick={ajouterTodo}>Ajouter</button>
      </div>

      {/* Recherche et filtres */}
      <div className="controls">
        <input
          value={recherche}
          onChange={(e) => setRecherche(e.target.value)}
          placeholder="Rechercher..."
        />

        <select value={filtre} onChange={(e) => setFiltre(e.target.value)}>
          <option value="tous">Tous</option>
          <option value="actifs">Actifs</option>
          <option value="termines">Terminés</option>
        </select>
      </div>

      {/* Liste des tâches */}
      <ul className="todo-list">
        {todosFiltres.map(todo => (
          <TodoItem
            key={todo.id}
            todo={todo}
            onToggle={toggleTodo}
            onDelete={supprimerTodo}
            onEdit={modifierTodo}
          />
        ))}
      </ul>
```

```
      {todosFiltres.length === 0 && (
        <p className="empty">Aucune tâche trouvée</p>
      )}
    </div>
  );
}

// Composant TodoItem
function TodoItem({ todo, onToggle, onDelete, onEdit }) {
  const [edition, setEdition] = useState(false);
  const [texte, setTexte] = useState(todo.texte);

  const sauvegarder = () => {
    if (texte.trim()) {
      onEdit(todo.id, texte);
      setEdition(false);
    }
  };

  return (
    <li className={`todo-item ${todo.termine ? 'termine' : ''}`}>
      <input
        type="checkbox"
        checked={todo.termine}
        onChange={() => onToggle(todo.id)}
      />

      {edition ? (
        <input
          value={texte}
          onChange={(e) => setTexte(e.target.value)}
          onBlur={sauvegarder}
          onKeyPress={(e) => e.key === 'Enter' && sauvegarder()}
          autoFocus
        />
      ) : (
        <span onClick={() => setEdition(true)}>
          {todo.texte}
        </span>
      )}

      <button onClick={() => onDelete(todo.id)}>🗑️</button>
    </li>
```

```
  );
}

export default TodoApp;
```

💡 Cette Todo App est si complète qu'elle pourrait même gérer votre vie... enfin presque !

Styles CSS pour le Projet

```
.todo-app {
  max-width: 600px;
  margin: 50px auto;
  padding: 20px;
  background: white;
  border-radius: 10px;
  box-shadow: 0 2px 10px rgba(0,0,0,0.1);
}

h1 {
  text-align: center;
  color: #3498db;
}

.stats {
  display: flex;
  justify-content: space-around;
  margin: 20px 0;
  padding: 15px;
  background: #ecf0f1;
  border-radius: 5px;
}

.add-todo {
  display: flex;
  gap: 10px;
  margin: 20px 0;
}

.add-todo input {
  flex: 1;
  padding: 10px;
  border: 2px solid #3498db;
  border-radius: 5px;
}

button {
  padding: 10px 20px;
  background: #3498db;
  color: white;
  border: none;
  border-radius: 5px;
```

```
  cursor: pointer;
}

button:hover {
  background: #2980b9;
}

.controls {
  display: flex;
  gap: 10px;
  margin: 20px 0;
}

.todo-list {
  list-style: none;
  padding: 0;
}

.todo-item {
  display: flex;
  align-items: center;
  gap: 10px;
  padding: 15px;
  margin: 10px 0;
  background: #f8f9fa;
  border-radius: 5px;
  transition: all 0.3s;
}

.todo-item:hover {
  background: #e9ecef;
  transform: translateX(5px);
}

.todo-item.termine span {
  text-decoration: line-through;
  color: #95a5a6;
}

.empty {
  text-align: center;
  color: #95a5a6;
  margin: 40px 0;
}
```

🎓 Conclusion

Félicitations ! Vous êtes maintenant un développeur React compétent !

Vous avez parcouru un long chemin depuis votre premier composant. Voici ce que vous maîtrisez maintenant :

- Les composants et le JSX
- Les Props et le passage de données
- useState et la gestion d'état
- useEffect et les effets de bord
- La gestion d'événements et de formulaires
- useContext pour le partage de données
- useRef pour les références DOM
- useMemo et useCallback pour l'optimisation
- La création de Custom Hooks

Prochaines Étapes

Continuez votre apprentissage avec ces sujets avancés :

- **React Router :** Pour créer des applications multi-pages
- **Redux ou Zustand :** Pour une gestion d'état avancée
- **Next.js :** Framework React pour le rendu côté serveur
- **React Query :** Pour la gestion des données asynchrones
- **Testing avec Jest et React Testing Library :** Pour des applications robustes

> 💡 *Rappelez-vous : même les meilleurs développeurs React ont commencé par un simple 'Hello World' !*

Bon code et bonne continuation ! 🚀

🎉 Bravo !

Vous avez terminé la Partie 2 ! Vous maîtrisez maintenant useEffect, la gestion d'événements, le rendu conditionnel, les listes et les formulaires. Dans la Partie 3, nous explorerons des concepts avancés et des patterns professionnels !

Fin de la Partie 2