

# React.js

Introduction

# Sommaire

1. Qu'est ce que React ?
2. La programmation fonctionnelle
3. Créer un composant React
4. Gestion d'état de l'application (MobX vs Redux vs Flux)
5. Redux
6. La gestion des routes
7. Tester une application React

**William HELIE-JOLY**  
william.heliejoly@gmail.com

# 1. Qu'est ce que React ?

# Qu'est ce que React ?

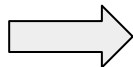
## History

2009 : Angular

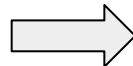
2013 : React

# Qu'est ce que React ?

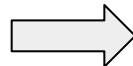
Pourquoi Facebook a t-il créé React ?



- MVC (Modèle-vue-contrôleur)
- Imposant moteur de templating HTML
- Rapide a prendre en main
- Syntaxe simple



- MVC (Modèle-vue-contrôleur)
- Fortes convention
- Simple a prendre en main



- MVC (Modèle-vue-collection)
- Simplification dans la déclaration d'événements
- Structure du code propre

# Qu'est ce que React ?

Pourquoi Facebook a t-il créé React ?

## FACEBOOK advertising campaign

**AD SET:** Define your audience, budget and schedule

**Audience**  
Define who you want to see your ads. [Learn more.](#)

NEW AUDIENCE ▾

Custom Audiences ⓘ  
Add Custom Audiences or Lookalike Audiences


Exclude | Create New ▾

Locations ⓘ  
✓ **Everyone in this location**  
People who live in this location  
People recently in this location  
People traveling in this location  
Include ▾ | Add locations  
Add Bulk Locations...

Age ⓘ  
35 ▾ - 65+ ▾

Gender ⓘ  
**All** Men Women


Languages ⓘ  
**English (All)**  
Enter a language...

**Audience Definition**  
  
Your audience is too specific for your ads to be shown. Try making it broader.

**Audience Details:**

- Location:
  - United States: Sonoma (+25 mi)  
California
- Age:
  - 35 - 65+
- Language:
  - English (All)
- Placements:
  - News Feed on desktop computers, News Feed on mobile devices or Right column on desktop computers
- People Who Match:
  - Interests: arts

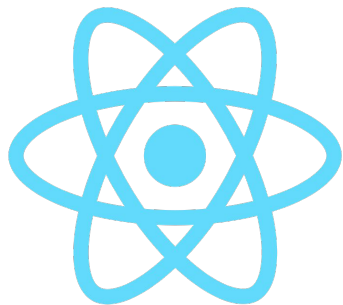
Potential Reach: Fewer than 1000 people

**Estimated Daily Reach**  
  
0 of 100 ⓘ  
This is only an estimate. Numbers shown are based on the average performance of ads targeted to your selected audience.

**Detailed Targeting ⓘ** INCLUDE people who match at least ONE of the following ⓘ  
Interests > Additional Interests

# Qu'est ce que React ?

Pourquoi Facebook a t-il créé React ?



**REACT**  
une nouvelle approche



# Qu'est ce que React ?

- Une bibliothèque javascript, créée par Facebook, qui sert à construire des interfaces riches et dynamiques.
- Correspond uniquement au V dans MVC
- Déclarative, prédictive et réactive
- Architecture basée sur le principe des web-components
- Utilise le Virtual DOM pour mettre à jour efficacement le DOM
- Utilise une syntaxe spéciale, le JSX, qui mêle HTML et JavaScript
- Server-Side Rendering
- Facilement testable

Qu'est ce que React ?

**V dans MVC**

# Qu'est ce que React ?

Le V dans MVC

React.js est une bibliothèque, pas un Framework comme Angular ou Ember.js

Pas de Model, Controller, pas de routeur, pas de gestion de requêtes serveur, React.js ne contient que le code qui s'occupe de mettre à jour le DOM quand l'état d'un composant change.

Qu'est ce que React ?

# Le Virtual DOM

# Qu'est ce que React ?

## Le Virtual DOM

### **Problèmes :**

Manipuler et modifier le DOM est coûteux et crée des problèmes de performance.

Pour avoir une interface fluide, il faut réduire au maximum le nombre de changements sur le DOM.

# Qu'est ce que React ?

Le Virtual DOM

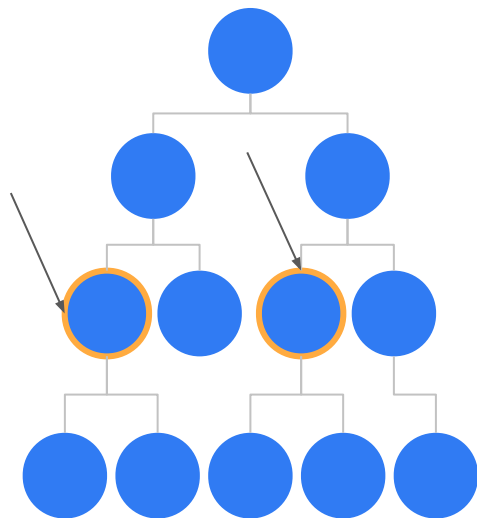
## **Solution :**

Manipuler une représentation virtuelle du DOM actuel pour mettre à jour uniquement les changements du DOM en une fois.

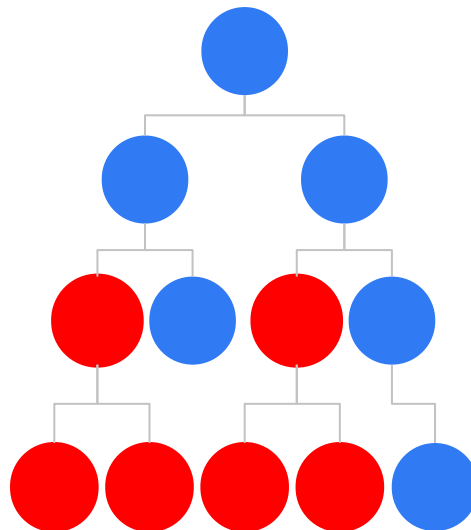
# Qu'est ce que React ?

## Le Virtual DOM

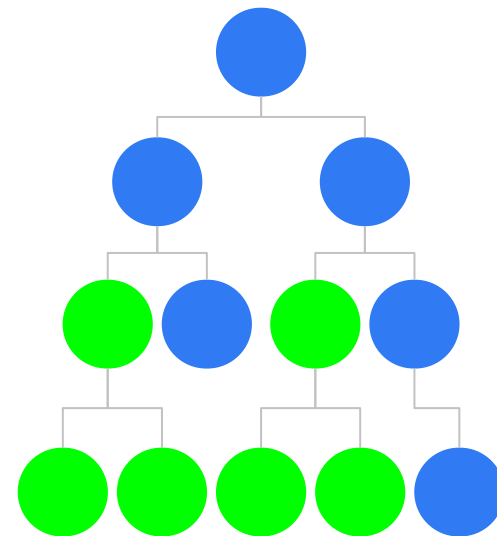
Changements d'état



Calcul des différences



Mise à jour du DOM



# Qu'est ce que React ?

## Le Virtual DOM

Le **VDOM** est comme

Git pour le DOM

12		slides/ticker-diff.html	View	
		@@ -37,13 +37,13 @@		
37	37	<div id="main">		
38	38	<div>		
39	39	<ul class="ticker">		
40		- <li class="loss">		
41		- <time datetime="2014-08-31 17:04:16">5:04:16pm</time>:		
42		- AAPL 412.52 (-7.10)		
	40	+ <li class="gain">		
	41	+ <time datetime="2014-08-31 17:04:18">5:04:18pm</time>:		
	42	+ GOOG 642.12 (+6.99)		
43	43	</li>		
44	44	<li class="loss">		
45		- <time datetime="2014-08-31 17:04:16">5:04:16pm</time>:		
46		- GOOG 642.10 (-37.01)		
	45	+ <time datetime="2014-08-31 17:04:17">5:04:17pm</time>:		
	46	+ AAPL 412.24 (-7.19)		
47	47	</li>		
48	48	<li class="loss">		
49	49	<time datetime="2014-08-31 17:04:16">5:04:16pm</time>:		
		@@ -80,4 +80,4 @@		



# Qu'est ce que React ?

## Le JSX

<https://facebook.github.io/react/docs/jsx-in-depth.html>

# Qu'est ce que React ?

## Le JSX

Le **JSX** est du sucre syntaxique au dessus de l'**API** React qui permet d'écrire plus simplement du code grâce à une syntaxe qui mélange du **XML** et du **JavaScript**.

# Qu'est ce que React ?

Le JSX

Sans JSX

```
React.createElement(  
  MyButton,  
  {color: 'blue', shadowSize: 2},  
  'Click Me'  
)
```

Avec JSX

```
<MyButton color="blue" shadowSize={2}>  
  Click Me  
</MyButton>
```

# Qu'est ce que React ?

## Le JSX

JSX produit des “**éléments**” React

Par exemple avec JSX nous pouvons écrire :

```
const element = <h1>Hello, world!</h1>;
```

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

# Qu'est ce que React ?

## Le JSX

```
// Variables
const name = "John";
const city = "Toulouse";

// Fonctions
const validateCity = (city) => city !== "";

const Title = (
  <div>
    <h1>Hello, {name}. </h1>
    /* Expressions */
    {validateCity(city) && <h2>Je viens de {city}.</h2>}
  </div>
);
```

En **JSX**, nous pouvons évaluer directement des **variables**, des **expressions**, voir même des **fonctions**. Cela reste du JavaScript.

Toutefois, l'évaluation reste **limitée**. Nous ne pouvons pas **déclarer de variables**, ni faire de **switch case**.

# Qu'est ce que React ?

## Le JSX

```
// Variables
const name = "John";
const city = "Toulouse";

// Fonctions
const validateCity = (city) => city !== "";

const Title = (
  <div>
    <h1>Hello, {name}. </h1>
    {/* Expressions */}
    {validateCity(city) && <h2>Je viens de {city}.</h2>}
  </div>
);
```

Le **JSX** est plus proche de JavaScript que de l'**HTML**, React DOM utilise la convention de dénomination de la propriété **camelCase** au lieu des noms d'attribut HTML.

Exemple :

**class** en HTML devient **className** en JSX.  
**tabindex** en HTML devient **tabIndex** en JSX.

Qu'est ce que React ?

# Les composants

# Qu'est ce que React ?

## Les composants

Les composants sont des morceaux d'interfaces indépendants, réutilisables et permettant d'ordonner l'interface en termes de fonctionnalités.

Ils sont comme des fonctions JavaScript :

- Sauf que les arguments sont appelés "**props**"
- Ils retournent ce qui doit être **affiché à l'écran**
- Ils possèdent un **état** et un **cycle de vie**.

On pourrait les comparer à des briques de legos.



# Qu'est ce que React ?

## Les composants

Un composant est divisé en **4 parties** :

1. Son état (state)
2. Les méthodes du cycle de vie
3. La méthode render qui retourne ce que le composant doit afficher
4. Ses propriétés appelées **props** (celles-ci sont en **read-only**)

# Qu'est ce que React ?

## Les composants

L'état d'un composant (**state**) est une variable privée au composant qui sert à conserver des changements internes du composant.

La mise à jour (**uniquement via setState()**) provoque un nouveau **render** du composant.

Celui-ci est un objet qui peut contenir n'importe quel type de données (object, function, Composants...)

# Qu'est ce que React ?

## Les composants

### Les méthodes du cycle de vie

#### Montage

- constructor()
- static getDerivedStateFromProps()
- render()
- componentDidMount()
- componentWillUnmount()

#### Mise à jour

- setState()
- forceUpdate()
- static getDerivedStateFromProps()
- shouldComponentUpdate()
- render()
- getSnapshotBeforeUpdate()
- componentDidUpdate()

#### Gestion erreur

- componentDidCatch()

# Qu'est ce que React ?

## Les composants

La méthode **render()** est appelée chaque fois qu'un composant doit être ré-affiché à l'écran.

Puisqu'elle est tout le temps appelée dès que l'état (**state**) ou les **props** d'un composant change, elle doit s'occuper de retourner uniquement la partie du **DOM** que le composant représente.

Pour faire simple, c'est dans cette méthode que l'on y trouve le **JSX** qui sera retranscrit en **HTML** au navigateur.

# Qu'est ce que React ?

## Les composants

Les **props** sont comme des arguments passés à une fonction, sauf que les composants **doivent** être des fonctions **pures**. C'est à dire que les **props** ne sont **jamais modifiés** par le composant directement.

Les composant peut recevoir de nouvelles **props** ou en envoyer aux composants enfants de nouvelles **props**.

# Qu'est ce que React ?

## Les composants

### Composant simple

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

### Composant imbriquant d'autres

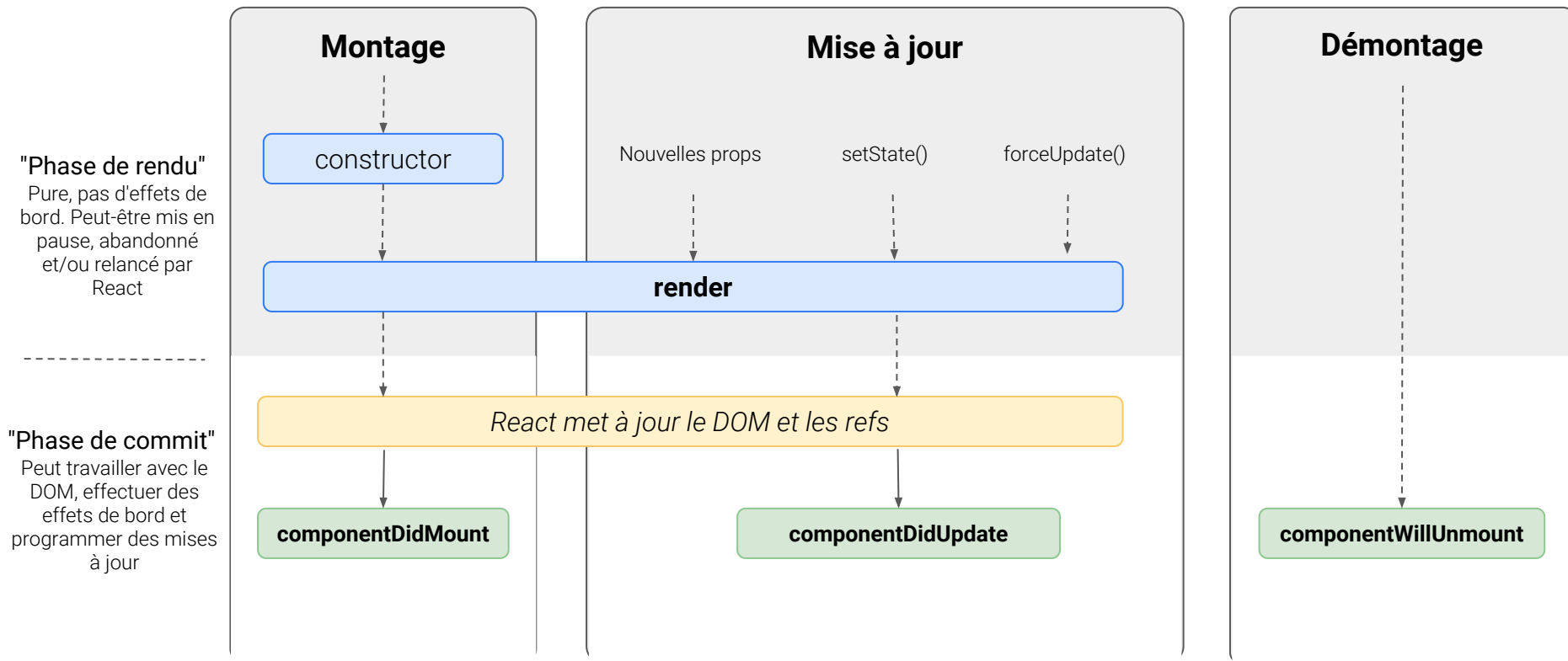
```
class Profile extends React.Component {  
  render() {  
    return (  
      <div className="UserInfo">  
        <Avatar user={props.user} />  
        <div className="UserInfo-name">  
          {props.user.name}  
        </div>  
      </div>  
    );  
  }  
}
```

Qu'est ce que React ?

# Cycle de vie des composants

# Qu'est ce que React ?

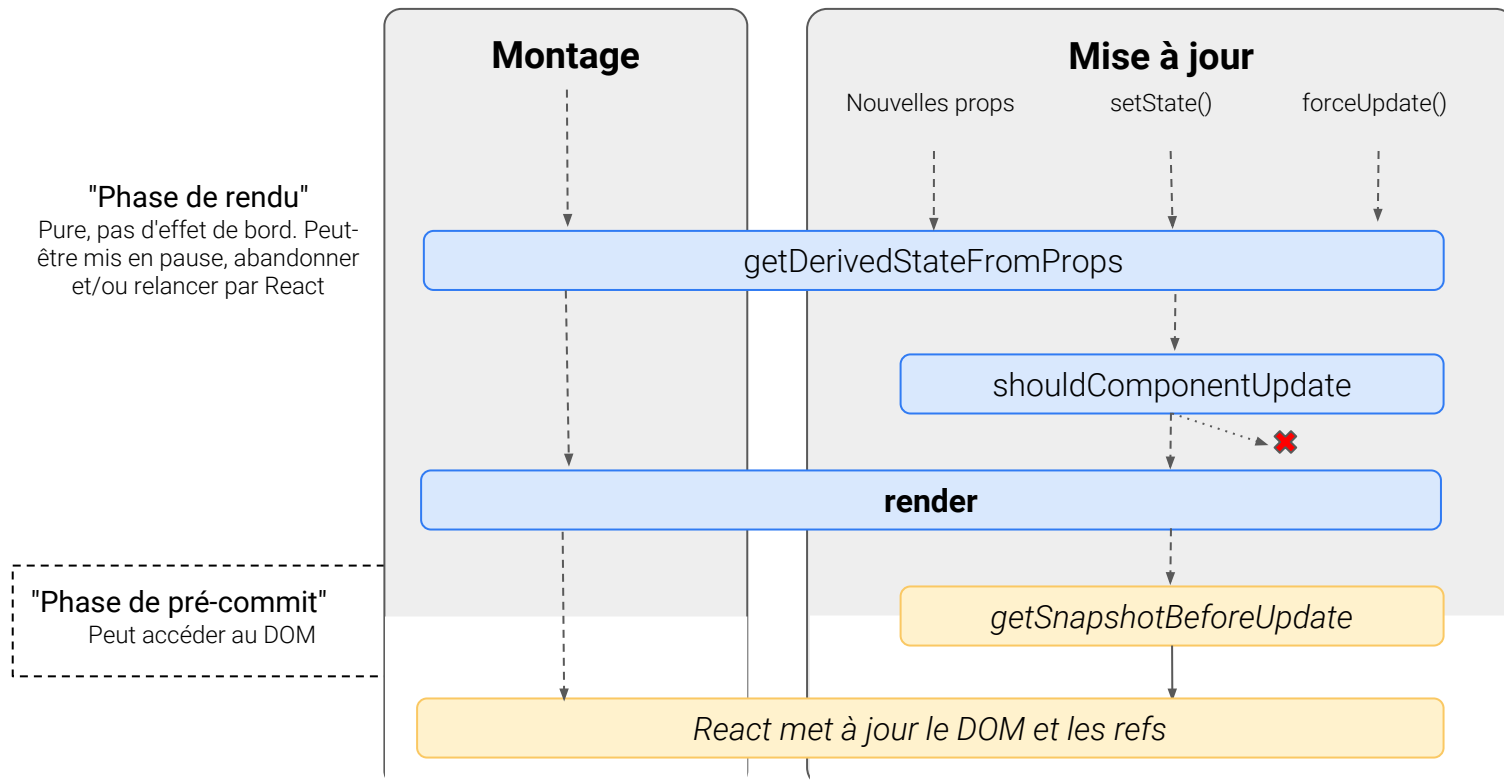
## Cycle de vie des composants





# Qu'est ce que React ?

## Cycle de vie des composants



## 2. La programmation fonctionnelle

# La programmation fonctionnelle

la **programmation fonctionnelle** met en avant l'application des fonctions, contrairement au modèle de programmation impérative qui met en avant les changements d'état.

# La programmation fonctionnelle

Les principes de la programmation fonctionnelle sont :

- L'immutabilité
- Les fonctions pures
- La composition

## L'immuabilité

# La programmation fonctionnelle

## Immutabilité

Un objet **immutable** est un objet dont la référence (donc l'état) ne peut être modifiée après avoir été créé. Cette immutabilité permet d'assurer l'intégrité des données tout au long du programme, notamment dans les langages supportant le multi-threading.

# La programmation fonctionnelle

## Immutabilité

*“Ou est le problème de modifier directement le contenu des objets ? Tout le monde fait ça depuis des années, on l'a même appris à l'école.”*

# La programmation fonctionnelle

## Immutabilité

### Exemple

```
// est-on sur du contenu de data ?  
var s = 0;  
for (var index = 0; index < data.length; index++)  
{  
  s += data[index];  
}  
  
// ou est passé a ?  
let b = 4;  
let a = b;  
a = c;
```

### Problèmes :

1. Le code ne reflète pas ce que l'on veut faire, mais il représente les changements d'états dans la machine. On écrit du code pour la machine
2. Si **data** vient à changer, on a aucun moyen de vérifier si le code fonctionne toujours. La mutabilité crée des effets de bords incontrôlables même avec la plus grande des rigueurs.
3. On peut simplement "perdre" des variables et causer des fuites de mémoires.



## Les fonctions

# La programmation fonctionnelle

## Les fonctions

En programmation, les **fonctions** se classent en 2 catégories : **pures** et **impures**

# La programmation fonctionnelle

## Les fonctions

### Fonction impure

```
function accelerate(car, accel) {  
  car.speed += accel;  
  return car;  
}
```

La valeur d'une des input est modifiée

### Fonction pure

```
function accelerate(car, accel) {  
  return { ...car, speed: car.speed + accel };  
}
```

Les input ne sont pas modifiées, juste utilisées

# La programmation fonctionnelle

## Les fonctions

Une fonction **impure** est une fonction qui **change l'environnement**, ou les **arguments** passés en paramètre.

Cette modification des paramètres nous oblige à connaître constamment l'état de chaque variable, à chaque appel de ces fonctions.

Cela rend le code plus obscur à **comprendre**, **tester**, et à **maîtriser** à cause de tous les effets de bords que cela peut engendrer.

# La programmation fonctionnelle

## Les fonctions

Une fonction est dite **pure** si son résultat dépend uniquement des paramètres qu'elle prend en entrée. Une fonction pure ne doit pas altérer l'environnement dans lequel elle est exécutée.

Les fonctions sont considérées comme des données. Elles peuvent être passées à d'autres fonctions comme paramètres et elles peuvent être retournées comme résultats de fonctions.

# La programmation fonctionnelle

## Les fonctions

### Avantage des fonctions pures

Les fonctions pures ont pour avantage d'être **prédictibles**. Ce qui permet de les tester plus facilement et surtout de mettre leur **résultat en cache** pour ne pas avoir à refaire le calcul pour des valeurs qu'on a déjà traitées.

Les fonctions pures sont souvent utilisées pour générer d'autres fonctions. Dans ce cas, elles sont appelées "Higher Order Functions" ou "Fonctions de rang supérieur".

*Note: Une fonction de rang supérieur peut ne pas être une fonction pure.*

# La programmation fonctionnelle

## Les fonctions

### Fonction d'ordre supérieur

```
const sum1 = (x) => (y) => (x + y);

function sum2(x) {
  return function (y) {
    return x + y;
  };
};
```

Elles ( *sum1* et *sum2* ) permettent de recevoir une valeur **x** puis elles retournent une fonction qui attend un **y** et cette dernière retourne la somme de **x** et **y**.

# La programmation fonctionnelle

## Les fonctions

Les Fonctions d'ordre supérieur utilisent le principe de fermeture (**closure**).

Une **closure** est une fonction accompagnée de son environnement lexical.

L'environnement lexical d'une fonction est l'ensemble des variables non locales qu'elle a capturé, soit par valeur (c'est-à-dire par copie des valeurs des variables), soit par référence (c'est-à-dire par copie des adresses mémoires des variables).

Une fermeture est donc créée, entre autres, lorsqu'une fonction est définie dans le corps d'une autre fonction et utilise des paramètres ou des variables locales de cette dernière.



# La programmation fonctionnelle

## Les fonctions

**Ok, mais à quoi ça sert des fonctions qui retournent des fonctions ?**

# La programmation fonctionnelle

## Les fonctions

La **closure** permet de résoudre le problème d'accès à certaines variables lors de l'exécution.

Exemple courant : on souhaite ajouter un argument à une **callback** sur un événement (**onClick**, **onMouseOver**...)

# La programmation fonctionnelle

## Les fonctions

### Exemple de closure

```
const button = $("#myButton");

function makeDeleteOnClick(id) {
  return function (evt) {
    $.post(`delete?id=${id}`, function(data) {
      $(".result").html(data);
    });
  }
}

button.onClick = makeDeleteOnClick(3);
```

La signature de la **callback** onClick est

```
// signature d'un handler d'événement javascript
function handler(evt) {
}
```

Ici, la closure permet d'accéder à **id** alors que celui-ci est normalement **inaccessible** du fait de l'appel du callback avec les paramètres de la signature.

# La programmation fonctionnelle

## Les fonctions

En résumé, une **closure** est une **boîte fermée** où tout le contenu interne est accessible qui retourne la forme de la fonction qu'elle englobe.

## La composition

# La programmation fonctionnelle

## La composition

La **composition de fonctions** est un concept mathématique qui permet de combiner plusieurs fonctions en une nouvelle fonction.

# La programmation fonctionnelle

## La composition

La clé de la **composition** est d'avoir des fonctions composables, c'est à dire des fonctions à **argument unique** et qui retourne une seule valeur.

On peut transformer toutes les fonctions à argument multiple en une fonction composable en appliquant le **currying**.

# La programmation fonctionnelle

## La composition

Le currying est l'action de transformer une fonction qui prend plusieurs arguments en une suite de retour de fonctions qui prennent un seul argument

```
const add1 = (x, y, z) => (x + y + z);  
add1(1, 2, 3)  
// => 6  
  
const add2 = (x) => (y) => (z) => (x + y + z);  
add2(1)(2)(3)  
// => 6
```



# La programmation fonctionnelle

La composition

**Apprentissage par l'exemple.**

Créons ensemble un petit moteur de template.

# La programmation fonctionnelle

## La composition

### Composition en action

```
const tag = (tag) => (content) => `<${tag}>${Array.isArray(content) ? content.join('') : content}</${tag}>`;
const b = tag('b');
const li = tag('li');
const ul = content => tag('ul')(content.map(li))
ul(['toto', 'titi'].map(b));
=> "<ul><li><b>toto</b></li><li><b>titi</b></li></ul>"
```

En quelques lignes de code, nous avons pu créer facilement toute une panoplie de fonctions qui génère les tags souhaités.

Notez surtout qu'ici, on décrit par fonctionnalités notre code, et non pas, ce que la machine est censée faire.

# La programmation fonctionnelle

HOC

**Introduction d'un HOC**  
High Order Component

# La programmation fonctionnelle

HOC

## Basic HOC

```
function ppHOC(WrappedComponent) {  
  return class PP extends React.Component {  
    render() {  
      return <WrappedComponent {...this.props}/>  
    }  
  }  
}
```

# La programmation fonctionnelle

HOC

## HOC pour le style

```
function ppHOC(WrappedComponent) {  
  return class PP extends React.Component {  
    render() {  
      return (  
        <div style={{display: 'block'}}>  
          <WrappedComponent {...this.props}/>  
        </div>  
      )  
    }  
  }  
}
```

# La programmation fonctionnelle

## HOC

```
function ppHOC(WrappedComponent) {  
  return class PP extends React.Component {  
    constructor(props) {  
      super(props)  
      this.state = {  
        name: ''  
      }  
  
      this.onChange = this.onChange.bind(this)  
    }  
    onChange(event) {  
      this.setState({  
        name: event.target.value  
      })  
    }  
    render() {  
      const newProps = {  
        name: {  
          value: this.state.name,  
          onChange: this.onChange  
        }  
      }  
      return <WrappedComponent {...this.props} {...newProps}/>  
    }  
  }  
}
```

# La programmation fonctionnelle

La composition

**PropTypes**

# La programmation fonctionnelle

La composition

## Typechecking With PropTypes

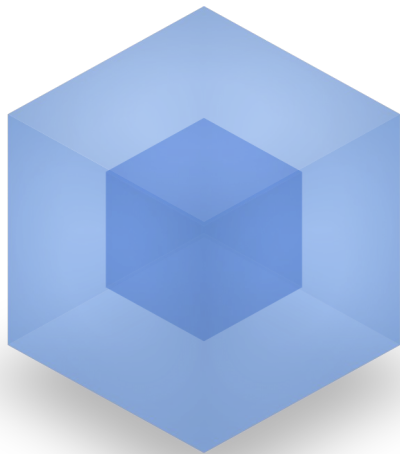
```
Developer.proptypes = {  
  language: PropTypes.string,  
}
```

```
Developer.defaultProps = {  
  language: 'JavaScript',  
}
```



### 3. Créer un composant React

# Créer un composant React



**webpack**  
MODULE BUNDLER

# Créer un composant React

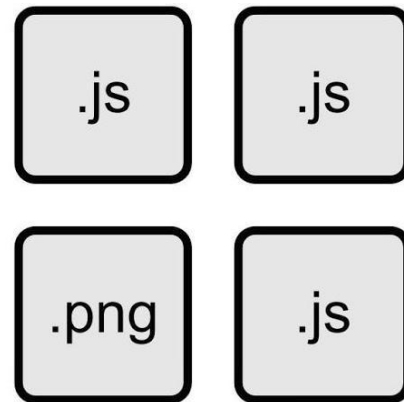
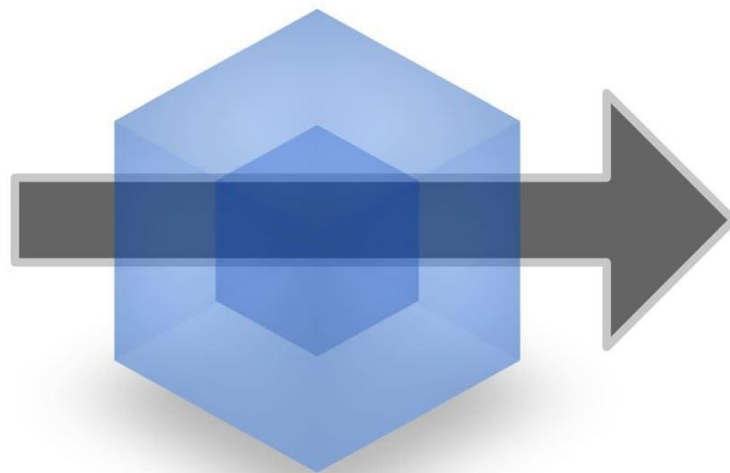
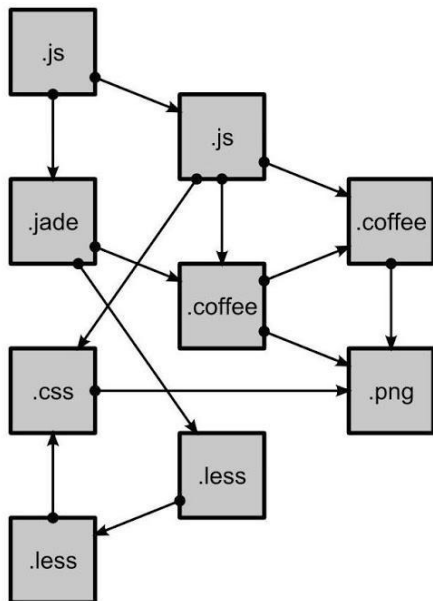
## Webpack : Les fonctionnalités

- Ressources statiques (CSS, images, fontes) disponible en tant que module
- Intégrer et consommer des bibliothèques tierces très simplement en tant que module
- Séparer votre **build** en plusieurs morceaux, chargés à la demande
- Garder un chargement initial très rapide si besoin
- Personnaliser la plupart des étapes du processus
- Adapté pour les gros projets

**Webpack** propose un système de **loader** qui permet de transformer tout et n'importe quoi en JavaScript.  
Ainsi, tout est consommable en tant que **module**.

# Créer un composant React

*Webpack transforme une multitude de fichiers en  
lots par responsabilité*



# Créer un composant React



# Créer un composant React

**NPM** est le gestionnaire de paquet officiel de **Node.js**.

Npm fonctionne avec un terminal, il permet de gérer les dépendances d'une application.

# Créer un composant React

*BABEL*

# Créer un composant React

**Babel** est un compilateur **open-source** utilisé dans le développement web.

**React** transpile le **JSX**  
et  
**Babel** transpile les versions de **JS**



**Mettre en place son  
environnement de travail**

# Créer un composant React

## Étape 1

Installer Node (<https://nodejs.org/en/>)

# Créer un composant React

## Étape 2

Installer les modules nodes

```
mkdir -p react-formation/exo1 && cd react-formation/exo1

npm init -y

# dépendances pour travailler
npm install webpack webpack-dev-server webpack-cli --save-dev
npm install babel-loader @babel/core @babel/preset-env @babel/preset-react --save-dev
npm install @babel/preset-env --save-dev

# dépendances utilisées par le navigateur
npm install react react-dom prop-types --save
```

# Créer un composant React

## Étape 3

### Architecture de dossiers

```
react-app
|- package.json
|- webpack.config.js
|- /dist
  |- bundle.js
  |- index.html
|- /src
  |- index.js
|- /node_modules
```

# Créer un composant React

## Étape 3

### Configuration

```
{
  "name": "votre nom",
  "description": "Mon 1er projet avec React",
  ...
  "private": true,
  "scripts": {
    "start": "webpack-dev-server --config ./webpack.config.js --mode development",
  },
  ...
}
```

# Créer un composant React

## Ajoutons la gestion de css

```
npm install --save-dev style-loader css-loader
```

## Modifier configuration webpack

### webpack.config.js

```
const path = require('path');
module.exports = {
  entry: './src/index.js',
  output: {
    filename: 'bundle.js',
    path: path.resolve(__dirname, 'dist')
  },
  devServer: {
    contentBase: './dist',
  },
  module: {
    rules: [
      {
        test: /\.css$/,
        use: [
          'style-loader',
          'css-loader'
        ]
      }
    ]
  }
}
```

# Créer un composant React

## Ajoutons Babel

.babelrc

```
{
  "presets": [
    "@babel/preset-env", "@babel/preset-react"
  ]
}
```

## Modifier configuration webpack

webpack.config.json

```
module.exports = {
  ...
  module: {
    rules: [
      ...
      { test: /\.js$/, exclude: /node_modules/, use: ["babel-loader"] }
    ]
  },
  resolve: {
    extensions: ['*', '.js', '.jsx']
  },
};
```

# Créer un composant React

## Base de page HTML

`./dist/index.html`

```
module.exports = {  
  ...  
  module: {  
    rules: [  
      ...  
      { test: /\.js$/, exclude: /node_modules/, use: ["babel-loader"] }  
    ]  
  },  
  resolve: {  
    extensions: ['*', '.js', '.jsx']  
  },  
};
```



# Créer un composant React

## Étape 4

Lancer l'application

```
npm run start
```

&

Lancer le navigateur

```
http://localhost:8080/
```

## Coder son 1er composant

# Créer un composant React

## **Apprentissage par l'exemple.**

Créons ensemble une petite calculette (très simple)

# Créer un composant React

**Ouvrez `react-formation/exo1/index.js`**

# Créer un composant React

On commence par indiquer que l'on va utiliser React via le mot clé **import**

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

# Créer un composant React

Ensuite on définit notre composant, via une **class**, en le faisant **hériter** de **React.Component**

```
class Calcullette extends React.Component {  
  render() {  
    return (  
      <div className="calcullette">  
        Ma calcullette  
      </div>  
    );  
  }  
}
```

On peut aussi définir un composant via une **fonction**

# Créer un composant React

Et pour finir, on indique à **React** ou faire le rendu de notre composant

```
ReactDOM.render(  
  <Calcullette />,  
  document.getElementById('app')  
);
```

# Créer un composant React

On va maintenant créer les touches de notre calculatrice

```
class Case extends React.Component {  
  render () {  
    const { children, ...rest } = this.props;  
    return (  
      <div className="case" {...rest}>  
        {children}  
      </div>  
    )  
  }  
}
```

Notez l'utilisation du **spread operator** qui permet de passer automatiquement toutes les propriétés **restantes** au div



# Créer un composant React

On peut ensuite utiliser le nouveau composant **Case** pour afficher les touches

```
<div className="cases">
  {[7, 8, 9, 4, 5, 6, 1, 2, 3, 0, '.'].map((c) => <Case key={c}>{c}</Case>)}
  <Case>=</Case>
</div>
```

Lors de l'affichage d'une liste, il ne faut pas oublier la **prop key**. Celle-ci permet à React d'**identifier** le composant dans le Virtual Dom.

# Créer un composant React

On va créer le composant qui va afficher le résultat

```
class Resultat extends React.Component {  
  render () {  
    const { children, ...rest } = this.props;  
    return (  
      <div className="resultat" {...rest}>  
        {children}  
      </div>  
    )  
  }  
}
```

# Créer un composant React

```
class Calcullette extends React.Component {  
  constructor(props) {  
    super(props);  
    this.state = {  
      resultat: 0,  
      operations: []  
    }  
    this.addOperation = this.addOperation.bind(this);  
  }  
  addOperation(op) {  
    this.setState({  
      operations: this.state.operations.concat(op)  
    });  
  }  
}
```

Dans le constructor, on définit l'état du composant et on vient bind notre méthode addOperation.

Cela nous permet de pouvoir appeler directement **this.addOperation** dans nos **callbacks**.

Dans **addOperation**, on vient mettre à jour le **state** de façon **pure** avec Array.concat

# Créer un composant React

On vient ajouter lors du click, via **onClick**, l'ajout de l'opération à effectuer, ici, ajouter la valeur de la case cliquée dans le tableau d'opérations.

```
{[7, 8, 9, 4, 5, 6, 1, 2, 3, 0, '.'].map((c) => <Case key={c} onClick={() => this.addOperation(c)}>{c}</Case>) }
```

Notez l'utilisation de la **closure** dans le **onClick** pour passer la valeur de **c** à **addOperation()**

# Gestion du SCSS de l'application

La gestion du **SCSS** est toujours une question dans un projet d'envergure.

Comment le gérer efficacement ?

- Une **structure** (architecture) définis et respecté (utiliser la logique **Webpack**)
  - Des librairies efficace tel que **styled-component**

## 4. Gestion de l'état de l'application (Flux vs Redux vs Mobx)

# Gestion de l'état de l'application

Le modèle le plus commun : **MVC** (modèle, vue, contrôleur)



# Gestion de l'état de l'application

**MVC** est très simple, facile à mettre en place et efficace.

Il est utilisé dans la plupart des framework et langages (**PHP**, **.Net**, **Java**)

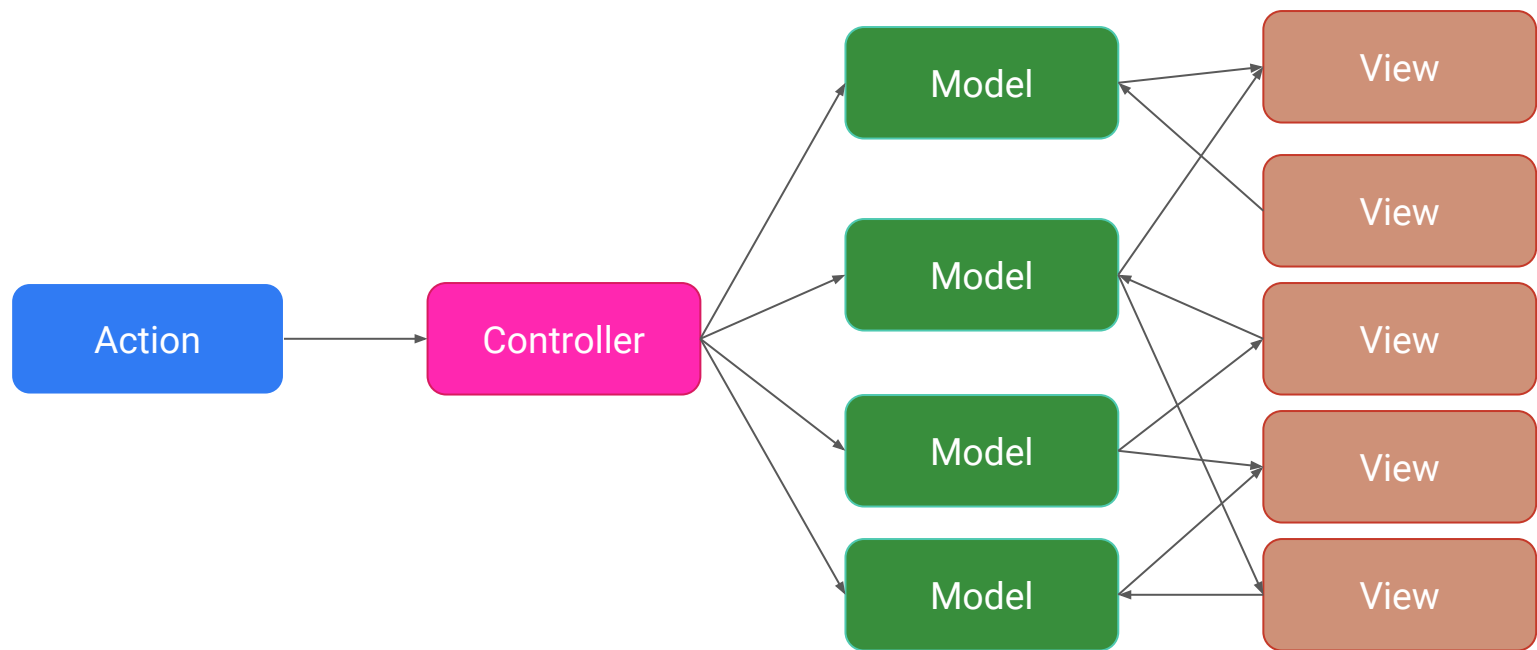
Il est scalable et marche très bien avec les technologies de rendu côté serveur car prédictif via l'URL appelée.

<http://www.exemple.com/controller/action/>



# Gestion de l'état de l'application

**MVC** n'est pas adapté pour gérer une application complexe en front uniquement.



# Gestion de l'état de l'application

Un **grand nombre** d'interactions entre les éléments de la page, **augmente** le nombre de **modèles** et de **vues**.

Les relations entre les éléments sont difficiles à suivre et à comprendre et l'ajout de fonctionnalités en devient impossible sans impacter et créer de bugs.

Le code produit, en partie impératif, finit par atteindre un point de rupture.

# Gestion de l'état de l'application

Les interactions entre les modèles et les vues vont dans les **deux sens**.  
On a un donc un flux de mise à jour **bidirectionnel**.

Cette interconnection entre les éléments empêche de déterminer à l'avance ce qui va se produire et quel sera l'**état** de l'application.

Nous sommes dans un modèle **non-prédictif** et assujetti aux problèmes d'informations **non à jour** dans différentes parties de notre application.

# Flux comme solution

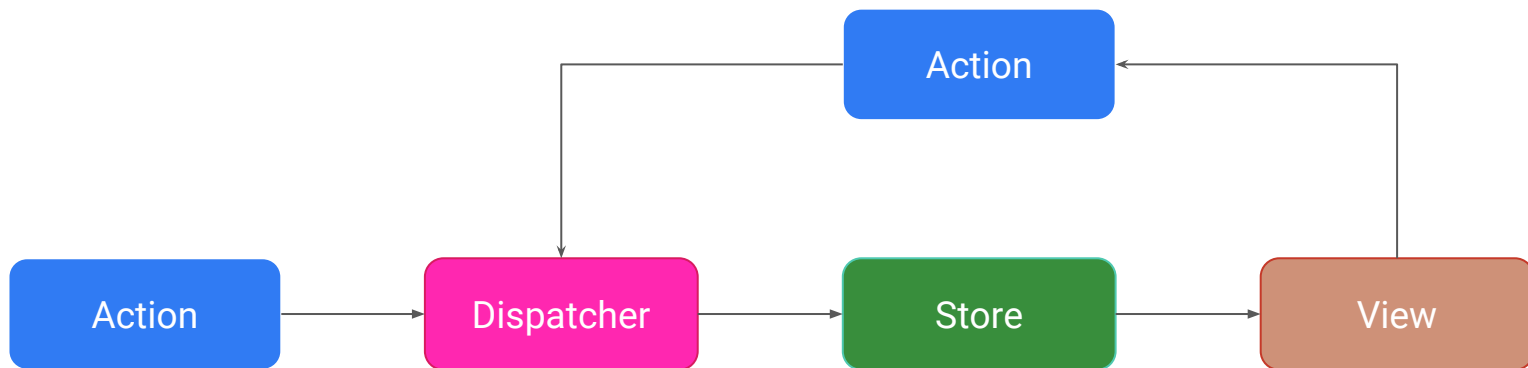
# Gestion de l'état de l'application

Le concept principal de **Flux** est que **toutes** les données *doivent* "**couler**" dans une seule **direction**, comme les props dans React.

Ce flux **unidirectionnel** est une des parties les plus signifiante de ce qui rend les choses de manière **prédictive**.

# Gestion de l'état de l'application

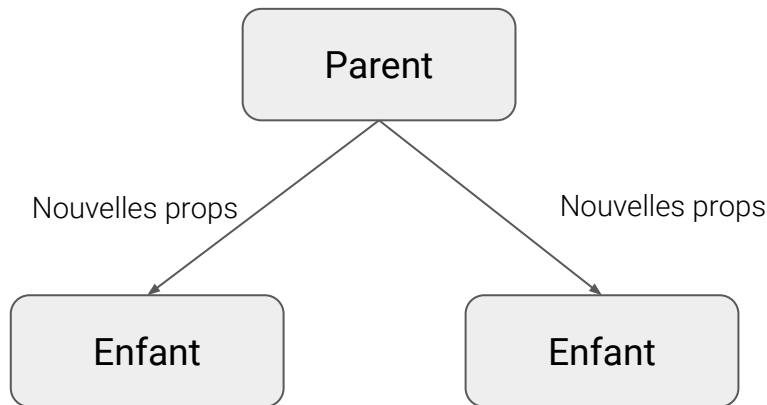
Schéma global de l'architecture **Flux**



# Gestion de l'état de l'application

## Petit rappel :

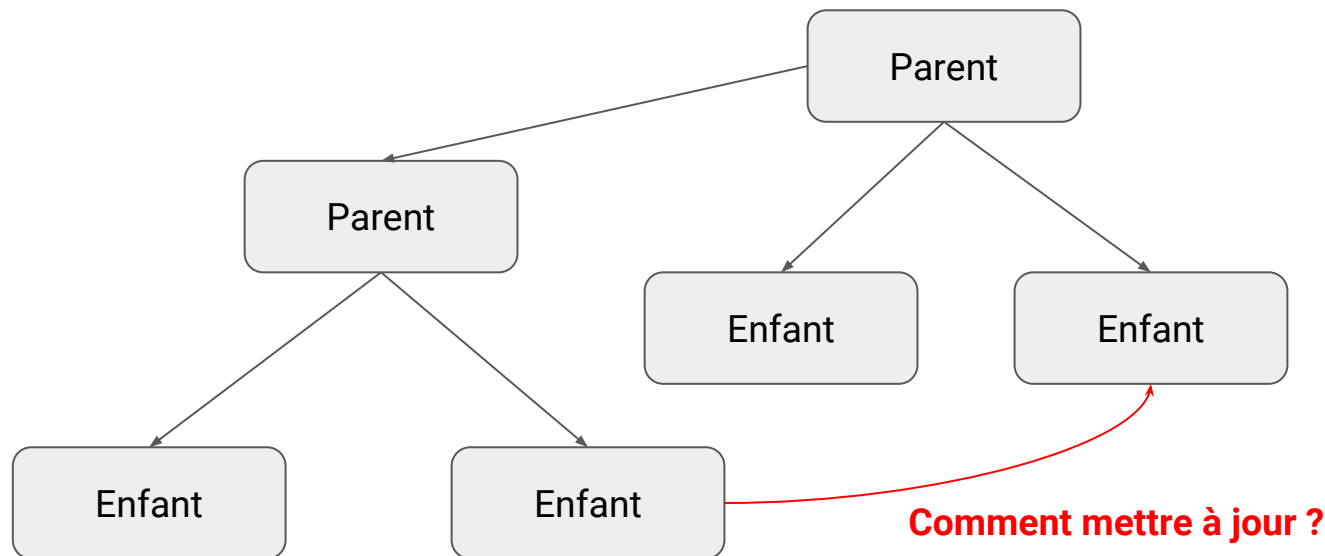
Dans **React**, les **props** d'un composant sont mis à jour par le composant **parent**



# Gestion de l'état de l'application

## Probleme :

Un enfant ne peut pas mettre à jour les **props** d'un autre enfant ou d'un parent





## **Les Actions**

# Gestion de l'état de l'application

Les composants utilisent des **Actions** pour dire qu'il y a une eu une **interaction** et donc, un **possible changement d'état** de l'application.

Ce sont des simples objets plats avec une propriété **type** qui sera utilisé pour décrire quel changement est effectué.

```
{  
  type: 'INCREMENT_COUNTER',  
  amount: '5',  
}
```

# Gestion de l'état de l'application

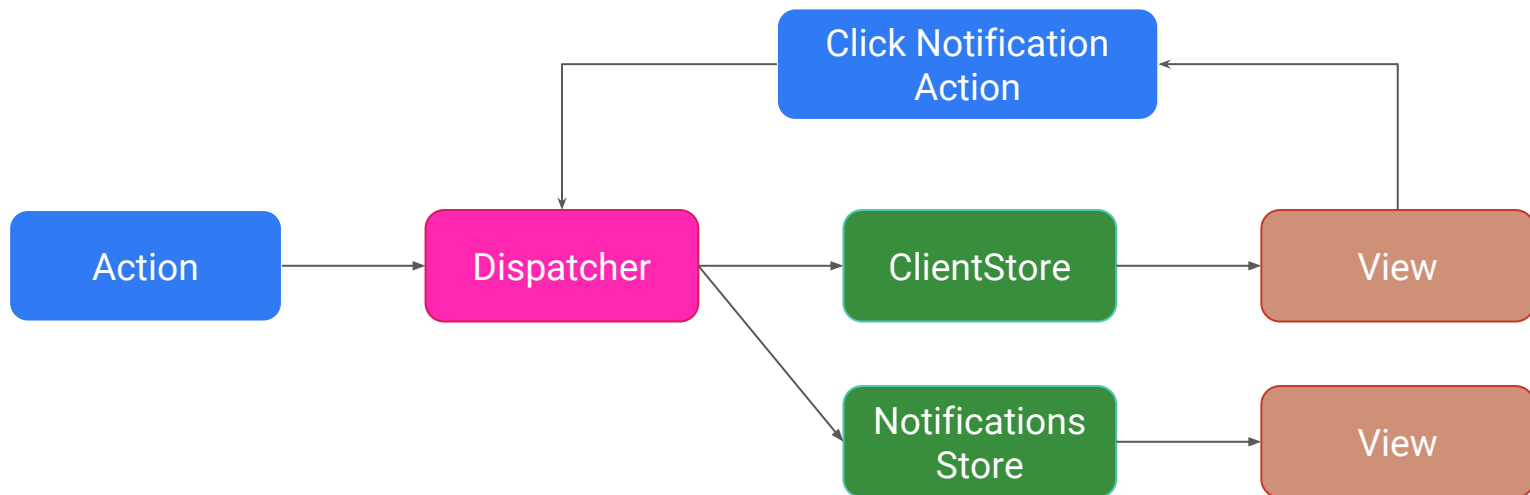
Les **Actions** sont récupérées par le **Dispatcher** et sont, ensuite, envoyées au **Store** de l'application.

```
{  
  type: 'INCREMENT_COUNTER',  
  amount: '5',  
}
```

# Gestion de l'état de l'application

**N.B :** Il y a un **SEUL *Dispatcher*** dans toute l'application.

Par contre, il peut y avoir **plusieurs *Store*** dans l'application et les ***Actions*** sont envoyées à **tous les stores**.



## **Les Stores**

# Gestion de l'état de l'application

Les **Stores** sont les structures qui contiennent tous les états de l'application, ainsi que la logique qui permet de mettre à jour ces états.

Ce sont, comme les **Actions**, des objets javascript (**class** ou **fonctions**).

Une application peut avoir différents **Stores** qui ne seront responsables que de leur portion de l'état de l'application.

# Gestion de l'état de l'application

Les **Stores**, au moment de leur création, s'enregistrent au **Dispatcher** et fonctionnent suivant quelques règles:

- L'état ne peut être changé qu'en réponse d'une **Action** reçue. Ils ne contiennent que des getters, pas de setters.
- À chaque fois qu'une donnée d'un store change, il doit émettre un *change event* qui sera broadcasté à toutes les vues pour effectuer un nouveau rendu.

# Gestion de l'état de l'application

Il est important de comprendre que les **Stores** sont des objets contenant à la fois une partie de l'état de l'application et le moyen de modifier cet état en réponse aux actions reçues.

Tous les **Stores** recevant toutes les **Actions**, un **Store** déterminera s'il doit ou non modifier son état lorsqu'il reçoit une **Action** en examinant le **type** de cette **Action**.

Si le **Store** a une logique liée à ce **type**, il effectuera les opérations appropriées, puis enverra un événement de modification aux **Views** configurées pour écouter les modifications apportées à partir de ce **Store**.



## **Les Views**

# Gestion de l'état de l'application

Les **Views** sont ce que l'utilisateur voit et avec lequel il interagit. Elles constituent l'interface permettant d'afficher les données des **Stores**, ainsi que de renvoyer les actions aux **Stores** via le *dispatcher*.

Puisque **Flux** a été conçu aux côtés de **React**, les deux se marient très bien.

Cependant, **Flux** peut fonctionner avec n'importe quel framework front-end

# Gestion de l'état de l'application

Maintenant que nous avons couvert tous les éléments, passons à l'ensemble du processus.

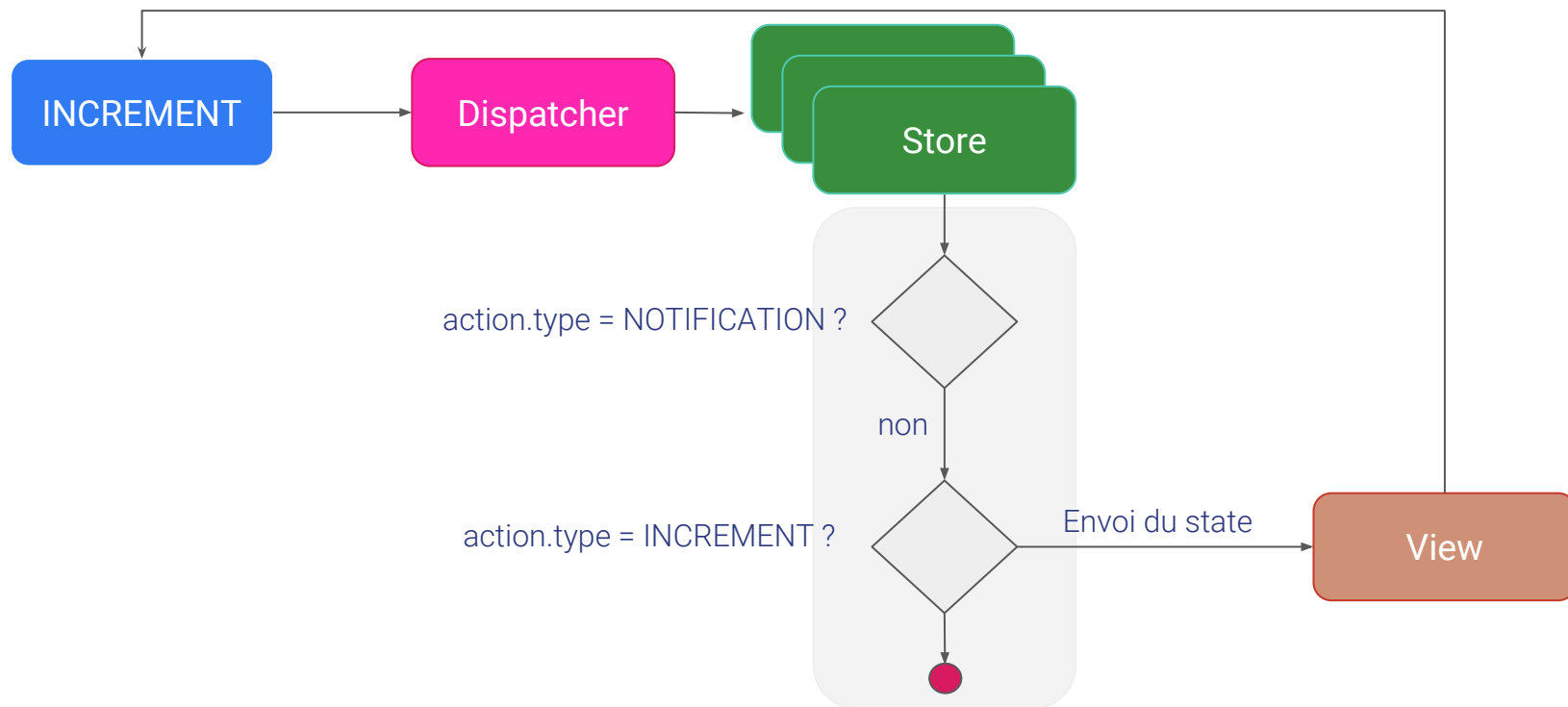
Supposons qu'un utilisateur clique sur un bouton de la **View**, intitulé «Compteur».

Cela envoie une **Action** avec le **type**: 'INCREMENT\_COUNTER' au **dispatcher**. Le **dispatcher** reçoit cette **Action** et l'achemine vers chacun des **Stores** de l'application (il peut y en avoir plusieurs).

Les **Stores** vérifient s'ils doivent réagir aux **Actions** 'INCREMENT\_COUNTER', si oui, ils modifient leur état interne, puis émettent un événement de modification, ce qui entraînera la restitution ultérieure des vues appropriées, affichant probablement une valeur mise à jour pour le compteur.

# Gestion de l'état de l'application

Envoi d'une action au Click() de type INCREMENT



## Redux

# Gestion de l'état de l'application

Co-écrit par Dan Abramov et Andrew Clark en 2015, **Redux** vise à simplifier et à rationaliser de nombreux concepts introduits par **Flux**.

**Redux** est l'implémentation la plus connue et la plus utilisée de **Flux**.

**Redux** et **Flux** sont similaires en ce sens qu'ils soulignent l'importance du flux de données **unidirectionnel** et qu'ils ajustent tous deux l'état par le biais d'**actions** comportant des champs de type. Cependant, ils ont quelques différences qu'il est important de noter:

# Gestion de l'état de l'application

**Redux** supprime le concept de **dispatcher** car il n'a qu'un seul **Store**.

Cela implique qu'il n'y a plus qu'une destination **unique** pour la diffusion de nouvelles **actions**, éliminant ainsi le besoin d'un **dispatcher**.

# Gestion de l'état de l'application

Un **Store** unique, moins complexe

Avec **Redux**, l'ensemble de l'état de votre application est situé dans un **Store** centralisé qui constitue la source **unique** de vérité de l'application.

De plus, les responsabilités du **Store** ont été réduites - il n'est désormais plus que responsable de la logique de mise à jour de l'état de l'application suivant les **Actions**.

La logique de mise à jour a maintenant été déléguée aux...



# Gestion de l'état de l'application

## Reducers

Les **Reducers** sont des **fonctions pures** qui prend en arguments l'état actuel et une action donnée et qui retourne soit l'état actuel non modifié, soit une nouvelle copie de l'état modifiée.

Le mot copie est important ici - **Redux** considère que l'état est **immuable**. Si l'état doit être changé, il n'est pas édité directement. Au lieu de cela, une copie de l'état est créée et le réducteur édite cette copie, puis la renvoie et remplace l'état d'origine par sa copie modifiée.

# Gestion de l'état de l'application

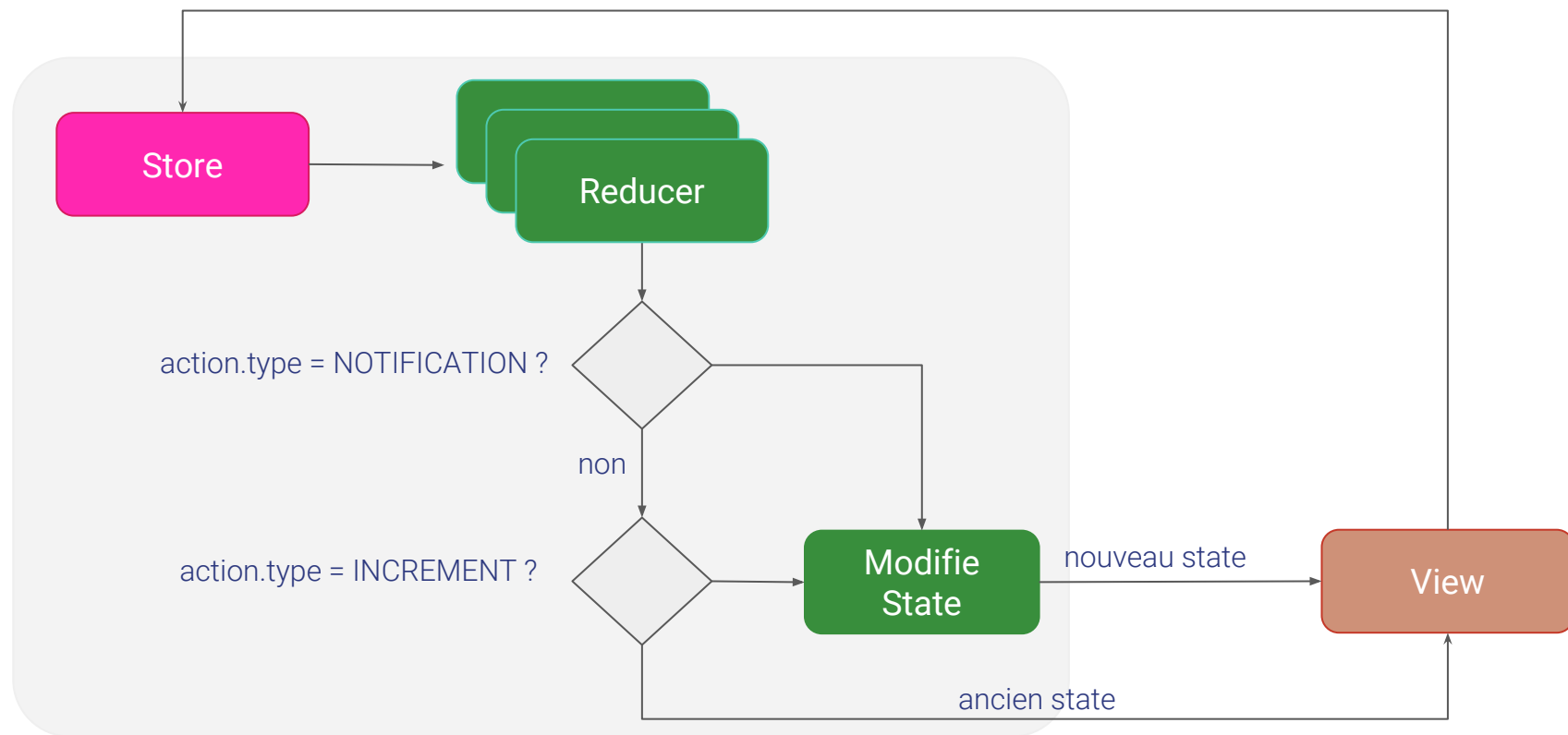
## Exemple d'implémentation d'un **Reducer**

```
const counterReducer = (state = { counter: 0 }, action) => {  
  switch (action.type) {  
    case 'INCREMENT_COUNTER':  
      return {  
        ...state,  
        counter: state.counter + 1  
      }  
    default:  
      return state  
  }  
}
```

L'utilisation de `{...}` permet de faire une copie de state et de retourner un nouvel objet sans modifier state directement

# Gestion de l'état de l'application

Dispatch d'une action au Click() de type INCREMENT



## Mobx

# Gestion de l'état de l'application

## Reducers vs Observables

La grosse différence entre **Mobx** et **Redux** se situent à ce niveau là. Dans **Mobx**, on **observe** le changement du **Store** modifié directement par les **Views**, alors que dans **Redux**, le **Store** est mis à jour par des **Actions** dispatchés par les **Views**.

Par contre, **Flux**, **Redux** et **Mobx** sont similaires car ils sont tous les 3 basés sur un flot de mises à jour **unidirectionnel**.

Les éléments principaux de **Mobx** sont : les **Actions**, les **Observables** et les **Computed**.

# Gestion de l'état de l'application

## Les Actions

Les **Actions** dans **Mobx** sont des fonctions **impures** (contrairement aux **Reducers** qui sont des fonctions **pures**) qui sont chargés de manipuler et de modifier les valeurs du **State** de l'application.

Elles sont implémentées directement dans les **Views** et non pas un **Reducer**.

# Gestion de l'état de l'application

## Les Observables

Les **Observables** dans **Mobx** sont des écouteurs qui sont chargés de surveiller les modifications d'état d'une variable (array, class, objet...).

En reliant un composant **React** avec **@observer**, le composant est automatiquement au courant quand la variable observée a été modifiée.

# Gestion de l'état de l'application

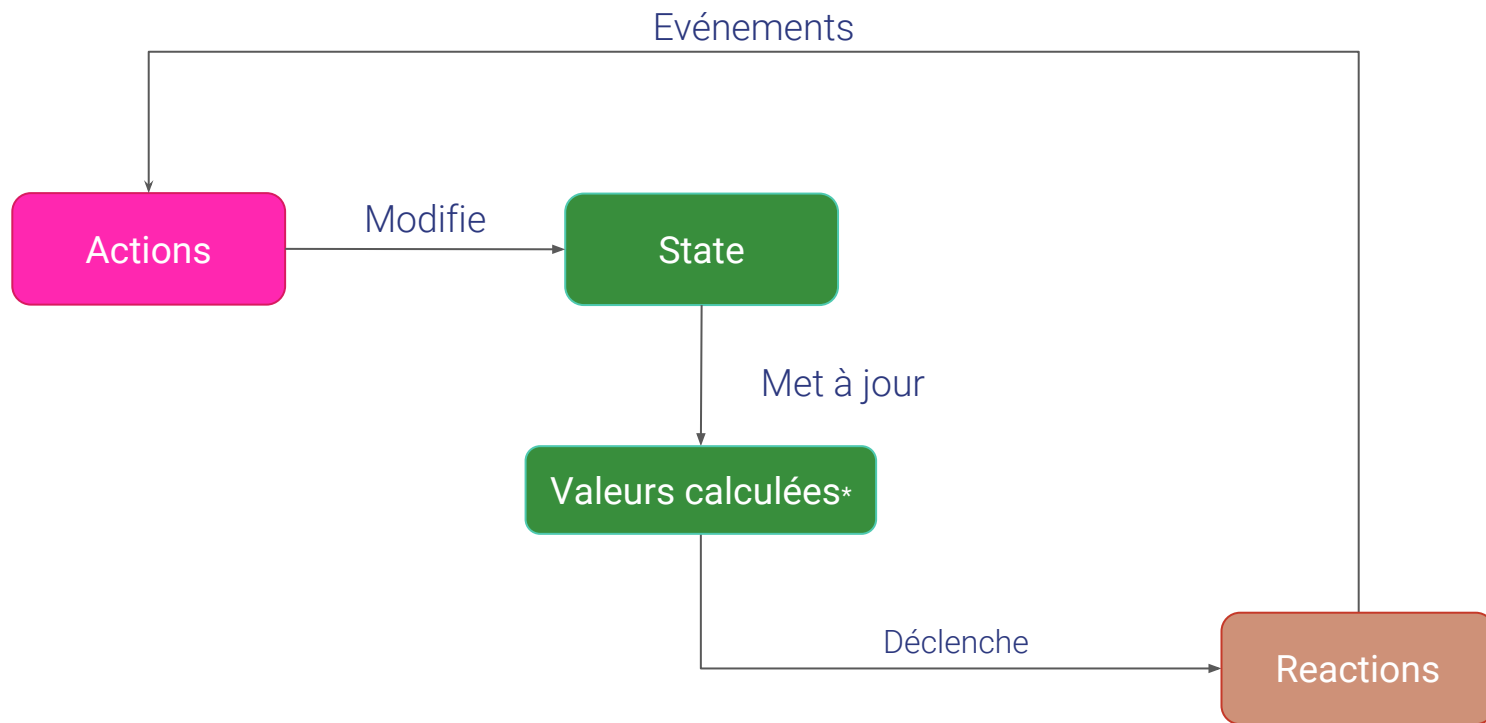
## Les Computed

Les **Computed** dans **Mobx** sont des fonctions **pures** qui produisent des valeurs dérivées du **State** de l'application.

Elles sont automatiquement appelées quand une des valeurs qu'elles utilisent sont modifiées.



# Gestion de l'état de l'application



\*Les valeurs calculées sont des valeurs dérivées du state observé, qui sont automatiquement mises à jour par MobX

# Gestion de l'état de l'application

## Apprentissage par l'exemple.

Améliorons ensemble notre petite calculette avec **Redux**

# Créer un composant React

**Ouvrez `react-formation/exo2/index.js`**

# Gestion de l'état de l'application

Comme expliqué précédemment, on va créer le **Reducer** principal chargé d'écouter *toutes* les **Actions**

./reducers.js

```
import { combineReducers } from 'redux';

function appReducer(state = {}, action) {
  switch (action.type) {
    default:
      return state;
  }
}

export default function createReducer() {
  return combineReducers({
    app: appReducer,
  });
}
```

# Gestion de l'état de l'application

```
function appReducer(state = {}, action) {  
  switch (action.type) {  
    default:  
      return state;  
  }  
}
```

*appReducer* est notre **reducer** de l'application.

Pour notre exemple, un seul suffit, mais on peut en créer autant que nécessaire suivant la taille et la structure de notre projet, via **combineReducers**, qui se charge de regrouper tous les reducers, en un seul.

# Gestion de l'état de l'application

```
import {  
  ADD_OPERATION,  
} from './constants';  
  
export function addOperationAction(operation)  
{  
  return {  
    type: ADD_OPERATION,  
    payload: operation,  
  };  
}
```

Ajoutons de la logique à nos composants en ajoutant une action qui permettra de mettre à jour, dans *appReducer*, la liste des opérations de notre calculatrice.

Celles-ci vont être stockées dans le **Store** et non plus dans le **state** du composant.

Notons l'utilisation d'une constante pour le type de l'action qui facilite le code.

# Gestion de l'état de l'application

```
import { ADD_OPERATION } from './constants';

function appReducer(
  state = { operations: [] },
  action,
) {
  switch (action.type) {
    case ADD_OPERATION:
      return {
        ...state,
        operations: state.operations.concat(action.payload),
      };
    default:
      return state;
  }
}
```

On vérifie le type de l'action, dans le switch case, et pour *ADD\_OPERATION*, on copie **state** et on vient ajouter l'opération à notre liste.

Notons l'utilisation de **concat** qui retourne un **nouveau** tableau. Restons **pure** ;)

# Gestion de l'état de l'application

```
import { createStore, applyMiddleware, compose } from 'redux';
import createReducer from './reducers';

export default function configureStore(initialState = {}) {
  const middlewares = [
    (action) => (action) => {
      console.log('Une action vient de passer dans le store', {
        action });
      return action;
    }
  ];

  const store = createStore(
    createReducer(),
    initialState,
    compose(applyMiddleware(...middlewares))
  );
  return store;
}
```

Beaucoup de magie ici, on ne va pas forcément s'étendre sur le sujet.

Le gros du travail s'effectue dans **createStore** qui est la fonction de redux qui permet de mettre en place le système de ***dispatch***.

On crée un petit middleware qui permettra d'afficher toutes les actions qui arriveront dans le store.



# Gestion de l'état de l'application

```
const mapDispatchToProps = (dispatch) => ({
  addOperation: (c) => dispatch(addOperationAction(c)),
});

const mapStateToProps = (state) => (console.log({state})), {
  operations: state.app.operations,
});

const ReduxCalcullette = connect(
  mapStateToProps,
  mapDispatchToProps
)(Calcullette);
```

*mapDispatchToProps* permet d'ajouter des **props** connecté au **Store** via la méthode **dispatch** de **Redux**.

*mapStateToProps* permet de piocher dans le **Store** et d'ajouter ce que l'on a été cherché en tant que **props** au composant.

On **connect** notre composant en le passant au **HOC connect()** de **Redux**

# Créer un composant React

On peut alors utiliser ***addOperation*** comme n'importe quelle **props** comme vu précédemment quand on modifiait le **state** de notre composant.

```
{[7, 8, 9, 4, 5, 6, 1, 2, 3, 0, '.'].map((c) => <Case key={c} onClick={() => this.addOperation(c)}>{c}</Case>) }
```

Notez l'utilisation de la **closure** dans le **onClick** pour passer la valeur de **c** à **addOperation()**

# Créer un composant React

Maintenant, à vous d'implémenter ***calculate***, en utilisant **Redux**, pour afficher et calculer le résultat

## Router

# Créer un composant React

**Ajoutons react-router**

```
npm install --save react-router-dom
```

# Gestion de l'état de l'application

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

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

const AppRouter = () => (
  <Router>
    <div>
      <Link to="/">Home</Link>
      <Link to="/about/">About</Link>
      <Route path="/" exact component={Index} />
      <Route path="/about/" component={About} />
    </div>
  </Router>
);

export default AppRouter;
```

Pour utiliser le **router**, rien de plus simple.

Le composant **Router** se charge automatiquement, par rapport au *path* donné à **Route** d'afficher ou non le composant passé dans la **props component**.

**Link** est le composant à utiliser à la place de l'élément html `<a>` pour naviguer dans l'application.

# Créer un composant React

Maintenant, à vous d'implémenter *une nouvelle page*, en utilisant **Router**, pour afficher la liste des anciens résultats de la calculatrice