



FORMATION

IT - Digital - Management

24/02/2024



m2iformation.fr

Bienvenue sur cette formation React

- Clément Hamon
- Développeur depuis 5 ans
- 3 ans de Javascript et React en entreprise
- 2 ans de formation
- clement.hamon35@gmail.com

Le réseau M2I Formation



Le groupe M2I

- Le groupe M2i est leader de la formation IT, Digital et Management en France depuis plus de 35 ans.
- L'engagement pour la qualité en étant certifié Qualiopi et Datadock.
- Plus de 300 collaborateurs dédiés à la montée en compétences de votre capital humain.
- Le catalogue M2I : <https://www.m2iformation.fr/catalogues/>
- La démarche qualité : <https://www.m2iformation.fr/demarche-qualite/>

Horaire et convocations

- 9h - 17h
- 15 mn de pause le matin
- 1h de pause déjeuner
- 15 mn de pause l'après midi

- Certifications
- Dernier jour à 16h30

Déroulé et structure de la formation et formalités

- Emargement le matin et l'après-midi
- Google Forms de demi-journée pour la validation des acquis et l'adaptabilité
- Évaluation du formateur le dernier jour au retour de la pause midi
- Notions théoriques suivis de mise en pratique

Tour de table et pré-requis

- Qui êtes-vous et quel est votre expérience dans l'informatique
- Droits admin
- Bonne connaissance de JavaScript
- Mail avant 10h30

Structure et versionning Github

- Pour chaque notion montrée, un commit sera fait sur ce git
- Support de cours sur Teams et Github

Objectifs de la formation

1. Vous familiariser avec la syntaxe et les fonctionnalités ES6
2. Présenter les concurrents de ReactJS
3. Présenter les principes de base du framework ReactJS
4. Gérer les évènements, rendu conditionnel et les listes dans le JSX
5. Présenter le fonctionnement de la navigation avec React et le router react-router-dom
6. Présenter les Hooks et les formulaires
7. Décrire le concept de store avec Redux
8. Décrire les tests avec Jest et RTL
9. Introduction à React Native
10. Introduction au SSR

Installation des outils

- Installation Node.js
- IDE Visual Code Studio
- React Developer Tools

1. ES6

2. Différence React et autres frameworks

React vs Angular

- React est souvent comparé à Angular, un autre framework populaire développé par Google.
- Contrairement à React, Angular est un **framework complet** avec une multitude d'outils intégrés p: formulaires, routes, services....
- React, plus léger, se concentre uniquement sur la partie "**V**" (**Vue**) du modèle MVC

React vs Vue

- Vue.js est souvent considéré comme un compromis entre React et Angular.
- Comme React, Vue.js est axé sur la création de composants et propose un système similaire de Virtual DOM. Cependant, Vue.js est un peu plus **opinionated** (il impose plus de conventions) que React, offrant une approche intégrée pour la gestion des états et des routes, mais sans la complexité d'Angular.
- React peut parfois sembler plus compliqué mais offre plus de flexibilité et d'évolutivité pour les projets de grande envergure.

Performance

- React et Vue.js sont souvent assez proches grâce à l'utilisation du Virtual DOM.
- Angular, étant un framework plus lourd avec plus de fonctionnalités intégrées, peut être plus lent dans certaines situations si toutes ses fonctionnalités ne sont pas nécessaires.
- React est donc un excellent choix pour les applications à page unique (SPA) où les parties de l'interface utilisateur changent fréquemment et nécessitent un rendu efficace.

3. Principes de base de React

Workflows de développement

- From scratch : Configuration manuel pour tout (Webpack, Babel, etc.). Plus complexe, mais très personnalisable.
 - `npm init`
- Intégration à une application Web existante : Introduire React progressivement.
- Utilisation de Create React App : Solution rapide et simple, sans avoir à se soucier des configurations.
 - `npx create-react-app formation-jvs-rea`

Composant

- React repose entièrement sur l'idée de **composants**.
- Partie autonome et indépendante d'une interface utilisateur
- Réutilisable et imbriquable dans d'autres composants
- Propre état et cycle de vie

Composant

Uploaded using RayThis Extension

App.js

```
import './App.css';
import HelloWorld from './HelloWorld';

function App() {
  return (
    <div className="App">
      <header className="App-header">
        <p>
          <HelloWorld />
        </p>
      </header>
    </div>
  );
}

export default App;
```

Uploaded using RayThis Extension

HelloWorld.jsx

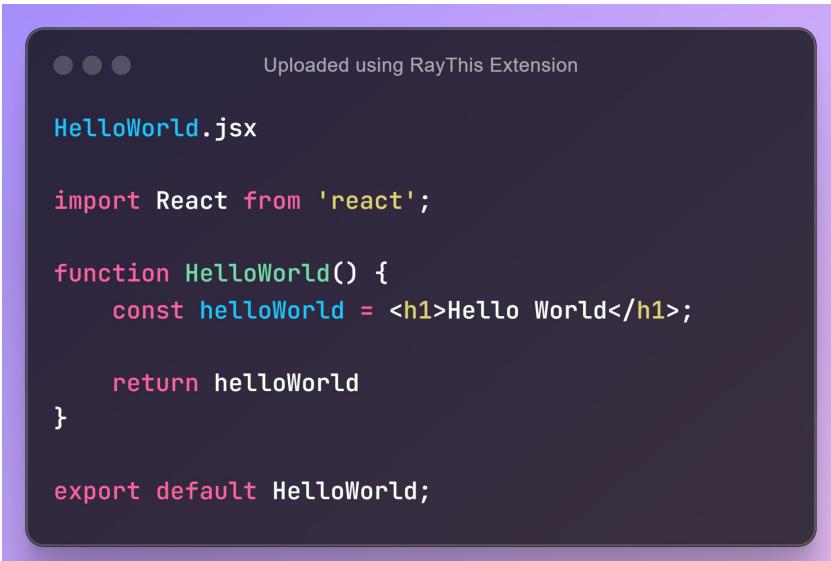
```
import React from 'react';

function HelloWorld() {
  return <h1>Hello World</h1>;
}

export default HelloWorld;
```

JSX (JavaScript XML)

- Permet de combiner JavaScript et HTML dans un seul fichier
- Rend le code plus lisible et structuré



Uploaded using RayThis Extension

• • •

HelloWorld.jsx

```
import React from 'react';

function HelloWorld() {
  const helloWorld = <h1>Hello World</h1>

  return helloWorld
}

export default HelloWorld;
```

Type de Composant

● Composants en mode classe et composants en mode fonction

Classe	Fonction
<pre>••• Uploaded using RayThis Extension ClasseOuFonction.jsx class ClassComponent extends Component { constructor(var1, var2) { this.var1 = var1; this.var2 = var2; } render() { return (<div> <h2>Composant en mode Classe</h2> </div>); } }</pre>	<pre>••• Uploaded using RayThis Extension ClasseOuFonction.jsx function FunctionComponent(var1, var2) { return (<div> <h2>Composant en mode Fonction</h2> </div>); }</pre>

Imbrication des composants

- Component Parent qui contient un ou des components Enfant

Uploaded using RayThis Extension

```
HelloWorld.jsx

import React from 'react';
import ChildComponent from './components/ChildComponent';

function HelloWorld() {

  const helloWorld = <h1>Ceci est une variable qui contient un élément JSX</h1>

  return (
    <section>
      <h1>Hello World</h1>
      {helloWorld}
      <ChildComponent /> //Compenent imbriqué (ou nested)
      <p>Je viens en dessous du composant</p>
    </section>
  );
}

export default HelloWorld;
```

Uploaded using RayThis Extension

```
ChildComponenent.jsx

import React from 'react'

function ChildComponent() {
  return (
    <div>
      <p>Je suis un enfant</p>
    </div>
  )
}

export default ChildComponent;
```

Exercice

Exercice 1 : Création d'un composant de type classe

1. Crée un composant de type classe `HelloWorld.jsx` appelé `HelloWorld` qui retourne un message "Hello World!" dans un élément `<p>`.
2. Affiche ce composant dans `App.js`.

Exercice 2 : Création de composants fonctionnels avec JSX

1. Crée un composant de type fonction dans `Welcome.jsx` appelé `Welcome` qui retourne un message de bienvenue en utilisant JSX.
2. Dans `App.js`, affiche trois fois le composant `Welcome`.

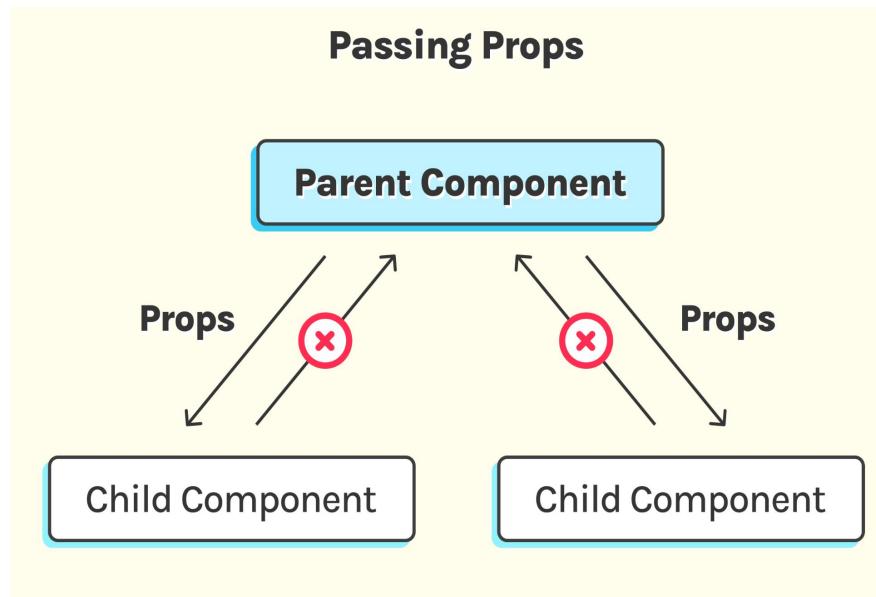
Exercice 3 : Imbrication de composants

1. Crée un composant fonctionnel dans `Titre.jsx` appelé `Titre` qui affiche un titre `<h3>`.
2. Crée un composant dans `Page.jsx` appelé `Page` qui retourne les composants `Titre` et `Welcome` (créé dans l'ex2) imbriqués ensemble.
3. Affiche le composant `Page` dans `App.js`.

Props



Les **props** (**propriétés**) sont des **informations** que l'on passe **d'un composant parent à un composant enfant**. Comme des arguments d'une fonction en JavaScript, les props permettent de rendre un composant dynamique en lui transmettant des valeurs spécifiques qu'il va utiliser.



Props

Uploaded using RayThis Extension

HelloWorld.jsx

```
function HelloWorld() {  
  
  return (  
    <section>  
      <h1>Hello World</h1>  
      <ChildComponentWithProps  
        message="Message du props"  
        nombre={10}  
        jsx={<p>Je suis un élément JSX</p>} />  
      <p>Je viens en dessous du composant</p>  
    </section>  
  );  
}
```

Uploaded using RayThis Extension

ChildComponentWithProps.jsx

```
export default function ChildComponentWithProps(props) {  
  return (  
    <div>  
      <h2>Composant Enfant avec Props</h2>  
      <p>Message : {props.message}</p>  
      <p> Nombre : {props.nombre}</p>  
      {props.jsx}  
    </div>  
  );  
}
```

Props

- `props.children`
- prop spéciale en React, utilisée pour rendre et injecter des éléments enfants au sein d'un composant.
- créé des composants conteneurs qui peuvent envelopper ou encapsuler d'autres éléments ou composants, en leur permettant de recevoir du contenu arbitraire depuis leur parent.

Props

Uploaded using RayThis Extension

App.js

```
<Box>
  <p>Ceci est une boîte avec du contenu personnalisé.</p>
</Box>
```

Uploaded using RayThis Extension

ComponentWithProps.children.jsx

```
export function Box(props) {
  return (
    <div style={{ border: '2px solid black', padding: '10px' }}>
      {props.children}
    </div>
  );
}
```

Ceci est une boîte avec du contenu personnalisé.

Exercice

Exercice 1 : Utilisation des props

1. Dans composant `WelcomeWithProps` , crée une prop `name` et affiche "Bienvenue, [name]".
2. Utilise ce composant dans `App.js` et passe-lui des noms différents en prop.

Exercice 2 : Plusieurs props

1. Crée un composant fonctionnel `UserInfo` qui prend deux props : `name` et `age`, et les affiche avec un petit texte de présentation.
2. Utilise ce composant dans `App.js` avec plusieurs valeurs différentes.

Exercice 3 : Utilisation de `props.children`

1. Crée un composant `Container` qui affiche son contenu dans une `div`, en utilisant `props.children`.
2. Dans `App.js`, utilise `Container` pour encapsuler du texte ou des éléments JSX, tu peux par exemple faire une description de ton parcours scolaire ou parler de ce que tu aimes.

Exercice 4 : Passer des composants comme `props.children`

1. Utilise les composants que tu as déjà créer pour les afficher grâce à `props.children`

Listes et fonctions

- **map()** : Rend une liste d'éléments
- La méthode **map()** est utilisée pour transformer un tableau en un autre tableau. En React, elle est souvent utilisée pour rendre une liste d'éléments à partir d'un tableau de données

```
•••          Uploaded using RayThis Extension

const fruits = ['Pomme', 'Banane', 'Orange', 'Ananas'];

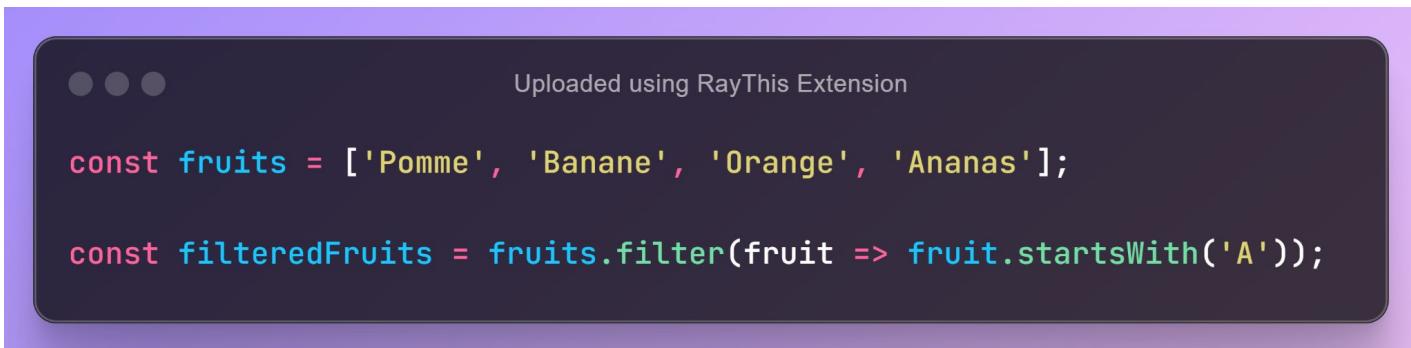
return (
  <ul>
    {fruits.map((fruit, index) => (
      <li key={index}>{fruit}</li>
    )));
  </ul>
);
```

Listes et fonctions

- Pourquoi utiliser des clés (key) ?
- Les **clés** sont des attributs spéciaux que React utilise pour identifier de manière **unique** chaque élément dans une liste de composants.
- Elles permettent à React de suivre les changements dans le Virtual DOM et de mettre à jour efficacement l'interface utilisateur, en ne modifiant que les éléments impactés et non toute la liste.

Listes et fonctions

- `filter()` : Filtre une liste d'éléments
- La méthode `filter()` est utilisée pour créer un nouveau tableau contenant uniquement les éléments qui satisfont une condition. En React, cela permet de rendre des éléments conditionnellement en fonction d'une logique métier.



Uploaded using RayThis Extension

```
const fruits = ['Pomme', 'Banane', 'Orange', 'Ananas'];

const filteredFruits = fruits.filter(fruit => fruit.startsWith('A'));
```

Listes et fonctions

Conclusion :

- `map()` permet de transformer un tableau en une liste d'éléments JSX, très utile pour rendre des listes en React.
- `filter()` permet de filtrer un tableau avant de rendre les éléments, pratique pour appliquer des conditions sur les données.
- Ces méthodes peuvent être combinées pour des transformations complexes et permettent d'écrire du code concis dans React.

Exercice

Exercice 1 : Utilisation de `map()` pour rendre une liste d'éléments

1. Crée un composant `ItemList` qui prend une prop `items`, un tableau de chaînes de caractères.
2. Utilise la méthode `map()` pour transformer ce tableau en une liste d'éléments `` dans une liste ``.
3. Utilise ce composant dans `App.js` avec un tableau d'exemple de chaînes de caractères.

Exercice 2 : Utilisation de `filter()` pour afficher des éléments filtrés

1. Crée un composant `FilteredItemList` qui prend deux props :
 - o `items`, un tableau de chaînes de caractères.
 - o `query`, une chaîne de caractères pour filtrer les éléments.
2. Utilise la méthode `filter()` pour ne garder que les éléments qui contiennent la chaîne `react`.
3. Utilise la méthode `map()` pour transformer les éléments filtrés en une liste d'éléments `` dans une liste ``.
4. Utilise ce composant dans `App.js` avec un tableau d'exemple et une chaîne de filtre.

Exercice bonus : Affichage de composants avec une liste

1. Crée un composant `User` qui prend une prop `name` et affiche un élément `<div>` avec le nom de l'utilisateur.
2. Crée un composant `UserList` qui prend une prop `users`, un tableau d'objets avec une propriété `name`.
3. Utilise la méthode `map()` pour transformer ce tableau en une liste de composants `User` dans le composant `UserList`.
4. Utilise ce composant dans `App.js` avec un tableau d'exemple d'utilisateurs.

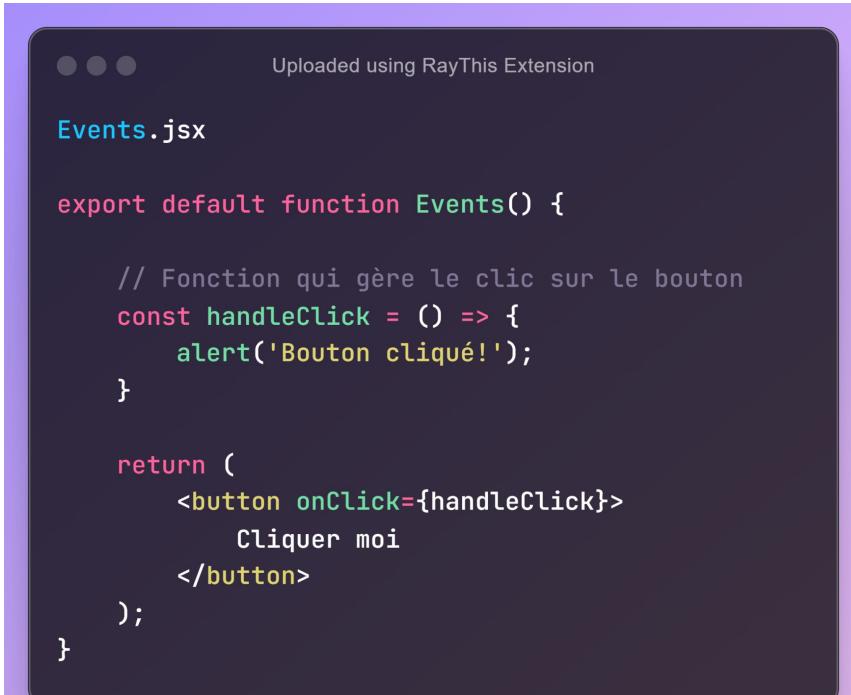
Virtual DOM

- Réplique légère du DOM réel
- Comparaison entre version virtuelle et version réelle, puis mise à jour des parties qui ont changées
- Amélioration des performances

4. Les événements et conditions

Événements

- Un gestionnaire d'événements est une fonction qui est définie pour réagir à un événement spécifique



Events.jsx

```
Uploaded using RayThis Extension

export default function Events() {

    // Fonction qui gère le clic sur le bouton
    const handleClick = () => {
        alert('Bouton cliqué!');
    }

    return (
        <button onClick={handleClick}>
            Cliquer moi
        </button>
    );
}
```

Dans cet exemple, `onClick` est l'événement de clic qui déclenche la fonction `handleClick`.

Événements

Voici quelques événements couramment utilisés en React :

- onClick : déclenché lorsque l'utilisateur clique sur un élément.
- onChange : déclenché lorsque la valeur d'un champ d'entrée change (utile pour les formulaires).
- onSubmit : déclenché lors de la soumission d'un formulaire.
- onMouseEnter / onMouseLeave : déclenchés lorsque la souris entre/sort d'un élément.
- onKeyDown / onKeyUp : déclenchés lorsqu'une touche est enfoncée/relâchée.

Exercice

Exercice 1 : Gestion d'un événement `onClick`

1. Crée un composant fonctionnel `Button` qui affiche un bouton avec le texte "Cliquez-moi".
2. Ajoute un événement `onClick` pour afficher un `alert` avec un message personnalisé lorsqu'on clique sur le bouton.

Exercice 2 : Événement `onMouseOver`

1. Crée un composant `HoverBox` qui affiche un `div` avec le texte "Survolez-moi".
2. Ajoute un événement `onMouseOver` pour changer la couleur du texte lorsque la souris passe dessus.

Exercice 3 : Événement `onMouseLeave`

1. Ajoute un événement `onMouseLeave` pour changer la couleur du texte lorsque la souris sort de la `div` avec le texte "Survolez-moi" ..

Exercice Bonus : Gestion des événements `onKeyUp` et `onKeyDown`

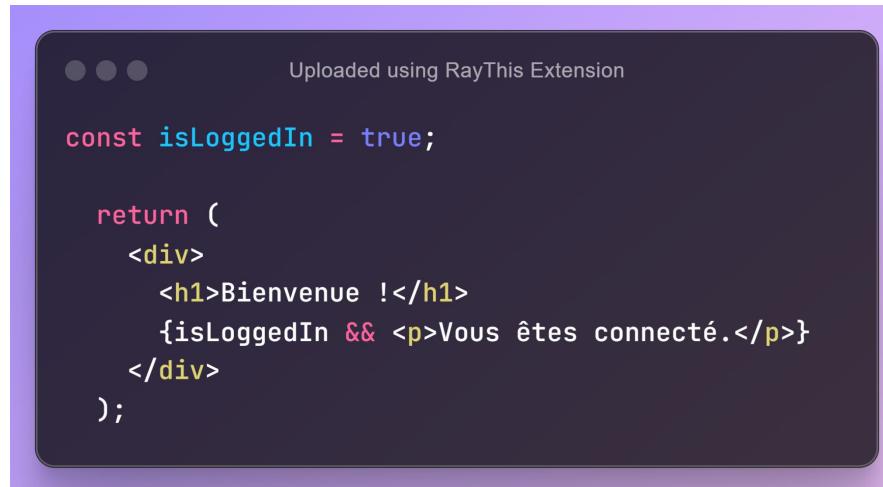
1. Crée un composant `KeyTracker` qui affiche un champ de texte `<input>`.
2. Ajoute deux événements :
 - o Un événement `onKeyDown` qui affiche un message dans la console indiquant quelle touche a été enfoncée.
 - o Un événement `onKeyUp` qui affiche un message indiquant que la touche a été relâchée.

Conditions

- Opérateur logique `&&` (ET logique)
- Dans cet exemple :

Si `isLoggedIn` est `true`, le paragraphe "Vous êtes connecté" sera affiché.

Si `isLoggedIn` est `false`, le paragraphe ne sera pas rendu.



Uploaded using RayThis Extension

```
const isLoggedIn = true;

return (
  <div>
    <h1>Bienvenue !</h1>
    {isLoggedIn && <p>Vous êtes connecté.</p>}
  </div>
);
```

Conditions

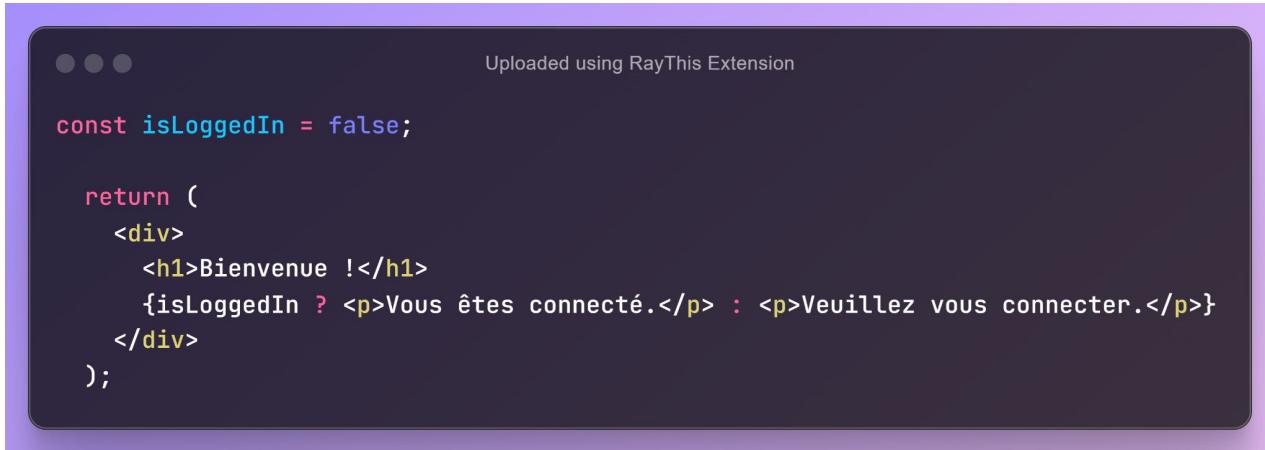
- Opérateur ternaire

Syntaxe : {condition ? contenuSiVrai : contenuSiFaux};

- Dans cet exemple :

Si `isLoggedIn` est `true`, le paragraphe "Vous êtes connecté" sera affiché.

Si `isLoggedIn` est `false`, le paragraphe "Veuillez vous connecter" sera affiché.



The screenshot shows a dark-themed code editor window with a purple header bar. The code in the editor is:

```
const isLoggedIn = false;

return (
  <div>
    <h1>Bienvenue !</h1>
    {isLoggedIn ? <p>Vous êtes connecté.</p> : <p>Veuillez vous connecter.</p>}
  </div>
);
```

At the top left of the editor, there are three small circular icons. At the top right, it says "Uploaded using RayThis Extension".

Conditions

- Attention ! En JavaScript, certaines valeurs comme 0, null, undefined, false, et "" (chaîne vide) sont évaluées comme false

Conditions

En résumé

- **Opérateur `&&`** : Affiche le contenu uniquement si la condition est vraie.
- **Opérateur ternaire** : Choisit entre deux contenus en fonction de la condition.
- Les deux opérateurs permettent de simplifier le rendu conditionnel sans avoir à écrire des blocs `if-else` plus longs.

Exercice

Exercice 1 : Utilisation de l'opérateur `&&`

1. Crée un composant fonctionnel `MessageDisplay` qui prend une prop `isTrue` (valeur booléenne true/false).
2. Utilise l'opérateur `&&` pour n'afficher le message "Bienvenue !" que si `isTrue` est vrai.
3. Utilise ce composant dans `App.js` avec différentes valeurs pour `isTrue`.

Exercice 2 : Utilisation de l'opérateur ternaire

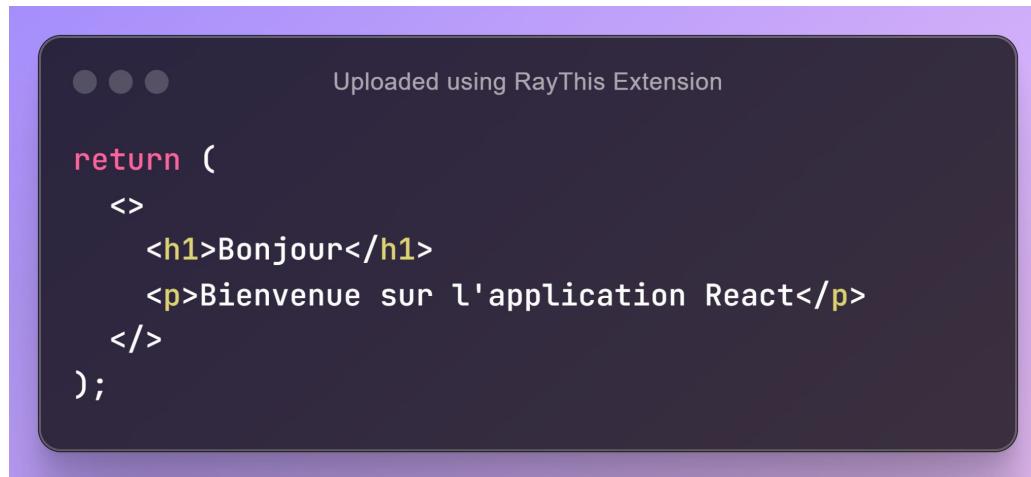
1. Crée un composant fonctionnel `LoginButton` qui prend une prop `isLoggedIn`.
2. Utilise l'opérateur ternaire pour afficher soit "Se déconnecter" si `isLoggedIn` est vrai, soit "Se connecter" si `isLoggedIn` est faux.
3. Utilise ce composant dans `App.js` en modifiant la valeur de `isLoggedIn`.

Exercice Bonus : Afficher le nombre de messages non lus

1. Crée un composant fonctionnel `UnreadMessages` qui prend une prop `unreadCount` (un nombre représentant le nombre de messages non lus).
2. Affiche "Vous avez [nombre] messages non lus".
3. Utilise ce composant dans `App.js` avec différentes valeurs pour `unreadCount`.

Fragments

Les **fragments** permettent de regrouper plusieurs éléments sans ajouter de nœud supplémentaire dans le DOM. Cela est utile lorsque vous souhaitez retourner plusieurs éléments d'un composant, mais sans introduire un élément parent inutile comme une `div`.



The screenshot shows a dark-themed code editor window with a purple header bar. The code inside the editor is:

```
return (
  <>
    <h1>Bonjour</h1>
    <p>Bienvenue sur l'application React</p>
  </>
);
```

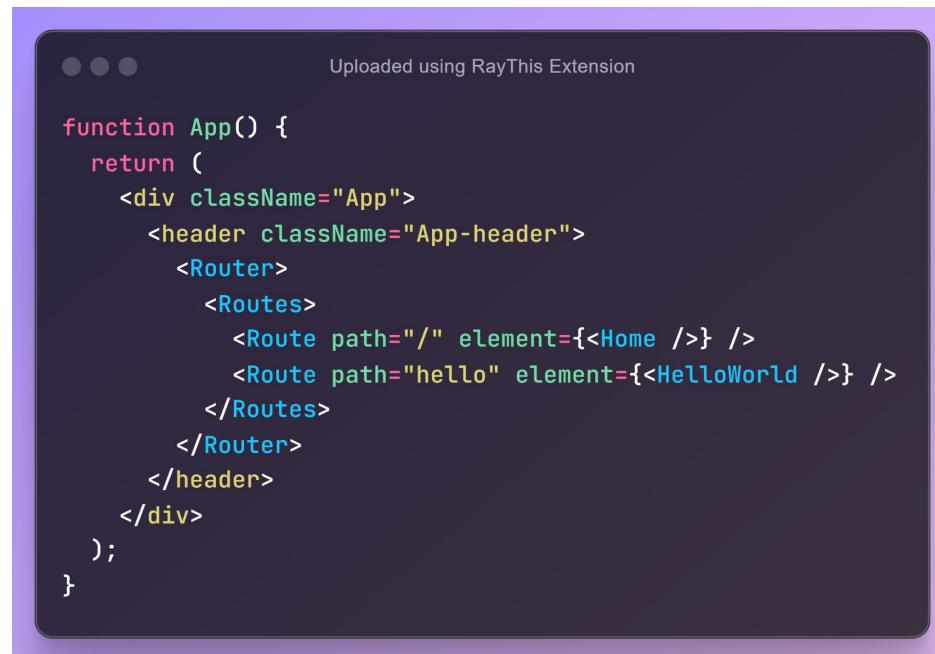
The code uses a fragment (`<>`) to group the `<h1>` and `<p>` elements without creating an unnecessary parent `div`. The editor interface includes three dots in the top-left corner and the text "Uploaded using RayThis Extension" in the top-right corner.

Dans cet exemple, les éléments `<h1>` et `<p>` sont contenus dans un fragment vide (`<> . . . </>`), ce qui évite d'ajouter un élément parent inutile dans le DOM.

5. Le routing et la navigation

Les routes

- **npm install react-router-dom**
- Le routeur de `react-router-dom` est configuré en utilisant le composant `BrowserRouter`. Ce composant doit entourer toute votre application pour permettre le routage.



Uploaded using RayThis Extension

```
function App() {
  return (
    <div className="App">
      <header className="App-header">
        <Router>
          <Routes>
            <Route path="/" element={<Home />} />
            <Route path="hello" element={<HelloWorld />} />
          </Routes>
        </Router>
      </header>
    </div>
  );
}
```

Liens de Navigation

```
import { BrowserRouter as Router, Routes, Route, Link } from 'react-router-dom';

function App() {
  return (
    <div className="App">
      <header className="App-header">
        <Router>
          <Routes>
            <Route path="/" element={
              <button>
                <Link to="/hello">Go to Hello World</Link>
              </button>
            } />
            <Route path="/hello" element={<HelloWorld />} />
          </Routes>
        </Router>
      </header>
    </div>
  );
}

export default App;
```

Uploaded using RayThis Extension

Les paramètres d'URL

- `:userId` est un **paramètre d'URL** dynamique. Dans votre composant `User`, vous pouvez accéder à ce paramètre avec `useParams()`

Uploaded using RayThis Extension

```
<Route path="user/:userId" element={<User />}/>
```

Uploaded using RayThis Extension

```
import { useParams } from 'react-router-dom';

export default function User() {
  const { userId } = useParams(); // Extraction du paramètre userId de l'URL
  return (
    <div>
      <h1>Page utilisateur pour l'ID utilisateur : {userId}</h1>
    </div>
  );
}
```

localhost:3000/user/12345

Page utilisateur pour l'ID utilisateur : 12345

Les routes imbriquées

- Lorsque l'utilisateur navigue vers `/user/42`, le composant `User` est affiché.
- Lorsque l'utilisateur clique sur le lien "View User Details", il est redirigé vers `/user/42/details`, et la route imbriquée `UserDetails` est rendue à l'intérieur du composant `User` grâce à l'élément `<Outlet>`.



Uploaded using RayThis Extension

```
<Route path="user/:userId" element={<User />}>
  <Route path="details" element={<UserDetails />} />
  <Route path="settings" element={<UserSettings />} />
</Route>
```



Uploaded using RayThis Extension

```
export default function User() {
  const { userId } = useParams();
  return (
    <>
      <h1>User Page for User ID: {userId}</h1>
      <nav>
        <Link to="details">View User Details</Link>
      </nav>
      <Outlet /> {/* Rend les sous-routes ici */}
    </>
  );
}
```

Exercice

Exercice 1 : Création de routes simples

1. Installe `react-router-dom` si ce n'est pas déjà fait avec `npm install react-router-dom`.
2. Crée un composant fonctionnel `Home` qui affiche un message "Accueil".
3. Crée un composant fonctionnel `About` qui affiche un message "À propos".
4. Configure un `BrowserRouter` dans `App.js` pour créer deux routes : `/home` pour le composant `Home` et `/about` pour le composant `About`.
5. Affiche les composants en fonction de la route actuelle.

Exercice 2 : Utilisation de `Link` pour la navigation

1. Ajoute un composant `Navigation` avec des liens `Link` pour naviguer entre les routes `/` et `/about`.
2. Utilise ce composant `Navigation` dans `App.js` pour permettre la navigation entre les pages grâce à un bouton.

Exercice 3 : Routes avec paramètres

1. Crée un composant fonctionnel `UserProfile` qui prend un paramètre d'URL `:id` et affiche "Profil de l'utilisateur [id]".
2. Configure une route `/user/:id` pour le composant `UserProfile` dans `App.js`.
3. Utilise `useParams` de `react-router-dom` pour accéder au paramètre d'URL dans `UserProfile`.

Exercice

Exercice 4 : Routes imbriquées

1. Crée un composant fonctionnel `UserProfile` qui prend un paramètre d'URL `:id` et affiche "Profil de l'utilisateur [id]".
2. Crée un composant fonctionnel `UserAbout` qui affiche "À propos de l'utilisateur [id]".
3. Configure les routes suivantes :
 - o `/user/:id` pour le composant `UserProfile`.
 - o `/user/:id/about` pour le composant `UserAbout`.
4. Utilise `useParams` de `react-router-dom` pour accéder au paramètre d'URL `:id` dans chaque composant.
5. Imbrique la route `settings` sous la route `/user/:id` pour le composant `UserSettings` dans `App.js`

6. Les Hooks

Hooks

- Permet de gérer l'état et d'autres fonctionnalités sans avoir à utiliser les composants de classe. Par exemple, le hook useState est utilisé pour gérer l'état d'un composant fonctionnel.

Hooks

- useState
- Le Hook useState permet d'ajouter un état local à des composants fonctionnels
- Il renvoie un tableau avec deux éléments : l'état actuel et une fonction pour le mettre à jour

Hooks



Uploaded using RayThis Extension

useState.jsx

```
export default function Counter() {
  const [count, setCount] = useState(0); // initialisation de l'état à 0

  return (
    <div>
      <p>Vous avez cliqué {count} fois</p>
      <button onClick={() => setCount(count + 1)}>Cliquez ici</button>
      <button onClick={() => setCount(0)}>Reset</button>
    </div>
  );
}
```

Hooks

- `useEffect`
- Permet d'exécuter du code après le rendu du composant
- `useEffect` s'exécute après chaque rendu

Hooks

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

function Timer() {
  const [seconds, setSeconds] = useState(0);

  useEffect(() => {
    const interval = setInterval(() => {
      setSeconds(prevSeconds => prevSeconds + 1);
    }, 1000);

    return () => clearInterval(interval);
  }, []);

  return <p>Temps écoulé : {seconds} secondes</p>;
}

export default Timer;
```

Uploaded using RayThis Extension

Exercice

Exercice 1 : Utilisation du hook useState

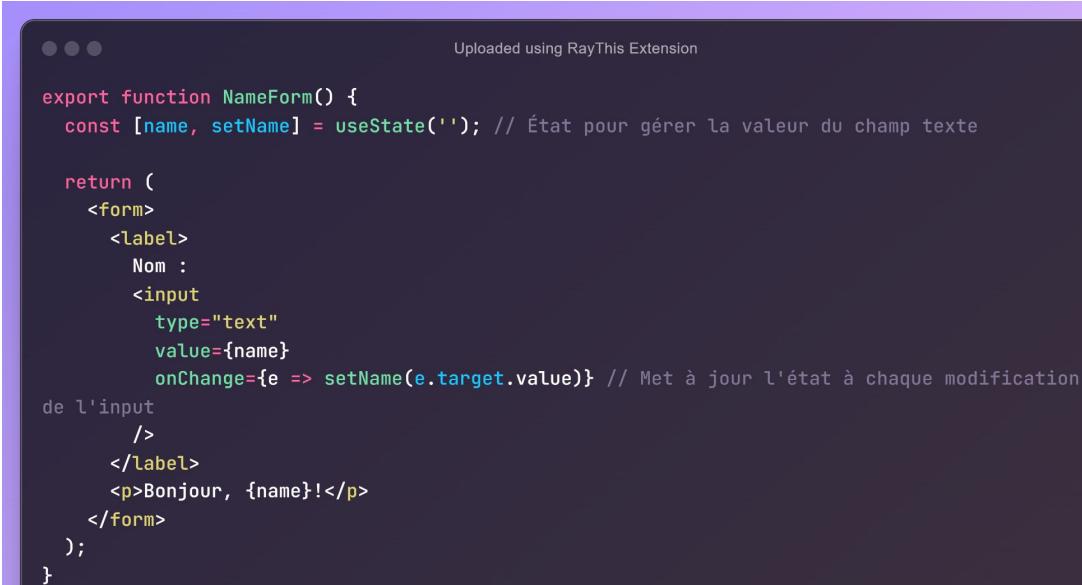
1. Crée un composant fonctionnel `Counter` qui affiche un compteur et un bouton pour incrémenter ce compteur.
2. Utilise le hook `useState` pour gérer l'état du compteur.
3. Chaque fois que le bouton est cliqué, incrémentez le compteur de 1.

Exercice 2 : Utilisation de useEffect pour la gestion du titre du document

1. Crée un composant fonctionnel `DocumentTitle` qui permet à l'utilisateur de saisir un titre dans un champ de texte.
2. Utilise le hook `useState` pour gérer la valeur du champ de texte.
3. Utilise le hook `useEffect` pour mettre à jour le titre du document (le texte affiché dans l'onglet du navigateur) chaque fois que la valeur du champ de texte change
4. PS: Le titre de l'onglet est stocké dans la variable `document.title`.
5. Pensez à ce qui gère les changements en React (diapo 32)

Composant Contrôlé

- Un composant contrôlé est un composant de formulaire dont la valeur est entièrement gérée par l'état React. Cela signifie que chaque modification de l'entrée de l'utilisateur est capturée via l'état, et l'interface utilisateur est ré-rendue en conséquence.



Uploaded using RayThis Extension

```
export function NameForm() {
  const [name, setName] = useState(''); // État pour gérer la valeur du champ texte

  return (
    <form>
      <label>
        Nom :
        <input
          type="text"
          value={name}
          onChange={e => setName(e.target.value)} // Met à jour l'état à chaque modification
          de l'input
        />
      </label>
      <p>Bonjour, {name}!</p>
    </form>
  );
}
```

Gestion des formulaires avec useState

Dans cet exemple :

- `handleChange` met à jour l'état `formData` en fonction des entrées de l'utilisateur.
- `handleSubmit` est appelé lors de la soumission du formulaire pour traiter les données.



```
const [formData, setFormData] = useState({
  username: '',
  email: ''
});

const handleChange = (e) => {
  setFormData({
    ...formData, // Copie des autres propriétés existantes
    [e.target.name]: e.target.value // Mise à jour de la propriété modifiée
  });
};

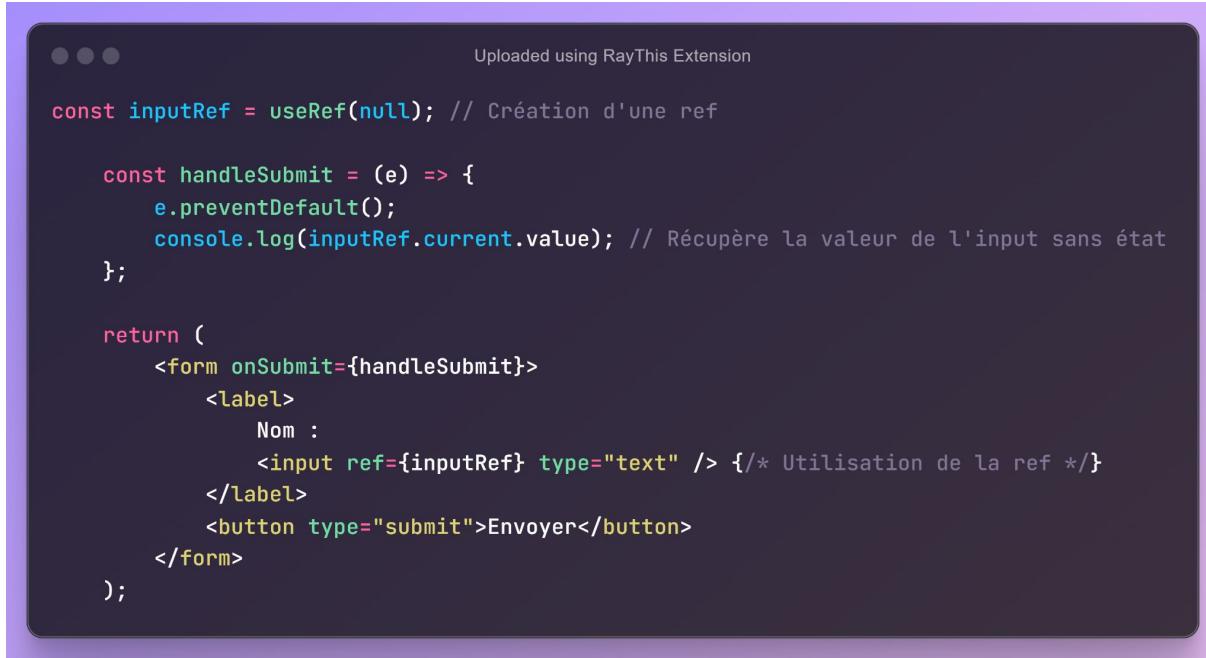
const handleSubmit = (e) => {
  e.preventDefault(); // Empêche le rechargement de la page
  console.log(formData);
};
```



```
return (
  <form onSubmit={handleSubmit}>
    <label>
      Nom d'utilisateur :
      <input
        type="text"
        name="username"
        value={formData.username}
        onChange={handleChange}
      />
    </label>
    <label>
      Email :
      <input
        type="email"
        name="email"
        value={formData.email}
        onChange={handleChange}
      />
    </label>
    <br />
    <button type="submit">Envoyer</button>
  </form>
);
```

Gestion des formulaires avec useRef

Contrairement à l'état (useState), une ref ne provoque **pas de ré-rendu** lorsque sa valeur change.



Uploaded using RayThis Extension

```
const inputRef = useRef(null); // Création d'une ref

const handleSubmit = (e) => {
  e.preventDefault();
  console.log(inputRef.current.value); // Récupère la valeur de l'input sans état
};

return (
  <form onSubmit={handleSubmit}>
    <label>
      Nom :
      <input ref={inputRef} type="text" /> /* Utilisation de la ref */
    </label>
    <button type="submit">Envoyer</button>
  </form>
);
```

Hooks

- **Les Hooks doivent être appelés au niveau supérieur** : Ne les appelez pas dans des boucles, des conditions ou des fonctions imbriquées !
- **Les Hooks ne peuvent être appelés que dans des composants React fonctionnels** ou dans des Hooks personnalisés !

Exercice

Exercice 1 : Gestion d'un formulaire avec useState

1. Crée un composant fonctionnel `ContactForm` avec un formulaire simple comprenant les champs suivants :
 - o Nom
 - o Email
 - o Message
2. Utilise le hook `useState` pour gérer l'état de chaque champ du formulaire.
3. Ajoute un bouton "Envoyer" qui, lorsqu'il est cliqué, affiche les valeurs des champs dans la console.

Exercice 2 : Gestion d'un formulaire avec useRef

1. Crée un composant fonctionnel `ContactFormWithRef` avec un formulaire simple comprenant les champs suivants :
 - o Nom
 - o Email
 - o Message
2. Utilise `useRef` pour référencer chaque champ du formulaire.
3. Ajoute un bouton "Envoyer" qui, lorsqu'il est cliqué, affiche les valeurs des champs dans la console en utilisant les références.

Hooks

- Fonctions JavaScript qui commencent par `use` et qui permettent de réutiliser de la logique basée sur des Hooks dans différents composants.
- Les custom Hooks permettent de mieux organiser le code et de le rendre plus réutilisable en partageant la logique entre plusieurs composants.

Hooks

- Exemple de customHook

Uploaded using RayThis Extension

```
function useCounter(initialValue = 0) {
  const [count, setCount] = useState(initialValue);

  const increment = () => setCount(count + 1);
  const decrement = () => setCount(count - 1);
  const reset = () => setCount(initialValue);

  return { count, increment, decrement, reset };
}
```

Uploaded using RayThis Extension

```
export default function CounterComponent() {
  const { count, increment, decrement, reset } = useCounter();

  return (
    <div>
      <p>Compteur : {count}</p>
      <button onClick={increment}>+1</button>
      <button onClick={decrement}>-1</button>
      <button onClick={reset}>Reset</button>
    </div>
  );
}
```

7. Les actions, les reducers et le store

Problème de la Gestion d'État

- Chaque changement de valeur du useState = re-render du parent ET de l'enfant

Uploaded using RayThis Extension

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

  return (
    <div>
      <Child count={count} />
      <button onClick={() => setCount(count + 1)}>Increment</button>
    </div>
  );
}

// Composant Enfant
function Child({ count }) {
  return <div>Count: {count}</div>;
}
```

Comment le résoudre ?

- **Actions** : Objets décrivant un événement. Chaque action a un *type* (qui identifie l'action) et, souvent, une charge utile (*payload*) avec des données supplémentaires.
- **Reducers** : Ce sont des fonctions pures qui reçoivent l'état *actuel* et une *action*, et renvoient un *nouvel état modifié*. Ils déterminent comment l'état de l'application change en réponse à une action donnée.
- **Store** : Le store est l'objet central qui contient l'état global de l'application. Il gère l'état, permet l'envoi des actions, et s'assure que les composants sont informés des changements d'état.

Redux

Un **slice** dans Redux regroupe l'état, les actions et les reducers pour une fonctionnalité donnée. Créons un slice pour un simple compteur :

```
import { createSlice } from '@reduxjs/toolkit';

const counterSlice = createSlice({
  name: 'counter',
  initialState: {
    value: 0,
  },
  reducers: {
    increment: (state) => {
      state.value += 1; // Vous pouvez modifier directement l'état grâce à Immer.
    },
    decrement: (state) => {
      state.value -= 1;
    },
    reset: (state) => {
      state.value = 0;
    },
    incrementByAmount: (state, action) => {
      state.value += action.payload;
    },
  },
});

export const { increment, decrement, reset, incrementByAmount } = counterSlice.actions;
export default counterSlice.reducer;
```

Redux

Maintenant, utilisons les hooks `useSelector` et `useDispatch` pour accéder à l'état et envoyer des actions depuis un composant.

```
import { useSelector, useDispatch } from 'react-redux';
import { increment, decrement, reset, incrementByAmount } from './counterSlice';

export function Counter() {
  const count = useSelector((state) => state.counter.value); // Accéder à l'état
  const dispatch = useDispatch(); // Pour envoyer des actions

  return (
    <div>
      <h1>Count: {count}</h1>
      <button onClick={() => dispatch(increment())}>Increment</button>
      <button onClick={() => dispatch(decrement())}>Decrement</button>
      <button onClick={() => dispatch(reset())}>Reset</button>
      <button onClick={() => dispatch(incrementByAmount(5))}>Increment by 5</button>
    </div>
  );
}
```

Le Composant Provider

- Le composant **Provider** est utilisé dans le fichier `index.js`. Il enveloppe l'ensemble de l'application pour rendre le store accessible à tous les composants enfants.

```
import { Provider } from 'react-redux';

const root = ReactDOM.createRoot(document.getElementById('root'));
root.render(
  <React.StrictMode>
    <Provider store={store}>
      <App />
    </Provider>,
  </React.StrictMode>
);
```

Exercice

Exercice 1 : Configuration de Redux avec `@reduxjs/toolkit`

1. Configure un store Redux avec `createSlice` pour gérer un compteur.

Consignes :

- Crée un slice `counter` avec les actions `increment` et `decrement`.
- Configure le store Redux avec ce slice.
- Crée un composant `Counter` qui affiche le compteur et deux boutons pour incrémenter et décrémenter le compteur.

Exercice Bonus : Gestion de la liste des tâches avec `@reduxjs/toolkit`

1. Configure un store Redux avec `createSlice` pour gérer les tâches.

Consignes :

- Crée un slice `tasks` avec les actions `addTask` et `removeTask`.
- Configure le store Redux avec ce slice.
- Crée un composant `TaskList` pour ajouter et afficher les tâches.

Introduction à Zustand

Contrairement à des solutions comme Redux, qui nécessitent plus de configuration, Zustand est simple à utiliser tout en restant puissante.

Pourquoi utiliser Zustand ?

- Simplicité : Pas besoin de configureStore, d'actions ou de reducers comme dans Redux.
- Hooks natifs : Utilise directement les hooks de React, ce qui rend son intégration naturelle.
- Performances : Très performant, avec une gestion fine des re-renders.

```
npm install zustand
```

Introduction à Zustand

- Création du store :

```
••• Uploaded using RayThis Extension

import create from 'zustand';

const useStore = create((set) => ({
  count: 0, // état initial
  increment: () => set({ count: state.count + 1 }),
  decrement: () => set({ count: state.count - 1 }),
}));
```

- Utiliser le store dans un component :

```
••• Uploaded using RayThis Extension

import useStore from './store'; // Importer le store Zustand

export function Counter () {
  const count = useStore((state) => state.count); // Sélectionner l'état
  const increment = useStore((state) => state.increment); // Sélectionner l'action
  const decrement = useStore((state) => state.decrement);

  return (
    <div>
      <h1>Count: {count}</h1>
      <button onClick={increment}>Increment</button>
      <button onClick={decrement}>Decrement</button>
    </div>
  );
}
```

8. Tests

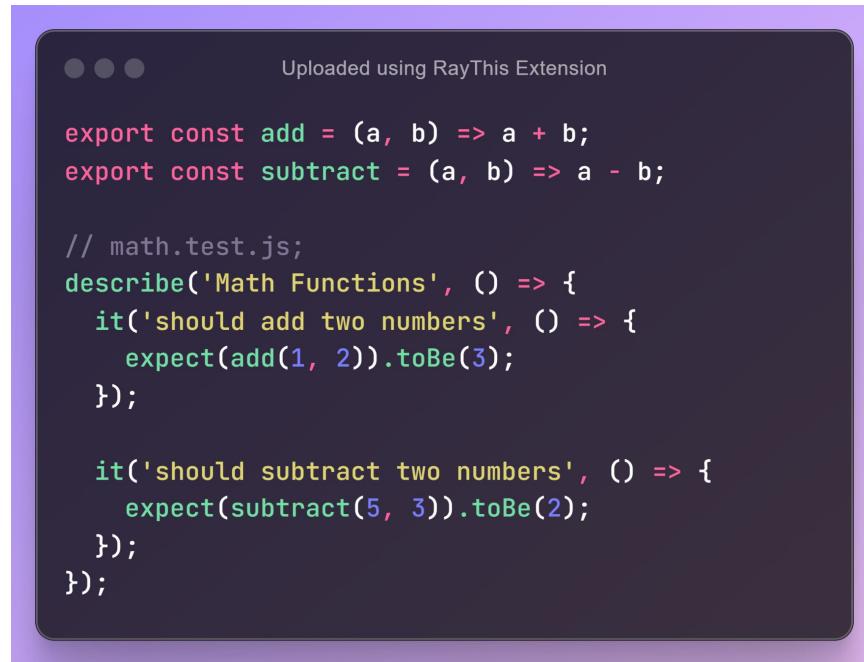
Les tests

- Les **tests unitaires** sont essentiels pour garantir la qualité et la *stabilité* de votre code.
- Ils permettent de vérifier qu'une petite portion de code (une unité, comme une fonction ou un composant) fonctionne correctement en *isolation*.

Concepts clés de Jest

- **describe** : Permet de regrouper des tests liés dans un bloc, souvent utilisé pour organiser les tests d'un composant ou d'une fonction.
- **it** : Définit un cas de test. Chaque cas de test doit vérifier un aspect spécifique du comportement.
- **expect** : Une assertion pour vérifier que le résultat correspond à ce qui est attendu.
- **setup et teardown** : Utilisés pour configurer l'environnement avant les tests (**beforeAll**, **beforeEach**) ou le nettoyer après (**afterAll**, **afterEach**).

- Exemple simple :
- npm run test



The screenshot shows a dark-themed code editor window with a purple header bar. The header bar has three dots on the left and the text "Uploaded using RayThis Extension" on the right. The main area contains the following Jest test code:

```
export const add = (a, b) => a + b;
export const subtract = (a, b) => a - b;

// math.test.js
describe('Math Functions', () => {
  it('should add two numbers', () => {
    expect(add(1, 2)).toBe(3);
  });

  it('should subtract two numbers', () => {
    expect(subtract(5, 3)).toBe(2);
  });
});
```

Exercice

Exercice 1 : Utilisation de Jest

1. Fonction de Modification de Texte :

```
function majuscule(str) {  
  
    return str.toUpperCase();  
  
}
```

Exercice 2 : Fonction `filterAndSortUsers` :

- **Description** : Filtre les utilisateurs pour ne conserver que ceux ayant un âge supérieur ou égal à `minAge`, puis trie les utilisateurs restants par nom en ordre alphabétique.
- **Paramètres** :
 - `users` : Un tableau d'objets utilisateurs, chacun avec un `name` et un `age`.
 - `minAge` : L'âge minimum requis pour qu'un utilisateur soit inclus dans le résultat.
- **Retour** : Un tableau d'objets utilisateurs filtrés et triés.

Tests :

- **Premier test** : Vérifie que les utilisateurs sont correctement filtrés par âge et triés par nom.
- **Deuxième test** : Vérifie le comportement lorsque aucun utilisateur ne correspond au critère d'âge (doit retourner une liste vide).
- Bonus : Faire la fonction `filterAndSortUsers` sans l'aide de la diapo suivante

```
/** * Filtre les utilisateurs par âge et les trie par nom.  
 * @param {Array} users - Liste d'utilisateurs, chaque utilisateur étant un objet avec  
 * `name` et `age`.  
 * @param {number} minAge - Âge minimum pour filtrer les utilisateurs.  
 * @returns {Array} Liste d'utilisateurs filtrés et triés.  
  
function filterAndSortUsers(users, minAge) {  
    return users .filter(user => user.age >= minAge) .sort((a, b) => {  
        a.name.localeCompare(b.name);  
    });  
}  
  
module.exports = filterAndSortUsers;
```

React Testing Library

Concepts clés de RTL

- **render** : Rendu du composant dans un environnement de test. Retourne des méthodes pour interagir avec le DOM (ex. `getByText`).
- **fireEvent** : Simule des événements (clics, changements d'input, etc.) sur des éléments.
- **cleanup** : Nettoie le DOM après chaque test pour éviter les effets de bord.

React Testing Library

- Exemple simple :
- npm run test

```
import '@testing-library/jest-dom'
import * as React from 'react'
import {render, fireEvent, screen} from '@testing-library/react'
import HiddenMessage from '../components/HiddenMessage'

test('shows the children when the checkbox is checked', () => {
  const testMessage = 'Test Message'
  render(<HiddenMessage>{testMessage}</HiddenMessage>

  expect(screen.queryByText(testMessage)).toBeNull()

  fireEvent.click(screen.getByLabelText(/show/i))

  expect(screen.getByText(testMessage)).toBeInTheDocument()
})
```

Exercice

Exercice 1 : Test d'un composant React simple avec React Testing Library

1. Crée un composant fonctionnel **Greeting** qui affiche un message de bienvenue avec le nom passé en prop.
2. Écris un test pour ce composant en utilisant Jest et React Testing Library pour vérifier que le message de bienvenue est correctement affiché.

Exercice 2 : Test d'un composant avec gestion d'état

1. Crée un composant fonctionnel **Counter** avec un bouton pour incrémenter un compteur.
2. Écris un test pour vérifier que le compteur augmente correctement lorsque le bouton est cliqué.

Exercice Bonus : Test d'un formulaire avec validation

1. Crée un composant fonctionnel **Form** avec un champ de texte et un bouton. Le formulaire doit afficher un message d'erreur si le champ est vide lorsque le bouton est cliqué.
2. Écris un test pour vérifier que le message d'erreur est affiché correctement lorsque le formulaire est soumis avec un champ vide.

9. React Native

React native

Points clés de React Native :

- **Multi-plateforme** : un seul code peut être utilisé pour iOS et Android
- **Utilise JavaScript et JSX** : proche de React pour le développement web, ce qui le rend accessible aux développeurs front-end
- **Performance native** : contrairement aux approches basées sur des vues Web (comme Cordova), React Native utilise des composants natifs, offrant ainsi une meilleure performance et expérience utilisateur
- **Rendu natif** : bien que vous écriviez en JavaScript, les composants sont transformés en éléments natifs des plateformes cibles.

React native

Points clés de React Native :

- **Multi-plateforme** : un seul code peut être utilisé pour iOS et Android
- **Utilise JavaScript et JSX** : proche de React pour le développement web, ce qui le rend accessible aux développeurs front-end
- **Performance native** : contrairement aux approches basées sur des vues Web (comme Cordova), React Native utilise des composants natifs, offrant ainsi une meilleure performance et expérience utilisateur
- **Rendu natif** : bien que vous écriviez en JavaScript, les composants sont transformés en éléments natifs des plateformes cibles.

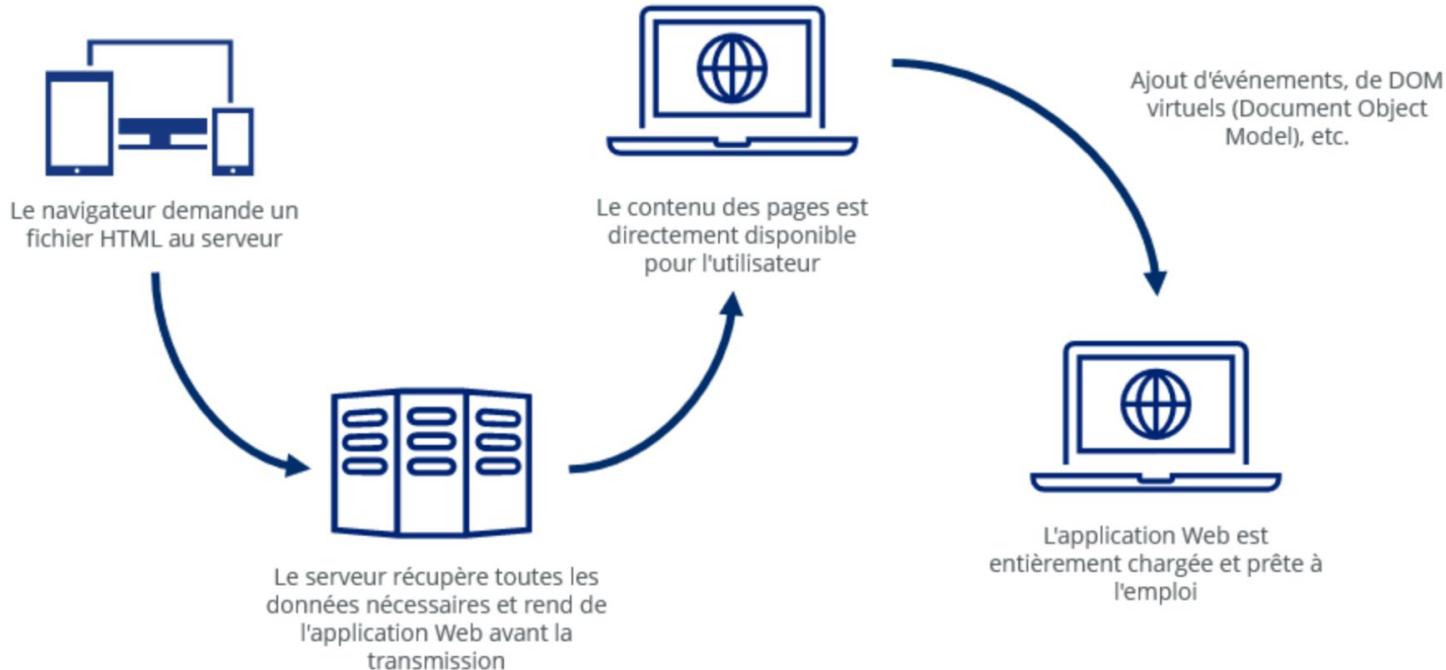
React native

Exemples d'applications créées avec React Native :

- Instagram
- Facebook
- Uber Eats
- Walmart
- Bloomberg

10. Server side rendering

Server Side Rendering



Avantages du Server Side Rendering

- **Amélioration des performances initiales :**

Les utilisateurs voient le contenu plus rapidement. Cela peut améliorer le temps de chargement initial par rapport au rendu côté client, où le navigateur doit d'abord télécharger JavaScript, l'exécuter et générer le contenu.

- **SEO (Optimisation pour les moteurs de recherche) :**

Les moteurs de recherche peuvent plus facilement indexer les pages rendues côté serveur. Les pages sont déjà prêtes à être explorées par les robots des moteurs de recherche, ce qui peut améliorer la visibilité et le classement dans les résultats de recherche.

- **Meilleure accessibilité :**

Les utilisateurs avec des connexions Internet lentes ou des appareils moins puissants peuvent avoir une meilleure expérience puisque le contenu est déjà rendu lorsqu'il arrive sur le client.

- **Amélioration de la performance perçue :**

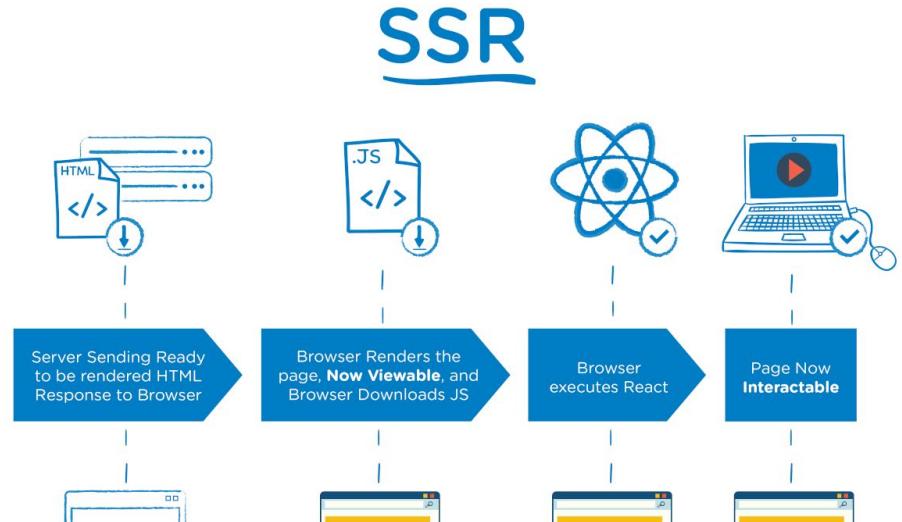
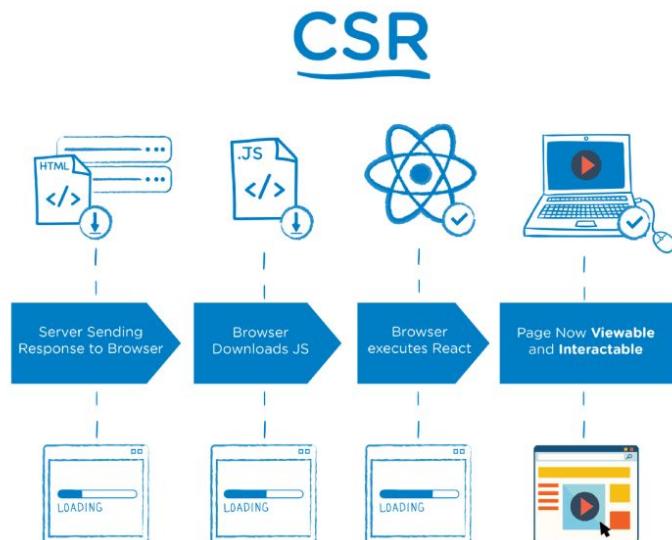
Les utilisateurs peuvent interagir avec le contenu plus rapidement, ce qui peut rendre l'application plus réactive et améliorer la perception globale de la performance.

- **Réduction du JavaScript initial :**

Avec SSR, moins de JavaScript est nécessaire pour le rendu initial de la page, ce qui peut réduire la charge sur le client et le temps nécessaire pour que la page devienne interactive.

Cas d'utilisation de Server Side Rendering

- Applications avec un contenu statique
- Sites avec des exigences SEO élevées
- Applications nécessitant un chargement rapide



Votre formateur

Clément Hamon

Formateur externe M2I

clement.hamon35@gmail.com

<https://www.linkedin.com/in/clément-hamon-135485209/>

Et encore merci !