

# **React JS -** **Développement** **d'applications Web**

# More about me

Gaël Baldous

Lead DevSecOps at  Semifir

## Mes missions :

- Sécurité ( Applicative )
- Développement
- Audit / Pilotage SMSI



<https://fr.linkedin.com/in/gaël-baldous>



# Programme :

## 1 – Introduction et rappels ES6

- Outils et IDE
- L'extension du navigateur React developer tools
- Packaging, npm
- Transpiler EcmaScript
- Let, variables locales et constantes
- Typage et types natifs
- Paramètres optionnels, valeurs par défaut
- Classes et interfaces
- Gestion des modules
- Arrow functions

## 2 – Le framework React JS

- Principes de base : comprendre l'intérêt de react par rapport à ses concurrents et la façon dont il a été pensé
- Philosophie « composant »
- Les workflows de développement : from scratch (customisé), intégration à une application web existante, utilisation d'un outil de création d'une application React (create-react-app)
- Le DOM Virtuel et la réconciliation



# Programme :

## 3 - Le JSX et les composants

- Définition d'un élément React (types, attributs, enfants)
- Liaison avec le DOM (ReactDOM.render())
- Une nouvelle syntaxe : Le JSX
- Le plugin de Babel pour le JSX
- Les règles du JSX (injection d'expression, protection XSS, balise parente)
- Les attributs JSX
- Les composants : définition et intérêt (réutilisabilité)
- Les composants en mode classe
- Les composants fonctionnels (nouvelle solution)
- Imbrication de composants (les balises de composant)

## 4 - Les props

- Définition (transmission de données, readonly)
- Envoyer des props
- Accéder au props (composants fonctionnels / classe)
- La props children



# Programme :

## 5 - Le State et les lifecycles

- Définition (persistance de données, singularisation du composant)
- Initialiser le state
- La méthode setState et ses 2 formes (synchrone/asynchrone)
- Le cycle de vie du composant
- Montage du composant (componentDidMount)
- Mise à jour du composant (componentDidUpdate)
- Démontage du composant (componentWillUnmount)
- Best pratiques (setState asynchrone, ne pas utiliser setState dans le constructeur)

## 6 - Les Hooks

- Définition
- Hooks vs composants en mode classe
- Le hook d'état
- Le hook d'effet et la liste de dépendance
- Les modes du hook d'effet : initialisation, mise à jour, nettoyage
- Les règles des hooks
- Les custom hooks



# Programme :

## 7 - Les événements

- Syntaxe des événements dans le JSX
- Méthodes de gestion d'événement (handler)
- Techniques de liaison du contexte d'exécution au handler (bind(), fonctions fléchées, ...)
- Objet d'événement
- Passage de paramètres supplémentaires au handler
- Envoyer un handler en props

## 8 - Rendu conditionnel et liste

- Contenu conditionnel et raccourcis (etet, ternaires)
- Listes et raccourcis (higher order functions : map, filter, ...)
- Les clés (key) et le DOM Virtuel
- Les fragments



# Programme :

## 9 - Les formulaires

- État du composant = source de vérité
- Composant contrôlé
- L'attribut de valeur universel des champs : value
- Soumission du formulaire
- Composants non contrôlés (input de type file)
- Les refs

## 10 - Le routing et la navigation

- Construire une SPA dont les urls sont bookmarkables
- La librairie react-router-dom (version 5)
- Le router
- Les liens
- Les routes
- Le switch
- Les paramètres d'url
- Les navigations imbriquées



# Programme :

## 11 - Introduction à Redux et architecture flux

- Immutabilité des variables partagées
- Les composants d'ordre supérieur
- Problème de la gestion d'état
- Les Systèmes de gestion d'état
- L'architecture flux (actions, dispatcher, store, ...)
- Redux : définition et installation
- Les actions
- Les reducers
- Le store
- Utilisation avec React (react-redux)
- Le composant Provider
- Les containers
- Le HOC connect
- La méthode mapStateToProps
- La méthode mapDispatchToProps

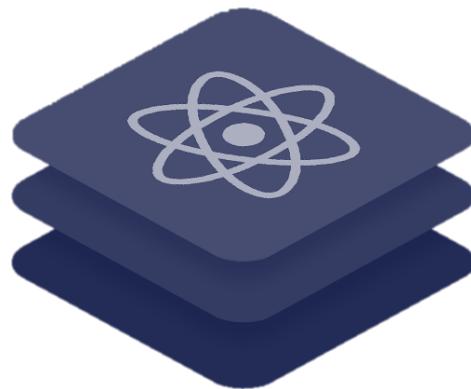
## 12 - Les tests

- Introduction au framework Jest (setup, teardown, describe, it)
- La React testing library (cleanup, render, fireEvent)



# Partie 1 :

## Introduction et rappels ES6



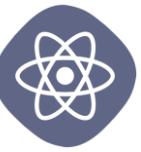


# Historique

## Informations générales

- Bibliothèque **JavaScript** libre développée par Facebook depuis 2013
- Son but : faciliter la création d'application web monopage (**SPA**)
- Fournie des composants déclaratifs permettant de créer des interfaces utilisateur de manière simple

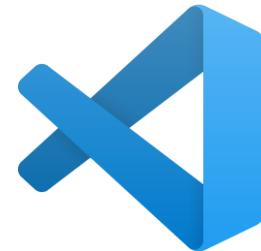




# Outils IDE

Les différents outils de développement

- Visual Studio Code
- Web Storm
- Reactide
- Codux

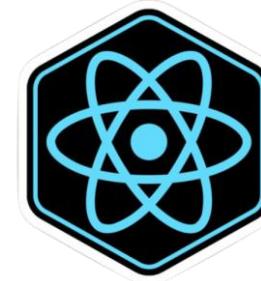
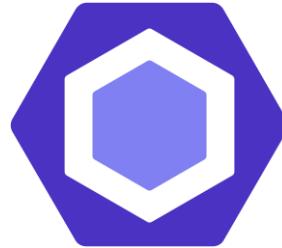




# Extensions

Quelques exemples d'extensions utiles

- **Navigateur :**
  - React Developer Tool
- **Visual Studio :**
  - React Snippet
  - Eslint
  - Prettier



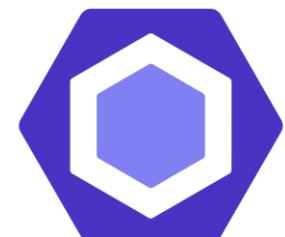


# Eslint

## Informations générales

**ESLINT** un outil de linting pour JavaScript. Il est conçu pour aider les développeurs à détecter et à corriger les erreurs de code, ainsi qu'à améliorer la qualité du code en imposant des normes de style de code cohérentes et en signalant les pratiques dangereuses.

**ESLint** examine le code JavaScript et signale les erreurs potentielles ou les violations de règles de style. Il est très flexible et peut être configuré pour s'adapter à différents styles de codage et à différents projets.



ESLint



# Eslint

Informations générales

Quelques règles générales :

- Variables non déclarées
- Variables inutilisées
- Boucles infinies
- Avertissement divers

Quelques règles par des entreprises :





# Paquets Manager

## Définitions et différences

Un gestionnaire de paquets, ou "package manager" en anglais, est un outil logiciel qui permet d'automatiser le processus d'installation, de mise à jour, de configuration et de suppression de logiciels ou de bibliothèques logicielles dans un système informatique. Il gère les dépendances entre les paquets et s'assure que tous les fichiers et configurations nécessaires sont installés correctement.





# ES6

## Ecmascript

**ECMAScript 6**, également connu sous le nom **d'ES6** et **ECMAScript 2015**, est une version majeure de la spécification du langage de programmation **JavaScript** qui a introduit des améliorations significatives et de nouvelles fonctionnalités.

Telles que les classes, les promesses, les flèches de fonction, et bien d'autres, visant à faciliter le développement et à rendre le code JavaScript plus lisible et puissant.

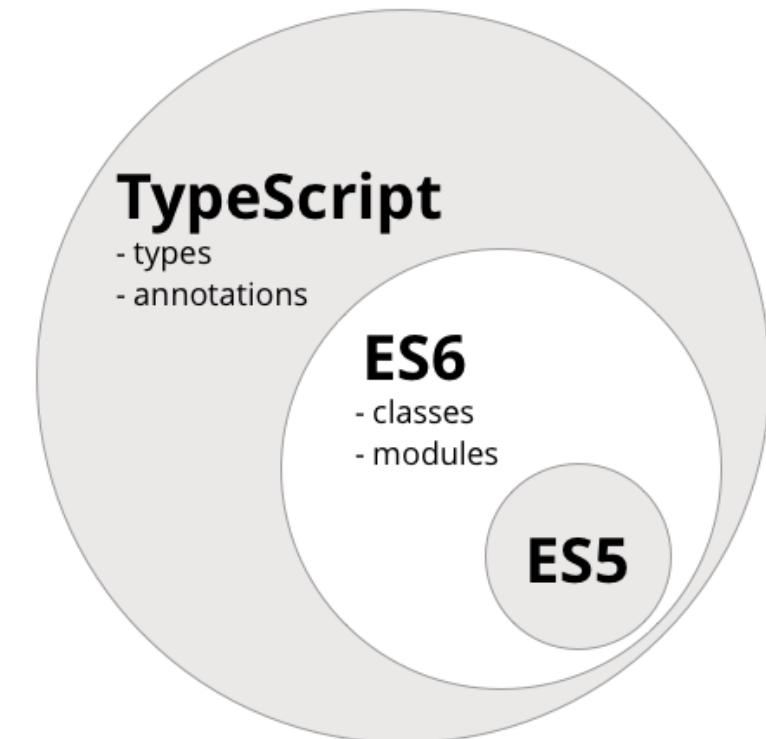




# ES6

## Transpilation

La transpilation d'ECMAScript 6, ou ES6, fait référence au processus de conversion du code écrit en ES6 en une version antérieure d'ECMAScript, comme ES5, qui est largement compatible avec la plupart des navigateurs et des environnements d'exécution de JavaScript.



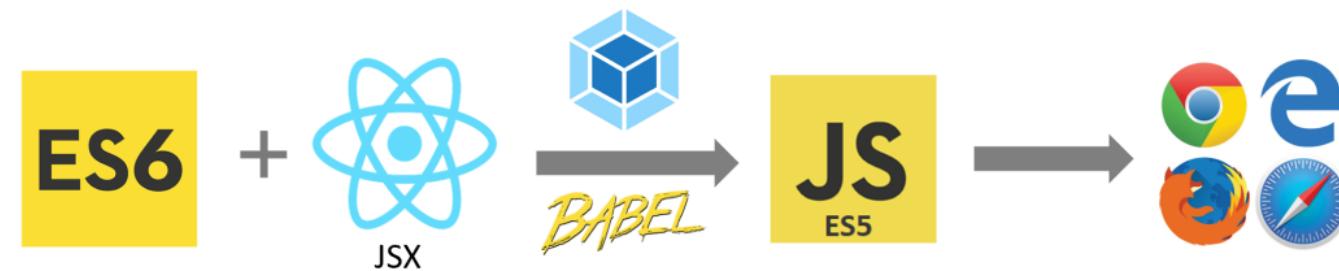


# ES6

## Exemple simple de Transpilation

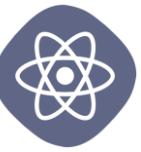
Plusieurs outils de transpilation existent :

- **Babel JS**
- **Webpack**



// Utilisation de la syntaxe de fonction fléchée d'ES6  
const greet = (name) => `Hello, \${name}!`;

// Le même code transpilé en ES5  
var greet = function(name) {  
 return 'Hello, ' + name + '!';  
};



# TypeScript

## TypeScript

**TypeScript** est un sur-ensemble typé de **JavaScript** développé par **Microsoft**, qui ajoute des types statiques et d'autres fonctionnalités pour améliorer la lisibilité et la maintenabilité du code, tout en offrant des outils robustes pour la détection d'erreurs lors de la phase de développement.





# ES6

## Variab le et mutabilité

Dans le langage de programmation **JavaScript**, les mots-clés **let**, **const**, et **var** sont utilisés pour déclarer des variables, chacun offrant des niveaux différents de portée et de mutabilité, permettant aux développeurs de gérer avec précision la disponibilité et la modification des données dans leurs programmes.



```
if (true) {  
    let x = 10; // x est accessible uniquement à l'intérieur de ce bloc  
}  
console.log(x); // Erreur: x n'est pas défini
```



```
const y = 20; // y doit être initialisé lors de la déclaration  
y = 30; // Erreur: l'assignation à une variable constante
```



```
if (true) {  
    var z = 40; // z est accessible à l'extérieur de ce bloc  
}  
console.log(z); // Affiche 40
```



# ES6

## Typage et type natifs

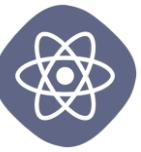
JavaScript est un langage à **typage dynamique**. Cela signifie que le type d'une variable est déterminé à l'exécution et peut changer au cours de la vie de la variable.

TypeScript est un sur-ensemble de JavaScript qui introduit le **typage statique**. Vous pouvez spécifier explicitement le type de variable lors de sa déclaration.

### Type natif :

- Number
- String
- Boolean
- Object
- Array
- Null
- Undefined
- Symbol (introduit avec ES6)





# ES6

## Paramètres optionnels, valeurs par défaut

En JavaScript et TypeScript, vous pouvez définir des **paramètres optionnels** et des **valeurs par défaut** pour les fonctions, ce qui vous donne une plus grande flexibilité dans la manière dont vous appelez ces fonctions.

```
● ○ ●  
function greet(name) {  
    console.log("Hello, " + name + "!");  
}  
  
greet(); // Affiche "Hello, undefined!"
```



```
● ○ ●  
function greet(name?: string) {  
    console.log("Hello, " + name + "!");  
}  
  
greet(); // Affiche "Hello, undefined!"
```

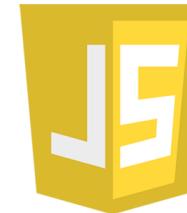


# ES6

## Classes et Interfaces

Les **classes** et les **interfaces** sont deux concepts clés de la programmation orientée objet en TypeScript et, dans une certaine mesure, en JavaScript avec l'introduction d'**ES6**.

```
● ● ●  
class Person {  
    constructor(name, age) {  
        this.name = name;  
        this.age = age;  
    }  
  
    greet() {  
        console.log(`Hello, my name is ${this.name} and I am ${this.age}  
years old.`);  
    }  
  
    const person = new Person("Alice", 30);  
    person.greet(); // Affiche "Hello, my name is Alice and I am 30 years old"
```



```
● ● ●  
class Person {  
    name: string;  
    age: number;  
  
    constructor(name: string, age: number) {  
        this.name = name;  
        this.age = age;  
    }  
  
    greet(): void {  
        console.log(`Hello, my name is ${this.name} and I am ${this.age}  
years old.`);  
    }  
}
```

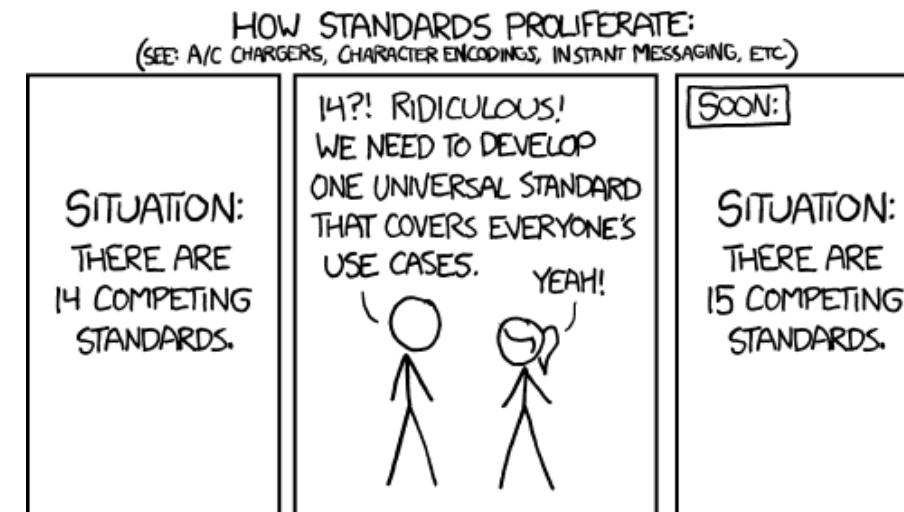


# ES6

## Gestion des modules

La **gestion des modules** est une fonctionnalité qui permet de diviser le code en plusieurs fichiers ou modules afin d'organiser et de structurer le code de manière plus claire.

Chaque module peut avoir ses propres dépendances et peut être réutilisé dans d'autres modules ou projets. Les systèmes de modules les plus couramment utilisés en JavaScript sont **CommonJS** et **ES6 Modules**.





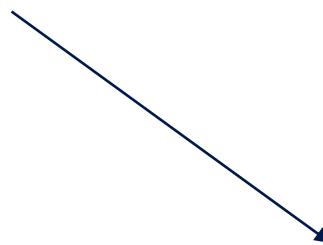
# ES6

## Gestion des modules

Nous avons donc maintenant une manière bien définie d'importer des modules

```
// Export par défaut
const MyComponent = () => {
  return <div>Hello World</div>;
};

export default MyComponent;
```



```
// Import par défaut
import MyComponent from './MyComponent';

function App() {
  return <MyComponent />;
}

export default App;
```



# ES6

## Les fonctions fléchées

Les **fonctions fléchées** (Arrow functions) ont été introduites dans ECMAScript 2015 (ES6) et offrent une syntaxe plus concise pour écrire des fonctions en JavaScript.

```
● ● ●

class MyClass {
  constructor() {
    this.myValue = 1;
  }

  traditionalFunction() {
    document.body.addEventListener('click', function() {
      console.log(this.myValue); // Erreur, `this` réfère à `document.body`
    });
  }

  arrowFunction() {
    document.body.addEventListener('click', () => {
      console.log(this.myValue); // Affiche 1, `this` réfère à l'instance de
`MyClass`;
    });
  }
}
```

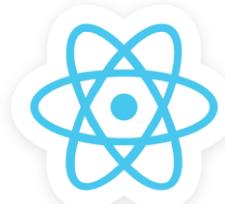


# ES6

## Les fonctions fléchées

Voyons des exemples en React :

```
● ● ●  
// Importation de React  
import React, { Component } from 'react';  
  
// Composant de classe  
class TraditionalComponent extends Component {  
  render() {  
    return (  
      <div>  
        <h1>Bonjour depuis un Composant Traditionnel !</h1>  
      </div>  
    );  
  }  
  
// Exportation du composant  
export default TraditionalComponent;
```



```
● ● ●  
// Importation de React  
import React from 'react';  
  
// Composant fonctionnel utilisant une fonction fléchée  
const ArrowFunctionComponent = () => {  
  return (  
    <div>  
      <h1>Bonjour depuis un Composant Fonctionnel avec Fonction  
      Fléchée !</h1>  
    </div>  
  );  
};  
  
// Exportation du composant  
export default ArrowFunctionComponent;
```



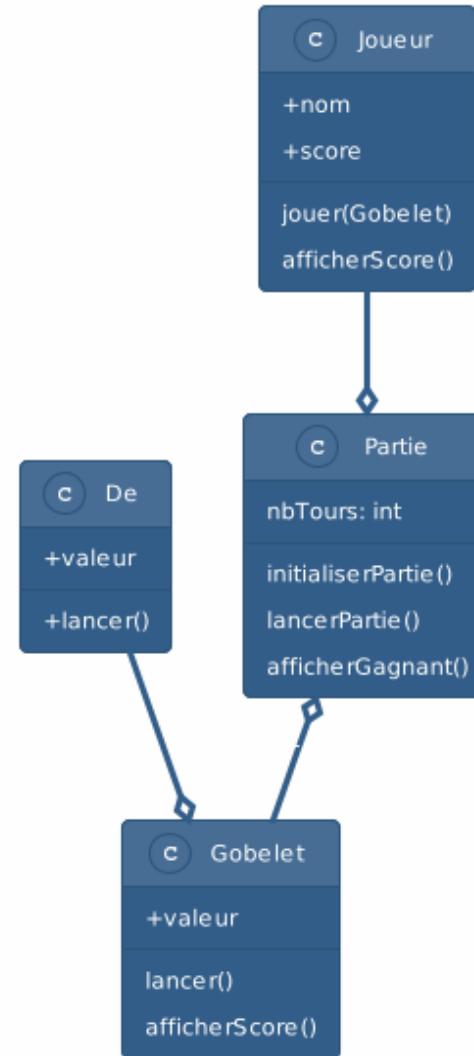
# Exercice

## Exercice Typescript

```
import Game from "./Classes/Game";
import Player from "./Classes/Player";

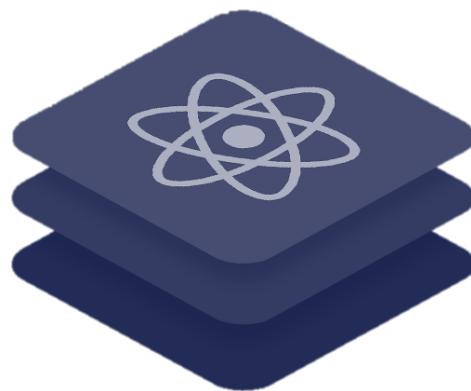
const player1: Player = new Player('player1');
const player2: Player = new Player('player2');
const player3: Player = new Player('player3');

const game: Game = new Game();
game.initializeGame(player1, player2, player3);
game.playGame();
```



# Partie 2 :

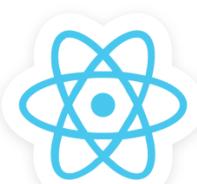
## Le framework ReactJS



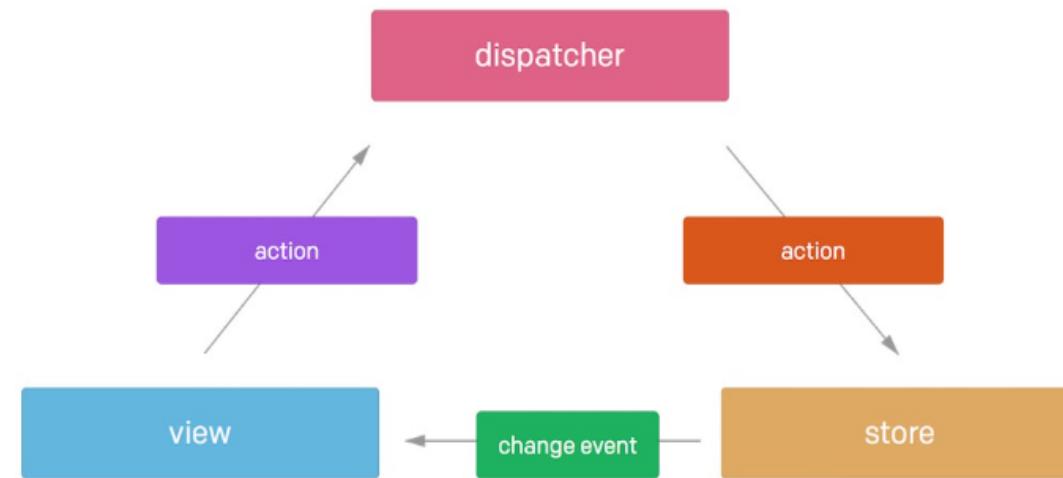
# Le framework React JS

## Principes de bases

React, développé et maintenu par Facebook, est l'une des bibliothèques JavaScript les plus populaires pour la construction d'interfaces utilisateur interactives. Voici quelques principes de base et avantages de React par rapport à ses concurrents tels que Angular, Vue, etc.



- ✓ Composants et Réutilisabilité
- ✓ Virtual DOM
- ✓ Unidirectional Data Flow
- ✓ Ecosystème et Communauté
- ✓ Flexibilité
- ✓ JSX

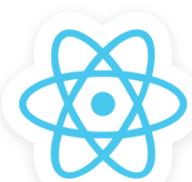




# Le framework React JS

## Philosophie composant

La philosophie de React repose fortement sur le concept de « composants ». Cette approche divise l'interface utilisateur en unités indépendantes, réutilisables et modulables.



- ✓ Tout est composant
- ✓ Réutilisabilité
- ✓ Composition
- ✓ État Local
- ✓ Isolation et Encapsulation
- ✓ Maintenance Évolutivité
- ✓ Cohérence de l'interface Utilisateur

```
// Importation de React
import React from 'react';

// Définition du type des props à l'aide d'une interface
interface GreetingProps {
  name: string;
}

// Création d'un composant fonctionnel avec une arrow function
const Greeting: React.FC<GreetingProps> = ({ name }) => {
  return (
    <div>
      <p>Hello, {name}!</p>
    </div>
  );
};

// Exportation du composant
export default Greeting;
```



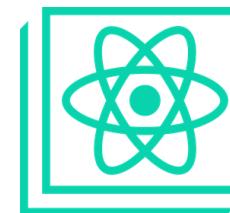
# Le framework React JS

## Workflow de développement

Les workflows de développement avec React peuvent varier considérablement selon le contexte du projet. Voici une description de trois scénarios courants : créer une application à partir de zéro, intégrer React dans une application web existante, et utiliser un outil de génération comme `create-react-app`.

Les questions à se poser :

- Taille ?
- Le type d'exécution
- Builder ?
- Native ESM / SSR ?





# Le framework React JS

Démonstration d'une Card avec du glassmorphism



**Exercice :**



- Création d'un starter react avec l'aide du CRA et de Vite



- Création d'un dossier composant
- Création d'un dossier Card à l'intérieur de celui-ci
- Création de deux fichiers :
  - Card.tsx
  - Card.css



# Strict Mode

## Informations générales

Le "**Strict mode**" en React est un outil de développement qui aide les développeurs à repérer les problèmes potentiels dans leur application React en émettant des avertissements supplémentaires et en activant certains comportements expérimentaux pour faciliter la détection des problèmes.



```
● ● ●  
import React from 'react';  
import ReactDOM from 'react-dom';  
import App from './App';  
  
ReactDOM.render(  
  <React.StrictMode>  
    <App />  
  </React.StrictMode>,  
  document.getElementById('root')  
>;
```



# Strict Mode

## Informations générales

Lorsque vous utilisez le "**Strict mode**", React effectue certaines vérifications supplémentaires lors de la création de l'application, telles que

- Vérification de la signature des méthodes du cycle de vie du composant.
- Avertissements lorsqu'une application utilise des API obsolètes de React.
- Vérification des changements dans le nombre de props d'un composant.

**Le Strict Mode a évolué dans la dernière mise à jour de react 18**



# Le framework React JS

## Démonstration Card.tsx

```
import './Card.css';

type CardProps = {
  title: string;
  content: string;
}

const Card = ({ title, content }: CardProps) => {
  return (
    <div className="card">
      <h2>{title}</h2>
      <p>{content}</p>
    </div>
  );
};

export default Card;
```



# Le framework React JS

## Démonstration Card.css

```
● ● ●

.card {
  background: rgba(255, 255, 255, 0.1);
  border-radius: 15px;
  padding: 20px;
  width: 300px;
  margin-bottom: 20px;
  margin-left: 20px;
  margin-right: 20px;
  backdrop-filter: blur(10px);
  border: 1px solid rgba(255, 255, 255, 0.2);
  box-shadow: 0 4px 6px rgba(0, 0, 0, 0.1);
  transition: 0.5s;
  cursor: pointer;
  overflow: hidden;
}

.card:hover {
  box-shadow: 0 8px 12px rgba(0, 0, 0, 0.1);
  transform: scale(1.05);
}

.card h1 {
  font-size: 1.5em;    margin-bottom: 10px;
}

.card p {
  font-size: 1em;
  color: #ccc;
}
```



# Le framework React JS

## Démonstration App.tsx

```
import './App.css'
import Card from './components/card/Card'
import Header from './components/header/Header'

function App() {

  const cardsData = {
    title: 'Orage', content: 'Température : 10 degrés'
  };

  return (
    <>
      <div style={{ background: 'linear-gradient(135deg, #667eea 0%, #764ba2 100%)',
minHeight: '100vh', padding: '20px' }}>
        <Header />
        <Card title={ cardsData.title } content={ cardsData.content } />
      </div>
    </>
  )
}
export default App;
```



# Le framework React JS

Test et pratique

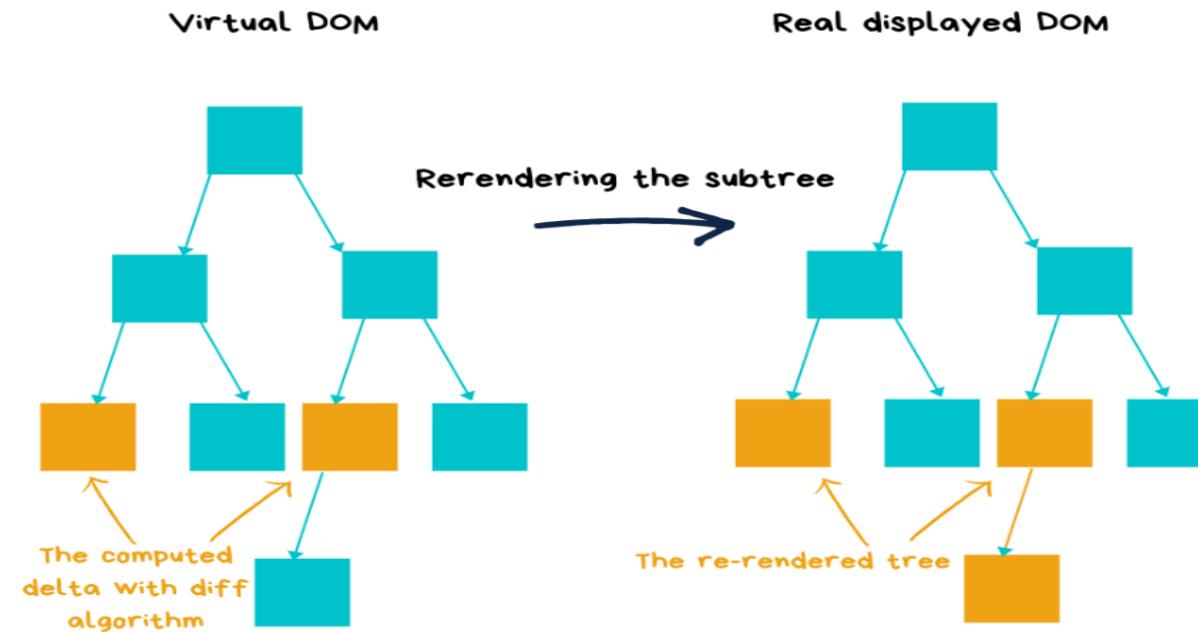




# Le framework React JS

## Dom virtuel et réconciliation

Le **DOM Virtuel** et la **Réconciliation** sont deux concepts clés qui permettent à React de construire des interfaces utilisateur performantes et efficaces.





# Le framework React JS

## Dom et ajout de nœud

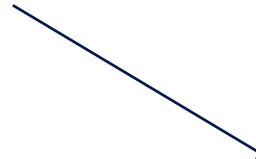
Lors de la création de composants React, vous pouvez rencontrer des situations où vous souhaitez rendre plusieurs éléments à la fois, mais sans les envelopper dans un élément parent supplémentaire (comme une div). Les fragments de React sont utilisés dans ces situations.



```
const Group = () => {
  return (
    <div>
      <p>Élément 1</p>
      <p>Élément 2</p>
    </div>
  );
};
```

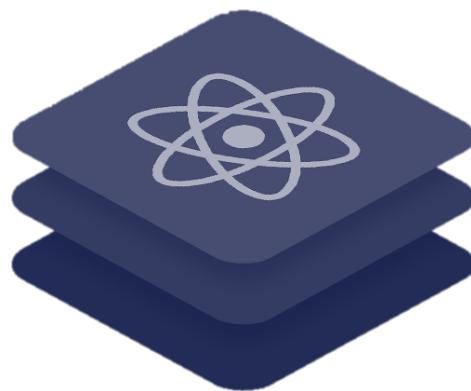


```
const Group = () => {
  return (
    <>
      <p>Élément 1</p>
      <p>Élément 2</p>
    </>
  );
};
```



# Partie 3 :

## Les JSX et les composants



# Les JSX et les composants

## Element react

Un **élément React** est un objet qui décrit ce que l'on souhaite voir dans l'interface utilisateur. C'est une description légère, immuable, et créée à partir de composants React.

Les éléments React peuvent être de divers types et contiennent des attributs ainsi que des enfants.

### Types :

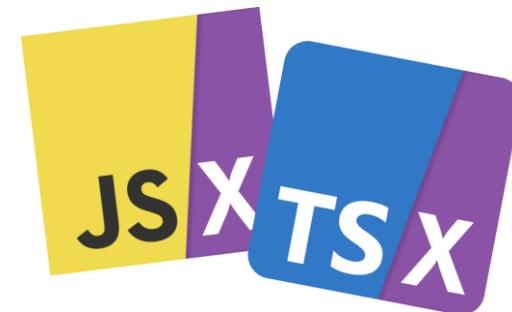
- élément html
- élément de composant

### Attributs:

- élément html
- élément de composant

### Composition :

- élément html
- élément de composant



# Les JSX et les composants

## Liaison avec le DOM

La liaison avec le DOM dans React est gérée principalement par le package **ReactDOM**. C'est **ReactDOM** qui fait le pont entre vos composants React (et l'arbre du DOM virtuel qu'ils produisent) et le DOM réel de la page web.

La méthode **ReactDOM.render()** est particulièrement importante dans ce processus

```
● ● ●

import React from 'react';
import ReactDOM from 'react-dom';

const App = () => {
  return <h1>Bonjour, monde !</h1>;
};

// Rendre le composant App dans l'élément avec l'ID 'root' du DOM
// réel
ReactDOM.render(<App />, document.getElementById('root'));
```

# Les JSX et les composants

Une nouvelle syntaxe le JSX

JSX est une extension de syntaxe pour JavaScript, recommandée par React, qui ressemble beaucoup à du XML ou du HTML.

Elle permet d'écrire des structures qui ressemblent à du HTML directement dans votre code JavaScript, rendant le code plus lisible et expressif.

La JSX permet plus simplement :

- Elément et balises
- Expressions
- Composants
- Attributs (attention aux quelques différences)
- Enfants

BABEL



```
const element = <h1>Bonjour, monde!</h1>;
```



```
const element = React.createElement('h1', null, 'Bonjour, monde!');
```

# Les JSX et les composants

## Les règles du JSX

Le JSX, bien qu'il ressemble à du HTML, a certaines règles et caractéristiques propres qui sont importantes à connaître lorsqu'on travaille avec React.

Échappement automatique :



```
const name = "Monde";
const element = <h1>Bonjour, {name}</h1>;
```

Parent Unique :



```
// Incorrect
const element = (
  <h1>Titre</h1>
  <p>Paragraphe</p>
);

// Correct
const element = (
  <div>
    <h1>Titre</h1>
    <p>Paragraphe</p>
  </div>
);
```

Fragment React :



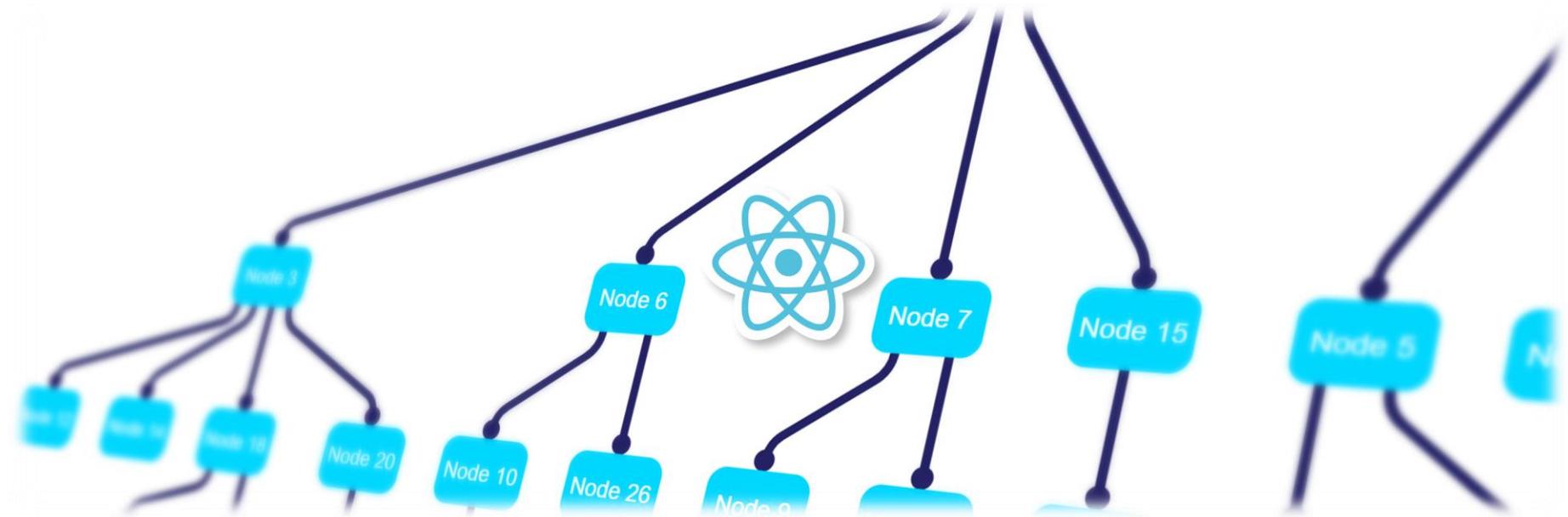
```
// Utilisation de fragment pour éviter l'ajout d'un nœud supplémentaire
const element = (
  <>
    <h1>Titre</h1>
    <p>Paragraphe</p>
  </>
);
```

# Les JSX et les composants

## Les composants

Un composant dans le contexte des bibliothèques de développement front-end comme **React** est une unité indépendante de code UI. Il représente généralement un élément de l'interface utilisateur, comme un bouton, une carte, un formulaire, etc. L'intérêt principal des composants est la réutilisabilité.

Un composant peut être défini une fois et utilisé à plusieurs endroits, facilitant la maintenance et la cohérence de l'interface utilisateur.



# Les JSX et les composants

## Les composants classe

Avant l'introduction des Hooks dans React 16.8, les composants basés sur des classes étaient le moyen standard pour créer des composants avec des états et des cycles de vie.

Un composant de classe est une classe ES6 qui étend **React.Component** et implémente une méthode **render()** pour définir ce qui doit être affiché.



```
import React, { Component } from 'react';

class MyComponent extends Component {
  render() {
    return <h1>Hello, World!</h1>;
  }
}

export default MyComponent;
```

# Les JSX et les composants

## Les composants fonctionnels

Avec l'introduction des Hooks, React a encouragé l'utilisation de composants fonctionnels, qui sont plus concis et offrent une meilleure lisibilité.

Les composants fonctionnels sont simplement des fonctions JavaScript qui prennent des **props** en tant qu'argument et retournent du JSX.

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

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

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

export default Counter;
```

# Les JSX et les composants

Test et pratique



# Les JSX et les composants

## Rendu Conditionnel

Les méthodes de tableau comme **map** et **filter** sont couramment utilisées pour traiter et afficher des listes de composants.

```
● ○ ●  
  
type User = {  
  id: number;  
  name: string;  
};  
  
const users: User[] = [  
  { id: 1, name: 'Alice' },  
  { id: 2, name: 'Bob' },  
  // ...  
];
```



```
● ○ ●  
  
import React from 'react';  
  
const UserList: React.FC<{ users: User[] }> = ({ users }) => {  
  return (  
    <ul>  
      {users.map(user => (  
        <li key={user.id}>{user.name}</li>  
      ))}  
    </ul>  
  );  
};
```

# Les JSX et les composants

## Rendu Conditionnel

Les méthodes de tableau comme **map** et **filter** sont couramment utilisées pour traiter et afficher des listes de composants.



```
const FilteredUserList: React.FC<{ users: User[] }> = ({ users }) =>
{
  return (
    <ul>
      {users
        .filter(user => user.name.startsWith('A'))
        .map(filteredUser => (
          <li key={filteredUser.id}>{filteredUser.name}</li>
        )));
    </ul>
  );
};
```

# Les JSX et les composants

## Exercice

**Objectif :** Créer un composant qui affiche une liste de tâches avec leur statut.



```
const tasks: Task[] = [
  { id: 1, title: 'Faire la vaisselle', isDone: false },
  { id: 2, title: 'Promener le chien', isDone: true },
  { id: 3, title: 'Faire les courses', isDone: false },
];
```

**Objectif 2 :** Créer un composant qui affiche une liste de tâches faites

# Les JSX et les composants

## Démonstration



- Création d'un composant **Header**
- Création d'un nouveau composant : "CardList" (composant parent de Card)
  - celui-ci affichera une liste de Card avec une fonction **map**
  - Vous n'êtes pas obligés d'utiliser les **props**

# Les JSX et les composants

## Project Structure

La structuration d'un projet React et la division de ses composants sont cruciales pour assurer la lisibilité, la maintenabilité et la réutilisabilité du code.

Il n'a pas de bonne solution...

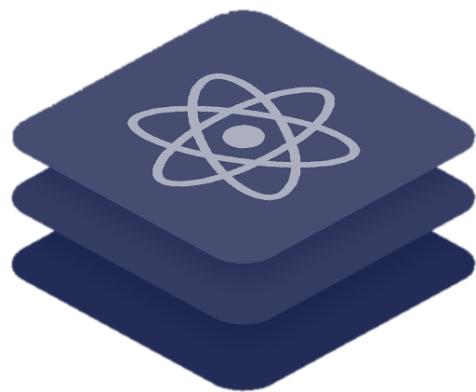
Indication :

- Division Simple (Composant, domaine métier, feature )
- Design Pattern (Adapter par exemple pour le CSS)



# Partie 4 :

## Les props





# Les Props

## Définition (transmission de données, readonly)

Les **props** (abréviation de "properties" ou propriétés) sont un mécanisme essentiel dans React qui permet de transmettre des données entre composants. Elles servent à passer des valeurs d'un composant parent à un composant enfant.



```
const Enfant = (props) => {
  return <p>Bonjour, {props.nom} !</p>;
};
```



```
const Parent = () => {
  return (
    <div>
      <h1>Exemple de Props</h1>
      <Enfant nom="Alice" />
      <Enfant nom="Bob" />
    </div>
  );
};
```



# Les Props

## Classe/Fonctionnel

L'accès aux **props** dans les composants React varie légèrement selon que vous travaillez avec des composants fonctionnels ou des composants de classe.



```
import React, { Component } from 'react';

// Composant de classe
class MonComposantClasse extends Component {
  render() {
    return <p>Bonjour, {this.props.nom} !</p>;
  }
}

// Utilisation
<MonComposantClasse nom="Alice" />;
```

### Fonctionnel :



```
// Destructuration des props
const MonComposantFonctionnel = ({ nom }) => {
  return <p>Bonjour, {nom} !</p>;
};

// Utilisation
<MonComposantFonctionnel nom="Alice" />;
```



# Les Props

## Spread Operator

Le spread operator permet de 'propager' ou de distribuer les propriétés d'un objet dans un autre

- **Passer Toutes les Props à un Enfant**
- **Combiner les Props avec D'autres Valeurs**
- **Extraire Certaines Props et Passer le Reste**



```
interface ChildProps {
  propA: string;
  propB: number;
}

const ChildComponent = ({ propA, propB }: ChildProps) => {
  // ... utilisez propA et propB
  return <div>{propA} {propB}</div>;
};

interface ParentProps extends ChildProps {
  propC: boolean;
}

const ParentComponent = ({ propC, ...childProps }: ParentProps) => {
  return (
    <div>
      {propC && <ChildComponent {...childProps} />}
    </div>
  );
};
```



# Les Props

Test et pratique





# Les Props

## Props Children

La prop **children** est une prop spéciale dans React qui permet de passer des composants comme enfants à d'autres composants dans le cadre de la décomposition.

```
● ● ●

// Composant Parent
const Boite = (props) => {
  return <div className="boite">{props.children}</div>;
};

// Utilisation
const App = () => {
  return (
    <Boite>
      <p>Ceci est un enfant !</p>
    </Boite>
  );
};
```



# Les Props

## Différence entre React.FC et Interface

Quand vous utilisez React.FC (ou React.FunctionComponent), TypeScript comprend que votre composant peut recevoir des children même si vous ne les déclarez pas explicitement dans l'interface de vos props. C'est parce que React.FC est déjà défini avec children comme partie de son type

```
● ● ●

// Sans React.FC, vous devez déclarer 'children' si vous voulez l'utiliser :
interface PropsWithoutReactFC {
  myProp: string;
  children?: React.ReactNode; // Children doit être déclaré explicitement
}

const MyComponentWithoutReactFC = ({ myProp, children }: PropsWithoutReactFC) => {
  // ...
};

// Avec React.FC, 'children' est déjà compris dans le type :
interface PropsWithReactFC {
  myProp: string;
  // Pas besoin de déclarer 'children', il est déjà inclus
}

const MyComponentWithReactFC: React.FC<PropsWithReactFC> = ({ myProp, children }) => {
  // ...
};
```



# Les Props

## Démonstration



- Création d'un composant CardImage, le composant parent sera la Card

Phase 2 :



- Création d'un composant Sidebar qui prendra des "props children"



# Les Props

## Props Function

Passer une fonction en tant que prop dans React est une pratique courante qui permet de créer des interactions dynamiques entre les composants. Cela peut être utile pour transmettre une fonction depuis un composant parent à un composant enfant afin que l'enfant puisse déclencher des actions dans le parent.

```
● ● ●  
import React, { FC, useCallback } from 'react';  
import Enfant from './Enfant';  
  
const Parent: FC = () => {  
  const handleChildClick = useCallback((message: string) => {  
    alert(`Enfant dit : ${message}`);  
  }, []);  
  
  return <Enfant onClick={handleChildClick} />;  
};  
  
export default Parent;
```



```
● ● ●  
import React, { FC } from 'react';  
  
interface EnfantProps {  
  onClick: (message: string) => void;  
}  
  
const Enfant: FC<EnfantProps> = ({ onClick }) => {  
  return (  
    <button onClick={() => onClick("Bonjour Parent !")}>  
      Cliquez moi  
    </button>  
  );  
};  
  
export default Enfant;
```



# Les Props

## Rappel sur les méthodes de gestion d'évènement

Les méthodes telles que onClose, onClick, onChange, et autres, sont appelées **gestionnaires d'événements** ou **handlers** en anglais. Ces fonctions sont exécutées en réponse à certaines actions de l'utilisateur, comme des clics de souris, des frappes de clavier, des mouvements de souris, etc.

```
● ● ●

function Bouton({ onClick }) {
  return <button onClick={onClick}>Cliquez sur moi</button>;
}

function App() {
  const handleClick = () => {
    console.log("Le bouton dans App a été cliqué");
  };

  return <Bouton onClick={handleClick} />;
}
```



# Les Props

## Rappel sur les méthodes de gestion d'évènement

Dans le JSX, les noms des événements sont écrits en camelCase et sont placés dans des accolades.



```
<button onClick={this.handleClick}>Cliquez sur moi</button>
```

Les gestionnaires d'événements sont des fonctions qui sont appelées en réponse à des événements déclenchés



```
handleClick() {  
  console.log('Le bouton a été cliqué');  
}
```

Vous pouvez passer des arguments supplémentaires à votre gestionnaire d'événements.



# Les Props

## Rappel sur les méthodes de gestion d'évènement

La liaison du contexte d'exécution dans React fait référence à la façon dont nous nous assurons que la valeur de `this` dans une méthode de gestion d'événement (handler) pointe vers l'instance correcte du composant



```
class MonComposant extends React.Component {
  constructor(props) {
    super(props);
    this.handleClick = this.handleClick.bind(this);
  }

  handleClick() {
    console.log('this est:', this); // Ici, `this` se réfère à l'instance de
MonComposant
  }

  render() {
    return <button onClick={this.handleClick}>Cliquez sur moi</button>;
  }
}
```



```
class MonComposant extends React.Component {
  handleClick = () => {
    console.log('this est:', this); // Ici aussi, `this` se réfère à
l'instance de MonComposant
  }

  render() {
    return <button onClick={this.handleClick}>Cliquez sur moi</button>;
  }
}
```



# Les Props

## Rappel sur les méthodes de gestion d'évènement

Dans les composants fonctionnels, il n'y a pas de concept de **this**, donc vous n'avez pas besoin de vous soucier de la liaison comme dans les composants de classe. Toutefois, la gestion des fonctions et des callbacks reste un aspect important de la performance et de la lisibilité du code

```
● ● ●  
function MonComposant() {  
  const handleClick = (e) => {  
    console.log('Bouton cliqué!');  
  };  
  
  return <button onClick={handleClick}>Cliquez sur moi</button>;  
}
```

```
● ● ●  
function Parent() {  
  const handleChildClick = (value) => {  
    console.log('La valeur reçue de Child est:', value); // La valeur reçue  
    de Child est: 1  
  };  
  
  return <Child onChildClick={handleChildClick} />;  
}  
  
function Child({ onChildClick }) {  
  return <button onClick={() => onChildClick(1)}>Cliquez sur moi</button>;  
}
```



# Les Props

Test et pratique





# Les Props

## Démonstration



- Crédation d'un composant Bouton (dans chaque Card)
- Crédation d'une alerte lors du click du Bouton (props function)



# Les Props

## Props drilling et bonnes pratiques

**Props Drilling** est un terme utilisé dans le contexte de React pour décrire le processus par lequel les **props** sont passées à travers plusieurs niveaux de composants imbriqués, même si certains de ces composants intermédiaires n'ont pas besoin de ces données pour leur propre logique

```
● ● ●  
import React, { FC, useState } from 'react';  
import Parent from './Parent';  
  
const GrandParent: FC = () => {  
  const [message, setMessage] = useState<string>('Message du GrandParent');  
  
  return (  
    <div>  
      <h1>GrandParent</h1>  
      <Parent message={message} />  
    </div>  
  );  
  
  export default GrandParent;
```

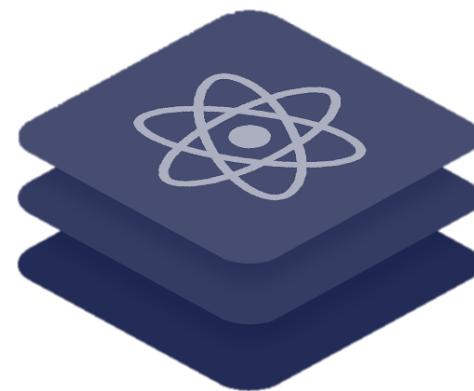


```
● ● ●  
import React, { FC } from 'react';  
import Enfant from './Enfant';  
  
interface ParentProps {  
  message: string;  
}  
  
const Parent: FC<ParentProps> = ({ message }) => {  
  return (  
    <div>  
      <h2>Parent</h2>  
      <Enfant message={message} />  
    </div>  
  );  
  
  export default Parent;
```



# Partie 5 :

## Le State et les lifecycles

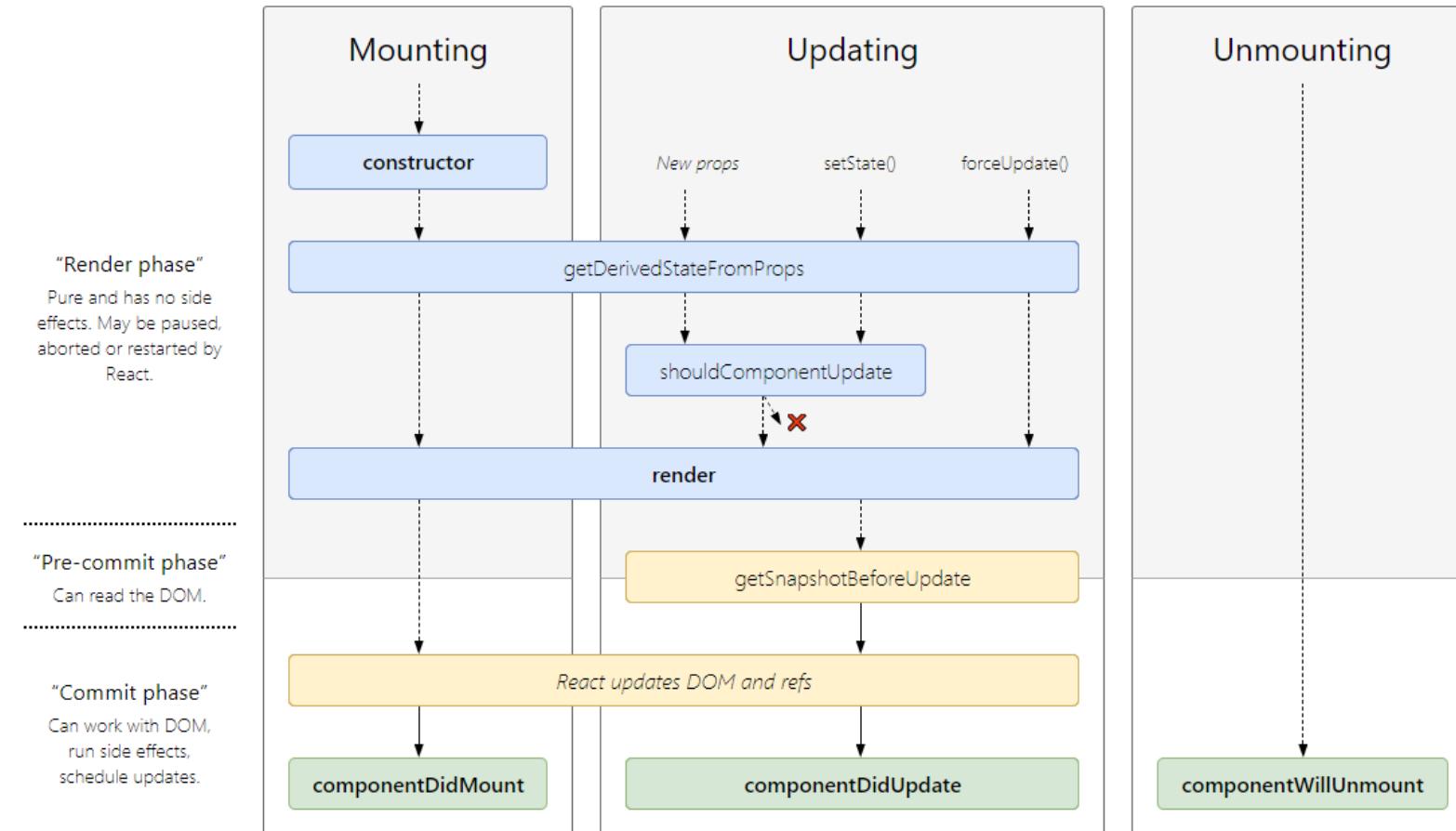




# Le State et les lifecycles

## Cycle de vie

Le cycle de vie d'un composant React peut être divisé en trois phases principales : **Montage**, **Mise à jour**, et **Démontage**.





# Le State et les lifecycles

Exemple avec un composant de classe

```
● ● ●

import React, { Component } from 'react';

class ComposantClasse extends Component {
  state = { compteur: 0 };

  componentDidMount() {
    console.log('Composant monté');
  }

  componentDidUpdate() {
    console.log('Composant mis à jour');
  }

  componentWillUnmount() {
    console.log('Composant va être démonté');
  }

  render() {
    return (
      <div>
        <p>Composant</p>
      </div>
    );
  }
}
```



# Le State et les lifecycles

## State

Le **state** dans React fait référence à l'état local d'un composant. C'est une manière de stocker et de gérer des données qui sont spécifiques à un composant et qui peuvent changer avec le temps. Lorsque le **state** d'un composant change, React re-rend automatiquement le composant pour refléter ces changements.

```
● ● ●

class MonComposant extends React.Component {
  constructor(props) {
    super(props);
    this.state = {
      compteur: 0
    };
  }

  incrementer = () => {
    this.setState({ compteur: this.state.compteur + 1 });
  };

  render() {
    return (
      <div>
        <p>Compteur: {this.state.compteur}</p>
        <button onClick={this.incrementer}>Incrémenter</button>
      </div>
    );
  }
}
```



# Le State et les lifecycles

## State et composant fonctionnel

Avec les composants fonctionnels, vous pouvez utiliser le Hook **useState** pour ajouter un état local.



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

const MonComposant = () => {
  const [compteur, setCompteur] = useState(0);

  const incrementer = () => {
    setCompteur(compteur + 1);
  };

  return (
    <div>
      <p>Compteur: {compteur}</p>
      <button onClick={incrementer}>Incrémenter</button>
    </div>
  );
};
```



# Le State et les lifecycles

## Décomposons

Le hook **useState** est un hook fondamental dans React pour gérer l'état local d'un composant. Il permet d'ajouter un état réactif à un composant fonctionnel, ce qui n'était possible auparavant qu'avec les composants de classe.



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

Plusieurs façons :



```
const [age, setAge] = useState(25);
const [fruit, setFruit] = useState('banane');
const [todos, setTodos] = useState([{ text: 'Apprendre React' }]);
```



# Le State et les lifecycles

Test et pratique





# Le State et les lifecycles

## Forme du State

Initialisation paresseuse :



```
const [someValue, setSomeValue] = useState(() => {  
  // quelques calculs coûteux  
  return calculatedValue;  
});
```



```
import React, { useState } from 'react';  
  
function Compteur() {  
  const [count, setCount] = useState(0);  
  
  const incrementer = () => {  
    // Mise à jour fonctionnelle : utilise la valeur courante de `count`  
    setCount(prevCount => prevCount + 1);  
  };  
  
  return (  
    <div>  
      <p>Vous avez cliqué {count} fois</p>  
      <button onClick={incrementer}>  
        Cliquez moi  
      </button>  
    </div>  
  );  
}
```

Mise à jour fonctionnelle :



# Le State et les lifecycles

## Test et pratique



- Créez une fonction de composant nommée Counter qui affiche un compteur et deux boutons, "Increment" et "Reset".
- Utilisez useState dans la fonction Counter pour déclarer une variable d'état nommée count initialisée à 0.
- Affichez la valeur de count dans le rendu du composant.
- Créez une fonction nommée handleIncrement qui incrémente la valeur de count de 1 lorsqu'elle est appelée.
- Créez une fonction nommée handleReset qui réinitialise la valeur de count à 0 lorsqu'elle est appelée.
- Assignez les fonctions handleIncrement et handleReset aux propriétés onClick des boutons respectivement.



# Le State et les lifecycles

## Rendu conditionnel

React permet d'inclure des expressions JavaScript directement dans le JSX, ce qui facilite le rendu conditionnel de composants ou d'éléments.



```
{condition ? <ComponentIfTrue /> : <ComponentIfFalse />}
```



```
{condition && <ComponentIfTrue />}
```



# Le State et les lifecycles

## Démonstration



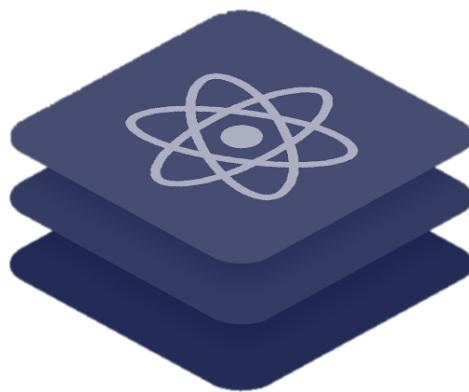
- Création d'un composant Alert afin de remplacer l'alert du navigateur lors du click sur le bouton de chaque Card

## Phase 2 :



- Modification de la barre latérale (Sidebar)
  - elle se replie lorsque la souris n'est plus dessus.

# Partie 6 : Les Hooks





# Les Hooks

## Définition

Les Hooks sont une addition à React introduite dans la version 16.8 qui permet d'utiliser l'état et d'autres fonctionnalités de React sans écrire une classe. Ils sont fondamentaux pour écrire des composants fonctionnels dans React.



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



```
useEffect(() => {
  // Code qui s'exécute après le rendu du composant
  return () => {
    // Code de nettoyage (équivalent à componentWillUnmount)
  };
}, [dependencies]); // La liste de dépendance
```



# Les Hooks

## Définition

**useEffect** remplace les méthodes de cycle de vie dans les composants fonctionnels. Il peut être configuré pour s'exécuter après chaque rendu ou seulement après certains rendus en définissant une liste de dépendance.

```
● ● ●  
useEffect(() => {  
  // Code qui s'exécute après le rendu du composant  
  return () => {  
    // Code de nettoyage (équivalent à componentWillUnmount)  
  };  
}, [dependencies]); // La liste de dépendance
```

```
● ● ●  
import { useEffect } from 'react';  
import { createConnection } from './chat.js';  
  
function ChatRoom({ roomId }) {  
  const [serverUrl, setServerUrl] = useState('https://localhost:1234');  
  
  useEffect(() => {  
    const connection = createConnection(serverUrl, roomId);  
    connection.connect();  
    return () => {  
      connection.disconnect();  
    };  
  }, [serverUrl, roomId]);  
  // ...  
}
```



# Les Hooks

Test et pratique





# Les Hooks

## Démonstration



- Vous allez remplacer le 'content' de votre Card.tsx par un appel à une api en utilisant le hook **useEffect**
- L'api est simulée avec json-server



# Les Hooks

## Les règles

- Appeler les hooks uniquement au niveau supérieur (pas dans des boucles, conditions, ou fonctions imbriquées).
- Appeler les hooks uniquement depuis des composants fonctionnels ou d'autres hooks personnalisés.

Pour **useEffect** :

- Déclarez des fonctions asynchrones à l'intérieur de useEffect
- Gestion des dépendances
- Nettoyage de l'effet
- Gérez les erreurs





# Les Hooks

## useContext

Le hook **useContext** en React est utilisé pour accéder à la valeur d'un Context depuis un composant fonctionnel. Le **Context** fournit un moyen de passer des données à travers l'arbre des composants sans avoir à passer manuellement les props à chaque niveau.

En d'autres termes, **useContext** vous permet de partager des valeurs entre les composants de façon plus directe que le prop drilling.



```
import React from 'react';
const MyContext = React.createContext(defaultValue);
```



```
<MyContext.Provider value={/* une valeur partagée */}
  {/* vos composants ici */}
</MyContext.Provider>
```



```
import React, { useContext } from 'react';
const value = useContext(MyContext);
```





# Les Hooks

Test et pratique





# Les Hooks

## Démonstration



- Changement du theme du composant Header avec un bouton

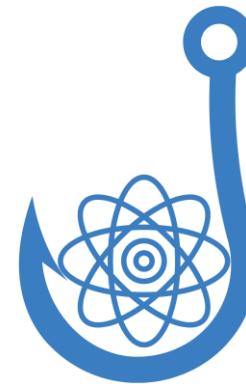


# Les Hooks

## Custom Hooks

Les custom hooks permettent de réutiliser de la logique d'état et de cycle de vie sans avoir besoin de transformer un comportement en un composant de classe ou d'encombrer le composant avec des duplications.

En fait, chaque fois que vous avez du code qui est partagé entre les composants et qui utilise les hooks, envisagez de l'écrire dans un custom hook.





# Les Hooks

Test et pratique





# Les Hooks

## Démonstration



- Création d'un custom hook permettant d'inscrire des valeurs dans le LocalStorage
- Création d'un composant permettant de lire et de mettre en oeuvre ce hook

# Utilisation des Hooks

Informations générales

**useRef :**

Permet la création une référence mutable

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

function Counter() {
  const count = useRef(0);
  const increment = () => {
    count.current += 1;
  };
  return (
    <>
      <p>Count: {count.current}</p>
      <button onClick={increment}>Increment</button>
    </>
  );
}
```

Exemple simple

# Utilisation des Hooks

Informations générales

**useTransition :**

permet aux composants d'éviter des états de chargement indésirables en attendant que le contenu soit chargé avant de transiter vers le prochain écran.



```
const SUSPENSE_CONFIG = { timeoutMs: 2000 };

const [startTransition, isPending] = useTransition(SUSPENSE_CONFIG);
```

Exemple simple

# Utilisation des Hooks

Informations générales

**useCallBack :**

Permet mise en cache d'une fonction



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

export default function ProductPage({ productId, referrer, theme }) {
  const handleSubmit = useCallback((orderDetails) => {
    post('/product/' + productId + '/buy', {
      referrer,
      orderDetails,
    });
  }, [productId, referrer]);
}
```

Exemple simple

# ErrorBoundary

## Informations générales

Un composant **ErrorBoundary** est un composant de React qui peut intercepter les erreurs lancées dans ses composants enfants pendant le rendu, le passage d'argument et la construction.



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

function MyComponent() {
  const [data, setData] = useState([]);
  const [error, setError] = useState(null);

  useEffect(() => {
    fetch('https://jsonplaceholder.typicode.com/todos')
      .then(response => response.json())
      .then(data => {
        setData(data);
      })
      .catch(error => {
        setError(error);
      });
  }, []);

  if (error) {
    throw new Error('Something went wrong.');
  }

  return (
    <div>
      <h2>My Component</h2>
      {data.map(item => <div key={item.id}>{item.title}</div>)}
    </div>
  );
}
```

# ErrorBoundary

Informations générales

**ErrorBoundary :**

Le composant **MyComponent** utilise les **hooks useState et useEffect** pour récupérer la liste des éléments à partir de l'API.



```
function ErrorBoundary({ children }) {
  const [hasError, setHasError] = useState(false);

  const handleCatch = (error, errorInfo) => {
    // Vous pouvez enregistrer l'erreur à des fins d'analyse ultérieure
    console.log('Error caught:', error, errorInfo);
    setHasError(true);
  };

  if (hasError) {
    return <h1>Something went wrong.</h1>;
  }

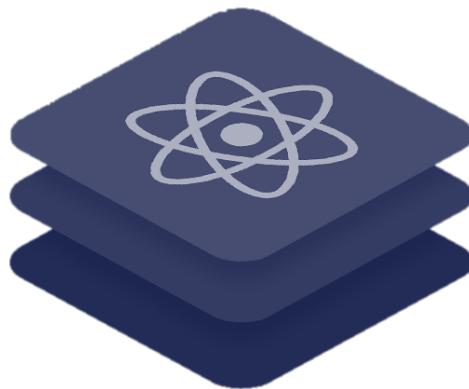
  return (
    <React.Fragment>
      {children}
      <ErrorBoundary handleCatch={handleCatch} />
    </React.Fragment>
  );
}

function App() {
  return (
    <div>
      <ErrorBoundary>
        <MyComponent />
      </ErrorBoundary>
    </div>
  );
}
```

Si une erreur se produit lors de la récupération des éléments, elle lance une erreur avec `throw new Error('Something went wrong.')`.

# Partie 7 :

# Le Routing et la navigation

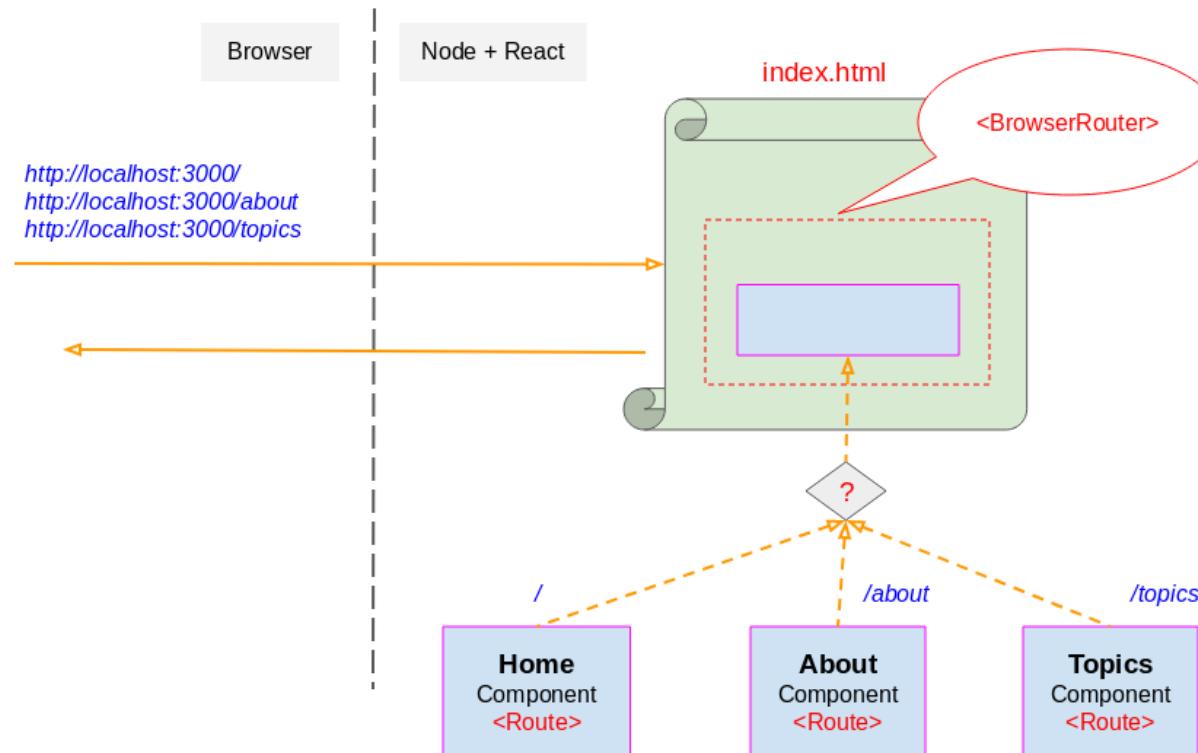




# Le Routing et la navigation

## Définition

Le routing est un aspect essentiel des applications web modernes, en particulier des Single Page Applications (SPA), où au lieu de charger une nouvelle page à chaque action, nous remplaçons le contenu de la page actuelle par du nouveau contenu, ce qui rend l'expérience utilisateur plus fluide et plus rapide.





# Le Routing et la navigation

## Installation

React ne vient pas avec un système de routing intégré, mais il existe des bibliothèques tierces très populaires qui gèrent le routing dans les applications React, la plus connue étant **react-router-dom**.



```
npm install react-router-dom localforage match-sorter sort-by
```



# Le Routing et la navigation

## La mise en place

Le composant `<BrowserRouter>` de React Router v5 est un wrapper qui utilise l'API HTML5 history pour garder votre UI synchronisée avec l'URL.

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

function App() {
  return (
    <BrowserRouter>
      {/* Le reste de votre UI */}
    </BrowserRouter>
  );
}

export default App;
```

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

const Home = () => <h2>Home</h2>;
const About = () => <h2>About</h2>;

const App = () => {
  return (
    <Router>
      <div>
        <nav>
          <ul>
            <li>
              <Link to="/">Home</Link>
            </li>
            <li>
              <Link to="/about">About</Link>
            </li>
          </ul>
        </nav>
        <Switch>
          <Route path="/about">
            <About />
          </Route>
          <Route path="/">
            <Home />
          </Route>
        </Switch>
      </div>
    </Router>
  );
};

export default App;
```



# Le Routing et la navigation

Test et pratique





# Le Routing et la navigation

Plusieurs méthodes de routing

Il existe plusieurs manières de définir des routes.

```
// AppRoutes.tsx

import React from 'react';
import { Routes, Route } from 'react-router-dom';
import HomePage from './HomePage';
import AboutPage from './AboutPage';
import NotFoundPage from './NotFoundPage';

const AppRoutes: React.FC = () => {
  return (
    <Routes>
      <Route path="/" element={<HomePage />} />
      <Route path="/about" element={<AboutPage />} />
      <Route path="*" element={<NotFoundPage />} />
    </Routes>
  );
};

export default AppRoutes;
```



# Le Routing et la navigation

Démonstration



- Transformation de notre sidebar avec le routing



# Le Routing et la navigation

## Élément complémentaire

Il existe le hook `useParams` de react-router-dom est utilisé pour accéder aux paramètres de l'URL dans une route.

```
// App.tsx ou votre composant principal

import React from 'react';
import { BrowserRouter as Router, Route, Switch } from 'react-router-dom';
import UserProfile from './UserProfile';

const App = () => {
  return (
    <Router>
      <Switch>
        <Route path="/user/:userId" component={UserProfile} />
      </Switch>
    </Router>
  );
};

export default App;
```

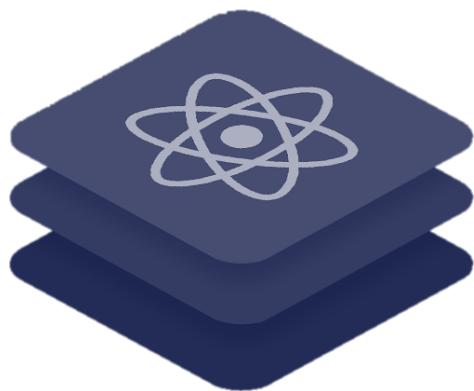
```
const UserProfile = () => {
  // Extraction des paramètres avec le hook useParams
  const { userId } = useParams<RouteParams>();

  return (
    <div>
      <h1>User Profile</h1>
      <p>User ID is: {userId}</p>
    </div>
  );
};
```

```
import { useEffect, useState } from "react";
import { useParams } from "react-router-dom";
import Profil from "../models/Profil";
import profils from "../models/profils";

const Profil = () => {
  const [profil, setProfil] = useState<Profil>();
  const { id } = useParams<{ id: string }>();
  useEffect(() => {
    if (id) {
      heros.forEach((profil) => {
        if (profil.id === +id) {
          setProfil(profil);
        }
      });
    }
  }, [id]);
  return (
    <div className="carte">
      <h2>{profil?.nom}</h2>
      <h3>{profil?.prenom}</h3>
      <div className="carteBody">
        <img src={profil?.image} alt={profil?.prenom} />
      </div>
      <div className="carteprofil">
        <p className="profil">Age : {profil?.age} ans</p>
        <p className="profil">Ville: {profil?.ville}</p>
      </div>
    </div>
  );
}
```

# Partie 8 : Les Formulaires



# Les Formulaires

## Définition

Dans React, les formulaires sont généralement gérés à l'aide de composants contrôlés, où l'état du composant sert de "source de vérité". Cela signifie que l'état de votre composant React (généralement géré par les hooks d'état comme useState) contrôle les données saisies dans les éléments du formulaire.



```
<form onSubmit={handleSubmit}>
  <input type="text" value={inputValue} onChange={handleChange} />
  <button type="submit">Submit</button>
</form>
```

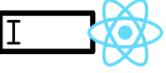
# Les Formulaires

## Contrôlé vs non Contrôlé

Un composant de formulaire contrôlé a sa valeur contrôlée par React. À chaque mise à jour de l'état, le champ de formulaire reflète cette valeur



```
<form>
  <input name="nonContrôlé" placeholder="nonContrôlé" />
  <input name="contrôlé" value="contrôlé" />
</form>
```



# Les Formulaires

## Construction d'un formulaire

Un composant de formulaire contrôlé a sa valeur contrôlée par React. À chaque mise à jour de l'état, le champ de formulaire reflète cette valeur

```
● ● ●

import { useState } from 'react';

interface FormProps {
    // définissez ici les props si nécessaire
}

const Form = (props: FormProps) => {
    const [inputValue, setInputValue] = useState<string>('');

    const handleChange = (event: React.ChangeEvent<HTMLInputElement>) => {
        setInputValue(event.target.value);
    };

    const handleSubmit = (event: React.FormEvent<HTMLFormElement>) => {
        event.preventDefault();
        // Traitement de la soumission du formulaire
        console.log(inputValue);
    };

    return (
        <form onSubmit={handleSubmit}>
            <input type="text" value={inputValue} onChange={handleChange} />
            <button type="submit"> Submit </button>
        </form>
    );
};

export default Form;
```

# Les Formulaires

Test et pratiques



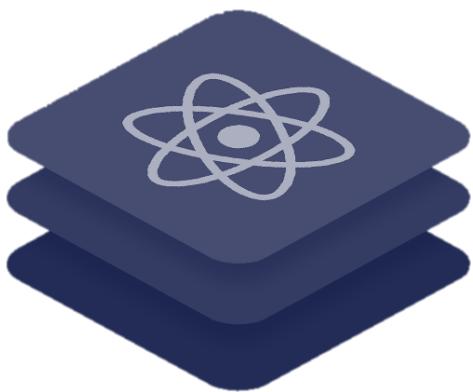
# Les Formulaires

## Démonstration



- Création d'une nouvelle route "formulaire"
- Création d'un composant formulaire qui comprendra
  - nom
  - email
  - description

# Partie 9 : Store



# Store

## Définition

À mesure que les applications grandissent, la gestion de l'état devient complexe.

Il existe plusieurs bibliothèques et architectures pour gérer l'état :



## Redux



## Jōtai

Primitive and flexible state  
management for React



# Store

## Redux

Inspiré par l'architecture Flux de Facebook et le langage de programmation Elm, Redux a été créé par Dan Abramov et Andrew Clark en 2015



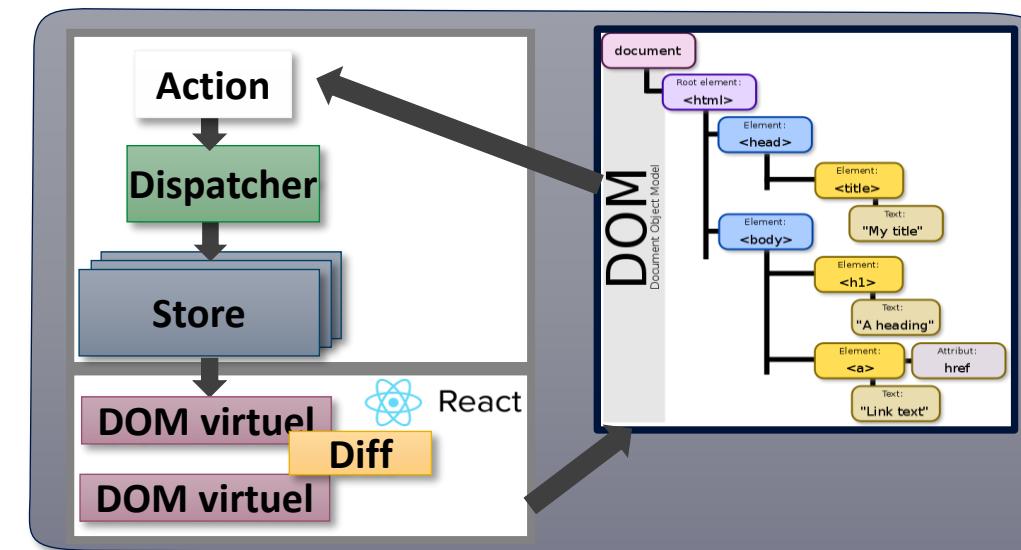
# Redux

# Store

## Architecture Flux

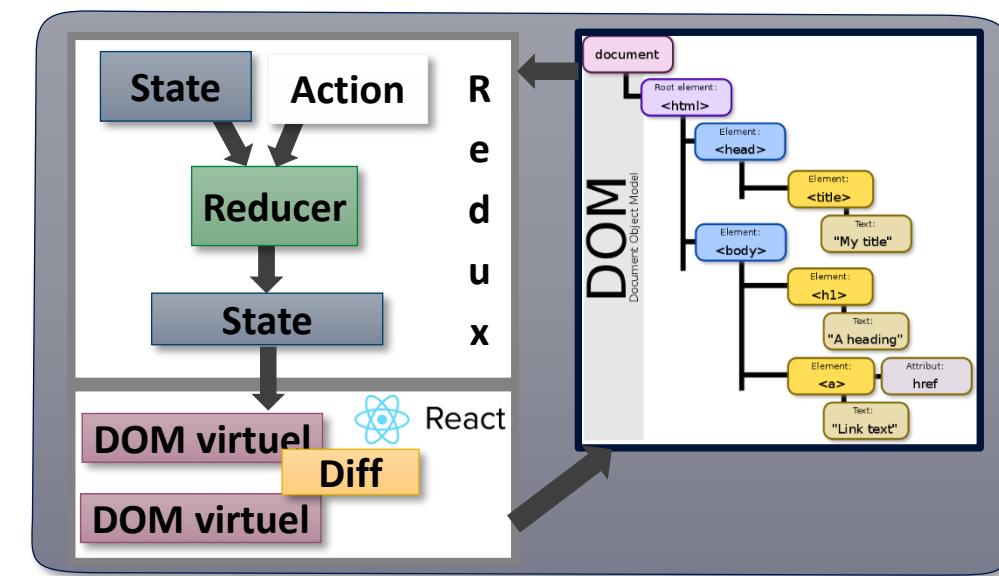
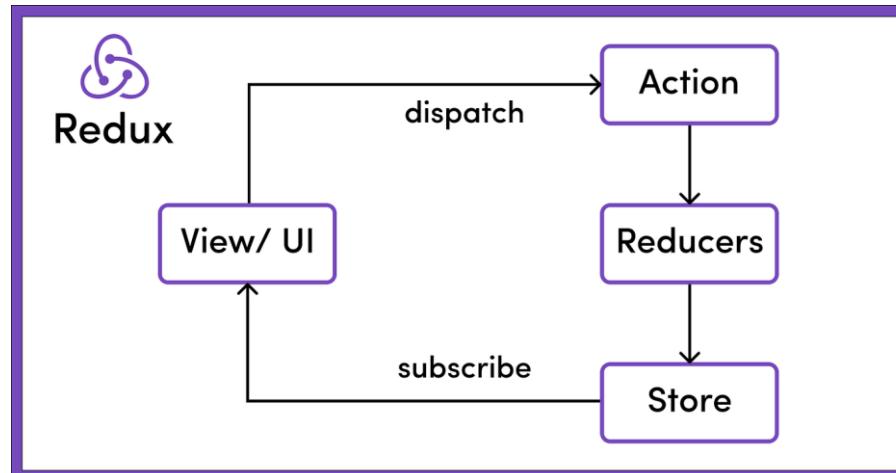
### Qu'est-ce que Flux ?

- **Un pattern de conception d'architecture** : Flux est une architecture qui a été créée par Facebook pour construire des interfaces utilisateur client riches avec React.
- **Unidirectionnel** : Flux promeut un flux de données unidirectionnel, ce qui le rend plus prévisible et facile à comprendre.



# Store

## Redux Architecture



# Store

## Installation



```
add redux react-redux @reduxjs/toolkit
```



# Redux



```
degit reduxjs/redux-templates/packages/vite-template-redux my-app
```

# Store

## Procédé

### Création d'un store



```
// store.ts
import { configureStore } from '@reduxjs/toolkit'

export const store = configureStore({
  reducer: {
    // Reducers ajoutés ici
  },
})
```

# Store

## Procédé

### Création d'un slice

```
● ● ●

// features/counter/counterSlice.ts
import { createSlice, PayloadAction } from '@reduxjs/toolkit'

interface CounterState {
  value: number
}

const initialState: CounterState = {
  value: 0,
}

export const counterSlice = createSlice({
  name: 'counter',
  initialState,
  reducers: {
    incremented: state => {
      state.value += 1
    },
  },
})

export const { incremented, decremented, incrementedByAmount } = counterSlice.actions
```

# Store

## Procédé

Modification de notre store pour ajouter notre reducer



```
// store.ts
import counterReducer from './features/counter/counterSlice'

export const store = configureStore({
  reducer: {
    counter: counterReducer,
  },
})
```

# Store

## Procédé

Mettre à disposition notre store



```
// App.tsx
import { Provider } from 'react-redux'
import { store } from './store'

function App() {
  return (
    <Provider store={store}>
      {/* Composants de l'application */}
    </Provider>
  )
}
```

# Store

## Procédé

Ajout du composant utilisant notre store



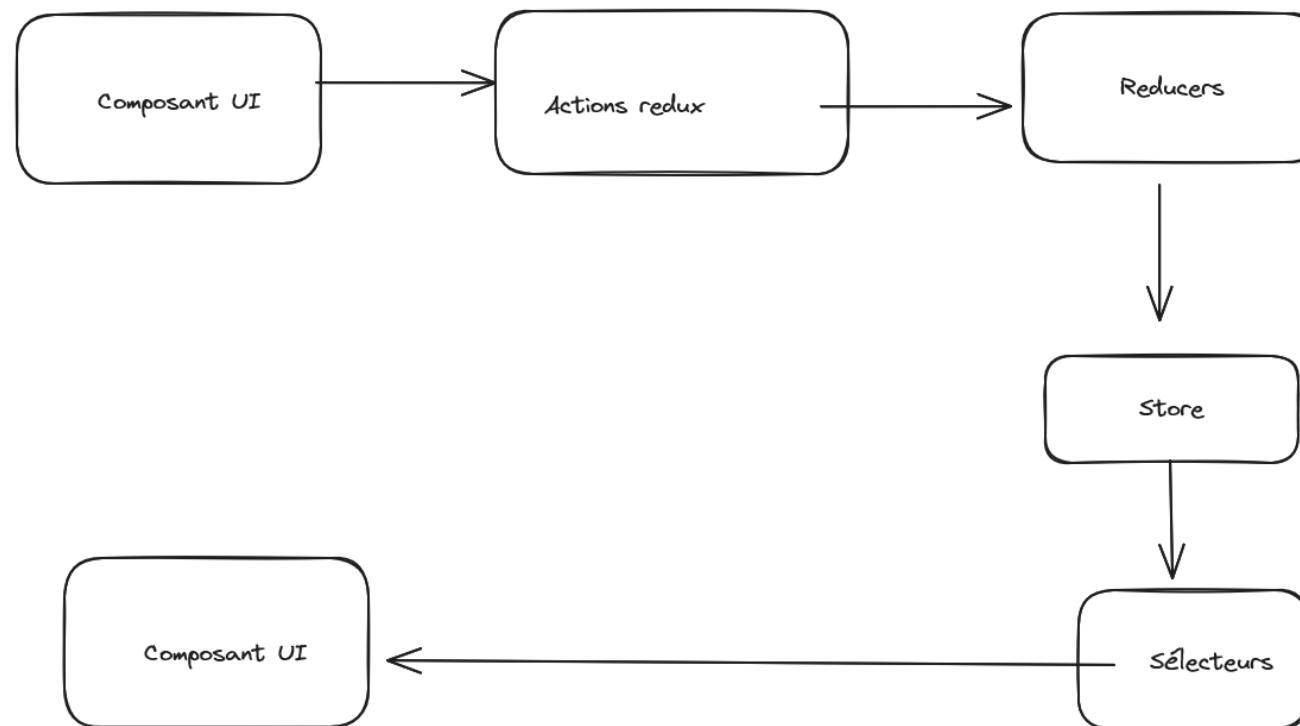
```
import { useAppSelector, useAppDispatch } from '..../hooks'

const CounterComponent = () => {
  const count = useAppSelector(state => state.counter.value)
  const dispatch = useAppDispatch()

  return (
    /* UI du compteur */
  )
}
```

# Store

## Récapitulons

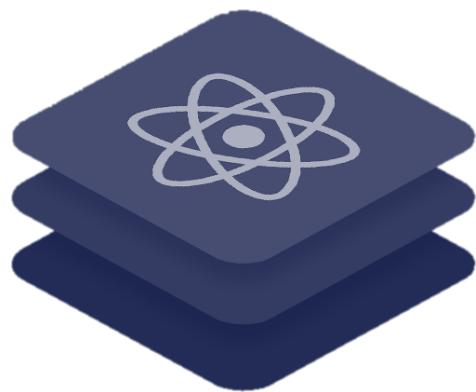


# Store Redux

Test et pratiques



# Partie 10 : Les Tests



# Test des Hooks

## Test à travers un exemple :

Supposons qu'on ait un **hook** personnalisé nommé **useCounter**, qui implémente un compteur simple. Il a deux méthodes : **increment** pour augmenter la valeur du compteur et **reset** pour réinitialiser la valeur du compteur à zéro.



```
import { renderHook, act } from '@testing-library/react-hooks';
import useCounter from './useCounter';

test('useCounter should increment and reset the counter', () => {
  const { result } = renderHook(() => useCounter());
  // Vérifie que le compteur est initialement à zéro
  expect(result.current.count).toBe(0);

  // Incrémente le compteur de 1
  act(() => {
    result.current.increment();
  });

  // Vérifie que le compteur a été incrémenté
  expect(result.current.count).toBe(1);

  // Réinitialise le compteur à zéro
  act(() => {
    result.current.reset();
  });

  // Vérifie que le compteur a été réinitialisé
  expect(result.current.count).toBe(0);
});
```

# Test de composants avec des Hooks

## Test avec React Testing Library

Pour tester des composants qui utilisent des **hooks**, vous pouvez utiliser des bibliothèques de test telles que **React Testing Library** ou **Enzyme**.

**React Testing Library** fournit une série d'utilitaires pour faciliter la création de tests pour les composants React, y compris la simulation d'événements, la recherche d'éléments dans le DOM, l'attente d'états asynchrones et bien plus encore.

Elle permet également de simuler les interactions utilisateur avec les composants en utilisant la méthode **fireEvent**

# Test de composants avec des Hooks

## Test avec React Testing Library

Voici un exemple de test de composant avec des hooks en utilisant **React Testing Library** :



```
import { render, fireEvent } from '@testing-library/react';
import Counter from './Counter';

test('Counter component should render correctly', () => {
  const { getByText } = render(<Counter />);

  // Vérifie que le texte affiché initialement est "Count: 0"
  expect(getByText('Count: 0')).toBeInTheDocument();

  // Clique sur le bouton d'incrémantation
  fireEvent.click(getByText('Increment'));
}
```



```
// Vérifie que le texte affiché après l'incrémantation est "Count: 1"
expect(getByText('Count: 1')).toBeInTheDocument();

// Clique sur le bouton de réinitialisation
fireEvent.click(getByText('Reset'));

// Vérifie que le texte affiché après la réinitialisation est "Count: 0"
expect(getByText('Count: 0')).toBeInTheDocument();
});
```

# Test asynchrone

## Exemple de tests asynchrone

Pour effectuer des tests asynchrones en **React**, vous pouvez utiliser les fonctions asynchrones telles que **async/await** ou les promesses avec **Jest**, qui est une bibliothèque de test couramment utilisée pour les applications React.



```
test('fetchData should return data from API', async () => {
  const data = await fetchData(); // Supposons que fetchData est une fonction
  // qui fait une requête asynchrone à une API

  expect(data).toEqual({ /* Les données attendues de l'API */ });
});
```

# Test asynchrone

## Exemple de tests asynchrone

Voici un exemple de test asynchrone avec Jest et les promesses



```
test('fetchData should return data from API', () => {
  return fetchData().then(data => {
    expect(data).toEqual({ /* Les données attendues de l'API */ });
  });
});
```

# Mock Avancés

## Exemple de mock avec Jest

Jest est un framework de test pour les applications en JavaScript, y compris les applications React.

Jest est très populaire dans la communauté React car il est facile à configurer et à utiliser. Il est également très rapide et peut être utilisé pour effectuer des tests unitaires, des tests d'intégration et des tests de snapshot.



# Mock Avancés

## Exemple de mock avec Jest

Les tests unitaires sont utilisés pour tester des parties spécifiques du code.  
Les tests d'intégration sont utilisés pour tester la façon dont les différentes parties du code fonctionnent ensemble.

Les tests de snapshot sont utilisés pour s'assurer que les changements apportés à une application ne modifient pas l'apparence ou le comportement de l'application de manière inattendue.



# Mock Avancés

## Exemple de mock avec Jest

```
● ● ●

import React from 'react';
import { render, screen } from '@testing-library/react';
import MyComponent from './MyComponent';

describe('MyComponent', () => {
  it('renders the component correctly', () => {
    render(<MyComponent />);

    const component = screen.getByTestId('my-component');
    expect(component).toBeInTheDocument();
  });

  it('displays the correct text', () => {
    const expectedText = 'Hello, world!';
    render(<MyComponent text={expectedText} />);

    const textElement = screen.getByText(expectedText);
    expect(textElement).toBeInTheDocument();
  });
});
```



# Mock Avancés

## Autres outils de test :

Il existe de nombreux outils afin de tester les composants et cela de plusieurs manières.

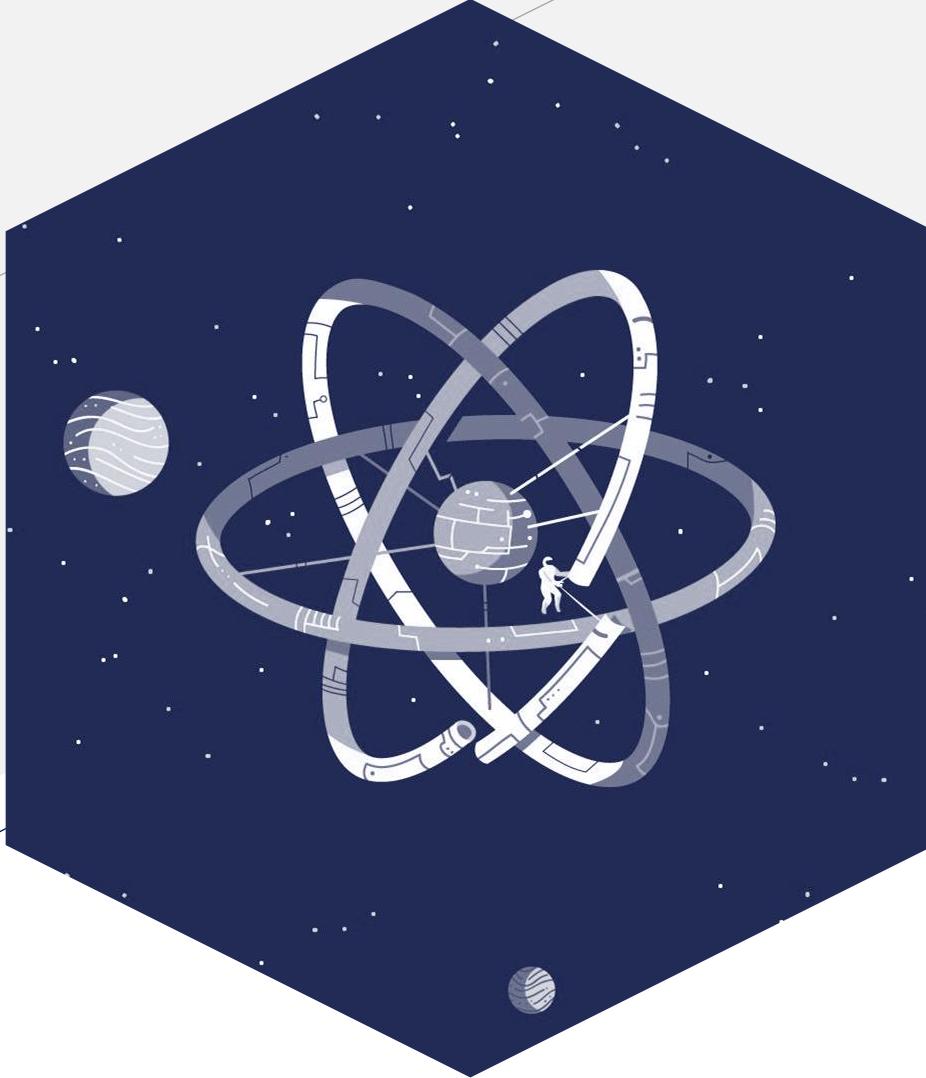


Site officiel : [Link](#)

Documentation : [Link](#)

The screenshot shows the Storybook interface. On the left, the sidebar navigation includes sections like DESIGN SYSTEM, Intro, Avatar, AvatarList, Badge, and others. The 'Badge' section is expanded, and its sub-sections are listed: positive, negative, warning, neutral, error, and with icon. The 'Docs' tab is selected in the top navigation bar, which is highlighted with a red box. The main content area displays the 'Badge' component documentation, titled 'Badge' with the subtitle 'Handy status label'. It explains how to use the `Badge` component to highlight key info with a predefined status. Below this, there's a preview area showing several badge components with different statuses: Positive (green), Negative (red), Neutral (grey), Error (orange), Warning (yellow), and with icon (light green). A 'Show code' button is located at the bottom right of this preview area. At the very bottom, there's a table with columns for Name, Description, and Default, showing the configuration for the 'status' prop.

Name	Description	Default
status	'positive'   'negative'   'neutral'   'error'   'warning'	""neutral""



# Merci.



Gael Baldous



<https://semifir.com>