

{ React Training }

**DÉVELOPPER UNE APPLICATION WEB
AVEC REACTJS**



SOMMAIRE

Rappels

React

Redux

redux-saga

React-router

Tests / debugging

Optimisations des applications

<https://github.com/soupramanien/react-basic-course>

RAPPELS

ECMAScript 2015 / ES6

JAVASCRIPT

JavaScript est un langage de programmation de plus en plus utilisé pour le développement Web.

Il est formé principalement de :

- ECMAScript, qui fournit les fonctionnalités centrales

mais aussi de :

- DOM (Document Object Model) qui fournit les fonctionnalités pour interagir avec une page web
- BOM (Browser Object Model) qui fournit les fonctionnalités pour interagir avec le navigateur

Versions de ECMAScript :

ES5, ECMAScript 5.1, publiée en 2011

ES6, ECMAScript 6, également appelé Harmony, publiée en 2015

ES7 (2016) et ES8 (2017), releases 'mineures'

ECMAScript 2015

Deuxième révision majeure de JavaScript.

Également connu sous le nom d'ES6 et d'ECMAScript 6

Beaucoup de nouvelles choses dans cette version majeure d'ECMAScript

La compatibilité pour ES6 des moteurs JS est visible sur :

- <https://kangax.github.io/compat-table/es6/>

Il est recommandé de transformer (compiler) le code ES6 en code ES5 à l'aide d'un transpiler tel que :

- Traceur
- Babel

VARIABLES ET CONSTANTES

Les mots-clés à utiliser sont :

- **let** pour une variable
- **const** pour une constante

```
let sum; // 'équivalent' à 'let sum = undefined;'  
let a, b, c = -1; // multiple déclaration  
let v = 100;  
...  
v = 'coucou'; // autorisé mais fortement déconseillé  
...  
const LIMIT = 10;  
const defaultName = 'toto';
```

Avertissement

On évitera dorénavant l'utilisation de `var` pour déclarer une variable.

PORTÉE DE BLOC

Les instructions **let** et **const** définissent des variables et constantes qui sont locales au bloc où elles apparaissent.

```
{  
    const x = 3;  
    console.log(x);  
}  
console.log(x); // error
```

Les paramètres de fonction sont aussi locaux à la fonction.

```
function f(x) {  
    return x + 3;  
}  
f(5); // 8  
console.log(x); // erreur
```

PORTÉE DE BLOC

Une variable définie dans un bloc sera accessible dans tout autre bloc interne (sauf si elle est masquée).

```
let x = 3; // variable de portée globale
let y = 5;
if (true) {
  console.log(x); // 3
  console.log(y); // 5
  const x = 8; // x global est masqué
  console.log(x); // 8
  if (true) {
    console.log(x); // 8
    console.log(y); // 5
  }
  console.log(x); // 8
}
console.log(x); // 3
```


TYPES SYMBOL

Le type symbol est un nouveau type (ES6) permettant de représenter des tokens uniques.

- Les valeurs sont créées avec la fonction Symbol() (pas de new)
- Il est possible de fournir une description en paramètre

```
const RED = Symbol();  
const ORANGE = Symbol("The color of a sunset!");  
RED === ORANGE; // false: every symbol is unique
```

STRING TEMPLATES

En utilisant les backticks ('`'), depuis ES6, à la place des guillemets (simples ou doubles), il est possible d'injecter la valeur d'une variable ou expression numérique dans une string.

```
let monday = 19.5;  
console.log("Temperature on Monday is " + Monday + "\u00b0C");  
let friday = 22;  
console.log(`Temperature on Friday is ${friday}\u00b0C`);
```

On obtient :

Temperature on Monday is 19.5°C

Temperature on Friday is 22°C



FONCTIONS |

FONCTIONS

La syntaxe classique pour définir une fonction est :

Remarque

Une fonction retourne toujours un résultat, même si rien ne l'indique au niveau de la signature (en-tête). Ce résultat est :

- soit une valeur retournée par l'instruction `return`,
- soit `undefined`.

```
function name(arg0, arg1, ..., argN) {  
    statements  
}
```

```
function hello() {  
    return 'Hello world';  
}  
hello(); // 'Hello world'
```

```
function sum(num1, num2) {  
    return num1 + num2;  
}  
sum(5,10); // 15
```

APPEL ET RÉFÉRENCE

Pour toute fonction *f* définie :

- *f()* représente l'appel de la fonction
- *f* représente la référence (adresse) de la fonction

```
function hello() {  
    return 'Hello world';  
}
```

```
const f = hello;  
f(); // 'hello world'  
const o = {};  
o.g = hello;  
o.g(); // 'hello world'  
const t = [1, 2, 3];  
t[1] = hello;  
t[1](); // 'hello world'
```

PARAMÈTRES

Un paramètre ne recevant pas de valeur à l'appel est *undefined*

Pour gérer un nombre variable de paramètres, on peut utiliser l'opérateur '*spread*' ...

```
function f(x) {  
    console.log('x = ' + x);  
}  
f(); // x = undefined  
function addPrefix(prefix, ...words) {  
    return words.map(w => prefix + w);  
}  
addPrefix('con', 'cept', 'te'); // ['concept', 'conte']
```

Remarque

La portée des variables représentant les paramètres est celle de la fonction.

ARGUMENTS (ES5)

Il existe toujours un tableau arguments implicite lors de l'appel a une fonction.

```
function f() {  
    console.log(arguments.length);  
    for (let a of arguments) console.log(a);  
}  
f("Nicolas"); // affiche 1 Nicolas  
f("Nicolas", 25); // affiche 2 Nicolas 25
```

Remarque

dorénavant, on utilisera plutôt l'operateur **'spread'** ... (d'autant plus que arguments ne peut être utilisé avec la notation fléchée).

PARAMÈTRES PAR DÉFAUT

Il suffit de préciser la valeur à la définition.

```
function f(a, b = 'default', c = 3) {  
    return `${a} - ${b} - ${c}`;  
}  
f(5, 6, 7); // '5 - 6 - 7'  
f(5, 6); // '5 - 6 - 3'  
f(5); // '5 - default - 3'  
f(); // 'undefined - default - 3'  
f(5, undefined, 7); // '5 - default - 7'
```

Remarque

A noter qu'on peut utiliser la valeur undefined à l'appel.

ECLATER LES VALEURS

Parfois, il est nécessaire d'éclater les valeurs d'un tableau, notamment lors d'un appel à une fonction.

```
function f(a, b, c) {  
    return a + b + c;  
}  
let t = [10, 3, 7];  
let result = f(...t);  
console.log(result); // 20
```

Remarque

En ES5, il aurait fallu écrire ***let result = f.apply(null,t);***

DÉFINIR UNE FONCTION

Il y a deux manières (sensiblement équivalentes) de définir une fonction.

La deuxième passe par la création d'une fonction anonyme et la récupération de son adresse.

Remarque

Il est possible de référencer une fonction par deux variables différentes.

```
function sum(n1, n2) {  
    return n1+n2;  
}  
sum(5,10); //15
```

```
let sum = function (n1, n2) {  
    return n1+n2;  
};  
sum(5,10); //15
```

```
function sum(n1, n2) {  
    return n1+n2;  
}  
let g = sum;  
g(5,10); //15
```

FONCTIONS « FLÉCHÉES »

Il s'agit d'une notation plus moderne et compacte pour les fonctions anonymes.

Remarque

S'il y a plus d'une instruction dans le corps de la fonction, il faut construire un corps de manière classique (avec accolades et instructions return si approprié).

```
let f1 = function() {  
    return "hello";  
}
```

```
let g1 = () => "hello";
```

```
let f2 = function(name) {  
    return "hello" + name;  
}
```

```
let g2 = name => "hello" + name;
```

```
let f3 = function(a, b) {  
    return a + b;  
}
```

```
let g3 = (a,b) => a + b;
```

FONCTIONS ET THIS

Il est important de noter qu'au sein d'une méthode, `this` est lié :

- à la manière dont une fonction est appelée : dynamic `this`,
- et non à l'endroit où la fonction est définie : lexical `this`

Remarque

Lors du second appel (`speak()` sans préfixe), `this` désigne l'objet global avec `node` (et l'objet `window` avec un navigateur).

```
let o = {
  name: 'Jo',
  speak: function() {
    return "My name is " + this.name;
  }
}
o.speak(); // "My name is Jo"

let speak = o.speak;
speak === o.speak; // true
speak(); // "My name is undefined"
```

FONCTIONS ET THIS

Si on utilise *this* dans une fonction auxiliaire, cela ne convient pas.

```
let o = {  
  name: 'Jo',  
  speakReverse() {  
    function reverse() {  
      return this.name.split('').reverse().join('');  
    }  
    return reverse() + " si eman ym";  
  }  
}  
o.speakReverse(); // TypeError: Cannot read property 'split' of undefined
```

FONCTIONS ET THIS

Une solution classique : sauver *this* dans *that* :

Remarque

Utiliser *that* est une convention. Parfois, on rencontre *self*.

```
let o = {  
  name: 'Jo',  
  speakReverse() {  
    const that = this;  
    function reverse() {  
      return that.name.split('').reverse().join('');  
    }  
    return reverse() + " si eman ym";  
  }  
}  
o.speakReverse(); // "oJ si eman ym"
```

FONCTIONS ET THIS

Autre solution : utiliser une fonction fléchée. Le problème ne se pose pas car *this* est défini lexicalement (*this* provient du code englobant).

```
let o = {  
  name: 'Jo',  
  speakReverse() {  
    let reverse = () => this.name.split('').reverse().join('');  
    return reverse() + " si eman ym";  
  }  
}  
o.speakReverse(); // "oJ si eman ym"
```

SPÉCIFIER THIS

Lorsque vous appelez la méthode **call** sur une fonction **f** :

- le premier argument représente la valeur de **this** pendant l'exécution de **f**
- les autres arguments représentent les paramètres de la fonction **f**

```
let jo = {  
  name: 'Jo'  
};  
let alice = {  
  name: 'alice'  
};  
function greet() {  
  return "Hello " + this.name;  
}  
greet(); // Hello  
greet.call(jo); // Hello Jo  
greet.call(alice); // Hello alice
```


SPÉCIFIER THIS

Remarque

La fonction ***apply*** est similaire à ***call*** mais place tous les arguments, hormis le premier, dans un tableau.

```
let jo = {  
  name: 'Jo'  
};  
let alice = {  
  name: 'alice'  
};
```

```
function update(weight, job) {  
  this.weight = weight;  
  this.job = job;  
}  
update.call(jo, 84, 'singer');  
jo; // { name: 'jo', weight: 84, job: 'singer' }  
update.call(alice, 47, 'actress');  
alice; // { name: 'alice', weight: 47, job: 'actress' }
```

```
let info = [77, 'pilot'];  
update.apply(jo, info);  
jo; // { name: 'jo', weight: 77, job: 'pilot' }
```

SPÉCIFIER THIS

Il est possible d'utiliser l'opérateur *spread* ...

La méthode *bind* permet de lier une valeur à *this* de manière permanente.

Remarque

Il est possible de lier définitivement d'autres paramètres avec *bind*.

```
update.call(jo, ...info); // same update as previously  
let t = [20, 5, 8, 3, 120];  
Math.min.apply(null, t); // 3  
Math.min.call(null, ...t); // 3  
Math.min(...t);
```

```
let updateJo = update.bind(jo);  
updateJo(73, 'doctor');  
jo; // f name: 'jo', weight: 73, job: 'doctor' g  
updateJo.call(alice, 55, 'speaker');  
jo; // f name: 'jo', weight: 55, job: 'speaker' g  
alice; // f name: 'jo', weight: 77, job: 'pilot' g
```

EN RÉSUMÉ : LES FONCTIONS EN JAVASCRIPT

Une fonction peut donc être définie de trois manières en JavaScript :

- comme fonction nommée
- comme fonction anonyme
- comme fonction anonyme, notation fléchée (ES6)

Mais également :

- comme définition de méthode (ES6)

```
function contientZero(tab) {  
    for (let v of tab) if (v == 0) return true;  
    return false;  
}
```

EN RÉSUMÉ : LES FONCTIONS EN JAVASCRIPT

Une fonction anonyme doit se placer à un endroit qui s'y prête. Le plus souvent, elle représente une fonction de rappel (callback).

```
// anonymous function
```

```
function (tab) {  
    for (let v of tab)  
        if (v == 0)  
            return true;  
    return false;  
}
```

```
// anonymous function (ES6)
```

```
(tab) => {  
    for (let v of tab)  
        if (v == 0)  
            return true;  
    return false;  
}
```

```
// anonymous function (ES6) -- methods on arrays
```

```
(tab) => tab.some(v => v == 0);
```

EXEMPLE : TRI D'UN TABLEAU

```
let t = [10, 3, 0, 12];  
t.sort();  
console.log("Après tri : " + t);
```

On peut utiliser une fonction (nommée) pour réaliser le tri.

```
function comparaison(v1,v2) {  
    if (v1 < v2)  
        return -1;  
    if (v1 > v2)  
        return 1;  
    return 0;  
}  
let t = [10, 3, 0, 12];  
t.sort(comparaison);  
console.log("Après tri : " + t);
```

EXEMPLE : TRI D'UN TABLEAU

On peut aussi utiliser une fonction anonyme pour réaliser le tri.

```
let t = [10, 3, 0, 12];
t.sort(function (v1,v2) {
    if (v1 < v2)
        return -1;
    if (v1 > v2)
        return 1;
    return 0;
});
console.log("Après tri : " + t);
```

On peut aussi simplifier le code.

```
let t = [10, 3, 0, 12];
t.sort(function (v1,v2) {
    return v1 - v2;
});
console.log("Après tri : " + t);
```

EXEMPLE : TRI D'UN TABLEAU

Encore mieux, en utilisant la nouvelle syntaxe ES6.

```
let t = [10, 3, 0, 12];  
t.sort((v1,v2) => v1 - v2);  
console.log("Après tri : " + t);
```

CLOSURE

Une fonction définie dans un certain contexte à accès aux variables de ce contexte. Si elle est appelée plus tard dans un tout autre contexte, elle gardera un accès aux variables de son contexte de définition. Il s'agit d'une *closure*.

```
let f;  
{  
  let x = 2;  
  f = function() {  
    return x;  
  }  
}  
f(); // 2
```


IIFE

Il est possible de définir une fonction et de l'appeler immédiatement.

Cela s'appelle une IIFE (Immediately Invoked Function Expression).

L'intérêt principal est qu'une IIFE possède son propre scope qui est protégé car inaccessible de l'extérieur.

```
(function() {  
    // this is the IIFE body  
})();
```

```
let message = (function() {  
    const code = 'coucou';  
    return 'length of the code : ' + code.length;  
})();  
message; // 6
```

IIFE

On peut simuler une variable statique (comme en java) avec une IIFE

```
let incrementer = (function() {  
    let i = 0;  
    return () => i++;  
})();  
incrementer(); // 0  
incrementer(); // 1  
incrementer(); // 2
```

Remarque

La variable `i` n'est pas globale et n'est pas accessible en dehors de la fonction.

ARRAYS, MAPS AND SETS



TABLEAUX

Les tableaux peuvent contenir des données de nature différente.

```
let t = []; // tableau vide
let colors = ['red', 'blue', 'green'];
let t1 = [1, 2, 3];
let t2 = ['one', 2, 'three'];
let t3 = [[1, 2, 3], ['one', 2, 'three']];
let t4 = [
    { name: 'jo', age: 33 },
    1,
    () => 'hello',
    'three'
]
t1[0]; // 1
t2[2]; // three
t3[1]; // ['one', 2, 'three']
t3[1][0]; // one
t4[2](); // hello
t1.length; // 3
t4.length; // 4
t4[1].length; // 3
```

TABLEAUX

Il est possible de connaître et même de modifier la longueur d'un tableau (champ *length*).

De nombreuses méthodes existent sur les tableaux.

```
colors[colors.length]="black"; // nouvelle couleur
colors[99]="pink";
colors.length; // 100
colors[50]; // undefined
colors.length=10; // plus que 10 cases
```

```
let colors = ['red', 'blue', 'green'];
colors; // red, blue, green
colors.join(";"); // red;blue;green
colors.push("black");
colors; // red, blue, green, black
let item = colors.pop();
console.log(item + " " + colors); // black red,blue,green
let item2 = colors.shift();
console.log(item2 + " " + colors); // red blue,green
```

NOMBREUSES MÉTHODES

Ajouter/supprimer un élément :

- en queue de tableau : push et pop
- en tête de tableau : unshift et shift
- Ajouter/supprimer des éléments à n'importe quelle position : splice
- Ajouter des éléments en queue de tableau : concat
- Extraire un sous-tableau : slice
- Remplir un tableau : fill
- Inverser et trier un tableau : reverse et sort
- copy and replace au sein d'un tableau : copyWithin
- trouver le premier ou dernier indice d'une valeur : indexOf et lastIndexOf

```
let t = [0, 1, 2, 3, 4];  
t.reverse(); // 4, 3, 2, 1, 0  
t.indexOf(3); // 1  
t.slice(2, 4); // 2, 1  
t.concat(4, 30); // 4, 3, 3, 1, 0, 4, 30  
t.shift(); // 3, 2, 1, 0  
t.fill(0); // 0, 0, 0, 0
```

FONCTIONS AVEC CALLBACKS SUR LES TABLEAUX

Il existe de nombreuses fonctions sur les tableaux utilisant des callbacks.

Certains de ces fonctions retournent une valeur simple.

```
let b1 = t.some(function (valeur) {  
    return valeur == 22;  
});  
console.log("Variable b1 = ", b1);  
let j = t.findIndex(function (valeur) {  
    return valeur == 22;  
});  
console.log("Variable j = ", j);  
let b2 = t.every(function (valeur) {  
    return valeur > 2;  
});  
console.log("Variable b2 = " + b2);
```

FONCTIONS AVEC CALLBACKS SUR LES TABLEAUX

D'autres fonctions avec callbacks sur les tableaux retournent un (nouveau) tableau.

```
let t2 = t.map(function (valeur) {  
    return valeur * 2;  
});  
console.log("Tableau t2 = " + t2);  
let t3 = t.filter(function (valeur) {  
    return valeur < 8;  
});  
console.log("Tableau t3 = " + t3);  
let t4 = ["cc", "BB", "aa", "DD"];  
t4.sort(function (x, y) {  
    return x.localeCompare(y);  
});  
console.log("Tableau t4 = " + t4);
```


FONCTIONS AVEC CALLBACKS SUR LES TABLEAUX

Les mêmes exemples en utilisant la notation fléchée (ES6).

```
let b1 = t.some(v => v == 22);
console.log("Variable b1 = ", b1);
let j = t.findIndex(v => v == 22);
console.log("Variable j = ", j);
let b2 = t.every(v => v > 2);
console.log("Variable b2 = " + b2);
let t2 = t.map(v => v * 2);
console.log("Tableau t2 = " + t2);
let t3 = t.filter(v => v < 8);
console.log("Tableau t3 = " + t3);
let t4 = ["cc", "BB", "aa", "DD"];
t4.sort((x, y) => x.localeCompare(y));
console.log("Tableau t4 = " + t4);
```

FONCTIONS AVEC CALLBACKS SUR LES TABLEAUX

Il est possible de parcourir un tableau avec la méthode `forEach`.

```
let t = [3, 4, 2, 8, 5];  
t.forEach(v => console.log(' carre : ' + v*v));
```

Remarque

Pour nombre de méthodes, il est possible d'indiquer un second paramètre lors des appels : il s'agit de la valeur à lier à *this*.

Remarque

Pour nombre de méthodes, il est possible d'utiliser une fonction callback à deux paramètres (et même à trois), dont le deuxième est l'indice courant.

```
let t = [3, 4, 12, 8, 22, 49, 5];  
let carre = (v) => Number.isInteger(Math.sqrt(v));  
t.find((v,i) => i>2 && carre(v)); // 49
```

MAP REDUCE

Un exemple avec map où les parenthèses sont nécessaires autour du corps de la fonction fléchée à cause des accolades de création d'objets.

```
let jeux = ['go', 'de'];  
let prix = [99, 15];  
let t = jeux.map((j,i) => ({nom: j, prix: prix[i]}));  
// [{ nom: 'go', prix: 99 g, f nom: 'de', prix: 15 }]
```

La fonction reduce permet de combiner toutes les valeurs d'un tableau.

Le premier argument est l'accumulateur, et les trois arguments suivants sont la valeur courante, l'indice courant et le tableau lui-même .

```
let t = [0, 1, 2, 3];  
let somme = t.reduce((a, b) => a + b);  
console.log("Somme=" + somme);
```

MAP REDUCE

Remarque

Si le premier élément du tableau ne peut pas servir d'initialisation de l'accumulateur, alors il faut le glisser en deuxième argument de reduce.

Exercice : écrire le code utilisant la fonction reduce permettant de calculer la valeur max présente dans un tableau t.

```
// (version a)
let max = t.reduce((a, b) => {
    if (b > a) return b;
    else return a;
});
console.log("max (a) = " + max);
// (version b)
max = t.reduce((a, b) => b > a ? b : a);
console.log("max (b) = " + max);
// (version c)
max = t.reduce((a, b) => Math.max(a, b));
console.log("max (c) " + max);
```

MAP REDUCE

Exercice : compter le nombre de lettres cumulé sur le nombre de mots d'un tableau donne.

```
let pets = ['cat', 'dog', 'fish'];  
let nb = pets.map(s => s.length).reduce((a, b) => a + b);  
console.log('nb = ' + nb);
```

Exercice : écrire le code permettant de construire un nouveau tableau obtenu en itérant toutes les valeurs paires d'un tableau, et en transformant chaque valeur restante en son carré.

```
let tab = t.filter(v => v % 2 == 0).map(v => v * v);  
console.log('Tableau tab : ' + tab);
```

MAPS

Pour construire des maps, depuis ES6 il ne faut plus utiliser les objets mais utiliser l'objet Map.

Remarque

Il est possible de passer un tableau de tableaux (avec deux cellules, l'une pour la clé, l'autre pour la valeur) à la construction de la map.

```
let map = new Map();
map.set('toto', 100);
map.set('titi', 50).set('tata', 20);
map.get('titi'); // 50
map.set('titi', 10);
map.get('titi'); // 10
map.has('tutu'); // false
map.has('toto'); // true
map.size; // 3
map.delete('toto');
map.size; // 2
map.clear();
map.size; // 0
```

MAPS

Il est possible d'itérer sur les clés, les valeurs ou les entrées. L'ordre d'insertion est préservé.

```
for (let k of map.keys())  
  console.log(k);  
for (let v of map.values())  
  console.log(v);  
for (let e of map.entries())  
  console.log(e[0] + ':' + e[1]);  
for (let [k, v] of map.entries())  
  console.log(k + ':' + v);  
for (let [k, v] of map)  
  console.log(k + ':' + v);  
map.forEach(k => console.log(k));  
map.forEach((k,v) => console.log(k + ':' + v));
```

SETS

Pour construire des sets, depuis ES6 on peut utiliser l'objet Set. Lors d'une itération, l'ordre d'insertion est préservé.

```
let set = new Set();  
set.add('user');  
set.add('admin').add('tech');  
set.size; // 3  
set.add('admin');  
set.size; // 3  
set.delete('user');  
set; // ['admin', 'tech']  
set.forEach(k => console.log(k));
```


DÉSTRUCTURATION D'AFFECTATION

Pour les tableaux, on peut copier les valeurs dans des variables

individuelles nommées en une seule instruction.

```
let t = [1, 2, 3];  
let [x, y] = t;  
x; // 1  
y; // 2  
z; // error  
let t = [1, 2, 3, 4, 5];  
let [x, y, ...rest] = t;  
x; // 1  
y; // 2  
rest; // [3, 4, 5]  
let x = 5, y = 10;  
[x, y] = [y, x];  
x; // 10  
y; // 5
```

DÉSTRUCTURATION D'AFFECTATION

Il est possible d'ignorer des valeurs, et de fournir des valeurs par défaut.

```
let [x, , y] = [1, 2, 3];  
x; // 1  
y; // 3  
let [x, , , ...y] = [1, 2, 3, 4, 5, 6];  
x; // 1  
y; // [4, 5, 6]  
let [x, y, z = 3] = [1, 2];  
z; // 3  
function f([a, b, c = 4] = [1, 2, 3]) {  
  console.log(a, b, c);  
}  
f(); // 1 2 3  
f(undefined); // 1 2 3  
f([10, 20]); // 10 20 4
```

DÉSTRUCTURATION D'AFFECTATION

Il y a de nombreux autres usages de l'opérateur ... sur les tableaux.

```
let t1 = [2, 3, 4];
let t2 = [1, ...t1, 5, 6];
console.log(t2); // [1, 2, 3, 4, 5, 6]
let t3 = [1];
t3.push(...t1);
console.log(t3); // [1, 2, 3, 4];
let a1 = [1], a2 = [2];
let a3 = [...a1, ...a2, ...[3, 4]], a4 = [5];
function f(a, b, c, d, e) {
    return a + b + c + d + e;
}
console.log(f(...a3, ...a4)); // 15
```

DÉSTRUCTURATION D'OBJETS

Pour les objets, il faut respecter le nom des champs (peu importe l'ordre).

```
let o = { b: 2, c: 3, d: 4 };  
let {a, b, c} = o;  
a; // undefined  
b; // 2  
c; // 3  
d; // erreur
```

Lorsque la déstructuration n'est pas effectuée au moment de la déclaration, il faut encadrer avec des parenthèses.

```
let o = { b: 2, c: 3, d: 4 };  
let d, b, c;  
{d, b, c} = o; // erreur  
({d, b, c} = o); // ok  
console.log(d, b, c); // 4 2 3
```

DÉSTRUCTURATION D'OBJETS

Il est possible d'utiliser des valeurs par défaut et d'autres noms que ceux des propriétés.

```
let {a, b, c = 3} = {a: 1, b: 2};  
console.log(a, b, c); // 1 2 3  
let o = {name: 'toto', age: 20 };  
let {name: x, age: y} = o;  
x; // 'toto'  
y; // 20  
let o = {name: 'toto', other: {age: 20}};  
let {name, other:{age}} = o;  
name; // 'toto'  
age; // 20
```

DÉSTRUCTURATION D'ARGUMENTS

Pour les tableaux, comme d'habitude, il faut prêter attention à l'ordre des arguments.

```
function sentence([subject, verb, object]) {  
    return subject + ' ' + verb + ' ' + object;  
}  
let t = ['I', 'love', 'JavaScript'];  
sentence(t); // 'I love JavaScript'
```

En fait, on aurait pu écrire :

```
function sentence(subject, verb, object) {  
    return subject + ' ' + verb + ' ' + object;  
}  
let t = ['I', 'love', 'JavaScript'];  
sentence(...t); // 'I love JavaScript'
```

DÉSTRUCTURATION D'ARGUMENTS

Pour les objets, le nom des champs permet le matching.

```
function sentence({subject, verb, object}) {  
    return subject + ' ' + verb + ' ' + object;  
}  
  
let o = {  
    verb: 'love',  
    object: 'JavaScript',  
    subject: 'I'  
};  
  
sentence(o);  
  
function f({name = 'toto', age = 20, profession = 'pilot'}  
= {}) {  
    console.log(name, age, profession);  
}  
  
f({name: 'titi', age: 30}); // 'titi 30 pilot'  
f(); // 'toto 20 pilot'  
f(undefined); // 'toto 20 pilot'
```

OBJECTS AND CLASSES



CREATION DIRECTE D'UN OBJET

Notation littérale à privilégier :

- symboles '{' et '}' encadrant la définition de l'objet
- introduction de champs (propriétés) et fonctions (méthodes), constitués d'un nom suivi de ':' et de la valeur
- symbole ',' comme séparateur entre les différents champs/fonctions

Remarque

Pour les méthodes, on privilégiera la notation concise ('method définition' comme ci-dessus).

```
let person = {  
  name: 'Alice',  
  age: 20,  
  job: 'singer',  
  introduction() {  
    return this.name + ' ' + this.age + ' ' + this.job;  
  }  
};
```

ACCÈS AUX MEMBRES D'UN OBJET

Pour accéder à tout membre (champ ou fonction) d'une variable de type object, on utilise généralement la notation pointée :

- nom de la variable suivi de '.' suivi du nom du membre

Toutefois, il est également possible d'utiliser les crochets pour accéder aux membres.

```
person.name; // Alice
person['name']; // Alice
let field='name';
person[field]; // Alice
person.introduction(); // Alice 20 singer
person['introduction'](); // Alice 20 singer
```

ACCÈS AUX MEMBRES D'UN OBJET

Il est important de noter qu'on utilise :

- `this` si on se trouve dans le code d'une fonction de l'objet
- le nom de la variable (objet) si on se trouve dans du code par ailleurs

```
let game = {  
  finished: false,  
  ...  
  isFinished() {  
    return this.finished;  
  },  
  getPieceAt(row, col) {  
    ...  
  }  
};  
if (game.isFinished()) {  
  ...  
}
```

PARCOURIR UN OBJET

On peut utiliser `for in`, mais il faut généralement prêter attention aux propriétés héritées.

```
let o = { apple: 1, orange: 2, apricot: 3, banana: 4 };
for (let p in o)
  if (o.hasOwnProperty(p))
    console.log(p + ':' + o[p]);
```

On utilise plutôt `Object.keys()` :

```
Object.keys(o)
  .filter(p => p.match(/^a/))
  .forEach(p => console.log(p + ':' + o[p]));
// apple : 1
// apricot : 3
```

OBJETS DYNAMIQUES

Il est possible de :

- modifier dynamiquement la structure d'un objet en ajoutant ou retirant (delete) des propriétés (champs ou méthodes).
- modifier la valeur d'un champ, et aussi d'une méthode (dangereux)

```
let obj = {}; // objet vide
obj.size=3;
obj.color='yellow';
obj; // f size: 3, color: 'yellow' g,
obj.hello = function() { return 'hello'; }
obj.hello(); // hello
obj.hello = function() { return 'goodbye'; }
obj.hello(); // goodbye
delete obj.hello;
obj.hello(); // erreur
obj.address = { street: 'rue de la paix', city: 'Paris' };
obj.address.city; // Paris
obj['address']['city']; // Paris
```

CRÉER UNE CLASSE

Dans une classe, on peut placer un constructeur (un seul !) et des méthodes (on n'utilise pas le mot-clé function).

```
class Point {  
    constructor(x, y) {  
        this.x = x;  
        this.y = y;  
    }  
    atNorthEast() {  
        return this.x >= 0 && this.y >= 0;  
    }  
}  
  
let a = new Point(3,-1);  
let b = new Point(2,8);  
a.atNorthEast(); // false  
b.atNorthEast(); // true  
a instanceof Car; // true  
b instanceof Array; //false
```

MÉTHODES STATIQUES

Ce sont des méthodes qui ne concernent pas des instance spécifiques mais la classe.

```
class Point {  
    constructor(x, y) {  
        this.x = x;  
        this.y = y;  
    }  
    static distance(a, b) {  
        const dx = a.x - b.x;  
        const dy = a.y - b.y;  
        return Math.hypot(dx, dy);  
    }  
}  
  
let p1 = new Point(5, 5);  
let p2 = new Point(10, 10);  
Point.distance(p1, p2);
```

VARIABLES DE CLASSE

Il faut définir une propriété au niveau de la classe.

```
class Car {  
    constructor(make) {  
        this.make=make;  
        this.id=Car.id++;  
    }  
}  
Car.id = 0;  
let c1 = new Car('Peugeot');  
let c2 = new Car('Renault');  
c1.id; // 0  
c2.id; // 1
```


HÉRITAGE

Il faut utiliser le mot-clé `extends` et appeler le super-constructeur si on insère un constructeur dans la sous-classe.

```
let v = new vehicle();
v.addPassenger('Jo');
v.addPassenger('Alice');
v.passengers; // ['Jo', 'Alice']
let c = new Car();
c.addPassenger('John');
c.addPassenger('Lucie');
c.passengers; // ['John', 'Lucie']
v.deployAirbags(); // erreur
c.deployAirbags(); // 'bwoosh'
```

```
class Vehicle {
  constructor() {
    this.passengers = [];
  }
  addPassenger(p) {
    this.passengers.push(p);
  }
}
class Car extends vehicle {
  constructor() {
    super();
  }
  deployAirbags() {
    console.log('bwoosh');
  }
}
```

COMPORTEMENT ASYNCHRONE

PROMISE

L'objet Promise (pour « promesse ») est utilisé pour réaliser des traitements de façon asynchrone.

Une promesse représente une valeur qui peut être disponible maintenant, dans le futur voire jamais.

```
const promise1 = new Promise((resolve, reject) => {  
    setTimeout(() => {    resolve('foo'); }, 300);  
});  
  
promise1.then((value) => { console.log(value); // expected output: "foo"});  
console.log(promise1); // expected output: [object Promise]
```

CALLBACK HELL

L'enchaînement d'appels de fonctions callback devient très vite illisible :

```
faireTruc1(function(result1) {  
    faireTruc2(result1, function(result2) {  
        faireTruc3(result2, function(result3) {  
            console.log('Résultat final :' + result3);  
        }, failureCallback);  
    }, failureCallback);  
}, failureCallback);
```

Avec les promesses, le code est beaucoup plus lisible

```
faireTruc1()  
    .then(result1 => faireTruc2(result1))  
    .then(result2 => faireTruc3(result2))  
    .then(result3 => console.log('Res. final :' + result3))  
    .catch(failureCallback);
```

CRÉER UNE PROMESSE

```
const maPremierePromesse = new Promise((resolve, reject) => {  
  // réaliser une tâche asynchrone et appeler :  
  
  // resolve(uneValeur); // si la promesse est tenue  
  // ou  
  // reject("raison d'echec"); // si elle est rompue  
});
```

ILLUSTRATION AVEC XHR

```
function ajax(url) {  
    return new Promise(function(resolve, reject) {  
        let r = new XMLHttpRequest();  
        r.open("GET",url);  
  
        r.addEventListener("load", function() {  
            if (r.status === 200)  
                resolve(r.responseText);  
            else  
                reject("Server Error : " + r.status);  
        },false);  
  
        r.addEventListener("error", function() {  
            reject("Cannot make AJAX Request");  
        },false);  
  
        r.send();  
    });  
}
```

ILLUSTRATION AVEC XHR

```
let url = "https://mdn.github.io/learning-area/javascript/oojs/json/superheroes.json";

ajax(url).then(function(value) {
    return JSON.parse(value);
}).then(function(value) {
    console.log(value.squadName);
    return value;
}).catch(function(reason) {
    console.log(reason);
});
```

ASYNC FUNCTION

La déclaration ***async function*** définit une fonction asynchrone qui renvoie un objet *AsyncFunction*.

Une fonction asynchrone est une fonction qui s'exécute de façon asynchrone grâce à la boucle d'évènement en utilisant une promesse (Promise) comme valeur de retour.

```
function resolveAfter2Seconds() {  
    return new Promise(resolve => {  
        setTimeout(() => {  
            resolve('resolved');  
        }, 2000);  
    });  
}  
  
async function asyncCall() {  
    console.log('calling');  
    const result = await resolveAfter2Seconds();  
    console.log(result); // expected output: "resolved"  
}  
asyncCall();
```


ASYNC FUNCTION

Une fonction **async** peut contenir une expression **await** qui interrompt l'exécution de la fonction asynchrone et attend la résolution de la promesse passée Promise.

La fonction asynchrone reprend ensuite puis renvoie la valeur de résolution.

Le mot-clé **await** est uniquement valide au sein des fonctions asynchrones.

Si ce mot-clé est utilisé en dehors du corps d'une fonction asynchrone, cela provoquera une exception `SyntaxError`.



MODULES

MODULE

Le mécanisme pour diviser les programmes JavaScript en plusieurs modules qu'on pourrait importer les uns dans les autres

Cette fonctionnalité était présente dans Node.js depuis longtemps et plusieurs bibliothèques et *frameworks* JavaScript ont permis l'utilisation de modules ([CommonJS](#), [AMD](#), [RequireJS](#) ou, plus récemment, [Webpack](#) et [Babel](#)).

L'utilisation des modules natifs JavaScript repose sur les instructions **import** and **export**

EXPORT DES FONCTIONNALITÉS

Afin d'utiliser les fonctionnalités d'un module, on devra les exporter à l'aide de l'instruction `export`

La méthode la plus simple consiste à placer cette instruction devant chaque valeur qu'on souhaite exporter

Il est possible d'exporter des fonctions, des variables (qu'elles soient définies avec `var`, `let` ou `const`) et aussi des classes

```
//square.js
export const name = 'square';

export function draw(ctx, length, x, y, color) {
  ctx.fillStyle = color;
  ctx.fillRect(x, y, length, length);

  return {
    length: length,
    x: x,
    y: y,
    color: color
  };
}
```

EXPORT DES FONCTIONNALITÉS

Une méthode plus concise consiste à exporter l'ensemble des valeurs grâce à une seule instruction située à la fin du fichier

- les valeurs sont séparées par des virgules et la liste est délimitée entre accolades :

```
//square.js
const name = 'square';

function draw(ctx, length, x, y, color) {
  ctx.fillStyle = color;
  ctx.fillRect(x, y, length, length);

  return {
    length: length,
    x: x,
    y: y,
    color: color
  };
}
export { name, draw };
```

IMPORTER DES FONCTIONNALITÉS

Lorsque des fonctionnalités sont exportées par un premier module, on peut les importer dans un script afin de les utiliser à l'aide de l'instruction ***import***

```
import { name, draw } from './square.js';
```

EXPORTS PAR DÉFAUT

Conçu pour simplifier l'export d'une fonction par module

On exporte une fonction par défaut

```
export default function(ctx) {
```

```
  ...
```

```
}
```

on importe la fonction par défaut avec cette ligne:

```
import randomSquare from './square.js';
```

Pas d'accolade car il n'y a qu'un seul export par défaut possible par module

Equivalent à import {default as randomSquare} from './square.js';



RAPPELS

Programmation fonctionnelle

FUNCTIONAL PROGRAMMING

C'est la forme de programmation caractérisée par l'enchaînement des appels de fonctions (method chaining).

```
let t = [2, 3, 4].map(v => v * 2).reduce((a,v) => a+v, 0);  
console.log("The total is", t); // 18
```

Nombre de langages intègrent aujourd'hui des mécanismes de programmation fonctionnelle. En Java 8, c'est par le biais de Stream.

```
List<String> list =  
Arrays.asList("a1", "a2", "b1", "c2", "c1");  
list  
    .stream()  
    .filter(s -> s.startsWith("c"))  
    .map(s -> s.toUpperCase())  
    .sorted()  
    .forEach(s -> System.out.print(s)); // c1c2
```

HIGHER ORDER FUNCTIONS

Les fonctions qui opèrent sur d'autres fonctions

- soit en les prenant comme arguments,
- soit en les retournant

Les fonctions d'ordre supérieur(HOF) nous permettent d'abstraire sur des *actions*

Elles se présentent sous plusieurs formes

- des fonctions qui créent de nouvelles fonctions
- des fonctions qui prennent en argument d'autres fonctions
- des fonctions qui changent d'autres fonctions

HIGHER ORDER FUNCTIONS

Des fonctions qui créent de nouvelles fonctions

Exemple:

```
function greaterThan(n) {  
  return m => m > n;  
}  
let greaterThan10 = greaterThan(10);  
console.log(greaterThan10(11));  
// → true
```

HIGHER ORDER FUNCTIONS

Des fonctions qui prennent en argument d'autres fonctions

Exemple:

```
function sum(array, f) {  
  if (typeof f !== 'function')  
    f = x => x;  
  return array.reduce((a,v) => a + f(v), 0);  
}  
let t = [1, 2, 3];  
sum(t); // 6  
sum(t, x => x*x); // 14  
sum(t, x => Math.pow(x,3)); // 36
```

HIGHER ORDER FUNCTIONS

Des fonctions qui changent d'autres fonctions

Cela permet de spécialiser une fonction en spécifiant un ou plusieurs paramètres.
Cela est connu sous le terme de currying.

Exemple:

```
function newSummer(f) {  
    return array => sum(array, f);  
}  
const sumOfSquares = newSummer(x => x*x);  
const sumOfCubes = newSummer(x => Math.pow(x,3));  
let t = [1, 2, 3];  
sum(t); // 6  
sumOfSquares(t); // 14  
sumOfCubes(t); // 36
```



REACT

REACT ?

Bibliothèque JavaScript

Créé par Facebook en 2013

Gère la vue (V de MVC)

Quelques principes

- Déclaratif
- DOM Virtuel
- Centré composant
- Réactif

DÉCLARATIF

Imperative

```
const element = React.createElement(  
  'h1',  
  {className: 'greeting'},  
  'Bonjour, monde !'  
);
```

Declarative

```
const element = (  
  <h1 className="greeting">  
    Bonjour, monde !  
  </h1>  
);
```


UN DOM VIRTUEL

Représentation en mémoire du DOM physique

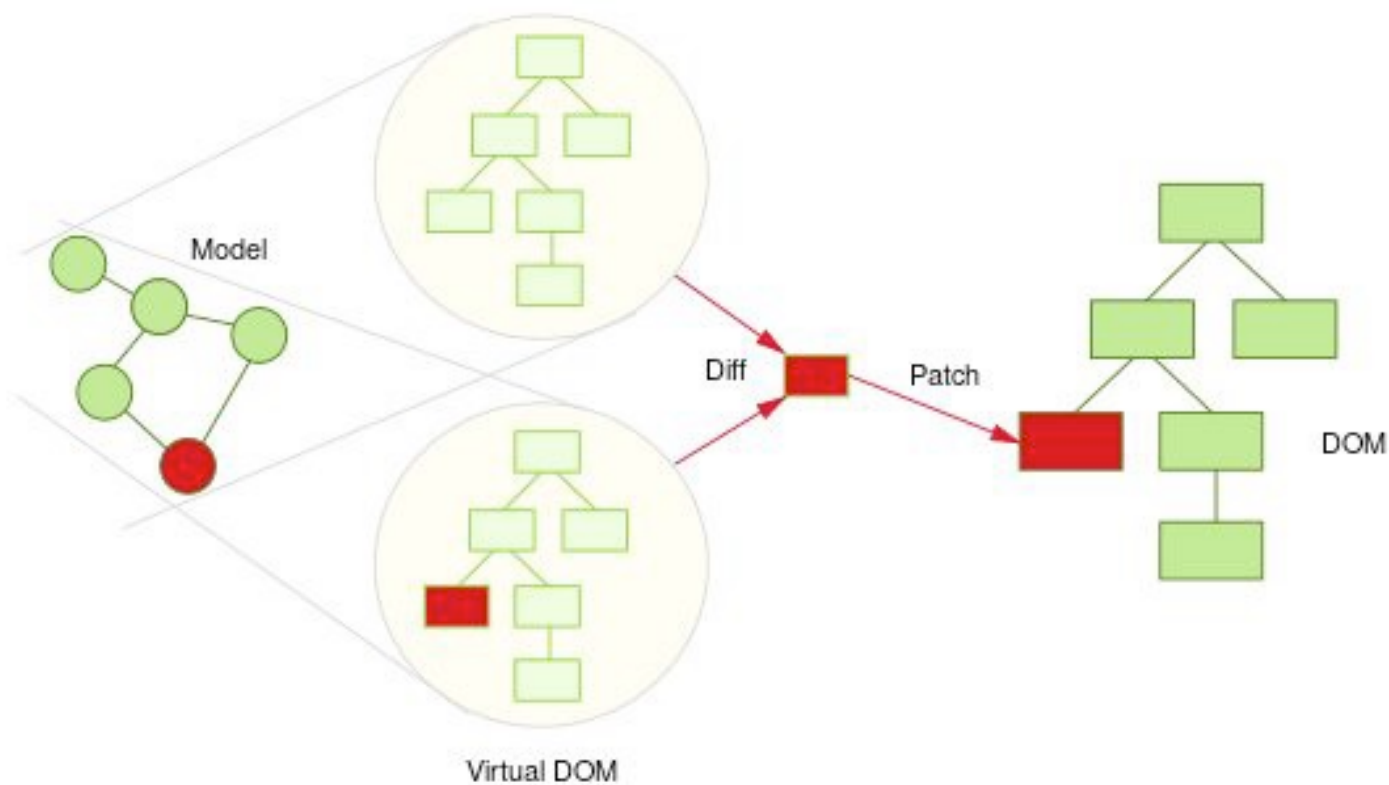
- les modifications se font sur ce DOM virtuel ensuite **React** s'occupe de les synchroniser vers le DOM physique en cas de nécessité.

Très rapide

Système d'évènements interne

Calcul les différences avec l'état précédent

UN DOM VIRTUEL



DES COMPOSANTS

☐ Only show products in stock

Name	Price
------	-------

Sporting Goods	
-----------------------	--

Football	\$49.99
----------	---------

Baseball	\$9.99
----------	--------

Basketball	\$29.99
------------	---------

Electronics	
--------------------	--

iPod Touch	\$99.99
------------	---------

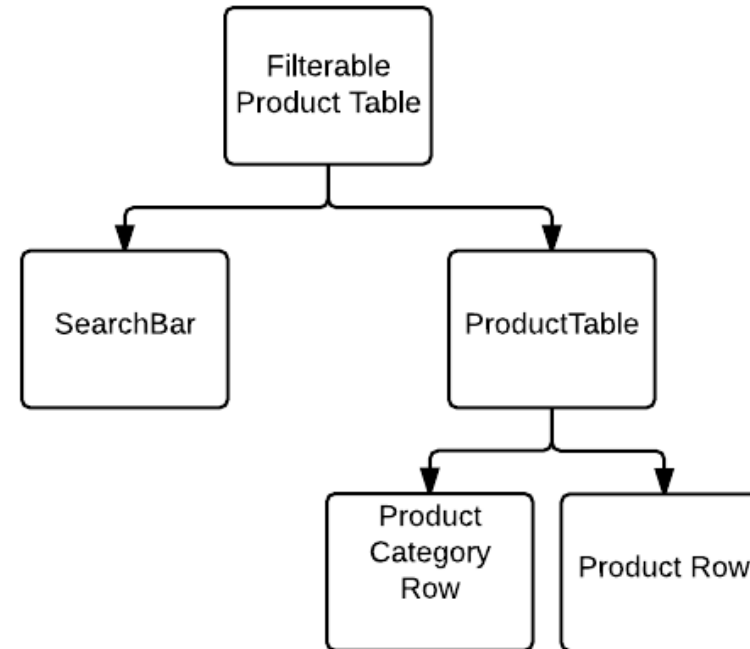
iPhone 5	\$399.99
----------	----------

Nexus 7	\$199.99
---------	----------

DES COMPOSANTS

☐ Only show products in stock

Name	Price
Sporting Goods	
Football	\$49.99
Baseball	\$9.99
Basketball	\$29.99
Electronics	
iPod Touch	\$99.99
iPhone 5	\$399.99
Nexus 7	\$199.99



RÉACTIF

Responsive

- Réponse en temps voulu, si possible
- Temps de réponses rapides et fiables (limites hautes)

Résilient

- Résiste à l'échec
- On fait en sorte qu'un échec n'impacte qu'un seul composant

Élastique

- Le système reste réactif en cas de variation de la charge de travail
 - Pas de point central
 - Pas de goulot
 - Distribution des entrées entre composants

Orienté message

- Passage de messages asynchrones
 - Couplage faible, isolation
- Pas de blocage, les composants consomment les ressources quand ils peuvent

QUELS LANGAGES UTILISE REACT?

HTML pour les vues

CSS pour les styles

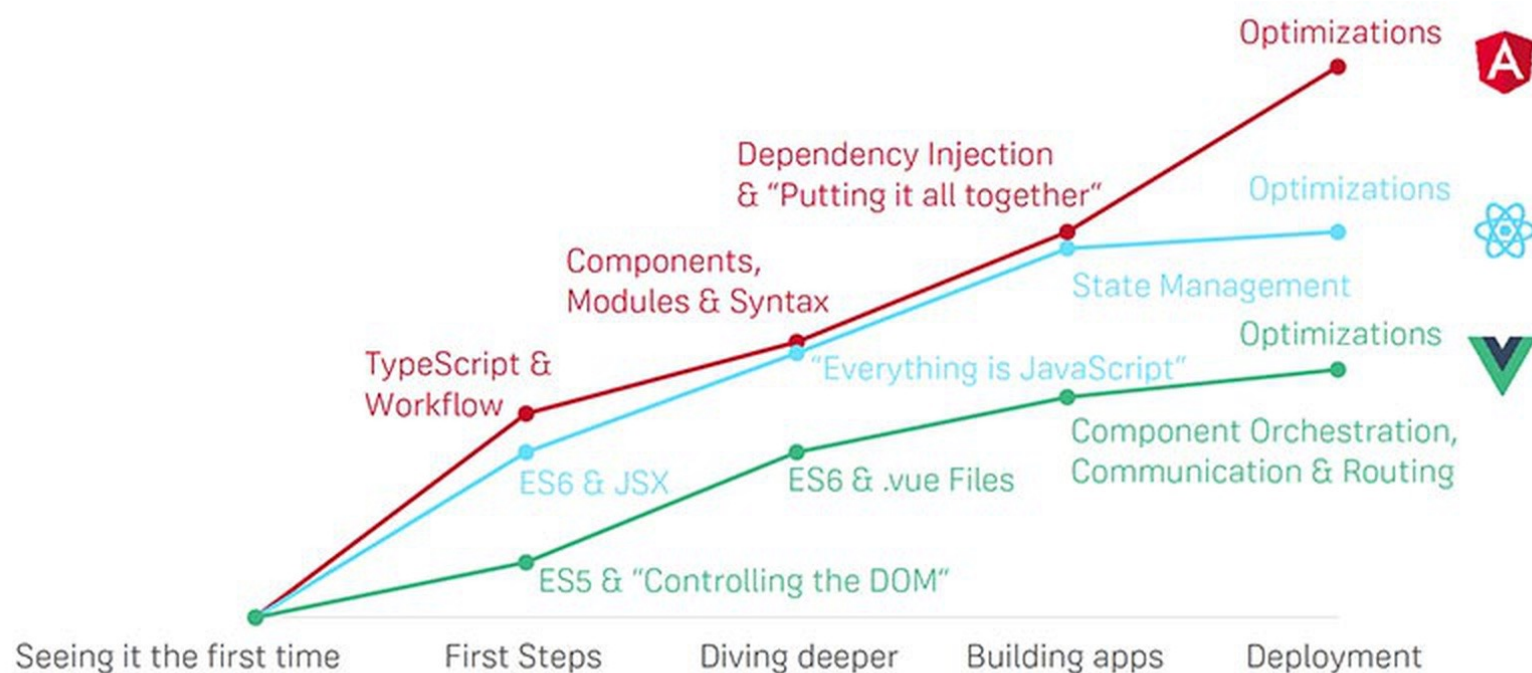
JSX (JavaScript XML) : pour les scripts



POURQUOI REACT |

COURBE D'APPRENTISSAGE

(Possible) Learning Curve



AVANTAGES DE REACTJS

Facile à apprendre, en raison de sa simplicité en ce qui concerne la syntaxe

Haut niveau de flexibilité et de réactivité

DOM virtuel (modèle d'objet de document)

Combiné avec ES6 / 7, ReactJS peut travailler avec la charge élevée d'une manière facile.

Librairie JavaScript 100% open source (reçoit beaucoup de mises à jour et d'améliorations quotidiennes)

La migration entre les versions est généralement très facile, avec « codemods », fournit par Google, pour automatiser une grande partie du processus.

INCONVÉNIENTS DE REACTJS

ReactJS est *unopinionated*, ce qui signifie que les développeurs ont parfois trop de choix

Long à maîtriser, ReactJS nécessite une connaissance approfondie de la façon d'intégrer l'interface utilisateur dans le framework MVC

LES AVANTAGES ET LES INCONVÉNIENTS DES 3 PRINCIPAUX LIBRAIRAIRES/Frameworks

	Angular	React	Vue.js
Performance	moyen	haute	haute
Scalabilité	haute	haute	faible
Apprentissage	difficile	moyen	facile
Disponibilité des développeurs	haute	haute	faible
Communauté des développeurs	grande	très grande	petite
Acceptation et confiance	haute	très haute	faible

LES MARQUES QUI ONT ADOPTÉES CHACUN DES FRAMEWORKS

Angular	React	VueJS
Microsoft	Facebook	Google
Deutsche Bank	Yahoo!	Apple
Google	New York Times	Nintendo
Forbes	Netflix	Trivago
PayPal	Airbnb	Gitlab
Samsung	DropBox	Trustpilot

LA POPULARITÉ SUR GITHUB

	Angular	React	VueJS
Etoiles sur GitHub	77k +	177K +	190K +
Commits	22 295	14 610	3 208
Contributors	1 471	1 505	390
Licence	MIT	MIT	MIT

DOCUMENTATION OFFICIELLE (EN FRANÇAIS)

<https://fr.reactjs.org/>

PRÉPARATION DE L'ENVIRONNEMENT DE DÉVELOPPEMENT

A thin, vertical blue line is positioned to the right of the main title text.

INSTALLATION DE NODE.JS

Outils de développement de React dépendent de l'environnement Node.js

Lien pour installer Node.js (la version 16.13.0 LTS) : <https://nodejs.org/en/>

Vérifier l'installation de Node:

```
▪ node -v      soupr@LAPTOP-JN1AN42I MINGW64 ~/Desktop/Misc2/Baobab-ingenierie/Formation/Oxiane
▪ V16.13.0     $ node -v
                v16.13.0
```

NPM (Node Package Manager) qui est un gestionnaire de paquetage des projets Node qui est installé par défaut.

```
▪ npm -v      soupr@LAPTOP-JN1AN42I MINGW64 ~/Desktop/Misc2/Baobab-ingenierie/Formation/Oxiane
▪ v8.1.0      $ npm -v
                8.1.0
```


INSTALLATION D'UN ÉDITEUR DE CODE

Visual Studio Code

Visual Studio Code est un éditeur de code extensible développé par Microsoft pour Windows, Linux et macOS.

Lien pour télécharger VS Code : <https://code.visualstudio.com/>

Extension VSC pour **React**

- ES7 React/Redux/React-Native/JS snippets
- JS JSX Snippets

INSTALLATION DE REACT-DEVTOOLS

Une extension de Chrome/Firefox

Débogage des applications React complexes

Liens pour télécharger l'extension :

<https://chrome.google.com/webstore/detail/react-developer-tools/fmkadmapgofadopljbjfkapdkoienihi>

<https://addons.mozilla.org/fr/firefox/addon/react-devtools/>

INSTALLATION DE CREATE-REACT-APP PACKAGE

Create React App est un environnement confortable pour apprendre React, et constitue la meilleure option pour démarrer une nouvelle application web monopage (SPA) en React.

Il configure votre environnement de développement de façon à vous permettre d'utiliser les dernières fonctionnalités de JavaScript, propose une expérience développeur agréable et optimise votre application pour la production.

CRÉATION D'UN PROJET

Exécutez la commande suivante sur le terminal/cmd:

npx create-react-app todo

Commande ***npx*** :

- Est installé par défaut avec Node.js
- Est utilisé pour exécuter les packages Node

Argument ***create-react-app***

- Demande ***npx*** d'exécuter le package ***create-react-app***
- Est utilisé pour créer des projets React

Argument ***todo***

- C'est le nom de projet à créer

```
soupr@LAPTOP-JN1AN42I MINGW64 ~/Desktop/Misc2/Baobab-ingenierie/Formation/Oxiane/react/projects
$ npx create-react-app todo
Need to install the following packages:
  create-react-app
Ok to proceed? (y)
npm WARN deprecated tar@2.2.2: This version of tar is no longer supported, and will not receive security updates. Please upgrade asap.

Creating a new React app in C:\Users\soupr\Desktop\Misc2\Baobab-ingenierie\Formation\Oxiane\react\projects\todo.

Installing packages. This might take a couple of minutes.
Installing react, react-dom, and react-scripts with cra-template...

yarn add v1.9.4
[1/4] Resolving packages...

Success! Created todo at C:\Users\soupr\Desktop\Misc2\Baobab-ingenierie\Formation\Oxiane\react\projects\todo
do
Inside that directory, you can run several commands:

  yarn start
    Starts the development server.

  yarn build
    Bundles the app into static files for production.

  yarn test
    Starts the test runner.

  yarn eject
    Removes this tool and copies build dependencies, configuration files
    and scripts into the app directory. If you do this, you can't go back!

We suggest that you begin by typing:

  cd todo
  yarn start

Happy hacking!
```

VERSION DE REACT

Pour vérifier la version de React installée, exécuter la commande

- **`npm view react version`**

Pour explorer la liste des commandes create-react-app, exécuter la commande

- **`npx create-react-app --help`**

OUTITS MIS À DISPO PAR CREATE-REACT-APP

babel

- transpileur
- Permet de traduire le code JS (ES6, ES7, ...) et JSX en JS (ES5) qui est exécutable par tous les navigateurs = phase de transpilation
- <https://babeljs.io>

webpack

- bundler JavaScript
- construit le graphe de dépendances regroupe des ressources de même nature (.js ou .css...) dans un ou plusieurs bundles
- fonctionne avec un système de module : un fichier JS est un module, un fichier CSS est un module...

Un serveur de développement préconfiguré qui utilise NodeJs

ARBORESCENCE D'UN PROJET REACT

node modules

- contenant les fichiers (dépendances) nécessaires de la librairie nodeJS pour un projet **React**

public

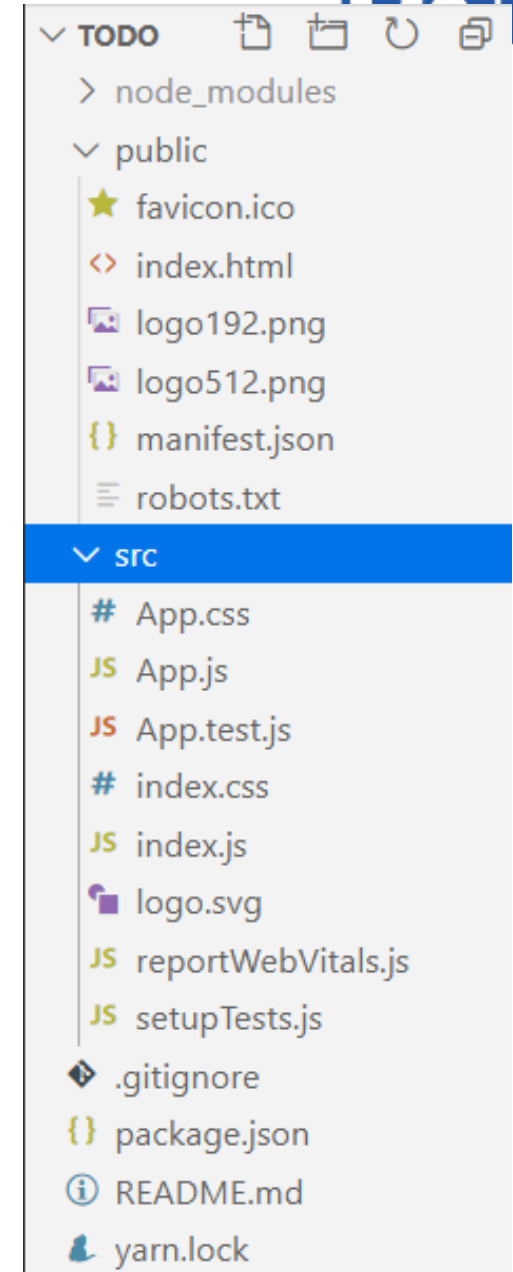
- contenant les fichiers accessibles de l'extérieur

src

- contenant les fichiers sources de l'application (exigé par **Webpack**)

package.json

- contenant l'ensemble de dépendance de l'application



ARBORESCENCE D'UN PROJET *REACT*

index.html

- L'unique fichier HTML d'une application *React*
- C'est le fichier HTML qui est chargé par le navigateur. Il contient un container (un élément div) dans lequel l'application React est affichée.

favicon.ico, logo192.png et logo512.png

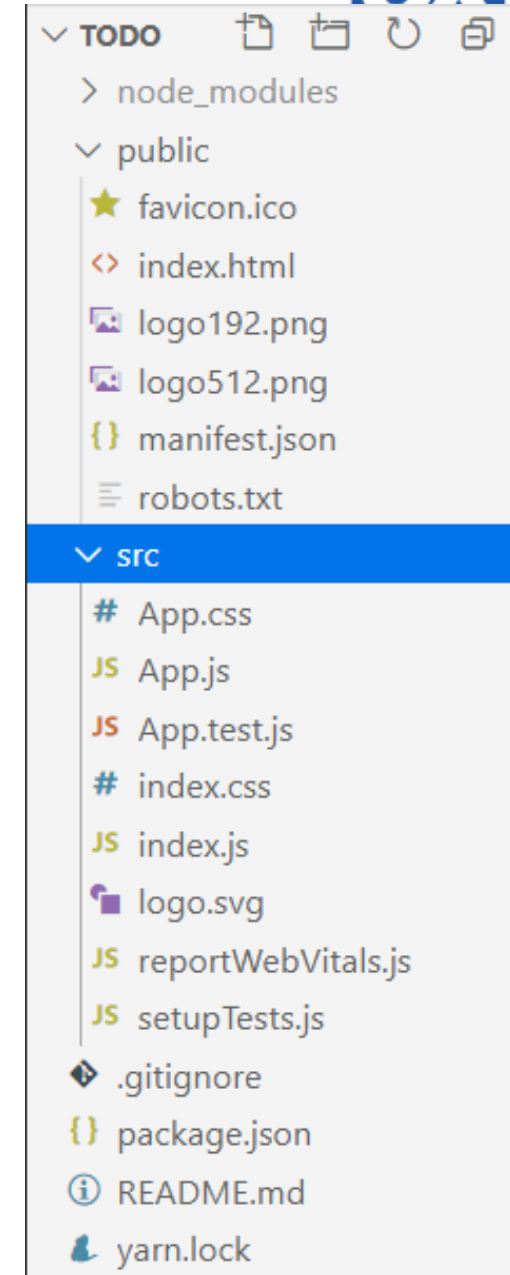
- les logo de **React**

manifest.json

- un fichier texte JSON permettant de décrire l'application (nom, auteur...) pour faciliter l'installation des applications Web sur l'écran d'accueil d'un appareil.

Robots.txt

- un fichier texte consulté par les moteurs de recherche pour savoir si l'affichage de contenu de l'application est autorisé dans les résultats de recherche.



QUE CONTIENT SRC?

index.js

- le point d'entrée de l'application

index.css

- la feuille de style associée au point d'entrée

App.js

- le premier composant

App.css

- la feuille de style associée au premier composant

App.test.js

- le fichier de test du premier composant

serviceWorker.js

- pour exécuter un script en arrière-plan séparément de la page Web.

setupTests.js

- le fichier de configuration globale de test

INDEX.JS — POINT D'ENTRÉE

```
import React from 'react';
import ReactDOM from 'react-dom';
import './index.css';
import App from './App';
import reportWebVitals from './reportWebVitals';
```

```
ReactDOM.render(
  <React.StrictMode>
    <App />
  </React.StrictMode>,
  document.getElementById('root')
);
```

React : API permettant de gérer les composants

ReactDOM : API permettant d'attacher les composants au DOM

DÉMARRAGE DES OUTILS DE DÉVELOPPEMENT

Exécutez la commande suivante sur le terminal/cmd:

npm start ou

yarn start

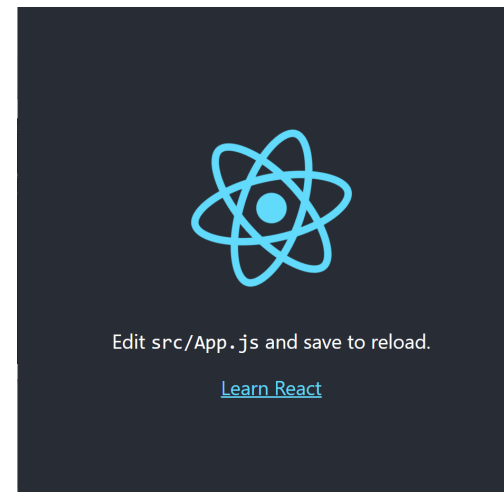
```
soupr@LAPTOP-JN1AN42I MINGW64 ~/Desktop/Misc2/Baobab-ingenierie/Formation/Oxiane/react/projects/todo (master)  
$ yarn start
```

Compiled successfully!

You can now view `todo` in the browser.

```
Local:      http://localhost:3000  
On Your Network: http://172.31.0.1:3000
```

Note that the development build is not optimized.
To create a production build, use **yarn build**.



L'ENVIRONNEMENT CLOUD POUR LE DEV

Quickly prototype ideas with code

Free, instant, collaborative sandboxes for rapid web development.



Create a Sandbox, it's free

<https://codesandbox.io/>

INTRODUCTION À JSX

JSX

JavaScript XML

une extension syntaxique à JavaScript

une expression JSX n'est ni du JavaScript ni
une chaîne de caractères

=> nécessite un *transpilage*

une expression placée entre accolades { }
est du code JavaScript interprété

attribut : class => className

```
render() {  
  return (  
    <div>  
      <h4 className="todo-header">  
        To Do List  
      </h4>  
    </div>  
  );  
}
```

EXPRESSIONS DANS JSX

On peut utiliser n'importe quelle expression JavaScript valide dans des accolades en JSX

```
function formatName(user) {  
  return user.firstName + ' ' + user.lastName;  
}  
  
const user = {  
  firstName: 'Kyllian',  
  lastName: 'Mbappé'  
};  
  
const element = (  
  <h1>  
    Bonjour, {formatName(user)} !  
  </h1>  
)  
;  
ReactDOM.render(  
  element,  
  document.getElementById('root')  
)  
;
```

ATTRIBUTS EN JSX

On peut utiliser des guillemets pour spécifier des littéraux chaînes de caractères dans les attributs

```
const element = <div tabIndex="0"></div>;
```

On peut aussi utiliser des accolades pour utiliser une expression JavaScript dans un attribut

```
const element = <img src={user.avatarUrl}></img>;
```


ÉLÉMENTS ENFANTS EN JSX

Les balises JSX peuvent contenir des enfants

```
const element = (  
  <div>  
    <h1>Bonjour !</h1>  
    <h2>Content de te voir ici.</h2>  
  </div>  
);
```

LA SÉCURITÉ

JSX empêche les attaques d'injection du code actif (XSS)

```
const title = response.potentiallyMaliciousInput;  
// Ceci est sans risque :  
const element = <h1>{title}</h1>;
```

TRANSPILATION: JSX À JS

Babel transpile le code JSX en code JS

```
const element = (  
  <h1 className="greeting">  
    Bonjour, monde !  
  </h1>  
);
```



```
const element = React.createElement(  
  'h1',  
  {className: 'greeting'},  
  'Bonjour, monde !'  
);
```

<https://babeljs.io/>

LE RENDU DES ÉLÉMENTS

Les éléments sont les blocs élémentaires d'une application React

Un élément décrit ce que l'on veut voir à l'écran

Les applications développées uniquement avec React ont généralement un seul nœud DOM racine :

```
<!-- index.html -->  
<div id="root"></div>
```

Pour faire le rendu d'un élément React dans un nœud DOM racine, on passe l'élément React et un nœud DOM racine à la méthode ReactDOM.render :

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



COMPOSANTS |

COMPOSANTS

Les composants nous permettent de découper l'interface utilisateur en éléments indépendants et réutilisables, nous permettant ainsi de considérer chaque élément de manière isolée.

Conceptuellement, les composants sont comme des fonctions JavaScript. Ils acceptent des entrées quelconques (appelées « props ») et renvoient des éléments React décrivant ce qui doit apparaître à l'écran.

Tous les composants React commencent par une majuscule (**obligatoire**) !

DÉFINITION DE COMPOSANT *REACT*

Deux solutions pour la définition de composant

Fonctions

- fonctions composants
- moyen le plus simple de définir un composant
- stateless = sans état
- possibilité d'ajout d'état avec les Hooks depuis la version 16.8
- renvoie un élément React

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

Classes

- Composant avec état ou stateful (complex)
- Déclaration en suivant les classes de ES6
- Méthode **render()** **obligatoire** qui renvoie un élément React

```
class Welcome extends React.Component {  
  render() {  
    return <h1>Bonjour, {this.props.name}</h1>;  
  }  
}
```

PROPS

Les props sont en lecture seule

- Que vous déclariez un composant sous forme de **fonction** ou de **classe**, il ne doit jamais modifier ses propres props

Fonctions « pures »

```
function sum(a, b) {  
  return a + b;  
}
```

- Ces fonctions sont dites « pures » parce qu'elles ne tentent pas de modifier leurs entrées et retournent toujours le même résultat pour les mêmes entrées

Fonctions « impures »

```
function withdraw(account, amount) {  
  account.total -= amount;  
}
```

- cette fonction est impure car elle modifie sa propre entrée

Tout composant React doit agir comme une fonction pure vis-à-vis de ses props

COMPOSITION DE COMPOSANTS

Les composants peuvent faire référence à d'autres composants dans leur sortie

Remarques:

- React considère les composants commençant par des lettres minuscules comme des balises DOM
- Par exemple, `<div />` représente une balise HTML div, mais `<Welcome />` représente un composant, et exige que l'identifiant `Welcome` existe dans la portée courante

```
function Welcome(props) {  
  return <h1>Bonjour, {props.name}</h1>;  
}  
function App() {  
  return (  
    <div>  
      <Welcome name="Sara" />  
      <Welcome name="Cahal" />  
      <Welcome name="Edite" />  
    </div>  
  );  
}  
ReactDOM.render(  
  <App />,  
  document.getElementById('root')  
);
```

EXTRACTION DES COMPOSANTS

Problème

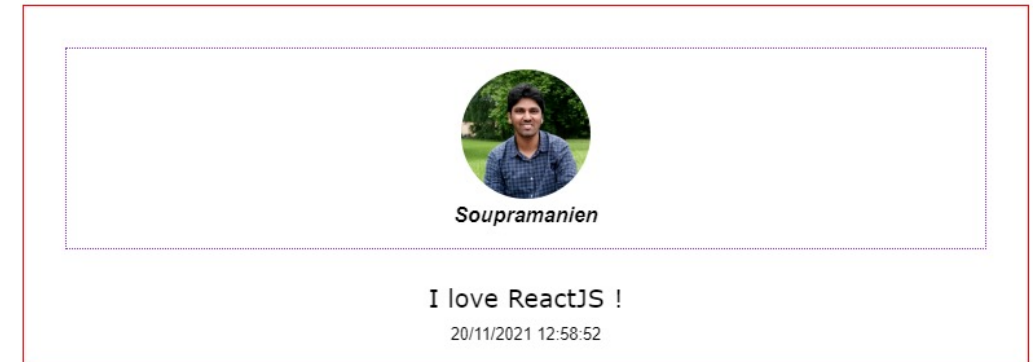
- Nombreuses imbrications au sein du composant le rendent difficile à maintenir, et nous empêchent d'en réutiliser des parties individuelles

Solution

- Scinder des composants en composants plus petits

ETUDE DE CAS

```
import React from "react";
import moment from "moment";
function formatDate(date) {
  return moment(date).format('DD/MM/YYYY hh:mm:ss');
}
export default function Comment(props) {
  return (
    <div className="Comment">
      <div className="UserInfo">
        <img
          className="Avatar"
          src={props.author.avatarUrl}
          alt={props.author.name}
        />
        <div className="UserInfo-name">{props.author.name}</div>
      </div>
      <div className="Comment-text">{props.text}</div>
      <div className="Comment-date">{formatDate(props.date)}</div>
    </div>
  );
}
```

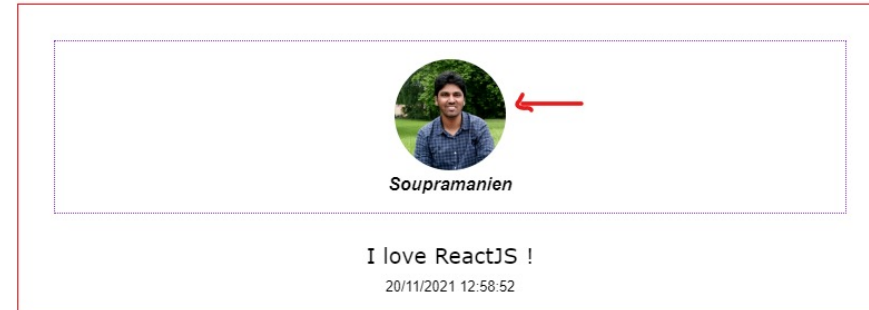


COMPOSANT AVATAR

Pour commencer, nous allons extraire le composant **Avatar**:

```
function Avatar(props) {
  return (
    <img className="Avatar"
      src={props.user.avatarUrl}
      alt={props.user.name}
    />
  );
}
```

Remarque: on donne un nom générique : **user** plutôt que **author**

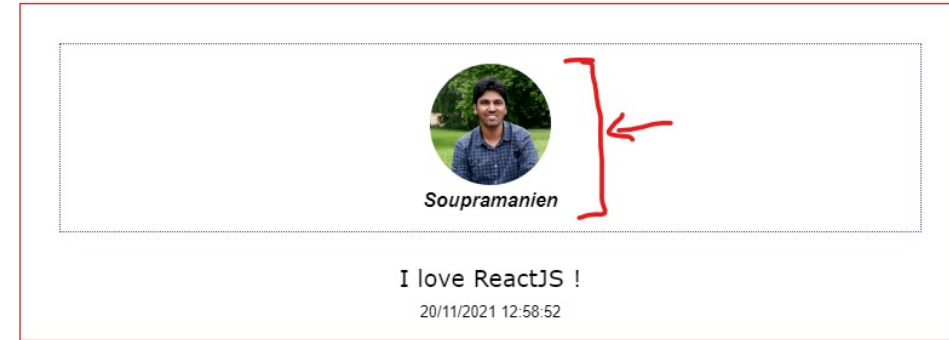


```
function Comment(props) {
  return (
    <div className="Comment">
      <div className="UserInfo">
        <Avatar user={props.author}/>
        <div className="UserInfo-name">
          {props.author.name}
        </div>
      </div>
      <div className="Comment-text">
        {props.text}
      </div>
      <div className="Comment-date">
        {formatDate(props.date)}
      </div>
    </div>
  );
}
```

COMPOSANT USERINFO

Ensuite, nous allons extraire un composant **UserInfo** qui affiche un **Avatar**:

```
function UserInfo(props) {
  return (
    <div className="UserInfo">
      <Avatar user={props.user} />
      <div className="UserInfo-name">
        {props.user.name}
      </div>
    </div>
  );
}
```



```
function Comment(props) {
  return (
    <div className="Comment">
      <UserInfo user={props.author}/>
      <div className="Comment-text">
        {props.text}
      </div>
      <div className="Comment-date">
        {formatDate(props.date)}
      </div>
    </div>
  );
}
```

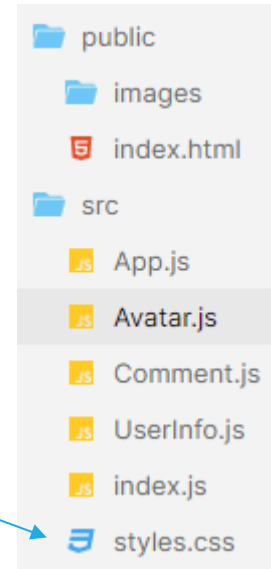
RÈGLES POUR TROUVER LE BON CANDIDAT POUR EXTRACTION

En règle générale, si une partie de votre interface utilisateur est utilisée plusieurs fois (Button, Panel, Avatar, productListElement), ou si elle est suffisamment complexe en elle-même (Comment, ProductList, App), c'est un bon candidat pour une extraction en tant que composant séparé

AJOUT DU STYLE

Pour ajouter du style à vos composants

- créer un fichier **styles.css** dans le dossier **src** contenant nos styles
- Dans **App.js**, importer ce fichier par la directive **import**
`import './styles.css';`
- dans nos composants d'ajouter les classes aux balises à l'aide de la propriété **className**



```
//styles.css
.Avatar {
  width: 100px;
  height: 100px;
  border-radius: 50%;
}
```

...

```
function Avatar(props) {
  return (
    <img className="Avatar"
      src={props.user.avatarUrl}
      alt={props.user.name} />
  );
}
```

FONCTIONS COMPOSANTS

FONCTIONS COMPOSANTS

fonction dont le résultat est le contenu du composant.

le nom d'un composant commence par une majuscule.

```
export default function App() {  
  return "Hello, World";  
}
```

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

COMPOSANTS À BASE DE CLASSES

COMPOSANTS À BASE DE CLASSES

Il est 14:38:56.

extends React.component

les propriétés **props** sont en paramètre du constructeur

- appel de **super(props)**
- props devient **this.props**

La variable d'instance **this.state** est un objet contenant l'état de composant qui **participe au flux de données**

c'est la méthode **render()** qui renvoie la vue du composant

le résultat ne peut avoir qu'un seul composant racine

approche déclarative : on décrit dans **render()** ce que l'on veut avoir

```
class Clock extends React.Component {
  constructor(props) {
    super(props);
    this.state = {date: new Date()};
  }
  render() {
    return (
      <div>
        <h1>{this.props.message}</h1>
        <h2>Il est {this.state.date.toLocaleTimeString()}.
        </h2>
      </div>
    );
  }
}
ReactDOM.render(
  <Clock />,
  document.getElementById('root')
);
```

DEUX FAÇONS DE GÉRER LES DONNÉES

Données qui changent (mutable) :

- on utilise un état (state)

Données qui ne changent pas (immutable) :

- on utilise des propriétés (props)

On essaie de minimiser les données qui changent

MÉTHODES DE CYCLE DE VIE

Problème

- Le composant **Clock** affiche l'heure mais ne se met pas à jour

Solution

- nous allons faire en sorte que le composant **Clock** mette en place un minuteur et se mette à jour toutes les secondes
- Nous voulons mettre en place un minuteur quand une Horloge apparaît dans le DOM pour la première fois. Le terme React « mount » désigne cette phase.
- Nous voulons également nettoyer le minuteur quand le DOM produit par l'Horloge est supprimé. En React, on parle de « unmount ».
- Nous pouvons déclarer des méthodes spéciales sur un composant à base de classe pour exécuter du code quand un composant est monté et démonté

COMPONENTDIDMOUNT

componentDidMount est une méthode de cycle de vie qui est exécutée après que la sortie du composant a été injectée dans le DOM.

C'est un bon endroit pour mettre en place le minuteur :

```
componentDidMount() {  
  this.timerID = setInterval(  
    () => this.tick(),  
    1000  
  );  
}
```

Remarque :

- `this.props` et `this.state` participent au flux de données
- Vous pouvez ajouter manuellement d'autres champs sur la classe si vous avez besoin de stocker quelque chose qui ne participe pas au flux de données (comme un ID de minuteur)

COMPONENTWILLUNMOUNT

componentWillUnmount est une méthode de cycle de vie qui est exécuté avant le démontage de composant du DOM

C'est un bon endroit pour détruire le minuteur

```
componentWillUnmount() {  
  clearInterval(this.timerID);  
}
```

SETSTATE

La méthode **setState** permet de **planifier** une mise à jour de l'état local du composant **React** invoque la méthode **render()** pour chaque appel de la méthode **setState** nous allons implémenter une méthode appelée **tick()** que le composant **Clock** va exécuter toutes les secondes

```
tick() {  
  this.setState({  
    date: new Date()  
  });  
}
```

Ici, on planifie la mise à jour de la propriété **date** du composant Clock


```
class Clock extends React.Component {
  constructor(props) {
    super(props);
    this.state = {date: new Date()};
  }
  componentDidMount() {
    this.timerID = setInterval(
      () => this.tick(),
      1000
    );
  }
  componentWillUnmount() {
    clearInterval(this.timerID);
  }
  tick() {
    this.setState({
      date: new Date()
    });
  }
  render() {
    return (
      <div>
        <h1>Bonjour, monde !</h1>
        <h2>Il est {this.state.date.toLocaleTimeString()}.</h2>
      </div>
    );
  }
}
ReactDOM.render(
  <Clock />,
  document.getElementById('root')
);
```

COMMENT UTILISER L'ÉTAT LOCAL CORRECTEMENT ?

Il y'a trois choses que vous devriez savoir à propos de **setState()**

1. Ne modifiez pas l'état directement

Par exemple, ceci ne déclenchera pas un rafraîchissement du composant :

```
// Erroné  
this.state.comment = 'Bonjour';
```

À la place, utilisez `setState()` :

```
// Correct  
this.setState({comment: 'Bonjour'});
```

Le seul endroit où vous pouvez affecter `this.state`, c'est le constructeur.

COMMENT UTILISER L'ÉTAT LOCAL CORRECTEMENT ?

2. Les mises à jour de l'état peuvent être asynchrones

React peut grouper plusieurs appels à **setState()** en une seule mise à jour pour des raisons de performance.

Comme **this.props** et **this.state** peuvent être mises à jour de façon asynchrone, vous ne devez pas vous baser sur leurs valeurs pour calculer le prochain état.

Par exemple, ce code peut échouer pour mettre à jour un **compteur** :

```
// Erroné
this.setState({
  counter: this.state.counter + this.props.increment,
});

// Correct
this.setState((state, props) => ({
  counter: state.counter + props.increment
}));
```

Pour remédier à ce problème, utilisez la seconde forme de **setState()** qui accepte une fonction à la place d'un objet. Cette fonction recevra l'état précédent comme premier argument et les **props** au moment de la mise à jour comme second argument :

COMMENT UTILISER L'ÉTAT LOCAL CORRECTEMENT ?

3. Les mises à jour de l'état sont fusionnées

Quand vous invoquez **setState()**, React fusionne les objets que vous donnez avec l'état actuel.

Par exemple, votre état peut contenir plusieurs variables indépendantes :

Ensuite, vous pouvez les mettre à jour indépendamment avec des appels séparés à **setState()** :

La fusion n'est pas profonde, donc **this.setState({comments})** laisse **this.state.posts** intacte, mais remplace complètement **this.state.comments**.

```
constructor(props) {  
  super(props);  
  this.state = {  
    posts: [],  
    comments: []  
  };  
}
```

```
componentDidMount() {  
  fetchPosts().then(response => {  
    this.setState({  
      posts: response.posts  
    });  
  });  
}
```

```
fetchComments().then(response => {  
  this.setState({  
    comments: response.comments  
  });  
});  
}
```

DEFAULTPROPS ET PROPTYPES

defaultProps et propTypes:

valeur par défaut et contraintes sur les propriétés

validation des valeurs des **props** à l'exécution, phase de développement

message *warning* dans la console en cas de non respect

```
import React from "react";
import PropTypes from "prop-types";
export default class Person extends React.Component {
  ...
}
Person.defaultProps = {
  name: "Anonymous",
};
Person.propTypes = {
  name: PropTypes.string,
  age: PropTypes.number.isRequired,
};
```

FLUX DE DONNÉES

LES DONNÉES DESCENDENT

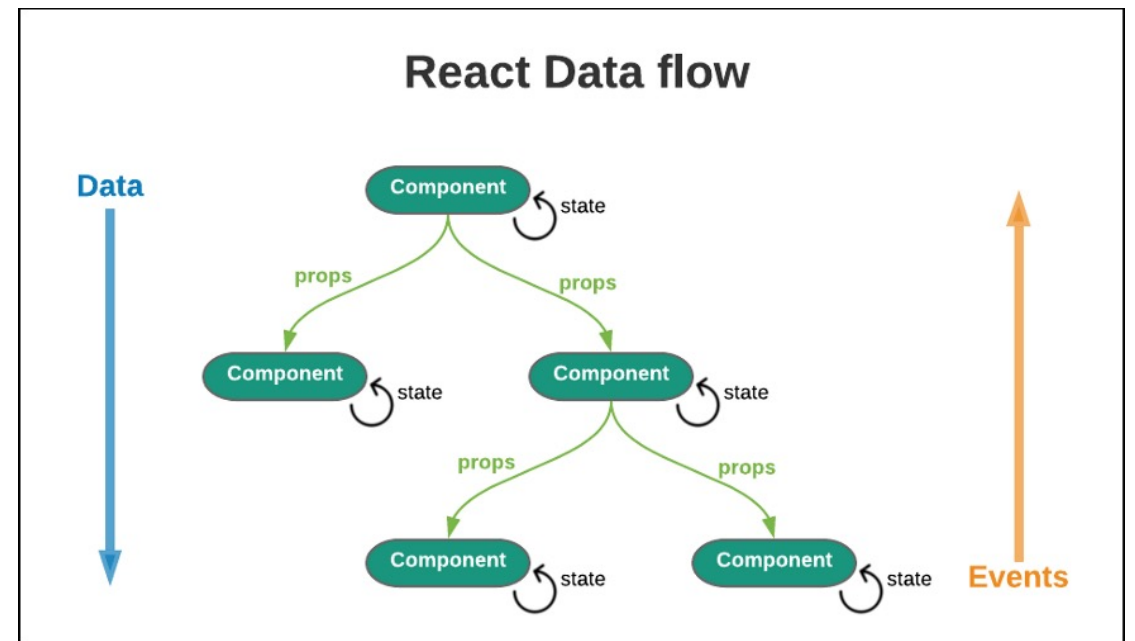
Un composant peut choisir de passer son état à ses enfants via des props

```
<FormattedDate date={this.state.date} />
```

```
function FormattedDate(props) {  
  return <h2>Il est  
    {props.date.toLocaleTimeString()}</h2>;  
}
```

C'est appelé un flux de données « du haut vers le bas » ou « unidirectionnel »

Un état local est toujours possédé par un composant spécifique, et toute donnée ou interface utilisateur dérivée de cet état ne peut affecter que les composants « en-dessous » de celui-ci dans l'arbre de composants.



LES ÉVÉNEMENTS EN REACT

LA GESTION DES ÉVÉNEMENTS

La gestion des événements pour les éléments React est très similaire à celle des éléments du DOM.

Il y a tout de même quelques différences de syntaxe :

- Les événements de React sont nommés en **camelCase** plutôt qu'en minuscules
- En JSX on passe une fonction comme gestionnaire d'événements plutôt qu'une chaîne de caractères

```
//HTML  
<button onclick="activateLasers()">  
  Activer les lasers  
</button>
```

```
//JSX  
<button onClick={activateLasers}>  
  Activer les lasers  
</button>
```

LA GESTION DES ÉVÉNEMENTS

Autre différence importante

- En React, on ne peut pas renvoyer **false** pour empêcher le comportement par défaut.
- Vous devez appeler explicitement `preventDefault`.
- Par exemple, en HTML, pour annuler le comportement par défaut des liens qui consiste à ouvrir une nouvelle page, vous pourriez écrire :

```
<a href="#"  
  onclick="console.log('Le lien a  
    été cliqué.');" return false">  
  Cliquez ici  
</a>
```

- En React, ça pourrait être :

```
function ActionLink() {  
  function handleClick(e) {  
    e.preventDefault();  
    console.log('Le lien a été cliqué.');  }  
  
  return (  
    <a href="#" onClick={handleClick}>  
      Cliquez ici  
    </a>  
  );  
}
```

COMPOSANTS À BASE DE CLASSE

Problème

Si le gestionnaire d'événements est une méthode de la classe et elle utilise **this**, la méthode de classes n'est pas liée par défaut donc **this** vaut **undefined**

Solution 1

Lier **this** la méthode avec la méthode **bind**

```
class Toggle extends React.Component {  
  constructor(props) {  
    super(props);  
    this.state = { isToggleOn: true };  
    // Cette liaison est nécessaire afin de permettre  
    // l'utilisation de `this` dans la fonction de rappel.  
    this.handleClick = this.handleClick.bind(this);  
  }  
  handleClick() {  
    this.setState(state => ({  
      isToggleOn: !state.isToggleOn  
    }));  
  }  
  render() {  
    return (  
      <button onClick={this.handleClick}>  
        {this.state.isToggleOn ? 'ON' : 'OFF'}  
      </button>  
    );  
  }  
}
```

COMPOSANTS À BASE DE CLASSE

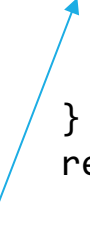
Problème

Si le gestionnaire d'événements est une méthode de la classe et elle utilise **this**, la méthode de classes n'est pas liée par défaut donc **this** vaut **undefined**

Solution 2

Utiliser la syntaxe de **fonction fléchée** pour définir la méthode de gestionnaire d'événements

```
class Toggle extends React.Component {  
  constructor(props) {  
    super(props);  
    this.state = { isToggleOn: true };  
  }  
  handleClick = () => {  
    this.setState(state => ({  
      isToggleOn: !state.isToggleOn  
    }));  
  }  
  render() {  
    return (  
      <button onClick={this.handleClick}>  
        {this.state.isToggleOn ? 'ON' : 'OFF'}  
      </button>  
    );  
  }  
}
```



PASSAGE DES ARGUMENTS À UN GESTIONNAIRE D'ÉVÉNEMENTS

Au sein d'une boucle, il est courant de vouloir passer un argument supplémentaire à un gestionnaire d'événements.

Par exemple, si **id** représente la ligne sélectionnée, on peut faire au choix :

```
<button onClick={(e) => this.deleteRow(id, e)}>Supprimer la ligne</button>  
<button onClick={this.deleteRow.bind(this, id)}>Supprimer la ligne</button>
```

Les lignes précédentes sont équivalentes et utilisent respectivement les **fonctions fléchées** et **Function.prototype.bind**.

Dans les deux cas, l'argument **e** représente l'événement React qui sera passé en second argument après **l'ID**.

Avec une **fonction fléchée**, nous devons passer l'argument explicitement, alors qu'avec **bind** tous les arguments sont automatiquement transmis.

AFFICHAGE CONDITIONNEL



AFFICHAGE CONDITIONNEL

L'affichage conditionnel en **React** fonctionne de la même façon que les conditions en **JavaScript**.

- Utilisation de l'instruction JavaScript **if** ou l'opérateur ternaire

```
function UserGreeting(props) {  
  return <h1>Bienvenue !</h1>;  
}
```

```
function GuestGreeting(props) {  
  return <h1>Veuillez vous inscrire.</h1>;  
}
```

```
function Greeting(props) {  
  const isLoggedIn = props.isLoggedIn;  
  if (isLoggedIn) {  
    return <UserGreeting />;  
  }  
  return <GuestGreeting />;  
}
```

```
function Greeting(props) {  
  const isLoggedIn = props.isLoggedIn;  
  (isLoggedIn) ? <UserGreeting /> :  
                <GuestGreeting />;  
}
```

```
class LoginControl extends React.Component {
  constructor(props) {
    super(props);
    this.state = {isLoggedIn: false};
  }
  handleLoginClick = () => this.setState({isLoggedIn: true})
  handleLogoutClick = () => this.setState({isLoggedIn: false})

  render() {
    const isLoggedIn = this.state.isLoggedIn;
    let button = (isLoggedIn) ? <LogoutButton onClick={this.handleLogoutClick} /> :
                        <LoginButton onClick={this.handleLoginClick} />;

    return (
      <div>
        <Greeting isLoggedIn={isLoggedIn} />
        {button}
      </div>
    );
  }
}
```

```
render() { //version alternative
  const isLoggedIn = this.state.isLoggedIn;
  return (
    <div>
      <Greeting isLoggedIn={isLoggedIn} />
      {(isLoggedIn) ? <LogoutButton onClick={this.handleLogoutClick} /> :
                      <LoginButton onClick={this.handleLoginClick} />}
    </div>
  );
}
```


CONDITION À LA VOLÉE AVEC L'OPÉRATEUR LOGIQUE &&

Vous pouvez utiliser n'importe quelle expression dans du JSX en l'enveloppant dans des accolades. Ça vaut aussi pour l'opérateur logique Javascript &&. Il peut être pratique pour inclure conditionnellement un élément :

```
function Mailbox(props) {  
  const unreadMessages = props.unreadMessages;  
  return (  
    <div>  
      <h1>Bonjour !</h1>  
      {unreadMessages.length > 0 &&  
        <h2>  
          Vous avez {unreadMessages.length} message(s) non-lu(s).  
        </h2>  
      }  
    </div>  
  );  
}
```

EMPÊCHER L’AFFICHAGE D’UN COMPOSANT

On renvoie **null** au lieu de son affichage habituel

```
function WarningBanner(props) {  
  if (!props.warn) {  
    return null;  
  }  
  
  return (  
    <div className="warning">  
      Attention !  
    </div>  
  );  
}
```



LA LISTE |

AFFICHER PLUSIEURS COMPOSANTS

On peut construire des collections d'éléments React à partir d'un tableau JavaScript

```
function NumberList(props) {  
  const numbers = props.numbers;  
  const listItems = numbers.map((number) =>  
    <li>{number}</li>  
  );  
  return (  
    <ul>{listItems}</ul>  
  );  
}
```

```
const numbers = [1, 2, 3, 4, 5];  
ReactDOM.render(  
  <NumberList numbers={numbers} />,  
  document.getElementById('root')  
);
```

- 1
- 2
- 3
- 4
- 5

En exécutant ce code, vous obtiendrez un avertissement disant qu'une **clé** devrait être fournie pour les éléments d'une liste

LA LISTE ET 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

La clé utilisée doit identifier de façon unique un élément d'une liste

- Ex: clé peut être ID de l'objet **todo** dans une **todo** liste

Règle simple à retenir

- chaque élément à l'intérieur d'un appel à `map()` a besoin d'une clé

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

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

INDEX DE L'ÉLÉMENT COMME CLÉS

Vous pouvez utiliser l'index de l'élément comme **clé**

- Si et seulement si l'ordre des éléments ne risque pas de changer

```
const todoItems = todos.map((todo, index) =>
  // Ne faites ceci que si les éléments n'ont pas d'ID stable et
  // l'ordre des éléments ne risque pas de changer
  <li key={index}>
    {todo.text}
  </li>
);
```

Il ne faut pas utilisé l'index de l'élément comme **clé**

- si l'ordre des éléments est susceptible de changer
- Ça peut avoir un effet négatif sur les performances, et causer des problèmes avec l'état du composant

EXTRACTION DES COMPOSANTS AVEC DES CLÉS

Les clés n'ont une signification que dans le contexte du tableau qui les entoure.

Par exemple, si on extrait un composant `ListItem`, on doit garder la clé sur l'élément `<ListItem />` dans le tableau, et non sur l'élément `` dans le composant `ListItem` lui-même.

```
function ListItem(props) {  
  // Correct ! Pas la peine de spécifier la clé ici :  
  return <li>{props.value}</li>;  
}  
  
function NumberList(props) {  
  const numbers = props.numbers;  
  const listItems = numbers.map((number) =>  
    // Correct ! La clé doit être spécifiée dans le tableau.  
    <ListItem key={number.toString()} value={number} />  
  );  
  return (  
    <ul>  
      {listItems}  
    </ul>  
  );  
}
```

LA CLÉ ET L'UNICITÉ

Les clés n'ont besoin d'être uniques qu'au sein de la liste

On peut utiliser les mêmes clés dans des tableaux différents

```
const posts = [  
  {id: 1, title: 'Bonjour, monde', content:  
    'Bienvenue sur la doc de React !'},  
  {id: 2, title: 'Installation', content: 'Vous  
pouvez installer React depuis npm.'}  
];  
ReactDOM.render(  
  <Blog posts={posts} />,  
  document.getElementById('root')  
);
```

```
function Blog(props) {  
  const sidebar = (  
    <ul>  
      {props.posts.map((post) =>  
        <li key={post.id}>  
          {post.title}  
        </li>  
      )}  
    </ul>  
  );  
  const content = props.posts.map((post) =>  
    <div key={post.id}>  
      <h3>{post.title}</h3>  
      <p>{post.content}</p>  
    </div>  
  );  
  return (  
    <div>  
      {sidebar}  
      <hr />  
      {content}  
    </div>  
  );  
}
```


KEY != ID

Les clés servent d'indicateur à React et le composant enfant n'a pas accès à la clé.

```
const content = posts.map((post) =>
  <Post
    key={post.id}
    id={post.id}
    title={post.title} />
);
```

Dans l'exemple ci-dessus, le composant **Post** peut accéder à **props.id**, mais pas à **props.key**.



FORMULAIRES

composants contrôlés et non-
contrôlés

FORMULAIRE HTML

Les formulaires HTML fonctionnent un peu différemment des autres éléments du DOM en React car ils possèdent naturellement un état interne

un formulaire HTML 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** ».

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

COMPOSANTS CONTRÔLÉS

Un champ de formulaire dont l'état est contrôlé par React est appelé un « composant contrôlé »

L'état modifiable est généralement stocké dans la propriété **state** des composants et mis à jour uniquement avec **setState()**

Le composant React qui affiche le formulaire contrôle aussi son comportement par rapport aux saisies de l'utilisateur

```

class NameForm extends React.Component {
  constructor(props) {
    super(props);
    this.state = {value: ''};

    this.handleChange = this.handleChange.bind(this);
    this.handleSubmit = this.handleSubmit.bind(this);
  }

  handleChange(event) {
    this.setState({value: event.target.value});
  }

  handleSubmit(event) {
    alert('Le nom a été soumis : ' + this.state.value);
    event.preventDefault();
  }

  render() {
    return (
      <form onSubmit={this.handleSubmit}>
        <label>
          Nom :
          <input type="text" value={this.state.value} onChange={this.handleChange} />
        </label>
        <input type="submit" value="Envoyer" />
      </form>
    );
  }
}

```

À présent que l'attribut **value** est défini sur notre élément de formulaire, la valeur affichée sera toujours **this.state.value**, faisant ainsi de l'état local React la source de vérité.

Puisque **handleChange** est déclenché à chaque frappe pour mettre à jour l'état local React, la valeur affichée restera mise à jour au fil de la saisie

Dans un composant contrôlé, la valeur du champ est en permanence pilotée par l'état React

GESTION DE PLUSIEURS SAISIES

Quand vous souhaitez gérer plusieurs champs contrôlés, vous pouvez ajouter un attribut **name** à chaque champ et laisser la fonction gestionnaire choisir quoi faire en fonction de la valeur de **event.target.name**.

```
class Reservation extends React.Component {
  constructor(props) {
    super(props);
    this.state = {
      isGoing: true, numberOfGuests: 2
    };
  }
  handleInputChange = (event) => {
    const target = event.target;
    const value = target.type === 'checkbox' ? target.checked : target.value;
    const name = target.name;
    this.setState({
      [name]: value
    });
  }
  render() {
    return (
      <form>
        <label>
          Participe :
          <input name="isGoing" type="checkbox" checked={this.state.isGoing} onChange={(e)=>this.handleInputChange(e)} />
        </label>
        <br />
        <label>
          Nombre d'invités :
          <input name="numberOfGuests" type="number" value={this.state.numberOfGuests}
onChange={(e)=>this.handleInputChange(e)} />
        </label>
      </form>
    );
  }
}
```

← Notez l'utilisation de la syntaxe des propriétés calculées pour mettre à jour la valeur de l'état correspondant au nom du champ

COMPOSANTS NON-CONTRÔLÉS

Composant contrôlé

- les données du formulaires sont gérées par le composant React.

composant non-contrôlé

- L'alternative à « composant contrôlé »
- Les données sont gérées par le DOM
- Utilisation de **ref** à la place de gestionnaire d'événements pour chaque mise à jour de l'état d'un composant (code plus concis)
- Intégration simple du code React à base de composants non-contrôlés avec du code non-React


```
class NameForm extends React.Component {
  constructor(props) {
    super(props);
    this.handleSubmit = this.handleSubmit.bind(this);
    this.input = React.createRef(); ← On crée un objet ref
  }
}
```

```
handleSubmit(event) {
  alert('Un nom a été envoyé : ' + this.input.current.value);
  event.preventDefault();
}
```

Quand une ref est passée à un élément dans render, une référence au nœud devient accessible via l'attribut **current** de la ref

```
render() {
  return (
    <form onSubmit={this.handleSubmit}>
      <label>
        Nom :
        <input type="text" ref={this.input} />
      </label>
      <input type="submit" value="Envoyer" />
    </form>
  );
}
```

On attache **ref** à l'élément React via l'attribut **ref**

FRAGMENTS

En React, il est courant pour un composant de renvoyer plusieurs éléments.

Les fragments nous permettent de grouper une liste d'enfants sans ajouter de nœud supplémentaire au DOM.

Rappel : en JSX, on ne peut renvoyer qu'un élément racine.

```
render() {  
  return (  
    <div> ← Inutile de créer un nœud div  
      <ChildA />  
      <ChildB />  
      <ChildC />  
    </div>  
  );  
}
```

```
render() {  
  return (  
    <React.Fragment>  
      <ChildA />  
      <ChildB />  
      <ChildC />  
    </React.Fragment>  
  );  
}
```

```
ou  
render() {  
  return (  
    <>  
      <ChildA />  
      <ChildB />  
      <ChildC />  
    </>  
  );  
}
```

FAIRE REMONTER L'ÉTAT |

FAIRE REMONTER L'ÉTAT

Il ne doit y avoir qu'une seule « source de vérité » pour toute donnée qui change dans une application React

Étapes :

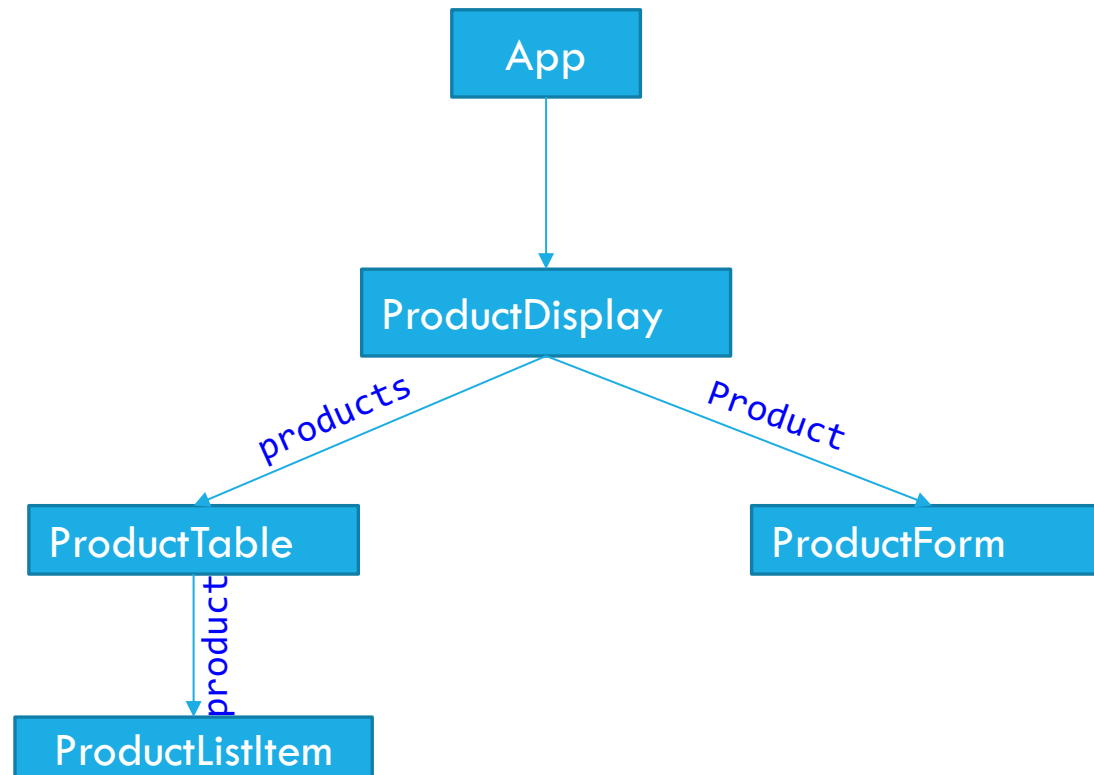
- Ajoute de l'état au composant qui en a besoin pour s'afficher
- Faire remonter l'état dans l'ancêtre commun le plus proche, si d'autres composants en ont également besoin
- Baser sur des données qui se propagent du haut vers le bas pour la synchronisation des états de différents composants

Faire remonter l'état implique d'écrire plus de code générique mais ça demande moins de travail pour trouver et isoler les bugs

Puisque tout état « vit » dans un composant et que seul ce composant peut le changer, la surface d'impact des bugs est grandement réduite

ETUDE DE CAS

Développer l'application qui permet de gérer les produits.



```
export default class ProductDisplay extends React.Component
{
  componentDidMount = () =>
  {
    productService.getProducts()
      .then((data) => {
        return data.json()
      })
      .then((res) => {
        this.setState({ products: res })
      })
      .catch((err) => {
        this.setState({ networkError: true })
      })
  }
  constructor(props) {
    super(props);
    this.state = {
      networkError: false,
      startEditing: false,
      product: {},
      products: []
    }
  }
  deleteProduct = (productId) => {
    productService.deleteProduct(productId);
  }
}
```

```
showForm = (product) => {
  this.setState({ startEditing: true, product: product });
}
cancel = () => {
  this.setState({ startEditing: false, product: {} });
}
save = (product) => {
  //ajout d'un nouveau produit
  if (!product.id) {
    productService.addProduct(product)
  }
  else {
    productService.updateProduct(product)
  }
}
render() {
  if (this.state.networkError) {
    return <p>problème de réseau !</p>
  } else {
    return this.state.startEditing ?
      <ProductForm product={this.state.product}
        cancelCallback={this.cancel}
        saveCallback={this.save} /> :
      <ProductTable products={this.state.products}
        showForm={this.showForm}
        deleteCallback={this.deleteProduct} />
    }
  }
}
```

```

export default class ProductTable extends React.Component{
  constructor(props){
    super(props);
  }
  render(){
    return(
      <React.Fragment>
        <table>
          <caption>Produits</caption>
          <tr>
            <th>ID</th>
            <th>Nom</th>
            <th>Catégorie</th>
            <th>Prix</th>
            <th>Image</th>
            <th>Action</th>
          </tr>
          {this.props.products.map((product)=>{
            return(
              <ProductListItem product={product} showForm={this.props.showForm}
deleteCallback={this.props.deleteCallback}/>
            );
          })}
        </table>
        <button onClick={()=>this.props.showForm({})}>Créer Produit</button>
      </React.Fragment>
    );
  }
}

```



```
export default function ProductListItem(props){
  const product = props.product;
  return (
    <tr key={product.id.toString()}>
      <td>{product.id}</td>
      <td>{product.name}</td>
      <td>{product.category}</td>
      <td>{product.price}&euro;</td>
      <td><img src={` /images/${product.name}.jpg` } width="50" height="50"/></td>
      <td>
        <button onClick={()=>props.showForm(product)}>Modifier</button>
        <button onClick={()=>props.deleteCallback(product.id)}>Supprimer</button>
      </td>
    </tr>
  )
}
```

```
export default class ProductForm extends React.Component{
  constructor(props){
    super(props);
    this.state = {
      id : props.product.id || "",
      name : props.product.name || "",
      category : props.product.category || "",
      price : props.product.price || "",
    }
  }
  handleChange = (evt)=>{
    evt.persist();
    this.setState((state)=>state[evt.target.name] = evt.target.value)
  }
  save = (evt)=>{
    evt.preventDefault();
    let product = {
      id : this.state.id,
      name : this.state.name,
      category : this.state.category,
      price : this.state.price,
    }
    this.props.saveCallback(product);
  }
  render(){
    return(
      <form>
        <input type="text" name="id" value={this.state.id} placeholder="id" readOnly/>
        <input type="text" name="name" value={this.state.name} placeholder="nom" onChange={this.handleChange}/>
        <input type="text" name="category" value={this.state.category} placeholder="catégorie" onChange={this.handleChange}/>
        <input type="number" name="price" value={this.state.price} placeholder="Prix" onChange={this.handleChange}/>
        <button onClick={this.save}>Enregistrer</button>
        <button onClick={this.props.cancelCallback}>Annuler</button>
      </form>
    );
  }
}
```

COMPOSANT D'ORDRE SUPÉRIEUR

Un composant d'ordre supérieur (*Higher-Order Component* ou *HOC*) est une technique avancée de React qui permet de réutiliser la logique de composants

Les HOC ne font pas partie de l'API de React à proprement parler, mais découlent de sa nature compositionnelle

Concrètement, **un composant d'ordre supérieur est une fonction qui accepte un composant et renvoie un nouveau composant.**

Les HOC sont courants dans des bibliothèques tierces de **React**, comme **connect** dans **Redux**

EXEMPLE

Une fonction qui prend en argument un composant quelconque et ajoute la fonctionnalité de log.

```
function logProps(WrappedComponent) {  
  return class extends React.Component {  
    componentDidUpdate(prevProps) {  
      console.log('Props actuelles : ', this.props);  
      console.log('Props précédentes : ', prevProps);  
    }  
    render() {  
      // Enrobe le composant initial dans un conteneur, sans le modifier. Mieux !  
      return <WrappedComponent {...this.props} />;  
    }  
  }  
}
```



HOOKS

HOOKS

Les *Hooks* sont arrivés avec React 16.8

Ils nous permettent de bénéficier d'un état local et d'autres fonctionnalités de React sans avoir à écrire une classe

Par convention, tous les hooks ont leur nom qui commence par « use »

POURQUOI LES HOOKS

Réutiliser de la logique à état sans modifier la hiérarchie de vos composants

Utiliser davantage de fonctionnalités de React sans recourir aux classes

Découper un composant en petites fonctions basées sur les parties qui sont intrinsèquement liées (comme la configuration d'un abonnement ou le chargement de données)

GÉRER LE STATE LOCAL AVEC USESTATE

Afin de gérer le state local et permettre sa modification, React propose le hook **useState**

Comme tous les **hooks**, il s'agit d'une fonction à appeler dans le composant.

useState renvoie deux valeurs (dans un tableau) :

- La valeur actuelle de l'élément du state
- Une fonction permettant de mettre à jour cette valeur, déclenchant ainsi un nouveau rendu du composant
- **useState** prend en unique paramètre la valeur initiale du state.

```
let [networkError, setNetworkError] = useState(false);  
let [product, setProduct] = useState({});  
let [startEditing, setStartEditing] = useState(false);  
let [products, setProducts] = useState([]);
```

```
const showForm = (product) => {  
  setProduct(product);  
  setStartEditing(true);  
};
```


REQUÊTE AJAX AU DÉMARRAGE AVEC USEEFFECT

useEffect

- permet de gérer en partie le cycle de vie du composant, en remplaçant notamment la méthode **componentDidMount**.
- Le hook **useEffect** prend en paramètre deux arguments
 - une fonction, qui sera appelée chaque fois que le composant est rendu
 - un tableau contenant les éléments dont la mise à jour déclenche appel de fonction passée en premier argument
 - Si tableau est vide, fonction s'exécute seulement au premier rendu

```
useEffect(() => {  
  productService  
    .getProducts()  
    .then((data) => {  
      return data.json();  
    })  
    .then((res) => {  
      setProducts(res);  
    })  
    .catch((err) => {  
      setNetworkError(true);  
    });  
}, []); //[] -> exécute seulement au premier rendu  
car on veut charger de données une seule fois
```

EXEMPLE

Reprenons le composant **ProductDisplay** de notre application (slide 226 et 227), et initialisons une nouvelle version, sous forme de fonction cette fois-ci.

```
//ProductDisplay.js
import { useEffect, useState } from "react";
import productService from "../productService";

export default function ProductDisplay(props) {
  useEffect(() => {
    productService
      .getProducts()
      .then((data) => {
        return data.json();
      })
      .then((res) => {
        setProducts(res);
      })
      .catch((err) => {
        setNetworkError(true);
      });
  }, []); //[] -> exécute seulement au premier rendu
  let [networkError, setNetworkError] = useState(false);
  let [product, setProduct] = useState({});
  let [startEditing, setStartEditing] = useState(false);
  let [products, setProducts] = useState([]);
  const deleteProduct = (productId) => {
    productService.deleteProduct(productId);
  };
  const showForm = (product) => {
    setProduct(product);
    setStartEditing(true);
  };
}
```

```
//ProductDisplay.js (suite)
const cancel = () => {
  setProduct({});
  setStartEditing(true);
};
const save = (product) => {
  //ajout d'un nouveau produit
  if (!product.id) {
    productService.addProduct(product);
  } else {
    productService.updateProduct(product);
  }
};
if (networkError) {
  return <p>problème de réseau !</p>;
} else {
  return startEditing ? (
    <ProductForm
      product={product}
      cancelCallback={cancel}
      saveCallback={save}
    />
  ) : (
    <ProductTable
      products={products}
      showForm={showForm}
      deleteCallback={deleteProduct}
    />
  );
}
```



REDUX |

REDUX

Redux est une bibliothèque JavaScript qui implémente un conteneur d'état de l'architecture à base de flux.

Redux peut être décrit en trois principes fondamentaux:

1. Source unique de vérité (magasin unique)
2. L'état est en lecture seule (nécessite des actions pour signaler un changement)
3. Les modifications sont effectuées avec des fonctions pures (cela crée un nouvel état en fonction des actions)

POURQUOI REDUX ?

Gestion de données en utilisant de **state** dans un composant parent et de le passer aux enfants via les **props**, peut devenir laborieux à gérer et maintenir lorsque l'application stocke beaucoup de données, qui seront utilisées par beaucoup de composants.

C'est pour répondre à ce problème que Redux entre en jeu.

CONCEPTS DE BASE : LE STATE

L'élément le plus important de Redux

C'est en effet sa principale fonction : stocker l'état d'une application.

Les données qu'il stocke peuvent être de tout type

- objets, nombres, tableaux, fonctions, etc.

Le state peut être lu (par un composant React ou autre) ;

En revanche, il ne sera jamais modifié directement.

Nous allons devoir indiquer à Redux comment en générer la version suivante.

CONCEPTS DE BASE : LES ACTIONS

Afin de mettre à jour le state, la première étape sera de **déclencher** une action, par exemple au clic sur un bouton. On dira alors que l'action est **dispatchée**.

Une action doit être un objet, constitué d'un **type**, et éventuellement d'autres **données**.

L'idéal est de concevoir une action comme un **verbe**, associé à des **paramètres**.

- Par exemple, si le state contient un compteur (exemple classique d'introduction à Redux), on pourrait avoir les actions `incrémenter` et `définirValeur(valeur)`, la seconde étant une action qui prend un paramètre `valeur`.

CONCEPTS DE BASE : LE REDUCER

Le **reducer** est une fonction, prenant en paramètres un state et une action, et renvoyant un nouveau state.

- Par exemple, pour un compteur, à partir d'un state valant 5 et d'une action incrémenter, nous renverrions comme nouveau state la valeur 6.
- Nous ne mettons donc pas à jour le state, nous indiquons simplement à Redux une nouvelle version à prendre en compte.

Note importante

- le reducer doit être une fonction pure, au sens défini par la programmation fonctionnelle, c'est-à-dire notamment :
- Que son comportement et sa valeur de retour doivent être déterministes : pour un ensemble de paramètres donnés, on doit toujours avoir le même retour.
- Qu'elle ne doit pas avoir d'effet de bord, c'est-à-dire modifier un état qui lui est extérieur. On ne pourra donc pas mettre à jour d'autres variables, dispatcher une autre action, ou encore déclencher un traitement asynchrone à base de promesses.

Cela peut paraître contraignant au premier abord, mais nous verrons comment Redux nous permet de nous en sortir tout de même de manière élégante. De plus ces contraintes sont à la base de ce qui rend une application Redux plus facile à maintenir.

CONCEPTS DE BASE : LE STORE

Le store n'est autre que **l'objet** unifiant le state, le **reducer** et les actions.

Nous l'initialisons au démarrage de l'application, en lui fournissant un **reducer** et un **state** initial. Puis nous pourrons

- lire le state courant
- dispatcher des actions
- souscrire aux changements du state (c'est-à-dire appeler une fonction dès que le state est mis à jour)

Notez que pour un store nous n'aurons toujours qu'un seul reducer, mais qu'il est aisé comme nous le verrons de combiner des **reducer** pour n'en faire qu'un seul ; ce ne sera donc jamais une contrainte.

REACT-REDUX

Une bibliothèque JavaScript permettant de connecter facilement notre magasin Redux à nos composants React

Cette bibliothèque permet de définir pour certains composants

- à quels éléments du state nous souhaitons avoir accès (pour les afficher par exemple) ;
- quelles actions nous souhaitons être mesure de dispatcher.

INSTALLATION DE DÉPENDANCES

Exécutez la commande suivante:

yarn add redux react-redux

ou

npm install redux react-redux

EXEMPLE

Il s'agira d'un simple compteur, avec un bouton permettant de l'incrémenter

CRÉATION DE COMPOSANT « COMPTEUR »

On attend que deux propriétés lui soient passées :

- la valeur du compteur **counter** et une fonction **increment** permettant d'incrémenter le compteur

< > ↺ <https://496ck.csb.app/>

Compteur:

```
// Counter.js
const Counter = (props) => (
  <p>
    Compteur:
    <span>{props.counter}</span>
    <button onClick={() => props.increment()}>+</button>
  </p>
)

export default Counter

// App.js
import Counter from "../Counter";

export default function App() {
  return <Counter />;
}
```

CRÉATION DE STORE : STATE INITIAL

Commençons par définir le **state initial** de notre application dans le fichier **store.js**.

Nous stockerons uniquement un **compteur** pour le moment, ce pourrait donc être une simple valeur numérique: 0.

Mais au cas où nous souhaiterions par la suite stocker d'autres informations, il est plus judicieux d'utiliser un objet:

```
// Initial state  
const initialState = { counter: 0 }
```


CRÉATION DE STORE : ACTIONS ET ACTIONS CREATORS

Définissons le type « increment » associé à l'action permettant d'incrémenter le compteur :

```
// Action types  
const INCREMENT = 'increment'
```

Afin de faciliter la création d'une action de ce type, définissons également une fonction **increment**:

```
// Action creators  
const increment = () => ({  
  type: INCREMENT  
})
```

CRÉATION DE STORE : REDUCER

Pour avoir les éléments nécessaires pour créer notre store, il ne nous manque que le **reducer**.

Il ne réagit qu'à un seul type d'action, **INCREMENT**, et dans ce cas renvoie comme compteur la valeur de l'attribut **counter** du state, incrémentée de 1 :

```
// Reducer
const reducer = (state = initialState, action) => {
  switch (action.type) {
    case INCREMENT:
      return { ...state, counter: state.counter + 1 }
    default:
      return state
  }
}
```

CRÉATION DE STORE : REDUCER (SUITE)

Quelques remarques sur les bonnes pratiques pour écrire un **reducer**:

- Le premier paramètre (le state courant), n'est pas défini au premier appel du reducer. Il est donc pratique d'utiliser notre initialState en valeur par défaut afin d'initialiser le state à ce moment.
- Structurer un **reducer** avec un bloc switch est la manière de faire la plus simple et la plus courante, mais n'a rien d'obligatoire.
- Utiliser la syntaxe `{ ...state, nouvellesValeurs }` permet d'une part de s'assurer qu'on ne modifie pas le state actuel, et d'autre part de ne mettre à jour que les attributs qui nous intéressent ici. En l'occurrence nous n'en avons qu'un seul (counter), mais la plupart du temps il y en aura d'autres, prenons donc dès maintenant de bonnes habitudes.
- Si l'action ne correspond à aucun type connu, nous devons tout de même renvoyer un state, nous renvoyons donc le state courant sans modification (cas default du switch). En effet Redux commence par dispatcher des actions à lui au démarrage.
- Par ailleurs, remarquez que cette fonction est pure, comme définie dans la section précédente: nous ne modifions pas de valeur extérieure à la fonction (notamment le state), et elle n'entraîne pas d'effet de bord.

CRÉATION DE STORE

À présent, nous pouvons créer notre store dans le fichier **index.js**

Commençons par importer(en haut du fichier) la fonction **createStore** de **Redux**:

Puis créons ce store en lui donnant en paramètre le **reducer**:

```
// Store  
const store = createStore(reducer)
```

CONNEXION REDUX - REACT

Connectons le magasin Redux à notre application Redux

Commençons par importer(en haut du fichier) le composant **Provider** de **react-redux**

On doit envelopper notre application dans un **Provider**, qui est un composant qui passe par votre magasin pour être utilisé par les composants enfants

L'avantage de Provider, c'est qu'il n'est nécessaire de l'utiliser qu'une seule fois

- tous les composants inclus à l'intérieur, y compris les composants appelés par d'autres composants, auront accès au store via l'utilisation de **connect**.

```
//index.js
import ReactDOM from "react-dom";
import { Provider } from "react-redux";
import { createStore } from "redux";
import App from "./App";
import { reducer } from "./store";

const store = createStore(reducer);
const rootElement = document.getElementById("root");
ReactDOM.render(
  <Provider store={store}>
    <App />
  </Provider>,
  rootElement
);
```

MAPSTATETOPROPS

mapStateToProps

- Fonction permettant de récupérer les éléments du state qui nous intéressent.
- Par convention, cette fonction est appelée mapStateToProps
- Comme son nom l'indique, cette fonction va nous permettre d'indiquer à React - Redux quel mapping(correspondances) nous souhaitons effectuer entre les attributs du state et les propriétés du composant.

```
const mapStateToProps = state => ({  
  counter: state.counter  
})
```

MAPDISPATCHTOPROPS

mapDispatchToProps

- Fonction permettant d'effectuer le mapping entre les actions que l'on veut dispatcher et les propriétés du composant;
- Par convention on l'appelle mapDispatchToProps

```
const mapDispatchToProps = dispatch => ({  
  increment: () => dispatch(increment())  
})
```

ou

```
const mapDispatchToProps = {  
  increment: () => actions.increment()  
};
```

CONNECT

connect

- Fonction permettant d'indiquer à react-redux comment utiliser les fonctions `mapStateToProps` et `mapDispatchToProps`
- elle renvoie une nouvelle fonction, à laquelle on passe en paramètre un composant pour obtenir en retour un nouveau composant (HOC), augmenté du lien avec Redux

```
const CounterWithRedux = connect(  
  mapStateToProps,  
  mapDispatchToProps  
) (Counter)
```



```
// Counter.js
import React from "react";
import { connect } from "react-redux";
import { actions } from "../store";

const Counter = ({ counter, increment }) => (
  <p>
    Compteur:
    <span>{counter}</span>
    <button onClick={() => increment()}>+</button>
  </p>
);

const mapStateToProps = (state) => ({
  counter: state.counter
});

const mapDispatchToProps = {
  increment: () => actions.increment()
};

export default connect(mapStateToProps,
mapDispatchToProps)(Counter);
```

```
// store.js
import { createStore } from 'redux'

const initialState = { counter: 0 }

const actionTypes = {
  INCREMENT: 'increment'
}

export const actions = {
  increment: () => ({
    type: actionTypes.INCREMENT
  })
}

const reducer = (state = initialState, action) => {
  switch (action.type) {
    case actionTypes.INCREMENT:
      return { ...state, counter: state.counter + 1 }
    default:
      return state
  }
}

export default createStore(reducer)
```

```
//index.js
import ReactDOM from "react-dom";
import { Provider } from "react-redux";
import App from "./App";
import store from "./store";

const rootElement = document.getElementById("root");
ReactDOM.render(
  <Provider store={store}>
    <App />
  </Provider>,
  rootElement
);
```



REDUX-SAGA |

REDUX-SAGA

Redux-Saga est une bibliothèque JavaScript permettant de gérer les effets de bord d'une application avec Redux.

Les opérations « effets de bord » (ou effets pour faire court) sont les opérations qui peuvent affecter d'autres composants et ne peuvent être réalisées pendant l'affichage

- Ex: chargement de données distantes

PRINCIPES ET DIFFICULTÉS DE REDUX-SAGA

Principes:

- Le principe sera de définir des sagas qui seront déclenchées sur des actions données et qui auront la possibilité d'effectuer des traitements, comme lire le state, appeler des fonctions asynchrones, ou déclencher de nouvelles actions.
- Tout cela s'intégrera dans notre store Redux à l'aide d'un middleware.

Difficultés

- L'une des principales difficultés à l'utilisation de Redux-Saga est qu'il repose sur une fonctionnalité de JavaScript assez peu utilisée au quotidien : les générateurs ou fonctions génératrices.
- Mais c'est aussi ce qui fait sa force, car les générateurs permettent d'appliquer les principes à la base de Redux-Saga, c'est-à-dire déclencher des effets.

ITÉRATEURS

Un **itérateur** est un objet sachant comment accéder aux éléments d'une collection un par un et qui connaît leur position dans la collection.

En JavaScript, un **itérateur** expose une méthode **next()** qui retourne l'élément suivant dans la séquence.

- Cette méthode renvoie un objet possédant deux propriétés : **done** et **value**.

Un itérateur est "terminé" lorsque l'appel à la méthode **next()** renvoie un objet dont la propriété **done** vaut true.

Une fois créé, un **itérateur** peut être utilisé explicitement en appelant sa méthode **next()**, ou implicitement en utilisant la boucle **for...in**.

EXEMPLE

Voici un exemple d'une fonction créant un **itérateur** qui parcourt l'intervalle défini par ses arguments (depuis **debut** (inclus) jusqu'à **end** (exclus) et avec **pas** comme incrément. La valeur finale qui est renvoyée correspond à la taille de la séquence créée.


```
function creerIterateurIntervalle(debut = 0, fin = Infinity, pas = 1) {  
  let prochainIndex = debut;  
  let nbIterations = 0;  
  
  const rangeIterator = {  
    next: function() {  
      let resultat;  
      if (prochainIndex < fin) {  
        resultat = { value: prochainIndex, done: false };  
        prochainIndex += pas;  
        nbIterations++;  
        return resultat;  
      }  
      return { value: nbIterations, done: true }  
    }  
  };  
  return rangeIterator;  
}
```

```
let it = creerIterateurIntervalle(1, 10, 2);
```

```
let resultat = it.next();  
while (!resultat.done) {  
  console.log(resultat.value); // 1 3 5 7 9  
  resultat = it.next();  
}
```

```
console.log("La séquence parcourue contenait ", result.value, " éléments.");
```

GÉNÉRATEUR

Un générateur est un type de fonction spécial qui fonctionne comme une fabrique (factory) **d'itérateurs**.

Une fonction devient un générateur lorsqu'elle contient une ou plusieurs expressions **yield** et qu'elle utilise la syntaxe **function***.

Exemple simple

```
function* idMaker(){  
  var index = 0;  
  while(true)  
    yield index++;  
}
```

```
var gen = idMaker();
```

```
console.log(gen.next().value); // 0  
console.log(gen.next().value); // 1  
console.log(gen.next().value); // 2
```

EXEMPLE COMPLEX

Imaginez que l'on souhaite définir un moteur d'exécution, permettant d'exécuter des petits programmes pour effectuer des opérations, demander des informations à l'utilisateur et lui afficher des résultats.

Par exemple, le programme suivant, en pseudo-code :

```
name = prompt "What's your name?"  
greetings = "Hello " + name + "!"  
show greetings
```

- Nous souhaitons que notre programme puisse être écrit en JavaScript, mais les fonctions du langage de haut niveau **prompt** et **show** ne sont pas connues.
- Il se peut qu'elles affichent une nouvelle fenêtre à l'utilisateur, ou qu'elles fassent appel à une API, ou encore qu'elles aillent lire des informations dans des fichiers.
- Elles ne sont peut-être même pas synchrones.
- En bref, il s'agit d'effets de bord du programme.

FONCTIONS UTILITAIRES ET PROGRAMME PRINCIPAL

Afin de gérer ces effets de bord, commençons par définir deux fonctions utilitaires (en JavaScript cette fois) :

```
const prompt = name => ({ type: 'prompt', name })
const print = message => ({ type: 'print', message })
```

Notez que ces fonctions ne « font » rien, si ce n'est nous aider à définir les effets de bord possibles de notre moteur d'exécution.

Mais en utilisant les générateurs de JavaScript, on peut à présent écrire notre programme ainsi :

```
function* myProgram() {
  const name = yield prompt('What's your name?')
  const greetings = `Hello ${name}!`
  yield print(greetings)
}
```

EXECUTION MANUELLE DU PROGRAMME

- `it = myProgram()`
- `> it.next()`
- `Object { value: {type: "prompt", name: "What's your name?"}, done: false }`
- `> it.next("John") Object { value: {type: "print", message: "Hello John!"}, done: false }`
- `> it.next() Object {value: undefined, done: true}`

Ici, c'est nous qui avons simulé le moteur d'exécution du programme, en appelant à plusieurs reprises la méthode `next` de l'itérateur :

- Tout d'abord, l'itérateur nous a renvoyé un effet de type « `prompt` ». Nous savons donc que l'on doit demander à l'utilisateur une information (avec le message « `What's your name?` »). Pour l'itération suivante, nous envoyons donc l'information demandée : « `John` ».
- À l'itération suivante, le programme reçoit la valeur demandée, il s'agit du retour de l'instruction `yield`. Il continue donc son exécution, et déclenche l'effet suivant, de type « `prompt` », avec le message « `Hello John!` ». Nous affichons (virtuellement) ce message à l'utilisateur), et appelons l'itération suivante.
- Le programme est terminé, et met donc fin à l'itérateur.

MOTEUR D'EXÉCUTION

```
function executeProgram(program) {  
  const it = program()  
  let res = it.next()  
  while (!res.done) {  
    const effect = res.value  
    switch (effect.type) {  
      case 'prompt':  
        const input = window.prompt(effect.desc)  
        res = it.next(input)  
        break  
      case 'print':  
        window.alert(effect.message)  
        res = it.next()  
        break  
      default:  
        throw new Error(`Invalid effect type: ${effect.type}`)  
    }  
  }  
}
```

APPORTS DES FONCTIONS GÉNÉRATRICES ET LE PRINCIPE DES EFFETS

Notre programme est défini par une fonction pure, c'est-à-dire que pour des paramètres donnés, l'exécution et le résultat retourné seront toujours les mêmes.

Notre programme est beaucoup plus facilement testable :

- en environnement réel, il sera interprété par un moteur d'exécution,
- mais en test il peut l'être par un autre, qui par exemple simulerait les appels à une API si cela est nécessaire.

La syntaxe du programme est somme toute relativement simple, une fois que l'on appréhende les fonction*et les yield.

LES EFFETS DE REDUX-SAGA

Le but principal de Redux-Saga est de proposer une manière de gérer des effets de bord dans une application utilisant Redux.

Pour cela, de la même manière que dans notre exemple précédent où nous déclenchions des effets pour demander une saisie de l'utilisateur ou lui afficher une valeur, nous allons déclencher des effets de Redux-Saga, cette fois-ci pour interagir avec Redux ou déclencher d'autres effets de bord.

Les effets qu'il est possible de déclencher dans Redux-Saga sont de plusieurs types.

call

- permet d'appeler une fonction, éventuellement asynchrone, et d'en récupérer le résultat.
- Typiquement, il s'agit d'une fonction ayant justement des effets de bord, comme une requête à une API.

DES EFFETS PERMETTENT DE MANIPULER LE STORE DE REDUX

select

- lit une valeur dans le store grâce à un sélecteur, de manière semblable au hook **useSelector**.

put

- dispatche une action.

take

- attend l'arrivée d'une action d'un type donné.

DES EFFETS PERMETTENT DE MANIPULER L'EXÉCUTION DE LA SAGA ELLE-MÊME

fork

- pour dupliquer l'exécution de la saga courante,

delay

- pour attendre une certaine durée

EXEMPLE

Nous allons réaliser une petite application permettant d'interroger une API de recherche d'adresse proposée par l'administration française : adresse.data.gouv.fr. Celle-ci permet à partir d'une requête (un texte libre) de récupérer une liste de résultats correspondant à des adresses réelles, avec des informations comme leur géolocalisation exacte.

Les cas d'utilisations possibles sont nombreux

- autocomplétion sur un champ de saisie d'adresse,
- recherche d'adresse sur une carte, etc.

L'avantage pour notre exemple est qu'à ce jour aucune clé n'est nécessaire pour l'utiliser l'API, elle est donc très simple à appeler (pas d'inscription nécessaire notamment).

EXEMPLE (SUITE)

Notre application présentera un champ de saisie pour la requête, un bouton permettant de lancer la recherche, et une liste de résultats. Elle devra gérer un affichage spécifique pour les cas suivants :

- la recherche n'a pas encore été lancée
- la recherche est en cours
- une erreur s'est produite
- aucun résultat n'est renvoyé
- au moins un résultat est renvoyé



120 rue de la paix **Go!**

Results

- 📍 120 Rue de la Paix 50100 Cherbourg-en-Cote...
- 📍 120 Rue de la Paix 62200 Boulogne-sur-Mer
- 📍 120 Rue de la Paix 94170 Le Perreux-sur-Mar...
- 📍 120 Rue de la Paix 78500 Sartrouville
- 📍 120 Rue de la Paix 59110 La Madeleine

INSTALLATION DES DÉPENDANCES

Exécutez la commande suivante:

yarn add redux react-redux redux-saga

Ou

npm install redux react-redux redux-saga

LES ACTIONS

```
// src/services/search/actions.js
export const updateQuery = query =>
  ({ type: 'updateQuery', payload: query })
export const search = () => ({ type: 'search' })
export const searchSuccess = results => ({
  type: 'searchSuccess',
  payload: results,
})
export const searchFailure = error => ({
  type: 'searchFailure',
  payload: error,
})
```

REDUCER

```
// src/services/search/reducer.js
const initialState = {
  query: '',
  isPending: false,
  hasError: false,
  results: undefined,
}

export const reducer = (state = initialState, action) => {
  switch (action.type) {
    case 'updateQuery':
      return { ...state, query: action.payload }
    case 'search':
      return { ...state, isPending: true, hasError: false }
    case 'searchSuccess':
      return {
        ...state, isPending: false,
        results: action.payload
      }
    case 'searchFailure':
      return { ...state, isPending: false, hasError: true }
    default:
      return state
  }
}
```


SELECTEURS

Afin de faciliter la lecture depuis le state dans les composants et dans la saga, nous fournissons également des sélecteurs.

Il s'agit de fonctions prenant en paramètre le state, et renvoyant la valeur demandée

```
// src/services/search/index.js
export const namespace = 'search'

// src/services/search/selectors.js
import { namespace } from '.'

export const selectQuery = state =>
  state[namespace].query
export const selectSearchIsPending = state =>
  state[namespace].isPending
export const selectSearchHasError = state =>
  state[namespace].hasError
export const selectSearchResults = state =>
  state[namespace].results
```

SAGA

Nous souhaitons réagir aux actions de type « search », déclenchées lorsque l'utilisateur clique sur le bouton « Go ».

À ce moment, nous devons récupérer la requête dans le state(en utilisant le sélecteur selectQuery),

puis appeler la fonction de recherche searchAddresses avec la requête en paramètre,

et enfin dispatcher une action searchSuccess avec les résultats.

Dans le cas où une erreur s'est produite(par exemple, pas de connexion au réseau), on dispatchera une action searchFailure.

Pour effectuer la recherche chaque fois qu'un caractère est tapé, mais en ajoutant un délai de sorte qu'il n'y ait pas trop de requêtes inutiles à l'API, on utilise **throttle**

throttle fonctionne exactement comme **takeEvery**, mais en ajoutant le délai avant de déclencher la saga

```
import { takeEvery, select, call, put, throttle }
  from '@redux-saga/core/effects'
import { searchAddresses } from './api'
import { selectQuery } from './selectors'
import { searchSuccess, searchFailure, search
  } from './actions'
function* searchSaga() {
  const query = yield select(selectQuery)
  try {
    const results = yield call(searchAddresses, query)
    yield put(searchSuccess(results))
  } catch (err) {
    yield put(searchFailure(err))
  }
}
function* updateQuerySaga() {
  const query = yield select(selectQuery)
  if (query.length > 3) {
    yield put(search())
  }
}
export function* rootSaga() {
  yield takeEvery('search', searchSaga)
  yield throttle(1000, 'updateQuery', updateQuerySaga)
}
```

API

```
// src/services/search/api.js
export const searchAddresses = async query => {
  const res = await fetch(
    'https://api-adresse.data.gouv.fr/search/?q='
    + encodeURIComponent(query),
  )
  const { features } = await res.json()
  return features
}
```

```
// src/store.js
import createSagaMiddleware from '@redux-saga/core'
import { fork } from '@redux-saga/core/effects'
import { createStore, applyMiddleware, combineReducers } from 'redux'

// Search service
import { namespace as searchNamespace } from '../services/search'
import { rootSaga as searchSaga } from '../services/search/saga'
import { reducer as searchReducer } from '../services/search/reducer'

const rootReducer = combineReducers({
  [searchNamespace]: searchReducer,
})

function* rootSaga() {
  yield fork(searchSaga)
}

const sagaMiddleware = createSagaMiddleware()
export const store = createStore(
  rootReducer,
  applyMiddleware(sagaMiddleware)
)
sagaMiddleware.run(rootSaga)
```

```
// src/components/App.js
import React from 'react'
import './App.css'
import { connect } from 'react-redux'
import {
  selectQuery,
  selectSearchIsPending,
  selectSearchHasError,
  selectSearchResults,
} from '../services/search/selectors'
import { updateQuery, search } from '../services/search/actions'

const App = ({
  query, isPending, hasError,
  results, updateQuery, search
}) => {
  const handleFormSubmit = event => {
    event.preventDefault()
    search()
  }
  const handleQueryChange = event =>
    updateQuery(event.target.value)
```

```
// src/components/App.js (suite)
```



```
return (  
  <div className="app">  
    <form onSubmit={handleFormSubmit}>  
      <input  
        type="text"  
        value={query}  
        onChange={handleQueryChange}  
        placeholder="Enter an address..."  
        required  
      />  
      <button type="submit">Go!</button>  
    </form>  
    {results !== undefined && (  
      <>  
        <h2>Results</h2>  
        {isPending && <p className="info">Loading...</p>}  
        {hasError &&  
          <p className="info error">An error occurred.</p>}  
        {results.length > 0 ? (  
          <ul>  
            {results.map(result => (  
              <li key={result.properties.id}>  
                {result.properties.label}  
              </li>  
            ))}  
          </ul>  
        ) : (  
          <p className="info">Search returned no result.</p>  
        )}  
      </>  
    )}  
  </div>  
)  
}
```

```
// src/components/App.js (suite)
const mapStateToProps = state => ({
  query: selectQuery(state),
  isPending: selectSearchIsPending(state),
  hasError: selectSearchHasError(state),
  results: selectSearchResults(state),
})

const mapDispatchToProps = {
  updateQuery,
  search,
}

export default connect(
  mapStateToProps,
  mapDispatchToProps,
)(App)
```



REACT ROUTER

REACT ROUTER

React Router est une bibliothèque de routage côté client et côté serveur complète pour React

React Router s'exécute partout où React s'exécute

- sur le web, sur le serveur avec node.js et sur React Native.

INSTALLATION

Pour utiliser React Router sur le web, Installez la librairie **react-router-dom** en exécutant la commande suivante:

npm add react-router-dom

ou

yarn add react-router-dom

CONNEXION DE L'APPLI À URL DU NAVIGATEUR

Le composant **BrowserRouter** permet de définir la portion de l'application concernée par le routage

Il est tout à fait possible d'inclure ce composant dans un élément précis de l'application (c'est-à-dire sans englober le tout), mais dans ce cas ce qui est à l'extérieur n'aura pas accès au contexte (c'est-à-dire l'endroit où se trouve l'utilisateur) et ne pourra pas non plus le mettre à jour (se rendre à un endroit spécifique)

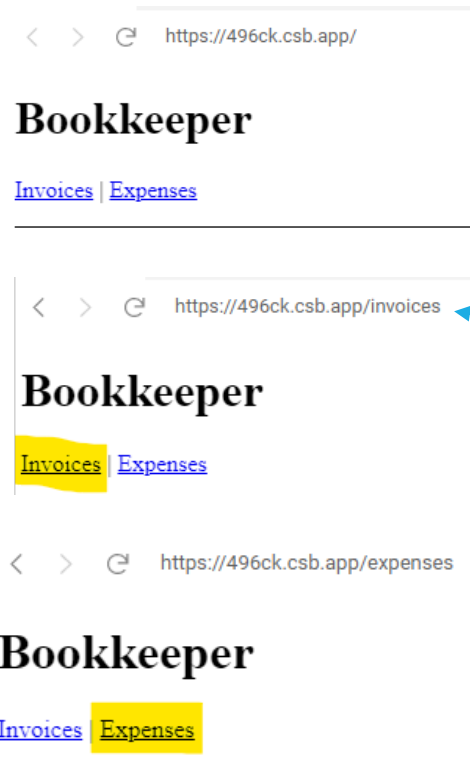
La plupart du temps, c'est toute l'application qui sera englobée dans un Router

```
//index.js
import ReactDOM from "react-dom";
import { BrowserRouter } from "react-router-dom";
import App from "./App";

ReactDOM.render(
  <BrowserRouter>
    <App />
  </BrowserRouter>,
  document.getElementById("root");
);
```

LINK

Link prend en paramètre l'URL où l'on souhaite se rendre au moment du clic



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

export default function App() {
  return (
    <div>
      <h1>Bookkeeper</h1>
      <nav
        style={{
          borderBottom: "solid 1px",
          paddingBottom: "1rem"
        }}
      >
        <Link to="/invoices">Invoices</Link> |{" "}
        <Link to="/expenses">Expenses</Link>
      </nav>
    </div>
  );
}
```

CRÉATION DES COMPOSANTS

On crée des composants **Expenses** et **Invoices**

```
export default function Expenses() {  
  return (  
    <main style={{ padding: "1rem 0" }}>  
      <h2>Expenses</h2>  
    </main>  
  );  
}
```

```
export default function Invoices() {  
  return (  
    <main style={{ padding: "1rem 0" }}>  
      <h2>Invoices</h2>  
    </main>  
  );  
}
```

DÉFINITION DES ROUTES

On définit des routes suivantes:

/ : affiche le composant **<App>**

/invoices : affiche le composant
<Invoices>

< > ↺ <https://496ck.csb.app/invoices>

Invoices

/expenses : affiche le composant
<Expenses>

< > ↺ <https://496ck.csb.app/expenses>

Expenses

```
//index.js
import ReactDOM from "react-dom";
import {
  BrowserRouter,
  Routes,
  Route
} from "react-router-dom";
import App from "./App";
import Expenses from "./expenses";
import Invoices from "./invoices";

ReactDOM.render(
  <BrowserRouter>
    <Routes>
      <Route path="/" element={<App />} />
      <Route path="expenses" element={<Expenses />} />
      <Route path="invoices" element={<Invoices />} />
    </Routes>
  </BrowserRouter>,
  document.getElementById("root")
);
```

ROUTES IMBRIQUÉES

```
<Route path="/" element={<App />}>  
  <Route path="expenses" element={<Expenses />} />  
  <Route path="invoices" element={<Invoices />} />  
</Route>
```

Lorsque les routes ont des enfants, elles font deux choses :

- Il imbrique les URL ("/" + "expenses" et "/" + "invoices")
- Il imbriquera les composants de l'interface utilisateur pour la mise en page partagée lorsque la route enfant correspond

ROUTES IMBRIQUÉES

Pour que l'imbrication fonctionne, ajouter Outlet dans la route « parent »

La route parent (App.js) persiste pendant les <Outlet>échanges entre les deux routes enfants (<Invoices>et <Expenses>) !

< > ↺ https://496ck.csb.app/invoices

Bookkeeper

[Invoices](#) | [Expenses](#)

Invoices

< > ↺ https://496ck.csb.app/expenses

Bookkeeper

[Invoices](#) | [Expenses](#)

Expenses

```
import { Outlet, Link } from "react-router-dom";

export default function App() {
  return (
    <div>
      <h1>Bookkeeper</h1>
      <nav
        style={{
          borderBottom: "solid 1px",
          paddingBottom: "1rem"
        }}
      >
        <Link to="/invoices">Invoices</Link> |{" "}
        <Link to="/expenses">Expenses</Link>
      </nav>
      <Outlet />
    </div>
  );
}
```


CRÉATION D'UNE LISTE DES « INVOICES »

Link prend en paramètre l'URL où l'on souhaite se rendre au moment du clic

Si vous cliquez sur ces liens, la page devient vierge !

C'est parce qu'aucune des routes que nous avons définies ne correspond à une URL comme celles vers lesquelles nous sommes liés : « `/invoices/1995` »

< > ↺ <https://496ck.csb.app/invoices/1995>

```
export default function Invoices() {
  let invoices = getInvoices();
  return (
    <div style={{ display: "flex" }}>
      <nav
        style={{
          borderRight: "solid 1px",
          padding: "1rem"
        }}
      >
        {invoices.map(invoice => (
          <Link
            style={{ display: "block", margin: "1rem 0" }}
            to={`/${invoices}/${invoice.number}`}
            key={invoice.number}
          >
            {invoice.name}
          </Link>
        ))}
      </nav>
    </div>
  );
}
```

```
//data.js
let invoices = [
  {
    name: "Santa Monica",
    number: 1995,
    amount: "$10,800",
    due: "12/05/1995"
  },
  {
    name: "Stankonia",
    number: 2000,
    amount: "$8,000",
    due: "10/31/2000"
  },
  {
    name: "Ocean Avenue",
    number: 2003,
    amount: "$9,500",
    due: "07/22/2003"
  },
];

export function getInvoices() {
  return invoices;
}
```

ROUTE « NO MATCH »

Il est recommandé de toujours gérer le cas de « no match »

Le "*" a ici une signification particulière. Il ne correspondra que lorsqu'aucun autre Route ne le fera.

< > ↻ <https://496ck.csb.app/invoices/1995>

Bookkeeper

[Invoices](#) | [Expenses](#)

There's nothing here!

```
<Routes>
  <Route path="/" element={<App />}>
    <Route path="expenses" element={<Expenses />} />
    <Route path="invoices" element={<Invoices />} />
    <Route
      path="*"
      element={
        <main style={{ padding: "1rem" }}>
          <p>There's nothing here!</p>
        </main>
      }
    />
  </Route>
</Routes>
```

PARAMÈTRE D'URL

La partie du chemin « :invoiceId » est un "param d'URL", ce qui signifie qu'elle peut correspondre à n'importe quelle valeur tant que le modèle est le même.

Le `<Route>` ajoute une deuxième couche d'imbrication de routes lorsqu'il correspond à

- `<App><Invoices><Invoice /></Invoices></App>`.

Étant donné que le `<Route>` est imbriqué, l'interface utilisateur sera également imbriquée.

```
export default function Invoice() {
  return <h2>Invoice #???</h2>;
}
```

Nous aimerions afficher le numéro de facture au lieu de "???"

```
<Routes>
  <Route path="/" element={<App />}>
    <Route path="expenses" element={<Expenses />} />
    <Route path="invoices" element={<Invoices />}>
      <Route path=":invoiceId" element={<Invoice />} />
    </Route>
  <Route
    path="*"
    element={
      <main style={{ padding: "1rem" }}>
        <p>There's nothing here!</p>
      </main>
    }
  />
</Route>
</Routes>
```

PARAMÈTRE D'URL

Étant donné que le `<Route>` est imbriqué, l'interface utilisateur sera également imbriquée

Nous devons ajouter **Outlet** à la Route de mise en page parent

```
export default function Invoices() {
  let invoices = getInvoices();
  return (
    <div style={{ display: "flex" }}>
      <nav
        style={{
          borderRight: "solid 1px",
          padding: "1rem"
        }}
      >
        {invoices.map(invoice => (
          <Link
            style={{ display: "block", margin: "1rem 0" }}
            to={`/invoices/${invoice.number}`}
            key={invoice.number}
          >
            {invoice.name}
          </Link>
        ))}
      </nav>
      <Outlet />
    </div>
  );
}
```



LECTURE DE PARAMS D'URL

La clé du paramètre sur l'objet **params** est la même que le segment dynamique dans le chemin de la route

- :invoiceld -> params.invoiceld

```
import { useParams } from "react-router-dom";  
  
export default function Invoice() {  
  let params = useParams();  
  return <h2>Invoice: {params.invoiceId}</h2>;  
}
```

RECHERCHER UN « INVOICE »

Ouvrons **src/data.js** et ajoutons une nouvelle fonction pour rechercher des factures par leur numéro

```
// ...  
  
export function getInvoices() {  
  return invoices;  
}  
  
export function getInvoice(number) {  
  return invoices.find(  
    invoice => invoice.number === number  
  );  
}
```

RECHERCHER UN « INVOICE »

Dans **invoice.js** nous utilisons le paramètre pour rechercher une facture et afficher plus d'informations

< > ↺ <https://496ck.csb.app/invoices/1995>

Bookkeeper

[Invoices](#) | [Expenses](#)

[Santa Monica](#)

[Stankonia](#)

[Ocean Avenue](#)

[Tubthumper](#)

[Wide Open Spaces](#)

Total Due: \$10,800

Santa Monica: 1995

Due Date: 12/05/1995

```
import { useParams } from "react-router-dom";
import { getInvoice } from "../data";

export default function Invoice() {
  let params = useParams();
  let invoice =
    getInvoice(parseInt(params.invoiceId));
  return (
    <main style={{ padding: "1rem" }}>
      <h2>Total Due: {invoice.amount}</h2>
      <p>
        {invoice.name}: {invoice.number}
      </p>
      <p>Due Date: {invoice.due}</p>
    </main>
  );
}
```


ROUTE « INDEX »

Problème

- Lorsque l'on clique sur le lien « Invoices » dans la navigation globale de notre application, la zone de contenu principale devient vide !

Solution

- Nous pouvons résoudre ce problème avec une route "index".

```
< Routes >
  <Route path="/" element={<App />}>
    <Route path="expenses" element={<Expenses />} />
    <Route path="invoices" element={<Invoices />}>
      <Route
        index
        element={
          <main style={{ padding: "1rem" }}>
            <p>Select an invoice</p>
          </main>
        }
      />
    <Route path=":invoiceId" element={<Invoice />} />
  </Route>
  <Route
    path="*"
    element={
      <main style={{ padding: "1rem" }}>
        <p>There's nothing here!</p>
      </main>
    }
  />
</Route>
</Routes >
```

ROUTE « INDEX »

Les routes d'index sont rendues dans la sortie des routes parentes au niveau du chemin de la route parente

Les routes d'index correspondent lorsqu'une route parent correspond mais qu'aucun des autres enfants ne correspond

Les routes d'index sont la route enfant par défaut pour une route parent

Les routes d'index sont rendues lorsque l'utilisateur n'a pas encore cliqué sur l'un des éléments d'une liste de navigation

LIENS ACTIFS

Lien actif

- URL de lien correspondant à l'URL du navigateur

Ajoutons liens actif à notre liste de « invoices » en échangeant **Link** contre **NavLink**

- Nous avons changé le **style** d'un simple objet en une fonction qui renvoie un objet.
- Nous avons changé la couleur de notre lien en regardant la valeur **isActive** qui est passée par **NavLink** à notre fonction de style

```
import { NavLink, Outlet } from "react-router-dom";
import { getInvoices } from "../data";
```

```
export default function Invoices() {
  let invoices = getInvoices();
  return (
    <div style={{ display: "flex" }}>
      <nav
        style={{
          borderRight: "solid 1px",
          padding: "1rem"
        }}
      >
        {invoices.map(invoice => (
          <NavLink
            style={({ isActive }) => {
              return {
                display: "block",
                margin: "1rem 0",
                color: isActive ? "red" : ""
              };
            }}
            to={`/invoices/${invoice.number}`}
            key={invoice.number}
          >
            {invoice.name}
          </NavLink>
        ))}
      </nav>
      <Outlet />
    </div>
  );
}
```

< > ↺ https://496ck.csb.app/invoices/2000

Bookkeeper

[Invoices](#) | [Expenses](#)

[Santa Monica](#)

[Stankonia](#)

[Ocean Avenue](#)

[Tubthumper](#)

[Wide Open Spaces](#)

Total Due: \$8,000

Stankonia: 2000

Due Date: 10/31/2000

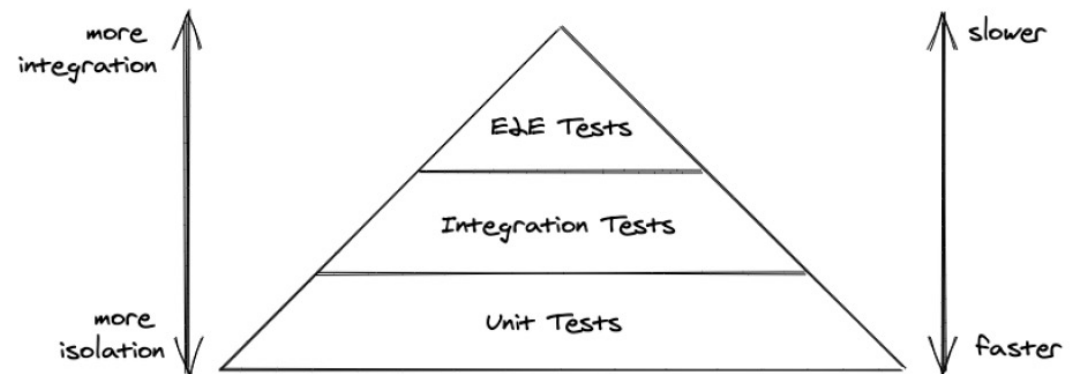


TESTS

TESTS

Il existe plusieurs façons de tester des composants React, lesquelles se divisent au final en deux grandes catégories :

- **Effectuer le rendu d'arborescences de composants** dans un environnement de test simplifié, et vérifier la sortie.
- **Exécuter une appli complète** dans un environnement navigateur réaliste (on parle alors de tests « de bout en bout ») (*end-to-end (e2e)*).



OUTILS

Jest

- est un framework de test JavaScript qui vous permet d'accéder au DOM via jsdom.
- Même si jsdom ne simule que partiellement le fonctionnement d'un navigateur, il est souvent suffisant pour tester vos composants React.
- Jest combine une excellente vitesse d'itération avec de puissantes fonctionnalités telles que l'isolation des modules et des horloges, afin que vous puissiez garder un contrôle fin sur la façon dont votre code s'exécute.

React Testing Library

- fournit un ensemble de fonctions utilitaires pour tester des composants React sans dépendre de leurs détails d'implémentation.
- Cette approche facilite le changement de conception interne et vous aiguille vers de meilleures pratiques en termes d'accessibilité.
- Même s'il ne fournit pas de moyen pour réaliser le rendu « superficiel » d'un composant (sans ses enfants), on peut y arriver avec un framework tel que Jest et ses mécanismes d'isolation.

MISE EN PLACE / NETTOYAGE

Pour chaque test, nous voulons habituellement réaliser le rendu d'un arbre React au sein d'un élément DOM attaché à document.

- Ce dernier point est nécessaire pour que le composant puisse recevoir les événements du DOM.

Et lorsque le test se termine, nous voulons « nettoyer » et démonter l'arbre présent dans document.

Une façon courante de faire ça consiste à associer les blocks **beforeEach** et **afterEach** afin qu'il s'exécutent systématiquement autour de chaque test, ce qui permet d'en isoler les effets

```
import { unmountComponentAtNode } from "react-dom";

let container = null;
beforeEach(() => {
  // met en place un élément DOM comme cible de rendu
  container = document.createElement("div");
  document.body.appendChild(container);
});

afterEach(() => {
  // nettoie en sortie de test
  unmountComponentAtNode(container);
  container.remove();
  container = null;
});
```

ACT()

Lorsqu'on écrit des tests UI, des tâches comme le rendu lui-même, les événements utilisateurs ou encore le chargement de données peuvent être considérées comme autant « d'unités » d'interaction avec l'interface utilisateur.

react-dom/test-utils fournit une fonction utilitaire appelée **act()** qui s'assure que toutes les mises à jour relatives à ces « unités » ont bien été traitées et appliquées au DOM avant que nous ne commençons à exprimer nos assertions

Ça nous aide à rapprocher nos tests du comportement que de véritables utilisateurs constateraient en utilisant notre application.

```
act(() => {  
  // rendu des composants  
});  
// exécution des assertions
```

Remarque

Le terme act vient de l'approche **Arrange-Act-Assert**.

TESTER LE RENDU D'UN COMPOSANT

Vous voudrez fréquemment vérifier que le rendu d'un composant est correct pour un jeu de **props** donné.

Prenons un composant simple qui affiche un message basé sur une **prop** :

```
// hello.js

import React from "react";

export default function Hello(props) {
  if (props.name) {
    return <h1>Bonjour, {props.name} !</h1>;
  } else {
    return <span>Salut, étranger</span>;
  }
}
```

```
// hello.test.js
import React from "react";
import { render, unmountComponentAtNode } from "react-dom";
import { act } from "react-dom/test-utils";
import Hello from "./hello";

let container = null;
beforeEach(() => {
  // met en place un élément DOM comme cible de rendu
  container = document.createElement("div");
  document.body.appendChild(container);
});

afterEach(() => {
  // nettoie en sortie de test
  unmountComponentAtNode(container);
  container.remove();
  container = null;
});

it("s'affiche avec ou sans nom", () => {
  act(() => {
    render(<Hello />, container);
  });
  expect(container.textContent).toBe("Salut, étranger");

  act(() => {
    render(<Hello name="Jenny" />, container);
  });
  expect(container.textContent).toBe("Bonjour, Jenny !");

  act(() => {
    render(<Hello name="Margaret" />, container);
  });
  expect(container.textContent).toBe("Bonjour, Margaret !");
});
```

TESTER LE CHARGEMENT DE DONNÉES

Au lieu d'appeler l'API réelle dans tous vos tests, vous pouvez simuler les requêtes et renvoyer des données factices.

Simuler le chargement de données avec de « fausses » données évite de fragiliser les tests lors d'un back-end indisponible, et les accélère en prime.

```
// user.js
import React, { useState, useEffect } from "react";

export default function User(props) {
  const [user, setUser] = useState(null);
  async function fetchUserData(id) {
    const response = await fetch("/") + id);
    setUser(await response.json());
  }

  useEffect(() => {
    fetchUserData(props.id);
  }, [props.id]);

  if (!user) {
    return "Chargement...";
  }

  return (
    <details>
      <summary>{user.name}</summary>
      <strong>{user.age}</strong> ans
      <br />
      vit à {user.address}
    </details>
  );
}
```

```
// user.test.js
import React from "react";
import { render, unmountComponentAtNode } from
"react-dom";
import { act } from "react-dom/test-utils";
import User from "../user";

let container = null;
beforeEach(() => {
  // met en place un élément DOM comme cible de rendu
  container = document.createElement("div");
  document.body.appendChild(container);
});

afterEach(() => {
  // nettoie en sortie de test
  unmountComponentAtNode(container);
  container.remove();
  container = null;
});

it("affiche les données utilisateur", async () => {
  const fakeUser = {
    name: "Joni Baez",
    age: "32",
    address: "123, Charming Avenue"
  };
});
```

```
// user.test.js suite
jest.spyOn(global, "fetch").mockImplementation(() =>
  Promise.resolve({
    json: () => Promise.resolve(fakeUser)
  })
);

// Utilise la version asynchrone de `act` pour appliquer les promesses
accomplies
await act(async () => {
  render(<User id="123" />, container);
});

expect(container.querySelector("summary").textContent).toBe(fakeUser.name);
expect(container.querySelector("strong").textContent).toBe(fakeUser.age);
expect(container.textContent).toContain(fakeUser.address);

// retire la simulation pour assurer une bonne isolation des tests
global.fetch.mockRestore();
});
```

TESTER LES ÉVÉNEMENTS

Nous allons déclencher de véritables événements DOM sur des éléments DOM, et de vérifier le résultat.

Prenons ce composant Toggle :

```
// toggle.js

import React, { useState } from "react";

export default function Toggle(props) {
  const [state, setState] = useState(false);
  return (
    <button
      onClick={() => {
        setState(previousState => !previousState);
        props.onChange(!state);
      }}
      data-testid="toggle"
    >
      {state === true ? "Éteindre" : "Allumer"}
    </button>
  );
}
```



```
// toggle.test.js

import React from "react";
import { render, unmountComponentAtNode } from "react-dom";
import { act } from "react-dom/test-utils";

import Toggle from "../toggle";

let container = null;
beforeEach(() => {
  // met en place un élément DOM comme cible de rendu
  container = document.createElement("div");
  // `container` *doit* être attaché à `document` pour que les événements
  // fonctionnent correctement.
  document.body.appendChild(container);
});

afterEach(() => {
  // nettoie en sortie de test
  unmountComponentAtNode(container);
  container.remove();
  container = null;
});
```

```
// toggle.test.js (suite)

it("change de valeur suite au clic", () => {
  const onChange = jest.fn();
  act(() => {
    render(<Toggle onChange={onChange} />, container);
  });

  // récupère l'élément bouton et déclenche quelques clics dessus
  const button = document.querySelector("[data-testid=toggle]");
  expect(button.innerHTML).toBe("Allumer");

  act(() => {
    button.dispatchEvent(new MouseEvent("click", { bubbles: true }));
  });

  expect(onChange).toHaveBeenCalledTimes(1);
  expect(button.innerHTML).toBe("Éteindre");

  act(() => {
    for (let i = 0; i < 5; i++) {
      button.dispatchEvent(new MouseEvent("click", { bubbles: true }));
    }
  });

  expect(onChange).toHaveBeenCalledTimes(6);
  expect(button.innerHTML).toBe("Allumer");
});
```

CAPTURE D'INSTANTANÉS

Les frameworks tels que Jest vous permettent aussi de sauvegarder des « instantanés » de données grâce à **toMatchSnapshot** / **toMatchInlineSnapshot**.

Avec elles, vous pouvez « sauver » la sortie de rendu d'un composant et vous assurer que toute modification qui lui sera apportée devra être explicitement confirmée en tant qu'évolution de l'instantané.

Dans l'exemple qui suit, nous affichons un composant et formatons le HTML obtenu grâce au module **pretty**, pour enfin le sauvegarder comme instantané en ligne :

```
// hello.test.js

import React from "react";
import { render, unmountComponentAtNode } from
"react-dom";
import { act } from "react-dom/test-utils";
import pretty from "pretty";

import Hello from "../hello";

let container = null;
beforeEach(() => {
  // met en place un élément DOM comme cible de rendu
  container = document.createElement("div");
  document.body.appendChild(container);
});

afterEach(() => {
  // nettoie en sortie de test
  unmountComponentAtNode(container);
  container.remove();
  container = null;
});
```

```
// hello.test.js (suite)
it("devrait afficher une salutation", () => {
  act(() => {
    render(<Hello />, container);
  });

  expect(
    pretty(container.innerHTML)
  ).toMatchInlineSnapshot(); /* ...rempli automatiquement par Jest... */

  act(() => {
    render(<Hello name="Jenny" />, container);
  });

  expect(
    pretty(container.innerHTML)
  ).toMatchInlineSnapshot(); /* ...rempli automatiquement par Jest... */

  act(() => {
    render(<Hello name="Margaret" />, container);
  });

  expect(
    pretty(container.innerHTML)
  ).toMatchInlineSnapshot(); /* ...rempli automatiquement par Jest... */
});
```

TESTS DE BOUT EN BOUT

Les tests de bout en bout sont utiles pour tester des scénarios plus longs, en particulier s'ils sont critiques à votre activité (comme des paiements ou des inscriptions).

Framework de tests E2E

- Selenium
- Puppeteer
- Cypress

Bookkeeper

select an invoice

Tubthumper

The screenshot shows the Chrome DevTools component inspector. The left pane displays the component tree with the following structure:

- App
 - Link ForwardRef
 - Link ForwardRef
 - Outlet
 - Route.Provider
 - Invoices
 - NavLink key="1995" ForwardRef** (selected)
 - Link ForwardRef
 - NavLink key="2000" ForwardRef
 - Link ForwardRef
 - NavLink key="2003" ForwardRef
 - Link ForwardRef
 - NavLink key="1997" ForwardRef
 - Link ForwardRef
 - NavLink key="1998" ForwardRef
 - Link ForwardRef
 - Outlet

The right pane shows the props and hooks for the selected **NavLink** component:

props

- children: "Santa Monica"
- style: `f style() {}`
- to: `"/invoices/1995"`
- new entry: ""

hooks

- Location:
- ResolvedPath:

rendered by

- Invoices
 - `createLegacyRoot()`
 - `react-dom@17.0.2`



OPTIMISATION |

OPTIMISATION

En interne, React fait appel à différentes techniques intelligentes pour minimiser le nombre d'opérations coûteuses sur le DOM nécessaires à la mise à jour de l'interface utilisateur (UI).

Pour de nombreuses applications, utiliser **React** offrira une UI rapide sans avoir à fournir beaucoup de travail pour optimiser les performances.

Néanmoins, il existe plusieurs façons d'accélérer votre application React.

UTILISER LA VERSION DE PRODUCTION

Si votre projet est construit avec **Create React App**, exécutez :

npm run build

Cela générera la version de production de votre application dans le répertoire `build/` de votre projet.

VIRTUALISER LES LISTES LONGUES

Si votre application génère d'importantes listes de données (des centaines ou des milliers de lignes), il est recommandé d'utiliser la technique de « fenêtrage » (*windowing*).

- Cette technique consiste à n'afficher à tout instant qu'un petit sous-ensemble des lignes, ce qui permet de diminuer considérablement le temps nécessaire au rendu des composants ainsi que le nombre de nœuds DOM créés.
- **react-window** et **react-virtualized** sont des bibliothèques populaires de gestion du fenêtrage. Elles fournissent différents composants réutilisables pour afficher des listes, des grilles et des données tabulaires. Vous pouvez également créer votre propre composant, comme l'a fait Twitter, si vous voulez quelque chose de plus adapté à vos cas d'usage spécifiques.

NE PAS EXÉCUTER AU PREMIER RENDU

Un exemple où `useEffect` ne doit pas s'exécuter au premier rendu car ici c'est inutile.

Il s'exécute à chaque re-render (mise à jour)

```
const useSemiPersistentState = (key, initialState) =>
{
  const isMounted = React.useRef(false);
  const [value, setValue] = React.useState(
    localStorage.getItem(key) || initialState
  );
  React.useEffect(() => {
    if (!isMounted.current) {
      isMounted.current = true;
    } else {
      console.log('A');
      localStorage.setItem(key, value);
    }
  }, [value, key]);
  return [value, setValue];
};
```

MERCI POUR VOTRE ATTENTION 🙏