

Contenu Théorique - Formation React

Notes rapides pour le formateur. À utiliser en complément des slides PDF.

1. Introduction à React

Historique

Année	Événement
2011	Jordan Walke crée FaxJS chez Facebook
2013	React open-sourcé (JSConf US)
2015	React Native annoncé
2016	React 15 - Réécriture du core
2019	React 16.8 - Les Hooks arrivent
2022	React 18 - Concurrent Mode, Suspense
2024	React 19 - Server Components stable

Pourquoi React ?

- **Déclaratif** : Tu décris CE QUE tu veux, pas COMMENT le faire
- **Composable** : Petits blocs réutilisables (LEGO)
- **Learn once, write anywhere** : Web, Mobile (Native), Desktop (Electron)
- **Écosystème massif** : 20M+ téléchargements/semaine sur npm

La Philosophie

```
UI = f(state)
```

L'interface est une **fonction pure** de l'état. Quand l'état change, React recalcule l'UI.

2. Virtual DOM (simplifié)

Le Problème

Manipuler le DOM réel est **lent** (reflow, repaint).

La Solution React

État change → Nouveau Virtual DOM → Diff avec l'ancien → Patch minimal sur le vrai DOM

Analogie

"Imagine que tu corriges un document Word. Tu ne réimprimes pas tout le document pour changer un mot. Tu changes juste le mot."

React fait pareil : il calcule la différence et applique uniquement les changements nécessaires.

3. JSX - Les 5 Règles d'Or

#	Règle	Exemple
1	Un seul élément parent	<code><div>...</div></code> ou <code><>...</></code>
2	Attributs en camelCase	<code>className, onClick, htmlFor</code>
3	JS avec accolades	<code>{variable}, {2 + 2}</code>
4	Balises auto-fermantes	<code>, <input /></code>
5	Commentaires spéciaux	<code>/* commentaire */</code>

JSX ≠ HTML

```
// JSX                                // HTML équivalent
<div className="box">
  <label htmlFor="name">
    <input type="text" />
  </div>                                <div class="box">
                                            <label for="name">
                                              <input type="text">
                                            </div>
```

4. Composants

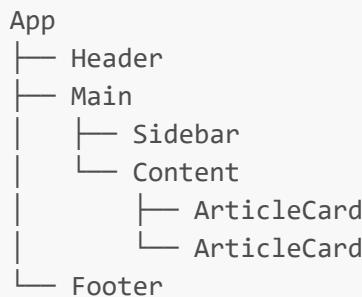
Définition

Un composant = une fonction qui retourne du JSX

Règles

1. Nom en **PascalCase** : `UserCard`, pas `userCard`
2. **Doit retourner** du JSX (ou `null`)
3. **Un fichier = un composant** (convention)

Analogie LEGO



Chaque brique est indépendante et réutilisable.

5. Props

C'est quoi ?

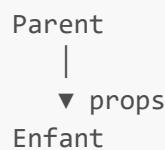
Les props sont les **paramètres** d'un composant. Elles permettent de passer des données du parent vers l'enfant.

Règle d'or

Les props sont en lecture seule (immutables)

Un composant ne doit JAMAIS modifier ses props.

Flux de données



Les données descendent. C'est le **flux unidirectionnel**.

Communication inverse

Pour qu'un enfant communique avec son parent : **callback props**

```
// Parent passe une fonction
<TodoItem onDelete={() => handleDelete(id)} />

// Enfant appelle la fonction
<button onClick={onDelete}>Supprimer</button>
```

6. État (State) avec useState

Props vs State

Props	State
Vient du parent	Appartient au composant
Lecture seule	Modifiable
Données externes	Données internes

Pourquoi pas une simple variable ?

```
// ✗ Ne fonctionne PAS
let count = 0;
function increment() {
  count++; // React ne sait pas qu'il faut re-render
}

// ✓ Fonctionne
const [count, setCount] = useState(0);
function increment() {
  setCount(count + 1); // Déclenche un re-render
}
```

useState est asynchrone

```
setCount(count + 1);
console.log(count); // Affiche l'ancienne valeur !

// Pour être sûr d'avoir la dernière valeur :
setCount(prev => prev + 1);
```

7. Listes et Keys

Pourquoi map() ?

```
const items = ['A', 'B', 'C'];

// On transforme le tableau en tableau de JSX
{items.map(item => <li key={item}>{item}</li>)}
```

Pourquoi key ?

React utilise les keys pour **identifier** chaque élément et optimiser les mises à jour.

Règles des keys

Faire	Ne pas faire
Utiliser un ID unique	Utiliser l'index (sauf liste statique)
key={user.id}	key={index}
Clé stable	Clé aléatoire (<code>Math.random()</code>)

8. Rendu Conditionnel

3 Méthodes

1. Ternaire (A ou B)

```
{isLoggedIn ? <Dashboard /> : <Login />}
```

2. ET logique (A ou rien)

```
{hasNotifications && <Badge count={3} />}
```

3. Variable (cas complexes)

```
let content;
if (loading) content = <Spinner />;
else if (error) content = <Error />;
else content = <Data />;

return <div>{content}</div>;
```

Piège du 0

```
// ✗ Affiche "0" si count === 0
{count && <span>{count} items</span>}

// ✅ Correct
{count > 0 && <span>{count} items</span>}
```

9. useEffect - Cycle de Vie

Les 3 Cas

```
// 1. À chaque render
useEffect(() => {
  console.log('Render !');
});

// 2. Au montage uniquement
useEffect(() => {
  console.log('Monté !');
}, []);

// 3. Quand une dépendance change
useEffect(() => {
  console.log('userId a changé :', userId);
}, [userId]);
```

Cleanup (nettoyage)

```
useEffect(() => {
  const timer = setInterval(() => {...}, 1000);

  // Appelé au démontage ou avant le prochain effet
  return () => clearInterval(timer);
}, []);
```

Cas d'usage courants

- Fetch de données
- Abonnements (WebSocket, events)
- Timers (setTimeout, setInterval)
- Mise à jour du titre de la page

10. Fetch Pattern

Structure Standard

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

useEffect(() => {
  const fetchData = async () => {
    try {
      const res = await fetch(url);
      if (!res.ok) throw new Error('Erreur');
      const json = await res.json();
      setData(json);
    } catch (err) {
      setError(err.message);
    }
  };
  fetchData();
}, []);
```

```

    } catch (err) {
      setError(err.message);
    } finally {
      setLoading(false);
    }
  };

  fetchData();
}, []);

```

Les 3 États de l'UI

Loading → Success (data) ou Error

11. useReducer

Quand l'utiliser ?

- État avec **plusieurs sous-valeurs**
- Logique de mise à jour **complexe**
- Actions **multiples** sur le même état

Pattern

```

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

function reducer(state, action) {
  switch (action.type) {
    case 'INCREMENT':
      return { ...state, count: state.count + 1 };
    case 'SET_NAME':
      return { ...state, name: action.payload };
    default:
      return state;
  }
}

// Utilisation
dispatch({ type: 'INCREMENT' });
dispatch({ type: 'SET_NAME', payload: 'Victor' });

```

useState vs useReducer

useState	useReducer
État simple	État complexe

useState	useReducer
1-2 valeurs	Objet avec plusieurs champs
Logique simple	Logique avec conditions

12. Custom Hooks

Pourquoi ?

Extraire et **réutiliser** de la logique entre composants.

Convention

Toujours préfixer par `use` : `useLocalStorage`, `useFetch`, `useDebounce`

Exemple : `useLocalStorage`

```
function useLocalStorage(key, initial) {
  const [value, setValue] = useState(() => {
    const stored = localStorage.getItem(key);
    return stored ? JSON.parse(stored) : initial;
  });

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

  return [value, setValue];
}
```

13. Context API

Le Problème : Prop Drilling

```
App → Layout → Sidebar → Menu → MenuItem → UserAvatar
                                         ↑
On doit passer "user" à travers 5 niveaux !
```

La Solution : Context

```
// 1. Créer le context
const UserContext = createContext();

// 2. Fournir la valeur (Provider)
<UserContext.Provider value={user}>
```

```

<App />
</UserContext.Provider>

// 3. Consommer (n'importe où dans l'arbre)
const user = useContext(UserContext);

```

Pattern Recommandé : Context + Hook

```

// UserContext.jsx
const UserContext = createContext();

export function UserProvider({ children }) {
  const [user, setUser] = useState(null);
  return (
    <UserContext.Provider value={{ user, setUser }}>
      {children}
    </UserContext.Provider>
  );
}

export function useUser() {
  const context = useContext(UserContext);
  if (!context) throw new Error('useUser must be within UserProvider');
  return context;
}

```

14. Performance (Mémo Rapide)

React.memo

Évite le re-render si les props n'ont pas changé.

```

const ExpensiveComponent = React.memo(function({ data }) {
  // ...
});

```

useCallback

Mémorise une **fonction**.

```

const handleClick = useCallback(() => {
  doSomething(id);
}, [id]);

```

useMemo

Mémorise une **valeur calculée**.

```
const sortedList = useMemo(() => {
  return list.sort((a, b) => a.name.localeCompare(b.name));
}, [list]);
```

Règle d'or

N'optimise pas prématulement. Utilise ces hooks seulement si tu as un problème de performance mesuré.

Aide-Mémoire Final

```
Composant = fonction → JSX  
Props = paramètres (lecture seule, descendant)  
State = données internes (useState, useReducer)  
useEffect = effets de bord (fetch, timers, subscriptions)  
Context = état global sans prop drilling  
Custom Hook = logique réutilisable (préfixe use)
```

Questions Fréquentes des Stagiaires

Q: Pourquoi React et pas Vue/Angular ?

React est une bibliothèque (pas un framework). Plus de liberté, mais plus de choix à faire. Écosystème le plus large.

Q: C'est quoi le Virtual DOM ?

Une copie légère du DOM en mémoire. React compare l'ancien et le nouveau, puis applique seulement les différences.

Q: useState ou useReducer ?

useState pour état simple (1-2 valeurs). useReducer pour état complexe avec plusieurs actions.

Q: Quand utiliser Context ?

Pour des données "globales" : thème, utilisateur connecté, langue. Pas pour tout l'état de l'app.

Q: Pourquoi mes useEffect tournent en boucle infinie ?

Tu as oublié le tableau de dépendances, ou tu modifies une dépendance dans l'effet lui-même.