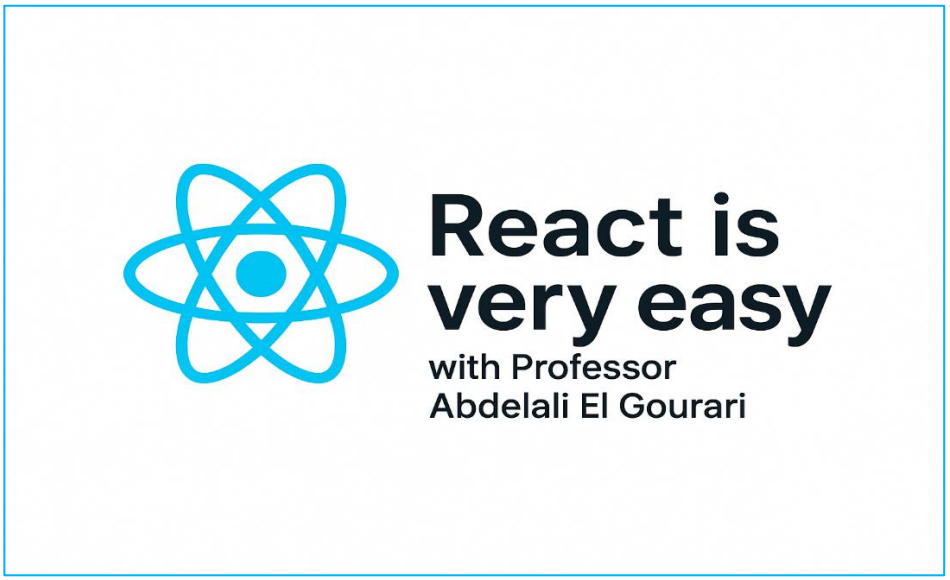
REACT.JS



Abdelali El Gourari holds a doctor of computer science degree, specializing in artificial intelligence (AI) and embedded systems at Cadi Ayyad University, Morocco. His research and professional activities focus on computer science, particularly in the areas of AI and embedded systems. Dr. El Gourari has published numerous computer science publications. His work includes research on modeling collaborative practical work processes in e-learning for electrical engineering education and exploring the application of virtual reality and augmented reality in education.

E-mail: a.elgourari@emsi.ma.

STRUCTURE GÉNÉRALE DU COURS

MODULE 1 : Introduction au Cours

- Présentation des objectifs pédagogiques
- Prérequis techniques
- Environnement de développement
- Méthodologie d'apprentissage

MODULE 2 : Introduction à React.js

Objectifs Pédagogiques

- Comprendre l'écosystème React
- Créer une première application
- Différencier bibliothèque vs framework

1. **Histoire et Contexte**

   ✓ Évolution des frameworks front-end (2006-2024)
   ✓ Positionnement de React dans l'écosystème

2. **Concepts Fondamentaux**

   ✓ Virtual DOM vs Real DOM
   ✓ Architecture composants
   ✓ Philosophie "UI = f(état)"

3. **Mise en Pratique**

   ✓ Installation et configuration
   ✓ Create React App
   ✓ Premier composant fonctionnel

Exercices Pratiques

- Composant `Etudiant` avec props basiques

- Composant `Message` avec date dynamique

MODULE 3 : JSX et Composants

Objectifs

- Maîtriser la syntaxe JSX
- Composer des interfaces modulaires
- Manipuler les props efficacement

1. **JSX Avancé**

✓ Règles de syntaxe
✓ Expressions JavaScript
✓ Rendering conditionnel

2. **Architecture Composants**

✓ Composants conteneurs vs présentationnels
✓ Arborescence d'application
✓ Réutilisabilité et maintenabilité

Exercices Pratiques

- Système de cartes de profils
- Liste de cours dynamique
- Formulaire de contact

MODULE 4 : State et Hooks Fondamentaux

Objectifs

- Gérer l'état local avec useState
- Maîtriser les effets avec useEffect
- Implémenter des interfaces interactives

1. **State Management**

- ✓ Immutabilité des données
- ✓ Mises à jour asynchrones
- ✓ Optimisation des re-rendus

2. **Hooks Essentiels**

- ✓ useState pour données simples
- ✓ useEffect pour effets secondaires
- ✓ Règles des Hooks

Exercices Pratiques

- Compteur intelligent avec pause/reprise
- Horloge temps réel
- Todo list interactive

MODULE 5 : Listes et Formulaires Avancés

Objectifs

- Gérer des collections dynamiques
- Créer des formulaires complexes
- Valider et transformer les données

1. **Gestion des Listes**

- ✓ Clés uniques et performances
- ✓ Operations CRUD sur tableaux
- ✓ Pagination virtuelle

2. **Formulaires Contrôlés**

- ✓ Champs complexes (select, checkbox, file)
- ✓ Validation en temps réel
- ✓ Gestion d'erreurs

Exercices Pratiques

- ✓ Système de gestion d'étudiants
- ✓ Formulaire d'inscription multi-étapes
- ✓ Recherche et filtrage dynamique

MODULE 6 : Routing et Navigation

Objectifs

- Implémenter la navigation SPA
- Gérer les routes dynamiques
- Sécuriser l'accès aux pages

1. **React Router v6+**

- ✓ Configuration avancée
- ✓ Routes imbriquées
- ✓ Navigation programmatique

2. **Patterns Avancés**

- ✓ Lazy loading des routes
- ✓ Gardes d'authentification
- ✓ Gestion des erreurs 404

Exercices Pratiques

- ✓ Application multi-pages complète
- ✓ Système d'authentification simulé
- ✓ Panneau d'administration protégé

MODULE 7 : State Management Global

Objectifs

- Choisir entre Context API et Redux
- Centraliser l'état de l'application

- Debugger efficacement l'état global

1. **Context API**

- ✓ Providers et Consumers
- ✓ Optimisation des performances
- ✓ Cas d'usage recommandés

2. **Redux Toolkit**

- ✓ Store configuration
- ✓ Slices et reducers
- ✓ Async actions avec Thunk

Exercices Pratiques

- ✓ Thème clair/sombre global
- ✓ Panier d'achat e-commerce
- ✓ Système de notifications

MODULE 8 : Stylisation Professionnelle

Objectifs

- Choisir la bonne méthodologie de styling
- Créer des interfaces responsives
- Implémenter un design system

1. **Méthodologies Comparées**

- ✓ CSS Modules vs Styled Components
- ✓ Utility-First avec Tailwind
- ✓ Component Libraries (MUI)

2. **Design Avancé**

- ✓ Thémisation dynamique

- ✓ Animations et transitions
- ✓ Accessibilité (a1ly)

Exercices Pratiques

- Dashboard responsive complet
- Système de thèmes personnalisables
- Composants animés interactifs

MODULE 9 : APIs et Data Fetching

Objectifs

- Consommer des APIs REST/GraphQL
- Gérer les états asynchrones
- Implémenter des mutations

1. **Data Fetching Moderne**

- ✓ SWR/React Query
- ✓ Gestion du cache
- ✓ Optimistic updates

2. **Architecture Data**

- ✓ Normalisation des données
- ✓ Gestion d'erreurs robuste
- ✓ Loading states avancés

Exercices Pratiques

- ✓ Application blog complète
- ✓ Dashboard de métriques en temps réel
- ✓ Système de commentaires

MODULE 10 : Performance et Optimisation

Objectifs

- Analyser les performances
- Optimiser le bundle
- Implémenter le lazy loading

1. **Optimisation Avancée**

- ✓ Memoization avec useMemo/useCallback
- ✓ Code splitting dynamique
- ✓ Virtualisation de listes

2. **Outils Professionnels**

- ✓ React DevTools
- ✓ Bundle analyzers
- ✓ Performance monitoring

Exercices Pratiques

- ✓ Optimisation d'applications lentes
- ✓ Mise en place de lazy loading
- ✓ Analyse de bundle et optimisation

MODULE 11 : Déploiement et Production

Objectifs

- Préparer l'application pour la production
- Déployer sur différentes plateformes
- Mettre en place le CI/CD

1. **Build et Déploiement**

- ✓ Configuration de build avancée
- ✓ Variables d'environnement
- ✓ Déploiement sur Vercel/Netlify/AWS

2. **Bonnes Pratiques Production**

- ✓ Error boundaries
- ✓ Monitoring et analytics
- ✓ SEO et meta tags

Exercices Pratiques

- Pipeline de déploiement complet
- Configuration multi-environnements
- Mise en place de monitoring

PROJETS FINAUX INTÉGRATEURS

Projet 1 : Application de Gestion Scolaire

- Gestion des étudiants, cours, notes
- Tableaux de bord interactifs
- Système d'authentification complet

Projet 2 : E-commerce Moderne

- Catalogue produits dynamique
- Panier d'achat persistant
- Processus de commande multi-étapes

Projet 3 : Réseau Social Minimaliste

- Flux d'actualités en temps réel
- Système de messagerie
- Profils utilisateurs personnalisables

Module 1 : Introduction à React.js

Objectifs pédagogiques

À la fin de ce module, l'étudiant sera capable de :

- Expliquer ce qu'est React et à quoi il sert.
- Comprendre la différence entre une bibliothèque et un framework.
- Identifier les avantages de React dans le développement front-end moderne.
- Créer et exécuter une première application React.

1. Qu'est-ce que React ?

React est une **bibliothèque JavaScript** open-source, initialement créée par Facebook en 2013, et aujourd'hui maintenue par une large communauté de développeurs..
Elle permet de **créer des interfaces utilisateurs (UI)** dynamiques et réactives pour les applications web.

Contrairement à des frameworks comme Angular, ou Vue.js, React ne gère **que la partie interface (View)**, ce qui le rend plus **léger, rapide et modulaire**.

Pourquoi utiliser React ?

- **Réutilisation du code** via les *composants*.
- **Mise à jour efficace du DOM (Document Object Model)** grâce au **Virtual DOM**.
- **Grande communauté** et **écosystème riche**.
- **Performance élevée** et **facilité d'intégration** avec d'autres technologies.

Exemple concret : sans React vs avec React

Méthode traditionnelle (JavaScript pur)

```
Html :
<!DOCTYPE html>
<html>
  <body>
    <div id="root"></div>
    <script>
      const root = document.getElementById('root');
      const message = document.createElement('h1');
      message.textContent = "Bonjour, JavaScript pur";
      root.appendChild(message);
    </script>
  </body>
</html>
```

Méthode moderne (React)

```
import React from 'react';

// Importing createRoot from 'react-dom/client'
import ReactDOM from 'react-dom/client';

// Defining a React element
const element = <h1>Bonjour, React</h1>;

// Creating a root to render the React element into the DOM
```

```
const root = ReactDOM.createRoot(document.getElementById('root'));

// Rendering the React element into the root
root.render(element);
```

Ici, React gère le **DOM virtuel**, ce qui optimise le rendu.

2. Histoire et évolution rapide

Année	Outil / Framework	Créateur	Particularité principale
2006	jQuery	John Resig	Simplifie le DOM
2010	AngularJS	Google	MVVM complet
2013	**React**	Facebook	Virtual DOM, composants
2014	Vue.js	Evan You	Simplicité et flexibilité
2016+	Angular (v2+)	Google	Framework complet moderne

React s'est imposé grâce à sa **philosophie simple et performante** :

"**UI = f(état)**" → L'interface est une fonction de l'état.

3. Comment fonctionne React ?

Le Virtual DOM

Le **Virtual DOM** est une **copie légère du DOM réel**.
Quand l'état change, React compare (diffing) les deux arbres DOM et met à jour **uniquement les parties modifiées**.

Exemple visuel simplifié :

```
État initial   →   DOM réel : <h1>Bonjour</h1>
Nouvel état    →   DOM virtuel : <h1>Bonsoir</h1>

React détecte le changement et met à jour seulement le texte.
```

Résultat : **meilleure performance** et **moins de rafraîchissements inutiles**.

4. Installer et exécuter une application React

Prérequis :

- **Node.js** installé sur ton ordinateur.
- **npm** (inclus avec Node).

```
# Vérification de l'installation
node -v
```


npm -v

Création d'une application :

npx create-react-app mon-premier-projet → ??? C:\Users\hp\AppData\Roaming\npm
cd mon-premier-projet
npm start

Structure du projet :

```
mon-premier-projet/
├── src/                          # Code source (composants, App.js, etc.)
│   ├── App.js
│   ├── index.js
│   └── App.css
├── public/                       # Fichiers statiques (index.html)
│   └── index.html
├── package.json                  # Configuration du projet
└── node_modules/                 # Dépendances installées
```

1. Le dossier src/ — le cœur du projet

C'est ici que tu écris **tout ton code React** : les composants, le style, la logique de ton application.

App.js

- C'est **le composant principal** de ton application React.
- Il définit ce que l'utilisateur voit à l'écran (la structure de ta page).
- Tous les autres composants seront **importés ici**.

Exemple :

```
function App() {
  return (
    <div>
      <h1>Bienvenue sur mon premier projet React </h1>
    </div>
  );
}
export default App;
```

App est un **composant fonctionnel** : il retourne du JSX (du HTML dans du JavaScript).
C'est **le point de départ visuel** de ton application.

index.js

- C'est le fichier d'entrée de ton application React.
- Il connecte ton application React à la page HTML du navigateur.
- Il dit à React : "Affiche le composant <App /> dans la page web !"

Exemple :


```
import React from "react";
import ReactDOM from "react-dom/client";
import App from "./App";
import "./App.css";

const root = ReactDOM.createRoot(document.getElementById("root"));
root.render(<App />);
```

ReactDOM.createRoot() dit à React **où afficher** ton application (dans le du HTML).

C'est ici que ton projet **démarre réellement**.

App.css

- Fichier de style CSS associé à ton composant App.js.
- Tu peux y écrire toutes les règles CSS (couleurs, marges, tailles, etc.).

Exemple :

```
body {
  background-color: #f4f6f8;
  font-family: Arial, sans-serif;
  text-align: center;
}
h1 {
  color: #007bff;
}
```

Chaque composant peut avoir son **propre fichier CSS** pour mieux organiser le style.

2. Le dossier public/ — les fichiers visibles par le navigateur

Ce dossier contient les fichiers **statiques** (images, HTML, favicon, etc.).

index.html

- C'est le seul fichier HTML de ton projet React.
- React ne crée pas plusieurs pages HTML : tout le contenu est inséré ici dynamiquement.
- Le navigateur charge ce fichier au démarrage.
- À l'intérieur, on trouve : `<div id="root"></div>`

C'est **dans ce "div"** que React va afficher ton application (App.js).

Tu peux modifier le titre de ta page ici : `<title>Mon Premier Projet React</title>`

3. package.json — le cerveau du projet

Ce fichier contient toutes les **informations et dépendances** de ton projet.

Exemple simplifié :

```
{
  "name": "mon-premier-projet",
  "version": "1.0.0",
  "dependencies": {
    "react": "^18.0.0",
    "react-dom": "^18.0.0"
```



```
  },
  "scripts": {
    "start": "react-scripts start",
    "build": "react-scripts build"
  }
}
```

Ce fichier est **créé automatiquement** par npm.

Il liste :

- les bibliothèques utilisées (React, React Router, etc.)
- les scripts de commandes :
  - ✓ npm start → Démarre le serveur de développement
  - ✓ npm run build → Crée la version finale du site

4. node_modules/ — la boîte à outils

C'est le dossier où **npm installe toutes les bibliothèques nécessaires** à ton projet.

Très important :

- Ce dossier est très lourd (plusieurs centaines de Mo).
- Ne le modifie jamais manuellement.
- Il est recréé automatiquement quand tu exécutes : npm install

Exercices Pratiques

Exercice 1.1 : Premier Composant

Créez un composant `Etudiant` affichant :

- Nom de l'étudiant
- Âge
- Filière

Résultat attendu :

Nom : Chaima
Âge : 21 ans
Filière : Informatique

Exercice 1.2 : Composant Message

Créez un composant `Message` qui :

- Affiche "Bonjour le monde "
- Affiche la date actuelle automatiquement

Sloution : Exercice 1.1

Jsx: Version 1 : Date fixe

```
// src/components/Etudiant.js
import React from "react";
function Etudiant() {
  const nom = "Abdelali";
```



```
  const age = 29;
  const filiere = "Informatique";
  return (
    <div>
      <h2>Informations de l'étudiant :</h2>
      <p><strong>Nom :</strong> {nom}</p>
      <p><strong>Âge :</strong> {age} ans</p>
      <p><strong>Filière :</strong> {filiere}</p>
    </div>
  );
}
export default Etudiant;
```

Explication

- Le composant Etudiant est une **fonction JavaScript** qui retourne du **JSX**.
- Les variables nom, age et filiere sont insérées dans le JSX à l'aide des **accolades { }**.
- Chaque donnée est affichée dans une balise <p> avec un style simple.

Sloution : Exercice 1.2

jsx: Version 2 : Horloge en temps réel

```
// src/components/Message.js
import React from "react";
function Message() {
  const dateActuelle = new Date().toLocaleString();
  return (
    <div>
      <h2>Bonjour le monde </h2>
      <p>Date actuelle : {dateActuelle}</p>
    </div>
  );
}
export default Message;
```

Explication

- new Date().toLocaleString() génère la **date et l'heure** selon la langue et le format local.
- Le composant s'actualise à chaque rechargement de la page.
- Si on voulait qu'elle **se mette à jour automatiquement en temps réel**, on pourrait utiliser useEffect + s etInterval.

```
// src/components/Message.js
import React, { useState, useEffect } from "react";
function Message() {
  const [date, setDate] = useState(new Date());
  useEffect(() => {
    const timer = setInterval(() => setDate(new Date()), 1000);
    return () => clearInterval(timer);
  }, []);
  return (
    <div>
      <h2>Bonjour le monde </h2>
      <p>Date actuelle : {date.toLocaleString()}</p>
    </div>
  );
}
export default Message;
```

- **Version 1** : Parfaite pour comprendre les bases (variables, JSX)
- **Version 2** : Idéale pour introduire les Hooks (useState, useEffect)

Résumé

Concept	Description
React	Bibliothèque pour interfaces utilisateurs
Virtual DOM	Copie optimisée du DOM pour performance
Composant	Fonction retournant du JSX
JSX	Syntaxe combinant JavaScript et HTML

MODULE 2 : JSX ET COMPOSANTS

Objectifs Pédagogiques

À la fin de ce module, l'étudiant sera capable de :

- Maîtriser la syntaxe JSX et ses règles
- Créer et composer des composants fonctionnels
- Utiliser les **props** pour passer des données
- Structurer une application modulaire

2.1 JSX (JavaScript XML)

Extension de syntaxe permettant d'écrire du HTML dans JavaScript.

Règles importantes :

- Un seul élément parent
- Fermeture obligatoire des balises
- Attributs en camelCase
- Les accolades `{}` pour le JavaScript

```
// Incorrect : multiple éléments
return (
  <h1>Titre</h1>
  <p>Paragraphe</p>
);
// Correct : un seul parent
```



```
return (
  <div>
    <h1>Titre</h1>
    <p>Paragraphe</p>
  </div>
);
```

2.2 Composants Fonctionnels

Fonction JavaScript qui retourne du JSX.

```
// Composant simple
function Bienvenue() {
  return <h1>Bienvenue sur mon site </h1>;
}
// Composant avec arrow function
const Bienvenue = () => {
  return <h1>Bienvenue sur mon site </h1>;
};
```

→ *Tu peux taper rfc* → *Enter*

2.3 Les Props (Propriétés)

Définition : Mécanisme pour passer des données du parent vers l'enfant.

```
// Composant enfant
function Bonjour(props) {
  return <h2>Bonjour, {props.nom} </h2>;
}
// Composant parent
function App() {
  return (
    <div>
      <Bonjour nom="Abdelali" />
      <Bonjour nom="Oualid" />
      <Bonjour nom="Yassine" />
    </div>
  );
}
```

Déstructuration des props (moderne) :

```
function Etudiant({ nom, age, filiere }) {
```



```
  return (
    <div>
      <h3>{nom}</h3>
      <p>{age} ans - {filiere}</p>
    </div>
  );
}
```

2.4 Composition de Composants

```
function Header() {
  return <header>En-tête du site</header>;
}
function Footer() {
  return <footer>Pied de page © 2024</footer>;
}
function App() {
  return (
    <div>
      <Header />
      <main>Contenu principal</main>
      <Footer />
    </div>
  );
}
```

Exercices Pratiques

Exercice 2.1 : Carte d'Étudiant

Créez un composant **CarteEtudiant** affichant :

- Nom, âge, filière et moyenne
- Phrase : "{nom} a une moyenne de {moyenne}/20"

Exercice 2.2 : Liste de Cours

Créez un composant **Cours** qui reçoit en props :

- ✓ titre, professeur, nombreHeures.
- ✓ Affichez une liste de 3 cours.

Solution : Exercice 2.1 — Carte d'Étudiant

Objectif :

Créer un composant `CarteEtudiant` qui affiche :

- Nom
- Âge
- Filière
- Moyenne

Et une phrase : "{nom} a une moyenne de {moyenne}/20"

```
// src/components/CarteEtudiant.js
import React from "react";
function CarteEtudiant({ nom, age, filiere, moyenne }) {
  return (
    <div
      style={{
        border: "2px solid #007bff",
        borderRadius: "10px",
        padding: "15px",
        width: "250px",
        margin: "10px",
        backgroundColor: "#f8f9fa",
      }}
    >
      <h3>Étudiant : {nom}</h3>
      <p>Âge : {age} ans</p>
      <p>Filière : {filiere}</p>
      <p>
        <strong>{nom}</strong> a une moyenne de <strong>{moyenne}/20</strong>
      </p>
    </div>
  );
}
export default CarteEtudiant;
```

Explication

- Le composant CarteEtudiant reçoit des **props** (nom, age, filiere, moyenne).
- Les accolades {} permettent d'insérer des valeurs JavaScript dans le JSX.
- On applique un petit **style inline** pour rendre la carte plus visuelle.

Utilisation dans App.js


```
// src/App.js
import React from "react";
import CarteEtudiant from "./components/CarteEtudiant";

function App() {
  return (
    <div>
      <h1>Liste des étudiants </h1>
      <CarteEtudiant nom="Chaima" age={21} filiere="Informatique" moyenne={17.5} />
      <CarteEtudiant nom="Oualid" age={22} filiere="Mathématiques" moyenne={14.8} />
      <CarteEtudiant nom="Yassine" age={23} filiere="Physique" moyenne={15.2} />
    </div>
  );
}
export default App;
```

Solution : Exercice 2.2 — Liste de Cours

Objectif :

Créer un composant `Cours` qui reçoit en **props** :

- titre
- professeur
- nombreHeures

Puis l'utiliser dans une liste pour afficher **3 cours**.

```
// src/components/Cours.js
import React from "react";
function Cours({ titre, professeur, nombreHeures }) {
  return (
    <div
      style={{
        border: "1px solid #28a745",
        borderRadius: "10px",
        padding: "10px",
        width: "300px",
        margin: "10px",
        backgroundColor: "#e8f5e9",
      }}
      <h3>{titre}</h3>
      <p>Professeur : {professeur}</p>
      <p>Durée : {nombreHeures} heures</p>
```



```
    </div>
  );
}
export default Cours;
```

Utilisation dans App.js

```
// src/App.js
import React from "react";
import Cours from "./components/Cours";
function App() {
  return (
    <div>
      <h1>Liste des Cours </h1>
      <Cours titre="React.js Avancé" professeur="Abdelali El Gourari" nombreHeures={12} />
      <Cours titre="Python pour la Data Science" professeur="Aïcha B." nombreHeures={10} />
      <Cours titre="Développement Web Full Stack" professeur="Yassine M." nombreHeures={14}
/>
    </div>
  );
}
export default App;
```

Explication

- Chaque composant Cours représente **un seul élément** de la liste.
- Les **props** permettent de réutiliser le même composant pour différents cours.
- L'approche **déclarative** rend le code plus lisible et modulaire.

Résumé

Concept	Description
JSX	Syntaxe HTML dans JavaScript
Composant	Fonction retournant du JSX
Props	Données passées au composant
Composition	Imbrication de composants

MODULE 3 : STATE ET HOOKS FONDAMENTAUX

Objectifs Pédagogiques

À la fin de ce module, l'étudiant sera capable de :

- Comprendre le concept de state et son importance
- Utiliser le Hook useState pour gérer l'état local
- Utiliser useEffect pour les effets secondaires
- Créer des composants interactifs

3.1 Le State (État)

Le **state** représente les **données dynamiques** d'un composant React.
C'est ce qui change au cours du temps (par exemple : un compteur, un texte saisi, une couleur, etc.).
Quand le state change, **le composant se met à jour automatiquement** à l'écran.

Exemple :

```
let [count, setCount] = useState(0);
```

Ici, `count` est le **state**, et `setCount` sert à le **modifier**.

Différence Props vs State :

Aspect	Props	State
Origine	Parent	Composant lui-même
Modifiable	Non	Oui
Rôle	Données entrantes	Données internes

3.2 Le Hook useState

Le **Hook `useState`** permet de **créer et gérer un state** dans un composant fonctionnel.
Il **retourne deux valeurs** :

1. La valeur actuelle du state.
2. Une fonction pour la modifier.

Exemple :

Javascript →

```
const [message, setMessage] = useState("Bonjour !");
```

- `message` → valeur du state
- `setMessage` → fonction pour changer cette valeur

Exemple : Compteur


```
import { useState } from 'react';
function Compteur() {
  const [count, setCount] = useState(0);
  return (
    <div>
      <p>Compteur : {count}</p>
      <button onClick={() => setCount(count + 1)}>
        Incrémenter
      </button>
    </div>
  );
}
```

Exemple : Champ de texte

```
function Formulaire() {
  const [nom, setNom] = useState('');
  return (
    <div>
      <input
        type="text"
        value={nom}
        onChange={(e) => setNom(e.target.value)}
        placeholder="Entrez votre nom"
      />
      <p>Bonjour {nom} !</p>
    </div>
  );
}
```

3.3 Le Hook useEffect

Le **Hook `useEffect`** permet d'exécuter du **code automatiquement** quand le composant :

- est **affiché** pour la première fois,
- ou quand **certaines données changent**.

Il sert souvent pour :

- Charger des données depuis une API
- Mettre à jour le titre de la page
- Gérer un minuteur, etc.

Exemple :


```
useEffect(() => {
  console.log("Composant affiché ou mis à jour !");
}, []);
```

Le deuxième paramètre `[]` indique que le code ne s'exécute **qu'une seule fois** (au chargement du composant).

Syntaxe :

Javascript :

```
useEffect(() => {
  // Code à exécuter
}, [dépendances]);
```

Sur changement de variable :

Jsx :

```
useEffect(() => {
  document.title = `Messages (${messages.length})`;
}, [messages]); // Exécuté quand messages change
```

Ligne 1 : `useEffect(() => {`

- `useEffect` est un hook React qui permet de gérer les "effets de bord"
- Un "effet de bord" = une action qui affecte l'extérieur du composant
- Ici, on veut modifier le titre de la page (qui est en dehors de notre composant)

Ligne 2 : `document.title =`Messages (${messages.length});

- `document.title` : Propriété qui contrôle le titre de l'onglet du navigateur
- `` `Messages (${messages.length})` `` : Template string (chaîne de caractères dynamique)
- `${messages.length}` : Insère le nombre de messages dans la chaîne
- **Exemple :** Si `messages.length` vaut 3 → titre devient "Messages (3)"

Ligne 3 : `}, [messages]);`

- `[messages]` : Tableau de dépendances
- **Important :** Cet effet s'exécute seulement quand la variable `messages` change
- Si `messages` reste identique, l'effet n'est pas relancé

Analogie pour comprendre :

Imaginez un assistant personnel qui :

- **Regarde** votre liste de messages
- **Met à jour** le titre de votre agenda quand le nombre de messages change
- **Ne travaille pas** si la liste ne change pas (pour économiser de l'énergie)

Nettoyage :

Jsx:

```
useEffect(() => {
  const timer = setInterval(() => {
    console.log('Timer executé');
  }, 1000);
  return () => clearInterval(timer); // Nettoyage
}, []);
```

Ligne 1 : `useEffect(() => {`

- On déclare un effet qui va s'exécuter après le rendu du composant

Ligne 2 : `const timer = setInterval(() => {`

- `setInterval` : Fonction JavaScript qui répète une action à intervalle régulier
- `timer` : Variable qui stocke la référence à l'intervalle (comme un numéro de ticket)

Lignes 3-4 : `console.log('Timer executé'); }, 1000);`

- Toutes les 1000ms (1 seconde), exécute `console.log('Timer executé')`
- **Résultat :** "Timer executé" s'affiche dans la console chaque seconde

Ligne 5 : `return () => clearInterval(timer);`

- **Fonction de nettoyage** : S'exécute quand le composant est démonté
- `clearInterval(timer)` : Arrête l'intervalle pour éviter les fuites mémoire
- **Essentiel** pour les performances !

Ligne 6 : `}, []);`

- `[]` : Tableau de dépendances VIDE
- **Signification :** Cet effet s'exécute UNE SEULE FOIS au montage du composant

Analogie pour comprendre :

Imaginez un réveil matin :

- **Montage** : Vous programmez le réveil pour qu'il sonne toutes les heures
- **Nettoyage** : Quand vous quittez la maison, vous éteignez le réveil
- **Sans nettoyage** : Le réveil continuerait à sonner dans une maison vide !

3.4 Exemple Complet :

Jsx:

```
function App() {
  const [messages, setMessages] = useState([]);
  const [compteur, setCompteur] = useState(0);
```



```
  // Effet 1 : Mise à jour du titre
  useEffect(() => {
    document.title = `Messages (${messages.length})`;
    console.log("Titre mis à jour !");
  }, [messages]); // Seulement quand messages change
  // Effet 2 : Timer
  useEffect(() => {
    console.log("Timer démarré !");
    const timer = setInterval(() => {
      console.log('Timer executé');
    }, 1000);
    return () => {
      console.log("Timer arrêté !");
      clearInterval(timer);
    };
  }, []); // Une seule fois
  return (
    <div>
      <button onClick={() => setMessages([...messages, "Nouveau message"])}>
        Ajouter message ({messages.length})
      </button>
      <button onClick={() => setCompteur(compteur + 1)}>
        Compteur: {compteur}
      </button>
    </div>
  );
}
```

Exercices Pratiques

Exercice 3.1 : Compteur Intelligent

Créez un composant **CompteurAuto** qui :

- Démarre à 0
- S'incrémente automatiquement toutes les 2 secondes
- Avoir un bouton pause/reprise

```
// Importation des hooks React nécessaires
import { useState, useEffect } from 'react';
function CompteurAuto() {
  // 1. Déclaration de l'état du compteur - initialisé à 0
  const [compteur, setCompteur] = useState(0);
  // 2. Déclaration de l'état pour savoir si le compteur est en pause
```



```
  const [enPause, setEnPause] = useState(false);
  // 3. useEffect pour gérer l'incrémentation automatique
  useEffect(() => {
    // Si le compteur est en pause, on ne fait rien
    if (enPause) return;
    // Création d'un intervalle qui s'exécute toutes les 2 secondes (2000ms)
    const interval = setInterval(() => {
      // Incrémentation du compteur de 1
      setCompteur(prevCompteur => prevCompteur + 1);
    }, 2000);
    // Nettoyage : cette fonction est appelée quand le composant est démonté
    // ou quand les dépendances changent
    return () => clearInterval(interval);
  }, [enPause]); // Dépendance : l'effet se relance quand enPause change
  // 4. Fonction pour gérer le bouton pause/reprise
  const togglePause = () => {
    setEnPause(!enPause); // Inverse la valeur actuelle (true devient false, etc.)
  };
  // 5. Rendu du composant
  return (
    <div style={{ textAlign: 'center', padding: '20px' }}>
      <h2>Compteur Auto: {compteur}</h2>
      {/* Bouton qui change de texte selon l'état */}
      <button onClick={togglePause}>
        {enPause ? 'Reprendre ' : 'Pause '}
      </button>
      {/* Affichage de l'état actuel */}
      <p>Statut: {enPause ? 'En pause' : 'En cours'}</p>
    </div>
  );
}
export default CompteurAuto;
```

Explications importantes :

useState :

- useState(0) : Crée une variable compteur avec valeur initiale 0
- setCompteur : Fonction pour modifier la valeur de compteur
- Quand setCompteur est appelé, le composant se re-rend

useEffect :

- S'exécute après chaque rendu du composant
- [enPause] : Liste de dépendances - l'effet se relance seulement si enPause change
- return () => clearInterval(interval) : Nettoyage essentiel pour éviter les fuites mémoire

Exercice 3.2 : Horloge Temps Réel

Créez un composant Horloge qui :

- Affiche l'heure actuelle mise à jour chaque seconde
- Permet de basculer entre format 12h/24h

```
// Importation des hooks React nécessaires
import { useState, useEffect } from 'react';
function Horloge() {
  // 1. État pour stocker l'heure actuelle
  const [heure, setHeure] = useState(new Date());
  // 2. État pour le format d'affichage (true = 24h, false = 12h)
  const [format24h, setFormat24h] = useState(true);
  // 3. useEffect pour mettre à jour l'heure chaque seconde
  useEffect(() => {
    // Création d'un intervalle qui s'exécute toutes les secondes (1000ms)
    const interval = setInterval(() => {
      setHeure(new Date()); // Met à jour l'heure avec l'heure actuelle
    }, 1000);
    // Nettoyage de l'intervalle
    return () => clearInterval(interval);
  }, []); // Tableau de dépendances vide = s'exécute une fois au montage
  // 4. Fonction pour formater l'heure selon le format choisi
  const formaterHeure = () => {
    if (format24h) {
      // Format 24h : "HH:MM:SS"
      return heure.toLocaleTimeString('fr-FR'); // Ex: "14:30:25"
    } else {
      // Format 12h : "HH:MM:SS AM/PM"
      return heure.toLocaleTimeString('en-US'); // Ex: "2:30:25 PM"
    }
  };
  // 5. Fonction pour basculer entre les formats
  const toggleFormat = () => {
    setFormat24h(!format24h); // Inverse la valeur actuelle
  };
  // 6. Rendu du composant
  return (
    <div style={{ textAlign: 'center', padding: '20px' }}>
```



```
      <h2>Horloge Temps Réel</h2>
      {/* Affichage de l'heure formatée */}
      <div style={{ fontSize: '2em', margin: '20px 0' }}>
        {formaterHeure()}
      </div>
      {/* Bouton pour changer le format */}
      <button onClick={toggleFormat}>
        Basculer en {format24h ? '12h' : '24h'}
      </button>
      {/* Affichage du format actuel */}
      <p>Format: {format24h ? '24 heures' : '12 heures'}</p>
    </div>
  );
}
export default Horloge;
```

Explications importantes :

Gestion du temps :

- `new Date()` : Crée un objet Date avec l'heure actuelle
- `toLocaleTimeString()` : Formate l'heure selon la locale
- `'fr-FR'` : Format français (24h)
- `'en-US'` : Format américain (12h avec AM/PM)

useEffect avec dépendances vides :

- `[]` : L'effet s'exécute seulement au montage du composant
- Parfait pour les initialisations qui ne doivent se faire qu'une fois

Utilisation dans l'application principale

```
import CompteurAuto from './CompteurAuto';
import Horloge from './Horloge';
function App() {
  return (
    <div>
      <CompteurAuto />
      <Horloge />
    </div>
  );
}
export default App;
```


Points pédagogiques à souligner :

- **État local** : Chaque composant gère son propre état
- **Effets de bord** : `useEffect` pour les opérations asynchrones
- **Événements** : `onClick` pour interagir avec l'interface
- **Rendu conditionnel** : Texte qui change selon l'état
- **Nettoyage** : Importance de `clearInterval` pour les performances

MODULE 4 : LISTES ET FORMULAIRES AVANCÉS

Objectifs Pédagogiques

À la fin de ce module, l'étudiant sera capable de :

- Afficher et manipuler des listes dynamiques
- Comprendre l'importance des clés (keys)
- Créer des formulaires contrôlés complexes
- Valider et gérer les données utilisateur

PARTIE 1 : LISTES DYNAMIQUES

En React, nous travaillons souvent avec des données qui changent. Les listes dynamiques nous permettent d'afficher et de mettre à jour des collections d'éléments de manière efficace.

Exemple :

Jsx :

```
function ListeSimple() {
  const [elements, setElements] = useState(['Pomme', 'Banane', 'Orange']);
  return (
    <ul>
      {elements.map((element, index) => (
        <li key={index}>{element}</li>
      ))}
    </ul>
  );
}
```

À Retenir

- map() transforme un tableau en composants JSX
- Toujours retourner un élément JSX dans le map
- L'état (useState) permet de mettre à jour la liste dynamiquement

PARTIE 2 : CLÉS (KEYS) - LE SECRET DES PERFORMANCES

Les **clés** aident React à identifier quels éléments ont changé, été ajoutés ou supprimés. Elles doivent être **uniques** et **stables**.

Exemple Critique

Jsx :

```jsx
// ❌ MAUVAIS - Problèmes de performances
{users.map((user, index) => (
  <UserComponent key={index} user={user} />
))}
// ✅ BON - Utilisation d'ID uniques
{users.map(user => (
  <UserComponent key={user.id} user={user} />
))}
```

À Retenir

- Les clés doivent être **uniques** et **stables**
- Jamais utiliser l'index comme clé pour des listes modifiables
- Meilleures performances et moins de bugs

PARTIE 3 : FORMULAIRES CONTRÔLÉS

Un formulaire contrôlé signifie que React gère l'état de chaque champ. C'est essentiel pour les formulaires complexes avec validation.

Jsx :

```jsx
function FormulaireControle() {
  const [formData, setFormData] = useState({
    email: '',
    password: ''
  });
  // Gestionnaire universel
  const handleChange = (e) => {
    setFormData({
      ...formData,
      [e.target.name]: e.target.value
    });
  };
  return (
    <form>
      <input
        name="email"
        value={formData.email}
        onChange={handleChange}
```



```
        />
        <input
          name="password"
          value={formData.password}
          onChange={handleChange}
        />
      </form>
    );
  }
```

À Retenir

- Chaque input a value et onChange
- Un seul state pour tout le formulaire
- Spread operator (...) pour préserver les autres champs

PARTIE 4 : VALIDATION DES DONNÉES

La validation assure que les données entrées par l'utilisateur sont correctes avant soumission.

Jsx :

```
  function ValidationSimple() {
    const [email, setEmail] = useState('');
    const [erreur, setErreur] = useState('');
    const validerEmail = (email) => {
      if (!email.includes('@')) {
        setErreur('Email invalide');
        return false;
      }
      setErreur('');
      return true;
    };
    const handleSubmit = (e) => {
      e.preventDefault();
      if (validerEmail(email)) {
        // Soumettre le formulaire
      }
    };
    return (
      <form onSubmit={handleSubmit}>
        <input
          value={email}
          onChange={(e) => setEmail(e.target.value)}
        />
```



```
        {erreur && <p style={{color: 'red'}}>{erreur}</p>}
        <button type="submit">Valider</button>
      </form>
    );
  }
```

SYNTHÈSE DES BONNES PRATIQUES

Pour les Listes

Jsx :

```
// BONNES PRATIQUES
{items.map(item => (
  <Component
    key={item.id}           // ID unique
    data={item}
  />
))}
```

Pour les Formulaires

Jsx :

```
// STRUCTURE OPTIMALE
const [form, setForm] = useState(initialValues);

const onChange = (e) => {
  setForm(prev => ({
    ...prev,
    [e.target.name]: e.target.value
  }));
};
```

CHECKLIST DE VÉRIFICATION

Avant de soumettre votre code, vérifiez :

Listes

- J'utilise map() pour afficher les tableaux
- J'ai ajouté une key unique à chaque élément
- Ma clé n'est pas l'index du tableau

Formulaires

- Mes inputs ont value et onChange
- J'utilise un state contrôlé
- Je gère la soumission avec onSubmit

Validation

- Je valide les données avant soumission
- J'affiche les messages d'erreur
- J'empêche la soumission si données invalides

Exercices Pratiques

Exercice 4.1 : Gestion d'Étudiants

Créez un système complet de gestion d'étudiants :

- Formulaire d'ajout (nom, âge, filière)
- Liste affichée sous forme de tableau
- Possibilité de suppression
- Recherche et filtrage

Exercice 4.2 : Formulaire Multi-étapes

Créez un formulaire d'inscription en 3 étapes :

- Informations personnelles
- Informations académiques
- Confirmation

Résumé

Concept	Description
.map()	Transformation de listes
key	Identifiant unique pour le rendu de liste
Formulaire contrôlé	State React contrôle les valeurs
Validation	Vérification des données utilisateur

MODULE 5 : ROUTING ET NAVIGATION

Objectifs Pédagogiques

À la fin de ce module, l'étudiant sera capable de :

- Configurer React Router pour la navigation
- Créer des routes dynamiques et imbriquées
- Gérer la navigation programmatique
- Implémenter des gardes d'authentification

5.1 Installation et Configuration

Installation : npm install react-router-dom

PARTIE 1 : CONFIGURATION REACT ROUTER

Définir les routes de votre application

Setup Minimal

```
// Installation et configuration de base
import { BrowserRouter, Routes, Route } from 'react-router-dom';

function App() {
  return (
    <BrowserRouter>
      <Routes>
        <Route path="/" element={<Accueil />} />
        <Route path="/apropos" element={<APropos />} />
        <Route path="/contact" element={<Contact />} />
      </Routes>
    </BrowserRouter>
  );
}
```

À Retenir

- BrowserRouter : Conteneur principal
- Routes : Conteneur des routes
- Route : Définit un chemin → composant

PARTIE 2 : ROUTES DYNAMIQUES ET IMBRIQUÉES

Routes avec paramètres et hiérarchie

Routes Dynamiques

```
function App() {
  return (
    <Routes>
      {/* Route avec paramètre */}
      <Route path="/utilisateur/:id" element={<ProfilUtilisateur />} />
      {/* Route optionnelle */}
      <Route path="/produit/:id?" element={<Produit />} />
    </Routes>
  );
}
// Récupération du paramètre dans le composant
function ProfilUtilisateur() {
  const { id } = useParams(); // Hook pour récupérer :id
  return <h1>Profil de l'utilisateur {id}</h1>;
}
```


Routes Imbriquées

```
function App() {
  return (
    <Routes>
      <Route path="/admin" element={<LayoutAdmin />}>
        {/* Routes imbriquées */}
        <Route path="dashboard" element={<Dashboard />} />
        <Route path="utilisateurs" element={<GestionUtilisateurs />} />
      </Route>
    </Routes>
  );
}
// Layout avec outlet pour les sous-routes
function LayoutAdmin() {
  return (
    <div>
      <nav>Menu admin</nav>
      <Outlet /> {/* Ici s'affichent les routes enfants */}
    </div>
  );
}
```

À Retenir

- :param pour les paramètres dynamiques
- useParams() pour récupérer les paramètres
- Outlet pour afficher les routes enfants

PARTIE 3 : NAVIGATION PROGRAMMATIQUE

Changer de route via le code JavaScript

Navigation Essentielle

```
import { useNavigate, Link } from 'react-router-dom';
function ComposantAvecNavigation() {
  const navigate = useNavigate();
  const handleLogin = () => {
    // Navigation après connexion
    navigate('/dashboard');
  };
  const handleGoBack = () => {
    // Retour en arrière
    navigate(-1);
  };
```



```
  return (
    <div>
      {/* Navigation via liens */}
      <Link to="/accueil">Accueil</Link>
      {/* Navigation programmatique */}
      <button onClick={handleLogin}>Se connecter</button>
      <button onClick={handleGoBack}>Retour</button>
    </div>
  );
}
```

À Retenir

- useNavigate() pour navigation dans le code
- Link pour la navigation via liens
- navigate(-1) pour revenir en arrière

PARTIE 4 : GARDES D'AUTHENTIFICATION

Protéger l'accès aux routes

Protection de Route Simple

```
// Composant de protection
function RouteProtegee({ enfants }) {
  const estConnecte = useSelector(state => state.auth.estConnecte);
  if (!estConnecte) {
    return <Navigate to="/connexion" replace />;
  }
  return enfants;
}
// Utilisation
function App() {
  return (
    <Routes>
      <Route path="/connexion" element={<Connexion />} />
      <Route
        path="/profil"
        element={
          <RouteProtegee>
            <Profil />
          </RouteProtegee>
        }
      />
    </Routes>
```



```
  );
}
```

Pattern Avancé avec Layout Protégé

```
// Layout protégé réutilisable
function LayoutProtege() {
  const estConnecte = checkAuth(); // Votre logique d'auth

  if (!estConnecte) {
    return <Navigate to="/login" replace />;
  }

  return (
    <div>
      <Header />
      <Outlet />
    </div>
  );
}

// Application avec zones protégées
function App() {
  return (
    <Routes>
      {/* Routes publiques */}
      <Route path="/" element={<Accueil />} />
      <Route path="/login" element={<Login />} />

      {/* Routes protégées */}
      <Route element={<LayoutProtege />}>
        <Route path="/dashboard" element={<Dashboard />} />
        <Route path="/settings" element={<Settings />} />
      </Route>
    </Routes>
  );
}
```

À Retenir

- Navigate pour rediriger automatiquement
- Vérifier l'authentification avant rendu
- Pattern de layout pour protéger plusieurs routes

SYNTHÈSE DES PATTERNS ESSENTIELS

Structure de Base Complète

```
function App() {
  return (
    <BrowserRouter>
      <Header />
      <Routes>
        {/* Public */}
        <Route path="/" element={<Accueil />} />
        <Route path="/login" element={<Login />} />

        {/* Protégé */}
        <Route element={<AuthGuard />}>
          <Route path="/dashboard" element={<Dashboard />} />
          <Route path="/profile/:id" element={<Profile />} />
        </Route>

        {/* 404 */}
        <Route path="*" element={<NotFound />} />
      </Routes>
      <Footer />
    </BrowserRouter>
  );
}
```

Hook Personnalisé pour l'Auth

```
function useAuth() {
  const navigate = useNavigate();
  const login = () => {
    // Logique de connexion
    navigate('/dashboard', { replace: true });
  };
  const logout = () => {
    // Logique de déconnexion
    navigate('/', { replace: true });
  };
  return { login, logout };
}
```

CHECKLIST DE VÉRIFICATION

Configuration

- J'ai installé react-router-dom
- J'ai wrappé mon app avec BrowserRouter

- J'utilise Routes et Route correctement

Routes

- Mes routes principales sont définies
- J'utilise useParams() pour les paramètres
- J'utilise Outlet pour les routes imbriquées

Navigation

- J'utilise Link pour les liens
- J'utilise useNavigate() pour la navigation code
- Je gère les retours en arrière

Sécurité

- J'ai protégé les routes sensibles
- Je redirige vers la page de login si non auth
- J'utilise replace pour éviter l'historique

Exercices Pratiques

Exercice 5.1 : Application Multi-pages

Créez une application avec les pages :
- Accueil (liste des articles)
- Article détail (/article/:id)
- À propos
- Contact
- Admin (protégée)

Exercice 5.2 : Authentification et Routes Protégées

Implémentez un système où :
- Seuls les utilisateurs connectés voient /dashboard
- Les non-connectés sont redirigés vers /login

MODULE 6 : STATE MANAGEMENT GLOBAL

Objectifs Pédagogiques

À la fin de ce module, l'étudiant sera capable de :
- Comprendre les limites du state local
- Utiliser Context API pour le state global
- Maîtriser Redux Toolkit pour les applications complexes
- Choisir la bonne solution selon le projet

6.1 Problème du Prop Drilling

Situation problématique :

Jsx :

```
<App user={user}>
  <Header user={user}>
    <Navigation user={user}>
      <UserMenu user={user} />
    </Navigation>
  </Header>
</App>
```

Solution : Context API

6.2 Context API

Création d'un contexte :

Jsx:

```
import { createContext, useContext, useState } from 'react';
const UserContext = createContext();
export function UserProvider({ children }) {
  const [user, setUser] = useState(null);
  const [isAuthenticated, setIsAuthenticated] = useState(false);
  const login = (userData) => {
    setUser(userData);
    setIsAuthenticated(true);
  };
  const logout = () => {
    setUser(null);
    setIsAuthenticated(false);
  };
  return (
    <UserContext.Provider value={{
      user,
      isAuthenticated,
      login,
      logout
    }}>
      {children}
    </UserContext.Provider>
  );
}
export const useUser = () => {
  const context = useContext(UserContext);
  if (!context) {
```



```
    throw new Error('useUser must be used within UserProvider');
  }
  return context;
};
```

Utilisation dans l'application :

Jsx:

```
// index.js
import { UserProvider } from './contexts/UserContext';
ReactDOM.render(
  <UserProvider>
    <App />
  </UserProvider>,
  document.getElementById('root')
);
// Composant utilisateur
function UserProfile() {
  const { user, logout } = useUser();
  return (
    <div>
      <h2>Bonjour {user?.name}</h2>
      <button onClick={logout}>Déconnexion</button>
    </div>
  );
}
```

6.3 Redux Toolkit (RTK)

Installation : npm install @reduxjs/toolkit react-redux

Configuration du store :

Jsx:

```
// store.js
import { configureStore, createSlice } from '@reduxjs/toolkit';
const counterSlice = createSlice({
  name: 'counter',
  initialState: { value: 0 },
  reducers: {
    increment: (state) => {
      state.value += 1;
    },
    decrement: (state) => {
```



```
      state.value -= 1;
    },
    incrementByAmount: (state, action) => {
      state.value += action.payload;
    },
  },
});
export const { increment, decrement, incrementByAmount } = counterSlice.actions;
export const store = configureStore({
  reducer: {
    counter: counterSlice.reducer,
  },
});
```

Utilisation avec React :

Jsx:

```
// index.js
import { Provider } from 'react-redux';
import { store } from './store';
ReactDOM.render(
  <Provider store={store}>
    <App />
  </Provider>,
  document.getElementById('root')
);
// Composant
import { useSelector, useDispatch } from 'react-redux';
import { increment, decrement } from './store';
function Counter() {
  const count = useSelector((state) => state.counter.value);
  const dispatch = useDispatch();
  return (
    <div>
      <span>{count}</span>
      <button onClick={() => dispatch(increment())}>+</button>
      <button onClick={() => dispatch(decrement())}>-</button>
    </div>
  );
}
```

6.4 Async Actions avec Redux Toolkit

Jsx:


```
// usersSlice.js
import { createSlice, createAsyncThunk } from '@reduxjs/toolkit';
export const fetchUsers = createAsyncThunk(
 'users/fetchUsers',
 async () => {
  const response = await fetch('/api/users');
  return response.json();
 }
);

const usersSlice = createSlice({
 name: 'users',
 initialState: {
  items: [],
  loading: false,
  error: null
 },
 reducers: {},
 extraReducers: (builder) => {
  builder
   .addCase(fetchUsers.pending, (state) => {
    state.loading = true;
   })
   .addCase(fetchUsers.fulfilled, (state, action) => {
    state.loading = false;
    state.items = action.payload;
   })
   .addCase(fetchUsers.rejected, (state, action) => {
    state.loading = false;
    state.error = action.error.message;
   });
 },
});
```

Exercices Pratiques

Exercice 6.1 : Panier d'Achat
Créez un contexte CartContext pour gérer :

- Ajout/suppression d'articles
- Calcul du total
- Persistance dans localStorage

Exercice 6.2 : Application avec Redux
Développez une application de gestion de tâches avec Redux Toolkit :

- Ajout/suppression de tâches
- Marquage comme terminé

- Filtrage par statut

Résumé

Solution	Cas d'usage	Complexité
State local	Données composant-specific	Simple
Context API	State global modéré	Moyenne
Redux Toolkit	Applications complexes	Élevée

MODULE 7 : STYLISATION PROFESSIONNELLE

Objectifs Pédagogiques

À la fin de ce module, l'étudiant sera capable de :

- Choisir la bonne méthodologie de stylisation
- Utiliser CSS Modules pour l'isolation
- Maîtriser Tailwind CSS pour le développement rapide
- Implémenter un design system cohérent

7.1 CSS Modules
Avantage : Isolation automatique des styles
Css :

```css
/* Button.module.css */
.button {
  padding: 12px 24px;
  border: none;
  border-radius: 8px;
  background-color: #007bff;
  color: white;
  cursor: pointer;
  transition: background-color 0.2s;
}
.button:hover {
  background-color: #0056b3;
}
.primary {
  composes: button;
  background-color: #28a745;
}
.primary:hover {
  background-color: #1e7e34;
}
```

Jsx:


```
// Button.js
import styles from './Button.module.css';

function Button({ children, variant = 'default' }) {
  const className = variant === 'primary'
    ? styles.primary
    : styles.button;

  return (
    <button className={className}>
      {children}
    </button>
  );
}
```

7.2 Styled Components

Installation : `npm install styled-components`

Utilisation :

Jsx:

```
import styled from 'styled-components';

const Button = styled.button`
  padding: 12px 24px;
  border: none;
  border-radius: 8px;
  background-color: ${props =>
    props.primary ? '#007bff' : '#6c757d'};
  color: white;
  cursor: pointer;
  transition: background-color 0.2s;

  &:hover {
    background-color: ${props =>
      props.primary ? '#0056b3' : '#545b62'};
  }
`;

function App() {
  return (
    <div>
```



```
      <Button>Normal</Button>
      <Button primary>Primary</Button>
    </div>
  );
    }
```

7.3 Tailwind CSS

Installation : npm install -D tailwindcss

npx tailwindcss init

Configuration :

Jsx:

```
// tailwind.config.js
module.exports = {
 content: ["./src/**/*.{js,jsx,ts,tsx}"],
 theme: {
  extend: {},
 },
 plugins: [],
}
```

Utilisation :

Jsx:

```
function Card({ title, content, image }) {
 return (
  <div className="max-w-sm rounded overflow-hidden shadow-lg bg-white">
   <img className="w-full" src={image} alt={title} />
   <div className="px-6 py-4">
    <div className="font-bold text-xl mb-2">{title}</div>
    <p className="text-gray-700 text-base">
     {content}
    </p>
   </div>
   <div className="px-6 pt-4 pb-2">
    <button className="bg-blue-500 hover:bg-blue-700 text-white font-bold py-2 px-4 rounded">
     Voir plus
    </button>
   </div>
  </div>
 );
```



```
}
```

7.4 Material-UI (MUI)

Installation : npm install @mui/material @emotion/react @emotion/styled

Utilisation :

Jsx:

```jsx
import { Button, Card, CardContent, Typography } from '@mui/material';
function ProductCard({ product }) {
 return (
  <Card sx={{ maxWidth: 345 }}>
   <CardContent>
    <Typography gutterBottom variant="h5" component="div">
     {product.name}
    </Typography>
    <Typography variant="body2" color="text.secondary">
     {product.description}
    </Typography>
    <Button
     variant="contained"
     sx={{ mt: 2 }}
    >
     Acheter
    </Button>
   </CardContent>
  </Card>
 );
}
```

7.5 Thème Clair/Sombre

Avec Context API :

Jsx:

```jsx
const ThemeContext = createContext();
export function ThemeProvider({ children }) {
 const [isDark, setIsDark] = useState(false);
 const theme = {
  isDark,
  toggleTheme: () => setIsDark(!isDark),
  colors: isDark ? darkColors : lightColors
 };
```



```
  return (
    <ThemeContext.Provider value={theme}>
      <div className={isDark ? 'dark-theme' : 'light-theme'}>
        {children}
      </div>
    </ThemeContext.Provider>
  );
}
```

Exercices Pratiques

Exercice 7.1 : Design System

Créez un design system réutilisable avec :

- Boutons (primary, secondary, danger)
- Cartes avec variations
- Typographie cohérente
- Thème clair/sombre

Exercice 7.2 : Dashboard Responsive

Développez un dashboard responsive avec :

- Sidebar navigation
- Cartes de statistiques
- Tableaux de données
- Graphiques (avec Chart.js ou autre)

Résumé

Méthode	Avantages	Inconvénients
CSS Modules	Isolation, familier	Configuration supplémentaire
Styled Components	CSS-in-JS, dynamique	Taille bundle, learning curve
Tailwind CSS	Rapide, cohérent	Classes verbeuses, learning curve
Material-UI	Composants riches	Personnalisation limitée

MODULE 8 : APIS ET DATA FETCHING

Objectifs Pédagogiques

À la fin de ce module, l'étudiant sera capable de :

- Effectuer des requêtes HTTP avec fetch et Axios
- Gérer les états de chargement et d'erreur
- Implémenter les opérations CRUD complètes

- Optimiser les performances de data fetching

8.1 Fetch API Native

GET Request :

Jsx:

```
function UsersList() {
  const [users, setUsers] = useState([]);
  const [loading, setLoading] = useState(true);
  const [error, setError] = useState(null);

  useEffect(() => {
    const fetchUsers = async () => {
      try {
        setLoading(true);
        const response = await fetch('https://jsonplaceholder.typicode.com/users');

        if (!response.ok) {
          throw new Error('Erreur réseau');
        }

        const data = await response.json();
        setUsers(data);
      } catch (err) {
        setError(err.message);
      } finally {
        setLoading(false);
      }
    };

    fetchUsers();
  }, []);

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

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



```
  );
}
```

8.2 Axios (Alternative à Fetch)

Installation : npm install axios

Utilisation :

Jsx:

```
import axios from 'axios';

function PostsManager() {
  const [posts, setPosts] = useState([]);

  useEffect(() => {
    const fetchPosts = async () => {
      try {
        const response = await axios.get(
          'https://jsonplaceholder.typicode.com/posts'
        );
        setPosts(response.data);
      } catch (error) {
        console.error('Erreur:', error);
      }
    };

    fetchPosts();
  }, []);
}
```

8.3 Custom Hook pour Data Fetching

Jsx:

```
function useApi(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);
```



```
        if (!response.ok) throw new Error('Erreur réseau');

        const result = await response.json();
        setData(result);
      } catch (err) {
        setError(err.message);
      } finally {
        setLoading(false);
      }
    };

    fetchData();
  }, [url]);

  return { data, loading, error };
}

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

  // Rendu conditionnel...
}
```

8.4 CRUD Complet

```
function ProductManager() {
  const [products, setProducts] = useState([]);
  const [editingProduct, setEditingProduct] = useState(null);

  // CREATE
  const createProduct = async (productData) => {
    const response = await fetch('/api/products', {
      method: 'POST',
      headers: { 'Content-Type': 'application/json' },
      body: JSON.stringify(productData),
    });
    const newProduct = await response.json();
    setProducts(prev => [...prev, newProduct]);
  };
```



```
  // READ
  const fetchProducts = async () => {
    const response = await fetch('/api/products');
    const products = await response.json();
    setProducts(products);
  };

  // UPDATE
  const updateProduct = async (id, updates) => {
    const response = await fetch(`/api/products/${id}`, {
      method: 'PUT',
      headers: { 'Content-Type': 'application/json' },
      body: JSON.stringify(updates),
    });
    const updatedProduct = await response.json();

    setProducts(prev =>
      prev.map(p => p.id === id ? updatedProduct : p)
    );
  };

  // DELETE
  const deleteProduct = async (id) => {
    await fetch(`/api/products/${id}`, { method: 'DELETE' });
    setProducts(prev => prev.filter(p => p.id !== id));
  };

  return (
    <div>
      {/* Interface utilisateur */}
    </div>
  );
}
```

8.5 Gestion Optimiste d'État

```
function OptimisticList() {
  const [items, setItems] = useState([]);

  const addItem = async (newItem) => {
    // Optimistic update
    const tempId = Date.now();
```



```
    const optimisticItem = { ...newItem, id: tempId };

    setItems(prev => [...prev, optimisticItem]);

    try {
      const response = await fetch('/api/items', {
        method: 'POST',
        body: JSON.stringify(newItem),
      });

      const savedItem = await response.json();

      // Remplacement par la vraie donnée
      setItems(prev =>
        prev.map(item =>
          item.id === tempId ? savedItem : item
        )
      );
    } catch (error) {
      // Rollback en cas d'erreur
      setItems(prev =>
        prev.filter(item => item.id !== tempId)
      );
      alert('Erreur lors de l\'ajout');
    }
  };
}
```

Exercices Pratiques

Exercice 8.1 : Application Blog Complète

Créez un blog avec :

- Liste des articles (GET)
- Création d'article (POST)
- Édition d'article (PUT)
- Suppression d'article (DELETE)
- Recherche et filtrage

Exercice 8.2 : Gestion de Cache

Implémentez un système de cache pour :

- Éviter les requêtes répétitives
- Invalider le cache quand nécessaire
- Gérer la synchronisation des données

MODULE 9 : PERFORMANCE ET OPTIMISATION

Objectifs Pédagogiques

À la fin de ce module, l'étudiant sera capable de :

- Identifier et résoudre les problèmes de performance
- Utiliser les techniques de memoization
- Implémenter le lazy loading des composants
- Analyser et optimiser le bundle

9.1 Memoization avec useMemo et useCallback

useMemo pour valeurs coûteuses :

```
function ExpensiveComponent({ items, filter }) {
  const filteredItems = useMemo(() => {
    return items.filter(item =>
      item.name.toLowerCase().includes(filter.toLowerCase())
    );
  }, [items, filter]); // Recalcul seulement si items ou filter change

  return (
    <div>
      {filteredItems.map(item => (
        <div key={item.id}>{item.name}</div>
      ))}
    </div>
  );
}
```

useCallback pour fonctions :

```
function ParentComponent() {
  const [count, setCount] = useState(0);
  const [value, setValue] = useState('');

  // ✅ Fonction memoized
  const handleClick = useCallback(() => {
    setCount(prev => prev + 1);
  }, []); // Dépendances vides = même fonction

  // ❌ Fonction recréée à chaque rendu
  const handleChange = (newValue) => {
```



```
    setValue(newValue);
  };

  return (
    <div>
      <ChildComponent onClick={handleClick} />
      <input value={value} onChange={handleChange} />
    </div>
  );
}

const ChildComponent = React.memo(({ onClick }) => {
  console.log('Child rendered'); // Seulement quand les props changent
  return <button onClick={onClick}>Click me</button>;
});
```

9.2 React.memo pour Composants

```
const UserCard = React.memo(({ user, onEdit }) => {
  return (
    <div className="user-card">
      <h3>{user.name}</h3>
      <p>{user.email}</p>
      <button onClick={() => onEdit(user.id)}>
        Éditer
      </button>
    </div>
  );
});

// Comparaison personnalisée
const UserCard = React.memo(({ user, onEdit }) => {
  // Composant
}, (prevProps, nextProps) => {
  // Re-rendu seulement si l'ID utilisateur change
  return prevProps.user.id === nextProps.user.id;
});
```

9.3 Lazy Loading avec React.lazy

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

// Chargement différé
```



```
const Dashboard = lazy(() => import('./components/Dashboard'));
const Settings = lazy(() => import('./components/Settings'));
const Analytics = lazy(() => import('./components/Analytics'));

function App() {
 const [currentPage, setCurrentPage] = useState('dashboard');

 const renderPage = () => {
  switch (currentPage) {
   case 'dashboard':
    return <Dashboard />;
   case 'settings':
    return <Settings />;
   case 'analytics':
    return <Analytics />;
   default:
    return <Dashboard />;
  }
 };

 return (
  <div>
   <nav>
    <button onClick={() => setCurrentPage('dashboard')}>
     Dashboard
    </button>
    <button onClick={() => setCurrentPage('settings')}>
     Settings
    </button>
    <button onClick={() => setCurrentPage('analytics')}>
     Analytics
    </button>
   </nav>

   <Suspense fallback={<div>Chargement...</div>}>
    {renderPage()}
   </Suspense>
  </div>
 );
}
```

9.4 Code Splitting avec React Router


```
import { Suspense, lazy } from 'react';
import { Routes, Route } from 'react-router-dom';

const Home = lazy(() => import('./pages/Home'));
const About = lazy(() => import('./pages/About'));
const Contact = lazy(() => import('./pages/Contact'));

function App() {
 return (
  <Suspense fallback={<div>Chargement de la page...</div>}>
   <Routes>
    <Route path="/" element={<Home />} />
    <Route path="/about" element={<About />} />
    <Route path="/contact" element={<Contact />} />
   </Routes>
  </Suspense>
 );
}
```

9.5 Optimisation du Bundle

Analyse du bundle :

```
npm install -D webpack-bundle-analyzer
npm run build
npx webpack-bundle-analyzer build/static/js/*.js
```

Configuration de split chunks :

Javascript:

```
// webpack.config.js (Create React App eject)
module.exports = {
 optimization: {
  splitChunks: {
   chunks: 'all',
   cacheGroups: {
    vendor: {
     test: /[\\/]node_modules[\\/]/,
     name: 'vendors',
     chunks: 'all',
    },
   },
  },
```



```
  },
};
```

9.6 Virtualisation de Listes

Avec react-window : `npm install react-window`

```
import { FixedSizeList as List } from 'react-window';

const BigList = ({ items }) => {
  const Row = ({ index, style }) => (
    <div style={style}>
      Item {items[index].name}
    </div>
  );

  return (
    <List
      height={400}
      itemCount={items.length}
      itemSize={50}
    >
      {Row}
    </List>
  );
};
```

Exercices Pratiques

Exercice 9.1 : Optimisation d'Application Lente
Analysez et optimisez une application existante :

- Identifiez les re-rendus inutiles
- Implémentez la memoization
- Ajoutez le lazy loading

Exercice 9.2 : Table de Données Performante
Créez un composant Table qui :

- Gère 10,000+ lignes efficacement
- Implémente la virtualisation
- Permet le tri et le filtrage

MODULE 10 : DÉPLOIEMENT ET PRODUCTION

Objectifs Pédagogiques

À la fin de ce module, l'étudiant sera capable de :

- Préparer une application React pour la production
- Déployer sur différentes plateformes (Vercel, Netlify, AWS)
- Configurer les variables d'environnement
- Mettre en place la surveillance et le monitoring

10.1 Build de Production

Commande de build : npm run build

Structure du dossier build :

```
build/
├── static/
│   ├── js/
│   │   ├── main.[hash].js
│   │   └── main.[hash].js.map
│   └── css/
│       └── main.[hash].css
├── index.html
├── manifest.json
└── robots.txt
```

10.2 Variables d'Environnement

.env.development :

Env:

```
REACT_APP_API_URL=http://localhost:3001/api
REACT_APP_DEBUG=true
```

.env.production :

Env:

```
REACT_APP_API_URL=https://api.monapp.com
REACT_APP_DEBUG=false
```

Utilisation dans le code :

```
const API_URL = process.env.REACT_APP_API_URL;
```



```
function App() {
  useEffect(() => {
    fetch(`${API_URL}/users`)
      .then(response => response.json())
      .then(data => console.log(data));
  }, []);
}
```

10.3 Déploiement sur Vercel

Configuration vercel.json :

Json :

```
{
  "version": 2,
  "builds": [
    {
      "src": "package.json",
      "use": "@vercel/static-build",
      "config": {
        "distDir": "build"
      }
    }
  ],
  "routes": [
    {
      "src": "/(.*)",
      "dest": "/index.html"
    }
  ]
}
```

Déploiement via CLI :

```
npm install -g vercel
vercel --prod
```

10.4 Déploiement sur Netlify

Configuration netlify.toml :

Toml:

```
[build]
```



```
  command = "npm run build"
  publish = "build"

[[redirects]]
  from = "/*"
  to = "/index.html"
  status = 200
```

Déploiement manuel :

- Drag & drop du dossier build sur Netlify

10.5 Déploiement sur AWS S3 + CloudFront

Configuration S3 :

```
# Installation AWS CLI
        aws configure
# Création du bucket
        aws s3 mb s3://mon-app-react
# Upload des fichiers
        aws s3 sync build/ s3://mon-app-react --delete
# Configuration du site statique
        aws s3 website s3://mon-app-react --index-document index.html
```

10.6 Error Boundaries

```
        class ErrorBoundary extends React.Component {
          constructor(props) {
            super(props);
            this.state = { hasError: false, error: null };
          }

          static getDerivedStateFromError(error) {
            return { hasError: true, error };
          }

          componentDidCatch(error, errorInfo) {
            // Envoyer l'erreur à un service de monitoring
            console.error('Error caught by boundary:', error, errorInfo);
          }

          render() {
```



```
    if (this.state.hasError) {
      return (
        <div className="error-boundary">
          <h2>Quelque chose s'est mal passé</h2>
          <details>
            {this.state.error && this.state.error.toString()}
          </details>
          <button onClick={() => this.setState({ hasError: false })}>
            Réessayer
          </button>
        </div>
      );
    }

    return this.props.children;
  }
}

// Utilisation
<ErrorBoundary>
  <ComponentQuiPeutPlanter />
</ErrorBoundary>
```

10.7 Monitoring et Analytics

Configuration de Google Analytics :

```
// hooks/useAnalytics.js
import { useEffect } from 'react';
import { useLocation } from 'react-router-dom';

export const useAnalytics = () => {
  const location = useLocation();

  useEffect(() => {
    if (window.gtag) {
      window.gtag('config', 'GA_MEASUREMENT_ID', {
        page_title: document.title,
        page_location: window.location.href,
      });
    }
  }, [location]);
};
```



```
// App.js
function App() {
  useAnalytics();

  return (
    // Votre application
  );
}
```

10.8 Performance Monitoring

```
// hooks/usePerformance.js
import { useEffect } from 'react';
export const usePerformance = () => {
  useEffect(() => {
    const observer = new PerformanceObserver((list) => {
      list.getEntries().forEach((entry) => {
        console.log(`${entry.name}: ${entry.duration}ms`);

        // Envoyer à votre service de monitoring
        if (entry.duration > 1000) {
          console.warn('Performance issue detected:', entry);
        }
      });
    });

    observer.observe({ entryTypes: ['measure', 'navigation'] });

    return () => observer.disconnect();
  }, []);
};
```

Exercices Pratiques

Exercice 10.1 : Pipeline de Déploiement Complet

Créez un pipeline CI/CD avec :

- Tests automatiques
- Build de production
- Déploiement sur Vercel/Netlify
- Notification des erreurs

Exercice 10.2 : Application Production-Ready

Prenez une application existante et :

- Ajoutez les Error Boundaries
- Configurez les variables d'environnement
- Implémentez le monitoring
- Déployez sur une plateforme

Résumé

Plateforme	Facilité	Coût	Fonctionnalités
Vercel	Très facile	Gratuit pour petit usage	Excellent pour Next.js
Netlify	Très facile	Gratuit pour petit usage	Fonctionnalités riches
AWS S3	Modérée	Payant à l'usage	Contrôle total
Firebase	Facile	Gratuit pour petit usage	Suite Google

PROJETS FINAUX INTÉGRATEURS

Projet 1 : Application de Gestion Scolaire

Fonctionnalités :

- Gestion des étudiants (CRUD)
- Gestion des cours et inscriptions
- Système de notes et bulletins
- Tableaux de bord administrateur
- Authentification et autorisation

Stack technique :

- React avec TypeScript
- Redux Toolkit pour le state management
- React Router pour la navigation
- Tailwind CSS pour le styling
- Axios pour les appels API

Projet 2 : Plateforme E-commerce

Fonctionnalités :

- Catalogue produits avec filtres
- Panier d'achat persistant
- Processus de commande en plusieurs étapes
- Système de paiement (simulé)
- Espace client avec historique

Stack technique :

- React avec Context API
- React Router
- Styled Components
- Integration avec Stripe (test)

Projet 3 : Réseau Social Minimaliste

Fonctionnalités :

- Flux d'actualités en temps réel
- Système d'amis et follow
- Messagerie instantanée
- Profils utilisateurs personnalisables
- Upload et gestion de médias

Stack technique :

- React avec Redux
- Socket.io pour le real-time
- Cloudinary pour les médias
- JWT pour l'authentification

ANNEXES

Annexe A : Glossaire React

Composant : Fonction ou classe qui retourne du JSX et décrit une partie de l'UI.

Hook : Fonction spéciale qui permet d'utiliser les fonctionnalités de React dans les composants fonctionnels.

JSX : Syntaxe étendue de JavaScript permettant d'écrire du HTML-like dans le code.

State : Données internes d'un composant qui peuvent changer au cours du temps.

Props : Données passées d'un composant parent à un composant enfant.

Virtual DOM : Représentation légère en mémoire du DOM réel.

Annexe B : Cheatsheet des Hooks

useState : Gestion d'état local

useEffect : Effets secondaires

useContext : Accès au contexte

useReducer : State complexe avec reducers

useMemo : Mémoization de valeurs

useCallback : Mémoization de fonctions

useRef : Références persistantes

useLayoutEffect : Effets synchrone après rendu

Annexe C : Bonnes Pratiques

1. **Composition over Inheritance :** Privilégiez la composition de composants
2. **Single Responsibility :** Un composant = une responsabilité
3. **Immutabilité :** Ne modifiez jamais le state directement
4. **Lifting State Up :** Remontez l'état au plus proche parent commun
5. **Controlled Components :** Utilisez des formulaires contrôlés

Annexe D : Ressources Recommandées

Documentation Officielle : https://react.dev

React Patterns : https://reactpatterns.com

Awesome React : https://github.com/enaqx/awesome-react

React TypeScript Cheatsheet : https://react-typescript-cheatsheet.netlify.app

CONCLUSION

Ce cours complet couvre l'ensemble des compétences nécessaires pour devenir un développeur React professionnel. Chaque module a été conçu pour construire progressivement une expertise solide, depuis les bases jusqu'aux concepts avancés.

Les projets pratiques et exercices permettent d'appliquer immédiatement les connaissances acquises, tandis que les bonnes pratiques et patterns architecturaux assurent la création d'applications maintenables et évolutives.

Prochaines étapes recommandées :

1. Pratique régulière sur des projets personnels
2. Contribution à des projets open source React
3. Exploration de l'écosystème (Next.js, Gatsby, React Native)
4. Veille technologique continue