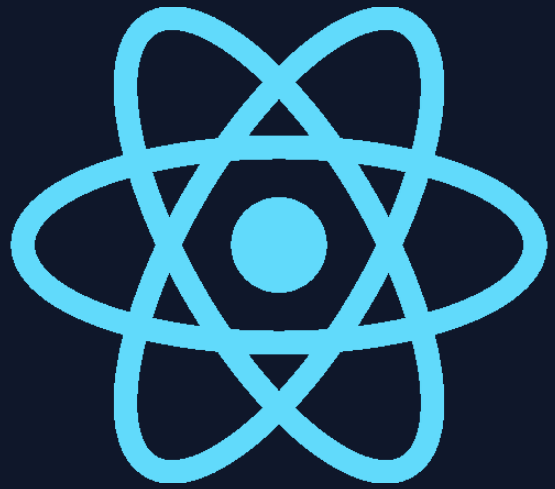


# Cours de ReactJS

## Module 1: Les Fondamentaux

Formateur : Kamel ABBASSI abbassi.kamel@gmail.com

# Module 1: Les Fondamentaux

Poser les bases de la pensée

React.

# Objectifs de ce Module

- Comprendre ce qu'est React et le problème qu'il résout (le V-DOM).
- Maîtriser la syntaxe **JSX** pour décrire les interfaces.
- Penser en **Composants** réutilisables.
- Utiliser les **Props** pour faire circuler les données.

# Qu'est-ce que React?

## Une Bibliothèque (UI)

React n'est *pas* un framework complet. C'est une bibliothèque dédiée à la construction d'interfaces utilisateur (la "Vue").

## Déclaratif

Vous décrivez à quoi ressemble l'interface à un instant T, et React se charge de la mettre à jour quand les données changent.

## Basé sur les Composants

L'UI est découpée en blocs réutilisables (composants) qui gèrent leur propre logique et apparence.

# Pourquoi React? Le DOM vs. Le Virtual DOM

## Le Problème: Le DOM est Lent

Le DOM (Document Object Model) est la structure de la page. Manipuler directement le DOM à chaque petit changement (comme taper du texte) est très coûteux en performances.

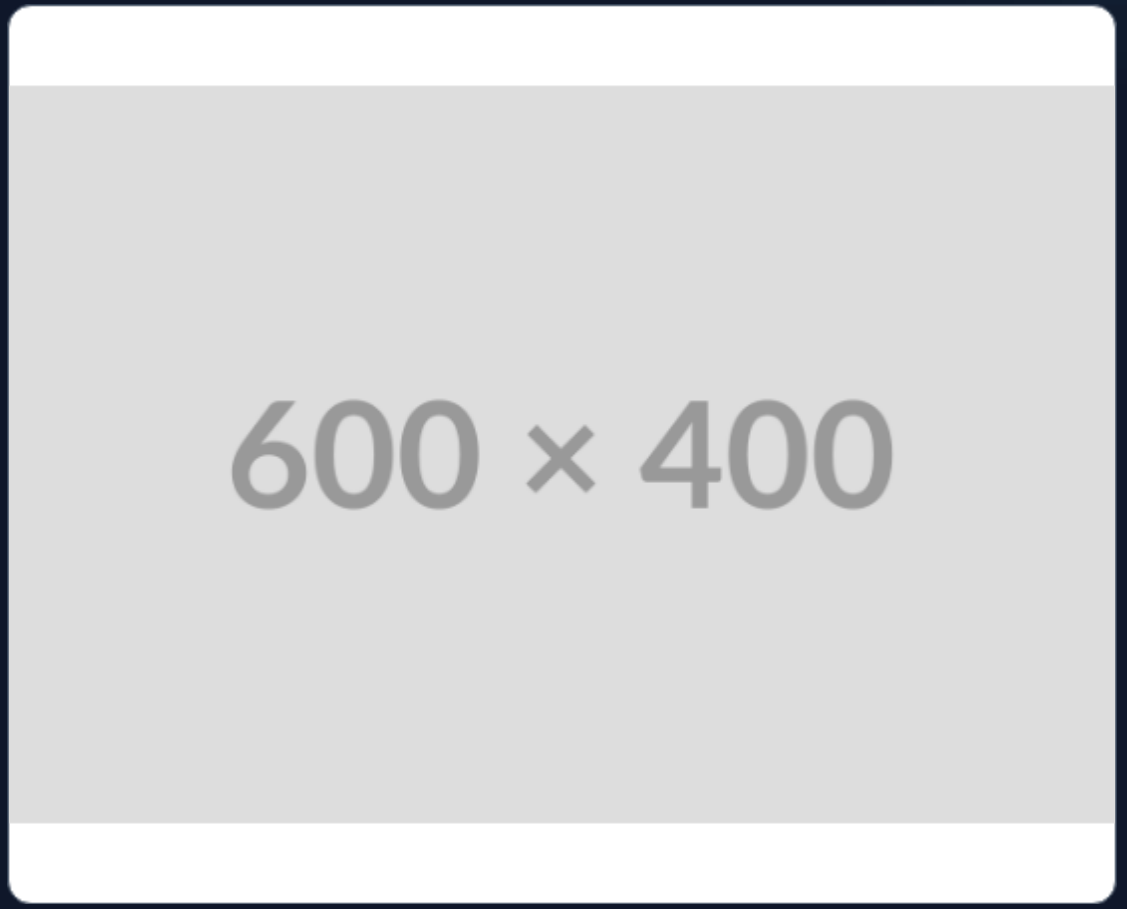
## La Solution: Le Virtual DOM

React garde une copie légère du DOM en mémoire (le "V-DOM"). Quand vos données changent, React met à jour ce V-DOM, le compare à l'ancien, et n'applique que les changements *minimaux* au DOM réel.

# Le Processus de "Réconciliation"

Ce processus s'appelle la **réconciliation**.

1. Vos données (l'état) changent.
2. React crée un nouveau Virtual DOM.
3. React "diff" (compare) l'ancien V-DOM et le nouveau.
4. Il identifie le plus petit lot de changements nécessaires.
5. Il applique ces changements au DOM réel, en une seule fois.



# Partie 1: Le JSX

La syntaxe pour décrire l'interface utilisateur.

# JSX: JavaScript + XML

Le JSX est une extension de syntaxe pour JavaScript. Il ressemble à du HTML, mais il est en fait transformé en JavaScript pur (React.createElement()) par un outil comme Babel.

Il permet de co-localiser la logique (JS) et la structure (HTML-like) dans un même fichier: le composant.

```
// Ce que vous écrivez (JSX)
const element =  className="salut">Bonjour;

// Ce que Babel génère (JS Pur)
const element = React.createElement(
  'h1',
  { className: 'salut' },
  'Bonjour'
);
```

# Règles du JSX (vs. HTML)

- **Attributs en camelCase:** class devient className. onclick devient onClick. for devient htmlFor.
- **Un seul élément racine:** Un composant doit toujours retourner un seul élément. Utilisez <>... (Fragment) pour envelopper.
- **Fermeture des balises:** Toutes les balises doivent être fermées (ex:  ou ).
- **Expressions JS avec {...}:** Utilisez les accolades pour insérer des variables ou des expressions.

# JSX: Rendu Conditionnel

On n'utilise pas if/else *dans* le JSX. On utilise des opérateurs ternaires ou la logique &&.

## Opérateur Ternaire

Idéal pour un if...else... simple.

## Opérateur &&

Idéal pour "afficher si vrai, sinon ne rien afficher".

```
function AuthButton({ isLoggedIn, unreadMessages }) {
  return (

      {/* Ternaire */}

        {isLoggedIn ? 'Bienvenue!' : 'Connectez-vous'}


      {/* Logique && */}
      {unreadMessages.length > 0 &&

          Vous avez {unreadMessages.length} messages.

      }

  );
}
```

# JSX: Rendu de Listes (.map())

On utilise la fonction .map() pour transformer un tableau de données en un tableau d'éléments JSX.

## L'attribut key

C'est **crucial**. React utilise la key pour identifier quel élément a changé, été ajouté ou supprimé.

Utilisez un ID unique (comme un ID de BDD), et *jamais* l'index du tableau (sauf en dernier recours).

```
const products = [
  { id: 'p1', name: 'Pomme' },
  { id: 'p2', name: 'Orange' },
];

function ProductList() {
  return (

      {products.map(product ⇒ (
         key={product.id}>
          {product.name}

      ))}

  );
}
```

# Partie 2: Les Composants

Les briques fondamentales de

React.

# Penser en Composants

## Découper l'interface

L'idée centrale de React est de découper une interface complexe en petits morceaux indépendants.

Pensez-y comme des briques de Lego. Chaque brique (Button, Header, ProfileCard) est indépendante, mais vous pouvez les assembler pour construire n'importe quoi.

# Anatomie d'un Composant Fonctionnel

Aujourd'hui, on écrit les composants React comme de simples fonctions JavaScript.

## Règles:

1. Le nom de la fonction **doit** commencer par une Majuscule (ex: MyComponent).
2. La fonction **doit** retourner du JSX (ou null).

```
// 1. Une fonction JS ...
// 2. ... qui commence par une Majuscule ...
function WelcomeMessage() {

  // (Logique JS ici ... )

  // 3. ... et retourne du JSX.
  return Bonjour, monde !;
}

// Utilisation du composant
function App() {
  return ;
}
```

# Composant vs. Élément vs. Instance

## Composant

C'est la "recette". C'est le code lui-même (la fonction ou la classe).

function Button() {...}

## Élément

C'est la "description" de ce qui doit être affiché. Un objet JS léger retourné par le JSX.

## Instance

C'est l'objet "vivant" géré par React, avec son propre état et cycle de vie. (Concept plus abstrait).

# Partie 3: Les Props

Passer des données aux composants.

# Que sont les "Props"?

## Des Arguments pour Composants

"Props" (propriétés) est la façon dont les composants communiquent. Les données "coulent" du parent vers l'enfant.

Si un composant est une "recette" (fonction), les props sont ses "ingrédients" (arguments).

Le flux de données est **unidirectionnel** (Top -> Down).

# Props en Action (Code)

**1. Parent:** Le parent *passe* les props comme des attributs HTML.

**2. Enfant:** L'enfant *reçoit* les props comme un objet, premier argument de la fonction.

## Déstructuration

Il est très courant de "déstructurer" l'objet props directement dans les arguments de la fonction pour un code plus propre.

```
// 1. Parent
function App() {
  return  name="Ana" age={30} />;
}

// 2. Enfant (recevant l'objet props)
function Greeting(props) {
  // props = { name: "Ana", age: 30 }
  return (
    Bonjour {props.name}! Vous avez {props.age} ans.
  );
}

// 3. Enfant (avec déstructuration)
function Greeting({ name, age }) {
  return (
    Bonjour {name}! Vous avez {age} ans.
  );
}
```

# Règles des Props

- **Lecture Seule (Immutabilité):** Un composant ne doit **jamais** modifier les props qu'il reçoit. C'est une règle fondamentale de React.
- **Flux Unidirectionnel:** Les données descendent toujours du parent vers l'enfant.
- **Validation (PropTypes):** Dans les gros projets, on utilise des bibliothèques (comme PropTypes ou TypeScript) pour s'assurer qu'un composant reçoit les bons types de props.

# La Prop Spéciale: children

props.children est une prop spéciale. Elle contient tout ce que vous placez *entre* les balises ouvrante et fermante d'un composant.

C'est très utilisé pour créer des "boîtes" ou des "conteneurs" génériques (Layouts, Cards, Modals).

```
// Composant "Card" (Enfant)
function Card({ title, children }) {
  return (
     className="card">
       className="card-title">{title}
       className="card-content">
        {children}  ⟵ Affiche le contenu


  );
}

// Utilisation (Parent)
function App() {
  return (
     title="Mon Profil">
      Ceci est mon profil.  ⟵ devient props.children
       src="..." />         ⟵ devient aussi props.chil

  );
}
```

# Fin du Module 1

Prochaine étape: Le Module 2 (État & Hooks)

# Module 2: État, Hooks & Cycle de Vie

Rendre vos composants interactifs et réactifs au temps.

# Props vs. State : Quelle est la Différence ?

## Props (Propriétés)

**Rôle:** Passer des données du parent à l'enfant (Arguments de la fonction).

**Mutabilité:** Immuables (Lecture Seule). L'enfant ne peut pas les modifier.

**Source:** Définies par le composant Parent.

**Exemple:** Le nom d'utilisateur reçu par un composant Card.

## State (État)

**Rôle:** Gérer des données privées *internes* au composant pour le rendre interactif.

**Mutabilité:** Modifiable. Le composant gère et change son propre état.

**Source:** Défini et géré par le composant lui-même.

**Exemple:** La valeur actuelle d'un compteur, ou si un modal est ouvert.

# Partie 1: Le Hook `useState`

Le moyen le plus simple d'ajouter de l'état local.

# Anatomie de `useState`

Le Hook `useState` est la fonction qui permet aux composants fonctionnels de conserver un état interne. Il retourne un tableau à deux éléments (déstructuration).

## 1. La valeur courante

La variable d'état actuelle (ex: `count`).

## 2. La fonction de mise à jour

Une fonction pour changer cette valeur (ex: `setCount`).

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

function Counter() {
  // [ Valeur, Fonction de mise à jour ] = useState(Vale
  const [count, setCount] = useState(0);

  return (

      Le compteur est à : {count}

      // Utilisation de la fonction de mise à jour
       onClick={() ⇒ setCount(count + 1)}>
        Incrémenter


  );
}
```

# Pourquoi utiliser `setCount` ?

- **Re-rendu:** Changer directement `count = 10` ne fonctionnerait pas. Seul `setCount()` demande à React de **re-rendre** le composant avec la nouvelle valeur.
- **Asynchrone:** La mise à jour de l'état est asynchrone (React la "batch" pour optimiser). Après `setCount()`, la variable `count` n'a pas encore la nouvelle valeur.
- **Isolé:** Chaque appel à `useState` crée une paire état/mise à jour indépendante pour cette instance du composant.

# Forme fonctionnelle de la mise à jour

Lorsque la nouvelle valeur dépend de l'ancienne (comme un compteur), utilisez la forme fonctionnelle de `setCount`.

## Pourquoi ?

Ceci garantit que vous utilisez la **dernière** valeur de l'état, surtout si l'état est mis à jour rapidement ou dans des opérations asynchrones. React vous fournit la valeur à jour dans cet argument.

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

  const incrementTwice = () => {
    // Mauvaise pratique : utilise l'état potentiellemme
    setCount(count + 1);
    setCount(count + 1); // Ne fera qu'incrémenter de 1

    // Bonne pratique : garantie d'utiliser la valeur pr
    setCount(prevCount => prevCount + 1);
    setCount(prevCount => prevCount + 1); // Fera bien +
  }

  return ( ... );
}
```

# Gestion de l'état: Objets et Tableaux (Immutabilité)

Pour mettre à jour un objet ou un tableau, vous **devez** créer une nouvelle copie de l'état. C'est l'immutabilité.

React se base sur la **référence** de l'objet ou du tableau : si l'objet est le même, React ne re-rend pas.

## Techniques:

Utilisez l'opérateur de propagation (`...`) ou les méthodes de tableau qui retournent une nouvelle copie (ex: `.map`, `.filter`).

```
const [user, setUser] = useState({ name: 'Ana', count: 0

// Mettre à jour l'objet (Bonne Pratique)
const incrementCount = () => {
  setUser(prevUser => ({
    // Copie de toutes les propriétés existantes
    ...prevUser,
    // Écrasement de la propriété à modifier
    count: prevUser.count + 1,
  }));
};

// Mettre à jour un tableau (Ex: Ajouter un élément)
const [items, setItems] = useState(['A', 'B']);

const addItem = () => {
  const newItem = 'C';
  setItems(prevItems => [...prevItems, newItem]);
};
```

# Partie 2: Le Hook `useEffect`

Gérer les effets secondaires (Side Effects).

# Qu'est-ce qu'un "Effet de Bord"?

- **Appels API:** Récupérer des données (`fetch`, `axios`).
- **Manipulation du DOM:** Changer le titre du document, se concentrer sur un champ.
- **Abonnements:** Écouter les événements du navigateur ou des WebSockets.
- **Timers:** Définir des intervalles ou des délais (`setTimeout`, `setInterval`).

# Anatomie de `useEffect`

`useEffect` prend deux arguments principaux :

## 1. La fonction d'effet

C'est là que vous placez votre code d'effet de bord (ex: l'appel API). Elle s'exécute **après** le rendu.

## 2. Le tableau de dépendances

C'est la clé de `useEffect`. Il contrôle **quand** l'effet doit s'exécuter à nouveau.

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

function DataFetcher({ userId }) {
  const [data, setData] = useState(null);

  useEffect(() => {
    // --- 1. Fonction d'effet (le Side Effect) ---
    async function fetchData() {
      // Ex: Appel API
      const response = await fetch(`/api/users/${userId}
      const result = await response.json();
      setData(result);
    }
    fetchData();
  }, [userId]); // --- 2. Tableau de dépendances ---

  return ( ... );
}
```

# Les 3 Cas d'Utilisation de `useEffect`

- **Cas 1: Exécuter après chaque rendu (Rare)**

- **Cas 2: Exécuter une seule fois au montage (API Initiale)**

- **Cas 3: Exécuter lors d'un changement spécifique (Le plus courant)**

# Nettoyage (Clean-up) de `useEffect`

Certains effets de bord doivent être nettoyés pour éviter les fuites de mémoire (memory leaks) ou les comportements inattendus.

L'effet peut retourner une fonction. Cette fonction de "nettoyage" est exécutée juste avant que l'effet ne soit ré-exécuté, ou lorsque le composant est **démonté** (retiré du DOM).

```
function TimerComponent() {
  useEffect(() => {
    // 1. L'effet : Démarrage d'un Timer
    const timerId = setInterval(() => {
      console.log("Le timer s'exécute ... ");
    }, 1000);

    // 2. La fonction de nettoyage (return)
    return () => {
      clearInterval(timerId); // Stoppe le timer
      console.log("Le timer est stoppé.");
    };
  }, []); // Dépendance vide : s'exécute uniquement au m

  return Timer actif;
}
```

# Quand Faut-il Nettoyer?

- **Timers:** Toujours nettoyer `setTimeout` et `setInterval`.
- **Listeners:** Toujours retirer les écouteurs d'événements (`window.removeEventListener`).
- **Abonnements:** Toujours se désabonner des WebSockets ou autres flux de données.
- **NON Nécessaire:** Les appels API (le navigateur les nettoie), la modification du titre du document, l'état local.

# Partie 3: Cycle de Vie et Hooks

Comprendre l'existence du composant.

# Les 3 Phases du Cycle de Vie

## 1. Montage (Mounting)

Le composant est créé et inséré dans le DOM (affichage initial).
**Hook associé:** `useEffect(..., [])`

## 2. Mise à Jour (Updating)

Les props ou l'état interne changent, ce qui force un re-rendu.
**Hook associé:** `useEffect(..., [deps])`

## 3. Démontage (Unmounting)

Le composant est retiré du DOM (ex: navigation sur une autre page).
**Hook associé:** La fonction de nettoyage de `useEffect` (`return () => {...}`).

# Représentation du Cycle de Vie

Les Hooks ont remplacé les méthodes de cycle de vie des anciennes Class Components (ex: `componentDidMount`, `componentDidUpdate`, `componentWillUnmount`).

Avec les Hooks, les logiques liées (initialisation et nettoyage) sont regroupées dans un seul `useEffect`, ce qui rend le code plus clair et moins sujet aux erreurs.

# Règle d'Or des Hooks

## Règle 1: Appeler au niveau supérieur

N'appelez les Hooks **qu'au niveau racine** d'un composant fonctionnel ou d'un Custom Hook. Jamais dans des conditions (`if`), des boucles (`for`) ou des fonctions imbriquées.

## Règle 2: Appeler à partir de Composants

N'appelez les Hooks React (comme `useState`, `useEffect`, etc.) qu'à partir de **Composants Fonctionnels React** ou de **Hooks Personnalisés**.

# Pourquoi les Règles d'Or?

- **Ordre:** React gère l'état et les effets en interne à l'aide d'un tableau indexé.
- **Cohérence:** Si vous appelez des Hooks conditionnellement, le tableau interne de React serait dans un ordre différent d'un rendu à l'autre, ce qui mènerait au chaos.
- **Outils:** Il existe des outils comme l'extension **ESLint Hook Rules** qui appliquent ces règles et vous avertissent automatiquement.

# Fin du Module 2

Prochaine étape: Le Module 3 (Gestion des Événements & Formulaires)

# Module 3: Gestion des Événements & Formulaires

Capturer les interactions utilisateur et gérer les entrées.

# Événements dans React : Conventions JSX

## Nommage en CamelCase

Dans HTML, les événements sont en minuscules (ex: onclick). En React (JSX), ils sont en **CamelCase** (ex: onClick, onChange).

**Exemple:**

```
// HTML (Mauvais en React)
 onclick="handleClick()">

// React (Bon)
 onClick={handleClick}>
```

## Fonctions comme Handlers

Au lieu d'une chaîne de caractères (comme en HTML pur), vous passez une **fonction JavaScript** comme gestionnaire d'événement (event handler).

**Exemple:**

```
function MyButton() {
  const handleClick = () => {
    alert('Clic détecté!');
  };
  return ( onClick={handleClick}>
      Cliquez-moi
  );
}
```

# L'Objet Événement Synthétique

Lorsqu'un événement est déclenché, React passe un objet **SyntheticEvent** au gestionnaire de fonction.

Cet objet enveloppe l'événement natif du navigateur, assurant que le comportement de l'événement est **cohérent** sur tous les navigateurs (Chrome, Firefox, Safari, etc.).

## Méthodes clés:

e.preventDefault(): Empêche l'action par défaut du navigateur (ex: soumission de formulaire).
e.stopPropagation(): Arrête la propagation de l'événement vers les parents.

```
function Button() {
  // L'événement (e) est l'objet SyntheticEvent
  const handleLinkClick = (e) ⇒ {
    // Empêche le navigateur de naviguer (action par déf
    e.preventDefault();
    console.log("Navigation bloquée par React.");
  };

  return (
     href="https://reactjs.org" onClick={handleLinkClick
      Lien vers React

  );
}
```

# Passer des arguments aux gestionnaires

Parfois, vous devez passer des données supplémentaires au gestionnaire d'événement (ex: l'ID d'un élément à supprimer).

## Méthode recommandée : Fonction Fléchée

Utilisez une fonction fléchée anonyme inline pour envelopper l'appel à la fonction de gestionnaire. Cela garantit que la fonction n'est appelée qu'au clic, et non lors du rendu.
L'objet événement `e` est automatiquement passé comme dernier argument par React si vous le définissez.

```
function ItemList({ items }) {

  const handleDelete = (itemId, event) => {
    console.log(`Suppression de l'élément: ${itemId}`);
    console.log(`Événement: ${event.type}`);
  };

  return (

      {items.map(item => (
         key={item.id}>
          {item.name}
          // Fonction fléchée pour passer l'ID
           onClick={(e) => handleDelete(item.id, e)}>
            X


      ))}

  );
}
```

# Partie 2: Les Formulaires et le State

Transformer les formulaires HTML en données utilisables.

# Composants Contrôlés (Controlled Components)

## Définition

En React, un "Controlled Component" est un élément de formulaire dont la valeur est gérée par l'**état React** (`useState`).
Le State devient la "source de vérité" unique pour la donnée.

## Flux de données (Data Flow)

Le composant rend l'élément de formulaire.

1. L'utilisateur tape (événement `onChange`).
2. Le gestionnaire `onChange` met à jour l'état (`setState`).
3. L'état mis à jour force un re-rendu.
4. La propriété `value` du champ reflète le nouvel état.

Pour qu'un champ soit contrôlé, il doit avoir deux propriétés clés:

## 1. `value`

Liaison à la variable d'état.

## 2. `onChange`

Appelle la fonction de mise à jour de l'état chaque fois que la valeur du champ change (la frappe de l'utilisateur).

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

function NameForm() {
  const [name, setName] = useState('');

  // Le gestionnaire d'événement
  const handleChange = (e) => {
    // e.target.value contient la nouvelle valeur du cha
    setName(e.target.value);
  };

  return (


        Nom:

          type="text"
          value={name} // 1. La source de vérité
          onChange={handleChange} // 2. Mise à jour de l
        />

      Vous avez tapé : {name}
  );
```

# Les différents types d'entrées

- **Entrée de texte (``):** Utilise la prop `value` et l'événement `onChange`.
- **Sélection (``):** Le `select` gère la prop `value` sur la balise parente, au lieu de définir `selected` sur l'option.
- **Case à cocher (``):** Utilise la prop `checked` (booléen) au lieu de `value`. L'événement `onChange` met à jour `checked`.

# Gestion de la Soumission de Formulaire

L'événement de soumission est déclenché par le `` lorsqu'un bouton de type `submit` est cliqué.

Le gestionnaire doit prendre l'objet événement `e` en argument pour pouvoir appeler e.preventDefault(), ce qui est **essentiel** pour empêcher le rechargement de la page par le navigateur.

```
function SubmitForm() {
  const [email, setEmail] = useState('');

  // 1. Le gestionnaire de soumission
  const handleSubmit = (e) => {
    e.preventDefault(); // ESSENTIEL: Empêche le recharg
    alert(`Soumission de l'email: ${email}`);
    setEmail(''); // Réinitialisation du formulaire
  };

  return (
     onSubmit={handleSubmit}>

        type="email"
        value={email}
        onChange={(e) => setEmail(e.target.value)}
        placeholder="Entrez votre email"
      />
       type="submit">S'inscrire

  );
}
```

# Partie 3: Composants Non Contrôlés

Utiliser les Refs pour accéder directement au DOM.

# Composants Non Contrôlés (Uncontrolled Components)

- **Définition:** Les données sont gérées par le DOM lui-même, pas par l'état React.
- **Accès:** Vous accédez à la valeur du champ directement après la soumission du formulaire, en utilisant le Hook `useRef`.
- **Cas d'usage:** Idéal pour les formulaires simples, les fichiers de type `file input`, ou lorsque vous voulez intégrer du code non-React.
- **Inconvénient:** Moins de contrôle en temps réel (validation instantanée, modification programmée du champ).

# contrôlé

Le Hook `useRef` vous permet de créer une référence qui peut être attachée à un élément DOM (comme un ``).

## Étapes:

1. Déclarez une Ref : const inputRef = useRef(null);

2. Attachez-la au champ :

3. Accédez à la valeur : inputRef.current.value lors de la soumission.

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

function UncontrolledForm() {
  // 1. Création de la Ref
  const nameInputRef = useRef(null);

  const handleSubmit = (e) => {
    e.preventDefault();

    // 3. Accès à la valeur via la Ref
    const name = nameInputRef.current.value;
    console.log(`Nom soumis (via Ref): ${name}`);
  };

  return (
     onSubmit={handleSubmit}>

        type="text"
        defaultValue="Prénom par défaut" // Utiliser def
        ref={nameInputRef} // 2. Attachement de la Ref
      />
        type="submit">Soumettre
```

# Recommandation : Composants Contrôlés vs Non Contrôlés

- **Privilégiez les Composants Contrôlés:** Dans la plupart des cas, les composants contrôlés sont la méthode React idiome, offrant une meilleure gestion de la validation et des modifications d'état en temps réel.
- **Formulaires Complexes:** Pour les formulaires très complexes ou multiples, des bibliothèques externes comme **Formik** ou **React Hook Form** sont recommandées pour simplifier la gestion de l'état et de la validation.
- **Cas Spéciaux:** Utilisez non contrôlés uniquement pour les cas où vous avez besoin d'un accès direct au DOM (ex: gestion des fichiers, intégration d'un plugin jQuery).

# Fin du Module 3

Prochaine étape: Le Module 4 (Routing et Navigation)

# Module 4: Routing et Navigation

Créer des applications multi-pages fluides avec React Router.

# Qu'est-ce que le Client-Side Routing ?

## Définition

C'est le mécanisme par lequel l'application gère les changements d'URL (routes) sans que le navigateur n'ait besoin de recharger toute la page (Single Page Application - SPA).

En React, un changement de route affiche simplement un nouvel ensemble de composants.

## Avantages des SPA

- **Rapidité:** Une fois les ressources initiales chargées, la navigation est instantanée.
- **Expérience Utilisateur:** Permet des transitions fluides et des états persistants (ex: un lecteur de musique ne s'arrête pas entre les pages).
- **API:** Utilise l'API History du navigateur (pushState, replaceState) pour mettre à jour l'URL sans requête serveur complète.

# React Router : La librairie incontournable

- **Utilité:** C'est la bibliothèque la plus populaire pour gérer le routage déclaratif dans les applications React.
- **Installation:** Nécessite une installation externe (ex: npm install react-router-dom).
- **Composants Clés:** Elle fournit des composants spéciaux pour lier l'état de l'application à l'URL.

Pour utiliser le routage, il faut envelopper l'application dans un "routeur". `BrowserRouter` est le plus couramment utilisé.

## BrowserRouter

Le routeur parent qui utilise l'API History du navigateur pour garder l'interface utilisateur synchronisée avec l'URL.

## Routes et Route

est un conteneur qui sélectionne la première `` qui correspond à l'URL actuelle.

mappe un chemin (`path`) vers un élément React

```
import { BrowserRouter, Routes, Route } from 'react-rout

// Composants de page (simulés)
const Home = () ⇒ Accueil;
const About = () ⇒ À Propos;
const NotFound = () ⇒ 404 - Page Non Trouvée;

function App() {
  return (


        // Route 1: Correspond à l'URL '/'
         path="/" element={ />} />

        // Route 2: Correspond à l'URL '/about'
         path="/about" element={ />} />

        // Route par défaut (Catch-all)
         path="*" element={ />} />
```

# Navigation avec le composant `Link`

## vs

N'utilisez **jamais** la balise HTML  pour la navigation interne dans une application React Router. Elle provoquerait un rechargement complet de la page, annulant l'effet SPA.
Le composant  de React Router intercepte le clic et gère le changement de route sans rechargement.

## Syntaxe

Le composant  remplace l'attribut href par to.

```
// Mauvais (recharge la page)
 href="/about">

// Bon (navigation SPA)
 to="/about">

// Lien vers l'Accueil
 to="/">
```

# Routes Paramétrées (Dynamic Routing)

Pour afficher les détails d'un produit, d'un utilisateur ou d'un article, vous utilisez des paramètres d'URL (ex: /products/123).

## Définir le Paramètre

Dans le path, utilisez le symbole deux-points (:) suivi du nom du paramètre (ex: :id).

## Accéder au Paramètre

Utilisez le hook useParams() dans le composant ciblé pour récupérer la valeur du paramètre.

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

// Composant qui affiche les détails
const ProductDetail = () => {
  // Récupère l'objet des paramètres { id: 'valeur' }
  const { id } = useParams();

  return (Détails du Produit #{id});
};

// Définition de la Route

  // Le chemin attend un paramètre nommé "id"
   path="/products/:id"
         element={ />} />
```

# Navigation Programmatique avec `useNavigate`

Parfois, la navigation n'est pas déclenchée par un clic sur un lien, mais par une action (ex: soumission de formulaire, succès d'une requête API, timeout).

## Le Hook `useNavigate`

Ce hook renvoie une fonction (souvent appelée `Maps`) qui vous permet de changer l'URL de manière programmatique.
Syntaxe: navigate('/chemin/cible')

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

function RedirectButton() {
  // 1. Initialiser la fonction de navigation
  const navigate = useNavigate();

  const handleLoginSuccess = () => {
    console.log("Authentification réussie ... ");

    // 2. Exécuter la navigation vers le tableau de bord
    navigate('/dashboard');
  };

  return (
     onClick={handleLoginSuccess}>
      Se Connecter

  );
}
```

# Routes Imbriquées (Nested Routes)

- **Concept:** Les routes imbriquées permettent à un composant parent de déterminer une partie de l'interface utilisateur, tandis que les routes enfants déterminent l'autre partie.
- **Exemple:** La navigation (sidebar) de la zone "Dashboard" reste constante, mais la zone de contenu principale change (/dashboard/profile, /dashboard/settings).
- **Composant Clé:** Vous utilisez le composant  dans le composant parent pour indiquer où les routes enfants doivent être rendues.

L'imbrication permet de partager la mise en page (Layout) entre les routes enfants.

Le composant `DashboardLayout` rend un `Navbar` et le `` où React Router injectera soit `Profile`, soit `Settings` en fonction de l'URL.

```
import { Route, Outlet } from 'react-router-dom';

// Composant de mise en page parent
const DashboardLayout = () => (

    ... Menu Dashboard ...
    // Ici l'enfant sera rendu
     />

);

// Définition des Routes

  // Route Parent: Définit le Layout pour les enfants
   path="/dashboard" element={ />}>

    // Route Enfant 1: Correspond à '/dashboard/profile'
     path="profile" element={ />} />

    // Route Enfant 2: Correspond à '/dashboard/settings
     path="settings" element={ />} />

    // Index Route: Correspond à '/dashboard' (layout se
     index element={ />} />
```

# Autres Hooks Utiles de React Router

- **useLocation():** Renvoie l'objet de localisation qui contient des informations sur l'URL actuelle (pathname, search, state). Utile pour déclencher des actions lors du changement de page.
- **useSearchParams():** Permet de lire et de modifier les paramètres de requête de l'URL (query parameters, ex: ?sort=price&filter=new).
- **useNavigate() pour l'historique:** Peut être utilisé pour revenir en arrière dans l'historique du navigateur: navigate(-1).

# Fin du Module 4

Prochaine étape: Le Module 5 (Fetch de Données et Hooks Avancés)

# Module 5: Fetch de Données et Hooks Avancés

Maîtriser la gestion des données et optimiser les performances.

# Partie 1: Stratégies de Fetch de Données

Interagir avec le monde extérieur (APIs, serveurs).

# Rappel: Fetch de données avec `useEffect`

- **Où:** Les appels API sont des effets de bord (Side Effects) et doivent donc être placés dans un `useEffect`.
- **Quand:** Pour un appel unique au chargement, utilisez un tableau de dépendances vide : `useEffect(..., [])`.
- **Le Problème:** Un simple `fetch` ne gère ni l'état de chargement (`loading`) ni les erreurs (`error`), ce qui est crucial pour une bonne UX.

Pour gérer correctement un appel API, vous avez besoin de 3 états distincts.

## 1. `data`

Pour stocker le résultat en cas de succès.

## 2. `loading`

Un booléen pour savoir si l'appel est en cours (utile pour afficher un spinner).

## 3. `error`

Pour stocker l'objet d'erreur en cas d'échec.

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

function UserList() {
  // 1. Déclarer les 3 états
  const [data, setData] = useState(null);
  const [loading, setLoading] = useState(true);
  const [error, setError] = useState(null);

  useEffect(() => {
    const fetchData = async () => {
      try {
        const response = await fetch('/api/users');
        if (!response.ok) throw new Error('Erreur réseau
        const result = await response.json();
        setData(result);
      } catch (err) {
        setError(err);
      } finally {
        setLoading(false); // Toujours stopper le charge
      }
    };
    fetchData();
  }, []);

  // 2. Gérer les 3 états dans le Rendu
  if (loading) return Chargement ... ;
  if (error) return Erreur: {error.message};
```

Que se passe-t-il si l'utilisateur quitte la page (démontage du composant) avant la fin de l'appel API ?

React vous avertira d'une "fuite de mémoire" car le setState est appelé sur un composant qui n'existe plus.

## La Solution: AbortController

Utilisez un AbortController pour annuler la requête fetch dans la fonction de nettoyage de `useEffect`.

```
useEffect(() => {
  // 1. Créer un contrôleur
  const controller = new AbortController();
  const signal = controller.signal;

  const fetchData = async () => {
    try {
      // 2. Passer le signal au fetch
      const res = await fetch('/api/data', { signal });
      // ... suite
    } catch (err) {
      if (err.name === 'AbortError') {
        console.log('Fetch annulé!');
      } else {
        setError(err);
      }
    } finally { /* setLoading(false) */ }
  };
  fetchData();

  // 3. Annuler dans la fonction de nettoyage
  return () => {
    controller.abort();
  };
```

# Partie 2: Les Hooks Personnalisés (Custom Hooks)

La solution pour une logique réutilisable.

# Qu'est-ce qu'un Hook Personnalisé ?

- **Définition:** C'est simplement une fonction JavaScript dont le nom commence par use.
- **Rôle:** Elle vous permet d'extraire la logique d'état (stateful logic) d'un composant pour la réutiliser.
- **Capacités:** Un Hook personnalisé peut appeler d'autres Hooks (useState, useEffect, useContext, ou même d'autres hooks personnalisés).

  **Le Problème Résolu:** La logique de fetch (data, loading, error, cleanup) est répétitive. Nous pouvons l'encapsuler.

Nous allons prendre toute la logique de la diapositive précédente et la placer dans une fonction réutilisable useFetch(url).

Cette fonction prend l'URL en argument et retourne les 3 états : data, loading, et error.

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

// La fonction commence par 'use'
function useFetch(url) {
  const [data, setData] = useState(null);
  const [loading, setLoading] = useState(true);
  const [error, setError] = useState(null);

  useEffect(() => {
    const controller = new AbortController();

    const fetchData = async () => {
      setLoading(true);
      try {
        const res = await fetch(url, { signal: controlle
        const json = await res.json();
        setData(json);
        setError(null);
      } catch (err) {
        if (err.name !== 'AbortError') setError(err);
      } finally {
        setLoading(false);
      }
    };
    fetchData();
    return () => controller.abort();
  }, [url]); // Dépend de l'URL
```

# Consommer le Hook `useFetch`

Maintenant, notre composant UserList est incroyablement propre et déclaratif.

Toute la logique complexe est cachée dans useFetch, et le composant se contente de gérer l'affichage.

```
// Fichier: useFetch.js (Défini précédemment)

// Fichier: UserList.js
import { useFetch } from './useFetch';

function UserList() {
  // 1. Consommer le Hook
  const { data, loading, error } = useFetch('/api/users'

  // 2. Gérer les 3 états dans le Rendu
  if (loading) return Chargement ... ;
  if (error) return Erreur: {error.message};

  return (

      {data?.map(user => (
         key={user.id}>{user.name}
      ))}

  );
}
```

# Partie 3: Hooks Avancés (useContext, useReducer)

Gérer l'état complexe et le partage d'état.

# Le Problème: Le "Prop Drilling"

- **Définition:** C'est l'action de faire passer des props à travers de multiples niveaux de composants imbriqués qui n'en ont pas besoin, juste pour les transmettre à un enfant lointain.
- **Exemple:** L'état "utilisateur authentifié" doit être passé de App -> Layout -> Navbar -> UserAvatar.
- **Inconvénient:** Rend le code difficile à maintenir et à refactoriser.

# La Solution: `useContext`

Le Hook `useContext` permet de créer un "conduit" de données global (ou semi-global) auquel n'importe quel composant enfant peut s'abonner, sans "prop drilling".

## Les 3 Étapes:

1. **Créer** le Contexte (`createContext`).

2. **Fournir** la valeur (`Context.Provider`).

3. **Consommer** la valeur (`useContext`).

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

// 1. Créer le contexte (souvent dans un fichier séparé)
const AuthContext = createContext(null); // null = valeu

function App() {
  const currentUser = { name: 'Alice' };

  return (
    // 2. Fournir la valeur à tous les enfants
     value={currentUser}>
       />

  );
}

function UserAvatar() {
  // 3. Consommer la valeur (pas de props !)
  const user = useContext(AuthContext);

  return {user.name};
}
```

# Quand utiliser `useContext` ?

- **Idéal pour:** Des données qui changent peu fréquemment.
- **Ex:** L'état d'authentification de l'utilisateur, le thème (dark/light mode), la langue préférée.
- **Attention:** `useContext` n'est pas optimisé pour des changements très fréquents (ex: état d'un formulaire). Lorsque la valeur du Provider change, *tous* les composants qui consomment ce contexte sont re-rendus.

# Le Problème: Logique d'état complexe

- **Limites de `useState`:** `useState` est parfait pour des états simples (booléens, chaînes, nombres).

  **Le défi:** Que faire si vous avez plusieurs états qui dépendent les uns des autres ? Ou si la prochaine valeur de l'état dépend d'une logique complexe basée sur la précédente ?

- **Exemple:** Gérer un panier d'achat (ajouter, supprimer, mettre à jour la quantité, calculer le total). Utiliser 10 `useState` différents devient chaotique.

Le Hook `useReducer` est une alternative à `useState` conçue pour gérer des logiques d'état complexes de manière prévisible.

Il centralise toute la logique de modification de l'état dans une seule fonction : le **Reducer**.

## Anatomie:

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

dispatch: Une fonction pour "envoyer" une action (instruction de changement) au reducer.

```
// 1. L'état initial
const initialState = { count: 0 };

// 2. Le Reducer (la logique de mise à jour)
function reducer(state, action) {
  switch (action.type) {
    case 'INCREMENT':
      return { count: state.count + 1 };
    case 'DECREMENT':
      return { count: state.count - 1 };
    case 'SET':
      return { count: action.payload };
    default:
      throw new Error('Action inconnue');
  }
}

function Counter() {
  // 3. Utiliser le Hook
  const [state, dispatch] = useReducer(reducer, initialS

  return (
    <>
      onClick={() => dispatch({ type: 'DECREMENT' })}>-
      {state.count}
      onClick={() => dispatch({ type: 'INCREMENT' })}>+
```

# Partie 4: Hooks d'Optimisation de Performance

Empêcher les re-rendus inutiles (re-renders).

# Le Problème: Les Re-Rendus Inutiles

React est rapide, mais il peut être ralenti si des composants lourds sont re-rendus sans raison.

**Cause 1: Calculs Coûteux.** Une fonction de filtrage sur une grande liste peut être appelée à chaque rendu, même si les données n'ont pas changé.

**Cause 2: Références.** Les objets (`{}`) et les fonctions (`() => {}`) sont recréés à chaque rendu, ce qui fait croire à React qu'une prop a changé.

Le Hook useMemo **mémorise (met en cache) la valeur de retour** d'un calcul coûteux.

Il ne recalcule la valeur que si l'une des dépendances (ex: `list`) a changé.

## Anatomie:

const valeurCache = useMemo(() => calcul(a, b), [a, b]);

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

// Supposons que c'est une fonction très lente
const filtreCoûteux = (items) => {
  console.log('FILTRAGE LENT EN COURS...');
  return items.filter(item => item.visible);
};

function MaListe({ items }) {
  // État local (ex: un compteur)
  const [count, setCount] = useState(0);

  // Sans useMemo, 'filtreCoûteux' s'exécute à chaque
  // fois que 'count' change (inutile !)

  // Avec useMemo, le filtrage ne s'exécute que si 'item
  const itemsVisibles = useMemo(() => {
    return filtreCoûteux(items);
  }, [items]); // Dépendance: items

  return (

       onClick={() => setCount(c => c + 1)}>
        Compteur: {count}

      {/* ... Affiche itemsVisibles ... */}
```

Similaire à `useMemo`, mais useCallback **mémorise (met en cache) la fonction elle-même**, pas sa valeur de retour.</p

C'est crucial lorsque vous passez des fonctions comme `props` à des composants enfants qui sont optimisés (ex: avec `React.memo`).

## Anatomie:

const funcCache = useCallback(() => action(a), [a]);

```
import { useCallback, useState } from 'react';
import { BoutonOptimisé } from './BoutonOptimisé';

function Parent() {
  const [count, setCount] = useState(0);
  const [name, setName] = useState('');

  // SANS useCallback:
  // Cette fonction est RECRÉÉE à chaque rendu.
  // BoutonOptimisé se re-rendra même si 'name' change.
  // const handleCountClick = () => setCount(count + 1);

  // AVEC useCallback:
  // Cette fonction n'est recréée que si 'count' change.
  const handleCountClick = useCallback(() => {
    setCount(count + 1);
  }, [count]); // Dépendance: count

  return (

       value={name} onChange={e => setName(e.target.valu
      // BoutonOptimisé ne se re-rend que si 'handleCoun
       onClick={handleCountClick}>
        Incrémenter le compteur
```

# Hook Avancé: `useRef` (Revisité)

**Rappel (Module 3):** Nous l'avons utilisé pour accéder aux éléments du DOM (composants non contrôlés).

**Nouvel Usage:** useRef est une "boîte" qui peut contenir n'importe quelle valeur.

**La Différence Clé:** Modifier la propriété `.current` d'une Ref **ne déclenche PAS** un nouveau rendu (contrairement à `useState`).

**Cas d'usage:** Stocker un ID de timer, des valeurs d'état précédentes, ou toute variable que vous voulez conserver entre les rendus sans provoquer de mise à jour.

# Fin du Module 5

Prochaine étape: Le Module 6 (Gestion d'État Globale & Écosystème)

# Module 6: Gestion d'État Globale & Écosystème

Passer à l'échelle, les outils modernes et les meilleures pratiques.

# Partie 1: Limites du Contexte et Solutions

Quand `useContext` n'est plus suffisant.

# Rappel et Limite de `useContext`

- **Performances:** Si la valeur passée au `Provider` change, **tous** les composants consommateurs se re-rendent, même s'ils n'utilisent qu'une petite partie de la valeur.
- **Logique:** Le contexte ne sépare pas clairement la logique de l'état (le `useState` et le `Context.Provider` sont souvent ensemble).
- **Architecture:** Pour les applications de grande envergure, le Context peut devenir un `Provider` unique et monolithique, rendant le débogage difficile.

# Les Outils Externes pour la Gestion d'État

- **Redux:** Le pionnier, basé sur le pattern Reducer (que nous avons vu avec `useReducer`). Offre une source unique de vérité et une prévisibilité totale. Souvent considéré comme trop "verbeux" pour les petites applications.
- **Zustand:** Un gestionnaire d'état minimaliste et moderne, s'inspirant de `useContext` et de `useReducer` mais optimisé pour les re-rendus. Extrêmement populaire pour sa simplicité.
- **Outils de Fetch (RTK Query / TanStack Query):** Pour la gestion des données de serveur. Ces outils gèrent automatiquement le `loading`, `error`, le cache, l'invalidation, et le fetch en arrière-plan. **Le choix moderne préféré.**

# Partie 2: Gestion des Données de Serveur (Caching)

Le nouveau standard pour les appels

API.

# TanStack Query (React Query)

- **Problème résolu:** Il fait la distinction entre l'état de l'interface utilisateur (UI State) et l'état du serveur (Server State).
- **Fonctionnalités clés:** Gère la mise en cache (caching), la déduplication des requêtes, la mise à jour en arrière-plan (stale-while-revalidate) et la synchronisation automatique des données.
- **Démarche simplifiée:** Au lieu d'écrire un `useFetch` avec `useState` et `useEffect`, on utilise un seul hook : `useQuery`.

En utilisant `useQuery`, toute la logique complexe de fetch, `loading`, `error`, nettoyage et mise en cache est gérée pour vous.

Le code devient extrêmement concis et déclaratif.

```
// 1. Importer le hook
import { useQuery } from '@tanstack/react-query';

// La fonction de fetch (une simple fonction async)
const getPosts = async () => {
  const res = await fetch('/api/posts');
  if (!res.ok) throw new Error('Failed to fetch');
  return res.json();
};

function PostList() {
  // 2. Utiliser useQuery
  const {
    data: posts,
    isLoading,
    isError,
    error
  } = useQuery({
    queryKey: ['posts'],
    queryFn: getPosts
  });

  // 3. Rendu conditionnel
  if (isLoading) return Chargement en cours ... ;
  if (isError) return Erreur: {error.message};

  return (
```

# Partie 3: Le Système de Navigation (Routing)

Organiser les différentes "pages" de votre application.

# React Router Dom

- **Rôle:** Permet de créer des URL uniques pour des vues spécifiques dans une application à page unique (Single Page Application - SPA).
- **Concept clé:** La navigation ne recharge pas la page ; elle change juste les composants affichés.
- **Composants principaux:**
  - : Entoure toute l'application.
  - et : Définissent les chemins (paths) et les composants.
  - : Remplace les balises `` pour la navigation interne.

Cet exemple montre comment structurer l'application pour gérer trois pages différentes : Accueil, À Propos et Détail du Produit (avec un paramètre dynamique :id).

```
import { BrowserRouter, Routes, Route, Link, useParams }

const Home = () => Page d'Accueil;
const About = () => À Propos;
const ProductDetail = () => {
  const { id } = useParams(); // Récupère l'ID de l'URL
  return Produit #{id};
};

function App() {
  return (

      // Navigation

         to="/">Accueil |
         to="/about">À Propos


      // Définition des Routes

         path="/" element={ />} />
         path="/about" element={ />} />
         path="/product/:id" element={ />} />
```

# Partie 4: Les Tests et la Qualité du Code

Assurer la robustesse et la maintenabilité.

# La Pyramide des Tests React

- **Tests Unitaires (Unit Tests):** Testent de petites unités de code de manière isolée (ex: une fonction, un Hook personnalisé). Outils : Jest.
- **Tests d'Intégration (Integration Tests):** Testent comment plusieurs unités (composants) travaillent ensemble (ex: un formulaire complet, incluant l'état et le rendu). Outils : React Testing Library.
- **Tests de Bout en Bout (E2E Tests):** Simulent l'expérience utilisateur complète dans un vrai navigateur (ex: "L'utilisateur se connecte, ajoute un article au panier, et finalise la commande"). Outils : Cypress, Playwright.

# React Testing Library (RTL)

- **Philosophie:** "Testez l'application comme un utilisateur la testerait."
- **Principe:** Au lieu de tester les détails d'implémentation (l'état interne), vous interagissez avec l'interface utilisateur et vérifiez le résultat visible.
- **Requêtes:** Utilise des méthodes basées sur l'accessibilité pour trouver les éléments, par exemple : getByRole('button', { name: /Ajouter/i }).

# Les Autres Outils de l'Écosystème

- **TypeScript:** Un sur-ensemble de JavaScript qui ajoute le typage statique. **Fortement recommandé** pour les projets de grande taille afin de prévenir les erreurs de type et améliorer l'autocomplétion.
- **Storybook:** Un outil pour développer, documenter et tester les composants d'interface utilisateur de manière isolée. Idéal pour construire un système de design réutilisable.
- **ESLint & Prettier:** Outils essentiels pour l'hygiène du code. ESLint détecte les problèmes de qualité et de style ; Prettier formate le code automatiquement.

# Félicitations !

Vous avez terminé le Cours Intensif ReactJS.

**Le voyage ne fait que commencer.**