

# FORMATION REACT.JS

Victor Dupré



# 1. Les fonctions

---

En JavaScript, les fonctions sont des objets de première classe.

Cela signifie qu'elles peuvent être manipulées et échangées, qu'elles peuvent avoir des propriétés et des méthodes, comme tous les autres objets JavaScript. Les fonctions sont des objets Function.

# Valeurs par défaut des arguments

---

```
Z: > Formation > JS formation.js > ...
1  ✓ function multiply(a, b = 1) {
2      |   return a * b;
3      |
4
5  console.log(multiply(5, 2));
6  // expected output: 10
7
8  console.log(multiply(5));
9  // expected output: 5
10 |
```

# Paramètres du reste (Rest parameters)

---

```
Z: > Formation > JS formation.js > multiply
  1  function sum(...args) {
  2    return args.reduce((previous, current) => {
  3      return previous + current;
  4    });
  5  }
  6
  7  console.log(sum(1, 2, 3));
  8  // expected output: 6
  9
 10 console.log(sum(1, 2, 3, 4));
 11 // expected output: 10
 12
```

# Utilisation des fonctions fléchées

---

Objectif : avoir une fonction plus courte.

```
Z: > Formation > JS formation.js > ...
1  const materials = [
2    'Hydrogen',
3    'Helium',
4    'Lithium',
5    'Beryllium'
6  ];
7
8  console.log(materials.map(material => material.length));
9  // expected output: Array [8, 6, 7, 9]
10 |
```

# Fonctions imbriquées

---

```
Z: > Formation > JS formation.js > ...
1  function ajouteCarres(a,b) {
2      function carre(x) {
3          return x * x;
4      }
5      return carre(a) + carre(b);
6  }
7  var a = ajouteCarres(2,3); // renvoie 13
8  var b = ajouteCarres(3,4); // renvoie 25
9  var c = ajouteCarres(4,5); // renvoie 41
```

# Fonctions anonymes

---

```
Z: > Formation > JS formation.js
● 1 ↵ function() {
  2   |   alert("Je suis une fonction anonyme");
  3   |
  4
  5
  6   |
```

# Fonction de Callback

---

Une fonction de callback est une fonction passée dans une autre fonction en tant qu'argument. Elle est ensuite invoquée à l'intérieur de la fonction externe pour accomplir une action.

```
Z: > Formation > JS formation.js > ...
1  function salutation(name) {
2      alert('Bonjour ' + name);
3  }
4
5  function processUserInput(callback) {
6      var name = prompt('Entrez votre nom.');
7      callback(name);
8  }
9
10 processUserInput(salutation);
11
12 |
```

# Closures

---

Une closure est une fonction interne qui va « se souvenir » et pouvoir continuer à accéder à des variables définies dans sa fonction parente même après la fin de l'exécution de celle-ci.

```
Z: > Formation > JS formation.js > ...
1 ~ function compteur() {
2     let count = 0;
3
4 ~     return function() {
5         |     |     return count++;
6     };
7 }
8
9 let plusUn = compteur();
10
11 |
```

# Le scope (la portée)

---

Exemple de variable locale :

```
Z: > Formation > JS formation.js > a
  1  function a() {
  2      var str = "Bonjour";
  3      console.log(str); // "Bonjour"
  4      function b() {
  5          var str = "Le monde";
  6          console.log("In b : ", str); // 'In b : Le monde'
  7      }
  8      b();
  9  }
```

# Le scope (la portée)

---

Exemple de variable globale:

```
Z: > Formation > JS formation.js > ...
1  var str = "Hello world";
2
3  function a() {
4      console.log(str); // "Hello world"
5  }
6
7  |
```

# Le scope (la portée)

---

Version ES6 : Let et Const

```
Z: > Formation > JS formation.js > b
1  // Avant
2  function a() {
3      var str = 'Hello';
4      if (true) {
5          var str = 'World'; // C'est la même variable !
6          console.log(str); // World
7      }
8      console.log(str); // World
9  }
10
11 function b() {
12     let str = 'Hello';
13     if (true) {
14         let str = 'World'; // c'est une variable différente
15         console.log(str); // World
16     }
17     console.log(str); // Hello
18 }
```



## 2. Les objets

---

JavaScript est conçu autour d'un paradigme simple, basé sur les objets. Un objet est un ensemble de propriétés et une propriété est une association entre un nom (aussi appelé clé) et une valeur. La valeur d'une propriété peut être une fonction, auquel cas la propriété peut être appelée « méthode ».

# Création d'un objet

---

```
Z: > Formation > JS formation.js > ...
1  var o = new Object();
```

Un objet vide est stocké dans "o". Ainsi, nous pouvons accéder aux méthodes du constructeur Objet.

Par exemple :

- > Assign
- > Create
- > entries
- > keys

```
Z: > Formation > JS formation.js > ...
1  var o = {};
```



# 3. Les classes

---

Les classes JavaScript ont été introduites avec ECMAScript 2015. Elles fournissent uniquement une syntaxe plus simple pour créer des objets et manipuler l'héritage.

# Définir une classe

---

```
Z: > Formation > JS formation.js > ...
```

```
1 ✓ class Rectangle {  
2 ✓   constructor(hauteur, largeur) {  
3   |     this.hauteur = hauteur;  
4   |     this.largeur = largeur;  
5   |   }  
6   }  
7  
8   |
```

# Les expressions de classes

---

```
Z: > Formation > JS formation.js > ...
1  // anonyme
2  let Rectangle = class {
3      constructor(hauteur, largeur) {
4          this.hauteur = hauteur;
5          this.largeur = largeur;
6      }
7  };
8
9  // nommée
10 let Rectangle = class Rectangle {
11     constructor(hauteur, largeur) {
12         this.hauteur = hauteur;
13         this.largeur = largeur;
14     }
15 };
16
17
```

# Ajout de méthodes

---

```
Z: > Formation > JS formation.js > ...
● 1  ✓ class Rectangle {
  2    ✓   constructor(hauteur, largeur) {
  3      |   this.hauteur = hauteur;
  4      |   this.largeur = largeur;
  5      |
  6
  7    ✓   get area() {
  8      |       return this.calcArea();
  9    }
 10
 11   ✓   calcArea() {
 12     |       return this.largeur * this.hauteur;
 13   }
 14
 15
 16   const carré = new Rectangle(10, 10);
 17
 18   console.log(carré.area);|
```

# Méthode "static"

---

```
Z: > Formation > JS formation.js > ...
1  class Point {
2    constructor(x, y) {
3      this.x = x;
4      this.y = y;
5    }
6
7    static distance(a, b) {
8      const dx = a.x - b.x;
9      const dy = a.y - b.y;
10     return Math.hypot(dx, dy);
11   }
12 }
13
14 const p1 = new Point(5, 5);
15 const p2 = new Point(10, 10);
16
17 console.log(Point.distance(p1, p2));
```

# Héritage

---

```
Z: > Formation > JS formation.js > 🐶 Chien
 1  class Animal {
 2    constructor(nom) {
 3      this.nom = nom;
 4    }
 5
 6    parle() {
 7      console.log(` ${this.nom} fait du bruit.`);
 8    }
 9  }
10
11 class Chien extends Animal {
12   constructor(nom) {
13     super(nom); // appelle le constructeur parent avec le paramètre
14   }
15   parle() {
16     console.log(` ${this.nom} aboie.`);
17   }
18 }
```



## 4. Les collections

---

Les collections incluent les tableaux et les objets semblables à des tableaux que sont les objets `Array` et les objets `TypedArray`.

# Créer un tableau

---

```
Z: > Formation > JS formation.js > ...
1  var arr = new Array(longueurTableau);
2  var arr = Array(longueurTableau);
3
4  // Cela aura le même effet que :
5  var arr = [];
6  arr.length = longueurTableau;
7
8  |
```

# Remplir un tableau

---

```
Z: > Formation > JS formation.js > ...
1  var arr = [];
2  arr[3.4] = "Oranges";
3  console.log(arr.length);           // 0
4  console.log(arr.hasOwnProperty(3.4)); // true
```

# Quelques méthodes de l'objet Tableau

---

## Le forEach

```
Z: > Formation > JS formation.js > ...
1  var tableau = ['premier', 'deuxième', , 'quatrième'];
2
3  // affiche ['premier', 'deuxième', 'quatrième'];
4  ✓ tableau.forEach(function(élément) {
5    console.log(élément);
6  });
7
8  if(tableau[2] === undefined) { console.log('tableau[2] vaut undefined'); } // true
9
10 var tableau = ['premier', 'deuxième', undefined, 'quatrième'];
11
12 // renvoie ['premier', 'deuxième', undefined, 'quatrième'];
13 ✓ tableau.forEach(function(élément) {
14   console.log(élément);
15 })
```

# Quelques méthodes de l'objet Tableau

---

La longueur

```
Z: > Formation > JS formation.js > ...
1  var chats = ['Marie', 'Toulouse', 'Berlioz'];
2  console.log(chats.length); // 3
3
4  chats.length = 2;
5  console.log(chats); // affiche "Marie,Toulouse" - Berlioz a été retiré
6
7  chats.length = 0;
8  console.log(chats); // affiche [], le tableau est vide
9
10 chats.length = 3;
11 console.log(chats); // [ <3 empty slots> ]
```

# Quelques méthodes de l'objet Tableau

---

```
Z: > Formation > JS formation.js > ...
1  var monTableau = new Array("Air", "Feu", "Eau");
2  monTableau.sort(); // trie le tableau [ "Air", "Eau", "Feu" ]
3
4  var monTableau = new Array ("1", "2", "3");
5  monTableau.reverse(); // monTableau vaut maintenant [ "3", "2", "1" ]
6
7  var monTableau = new Array("1", "2", "3");
8  var premier = monTableau.shift(); // monTableau vaut désormais [ "2", "3" ], premier vaut "1"
9
10 var monTableau = new Array("1", "2");
11 monTableau.push("3"); // monTableau vaut désormais [ "1", "2", "3" ]
12
13 var monTableau = new Array("Air", "Eau", "Feu");
14 var list = monTableau.join(" - "); // list sera "Air - Eau - Feu"
15
16 |
```



## 5. Les erreurs

---

La première chose qu'il y a à savoir lorsqu'on souhaite prendre en charge les erreurs est que lorsqu'une erreur d'exécution survient dans un script le JavaScript crée automatiquement un objet à partir de l'objet global Error qui est l'objet de base pour les erreurs en JavaScript.

# Attraper des erreurs

---

```
Z: > Formation > JS formation.js > ...
1  try{
2      prenom
3      alert('Bonjour');
4  }catch(err){
5      alert('Erreur rencontrée. ' +
6          '\nNom de l\'erreur : ' + err.name +
7          '\nMessage d\'erreur : ' + err.message +
8          '\nEmplacement de l\'erreur : ' + err.stack);
9  }
10 alert('Ce message s\'affiche correctement');
11
```



# 6. Les modules

---

Les modules permettent de fournir un mécanisme pour diviser les programmes JavaScript en plusieurs modules qu'on pourrait importer les uns dans les autres. Cette fonctionnalité était présente dans Node.js depuis longtemps et plusieurs bibliothèques et frameworks JavaScript ont permis l'utilisation de modules (CommonJS, AMD, RequireJS ou, plus récemment, Webpack et Babel).

# Exportation de module

---

```
Z: > Formation > JS formation.js >  draw
  1  export const name = 'square';
  2
  3  export function draw(ctx, length, x, y, color) {
  4      ctx.fillStyle = color;
  5      ctx.fillRect(x, y, length, length);
  6
  7  return {
  8      length: length,
  9      x: x,
 10      y: y,
 11      color: color
 12  };
 13 }
```

# Importation de module

---

```
Z: > Formation > JS formation.js
1  import { name, draw, reportArea, reportPerimeter } from './modules/square.mjs';
2
3
4  |
```



## 7. Les évènements

---

Les événements sont des actions ou des occurrences qui se produisent dans le système que vous programmez et dont le système vous informe afin que vous puissiez y répondre d'une manière ou d'une autre si vous le souhaitez.

# Exemple d'évènement

---

```
Z: > Formation > JS formation.js > onclick
1  var btn = document.querySelector('button');
2
3  ↵ function random(number) {
4  |   return Math.floor(Math.random()*(number+1));
5  }
6
7  ↵ btn.onclick = function() {
8  |   var rndCol = 'rgb(' + random(255) + ',' + random(255) + ',' + random(255) + ')';
9  |   document.body.style.backgroundColor = rndCol;
10 }
```

# Exemple d'évènement - Listener

---

```
Z: > Formation > JS formation.js > ...
1  var btn = document.querySelector('button');
2
3  ↘ function bgChange() {
4      var rndCol = 'rgb(' + random(255) + ', ' + random(255) + ', ' + random(255) + ')';
5      document.body.style.backgroundColor = rndCol;
6  }
7
8  btn.addEventListener('click', bgChange);
```



## 8. Les promesses

---

Une promesse est un objet (Promise) qui représente la complétion ou l'échec d'une opération asynchrone. La plupart du temps, on « consomme » des promesses.

# Anciennement :

```
Z: > Formation > JS formation.js > ...
1  function faireQqcALAncienne(successCallback, failureCallback){
2      console.log("C'est fait");
3      // réussir une fois sur deux
4      if (Math.random() > .5) {
5          successCallback("Réussite");
6      } else {
7          failureCallback("Échec");
8      }
9  }
10
11 function successCallback(résultat) {
12     console.log("L'opération a réussi avec le message : " + résultat);
13 }
14
15
16 function failureCallback(erreur) {
17     console.error("L'opération a échoué avec le message : " + erreur);
18 }
19
20 faireQqcALAncienne(successCallback, failureCallback);
```

# Maintenant :

---

```
Z: > Formation > JS formation.js > ...
1  ↵ function faireQqc() {
2  ↵   ↵   return new Promise((successCallback, failureCallback) => {
3  ↵   ↵   ↵   console.log("C'est fait");
4  ↵   ↵   ↵   // réussir une fois sur deux
5  ↵   ↵   ↵   if (Math.random() > .5) {
6  ↵   ↵   ↵   ↵   successCallback("Réussite");
7  ↵   ↵   ↵   } else {
8  ↵   ↵   ↵   ↵   failureCallback("Échec");
9  ↵   ↵   ↵   }
10  ↵   ↵   })
11  ↵   }
12
13  ↵ function successCallback(résultat) {
14  ↵   console.log("L'opération a réussi avec le message : " + résultat);
15  ↵   }
16
17
18  ↵ function failureCallback(erreur) {
19  ↵   console.error("L'opération a échoué avec le message : " + erreur);
20  ↵   }
21
22  faireQqc().then(successCallback, failureCallback);
```



# 9. Fetch

---

L'API Fetch fournit une interface JavaScript pour l'accès et la manipulation des parties de la pipeline HTTP, comme les requêtes et les réponses. Cela fournit aussi une méthode globale `fetch()` qui procure un moyen facile et logique de récupérer des ressources à travers le réseau de manière asynchrone.

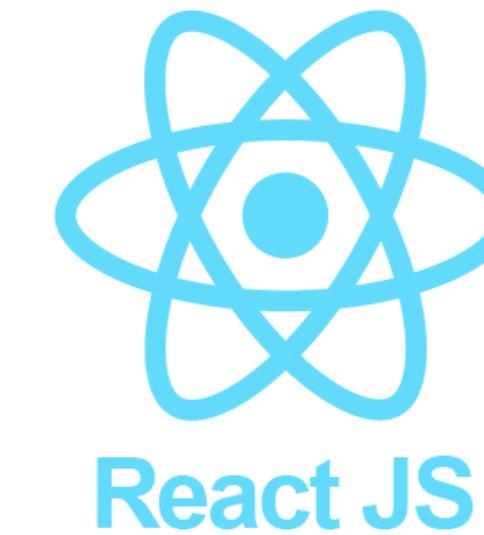
# Exemple :

```
Z: > Formation > JS formation.js > ...
1  var myHeaders = new Headers();
2
3  var myInit = { method: 'GET',
4                headers: myHeaders,
5                mode: 'cors',
6                cache: 'default' };
7
8  var myRequest = new Request('flowers.jpg',myInit);
9
10 fetch(myRequest,myInit)
11   .then(function(response) {
12     return response.blob();
13   })
14   .then(function(myBlob) {
15     var objectURL = URL.createObjectURL(myBlob);
16     myImage.src = objectURL;
17   });

```



# Introduction



ReactJS est un framework frontend open-source créé par Facebook en 2013 pour répondre à des besoins spécifiques dans la création d'interfaces utilisateur complexes et dynamiques. Aujourd'hui, ReactJS est utilisé par de nombreuses entreprises de premier plan telles que Netflix, Airbnb et Instagram.



# Les avantages

La composition de composants : Le concept de la composition de composants est l'un des avantages clés de ReactJS. Il permet aux développeurs de créer des interfaces utilisateur réutilisables en combinant plusieurs petits composants pour créer des composants plus complexes. Cela facilite la maintenance du code, la réutilisation des composants et la création de designs modulaires.

Virtual DOM : Le Virtual DOM est un autre avantage clé de ReactJS. Il permet de créer des interfaces utilisateur hautement réactives et efficaces en minimisant les modifications apportées à la page. Plutôt que de mettre à jour l'ensemble de la page lorsqu'un utilisateur interagit avec l'application, ReactJS met à jour uniquement les parties de l'interface utilisateur qui ont été modifiées, ce qui réduit le temps de traitement et améliore les performances globales de l'application.



# Les avantages

Large communauté de développeurs : ReactJS est soutenu par une grande communauté de développeurs, ce qui signifie qu'il existe de nombreux outils, bibliothèques et modules disponibles pour aider les développeurs à créer des applications de haute qualité. Cette communauté permet également aux développeurs de trouver des solutions aux problèmes et des réponses à leurs questions rapidement et facilement.

Facilité d'utilisation : ReactJS est relativement facile à apprendre et à utiliser, ce qui en fait une technologie populaire auprès des débutants et des experts. Il est également livré avec une grande variété de documentation et de guides pour aider les développeurs à apprendre rapidement et à résoudre les problèmes plus facilement.



# Différences entre framework

---

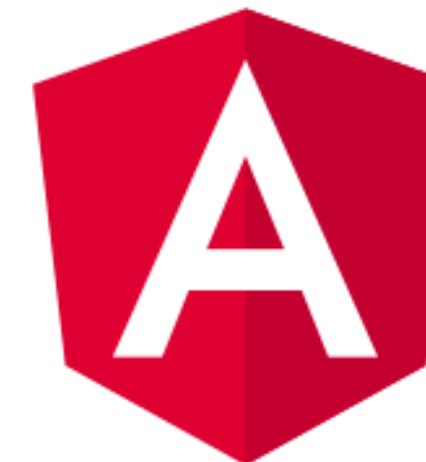


Vue.js : Vue.js est un autre framework frontend qui est souvent comparé à ReactJS. Comme ReactJS, Vue.js utilise une syntaxe similaire à HTML pour créer des composants réutilisables. Cependant, Vue.js offre une courbe d'apprentissage plus douce que ReactJS, avec une syntaxe plus simple et des concepts plus faciles à comprendre.



# Différences entre framework

---



Angular : Angular est un framework frontend développé par Google. Contrairement à ReactJS, Angular est un framework complet qui inclut des fonctionnalités telles que la liaison de données bidirectionnelle, les services, les directives et les modules. Angular est également plus strict dans sa syntaxe et nécessite une courbe d'apprentissage plus raide que ReactJS. En revanche, Angular fournit une expérience de développement plus cohérente et structurée qui facilite la maintenance du code.



# Explication du concept de Virtual DOM

---

Le Virtual DOM est une technique clé utilisée par ReactJS pour améliorer les performances de l'interface utilisateur. Le Virtual DOM est une version virtuelle et légère du DOM (Document Object Model) qui est utilisée pour minimiser les modifications apportées à la page lorsqu'un utilisateur interagit avec l'application.

Le DOM est une représentation en arborescence des éléments HTML de la page web, créée par le navigateur lorsqu'il analyse le code HTML de la page. Le DOM est la structure de données utilisée pour représenter les éléments HTML de la page et il est utilisé pour manipuler dynamiquement les éléments HTML de la page à l'aide de JavaScript.



# Explication du concept de Virtual DOM

---

Lorsqu'un utilisateur interagit avec une page web, le navigateur doit modifier le DOM pour refléter les changements apportés par l'utilisateur. Cependant, la modification du DOM peut être coûteuse en termes de performances, car cela peut nécessiter une mise à jour complète de la page, même si seule une petite partie de la page a été modifiée.

Le Virtual DOM est une solution à ce problème. Au lieu de modifier directement le DOM, ReactJS utilise une version virtuelle de celui-ci. Cette version virtuelle est une représentation légère et rapide du DOM, qui est mise à jour chaque fois que l'état de l'application change.



# Explication du concept de Virtual DOM

---

Lorsqu'un utilisateur interagit avec l'application, ReactJS compare la version virtuelle du DOM avec la version réelle du DOM et identifie les différences entre les deux. Ensuite, il met à jour uniquement les parties du DOM qui ont changé, au lieu de recharger l'ensemble de la page.

Le Virtual DOM permet donc de minimiser les modifications apportées à la page, ce qui améliore les performances globales de l'application. Le Virtual DOM est également efficace pour les applications avec des mises à jour fréquentes, car il permet de réduire la charge de travail du navigateur en minimisant les modifications apportées au DOM.



# Configuration de l'environnement de développement avec Node.js et npm

---

Node.js est une plateforme logicielle qui permet aux développeurs de créer des applications web en utilisant JavaScript du côté serveur. Npm (Node Package Manager) est un gestionnaire de packages pour Node.js qui permet aux développeurs d'installer et de gérer des packages JavaScript tiers.



# Tour d'horizon des outils de développement et d'intégration actuels :

Visual Studio Code : Visual Studio Code est un éditeur de code source gratuit et open source, développé par Microsoft. Il est utilisé pour écrire du code dans de nombreux langages de programmation, y compris JavaScript. Visual Studio Code offre de nombreuses fonctionnalités telles que la coloration syntaxique, l'achèvement de code, la débogage, etc.

Webpack : Webpack est un outil de compilation de modules pour les applications web. Il est utilisé pour regrouper le code JavaScript en modules, de manière à ce qu'il puisse être chargé plus rapidement par le navigateur. Webpack offre également des fonctionnalités telles que l'optimisation des images, la minification du code, la configuration du serveur de développement, etc.



# Tour d'horizon des outils de développement et d'intégration actuels :

Babel : Babel est un outil de transpilation qui permet aux développeurs d'écrire du code JavaScript moderne, qui peut être compilé pour être exécuté dans des navigateurs plus anciens. Babel peut transpiler du code écrit en ES6, ES7, etc. en code ES5 compatible avec les navigateurs plus anciens.

ESLint : ESLint est un outil de linter pour JavaScript. Il est utilisé pour détecter les erreurs de syntaxe et de style dans le code JavaScript. ESLint peut également être configuré pour détecter des erreurs spécifiques, telles que des variables non déclarées ou des fonctions inutilisées.



# Utilisation de la commande "create-react-app"

Installation de Node.js et npm : Avant d'utiliser la commande "create-react-app", vous devez vous assurer que Node.js et npm sont installés sur votre ordinateur. Vous pouvez vérifier que Node.js est installé en utilisant la commande "node -v" et que npm est installé en utilisant la commande "npm -v".

Installation de la commande "create-react-app" : La commande "create-react-app" est disponible via le package "create-react-app" sur npm. Pour l'installer, vous devez exécuter la commande "npm install -g create-react-app" dans votre terminal.



# Utilisation de la commande "create-react-app"

Génération d'une application React : Une fois que la commande "create-react-app" est installée, vous pouvez générer une nouvelle application React en utilisant la commande "create-react-app my-app", où "my-app" est le nom de votre application. Cette commande créera un nouveau dossier "my-app" contenant une application React préconfigurée.

Exécution de l'application : Pour exécuter l'application, vous devez vous rendre dans le dossier "my-app" en utilisant la commande "cd my-app" et exécuter la commande "npm start". Cette commande lancera l'application sur votre navigateur par défaut.



# Le JSX

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

Cette drôle de syntaxe n'est ni une chaîne de caractères ni du HTML.

Ça s'appelle du JSX, et c'est une extension syntaxique de JavaScript.

JSX produit des « éléments » React.

# Quelques exemples

---

```
1 const complete_name = 'Clarisse Agbegenou';
2 const element = <h1>Bonjour, {complete_name}</h1>;
```

```
1 const element = <div tabIndex="0"></div>;
2 const element = <img src={user.avatarUrl}></img>;
```

# Afficher un élément dans le DOM

---

Supposons qu'il y ait une balise <div> quelque part dans votre fichier HTML :

```
1 <div id="root"></div>
```

Nous parlons de nœud DOM « racine » car tout ce qu'il contient sera géré par React DOM.

```
1 const element = <h1>Bonjour, monde</h1>;
2 ReactDOM.render(element, document.getElementById('root'));
```

# Mettre à jour un élément

---

```
1  function tick() {
2    const element = (
3      <div>
4        <h1>Bonjour, monde !</h1>
5        <h2>Il est {new Date().toLocaleTimeString()}</h2>
6      </div>
7    );
8    ReactDOM.render(element, document.getElementById('root'));
9  }
10
11  setInterval(tick, 1000);
```

À chaque seconde, nous appellons ReactDOM.render() depuis une fonction de rappel passée à setInterval().

<https://codepen.io/pen?&editors=0010>



# Les composants

```
1 <function Welcome(props) {  
2   return <h1>Bonjour, {props.name}</h1>;  
3 }
```

```
1 const element = <Welcome name="Sara" />;
```

# Réutilisation des composants

---

```
1  function Welcome(props) {
2      return <h1>Bonjour, {props.name}</h1>;
3  }
4
5  function App() {
6      return (
7          <div>
8              <Welcome name="Sara" />
9              <Welcome name="Cahal" />
10             <Welcome name="Edite" />
11         </div>
12     );
13 }
14
15 ReactDOM.render(
16     <App />,
17     document.getElementById('root')
18 );
```

# Extraire des composants

---

```
1  function Comment(props) {
2    return (
3      <div className="Comment">
4        <div className="UserInfo">
5          <img className="Avatar"
6            src={props.author.avatarUrl}
7            alt={props.author.name}
8          />
9          <div className="UserInfo-name">
10            {props.author.name}
11          </div>
12        </div>
13        <div className="Comment-text">
14          {props.text}
15        </div>
16        <div className="Comment-date">
17          {formatDate(props.date)}
18        </div>
19      </div>
20    );
21  }
```

# Extraire des composants

```
1  ↵ function Avatar(props) {
2    ↵   return (
3    ↵     | <img className="Avatar"
4    ↵       |   src={props.user.avatarUrl}
5    ↵       |   alt={props.user.name}
6    ↵       |   />
7    ↵   );
8  ↵ }
9  ↵ function UserInfo(props) {
10 ↵   return (
11 ↵     | <div className="UserInfo">
12 ↵       |   <Avatar user={props.user} />
13 ↵       |   <div className="UserInfo-name">
14 ↵         |   {props.user.name}
15 ↵       |   </div>
16 ↵     | </div>
17 ↵   );
18
19 ↵ function Comment(props) {
20 ↵   return (
21 ↵     | <div className="Comment">
22 ↵       |   <UserInfo user={props.author} />
23 ↵       |   <div className="Comment-text">
24 ↵         |   {props.text}
25 ↵       |   </div>
26 ↵       |   <div className="Comment-date">
27 ↵         |   {formatDate(props.date)}
28 ↵       |   </div>
29 ↵     | </div>
30 ↵   );
31 }
```



# Les Props dans React.js

---

Les Props sont un moyen fondamental de transmission de données entre les composants dans React. Ils permettent de rendre les composants réutilisables et maintenables.

- Les Props sont des objets qui permettent de passer des données d'un composant parent à un composant enfant.
- Ils sont en lecture seule (readonly), ce qui signifie qu'un composant enfant ne peut pas modifier directement les props reçues.

# Envoyer des props

---

Pour envoyer des props à un composant enfant, ajoutez des attributs personnalisés au composant lors de son utilisation.

Exemple:

```
function Parent() {  
  return <Enfant nom="John Doe" age={30} />;  
}
```

# Accéder aux Props - Composants fonctionnels

---

Dans les composants fonctionnels, les props sont accessibles via le premier argument de la fonction.

```
function Enfant(props) {  
  return (  
    <div>  
      <h1>Nom: {props.nom}</h1>  
      <p>Age: {props.age}</p>  
    </div>  
  );  
}
```

# Accéder aux Props - Composants de classe

---

Dans les composants de classe, les props sont accessibles via `this.props`.

```
class Enfant extends React.Component {
  render() {
    return (
      <div>
        <h1>Nom: {this.props.nom}</h1>
        <p>Age: {this.props.age}</p>
      </div>
    );
  }
}
```

# La Props Children

---

La props children permet de passer du contenu entre les balises ouvrantes et fermantes d'un composant.

```
function Parent() {
  return (
    <Enfant>
      <p>Ceci est un contenu enfant.</p>
    </Enfant>
  );
}

function Enfant(props) {
  return <div>{props.children}</div>;
}
```



# Astuces et bonnes pratiques avec les Props

# Utiliser PropTypes pour la validation des props

---

Pour assurer la qualité et la prévisibilité de votre code, vous pouvez utiliser les PropTypes pour valider les types des props et marquer les props obligatoires.

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

function Enfant(props) {
  // ...
}

Enfant.propTypes = {
  nom: PropTypes.string.isRequired,
  age: PropTypes.number,
};
```

# Déstructurer les props

---

Pour un code plus lisible et maintenable, vous pouvez déstructurer les props dans les composants fonctionnels.

```
function Enfant({ nom, age }) {
  return (
    <div>
      <h1>Nom: {nom}</h1>
      <p>Age: {age}</p>
    </div>
  );
}
```

# Utiliser des valeurs par défaut pour les props

---

Pour éviter les erreurs et les comportements inattendus, définissez des valeurs par défaut pour les props qui ne sont pas obligatoires.

```
Enfant.defaultProps = {  
  age: 18,  
};
```



# Les Hooks dans React.js

Les Hooks sont une nouveauté introduite dans React 16.8 qui permettent d'utiliser l'état et les autres fonctionnalités de React dans les composants fonctionnels, sans avoir à recourir aux composants de classe.

- Les Hooks sont des fonctions qui permettent de "accrocher" l'état et le cycle de vie des composants fonctionnels.
- Ils offrent une approche plus simple et plus élégante pour gérer l'état et les effets de côté dans les composants fonctionnels.

# Hooks vs composants en mode classe

---

- Les Hooks permettent d'éviter la complexité et la verbosité des composants de classe.
- Ils permettent une meilleure séparation des préoccupations en regroupant la logique liée à l'état et aux effets de côté.
- Les Hooks facilitent la réutilisation et le partage de logique entre composants.

# Le hook d'état

---

Le Hook d'état, **useState**, permet d'ajouter un état local à un composant fonctionnel.

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

function Compteur() {
  const [compte, setCompte] = useState(0);

  return (
    <div>
      <p>Compte: {compte}</p>
      <button onClick={() => setCompte(compte + 1)}>Incrémenter</button>
    </div>
  );
}
```

# Hook d'effet et la liste de dépendance

---

Le Hook d'effet, `useEffect`, permet d'effectuer des effets de côté dans les composants fonctionnels, tels que les requêtes HTTP, les manipulations du DOM, etc.

La liste de dépendance est un tableau de dépendances qui déclenche l'effet lorsqu'une dépendance change.

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

function Exemple() {
  const [donnees, setDonnees] = useState([]);

  useEffect(() => {
    fetch('https://api.example.com/data')
      .then((response) => response.json())
      .then((data) => setDonnees(data));
  }, []); // L'effet ne se déclenche qu'au montage du composant

  // ...
}
```

# Les modes du Hook d'effet

---

- Initialisation : L'effet se déclenche au montage du composant.
- Mise à jour : L'effet se déclenche lorsqu'une dépendance change.
- Nettoyage : L'effet peut retourner une fonction de nettoyage pour libérer des ressources lors du démontage du composant.

```
useEffect(() => {
  const timer = setInterval(() => {
    // Mise à jour de l'état ou autre logique
  }, 1000);

  return () => {
    clearInterval(timer); // Nettoyage
  };
}, []);
```

# Règles des hooks

---

Il y a deux règles principales à suivre lors de l'utilisation des Hooks :\*

- N'utilisez les Hooks qu'au niveau le plus haut de votre composant : Évitez de les utiliser dans des boucles, des conditions ou des fonctions imbriquées.
- N'utilisez les Hooks que dans les composants fonctionnels React ou d'autres hooks : Ne les utilisez pas dans des fonctions JavaScript classiques ou dans des classes.

Ces règles garantissent le bon fonctionnement des Hooks et la prévisibilité du code.

# Création d'un custom Hook

---

Pour créer un custom Hook, écrivez une fonction JavaScript dont le nom commence par "use", puis utilisez les Hooks natifs ou d'autres custom Hooks à l'intérieur.

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

function useLocalStorage(key, initialValue) {
  const [value, setValue] = useState(() => {
    const storedValue = localStorage.getItem(key);
    return storedValue ? JSON.parse(storedValue) : initialValue;
  });

  useEffect(() => {
    localStorage.setItem(key, JSON.stringify(value));
  }, [key, value]);

  return [value, setValue];
}
```

# Utiliser un custom Hook

---

Pour utiliser un custom Hook dans un composant fonctionnel, importez-le et appelez-le comme une fonction.

```
import useLocalStorage from './useLocalStorage';

function MonComposant() {
  const [compteur, setCompteur] = useLocalStorage('compteur', 0);

  return (
    <div>
      <p>Compteur: {compteur}</p>
      <button onClick={() => setCompteur(compteur + 1)}>Incrémenter</button>
    </div>
  );
}
```



# Le state en class component

Le state est un objet qui stocke les données d'un composant. Il permet de :

1. Conserver les données entre les rendus (persistance de données)
2. Rendre chaque instance d'un composant unique (singularisation du composant)

# Initialisation

---

Dans les composants de classe, initialisez le state dans le constructeur :

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

# La méthode `setState`

---

`setState` est utilisée pour mettre à jour le state d'un composant.

Elle possède deux formes :

1.Synchrone : passez un objet contenant les modifications du state.

2.Asynchrone : passez une fonction qui reçoit l'état actuel et les props et renvoie un objet avec les modifications du state.

```
// Forme synchrone
this.setState({ compteur: this.state.compteur + 1 });

// Forme asynchrone
this.setState((prevState, props) => {
  return { compteur: prevState.compteur + 1 };
});
```

# **Le cycle de vie du composant**

---

Les composants de classe ont des méthodes spéciales appelées méthodes du cycle de vie. Elles sont exécutées à différentes étapes de la vie d'un composant :

1. Montage
2. Mise à jour
3. Démontage

# Montage du composant (componentDidMount)

---

componentDidMount est appelée après le rendu initial du composant. Utilisez cette méthode pour effectuer des actions telles que la récupération de données ou l'initialisation de listeners d'événements.

```
class MonComposant extends React.Component {  
  componentDidMount() {  
    console.log('Le composant a été monté');  
  }  
}
```

# Mise à jour du composant (componentDidUpdate)

---

componentDidUpdate est appelée après chaque mise à jour du composant. Utilisez cette méthode pour réagir aux changements de state ou de props.

```
class MonComposant extends React.Component {  
  componentDidUpdate(prevProps, prevState) {  
    console.log('Le composant a été mis à jour');  
  }  
}
```

# Démontage du composant (componentWillUnmount)

---

componentWillUnmount est appelée juste avant que le composant soit démonté et détruit. Utilisez cette méthode pour nettoyer les ressources utilisées par le composant (timers, listeners d'événements, etc.).

```
class MonComposant extends React.Component {  
  componentWillMount() {  
    console.log('Le composant va être démonté');  
  }  
}
```

# Bonnes pratiques

---

1. Utilisez toujours la forme asynchrone de setState lorsque le nouvel état dépend de l'état actuel ou des props.
2. Ne pas utiliser setState dans le constructeur. Utilisez plutôt le state initial directement.
3. Utilisez les hooks (useState, useEffect) dans les composants fonctionnels pour gérer le state et les effets de bord de manière plus simple et concise.



# Les événements dans React.js

# Syntaxe des événements dans le JSX

---

La syntaxe des événements dans le JSX est similaire à celle du HTML, mais avec quelques différences :

1. Les noms des événements sont en camelCase (onClick, onMouseDown, etc.).
2. Les gestionnaires d'événements sont des fonctions, pas des chaînes de caractères.

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

# Méthodes de gestion d'événement (handler)

---

Un gestionnaire d'événement (handler) est une fonction qui réagit à un événement spécifique.

```
class MonComposant extends React.Component {
  handleClick() {
    console.log('Le bouton a été cliqué');
  }

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

# Liaison du contexte d'exécution au handler

Pour accéder à `this` dans un gestionnaire d'événement, vous devez lier le contexte d'exécution. Il existe plusieurs méthodes pour cela :

1. Utilisez `bind()` dans le constructeur :

```
constructor(props) {
  super(props);
  this.handleClick = this.handleClick.bind(this);
}
```

2. Utilisez une fonction fléchée pour définir le gestionnaire d'événement :

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

3. Utilisez une fonction fléchée lors de la déclaration de l'événement dans le JSX :

```
<button onClick={() => this.handleClick()}>Cliquez ici</button>
```

# Objet d'événement

---

React crée un objet d'événement synthétique qui encapsule l'événement natif du navigateur. Cet objet est passé en tant que premier argument à votre gestionnaire d'événement.

```
handleClick(event) {  
  console.log('L\'événement est :', event);  
}
```

# Passage de paramètres supplémentaires au handler

---

Pour passer des paramètres supplémentaires à votre gestionnaire d'événement, utilisez une fonction fléchée lors de la déclaration de l'événement dans le JSX.

```
<button onClick={(event) => this.handleClick(event, 'paramètre supplémentaire')}>C
```

# Envoyer un handler en props

---

Vous pouvez également passer un gestionnaire d'événement en tant que prop à un autre composant.

```
// Composant parent
class Parent extends React.Component {
  handleClick() {
    console.log('Le bouton du composant enfant a été cliqué');
  }

  render() {
    return <Enfant onEnfantClick={this.handleClick} />;
  }
}

// Composant enfant
function Enfant(props) {
  return <button onClick={props.onEnfantClick}>Cliquez ici</button>;
}
```

# Envoyer un handler en props

---

Vous pouvez également passer un gestionnaire d'événement en tant que prop à un autre composant.

```
import React from 'react';

// Composant parent
function Parent() {
  const handleClick = () => {
    console.log('Le bouton du composant enfant a été cliqué');
  };

  return <Enfant onEnfantClick={handleClick} />;
}

// Composant enfant
function Enfant(props) {
  return <button onClick={props.onEnfantClick}>Cliquez ici</button>;
}
```



# Rendu conditionnel et listes dans React.js

# Opérateur ET (&&) :

```
function MonComposant({ condition }) {  
  return <div>{condition && <p>Condition remplie</p>}</div>;  
}
```

# Opérateur ternaire :

```
function MonComposant({ condition }) {  
  return <div>{condition ? <p>Condition remplie</p> : <p>Condition non remplie</p>}</div>;  
}
```

# Listes

```
function ListeDeNombres({ nombres }) {
  return (
    <ul>
      {nombres.map((nombre) => (
        <li key={nombre}>{nombre}</li>
      )));
    </ul>
  );
}
```

# Les clés (key) et le DOM Virtuel

---

Lorsque vous créez une liste, vous devez fournir une clé unique (key) pour chaque élément. React utilise ces clés pour identifier quels éléments ont été modifiés, ajoutés ou supprimés lors des mises à jour du DOM.

```
function ListeDeNombres({ nombres }) {
  return (
    <ul>
      {nombres.map((nombre) => (
        <li key={nombre.toString()}>{nombre}</li>
      ))}
    </ul>
  );
}
```

# Les fragments

---

Les fragments vous permettent de regrouper plusieurs éléments sans ajouter de nœuds supplémentaires au DOM.

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

function MonComposant() {
  return (
    <Fragment>
      <h1>Titre</h1>
      <p>Paragraphe</p>
    </Fragment>
  );
}

// Ou avec la syntaxe courte :
function MonComposant() {
  return (
    <>
      <h1>Titre</h1>
      <p>Paragraphe</p>
    </>
  );
}
```



# Les formulaires dans React.js

# Exemple de formulaire basique

---

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

function Formulaire() {
  const [nom, setNom] = useState('');

  const handleChange = (event) => {
    setNom(event.target.value);
  };

  return (
    <form>
      <label>
        Nom :
        <input type="text" value={nom} onChange={handleChange} />
      </label>
    </form>
  );
}
```

# Soumission de formulaire

---

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

function Formulaire() {
  const [nom, setNom] = useState('');

  const handleChange = (event) => {
    setNom(event.target.value);
  };

  const handleSubmit = (event) => {
    event.preventDefault();
    console.log('Le nom soumis est:', nom);
  };

  return (
    <form onSubmit={handleSubmit}>
      <label>
        Nom :
        <input type="text" value={nom} onChange={handleChange} />
      </label>
      <button type="submit">Soumettre</button>
    </form>
  );
}
```

# Etat non contrôlé

---

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

function Formulaire() {
  const inputRef = useRef();

  const handleSubmit = (event) => {
    event.preventDefault();
    console.log('Le fichier sélectionné est:', inputRef.current.files[0]);
  };

  return (
    <form onSubmit={handleSubmit}>
      <label>
        Fichier :
        <input type="file" ref={inputRef} />
      </label>
      <button type="submit">Soumettre</button>
    </form>
  );
}
```



# Formik

# Utilisation de Formik



---

Pour utiliser Formik,  
importez le composant  
Formik, les composants  
Form et Field et créez un  
composant de formulaire.

```
import { Formik, Form, Field } from 'formik';

function Formulaire() {
  const initialValues = {
    nom: '',
  };

  const handleSubmit = (values) => {
    console.log('Les valeurs soumises sont:', values);
  };

  return (
    <Formik initialValues={initialValues} onSubmit={handleSubmit}>
      {() => (
        <Form>
          <label>
            Nom :
            <Field type="text" name="nom" />
          </label>
          <button type="submit">Soumettre</button>
        </Form>
      )}
    </Formik>
  );
}
```

# Validation Formik

```
import { Formik, Form, Field, ErrorMessage } from 'formik';

function Formulaire() {
  const initialValues = {
    nom: '',
  };

  const handleSubmit = (values) => {
    console.log('Les valeurs soumises sont:', values);
  };

  const validate = (values) => {
    const errors = {};
    if (!values.nom) {
      errors.nom = 'Le nom est requis';
    }
    return errors;
  };

  return (
    <Formik initialValues={initialValues} onSubmit={handleSubmit} validate={validate}>
      {() => (
        <Form>
          <label>
            Nom :
            <Field type="text" name="nom" />
            <ErrorMessage name="nom" />
          </label>
          <button type="submit">Soumettre</button>
        </Form>
      )}
    </Formik>
  );
}
```

# Validation Yup

```
import { Formik, Form, Field, ErrorMessage } from 'formik';
import * as Yup from 'yup';

function Formulaire() {
  const initialValues = {
    nom: '',
  };

  const handleSubmit = (values) => {
    console.log('Les valeurs soumises sont:', values);
  };

  const validationSchema = Yup.object({
    nom: Yup.string().required('Le nom est requis'),
  });

  return (
    <Formik initialValues={initialValues} onSubmit={handleSubmit} validationSchema={validationSchema}>
      {() => (
        <Form>
          <label>
            Nom :
            <Field type="text" name="nom" />
            <ErrorMessage name="nom" />
          </label>
          <button type="submit">Soumettre</button>
        </Form>
      )}
    </Formik>
  );
}
```

# Avec des custom fields

```
import { Formik, Form, Field } from 'formik';
import * as Yup from 'yup';

function CustomInput({ field, ...props }) {
  return <input {...field} {...props} />;
}

function Formulaire() {
  const initialValues = {
    nom: '',
  };

  const handleSubmit = (values) => {
    console.log('Les valeurs soumises sont:', values);
  };

  const validationSchema = Yup.object({
    nom: Yup.string().required('Le nom est requis'),
  });

  return (
    <Formik initialValues={initialValues} onSubmit={handleSubmit} validationSchema={validationSchema}
      {() => (
        <Form>
        <label>
          Nom :
          <Field name="nom" component={CustomInput} />
        </label>
        <button type="submit">Soumettre</button>
      </Form>
    )}
  );
}
```



# React Router

**npm install react-router-dom**

# Initialisation

---

Nous allons utiliser le BrowserRouter pour notre application. On crée le router avec la méthode createBrowserRouter, qui accepte en paramètre notre configuration. On utilise également le composant RouterProvider à la racine de l'application.

```
1 import React from "react";
2 import ReactDOM from "react-dom/client";
3 import { createBrowserRouter, RouterProvider } from "react-router-dom";
4 import "./index.css";
5 import App from "./App.jsx";
6
7 const router = createBrowserRouter([
8   {
9     path: "/",
10    element: <App/>,
11  },
12]);
13
14 ReactDOM.createRoot(document.getElementById("root")).render(
15  <React.StrictMode>
16    <RouterProvider router={router} />
17  </React.StrictMode>
18);
19
```

# Les liens

---

Utilisez le composant Link pour créer des liens de navigation dans l'application.

```
1 import { Link } from "react-router-dom";
2
3 function UsersIndexPage({ users }) {
4   return (
5     <div>
6       <h1>Users</h1>
7       <ul>
8         {users.map((user) => (
9           <li key={user.id}>
10             <Link to={user.id}>{user.name}</Link>
11           </li>
12         )));
13       </ul>
14     </div>
15   );
16 }
17
18 export default UsersIndexPage;
19
```

# Les routes imbriquées

Utilisez l'array `children` sur une route pour lui ajouter des routes imbriquées, étendant son url de base.

```
1  const router = createBrowserRouter([
2    {
3      path: "/",
4      element: <Root />,
5      errorElement: <ErrorPage />,
6      children: [
7        {
8          path: "contacts/:contactId",
9          element: <Contact />,
10         },
11       ],
12     },
13   ]);
```

# Les navigations imbriquées

---

- Utilisez les routes imbriquées (via les routes children) pour gérer la navigation dans les sous- composants et créer des layouts réutilisables

```
1 import { Outlet } from "react-router-dom";
2
3 export default function Root() {
4   return (
5     <>
6       {/* all the other elements */}
7       <div id="detail">
8         <Outlet />
9       </div>
10     </>
11   );
12 }
13
```

# Les paramètres d'URL

---

Utilisez les paramètres d'URL pour passer des données à travers les routes.

```
1 import * as React from "react";
2 import { useParams } from "react-router-dom";
3
4 function ProfilePage() {
5   // Get the userId param from the URL.
6   let { userId } = useParams();
7   // ...
8 }
9
```



# Redux

---

Redux est une bibliothèque de gestion d'état pour les applications JavaScript. Il est souvent utilisé avec des frameworks tels que React et Angular pour gérer l'état de l'application de manière centralisée. Redux permet de stocker l'état de l'application dans un store global, qui peut être consulté et modifié par des actions déclenchées par l'utilisateur ou par l'application elle-même.



# Installation

---

`npm install redux`



# Création d'actions

---

La première étape consiste à créer les actions nécessaires pour gérer les articles. Les actions sont des objets JavaScript qui décrivent les modifications à apporter au store Redux. Voici un exemple d'actions pour gérer les articles :

```
export const ADD_POST = 'ADD_POST';
export const DELETE_POST = 'DELETE_POST';

export function addPost(title, content) {
  return {
    type: ADD_POST,
    payload: { title, content },
  };
}

export function deletePost(id) {
  return {
    type: DELETE_POST,
    payload: { id },
  };
}
```



# Création du reducer

Une fois les actions créées, la prochaine étape consiste à créer un reducer pour gérer les articles. Le reducer est une fonction pure qui prend en entrée l'état actuel du store Redux et une action, et renvoie un nouvel état qui reflète les modifications apportées par l'action.

Voici un exemple de reducer pour gérer les articles :

```
import { ADD_POST, DELETE_POST } from './actions';

const initialState = {
  posts: [],
};

function postsReducer(state = initialState, action) {
  switch (action.type) {
    case ADD_POST:
      return {
        ...state,
        posts: [...state.posts, action.payload],
      };
    case DELETE_POST:
      return {
        ...state,
        posts: state.posts.filter(post => post.id !== action.payload.id),
      };
    default:
      return state;
  }
}

export default postsReducer;
```



# Création du store

---

La dernière étape consiste à créer le store Redux lui-même. Le store est un objet qui contient l'état de l'application et les méthodes nécessaires pour le modifier. Voici un exemple de création du store "posts" :

```
import { createStore } from 'redux';
import postsReducer from './postsReducer';

const store = createStore(postsReducer);

export default store;
```



# Création d'un Provider

```
import React from 'react';
import { Provider } from 'react-redux';
import { BrowserRouter as Router, Route, Switch } from 'react-router-dom';
import store from './store';
import Home from './Home';
import PostList from './PostList';
import PostDetails from './PostDetails';

function App() {
  return (
    <Provider store={store}>
      <Router>
        <Switch>
          <Route exact path="/" component={Home} />
          <Route exact path="/posts" component={PostList} />
          <Route exact path="/posts/:id" component={PostDetails} />
        </Switch>
      </Router>
    </Provider>
  );
}

export default App;
```



# Comment utiliser Redux avec des composants fonctionnels ?

---

Redux peut être utilisé avec des composants fonctionnels en utilisant la bibliothèque react-redux. react-redux fournit des hooks qui permettent aux composants fonctionnels de se connecter au store Redux. Voici les principaux hooks que vous utiliserez pour connecter des composants fonctionnels au store Redux :



# Exemple d'utilisation de useSelector

Dans cet exemple, nous utilisons useSelector pour accéder à la liste des articles (posts) stockés dans le store Redux et les afficher dans un composant PostList.

```
import React from 'react';
import { useSelector } from 'react-redux';

function PostList() {
  const posts = useSelector(state => state.posts);

  return (
    <div>
      {posts.map(post => (
        <div key={post.id}>
          <h2>{post.title}</h2>
          <p>{post.content}</p>
        </div>
      ))}
    </div>
  );
}

export default PostList;
```

# Exemple d'utilisation de useDispatch



```
import React, { useState } from 'react';
import { useDispatch } from 'react-redux';
import { addPost } from './actions';

function AddPostForm() {
  const dispatch = useDispatch();
  const [title, setTitle] = useState('');
  const [content, setContent] = useState('');

  const handleSubmit = e => {
    e.preventDefault();
    dispatch(addPost(title, content));
    setTitle('');
    setContent('');
  };

  return (
    <form onSubmit={handleSubmit}>
      <label>
        Titre :
        <input type="text" value={title} onChange={e => setTitle(e.target.value)} />
      </label>
      <label>
        Contenu :
        <textarea value={content} onChange={e => setContent(e.target.value)} />
      </label>
      <button type="submit">Ajouter</button>
    </form>
  );
}

export default AddPostForm;
```

Dans cet exemple, nous utilisons `useDispatch` pour déclencher une action qui ajoute un nouvel article (post) dans le store Redux. Le composant `AddPostForm` permet aux utilisateurs de saisir un titre et un contenu pour un nouvel article, puis de cliquer sur le bouton "Ajouter" pour déclencher l'action.



# Plusieurs reducers ?

```
import React from 'react';
import { Provider } from 'react-redux';
import { BrowserRouter as Router, Route, Switch } from 'react-router-dom';
import { createStore, combineReducers } from 'redux';
import postsReducer from './postsReducer';
import usersReducer from './usersReducer';
import Home from './Home';
import PostList from './PostList';
import PostDetails from './PostDetails';

const rootReducer = combineReducers({
  posts: postsReducer,
  users: usersReducer,
});

const store = createStore(rootReducer);

function App() {
  return (
    <Provider store={store}>
      <Router>
        <Switch>
          <Route exact path="/" component={Home} />
          <Route exact path="/posts" component={PostList} />
          <Route exact path="/posts/:id" component={PostDetails} />
        </Switch>
      </Router>
    </Provider>
  );
}

export default App;
```

```
1 import React, { useState } from "react"
2 import Article from "../components/Article/Article"
3 import AddArticle from "../components/AddArticle/AddArticle"
4
5 const Articles = () => {
6   const [articles, setArticles] = useState([
7     { id: 1, title: "post 1", body: "Quisque cursus, metus vitae pharetra" },
8     { id: 2, title: "post 2", body: "Quisque cursus, metus vitae pharetra" },
9   ])
10  const saveArticle = e => {
11    e.preventDefault()
12    // the logic will be updated later
13  }
14
15  return (
16    <div>
17      <AddArticle saveArticle={saveArticle} />
18      {articles.map(article => (
19        <Article key={article.id} article={article} />
20      ))}
21    </div>
22  )
23}
24
25 export default Articles
26
```

# Reducer.js

---

```
1  const initialState = {  
2    articles: [  
3      { id: 1, title: "post 1", body: "Quisque cursus, metus vitae pharetra" },  
4      { id: 2, title: "post 2", body: "Quisque cursus, metus vitae pharetra" },  
5    ],  
6  }  
7  
8  const reducer = (state = initialState, action) => {  
9    return state  
10  }  
11  export default reducer
```

# App.js

---

```
1 import React from "react"
2 import ReactDOM from "react-dom"
3 import { createStore } from "redux"
4 import { Provider } from "react-redux"
5
6 import "./index.css"
7 import App from "./App"
8 import reducer from "./store/reducer"
9
10 const store = createStore(reducer)
11
12 ReactDOM.render(
13   <Provider store={store}>
14     <App />
15   </Provider>,
16   document.getElementById("root")
17 )
```

# Injection de la donnée

```
1  import React from "react"
2  import { connect } from "react-redux"
3
4  import Article from "../components/Article/Article"
5  import AddArticle from "../components/AddArticle/AddArticle"
6
7  const Articles = ({ articles }) => {
8    const saveArticle = e => {
9      e.preventDefault()
10     // the logic will be updated later
11   }
12   return (
13     <div>
14       <AddArticle saveArticle={saveArticle} />
15       {articles.map(article => (
16         <Article key={article.id} article={article} />
17       ))}
18     </div>
19   )
20 }
21
22 const mapStateToProps = state => {
23   return {
24     articles: state.articles,
25   }
26 }
27
28 export default connect(mapStateToProps)(Articles)
```

# Création d'une action

```
1 import React from "react"
2 import { connect } from "react-redux"
3
4 import Article from "../components/Article/Article"
5 import AddArticle from "../components/AddArticle/AddArticle"
6 import * as actionTypes from "../store/actionTypes"
7
8 const Articles = ({ articles, saveArticle }) => (
9   <div>
10    <AddArticle saveArticle={saveArticle} />
11    {articles.map(article => (
12      <Article key={article.id} article={article} />
13    ))}
14  </div>
15)
16
17 const mapStateToProps = state => {
18  return {
19    articles: state.articles,
20  }
21}
22
23 const mapDispatchToProps = dispatch => {
24  return {
25    saveArticle: article =>
26      dispatch({ type: actionTypes.ADD_ARTICLE, articleData: { article } }),
27  }
28}
29
30 export default connect(mapStateToProps, mapDispatchToProps)(Articles)
```

# Reducer.js

---

```
1 import * as actionTypes from "./actionTypes"
2
3 const initialState = {
4   articles: [
5     { id: 1, title: "post 1", body: "Quisque cursus, metus vitae pharetra" },
6     { id: 2, title: "post 2", body: "Quisque cursus, metus vitae pharetra" },
7   ],
8 }
9
10 const reducer = (state = initialState, action) => {
11   switch (action.type) {
12     case actionTypes.ADD_ARTICLE:
13       const newArticle = {
14         id: Math.random(), // not really unique but it's just an example
15         title: action.article.title,
16         body: action.article.body,
17       }
18       return {
19         ...state,
20         articles: state.articles.concat(newArticle),
21       }
22     }
23   return state
24 }
25 export default reducer
```



# Import de middleware

npm install redux-thunk

```
import { createStore, applyMiddleware } from 'redux';
import thunk from 'redux-thunk';
import rootReducer from './reducers';

const store = createStore(rootReducer, applyMiddleware(thunk));

export default store;
```



# Utilisation dans une action

Dans cet exemple, nous avons utilisé la fonction `applyMiddleware` fournie par Redux pour appliquer `thunk` en tant que middleware à notre store Redux. `thunk` est un middleware qui permet de retourner des fonctions à la place des objets d'action. Ces fonctions peuvent effectuer des opérations asynchrones, telles que des appels à une API, avant de déclencher une action.

Pour utiliser Redux Thunk, nous devons également modifier nos actions pour qu'elles retournent des fonctions plutôt que des objets d'action. Voici un exemple d'une action qui récupère la liste des articles à partir d'une API et la stocke dans le store Redux :

```
export function fetchPosts() {
  return async dispatch => {
    try {
      const response = await fetch('https://my-api.com/posts');
      const posts = await response.json();
      dispatch({ type: FETCH_POSTS_SUCCESS, payload: posts });
    } catch (error) {
      dispatch({ type: FETCH_POSTS_FAILURE, payload: error.message });
    }
  };
}
```



# Intégration de Redux Persist

---

Redux Persist est une bibliothèque qui permet de persister l'état du store Redux dans le stockage local (localStorage ou AsyncStorage) du navigateur ou du dispositif mobile. Cela permet de conserver les données de l'application même après une actualisation ou une fermeture de l'application.

`npm install redux-persist`



# Configuration

Ensuite, nous devons configurer Redux Persist en créant un persisteur et en le combinant avec notre réducteur racine. Voici un exemple de configuration de Redux Persist :

```
import { createStore } from 'redux';
import { persistStore, persistReducer } from 'redux-persist';
import storage from 'redux-persist/lib/storage';
import rootReducer from './reducers';

const persistConfig = {
  key: 'root',
  storage,
};

const persistedReducer = persistReducer(persistConfig, rootReducer);

export const store = createStore(persistedReducer);
export const persistor = persistStore(store);
```



# Ajouter le persisteur à notre application

Enfin, nous devons ajouter le persisteur à notre application en enveloppant notre composant Provider avec un composant PersistGate fourni par Redux Persist. Voici un exemple de configuration de PersistGate :

```
import React from 'react';
import { Provider } from 'react-redux';
import { PersistGate } from 'redux-persist/integration/react';
import { store, persistor } from './store';
import App from './App';

function Root() {
  return (
    <Provider store={store}>
      <PersistGate loading={null} persistor={persistor}>
        <App />
      </PersistGate>
    </Provider>
  );
}

export default Root;
```



# Gestion des erreurs avec les "Error Boundaries"

---

Les "Error Boundaries" sont une fonctionnalité de React qui permet de capturer les erreurs qui se produisent dans les composants enfants lors de la phase de rendu. Les "Error Boundaries" peuvent être utilisés pour empêcher l'application de planter en cas d'erreur et pour afficher un message d'erreur convivial à l'utilisateur.



# Exemple d'utilisation d'"Error Boundaries"

Voici un exemple d'utilisation d'un "Error Boundary" pour capturer les erreurs qui se produisent dans un composant enfant :

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

class ErrorBoundary extends Component {
  constructor(props) {
    super(props);
    this.state = { hasError: false };
  }

  componentDidCatch(error, info) {
    // Mettre à jour l'état pour afficher un message d'erreur convivial
    this.setState({ hasError: true });
    // Enregistrer l'erreur quelque part
    console.log(error, info);
  }

  render() {
    if (this.state.hasError) {
      return <h1>Oops, quelque chose s'est mal passé.</h1>;
    }
    return this.props.children;
  }
}

export default ErrorBoundary;
```