

React JS

# SOMMAIRE



**ECMA Script 6**



**Type Script**



**React**

# Avant de commencer



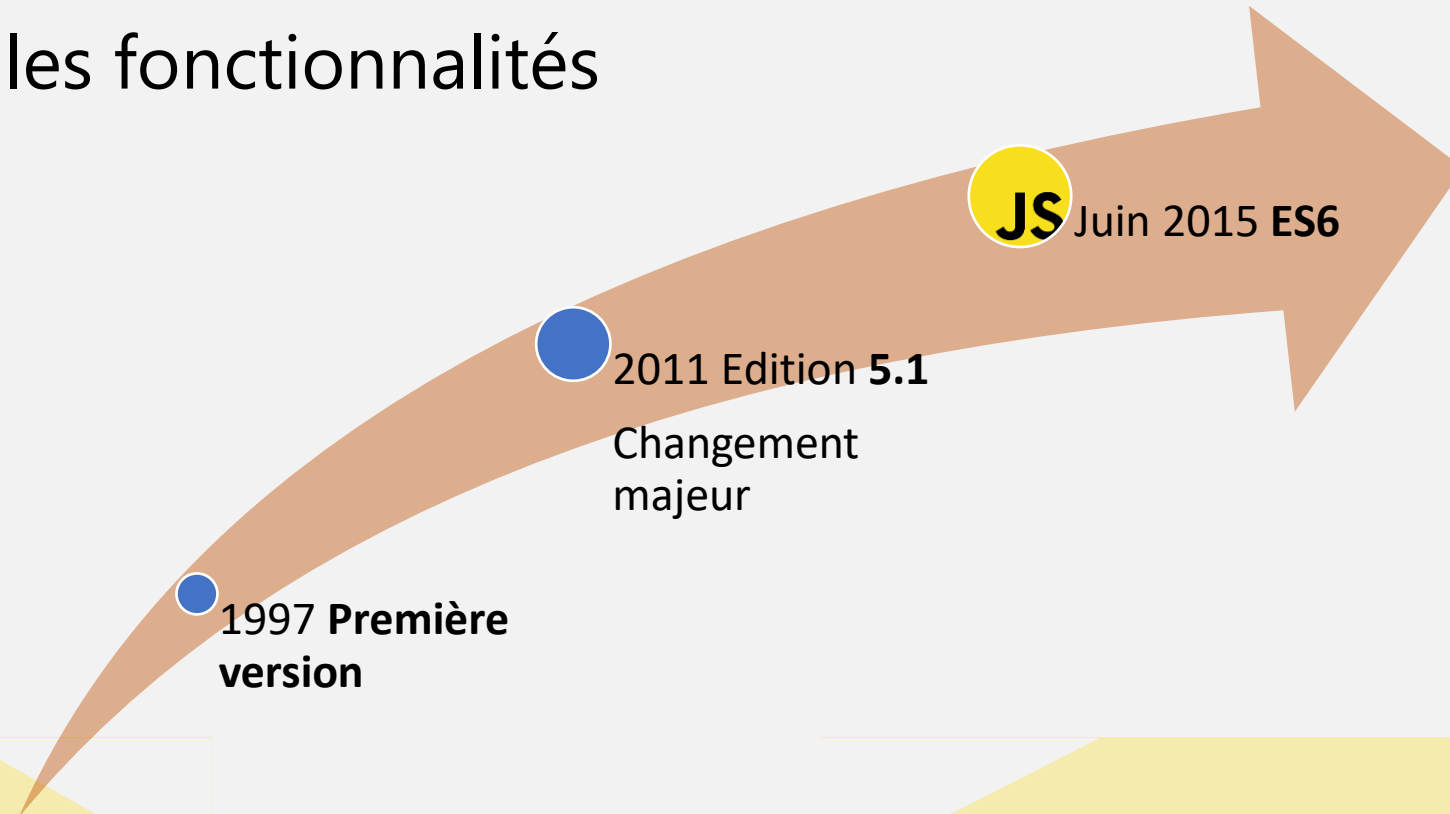


# ECMA Script 6

*Le JavaScript de demain, aujourd'hui*



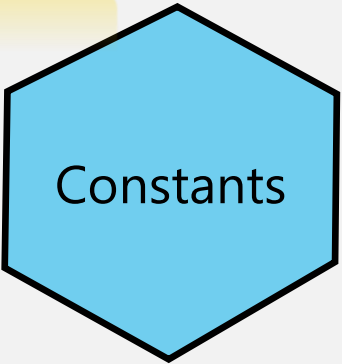
- Aussi appelé ES6 ou ECMA Script 2015
- Offre de nouvelles fonctionnalités





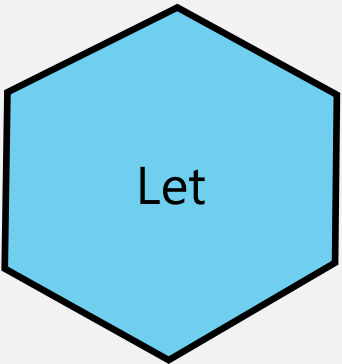
# ECMA Script 6

*Le JavaScript de demain, aujourd'hui*



- Valeur immuable. Impossible de réassigner une nouvelle valeur.
- Utilisation du mot clé **const**

```
const iamConstante = "Variable  
Immuable";  
iamConstante = "problèmes";
```



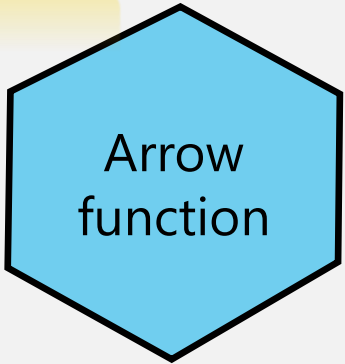
- Valeur muable. Peut être changé à tout moment
- Utilisation du mot clé **let**

```
let variabeTest = "Variable de  
test";  
variabeTest = "Pas de problème";
```



# ECMA Script 6

*Le JavaScript de demain, aujourd'hui*



- Définition de fonction plus courte.
- Ne redéfinit pas le contexte courant (**this**).

**{ } ES6**

```
const firstFunction = value => value + 1;
```

```
const secondFunction = value => ({value1: value, value2: value + 1});
```

```
const thirdFunction = (value1, value2) => value1 + value2;
```

**{ } ES5**

```
var firstFunctionEs5 = function (value) {  
  return value + 1; };
```

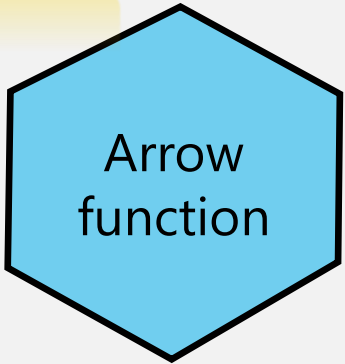
```
var secondFunctionEs5 = function (value) {  
  return { value1: value, value2: value + 1 }; };
```

```
var thirdFunctionEs5 = function (value1, value2) {  
  return value1 + value2; };
```



# ECMA Script 6

*Le JavaScript de demain, aujourd'hui*



**{ } ES6**

```
this.nums.forEach((v) => {  
  if (v % 5 === 0)  
    this.fives.push(v)  
})
```

**{ } ES5**

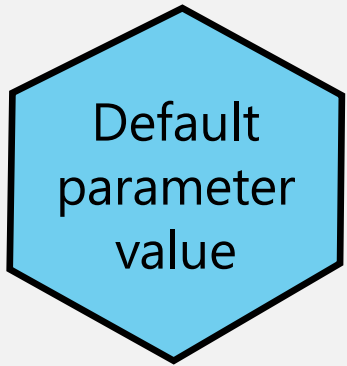
```
var self = this;  
this.nums.forEach(function (v) {  
  if (v % 5 === 0)  
    self.fives.push(v);  
});
```

Le contexte parent est accessible dans une arrow function. Ce dernier n'est pas redéfini.



# ECMA Script 6

*Le JavaScript de demain, aujourd'hui*



- Initialisation des paramètres dans la définition de la fonction.

**{ } ES6**

```
function functionTest (param1, param2  
= 8, param3 = 16) {  
    return x + y + z  
}  
f(1) === 25
```

**{ } ES5**

```
function functionTest (param1, param2,  
param3) {  
    if (y === undefined)  
        y = 8;  
    if (z === undefined)  
        z = 16;  
    return x + y + z;  
};  
f(1) === 25;
```





# ECMA Script 6

*Le JavaScript de demain, aujourd'hui*



- Opérateur de décomposition.

**{ } ES6**

```
const params = [ "hello", true, 7 ]  
const other = [ 1, 2, ...params ]  
// [ 1, 2, "hello", true, 7 ]
```

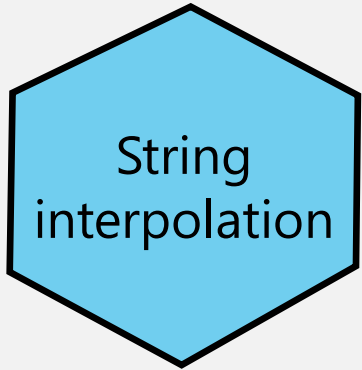
**{ } ES5**

```
var params = [ "hello", true, 7 ];  
var other = [ 1, 2 ].concat(params);  
// [ 1, 2, "hello", true, 7 ]
```



# ECMA Script 6

*Le JavaScript de demain, aujourd'hui*



- Concaténation de chaîne simple ou multiple.
- Plus rapide et concis.

**{ } ES6**

```
const customer = { name: "Foo" };  
const card = { amount: 7, product: "Bar",  
unitprice: 42 };  
const message = `Hello ${customer.name}`;
```

**{ } ES5**

```
var customer = { name: "Foo" };  
var card = { amount: 7, product:  
"Bar", unitprice: 42 };  
var message = "Hello " +  
customer.name ;
```



# ECMA Script 6

*Le JavaScript de demain, aujourd'hui*



- Syntaxe de création d'objet réduite et rapide.

**{ } ES6**

```
let x = 0, y = 0;  
obj = { x, y };
```

**{ } ES5**

```
var x = 0, y = 0;  
obj = { x: x, y: y  
};
```



# ECMA Script 6

*Le JavaScript de demain, aujourd'hui*



- Ajout d'une propriété à la volée via le contexte courant.

## { } ES6

```
let obj = {  
  foo: "bar",  
  [ "baz" ]: 42  
}
```

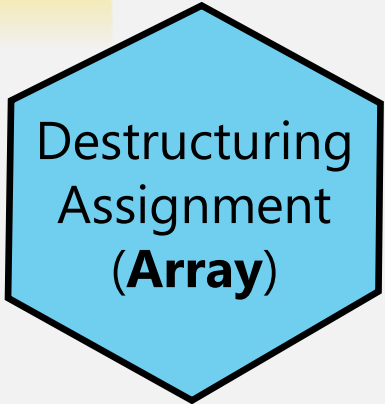
## { } ES5

```
var obj = {  
  foo: "bar"  
};  
obj[ "baz" ] = 42;
```



# ECMA Script 6

*Le JavaScript de demain, aujourd'hui*



- Assigner des variables provenant d'un objet ou tableau reposant sur leur structure.
- Très pratique pour la récupération de certaines variables ou constante par exemple.

**{ } ES6**

```
const list = [ 1, 2, 3 ]  
const [ a, b , c ] = list;
```

**{ } ES5**

```
var list = [ 1, 2, 3 ];  
var a = list[0], b = list[1], c= list[2]
```



# ECMA Script 6

*Le JavaScript de demain, aujourd'hui*



Destructuring  
Assignment  
(**Object**)

Destructuring  
Assignment  
(**Object**)  
**\*deep**

- Assigner des variables provenant d'un objet ou tableau reposant sur leur structure.
- La récupération est similaire à la précédente.

**{ } ES6**

```
var { op: a, lhs: { op: b }, rhs: c } = getASTNode();
```

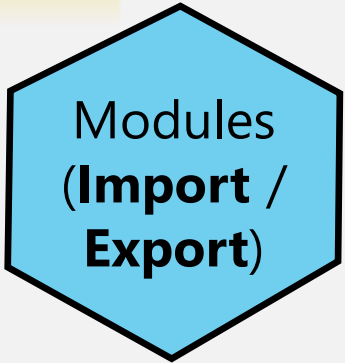
**{ } ES5**

```
var tmp = getASTNode();  
var a = tmp.op;  
var b = tmp.lhs.op;  
var c = tmp.rhs;
```



# ECMA Script 6

*Le JavaScript de demain, aujourd'hui*



- Permet de cloisonner des parcelles de code sous un espace de nom particulier.
  - Importer ou exporter des valeurs ou composants nommés ou non.
- **Deux types** d'export :

## Export nommé

- Utile pour exporter plusieurs valeurs.
- Obligation d'utiliser le même nom de l'objet exporté lors de l'importation

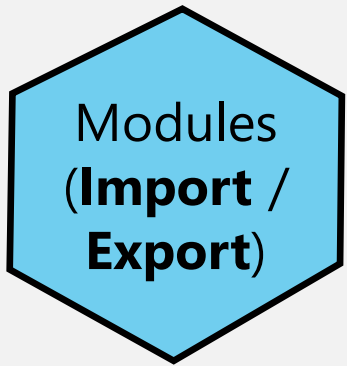
## Export par défaut

- Utile pour exporter une seule valeur.
- Peut avoir n'importe quel nom lors de l'importation



# ECMA Script 6

*Le JavaScript de demain, aujourd'hui*



## { } **Export** et **Import** nommé

```
export const myFunctionToExport = ()=> {};  
import {myFunctionToExport} from  
  './myFile';
```

## { } **Export** et **Import** par défaut

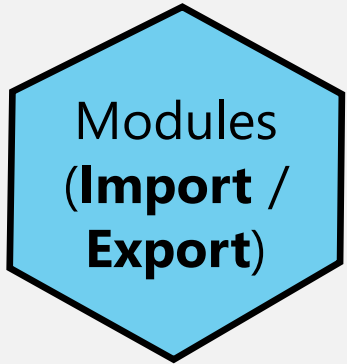
```
export default function (){};  
import myfunction from './myfile';
```





# ECMA Script 6

*Le JavaScript de demain, aujourd'hui*



- Importer **l'intégralité** d'un module.

```
import * as myModule from  
'/myModule.js';
```

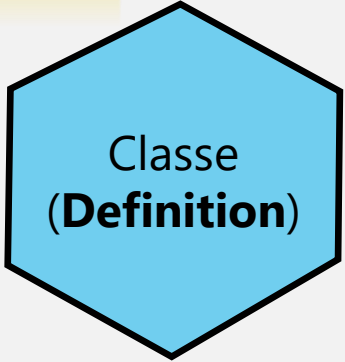
- Importer un objet exporté avec un **alias**.

```
import {nameOfObjectIReallyReallyBig as shortName} from '/myModule.js';
```



# ECMA Script 6

*Le JavaScript de demain, aujourd'hui*



- **Sucre syntaxique**\* plus pratique et lisible permettant de définir une classe comme en POO. Utilisation de « **class** ».

*\*Les sucres syntaxique sont des expressions permettant de faciliter la vie du développeur. Utilisé dans un langage de programmation.*

**{ } ES6**

```
class User {  
  constructor (firstName, lastName) {  
    this.firstName = firstName;  
    this.lastName = lastName;  
    this.printUser(firstName, lastName);  
  }  
  
  printUser (firstName, lastName) {  
    console.log(`Hello my name is  
    ${firstName} ${lastName}`);  
  }  
}
```

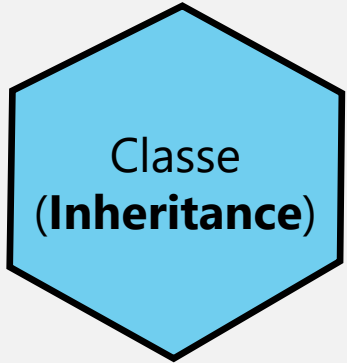
**{ } ES5**

```
var User = function(firstName, lastName){  
  this.firstName = firstName;  
  this.lastName = lastName;  
}  
  
User.prototype.printUser = function(){  
  console.log('Hello my name is' + this.firstName + ' '  
  + this.lastName);  
};
```



# ECMA Script 6

*Le JavaScript de demain, aujourd'hui*



- Utilisation du sucre « **extends** ».

**{ } ES6**

```
class Human {  
  constructor(age){  
    this.age = age;  
  }  
}  
class User extends Human {.....
```

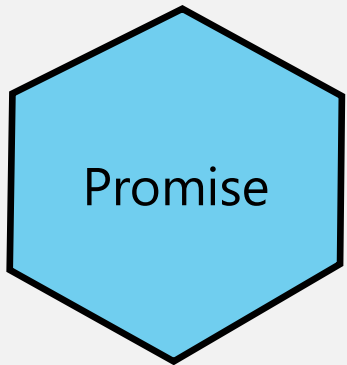
**{ } ES5**

```
var User = function(firstName, lastName, age){  
  this.firstName = firstName;  
  this.lastName = lastName;  
  Human.call(this, age);  
}  
User.prototype =  
  Object.create(Human.prototype);  
User.prototype.constructor = User;
```



# ECMA Script 6

*Le JavaScript de demain, aujourd'hui*



- Objet permettant la gestion des traitements asynchrones.
- Une « promesse », représente une valeur qui peut être disponible, maintenant, dans le futur, voir jamais.
- Utilisation de l'objet **Promise**

```
const myFirstPromise = new Promise((resolve,  
reject)=>{  
  //Si la promesse réussit  
  //resolve(valeur de retour)  
  
  //Si la promesse est rejetée  
  //reject(valeur de retour)  
});
```



- Une fonction peut retourner une promesse. C'est même une très bonne pratique
- Utilisation de **then()** en cas de réussite et **catch()** en cas d'échec

```
const myFirstPromiseReturnFunction = ()=>
{
return new Promise((resolve, reject)=>{
// resolve ou reject ....
});
}
myFirstPromiseReturnFunction.then((value)=>{
}).catch(()=>"Promesse rompue");
```



# ECMA Script 6

*Le JavaScript de demain, aujourd'hui*



- Possibilité de combiner plusieurs promesses.
- Il est possible d'attendre la fin de tous les traitements avant d'en exécuter un nouveau.
- Utilisation de **Promise.all()**

```
function fetchAsync (url, timeout, onData, onError) { ... }
```

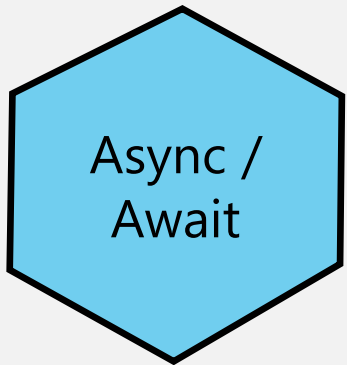
```
let fetchPromised = (url, timeout) => {  
  return new Promise((resolve, reject) => { fetchAsync(url,  
    timeout, resolve, reject) })  
}
```

```
Promise.all(  
  [ fetchPromised("http://backend/foo.txt", 500),  
    fetchPromised("http://backend/bar.txt", 500),  
    fetchPromised("http://backend/baz.txt", 500) ]  
)  
  .then((data) => {  
    let [ foo, bar, baz ] = data console.log(`success: foo=${foo}  
    bar=${bar} baz=${baz}`) }, (err) => { console.log(`error: ${err}`)  
  })
```



# ECMA Script 6

*Le JavaScript de demain, aujourd'hui*



- Utilisation de ce sucre permettant la lecture et la gestion plus facile des méthodes asynchrones.

```
const functionAsync = async()  
=>{  
  
} ;
```

```
const callFunctionAsync = await  
functionAsync();
```



# ECMA Script 6

*Le JavaScript de demain, aujourd'hui*



## □ Résumé

- De nombreuses améliorations et confort du langage
- **Const** et **let** sont block scopés à la différence du var qui est fonction scopé. L'assignation fonctionne par référence et non par valeur.
- **Arrow function**, ne dispose pas de son propre contexte.





# ECMA Script 6

*Le JavaScript de demain, aujourd'hui*



## □ Résumé

- **Default parameter**, utile pour passer des valeurs par défaut au paramètre de fonction.
- **Rest and spread operator**
  - Rest : Assemble plusieurs valeurs dans un tableau.
  - Spread : Eclater un tableau en une liste finie de valeurs.
- **Property Shortand** : pas obliger de réécrire les propriétés.



# ECMA Script 6

*Le JavaScript de demain, aujourd'hui*



## □ Résumé

- **Destructuring** sur les tableaux, objet et nested objet
- **Module** : cloisonné le code.
- **Promise** : Gestion asynchrone.

# A vous de jouer !





# Avant de commencer



- Tous les développements doivent être effectués en mode **"TDD"**
- Utilisation du module **Jest** pour la mise en place des tests unitaires.



# A vous de jouer !



`{}` **Etape 1 :** `https://github.com/Gparquet/EcmaScript6Experience.git`

`{}` **Etape 2 :** Aller dans le dossier puis taper `npm install`

`{}` **Etape 3 :** Dans le dossier **contact**, créer deux fichier - **contact.service.js**. (Ce dernier contiendra **toutes les méthodes** permettant de parser le fichier **contact.json**).

- **contact.service.spec.js**. (Contiendra tous les tests relatifs au fichier **contact.service**).

# A vous de jouer !



## `{}` Etape 4 : Enrichissement du service contact

- Création d'une méthode qui retourne les contacts par nom et prénom. Si l'utilisateur n'est pas trouvé renvoyer un message.

- ☐ *Utilisation des arrow function*
- ☐ *littéral expression*
- ☐ *forEach itérateur, module*



# A vous de jouer !



## `{}` Etape 5 :

- Création d'une méthode qui retourne les contacts par leur identifiant et nom. Si l'utilisateur n'est pas trouvé renvoyer un message.
  - ☐ *Utilisation des arrow function*
  - ☐ *Littéral expression*
  - ☐ *forEach itérateur, module*



# A vous de jouer !



## { } Etape 6 :

- Création d'une méthode permettant d'ajouter un nouveau contact.
  - ❑ *Utilisation des arrow function*
  - ❑ *littéral expression*
  - ❑ *Default parameter value*
  - ❑ *Destructuring, module*





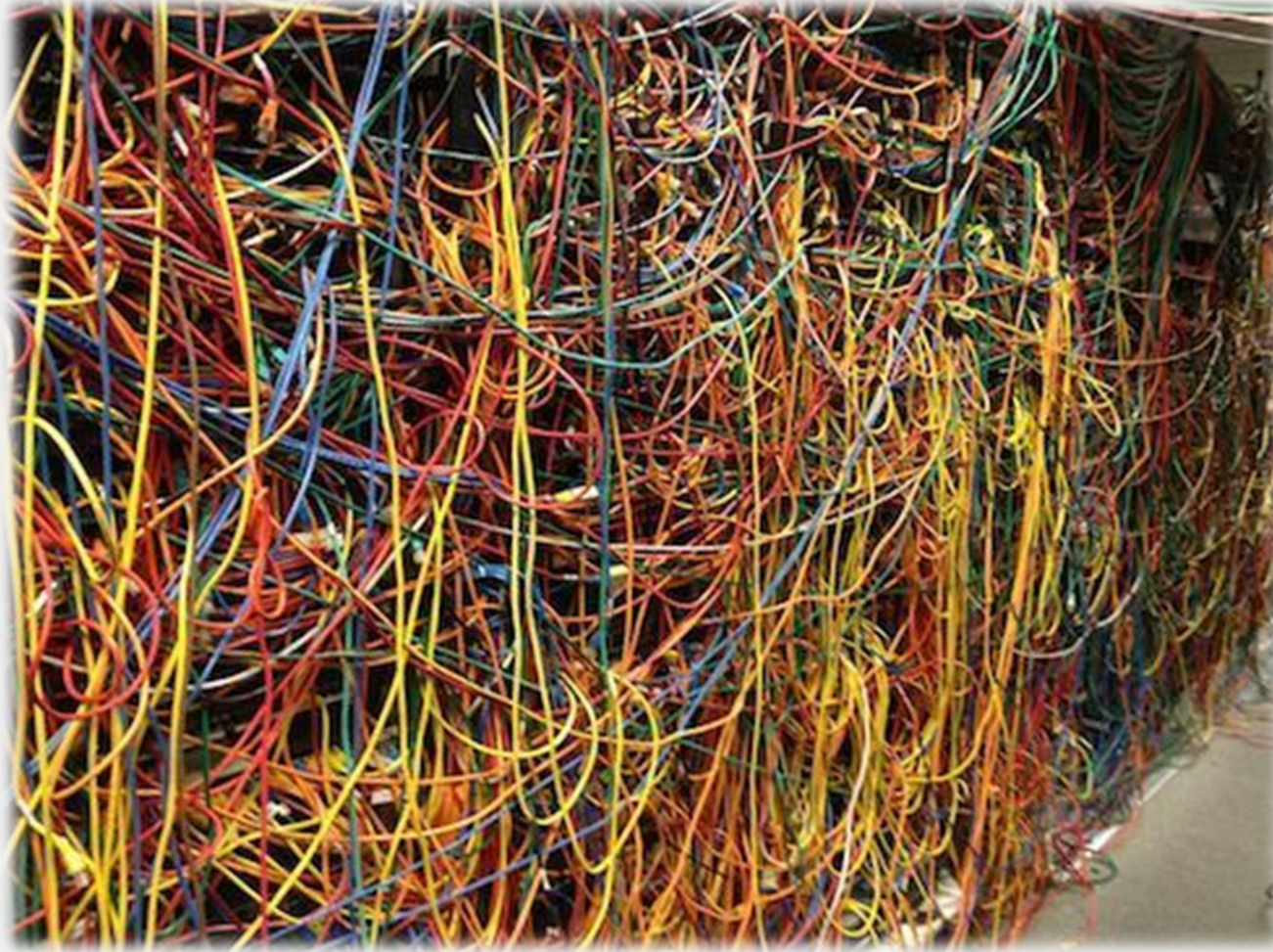
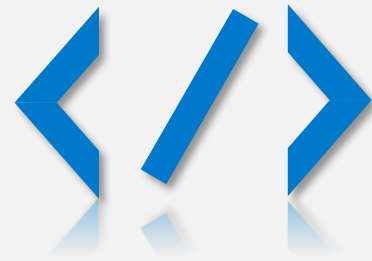
# TypeScript



- Langage de programmation open source développé par Microsoft en 2012
- Intègre la notion de typage et de programmation d'orientée objet

TS

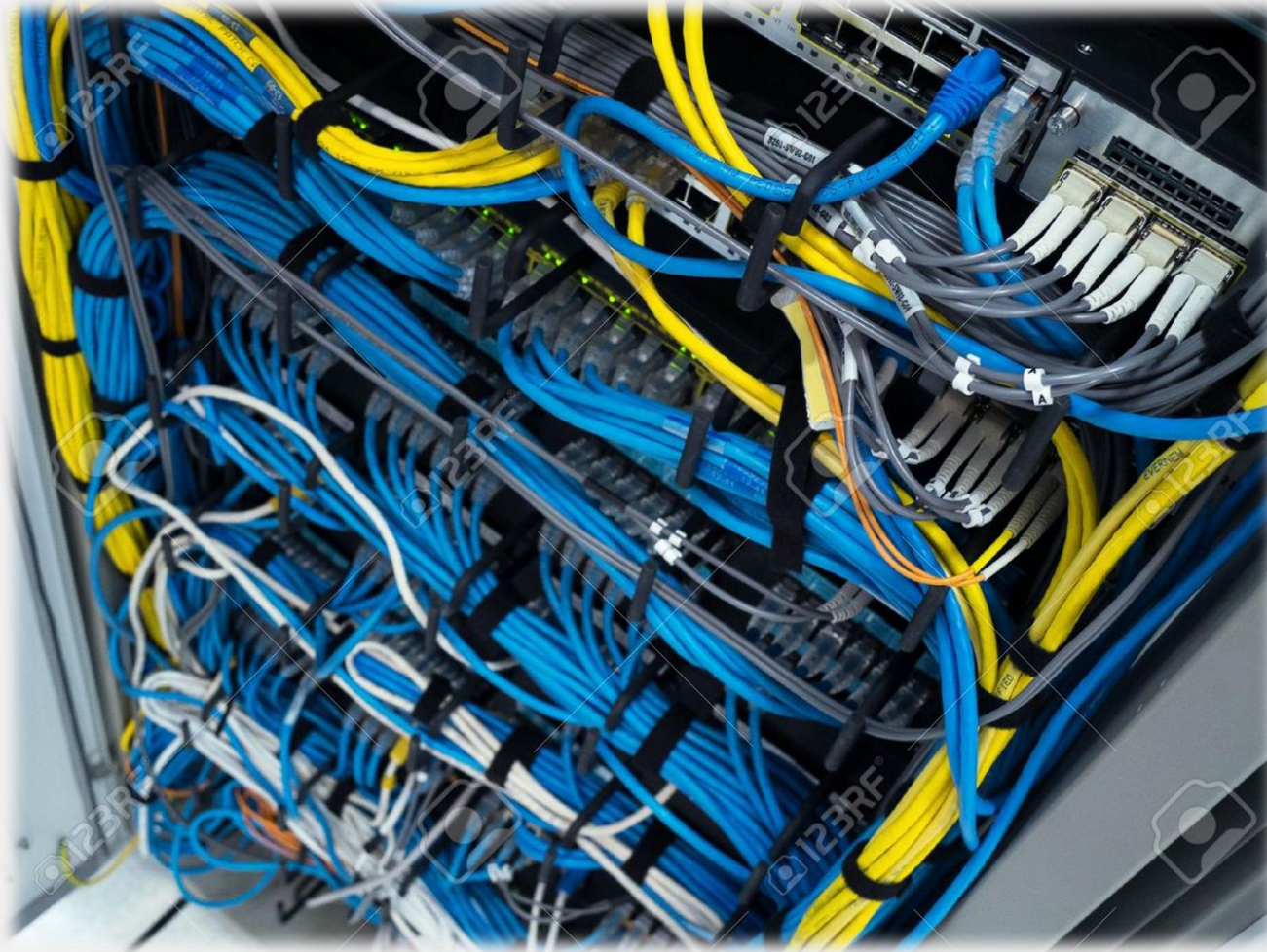
# Pourquoi utiliser TypeScript ?





TS

# Pourquoi utiliser TypeScript ?





# JavaScript rappel

*Le bon, la brut et le truand ....*



## JavaScript = Typage dynamique

### ☐ Le Bon

- Les variables peuvent contenir tout type d'objet
- Les types sont déterminés à la volés

### ☐ La Brute

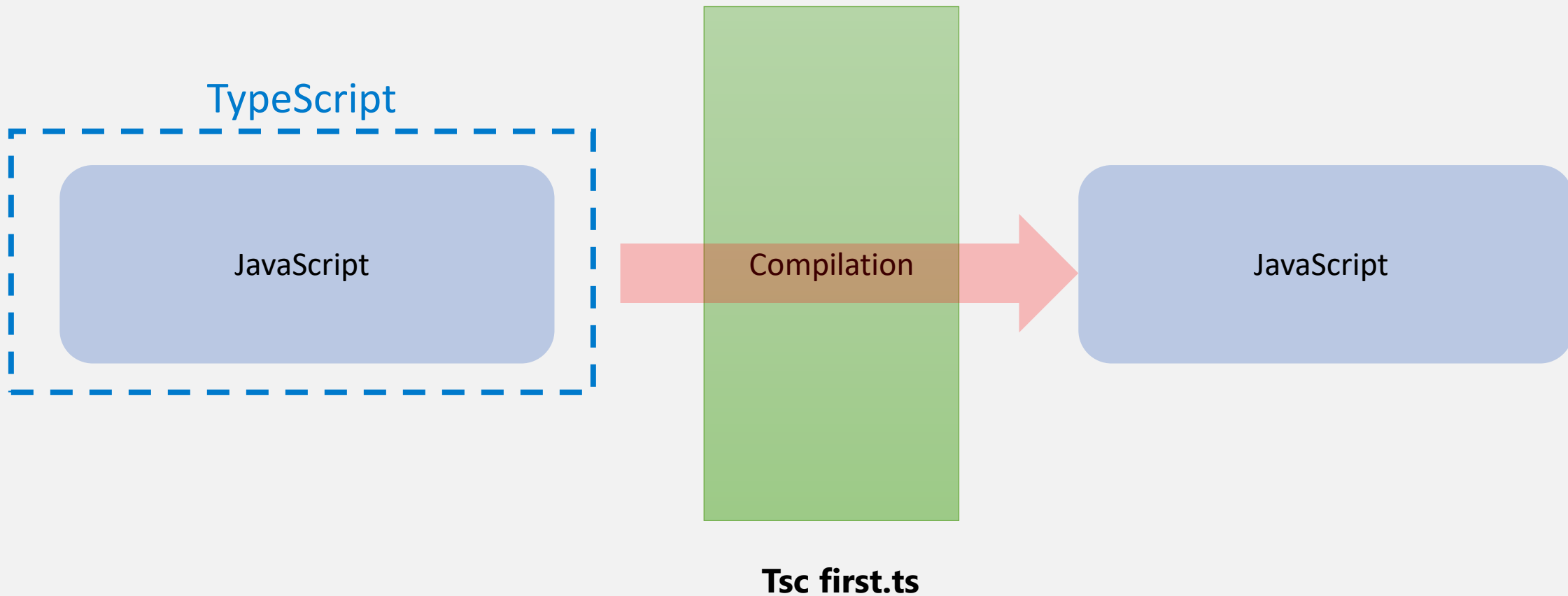
- Certaines applications JavaScript peuvent atteindre plus de 10 000 lignes de code.

### ☐ Le Truand

- Difficile de garantir le type de la variable
- Tous les développeur n'utilise pas === pour la comparaison



# TypeScript et ses Fonctionnalités





TS

# TypeScript et ses Fonctionnalités



Support le  
JavaScript  
Standard

Static  
Typing

Encapsulati  
on via les  
classes et  
modules

Support les  
constructeu  
rs,  
propriétés  
et fonctions

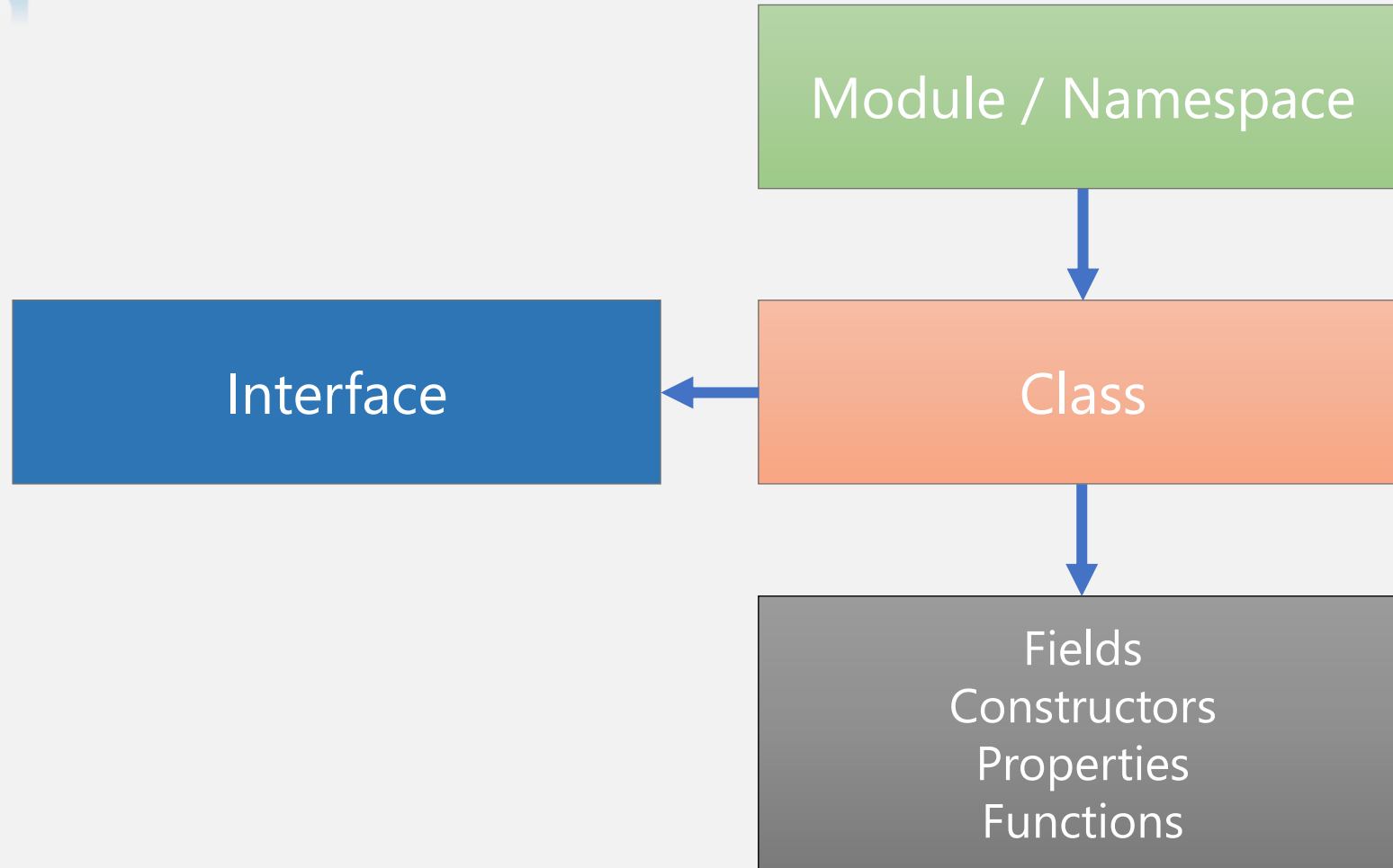
Définition  
d'interface

Support  
des  
fonctions  
lambda

Intellisense  
and  
vérification  
de la  
syntaxe



# Code hiérarchie





# Grammaire, Déclaration, Annotation



Grammaire : Inférence de type

```
const num = 2;
```

Grammaire : Annotation de type

```
const num: number = 2;
```





# Grammaire, Déclaration, Annotation



```
const any;
```

Le type peut être de n'importe quel type

```
const num: number;
```

Annotation de type

```
const num: number = 2;
```

Annotation de type et initialisation avec variable



# Enums



- Possibilité de définir des énumérateurs de type **numérique** ou **chaîne de caractère** ou **hétérogène**.
- Pour rappel, un enum est un ensemble de constantes nommées.
- Utilisation avec le mot clé *enum*



# Fonctions



- Il est possible de :
  - ✓ *Typer les paramètres*
  - ✓ *Typer la fonction complète*
  - ✓ *Rendre des paramètres optionnels*
  - ✓ *D'ajouter des valeurs par défaut au paramètre (également faisable en ES6).*



# Classes



- Tous ce que nous faisons avec un langage orienté objet (C#, Java ...), est faisable avec TypeScript :
  - ✓ *Instanciación d'objet*
  - ✓ *Héritage de classe / interface*
  - ✓ *Polymorphisme*
  - ✓ *Public, privé, et modificateur protégé*



# Interfaces



Avec TypeScript, les interfaces remplissent le rôle de :

- **Nommer les types**
- **Définir des contrats** dans votre code ainsi que des contrats avec du code en dehors de votre projet



# LIVE CODING



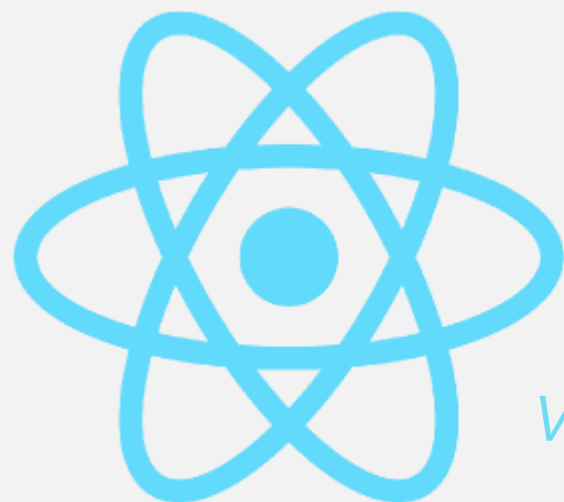
*Les types*

*Enum*

*Fonctions*

*Classes*

*Interfaces*



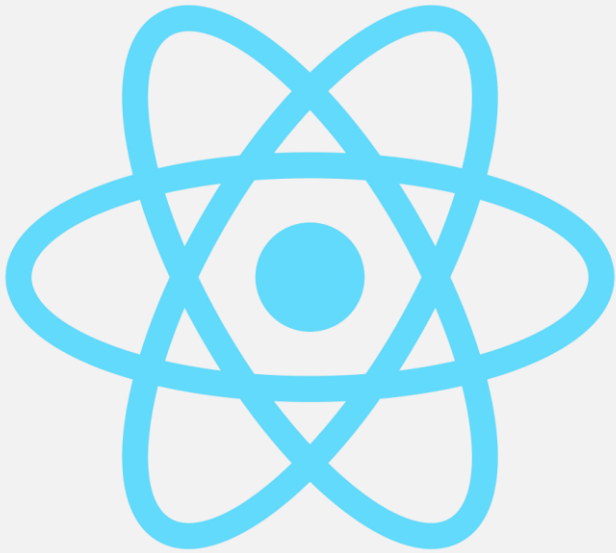
# React

*Vers un monde meilleur*



# ReactJS

*Vers un monde meilleur !*



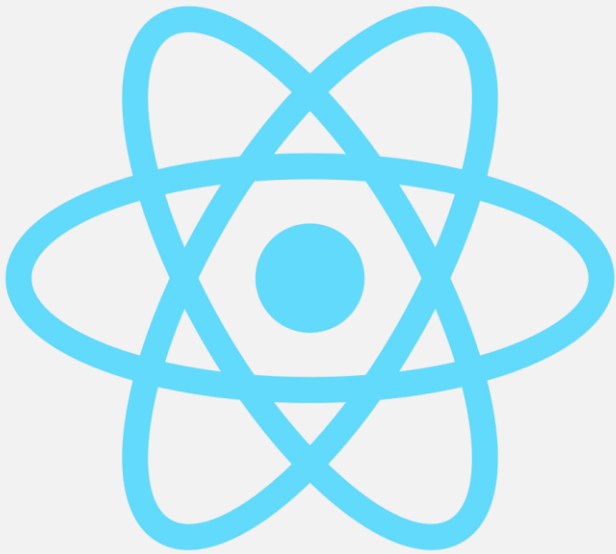
- Bibliothèque, non un Framework !
- Gestion uniquement de la vue (interface de l'application).
- Créé par Facebook depuis 2013.
- Pourquoi ????
  - Répondre au problème de l'application mono-page.





# ReactJS

*Vers un monde meilleur !*



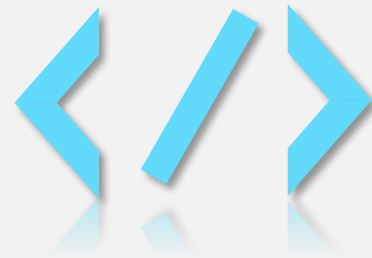
- ReactJS c'est aussi .....
  - ❑ Simplicité
  - ❑ Rapidité
  - ❑ Flexibilité
- Peut également être couplé avec un Framework tel que Angular par exemple.





# ReactJS

*Vers un monde meilleur !*



- ReactJS manipule un **DOM virtuel** et non celui du navigateur !!
- Tout est en composant en ReactJS. Ce dernier ne crée pas de HTML, mais une représentation sous forme d'objet et de nœuds de ce à quoi le HTML doit ressembler.

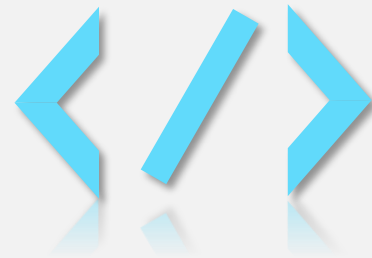
□ Pour rappel :

- ✓ **DOM** = **D**ocument **O**bject **M**odel.
- ✓ Interface entre le code et le HTML créé.
- ✓ Représentation à un instant T la page visible par l'utilisateur.



# ReactJS

*Vers un monde meilleur !*



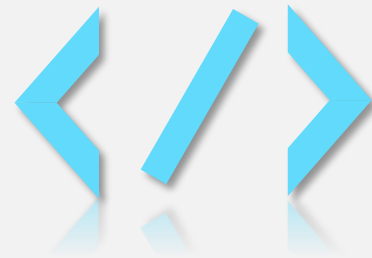
## ❑ Avant l'arrivée de ReactJS, les limitations :

- ✓ Performance réduite (*très visible sur les produits nomades*) lors d'une simple modification sur un DOM complexe.
- ✓ Suivre les changements d'état encore plus difficile. AngularJS en est l'exemple parfait avec sa méthode « **Dirty Checking** ».
  - ❑ Tous les objets sont surveillés en permanence, y compris ceux qui ne changent pas d'état.



# ReactJS

*Vers un monde meilleur !*



- **DOM virtuel**

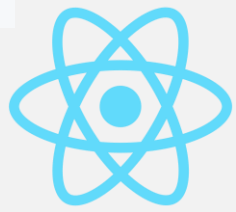
- C'est quoi ?

*(virtual-dom)*

- Outil permettant la représentation du DOM actuel complètement décorrélé de son homologue.
    - Gestion des actions minimales à exécutées pour mixer les changements du DOM virtuel avec celui utilisé par le navigateur.

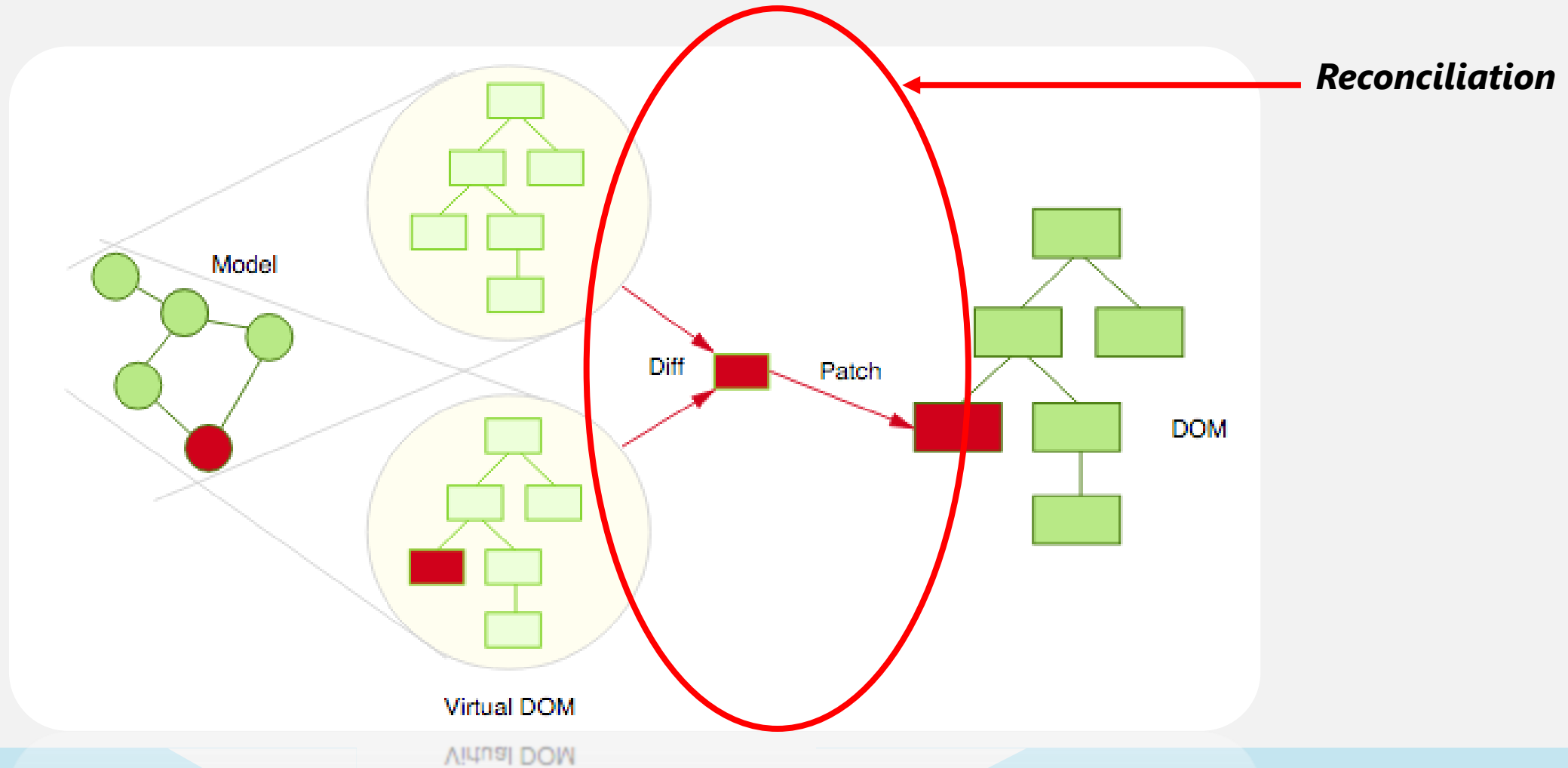
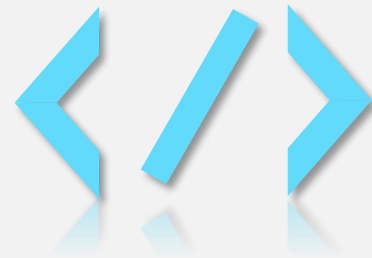
- Intérêt ?

- ✓ Gestion de la comparaison, modification du DOM coté JavaScript.
    - ✓ Application plus rapide !



# ReactJS

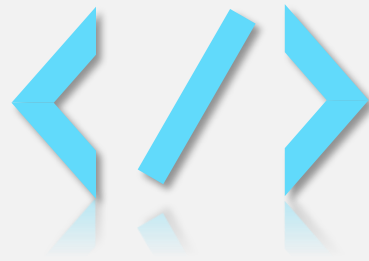
*Vers un monde meilleur !*





# ReactJS

*Vers un monde meilleur !*



- **Deux possibilités de créer une application React**

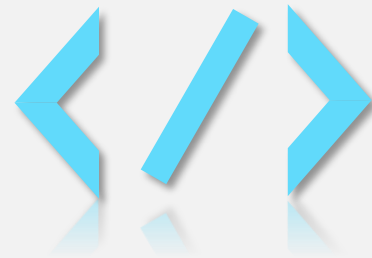
- ☐ CSR (**C**lient **S**ide **R**endering)

- ☐ SSR (**S**erver **S**ide **R**endering)



# ReactJS

*Vers un monde meilleur !*



## ❖ Client Side Rendering



Application interactive et dynamique

Une fois JS chargé tout est exécuté côté server.

Il n'y a plus de chargement de contenu à chaque chargement de page

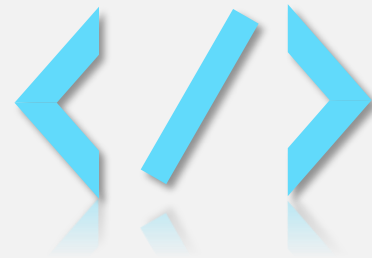
Optimisation compliquée du référencement ou SEO (Search Engine Optimization)

Temps de rechargement initial plus élevé

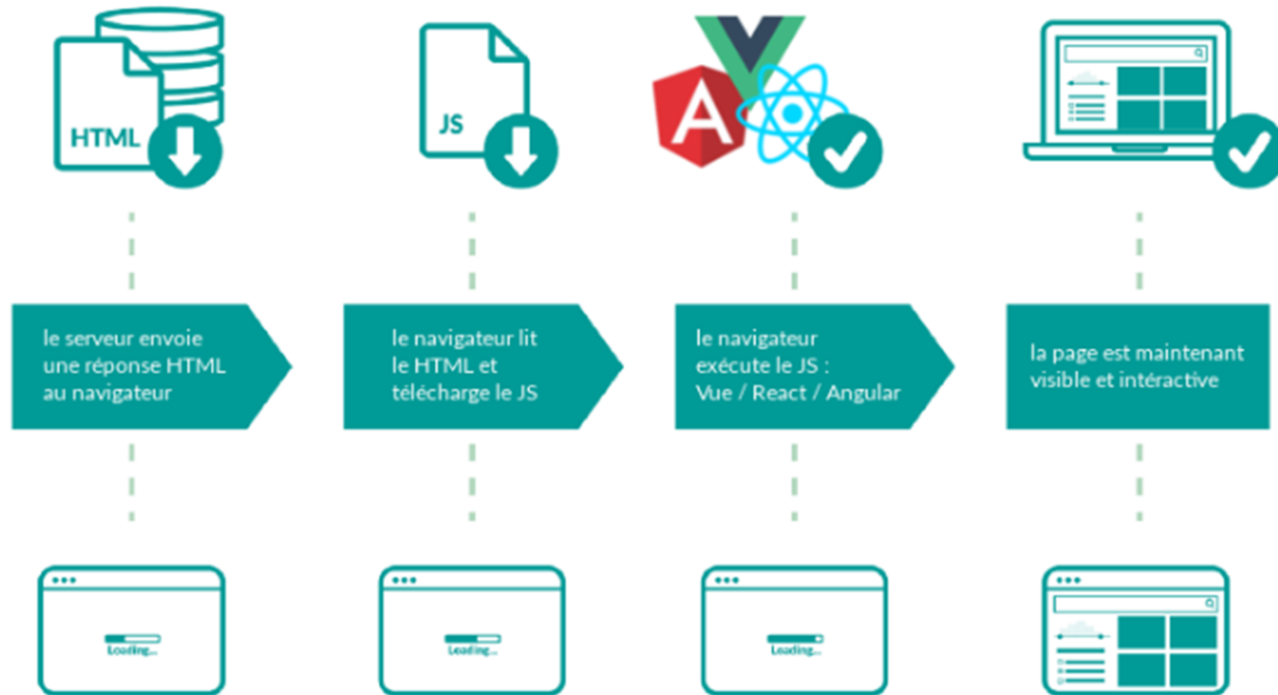


# ReactJS

*Vers un monde meilleur !*



## CSR

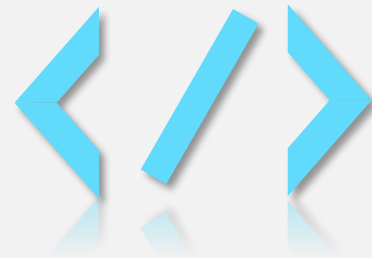






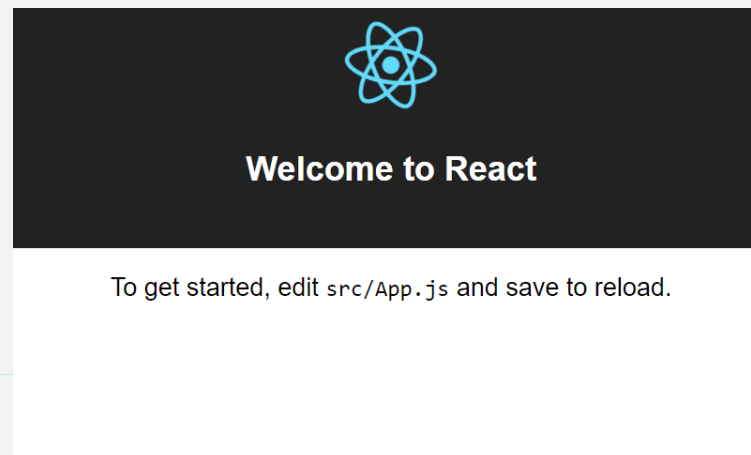
# ReactJS

*Vers un monde meilleur !*



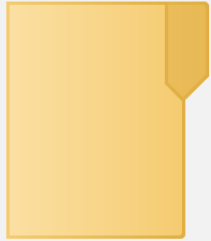
## □ Create-react-app

- Taper la commande **npx create-react-app formation.react.csr**
- Aller dans le dossier puis aller dans le dossier **formation.react.csr** puis taper **npm start**



# ReactJS

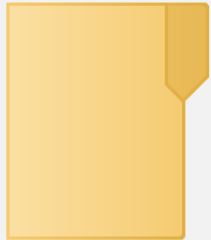
*Vers un monde meilleur !*



## Public



**Index.html** : page de démarrage de l'application



## Src



**App.js** : composant principal

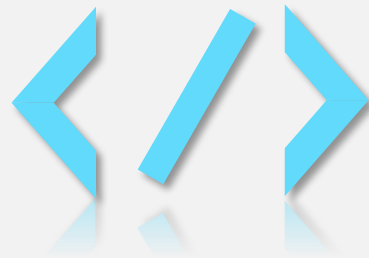


**Index.js** : point d'entrée principal pour le rendu des composants



# ReactJS

*Vers un monde meilleur !*



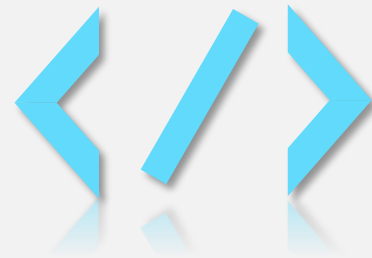
**serviceWorker.js** : script d'arrière plan

- Invisible pour l'utilisateur
- Ne nécessite pas de page Web pour être lancé
- Permet d'afficher du contenu même s'il n'y a pas de connexion internet



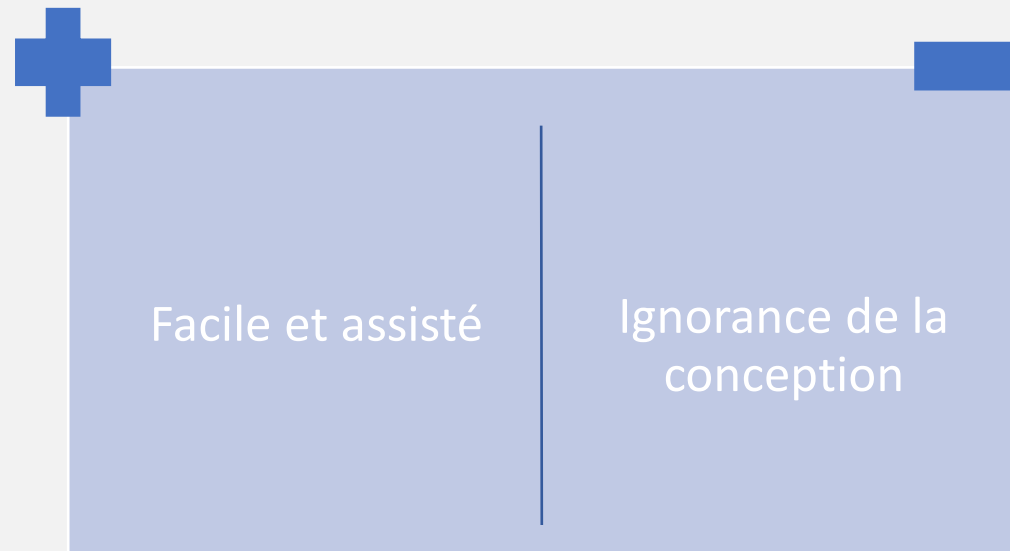
# ReactJS

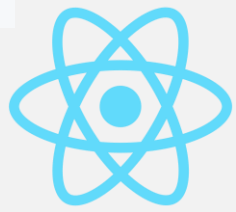
*Vers un monde meilleur !*



## ❑ Create-react-app

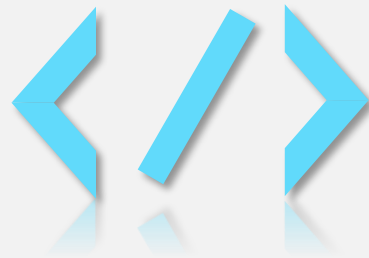
- C'est un module nodeJS **permettant la création d'une application ReactJS** disposant de toute la configuration nécessaire.





# ReactJS

*Vers un monde meilleur !*



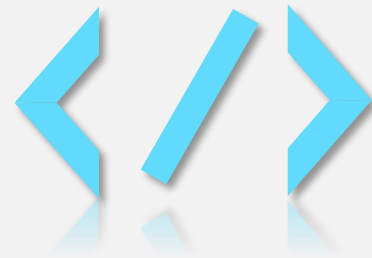
## ❖ **Server Side Rendering**

- Génère le code HTML d'une page sur le server en réponse à la requête
- Technique utilisée pour les SPA

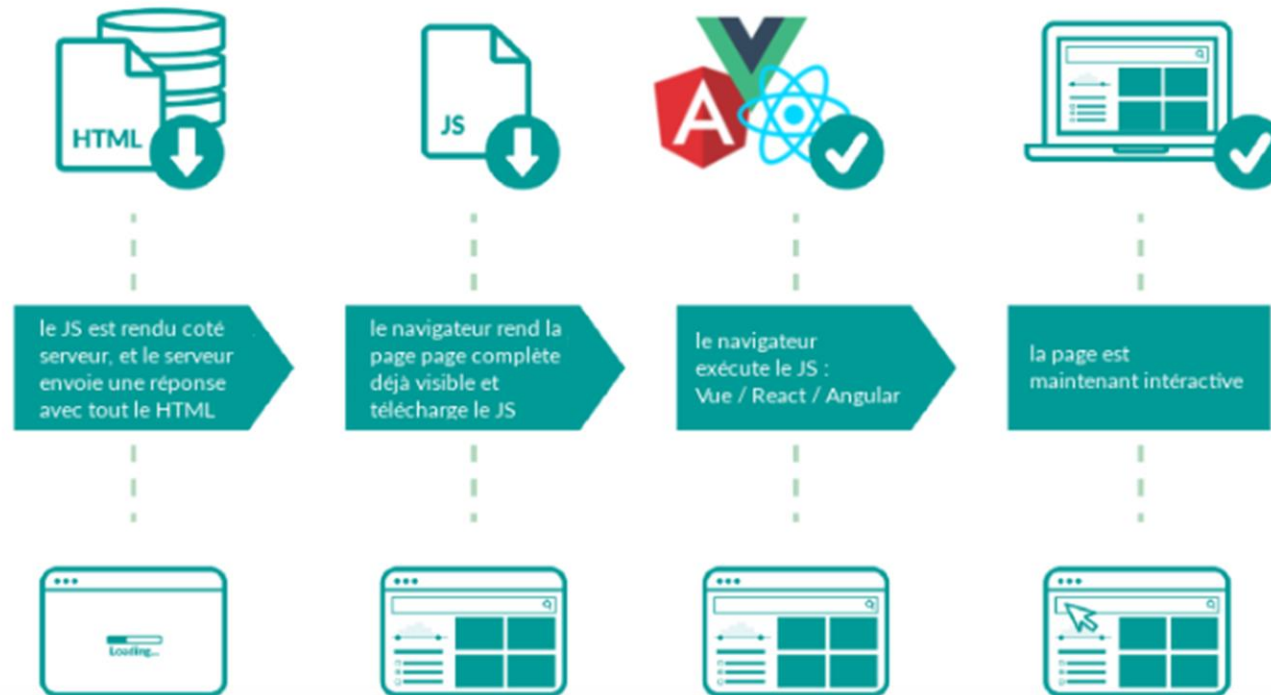


# ReactJS

*Vers un monde meilleur !*



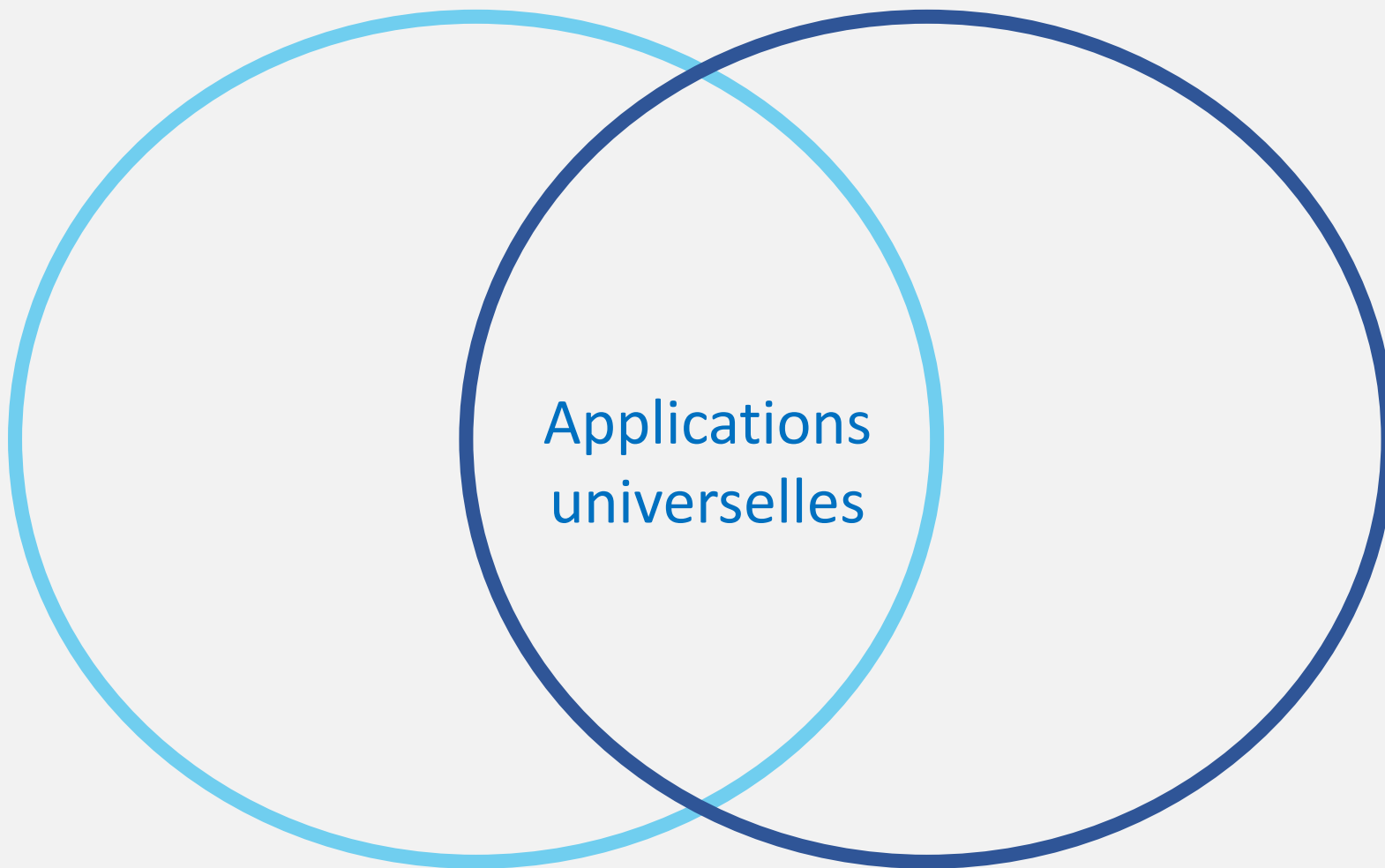
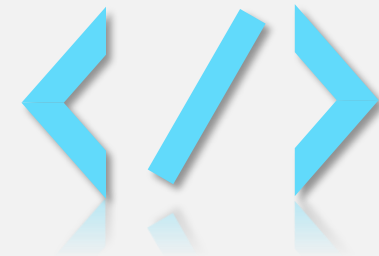
## SSR





# ReactJS

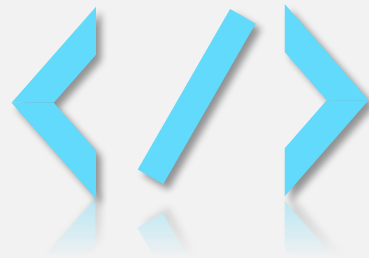
*Vers un monde meilleur !*





# ReactJS

*Vers un monde meilleur !*



## ❖ Application universelle

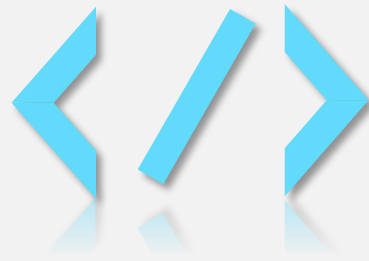






# ReactJS

*Vers un monde meilleur !*



❑ Application de zéro ..... Presque !

*BABEL*

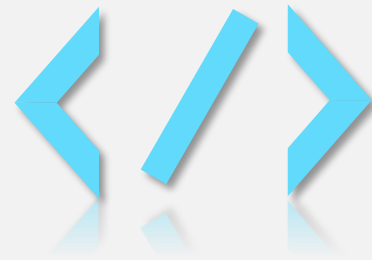


**ES6**



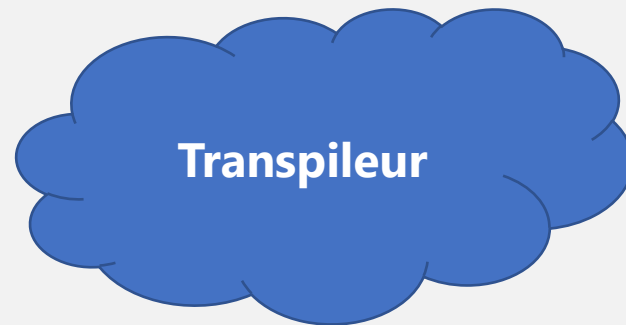
# ReactJS

*Vers un monde meilleur !*

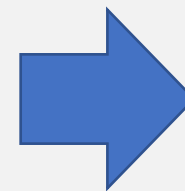


## BABEL

`(x) => x +1;`



Transpileur

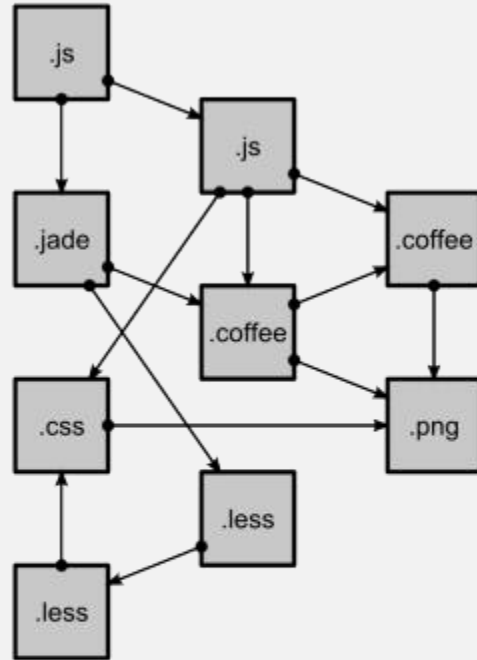
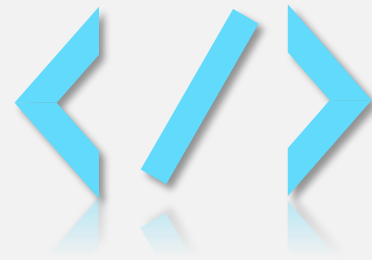


```
function(x){  
  return x+1;  
}
```

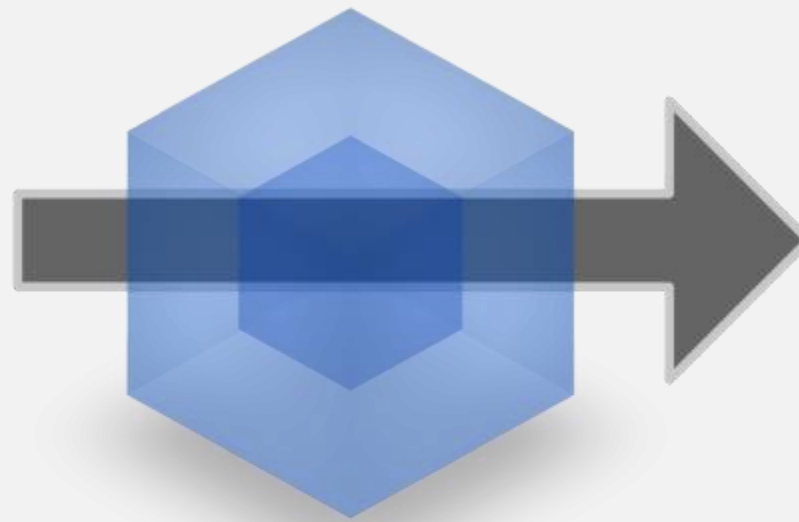


# ReactJS

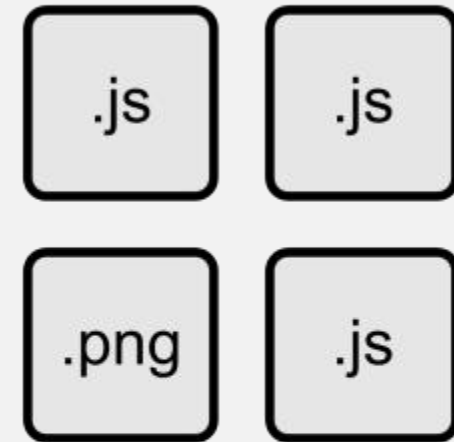
*Vers un monde meilleur !*



modules  
with dependencies



**webpack**  
MODULE BUNDLER

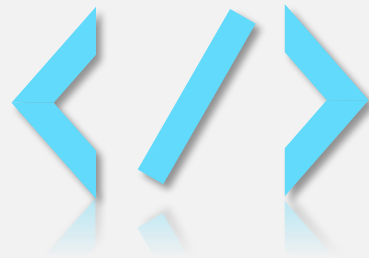


static  
assets



# ReactJS

*Vers un monde meilleur !*



☐ Orienté composant

StateFull

StateLess

# ReactJS

*Vers un monde meilleur !*

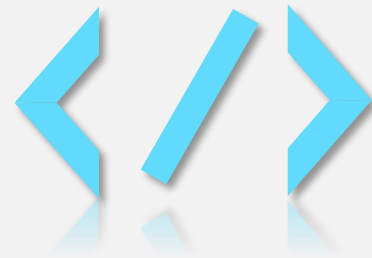
Orienté **Composant**  
**StateFull**





# ReactJS

*Vers un monde meilleur !*



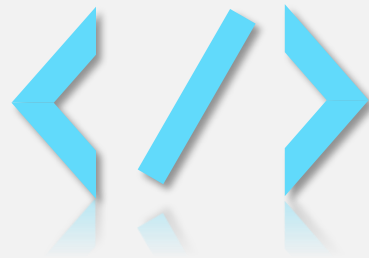
## Orienté composant : *StateFull*

- Qu'est ce que le principe de séparation de concept ??
- **ReactJS, ne fait pas de séparation de concept.** HTML et le JavaScript sont couplés dans un même composant.
- Un composant **peut contenir ou non** ses propres **états**.



# ReactJS

*Vers un monde meilleur !*



## Orienté composant : *StateFull*

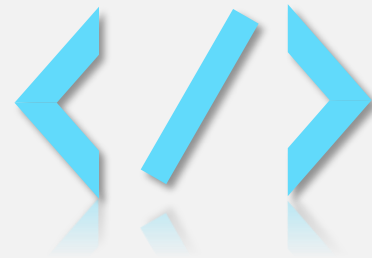
- La première chose à importer lors de la création d'un composant

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



# ReactJS

*Vers un monde meilleur !*



**Orienté composant : *StateFull***

❑ **Exemple :**

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

class Person extends Component{
  render(){
    return <h1>Je suis un composant personne</h1>;
  }
}

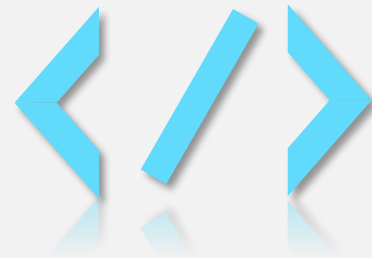
export default Person;
```





# ReactJS

*Vers un monde meilleur !*



**Orienté composant : *StateFull***

☐ Gestion des données

- Deux types de données en ReactJS (**props** and **state**).

state

● **Données  
privées**

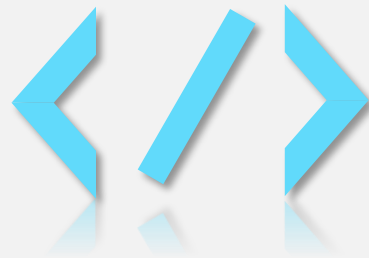
props

● **Données  
externes**



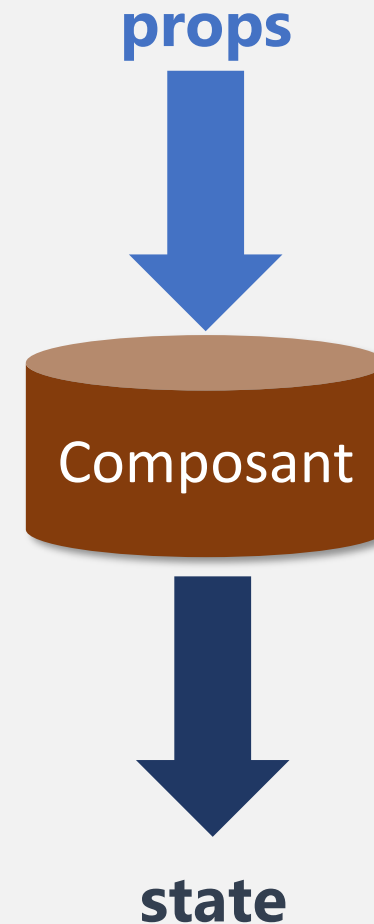
# ReactJS

*Vers un monde meilleur !*



Orienté composant : **StateFull**

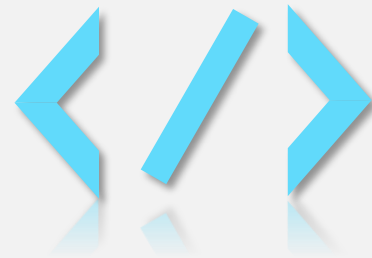
Un composant peut changer ses données interne (**state**) mais pas ses données externe (**props**)





# ReactJS

*Vers un monde meilleur !*



**Orienté composant : *StateFull***

**{ } Déclaration**

```
class Contact extends Component{  
  render(){  
    return  
    <div id="container">  
      <h1>Je suis un composant personne</h1>  
      <p>  
        Nom : {this.props.firstName}  
        <br />  
        Prenom : {this.props.lastName}  
      </p>  
    </div>  
  }  
}
```

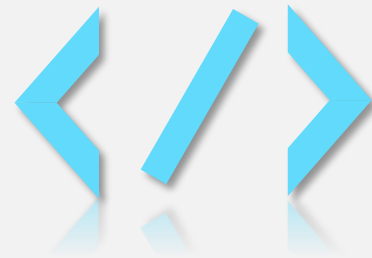
**{ } Appel**

```
<Contact  
  firstName="geoffrey"  
  lastName="parquet" />
```



# ReactJS

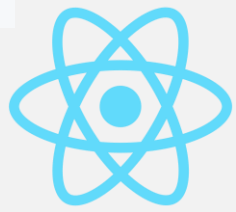
*Vers un monde meilleur !*



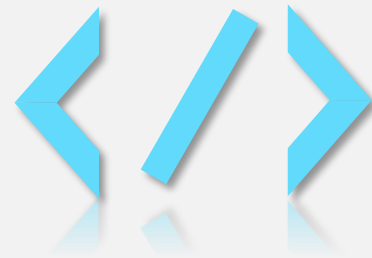
Orienté composant : **StateFull**

## { } Initialisation

- Tout ce passe via le **constructeur**
- Appel de la méthode **super()** permettant d'initialiser l'objet courant (**this**).



# ReactJS



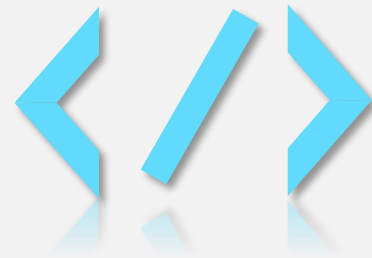
*Vers un monde meilleur !*

```
class Contact extends Component {  
  constructor(){  
    super();  
    this.state = {  
      firstName: '',  
      lastName: ''  
    };  
  }.....  
}  
  
render() {  
  return  
    <div id="container">  
      <h1>Je suis un composant  
      personne</h1>  
      <p>  
        Nom :  
        {this.props.firstName}  
        <br />  
        Prenom :  
        {this.props.lastName}  
      </p>  
    </div>
```



# ReactJS

*Vers un monde meilleur !*



Orienté composant : **StateFull**

## { } Mise à jour du state

- **Bind** de **this** à la méthode. Si ce dernier n'est pas passé, **il ne peut pas être utilisé dans cette dernière.**

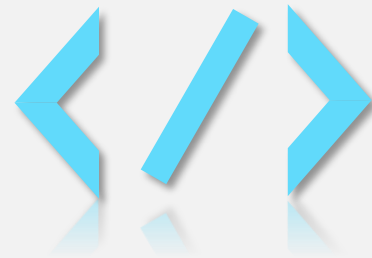
```
this.updateFirstName = this.updateFirstName.bind(this);  
updateFirstName(value){  
  this.setState({  
    ...this.state,  
    firstName: value  
  }):  
}
```
- Utilisation de la méthode **setState()**.

```
<button onClick={this.updateFirstName('Nouveau prénom')}></button>
```



# ReactJS

*Vers un monde meilleur !*



## Orienté composant : **StateFull**

### ☐ Résumé

- Composant appelé **StateFull**
- Utilisation de la classe **Component** de **React**
- Deux types de propriété **state** et **props**
- State **privé au composant**, props **extérieur au composant**
- Modification du state par la méthode **setState()**
- Ne pas oublier de **bind this** au méthode si ce dernier doit être utilisé.

# ReactJS

*Vers un monde meilleur !*

## REACT-DOM

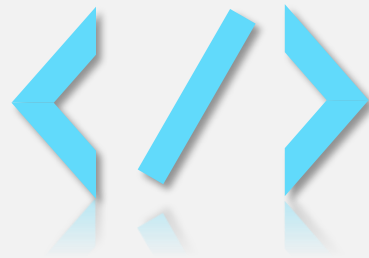






# ReactJS

*Vers un monde meilleur !*

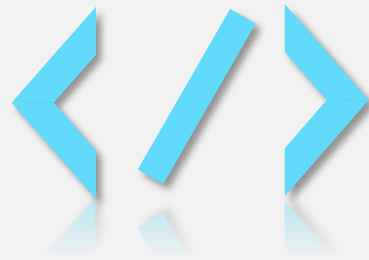


- **ReactDOM** s'assure de mettre à jour le DOM virtuel et le DOM physique
- Pour rendre un élément dans le DOM, il faut utiliser la méthode **render()** de **ReactDOM**
- Ce qui est initialisé dans la méthode, doit être la **racine de l'application**



# ReactJS

*Vers un monde meilleur !*



## □ Exemple :

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

# ReactJS

*Vers un monde meilleur !*

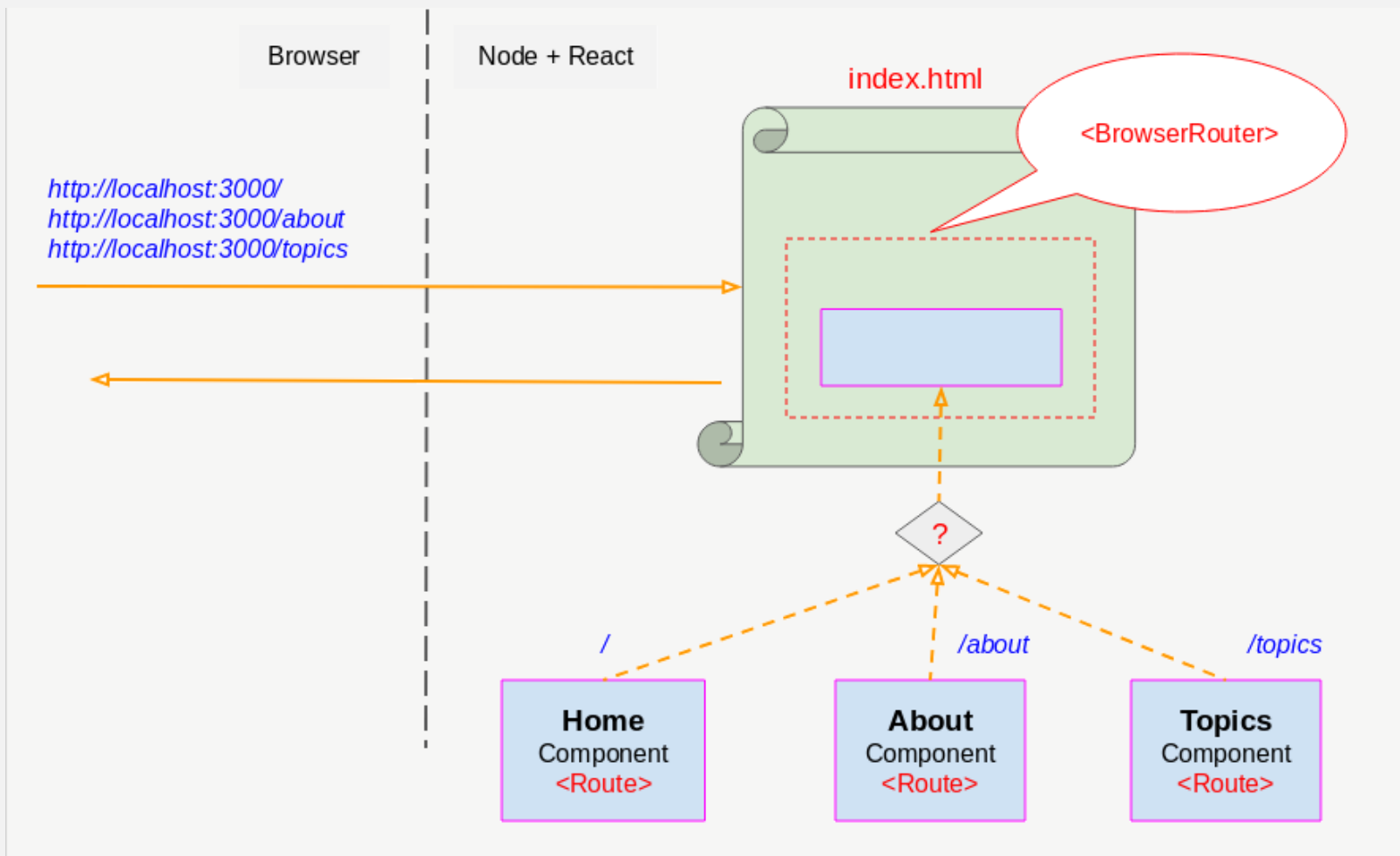
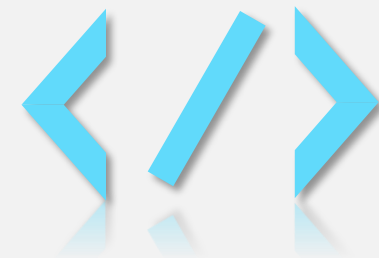
## REACT-ROUTER-DOM

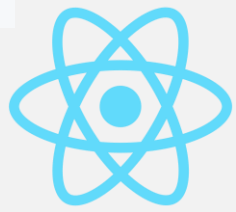




# ReactJS

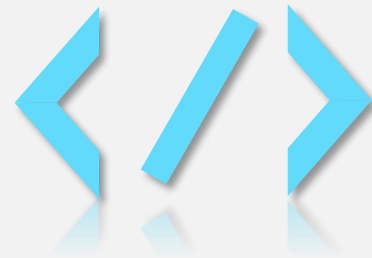
*Vers un monde meilleur !*





# ReactJS

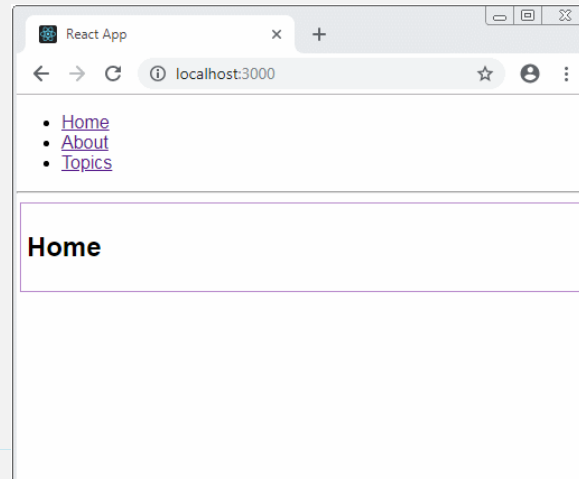
*Vers un monde meilleur !*



**React Router** : utilisation du package **react-router-dom**

❑ Quèsaco ???

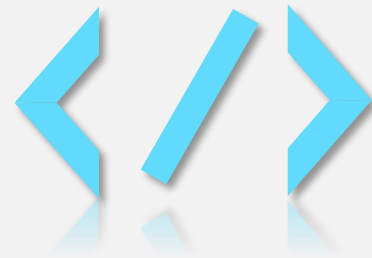
- Permet de **définir des URL dynamique** et de **sélectionner** un **composant** pour l'affichage (render) sur le navigateur de l'utilisateur.





# ReactJS

*Vers un monde meilleur !*



## React Router : Fournit deux composants

### ❑ <BrowserRouter>

- Utilisé le plus couramment.
- Il utilise le **History API** incluse dans **HTML5** pour surveiller l'historique du routeur.

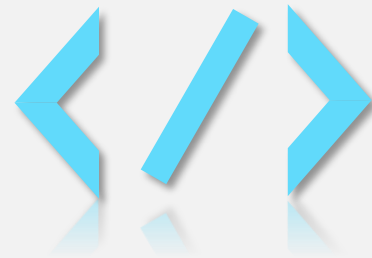
### ❑ <HashRouter>

- Utilise le hash de l'URL (***window.location.hash***) pour tout mémoriser.
- Utile pour les anciens navigateurs.



# ReactJS

*Vers un monde meilleur !*



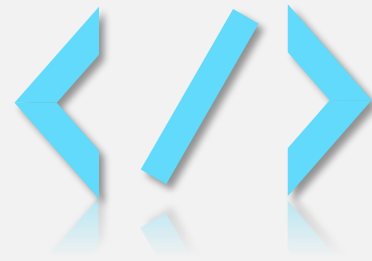
## React Router : Fournit deux composants

```
<BrowserRouter>  
  <Route exact path="/" component={Home}/>  
  <Route path="/about" component={About}/>  
  <Route path="/topics" component={Topics}/>  
</BrowserRouter>
```

```
<HashRouter>  
  <Route exact path="/" component={Home}/>  
  <Route path="/about" component={About}/>  
  <Route path="/topics" component={Topics}/>  
</HashRouter>
```



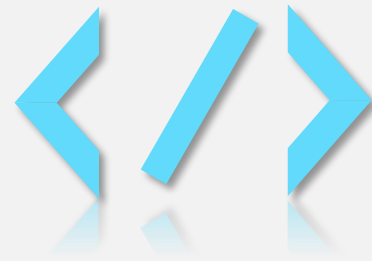
# A vous de jouer !







# A vous de jouer !



`{}` **Etape 1 :** `https://github.com/Gparquet/mediafuture.git`

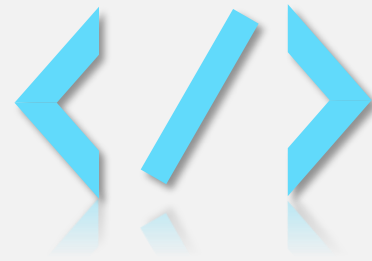
`{}` **Etape 2 :** Placer vous dans le dossier, puis exécuter `npm install`

`{}` **Etape 3 :** Une fois terminé, exécuter la commande `npm start`





# A vous de jouer !



**{}** **Etape 4** : Créer une nouvelle branche **feat/home** de la branche **master**

**{}** **Etape 5** : Créer un dossier **layout** et dans ce dernier créer un dossier **App**.

Placez-y les éléments relatifs au domaine App.

Faites en sorte que l'application compile et s'affiche.

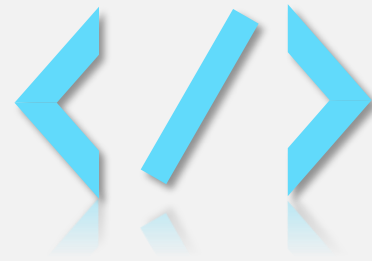
**{}** **Etape 6** : Modification de App.js en ajoutant un composant Header et Footer.

- *Réutilisation des composants* :

- @axa-fr/react-toolkit-layout-header
- @axa-fr/react-toolkit-layout-footer

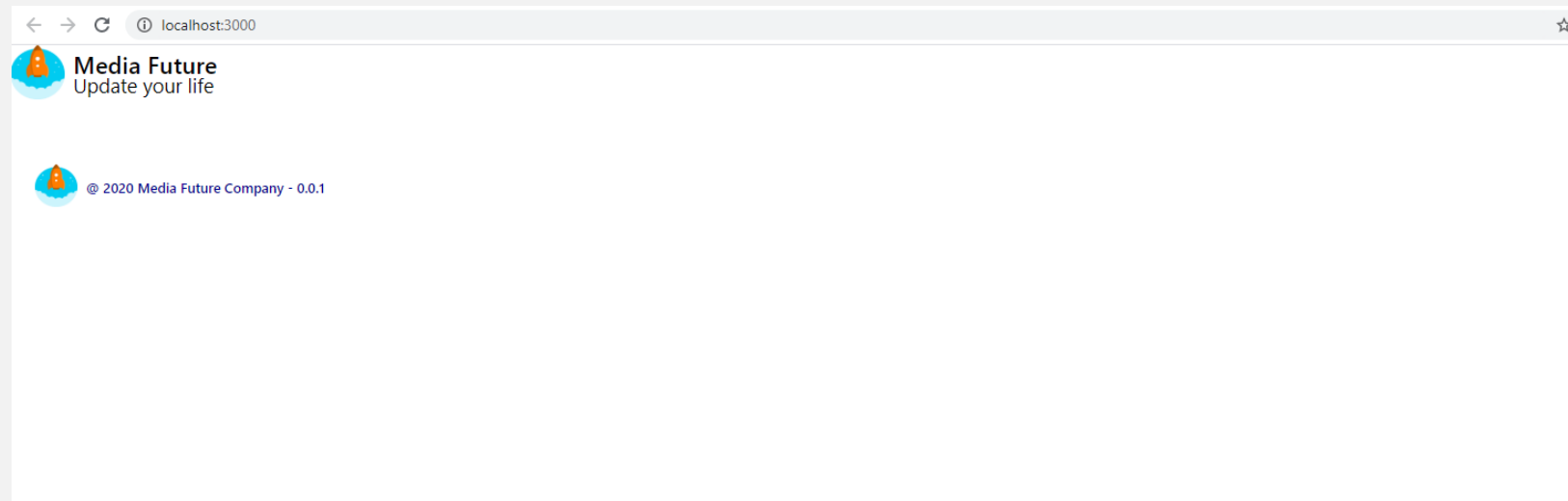


# A vous de jouer !



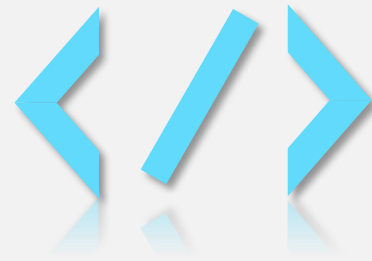
`{}` **Etape 7** : Ajouter le logo rocket.png, titre, etc... au Header et Footer.

**Cf. l'image ci-dessous :**





# A vous de jouer !



`{}` **Etape 8** : Ajouter un router à App.js permettant de faire pointer sur un composant Home.js de **type stateFull**.

Ce composant contiendra 4 zones, cf. image ci-dessous :



Le composant contiendra le titre des boutons en tant que propriété d'état

# ReactJS

*Vers un monde meilleur !*

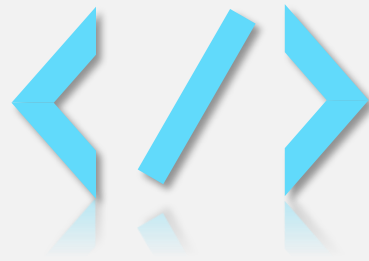
Gestion du **cycle de vie** d'un  
composant





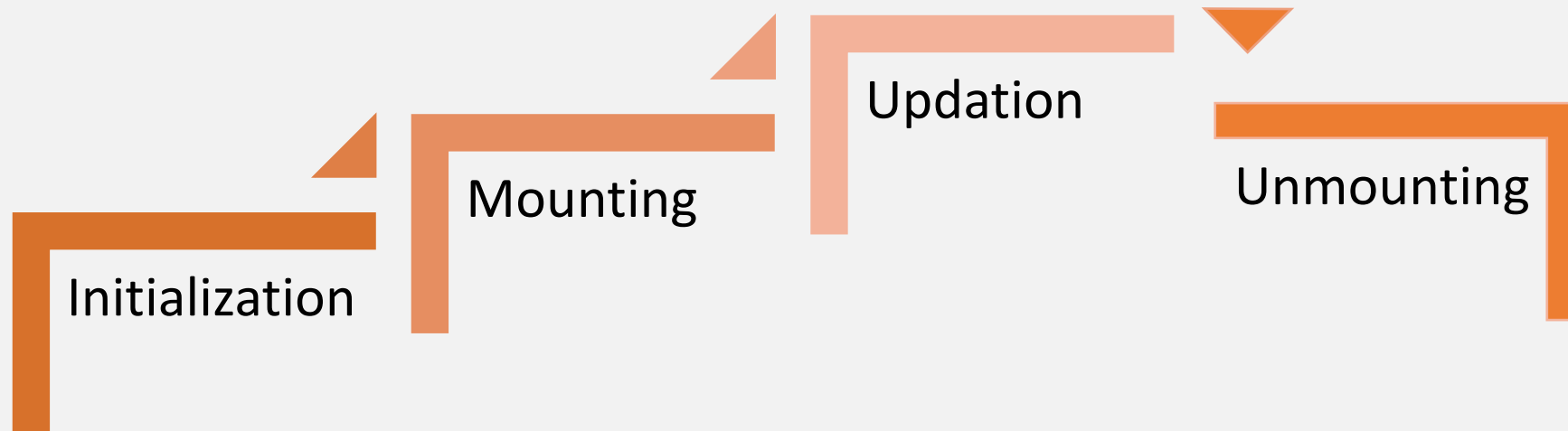
# ReactJS

*Vers un monde meilleur !*



## Gestion du cycle de vie des états avec les *Hooks*

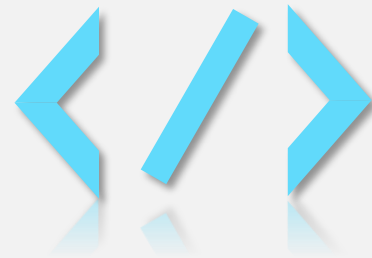
- 4 phases d'un composant React





# ReactJS

*Vers un monde meilleur !*



## Initialization

setup props and state

## Mounting

componentWillMount

render

componentDidMount

## Updation

### props

componentWillReceiveProps

shouldComponentUpdate

true

false

componentWillUpdate

render

componentDidUpdate

### states

shouldComponentUpdate

true

false

componentWillUpdate

render

componentDidUpdate

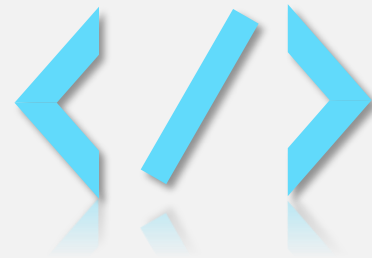
## Unmounting

componentWillUnmount



# ReactJS

*Vers un monde meilleur !*



- **Setup** props and state

```
class Contact extends Component {  
  constructor(){  
    super();  
    this.state = {  
      firstName: '',  
      lastName: ''  
    };  
  }  
  Contact.defaultProps = { theme:  
    'dark' }  
}
```

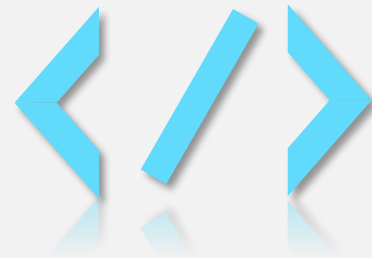
- *Setup du state initiale dans le **constructeur***
- *Setup des props par défaut par la propriété **defaultProps***





# ReactJS

*Vers un monde meilleur !*



- **Mounting**

- `componentWillMount`

- Exécuté juste avant la montée du composant dans le DOM
    - Exécuté juste avant le première appel de la méthode **Render()**

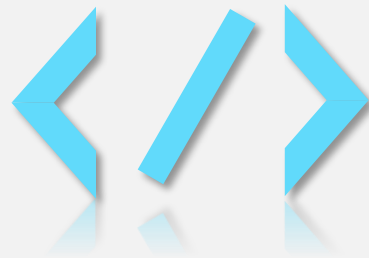
- `render`

- Méthode pure, pour la montée du composant dans le navigateur.



# ReactJS

*Vers un monde meilleur !*



- **Mounting**

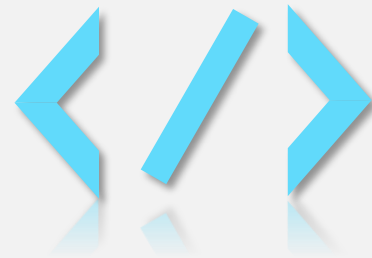
- `componentDidMount`

- Exécuté juste après la montée du composant dans le DOM
    - Utile pour la récupération des données par appel d'API (*par exemple*)



# ReactJS

*Vers un monde meilleur !*



- **Updation**

- Cette étape survient lorsque le composant **reçoit une mise à jour**
- Démarrage de cette étape **lors de l'appel de la méthode `setState()`**

- `souldComponentUpdate`

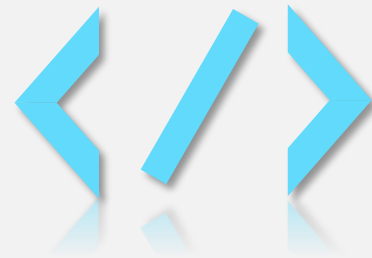
- Dit à React que le composant **reçoit une nouvelle demande de mise à jour** des props **et ou** state.

*Cette dernière porte bien son nom : **Est-ce que le composant doit être mise à jour.***



# ReactJS

*Vers un monde meilleur !*



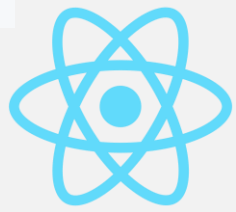
- **Updation**

- `componentWillUpdate`

- Exécute **après** que la méthode **shouldComponentUpdate** renvoie **true**
    - Appel de la méthode `render`, une fois cette dernière exécutée.

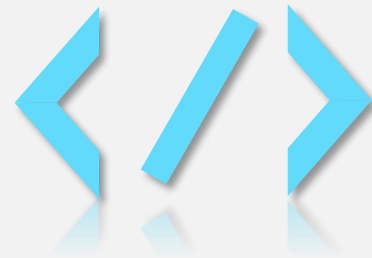
- `componentDidUpdate`

- Est exécuté quand la mise à jour a été effectué dans le DOM
    - Cette méthode est utilisée pour relancer les bibliothèques tierces utilisées pour s'assurer que ces bibliothèques se mettent également à jour et se rechargent.



# ReactJS

*Vers un monde meilleur !*



- **Updation**

- Liste des méthodes qui seront appelées lorsque le parent enverra de nouvelles props sont les suivantes :

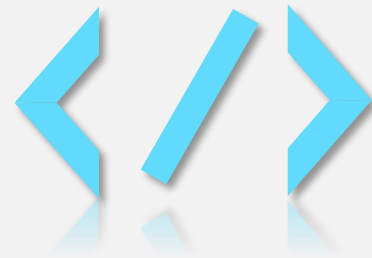
- ❑ `componentWillReceiveProps`

- Est exécuté lorsque les props ont changées et ne sont pas d'abord rendus.



# ReactJS

*Vers un monde meilleur !*



- **Unmounting**

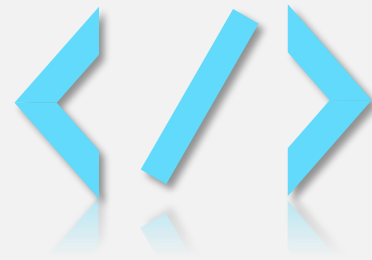
- Dans cette phase, le composant n'est pas nécessaire et il sera « démonté » du DOM. La méthode appelée dans cette phase est la suivante :

- `componentWillUnmount`

- Dernière méthode du cycle de vie
    - Exécuté avant que le composant soit démonté du DOM
    - Utile pour nettoyer par exemple des informations sensibles des informations d'authentification

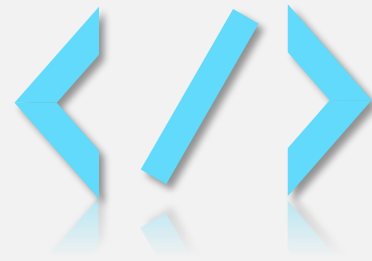


# A vous de jouer !






# A vous de jouer !



**{}** **Etape 1 :** Créer une nouvelle branche `feat/badge` puis installer le package `material-ui/core` (`npm install @material-ui/core`)

**{}** **Etape 2 :** Créer un composant `categories` de type `statefull` permettant d'afficher pour chaque catégorie le nombre d'éléments. Cf. exemple ci-dessous :



Media Future


Update your life

Book<sup>1</sup>

eBook<sup>1</sup>

Album<sup>2</sup>

Movie<sup>2</sup>



@ 2020 Media Future Company - 0.0.1



# ReactJS

*Vers un monde meilleur !*

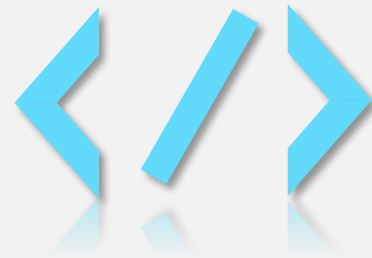
## PropTypes





# ReactJS

*Vers un monde meilleur !*



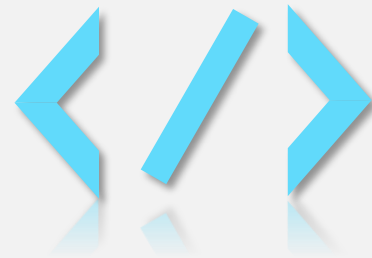
## Définir fortement les props avec **PropTypes**

- Probablement aucune erreur, aucun avertissement dans la console
- Pour configurer correctement le composant, il faut lui passer les bonnes *props*.
- Les *props* sont, véritablement, **l'API du composant**. Pour cela, il est fondamental de définir formellement cette API (la liste des props).



# ReactJS

*Vers un monde meilleur !*



## Définir fortement les props avec **PropTypes**

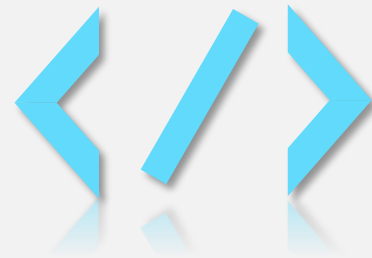
❑ Comment ça fonctionne ???

- React examine sur chaque composant une propriété statique nommé **propTypes**.
- C'est un objet **dont les clés sont les noms des props attendues**, et les valeurs des validateurs de props.
- Le module standard **prop-types** fournit une série de validateur.



# ReactJS

*Vers un monde meilleur !*



## Définir fortement les props avec **PropTypes**

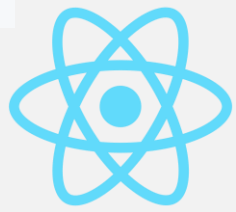
- Le développeur pourra avoir un retour instantané si une props est requise ou si la valeur n'est pas du bon type par exemple.

### ❑ Mise en place

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

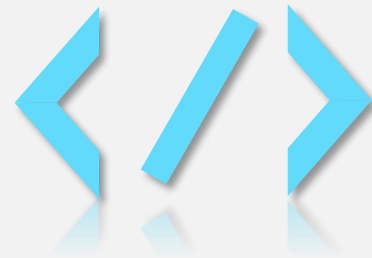
***Juste avant l'export du composant***

```
Contact.propTypes = {  
  ....  
};
```



# ReactJS

*Vers un monde meilleur !*



## Définir fortement les props avec **PropTypes**

- Sans mécanisme de définition formel, les bugs sont parfois difficile à repérer

```
<Contact props={{firstName : 'toto'}} />
```

### ❑ Exemple :

```
<Contact Props={{firstName : 'toto'}} />
```

# ReactJS

*Vers un monde meilleur !*

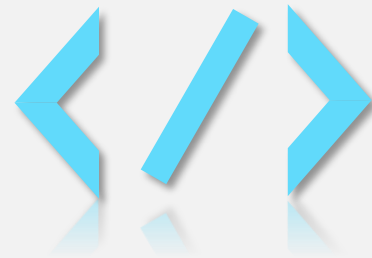
Orienté Composant  
StateLess





# ReactJS

*Vers un monde meilleur !*



## Orienté composant : *StateLess*

```
import React from 'react';

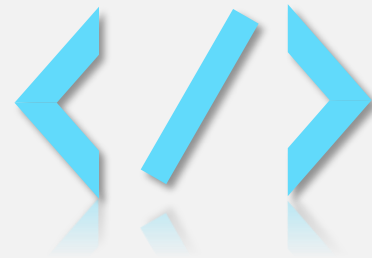
const Contact = (props)=>{
  return (
    <div>
      je suis un composant
      Contact {props.firstName}
    </div>
  );
}
```

- Fonction ayant la signature **object => JSX**
- Composant **sans gestion de son propre état.**
- **Props est passé par le parent.**  
Lecture seule uniquement



# ReactJS

*Vers un monde meilleur !*



## Orienté composant : *StateLess*

- Pour passer une valeur à un composant :

```
<Contact props={{firstName : 'toto'}} />
```

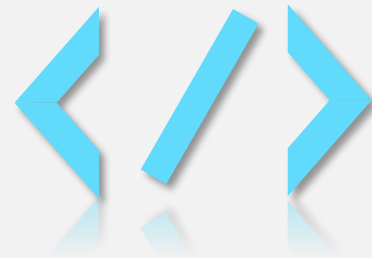
- Le composant créé est aussi appelé « **Composant fonctionnel** » car il est créé en utilisant une fonction pure.
- Le résultat **ne dépend** que des **arguments reçus et rien d'autre**





# ReactJS

*Vers un monde meilleur !*



## HOC : **H**igher **O**rders **C**omponent

❑ Quèsako ???

- C'est une fonction de base qui renvoie une autre fonction améliorée.





# ReactJS

*Vers un monde meilleur !*



## HOC : **H**igher **O**rders **C**omponents

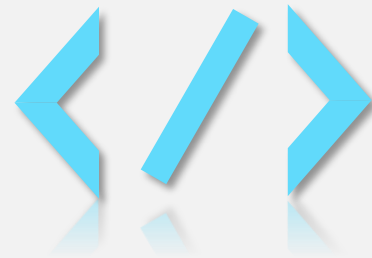
□ Le but ??

- Rendre le **code plus lisible** en **cachant la logique** derrière une fonction
- Appelé aussi « **pattern décorateur** »



# ReactJS

*Vers un monde meilleur !*



## HOC : Higher Order Component

### ❑ Exemple :

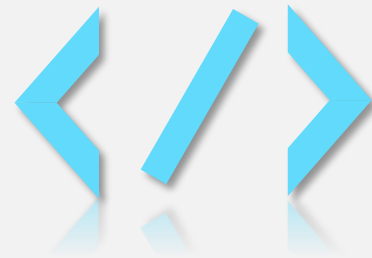
```
const Contact = (props) =>
{
  return props.loading
    ? <StylishSpinner />
    : <div>
      {props.firstName}
    </div>
}
```

- Affiche un spinner de chargement tant que le contact n'est pas chargé
- **Problème** : L'affichage qui apporte de la valeur (***comment afficher un utilisateur***) est noyée avec la partie qui s'occupe de l'affichage du chargement.



# ReactJS

*Vers un monde meilleur !*

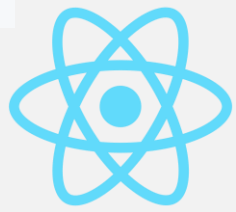


## HOC : **H**igher **O**rders **C**omponent

### Première extraction

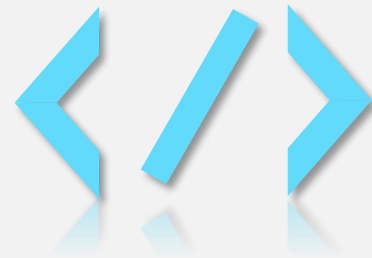
```
const FillContact = (props) =>
{
  return (
    <div>
      {props.user.name}
    </div>
  )
}
```

```
const Contact = (props) => {
  return props.loading
    ? <loadSpinner />
    : <FillContact {...props} />
}
```



# ReactJS

*Vers un monde meilleur !*



## HOC : **H**igher **O**rders **C**omponent

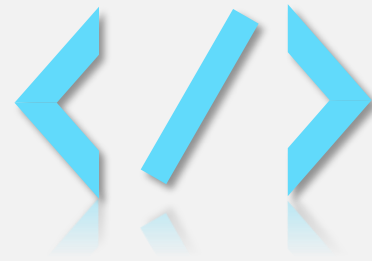
### Deuxième extraction

```
const withLoading =(isLoading,
BaseComponent)=> {
  return (props)=> {
    isLoading(props)
    ? <StylishSpinner />
    : <BaseComponent {...props} />
  }
}
```

```
const Contact =
withLoading(
  (props) =>
    props.loading,
  FillContact
)
```



# A vous de jouer !



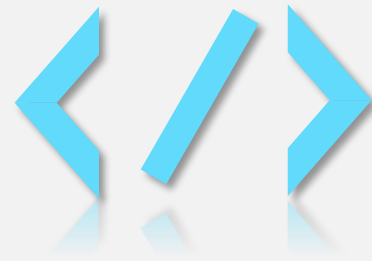
`{}` **Etape 1 :** Créer une nouvelle branche `feat/categoriesStateless`

`{}` **Etape 2 :** Modifier le composant `Categories` pour le passer en fonction pure sans état.

`{}` **Etape 3 :** Créer un composant « supérieur » `CategoriesContainer` dans le dossier `Categories` pour gérer l'état du composant `Categories` précédemment modifié



# A vous de jouer !



`{}` **Etape 4** : Ajouter les proptypes nécessaires pour le composant  
Categories

# ReactJS

*Vers un monde meilleur !*

La bibliothèque **Recompose**

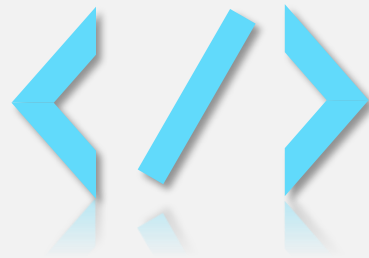






# ReactJS

*Vers un monde meilleur !*



## Recompose :

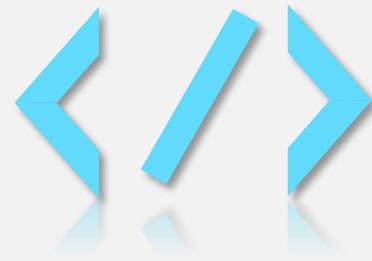
❑ Quèsaco ???

- Assistant de gestion d'état pour les composants HOC
- Dispose de nombreuses méthode d'aide pour la gestion des états



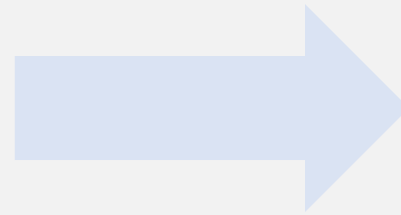
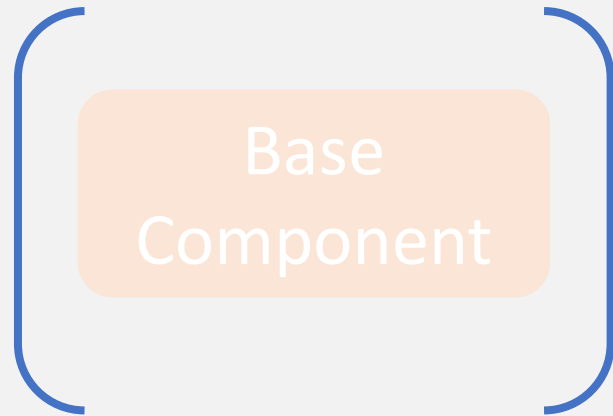
# ReactJS

*Vers un monde meilleur !*



## Exemple de HOC les containers :

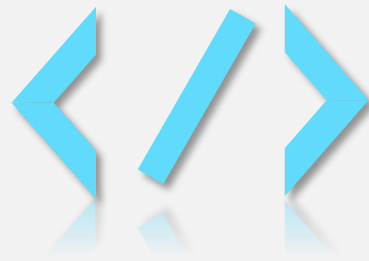
❑ Quèsaco ???



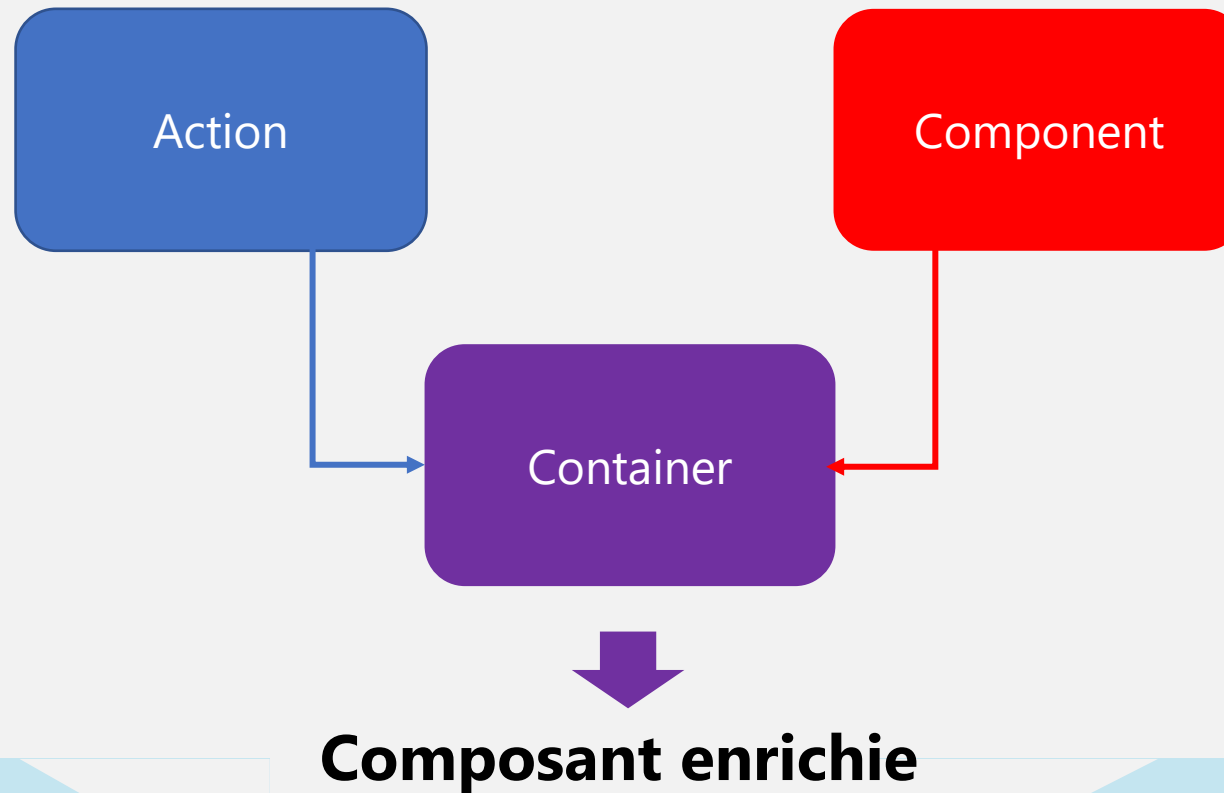


# ReactJS

*Vers un monde meilleur !*



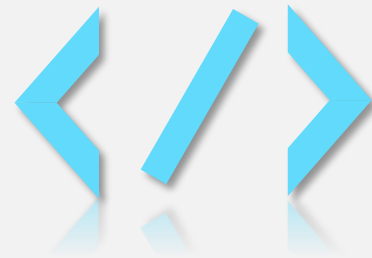
## Exemple de HOC les containers :





# ReactJS

*Vers un monde meilleur !*



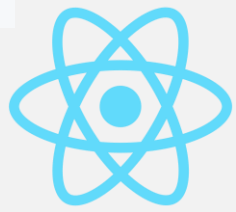
## Recompose :

### ❑ `compose()`

- Composition de plusieurs HOC pour les fusionner en un seul

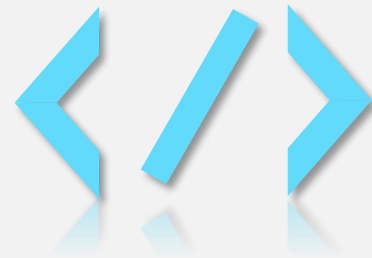
### ❑ `withHandlers()`

- Ajoute de nouveau handler d'événement comme propriété du composant encapsulés



# ReactJS

*Vers un monde meilleur !*



## Recompose :

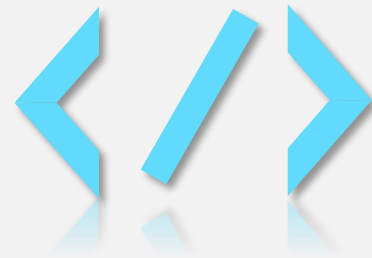
### ❑ useState()

- Ajout d'un état local au composant
- Elle prend en paramètre :
  - Le **nom de l'attribut dans l'état** qui sera donné comme propriété au composant enfant
  - Le **nom de la propriété contenant la fonction** pour mettre à jour cette état
  - La valeur initiale (**statique ou fonction** prenant en paramètre les propriétés et retournant la valeur initiale)



# ReactJS

*Vers un monde meilleur !*



## Recompose :

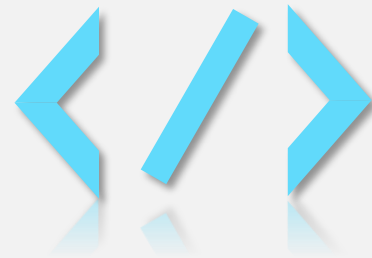
❑ `useState()`

```
useState(  
  "inputValue",  
  "setInputValue",  
  // `formatValue` est l'un des paramètres de  
  // notre HOC  
  props => formatValue(props.value),  
);
```



# ReactJS

*Vers un monde meilleur !*



## Recompose :

- Tout comme le composant StateFull, il est possible de gérer le cycle de vie d'un composant

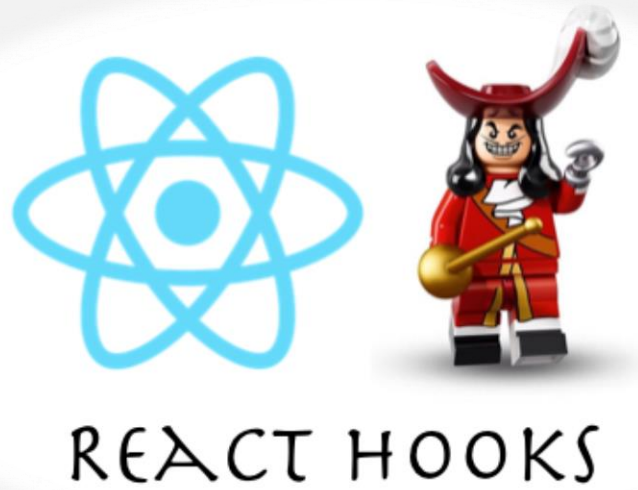
### ❑ `lifecycle()`

- Ajout d'un « hook » permettant d'utiliser une partie du cycle de vie d'un composant.

```
lifecycle({  
  componentDidMount: ()=> {  
    this.props.getList(this.props);  
  }  
})
```

# ReactJS

*Vers un monde meilleur !*

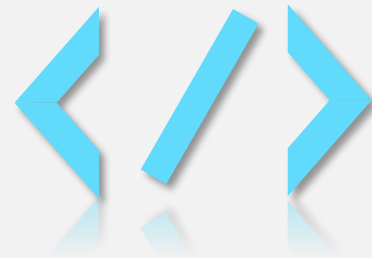






# ReactJS

*Vers un monde meilleur !*



## Hooks API : *le futur de React*

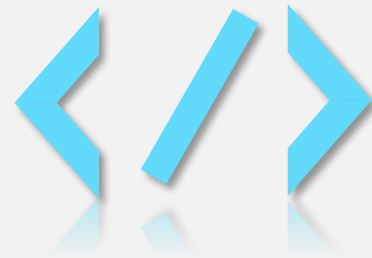
### ❑ Quèsaco ???

- Disponible depuis la **version 16.8** de ReactJS
- Permet :
  - D'écrire du code plus propre et épuré.
  - De ce passer complètement du mot clé class
  - D'utiliser la gestion d'état centralisé dans des composants "Stateless".



# ReactJS

*Vers un monde meilleur !*



## **Hooks API: le futur de React**

- **useState :**
  - Permet d'ajouter un état à un composant
- **useEffect :**
  - Permet d'ajouter un événement de gestion du cycle de vie (componentDidMount, componentDidUpdate).
- **useReducer :**
  - Permet d'ajouter un reducer permettant de regrouper le traitement des données
- **useContext :**
  - Permet d'ajouter un context global à l'application évite l'effet « props drilling »

# ReactJS

*Vers un monde meilleur !*

useState

useEffect

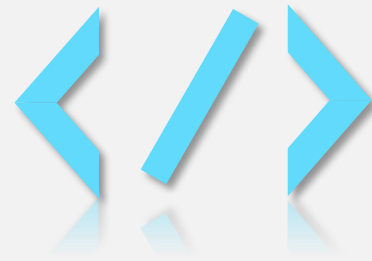
# Live Coding

useContext

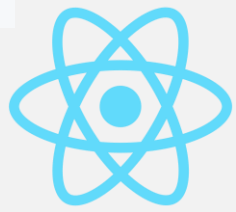
useEffect



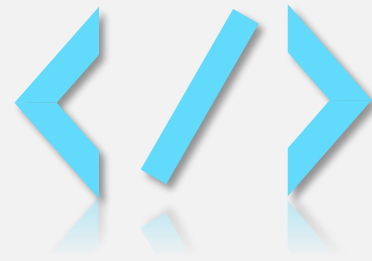
# A vous de jouer !



- Mettre en place une gestion d'état centralisé.
- Avec la méthode `useEffect`, récupérer les contacts du serveur de manière asynchrone afin de les ajouter dans l'état centralisé
- Transformer les composants afin de récupérer les données de l'état centralisé



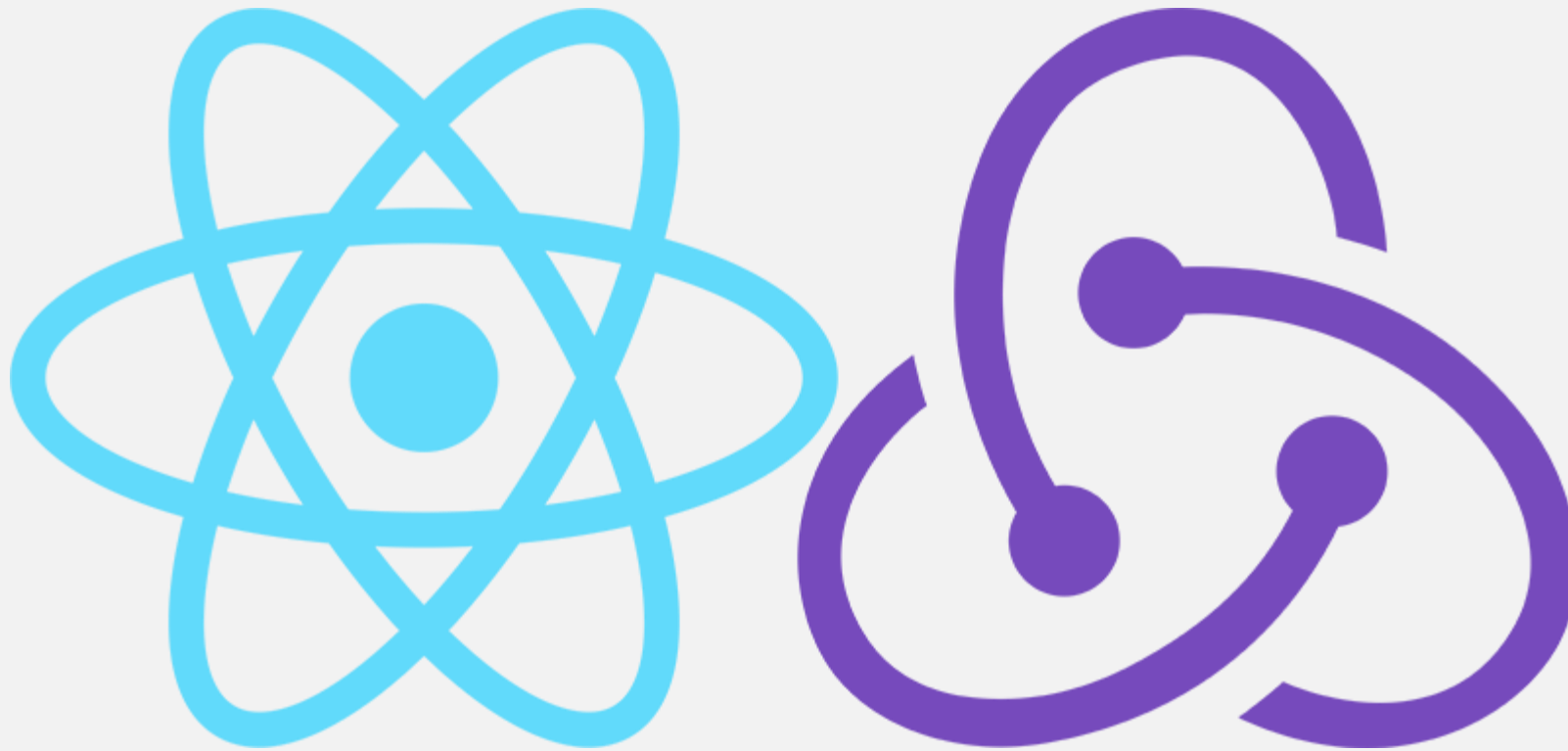
# Piqure de rappel



- Des composants, des composants et encore des composants.
- StateFull et StateLess
- Les composants dispose d'un cycle de vie
  - Initilizing
  - Mounting
  - Updating
  - Unmounting
- Filet de sécurité avec les prop-types

# Redux

*Vers une meilleur gestion des états !*



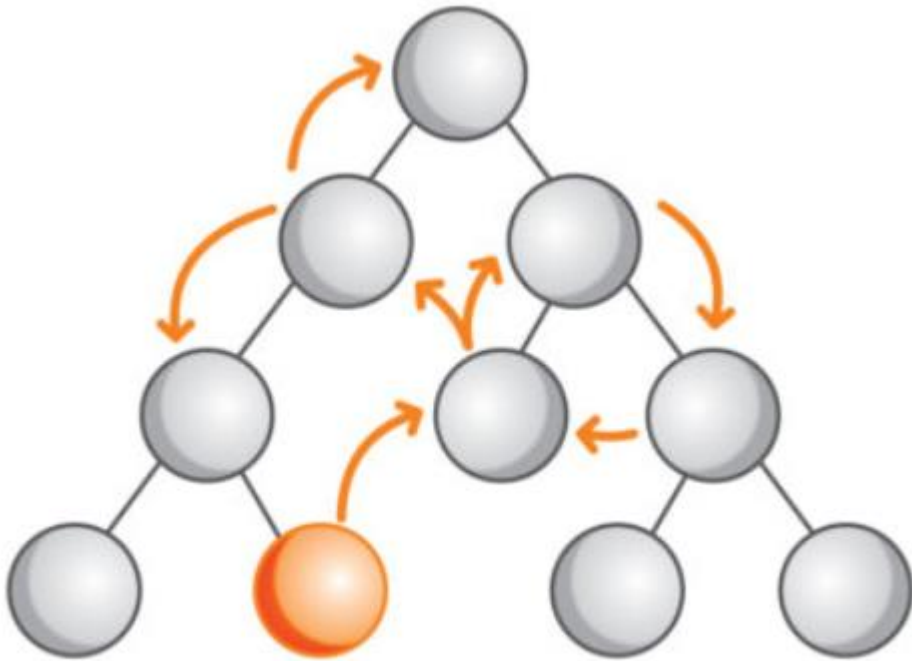


# Comment gérer au mieux les états ?





# Props drillings et problématique de communication inter composant



- Si nous devons **modifier une composant** enfant se trouvant en **bas de pile**, les props (données) doivent y être **amené jusqu'en bas**.
- Si un **composant doit communiquer** avec **un autre composant n'étant pas son arbre direct**, il faut **rapatrier la donnée à travers une multitude de composant**.





# Flux : Introduction



□ Quèsaco ?



- Une architecture (manière de pensée)
- Permet de compléter les composants avec un flux unidirectionnel de données.



# Flux



## □ 4 parties principales



**Action**

- Action à exécuter.
- Interaction avec l'utilisateur

**Dispatcher**

- Répartiteur de la logique métier.
- Réceptionne les actions et les transmet au store

**Store(s)**

- Magasin de données centralisées
- Réagit à l'action depuis le dispatcher

**Views**

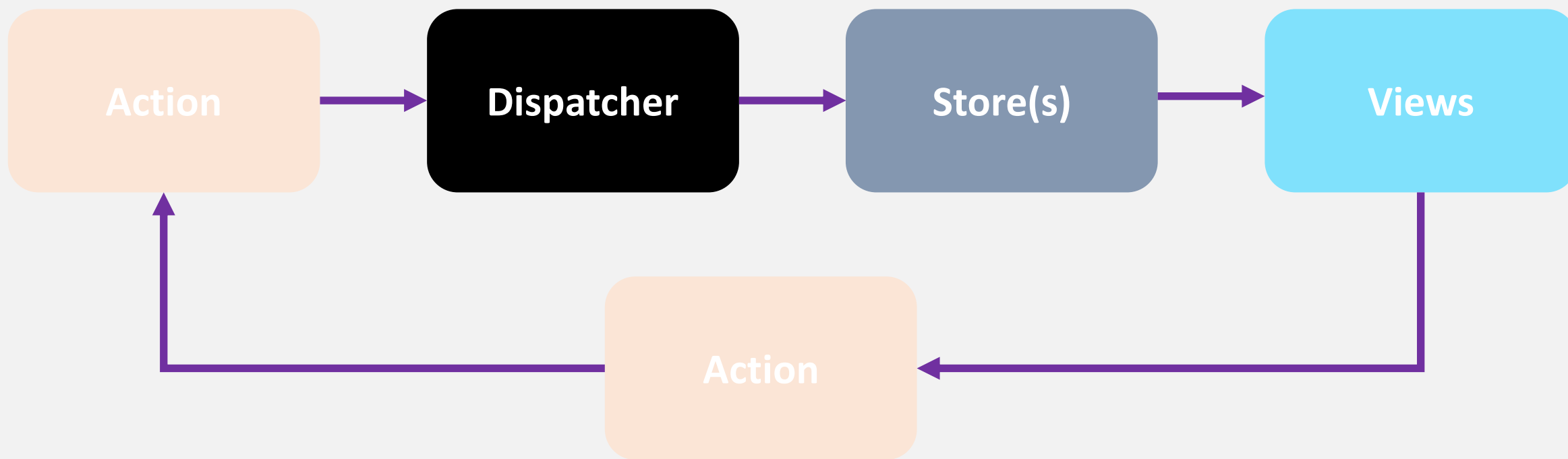
- Vue(s) react.
- Rendu des données



# Flux

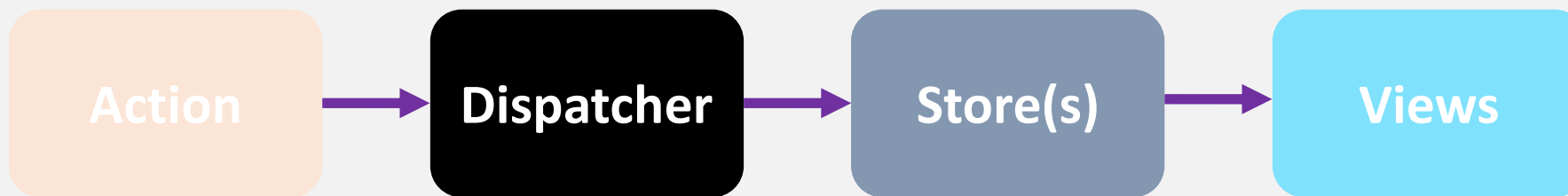


❑ Mode de pensée unidirectionnel





# Flux



- Lorsqu'un **utilisateur interagit** avec une **vue**, cette dernière **propage une action** via un répartiteur (**dispatcher**) central à **tous les stores**.
- Fonctionne très bien avec le mode déclaratif de React qui **permet au store d'envoyer des mises à jours sans spécifier comment transformer les vues entre les états**.



# Flux



Cependant il reste un problème, **la gestion de multi-store .....**





# Redux



Un store pour les gouverner tous !





# Redux



□ Quèsaco ?



- Une librairie et non un pattern. Cette dernière englobe les bonnes pratiques de flux.
- Centralisation des données en un seul store.
- 4 concepts de base avec une responsabilité identifiée



# Redux : 4 grands concepts

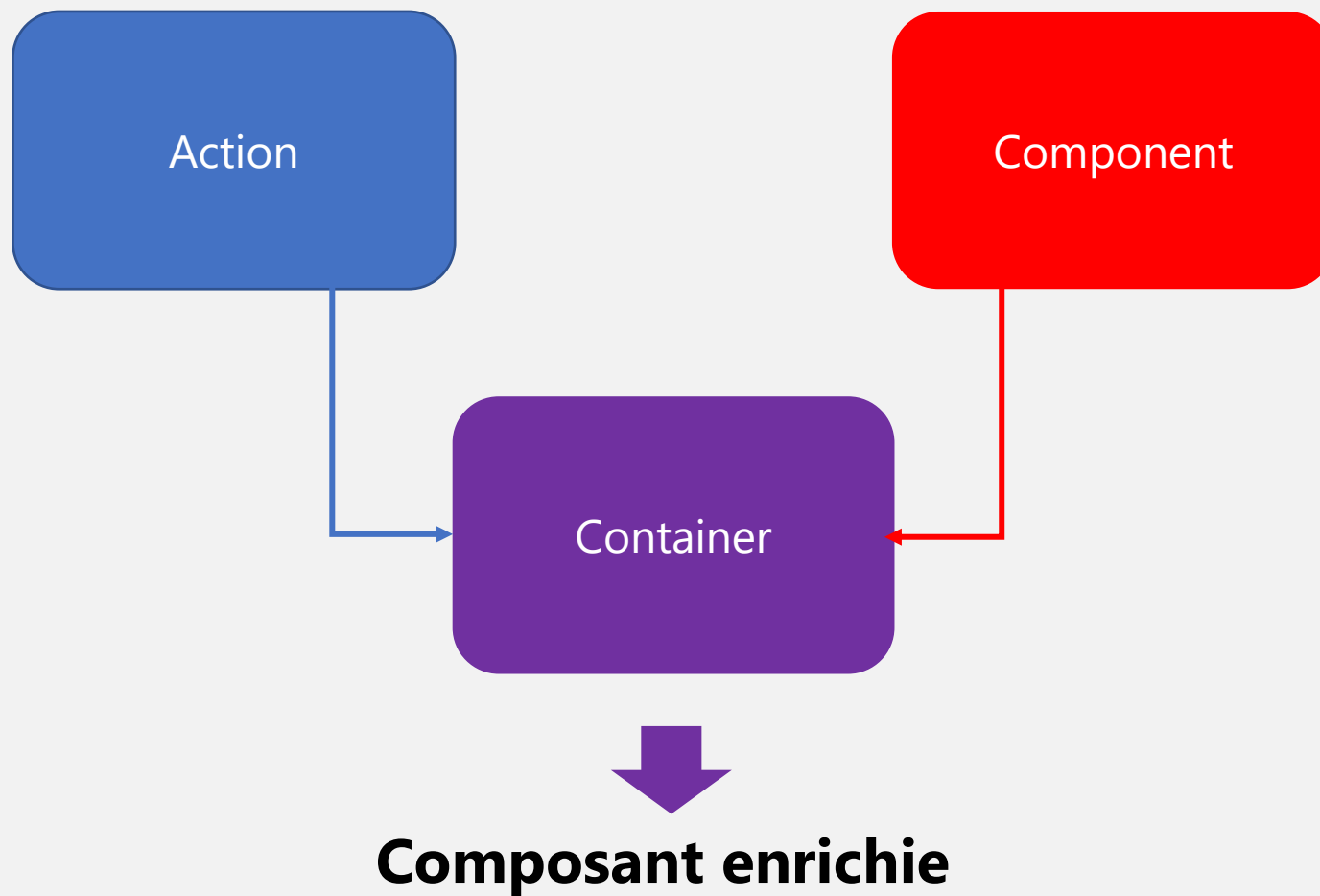


- ❑ **Actions :**
  - Action serveur / interaction avec l'utilisateur
- ❑ **Reducers :**
  - Gestion de la mise à jour du store
- ❑ **Store :**
  - Contient les données et réagit aux actions depuis le dispatcher
- ❑ **Views :**
  - Rendu des données



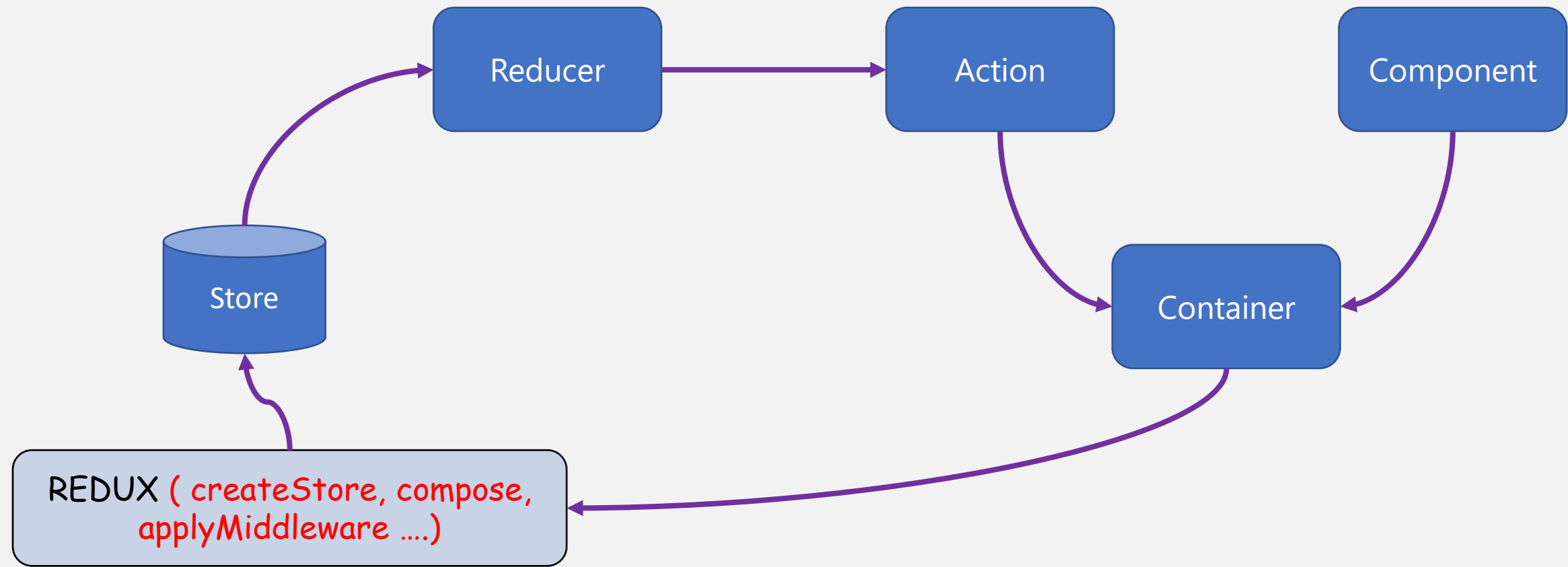


# Redux : Les containers



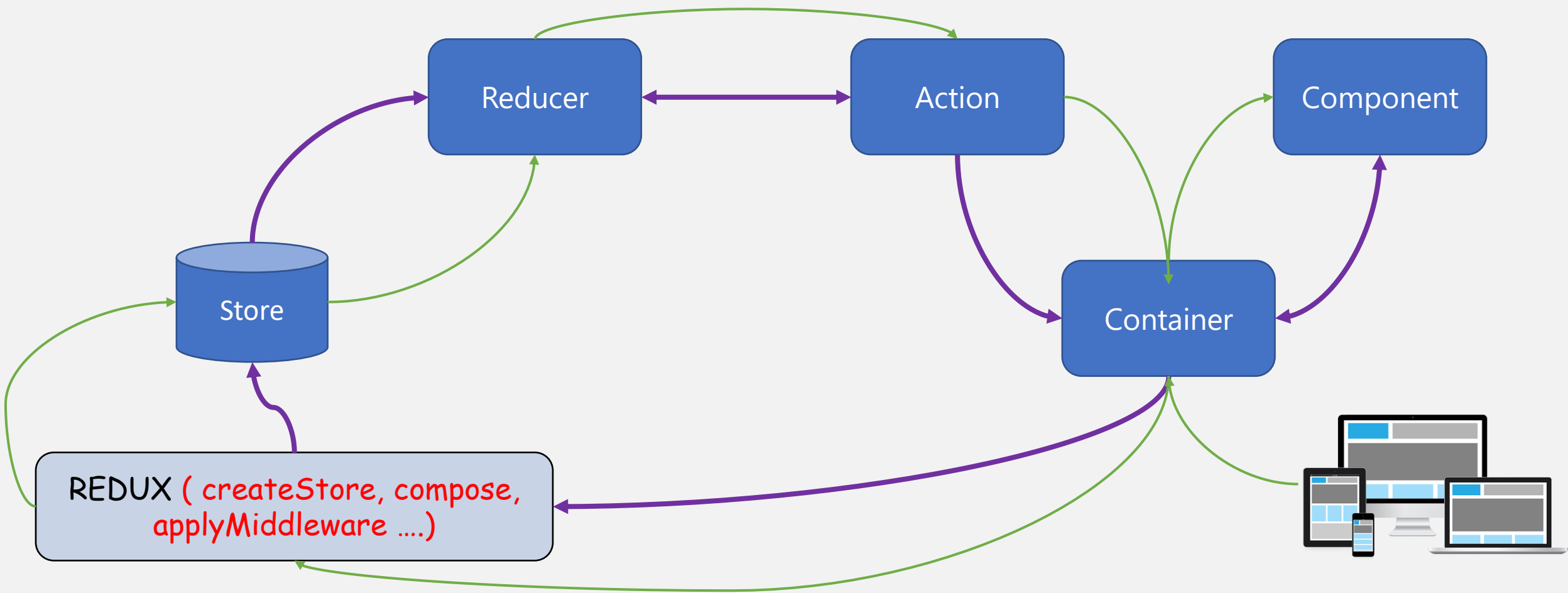


# Redux : En image 1/2





# Redux : En image 2/2





# A vous de jouer !



**{}** Etape 1 : Créer une nouvelle branche `feat/useRedux`

**{}** Etape 2 : Installer redux

- `npm install redux react-redux`
- `npm install npm install --save-dev redux-devtools`

**{}** Etape 3 : Création d'un provider dans le point d'entrée de l'application react (`index.js`)



MERCI