



1. Comprendre les concepts de base du web front-end

1.1. Fondamentaux des sites web statiques

Un site Web statique consiste en une série de fichiers HTML, chacun représentant une page physique d'un site Web. Sur les sites statiques, chaque page est un fichier HTML distinct. Lorsque vous visitez la page d'accueil, vous ne visualisez que le fichier de la page d'accueil réelle.

Les sites Web statiques sont assez simples et tous les sites Web ont été construits de cette façon pendant les premières années du World Wide Web.

1.2. Différence entre front End et back end :

C'est quoi front-end

Le front-end est construit à l'aide d'une combinaison de technologies telles que le langage de balisage hypertexte (HTML), JavaScript et les feuilles de style en cascade (CSS).

Les développeurs front-end conçoivent et construisent les éléments de l'expérience utilisateur sur la page Web ou l'application, y compris les boutons, les menus, les pages, les liens, les graphiques, etc.

C'est quoi back-end

Le back-end, également appelé côté serveur, se compose du serveur qui fournit les données à la demande, de l'application qui les canalise et de la base de données qui organise les informations.

Par exemple, lorsqu'un client parcourt des chaussures sur un site Web, il interagit avec le frontal.

Une fois qu'ils ont sélectionné l'article qu'ils souhaitent, l'ont mis dans le panier et autorisé l'achat, les informations sont conservées dans la base de données qui réside sur le serveur.

Quelques jours plus tard, lorsque le client vérifie l'état de sa livraison, le serveur extrait les informations pertinentes, les met à jour avec les données de suivi et les présente via le front-end.

1.3. Présentation des Frameworks front end :

Qu'est-ce qu'un framework ?

En gros, il s'agit d'une boîte à outils couplée à une bibliothèque pour programmeur informatique.

Il permet d'aider les programmeurs dans leur travail en leur proposant des morceaux de code pour mettre en place des fonctionnalités couramment demandées pour des applications. Par exemple, le multilinguisme, la gestion de la sécurité, la modularité (c'est-à-dire la possibilité pour l'utilisateur de personnaliser son interface).

L'avantage est que les programmeurs ne réinventent pas la roue chacun dans leur coin en programmant de A à Z tous les aspects de l'application demandée par le client.



Il faut savoir aussi qu'un projet nécessitera souvent l'utilisation de plusieurs frameworks. En effet, les frameworks sont généralement spécialisés dans un domaine bien précis :

- **Un framework back-end** permet à l'application de communiquer avec la base de données qui renferme des renseignements aussi précieux que les informations sur les utilisateurs, les sessions en cours, les articles en vente sur votre site... **Node JS** est un exemple de framework backend, tout comme **Ruby**, **Laravel**, **Django**, **Flask**...
- **Un framework front-end applicatif** permet de créer une interface pour que l'utilisateur puisse utiliser l'application. C'est le cas d'**Angular JS** mais aussi de **React**, **Vue JS**.
- Enfin, un **framework front-end de présentation** permet de définir l'apparence de l'application pour l'utilisateur. C'est le cas de **Bootstrap**.

Pourquoi utiliser les frameworks ?

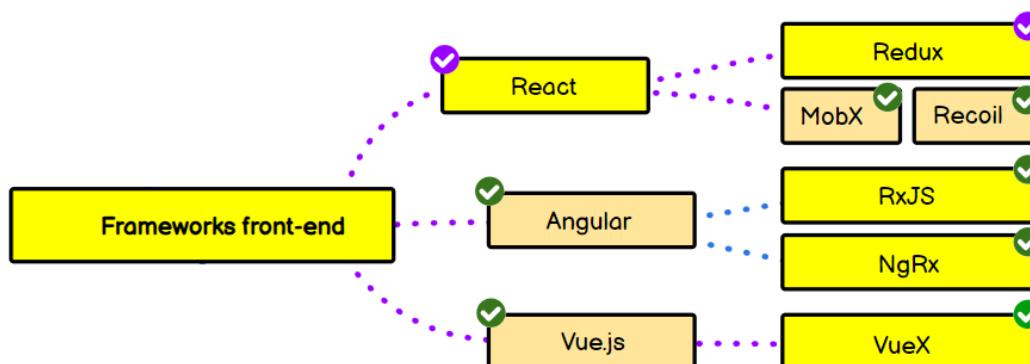
L'utilisation d'un framework n'est pas nécessaire dans un projet informatique, quelle que soit sa taille. Vous pouvez très bien vous en passer. Cependant, un framework est comme un squelette applicatif avec sa boîte à outils qui permet de développer plus rapidement, puisqu'il intègre une bonne partie de l'architecture, et invite les développeurs à suivre les normes de développement et de nommage tendant à une meilleure qualité du code.

Les avantages d'un framework ?

Utiliser un framework offre des avantages non négligeables :

- Les développeurs se concentrent uniquement sur la partie métier puisque toutes les couches techniques sont déjà intégrées dans le framework.
- L'architecture permet la séparation des couches techniques logiques afin de faciliter le développement en équipe, la maintenance et l'évolution.
- La maintenance et l'évolution du framework sont gérées par l'organisme fondateur.

Les frameworks front-end





React :



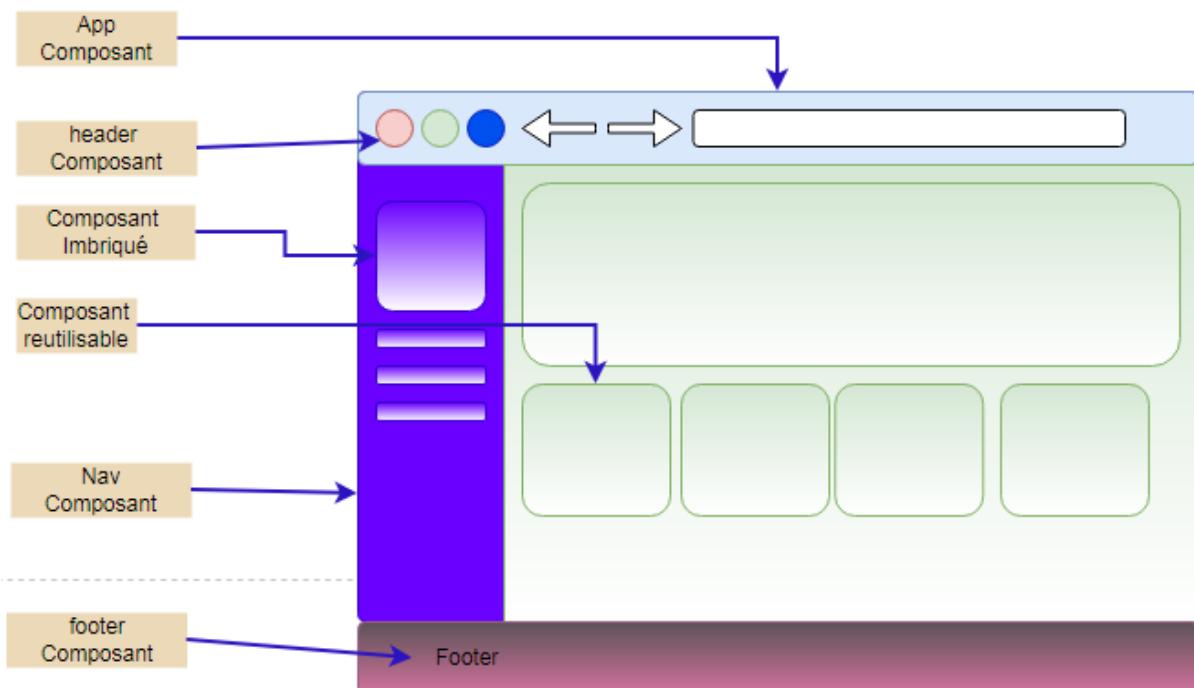
React c'est une librairie java script pour créer des interfaces utilisateur frontend, certaine personne l'appelle Framework, en général il ne faut pas prendre la tête avec est ce que React est une bibliothèque ou Framework, bref React c'est un utilitaire qui va nous aider à développer des interfaces utilisateur plus rapidement et facilement.

React est une librairie développée par Facebook est qui toujours maintenu avec Facebook

Pourquoi un Framework ou bibliothèque frontend ?

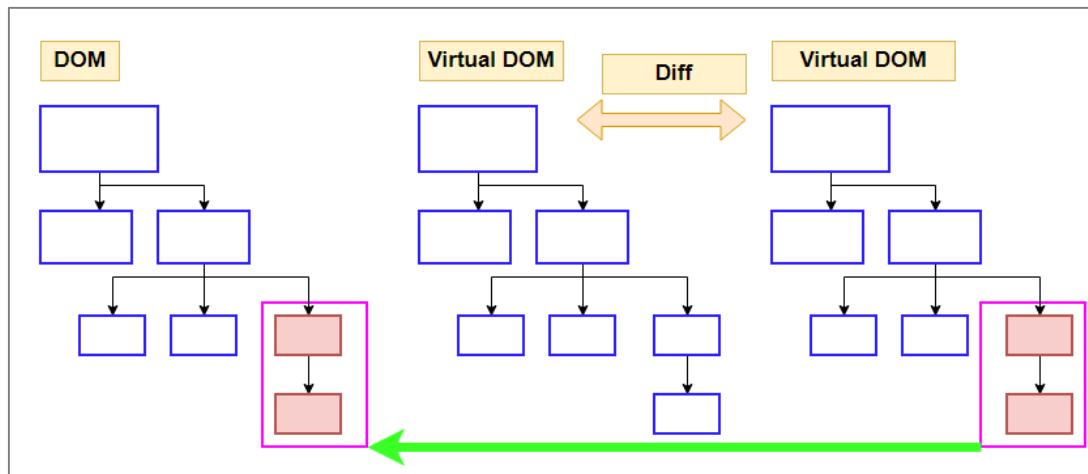
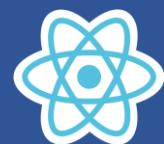
Avant l'arrivée des bibliothèques et Framework Frontend, les sites web ont été développé avec HTML, CSS, Java Script native, le site est constitué par plusieurs pages html, et quand l'utilisateur demande ou poste une information toutes la page est envoyée ce qui présente un lenteur de rafraîchissement, une charge sur le serveur et le navigateur.

L'approche des bibliothèques et Framework Frontend consiste à découper l'interface utilisateur en un ensemble de morceaux dite composants réutilisable.



Virtuel DOM :

Le DOM virtuel (VDOM) est un concept de programmation dans lequel une représentation idéale ou "virtuelle" d'une interface utilisateur est conservée en mémoire et synchronisée avec le "vrai" DOM par une bibliothèque telle que ReactDOM. Ce processus s'appelle la réconciliation.



Ajouter l'outils rendering a Google chrome pour afficher la partie du dom qui est rendu sera encadré

Paint flashing
 Highlights areas of the page (green) that need to be repainted. May not be suitable for people prone to photosensitive epilepsy.

React est le framework open-source développé et créé par Facebook. Ce framework est le meilleur framework d'interface utilisateur de 2022, utilisé par la majorité des développeurs frontaux selon Framework frontal, React se distingue par son modèle d'objet de document (DOM) virtuel, qui présente ses excellentes fonctionnalités.

Avantages

- Gain de temps lors de la réutilisation des composants
- Virtual DOM améliore à la fois l'expérience des utilisateurs et le travail du développeur
- Une bibliothèque open-source avec une diversité d'outils

Quand on utilise REACT :

React est utilisé pour développer une interface utilisateur riche, en particulier lorsque vous devez créer des applications d'une seule page. C'est le framework front-end le plus robuste lorsque vous avez besoin de créer une interface interactive en moins de temps car il prend en charge les composants réutilisables.

React requière une bonne maîtrise de JavaScript !!!



Quelques framework js pour le développement FrontEnd

ANGULAR :



Angular est une plate-forme et un framework permettant de créer des applications clientes single page à l'aide de HTML et de TypeScript.

Angular est écrit en TypeScript. Il implémente les fonctionnalités de base et facultatives sous la forme d'un ensemble de bibliothèques TypeScript que vous importez dans vos applications.

Angular est un framework JavaScript open-source écrit en TypeScript. Il est maintenu par Google.

Angular permet de développer des applications web SPA single page application. En tant que framework, Angular présente des avantages évidents tout en fournissant une structure standard avec laquelle les développeurs peuvent travailler. Il permet aux utilisateurs de créer de grandes applications de manière maintenable.

VUE JS



Vue est un framework JavaScript pour la construction d'interfaces utilisateur. Il s'appuie sur les standards HTML, CSS et JavaScript et fournit un modèle de programmation déclaratif et basé sur des composants qui vous aide à développer efficacement des interfaces utilisateur, qu'elles soient simples ou complexes.

Quels sont les avantages de Vue.JS ?

Flexibilité

L'écriture d'une application à l'aide de Vue.JS est rapide dû au fait qu'il est possible de l'exécuter via son navigateur. Cela facilite également le processus de test. Des applications beaucoup plus complexes comme ES6, JSX, Routing, Components et Bundlers peuvent également être construites en utilisant Vue.JS. Les développeurs peuvent l'utiliser de nombreuses façons différentes, car ce framework offre une grande flexibilité dans l'expression de leur code.

Intégration simple

Vue.JS offre également de grandes possibilités d'intégration avec les applications existantes, ce qui explique sa popularité auprès des développeurs. En effet, ce framework est basé sur JavaScript et peut donc être facilement intégré à d'autres plateformes utilisant JavaScript. Grâce à cette capacité, les développeurs peuvent travailler avec l'application en cours sans avoir à développer l'application à partir de zéro.



1.4. Les aspects avancés de JavaScript :

Le développement Frontend avec React nécessite une bonne maîtrise de Java script ES6.

Apprendre les fonctionnalités JavaScript est vraiment conseillé pour que vous soyez efficace dans la création d'applications avec React. Voici donc quelques fonctionnalités JavaScript que je vous recommande à apprendre afin que vous puissiez être aussi efficace que possible en travaillant avec React.

Les classes d'objets en ES6

Les classes sont des modèles pour créer des objets. Ils encapsulent les données avec du code pour travailler sur ces données.

JavaScript permet de créer des classes d'objets, mais dans les versions précédentes (avant ES6), la syntaxe est compliquée et ne ressemble pas à celle que l'on utilise dans les langages orientés objets similaires. ES6 propose une syntaxe plus classique, qui sera abondamment utilisée avec React.

Création d'une classe :

On utilise le nouveau mot-clé `class` (comme dans beaucoup d'autres langages). La création d'un objet de cette classe s'effectue au moyen de `new`. On définit ci-après une classe `Etudiant` (remarquez la majuscule sur la première lettre du nom de la classe, car tous les noms de classe doivent commencer par une majuscule), puis on crée un objet `etudiant` (ici, le nom de la variable commence par une minuscule, contrairement aux noms de classe).

```
class Etudiant{  
    constructor(nom,age){  
        this.nom=nom;  
        this.age=age  
    }  
}  
let et1= new Etudiant("Rami",23)  
let et2= new Etudiant("Karimi",21)
```

La méthode `constructor()` possède désormais les paramètres qui seront utilisés lors de la création des objets de la classe `Etudiant`. Afin de les conserver en tant qu'attributs dans la classe (et pouvoir les utiliser dans des méthodes qui seront définies dans cette classe), on les mémorise dans l'objet `this` (au moyen de `this.nom` et `this.age`). Remarquez que lors de la construction des objets par `new`, il faut maintenant indiquer en arguments les valeurs des paramètres utilisés, qui seront transmis dans la méthode `constructor()`. Si la méthode `constructor()` utilise des paramètres par défaut, les arguments correspondants peuvent être absents lors de la création des objets associés. On voit ici le contenu des deux objets créés, avec, entre accolades, les attributs créés au moyen de `this` dans le constructeur. Une classe est souvent enrichie au moyen de méthodes, qui seront ensuite utilisées par les objets de cette classe. Par



exemple, on peut créer une fonction info() dans la classe Personne qui permettra de visualiser le nom et age des objets de cette classe.

Création de la méthode info() dans la classe Etudiant :

```
class Etudiant{
    constructor(nom,age){
        this.nom=nom;
        this.age=age
    }
    info(){
        return `Etudiant nom:${this.nom} a pour
age:${this.age}`
    }
}
```

Appel de la méthode info() par les objets et1 et2

```
let et1= new Etudiant("Rami",23);
console.log(et1.info());
let et2= new Etudiant("Karimi",21) ;
console.log(et2.info());
```

Rendu

Etudiant nom:Rami a pour age:23
Etudiant nom:Karimi a pour age:21

Héritage de classe

L'héritage de classe permet de créer une nouvelle classe à partir d'une classe déjà existante. On dit que la nouvelle classe est dérivée (héritée) de la première. Il est ainsi possible d'enrichir (améliorer) une classe existante sans avoir à modifier son code interne. On peut alors réutiliser des parties de code existants (c'est-à-dire utiliser d'autres classes qui ont déjà été écrites et qui fonctionnent correctement). Pour illustrer cela, on souhaite créer une nouvelle classe, appelée Stagiaire, dérivée de la classe Etudiant.

```
class Stagiaire extends Etudiant{
    constructor(nom,age,stage){
        super(nom,age);
        this.stage=stage; }
    info(){return `Stagiaire nom:${this.nom} a pour
age:${this.age} stage:${this.stage}`}
}
```



La méthode info() peut être codé de la manière suivante

```
info(){
    return `${super.info()} stage:${this.stage}`
}
```

super(nom,age) : fait appelle au constructeur de la classe mère Etudiant

Création de deux instances de Stagiaire stg1 et stg2

```
let stg1= new Stagiaire("Rami",23,"dev mobile");
console.log(stg1.info());
let stg2= new Stagiaire("Karimi",21,"dev web")
console.log(stg2.info());
```

Rendu

```
Stagiaire nom:Rami a pour age:23 stage:dev mobile
Stagiaire nom:Karimi a pour age:21 stage:dev web
```

Modules

Le but des modules ES6 est d'aider à utiliser ces fichiers (ici, Etudiant.js et Index.js) sans avoir à connaître ces informations. En effet, ces dernières seront insérées sous une certaine forme dans le fichier lui-même, qui devient ainsi un module. C'est le rôle de l'export et de l'import des données, notions que nous détaillons dans les paragraphes qui suivent.

Un module c'est un fichier Java Script, qui contient les mots clés **export default** ou **export** elles les valeurs, classes, et fonctions qu'on veut exporter

C'est quoi la différence entre export default et export ?

Dans un module on peut faire seulement un seul export default, alors qu'on peut faire plusieurs exports.

Dans le module ci-dessous il un seul export default : **export default Etudiant** et deux export :

export {Etablissement,info}

L'élément exporté par default est importé sans accolades, tandis les éléments exportés par export sont importés avec des accolades

Voir module Index.js

```
import Etudiant,{Etablissement,info} from './Etudiant.js'
```

Module Etudiant.js

```
const Etablissement='ISGI'
class Etudiant{
    constructor(nom,age){
        this.nom=nom;
```



```

        this.age=age
    }
}
function info(etudiant){
    return `nom:${etudiant.nom} age:${etudiant.age}`
}
export default Etudiant
export {Etablissement,info}

```

Il est identique à :

```

export const Etablissement='ISGI'
export default class Etudiant{
    constructor(nom,age){
        this.nom=nom;
        this.age=age
    }
}
export function info(etudiant){
    return `nom:${etudiant.nom} age:${etudiant.age}`
}

```

Module Index.js

```

import Etudiant,{Etablissement,info} from './Etudiant.js'
let etd1= new Etudiant("Rami",33)
console.log(info(etd1))

```

Utilisation des aléas :

```

import Etd,{Etablissement as Etab,info} from './Etudiant.js'
let etd1= new Etd("Rami",33)
console.log(info(etd1)+" Etablissement: "+Etab)

```

Utilisation des modules dans la balise <script>

La page index.html

```

<!DOCTYPE html>
<html lang="en">
<head><meta charset="UTF-8"></head>
<body>
<script src="Index.js" type="module"></script>
</body>
</html>

```



Remarque : il faut ajouter l'attribut type= "module" à l'élément script

Template literals :

```
const salutation="salut "
const nom="Rami"
//avec template literal
console.log(` ${salutation} ${nom}` )
//est identique à
console.log(salutation+ " "+nom)
```

Rendu

salut	Rami
salut	Rami

En React : Voir Annexe 1

L'opérateur conditionnel ternaire

L'opérateur conditionnel (ternaire) est un opérateur JavaScript qui prend trois opérandes : une condition suivie d'un point d'interrogation (?), puis une expression à exécuter si la condition est vraie suivie de deux-points (:), et enfin l'expression à exécuter si la condition est fausse. Cet opérateur est fréquemment utilisé comme alternative à une instruction if...else.

L'opérateur conditionnel ternaire est très utilisé en react notamment en JSX exemple en Java Script ECS 6 :

```
let isMember=true;
let remise=isMember==true?0.2:0.1
console.log("remise: ",remise)
```

Rendu :

remise: 0.2

En React : Voir Annexe 2

Gestion des valeurs nulles avec opérateur conditionnel ternaire

Une utilisation courante consiste à gérer une valeur pouvant être nulle :

Exemple 1:

```
let nom
let salutation=nom?"salut "+nom:"inconnu"
console.log(salutation)
```

Rendu :

inconnu



Exemple 2 :

```
let nom="Rami"
let salutation=nom?"salut "+nom:"inconnu"
console.log(salutation)
```

Le rendu

salut Rami

Object destructuring

La syntaxe d'affectation de déstructuration est une expression JavaScript qui permet de décompresser des valeurs de tableaux, ou des propriétés d'objets, dans des variables distinctes.

```
const personne={
    nom:"fatihi",
    age:23,
    adresse:{rue:14,ville:"casa"}
}
const nom=personne.nom
const age=personne.age
const rue=personne.adresse.rue
const ville=personne.adresse.ville
console.log(nom,age,rue,ville)

//est identique à:
const {nom,age,adresse:{rue},adresse:{ville}}=personne
console.log(nom,age,rue,ville)
```

Le rendu

fatihi 23 14 casa

Destructuring avec fonction

```
const personne={
    nom:"fatihi",
    age:23,
    adresse:{rue:14,ville:"casa"}
}
function presentation({nom,age,adresse:{rue},adresse:{ville}}){
    console.log("salut ",nom,age,rue,ville)
}
presentation(personne)
```

Le rendu

salut fatihi 23 14 casa

En React : Voir Annexe 3



Opérations sur Array

La méthode map :

map est une méthode très populaire elle permet de :

- Retourne un nouveau Array
- Le nouveau Array a la même taille que l'Array d'origine
- Utilise les valeurs de l'Array d'origine pour faire le nouveau Array

Exemple : création d'un Array contenant les ages des personnes :

```
const personnes = [  
  { nom: "Rami", age: 33, estMember: true },  
  { nom: "Fatihi", age: 24, estMember: false },  
  { nom: "Chakib", age: 45, estMember: true },  
  { nom: "Mounir", age: 37, estMember: false },  
];  
  
const noms=personnes.map(function(pers){return pers.nom})  
console.log(noms)
```

Le rendu : ▶ (4) ['Rami', 'Fatihi', 'Chakib', 'Mounir']

Exercice 1 :

Soit l'Array nums=[2,5,8,7,3]

Utiliser la méthode map pour créer un Array contenant les éléments de nums multipliés par 3

nouvNums=[6,15,24,21,9]

La méthode filter

La méthode filter() crée une copie d'une partie d'un tableau donné, filtrée uniquement pour les éléments du tableau donné qui réussissent la condition implémentée par la fonction fournie.

Exemple :

```
const personnes = [  
  { nom: "Rami", age: 33, estMember: true },  
  { nom: "Fatihi", age: 24, estMember: false },  
  { nom: "Chakib", age: 45, estMember: true },  
  { nom: "Mounir", age: 37, estMember: false },  
];  
const persAgés=personnes.filter(function(pers){return  
pers.age>=35})  
console.log(persAgés)
```



Le rendu :

```
▼ (2) [{...}, {...}] ⓘ
▶ 0: {nom: 'Chakib', age: 45, estMember: true}
▶ 1: {nom: 'Mounir', age: 37, estMember: false}
  length: 2
▶ [[Prototype]]: Array(0)
```

L'Array persAgés contient les éléments de personnes qui vérifient la condition age>=35

Exercice 2 :

En utilisant les méthodes map et filter
créer un Array nomAgés à partir de l'Array personnes contenant les noms des personnes qui sont membre c-à-d estMember est true

En React : Voir Annexe 4

La méthode find

La méthode find() renvoie le premier élément du tableau fourni qui satisfait la fonction de test fournie. Si aucune valeur ne satisfait la fonction de test, undefined est renvoyé.

Exemple :

```
const personnes = [
  { nom: "Rami", age: 33, estMember: true },
  { nom: "Fatihi", age: 24, estMember: false },
  { nom: "Chakib", age: 45, estMember: true },
  { nom: "Mounir", age: 37, estMember: false },
];
const pers1=personnes.find(function(pers){return
pers.age>=45})
console.log(pers1)
```

Le rendu :

```
▶ {nom: 'Chakib', age: 45, estMember: true}
```

Exemple 2 :

```
const pers1=personnes.find(function(pers){return
pers.age==50})
console.log(pers1)
```

Le rendu :

```
undefined
```



La méthode reduce

La méthode `reduce()` exécute une fonction de rappel "reducer" fournie par l'utilisateur sur chaque élément du tableau, dans l'ordre, en transmettant la valeur de retour du calcul sur l'élément précédent. Le résultat final de l'exécution du réducteur sur tous les éléments du tableau est une valeur unique.

La première fois que le rappel est exécuté, il n'y a pas de "valeur de retour du calcul précédent". Si elle est fournie, une valeur initiale peut être utilisée à sa place. Sinon, l'élément de tableau à l'index 0 est utilisé comme valeur initiale et l'itération commence à partir de l'élément suivant (index 1 au lieu de l'index 0)

```
const clients = [
  { nom: "Rami", montant: 4500 },
  { nom: "Karimi", montant: 2300 },
  { nom: "Chaouki", montant: 5500 },
  { nom: "Ramoun", montant: 7700 },
];
const totalMontants=clients.reduce(function(total,client){
  return total+=client.montant
},0);
console.log(totalMontants)
```

Destructuring d'un array

Moyen plus rapide et plus simple d'accéder à la variable de décompression à partir d'un tableau ou d'objets.

```
let numbers=[10, 20, 30, 40, 50]
const [a, b,...rest] = numbers;
console.log(a,b,rest);
```

Le rendu : 10 20 ► (3) [30, 40, 50]

Remarque :

rest est un array contenant une copie de Array numbers sans le premier et le deuxième élément

Créer une copie d'une Array

```
const numbers2=[...numbers]
console.log(numbers2)
```

Le rendu

► (5) [10, 20, 30, 40, 50]



Permuter les valeurs de deux variables :

```
let x=10;
let y=20;
[x,y]=[y,x]
console.log("x=",x," y=",y)
```

Le rendu :

x= 20 y= 10

La programmation fonctionnelle

La programmation fonctionnelle est un paradigme de construction de programmes informatiques à l'aide d'expressions et de fonctions sans mutation d'état et de données. En respectant ces restrictions, la programmation fonctionnelle vise à écrire du code plus clair à comprendre et plus résistant aux bogues.

Programmation fonctionnelle

modulaire

La fonctionnalité de votre programme doit être divisée en modules indépendants, chacun contenant ce dont il a besoin pour exécuter un aspect de la fonctionnalité du programme. Les modifications apportées à un module ou à une fonction ne doivent pas affecter le reste du code

compréhensible

Un lecteur de votre programme devrait être capable de discerner ses composants, leurs fonctions et leurs relations sans effort excessif. Ceci est étroitement lié à la maintenabilité du code ; votre code devra être maintenu à un moment donné dans le futur, que ce soit pour être modifié ou pour ajouter de nouvelles fonctionnalités

testable

Les tests unitaires testent de petites parties de votre programme, vérifiant leur comportement en fonction du reste du code. Votre style de programmation doit favoriser l'écriture de code qui simplifie le travail d'écriture de tests unitaires. Les tests unitaires sont également comme de la documentation en ce sens qu'ils peuvent aider les lecteurs à comprendre ce que le code est censé faire

extensible

C'est un fait que votre programme nécessitera un jour une maintenance, peut-être pour ajouter de nouvelles fonctionnalités. Ces modifications ne devraient avoir qu'un impact minime sur la structure et le flux de données du code d'origine (voire pas du tout). De petits changements ne doivent pas impliquer une refactorisation importante et sérieuse de votre code

réutilisable

La réutilisation du code a pour objectif d'économiser des ressources, du temps et de l'argent, et de réduire la redondance en tirant parti du code précédemment écrit. Certaines caractéristiques contribuent à cet objectif, telles que la modularité (que nous avons déjà mentionnée), une forte cohésion (toutes les pièces d'un module vont ensemble), un faible couplage (les modules sont indépendants les uns des autres)



JavaScript n'est pas un langage purement fonctionnel, mais il possède toutes les fonctionnalités dont nous avons besoin pour qu'il fonctionne comme s'il l'était. Les principales caractéristiques du langage que nous allons utiliser sont les suivantes

function as first-class objects

recursive

Arrow functions

closures

spread

La récursivité

La récursivité est l'outil le plus puissant pour développer des algorithmes et une grande aide pour résoudre de grandes classes de problèmes. L'idée est qu'une fonction peut à un certain point s'appeler elle-même, et quand cet appel est fait, continuer à travailler avec le résultat qu'elle a reçu. Ceci est généralement très utile pour certaines classes de problèmes ou de définitions. L'exemple le plus souvent cité est la fonction factorielle (la factorielle de n s'écrit $n!$) telle que définie pour des valeurs entières positives :

```
function fact(n){  
    if (n==0){  
        return 1  
    }  
    else{  
        return n* fact(n-1)  
    }  
}  
  
console.log(fact(4))
```

Fermetures

Les fermetures sont un moyen d'implémenter le masquage des données (avec des variables privées), ce qui conduit à des modules et à d'autres fonctionnalités



intéressantes. Le concept clé des fermetures est que lorsque vous définissez une fonction, elle peut faire référence non seulement à ses propres variables locales, mais également à tout ce qui se trouve en dehors du contexte de la fonction. Nous pouvons écrire une fonction de comptage qui gardera son propre compte au moyen d'une fermeture

```
function compteur(){
    let count=0
    return function(){
        count++
        return count
    }
}
let next=compteur()
console.log(next())
console.log(next())
console.log(next())
console.log(next())
```

Le rendu

1
2
3
4

Fonction as first-class object

Dire que les fonctions sont des objets de première classe (également appelés citizens de première classe) signifie que vous pouvez tout faire avec des fonctions que vous pouvez faire avec d'autres objets. Par exemple, vous pouvez stocker une fonction dans une variable, vous pouvez la passer à une fonction, vous pouvez l'imprimer, etc. C'est vraiment la clé de la FP ; nous passerons souvent des fonctions en tant que paramètres (à d'autres fonctions) ou retournant une fonction à la suite d'un appel d'une fonction.

Exemple compteur voir TD:

```
spanHeure=document.getElementById("spHeure")
spanMinute=document.getElementById("spMinute")
spanSecond=document.getElementById("spSecond")
let initTime={heure:10,minute:23,second:5}
let currentTime=initTime

let timer=null
function incrementer(temp){
    time={...temp}
    return function(){
        time.second++
        if(time.second>=60){
```



```
        time.second=0
        time.minute++
        if(time.minute>=60){
            time.minute=0
            time.heure++
        }
    }
    affiche(time)
    currentTime=time
    return time
}

function affiche(time){
spanHeure.innerHTML=time.heure;
spanMinute.innerHTML=time.minute;
spanSecond.innerHTML=time.second;
}

function stopTimer() {
    if(timer){
        clearInterval( timer);
    }
    timer = null;
}

function incrementerTimer() {

    stopTimer()
    let next=incrementer(currentTime)
    timer=setInterval(next,1000);
}
```

Ici on a :

- ✓ La fonction incrementer est une fermeture elle retourne une fonction !!!
- ✓ La fonction affiche est appelé au sein de la fonction incrementer,
- ✓ La fonction **next** est passée en paramètre dans la fonction setInterval



Arrow fonction

Les arrow fonctions sont juste un moyen plus court et plus succinct de créer une fonction (sans nom). Les arrow fonctions peuvent être utilisées presque partout où une fonction classique peut être utilisée, sauf qu'elles ne peuvent pas être utilisées comme constructeurs

La fonction somme :

```
function somme(a,b){  
    return a+b  
}
```

Peut-être écrite en utilisant arrow fonction

```
const mySomme=(a,b)=>a+b  
console.log(mySomme(3,4))
```

Remarque :

Les fonctions fléchées sont généralement appelées fonctions anonymes en raison de leur absence de nom. Si vous avez besoin de faire référence à une fonction fléchée, vous devrez l'affecter à une variable ou à un attribut d'objet, comme nous l'avons fait ici ; sinon, vous ne pourrez pas l'utiliser.

Exemple :

```
const fact3 = n => (n === 0 ? 1 : n * fact3(n - 1));
```

L'opérateur de propagation spread

Vous permet de développer une expression aux endroits où vous nécessiterait autrement plusieurs arguments, éléments ou variables. Par exemple, vous pouvez remplacer des arguments dans un appel de fonction, comme illustré dans le code suivant :

```
function produit(a,b,c){  
    return a*b*c  
}  
  
let numbers=[2,3,4]  
console.log(produit(...numbers))
```

Pures fonctions :

Les fonctions pures acceptent des entrées (arguments) et renvoient une nouvelle valeur sans avoir à modifier en permanence des variables en dehors de sa portée.

Dans la fonction pure ci-dessous ‘pureFonction’ , nous avons pu récupérer une nouvelle valeur sans modifier la variable ‘immutableValue’ .

Alors que la fonction ‘impureFonction’ a changée la valeur de la variable ‘mutateValue’ d'où ‘impureFonction’ n'est pas pure !!!.



```
let mutateValue = 10;
function impureFonction(newValue) {
    return mutateValue += newValue;
}
impureFonction(20)
console.log(mutateValue)
//rendu 30
immutableValue=10
function pureFonction(newValue){
    return immutableValue+newValue
}
pureFonction(20)
console.log(immutableValue)
//rendu 10
```

Alors pourquoi la fonction pure est-elle si importante ?

Les fonctions pures favorisent la maintenabilité et la réutilisabilité ; à l'opposé, la fonction impure limite ces qualités car lorsqu'elles mutent des valeurs en dehors de son champ d'application, l'effet secondaire qu'elle provoque peut être indésirable. Cela favoriserait à son tour le code bourré de bogues.

Chose intéressante, React + Redux (qui, à mon avis, devient le framework/bibliothèque Javascript standard de l'industrie) pratique des fonctions pures pour sa gestion d'état. Inutile de dire que développer une habitude d'écrire des fonctions pures est important pour une transition facile dans le framework.

Pures fonctions et Arrays :

Ajouter un élément dans un Array

```
//ajouter un element dans la liste
myList=[1,2,3,4,5]
function pureAppend(value){
appendMyList=[...myList,value];
return appendMyList;}
//affichage
console.log(pureAppend(10))
//rendu [1,2,3,4,5,10]
console.log(myList)
//rendu [1,2,3,4,5]
```

On remarque myList n'a pas changé





Exercice 3 :

Soit l'Array

```
const inscriptions=[  
    {id:10,nom:'Rami',filiere:'DEV'},  
    {id:11,nom:'Kamali',filiere:'DEV'},  
    {id:12,nom:'Fahmi',filiere:'DEV'},  
    {id:13,nom:'Chaouki',filiere:'DEV'}  
];
```

Créer la pure fonction qui permet d'ajouter une inscription

Insérer dans un Array

La méthode splice() modifie le contenu d'un tableau en supprimant ou en remplaçant des éléments existants et/ou en ajoutant de nouveaux éléments en place. Pour accéder à une partie d'un tableau sans le modifier

```
const months = ['Jan', 'March', 'April', 'June'];  
months.splice(1, 0, 'Feb');  
// inserts at index 1  
console.log(months);  
// expected output: Array ["Jan", "Feb", "March", "April", "June"]  
  
months.splice(4, 1, 'May');  
// replaces 1 element at index 3  
console.log(months);  
// expected output: Array ["Jan", "Feb", "March", "April", "May"]
```

Inject avec pure fonction

```
let dontMutateInjection = [6, 7, 8];  
  
function pureInject(index,newValue) {  
    return [  
        ...dontMutateInjection.slice(0, index),  
        newValue,  
        ...dontMutateInjection.slice(index)  
    ]  
}  
console.log('-----')  
console.log(pureInject(1,10))  
//rendu [6,10,7,8]  
console.log(dontMutateInjection)  
//rendu [6,7,8]
```



Supprimer un élément d'un Array

Soit :

```
const inscriptions=[  
    {id:10,nom:'Rami',filiere:'DEV'},  
    {id:11,nom:'Kamali',filiere:'DEV'},  
    {id:12,nom:'Fahmi',filiere:'DEV'},  
    {id:13,nom:'Chaouki',filiere:'DEV'}  
];
```

```
function pureDeleteFonction(id){  
    return [...inscriptions.filter((ins)=>ins.id!=id)]  
}  
let deleteInscriptions= pureDeleteFonction(12)  
console.log('---pureDeleteFonction-----')  
console.log(deleteInscriptions)  
console.log(inscriptions)
```

Le rendu

```
▼ (3) [{} , {} , {} ] ⓘ  
▶ 0: {id: 10, nom: 'Rami', filiere: 'DEV'}  
▶ 1: {id: 11, nom: 'Kamali', filiere: 'DEV'}  
▶ 2: {id: 13, nom: 'Chaouki', filiere: 'DEV'}  
  length: 3  
▶ [[Prototype]]: Array(0)  
▼ (4) [{} , {} , {} , {} ] ⓘ  
▶ 0: {id: 10, nom: 'Rami', filiere: 'DEV'}  
▶ 1: {id: 11, nom: 'Kamali', filiere: 'DEV'}  
▶ 2: {id: 12, nom: 'Fahmi', filiere: 'DEV'}  
▶ 3: {id: 13, nom: 'Chaouki', filiere: 'DEV'}  
  length: 4  
▶ [[Prototype]]: Array(0)
```

Modifier un élément d'un Array

Soit

```
const notes=[  
    {id:10,module:'Algorithme',note:15},  
    {id:10,module:'POO',note:17},  
    {id:11,module:'Algorithme',note:16},  
    {id:11,module:'POO',note:14},  
];
```



On souhaite changer la note POO de l'étudiant dont le id =10

```
//pure fonction modification
function pureUpdateNote(id,module,note){
  const updatednotes= [...notes.filter(note=> !(note.id==id &&
note.module==module)),{id:id,module:module,note:note}]
  return updatednotes;
}
console.log("----pureUpdateNote-----")
console.log(pureUpdateNote(10,'POO',18))
console.log(notes)
```

Le rendu :

```
▼ (4) [{} , {} , {} , {}] 1
▶ 0: {id: 10, module: 'Algorithme', note: 15}
▶ 1: {id: 11, module: 'Algorithme', note: 16}
▶ 2: {id: 11, module: 'POO', note: 14}
▶ 3: {id: 10, module: 'POO', note: 18}
  length: 4
  ▶ [[Prototype]]: Array(0)

▼ (4) [{} , {} , {} , {}] 1
▶ 0: {id: 10, module: 'Algorithme', note: 15}
▶ 1: {id: 10, module: 'POO', note: 17}
▶ 2: {id: 11, module: 'Algorithme', note: 16}
▶ 3: {id: 11, module: 'POO', note: 14}
  length: 4
  ▶ [[Prototype]]: Array(0)
```

Changer un objet

Soit l'objet

```
personne={id:10,nom:'Rami'}
```

on souhaite avoir un objet de type personne avec en plus l'attribut age sans modifier l'objet initial

```
personne={id:10,nom:'Rami'}
//changer un objet en utilisant pure fonction
function addAge(age){
  let pers= {...personne,age:age}
  return pers;

}
console.log('changer personne')
console.log(addAge(33))
console.log(personne)
```

La fonction peut être écrite autrement

```
function addAge(age){
  return Object.assign({}, personne, {age:age});
}
```



2. Créer des applications web monopage (SPA)

2.1. Notion d'application web monopage SPA

Qu'est-ce qu'une SPA ?

Une application sur une seule page (SPA) diffère d'une page conventionnelle en cela qu'elle est rendue côté client et qu'elle est principalement pilotée par JavaScript, en utilisant les appels Ajax pour charger les données et mettre la page à jour dynamiquement. La plupart ou la totalité du contenu est récupérée une fois au chargement d'une seule page avec des ressources supplémentaires chargées de manière asynchrone, selon les besoins, en fonction de l'interaction de l'utilisateur avec la page.

Cela limite la nécessité d'actualiser la page et offre à l'utilisateur une expérience harmonieuse, rapide et rappelant davantage l'expérience d'une application native.

La raison d'être des SPA ("Single Page Application" ou "Application Monopage") est de rendre les applications utilisables sur presque tous les appareils.

Pourquoi une SPA ?

Plus rapide, fluide et ressemblant davantage à une application native, une SPA, de par son fonctionnement, offre une expérience très attrayante, non seulement pour le visiteur de la page web, mais aussi pour les spécialistes du marketing et les développeurs.

2.2. Single Page Application (SPA) vs Multi-Page Application (MPA)

MPA (Multiple Page Application) – concept de base

SPA et MPA sont tous deux basés sur le protocole HTTP.

Dans une architecture MPA client/serveur traditionnelle, chaque clic de l'utilisateur déclenche une requête HTTP vers le serveur. Le résultat de cette nouvelle requête est un rafraîchissement complet de la page, même si une partie du contenu reste inchangée.

SPA – Ajax et JavaScripts

Le cœur d'un SPA est basé sur Ajax, un ensemble de techniques de développement qui permet au client d'envoyer et de récupérer des données du serveur de manière asynchrone (en arrière-plan) sans interférer avec l'affichage et le comportement de la page web. Ajax permet aux pages web et, par extension, aux applications web, de modifier le contenu de manière dynamique sans avoir à recharger la page entière.

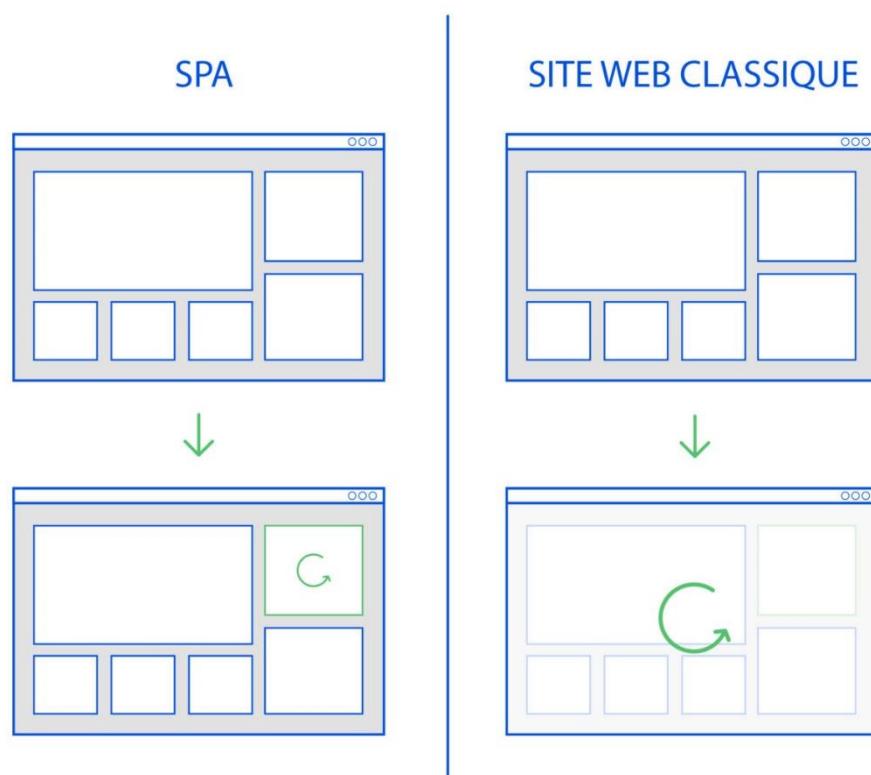
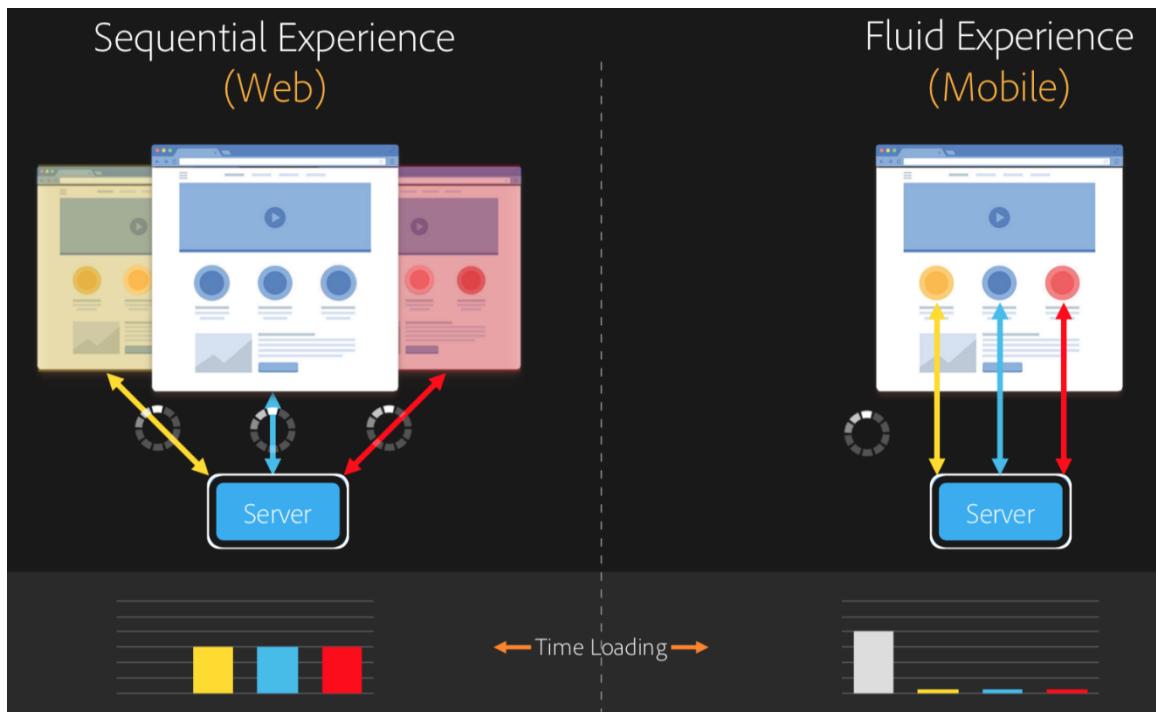
2.3. Fonctionnement d'une application web de type SPA

L'idée Principale derrière une SPA est que les appels et la dépendance à l'égard d'un serveur sont réduits afin de minimiser les retards causés par les appels au serveur, de sorte que SPA s'approche de la réactivité d'une application native.

Sur une page web séquentielle traditionnelle, seules les données nécessaires à la page immédiate sont chargées. Cela signifie que lorsque le visiteur passe à une autre page, le serveur est appelé pour que les ressources supplémentaires soient mises à



disposition. Des appels supplémentaires peuvent être nécessaires lorsque le visiteur interagit avec les éléments de la page. Ces appels multiples peuvent donner une impression de retard ou de lenteur, car la page doit rattraper les requêtes du visiteur.



Pour offrir une expérience plus fluide, qui approche ce qu'un visiteur attend des applications mobiles natives, un SPA charge toutes les données nécessaires pour le visiteur au premier chargement. Bien que cette opération puisse nécessiter au début



un peu plus de temps, elle élimine ensuite la nécessité d'appels supplémentaires au serveur.

En effectuant un rendu côté client, l'élément de page réagit plus rapidement et les interactions du visiteur avec la page sont immédiates. Toutes les données supplémentaires qui peuvent être nécessaires sont appelées de manière asynchrone afin d'optimiser la vitesse de la page.

2.4. Avantages et inconvénients d'une application web de type SPA

Les avantages d'une application à unique page SPA

Les applications à page unique sont connues pour tirer parti des mises en page répétitives et du contenu à la demande. Comme elles tirent parti de ces deux concepts, elles sont beaucoup plus efficaces, coûtent nettement moins cher que les applications traditionnelles et consomment également moins d'énergie. Voici quelques avantages à l'utilisation des applications à page unique.

Vitesse accrue

La vitesse doit être l'un des plus grands avantages d'opter pour le développement d'une application à page unique. Elles sont beaucoup plus rapides que les applications web traditionnelles car elles peuvent charger de nouvelles informations dans une seule page à la demande du client au lieu de relier plusieurs pages HTML à l'architecture du site.

Expérience utilisateur (UX)

Comme nous l'avons vu précédemment, les applications à page unique sont les leaders en matière d'expérience utilisateur pratique et directe. La plupart des utilisateurs sont souvent déroutés et irrités de devoir cliquer sur de nombreux liens dans une application traditionnelle pour arriver là où ils veulent.

Capacités de mise en cache

Une application à page unique est capable de mettre en cache les données locales plus efficacement. Comme l'application à page unique n'envoie qu'une seule requête au serveur et stocke toutes les données qu'elle reçoit, elle peut également utiliser ces données pour travailler hors ligne.

Débogage avec Chrome

Le débogage des SPA avec Chrome est beaucoup plus facile que le débogage de n'importe quelle application à pages multiples. De plus, comme les SPA sont construites avec des frameworks tels que Angular et React, ces frameworks ont leurs propres outils de débogage Chrome, ce qui facilite grandement leur débogage.

Les inconvénients d'une application à unique page SPA

Historique du navigateur

L'un des inconvénients d'une expérience utilisateur transparente et de l'absence de chargement de page avec les applications à page unique est que les SPA n'enregistrent pas les sauts des utilisateurs entre les différents états.



Cela signifie que lorsque l'utilisateur appuie sur le bouton « retour », il ne peut pas revenir en arrière. Le bouton retour ne ramène l'utilisateur qu'à la page précédente et non à l'état précédent dans l'application.

Optimisation du référencement

L'un des plus grands signaux d'alarme qui empêchent le développeur d'explorer les applications à page unique est la difficulté d'optimisation du référencement.

Problèmes de sécurité

Par rapport aux applications multi pages, les applications monopages sont plus vulnérables aux attaques par cross-site scripting. En effet, les pirates peuvent injecter des scripts côté client dans les applications web en utilisant le XSS.

Une application à page unique est parfaite comme base pour le développement futur d'applications mobiles. Les SPA visent à offrir une expérience utilisateur exceptionnelle en essayant de reproduire un environnement naturel dans le navigateur : pas de recharge de page ni de temps d'attente supplémentaire.



3. Comprendre les concepts de React JS

3.1. Présentation de React :

React (Ou ReactJS, React.js) est une bibliothèque JavaScript développée par Facebook, utilisée pour créer des composants d'affichage réutilisables.

Pour créer une application avec React, on va donc créer des composants d'affichage (en général des classes ou des fonctions) qui seront ensuite assemblés afin de former l'application finale. L'intérêt de React, outre le fait de créer des composants réutilisables, et de permettre aussi une mise à jour rapide de la page HTML.

En effet, lors de l'utilisation de bibliothèques telles que jQuery (qui manipulent les éléments HTML de la page, c'est-à-dire le DOM, directement), les performances sont bien moindres car la manipulation du DOM directement en JavaScript est lente lors de l'exécution. React ne manipule pas le DOM directement mais une copie interne de celui-ci (les objets ou éléments React, appelés le DOM virtuel), et produit les modifications sur l'affichage uniquement lorsque cela s'avère nécessaire.

Enfin, React n'est pas seulement une bibliothèque permettant d'effectuer des affichages sur le Web, elle peut également servir à produire des applications natives iPhone et Android, en utilisant une variante nommée React Native.

3.2. L'écosystème React :

NPM

npm fait deux choses : gérer vos dépendances et lancer des scripts

La gestion de dépendance vous assure que les librairies que vous utilisez sont à jour. Si vous avez un background plus orienté back-end, c'est l'équivalent de composer pour PHP, maven pour Java, encore pip pour Python, ou encore Bundler pour Ruby. Côté front-end, npm c'est beaucoup plus suivi par la communauté React, et il gérera aussi toutes vos dépendances en node.

Le lanceur de script exécute les différentes tâches dont vous avez besoin dans votre projet : la compilation, le démarrage de l'application, etc. C'est aussi un excellent moyen d'unifier les projets en proposant des standards valables sur tous les projets.

Quand vous récupérez un projet dans le Grand Internet, généralement vous aurez les commandes suivantes :

- **npm install** : installe le projet sur votre machine.
- **npm start** : démarre le projet.
- **npm test** : lance les tests du projet.
- **npm run dev** : s'occupe de lancer un environnement de développement agréable.



ECMAScript6

ECMAScript est un standard de langage de programmation, il définit : La syntaxe, Les types de variable et encore pas mal d'autres choses...

ES6 : Il a été publié en 2015, d'ailleurs vous pouvez aussi le voir aussi sous le nom ES2015. Beaucoup de choses ont évolué en ES6... **Fonctions Fléchées, Classes, Modules...**

Babel

Afin de faire de l'ES6 dès aujourd'hui, il faut le compiler, ou le transpiler. Pour cela il y a notamment **Babel** qui va transformer pour vous le code en ES5 (version qui est supportée par tous les navigateurs modernes). Il peut aller plus loin que de l'ES6 et a tendance à proposer des solutions via un système de plugin pour les propositions qui sont parfois encore en draft.

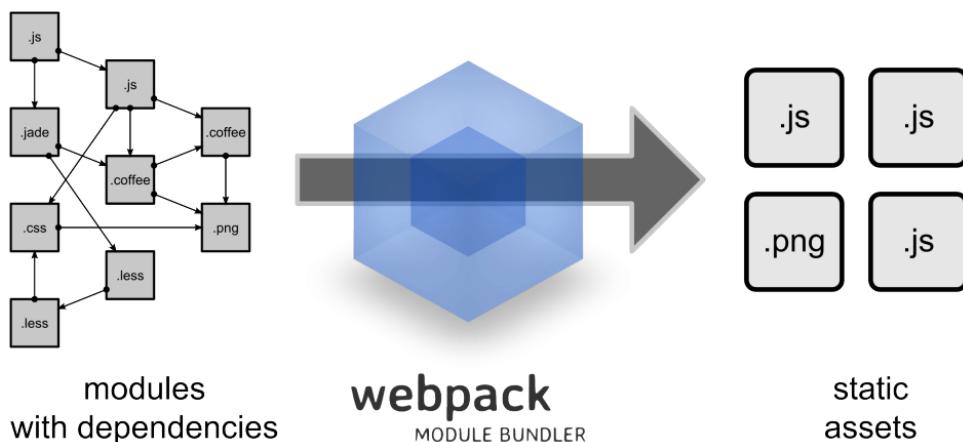
Babel fonctionne avec des presets, qui sont un ensemble de plug-ins permettant de compiler toutes les tâches liées à la technologie qu'on voudra compiler. En ce qui nous concerne, on aura besoin des presets suivants : babel-preset-es2015 et babel-preset-react.

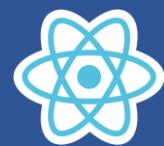
Bundlers

Le bundler condense tout le code de votre application pour n'avoir qu'un seul fichier javascript à importer dans votre page HTML. L'intérêt majeur c'est que vous n'êtes plus contraints d'avoir un fichier obèse quand vous codez du JavaScript, où il est impossible de retrouver votre fonction. Vous pouvez faire des modules (fichiers isolés), y faire référence dans un autre fichier et tout de même être capable de les utiliser dans le navigateur sans ajouter une balise <script> à chaque fois que vous ajoutez un fichier.

Les deux outils majeurs sur le marché sont Browserify et Webpack.

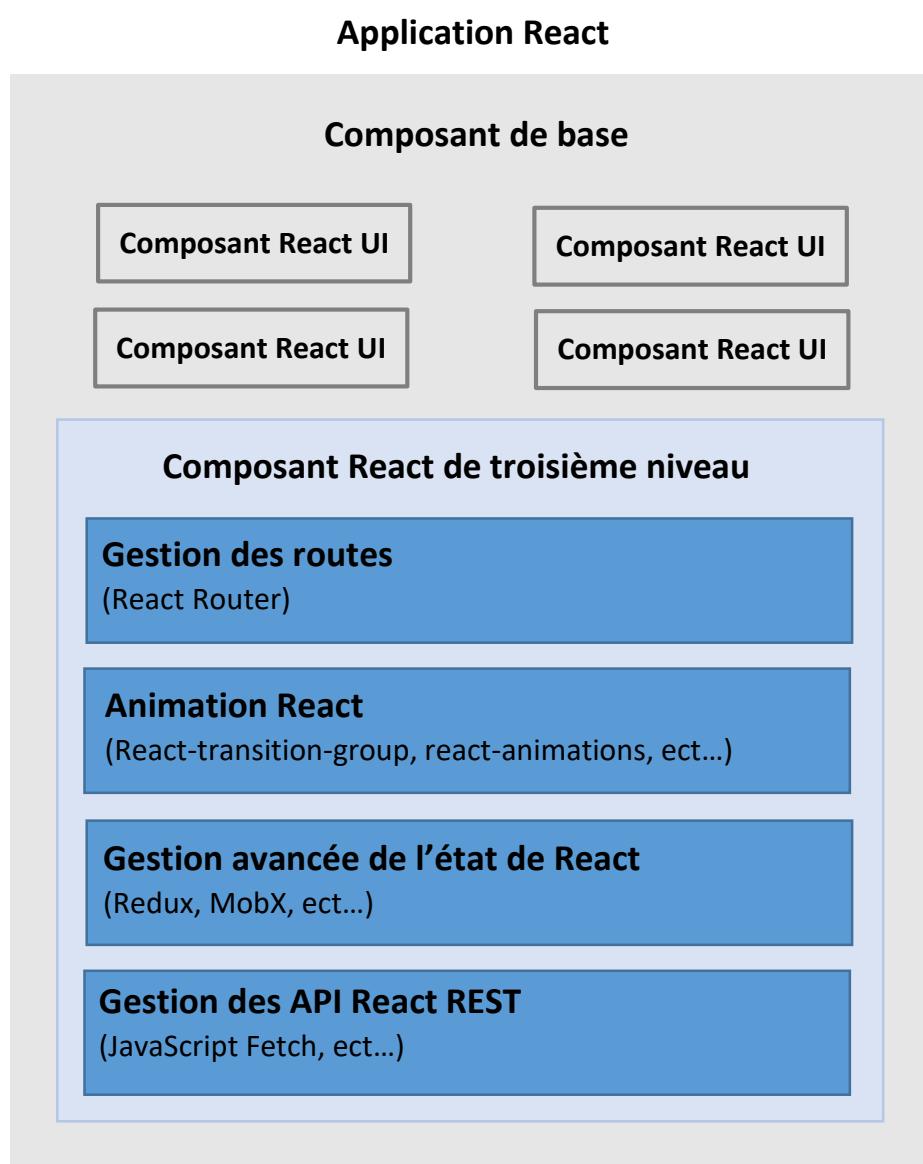
Webpack est un outil qui permet de prendre en compte la modularité du code JavaScript. Il peut le faire selon les différents standards (commonJs, AMD, etc.). En revanche, à lui tout seul, il est incapable de transformer le code ES2015 ou JSX.





3.3. Architecture d'une application React :

Afin d'écrire un bon code, la bibliothèque React propose différents concepts comme : **le composant d'ordre supérieur, le contexte, les accessoires de rendu, les références**, etc. **React Hooks** est utilisé pour faire gérer de grands projets. Examinons l'architecture d'une application React :



1. L'application React commence à démarrer avec un seul composant racine ;
2. Le composant racine est créé grâce à un ou plusieurs composants ;
3. Chaque composant est formé comme un composant de composants plus petits au lieu de dépendre d'un autre composant ;
4. La grande partie des composants sont des composants d'interface utilisateur ;
5. L'application React a la possibilité d'ajouter un composant tiers à des fins spécifiques comme le routage, l'animation, la gestion de l'état, etc.



Créer votre toute première application React.js (*Voir le document des travaux pratiques*)

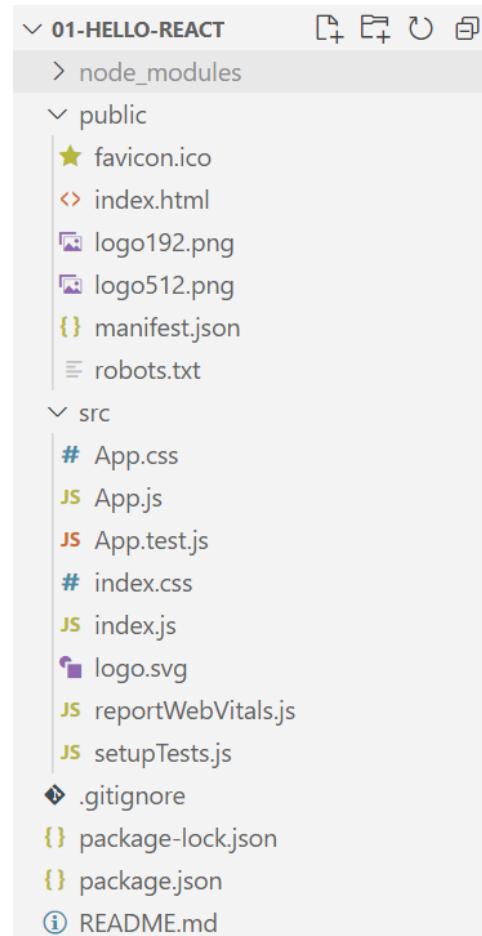
Structure de projet:

01-HELLO-REACT

```

    └── README.md
    └── node_modules
    └── package-lock.json
    └── package.json
    └── .gitignore
    └── public
        └── favicon.ico
        └── index.html
        └── logo192.png
        └── logo512.png
        └── manifest.json
        └── robots.txt
    └── src
        └── App.css
        └── App.js
        └── App.test.js
        └── index.css
        └── index.js
        └── logo.svg
        └── reportWebVitals.js
        └── setupTests.js

```



Le dossier **node_modules** contient toutes les librairies javascript dont vous aurez besoin

Le dossier **public** vous y trouverez le fichier index.html qui est notre point départ avec la div qui a pour class root qui permet de signifier à React où est ce qu'il doit venir s'ancrer.

Le fichier **.gitignore** a pour rôle de signifier à git les fichiers qui ne doivent pas être pusher sur le repo distant notamment le dossier node module.

Le fichier **package.json** est certainement le fichier qui contient plusieurs informations sur votre application React notamment les scripts, les dépendances.

Le fichier **Readme** permet de documenter votre app en Markdown.

Le dossier **src** est l'endroit où toute la magie de votre application va opérer, on y retrouve le fichier index.js qui est le fichier qui charge notre premier composant le composant App qui est ensuite greffer à la div avec la class root qui se trouve dans le fichier index.hmtl au niveau du dossier public.



3.4. Explication du code :

Quand vous créez un projet React avec **create-react-app**, il y a deux fichiers principal, le fichier **public/index.html** et le fichier **src/index.js**

Commençons par le fichier **public/index.html** :

Ce fichier ne contient pas grand-chose, c'est là qu'est défini la structure de base d'une page HTML, plus important il y est défini l'id root

```
<!DOCTYPE html>
<html lang="en">
  <head>
    <meta charset="utf-8" />
    <link rel="icon" href="%PUBLIC_URL%/favicon.ico" />
    <link rel="apple-touch-icon" href="%PUBLIC_URL%/logo192.png" />
    <link rel="manifest" href="%PUBLIC_URL%/manifest.json" />
    <title>React App</title>
  </head>
  <body>
    <noscript>You need to enable JavaScript to run this app.</noscript>
    <div id="root"></div>
  </body>
</html>
```

C'est en effet dans cette div que notre application sera afficher.

Passons ensuite au fichier **src/index.js** :

Ce fichier va permettre de récupérer nos composants et les afficher dans la div#root de public/index.html

```
1 import React from 'react';
2 import ReactDOM from 'react-dom/client';
3 import './index.css';
4 import App from './App';
5 import reportWebVitals from './reportWebVitals';
6
7 const root =
  ReactDOM.createRoot(document.getElementById('root'));
  root.render(
    <React.StrictMode>
      <App />
    </React.StrictMode>
  );
  reportWebVitals();
```



La ligne 1 : importe la bibliothèque React elle-même. Comme React transforme l'instruction JSX que nous écrivons en `React.createElement()`, tous les composants React doivent importer le module React. Si vous ignorez cette étape, votre application produira une erreur.

La ligne 2 : importe Le module react-dom qui fournit des méthodes spécifiques au DOM que vous pouvez utiliser au niveau racine de votre application et comme échappatoire pour travailler hors du modèle React si vous en avez besoin.

La ligne 3 : importe la feuille de style `src/index.css`

La ligne 4 : importe le fichier `src/App.js` qui définit une fonction nommée `App`.

La ligne 6 : affiche l'élément React `<App />` au sein du nœud DOM `root` et renvoie une référence sur le composant.

Ligne 5 et 7 : Web Vitals sont un ensemble de mesures utiles qui visent à capturer l'expérience utilisateur d'une page Web.

Avec la fonction `reportWebVitals`, vous pouvez envoyer n'importe résultats à un service d'analyse d'audience comme Google Analytics.

Passons maintenant au composant App (`src/App.js`) :

```
import logo from './logo.svg';
import './App.css';
function App() {
  return (
    <div className="App">
      <header className="App-header">
        <img src={logo} className="App-logo" alt="logo" />
        <p>
          Edit <code>src/App.js</code> and save to reload.
        </p>
        <a
          className="App-link"
          href="https://reactjs.org"
          target="_blank"
          rel="noopener noreferrer"
        >
          Learn React
        </a>
      </header>
    </div>
  );
}
export default App;
```

La fonction `App` renvoie une expression JSX. Cette expression définit ce que votre navigateur restitue finalement au DOM.

Tout en bas du fichier `App.js`, l'instruction `export default App` rend notre composant App disponible pour les autres modules.



4. Le langage JSX

Les concepteurs de React ont donc cherché un moyen d'alléger l'écriture et leur choix s'est porté sur l'utilisation de JSX (JavaScript eXtension). JSX est une forme d'écriture des éléments React, plus simple à lire et à écrire que les instructions **React.createElement()**. Cette syntaxe est donc abondamment utilisée dans les programmes React.

Au fur et à mesure, vous verrez que le JSX est un langage très intuitif à utiliser. Voici les deux propriétés de base pour ce qui est des balises utilisées :

- Toute balise commençant par une minuscule (div, span, label, etc.) est réservé aux éléments HTML. Ces éléments sont déclarés par React DOM, et vous obtiendrez une erreur si vous utilisez un élément inexistant.
- Toute balise commençant par une majuscule (Greetings, App, etc.) doit être déclarée explicitement, ce doit donc être un élément du scope courant : fonction déjà déclarée, composant importé d'une bibliothèque ou d'un autre fichier...

4.1. Hello React avec JSX

JSX est donc une forme nouvelle d'écriture des éléments React. Par exemple, voici un para-graphe contenant Hello React dans son texte.

Un paragraphe contenant Hello React en JSX

```
var p = <p>Hello React</p>; // Code JSX
```

Pour afficher ce paragraphe, on utilise l'outil Babel pour interpréter le code JSX et de le transformer en interne en code JavaScript compréhensible par le navigateur. D'autres outils existent, nous utiliserons celui-ci.

La page HTML utilisant Babel s'écrit de la façon suivante, en intégrant notre code JSX.

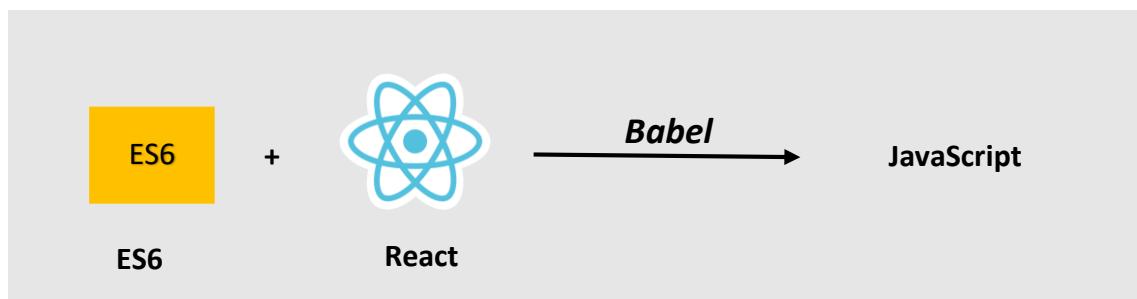
Fichier index.html utilisant Babel afin d'interpréter le code JSX.

```
<html>
  <head>
    <script
      src="https://unpkg.com/react@16/umd/react.development.js"
      crossorigin></script>
    <script src="https://unpkg.com/react-dom@16/umd/react-
      dom.development.js" crossorigin></script>
    <script
      src="https://unpkg.com/@babel/standalone/babel.min.js"></script>
  </head>
  <body>
    <div id="app"></div>
  </body>
```



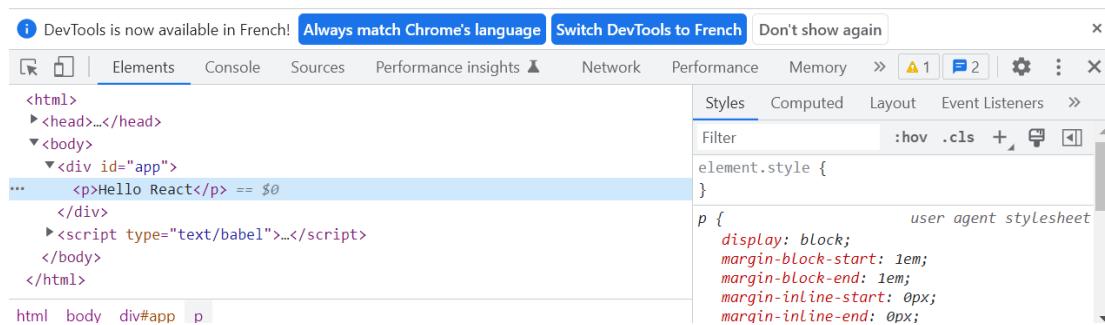
```
<script type="text/babel">
  var p = <p>Hello React</p>; // Code JSX
  console.log(p); // Affichage dans la console de l'élément
  React
  ReactDOM.render(p, document.getElementById("app"));
</script>
</html>
```

On inclut le fichier JavaScript de **Babel** au moyen de la balise `<script src="...">` (cela correspond à l'interpréteur qui traduira le code JSX en code JavaScript), puis on indique quelle partie du code JavaScript est à interpréter par Babel. Pour cela, on inclut l'attribut `type="text/babel"` dans la balise `<script>` contenant notre code JavaScript (et JSX).



Le code JavaScript permettant la création des éléments React est écrit en JSX (et sera traduit en JavaScript pur par Babel), tandis que les éléments React ainsi créés seront insérés dans la page HTML au moyen de l'instruction `ReactDOM.render()`.

L'affichage correspond au paragraphe contenant "Hello React", tandis que l'onglet React de la fenêtre des outils de développement montre l'élément React créé suite à la transformation du code JSX par Babel.



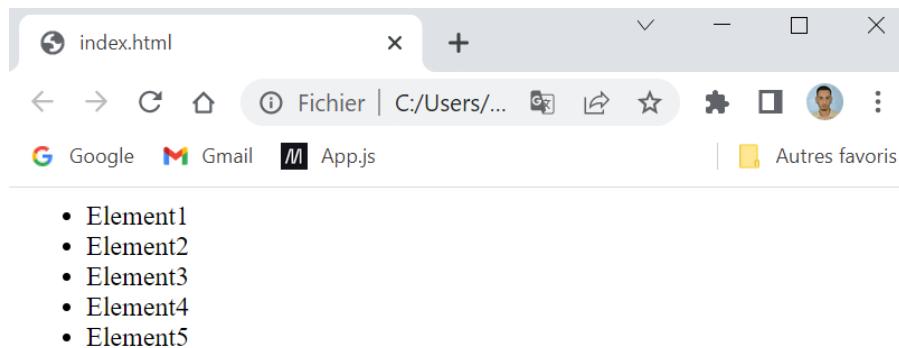


Remarquons que l'inclusion de Babel pour interpréter le code JSX ralentit le programme, vu qu'une étape de traduction est nécessaire avant d'exécuter le code JavaScript. Par conséquent, l'utilisation de Babel ne peut être viable que dans le cadre de l'écriture du programme (en mode développement). Cet outil ne peut pas être utilisé dans le cadre d'un déploiement (mode production). Dans ce dernier cas, on utilisera d'autres outils tels que Webpack pour créer un package plus compact.

Créer une liste de cinq éléments en JSX

```
var liste = <ul>
    <li> Element1 </li>
    <li> Element2 </li>
    <li> Element3 </li>
    <li> Element4 </li>
    <li> Element5 </li>
</ul>;
ReactDOM.render(liste, document.getElementById("app"));
```

Le code JSX est facile à lire et à écrire. Il s'écrit comme du code HTML, mais il est saisi dans la partie réservée au code JavaScript (dans la balise `<script>`). Une même instruction peut s'écrire sur plusieurs lignes et doit obligatoirement commencer par une balise ouvrante et se terminer par balise fermante (ici, `` et ``).



Ajouts d'attributs dans le code JSX

Le code JSX comporte les éléments qui seront affichés dans la page HTML. Ces éléments peuvent avoir des attributs tels que `id`, `style` ou `className` (l'attribut `class` est remplacé par l'attribut `className`).

Commençons par ajouter les attributs `id` et `className`. Pour cela, on définit la classe CSS `red` dans la balise `<style>` de la page.

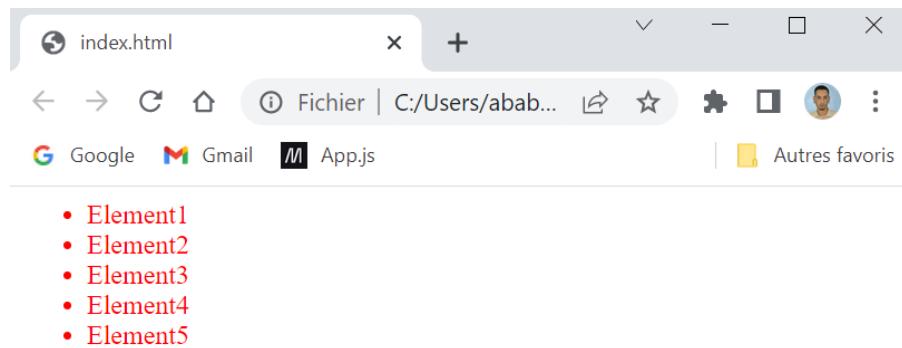
```
<style type="text/css">
    .red {
        color : red;
    }
</style>
```

Définir les attributs `id` et `className` dans le code JSX



```
var liste = <ul id="list1" className="red">
    <li> Element1 </li>
    <li> Element2 </li>
    <li> Element3 </li>
    <li> Element4 </li>
    <li> Element5 </li>
</ul>;
ReactDOM.render(liste, document.getElementById("app"));
```

La liste définie par `` possède bien l'id "list1", tandis que la classe CSS red est bien définie sur la liste (les éléments de liste sont de couleur rouge).



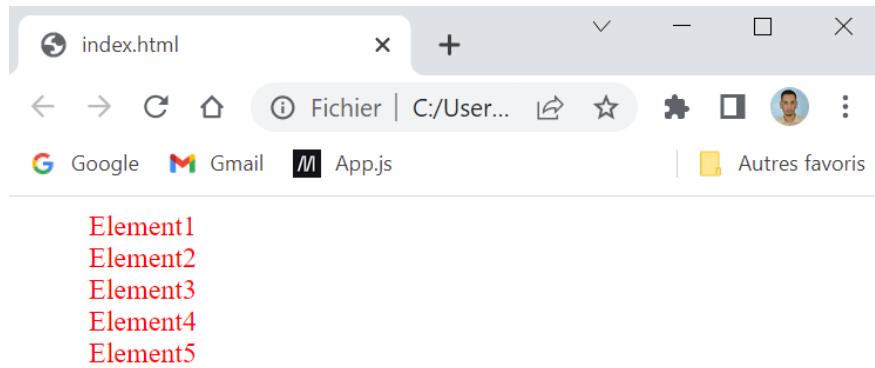
Ajout de l'attribut style en JSX

La syntaxe à utiliser pour insérer l'attribut style est légèrement différente. On souhaite maintenant définir dans le style de la liste (élément ``) la propriété CSS `list-style-type` et lui attribuer la valeur "none", ce qui signifie que les éléments de liste s'affichent sans être précédés par un point. La propriété `color` sera également définie dans le style à la valeur "red" (on suppose que l'on enlève l'attribut `className` utilisé précédemment de façon à ce que la couleur des éléments de liste ne soit pas définie à deux endroits).

Définir l'attribut style dans le code JSX

```
var liste = <ul id="list1" style={{listStyleType:"none",
color:"red"}}>
    <li> Element1 </li>
    <li> Element2 </li>
    <li> Element3 </li>
    <li> Element4 </li>
    <li> Element5 </li>
</ul>;
ReactDOM.render(liste, document.getElementById("app"));
```

La propriété `list-style-type` s'écrit dans le code JavaScript sous la forme `listStyleType`, en remplaçant comme d'habitude chaque tiret et la lettre qui le suit par une majuscule. L'attribut `style` est défini en utilisant les caractères `{}`, soit deux accolades ouvrantes puis deux fermantes. Les accolades extérieures indiquent que l'expression à l'intérieur est une expression JavaScript. De plus, le style doit dans ce cas être défini au moyen d'un objet JavaScript.



The screenshot shows a browser window with the title bar "index.html". The address bar shows "Fichier | C:/User...". Below the address bar are icons for Google, Gmail, and App.js. A sidebar on the right lists "Autres favoris". The main content area displays a list of five items: "Element1", "Element2", "Element3", "Element4", and "Element5", all rendered in red text.

4.2. Utilisation d'instructions JavaScript dans le code JSX

On peut utiliser des instructions JavaScript dans du code JSX, à condition d'entourer les instructions JavaScript avec des accolades. Chaque instruction entourée d'accolades est évaluée par le navigateur, et son résultat est inséré en lieu et place de l'instruction JavaScript évaluée. Ceci permet de créer du code JSX qui s'adapte aux conditions définies dans le programme.

Définir des instructions JavaScript qui calculent le style de l'élément en JSX

```
var color = "red";
var styleListe = { listStyleType:"none", color:color };
var liste = <ul id="list1" style={styleListe}>
    <li> Element1 </li>
    <li> Element2 </li>
    <li> Element3 </li>
    <li> Element4 </li>
    <li> Element5 </li>
</ul>;
ReactDOM.render(liste, document.getElementById("app"));
```

L'instruction JavaScript `{styleListe}` indique de calculer la valeur de l'expression `styleListe`, puis d'affecter cette valeur au style de l'élément dans le code JSX.



Insérer les éléments de liste définis dans un tableau elems

On peut améliorer notre code en insérant les éléments de liste au moyen d'un bloc de code JavaScript. Les éléments de la liste sont placés dans un tableau elems qui est ensuite parcouru par le code JavaScript et JSX.

```
var color = "red";
var styleListe = { listStyleType: "none", color:color };
var elems = ["Element1", "Element2", "Element3", "Element4",
"Element5"];
var liste = <ul id="list1" style={styleListe}>
{
    elems.map(function(elem, index) {
        return <li key={index}>{elem}</li>
    })
}
</ul>;
ReactDOM.render(liste, document.getElementById("app"));
```

Les instructions JavaScript dans un bloc de code JSX doivent être entourées par des accolades, en particulier l'instruction `elems.map()`. De même, dans chaque instruction JSX, toute expression JavaScript doit être encadrée par des accolades, d'où leur présence dans `{styleListe}` et `{index}`.

L'attribut `key` est similaire à celui utilisé dans le précédent chapitre et permet d'éviter un avertissement lors de l'exécution du code (message d'erreur «Each child in an array or iterator should have a unique "key" prop.»).

Utiliser la notation ES6 pour définir la fonction

En utilisant la notation `=>` (disponible dans ES6) pour définir la fonction de callback, on peut écrire plus simplement le code suivant.

```
var color = "red";
var styleListe = { listStyleType: "none", color:color };
var elems = ["Element1", "Element2", "Element3", "Element4",
"Element5"];
var liste = <ul id="list1" style={styleListe}>
{
    elems.map((elem, index) => {
        return <li key={index}>{elem}</li>
    })
}
</ul>;
ReactDOM.render(liste, document.getElementById("app"));
```



Utiliser la notation ES6 sans accolades ni instruction return dans la fonction de callback

Ce qui peut aussi s'écrire de façon encore plus raccourcie (les accolades et l'instruction return dans la fonction de callback ne sont pas nécessaires si une seule instruction est présente dans les accolades).

```
var color = "red";
var styleListe = { listStyleType:"none", color:color };
var elems = ["Element1", "Element2", "Element3", "Element4",
"Element5"];
var liste = <ul id="list1" style={styleListe}>
{
    elems.map((elem, index) =>
        <li key={index}>{elem}</li>
    )
}
</ul>;
ReactDOM.render(liste, document.getElementById("app"));
```



4.3. Créer un élément JSX avec une fonction

L'intérêt de JSX est qu'il permet de créer ses propres éléments HTML, qui seront vus comme des éléments React (écrits en JSX). On va donc ici apprendre à créer l'élément `<ListeElements>` qui représentera la liste `` contenant les éléments ``.

Créer une fonction qui retourne du code JSX

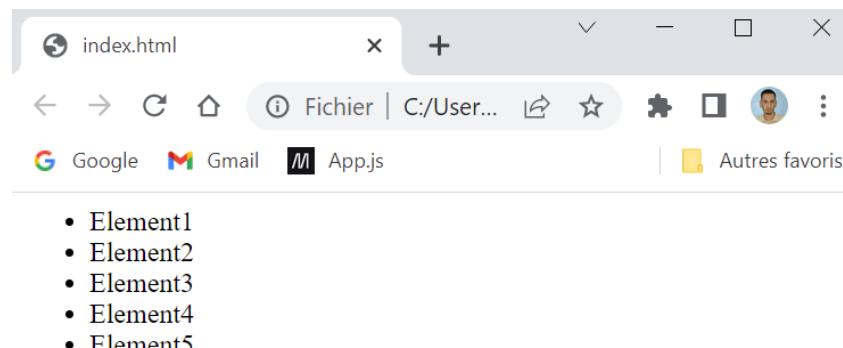
Améliorons le précédent programme pour le transformer en une fonction qui retourne le code JSX nécessaire à la création de la liste. Dans le chapitre précédent, nous avions réalisé une fonction similaire, mais qui retournait la liste au moyen des instructions `React.createElement()`. Ici, nous n'utilisons pas ces instructions mais plutôt le code JSX.

```
var elems = ["Element1", "Element2", "Element3", "Element4",
"Element5"];
var ListeElements = function() {
    return <ul>
        {
            elems.map(function(elem, index) {
                return <li key={index}>{elem}</li>;
            })
        }
    </ul>
}
ReactDOM.render(<ListeElements/>, document.getElementById("app"));
```

La méthode `ReactDOM.render()` prend ici en premier argument un élément React défini en JSX (`<ListeElements/>`). Cet élément correspond à une fonction du même nom qui crée et retourne les éléments React définis également en JSX.

Remarquez qu'un élément défini en JSX, tel que `<ListeElements/>`, doit obligatoirement commencer par une majuscule, sinon React produit une erreur. La fonction associée correspondante doit donc également commencer par une majuscule. Les seuls éléments JSX pouvant commencer par une minuscule sont ceux correspondants à des balises HTML, telles que ``, ``, etc.

On voit dans l'onglet React qu'un élément React nommé `<ListeElements>` a été créé par React, et qu'il contient la liste définie par ``.





Retourner uniquement les éléments dans la fonction (sans l'élément)

```
var elems = ["Element1", "Element2", "Element3", "Element4",
"Element5"];
var ListeElements = function() {
    return elems.map(function(elem, index) {
        return <li key={index}>{elem}</li>;
    })
}
ReactDOM.render(<ul><ListeElements/></ul>,
document.getElementById("app"));
```

La fonction ne retourne plus l'élément , donc les accolades qui servaient à indiquer le code JavaScript à l'intérieur du code JSX ne sont ici plus nécessaires (et si vous les laissez, elles provoquent une erreur).

En revanche, la méthode **ReactDOM.render()** doit retourner le code JSX complet, incluant l'élément .

Même si l'affichage de la liste est identique au précédent, on voit ici que les éléments React et <ListeElements> ont été inversés dans l'arborescence.

Transmettre des attributs dans un élément JSX

On peut transmettre des attributs aux éléments React définis ici en JSX. Par exemple, le tableau **elems** pourrait être transmis dans l'attribut **elems** de l'élément JSX. On peut créer les attributs que l'on souhaite dans un élément JSX, ces attributs seront transmis en paramètres de la fonction de traitement dans l'objet **props**.

```
var elems = ["Element1", "Element2", "Element3", "Element4",
"Element5"];
var ListeElements = function(props) {
    return <ul>
    {
        props.elems.map(function(elem, index) {
            return <li key={index}>{elem}</li>;
        })
    }
    </ul>
}
ReactDOM.render(<ListeElements elems={elems}/>,
document.getElementById("app"));
```

L'attribut **elems** est défini lors de l'écriture de l'élément JSX **<ListeElements elems={elems}/>**. Les attributs d'un élément défini par une fonction sont transmis dans l'objet **props** en paramètres de la fonction. Ainsi, pour accéder à l'attribut **elems** dans la fonction, on utilise **props.elems**.



Transmission de l'attribut style dans l'élément JSX

Transmettons maintenant l'attribut style dans l'élément JSX. Le style indiqué sera affecté aux éléments de la liste.

```
var elems = ["Element1", "Element2", "Element3", "Element4",
"Element5"];
var ListeElements = function(props) {
    return <ul>
        {
            props.elems.map(function(elem, index) {
                return <li key={index}
style={props.style}>{elem}</li>;
            })
        }
    </ul>
}
ReactDOM.render(<ListeElements elems={elems}
style={{color:"red"}}/>, document.getElementById("app"));
```

Le style est indiqué comme d'habitude sous forme d'objet JSON (ici, { color:"red" }), et comme c'est une instruction JavaScript, il faut l'entourer des accolades, d'où les doubles accolades que l'on peut voir ici dans l'élément JSX.

Ce style est récupéré dans la fonction au moyen du paramètre props, et il est accédé à l'aide de props.style dans l'élément JSX définissant chaque élément .

Écriture du programme en déstructurant l'objet props (en ES6)

```
var elems = ["Element1", "Element2", "Element3", "Element4",
"Element5"];
var ListeElements = function({elems, color}) {
    return <ul>
        {
            elems.map(function(elem, index) {
                return <li key={index} style={{color:color}}>
{elem}</li>;
            })
        }
    </ul>
}
ReactDOM.render(<ListeElements elems={elems} color="red"/>,
document.getElementById("app"));
```

On accède maintenant directement aux variables **elems** et **color** précédemment définies comme propriétés dans l'objet **props**.



Créer la liste au moyen de composants

Un élément JSX créé par notre programme est également appelé par un composant React. Ici, le composant est <ListeElements> qui représente la liste des éléments à afficher sous forme de liste.

Toutefois, React encourage d'aller plus loin, et de créer un maximum de composants dans nos programmes React. En effet, le but est d'écrire des composants indépendants qui pourront être utilisés à divers endroits du programme, voire dans d'autres programmes. Cela permet la modularité et la réutilisation du code grâce aux composants.

Dans notre programme, il n'est pas difficile de trouver un nouveau composant à écrire. Il pourrait s'appeler <Element> et correspondrait à un élément de la liste. Cela correspond à la philosophie de React qui consiste à organiser le code en différents composants qui s'utilisent les uns avec les autres. Le composant principal <ListeElements> est donc fait de plusieurs composants <Element>.

Écrivons le composant <Element> utilisé par le composant <ListeElements>.

```
var elems = ["Element1", "Element2", "Element3", "Element4",
"Element5"];
var Element = function({color, elem}) {
    return <li style={{color:color}}>{elem}</li>;
}
var ListeElements = function({elems, color}) {
    return <ul>
    {
        elems.map(function(elem, index) {
            return <Element key={index} elem={elem}
color={color} />
        })
    }
    </ul>
}
ReactDOM.render(<ListeElements elems={elems} color="red" />,
```

Le composant <Element> est lui aussi créé avec une fonction dans laquelle les attributs index, color et elem sont transmis en paramètres dans l'objet props (ici, utilisé sous forme déstruc-turée). L'attribut key est utilisé pour éviter l'erreur classique de React indiquant que cet attribut est obligatoire. Toutefois, il ne sert qu'à mettre une clé différente sur les éléments issus d'une fonction d'itération, donc il est utilisé dans l'écriture de l'élément <Element> (écrit dans une boucle d'itération), mais pas dans les paramètres de la fonction Element().



4.4. Créer un élément JSX avec une classe

Dans la section précédente, nous avons vu comment créer un élément JSX à partir d'une fonction. Mais on sait que l'on peut également créer des éléments React (et JSX) à partir d'une classe dérivant de la classe **React.Component**.

Créons maintenant deux classes correspondant aux deux composants utilisés précédemment (**<Element>** et **<ListeElements>**). Ces deux classes dérivent de la classe **React.Component**.

Créer les classes associées aux composants **<Element>** et **<ListeElements>**

```
var elems = ["Element1", "Element2", "Element3", "Element4",
"Element5"];
class Element extends React.Component {
    constructor(props) {
        super(props);
    }
    render() {
        return <li
style={{color:this.props.color}}>{this.props.elem}</li>
    }
}
class ListeElements extends React.Component {
    constructor(props) {
        super(props);
    }
    render() {
        return <ul>
            {
                this.props.elems.map((elem, index) => {
                    return <Element key={index} elem={elem}
color={this.props.color} />
                })
            }
        </ul>
    }
}
ReactDOM.render(<ListeElements elems={elems} color="red" />,
document.getElementById("app"));
```

L'instruction **ReactDOM.render()** est la même que celle utilisée dans la section précédente. On transmet dans la classe **ListeElements** les attributs **elems** et **color**, utilisés dans la classe via l'objet **this.props** qui les contient.

Remarquez que la fonction de callback utilisée dans la méthode **map()** est définie via la notation ES6 (avec **=>** au lieu de **function**), ceci afin de ne pas perdre la valeur de l'objet **this** dans la fonction de callback (**this.props** peut donc être accessible dans la fonction de call-back afin que sa propriété **color** soit utilisée).



Dans la classe **Element**, remarquez l'utilisation des doubles accolades pour définir le style: la première paire d'accolades est utilisée pour indiquer une instruction JavaScript, la seconde est utilisée pour écrire l'objet sous forme JSON.

On voit sur l'exemple précédent l'utilité de la notation des fonctions en ES6 (avec les caractères `=>`) qui évite de perdre la valeur de `this` dans une fonction de callback. Toutefois, on peut écrire le programme de façon légèrement différente et ne pas perdre la valeur de `this` tout en utilisant le mot-clé `function` pour la fonction de callback.

4.5. Utiliser une fonction ou une classe pour créer les composants en JSX ?

Cette question se pose car les deux manières vues précédemment sont similaires et aboutissent visiblement aux mêmes résultats.

Comme lorsque l'on s'était posé la question au sujet de la création des fonctions ou des classes avec les éléments React (par `React.createElement()` dans le chapitre précédent), la réponse est similaire:

- on utilisera une fonction si l'on n'a pas besoin de créer des propriétés ou des méthodes pour faciliter les traitements;
- on utilisera plutôt une classe si des propriétés ou des méthodes sont nécessaires pour les traitements.

En fait, une propriété très importante d'un composant sera la propriété `state`, permettant de gérer l'état du composant (ceci est étudié dans le chapitre suivant). La règle observée est que si le composant possède un état, on utilisera une classe pour le définir (c'est même dans ce cas obligatoire), sinon une fonction sera suffisante.

4.6. Règles d'écriture du code JSX

Un seul élément parent peut être retourné

Plusieurs éléments JSX de même niveau ne peuvent pas être retournés simultanément, il est obligatoire qu'ils soient encapsulés dans un élément parent, qui sera celui retourné (pour être unique), les autres éléments étant ses enfants. En général on utilise un élément `<div>` englobant l'ensemble, mais React propose aussi d'utiliser un composant `<React.Fragment>` jouant ce rôle.

Remarque : Cette règle est valable également si on utilise la méthode `React.createElement()`, avec laquelle on doit retourner également un seul élément React parent.

Utiliser un fragment avec le composant <React.Fragment>

L'ajout d'un parent, tel qu'un élément `<div>`, fonctionne lorsqu'on souhaite encapsuler plusieurs éléments retournés dans un seul. L'inconvénient de cette solution est que cela ajoute un élément `<div>` supplémentaire dans le code JSX, sans que cela soit vraiment nécessaire pour l'application React.

Pour cela, React propose un composant spécifique appelé `<React.Fragment>` que l'on peut utiliser pour ces cas-là.

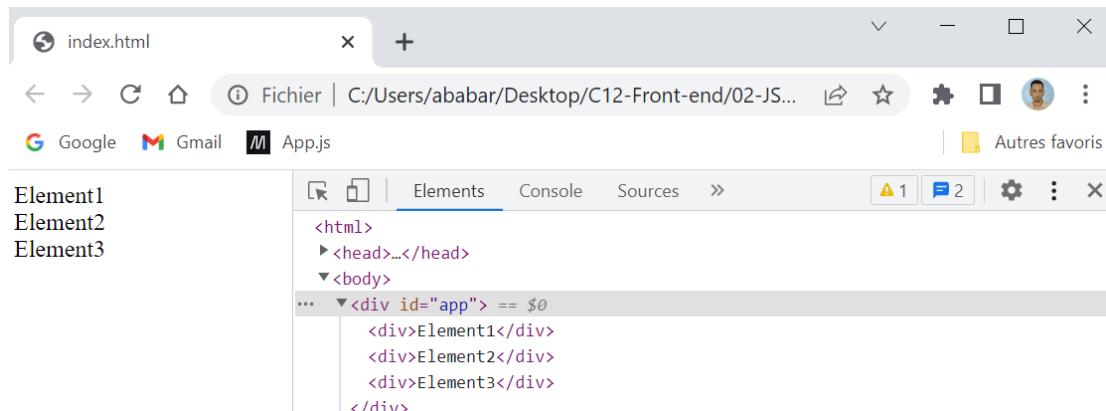


Utilisons le composant **<React.Fragment>** pour englober un ensemble de trois éléments **<div>** sans parents. L'élément **<React.Fragment>** va devenir le parent des trois éléments **<div>**, sans apparaître pour autant dans l'arborescence des éléments React.

```
function ListeElements(props) {
  return <React.Fragment>
    <div>Element1</div>
    <div>Element2</div>
    <div>Element3</div>
  </React.Fragment>
}
ReactDOM.render(<ListeElements />,
  document.getElementById("app"));
```

L'élément **<React.Fragment>** permet de retourner un seul parent, en évitant l'ajout d'un nouvel élément parent non nécessaire.

Remarquez que React ne visualise pas l'élément **<React.Fragment>** dans l'arborescence des éléments React.



Utiliser des parenthèses en début et en fin du code JSX

Lorsqu'on retourne un code JSX sur plusieurs lignes (par exemple un élément **** suivi de plusieurs éléments ****), l'instruction **return** doit comporter à la suite, sur la même ligne, le premier élément JSX retourné, sinon une erreur se produit. Cela oblige à décaler vers la droite le code JSX du premier élément retourné.

Afficher une liste d'éléments sans utiliser des parenthèses

```
function ListeElements(props) {
  return <ul>
    <li>Element1</li>
    <li>Element2</li>
    <li>Element3</li>
    <li>Element4</li>
    <li>Element5</li>
  </ul>
}
ReactDOM.render(<ListeElements />, document.getElementById("app"));
```



Afficher une liste d'éléments en utilisant des parenthèses

```
function ListeElements(props) {
  return (
    <ul>
      <li>Element1</li>
      <li>Element2</li>
      <li>Element3</li>
      <li>Element4</li>
      <li>Element5</li>
    </ul>
  )
}
ReactDOM.render(<ListeElements />, document.getElementById("app"));
```

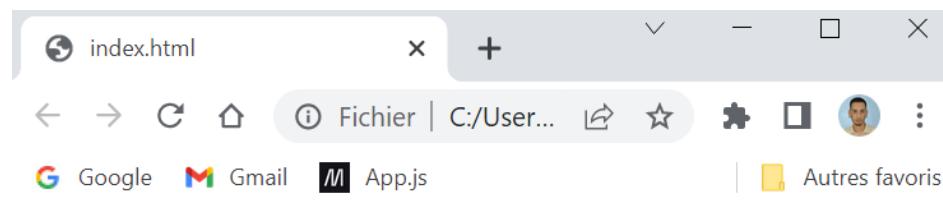
Commentaires dans le code JSX

On utilise les commentaires /* et */ pour indiquer respectivement le début et la fin du code JSX à commenter, à la condition d'entourer l'ensemble avec des accolades { et }.

Les commentaires avec // ne fonctionnent pas avec le code JSX... Par exemple, mettons en commentaires les "Element2" et "Element3" de la liste précédente.

```
function ListeElements(props) {
  return (
    <ul>
      <li>Element1</li>
      {/* <li>Element2</li>
      <li>Element3</li>*/}
      <li>Element4</li>
      <li>Element5</li>
    </ul>
  )
}
ReactDOM.render(<ListeElements />, document.getElementById("app"));
```

Dans les deux exemples de programmes, les éléments mis en commentaires n'apparaissent pas à l'affichage.



- Element1
- Element4
- Element5

Les éléments "Element2" et "Element3" ne sont pas affichés.

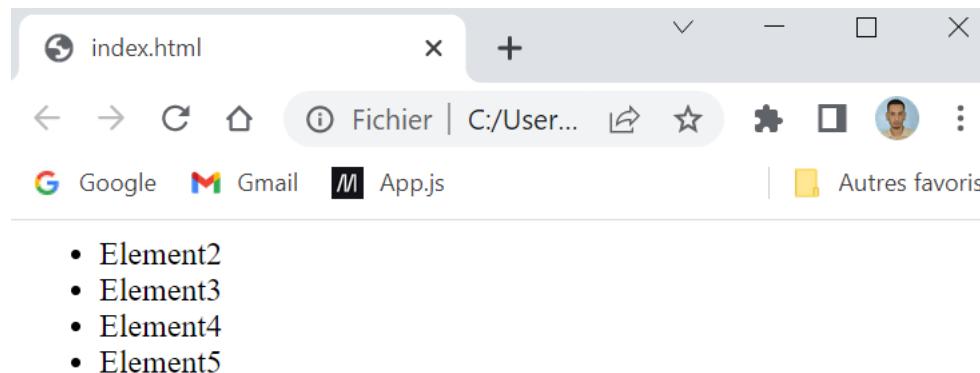


Utiliser des expressions conditionnelles dans le code JSX retourné

Il est possible d'utiliser des expressions conditionnelles avec ? et : dans le code JSX, à condition d'entourer l'ensemble avec des accolades { et } (car cela correspond à une expression JavaScript qui est évaluée).

Supposons que l'on ait un attribut dans le composant <ListeElements> permettant d'indiquer si l'on doit cacher ou pas le premier élément de la liste. L'attribut se nommera hideFirstItem et vaut true si l'on doit cacher cet élément, false sinon.

```
function ListeElements(props) {
  return (
    <ul>
      { props.hideFirstItem ? null : <li>Element1</li> }
      <li>Element2</li>
      <li>Element3</li>
      <li>Element4</li>
      <li>Element5</li>
    </ul>
  )
}
ReactDOM.render(<ListeElements hideFirstItem={true} />,
document.getElementById("app"));
```



Le premier élément de la liste n'apparaît pas.



5. Composant en React

5.1. Introduction :

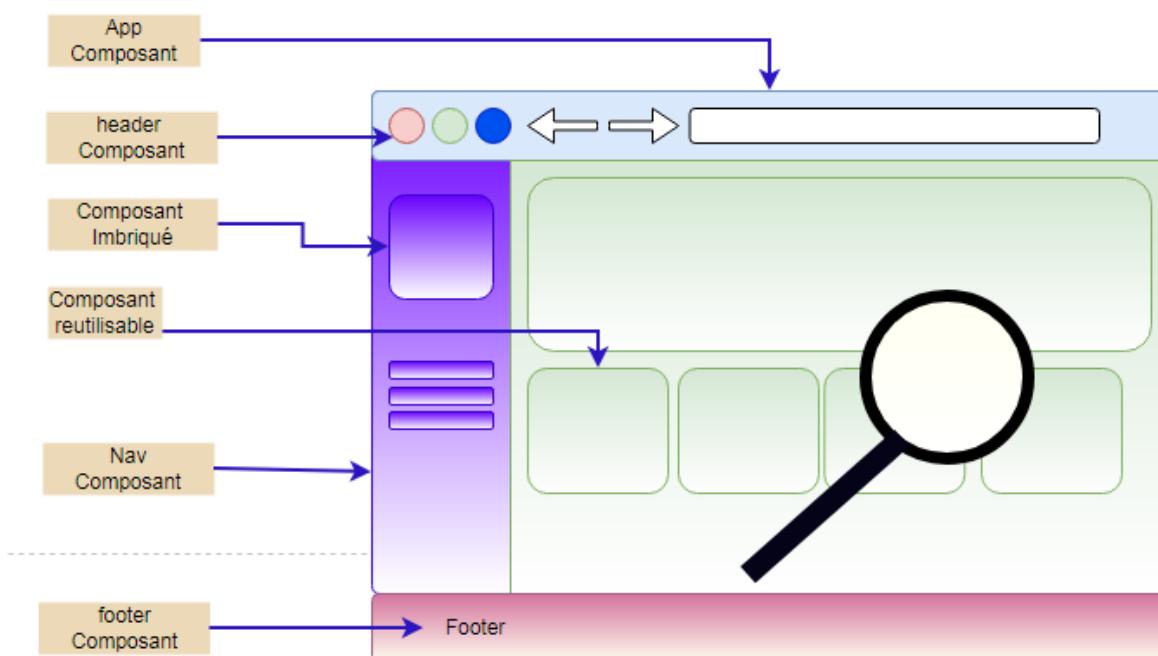
Chaque application React est constitué d'un ensemble de composants.

Avant de construire une Application React on doit identifier les parties qui constituent des composants réutilisables.

Avec les composants vous allez pouvoir avec un même ensemble de code regrouper la structure les styles et les comportements, et vous pouvez utiliser vos composants autant que vous voulez en vous faisant gagner beaucoup du temps.

Dans cette séance vous allez découvrir comment créer des éléments web avec des composants React

On va voir aussi les spécificités du JSX, notamment comment combiner plusieurs composants et comment utiliser des expressions java script directement dans les composants



5.2. Prise en main création d'une application React

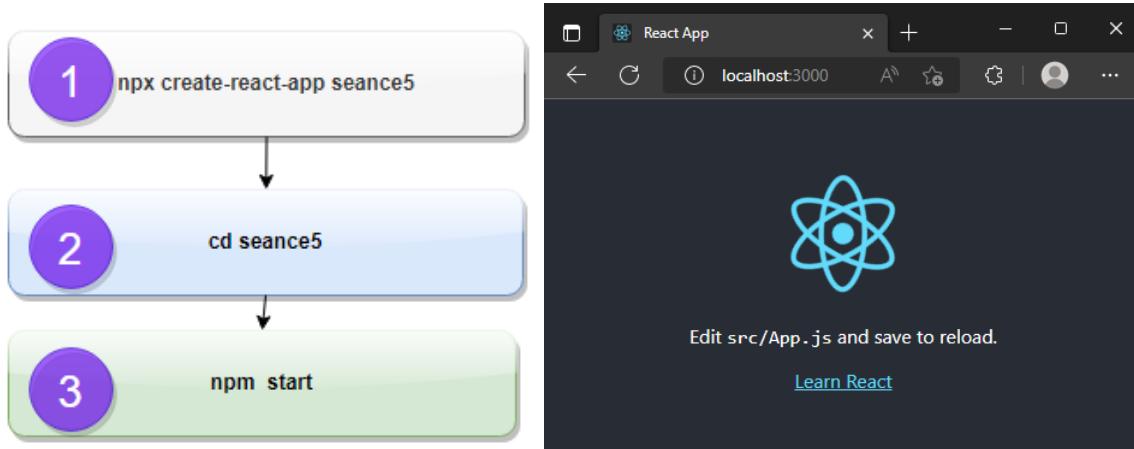
Après avoir créer un projet React voir séance 3

Création de l'application seance5

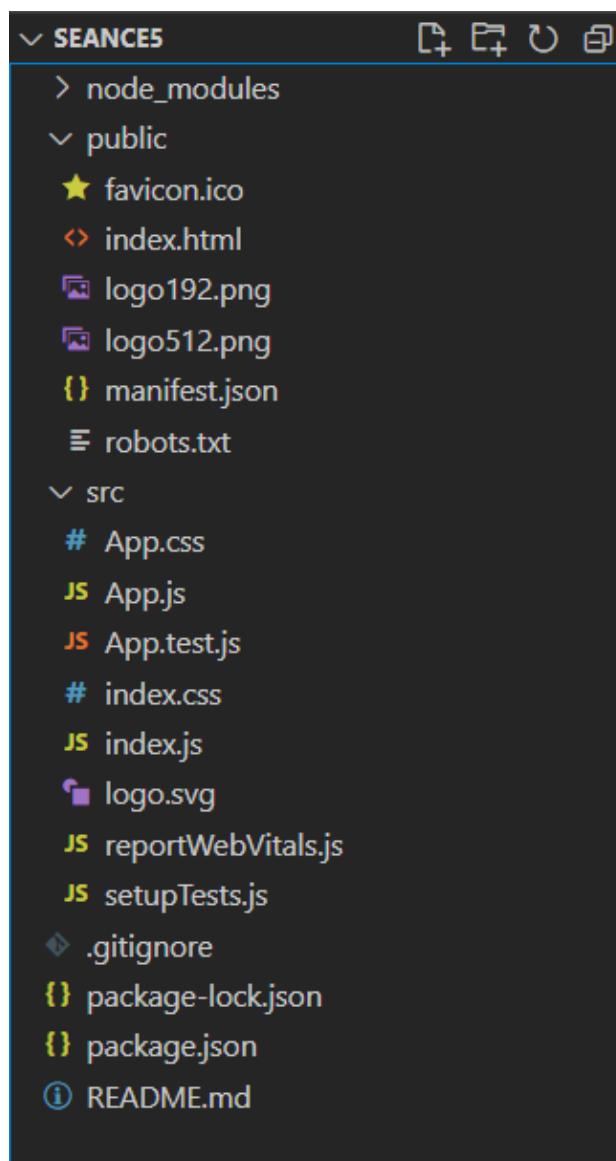
```
npx create-react-app seance5
Creating a new React app in [REDACTED] seance5.
Installing packages. This might take a couple of minutes.
Installing react, react-dom, and react-scripts with cra-template...
[REDACTED] | idealTree:webpack-dev-server: sill fetch manifest emojis-list@^3.0.0
```



Etapes à suivre



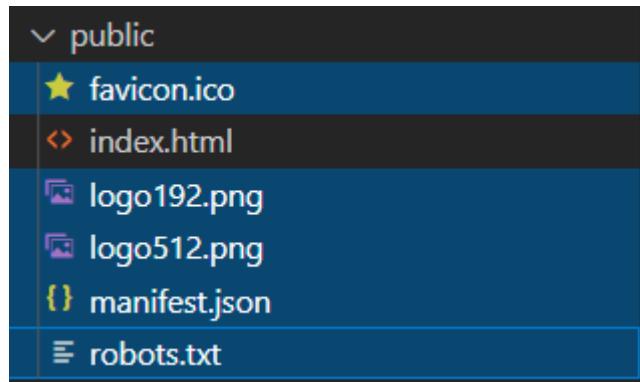
Structure d'une application React :



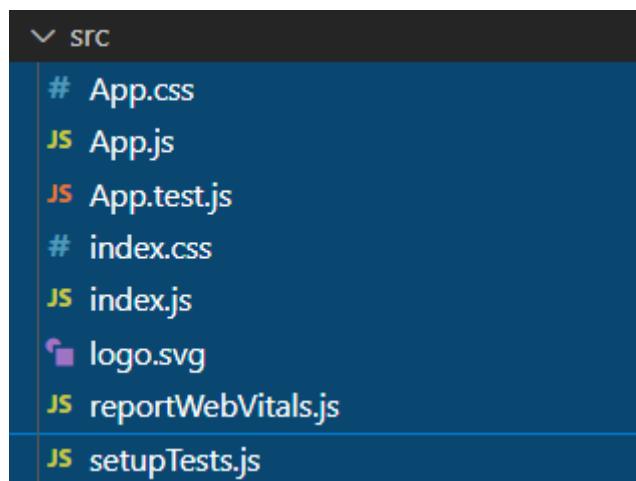


Pour vocation pédagogique, et pour démarrer un projet React avec le minimum de fichiers. dans un premier temps on va :

- Supprimer tous les fichiers qui se trouvent dans le dossier src
- Supprimer tous les fichiers qui se trouvent dans le dossier public sauf le fichier index.html

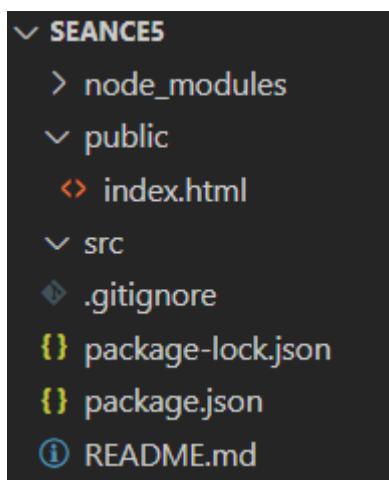


- Supprimer les fichiers sélectionnés de dossier public



- Supprimer tous les fichiers du dossier src

La structure de projet après modification





Dans le dossier src ajouter le fichier index.js

Voir contenu du fichier index.js

```
// 1) importer React et ReactDOM
import React from 'react';
import ReactDOM from 'react-dom/client'

//2) faire référence à div id=root de index.html

const element=document.getElementById("root");
//3) prendre le contrôle de l'element par React

const root=ReactDOM.createRoot(element)
// 4) Creation d'un composant (component)

//un composant est une fonction qui retourne jsx

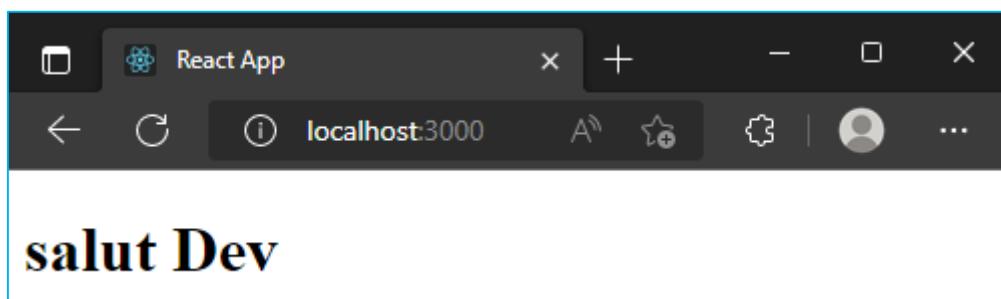
function App(){

    return(
        <h1>salut Dev</h1>
    )
}

// 5) afficher le composant dans le browser

root.render(<App/>)
```

Le rendu sera :

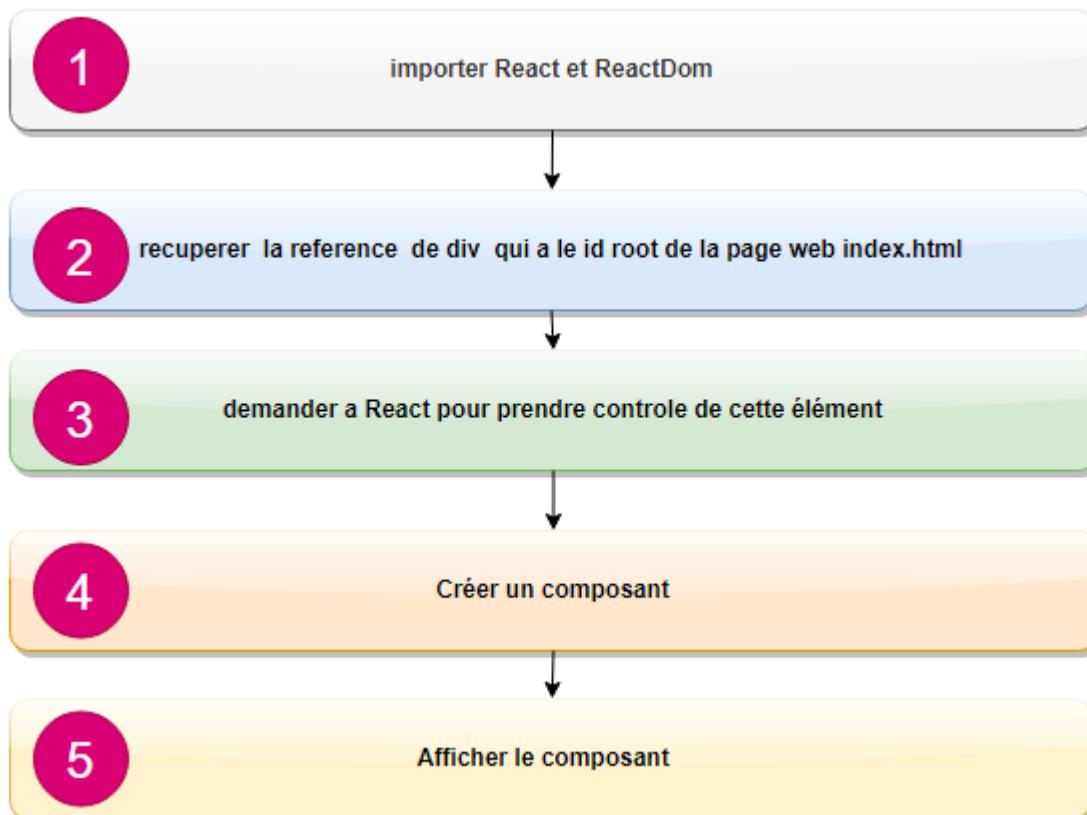




5.3. Explication des étapes pour créer un composant :

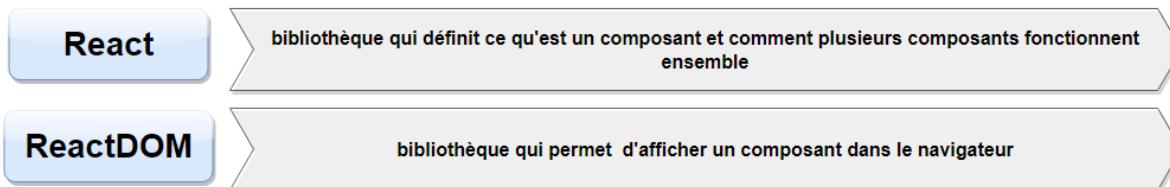
Un composant est **fonction** qui retourne de JSX

Ou une **classe** qui hérite de la classe **React.Component** possédant la fonction render qui retourne du JSX



Etape 1 :

```
// 1) importer React et ReactDOM
import React from 'react';
import ReactDOM from 'react-dom/client'
```



Le package react-dom/client fournit des méthodes spécifiques au client utilisées pour initialiser une application sur le client. La plupart de vos composants ne devraient pas avoir besoin d'utiliser ce module.



Etape 2 :

```
//2) faire référence à div id='root' de index.html

const element=document.getElementById("root");
```

on crée une référence vers l'élément div id='root' de la page public/index.html (la page index.html contient <div id= 'root'></div>)

C'est dans cette élément que React va injecter le code HTML

Etape 3 :

```
//3) prendre le contrôle de l'element par React

const root=ReactDOM.createRoot(element)
```

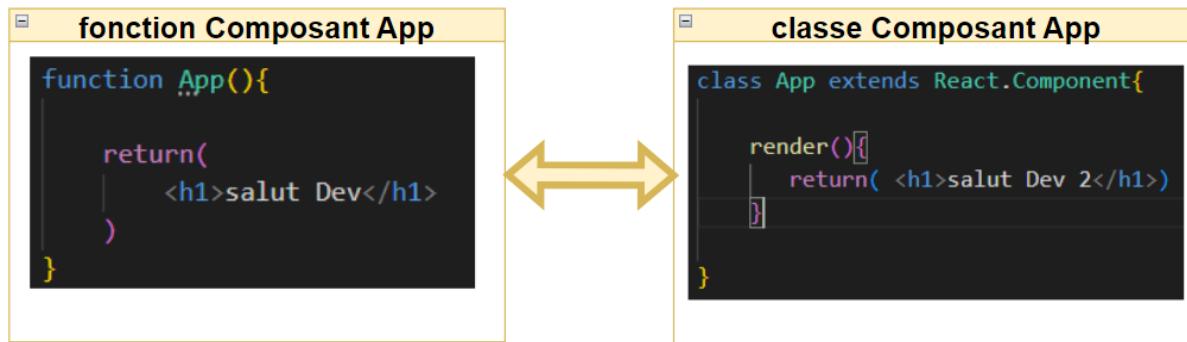
On prend le contrôle de l'élément div id='root' de la page public/index.html par React

Créez une racine React pour le conteneur fourni et renvoyez la racine. La racine peut être utilisée pour restituer un élément React dans le DOM avec render :

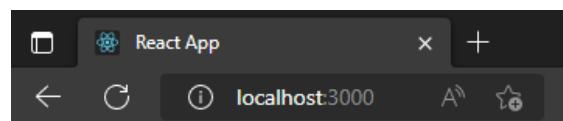
```
root.render(<App/>)
```

Etape 4

On crée un composant qui s'appelle App, comme on avait expliqué avant un composant est une fonction qui retourne de JSX, il peut être aussi une classe qui hérite de la classe React.Component possédant la fonction render qui retourne du JSX,



Remarque : Quand vous remplacez la fonction App par la classe App vous allez avoir le résultat



salut Dev 2

Pour une bonne organisation de l'application chaque composant doit être dans fichier js portant le même nom de composant.



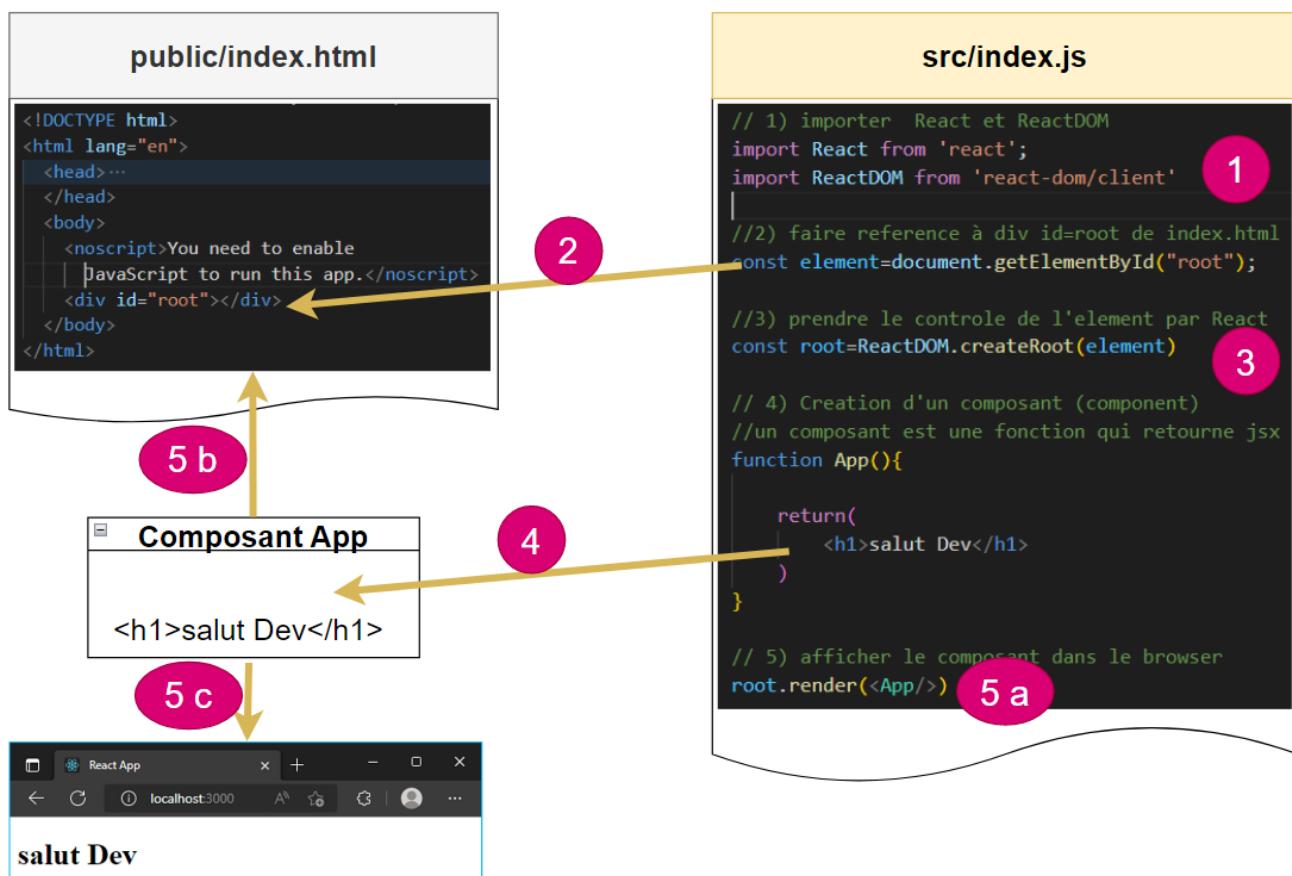
Etape 5

```
// 5) afficher le composant dans le browser
root.render(<App/>)
```

on passe App composant comme JSX élément à la méthode root.render()

permet d'afficher le contenu du composant App dans le <div id='root'></div> de la page idx.html

Comment ça marche



Comment se fait l'interprétation de JSX

Utilisation de l'outil : babeljs.io/repl

Outil pour vous montrer en quoi votre jsx est transformé :

The screenshot shows the Babel REPL interface with the following details:

- SETTINGS** (Left sidebar):
 - Evaluate
 - Line Wrap
 - Prettify
 - File Size
 - Time Travel
- Code Area** (Center):


```
1 <h1>salut Dev</h1>
```
- Output Area** (Right):


```
1 "use strict";
2
3 /*#__PURE__*/
4 React.createElement("h1", null, "salut Dev");
```
- Header** (Top):

BABEL Docs Setup Try it out Videos Blog Search Donate Team GitHub



```


- un
- deux
- trois


```

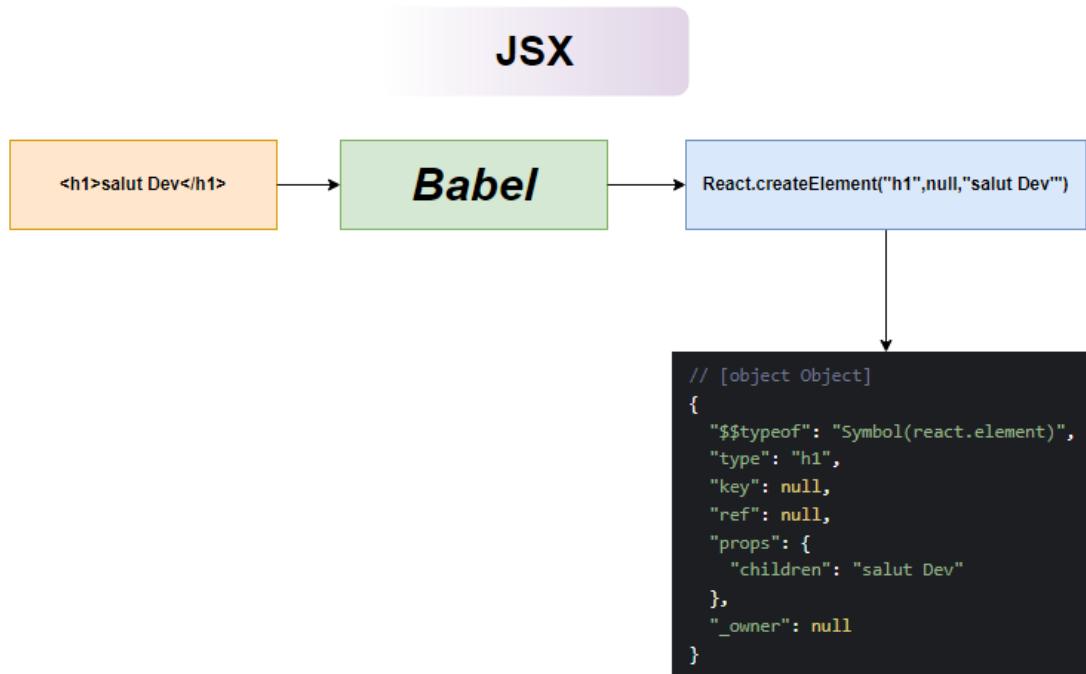
```

1 "use strict";
2
3 /*#__PURE__*/
4 React.createElement("div", {
5   id: "root"
6 }, /*#__PURE__*/React.createElement("ul", null,
/*#__PURE__*/React.createElement("li", null, "un"),
/*#__PURE__*/React.createElement("li", null, "deux"),
/*#__PURE__*/React.createElement("li", null, "trois")));

```

Comme vous remarquer le code JSX est très facile et compréhensible alors que le code java script correspondant est difficile pour le comprendre beaucoup d'imbrications d'appel de createElement.

JSX facilite beaucoup le travail de développeur.



Le code JSX <h1> salut Dev </h1> est transformé par la Bibliothèque Babel en code javascript

React.createElement("h1",null,"salut Dev") qui retourne un objet java script voir figure

Cet objet contient toutes les informations nécessaires à React pour connaitre l'élément à afficher dans le navigateur.

Par exemple 'type' :'h1' précise le type de l'élément

'props' :{'children' :salut Dev'} précise le texte contenu dans h1



```
<h1>saut Dev</h1>
```

cette écriture ne fait pas automatiquement d'affichage sur le navigateur

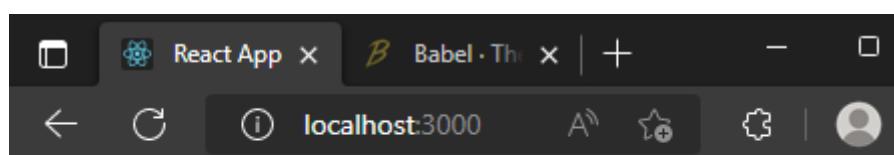
mais cette écriture a de sens pour React
React crée un élément h1

cette écriture doit être retournée par un composant pour que React puisse l'utiliser et l'afficher

5.4. Ajouter de variables et expression java script dans JSX

```
function App(){
  const nom='RAMI'
  return(
    <h1>salut {nom}</h1>
  )
}
```

Le rendu



salut RAMI

Si on veut utiliser une variable ou expression js dans JSX on utilise les accolades {}, ici on passe la variable nom dans JSX

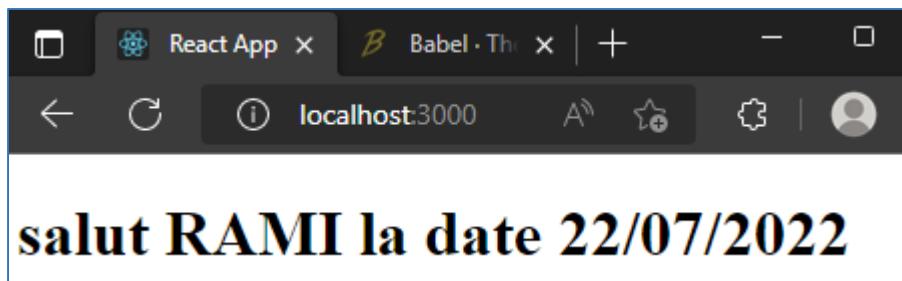
```
<h1>salut {nom}</h1>
```



Exemple 2 :

```
function App(){
  const nom='RAMI'
  const time=new Date().toLocaleDateString();
  return(
    <h1>salut {nom} la date {time}</h1>
  )
}
```

Rendu



Exemple 2 bis expression js dans jsx

```
function App(){
  const nom='RAMI'
  return(
    <h1>salut {nom} la date {new Date().toLocaleDateString()}</h1>
  )
}
```

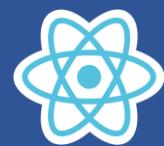
Exemple 3 :

Afficher la remise

Si le nombre d'article acheté est supérieur à 5 remise 2% si non remise 0%

Solution :

```
function App(){
  const nbArticle=6
  let remise=0
  if(nbArticle>=5){
    remise=2
  }
  return(<div>
    <h2> Remise </h2>
    <p> votre remise est: {remise} %</p>
    </div>
  )
}
```



Rendu



votre remise est: 2 %

Exercice d'entraînement :

On souhaite afficher le nom, le prenom et l'âge dans l'élément p du composant suivant :

Pour calculer l'âge utiliser la fonction getAge()

```
import React from 'react';
import ReactDOM from 'react-dom/client';
const element=document.getElementById("root");
const root=ReactDOM.createRoot(element)
//fonction getAge reçoi la date en format DD/MM/YYYY
function getAge(dateNaissance){
    let from = dateNaissance.split("/");
    let birthdateTimeStamp = new Date(from[2], from[1] - 1, from[0]);
    let cur = new Date();
    let diff = cur - birthdateTimeStamp;
    // difference en milliseconds
    let currentAge = Math.floor(diff/31557600000);
    // Division par 1000*60*60*24*365.25
    return currentAge;
}
function App(){
    const nom='RAMI'
    const prenom='AHMED'
    const dateNaissance='23/03/2000'
    return(<div>
        <h2> Informations </h2>
        <p> -----</p>
        </div>
    )
}
root.render(<App/>)
```

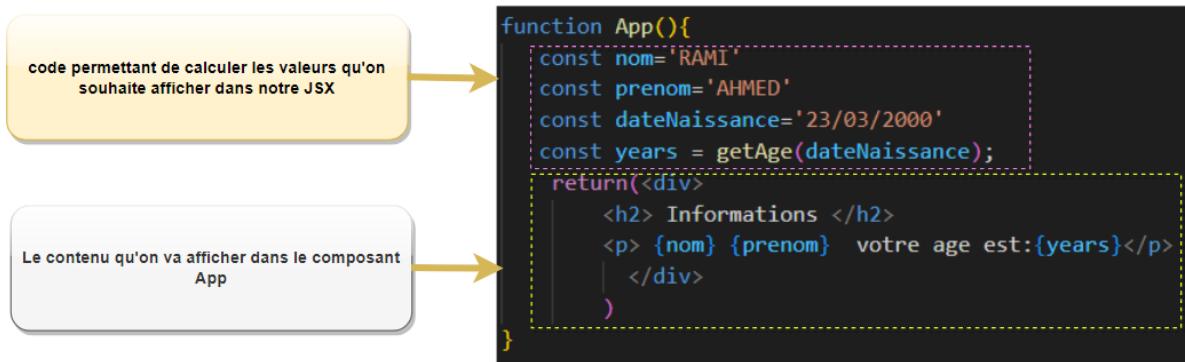
Le rendu

Informations

RAMI AHMED votre age est:22



Solution :



Reprendre l'exercice précédent en utilisant classe composante

Solution :

```

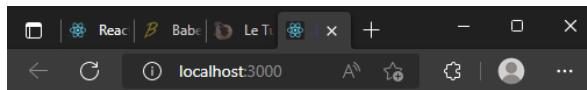
import React from "react";
function getAge(dateNaissance){
  let from = dateNaissance.split("/");
  let birthdateTimeStamp = new Date(from[2], from[1] - 1, from[0]);
  let cur = new Date();
  let diff = cur - birthdateTimeStamp;
  // difference en milliseconds
  let currentAge = Math.floor(diff/31557600000);
  // Division par 1000*60*60*24*365.25
  return currentAge;
}
export default class App extends React.Component{
  constructor(props) {
    super(props)
    this.nom="RAMI";
    this.prenom="AHMED";
    this.dateNaissance="10/03/2000"
  }
  render(){
    return(<div>
      <h2> Informations </h2>
      <p> {this.nom} {this.prenom} votre age est
      {getAge(this.dateNaissance)} ans</p>
    </div>
  )
}
  
```



6. Manipuler les propriétés et gérer les états

6.1. Introduction :

Dans la séance précédente, on avait dit que les composants sont réutilisables, si on est dans la situation où on souhaite afficher le même composant plusieurs fois mais avec des informations différentes.



Salut Rami Ahmed

Salut Kamali Ali

Salut Fahmi Khalid

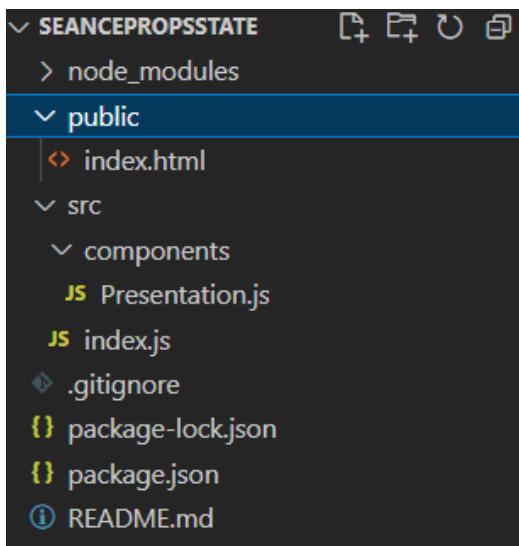
React utilise un Object props qui va nous permettre de passer les informations nécessaires au composant, ce qui rend le composant dynamique.

6.2. Manipulation des props dans un composant créé via une fonction

Pour illustrer ce besoin on va créer un composant Presentation, dans un premier temps on va créer le composant via une fonction puis dans un deuxième temps on va créer le composant via une classe.

Pour ce faire, il est souhaitable de créer le composant dans un fichier js Presentation.js qui se trouvera dans le dossier components.

- Créer le projet **seancePropsState**
- Créer le dossier components
- Créer le fichier Presentation.js dans le dossier src/components
- Ajouter le code suivant dans Presentation.js





Presentation.js

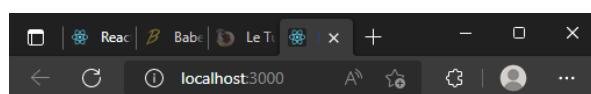
```
import React from 'react'
export default function Presentation(props){
  console.log(props)
  return (
    <div>
      <h2>Salut {props.nom} {props.prenom}</h2>
      <hr/>
    </div>
  )
}
```

Le fichier index.js

```
import React from 'react';
import ReactDOM from 'react-dom/client';
import Presentation from './components/Presentation';

const element=document.getElementById("root")
const root=ReactDOM.createRoot(element)
function App(){
  return(
    <div>
      <Presentation nom="Rami" prenom="Ahmed"/>
      <Presentation nom="Kamali" prenom="Ali" />
      <Presentation nom="Fahmi" prenom="Khalid"/>
    </div>
  )
}
root.render(<App/>)
```

Le rendu



Salut Rami Ahmed

Salut Kamali Ali

Salut Fahmi Khalid

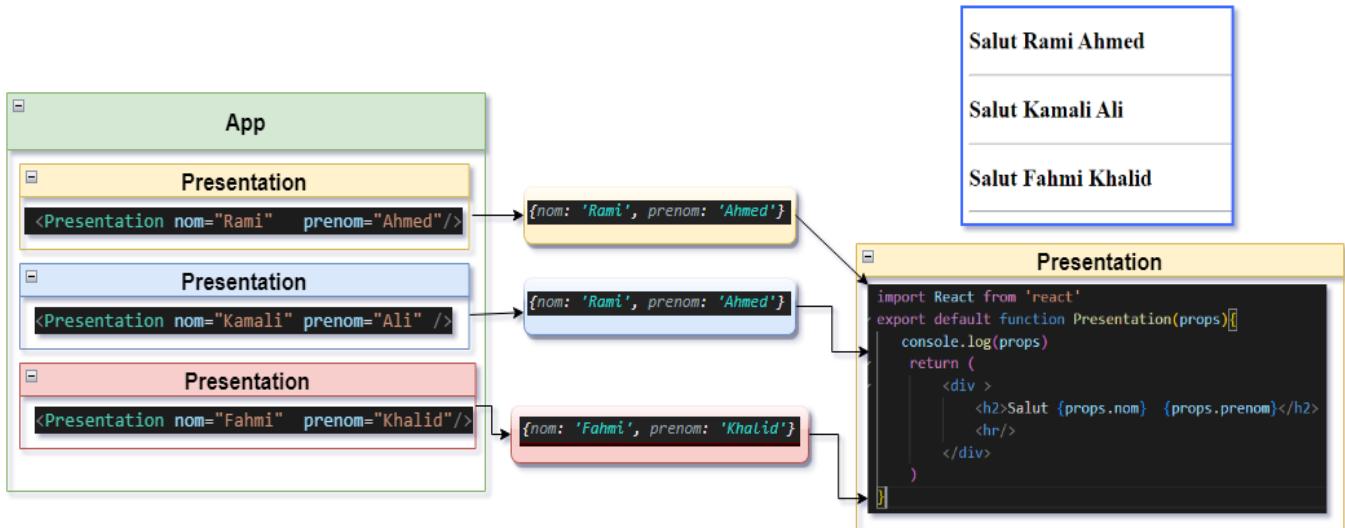
Le rendu de la console

```
▶ {nom: 'Rami', prenom: 'Ahmed'}
▶ {nom: 'Kamali', prenom: 'Ali'}
▶ {nom: 'Fahmi', prenom: 'Khalid'}
```



On remarque que **props** c'est un objet JavaScript

L'objet props est immuable c.-à-d. on ne peut pas changer les propriétés
Cette écriture n'est pas permise : **props.nom='Jamil'**

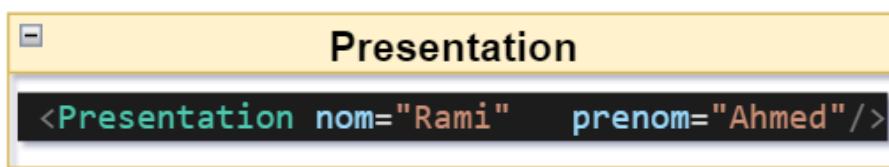


L'écriture :

```
<Presentation nom="Rami" prenom="Ahmed"/>
```

Fait appel à la fonction **Presentation** en passant en argument l'objet **props** qui fait référence à l'objet **{nom: 'Rami', prenom: 'Ahmed'}**, cet objet est créé à partir des attributs nom, prenom de l'élément **Presentation**.

Il est recommandé de choisir le nom props



La fonction **Presentation** retourne le JSX qui affiche

Salut Rami Ahmed



```
import React from 'react'
export default function Presentation(props){
    console.log(props)
    return (
        <div>
            <h2>Salut {props.nom} {props.prenom}</h2>
            <hr/>
        </div>
    )
}
```

Cette écriture permet de récupérer les propriétés nom et prenom de l'objet props passé en argument de la fonction Presentation

```
<h2>Salut {props.nom} {props.prenom}</h2>
```

Remarque : On peut passer dans l'objet props :

- une valeur
- un objet
- une liste

Passage d'un objet dans props

```
function App(){
    let personne1={nom:"Rami",prenom:"Ahmed"}
    let personne2={nom:"Kamali",prenom:"Ali"}
    let personne3={nom:"Fahmi",prenom:"Khalid"}
    return(
        <div>
            <Presentation personne={personne1}/>
            <Presentation personne={personne2} />
            <Presentation personne={personne3} />
        </div>
    )
}
```

Remarque : on peut passer directement l'objet

```
<Presentation personne={{nom:"Fahmi",prenom:"Khalid"}} />
```

Le rendu de console

```
▼ {personne: {...}} ⓘ
  ▶ personne: {nom: 'Rami', prenom: 'Ahmed'}
  ▶ [[Prototype]]: Object
  ▶ {personne: ...}
  ▶ {personne: ...}
```

Cette fois ci props c'est l'objet **{personne :{nom :’Rami’,prenom :’Ahmed’}}**



Récupération des données passées dans l'objet props

```
import React from 'react'
export default function Presentation(props){
    console.log(props)
    return (
        <div>
            <h2>Salut {props.personne.nom} {props.personne.prenom}</h2>
            <hr/>
        </div>
    )
}
```

On peut utiliser le destructeur objet JavaScript ES6 voir cours séance1

```
import React from 'react'
export default function Presentation(props){
    console.log(props)
    const {nom,prenom}=props.personne
    return (
        <div>
            <h2>Salut {nom} {prenom}</h2>
            <hr/>
        </div>
    )
}
```

Passer un Array dans props

```
import React from 'react';
import ReactDOM from 'react-dom/client'
import Presentation from './components/Presentation';

const element=document.getElementById("root")
const root=ReactDOM.createRoot(element)
function App(){
    let personne1={nom:"Rami",prenom:"Ahmed"}
    let diplomes=["Bac","Licence","Master"]
    return(
        <div>
            <Presentation personne={personne1} diplomes={diplomes} />
        </div>
    )
}
root.render(<App/>)
```



Composant Presentation

```
import React from 'react'
export default function Presentation(props){
  console.log(props)
  const {nom,prenom}=props.personne
  return (
    <div>
      <h2>Salut {nom} {prenom}</h2>
      <hr/>
      <h3>Diplomes</h3>
      <p>{props.diplomes}</p>
    </div>
  )
}
```

Le rendu console

```
▼ {personne: {...}, diplomes: Array(3)} ⓘ
  ► diplomes: (3) ['Bac', 'Licence', 'Master']
  ► personne: {nom: 'Rami', prenom: 'Ahmed'}
  ► [[Prototype]]: Object
```

Maintenant props c'est l'objet JavaScript

```
{personne :{ nom:"Rami",prenom:"Ahmed"},diplomes :["Bac","Licence","Master"]}
```

Salut Rami Ahmed

Diplomes

Bac Licence Master

Comme vous remarquez React fait la concaténation des éléments de la liste diplomes dans une chaîne de caractère {props.diplomes}, il y aura une séance ultérieure qui traite la manipulation des éléments d'un Array en utilisant la méthode map.



6.3. Manipulation des props dans un composant créé via une classe

On utilise le même projet

On utilisera le composant Salutation créé via classe
pour ce faire créer un fichier Salutation.js dans le dossier components

Salutation.js

```
import React from 'react'
export default class Salutation extends React.Component{
render(){
  console.log(this.props)
  return (
    <div>
      <h2>Salut {this.props.nom} {this.props.prenom}</h2>
      <hr/>
    </div>
  )
}
```

Vous allez remarquer que l'objet props est accessible directement dans l'objet Salutation, mais il ajouter le mot clé **this** : **{this.props.nom}**

Index.js

```
import React from 'react';
import ReactDOM from 'react-dom/client';
import Salutation from './components/Salutation';
const element=document.getElementById("root")
const root=ReactDOM.createRoot(element)
function App(){
  return(
    <div>
      <Salutation nom="Rami" prenom="Ahmed" />
      <Salutation nom="Kamali" prenom="Ali" />
      <Salutation nom="fahmi" prenom="Khalid" />

    </div>
  )
}
root.render(<App/>)
```

L'écriture :

```
<Salutation nom="Rami" prenom="Ahmed"/>
```

Crée une instance de la classe Salutation apré la création, la méthode render est exécutée, elle retourne du JSX qui affiche

Salut Rami Ahmed

On peut passer dans l'objet props une valeur, un objet et une liste



Passage d'un objet dans props

Index.js

```
import React from 'react';
import ReactDOM from 'react-dom/client'
import Salutation from './components/Salutation';
const element=document.getElementById("root")
const root=ReactDOM.createRoot(element)
function App(){
    let personne1={nom:"Rami",prenom:"Ahmed"}
    let personne2={nom:"Kamali",prenom:"Ali"}
    let personne3={nom:"Fahmi",prenom:"Khalid"}

    return(
        <div>
            <Salutation personne={personne1} />
            <Salutation personne={personne2} />
            <Salutation personne={personne3} />
        </div>
    )
}
root.render(<App/>)
```

Salutation.js

```
import React from "react";

export default class Salutation extends React.Component {
    render() {
        console.log(this.props);
        return (
            <div>
                <h2>
                    Salut {this.props.personne.nom} {this.props.personne.prenom}
                </h2>
                <hr />
            </div>
        );
    }
}
```



Passage d'un Array dans props

Index.js

```
import React from "react";
import ReactDOM from "react-dom/client";
import Salutation from "./components/Salutation";
const element = document.getElementById("root");
const root = ReactDOM.createRoot(element);
function App() {
  let personne1 = { nom: "Rami", prenom: "Ahmed" };
  let diplomes = ["Bac", "Licence", "Master"];
  return (
    <div>
      <Salutation personne={personne1} diplomes={diplomes} />
    </div>
  );
}
root.render(<App />);
```

Salutation.js

```
import React from "react";

export default class Salutation extends React.Component {
  render() {
    console.log(this.props);
    return (
      <div>
        <h2>
          Salut {this.props.personne.nom} {this.props.personne.prenom}
        </h2>
        <h3>Diplomes</h3>
        <p>{this.props.diplomes}</p>
        <hr />
      </div>
    );
  }
}
```



6.4. Passer dynamique contenu a un composant

Retournant a notre exemple Presentation

```
import React from "react";
import ReactDOM from "react-dom/client";
import Presentation from "./components/Presentation";
const element = document.getElementById("root");
const root = ReactDOM.createRoot(element);
function App() {
  return (
    <div>
      <Presentation nom="Rami" prenom="Ahmed">
        <p>ce ci est un children props</p>
      </Presentation>
      <Presentation nom="Kamali" prenom="Ali">
        <button>quitter</button>
      </Presentation>
    </div>
  );
}
root.render(<App />);
```

Dans ce cas l'élément Presentation contient un élément enfant entre la balise ouvrante et fermante.

Les éléments enfants sont différents, un paragraphe et un bouton

```
import React from 'react'
export default function Presentation(props){
  console.log(props)
  return (
    <div >
      <h2>Salut {props.nom} {props.prenom}</h2>
      {props.children}
      <hr/>

    </div>
  )
}
```

Pour récupérer l'élément enfant en utilise la propriété children : **{props.children}**

Le rendu

Salut Rami Ahmed

ce ci est un children props

Salut Kamali Ali

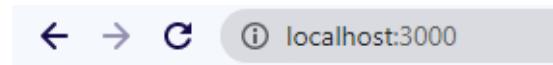
quitter



6.5. Utiliser l'objet state pour mettre à jour un composant créer par classe

State est un objet pouvant être mis à jour qui peut être utilisé pour contenir les données et contrôler le comportement du composant. Seuls les composants de classe peuvent avoir un état, pas les composants fonctionnels. Lorsque l'état est modifié, React restitue automatiquement le composant qui provoque une mise à jour d'affichage.

Pour expliquer l'objet state, considérons que nous souhaitons écrire un composant Message simple qui permet d'afficher le rendu suivant

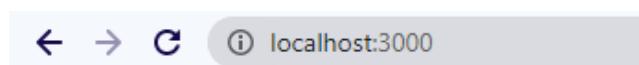


Bien venu visiteur

inscription

Si l'utilisateur clique sur le bouton inscription

Le rendu devient



votre inscription est effectuée

merci

Vous allez remarquer que après le click sur le bouton le message Bien Venu visiteur change pour devenir votre inscription est effectuée, le texte de bouton inscription change pour devenir merci.

D'où il faut avoir deux variables message et btnMessage dont les valeurs vont être modifiées après la clique sur le bouton, cette modification doit provoquer un rafraîchissement d'affichage.

Pour illustrer ces notions de gestion d'état, on va expliquer ensemble le code de cet exemple.

Etapes à suivre pour créer l'application Message.

Ajouter un fichier Message.js dans le dossier src/components.

Message.js

```
import React from 'react'
export default class Message extends React.Component{
  constructor(){
    super()
    this.state={message:"Bien venu visiteur",btnMessage:"inscription"}
  }
}
```



```
inscription(){
this.setState({message:"votre inscription est effectuée",btnMessage:"merci"})
}
render(){
return(<div>
<h2>{this.state.message}</h2>
<button onClick={()=>this.inscription()}>{this.state.btnMessage}</button>
</div>
)
}
```

index.js

```
import React from "react";
import ReactDOM from "react-dom/client";
import Message from "./components/Message";
const element = document.getElementById("root");
const root = ReactDOM.createRoot(element);
function App() {
  return (
    <div>
      <Message/>
    </div>
  );
}
root.render(<App />);
```

Explication du code de la classe Message

```
constructor(){
super()
this.state={message:"Bien venu visiteur",btnMessage:"inscription"}
}
```

Dans le constructeur on fait appel au constructeur de la classe mère React.Component par

super()

création de l'objet state qui contient les deux propriétés message et btnMessage ,les deux propriétés sont initialisées,

this.state={message:"Bien venu visiteur",btnMessage:"inscription"}

on remarque que state est un objet de la classe Message
mise à jour de state

```
inscription(){
  this.setState({message:"votre inscription est effectuée",btnMessage:"merci"})
}
```

Comme vous remarquez, la modification des valeurs des propriétés se fait via la fonction **this.setState()**,



La fonction `setState` modifie les propriétés puis provoque un appel de la méthode `render()`, c'est ainsi que l'interface utilisateur est mise à jour avec les nouvelles valeurs de `{this.state.message}` et `{this.state.btnMessage}`

Remarque Très importante

Il ne faut pas modifier l'état state directement

~~`this.state.message= "message_1"`~~

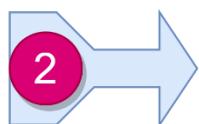
~~`this.state.btnMessage="message_2"`~~

en effet cette modification directe change l'état mais elle ne provoque pas l'appel de la méthode `render()`, d'où l'interface utilisateur n'affiche pas les nouvelles valeurs des propriétés de l'état .

regles de state



utilisé dans un classe composant



Attention de confondre props et state



state est un java script objet contenant des données relative au composant



la mise à jour de 'state' sur le composant provoque le rendu instantané du composant



state doit être initialisé quand le composant est créé



state doit être mise à jour uniquement par la fonction '`setState`'



Il faut impérativement faire attention à la dernière règle, la mise à jour de l'état (state) doit être fait uniquement via la fonction setState

```
import React from 'react'
export default class Message extends React.Component{ 1
    2
    constructor(props){
        super(props)
        5 this.state={message:"Bien venu visiteur",btnMessage:"inscription"} 3
    }

    inscription(){
        6     this.setState({message:"votre inscription est effectuée",btnMessage:"merci"})
    }

    render(){4}

    return(<div>
        <h2>{this.state.message}</h2>
        <button onClick={()=>this.inscription()}>{this.state.btnMessage}</button>
    </div>
)
}
```



8.1. Gestion des événements

La gestion des événements avec des éléments React est très similaire à la gestion des événements sur les éléments DOM. Il existe quelques différences de syntaxe :

- Les événements React sont nommés en camelCase, plutôt qu'en minuscules.
- Avec JSX, vous transmettez une fonction en tant que callBack fonction ou arrow fonction, plutôt qu'une chaîne.

Exemples :

onClick

HTML	<pre><button onclick="ajouter()"> Ajouter article </button></pre>
REACT	<pre>//callBack function <button onClick={ajouter}> Ajouter article </button></pre>
REACT	<pre>//Arrow function <button onClick={ajouter}> Ajouter article </button></pre>

onSubmit

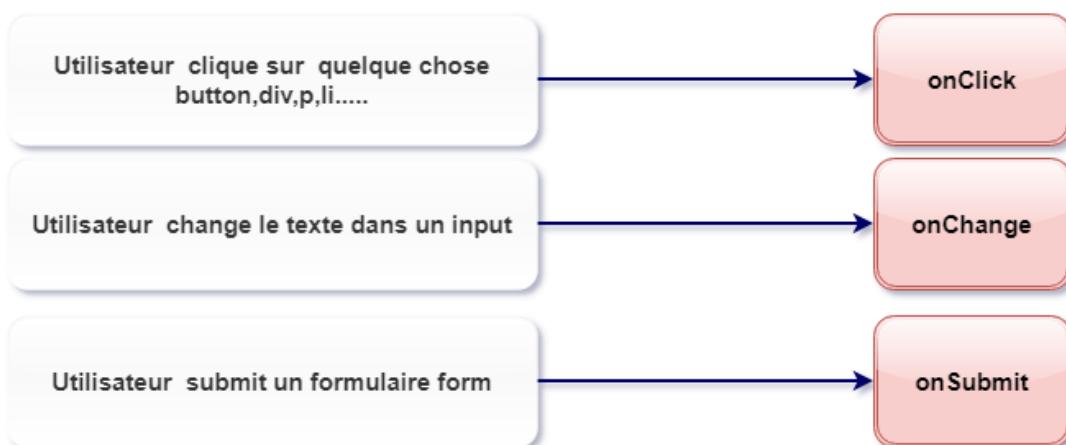
HTML	<pre><form onsubmit="console.log('You clicked submit.'); return false"> <button type="submit">Submit</button> </form></pre>
REACT	<pre>//arrow fonction import React from 'react' export default function Form() { function handleSubmit(e) { e.preventDefault();console.log('You clicked submit.'); } return (//arrow function <form onSubmit={(event)=>handleSubmit(event)}> <button type="submit">Submit</button> </form>); }</pre>



REACT	<pre>//callBack function import React from 'react' export default function Form() { function handleSubmit(e) { e.preventDefault();console.log('You clicked submit.'); } return (//callBack function <form onSubmit={handleSubmit}> <button type="submit">Submit</button> </form>); }</pre>
-------	--

onChange

HTML	<input type="text" onchange="changeNom()"></input>
REACT	<input type="text" onChange={changeNom}></input>



On peut utiliser les callBack et les Arrow fonctions pour gérer les évènements

Exercice d'application 1

Inscription

Nom: ahmed

Prenom:

Afficher

nom:RAMI prenom:AHMED



Quand l'utilisateur clique sur Afficher s'affiche le message contenant le nom et le prenom

Solution :

```
import React, { useState } from "react";
export default function Inscription(){
const [nom, setNom]=useState()
const [prenom, setPrenom]=useState()
const [information, setInformation]=useState()

function envoyer(){
setInformation(`nom:${nom} prenom:${prenom}`)
}

return(
<div>
  <h2></h2>
  <div>
    <label>Nom:</label><input type="text"
onChange={(e)=>{setNom(e.target.value.toUpperCase())}}/>
  </div>
  <div>
    <label>Prenom:</label><input type="text"
onChange={(e)=>{setPrenom(e.target.value.toUpperCase())}}/>
  </div>
  <button onClick={envoyer}>Afficher</button>
  <p>{information}</p>
</div>
)
}
```

Remarque importante : Si on remplace cette écriture

<button onClick={envoyer}>Afficher</button>

Par

<button onClick={envoyer()}>Afficher</button>

On aura le message d'erreur

► **Uncaught Error: Too many re-renders. React limits the number of renders to prevent an infinite loop.**

Car envoyer est exécuter après chaque render de composant, de plus la méthode envoyer contient le code suivant :

```
setInformation(`nom:${nom} prenom:${prenom}`)
```

qui a son tour déclenche un render à cause de Hook **useState** **setInformation**



Exercice Application 2 :

On peut valider les éléments input facilement en utilisant le Système gestion d'état de Rect.

Essayons ça en créant un simple validateur de mot de passe. Ce dernier va être un input text qui nécessite l'utilisateur d'entrer un mot de passe qui contient au moins 4 caractères.

Si l'utilisateur entre un mot de passe moins de 4 caractères. un message d'erreur va être affiché «**Password doit avoir au moins de 4 caractères**»

Entrer votre password:
Password doit avoir au moins t 4 caractères

Entrer votre password:

Solution :

Avec fonctionnel composant

```
import React, { useState } from 'react'
export default function Validator(){

    const [password, setPassword]=useState('')

    return(
        <div>
            <div>
                <label>Entrer votre password:</label>
                <input type="password"
                    value={password}
                    onChange={(event)=>setPassword(event.target.value)}
                />
            </div>
            {password.length<4?"Password doit avoir au moins t 4 caractères":""}
        </div>
    )
}
```



Avec classe composant

```
import React, { useState } from 'react'
export default class Validator extends React.Component{



    constructor (){
        super();
        this.state={password: ''}
    }
    render(){

        return(
            <div>
                <div>
                    <label>Entrer votre password:</label>
                    <input type="password"
                        value={this.state.password}
                        onChange={(event)=>
                            this.setState({password:event.target.value})}
                        />
                    </div>
                    {this.state.password.length<4?"Password doit avoir au moins t 4
caractères":""}
                </div>
            )
    }
}
```



8.2. Communication inter-composant (envoi, réception de données):

Exemple :

Composant App

composant ChercheBar

Entrer le mot clé de recherche:

le type :LEGUME

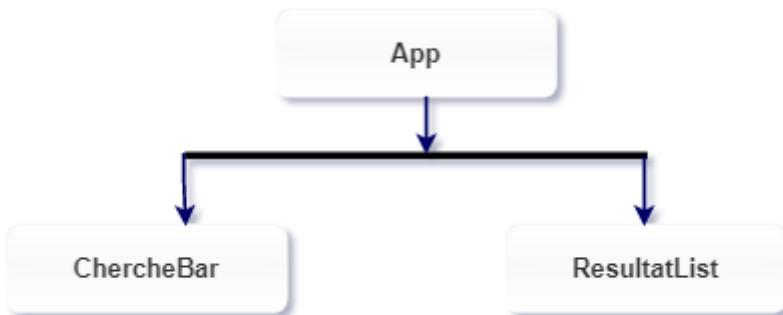
composant ResultatList

- tomate
- carotte
- pomme de terre
- navet
- poivron

On suppose que nous disposons de trois composants :

- App
- ChercherBar
- ResultatList

Les composants ChercherBar et ResultatList sont imbriqués dans le composant App





On dispose d'une liste

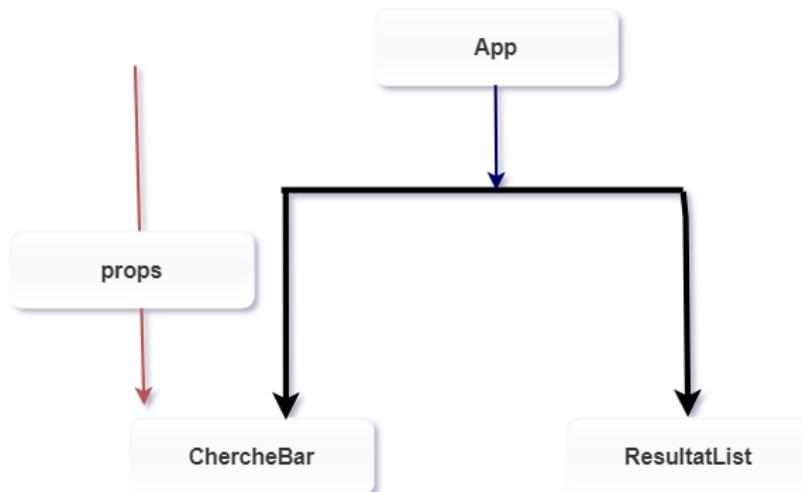
```
const list=[  
{nom:"banane",type:"fruit"},  
{nom:"orange",type:"fruit"},  
{nom:"pomme",type:"fruit"},  
{nom:"raisins",type:"fruit"},  
{nom:"kiwi",type:"fruit"},  
{nom:"tomate",type:"legume"},  
{nom:"carotte",type:"legume"},  
{nom:"pomme de terre",type:"legume"},  
{nom:"navet",type:"legume"},  
{nom:"poivron",type:"legume"}  
]
```

Pour des raisons pédagogiques les éléments de la liste sont de type "legume" ou "fruit"

Cette liste est une constante dans le composant App

L'utilisateur saisi le type dans le composant ChercheBar, la soumission du formulaire déclenche un callBack de la fonction qui aura la valeur saisie dans Cherchebar comme argument et qui va filtrer la liste selon le type saisie, puis le composant ResultatList affiche les éléments filtrés.

8.2.1. Communication de parent vers enfant



Déjà dans la séance 6 on a vu comment passer les informations à un composant via les props

<Presentation nom="Rami" prenom="Ahmed"/>



Code App.js

```
import React, { useState } from 'react'
import ChercheBar from './components/ChercheBar';
import ResultatList from './components/ResultatList';
const list=[
{nom:"banane",type:"fruit"},  

{nom:"orange",type:"fruit"},  

{nom:"pomme",type:"fruit"},  

{nom:"raisins",type:"fruit"},  

{nom:"kiwi",type:"fruit"},  

{nom:"tomate",type:"legume"},  

{nom:"carotte",type:"legume"},  

{nom:"pomme de terre",type:"legume"},  

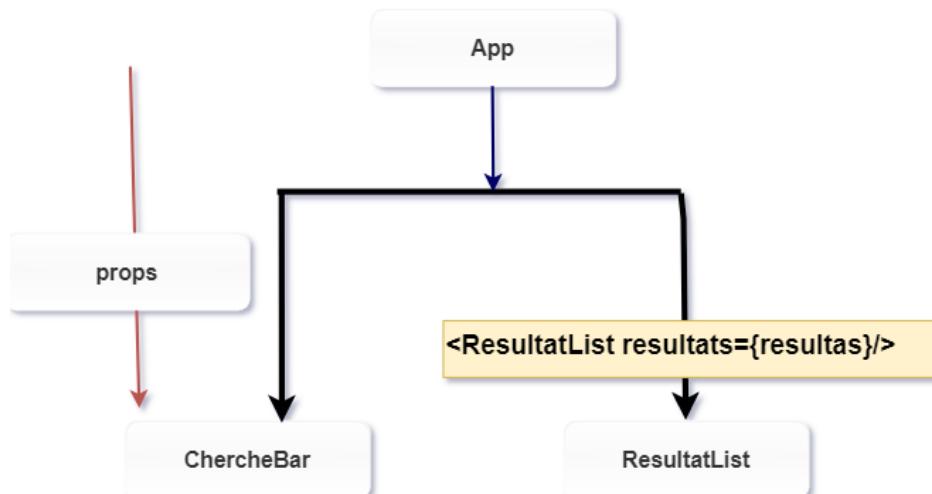
{nom:"navet",type:"legume"},  

{nom:"poivron",type:"legume"}  

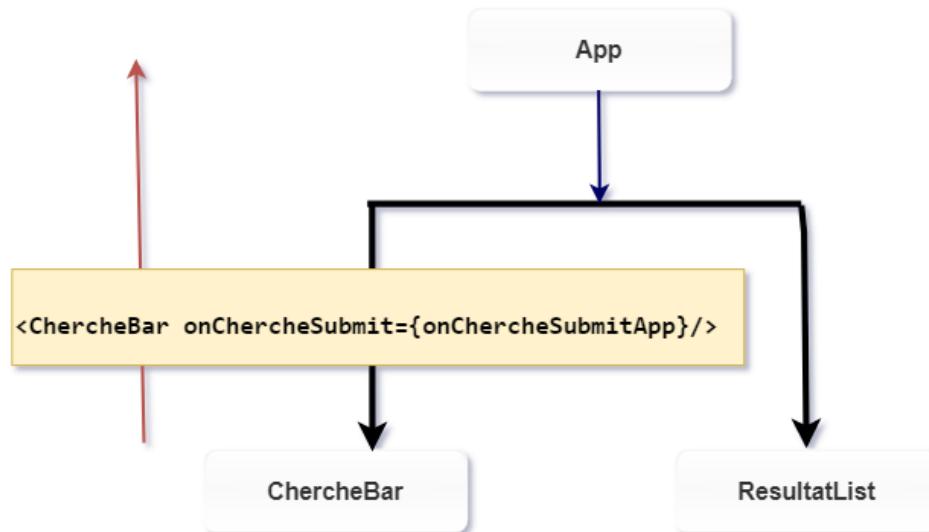
]  
  
export default function App() {  
  const [type,setType]=useState('')  
  const [resultas,setResultas]=useState([])  
  function onChercheSubmitApp(type){  
    setType(type)  
    setResultas(list.filter((item)=>item.type.toUpperCase()==type))  
  }  
  return (  
    <div className='App'>  
      <h1>Composant App</h1>  
      <ChercheBar onChercheSubmit={onChercheSubmitApp}/>  
      <div>  
        <p>Le type:<span style={{color:'rgb(36, 44, 33)",fontWeight:"bold"}}>{type}</span></p>  
      </div>  
      <ResultatList resultats={resultas}/>  
    </div>  
  );  
}
```

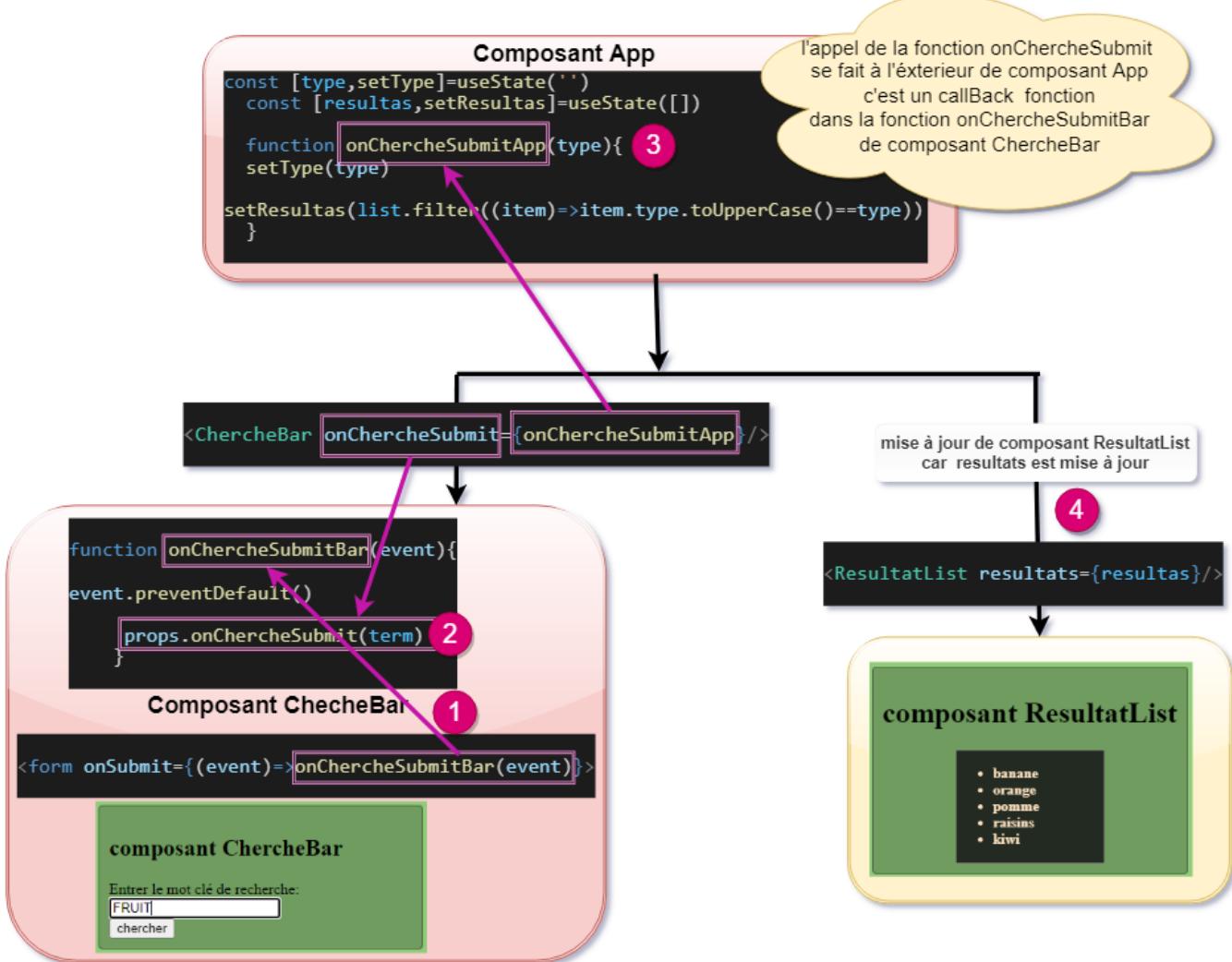
```
<ResultatList resultats={resultas}/>
```

Le composant ResultatList reçoit l'information resultats via le props resultats



8.2.2. Communication d'enfant vers parent





Explication du code de composant App

```
<ChercheBar onChercheSubmit={onChercheSubmitApp}/>
```

On a passé au composant ChercheBar le props `onChercheSubmit={onChercheSubmitApp}`

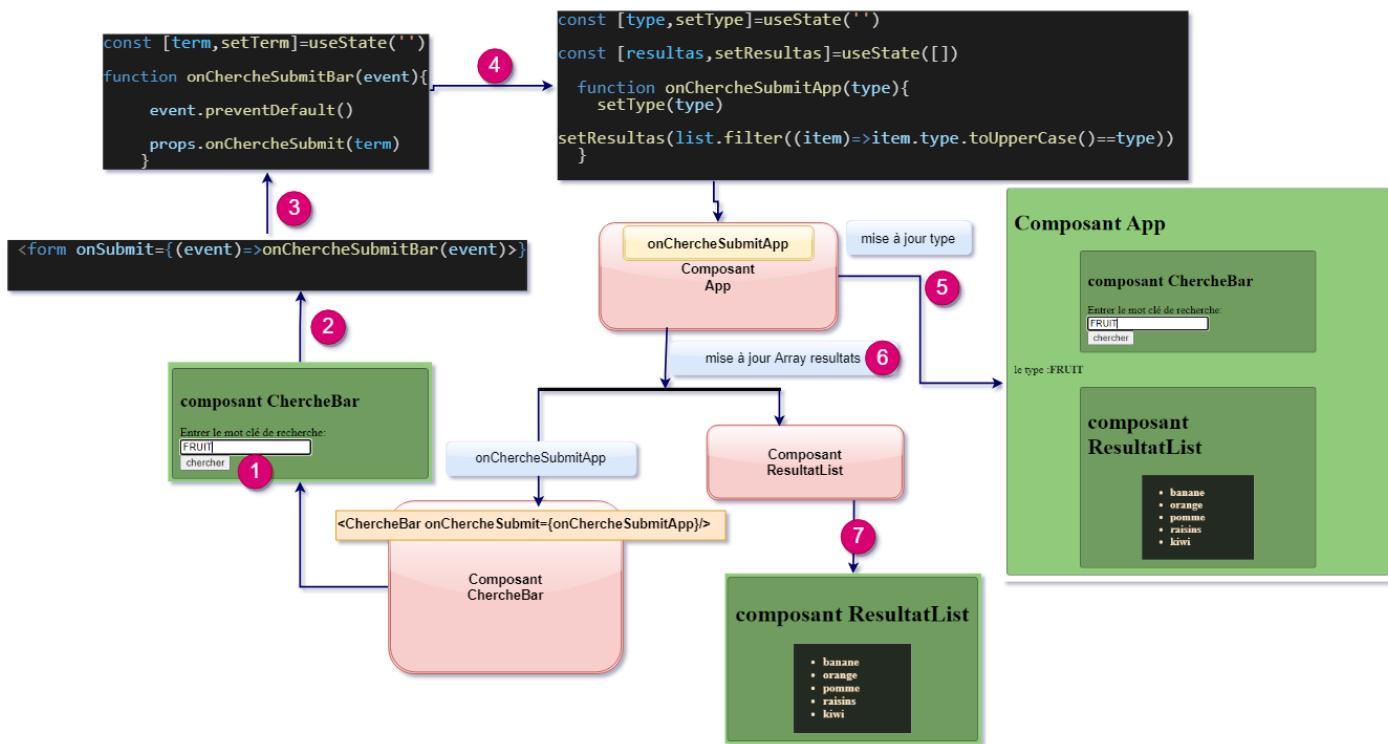
Le props `onChercheSubmit` contient la callback fonction `onChercheSubmitApp`

```
const [type, setType] = useState('')
const [resultas, setResultas] = useState([])
function onChercheSubmitApp(type){
    setType(type)
    setResultas(list.filter((item)=>item.type.toUpperCase()==type))
}
```

`onChercheSubmitApp` met à jour le type par la valeur de l'argument, puis filtre la liste sur le type dont la valeur est passée en argument.

`onChercheSubmitApp` sera exécuté quand l'utilisateur submit le formulaire

Le composant ResultatList est ré-rendu avec la liste résultats filtrée explication par schéma, les numéros marquent l'ordre chronologique des étapes d'exécutions



Le code de ChercheBar.js

```
import React, { useState } from "react";
export default function ChercheBar(props) {
  const [term, setTerm] = useState('')
  function onChercheSubmitBar(event){
    event.preventDefault()
    props.onChercheSubmit(term)
  }
  return (
    <div className="Child">
      <form onSubmit={(event)=>onChercheSubmitBar(event)}>
        <h2>composant ChercheBar</h2>
        <div>
          <label>Entrer le mot clé de recherche:</label>
          <input type="text" value={term}>
          <input type="text" value={term} onChange={(event)=>setTerm(event.target.value.toUpperCase())}>
        </div>
        <button type="submit">chercher</button>
      </form>
    </div>
  );
}
```



Le code de ResultatList.js

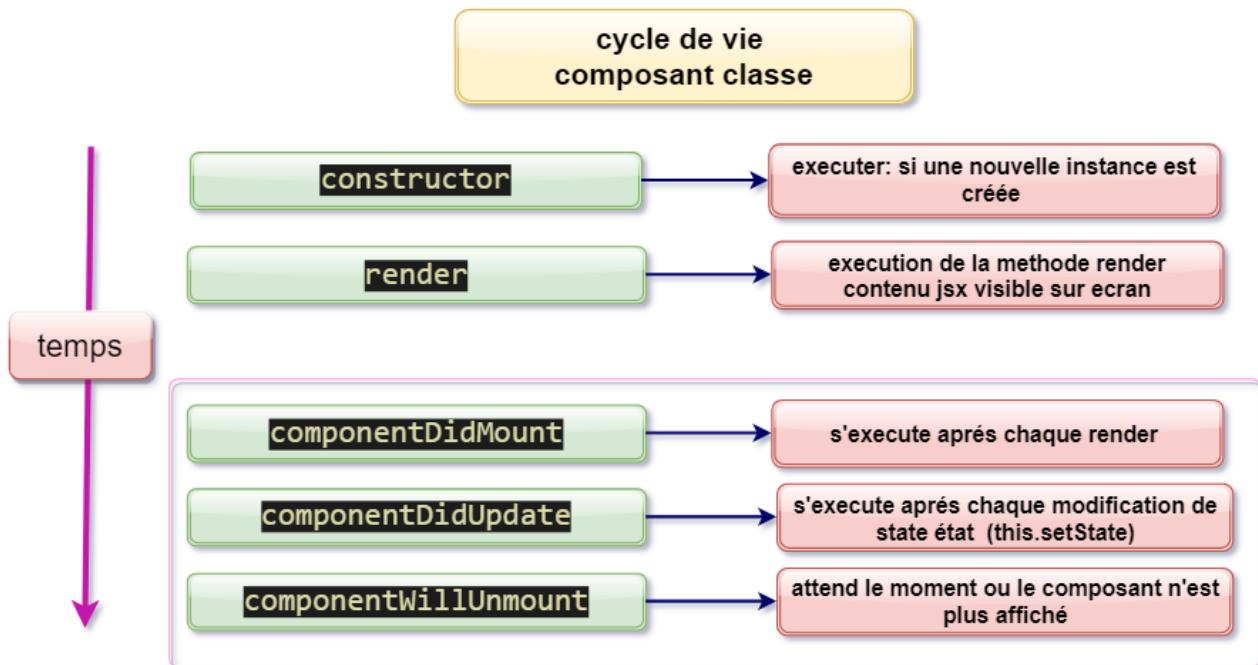
```
import React from "react";
export default function ResultatList(props) {
  return (
    <div className="Child">
      <h1>composant ResultatList</h1>
      {props.resultats.length == 0 ? (
        <p>pas de resultats</p>
      ) : (
        <div className="list">
          <ul>
            {props.resultats.map((item) => {
              return <li key={item.nom}>{item.nom}</li>;
            })}
          </ul>
        </div>
      )}
    </div>
  );
}
```

8.3. Cycle de vie des composants

8.3.1. Cycle de vie Composants créer via classes

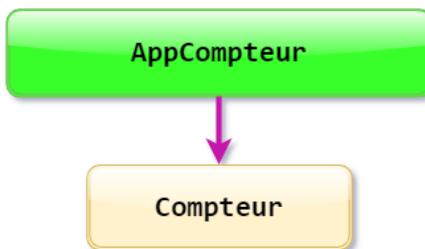
Composant life cycle est caractérisée par les méthodes optionnelles qu'on peut définir dans un composant créé par une classe.

React est responsable de faire l'appel automatique de ces méthodes au moment convenable (voir schéma) pendant le cycle de vie de composant





Exemple qui va illustrer le cycle de vie d'un composant classe



Le composant **Compteur** contient un compteur qu'on peut l'incrémenter et décrémenter

Le composant **AppCompteur** contient deux boutons pour charger et décharger le composant Compteur

Le composant **AppCompteur** est utilisé seulement pour vérifier la méthode componentWillUnmount

Monter Démonter

composant Compteur

compteur:0

Incrementer Decrementer

Rendu console au premier chargement

```

constructeur
Render
Component Did mount
-----
```

Quand je clique sur le bouton incrémenter ou décrémenter

```

Render
Component Did update
-----
```

Quand je click sur le bouton Démonter

```
Component est démonté
```

Code source : [Index.js](#)

```

import React from "react";
import ReactDOM from "react-dom/client";
import AppCompteur from "./components/AppCompteur";
const root=ReactDOM.createRoot(document.getElementById("root"))
root.render(<AppCompteur/>);
```



Compteur.js

```
import React from 'react'
export default class Compteur extends React.Component{
    constructor(props) {
        console.log("constructeur")
        super(props)

        this.state = {
            compteur:0
        }
        this.incremente=()=>{this.setState({compteur:this.state.compteur+1})}
        this.decremente=()=>{this.setState({compteur:this.state.compteur-1})}
    }

    componentDidMount(){
        //cette méthode est exécuté après render
        console.log("Component Did mount")
        console.log('-----')
    }

    componentDidUpdate(){
        //cette méthode est exécuté après mise à jour par setState
        console.log("Component Did update")
        console.log('-----')
    }
    componentWillUnmount(){
        console.log("Component est démonté")
    }
    render(){
        console.log('Render')
        return(
            <div style={{background:"yellow"}} >
                <h3>composant Compteur</h3>
                <p>compteur:{this.state.compteur}</p>
                <button onClick={this.incremente}>Incrementeur</button>
                <button onClick={this.decremente}>Decrementer</button>

            </div>
        )
    }
}
```



AppCompteur.js

```
import React from 'react'
import Compteur from './Compteur'
export default class AppCompteur extends React.Component{

    constructor(props) {

        super(props)
        this.state={isMonter:true}
        this.monter=()=>{this.setState({isMonter:true})}
        this.demonter=()=>{this.setState({isMonter:false})}
    }

    render(){

        return(
            <div >
                <button onClick={this.monter}
disabled={this.state.isMonter}>Monter</button>
                <button onClick={this.demonter}
disabled={!this.state.isMonter}>Démonter</button>
                { this.state.isMonter? <Compteur/>:null}
            </div>
        )
    }
}
```

8.3.2. Cycle de vie Composants créer via fonction

Vous pouvez tirer parti du Hook **useEffect** pour obtenir les mêmes résultats qu'avec les méthodes

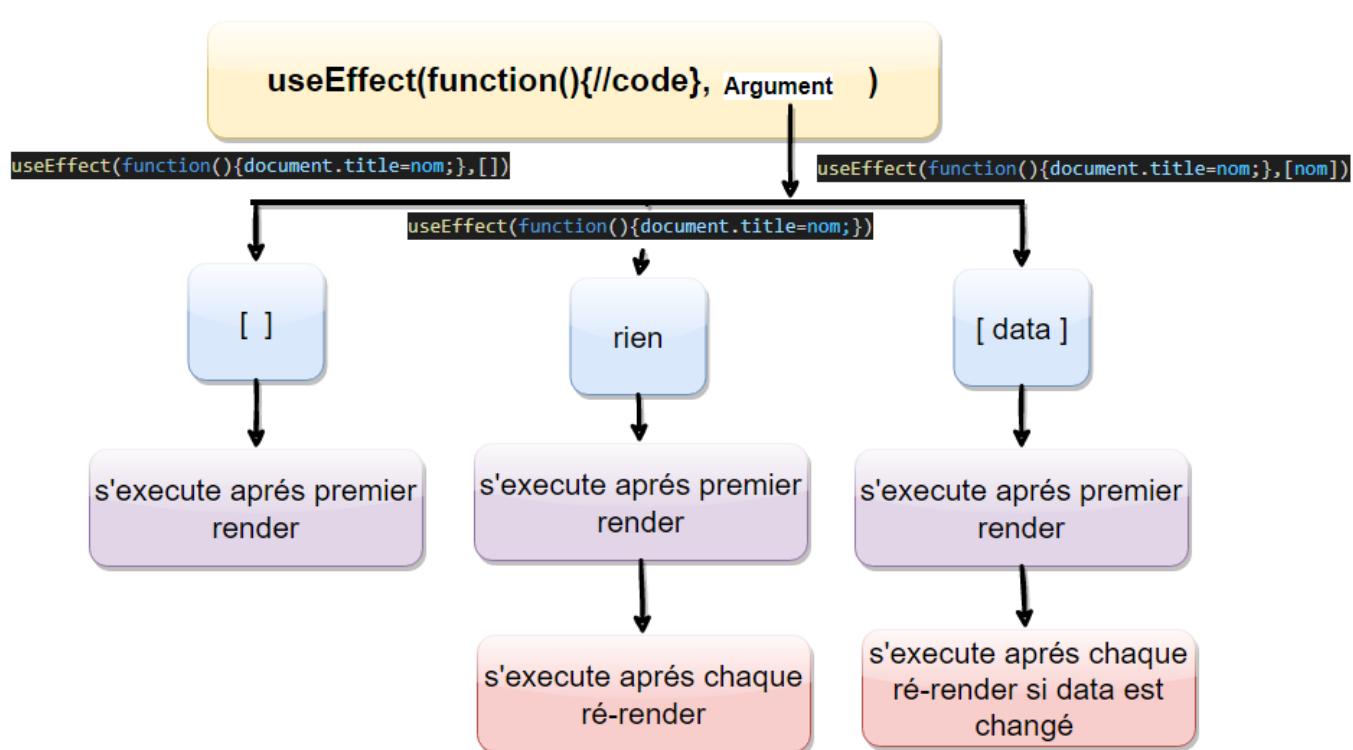
- componentDidMount
- componentDidUpdate
- componentWillUnmount

useEffect accepte deux paramètres. Le premier est un rappel qui s'exécute après le rendu, un peu comme dans componentDidMount.

Le deuxième paramètre est le tableau des dépendances d'effet. Si vous souhaitez l'exécuter uniquement lors du montage et du démontage, passez un tableau vide [].



Hook useEffect



Voir cours séance 12



9. Personnaliser les composants React

9.1. Styliser les composants React

Il existe différentes manières de styler les composants React.

Inline CSS

Il s'agit du style CSS envoyé à l'élément directement via HTML ou JSX. Vous pouvez inclure un objet JavaScript pour CSS dans les composants React.

Exemple

```
function Button() {
  return (
    <button style={{color:"#fff", borderColor:"#5C5cff",
padding:"10px", backgroundColor:"#5C5cff" }}>Nom du bouton</button>
  );
}
const root = ReactDOM.createRoot(document.getElementById('root'));
root.render(
  <Button />
);
```



Nom du bouton

Modules CSS

Les modules CSS s'assurent que tous les styles d'un composant sont regroupés à un seul endroit et s'appliquent à ce composant particulier. Cela résout certainement le problème de portée globale du CSS. La fonctionnalité de composition agit comme une arme pour représenter les styles partagés entre les états.

Exemple

Index.js

```
import React from 'react';
import ReactDOM from 'react-dom/client';
import './index.css';
function Button() {
  return (
    <button className='buttonStyle'>Nom du bouton</button>
  );
}
const root = ReactDOM.createRoot(document.getElementById('root'));
root.render(
  <Button />
);
```



Index.css

```
.buttonStyle {  
    color: #fff;  
    border-color: #5C5cff;  
    padding: 10px;  
    background-color: #5C5cff;  
}
```

Nom du bouton

Composants stylisés

Les composants stylisés sont l'une des nouvelles façons d'utiliser CSS dans le JavaScript moderne. C'est censé être le successeur des modules CSS, un moyen d'écrire du CSS qui est limité à un seul composant, et ne fuit pas vers un autre élément de la page

Les composants stylisés vous permettent d'écrire du CSS brut dans vos composants sans vous soucier des collisions de noms de classe.

Installer des composants stylisés en utilisant npm :

```
npm install styled-components
```

C'est tout ! Il ne vous reste plus qu'à ajouter cette importation :

```
import styled from 'styled-components'
```

Avec le styled objet importé, vous pouvez maintenant commencer à créer des composants stylisés. Voici le premier:

```
const Button = styled.button`  
color: #fff;  
border-color: #5C5cff;  
padding: 10px;  
background-color: #5C5cff;  
`
```

Maintenant, ce composant peut être rendu dans notre conteneur en utilisant la syntaxe normale de React:

```
return <Button>Nom du bouton</Button>;
```

Les composants stylisés offrent d'autres fonctions que vous pouvez utiliser pour créer d'autres composants, non seulement button, aimer section, h1, input et plein d'autres.

Nom du bouton

Il existe d'autres façons pour styler les éléments tels que CSS dans JS, ass & SCSS et Less.



9.2. Affichage conditionnel

L'affichage conditionnel en React fonctionne de la même façon que les conditions en Javascript. On utilise l'instruction Javascript `if` ou l'**opérateur ternaire** pour créer des éléments représentant l'état courant, et on laisse React mettre à jour l'interface utilisateur (UI) pour qu'elle corresponde.

Considérons ces deux composants :

Connecte.js	Anonymous.js
<pre>function Connecte() { return (<h1>Bienvenue !</h1>); } export default Connecte;</pre>	<pre>function Anonymous() { return (<h1>Veuillez vous inscrire.</h1>); } export default Anonymous;</pre>

Nous allons créer un composant App qui affiche un de ces deux composants, selon qu'un utilisateur est connecté ou non :

```
import Anonymous from './Anonymous';
import Connecte from './Connecte';

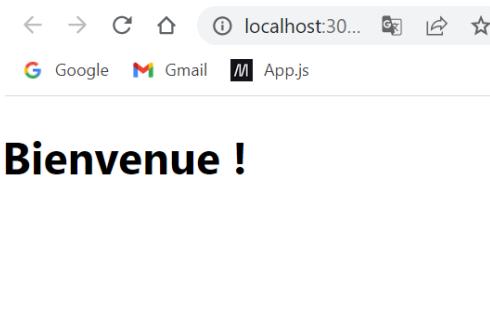
function App(props) {
  const isLoggedIn = props.isLoggedIn;
  if (isLoggedIn) {
    return <Connecte />;
  }
  return <Anonymous />;
}
export default App;
```

Cet exemple affiche un message différent selon la valeur de la prop `isLoggedIn`.

Code	Affichage
<pre>const root = ReactDOM.createRoot(document.getElementById('root')); root.render(<React.StrictMode> <App isLoggedIn={false} /> </React.StrictMode>);</pre>	<p>Veuillez vous inscrire.</p>



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



9.3. Listes et clés

Les listes

La méthode **map()** est utilisé pour prendre un tableau de nombres et doubler leurs valeurs. On peut construire des collections d'éléments et les inclure dans du JSX en utilisant les accolades {}.

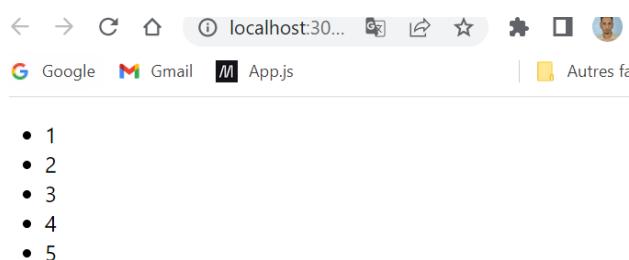
Ci-dessous, on itère sur le tableau de nombres en utilisant la méthode JavaScript map(). On retourne un élément pour chaque entrée du tableau. Enfin, on affecte le tableau d'éléments résultant à listItems :

```
const numbers = [1, 2, 3, 4, 5];
const listItems = numbers.map((numbers) =>
  <li>{numbers}</li>
);
```

On inclut tout le tableau listItems dans un élément , et on l'affiche dans le DOM :

```
const root = ReactDOM.createRoot(document.getElementById('root'));
root.render(
  <ul>{listItems}</ul>
);
```

Ce code affiche une liste à puces de nombres entre 1 et 5.





Les clés

Les clés aident React à identifier quels éléments d'une liste ont changé, ont été ajoutés ou supprimés. Vous devez donner une clé à chaque élément dans un tableau afin d'apporter aux éléments une identité stable :

```
const numbers = [1, 2, 3, 4, 5];
const listItems = numbers.map((number) =>
  <li key={number.toString()}>
    {number}
  </li>
);
```

Le meilleur moyen de choisir une clé est d'utiliser quelque chose qui identifie de façon unique un élément d'une liste parmi ses voisins. Le plus souvent on utilise l'ID de notre donnée comme clé :

```
const todoItems = todos.map((todo) =>
  <li key={todo.id}>
    {todo.text}
  </li>
);
```



10. Créer des formulaires

Les formulaires HTML fonctionnent un peu différemment des autres éléments du DOM en React car ils possèdent naturellement un état interne. Par exemple, ce formulaire en HTML qui accepte juste un nom :

```
<form>
  <label>
    Nom :
    <input type="text" name="name" />
  </label>
  <input type="submit" value="Envoyer" />
</form>
```

Ce formulaire a le comportement classique d'un formulaire HTML et redirige sur une nouvelle page quand l'utilisateur le soumet. Si vous souhaitez ce comportement en React, vous n'avez rien à faire. Cependant, dans la plupart des cas, vous voudrez pouvoir gérer la soumission avec une fonction JavaScript, qui accède aux données saisies par l'utilisateur. La manière classique de faire ça consiste à utiliser les « composants contrôlés ».

10.1. Composants contrôlés

En HTML, les éléments de formulaire tels que `<input>`, `<textarea>`, et `<select>` maintiennent généralement leur propre état et se mettent à jour par rapport aux saisies de l'utilisateur. En React, l'état modifiable est généralement stocké dans la propriété `state` des composants et mis à jour uniquement avec `setState()`.

On peut combiner ces deux concepts en utilisant l'état local React comme « source unique de vérité ». Ainsi le composant React qui affiche le formulaire contrôle aussi son comportement par rapport aux saisies de l'utilisateur. Un champ de formulaire dont l'état est contrôlé de cette façon par React est appelé un « composant contrôlé ».

Par exemple, en reprenant le code ci-dessus pour afficher le nom lors de la soumission, on peut écrire le formulaire sous forme de composant contrôlé :

Méthode 1 : Composant fonctionnel

```
NameForm.js
import { useState } from "react";

const NameForm = () => {
  const [nom, setNom] = useState('');
  const handleChange =(event)=>{
    setNom(event.target.value);
  }

  const handleSubmit =(event)=>{
    alert('Le nom a été soumis : ' + nom);
    event.preventDefault();
  }
}
```



```

    return (
      <form onSubmit={handleSubmit}>
        <label>
          Nom :
          <input type="text" name="nom" value={nom}
            onChange={handleChange} />
        </label>
        <input type="submit" value="Envoyer" />
      </form>
    );
  };
export default NameForm;

```

Méthode 2 : Composant de classe

NameForm.js

```

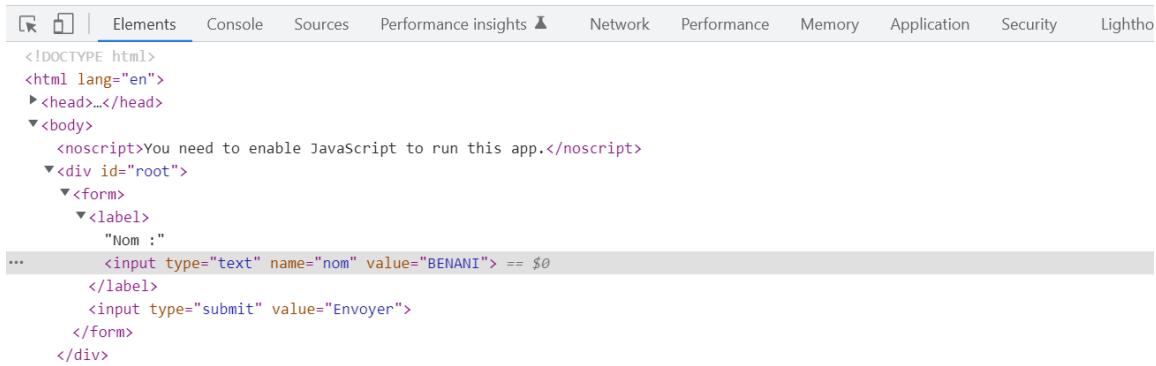
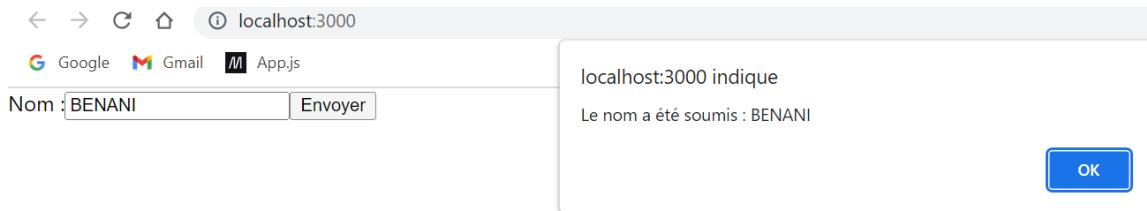
import React from 'react';

class NameForm extends React.Component {
  constructor(props) {
    super(props);
    this.state = {nom: ''};
    this.handleChange = this.handleChange.bind(this);
    this.handleSubmit = this.handleSubmit.bind(this);
  }
  handleChange(event) {
    this.setState({nom: event.target.value});
  }
  handleSubmit(event) {
    alert('Le nom a été soumis : ' + this.state.nom);
    event.preventDefault();
  }
  render() {
    return (
      <form onSubmit={this.handleSubmit}>
        <label>
          Nom :
          <input type="text" name="nom" value={this.state.nom}
            onChange={this.handleChange} />
        </label>
        <input type="submit" value="Envoyer" />
      </form>
    );
  }
}
export default NameForm;

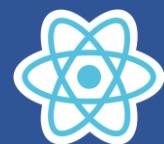
```



Après avoir affiché le composant et tapé quelques caractères dans le champ de saisie, nous obtenons l'affichage suivant :



```
<!DOCTYPE html>
<html lang="en">
  <head>...</head>
  <body>
    <noscript>You need to enable JavaScript to run this app.</noscript>
    <div id="root">
      <form>
        <label>
          "Nom :"
        <input type="text" name="nom" value="BENANI"> == $0
        </label>
        <input type="submit" value="Envoyer">
      </form>
    </div>
```



10.2. Gérer les champs de saisie multilignes

Un champ de saisie multiligne se définit grâce à l'élément <textarea>. Créons un composant <TextArea> qui «améliore» l'élément <textarea> traditionnel.

Commençons par un composant de base permettant de créer un élément <textarea> classique. Il s'utilisera sous la forme suivante:

```
<TextArea cols={40} rows={10} value="Tapez votre texte ici" />
```

Comme on l'avait vu précédemment pour les champs de saisie, il faut implémenter l'événement onChange afin que la saisie soit prise en compte dans le champ. De plus, le state doit comporter la valeur saisie dans le champ, afin que cette valeur soit automatiquement rafraîchie lors de la saisie.

La propriété value définie dans le composant devra donc servir à initialiser la propriété value du state.

Implémentation du composant <TextArea> de base

Méthode 1 : Composant fonctionnel

TextArea.js

```
import { useState } from "react";
const FTextArea = (props) => {
    const [message, setMessage] = useState(props.value);
    const handlerChange=(event)=>{
        setMessage(event.target.value);
    }
    const handlerFocus=()=>{
        setMessage("");
    }
    return (
        <textarea cols={props.cols}
        rows={props.rows}
        value={message}
        onFocus={handlerFocus.bind(this)}
        onChange={handlerChange.bind(this)} />
    );
};
export default FTextArea;
```

Méthode 2 : Composant de classe

TextArea.js

```
import React from 'react';
class TextArea extends React.Component {
    constructor(props) {
        super(props);
        this.state = { message : props.value };
    }
    handlerChange(event) {
```

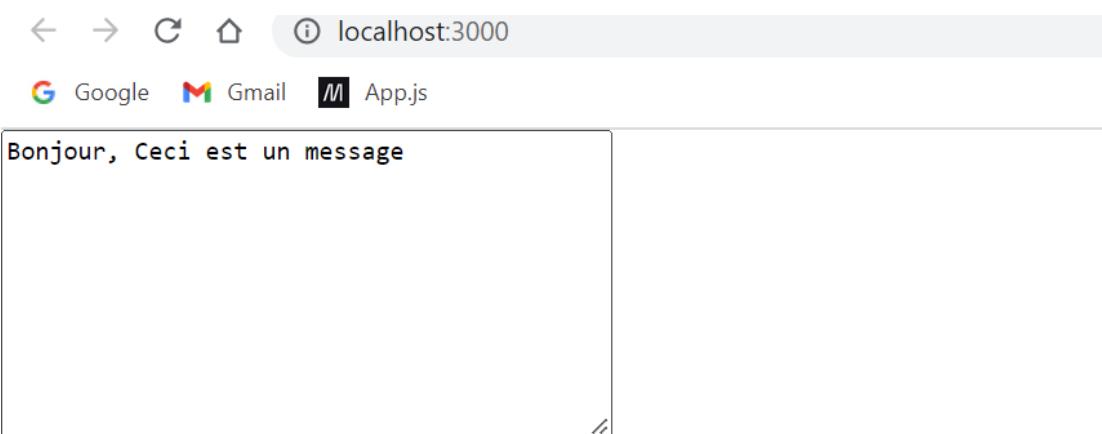


```

        this.setState({message : event.target.value});
    }
    handlerFocus(event) {
        this.setState({message : ""});
    }
    render() {
        return (
            <textarea cols={this.props.cols}
                      rows={this.props.rows}
                      value={this.state.message}
                      onFocus={this.handlerFocus.bind(this)}
                      onChange={this.handlerChange.bind(this)} />
        )
    }
}
export default TextArea;

```

Après avoir affiché le composant et tapé quelques caractères dans le champ de saisie, nous obtenons l'affichage suivant :



A screenshot of a web browser window. The address bar shows "localhost:3000". Below the address bar, there are tabs for Google, Gmail, and App.js. The main content area displays a text area containing the text "Bonjour, Ceci est un message".



A screenshot of the Chrome DevTools Elements tab. The DOM tree shows the following structure:

```

<!DOCTYPE html>
<html lang="en">
  <head>...</head>
  ...<body> == $0
    <noscript>You need to enable JavaScript to run this app.</noscript>
    <div id="root">
      <textarea cols="40" rows="10">Bonjour, Ceci est un message</textarea>
    </div>
  </body>
</html>

```



10.3. Gérer les listes de sélection

Les listes de sélection correspondent aux éléments <select> intégrant les éléments <option> décrivant les items de la liste. Par exemple, voici une liste écrite en HTML intégrant cinq éléments de liste :

```
<select>
  <option value="1">Element1</option>
  <option value="2">Element2</option>
  <option value="3">Element3</option>
  <option value="4">Element4</option>
  <option value="5">Element5</option>
</select>
```

On utilise pour cela un composant <Select> dans lequel on transmet une propriété options qui est un tableau indiquant la liste des éléments à afficher (sous forme de chaînes de caractères). Les éléments React sont créés avec JSX.

Méthode 1 : Composant fonctionnel

Select.js

```
const Select = (props) => {
  const handlerChange=(event)=>{
    console.log('key=' + event.target.value);
  }
  return (
    <select onChange={handlerChange.bind(this)}>
      {props.options.map(function(option, index) {
        return <option key={index+1}
value={index+1}>{option}</option>
      })}
    </select>
  );
}
export default Select;
```



Méthode 2 : Composant de classe

Select.js

```
import React from 'react';
class Select extends React.Component {
    constructor(props) {
        super(props);
    }
    render() {
        return (
            <select>
                {this.props.options.map(function(option, index) {
                    return <option key={index+1}
value={index+1}>{option}</option>
                })
            }
            </select>
        )
    }
    export default Select;
```

Après avoir affiché le composant

```
<Select options={['Element1", "Element2", "Element3", "Element4",
"Element5"]}> />
```

Nous obtenons l'affichage suivant :

The screenshot shows a browser window with the address bar displaying "localhost:3000". Below the address bar is a navigation bar with icons for back, forward, refresh, and home. The main content area shows a dropdown menu with the value "Element1". Below the dropdown is a screenshot of the browser's developer tools Elements tab. The DOM tree is visible, starting with the root element <html>. Under <body>, there is a <div id="root"> element, which contains a <select> element with five <option> elements, each labeled "Element1" through "Element5" respectively. The <select> element is highlighted with a blue selection bar.

```
<!DOCTYPE html>
<html lang="en">
  <head>...</head>
  <body>
    ... <div id="root"> == $0
      <select>
        <option value="1">Element1</option>
        <option value="2">Element2</option>
        <option value="3">Element3</option>
        <option value="4">Element4</option>
        <option value="5">Element5</option>
      </select>
    </div>
  </body>
</html>
```



Afficher l'attribut value de l'élément sélectionné

On utilise l'attribut onChange sur l'élément <select> afin d'effectuer le traitement lors de la sélection d'un nouvel élément dans la liste.

```
import React from 'react';
class Select extends React.Component {
    constructor(props) {
        super(props);
    }
    handlerChange(event) {
        console.log('key=' + event.target.value);
    }
    render() {
        return (
            <select onChange={this.handlerChange.bind(this)}>
                {this.props.options.map(function(option, index) {
                    return <option key={index+1}
value={index+1}>{option}</option>
                })}
            </select>
        )
    }
}
export default Select;
```

La valeur de l'élément sélectionné est récupérée au moyen de ***event.target.value***

Après avoir sélectionné plusieurs fois des éléments dans la liste, on obtient l'affichage suivant :

The screenshot shows a browser window with the address bar set to 'localhost:3000'. Below the address bar is a navigation bar with icons for back, forward, refresh, and home. A tab labeled 'App.js' is open. The main content area contains a dropdown menu with the value 'Element2'. At the bottom of the page is the browser's developer tools console. The 'Console' tab is selected. The console output shows the text 'key=2'.

Les valeurs associées aux éléments sélectionnés sont affichées dans la console.

Sélectionner l'élément de liste dont l'attribut value vaut 4

```
<Select defaultValue={4} />
```



10.4. Gérer les boutons radio

Nous utilisons ici un seul composant React nommé <RadioGroup> qui est chargé de gérer les boutons radio qui lui sont transmis. Le composant est utilisé de la façon suivante:

Méthode 1 : Composant fonctionnel

RadioGroup.js

```
const FRadioGroup = (props) => {
    return (
        <div>
            props.radios.map((radio, index) => {
                return (
                    <label key={index+1}><span>{radio.text}</span>
                    <input type="radio" value={radio.value}>
                    name="radioname" />
                    </label>
                )
            })
        </div>
    );
}
export default FRadioGroup;
```

Méthode 2 : Composant de classe

RadioGroup.js

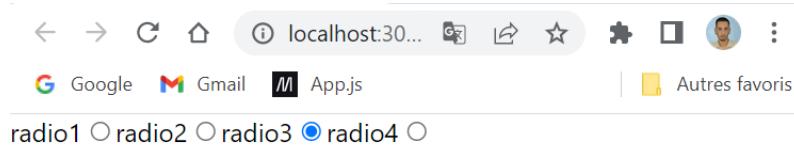
```
import React from 'react';
class RadioGroup extends React.Component {
    constructor(props) {
        super(props);
    }
    render() {
        return (
            <div>
                this.props.radios.map((radio, index) => {
                    return (
                        <label
                            key={index}><span>{radio.text}</span>
                            <input type="radio" value={radio.value}>
                            name={radio.name} checked={radio.checked} />
                            </label>
                    )
                })
            </div>
        )
    }
}
export default RadioGroup;
```



Après avoir affiché le composant

```
import RadioGroup from './RadioGroup';
var radios = [
  { value: 1, text: "radio1" },
  { value: 2, text: "radio2" },
  { value: 3, text: "radio3", checked: true },
  { value: 4, text: "radio4" }
];
const root = ReactDOM.createRoot(document.getElementById('root'));
root.render(<RadioGroup radios={radios} name="group1" />);
```

On obtient l'affichage suivant :



Chaque bouton sélectionné désélectionne le bouton précédemment sélectionné.



10.5. Gérer les cases à cocher

La gestion des cases à cocher est similaire à celle des boutons radio. Cependant, plusieurs cases à cocher peuvent être sélectionnées en même temps, contrairement aux boutons radio d'un même groupe.

Nous utiliserons pour cela deux composants React:

- le composant `<CheckBox>` permettra de gérer une case à cocher (via `this.state.checked`) ;
- le composant `<CheckBoxGroup>` permettra de gérer l'ensemble des cases à cocher à afficher.

CheckBoxGroup.js

```
import React from "react";
class CheckBox extends React.Component {
    constructor(props) {
        super(props);
        this.state = { checked : props.checked || false };
    }
    handlerChange(event) {
        this.setState({checked : event.target.checked});
    }
    render() {
        return (
            <label>
                <span>{this.props.text}</span>
                <input type="checkbox" value={this.props.value}
                    checked={this.state.checked}
                    onChange={this.handlerChange.bind(this)} />
                <br/>
            </label>
        )
    }
}
class CheckBoxGroup extends React.Component {
    constructor(props) {
        super(props);
    }
    render() {
        return (
            <div>
                {this.props.checkboxes.map((checkbox, index) => {
                    return (
                        <CheckBox key={index} text={checkbox.text}
                            value={checkbox.value}
                            checked={checkbox.checked}
                        />
                    )
                })
            }
        )
    }
}
```

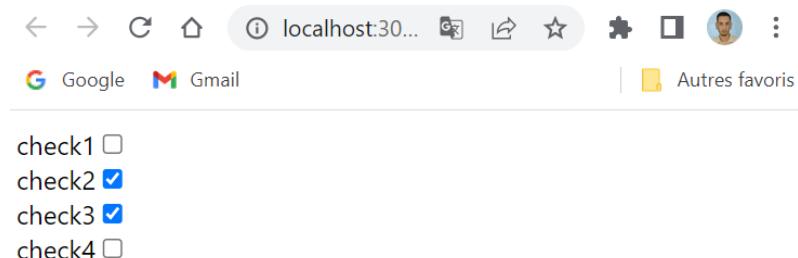


```
        })
      }
    )
}
export default CheckBoxGroup;
```

Après avoir affiché le composant

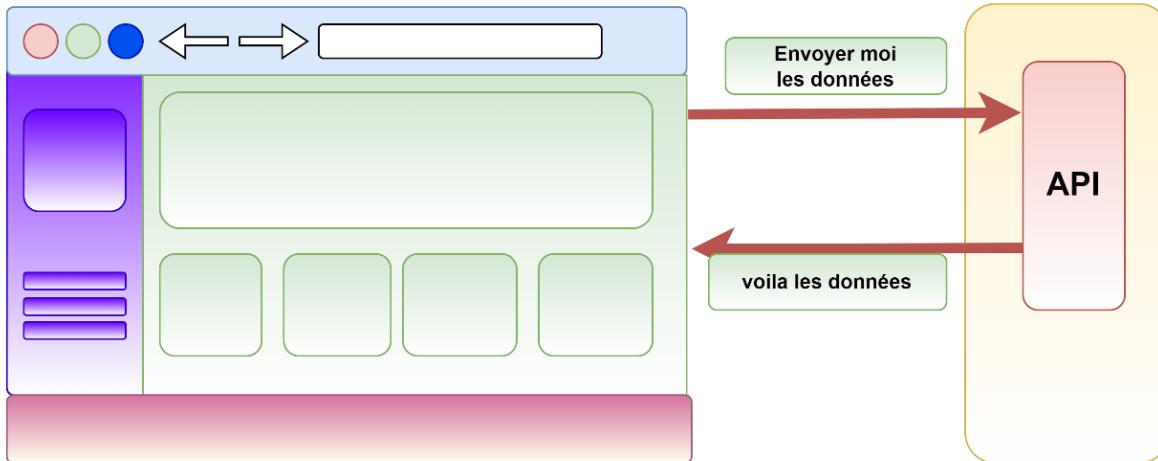
```
import CheckBoxGroup from './CheckBoxGroup';
var checkboxes = [
  { value : 1, text : "check1" },
  { value : 2, text : "check2", checked : true },
  { value : 3, text : "check3", checked : true },
  { value : 4, text : "check4" }
];
const root = ReactDOM.createRoot(document.getElementById('root'));
root.render(<CheckBoxGroup checkboxes={checkboxes} />);
```

On obtient l'affichage suivant :





11.1. React et AJAX



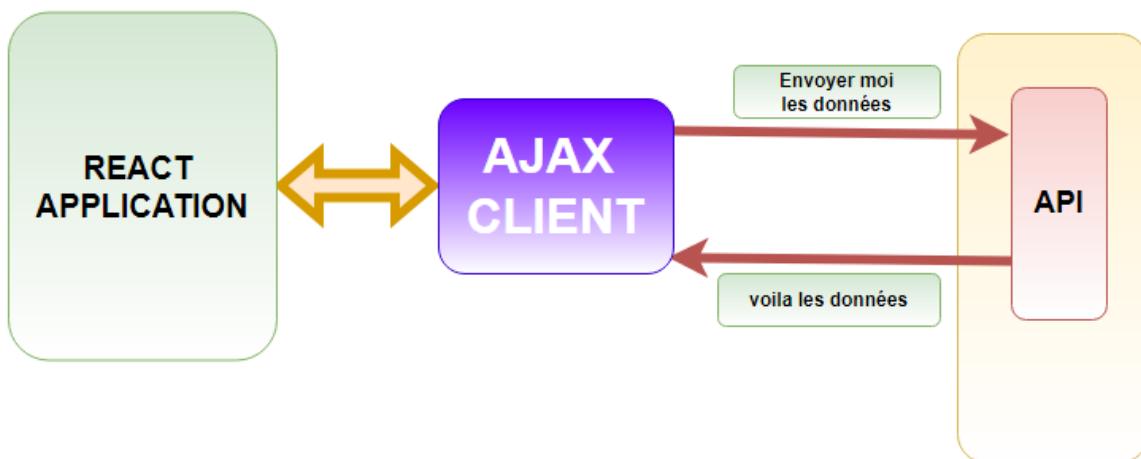
Ajax est un concept permettant de mettre à jour une partie de la page HTML affichée en effectuant une requête au serveur. Ce concept est très utilisé de nos jours dans la plupart des sites web, car il permet une mise à jour dynamique de la page sans avoir à la recharger totalement.

React n'intègre pas une API permettant d'effectuer des requêtes Ajax, mais il est possible d'utiliser d'autres API fournies par d'autres bibliothèques, voire d'utiliser l'API interne du navigateur.

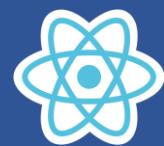
Nous étudions ici ces deux possibilités :

- utilisation de l'API interne du navigateur (en utilisant la méthode `fetch()`).
- Utilisation de l'API AXIOS

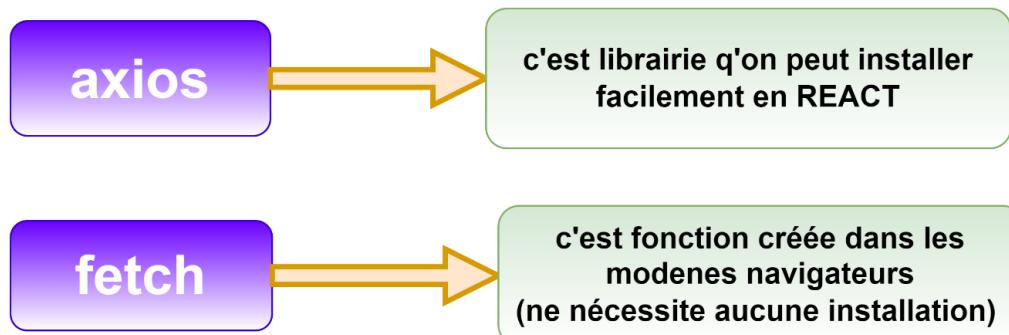
Pour le fournisseur d'API on va utiliser nos propres API sur un serveur PHP ou bien utiliser l'API en ligne :



Il est à préciser que REACT est une librairie destinée pour faire l'affichage de contenu HTML au client, interagir avec le client, récupérer les requêtes client.



REACT n'est pas responsable pour consommer un API, c'est le rôle de développeur d'ajouter la partie du code responsable de cette opération (utiliser par exemple fetch, axios.....)



L'endpoint de l'API est sous la forme :

<https://jsonplaceholder.typicode.com/todos>

Quand on navigue à cette url on aura le rendu suivant :



A screenshot of a browser window showing the JSON response from the endpoint. The URL in the address bar is 'https://jsonplaceholder.typicode.com/todos'. The response is a JSON array of three objects, each representing a todo item:

```
// 20220730191634
// https://jsonplaceholder.typicode.com/todos

[
  {
    "userId": 1,
    "id": 1,
    "title": "delectus aut autem",
    "completed": false
  },
  {
    "userId": 1,
    "id": 2,
    "title": "quis ut nam facilis et officia qui",
    "completed": false
  },
  {
    "userId": 1,
    "id": 3,
    "title": "fugiat veniam minus",
    "completed": false
  }
]
```

Cet API retourne les données en format JSON



Les différents EndPoint de cet API

endpoint	Array de :
https://jsonplaceholder.typicode.com/todos	{ "userId": 1, "id": 1, "title": "delectus aut autem", "completed": false }
https://jsonplaceholder.typicode.com/comments	{ "postId": 1, "id": 1, "name": "id labore ex et quam laborum", "email": "Eliseo@gardner.biz", "body": "laudantium enim quasi est quidem magnam voluptate ipsam eos\\ntempora quo necessitatibus\\ " } ,
https://jsonplaceholder.typicode.com/comments	{ "userId": 1, "id": 1, "title": "quidem molestiae enim" },
https://jsonplaceholder.typicode.com/photos	{ "albumId": 1, "id": 1, "title": "accusamus beatae ad facilis cum similique qui sunt", "url": "https://via.placeholder.com/600/92c952", "thumbnailUrl": "https://via.placeholder.com/150/92c952" },
https://jsonplaceholder.typicode.com/todos	{ "userId": 1, "id": 1, "title": "delectus aut autem", "completed": false }
https://jsonplaceholder.typicode.com/users	{ "id": 1, "name": "Leanne Graham", "username": "Bret", "email": "Sincere@april.biz", "address": { "street": "Kulas Light", "suite": "Apt. 556", "city": "Gwenborough", "zipcode": "92998-3874", "geo": { "lat": "-37.3159", "lng": "81.1496" }}



Les exemples qui vont être utilisés dans le cours vont être traités par à la fois **FETCH** et **AXIOS**

11.2. Consommation d'un API par AXIOS fonctionnel composant

Axios Client HTTP basé sur les promesses pour navigateur et node.js

Axios est un client HTTP simple basé sur les promesses compatibles avec le navigateur et node.js. Il propose une librairie facile à utiliser et à étendre, le tout dans un tout petit package.

Installation de axios

```
npm install --save axios
```

Utilisation de axios pour consommer l'API : <https://jsonplaceholder.typicode.com/users>

Il faut importer axios

```
import axios from 'axios'
```

Le code suivant :

```
axios.get('https://jsonplaceholder.typicode.com/users').then(
  (res)=>{ console.log(res)}
)
```

Affiche le rendu console

```
▶ {data: Array(10), status: 200, statusText: '', headers: {...}, config: {...}, ...}
```

```
▼ {data: Array(10), status: 200, statusText: '', headers: {...}, config: {...}, ...} ⓘ
▶ config: {transitional: {...}, transformRequest: Array(1), transformResponse: Array(1), timeout: 0, adapter: f, ...}
▼ data: Array(10)
  ▶ 0: {id: 1, name: 'Leanne Graham', username: 'Bret', email: 'Sincere@april.biz', address: {...}, ...}
  ▶ 1: {id: 2, name: 'Ervin Howell', username: 'Antonette', email: 'Shanna@melissa.tv', address: {...}, ...}
  ▶ 2: {id: 3, name: 'Clementine Bauch', username: 'Samantha', email: 'Nathan@yesenia.net', address: {...}, ...}
  ▶ 3: {id: 4, name: 'Patricia Lebsack', username: 'Karianne', email: 'Julianne.OConner@kory.org', address: {...}, ...}
  ▶ 4: {id: 5, name: 'Chelsey Dietrich', username: 'Kamren', email: 'Lucio_Hettinger@annie.ca', address: {...}, ...}
  ▶ 5: {id: 6, name: 'Mrs. Dennis Schulist', username: 'Leopoldo_Corkery', email: 'Karley_Dach@jasper.info', address: {...}, ...}
  ▶ 6: {id: 7, name: 'Kurtis Weissnat', username: 'Elwyn.Skiles', email: 'Telly.Hoeger@billy.biz', address: {...}, ...}
  ▶ 7: {id: 8, name: 'Nicholas Runolfsdottir V', username: 'Maxime_Nienow', email: 'Sherwood@rosamond.me', address: {...}, ...}
  ▶ 8: {id: 9, name: 'Glenna Reichert', username: 'Delphine', email: 'Chaim_McDermott@dana.io', address: {...}, ...}
  ▶ 9: {id: 10, name: 'Clementina DuBuque', username: 'Moriah.Stanton', email: 'Rey.Padberg@karina.biz', address: {...}, ...}
  ▶ length: 10
  ▶ [[Prototype]]: Array(0)
▶ headers: {cache-control: 'max-age=43200', content-type: 'application/json; charset=utf-8', expires: '-1', pragma: 'no-cache'}
▶ request: XMLHttpRequest {onreadystatechange: null, readyState: 4, timeout: 0, withCredentials: false, upload: XMLHttpRequestUpload, ...}
status: 200
statusText: ""
▶ [[Prototype]]: Object
```

Le rendu est un objet javascript qui contient les propriétés data, status, statusText, headers....

On peut remarquer que la propriété data contient un Array constitué par les objets JavaScript user.

Par ailleurs si on écrit :

```
axios.get('https://jsonplaceholder.typicode.com/users').then(
  (res)=>{ console.log(res.data)}
)
```



On aura le rendu suivant sur la console:

```
▼ (10) [{}]
  ▶ 0: {id: 1, name: 'Leanne Graham', username: 'Bret', email: 'Sincere@april.biz', address: {...}, ...}
  ▶ 1: {id: 2, name: 'Ervin Howell', username: 'Antonette', email: 'Shanna@melissa.tv', address: {...}, ...}
  ▶ 2: {id: 3, name: 'Clementine Bauch', username: 'Samantha', email: 'Nathan@esenia.net', address: {...}, ...}
  ▶ 3: {id: 4, name: 'Patricia Lebsack', username: 'Karianne', email: 'Julianne.OConner@kory.org', address: {...}, ...}
  ▶ 4: {id: 5, name: 'Chelsey Dietrich', username: 'Kamren', email: 'Lucio_Hettinger@annie.ca', address: {...}, ...}
  ▶ 5: {id: 6, name: 'Mrs. Dennis Schulist', username: 'Leopoldo_Corkery', email: 'Karley_Dach@jasper.info', address: {...}, ...}
  ▶ 6: {id: 7, name: 'Kurtis Weissnat', username: 'Elwyn.Skiles', email: 'Telly.Hoeger@billy.biz', address: {...}, ...}
  ▶ 7: {id: 8, name: 'Nicholas Runolfsdottir V', username: 'Maxime_Nienow', email: 'Sherwood@rosamond.me', address: {...}, ...}
  ▶ 8: {id: 9, name: 'Glenna Reichert', username: 'Delphine', email: 'Chaim_McDermott@dana.io', address: {...}, ...}
  ▶ 9: {id: 10, name: 'Clementina DuBuque', username: 'Moriah.Stanton', email: 'Rey.Padberg@karina.biz', address: {...}, ...}
  length: 10
  ▶ [[Prototype]]: Array(0)
```

Par conséquent res.data est un Array de 10 users.

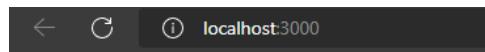
Maintenant on doit se demander où nous devons placer le code de consommation de l'API

Si on est dans un fonctionnel composant, le code doit être placé dans useEffect mais le deuxième argument doit contenir [], pour que l'exécution se lance uniquement au premier rendu

```
useEffect(()=>{
  axios.get('https://jsonplaceholder.typicode.com/users').then(
    (res)=>{ console.log(res.data)}
  )
},[])
```

Si on ne met pas l'argument [], le code sera exécuté à chaque changement des propriétés d'état useState

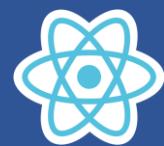
Affichant maintenant le nombre des users, pour ce faire il faut créer une propriété d'état useState utilisateurs destinée pour récupérer le résultat users de l'API



nombre d'utilisateurs: 10

Code App.js:

```
import React, { useEffect, useState } from 'react'
import axios from 'axios'
export default function App(){
  const [utilisateurs, setUtilisateurs] = useState([])
  useEffect(()=>{
    axios.get('https://jsonplaceholder.typicode.com/users')
      .then((res)=>{ console.log(res.data); setUtilisateurs(res.data)})
  },[])
  return(
    <div>
      <h1>nombre d'utilisateurs: {utilisateurs.length}</h1>
    </div>
  )
}
```



Retenons que l'objet user est de la forme :

```
{  
  "id": 1,  
  "name": "Leanne Graham",  
  "username": "Bret",  
  "email": "Sincere@april.biz",  
  "address": {  
    "street": "Kulas Light",  
    "suite": "Apt. 556",  
    "city": "Gwenborough",  
    "zipcode": "92998-3874",  
    "geo": {  
      "lat": "-37.3159",  
      "lng": "81.1496"  
    }  
  }  
}
```

Nous allons afficher les utilisateurs sur l'écran, pour ce faire on ajoute le code jsx suivant



The screenshot shows a web browser window with the URL `localhost:3000`. The page displays the title **nombre d'utilisateurs: 10**. Below the title, there are three user profiles, each enclosed in a green box:

- nom: Leanne Graham Bret**
email:Sincere@april.biz
ville:Gwenborough rue:Kulas Light
- nom: Ervin Howell Antonette**
email:Shanna@melissa.tv
ville:Wisokyburgh rue:Victor Plains
- nom: Clementine Bauch Samantha**
email:Nathan@yesenia.net
ville:McKenziehaven rue:Douglas Extension



Code finale de App.js

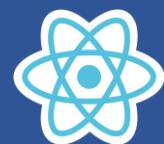
```
import React, { useEffect, useState } from 'react'
import axios from 'axios'
export default function App(){
    const [utilisateurs, setUtilisateurs]=useState([])
    useEffect(()=>{
        axios.get('https://jsonplaceholder.typicode.com/users')
            .then((res)=>{setUtilisateurs(res.data)})
    },[])
    return(
        <div>
            <h3>liste utilisateurs</h3>
            <div>
                <h1>nombre d'utilisateurs: {utilisateurs.length}</h1>
                {utilisateurs.map(user=>{
                    return(
                        <div className='child' key={user.id}>
                            <h3 style={{color:"rgb(92, 62, 3)"}}>nom:{user.name} {user.username}</h3>
                            <p>email:{user.email}</p>
                            <p>ville:{user.address.city}</p>
                            rue:{user.address.street} </p>
                        </div>
                    )
                })}</div>
        </div>
    )}
```

On peut utiliser **async await**

```
useEffect(()=>{
    const getData=async ()=>{
        const users=
    await axios.get('https://jsonplaceholder.typicode.com/users')
        setUtilisateurs(users.data)
    }
    getData()
},[])
```

Le style utilisé pour mettre en forme le rendu

```
.child{
    background-color:rgb(158, 227, 133);
    width:40%;
    margin:4px auto;
    padding:10px;
    border: 1px solid rgb(70, 88, 64);
    border-radius: 4px;
    box-shadow:8px rgb(210, 217, 208); ;
```



11.3. Consommation d'un API par fetch fonctionnel composant

```
fetch('https://jsonplaceholder.typicode.com/users')
    .then((response)=>{ console.log(response); return response.json()})
    .then((users)=>{console.log(users);setUtilisateurs(users)})
```

Rendu console

```
▶ Response {type: 'cors', url: 'https://jsonplaceholder.typicode.com/users', redirected: false, status: 200, ok: true, ...}
▶ (10) [{...}, {...}, {...}, {...}, {...}, {...}, {...}, {...}, {...}, {...}]
```

Etant donné que les données response ne sont pas du json, on applique la méthode json à response qui retourne une promesse d'où l'enchainement de la deuxième then.

La variable users contient cette fois ci un Array contenant des objet java script user , en suite on met à jour la variable d'état utilisateurs en utilisant le setter setUtilisateurs(users)

L'utilisation **async await** donne le même résultat (voir cours Java script Avancé):

```
const [utilisateurs, setUtilisateurs]=useState([])
  useEffect(()=>{
    const getData=async (rep)=>{
      const response= await fetch('https://jsonplaceholder.typicode.com/users')
      const users=await response.json()
      setUtilisateurs(users)
    }
    getData()
  },[])
```

11.4. Consommation d'un API par Axios classe composant

Pour les classes composant on met le code de consommation API dans **componentDidMount()**

```
componentDidMount(){
  axios.get('https://jsonplaceholder.typicode.com/users')
    .then((res)=>{this.setState({utilisateurs:res.data})})
}
```

Code complet :

App.js

```
import React from "react";
import axios from 'axios'
export default class App extends React.Component{
  constructor(props) {
    super(props)
    this.state = {
      utilisateurs:[]
    }
  }
  componentDidMount(){
```



```

    axios.get('https://jsonplaceholder.typicode.com/users')
      .then((res)=>{this.setState({utilisateurs:res.data})})
    }

    render(){
      return(
        <div>
          <h3>liste utilisateurs</h3>

          <div>
            <h1>nombre d'utilisateurs:</h1>
            {this.state.utilisateurs.length}
            {this.state.utilisateurs.map(user=>{

              return(
                <div className='child' key={user.id}>
                  <h3 style={{color:"rgb(92, 62, 3)"}>nom:{user.name} {user.username}</h3>
                  <p>email:{user.email}</p>
                  <p> ville:{user.address.city}</p>
                  rue:{user.address.street} </p>
                </div>
              )
            })
          }</div>
        </div>
      )
    }
  }
}

```

11.5. Consommation d'un API par fetch classe composant :

On change seulement le code de la méthode **componentDidMount**

```

componentDidMount(){

  fetch('https://jsonplaceholder.typicode.com/users')
    .then((response)=>{ console.log(response); return response.json()})
    .then((users)=>{this.setState({utilisateurs:users})})

}

```

Exercice d'application 1 avec solution :

Sachons que le endpoint <https://jsonplaceholder.typicode.com/users/3>

Retourne l'objet user qui a le id égal à 3



```
{
  "id": 3,
  "name": "Clementine Bauch",
  "username": "Samantha",
  "email": "Nathan@yesenia.net",
  "address": {
    "street": "Douglas Extension",
    "suite": "Suite 847",
    "city": "McKenziehaven",
    "zipcode": "59590-4157",
    "geo": {
      "lat": "-68.6102",
      "lng": "-47.0653"
    }
  },
  "phone": "1-463-123-4447",
  "website": "ramiro.info",
  "company": {
    "name": "Romaguera-Jacobson",
    "catchPhrase": "Face to face bifurcated interface",
    "bs": "e-enable strategic applications"
  }
}
```

Créer l'application React

Details utilisateur

donner le id:

id:1 nom: Leanne Graham Bret
 email:Sincere@april.biz
 telephone:1-770-736-8031 x56442
 site web:hildegard.org
 rue : Kulas Light
 ville : Gwenborough

L'utilisateur saisie le id suite à l'événement onChange, les informations de l'utilisateur s'affichent



Details utilisateur

donner le id:
svp choisir un id valide!!!!

Le message 'svp choisir un id valide !!!!' S'affiche si l'utilisateur n'a rien saisi ou bien le id n'existe pas

Eléments de solution :

```
import React, { useEffect, useState } from 'react'
export default function App(){
  const [id,setId]=useState(1)
  const [utilisateur,setUtilisateur]=useState({})
  const [address,setAddress]=useState({})
  function handelChangeId(event){
    setId(event.target.value)
  }
  useEffect(
()=>{
fetch(`https://jsonplaceholder.typicode.com/users/${id}`)
.then((response)=>{ return response.json()})
.then((user)=>{setAddress(user.address);setUtilisateur(user);})
,[id] )
  return(
    <div>
      <h1>Details utilisateur</h1>
      <div>
        <label>donner le id:</label>
        <input type="text" onChange={handelChangeId} value={id}></input>
      </div>
      {utilisateur && address ?
      <div className='child' key={utilisateur.id}>
        <h3 style={{color:"rgb(92, 62, 3)"}}> id:{utilisateur.id} nom:{utilisateur.name} {utilisateur.username}</h3>
        <p>email:{utilisateur.email}</p>
        <p> telephone:{utilisateur.phone} </p>
        <p> site web:{utilisateur.website} </p>
        <p> rue : {address.street} </p>
        <p> ville : {address.city} </p>
      </div>:"svp choisir un id valide!!!!" }
      </div>
    )
}
}
```

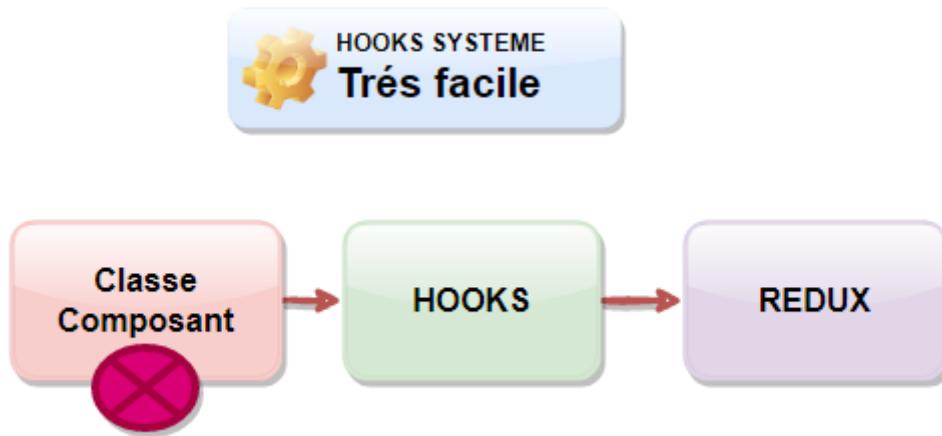


CSS :

```
.child{  
background-color:rgb(158, 227, 133);  
width:40%;  
margin:4px auto;  
padding:10px;  
border: 1px solid rgb(70, 88, 64);  
border-radius: 4px;  
box-shadow:8px rgb(210, 217, 208); ;  
}
```



12. Incorporer des données dans une application React avec les hooks



12.1. Utilisation des hooks pour gérer l'état de composant créé par fonction

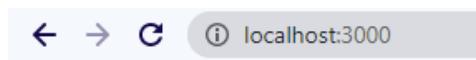
Les Hooks ont été ajoutés à React dans la version 16.8.

Les Hooks permettent aux composants fonctionnels d'accéder à l'état et à d'autres fonctionnalités de React. Pour cette raison, les composants de classe ne sont généralement plus nécessaires.

Bien que les Hooks remplacent généralement les composants de classe, il n'est pas prévu de supprimer des classes de React.

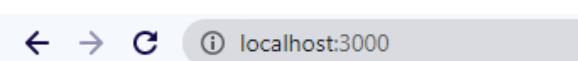
Retenant l'exemple précédent en utilisant cette fois ci un composant AutreMessage créé par fonction

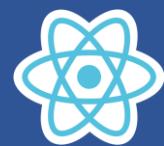
Pour expliquer l'objet state, considérons que nous souhaitons écrire un composant Message simple qui permet d'afficher le rendu suivant



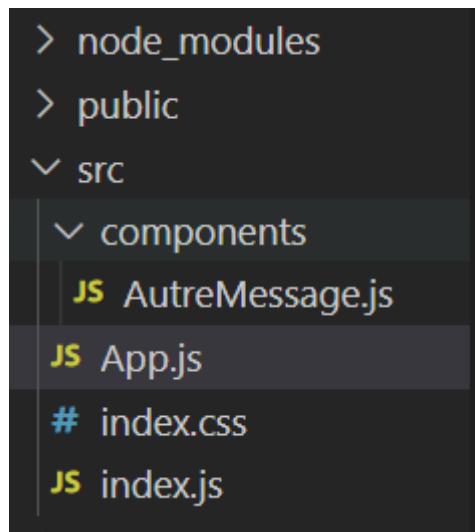
Si l'utilisateur clique sur le bouton inscription

Le rendu devient





Vous allez remarquer que après le clic sur le bouton le message Bien Venu visiteur change pour devenir votre inscription est effectuée, le texte de bouton inscription change pour devenir merci



Créer le fichier AutreMessage.js dans le dossier src/components

AutreMessage.js

```
import React, { useState } from "react";
export default function AutreMessage() {

  const [message, setMessage] = useState("Bien venu visiteur");
  const [btnMessage, setBtnMessage] = useState("inscription");

  function inscription() {
    setMessage("votre inscription est effectuée");
    setBtnMessage("merci");
  }

  return (
    <div>
      <h2>{message}</h2>
      <button onClick={() => inscription()}>{btnMessage}</button>
    </div>
  );
}
```



App.js

Le fichier App est créé directement dans le dossier src

```
import React from 'react'
import AutreMessage from './components/AutreMessage';
export default function App() {
  return (
    <div>
      <AutreMessage/>

    </div>
  );
}
```

index.js

```
import React from "react";
import ReactDOM from "react-dom/client";
import App from "./App";
const root=ReactDOM.createRoot(document.getElementById("root"))
root.render(<App/>);
```

Vous allez remarquer que cette fois ci on a créé le composant App dans fichier App.js

explication du code de composant fonctionnel AutreMessage

Pour utiliser le Hook useState il faut l'importer

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

Pour cet exemple on a besoin de gérer l'état de message et le texte du bouton.

```
const [message, setMessage] = useState("Bien venu visiteur");
```



```
const stateMessage=useState("Bien venu visiteur")
const message=stateMessage[0]
const setMessage=stateMessage[1]
```



ces deux écritures sont identiques
on a utilisé dans la deuxième écriture (Array
destructeur JS ES6)

```
const [message, setMessage] = useState("Bien venu visiteur")
```

```
const [message, setMessage] = useState("Bien venu visiteur")
```



valeur
reactive



setter
méthode

Le hook useState est une fonction fournie par REACT qui vous permet d'ajouter les fonctionnalités de React dans votre composant fonctionnel.

Le hook useState est une fonction qui retourne un Array contenant deux éléments

Le premier élément c'est la propriété message que l'on peut utiliser dans le composant fonctionnel, la propriété message est initialisée par l'argument de la fonction useState, dans ce cas message est initialisé par "Bien venu visiteur".

Le deuxième élément retourné c'est la fonction setMessage.

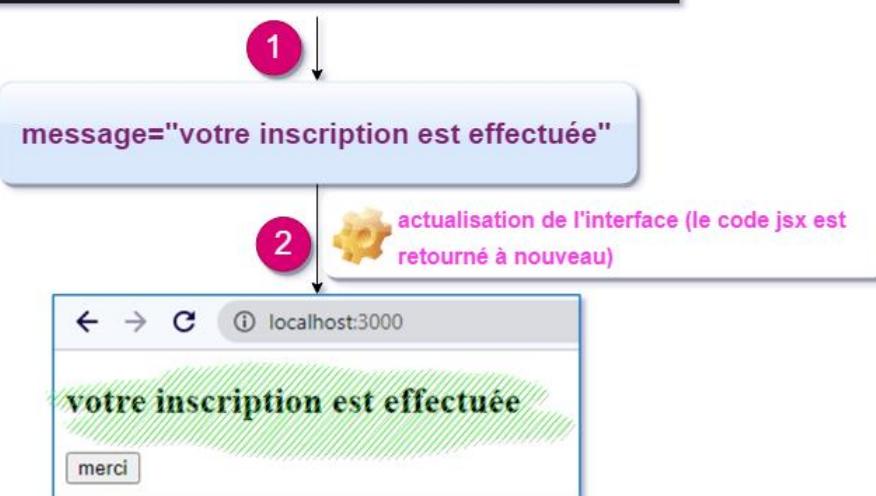
la méthode setMessage permet de changer la valeur de la propriété message puis lance un rafraîchissement de user interface.



```
const [message, setMessage] = useState("Bien venu visiteur")
```



```
setMessage("votre inscription est effectuée");
```



Pour cette écriture

```
<button onClick={() => inscription()}>{btnMessage}</button>
```

Vous allez remarquer qu'on a utilisé un arrow fonction qui fait appel à la fonction inscription après chaque événement click du bouton.

On peut utiliser aussi un callBack fonction

```
<button onClick={inscription}>{btnMessage}</button>
```

```
function inscription() {
  setMessage("votre inscription est effectuée");
  setBtnMessage("merci");
}
```



```
setMessage("votre inscription est effectuée");
```

setMessage modifie la valeur de la propriété message puis déclenche un rafraîchissement de l'interface, de même pour setBtnMessage

Attention si on fait appel de la fonction inscription directement

```
<button onClick={inscription()}>{btnMessage}</button>
```

Vous allez avoir ce message d'erreur sur le console

```
✖ ▶ Uncaught Error: Too many re-renders. React limits the number of renders to prevent an infinite loop.
    at renderWithHooks (react-dom.development.js:16317:1)
    at updateFunctionComponent (react-dom.development.js:19588:1)
    at beginWork (react-dom.development.js:21601:1)
    at HTMLUnknownElement.callCallback (react-dom.development.js:4164:1)
    at Object.invokeGuardedCallbackDev (react-dom.development.js:4213:1)
    at invokeGuardedCallback (react-dom.development.js:4252:1)
```

Ceci est dû à une boucle infinie, en effet la fonction inscription est appelée au chargement de l'interface, cette même fonction inscription contient

Les instructions

```
setMessage("votre inscription est effectuée");
setBtnMessage("merci");
```

qui à leurs tours déclenchent un ré-render de l'interface, et voilà on est dans une boucle infinie

Remarque : Attention Il ne faut pas changer directement les propriétés d'états !!!!

```
let [message, setMessage] = useState("Bien venu visiteur");
let [btnMessage, setBtnMessage] = useState("inscription");

function inscription() {
  message="votre inscription est effectuée";
  btnMessage="merci";
  console.log(message);
  console.log(btnMessage);
```

On a changé const par let

Puis dans la méthode inscription on a changé directement les propriétés d'Etats

On remarque que après le click l'interface n'a pas changé

Le rendu du console :

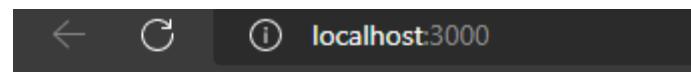
```
votre inscription est effectuée
merci
```

Donc il faut faire Attention !!!!!



Exercice d'application

Réalisation d'un compteur



compteur: 0

Le click sur le bouton incrementer incrémente le compteur

Le click sur le bouton decrementer décrémente le compteur

La valeur du compteur doit être affiché

D'où on a besoin d'une propriété d'état on va la nommée valeur

Composant Compteur.js

```
import React, { useState } from "react";
export default function Compteur(){
  const [valeur, setValue]=useState(0)
  return(
    <div>
      <h1>compteur: {valeur}</h1>
      <input type="button" value="incrementer"/>
      <input type="button" value="decrementer"/>

    </div>
  )
}
```

Index.js

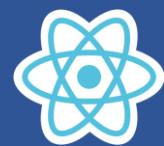
```
import React from "react"
import ReactDOM from "react-dom/client";
import Compteur from "./Compteur";
const root=ReactDOM.createRoot(document.getElementById("root"))
root.render(<Compteur/>);
```

On remarque que l'interface affiche la valeur 0 car

```
const [valeur, setValue]=useState(0)
```

La propriété valeur est initialisé par 0

Maintenant on gère les événements clicks des deux boutons



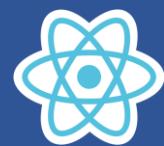
le code devient :

```
import React, { useState } from "react";
export default function Compteur(){
const [valeur,setValue]=useState(0)

function incrementer(){
    setValue(valeur+1)
}
function decrementer(){
    setValue(valeur-1)
}
return(
    <div>
        <h1>compteur: {valeur}</h1>
        <input type="button" value="incrementer" onClick={incrementer}/>
        <input type="button" value="decrementer" onClick={decrementer}/>
    </div>
)
}
```

On peut l'écrire autrement en utilisant un arrow fonction :

```
import React, { useState } from "react";
export default function Compteur(){
const [valeur,setValue]=useState(0)
const incrementer=()=>{setValue(valeur+1)}
const decrementer=()=>{setValue(valeur-1)}
return(
    <div>
        <h1>compteur: {valeur}</h1>
        <input type="button" value="incrementer" onClick={incrementer}/>
        <input type="button" value="decrementer" onClick={decrementer}/>
    </div>
)
}
```



Ou encore

```
import React, { useState } from "react";
export default function Compteur(){
const [valeur, setValeur]=useState(0)

return(
    <div>
        <h1>compteur: {valeur}</h1>
<input type="button" value="incrémenter" onClick={()=>{setValeur(valeur+1)}}/>
<input type="button" value="décrémenter" onClick={()=>{setValeur(valeur-1)}}/>

    </div>
)
}
```

C'est à vous de choisir ;)

**Si on assimile la logique des états (state) en React ça va nous permettre
d'ouvrir les portes de React**



12.2. useEffect :

Le Hook **useEffect**, ajoute la possibilité d'effectuer des effets secondaires à partir d'un fonctionnel composant . Il a le même objectif que **componentDidMount**, **componentDidUpdate** et **componentWillUnmount** dans les classes React, mais unifié en une seule API.

useEffect s'exécute après le rendu initial et après chaque mise à jour, donc par défaut, il est cohérent avec ce qu'il a rendu et vous pouvez désactiver ce comportement si vous le souhaitez pour des raisons de performances et/ou si vous avez une logique spéciale.

useEffect est déclaré à l'intérieur du composant, pour ce la useEffect nous donne un accès aux variables d'état.

Si nous voulons utiliser un useEffect de notre composant, importer **useEffect** à partir de **react**

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

et en suite nous demandons à React quoi faire après chaque changement

```
useEffect(()=>{document.title=nom;},[nom])
```

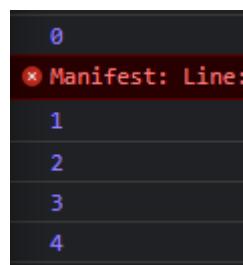
Exemple 1 :

Prenant l'exemple de compteur, on souhaite afficher sur la console la valeur de compteur. pour ce faire, le code doit être exécuté après chaque rendu de l'interface, le bon emplacement du code sera dans useEffect

localhost3000

compteur: 4

incrementer decrementer



```
0
✖ Manifest: Line:
1
2
3
4
```

Le code :

```
import React, { useEffect, useState } from "react";
export default function Compteur(){
const [valeur,setValue]=useState(0)
function incrementer(){setValue(valeur+1)}
function decrementer(){setValue(valeur-1)}
```



```
useEffect(()=>{console.log(valeur)})
  return(
    <div>
      <h1>compteur: {valeur}</h1>
      <input type="button" value="incrémenter" onClick={incrémenter}/>
      <input type="button" value="décrémenter" onClick={décrémenter}/>
    </div>
  )
```

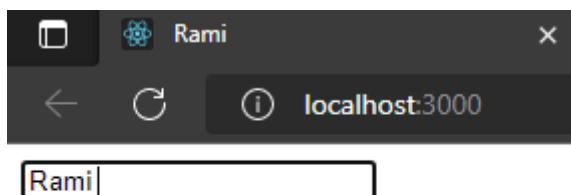
On peut aussi utiliser useEffect avec le deuxième argument :

```
useEffect(()=>{console.log(valeur)},[valeur])
```

comme ça on est sûre que l'exécution se fait seulement si la valeur est modifiée.

Exemple 2 :

Le titre de document va prendre la valeur du nom après chaque modification du nom



Salutation

salut Rami

Le composant fonctionnel Salutation permet d'afficher automatiquement le Nom dans le paragraphe en bas puis aussi dans le titre du document.

Le nom est modifié par l'élément input

Index.js

```
import React from "react";
import ReactDOM from "react-dom/client";
import App from "./App";
const root=ReactDOM.createRoot(document.getElementById("root"))
root.render(<App/>);
```

App.js

```
import React from 'react'

import Salutation from './components/Salutation';
export default function App() {
  return (
    <div>
```



```
<Salutation/>
</div>
);
}
```

Salutation.js

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

export default function Salutation(props){
const [nom,setName]=useState("Rami")
function changeNom(e){
setName(e.target.value)
}
useEffect(()=>{document.title=nom;},[nom])
return(
    <div>
        <input type="text" value={nom} onChange={changeNom}></input>
        <h2>Salutation</h2>
        <p>salut {nom}</p>
    </div>
)
}
```

```
useEffect(()=>{document.title=nom;},[nom])
```

useEffect est exécutée en passant en argument un arrow fonction qui contient le code qui va être executé, Le deuxième argument est un Array optionnel ,si il est omis useEffect s'exécute après le rendu initial et après chaque mise à jour de tous les propriétés states .

Si le deuxième argument est renseigné, useEffect s'execute après le rendu initial et après la modification seulement des propriétés qui se trouvent dans l'Array

Dans notre exemple

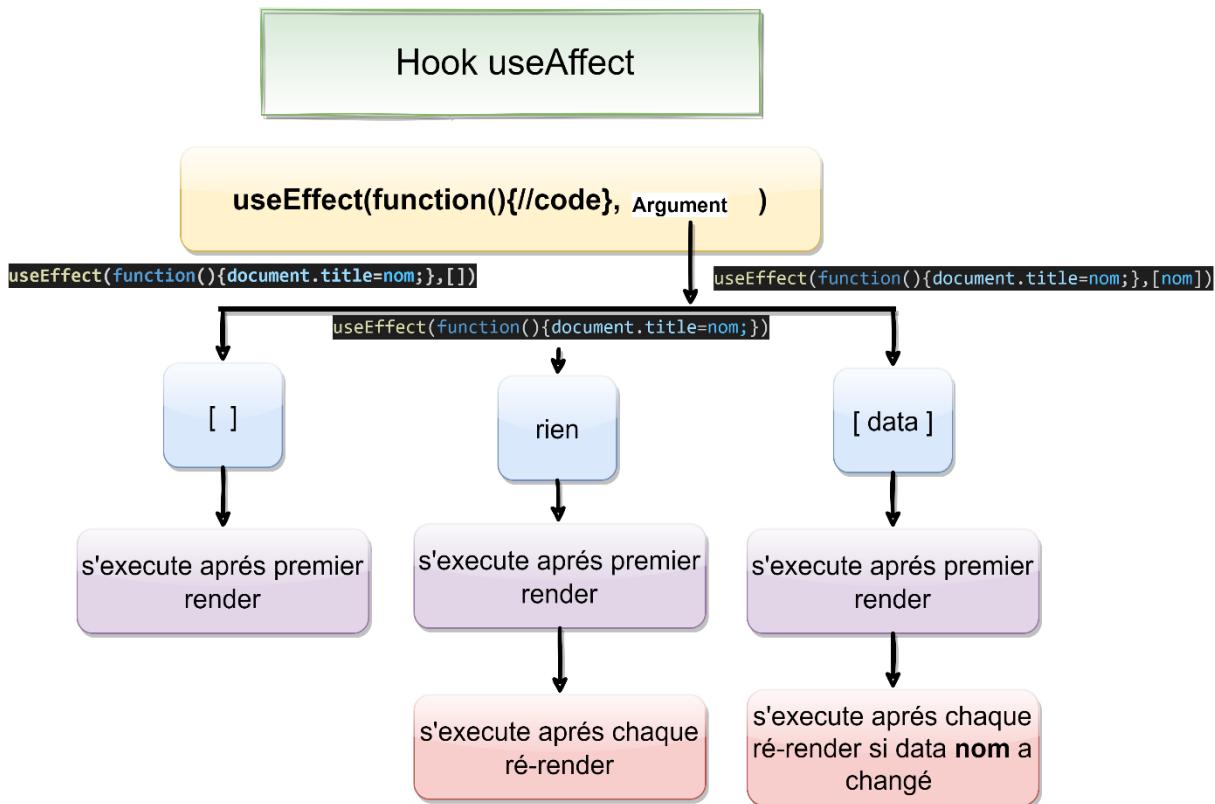
```
useEffect(()=>{document.title=nom;},[nom])
```

useEffect s'exécute après le rendu initial et après la modification seulement de la propriété nom

d'où le code **document.title=nom;** va être exécuté que si la propriété nom est modifiée



Les différents comportements de useEffect selon le deuxième argument



Argument peut être :

- ✓ []
- ✓ [prop1,prop2.....]
- ✓ rien

Exercice d'application

On reprend l'exercice précédent, en ajoutant le prenom et l'âge cette fois ci on souhaite que le titre de la page prend les valeurs nom et le prenom

Au chargement

Le rendu est :

The screenshot shows a browser window with the address bar displaying "localhost:3000". Below the address bar is a navigation bar with back, forward, and search icons. The main content area contains three input fields: "nom:" followed by an input field containing "----", "prénom:" followed by an input field containing "----", and "age:" followed by an input field containing "0".

Salutation

salut ---- vote age est:0

Les propriétés d'états prennent les valeurs d'initialisation

```
const [nom, setNom]=useState("----")
```



```
const [prenom, setPrenom]=useState("---")
const [age, setAge]=useState(0)
```

On remarque que le titre du document prend aussi les valeurs initiales

Après la saisie des informations on a le rendu suivant :



Salutation

salut Rami Ahmed vote age est:33

Le titre du document est mis à jour aussi le message est mis à jour

Le code du composant Salutation

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

export default function Salutation(props){
const [nom, setNom]=useState("---")
const [prenom, setPrenom]=useState("---")
const [age, setAge]=useState(0)
function changeNom(e){
    setNom(e.target.value)
}
function changePrenom(e){
    setPrenom(e.target.value)
}
function changeAge(e){
    setAge(e.target.value)
}
useEffect(function(){document.title=nom+ " "+prenom;},[nom,prenom])
return(
    <div>
    <label>nom:</label>
<input type="text" value={nom} onChange={changeNom}></input>

    <label>prénom:</label>
<input type="text" value={prenom} onChange={changePrenom}></input>
    <label>age:</label>
<input type="text" value={age} onChange={changeAge}></input>
    <h2>Salutation</h2>
    <p>salut {nom} {prenom} vote age est:{age}</p>
    </div>
)
}
```



Analysant ce code

```
useEffect(function(){document.title=nom+" "+prenom;},[nom,prenom])
```

Deuxième argument contient l'Array [nom,prenom]

Dans cette situation le deuxième argument de useEffect est un Array contenant les propriétés d'états nom et prenom, par conséquence useEffect s'exécute au premier rendu et à chaque modification des propriétés d'états nom et prenom

Maintenant si on utilise cette écriture

```
useEffect(function(){document.title=nom+" "+prenom;},[nom])
```

Deuxième argument contient [nom]

Le rendu après avoir renseigné les informations

A screenshot of a browser window titled "Rami ---". The address bar shows "localhost:3000". Below the address bar are three input fields: "nom: Rami", "prénom: Ahmed", and "age: 33".

Salutation

salut Rami Ahmed vote age est:33

useEffect est exécutée au premier rendu et seulement au changement du nom, car le deuxième argument de useEffect est [nom].

Par contre le message est toujours mis à jour

Regardons maintenant si on utilise cette écriture

```
useEffect(function(){document.title=nom+" "+prenom;},[])
```

Deuxième argument contient un Array vide

Le rendu relatif à cette écriture après avoir renseigné les informations :

A screenshot of a browser window titled "----". The address bar shows "localhost:3000". Below the address bar are three input fields: "nom: Rami", "prénom: Ahmed", and "age: 33".

Salutation

salut Rami Ahmed vote age est:33

Vous allez remarquer que useEffect est exécuté seulement au premier rendu,

En fin si on utilise cette écriture :

```
useEffect(function(){document.title=nom+" "+prenom+" "+age;})
```

useEffect sans deuxième argument



Le rendu relatif à cette écriture après avoir renseigné les informations :



A screenshot of a browser window titled "Rami Ahmed 33". The address bar shows "localhost:3000". Below the address bar are three input fields: "nom: Rami", "prénom: Ahmed", and "age: 33".

Salutation

salut Rami Ahmed vote age est:33

Vous allez remarquer que useEffect maintenant est executé au premier rendu et a tous les mis à jour des propriétés d'état nom, prenom et age

Quiz :

Question 1 :

Jetez un œil au code suivant. Après l'avoir exécuté, combien de console.log vous attendriez-vous à voir, et quand les verriez-vous ?

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

import ReactDOM from 'react-dom';

const App = () => {

  useEffect(() => {

    console.log('TEST!');

  }, []);

  return <div>test composant</div>;
};

ReactDOM.render(<App />, document.querySelector('#root'));
```

- a- Je vais voir un log de 'TEST', il est affiché après le composant est rendu
- b- Je ne vais voir aucun log
- c- Je vais voir un log de 'TEST', il est affiché avant le composant est rendu
- d- Je vais voir deux log de 'TEST',ils sont affichés après le composant est rendu



Question 2 :

Jetez un œil au code suivant. Imaginez qu'il soit exécuté, puis qu'un utilisateur a cliqué trois fois sur l'élément bouton. Combien d'instructions de log 'TEST' vous attendriez-vous à voir imprimées ?

```
import React, { useEffect, useState } from 'react';
import ReactDOM from 'react-dom';
const App = () => {
  const [count, setCount] = useState(0);

  useEffect(() => {
    console.log('TEST!');
  }, []);

  const onClick = () => {
    setCount(count + 1);
  }

  return (
    <div>
      <h1>Count: {count}</h1>
      <button onClick={onClick}>Click me!</button>
    </div>
  );
}

ReactDOM.render(<App />, document.querySelector('#root'));
```

- a- Je vais voir trois console.log 'TEST'
- b- Je vais voir quatre console.log 'TEST'
- c- Je vais voir un console.log 'TEST'

Question 3 :

Jetez un œil à l'extrait de code suivant. C'est identique à la dernière question, sauf que maintenant useEffect a un tableau avec un seul argument à l'intérieur.

Combien de fois vous attendez-vous à voir l'instruction de console.log 'TEST' après qu'un utilisateur a cliqué trois fois sur le bouton ?

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

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

  useEffect(() => {console.log('TEST!')}, [count]);

  const onClick = () => {
    setCount(count + 1);
  }

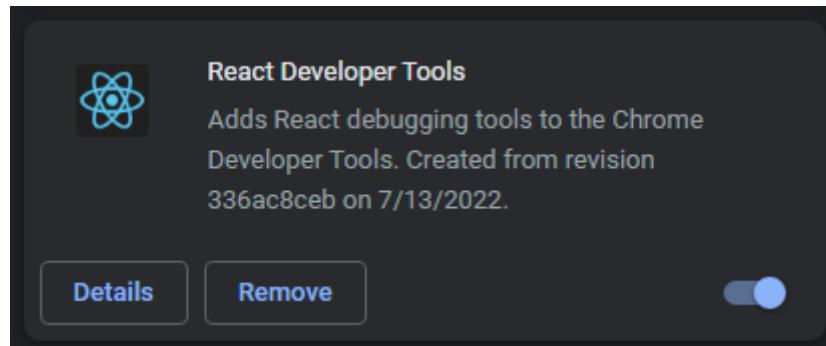
  return (
    <div>
      <h1>Count: {count}</h1>
      <button onClick={onClick}>Click me!</button>
    </div>
  );
}

ReactDOM.render(<App />, document.querySelector('#root'));
```

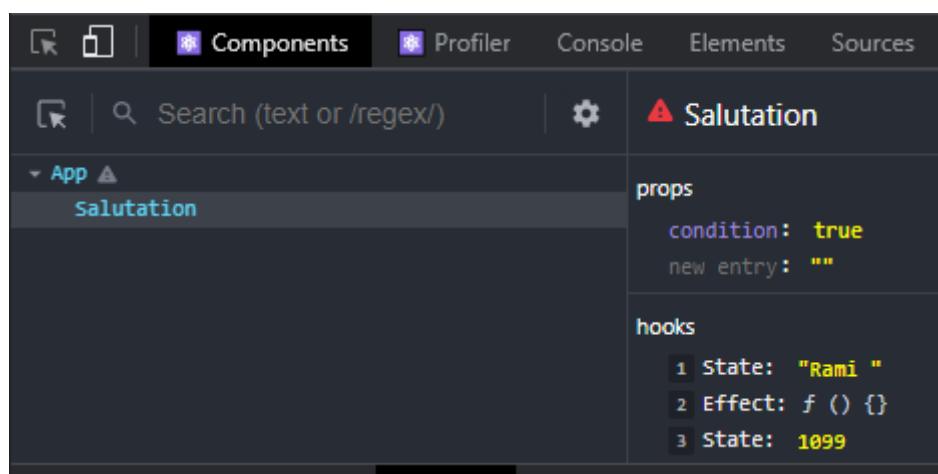
- a- Je vais voir un console.log 'TEST'
- b- Je vais voir deux console.log 'TEST'
- c- Je vais voir trois console.log 'TEST'
- d- Je vais voir quatre console.log 'TEST'



L'extension React Developer Tools, permet de faire le debugging



Voir le volet Component



12.3. Utilisation de useEffect pour charger les données d'un API

Le chargement des données provenant d'un API se fait seulement au chargement de user interface c-à-d au premier rendu du composant

Soit le endPoint suivant :

<https://jsonplaceholder.typicode.com/posts>

qui retourne un array des posts sous la forme

```
[  
  {  
    "userId": 1,  
    "id": 1,  
    "title": "sunt aut facere repellat provident occaecati excepturi",  
    "body": "test"  
  },  
  {  
    "userId": 1,  
    "id": 2,  
    "title": "qui est esse",  
    "body": "test2"  
  },.....]
```



Soit l'interface



liste des posts à partir d'un API

- sunt aut facere repellat provident occaecati excepturi optio reprehenderit
- qui est esse
- ea molestias quasi exercitationem repellat qui ipsa sit aut
- eum et est occaecati
- nesciunt quas odio
- dolorem eum magni eos aperiam quia
- magnam facilis autem

App.js

```
import React, { useEffect, useState } from "react";
//import usePosts from './hooks/usePosts'
export default function App(){
const [posts, setPosts] = useState([])
useEffect(()=>{
  fetch('https://jsonplaceholder.typicode.com/posts')
    .then(res=>res.json())
    .then(response=>setPosts(response))
},[])
return (<div>
  <h1>liste des posts à partir d'un API</h1>
  <ul>
    {posts.map(post=>{
      return (<li key={post.id}>{post.title}</li>)
    })}
  </ul>
</div>
)}
```

Index.js

```
import React from "react";
import ReactDOM from "react-dom/client";
import App from "./App";
const root=ReactDOM.createRoot(document.getElementById("root"))
root.render(<App/>);
```

Comme vous remarquez on fait l'appel de fetch dans la fonction callback de useEffect, le deuxième argument de useEffect contient [], de ce fait fetch est exécuté seulement au premier chargement de user interface, par conséquent si on effectue des mise à jour de propriétés state, fetch ne sera pas réexécuté.



12.4. Construire vos propres Hooks

Les customs Hooks ou bien les Hooks personnalisés sont des fonctions réutilisables.

Lorsque vous avez une logique de composant qui doit être utilisée par plusieurs composants, nous pouvons extraire cette logique dans un Hook personnalisé.

Les Hooks personnalisés commencent par convention par "use". Exemple : usePosts.

Dans l'exemple précédent la logique de récupération des Posts par fetch peut être utilisé par plusieurs composants, pour ce faire on externalise cette logique dans Hooks usePosts.

Par convention on met le fichier usePosts.js dans le dossier hooks

usePosts.js

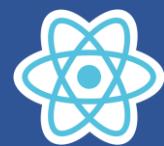
```
import React,{useState,useEffect} from "react";

function usePosts(){
const [posts,setPosts]=useState([])
useEffect(()=>{
fetch('https://jsonplaceholder.typicode.com/posts')
.then(res=>res.json())
.then(response=>setPosts(response))

},[])
return posts
}
export default usePosts
```

App.js

```
import React, { useEffect, useState } from "react";
import usePosts from "./hooks/usePosts";
export default function App(){
const posts=usePosts()
return (<div>
<h1>liste des posts à partir d'un API</h1>
<ul>
{posts.map(post=>{
    return (<li key={post.id}>{post.title}</li>)
})}
</ul>
</div>
)
```



usePosts.js

```
import React,{useState,useEffect} from "react";
function usePosts(){
  const [posts,setPosts]=useState([])
  useEffect(()=>{
    fetch('https://jsonplaceholder.typicode.com/posts')
    .then(res=>res.json())
    .then(response=>setPosts(response))
  },[])
  return posts
}
export default usePosts
```

App.js

```
import React, { useEffect, useState } from "react";
import usePosts from "./hooks/usePosts";
export default function App(){
  const posts=usePosts()
  return (<div>
    <h1>liste des posts à partir d'un API</h1>
    <ul>
      {posts.map(post=>{
        return (<li key={post.id}>{post.title}</li>)
      })}
    </ul>
  </div>
)
```

```
import usePosts from "./hooks/usePosts";
```

```
const posts=usePosts()
```

Posts fait référence à posts retourné par la fonction **usePosts** du hook usePosts.js

Par conséquent chaque composant qui nécessite cette logique peut utiliser ce hook usePosts



13. Gestion du routage

13.1. Qu'est-ce que le routage ?

Le routage est la capacité d'afficher différentes pages à l'utilisateur. Cela signifie qu'il donne la possibilité de naviguer entre les différentes parties d'une application en entrant une URL ou en cliquant sur un élément.

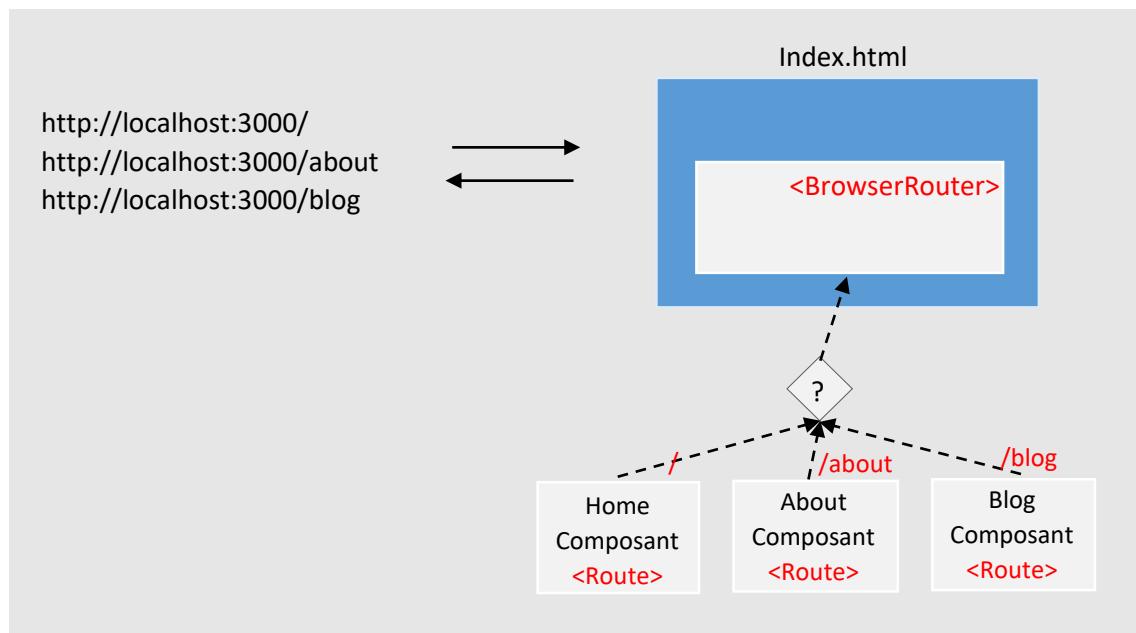
L'idée de Router (Routeur) est tellement utile car vous travaillez avec React, une bibliothèque Javascript pour programmer des applications d'une seule page (Single Page Application). Afin de développer une application React il vous faut écrire plusieurs Component mais il a besoin d'un seul fichier afin de servir des utilisateurs, c'est index.html.

Par défaut, React vient sans routage. Et pour l'activer, nous devons ajouter une bibliothèque nommée react-router.

Pour l'installer, vous devrez exécuter la commande suivante dans votre terminal :

```
npm install react-router-dom
```

Le React Router vous permet de définir des URL dynamiques et de sélectionner un Component approprié pour render (afficher) sur le navigateur d'utilisateur en correspondance à chaque URL.



13.2. <BrowserRouter> vs <HashRouter>

Le React Router vous fournit deux composants tels que <BrowserRouter> & <HashRouter>. Ces deux composants sont différents dans le type de URL qu'ils créent et synchronisent avec.

```
// <BrowserRouter>
http://example.com/about
// <HashRouter>
http://example.com/#/about
```



<BrowserRouter> est utilisé plus couramment, il utilise le History API incluse dans HTML5 pour surveiller l'historique de votre routeur tandis que <HashRouter> utilise le hash de l'URL (window.location.hash) pour tout mémoriser. Si vous avez l'intention de soutenir des anciens navigateurs, vous devez être scellé contre le <HashRouter> ou créer une application React à l'aide du routeur côté client. <HashRouter> est un choix raisonnable.

13.3. <Route>

Le composant <Route> définit une relation (mapping) entre une URL et un Component. Cela signifie que lorsque l'utilisateur visite une URL sur le navigateur, un Component correspondant doit être rendu sur l'interface.

```
<BrowserRouter>
  <Route exact path="/" component={Home}/>
  <Route path="/about" component={About}/>
  <Route path="/topics" component={Topics}/>
</BrowserRouter>
```

L'attribut exact est utilisé dans le <Route> afin de dire que ce <Route> ne fonctionne que si la URL sur le navigateur correspond absolument à la valeur de son attribut path.

13.4. Exemple :

Créer un projet 13-router et installer react-router

```
npx create-react-app 13-routage
cd 13-routage
install --save react-router-dom
```

Supprime tous les contenus des deux fichiers [App.css](#) & [App.js](#), nous allons écrire le code de ces deux fichiers.

[App.css](#)

```
.main-route-place {
  border: 1px solid #bb8fce;
  margin: 10px;
  padding: 5px;
}
```

[App.js](#)

```
import React, { Component } from 'react';
import './App.css';
import { Routes, Route, Link } from 'react-router-dom';
class App extends Component {
  render() {
    return (
      <div>
        <ul>
          <li>
```



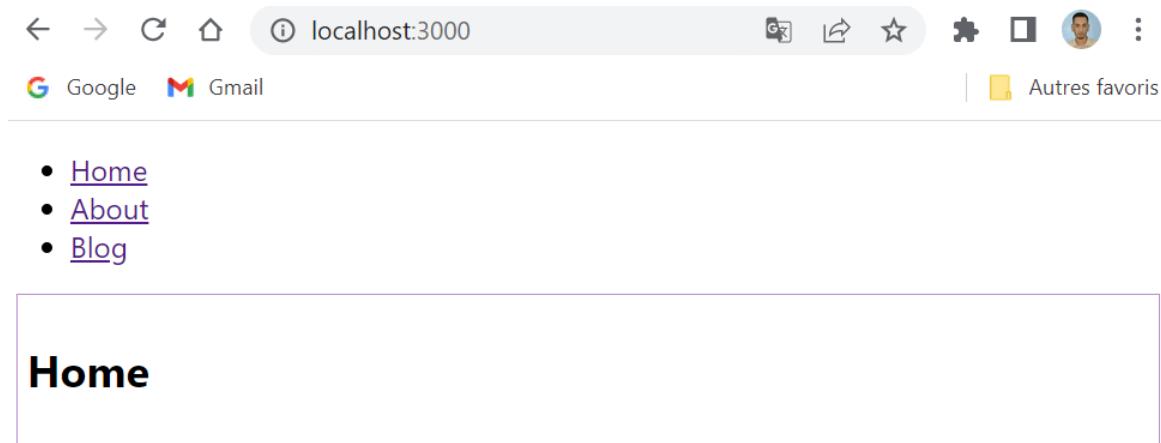
```
<Link to="/">Home</Link>
</li>
<li>
    <Link to="/about">About</Link>
</li>
<li>
    <Link to="/blog">Blog</Link>
</li>
</ul>
<div className="main-route-place">
    <Routes>
        <Route path="/" element={<Home />} />
        <Route path="/about" element={<About />} />
        <Route path="/blog" element={<Blog />} />
    </Routes>
</div>
</div>
);
}
}
class Home extends React.Component {
render() {
    return (
        <div>
            <h2>Home</h2>
        </div>
    );
}
}
class About extends React.Component {
render() {
    return (
        <div>
            <h2>About</h2>
        </div>
    );
}
}
class Blog extends React.Component {
render() {
    return (
        <div>
            <h2>Blog</h2>
        </div>
    );
}
}
export default App;
```



Index.js

```
import React from 'react';
import ReactDOM from 'react-dom/client';
import './index.css';
import App from './App';
import { BrowserRouter } from "react-router-dom";
const root = ReactDOM.createRoot(document.getElementById('root'));
root.render(
  <BrowserRouter>
    <App />
  </BrowserRouter>
);
```

Résultat :



localhost:3000

- [Home](#)
- [About](#)
- [Blog](#)

Home



14.Tester une application React

14.1. Que tester dans une application React ?

Sur une application front-end, qu'elle soit en React ou non, la question se pose régulièrement de savoir quoi tester unitairement. En effet, si l'on prend par exemple un composant qui n'est responsable que de présentation en générant du HTML, il serait laborieux de tester précisément le rendu du composant dans le navigateur. De plus, il est possible que le moindre changement dans le CSS associé au composant change le rendu au point de mettre le test en échec, ce qui n'est généralement pas le but.

Pas certains aspects, il est tout de même intéressant de tester quelques composants :

- pour tester les informations affichées
- pour tester le comportement du composant en réponse à des actions de l'utilisateur.

14.2. Test unitaire de composants avec Enzyme

Enzyme est un utilitaire de test JavaScript permettant de tester facilement les composants React. Il aide à rendre les composants React en mode test.

Pour démarrer avec Enzyme, installez-le via npm avec la commande suivante.

```
npm install --save-dev enzyme
npm install --save-dev enzyme-adapter-react-16 --force
```

Rédaction du premier cas de test

Étape 1 : Nous allons rendre un simple bouton nommé **Click Me** en utilisant le code suivant.

```
import React, { Component } from 'react';
import './App.css';

class App extends React.Component {
  constructor(props) {
    super(props);
    this.state = {};
  }
  render() {
    return (
      <div>
        <button className="click-me" id="ClickMe">Click Me</button>
      </div>
    )
  }
}

export default App;
```



Étape 2 : Ajoutez le code suivant dans le fichier **App.test.js**, qui est le fichier dans lequel nous écrivons les cas de test.

```
import React from 'react'
import Enzyme, { shallow } from 'enzyme'
import Adapter from 'enzyme-adapter-react-16'
import App from './App'

Enzyme.configure({ adapter: new Adapter() })

describe('Test Case For App', () => {
  it('should render button', () => {
    const wrapper = shallow(<App />)
    const buttonElement = wrapper.find('#ClickMe');
    expect(buttonElement).toHaveLength(1);
    expect(buttonElement.text()).toEqual('Click Me');
  })
})
```

Étape 3 : Utilisez la commande suivante pour exécuter les scénarios de test.

```
npm test
```

Les résultats du test seront affichés comme dans la capture d'écran suivante.

```
PASS  src/App.test.js
      Test Case For App
        ✓ should render button (9 ms)

      Test Suites: 1 passed, 1 total
      Tests:       1 passed, 1 total
      Snapshots:  0 total
      Time:       2.526 s
      Ran all test suites.

      Watch Usage: Press w to show more.
```



Scénario de test pour la variable d'état

Créons un nouveau cas de test pour vérifier l'état désactivé/activé du bouton à l'aide de la variable d'état.

Étape 1 : Ajoutez l'extrait de code ci-dessous dans le fichier **App.js**.

```
import React, { Component } from 'react';
import './App.css';

class App extends React.Component {
  constructor(props) {
    super(props);
    this.state = {
      ClickCount:0,
      IamDisabled : true
    };
    this.ClickMe = this.ClickMe.bind(this);
  }
  ClickMe(){
    this.setState({
      ClickCount:this.state.ClickCount + 1
    });
  }
  render() {
    return (
      <div>
        <button className="click-me" id="ClickMe"
onClick={this.ClickMe}>Click Me</button>
        <p>You clicked me :: {this.state.ClickCount}</p>
        <button className="click-me"
disabled={this.state.IamDisabled}>Disabled</button>
      </div>
    )
  }
}
export default App;
```



Étape 2 : Ajoutez l'extrait de code suivant dans le fichier App.test.js.

```
import React from 'react'
import Enzyme, { shallow } from 'enzyme'
import Adapter from 'enzyme-adapter-react-16'
import App from './App'

Enzyme.configure({ adapter: new Adapter() })

describe('Test Case For App', () => {
  it('should render button', () => {
    const wrapper = shallow(<App />)
    const buttonElement = wrapper.find('#ClickMe');
    expect(buttonElement).toHaveLength(1);
    expect(buttonElement.text()).toEqual('Click Me');
  }),

  it('increments count by 1 when button is clicked', () => {
    const wrapper = shallow(<App />);
    const buttonElement = wrapper.find('#ClickMe');
    buttonElement.simulate('click');
    const text = wrapper.find('p').text();
    expect(text).toEqual('You clicked me :: 1');
  });
})

describe('Test Case for App Page', () => {
  test('Validate Disabled Button disabled', () => {
    const wrapper = shallow(
      <App />
    );
    expect(wrapper.state('IamDisabled')).toBe(true);
  });
})
```

Les résultats du test seront affichés comme dans la capture d'écran suivante.

```
PASS  src/App.test.js
  Test Case For App
    ✓ should render button (12 ms)
    ✓ increments count by 1 when button is clicked (3 ms)
  Test Case for App Page
    ✓ Validate Disabled Button disabled (1 ms)

  Test Suites: 1 passed, 1 total
  Tests:       3 passed, 3 total
  Snapshots:   0 total
  Time:        2.637 s
  Ran all test suites related to changed files.
```



16. Concevoir une application avec Redux

16.1. Qu'est-ce que Redux ?

En 2013, Facebook a dit que AngularJS de Google est lent et lourd, donc en cette année-là, il a introduit ReactJS à la communauté des développeurs. Pour le ReactJS était seulement une bibliothèque pour créer des Component et rendre ces Component sur l'interface. Le ReactJS n'avait pas la capacité de gérer l'état des applications. Peu de temps après, Facebook a introduit une bibliothèque Javascript nommé **Flux** pour gérer l'état des applications et elle est une bibliothèque créée pour soutenir le React.

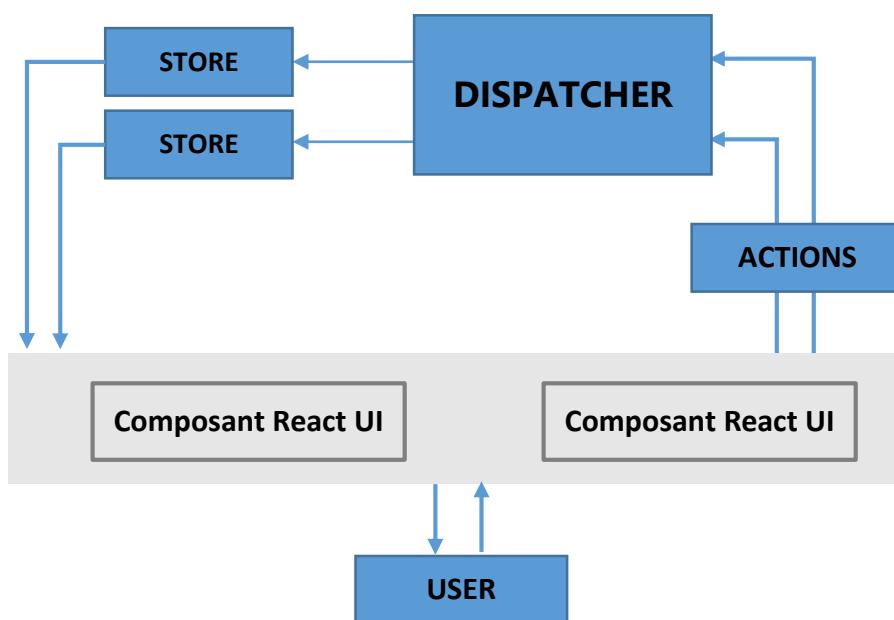
En ce temps-là, Dan Abramov a étudié le Flux de Facebook et le langage ELM. Il était influencé par l'architecture ELM, et a trouvé la complexité de Flux. En mai 2015 Dan Abramov a annoncé une nouvelle bibliothèque appelée **Redux**, elle est basée sur l'architecture de ELM et éliminé la complexité de Flux.

Après sa naissance, le Redux a suscité un vif écho et a immédiatement attiré l'attention de la communauté React et même Facebook a également invité Dan Abramov à travailler. Actuellement, le Redux et le Flux existent en parallèle, mais le Redux est populaire et plus largement utilisé.

16.2. L'architecture de Flux

L'architecture de Flux était introduit pour la première fois par Bill Fisher et Jing Chen à la conférence Facebook F8 en 2014. L'idée est redéfini le modèle MVVM (Model View - View Model) qui était largement utilisé auparavant avec le concept de "flux de données unidirectionnel" (unidirectional data flow).

Des actions (actions) et des événements (events) dans le Flux vont passer par un "circuit fermé" avec la forme ci-dessous :





Les parties dans l'architecture de FLUX :

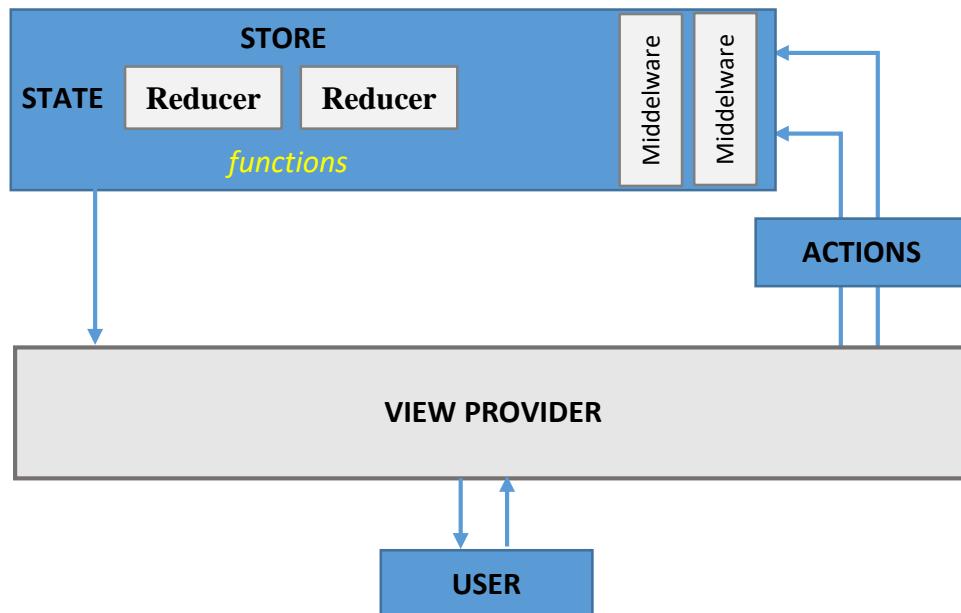
- **VIEW** : une composition hiérarchique (hierarchical composition) des React Component.
- **ACTION** : Est un objet pur créé pour stocker les informations relatives à un événement de l'utilisateur (Clique sur l'interface,...), il comprend les informations telles que : type de l'action, temps et location, ses coordonnées et quel state qu'il vise à changer.
- **DISPATCHER** : Le seul point de l'application à recevoir des objets Action à gérer.
- **STORE** : Store écoute des Action, gère des données et l'état de l'application.
Les Store basent sur les objets action pour répondre au USER INTERFACE correspondant.

16.3. L'architecture de Redux

Redux apprend l'architecture de Flux mais il omit la complexité inutile.

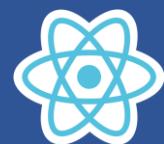
- Redux n'a pas de concept DISPATCHER.
- Redux n'a que STORE au lieu de plusieurs STORE comme le Flux.
- Les objets Action sera recus et gérés directement par STORE.

Ci-dessous l'illustration de l'architecture de REDUX :



Les parties dans l'architecture de REDUX :

- **VIEW PROVIDER** : Représente un View Framework pour inscrire avec STORE. Dans lequel, View Framework peut être React ou Angular,...
- **ACTION** : un objet pur créé pour stocker les informations relatives à l'événement d'un utilisateur (cliquez sur l'interface, ...). Il inclut les informations telles que: le type d'action, l'heure de l'événement, l'emplacement de l'événement, ses coordonnées et quel state qu'il vise à modifier.



- **STORE** : Gère l'état de l'application et a fonction dispatch (action).
- **MIDDLEWARE** : (Logiciel intermédiaire) Fournit un moyen d'interagir avec les objets Action envoyés à STORE avant leur envoi à REDUCER. A Middleware, vous pouvez effectuer des tâches telles que la rédaction de journaux, la génération d'erreurs, la création de requêtes asynchrones (asynchronous requests) ou la distribution (dispatch) de nouvelles actions, ...
- **REDUCER** : (Modificateur) Une fonction pure pour renvoyer un nouvel état à partir de l'état initial. Remarque : REDUCER ne modifie pas l'état de l'application. Au lieu de cela, il créera une copie de l'état d'origine et le modifiera pour obtenir un nouvel état.

Store

Redux permet de stocker dans un seul objet appelé "the store" tous les états (state) de l'application. Cet objet est "la seule source de vérité" (single source of truth) et il est accessible par tous les composants.

L'objet store possède trois méthodes :

- **subscribe** qui permet à tout écouteur (listener) d'être notifié en cas de modification du store. Les gestionnaires de vues (comme React) vont souscrire au store pour être notifié des modifications et effectuer mettre à jour l'interface graphique en conséquence.
- **dispatch** qui prend en paramètre une action et exécute le reducer qui va, à son tour, mettre à jour le store avec un nouvel état.
- **getState** qui retourne l'état courant du store. L'objet retourné ne doit pas être modifié.

Reducer

Pour mettre à jour le **store**, il n'est pas possible de le faire directement puisque redux s'inscrit dans le paradigme de la programmation fonctionnelle et donc de l'immuabilité.

On va utiliser la fonction appelée "**reducer**" qui reçoit en arguments le "state" et une "action" et qui retourne le nouveau "state" en fonction de l'"action". C'est au programmeur de décider quelle technique il emploie pour assurer l'immuabilité : spread operator, assign, immer (produce)

Il y aura en général plusieurs "reducer" dans une application (un par composant).

Action

Action est un **objet littéral** (plain object) qui représente ce qu'il vient de se passer. On peut l'assimiler à un événement. Cet objet (événement) est envoyé en argument à la méthode "**dispatch**" de l'objet "**store**". La méthode "dispatch" est l'unique point d'entrée du "store" ce qui va permettre de contrôler plus facilement les actions des utilisateurs. C'est grâce à cela que l'on va pouvoir "loger" toutes les modifications de l'interface (cf Redux dev tools) ou que l'on va pouvoir facilement mettre en place des mécanismes de "défaire et refaire".



17. Manipuler des states complexes d'une application React avec Redux

17.1. Redux dans une app React

Dans la séance précédente nous avons découvert Redux. Il nous a permis de **mieux organiser la logique** des applications grâce à un système en trois parties : state, actions et reducer.

Nous avons également vu le store qui permet de lier ces éléments et de les faire fonctionner ensemble.

Pour utiliser React et Redux ensemble de manière efficace, on utilise une librairie tierce : React-Redux.

Pour utiliser React-Redux, il faut :

- Envoyer le **store** grâce au **Provider** qui englobe toute l'application.
- Utiliser **useDispatch** pour envoyer des actions depuis les composants.
- Utiliser **useSelector** pour extraire des morceaux de state et mettre à jour le composant en cas de changement de state.

17.2. State avec Immer

En Javascript, il existe 2 types de variables : les variables primitives (Booléen, Null, Undefined, Nombre, BigInt, Chaîne de caractères / String, Symbole ; qui sont non-mutables) et les objets prédéfinis (array, function, object, ... qui sont donc mutables).

Le contenu des objets prédéfinis peuvent donc être modifiés après leur initialisation. Par exemple, si nous définissons un objet :

```
const myObject = { test: true };
```

Nous pouvons très bien, ajouter ou supprimer des attributs par la suite :

```
myObject.otherAttribute = false;
```

Dans ce cas, l'objet garde la même référence en mémoire, il ne sera donc pas possible de savoir si l'objet a été modifié ou non après coup.

Il en est de même pour les tableaux, l'ajout de nouveaux éléments à un tableau ne permet pas de savoir s'il a modifié ou non car la référence est toujours la même.

Par défaut, JS ne propose de solution pour rendre les tableaux et objets immutables (bien que l'utilisation de proxy permette de contourner un peu le problème). De plus, certaines fonctions sont piégeuses car elles vont modifier le tableau. Nous pouvons citer :

- Ajout/suppression d'un élément : **push**, **shift**, **unshift**, **pop**, **delete**
- Ajout/suppression de plusieurs éléments : **splice**
- Modification de l'ordre : **reverse**, **sort**



Alors que les fonctions suivantes renvoient un nouveau tableau :

- Ajout d'éléments : **concat**
- Opération sur les éléments : **map, filter, reduce**
- Copie superficielle du tableau : **slice**

Pour éviter les problèmes de fonctions qui modifient ou non les tableaux ou objets, nous allons utiliser la librairie **immer** : <https://immerjs.github.io/immer/>

D'après les séances précédentes, Redux est basé sur 3 principes :

- Une unique source de vérité : il y a un seul état global dans votre application
- L'état global est accessible uniquement en lecture seule
- Les modifications de l'état global sont réalisées uniquement à l'aide de fonctions pures

Les 2 types de méthodes utilisées par Redux sont :

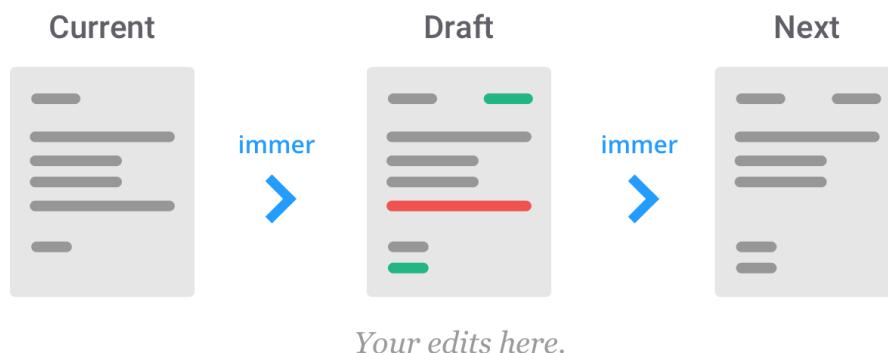
- Les **actions** : effectuent les appels externes (appels API, ...)
- Les **reducers** : mettent à jour l'état global à partir des nouveaux données reçues dans les actions

Chaque méthode dans un reducer doit être une fonction "pure", c'est-à-dire, elle ne peut pas :

- **Modifier les arguments en paramètres**
- Utiliser des variables globales
- Modifier des variables globales
- Aucun appel externe (comme API)
- Appeler une autre fonction non "pure" (comme les dates ou des nombres aléatoires)

Pour résoudre le problème principal de modification des données, par inadvertance, contenues dans l'état global, il est fortement conseillé de se reposer sur des données immutables.

La librairie **Immer** va permettre de **modifier les données** comme si elles étaient de simple tableau ou objet mais sans impacter directement l'état global.





Un exemple pour la comparaison ;

```
const baseState = [
  {
    title: "Learn TypeScript",
    done: true
  },
  {
    title: "Try Immer",
    done: false
  }
]
```

Imaginez que nous avons l'état de base ci-dessus et que nous devons mettre à jour la deuxième tâche et ajouter une troisième. Cependant, nous ne voulons pas muter **baseState** d'origine, et nous voulons également éviter le clonage en profondeur (pour préserver la première tâche).

```
▼ (3) [{...}, {...}, {...}] ⓘ
▶ 0: {title: 'Learn TypeScript', done: true}
▶ 1: {title: 'Try Immer', done: true}
▶ 2: {title: 'Tweet about it'}
  length: 3
▶ [[Prototype]]: Array(0)
```

Sans Immer

Sans Immer, nous devrons copier chaque niveau de la structure d'état qui est affecté par notre changement :

```
const nextState = baseState.slice() // shallow clone the array
nextState[1] = {
  // replace element ...
  ...nextState[1], // with a shallow clone of element 1
  done: true // ...combined with the desired update
}
nextState.push({title: "Tweet about it"})
```

Avec Immer

```
import produce from "immer"
const nextState = produce(baseState, draft => {
  draft[1].done = true
  draft.push({title: "Tweet about it"})
})
```



17.3. Les sélecteurs

- Un selector est une fonction que l'on passe à **useSelector** pour extraire un morceau du state.
- Il est possible d'utiliser les props dans les selectors, cela permet de faire des composants dynamiques.
- Pour simplifier les composants, on peut déclarer les selectors dans un fichier séparé ; si le composant utilise des props, il faut utiliser une fonction qui retourne le selector.
- Il ne faut jamais créer de référence dans les selectors, car React-Redux va croire qu'ils changent à chaque changement de state.

17.4. Les événements asynchrones

- Redux n'est pas capable de manipuler des événements asynchrones, mais on peut interagir avec dans une fonction asynchrone.
- Dans les composants React, on peut utiliser le hook **useStore** pour accéder au store et utiliser **getState** et **dispatch** dans notre action.
- Il est recommandé de déclarer les actions asynchrones dans des fonctions séparées (dans le fichier **store.js**, par exemple) ; il faut alors passer le store en paramètre.



17.1. Manipulation des states complexes

REDUX est une librairie pour gestion d'état pour Java script application

REACT



ANGULAR



VUE



VANILLA JS



REDUX

Redux n'est pas lié à une librairie ou Framework User Interface

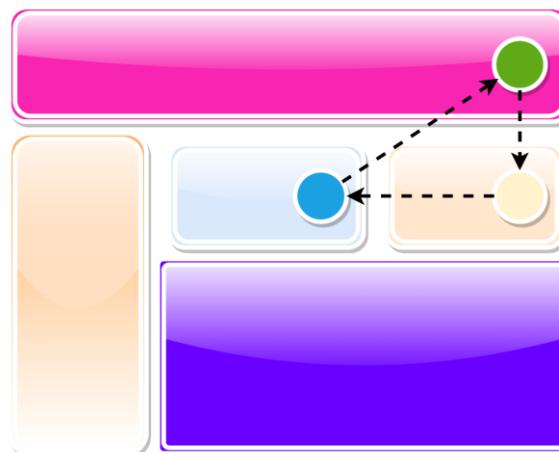
Mais pourquoi on a besoin à un gestionnaire d'état comme REDUX

Quand on crée une application avec un complexe UI, l'application est constituée par plusieurs composants dont on a besoin de garder leurs états asynchrones, un changement d'état d'un composant pourra provoquer immédiatement la modification d'état d'autres composants.

Dans ce cas la modification des données dans un composant, déclenchera la modification d'état d'autres composants.

Dans le cas d'un scénario complexe, les données peuvent être aussi modifiées par une arrivée des données provenant d'une requête réseau API ou d'un background tâche.

Dans cette situation les données peuvent circuler d'une partie à une autre d'UI, puis les données sont changées d'une manière imprévisible, en conséquence il faut écrire beaucoup de code pour garder l'état des composants asynchrone. Quand quelque chose ne va pas, il faut comprendre d'où elles viennent les données, chose qui est très complexe à faire directement avec du code





SOLUTION

Utilisation d'un gestionnaire d'état

En Redux au lieu de stocker les données sur plusieurs partie de l'UI, en stocke les données dans un centrale repository il s'agit d'un objet java script nommé store, on peut le considérer comme une petite base de données pour le front end, en conséquence pour cette architecture, les différentes partie de l'UI

ne maintiendront plus leur propre état au lieu de cela, ils obtiendront ce dont ils ont besoin dans le **store**.

si nous devons mettre à jour les données, il y a un seul endroit à mettre à jour dans le store, ce qui résout immédiatement le problème de synchronisation des données entre différentes parties de l'UI

l'architecture redux permet également de comprendre facilement comment les données changent dans nos applications si quelque chose ne va pas, nous pouvons voir exactement comment les données ont changé pourquoi quand et où.

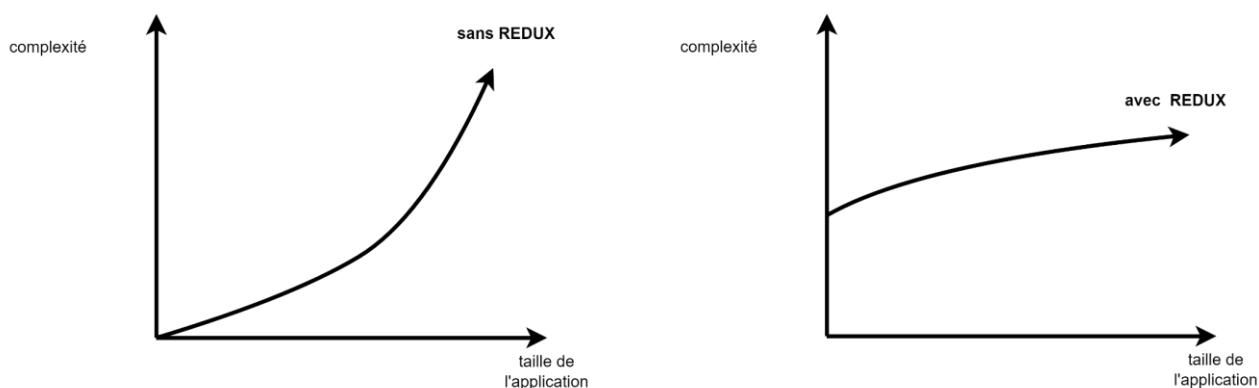
Avantages REDUX :

AVANTAGES

- ✓ Changements d'état prévisibles
- ✓ état centralisé
- ✓ Débogage facile
- ✓ conserver l'état de la page
- ✓ undo/redu
- ✓ modules complémentaires de l'écosystème angular,vue,react ,vanilla js
- ✓

Inconvénients :

- ✓ Complexité
- ✓ Verbosité

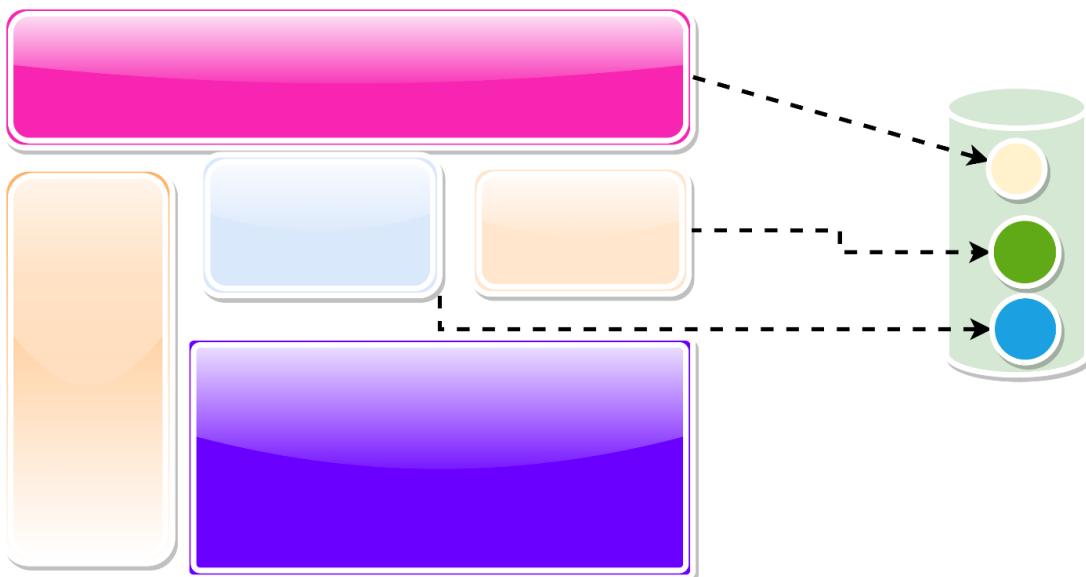


Pour tout projet ou application REDUX, nous devons déterminer le problème qu'on doit résoudre, quelles sont les contraintes, quelle est la solution optimale pour le résoudre efficacement



Quand on ne doit pas utiliser REDUX :

- application de petite ou moyenne taille
- budget serré
- petit flux de données d'interface utilisateur
- données statiques



L'idée de Redux est de centraliser la gestion des états dans un objet unique appelé store, qui centralise tous les états de tous les composants. Cet objet store (unique pour toute l'application) est géré par les méthodes de Redux.

17.2. Utilisation de REDUX

`createStore`

La fonction `createStore(reducer)` prend en paramètre une fonction de callback (appelée un reducer) ayant les paramètres `state` et `action`. La fonction `createStore(reducer)` retourne un objet appelé store.

La fonction de callback utilisée en paramètre de `createStore()` est donc appelée un reducer. On étudie ci-après cette fonction.

Les données de store ne peuvent être lu et mis à jour qu'à travers les méthodes suivantes, fournies sur l'objet store

`store.dispatch(action)`

La méthode `store.dispatch(action)` qui permet d'effectuer une modification de l'état, en fonction de l'action indiquée. En effet, seules les actions permettent d'effectuer une modification de l'état. Par exemple, une action pourrait être « Ajouter cet élément dans la liste » ou « Retirer le deuxième élément de la liste ». La différence avec React est que la modification de l'état n'est pas faite directement en indiquant sa valeur (par `this.setState()`), mais plutôt en exécutant une action qui provoquera la modification de l'état (par `store.dispatch(action)`).



store.getState()

La méthode `store.getState()` permet de récupérer sous forme d'objet l'état (stocké dans l'objet `store`). Cette méthode `store.getState()` ressemble au fonctionnement de l'objet `this.state` utilisé dans React, à la différence qu'elle retourne l'état de toute l'application et pas seulement celui qui est associé à un composant.

store.subscribe(listener)

La méthode `store.subscribe(listener)` permet d'effectuer un traitement lors de chaque action. Le traitement est effectué dans la fonction de callback `listener()`.

Globalement, on voit donc que l'on a des actions qui mettent à jour l'état et que l'on peut « écouter » les éventuels changements d'état lorsque les actions sont exécutées. La mise à jour de l'état est effectuée suite aux actions « dispatchées », mais la procédure de mise à jour de l'état est centralisée dans la fonction de callback indiquée en paramètre de la fonction `createStore(reducer)`. Le reducer (méthode de la forme `reducer(state, action)`) est donc une partie importante de l'application car il permet d'indiquer comment l'état est mis à jour (en fonction de l'ancien état et de l'action effectuée).

17.3. Actions dans Redux

Actions dans Redux

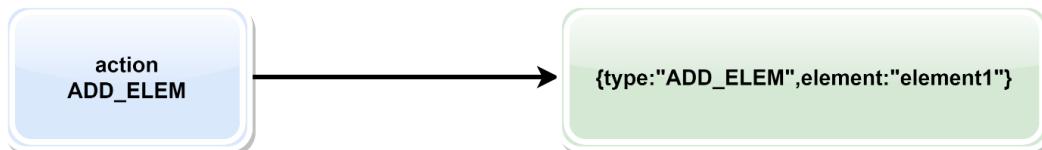
Les actions sont les éléments de l'application qui permettent de modifier l'état. Elles correspondent à tout ce qui peut produire un changement d'état de notre application. Par exemple, en supposant que notre application gère une liste d'éléments, des actions possibles pourraient être :

- ajouter un nouvel élément en fin de liste ;
- supprimer un élément dans la liste ;
- inverser l'ordre des éléments de la liste ;
- afficher uniquement les éléments qui contiennent un mot recherché.

Une action dans Redux est définie sous forme d'un objet JavaScript qui sert à la décrire.

une action doit posséder un attribut `type` qui permettra de la différencier d'une action d'un autre type. Le type est un attribut quelconque, mais il est souvent défini sous forme de constante entière ou de chaîne de caractères

une action peut posséder d'autres attributs qui sont nécessaire pour faire l'action ici on a ajouté l'attribut `element` qui nécessaire pour ajouter un élément à la liste

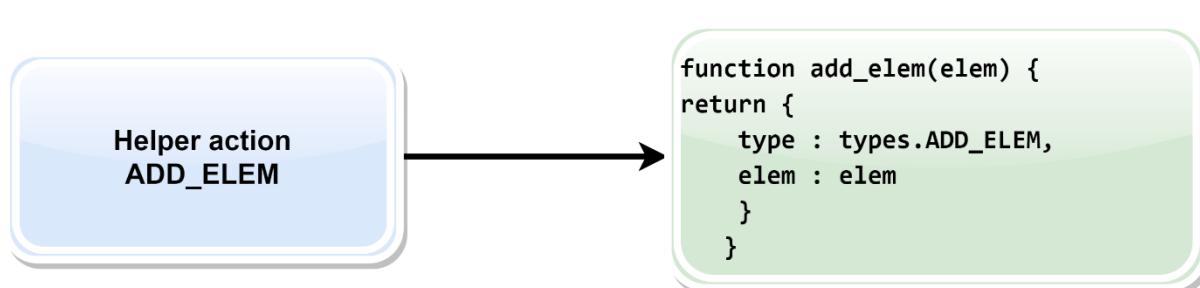




Ces actions peuvent modifier ou non l'état. Par exemple, si une liste possède un seul élément, le fait d'inverser cette liste ne produira aucune modification à l'affichage. Mais si l'on souhaite en plus conserver l'ordre d'insertion (ordre alphabétique croissant ou non), l'état sera modifié en interne (booléen indiquant croissant ou non) même si l'affichage n'est pas modifié.

Une action dans Redux est définie sous forme d'un objet JavaScript qui sert à la décrire. La seule contrainte est que chaque action doit posséder un attribut type qui permettra de la différencier d'une action d'un autre type. Le type est un attribut quelconque, mais il est souvent défini sous forme de constante entière ou de chaîne de caractères

fonction createur d>Action on peut utiliser une fonction helper qui retourne une action



17.4. les reducers dans Redux

Le reducer est la fonction de callback utilisée en paramètre de la fonction createStore(reducer). Elle permet d'indiquer les changements d'états de l'application, en fonction des actions effectuées.

Son fonctionnement est le suivant : la fonction reducer(state, action) utilise l'état actuel de l'application, et selon l'action indiquée en paramètre, la fonction retourne un nouvel état.

Elle ne doit rien faire d'autre, et en particulier elle ne doit surtout pas :

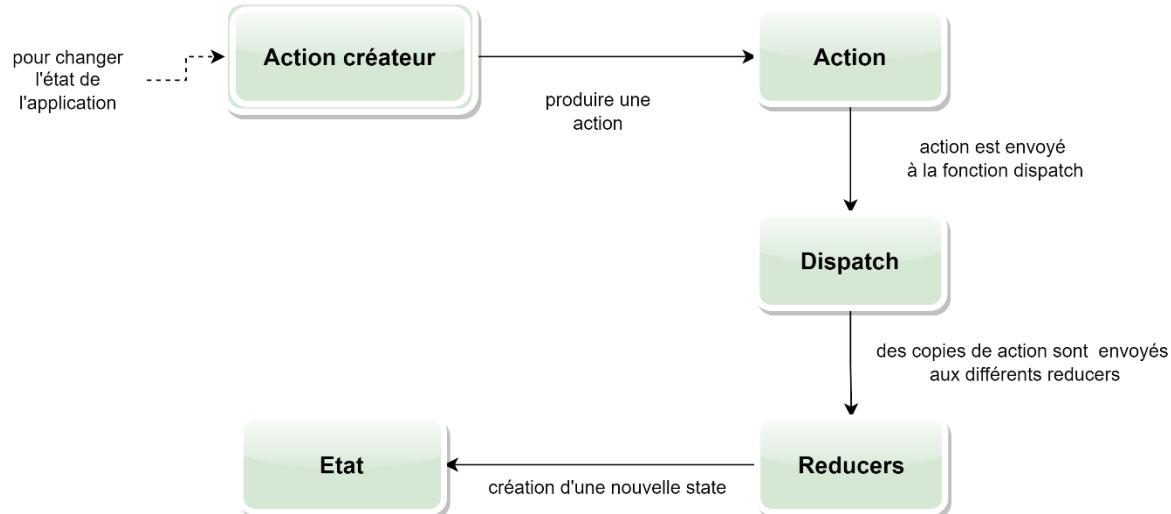
- mettre à jour des valeurs externes (bases de données, variables, etc.) ;
- utiliser des données variables autres que celles indiquées en paramètres (state et action) ;
- modifier l'état actuel (donc on s'interdit d'effectuer une action quelconque dans le reducer, ce qui modifierait l'état).

En fait, un reducer est ce que l'on appelle une « fonction pure » :

Si l'on exécute plusieurs fois la fonction avec les mêmes arguments, son comportement doit toujours être identique. Et pour cela, le monde extérieur ne doit pas influencer sur son comportement.



REDUX cycle



17.5. Atelier d'application Redux

Dans cet atelier on va essayer de gérer les inscriptions et notes des stagiaires

Un stagiaire est caractérisé par un id,nom et filière, les données des stagiaires seront stockées dans array inscriptionHistory

Un autre array notationHistory va contenir les informations de notation, une note est attribué à un stagiaire pour un module donné

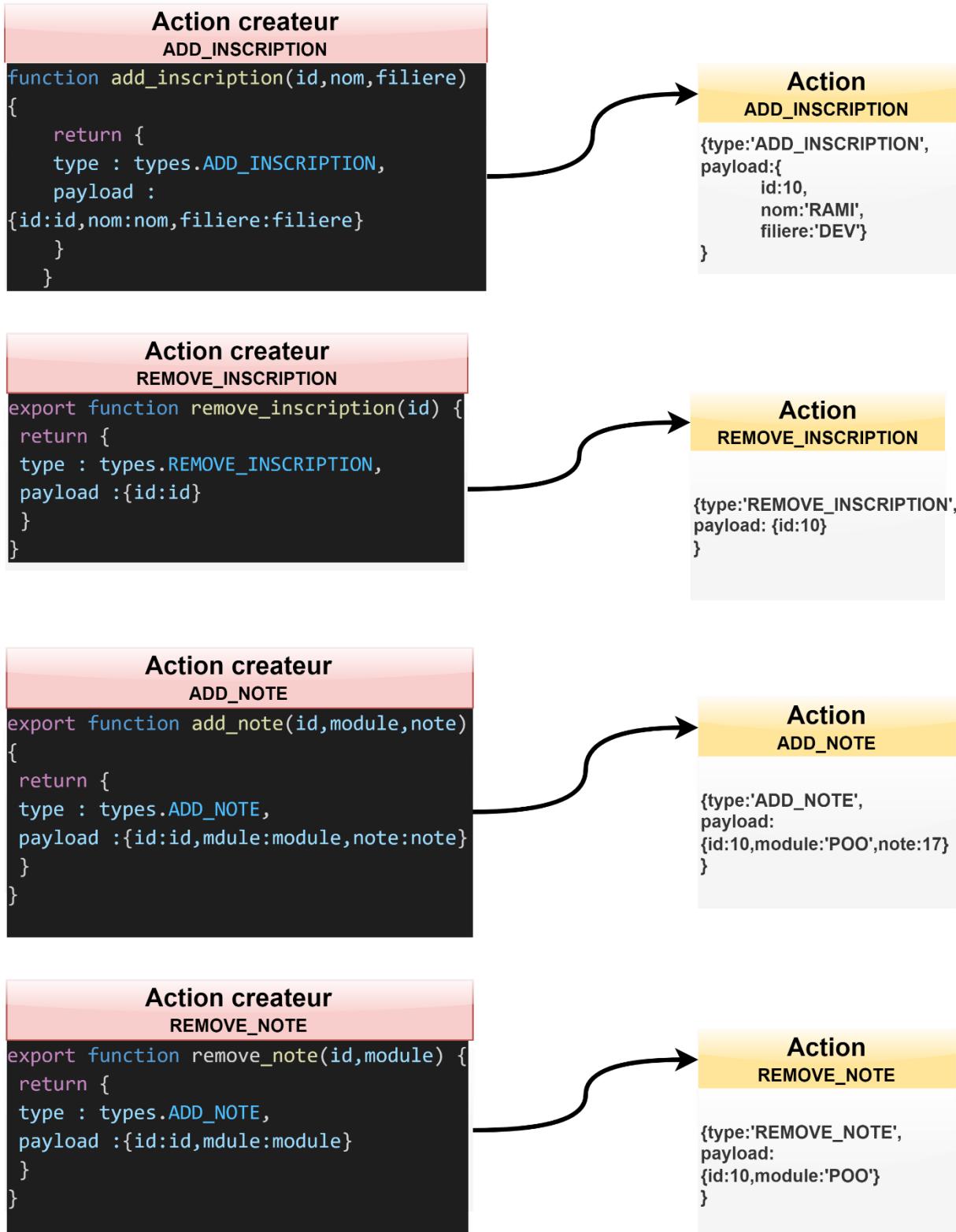
Par convention on regroupe les différents types dans un fichier types.js

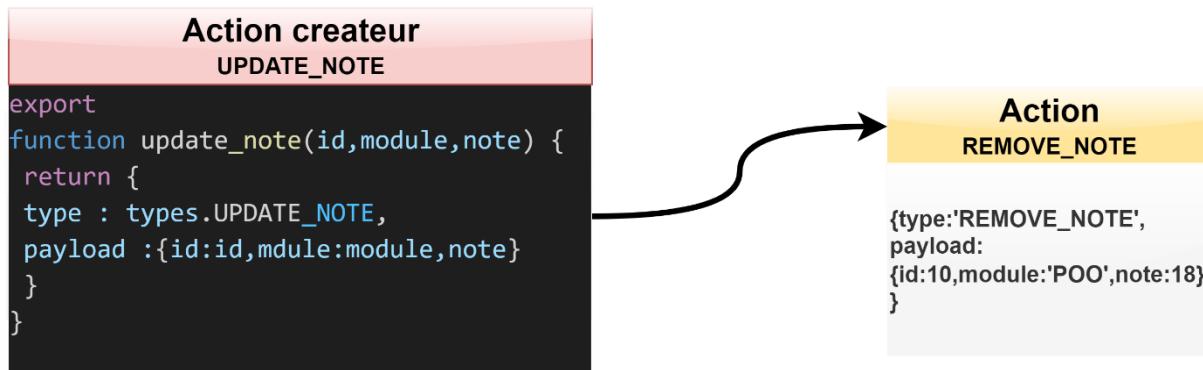
Types.js

```
// Constantes définissant les types d'actions
export const ADD_INSCRIPTION = "ADD_INSCRIPTION";
export const REMOVE_INSCRIPTION = "REMOVE_INSCRIPTION";
export const ADD_NOTE = "ADD_NOTE";
export const REMOVE_NOTE = "REMOVE_NOTE";
export const UPDATE_NOTE = "UPDATE_NOTE";
```



Les actions :





Actions.js

```

import * as types from './types'
// créateurs d'actions
export function add_inscription(id,nom,filiere) {
  return {
    type : types.ADD_INSCRIPTION,
    payload :{id:id,nom:nom,filiere:filiere}
  }
}
export function remove_inscription(id) {
  return {
    type : types.REMOVE_INSCRIPTION,
    payload :{id:id}
  }
}
export function add_note(id,module,note) {
  return {
    type : types.ADD_NOTE,
    payload:{id:id,module:module,note:note}
  }
}
export function remove_note(id,module) {
  return {
    type : types.REMOVE_NOTE,
    payload : {id:id,module:module}
  }
}
export function update_note(id,module,note) {
  return {
    type : types.UPDATE_NOTE,
    payload : {id:id,module:module,note:note}
  }
}

```



17.6. Crédit de reducers

```
//reducer inscription
function inscriptionHistory(listInscription = [], action) {
  if(action.type === types.ADD_INSCRIPTION){
    return [...listInscription,action.payload]
  }
  if(action.type === types.REMOVE_INSCRIPTION){
    return listInscription.filter(insc=> insc.id !== action.payload.id)
  }

  return listInscription;
}

//reducer notation
function notationHistory(listNotation = [],action){
  if(action.type === types.ADD_NOTE){
    return [...listNotation,action.payload]
  }
  if(action.type === types.REMOVE_NOTE){
    return listNotation.filter(note=> !(note.id === action.payload.id &&
note.module === action.payload.module))
  }
  if(action.type === types.UPDATE_NOTE){
    const {id,module,note} = action.payload
    return [...listNotation.filter(note=> !(note.id === id &&
note.module === module)),{id,module,note}]
  }
  return listNotation
}
```

combineReducers(reducers)

Au fur et à mesure que votre application devient plus complexe, vous souhaiterez diviser votre fonction de réduction en fonctions distinctes, chacune gérant des parties indépendantes de l'état.

La fonction d'assistance combineReducers transforme un objet dont les valeurs sont différentes fonctions reducers en une seule fonction reducer que vous pouvez transmettre à createStore.

Le reducer résultant appelle chaque réducteur enfant et rassemble leurs résultats dans un seul objet d'état. L'état produit par combineReducers() est constitué par les états de chaque réducteur sous leurs clés tels qu'ils sont passés à combineReducers().

```
import { createStore, combineReducers } from 'redux'
const Etablissement = combineReducers({
  inscriptionHistory: inscriptionHistory,
  notationHistory: notationHistory
});
const store = createStore(Etablissement);
```

Dans cette situation



```
console.log(store.getState());
```

retourne un objet java script

```
► {inscriptionHistory: Array(0), notationHistory: Array(0)}
```

Contenant les états de chaque reducer inscriptionHistory et notationHistory

Modifiant maintenant l'état de notre application, en ajoutant quelques inscriptions et quelques notations

```
store.dispatch(add_inscription(10, 'RAMI', 'DEV'));//ajouter un stagiaire
store.dispatch(add_inscription(11, 'KAMALI', 'INFRA'));//ajouter un stagiaire
store.dispatch(add_note(10, 'POO', 17));//ajouter une note
store.dispatch(add_note(10, 'Web', 15));//ajouter une note
store.dispatch(add_note(10, 'JS', 13));//ajouter une note
store.dispatch(remove_note(10, 'POO'));//supprimer une note
store.dispatch(update_note(10, 'JS', 18));//supprimer une note
```

```
console.log(store.getState());
```

rendu de console.log

```
▼ {inscriptionHistory: Array(2), notationHistory: Array(2)} ⓘ
  ▼ inscriptionHistory: Array(2)
    ► 0: {id: 10, nom: 'RAMI', filiere: 'DEV'}
    ► 1: {id: 11, nom: 'KAMALI', filiere: 'INFRA'}
      length: 2
    ► [[Prototype]]: Array(0)
  ▼ notationHistory: Array(2)
    ► 0: {id: 10, module: 'Web', note: 15}
    ► 1: {id: 10, module: 'JS', note: 18}
      length: 2
    ► [[Prototype]]: Array(0)
  ► [[Prototype]]: Object
```

17.7. Utilisation du module react-redux

Installation de module react-redux

Le module "react-redux" s'installe au moyen de la commande npm install react-redux.

La méthode connect()

Le module "react-redux" comporte principalement la méthode connect(), qui va faciliter l'accès au store :

- pour lire l'état ;
- pour déclencher les actions qui le mettent à jour.

La lecture de l'état ne se fera plus par store.getState() mais par des propriétés qui seront ajoutées au composant (c'est la méthode connect() qui les ajoute pour nous).



Par exemple, si dans l'état de Redux on a indiqué l'attribut inscriptionHistory, on pourra également accéder à la propriété inscriptionHistory dans un composant React (et ceci bien que cette propriété inscriptionHistory n'ait jamais été créée par nous dans le composant).

Le déclenchement des actions de Redux suit le même principe. Des propriétés (correspondant aux actions) seront ajoutées au composant par la méthode connect(), qui permettront d'utiliser ces actions dans le composant React sans passer par le store (c'est la méthode connect() qui rend ce processus invisible). Par exemple, si dans les actions de Redux on a indiqué l'action add_inscription(id,nom, filiere) qui permet d'insérer une nouvelle inscription dans la liste, on pourra accéder à la propriété add_inscription dans un composant React (et ceci bien que nous n'ayons jamais créé la propriété add_inscription dans le composant).

Utilisation de connect()

La méthode connect(), accessible grâce à l'import du module "react-redux", permet de transformer un composant (donc écrit sous forme de fonction ou de classe) afin de lui apporter de nouvelles propriétés, qui permettront l'accès aux attributs de l'état et aux actions souhaitées. Le store devient ainsi invisible (caché grâce à la méthode connect()) et nous n'avons plus à le transmettre dans les composants.

Le store doit être créé via createStore(). Et pour que sa valeur soit transmise dans les composants par le module "react-redux", il faut créer un composant parent de tous les autres (appelé <Provider>), auquel la valeur du store est transmise dans la propriété store. fichier index.js

Le composant <Provider> avec le store

```
import React from "react"
import ReactDOM from "react-dom"
import { createStore } from "redux"
import { Provider } from "react-redux"

import App from "./App"
import { Etablissement } from "./reducers"

const store=createStore(Etablissement)
ReactDOM.render(
<Provider store={store}>
<App/>
</Provider>,
document.querySelector("#root"))
```

Le composant <Provider> est un composant interne du module "react-redux", c'est pour cela qu'il est importé de ce module. Il englobe le composant principal de l'application, ici le composant <App>.

```
<Provider store={store}>
<App/>
</Provider>
```



Forme de la méthode connect()

Extrait du fichier App.js

```
import { connect } from 'react-redux'  
export default connect(mapStateToProps, mapDispatchToProps)(App)
```

La méthode connect(mapStateToProps, mapDispatchToProps) retourne une nouvelle fonction, cette fonction prend en paramètre un nom de composant (ici appelé App). Le composant indiqué en paramètre est transformé selon les règles de correspondance indiquées dans les fonctions mapStateToProps(state) et mapDispatchToProps(dispatch).

C'est ce composant transformé qui est retourné par la méthode connect()(App). Le composant indiqué en paramètre est soit un nom de fonction, soit un nom de classe (comme on sait le faire pour créer un composant React).

La transformation du composant concerne l'ajout de nouvelles propriétés dans ce composant, ce qui lui permet d'accéder à l'état et aux actions de Redux.

```
function mapStateToProps(state) {  
    return {  
        inscriptionHistory : state.inscriptionHistory  
    }  
}
```

```
function mapDispatchToProps(dispatch) {  
    return {  
        add_Inscription :(id,nom,filiere)=>{  
  
            dispatch(add_inscription(id,nom,filiere))  
    }  
}
```

La fonction mapStateToProps(state)

La fonction mapStateToProps(state) est passée en premier argument de la méthode connect().

Elle est utilisée dans le cas où l'on souhaite permettre au composant d'accéder en lecture à l'état (ou à une partie de celui-ci) de Redux.

Dans le cas où le composant ne veut pas accéder en lecture à l'état de Redux, on met null dans le premier paramètre de la méthode connect().

Dans notre exemple la propriété inscriptionHistory est maintenant définie dans le composant App, elle est donc accessible par l'objet props disponible dans celui-ci.

La fonction mapDispatchToProps(dispatch)

mapDispatchToProps() est une fonction permettant de rendre les actions de Redux accessible dans le composant, sous forme de propriétés (ici des méthodes accessibles à travers l'objet props).

Ces actions permettent de mettre à jour l'état de Redux (alors que la fonction mapStateToProps() vue précédemment permet de rendre l'état de Redux accessible en lecture seulement).



La fonction `mapDispatchToProps()` s'inscrit en deuxième paramètre de la méthode `connect()`. Elle retourne un objet indiquant les nouvelles propriétés disponibles dans le composant, liées aux actions de Redux.

Par exemple, si l'action `add_inscription(id,nom,filiere)`

existe dans les actions de Redux, il est possible pour un composant d'accéder à cette action en écrivant la fonction `mapDispatchToProps(dispatch)` suivante.

Extrait du code fichier index.js

```
import {add_inscription} from './actions'
```

```
function mapDispatchToProps(dispatch) {
  return {
    add_Inscription :(id,nom,filiere)=>{
      dispatch(add_inscription(id,nom,filiere))
    }
}
```

```
export default connect(mapStateToProps,mapDispatchToProps)(App)
```

Désormais la propriété `add_Inscription` est maintenant définie dans le composant `App`, elle est donc accessible par l'objet `props` disponible dans celui-ci.

On peut écrire

```
export default connect(null,mapDispatchToProps)(App)
```

Si on n'a pas besoin des informations State de store

et on peut écrire aussi

```
export default connect(mapStateToProps,null)(App)
```

Si on n'a pas besoin des actions de Redux

A compléter !!!!