



# React: Développement d'applications Web

Thaïs Labouré  
[thais.laboure@outlook.com](mailto:thais.laboure@outlook.com)  
Mis à jour le 16/09/2025



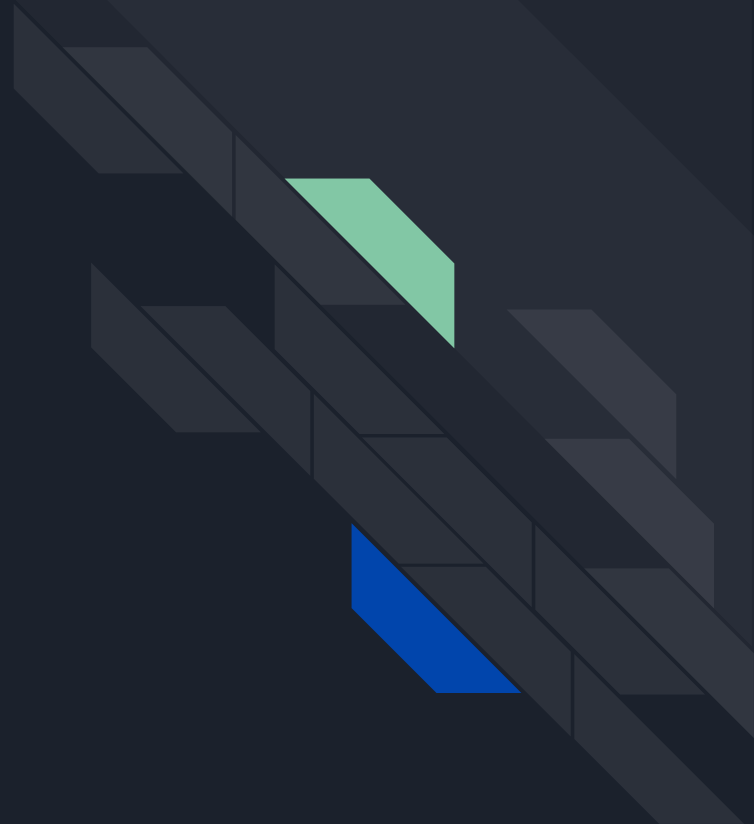
ABC



# Objectifs de la formation

- Comprendre les bases de React
- Explorer les concepts fondamentaux d'ES6 et TypeScript.
- Apprendre à développer des applications web modernes.
- Mettre en pratique les compétences acquises à travers un projet final.

# Rappels EcmaScript et Typescript





# Qu'est-ce qu'EcmaScript ?

Qu'est-ce qu'EcmaScript ?

- Spécification standard de JavaScript.
- ES6 (2015) a introduit des évolutions majeures pour simplifier et structurer le code.



# Nouveautés importantes d'EcmaScript

## Variables et constantes

`let` et `const` pour une meilleure portée des variables.



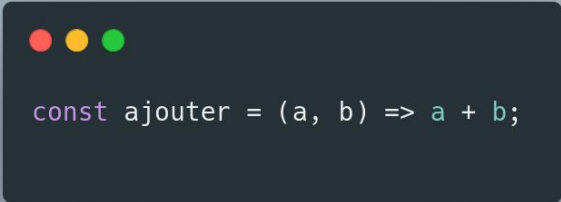
```
const PI = 3.14;  
let compteur = 0;
```



# Nouveautés importantes d'EcmaScript

## Fonctions fléchées

- Syntaxe concise pour les fonctions.
- Retour implicite



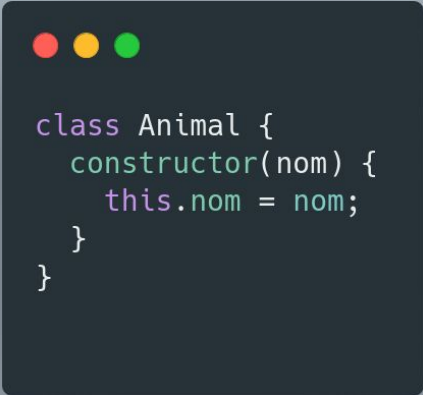
```
const ajouter = (a, b) => a + b;
```



# Nouveautés importantes d'EcmaScript

## Classes et héritage

- Structure pour les objets orientés classe.



```
class Animal {  
  constructor(nom) {  
    this.nom = nom;  
  }  
}
```






# Nouveautés importantes d'EcmaScript

## Template literals

- Création de chaînes de caractères multi-lignes et dynamiques.
- ( alt-gr + 7 )




```
const nom = 'Toto';  
const message = `Bonjour, ${nom}!`;
```



# Nouveautés importantes d'EcmaScript

## Modules

- Import/export de fonctionnalités.




```
import { addition } from './math.js';  
import Animal from './animal.js';
```



# Nouveautés importantes d'EcmaScript

## Destructuration

- Extraction de valeurs depuis des objets ou des tableaux.



```
const personne = {nom: 'Jean Michel', age: 452}  
const { nom, age } = personne;  
const fruits = ['Pêche', 'Pomme', 'Poire'];  
const [fruit1, fruit2] = fruits;
```



# Introduction à Typescript

Qu'est-ce que TypeScript ?

- Surensemble de JavaScript développé par Microsoft.
- Ajoute un typage statique et des fonctionnalités avancées.
- Transpilé en JavaScript standard.



TS



# Avantages de TypeScript

- Typage statique
  - Réduit les erreurs au moment de la compilation.
- Classes et Interfaces avancées
  - Structure claire et réutilisable pour les objets.
- Support des ES6+
  - Utilisation des dernières fonctionnalités tout en restant compatible avec les navigateurs.
- Outils de développement
  - Auto-complétion et débogage améliorés.

# Comparaison Typescript / Javascript



```
const foo = 'Hello World';  
function ajouter(a, b) {  
  return a + b;  
}
```



```
const foo: string = 'Hello World';  
function ajouter(a: number, b: number): number {  
  return a + b;  
}
```



# Présentation de Flow

Citère	Flow	Typescript
Créateur	Facebook	Microsoft
Typage	Typage optionnel et flexible	Typage strict
Syntaxe	Proche du JS standard	Superset de JS
Ecosystème	Moins adopté que TS	Populaire et communautaire
Outils associés	Se concentre sur le typage	Inclut un compilateur



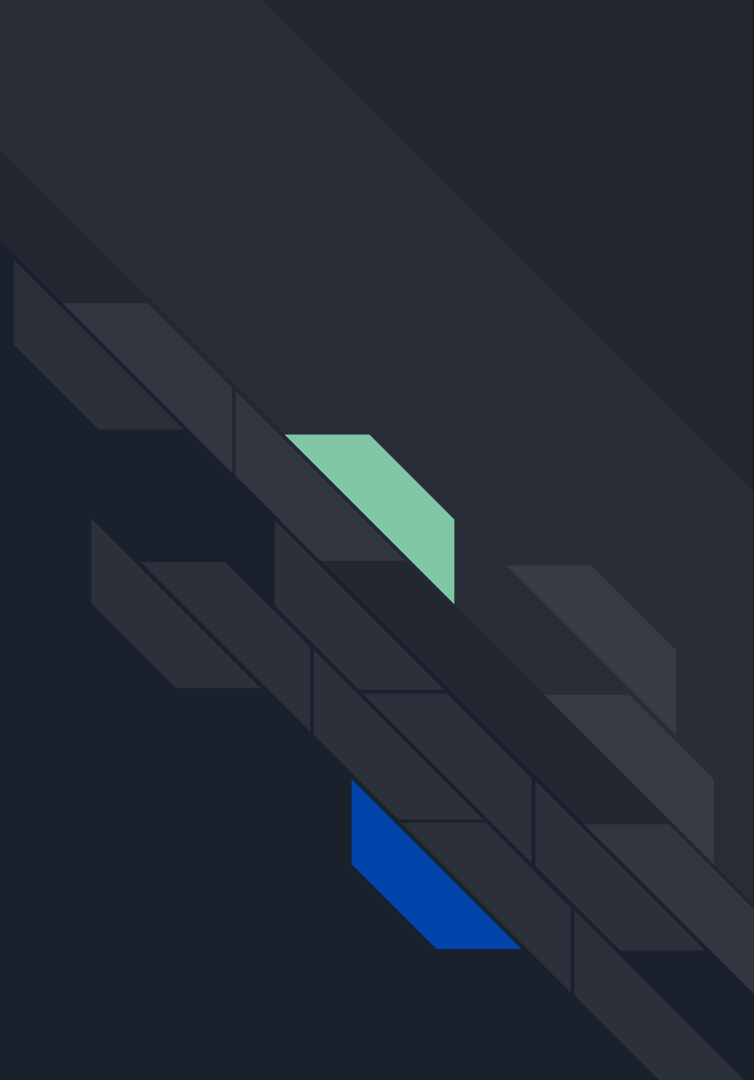
# Présentation de Flow



```
// @flow
function square(n: number): number {
  return n * n;
}
square("hello"); // Erreur détectée par Flow
```



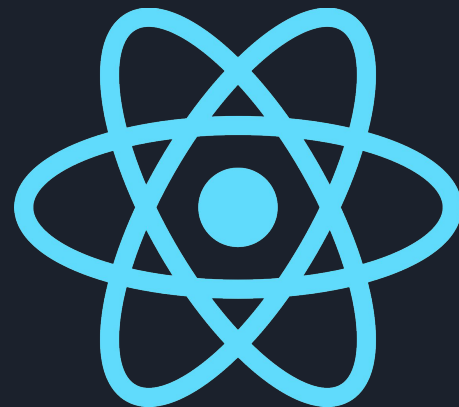
# Présentation de React





# React

- Créé par Facebook en 2013
- Librairie légère permettant de créer des Single Page Application ( SPA )
- Écosystème varié : React Router, Redux, React Query, ect...
- Large communauté
- Framework Next.JS pour créer des applications SSR





# Alternatives

## Angular

- Créé par Google en 2016
- Framework complet
- Basé sur Typescript
- Inclut de nombreuses librairies nativement : router, RXJS, HTTP Client,...

## Vue.JS

- Créé par Evan You en 2014
- Permet de créer des SPA
- Flexible, peut être intégré dans de nombreux environnements ( Symfony, Laravel, projet HTML / CSS natif... )
- Framework Nuxt permettant de créer des applications SSR



# Autres architectures courantes

Pour le développement web, on retrouve d'autres architectures courantes :

Le **MVC** ( *Model, Vue, Controller* ) : Symfony, Spring MVC, Node...

Le **SSR** ( *Server Side Rendering* ) : dérivé des SPA, les scripts sont exécutés côté serveur, pour n'envoyer "que" du HTML au navigateur : Next.js ( Framework pour React ), Nuxt ( Vue ), Angular avec ng-universal, Svelte Kit.



# L'écosystème React

JavaScript moderne utilise des outils pour :

- Transpiler le code pour la compatibilité avec les anciens navigateurs (Babel).
- Bundler les fichiers pour une meilleure performance (Webpack).
- Simplifier le démarrage avec des outils comme CRA ou Vite.

**Graphique** : Schéma du flux de développement (code source → transpilation → bundle → production).



# Babel

**Définition :** Babel est un **transpileur JavaScript** qui permet de convertir du code ES6+ (ou JSX) en JavaScript compatible avec les navigateurs plus anciens.

## Pourquoi l'utiliser avec React ?

- Convertir JSX en JavaScript.
- Support des nouvelles fonctionnalités JS.



# Webpack

**Définition :** Webpack est un **bundler**. Il regroupe tous les fichiers de votre projet (JS, CSS, images, etc.) en un seul fichier optimisé pour le navigateur.

## Pourquoi l'utiliser ?

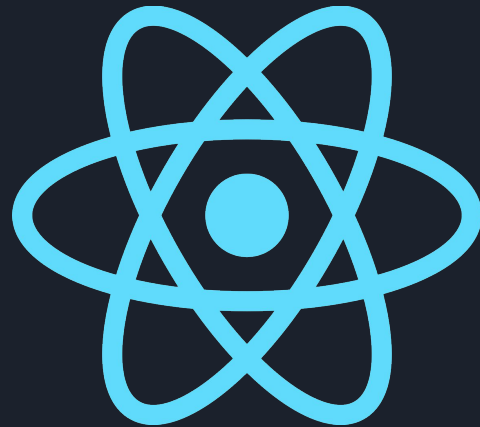
- Gestion des dépendances.
- Optimisation pour la production.

## Fonctionnalités clés :

- Loaders : Convertir des fichiers non-JS (ex. : CSS, images).
- Plugins : Ajouter des fonctionnalités (ex. : minification, hot reload).



# CRA : Create React App



## Description :

- Un outil CLI pour démarrer un projet React avec zéro configuration.
- Basé sur Webpack et Babel.

## Avantages :

- Simple pour les débutants.
- Configuration prête à l'emploi (hot reload, JSX, CSS).

## Inconvénients :

- Poids élevé en développement.
- Peu flexible sans éjection.





# Vite



## Description :

- Un outil moderne basé sur ESBuild pour le développement rapide.
- Conçu pour des performances élevées.

## Avantages :

- Temps de démarrage ultra-rapide.
- Support natif des modules ES (pas de bundle initial en dev).

## Exemple :

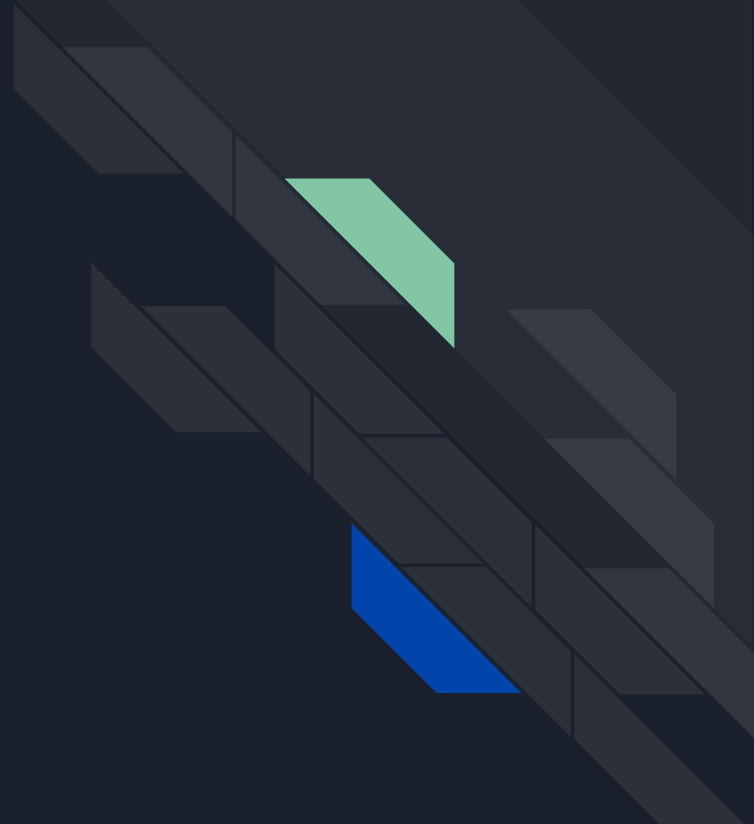
- Installation : `npm create vite@latest my-app --template react`
- Démarrage : `npm run dev`



# En résumé

- React est une librairie permettant de créer des Single Page Applications
- Pour créer une application React, deux solutions :
  - CRA: rapide et facile, parfait pour apprendre
  - Vite: la solution moderne

# Les Composants





# Qu'est-ce qu'un composant

Le principe de React est de créer des composants.

Un composant :

- Est un ensemble de code HTML / CSS / JS
- Permet de créer des parties d'interfaces, des pages
- Inclut le contenu, le style, et les réactions d'un élément
- Est réutilisable



# Le JSX

Les composants React utilisent majoritairement la syntaxe JSX :

JSX :

- Javascript XML.
- ressemble à du HTML, en respectant les normes XML : exemple les balises sans contenu doivent être orpheline.

Les noms de balises HTML doivent être en minuscule, les noms de composants se mettent en PascalCase.

Le JSX est plus proche du JS que du HTML. Les mots réservés en JS ne peuvent pas être utilisés en JSX.

- class : className
- for : htmlFor

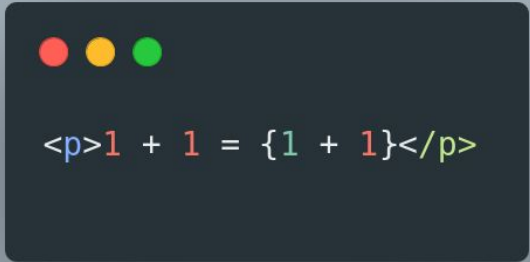
Les attributs html en 2 mots ou + doivent être écrit en camelCase ( et pas en kebab-case, ni tout en minuscule ) : tabIndex : tabIndex.



# L'interpolation

L'interpolation, c'est le fait d'interpréter du Javascript dans le template.

En React, on utilise les accolades pour interpréter du JS :



```
<p>1 + 1 = {1 + 1}</p>
```



# Le Data Binding

L'interprétation d'expressions HTML dans le template, et plus précisément, dans les attributs HTML. L'interpolation est un type de Data Binding. Pour les attributs, on parle aussi d'Attribute Binding.



```
<script>
  let classe = 'red';
</script>

<!-- reste du composant -->
<p className={classe}>...</p>
```



# Rendu Conditionnel

Le JSX interprétant du JS, on peut rendre du contenu en fonction d'une condition.

On peut utiliser le ternaire, des fonctions, ou des expressions conditionnelles.



```
<script>
  let bool = true;
</script>

<!-- reste du composant -->
<p>{bool ? `C'est oui !` : `C'est non !`}
```





# Le rendu de listes

React peut également rendre des listes.

Il va recréer un élément pour chaque élément de la liste.

Attention à bien assigner une *key* unique à chaque élément !

```
<script>
  let fruits = ['Pomme', 'Poire', 'Kaki']
</script>

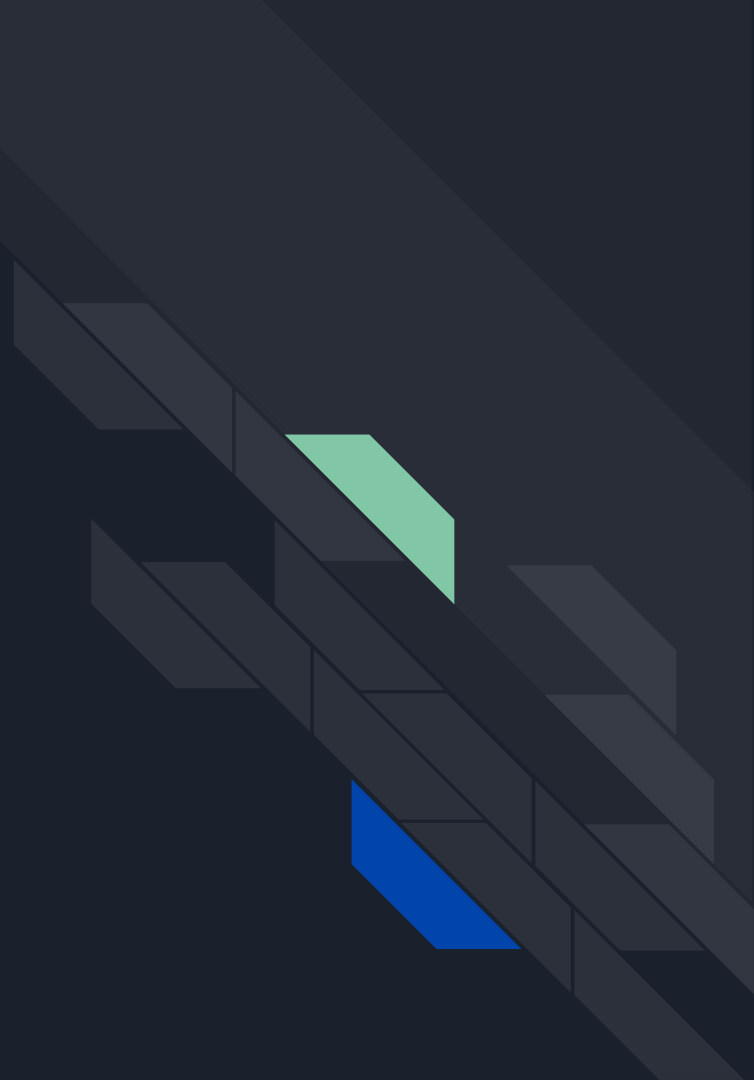
<!-- reste du composant -->
<ul>
  {fruits.map((f) => <li key={f}>{f}</li>)}
</ul>
```



## En résumé

- React permet de créer des composants en utilisant les JSX.
- Les JSX permettent d'interpréter du Javascript entre accolades
- Cela permet de faire de l'interpolation, du data binding, du rendu conditionnel, et du rendu de listes
- Le JS interprété dans le JSX doit retourner du contenu ( texte, JSX )

# React Router





# Présentation

React Router est une bibliothèque utilisée pour gérer la navigation et le routage dans une application React. Elle permet de créer des **applications monopages (SPA)** avec une navigation fluide.

## Pourquoi l'utiliser ?

- Gérer les URL pour afficher différentes vues.
- Mettre en œuvre des routes dynamiques.
- Navigation sans rechargement de page (client-side routing).

## Fonctionnalités clés :

- Définir des routes (via `<Route>`).
- Naviguer entre les pages (via `<Link>` ou `<NavLink>`).
- Gestion des paramètres d'URL.

S'installe avec `npm install react-router-dom`



# Configurer les routes

On configure les routes du router dans le composant App.jsx.

- Englober l'application dans le composant `<BrowserRouter>`
- Définir les `<Route>` dans `<Routes>`
- Une route peut contenir des sous-routes.

```
<BrowserRouter>
  <Navigation />
  <Routes>
    <Route path="/" element={<Home/>} />
    <Route path="/contact" element={<Contact/>} />
  </Routes>
</BrowserRouter>
```

# Naviguer entre les routes

- On navigue entre les routes avec les `<Link>` ou `<NavLink>`
- On indique la route à suivre avec l'attribut `to`
- `NavLink` ajoute une classe additionnelle aux liens actifs, qui permet de les styliser

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

function Navbar() {
  return (
    <nav>
      <Link to="/">Home</Link>
      <Link to="/about">About</Link>
      <Link to="/products">Products</Link>
    </nav>
  );
}
```



# Les paramètres

- On peut ajouter des paramètres à une route avec le data binding :  
`<Route path="/product/:id" element={<Product />} />`
- On peut ensuite récupérer ce paramètre dans la page avec `useParams()`



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

function Product() {
  const { id } = useParams();
  return <h1>Produit ID : {id}</h1>;
}
```



# Redirection programmatique

On peut rediriger  
programmatically ( dans le  
Javascript ) avec `useNavigate()`



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

function RedirectButton() {
  const navigate = useNavigate();
  return <button onClick={() => navigate('/about')}>Go to About</button>;
}
```





# Navigation Conditionnelle

On peut utiliser la navigation conditionnelle pour rediriger un•e utilisateur•trice vers une page ou une autre en fonction d'une condition.

Par exemple, pour protéger une route.

```
<Route
  path="/dashboard"
  element={isLoggedIn ? <Dashboard /> : <Navigate to="/login" />}
/>
```




# Le Lazy Loading

Par défaut, tous les composants importés et utilisés dans le routeur se chargent automatiquement au démarrage de l'application.

Avantage : le reste de la navigation est fluide.

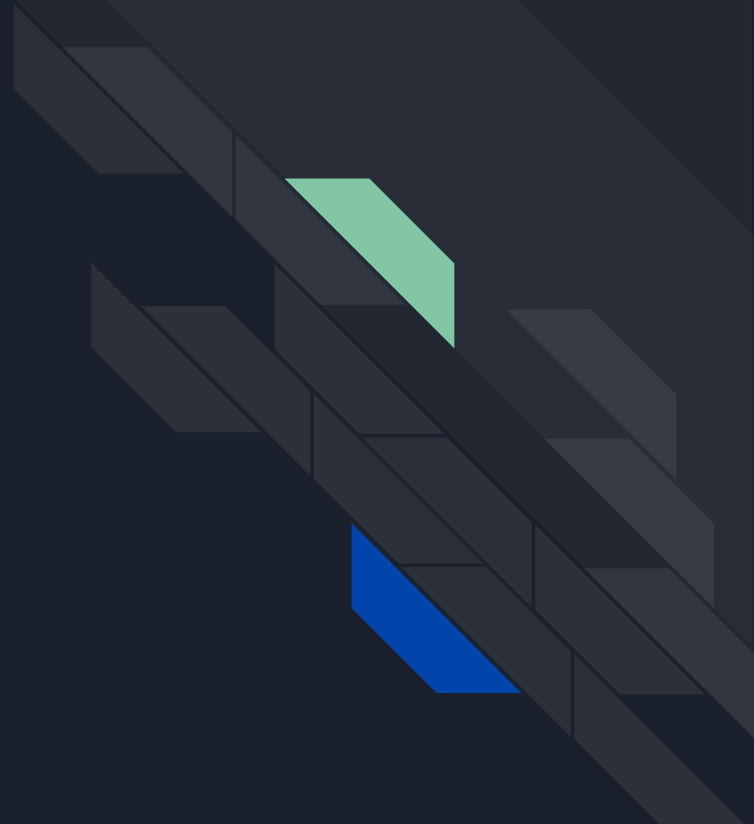
Inconvénient : le premier affichage peut-être très long, et on charge des composants dont on n'a pas forcément besoin.

Solution : le lazy-loading



```
const Admin = lazy(() => import('./pages/Admin'));
```

# La Réactivité





# Le Hook d'état

## Définition :

useState est un **hook** React qui permet de gérer l'état local d'un composant fonctionnel.

## Pourquoi utiliser useState ?

- Permet de rendre un composant réactif.
- Gère les données qui changent dynamiquement (ex. : texte saisi, compteur).

## Utilisation :

- state : l'état de la donnée à un moment T
- setState : une fonction pour modifier la donnée ( modifie et met à jour / re-render le composant )
- initialValue : la donnée initiale



```
const [state, setState] = useState(initialValue);
```



# Mettre à jour la donnée

La fonction retournée par `useState` permet de mettre à jour la valeur et de re-render le composant, pour mettre à jour l'affichage et les éléments utilisant cette donnée.

On ne modifie jamais directement une donnée réactive !



```
const [state, setState] = useState(initialValue);  
setState('Nouvelle valeur');  
setState(previous => 'Nouvelle valeur')
```

# Les événements





# Présentation

## Définition :

Les événements en React sont similaires à ceux du DOM, mais sont encapsulés dans le système de gestion des événements de React.

## Différences avec les événements DOM :

- React utilise une **syntaxe camelCase** (ex. : `onClick`, `onChange`).
- Les gestionnaires d'événements sont des **fonctions JavaScript** au lieu de chaînes de caractères.
- Les événements de React sont des objets `SyntheticEvent`, une couche d'abstraction compatible avec tous les navigateurs.



# Fonctionnement



```
function Button() {  
  const handleClick = () => alert('Button clicked!');  
  
  return <button onClick={handleClick}>Click me</button>;  
}
```



# Passer des paramètres

- React passe un objet *Event* en paramètre à la fonction appelée.
- Cela permet de gérer l'événement en fonction des besoins (*stopPropagation()*, *preventDefault()*)
- Pour passer des paramètres, il faut passer une fonction qui appelle la fonction



```
function Button() {  
  const handleClick = (event) => console.log(event);  
  
  return <button onClick={handleClick}>Click me</button>;  
}
```



```
function greet(name) {  
  alert(`Hello, ${name}!`);  
}  
  
<button onClick={() => greet('Alice')}>Say Hi</button>
```



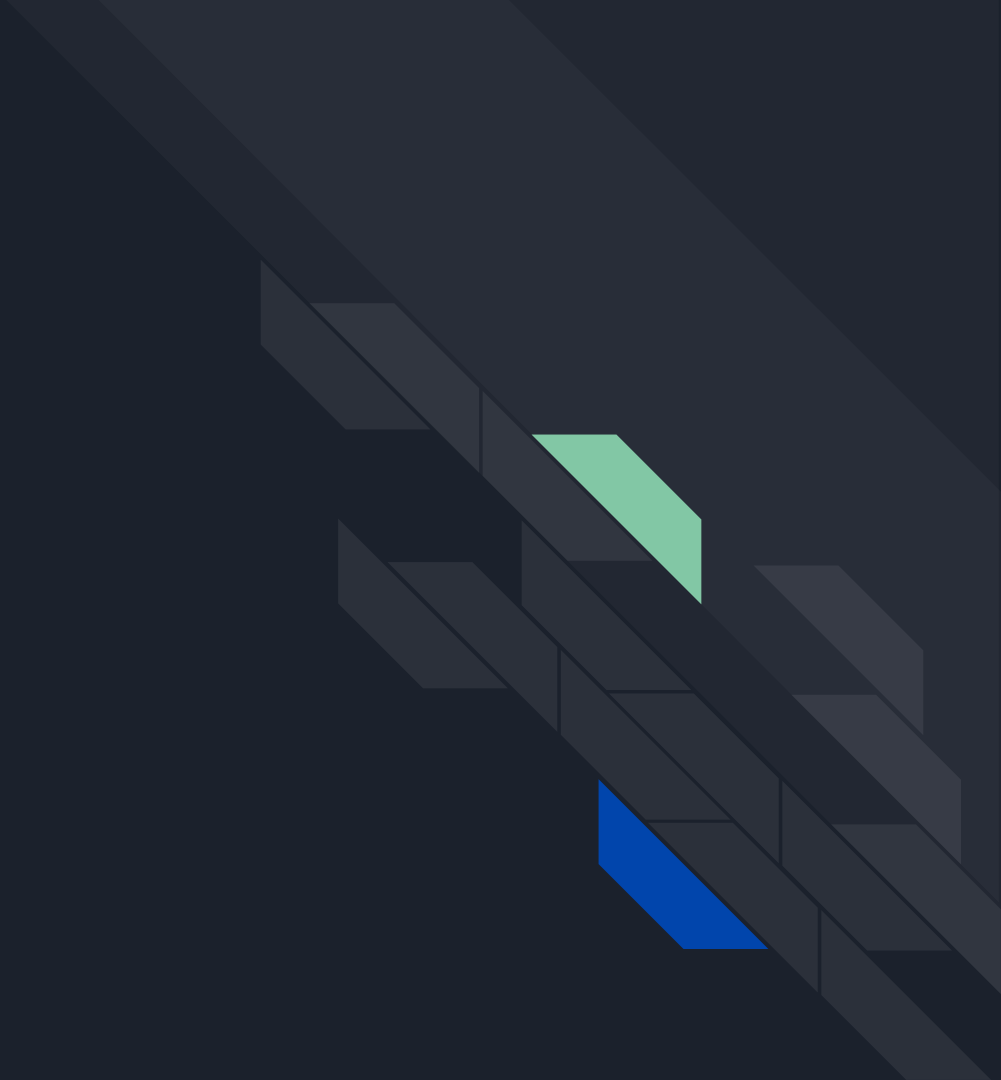
# Gérer plusieurs événements

Il est possible de gérer plusieurs événements.

Tous les événements natifs du DOM sont accessibles.

```
function InputField() {  
  const handleFocus = () => console.log('Focused!');  
  const handleBlur = () => console.log('Blurred!');  
  
  return <input onFocus={handleFocus} onBlur={handleBlur} />;  
}
```

# Les Props





# Présentation

Les props sont un moyen de créer des composants réutilisables qui s'adaptent aux besoins.

Elles permettent de gérer les données passer d'un composant parent vers un composant enfant ( un composant appelé par le parent ).

Un composant peut attendre aucune, une, ou plusieurs props de types différents.

## Caractéristiques principales :

- Les props sont **en lecture seule** (immutables).
- Utilisées pour **transférer des données** et **configurer des composants**.

## Pourquoi les utiliser ?

- Rendre les composants réutilisables et dynamiques.
- Faciliter la communication unidirectionnelle (de parent à enfant).



# Exemple simple

```
function Greeting(props) {  
  return <h1>Hello, {props.name}!</h1>;  
}  
  
function App() {  
  return <Greeting name="Alice" />;  
}
```



# La props Children

Une prop spéciale pour passer du contenu (HTML, JSX, ou autres composants) au composant enfant.

Permet de passer du texte, du contenu dynamiquement comme dans une “vrai” balise HTML.

```
function Card({ children }) {  
  return <div className="card">{children}</div>;  
}  
  
function App() {  
  return (  
    <Card>  
      <h1>Title</h1>  
      <p>Content inside the card</p>  
    </Card>  
  );  
}
```



# Validation avec PropTypes

npm i prop-types

## Pourquoi valider les props ?

- Garantir que les composants reçoivent les bonnes données.
- Faciliter le débogage.

## Types disponibles :

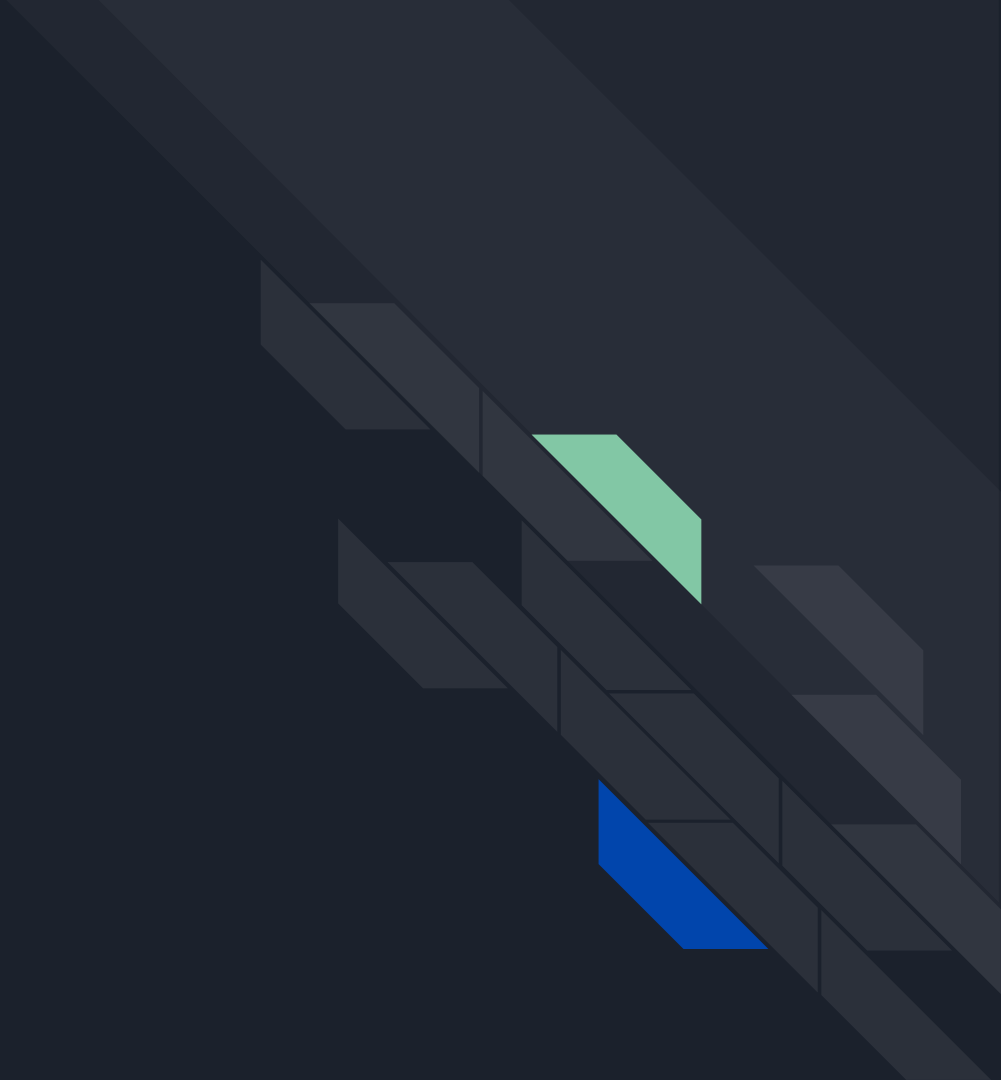
- string, number, array, object, bool, func, etc.
- `isRequired` pour une prop obligatoire.

```
import PropTypes from 'prop-types';

function Greeting({ name, age }) {
  return <p>{name} is {age} years old.</p>;
}

Greeting.propTypes = {
  name: PropTypes.string.isRequired,
  age: PropTypes.number.isRequired,
};
```

# Les Hooks







# Les Class Components

Avant les fonctions, React utilisait les classes pour créer des composants.

Ces classes exposent des méthodes de cycle de vie permettant de réagir à différentes étapes du projet : `componentDidMount()`, `componentDidUpdate()`, `componentWillUnmount()`, ect...



# UseEffect

A chaque modification du state ou de props, React réexécute la fonction du composant.

De ce fait, React recrée les variables déclarées dans la fonction.

UseEffect est un Hook qui permet de réagir lors d'effet de bord / du cycle de vie du composant.

Il remplace les méthodes *componentDidMount()*, *componentDidUpdate()*, et *componentWillUnmount()*.

Il prend en paramètre une callback ( l'action à effectuer ), et un tableau de dépendances ( les valeurs à surveiller ).



```
useEffect(() => {  
    console.log(`User updated`)  
}, [user])
```



# Memoïsation

La Memoization permet à React de garder en mémoire des calculs / fonctions, et de ne les recréer que lorsque que les valeurs utilisées par la fonction, et surveillée par la fonction sont modifiées

Il existe un Hook similaire, *useCallback*

UseCallback fonctionne de la même façon. La différence, c'est qu'on utilise useMemo comme une valeur, et useCallback, comme une fonction.

```
const [name, setName] = useState('La Tourte');
const [firstname, setFirstname] = useState('Jean Michel');
const fullnamme = useMemo(() => `${firstname} ${same}`, [name, firstname]);
```



# Reducer

Un hook React pour gérer des **états complexes** ou interdépendants.

- Alternative à useState.
- Basé sur le pattern Redux-like avec des actions et un reducer.

## Quand l'utiliser ?

- Lorsque l'état est complexe ou comporte plusieurs sous-états.
- Pour des logiques de mise à jour impliquant plusieurs étapes ou conditions.

Un reducer est une fonction qui détermine comment l'état doit changer en fonction d'une action.



# Le Contexte

Le **Context API** permet de partager des données (comme des thèmes, des utilisateurs authentifiés, ou des paramètres globaux) à travers une hiérarchie de composants, **sans avoir à passer les props manuellement** à chaque niveau.

## Pourquoi utiliser le Contexte ?

- Évite le **prop drilling** :
  - Le prop drilling consiste à passer des données à des composants enfants via plusieurs niveaux inutiles.
- Simplifie le partage de données entre composants qui ne sont pas directement liés.
- Idéal pour des données globales comme :
  - **Thème** (clair/sombre).
  - **Utilisateur authentifié**.
  - **Langue de l'application**.



# Le Contexte

## Avantages du Contexte

- Réduction de la complexité dans le partage de données.
- Facilite la gestion d'états globaux dans des applications complexes.
- Fonctionne avec n'importe quel type de données.

## Limites du Contexte

- Les **rendus inutiles** : tous les composants consommateurs se re-rendent si la valeur du contexte change.
- Pas destiné à remplacer un état global comme Redux dans des applications complexes.

## Points clés à retenir

- Le Contexte est idéal pour **les données globales** peu complexes.
- Combinez-le avec useReducer ou une solution comme **Redux** pour des logiques complexes ou un état global plus structuré.



# Les Hooks personnalisés

Un **hook personnalisé** (custom hook) est une fonction JavaScript qui utilise un ou plusieurs hooks React (useState, useEffect, etc.) pour encapsuler une logique réutilisable.

## Pourquoi les utiliser ?

- Réduction de la duplication de code.
- Isolation de logiques complexes (exemple : récupération de données, gestion d'abonnements).
- Amélioration de la lisibilité et de la maintenabilité des composants.

## Convention de nommage :

Un hook personnalisé commence toujours par use (exemple : useFetch, useAuth).

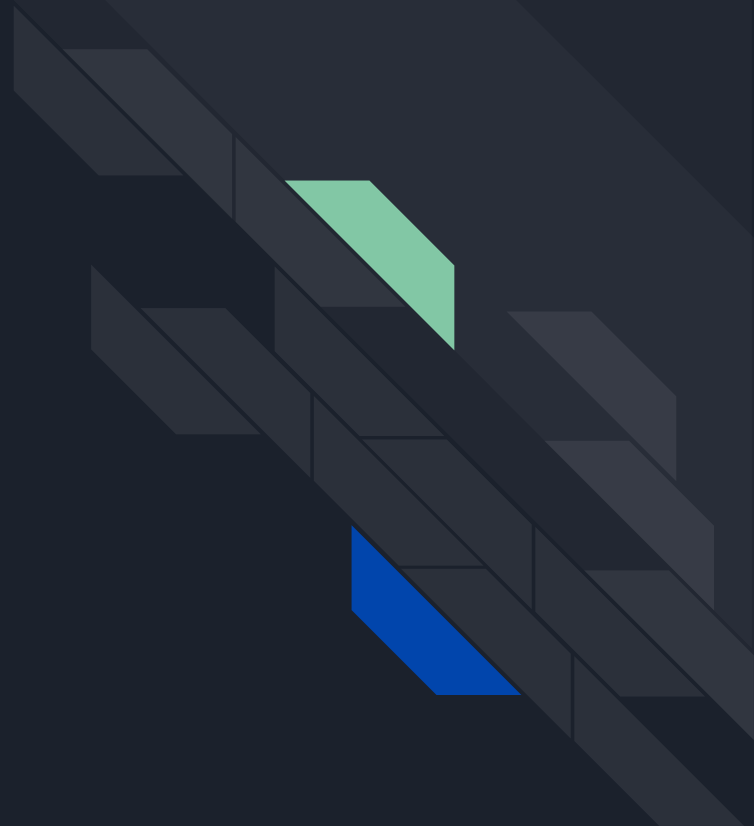


# Custom Hooks

```
function useMonHookPersonnalise() {  
  // Utilisation de hooks React ici  
  const [state, setState] = useState(initialValue);  
  
  useEffect(() => {  
    // Logique de gestion  
  }, []);  
  
  return state; // Ou un objet si plusieurs valeurs  
}
```



Librairies





# React Hook Form

```
npm install react-hook-form
```

Une bibliothèque légère et performante pour la gestion des **formulaires** dans React.

## Avantages :

- Minimise les **re-rendus** des composants.
- Simplifie la gestion des erreurs de validation.
- Compatible avec des bibliothèques tierces comme **Yup** pour des validations complexes.



# Redux

- Redux est une bibliothèque de gestion d'état prédictible pour les applications JavaScript. Il permet de gérer de manière centralisée l'état de l'application, rendant les flux de données plus clairs et maintenables.
- **Principes clés de Redux :**
  - **Store unique** : Un seul objet qui contient toute l'application.
  - **Actions** : Ce sont des objets qui décrivent un événement ou une intention de modification dans l'état.
  - **Reducers** : Fonction pure qui spécifie comment l'état change en réponse à une action.
  - **Dispatch** : Fonction qui permet d'envoyer des actions pour déclencher des changements dans l'état.
  - **State immuable** : L'état ne doit jamais être modifié directement. On crée une nouvelle copie de l'état à chaque modification.



# Fonctionnement de Redux

**Action** : Un événement ou une intention (exemple : utilisateur soumet un formulaire).

**Dispatch** : Envoie l'action au store.

**Reducer** : Le reducer prend l'action et l'état actuel pour produire un nouvel état.

**Store** : Le store contient l'état de l'application et met à jour les composants réactifs lorsque l'état change.

**View (UI)** : Les composants React se mettent à jour en fonction de l'état actuel du store.



# React Query

```
npm install @tanstack/react-query
```

React Query est une bibliothèque qui simplifie la **récupération**, le **cache**, la **synchronisation** et la **mise à jour des données** dans les applications React.

## Pourquoi l'utiliser ?

- Gestion simplifiée des états liés aux données distantes (chargement, erreurs, données).
- Mise en cache automatique des requêtes.
- Synchronisation en arrière-plan.
- Prise en charge native des fonctionnalités comme **pagination** ou **invalidation du cache**.



# React Developer Tools

Une extension de navigateur (disponible pour Chrome et Firefox) qui permet d'inspecter la structure des composants React, de visualiser leur état et leurs props en temps réel.

- **Fonctionnalités principales :**
  - **Inspecter les composants :** Voir la hiérarchie des composants et leurs props.
  - **État local et contextuel :** Accéder aux états des composants via `useState` et `useReducer`.
  - **Profiler :** Analyser les performances des composants, détecter les rendus inutiles.
  - **Modification en direct :** Modifier les props ou l'état des composants à la volée pour tester des scénarios.
- **Installation :**
  - Téléchargez l'extension **React Developer Tools** depuis le **Chrome Web Store** ou **Firefox Add-ons**.

Conclusion





# Une bibliothèque puissante et flexible

## **Composants réutilisables :**

React permet de créer des interfaces modulaires et réutilisables grâce à ses composants, facilitant ainsi le développement d'applications évolutives.

## **Gestion de l'état optimisée :**

Grâce à des hooks comme `useState`, `useReducer` et des outils comme `Redux`, React facilite la gestion de l'état d'une application de manière prévisible et performante.

## **Écosystème riche :**

React s'intègre avec un large éventail d'outils et de bibliothèques comme **React Router**, **React Query**, **React Native** et plus encore, permettant de développer des applications web et mobiles robustes.





# A Retenir

## **Développement unidirectionnel :**

Le flux de données dans React est unidirectionnel, ce qui simplifie la gestion de l'état et la détection des erreurs.

## **Performances optimisées :**

Des outils comme `React.memo`, `useMemo` et `useCallback` aident à éviter des rendus inutiles et à améliorer les performances des applications React.

## **Évolution continue :**

React évolue constamment avec des ajouts réguliers de fonctionnalités (comme les Hooks) et des améliorations de la performance, garantissant que l'écosystème reste moderne et performant.

Questions ?

