

# React

Animé par Mazen Gharbi


# Présentation



```
constructor() {  
  let user = new User();  
  user.name = 'Mazen GHARBI';  
  user.addSkills(['Angular', 'React', 'Vue', 'Node', 'PHP', 'Symfony', ...LIST_OTHERS]);  
  user.company = 'Hiracle';  
  user.email = 'mazen@hiracle.fr';  
}
```

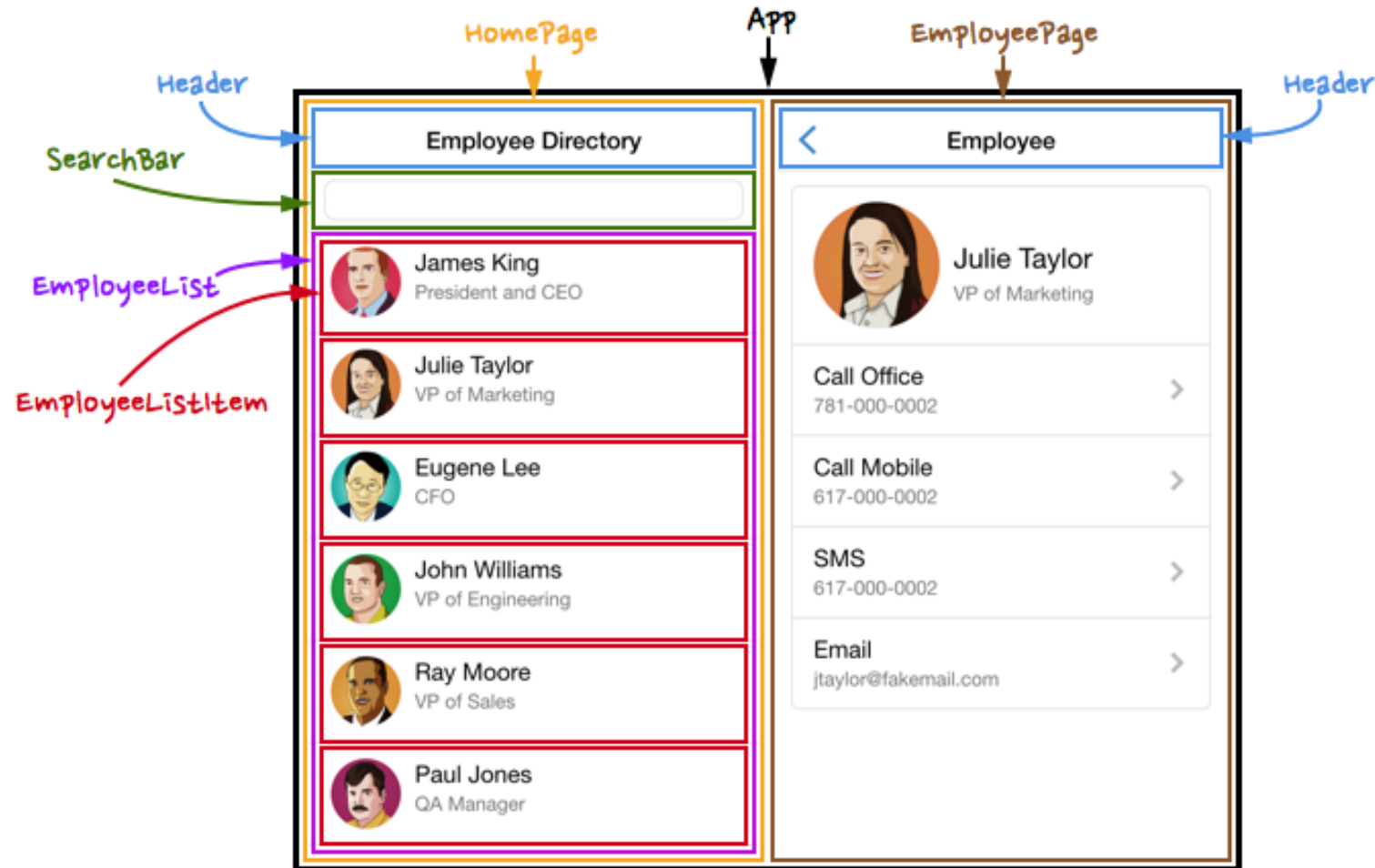


# Introduction

- ▷ React est une bibliothèque (et non un framework)
- ▷ Créée en 2013 par Facebook 
- ▷ Rapide à apprendre..
- ▷ Des milliers de librairies pour React créées par la communauté
- ▷ Orienté composant !



# Orienté composants



# JavaScript XML

▷ JSX est le langage permettant « d'écrire le HTML dans le JS » ;

```
function Button(props) {  
  
  const handleClick = () => {  
    props.clickHandler(props.value);  
  };  
  
  return (  
    <div className={'Button ' + props.class} onClick={handleClick}>  
      <div>  
        {props.value}  
      </div>  
    </div>  
  );  
}
```

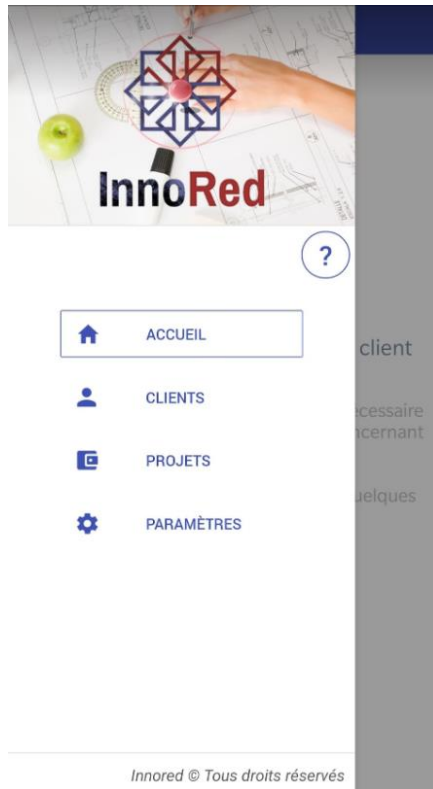
# Single Page Application

- ▷ Parfaitement adapté pour la mise en place de **SPAs**



- ▷ Une SPA est une application à **page unique**. Le contenu change **dynamiquement** sans requêter le serveur à chaque fois !

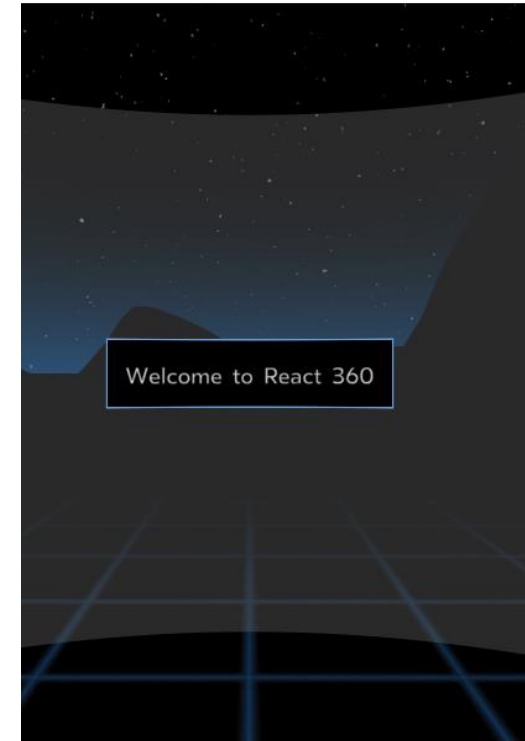
# Ce qu'on peut faire avec React



Applications mobiles



Sites webs

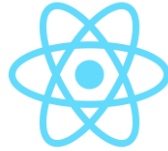


VR

Merci le Virtual DOM !

# Sommaire

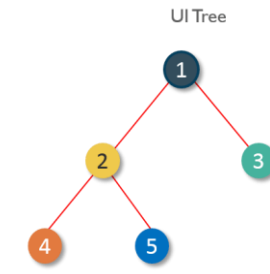
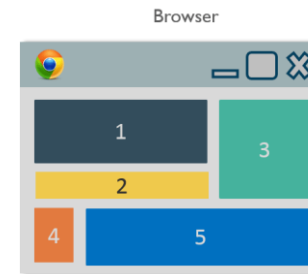
1. Introduction à React ;



2. Rappels Javascript ;

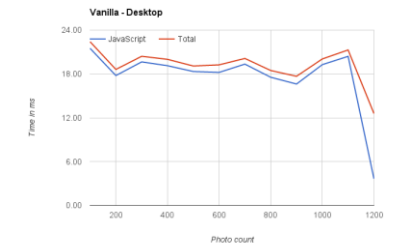


3. Création des premiers composants ;



4. React – Optimisez les performances de votre application ;

5. Les bibliothèques fondamentales React ;

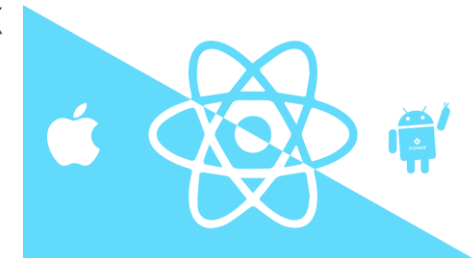


6. Stratégie de gestion de données avec Redux Toolkit ;



**Redux**

7. Introduction au développement mobile avec React ;







# Rappels Javascript

# Les objets

```
/* Create user object. */  
var user = { firstName: 'Foo' };  
  
/* Add `lastName` attribute. */  
user.lastName = 'BAR';  
  
/* Remove `firstName` attribute. */  
delete user.firstName;  
  
/* Change value */  
user.lastName = 'Foo';
```

# Clôner un objet

- ▷ Parfois, il peut s'avérer nécessaire de clôner un objet en JS
- ▷ Pour ce faire, on utilise la méthode « assign » de Object

```
var obj = {a: 1};  
var monClone = Object.assign({}, obj);
```

Objet vide dans lequel on souhaite clôner

- ▷ Il est également possible de clôner ET de modifier / ajouter une propriété en même temps

```
var obj = {a: 1, b: 3};  
var monClone = Object.assign({}, obj, {a: 2}); // {a: 2, b: 3}
```

# Les fonctions

```
var userName = function userName(user) {  
  return user.firstName + ' ' + user.lastName;  
};
```

```
var user = { firstName: 'Foo', lastName: 'BAR' };  
userName(user) // Foo BAR  
userName(user, 1, 2, 3, 4) // =>      Foo BAR  
userName() // =>      TypeError: Cannot read property...
```

# Closures

## Fermeture (informatique)

🔗 Pour les articles homonymes, voir [Closure](#) et [Fermeture](#).

Dans un [langage de programmation](#), une **fermeture** ou **clôture** (en anglais : ***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)<sup>1</sup>. 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.

Une fermeture peut être passée en argument d'une fonction dans l'environnement où elle a été créée (passée *vers le bas*) ou renvoyée comme valeur de retour (passée *vers le haut*). Dans ce cas, le problème posé alors par la fermeture est qu'elle fait référence à des données qui auraient typiquement été allouées sur la [pile d'exécution](#) et libérées à la sortie de l'environnement. Hors optimisations par le compilateur, le problème est généralement résolu par une allocation sur le [tas](#) de l'environnement.

### ▷ Pour faire simple :

- › Peut être appelé dans [n'importe](#) quel contexte ;
- › [Se souvient](#) du contexte dans lequel l'appel a été fait.

# Closures

```
var value = null;
```

```
setTimeout(function () {  
    value = 'value has been set';  
}, 100 /* 100 ms. */);
```

```
console.log(value); // => null
```

```
setTimeout(function () {  
    console.log(value); // => 'value has been set'  
}, 200);
```

```
(function (value) {  
    console.log(value); // => undefined  
})();
```

# Attention aux abus !

```
var lastInfo = null;

server.loadUser(function (user) {
  user.loadInfos(function (infos) {
    infos[0].save(function (info) {
      lastInfo = infos[0] = wish;
    });
  });
});
```

```
function register()
{
  if (!empty($_POST)) {
    $msg = '';
    if ($_POST['user_name']) {
      if ($_POST['user_password_new']) {
        if ($_POST['user_password_new'] === $_POST['user_password_repeat']) {
          if (strlen($_POST['user_password_new']) > 5) {
            if (strlen($_POST['user_name']) < 65 && strlen($_POST['user_name']) > 1) {
              if (preg_match('/^[a-z\d]{2,64}$/i', $_POST['user_name'])) {
                $user = read_user($_POST['user_name']);
                if (!isset($user['user_name'])) {
                  if ($_POST['user_email']) {
                    if (strlen($_POST['user_email']) < 65) {
                      if (filter_var($_POST['user_email'], FILTER_VALIDATE_EMAIL)) {
                        create_user();
                        $_SESSION['msg'] = 'You are now registered so please login';
                        header('Location: ' . $_SERVER['PHP_SELF']);
                        exit();
                      } else $msg = 'You must provide a valid email address';
                    } else $msg = 'Email must be less than 64 characters';
                  } else $msg = 'Email cannot be empty';
                } else $msg = 'Username already exists';
              } else $msg = 'Username must be only a-z, A-Z, 0-9';
            } else $msg = 'Username must be between 2 and 64 characters';
          } else $msg = 'Password must be at least 6 characters';
        } else $msg = 'Passwords do not match';
      } else $msg = 'Empty Password';
    } else $msg = 'Empty Username';
    $_SESSION['msg'] = $msg;
  }
  return register_form();
}
```



## WATERFALL SUICIDE

## PYRAMID OF DOOM

```
pan.pourWater(function() {
  range.bringToBoil(function() {
    range.lowerHeat(function() {
      pan.addRice(function() {
        setTimeout(function() {
          range.turnOff();
          serve();
        }, 15 * 60 * 1000);
      });
    });
  });
});
```

pyramid of doom

mozilla



# Les promesses

- ▷ Fonctionnalité EcmaScript 2015 !
- ▷ Permet d'éviter les callbacks successifs
- ▷ Un petit peu difficile à comprendre au premier abord.. Mais indispensable pour la mise en place d'un code propre !

```
const promise = new Promise(function (resolve, reject) {  
    resolve('Tout va bien');  
});  
  
promise.then(function (resultat) {  
    console.log(resultat); // Résultat de la promesse  
});
```

# Promesses

- ▷ Les promesses font désormais parti des fonctionnalités ES6.
- ▷ *Malheureusement les « promises » ES6 n'implémentent pas la méthode `finally`... maintenant si !*
- ▷ Les promises ne sont pas « lazy » ;
- ▷ Les promises ne sont pas annulables.

```
let promise = new Promise((resolve, reject) => {
  setTimeout(() => {
    resolve('Résultat positif');
  }, 1000);
});

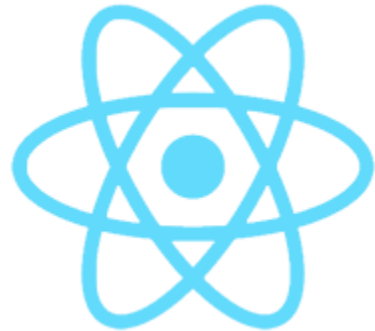
promise
  .then((res) => {
    console.log(res);

    return new Promise((resolve, reject) => resolve('Encore OK!'));
  })
  .then((res) => {
    console.log(res);

    return 'Primitive';
  })
  .then((res) => {
    console.log(res);
    throw new Error('On rentre dans le catch !');
  })
  .catch((err) => {
    console.error(err);
  });
```

# Quelques limites du langage

- ▷ Très permissif.. Trop.
- ▷ Pas adapté pour les grosses applications



React



- ▷ Pas d'introspection

# EcmaScript 6 / 2015

▷ Les classes !! Avec héritage

▷ Modules



▷  Arrow Functions

▷ { Template Strings }

▷ Spread & Rest

▷ Déstructuration (objet et array)

# Scope du mot-clé 'let'

```
let x = 1;

if (x === 1) {
  let x = 2;
  console.log(x); // => 2
}

console.log(x); // => 1
```


# Arrow functions

```
let maFonction = (a) => {  
  let result = a + 1;  
  return result;  
}  
  
console.log(maFonction(2));
```

---

```
var materials = ['Hydrogen', 'Helium', 'Lithium', 'Beryllium'];  
  
// Affichera [8, 6, 7, 9]  
console.log(materials.map(material => material.length));
```

# Template Strings

- ▶ Les templates string sont une nouvelle manière d'écrire vos chaînes de caractères
- ▶ On crée une template string avec « ` » (Alt Gr + 7 sur Windows) 

```
let a = 1;  
let maTemplateString = `  
    Passage à la ligne bien prit en compte  
    Evaluation de variable : ${a + 1} = 2  
`;  
;
```



# Destructuration d'array

```
let userList = [  
  {firstName: 'Foo'},  
  {firstName: 'Mads'}  
];
```

```
let [user1, user2] = userList;
```


```
console.log(user1); // { firstName: 'Foo' }  
console.log(user2); // { firstName: 'Mads' }
```

# Destructuration d'objet


```
let user = {  
  firstName: 'Foo',  
  lastName: 'BAR',  
  email: 'foo.bar@toto.com'  
};
```


```
let {lastName, firstName} = user;  
console.log(firstName); // Foo  
console.log(lastName); // BAR
```

# Spread & Rest

**function** *add*(...numbers) {  
**return** numbers.reduce((lastSum, num) => lastSum + num);  
}

**const** somme = *add*(1, 2, 3, 4);

**const** arr1 = [1, 2, 3];  
**const** arr2 = [...arr1, 4]; // [1, 2, 3, 4] 

**const** obj1 = {a: 1, b: 2};  
**const** obj2 = {...obj1, a: 3, c: 1}; // {a: 3, b: 2, c: 1} 

Spread !

# Modules

Depuis ES6, il est possible d'importer des fonctionnalités d'un fichier javascript vers un autre :

```
export class Personne {
  firstname;
}
```

personne.class.js

```
import { Personne } from './personne.class';
```

main.js

```
let pers = new Personne();
```

Obligatoire si  
pas de « default »

# Modules – export default

▷ Même cas de figure mais avec le mot-clé « default » cette fois :

Un seul default par fichier

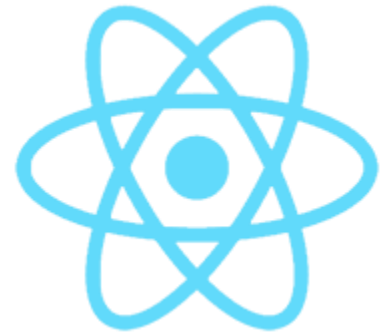
```
export default class Personne {  
  firstname;  
}
```

personne.class.js

```
import Toto from './personne.class';
```

main.js

```
let pers = new Toto();
```



# React

---

Premier projet React

# Introduction

- ▷ En React, on mélange JavaScript et HTML...
- ▷ Voici un exemple de code :

## Vue d'un programme React Native

```
{  
  hasError &&  
  <Text style={styles.error}>Une erreur est survenue</Text>  
}  
  
<Text style={{ fontSize: 17, marginTop: 10, color: '#C1C1C1', textAlign: 'center', padding: 10 }}>  
  { `Veuillez renseigner les informations de votre client ci-dessous` }  
</Text>
```

React Native

# Introduction

- ▷ Librairie créée en 2013 par Facebook, directement inspirée de la surcouche PHP XHP ;
  - › *extension de PHP pour permettre la syntaxe XML dans le but de créer des éléments HTML personnalisés et réutilisables*

## PHP classique

```
if ($_POST['name']) {  
?>  
    Hello, <?=$_POST['name']?>.  
} else {  
?>  
    <form method="post">  
        What is your name?  
        <input type="text" name="name">  
        <input type="submit">  
    </form>  
}
```

## Avec XHP

```
if ($_POST['name']) {  
    echo Hello, {$_POST['name']};  
} else {  
    echo  
    <form method="post">  
        What is your name?  
        <input type="text" name="name" />  
        <input type="submit" />  
    </form>;  
}
```



# Premiers pas

```
<!DOCTYPE html>
<html>
```

```
<head>
```

```
  <meta charset="utf-8">
```

```
  <meta name="viewport" content="width=device-width">
```

```
  <script src="https://unpkg.com/react@18.2/umd/react.development.js"></script>
```

```
  <script src="https://unpkg.com/react-dom@18.2/umd/react-dom.development.js"></script>
```

```
  <script src="https://unpkg.com/babel-standalone@6.15.0/babel.min.js"></script>
```

```
  <title>JS Bin</title>
```

```
</head>
```

```
<body>
```

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

```
  <script type="text/babel">
```

```
    const containerRoot = ReactDOM.createRoot(document.querySelector("#root"));
```

```
    containerRoot.render(
```

```
      <h1>Hello world!</h1>
```

```
    );
```

```
  </script>
```

```
</body>
```

```
</html>
```

index.html

1

2

3

Output

Run with JS

# Hello world!

# Comment ça marche

- ▷ Pour écrire une première page avec React, nous avons ajouté trois scripts: [React](#), [ReactDOM](#) et [Babel](#) ;
- ▷ La différence entre React et ReactDOM est que le premier gère le cœur de la librairie (composants, state, props) et le deuxième gère l'intégration avec les APIs DOM ;
- ▷ Cette séparation a été faite par les développeurs afin de permettre l'émergence de moteurs de rendu pour d'autres plateformes comme :
  - › [React Native](#) pour créer des applications mobiles ;
  - › [React Blessed](#) pour créer des interfaces sur le terminal ;
  - › [React VR](#) pour créer des sites web utilisant la VR ;

# Tout est JavaScript

▷ En React, le HTML n'est qu'une illusion !

main.js

```
const root = ReactDOM.createRoot(document.querySelector("#root"));

root.render(
  <h1>Hello world!</h1>
);
```

Babel réalise une « transpilation » de ce code !

```
root.render(React.createElement("h1", null, "Hello world!"));
```

C'est ce qu'on appelle un nœud virtuel

# Un langage bien différent

App.js

JSX

```
function App () {  
  return (  
    <div className="App">  
      <header className="App-header">  
        <img src={logo} className="App-logo" alt="logo" />  
        <h1 className="App-title">Welcome to React</h1>  
      </header>  
      <p className="App-intro">  
        To get started, edit <code>src/App.js</code> and save to reload.  
      </p>  
    </div>  
  );  
}
```

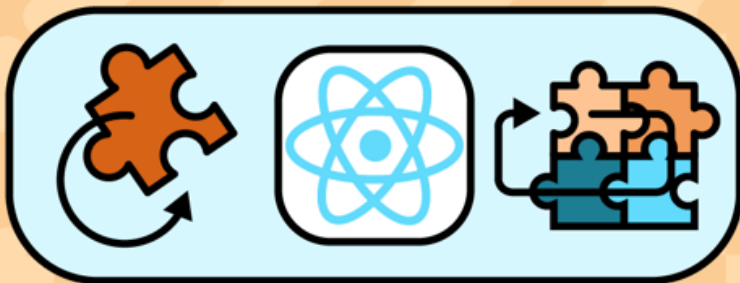
# Et voici l'équivalent en Javascript...

```
React.createElement(  
  "div",  
  { className: "App" },  
  React.createElement(  
    "header",  
    { className: "App-header" },  
    React.createElement("img", { src: logo, className: "App-logo", alt: "logo" }),  
    React.createElement(  
      "h1",  
      { className: "App-title" },  
      "Welcome to React"  
    )  
  ),  
  React.createElement(  
    "p",  
    { className: "App-intro" },  
    "To get started, edit ",  
    React.createElement(  
      "code",  
      null,  
      "src/App.js"  
    ),  
    " and save to reload."  
  )  
);
```

C'est quand même mieux en JSX

# JSX

- ▷ En React, la vue est **directement intégrée dans le modèle** ;
- ▷ Les développeurs ont poussé jusqu'au bout leur vision à avoir la **logique intimement reliée à un élément**.
- ▷ On parlera plus tard de composant **mono-fichier**



# JSX

▷ Si on veut écrire du JS dans notre composant, il faut l'entourer d'accolades :

JSX :

```
<h1 className="maClasse">  
  {"SaLUt l'éQUIPE !!".toLowerCase()}  
  {2 * 4}  
</h1>
```

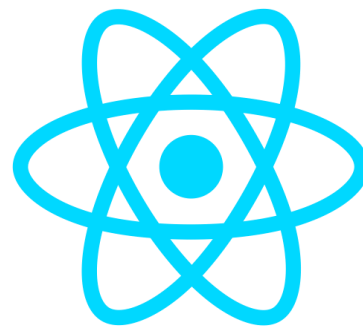
HTML :

```
<h1 class="maClasse">salut l'équipe !!8</h1>
```

# JSX

- ▷ Finalement le JSX est juste une manière plus simple d'écrire du HTML purement en JavaScript en évitant le fameux innerHTML
- ▷ De plus, vous pouvez être sûr que vos balises seront valides car le parseur JSX est stricte.
- ▷ Il évite aussi les failles XSS en échappant tous les caractères spéciaux





# Utiliser la React CLI

# Créer un projet React

- ▷ Pour simplifier le développement en ReactJS, nous allons utiliser « **create-react-app** »
- ▷ Aide à la création d'un projet React
- ▷ Ce script va installer ces outils :
  - › Webpack, outil de build pour le Front end
  - › ESLint, linter pour JavaScript
  - › Jest, solution de testing en JavaScript préconfigurée pour React
  - › Babel, préconfiguré pour transpiler du code ES6, JSX vers du ES5

# Create-react-app

- ▷ Génère une appli et un automatisateur de tâches
  - › Géré par Webpack comme vu précédemment
- ▷ Permet l'automatisation des tâches suivantes :
  - › Transpilation ES6 et JSX ;
  - › Serveur de développement avec rechargement de module à chaud ;
  - › Linting code ;
  - › Préfixe CSS ;
  - › Créer un script avec JS, CSS et regroupement d'images, et des sourcemaps ;
  - › Cadre de test Jest.

# Installation

- ▷ Tout d'abord, installez **create-react-app** globalement avec le gestionnaire de packages (npm) ;
  - › NodeJS est nécessaire, téléchargez-le [ici](#)
- ▷ Ouvrez votre terminal, naviguez (*cd*) vers le répertoire où vous souhaitez installer et tapez :

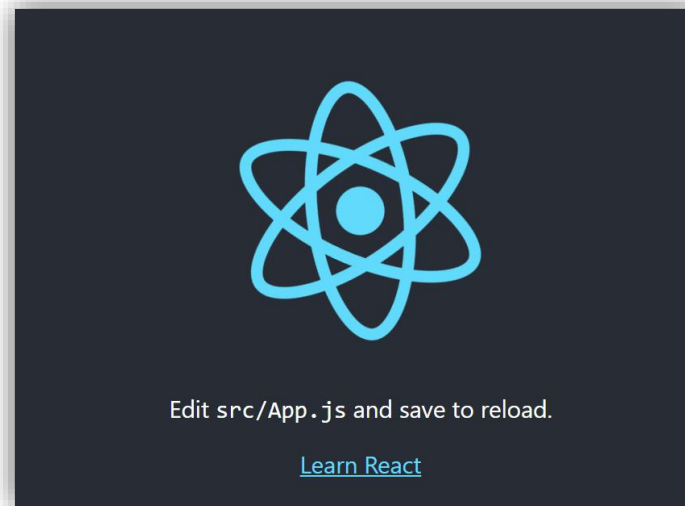
```
> npx create-react-app nom-de-mon-projet
```

# Lancez votre projet

- ▷ Attendez l'installation, entrez dans le dossier et lancez :

```
> npm start
```

- ▷ Et sous nos yeux ébahis



# Fichiers générés

reddit-like/

- node\_modules/ // Contient toutes les dépendances définies dans package.json
- **public/** // Fichiers statiques envoyés par le serveur (images / css / etc.)
- **src/** // Notre code source
- .gitignore
- package-lock.json // Sous dépendances
- package.json // Liste de nos dépendances ainsi que des commandes
- README.md

# Premier projet

- ▷ Modifier le fichier App.js de sorte à afficher la date dynamiquement cette fois-ci
- ▷ Utilisez l'API Date de JavaScript
  - › [Accéder à la documentation](#)
- ▷ L'affichage ne doit pas se remettre à jour à chaque seconde !
  - › Mais la date et l'heure doivent être correctes au moment où la page est chargée

# TP – Créez votre premier environnement

- ▷ Créez votre premier projet React ;
- ▷ L'objectif : Afficher la date d'aujourd'hui sous le format suivant :

Bonjour, nous sommes le 28/11/2022 et il est 09h45

- ▷ L'affichage de la date doit se faire **de manière dynamique** !



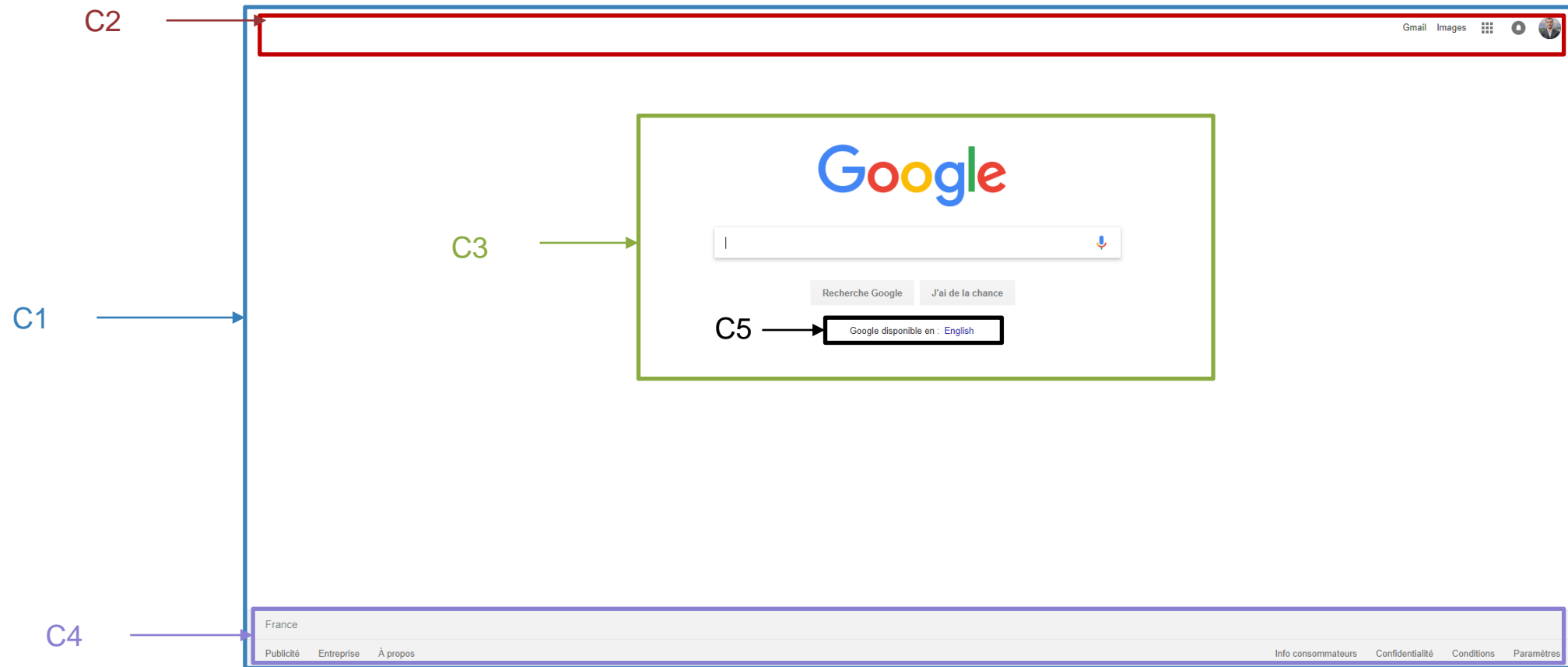


# Les composants

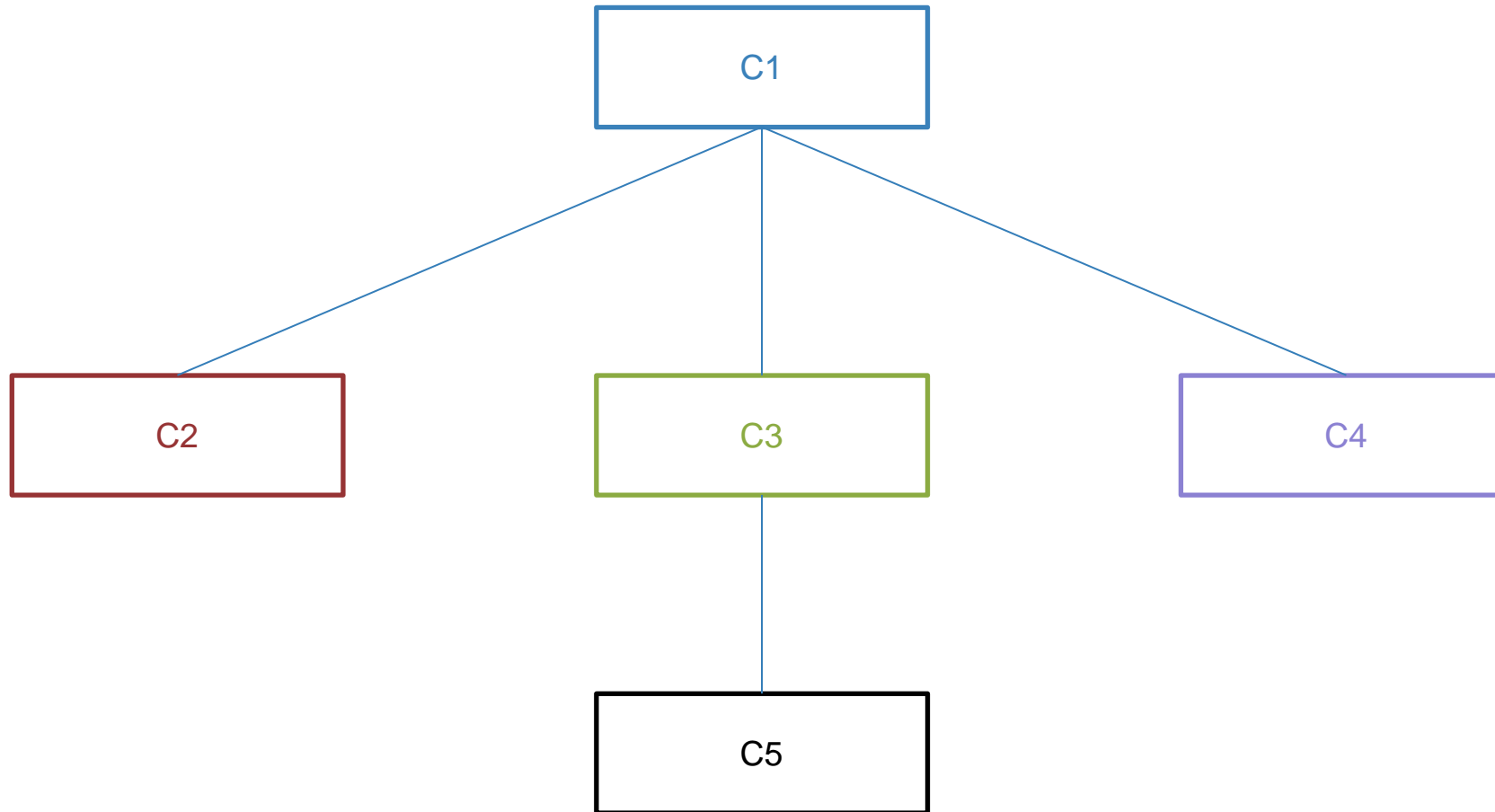
# Pilier de la librairie

- ▷ Un composant contrôle une vue ou une partie d'une vue
- ▷ L'un des principaux concepts de React est de voir une application comme une **arborescence de composants**.
- ▷ Les composants permettent une meilleure décomposition de l'application, facilitent le **refactoring** et le **testing**.
- ▷ Chaque composant est **isolé** des autres composants. Il n'hérite pas implicitement des attributs des composants parents.

# Séparation par composants



# Séparation par composants



# Différentes manières de faire

- ▷ Depuis **React 16.8**, une nouvelle manière de développer les composants est apparu
  - › Les **Hooks** !
- ▷ L'objectif de cette manœuvre : **Nous simplifier la vie !**
- ▷ L'inconvénient : **Des méthodologies différentes pour un résultat identique**
  - › Il convient donc de faire attention à ce que vous lirez sur internet
- ▷ Nous y reviendrons plus tard



# Créer et appeler un composant

Création du composant

HelloWorld.js

```
import React from "react";

function HelloWorld() {
  return <h1>Hello, World!</h1>;
}

export default HelloWorld;
```

Appel du composant

main.js

```
import HelloWorld from './Hello';

function App() {
  return (
    <HelloWorld />
  );
}

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

Fait le lien avec le index.html

- ▷ Le navigateur ne comprend pas le JSX. Cette syntaxe étant invalide.
- ▷ C'est Babel qui va « transformer » le JSX en code valide  
(transpiler)

# Functionnal Components React

- ▷ Un composant, c'est en fait une fonction

```
export default function HelloScreen() { ...
```

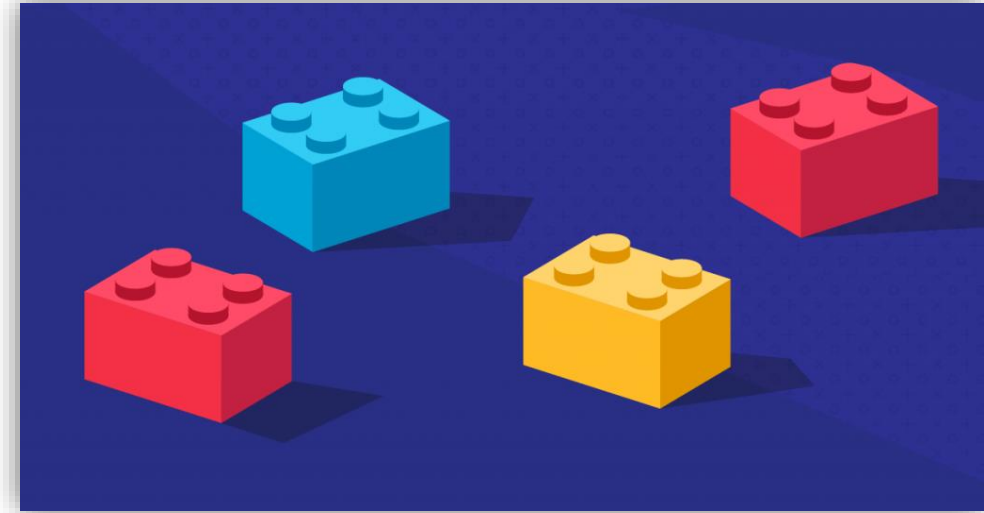
- ▷ Le retour de cette fonction correspond à la vue qu'elle représente

```
...  
return <div>Ma vue</div>;
```

- ▷ Ces fonctions peuvent contenir d'autres fonctions

# Passage de propriétés

- ▷ Un composant peut être configuré à l'aide de propriétés
- ▷ Celles-ci peuvent être apparentées à des paramètres de fonction



```
<QuiEstLeMeilleur color='red' />
```

**React c'est le plus mieux**

```
<QuiEstLeMeilleur color='blue' />
```

**React c'est le plus mieux**

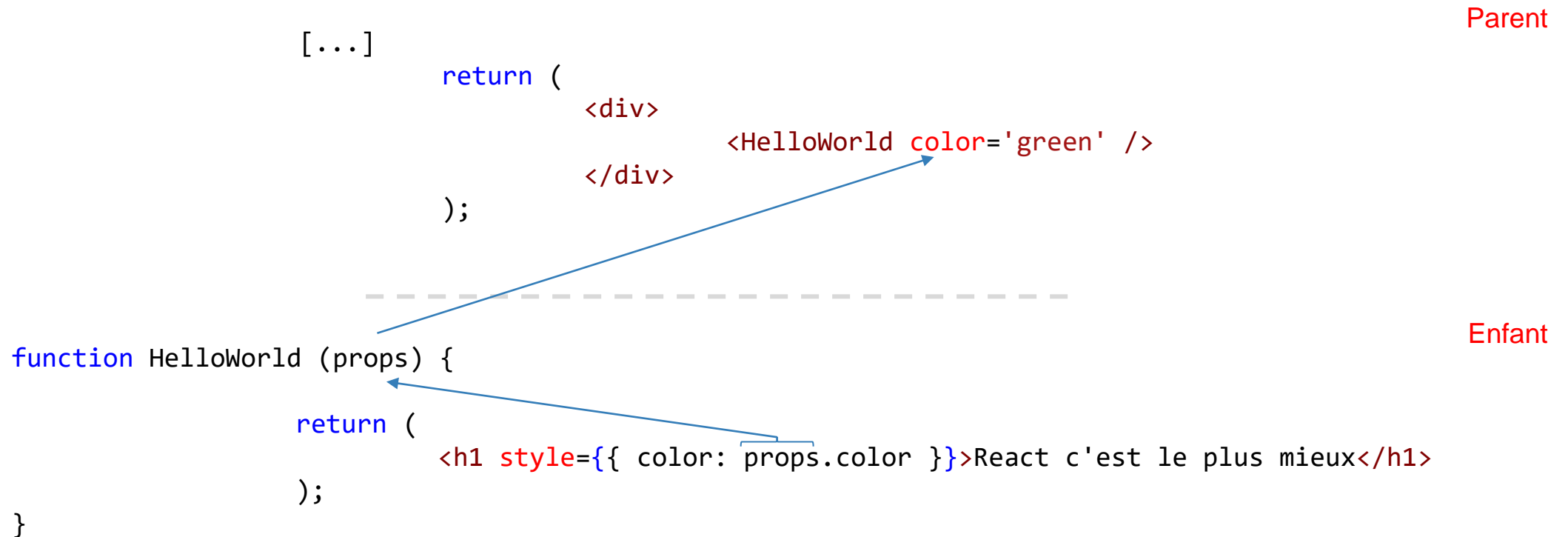
```
<QuiEstLeMeilleur color='green' />
```

**React c'est le plus mieux**



# Passage de propriétés

- ▷ Les propriétés d'un composant enfant sont accessibles au travers un objet « **props** »



# Props

▷ Seulement trois props sont **réservées** par React :

- › key: différencie les composants dans une liste à l'aide d'un identifiant unique (en savoir plus)
- › ref: permet de manipuler directement l'élément dans le DOM
- › children: liste des composants enfants,

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

← Bonne pratique, chaque clé doit être unique

# TP - Modulariser notre vue

- ▷ Créez un fichier « **DisplayTimer.jsx** »
  - › Le format jsx ne diffère en rien du format js, le contenu sera interprété pareillement. Seulement, cela permettra à votre IDE de reconnaître le format React et ainsi profiter des outils de l'éditeurs (indentation etc.)
- ▷ Le composant DisplayTimer doit prendre en paramètre **2 propriétés** :
  - › Le temps à afficher (*timeToDisplay*) ;
  - › Le prénom (*firstname*).
- ▷ Codez et appelez ce composant de sorte à obtenir cette vue :

Output

Run with JS

Auto-run JS



Bonjour Joe, nous sommes le 28/11/2022 et il est 09h45



# Etat d'un composant

```
✓ JSONSchema
1 {
2   "title": "A registration form",
3   "description": "A simple form example.",
4   "type": "object",
5   "required": [
6     "firstName",
7     "lastName"
8   ],
9   "properties": {
10    "firstName": {
11      "type": "string",
12      "title": "First name"
13    },
14    "lastName": {
15      "type": "string",
```

```
✓ UISchema
1 {
2   "firstName": {
3     "ui:autofocus": true
4   },
5   "age": {
6     "ui:widget": "updown"
7   },
8   "bio": {
9     "ui:widget": "textarea"
10  },
11  "password": {
12    "ui:widget": "password",
13    "ui:help": "Hint: Make it
14    strong!"
15  },
16 }
```

```
✓ formData
1 {
2   "firstName": "Chuck",
3   "lastName": "Norris",
4   "age": 75,
5   "bio": "Roundhouse kicking
6   asses since 1940",
7   "password": "noneed"
8 }
```

## A registration form

A simple form example.

**First name\***

**Last name\***

**Age**

**Bio**

**Password**

Hint: Make it strong!

# State

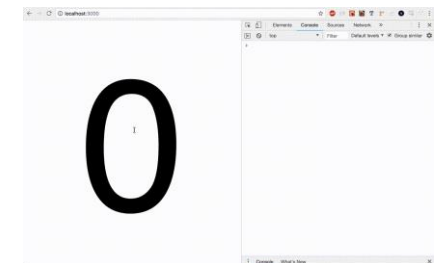
- ▷ Chaque composant peut avoir un état (ou **state**) qui lui est propre.
  - › Ce state n'est pas visible par les autres composants !
- ▷ Permet d'avoir de l'interactivité dans notre composant
- ▷ **Déclenche le réaffichage de son composant quand il est modifié**
  - › Doit être déclaré dans le constructeur du composant
- ▷ Equivalent à la propriété data dans VueJS ou \$scope dans AngularJS.

# State

- ▷ En React, la vue ne se met jamais à jour automatiquement
  - › Ceci pour être bien plus performant



- ▷ Pour provoquer un réaffichage, il est nécessaire de mettre à jour **l'état du composant**
- ▷ Chaque composant possède **un état unique** constitué de plusieurs variables
- ▷ La mise à jour d'une de ces variables provoquera automatiquement le réaffichage, sinon, rien ne se passe !



# Gestion de l'état avec useState

- ▷ « useState » est la méthode permettant de créer une variable d'état dans un composant

```
const [val, setVal] = useState("Valeur initiale");
```



On peut mettre les noms que l'on veut

- ▷ Renvoie un tableau contenant la valeur initiale et un setter
- ▷ Appeler le « setVal » provoque la réinterprétation de la vue
- ▷ Fonctionne également pour un objet

```
const [val, setVal] = useState({ prop: "Valeur initiale" });
```



# State

- ▷ Le state est mis à jour de manière asynchrone pour des raisons de performance. Il est modifié seulement avec la méthode `setState` d'un composant ;
- ▷ Il existe deux manières d'appeler `setState` :
  - › Statiquement: simple à écrire mais ne permet pas modifier correctement en mode batch
  - › Dynamiquement: plus verbeuse à écrire mais gère les cas plus complexes

[Plus d'infos ici](#)

# Mise à jour statique de l'état

```
function HelloWorld() {  
  const [monCompteur, setMonCompteur] = useState(0);  
  
  increment() {  
    for(let i = 0; i < 10; i++) {  
      setMonCompteur( monCompteur + 1 );  
    }  
  }  
  
  ...  
}
```

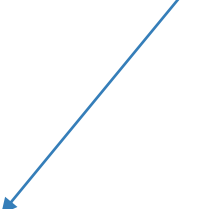
- ▷ La mise à jour d'un état est **asynchrone**
  - › Cela signifie que **c'est React qui décide** quand la mise à jour de la variable (et donc de la vue) se fera
- ▷ Dans l'exemple précédent, **monCompteur** vaudra 1 et non pas 10 comme prévu...

# State

- ▷ Comme dit précédemment, `setState` est exécutée de manière asynchrone ;
- ▷ Pour éviter notre problème, on préférera cette écriture :

```
function HelloWorld() {  
  const [monCompteur, setMonCompteur] = useState(0);  
  
  increment() {  
    for(let i = 0; i < 10; i++) {  
      setMonCompteur( (lastValueCompteur) => lastValueCompteur + 1 );  
    }  
  }  
  
  ...  
}
```

C'est désormais un callback



# State - Affichage

Variable d'état 😊

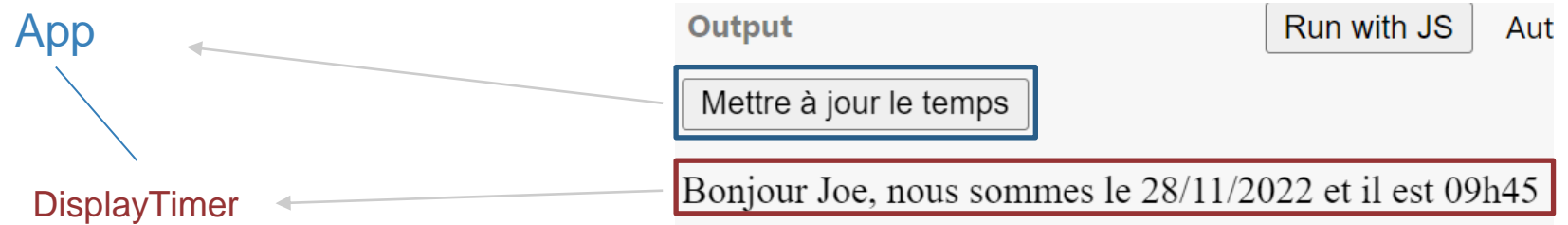
```
// La méthode onClick réagit au click de l'utilisateur sur l'élément
// Attend une référence de fonction !
return (
  <div>
    <h1>Valeur actuelle du compteur : {value}</h1>
    <p onClick={increment}>Cliquez ici pour incrémenter la valeur</p>
  </div>
)
```

Peut être appliqué sur n'importe quel élément

*Un composant enfant n'hérite pas du state père !*

# TP – Dynamiser notre contenu

- ▷ Voici notre architecture de composants actuelle :



- ▷ Avec 2 propriétés envoyées du père vers l'enfant
- ▷ Créez un état dans **App** correspondant au temps actuel
- ▷ Ajoutez un bouton dans la vue de **App** pour mettre à jour le temps
  - › Vérifiez que le temps se mette correctement à jour dans la vue de **DisplayTimer**

# Smart & Dumb

- ▷ Il existe 2 catégories de composants : les **Smart** et les **Dumb**
- ▷ Non obligatoire mais permet d'avoir une **structure pérenne**
- ▷ C'est l'une des architecture la plus **importante** et **difficile** à mettre en œuvre durant vos développements React.

## SMART

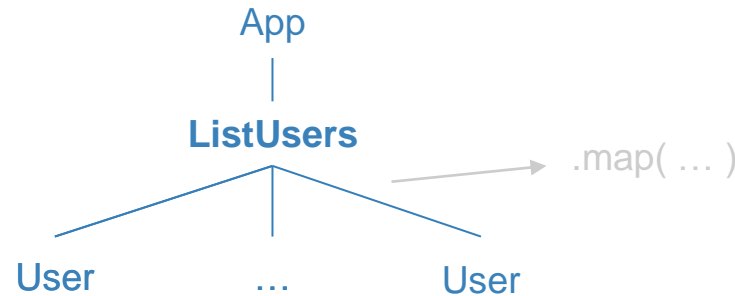
- Gère la « **business logic** » ;
- Gère l'**algorithmique** ;
- Gère la **donnée** ;
- Réalise les appels serveurs ;
- **N'est pas réutilisable.**

## DUMB

- Réalise l'affichage ;
- Délègue les traitements au père ;
- Possède beaucoup de propriétés ;
- **Est ré-utilisable.**

# Mise en place de composants Smart & Dumb

▷ Soit une architecture comme celle-ci permettant de gérer l'âge de nos utilisateurs :



▷ Avec le code suivant :

```

const users = [
  { id: 0, firstname: 'Joe', age: 42 },
  { id: 1, firstname: 'Youssef', age: 28 },
  { id: 2, firstname: 'Toto', age: 33 },
]

function ListUsers() {
  return <div>
    {
      users.map(u => <User firstname={u.firstname} age={u.age} />)
    }
  </div>;
}
  
```

ListUsers.jsx

```

function User({ firstname, age }) {
  const [ageToDisplay, setAgeToDisplay] = useState(age);

  return (<div>
    <p>{firstname} a {ageToDisplay} an(s)</p>
    <button onClick={() => setAgeToDisplay(ageToDisplay + 1)}>
      Souhaitez l'anniversaire
    </button>
  </div>);
}

export default User;
  
```

Users.jsx

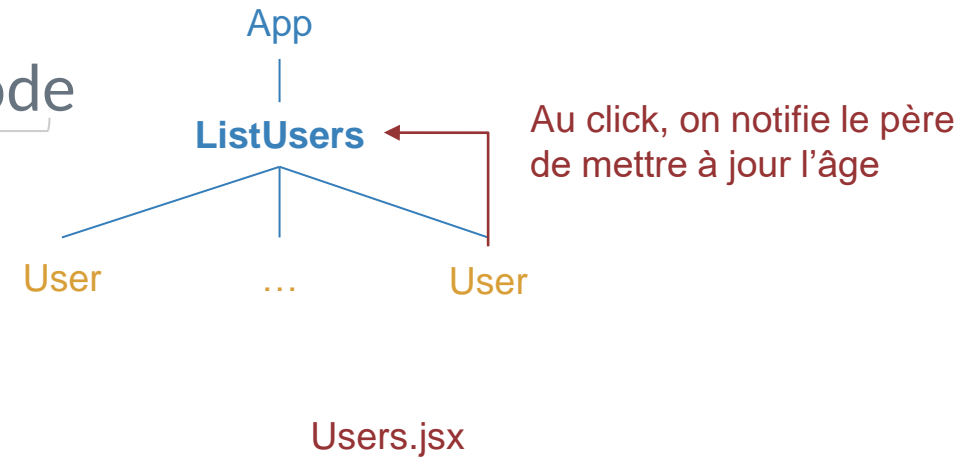
**MAUVAISE PRATIQUE !**

export default ListUsers;

# Mise en place de composants Smart & Dumb

► Pour respecter les bonnes pratiques, c'est **ListUsers** qui doit gérer l'incrémentation de l'âge

1. **ListUsers** envoie en paramètre une méthode
2. L'**enfant** l'appelle pour notifier le **père**
3. Le **père** modifie l'état qui se propage



```
import React from "react";

function User({ firstname, age, incrementAge }) {
  return (
    <div>
      <p>{firstname} a {age} an(s)</p>
      <button onClick={incrementAge}>Souhaitez l'anniversaire</button>
    </div>
  );
}

export default User;
```



# Mise en place de composants Smart & Dumb

```
import React, { useState } from "react";
import User from './User';
```

```
function ListUsers() {
  const [users, setUsers] = useState([
    { id: 0, firstname: 'Joe', age: 42 },
    { id: 1, firstname: 'Youssef', age: 28 },
    { id: 2, firstname: 'Toto', age: 33 },
  ]);
```

```
  function updateAge(index, newAge) {
    const newUsers = users.map((u, i) => {
      if(index === i) {
        return { ...u, age: newAge };
      }

      return u;
    });
    setUsers(newUsers);
  }
```

(suite du code)

```
    return (<div>
      {
        users.map((user, index) => {
          return <User firstname={user.firstname}
            incrementAge={() => updateAge(index, user.age + 1)}
            age={user.age} />
        })
      }
    </div>);
  } // Fin de la fonction (component ListUsers)
```

```
  export default ListUsers;
```

# Explications

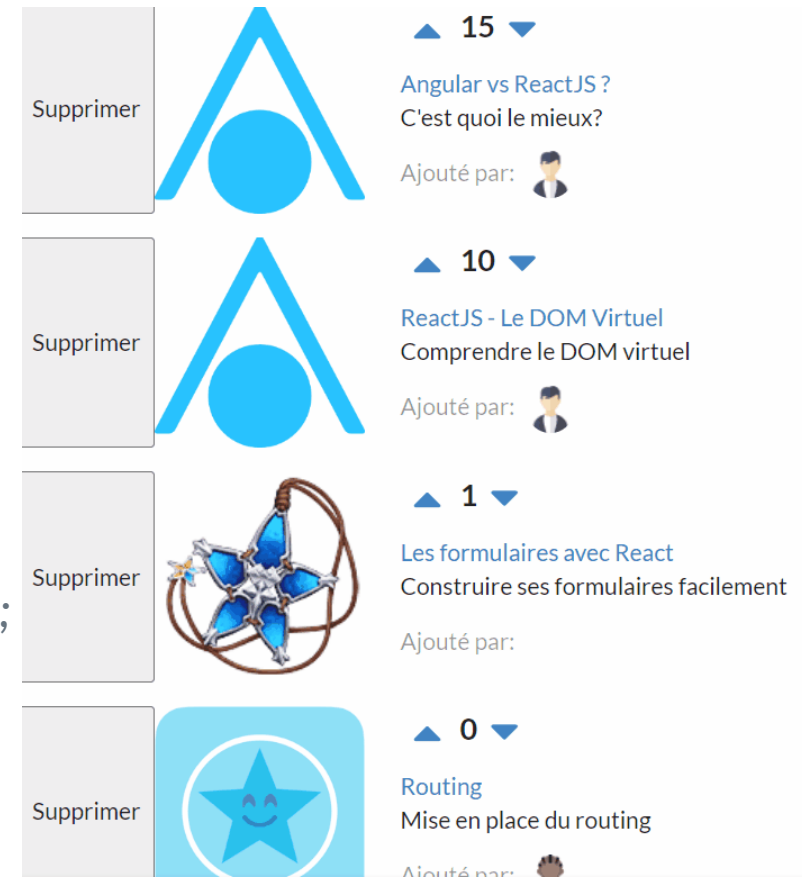
- ▷ Souvenez-vous, un état doit être **IMMUABLE**
- ▷ Cette écriture ne fonctionnera pas !

```
function ListUsers() {  
  const [users, setUsers] = useState([  
    { id: 0, firstname: 'Joe', age: 42 },  
    ...  
  ]);  
  
  function updateAge(index, newAge) {  
    users[index].age += 1; // PROBLEME ICI  
  }  
  ...  
}
```

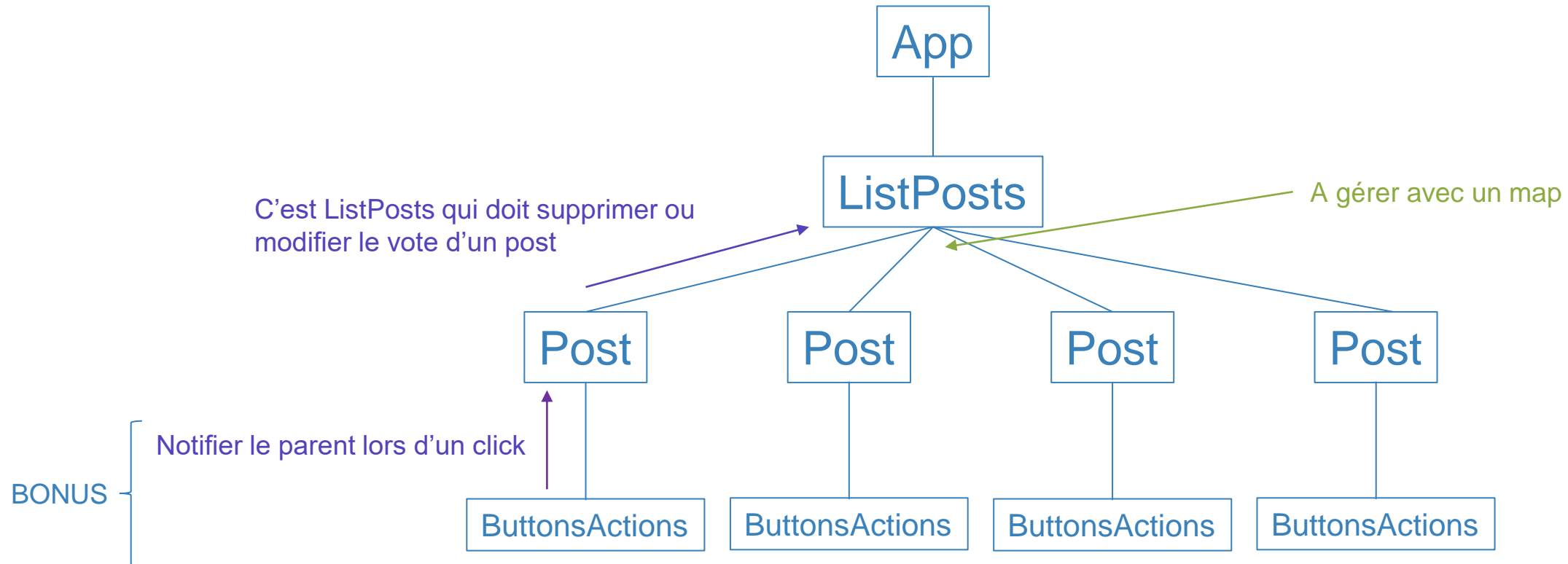
- ▷ On recrée donc entièrement le tableau en recréant uniquement l'objet à modifier
  - › Cela nous assure que l'état précédent ne sera jamais modifié

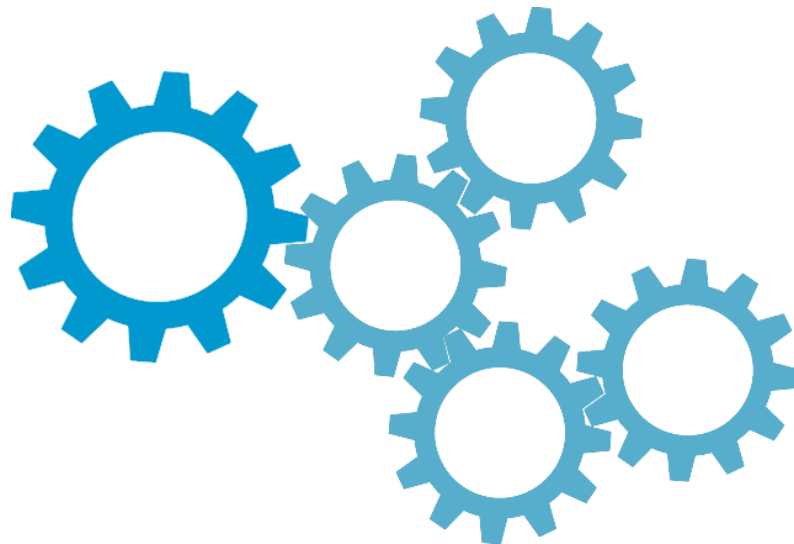
# Un projet aux grandes ambitions

- ▷ Créons un petit forum 😊
- ▷ Rajoutez un « downvote » ;
- ▷ Créez un composant enfant permettant de gérer les boutons de votes ;
- ▷ Ne pas afficher les posts qui ont moins de 0 upvotes ;
- ▷ (Bonus) Ajouter un bouton (*nouveau composant*) permettant de supprimer un post ;
- ▷ (Bonus) Affichez en vert les posts avec plus de 10 likes et en rouge ceux avec moins de 2 likes



# Architecture de composants

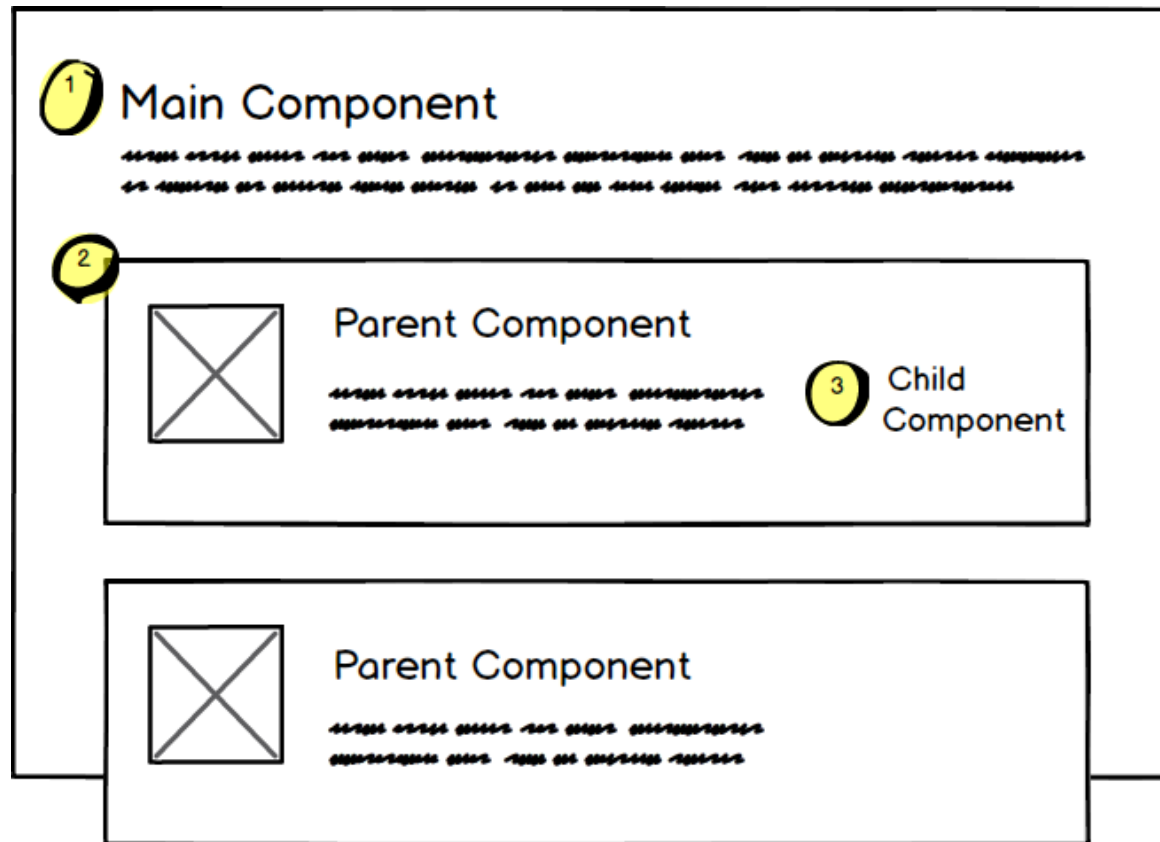




# Les bibliothèques fondamentales

# Comprendre les composants

*“ Not only do React components map cleanly to UI components, but they are self-contained. The markup, view logic, and often component-specific style is all housed in one place. This feature makes React components reusable. ”*

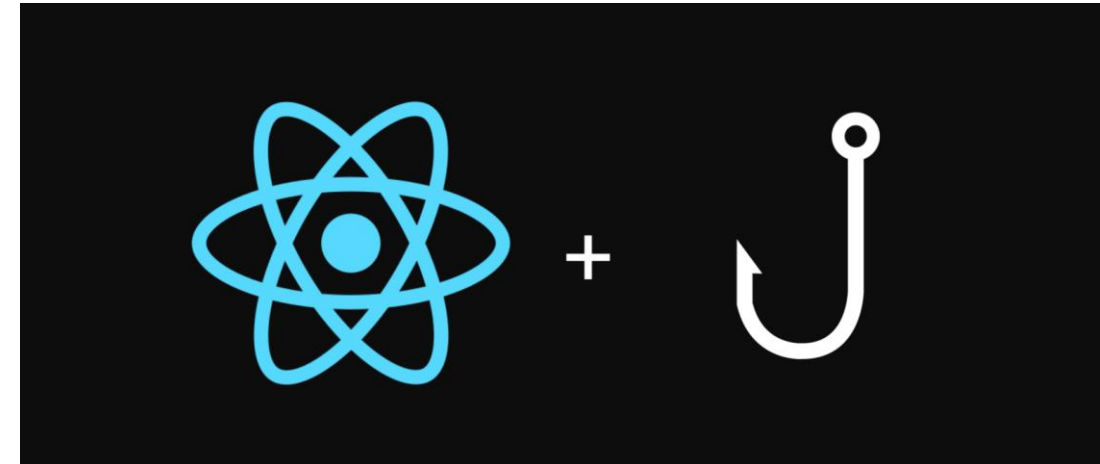


# Les bonnes pratiques ont changées

- ▷ Désormais, il s'agira d'implémenter des « functional component » **TOUT le temps !**
- ▷ Simplifie le développement
  - › Evite les méthodes alambiquées (componentDidMount / etc.)
  - › Solutionne le problème du binding this
  - › Plus accessible pour un débutant React
- ▷ Mais la **gestion du state évolue.**

# Les hooks

- ▷ Avec les composants fonctions, plus de « `this.state` »
  - › Et plus de mot-clé `this` avec le comportement étudié précédemment !
- ▷ Pour les remplacer, l'équipe React a intégrée les « Hooks »
- ▷ Un hook permet de gérer un état localisé
- ▷ On peut avoir plusieurs hooks !
- ▷ Nouvelle méthode : « `useState` »
  - › Renvoie 2 valeurs !





# Exemple de hook

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

function Example() {
  // Declare a new state variable, which we'll call "count"
  const [count, setCount] = useState(0);

  return (
    <div>
      <p>You clicked {count} times</p>
      <button onClick={() => setCount(count + 1)}>
        Click me
      </button>
    </div>
  );
}
```

# Hooks d'effets

- ▶ Avec les class components, nous avions des méthodes permettant de gérer le cycle de vie du composant actuel

```
class MyComp extends React.Component {  
  componentDidMount() {}  
  
  componentDidUpdate() {}  
  
  componentWillUnmount() {}  
}
```

- ▶ Cette logique a été remplacée par les hook d'effet :

```
const [val, setVal] = useState("Ma valeur initiale");  
  
// Permet de réagir à la création du composant ET à chaque modification des variables  
useEffect(() => {  
  console.log("Hello ! Vue réinterprétée :)");  
});
```

# Hooks d'effets - Nettoyer la mémoire

- ▷ En retournant une fonction, nous pouvons appliquer un comportement à la « mort » du composant :

```
useEffect(() => {  
  return () => {  
    // Code à exécuter lors de la mort de l'effet  
  };  
}, []);
```

- ▷ Un second paramètre permet de spécifier à quelles variables on souhaite réagir

```
export default function App({ color }) {  
  useEffect(() => {  
    return () => {  
      console.log('La couleur a changée !');  
    };  
  }, [color]);  
}
```

# « useEffect » est réexécuté à chaque fois !

▷ Ce qui signifie qu'un code comme celui-ci :

```
useEffect(() => {  
  UsersAPI.connect(onUserUpdate, props.user.id);  
  return () => {  
    UsersAPI.disconnect(onUserUpdate);  
  };  
});
```

▷ Va provoquer l'abonnement et le désabonnement au service à chaque réinterprétation... Pas toujours le meilleur choix !

› React évite ainsi certains bugs si les props changent entre temps

```
useEffect(() => {  
  UsersAPI.connect(onUserUpdate, props.user.id);  
  return () => {  
    UsersAPI.disconnect(onUserUpdate);  
  };  
}, [props.user.id]);
```

→ Bien mieux pour les performances !

# Construire nos Hooks personnalisés

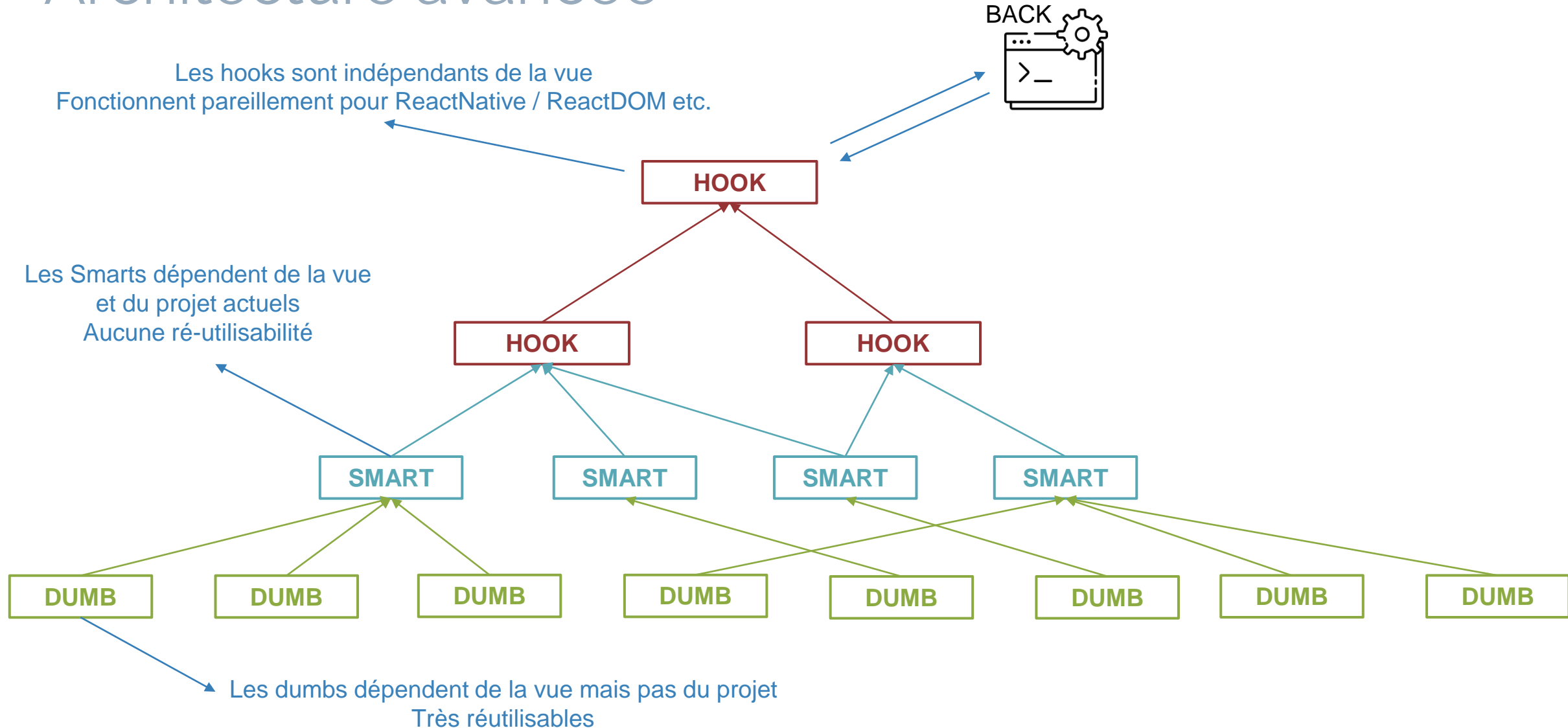
▷ Comme énoncé au départ, objectif : réutilisabilité !

```
function useLoadUsers() {  
  const [users, setUsers] = useState(null);  
  
  useEffect(() => {  
    const subscription = UserAPI.loadUsers().subscribe((users) => {  
      setUsers(users); // Provoque la réinterprétation de la vue  
    });  
  
    return () => {  
      subscription.unsubscribe();  
    };  
  }, []); // [] => Permet d'enclencher le hook uniquement 1 fois au chargement du composant  
  
  return users;  
}
```

---

```
function DisplayUsers(props) {  
  const users = useLoadUsers();  
  
  return users ? 'Aucun utilisateur' : `Il y a ${users.length} utilisateurs`;  
}
```

# Architecture avancée



# PropTypes

- ▷ Permettent de définir les types des propriétés attendues
  - › Non obligatoires mais vivement recommandées ;
- ▷ Vérification qui facilite la collaboration entre différents développeurs
  - › Affiche un **warning** dans la console si une propriété ne respecte par la propTypes définie ;
- ▷ N'impactent pas les performances de votre application en prod
  - › Car effacées lors de la création de la phase de build

[En savoir plus ici](#)

# PropTypes

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

export default function Hello(props) {
  const { firstname, lastname } = props;
  return (
    <div>
      Bonjour {firstname} {lastname}, comment tu vas ?
    </div>
  );
}

Hello.propTypes = {
  firstname: PropTypes.string.isRequired,
  lastname: PropTypes.string
};
```

Destructuration d'objet

Permet d'indiquer que ce champs est requis



# Children

## Souvenez-vous

- ▷ « Seulement trois props sont réservées par React :
  - key: différencie les composants dans une liste à l'aide d'un identifiant unique ([en savoir plus](#))
  - ref: permet de manipuler directement l'élément dans le DOM
  - children: liste des composants enfants »
- ▷ Il est possible d'entrer du contenu dans le corps d'appel d'un component afin de le récupérer par la suite via la propriété children

# Children

```
function FancyBorder(props) {  
  return (  
    <div className={"FancyBorder FancyBorder-" + props.color}>  
      {props.children}  
    </div>  
  );  
}  
-----  
function Dialog(props) {  
  return (  
    <FancyBorder color="blue">  
      {  
        <h1 className="Dialog-title">{props.title}</h1>  
        <p className="Dialog-message">{props.message}</p>  
      }  
    </FancyBorder>  
  );  
}
```

Nos childrens

## TP - Rendre clean notre code

- ▷ Récupérez votre code du **TP « Forum »** ;
- ▷ Le père « ListPosts » doit désormais envoyer 2 propriétés supplémentaires à « Post » :
  - › **color** – Chaîne de caractère permettant de définir la couleur des titres ;
  - › **uppercase** – Boolean permettant de savoir si les éléments doivent s'afficher en majuscule ou pas
- ▷ Votre mission : **Implémenter les propriétés et les fonctionnalités associées & mettre en place la logique *PropTypes* !**

# Construire son projet

- ▷ On va essayer de créer des composants de type « **data-driven** » :
  - › Pour y arriver, on maximisera l'utilisation des « props » ;
  - › Facilitera ainsi la **ré-utilisabilité**
- ▷ Les props vous permettent également de passer des références de fonction

The diagram illustrates a transformation in JSX syntax. On the left, a JSX element `<Product>` is shown with multiple individual props: `key`, `id`, `title`, `description`, `url`, `votes`, `submitterAvatarUrl`, `productImageUrl`, and `onVote`. A horizontal blue arrow points from this element to the right, crossing a vertical dashed line. On the right side of the line, the same `<Product>` element is shown, but with a more concise syntax: the first seven props are grouped into a single `{...product}` spread prop, and the `onVote` prop remains as is. The closing tag `</>` is also present on the right.

```
<Product
  key={"product-" + product.id}
  id={product.id}
  title={product.title}
  description={product.description}
  url={product.url}
  votes={product.votes}
  submitterAvatarUrl={product.submitterAvatarUrl}
  productImageUrl={product.productImageUrl}
  onVote={this.handleProductUpVote}
/>
```

```
<Product
  key={"product-" + product.id}
  {...product}
  onVote={this.handleProductUpVote}
/>
```



# Routing

# Deep Linking

**https://www.m2g-intellect.fr/admin/users.php**

Protocole      Nom de domaine      Chemin vers la ressource

-----

- ▶ Habituellement, le « **pathname** » correspond au chemin vers la ressource que l'on souhaite récupéré du serveur ;
- ▶ Nous travaillons actuellement sur une Single Page Application ;
  - › Nous n'accéderons plus à des ressources serveur directement ;
- ▶ Mais un système de routing est tout de même nécessaire
  - › Partager une page du site ;
  - › Recharger la page et rester sur la même vue *etc.* ;

# Routes

- ▷ Le système de routing en React est une sorte de « *mémoire d'état externe* »
  - › Il sert par exemple à ce qu'un utilisateur enregistre la page actuelle en favori
- ▷ Comme toute nouvelle fonctionnalité, il est nécessaire d'installer une librairie pour manipuler le routing :

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

- ▷ Meilleure librairie du moment pour la gestion des routes

Le serveur n'est pas requêté aux changements de pages en React !

# Mise en place

- ▶ Pour mettre en place les routes, il va être nécessaire de définir un **Context Route** autour de composant principal :

```
import { BrowserRouter as Router } from 'react-router-dom';  
  
<Router>  
  <App />  
</Router>
```

Alias utilisé

Composant racine

- ▶ Ainsi, le système de routing est initialisé et on pourra :
  - › Définir nos routes ;
  - › Naviguer entre nos pages.




# Déclaration des pages

- ▷ En React, chaque page est définie par un composant

```
<Routes>  
  // On est actuellement dans le composant App  
  <Route path='/accueil' element={<HomeComponent />} />  
  <Route path='/utilisateurs' element={<UsersComponent />} />  
  <Route path='/details' element={<DetailsComponent />} />  
</Routes>
```

Composants créés  
au préalable



- ▷ Chaque page est représentée par un composant

# Naviguer entre les routes

- ▷ Pour la redirection, « react-router-dom » fournit un composant « **Link** »
- ▷ Un attribut « to » permet de spécifier vers quel route on souhaite rediriger :

```
<Link to="/utilisateurs">Voir les utilisateurs du site :)</Link>
```

- ▷ Peut également être un objet :

```
<Link  
to={{  
  pathname: '/courses',  
  search: '?sort=name',  
  hash: '#the-hash',  
}}  
state={{ fromDashboard: true }}  
>;
```

Route visé

Paramètres GET

Ancre

Envoyer un état dans le composant de la route sur laquelle on arrive

# Routes

app.js

```
function App () {  
  return (  
    <Router>  
      <Routes>  
        <Menu />  
        <Route path='/accueil' element={<Accueil />} />  
        <Route path='/contact' element={<Contact />} />  
        <Route path='/forum' element={<Forum />} />  
      </Routes>  
    </Router>  
  );  
}
```

# Routes imbriquées

App.js

```

<Routes>
...
  <Route path='/accueil' element={<Accueil />} >
    <Route path='enfant1' element={<RouteEnfant1 />} />
    <Route path='enfant2' element={<RouteEnfant2 />} />
  </Route>
</Routes>

```

On déclare toutes les routes enfants comme children de la route parent

Accueil.js

```

return (
  <div>
    Contenu de le page Accueil -
    <Link to='/accueil/details'>
      <a> Détails</a>
    </Link>
    <Outlet />
  </div>
);

```

Provoque l'affichage des routes enfants

# Paramètres de route

- ▷ Les paramètres d'URL sont des paramètres dont les valeurs sont dynamiques dans l'URL ;
- ▷ Un exemple pratique serait les pages de profil de Twitter. Si rendu par React Router, cette route peut ressembler à cela :

```
import Profile from './pages/profile.component';
```

```
<Route path('/:handle' element={Profile} />
```



Le « : » indique que c'est un paramètre de route qui porte le nom « handle »

# Paramètres de route

```
import React, { useState } from 'react';
import { useParams } from 'react-router';

function Profile () {
  const [user, setUser] = useState(null);

  // Récupération du paramètre de route
  const { handle } = useParams();

  // fetch permet de requêter un serveur
  fetch(`https://api.twitter.com/user/${handle}`).then((user) => {
    setUser((previousState) => user)
  });

  return (
    ...
  )
}
```

C'est ici que tout se joue

# Paramètres URL

▷ Voici un exemple : <https://stackblitz.com/edit/macademia-react-router-params>

Dans render()

```
<Routes>
  <Menu />
  <Route exact path="/" element={<Accueil />} />
  <Route path="/contact/:id" element={<Contact />} />
  <Route path="/forum" element={<Forum />} />
</Routes>
```

app.js

---

```
return (
  <div>
    Contenu de la page Contact { useParams().id }
  </div>
)
```

Tout est là

contact.js

# Rediriger avec les hooks !

▷ <https://stackblitz.com/edit/react-router-avec-hooks>

```
export default function App() {  
  const nav = useNavigate();  
  const location = useLocation();  
  
  return (  
    <div>  
      <p>Route actuelle : {location.pathname}</p>  
    <ul>  
      <button onClick={() => nav('/home')}>Home</button>
```

---

```
<Route path="/home" component={Home} />  
<Route path="/contact/:name" component={Contact} />
```

```
export default function Contact() {  
  const { name } = useParams();  
  ...
```



# Lazy loading

- ▷ Chargement **différé** des pages
  - › Amélioration nette des performances
- ▷ **React.lazy()** permet de charger dynamiquement un composant

```
const UserLazy = React.lazy(() => import('./pages/User'));
```

- ▷ **Suspense** permet de suspendre le rendu jusqu'à ce que le composant soit prêt à être rendu.

```
<Suspense fallback={<div>Veuillez patienter.. On charge la page !</div>}>  
  <Routes>  
    <Route path="user/:userId" element={<UserLazy />} />  
  </Routes>  
</Suspense>
```

## TP - Login + Home

- ▷ Mettez en place un simple **page Login** avec pour seul contenu un bouton « Me connecter »
- ▷ Au click, vous aurez la responsabilité de rediriger l'utilisateur vers une nouvelle page **Home**
- ▷ Dans la page Home, un lien nous permettra de **revenir vers la page Login**
- ▷ *(Bonus) Mettez en place un système de routing imbriqué avec l'envoi de paramètre de la route père vers la route enfant*



# Formulaire

# Formulaires en React

- ▷ Nos formulaires en React seront gérés à travers nos composants ;
- ▷ Il existe 2 types de composants pour les gérer :
  - › **Controlled** components ;
  - › **Uncontrolled** components.
- ▷ Quelle différence ?

# Controlled component

- ▷ Un « **Controlled Component** » est un composant sur lequel nous choisissons nous-même la valeur qui apparaîtra :

```
function ControlledInput () {  
  
    return <input type="text" value="valeur de l'input"/>  
  
}
```

ControlledInput.js

- ▷ Si l'utilisateur insère une nouvelle valeur dans l'input, rien ne se passera. Afin de contourner ce problème, on fait en sorte que le composant React écoute les changements de l'utilisateur pour mettre à jour la valeur

# Controlled component

▷ Voici ce qu'on obtiendra :

ControlledInput.js

```
function ControlledInput {  
  const [value, setValue] = useState('');  
  
  function onChange(event) {  
    setValue(event.target.value);  
  }  
  
  return <input  
    type="text"  
    value={value}  
    onChange={onChange}  
  />  
}
```

Le state décide du contenu de l'input

Evenement React, réagit aux changements sur l'input

# Uncontrolled component

- ▷ Un « *Uncontrolled Components* » est un composant sur lequel la mise à jour de la valeur ne change pas l'état de notre composant. La valeur de l'input sera donc toujours celle envoyée par l'utilisateur. (inverse du controlled component)

```
function UncontrolledInput {  
  return <input type="text" defaultValue="valeur de l'input"/>  
}
```

- ▷ Comment récupérer la donnée entrée par l'utilisateur ?

# Uncontrolled component

## ▷ Via une référence :

```
<input type="text" ref={ (ref) => input = ref} />
```

## ▷ Avec un listener :

```
function Form {  
  const onSubmit = (event) => {  
    event.preventDefault();  
    let data = new FormData(event.target);  
  };  
  
  return (  
    <form onSubmit={onSubmit}>  
      <input type="text" name="input_name"/>  
      <button>Submit</button>  
    </form>  
  )  
}
```



# Et avec les functional components ?

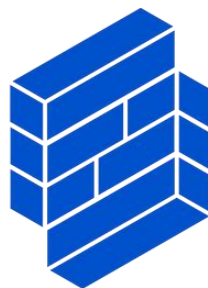
- ▷ C'est une autre paire de manche
- ▷ Il va être nécessaire de créer une référence au préalable :

```
export default function MiniForm() {  
  const firstnameRef = useRef();  
  
  function onSubmitForm(event) {  
    event.preventDefault();  
    console.log(firstnameRef.current.value);  
  }  
  
  return (  
    <form onSubmit={onSubmitForm}>  
      <input ref={firstnameRef} />  
      <button type="submit">Valider</button>  
    </form>  
  );  
}
```

Création de la référence

Annule le rechargement de la page

On applique la référence à l'input



Formik

# Formik – un incontournable

- ▷ L'une des grandes forces de React, c'est sa gestion dynamique de formulaire
- ▷ Mais le code pour implémenter une gestion efficace est **lourd** et **redondant**
- ▷ Formik va nous faire gagner beaucoup de temps et d'énergie

```
> npm install formik --save
```

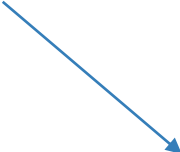
# Un composant Formik très puissant

- ▷ Formik fournit un composant paramétrable pour englober vos formulaires
  - › On va mettre en place un **contexte Formik** autour du formulaire

```
const valeursInitiales = {firstname: 'Joe', lastname: 'Dupond'};

const onValidate = values => {
  // Renvoie un objet vide si tout est OK
  // Et contenant des propriétés si des erreurs sont repérées
  console.log(values); // {firstname: '...', lastname: '...'}
  return {firstname: 'Une erreur car je le veux.'};
};

const onSubmit = (values, meta) => {
  console.log(values); // {firstname: '...', lastname: '...'}
  // Si ça nous va, alors on envoie l'information au serveur
  setTimeout(() => { // Simule l'appel au back
    meta.setSubmitting(false); // On indique que le chargement est terminé
  }, 1000)
};
```



```
<Formik
  initialValues={ valeursInitiales }
  validate={ onValidate }
  onSubmit={ onSubmit }
  // Et pleins d'autres paramètres optionnels
>
  <!-- Notre formulaire ici -->
</Formik>
```

# Formulaire avec les composants Formik

- ▷ Une fois le contexte préparé, il s'agit maintenant de créer notre formulaire puis de le relier à Formik
  - › Nous allons pleinement profiter du contexte Formik en mettant en place des composants (consumer) fournis par la librairie

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

// ...

<Formik initialValues={ valeursInitiales } validate={ onValidate } onSubmit={ onSubmit }>
  {{{ isSubmitting, ... }} => (
    <Form>
      <Field type="text" name="firstname" />
      <ErrorMessage name="firstname" component="div" />
      <Field type="text" name="lastname" />
      <ErrorMessage name="lastname" component="div" />
      <button type="submit" disabled={ isSubmitting }>
        Valider
      </button>
    </Form>
  )}
</Formik>
```

# Formulaire avec les composants Formik

▷ Sinon, on peut également s'abstraire des composants Formik...

```
import { Formik } from 'formik';
// ...
<Formik initialValues={ valeursInitiales } validate={ onValidate } onSubmit={ onSubmit }>
  ({ values, errors, isSubmitting, handleSubmit, handleChange, handleBlur, touched }) => (
    <form onSubmit={handleSubmit}>
      <input
        name="firstname"
        onChange={handleChange}
        onBlur={handleBlur}
        value={values.firstname}
      />
      { errors.firstname && touched.firstname && <p>{errors.firstname}</p> }
      /* ... */
      <button type="submit" disabled={isSubmitting}>
        Valider
      </button>
    </form>
  )
</Formik>
```

# Validation

- ▷ On gagne effectivement du temps sur la gestion du formulaire..  
Mais qu'en est-il de **la validation** ?
  - › **C'est la limite de Formik**
- ▷ Bah... C'est à nous de la mettre en place – manuellement !
- ▷ Pour la gagner du temps, nous partirons plutôt avec une librairie complémentaire très populaire : **YUP**

```
➤ npm install yup --save
```

# YUP – Validation des champs

- ▷ Formik possède une configuration adaptée et spécifique pour Yup
- ▷ L'implémentation se fera au travers des **schémas de validation**
- ▷ Et la mise en place correspond à la création du **schéma** :

```
const ConnectionSchema = Yup.object().shape({  
  firstname: Yup.string()  
    .min(2, 'Trop court :')  
    .max(50, 'Trop long cette fois !')  
    .required('Ce paramètre est requis'),  
  lastname: Yup.string()  
    .min(2, '...')  
    .max(50, '...')  
});
```



# Mise en place du schéma de validation

▷ Pour ça, rien de plus simple :

```
<Formik
  initialValues={ valeursInitiales }
  validate={ onValidate }
  onSubmit={ onSubmit }
  validationSchema={ ConnectionSchema }
  // Et pleins d'autres paramètres optionnels
>
  <!-- Notre formulaire ici -->
</Formik>
```

▷ Et c'est gagné, la gestion d'erreur sera automatiquement relié à Yup

```
{errors.firstname && touched.firstname && <p>errors.firstname</p>}
```



# ImmerJS

# Les variables d'états sont immuables

- ▷ Souvenez-vous, il est **interdit de modifier un état précédent !**
- ▷ Ce code est une mauvaise pratique :

```
const newState = {...state}; // Nouvelle référence certes  
// Mais ici on modifie l'ancien state  
newState.users[action.payload.index].firstname = action.payload.newFirstname;  
setState(newState);
```

# Différentes stratégies de clônage

▷ Voici les solutions que l'on pourrait mettre en place intuitivement :

▷ Cette solution fonctionne mais est très peu performante

```
const newState = JSON.parse(JSON.stringify(oldState));
```

▷ Celle-ci est mieux, mais pourquoi cloner recréer tout l'objet si l'on souhaite modifier un et unique élément à l'intérieur ?

```
const _ = require('underscore');  
const object = {foo: {bar: 123}};  
const objectClone = _.clone(object);
```

# La bonne solution

▷ Pour être efficace, il va être nécessaire de cloner dans l'objet uniquement les propriétés qui doivent être modifiées

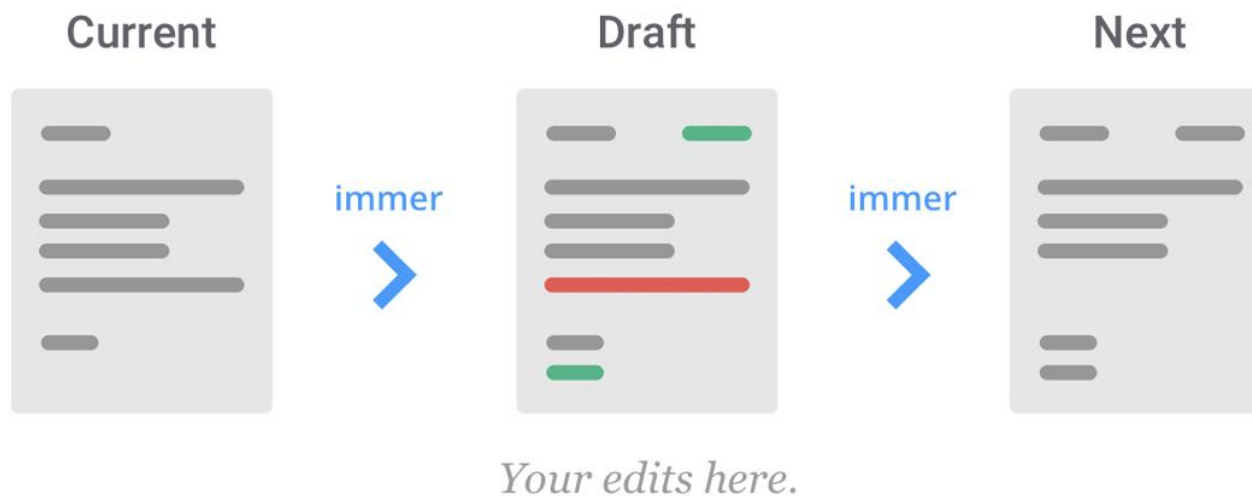
› Voici un exemple :

```
function happyBirthday(userId) {  
  const newUsers = users.map(u => {  
    if(u.id === userId) {  
      return {  
        ...u,  
        age: u.age + 1  
      }  
    }  
  });  
  
  return u;  
})  
  
updateUsers(newUsers);  
}
```

▷ Oui, on connaît.. Mais c'est très lourd à écrire !

# ImmerJS

- ▷ C'est là qu'intervient Immer
- ▷ Nous permet de manipuler un « draft » d'un objet
- ▷ Lui se chargera de calculer les différences pour les appliquer au nouvel objet « cloné »
  - › Une sorte de diffing



# Installer ImmerJS

- ▷ Pour l'installer, c'est très simple :

```
npm install immer use-immmer
```

- ▷ Un nouveau hook va faire son apparition : « useImmer »
- ▷ Nous permettra de créer des **variables d'état**
  - › Exactement comme useState
- ▷ Ces variables sont gérées par ImmerJS !

# useImmer

## ▷ Voici un exemple :

```
function App() {  
  const [person, updatePerson] = useImmer({  
    name: "Michel",  
    age: 33  
  });  
}
```

→ Crée une variable d'état

## ▷ Lors de la modification, immer nous fournit un « draft » que l'on pourra modifier **comme bon nous semble** !

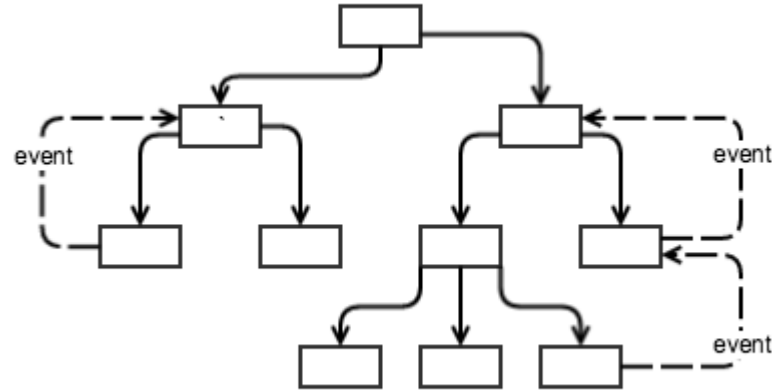
```
// ...  
function updateName(name) {  
  updatePerson(draft => {  
    draft.name = name;  
  });  
}  
// ...
```

→ C'est Immer qui se chargera de trouver les différences puis de cloner l'objet pour mettre à jour l'état



# TP - Sécuriser notre application

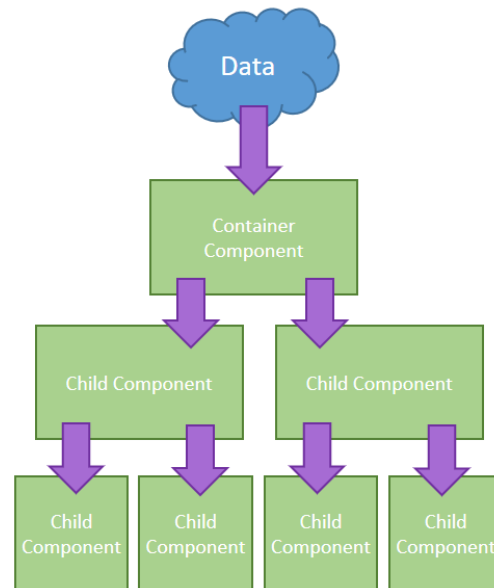
- ▷ Récupérons le **TP précédent** ;
- ▷ Dans la page Login, **ajoutez un formulaire** avec 2 champs : « pseudo » & « password » ;
- ▷ Si l'user rentre en pseudo : « admin » et en password « toto », alors la **redirection** se fera vers la page Home, sinon un message d'erreur est affiché !
- ▷ *(Bonus) Faites en sorte d'enregistrer un token à la connexion qui permettra de ne pas se reconnecter lors du rafraichissement de la page*
- ▷ *(Bonus) Simuler un appel serveur avec un délai d'attente de 3 secondes (setTimeout) et faite apparaître un composant gérant le loading en attendant la fin de l'appel*



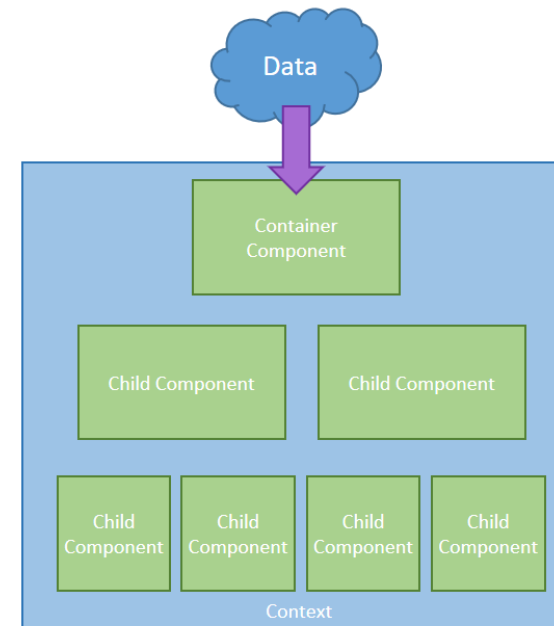
# Stratégie pour la gestion de données

# Context API

- React offre un moyen de partager de l'information à travers les composants



*prop drilling*



*context API*

# Créer un contexte

```
/* Mise en place du contexte pour les clients */  
export const AppContextCustomers = React.createContext({  
  customers: [],  
  addCustomer: () => { },  
  removeCustomer: () => { },  
  editCustomer: () => { },  
});
```

---

customers.context.ts

```
<AppContextCustomers.Provider value={this.state.customers}>  
  <Root>  
    <AppRoute />  
  </Root>  
</AppContextCustomers.Provider>
```

app.ts

# Utilisation du contexte

clients.screen.ts

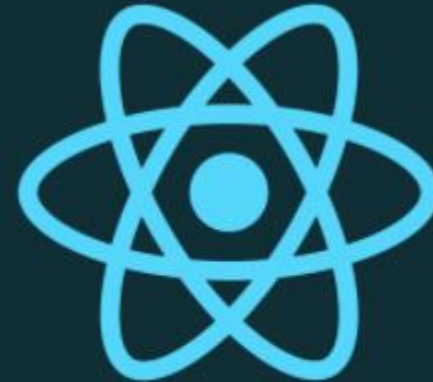
```
render() {  
  return (  
    <AppContextCustomers.Consumer>  
      ({ { customers } }) => (  
        (customers.length)  
        ? this.displayClients()  
        : this.displayNoClients()  
      )  
    </AppContextCustomers.Consumer>  
  );  
}
```

```
function App() {  
  // Récupération de la valeur du contexte  
  const userContext = useContext(AppContextCustomers);  
  
  return <span>{userContext.customers.length}</span>;  
}
```

# Pas de magie



Redux



React Context API

# Introduction

- ▷ Redux est une librairie pour la gestion d'état de manière prévisible, créée par Dan Abramov pour les applications JavaScript ;
- ▷ Elle est basée sur le concept de circulation unidirectionnel de données, popularisé par l'équipe Facebook avec son architecture [Flux](#) ;
- ▷ Elle n'a aucune dépendance avec ReactJS, et donc peut être utilisée avec d'autres frameworks JS ;
- ▷ Elle est la librairie préférée pour la gestion

# Application State vs UI State

- ▷ Il y a généralement deux « states » dans une app :
  - › Application State : état général d'une application. Peut être stocké dans une base de données ou ailleurs
  - › UI State : état propre à une partie de l'application (ex: formulaire), éphémère et qui peut être effacé
  
- ▷ Une bonne pratique est de gérer l'Application State par Redux et le UI State par setState
  
- ▷ *Note: Cette règle n'est pas inscrite dans le marbre, vous pouvez utiliser Redux ou setState suivant vos propres besoins.*



# Redux - Installation de l'environnement

- ▷ On commence par installer la dépendance Redux

```
> npm install --save redux
```

- ▷ Malgré le fait que Redux n'ait aucune dépendance avec ReactJS, il y a une librairie officielle (créée par l'équipe Redux) qui harmonise les deux et permet ainsi d'écrire moins de code :

```
> npm install --save react-redux
```

# Implémentation de toolkit

- ▷ Depuis peu, une nouvelle manière d'implémenter Redux est apparu
  - › Bien plus simple ;
  - › Bien plus lisible !
- ▷ Redux Toolkit !
- ▷ Permet de « rendre » l'application state **mutable**
- ▷ Nouvelle bibliothèque à part entière :

```
npm install @reduxjs/toolkit
```

# Implémentation

- ▷ Rien ne change pour :
  - › La création du store ;
  - › La mise en place des containers ;
  - › La communication entre nos composants et notre Store.
  
- ▷ Les changements s'appliqueront principalement pour nos **Actions** et **Reducers** ;
  
- ▷ Ces 2 entités vont être fusionnées en une et unique : Les **slices**

# Implémentation

```
export const MyCustomSlice = createSlice({
  name: "monPremierSlice",
  ...
})
```

- ▷ Votre Slice représente un « bout » de votre store ;
- ▷ Voici une architecture basique :

```
export const MySlice = createSlice({
  name: "mySlice",
  initialState: {
    ...
  },
  reducers: {
    ...
  }
});
```

# Implémentation

- ▷ Le changement le plus brutal : Il n'y a **plus d'action creators** !
- ▷ Désormais, il n'y que des reducers, ceux-ci pourront être utilisés comme actions creators également
  - › Grâce à **ImmerJS**, l'état géré par vos reducers est **mutable** !

```
import { createSlice } from "@reduxjs/toolkit";
export const MySlice = createSlice({
  name: "mySlice",
  initialState: {
    data: [1, 2, 3],
  },
  reducers: {
    addNewData(state, { payload: { dataToAdd } }) {
      state.data.push(dataToAdd);
    },
    removeDataByIndex(state, action) {
      state.data.splice(action.payload); // Quel bonheur
    },
  },
});
```

# Utilisation des Slices

- ▷ Pour interagir avec votre Store, il va être nécessaire de fournir :
  - › Un reducer à votre Store ;
  - › Des actions à vos component.
- ▷ Redux toolkit a tout prévu, ajoutons ces 2 lignes à la fin du fichier :

```
export const { addNewData, removeDataByIndex } = MySlice.actions;
```

MySlice.slice.js

Action creators

```
export default MySlice.reducer;
```

Reducer

Nouvelle bonne  
pratique

```
import MySliceReducer from './MySlice.slice';

const store = configureStore({
  reducer: {
    slices: MySliceReducer
  }
});
```

App.js

# Actions

- ▷ Encore une fois, rien ne change pour les actions et container
- ▷ Il s'agira simplement d'importer vos actions précédemment exportés du slice dans votre conteneur :

```

import { addNewData, removeDataByIndex } from './MySlice.slice';
let dispatchMapping = (dispatch) => ({
  actions: bindActionCreators({ addNewData, removeDataByIndex }, dispatch)
});
    
```

comp.container.js

# useSelector & useDispatch

▷ Plus besoin de créer de conteneurs !

▷ Soit le slice suivant :

counter.slice.js

```
export const counterSlice = createSlice({
  name: 'counter',
  initialState: { count: 1 },
  reducers: {
    increment: (state) => {
      state.count = state.count + 1;
    },
    decrement: (state) => {
      state.count = state.count - 1;
    },
  },
});

export const { increment, decrement } = counterSlice.actions;
export default counterSlice.reducer;
```



# useSelector & useDispatch

- Le composant peut se « connecter » au store grâce au « useSelector »

**counter.component.js**

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

const Counter = (props) => {
  const count = useSelector((state) => {
    return state.counter.count;
  });

  return (
    <div>
      <h1>Compteur: {count}</h1>
    </div>
  );
}
```

----->

```
export const counterSlice = createSlice({
  name: 'counter',
  initialState: { count: 1 },
});
```

# useSelector & useDispatch

## ▷ Même principe pour dispatcher une action

```


import { useDispatch } from 'react-redux';
import { increment } from './store/counterSlice';

const Counter = (props) => {
  const dispatch = useDispatch();

  return (
    <div>
      <button onClick={() => dispatch(increment())}>
        Incrémenter
      </button>
    </div>
  );
};
    
```

counter.component.js

export const { increment, decrement } = counterSlice.actions;



<https://stackblitz.com/edit/react-redux-toolkitjs-demo>

# Communiquer avec un serveur

- ▷ Il est rare de mettre en place un store sans un **serveur** pour lui fournir la donnée
- ▷ Il s'agira de :
  - › Réaliser un **appel au serveur** lors de certaines actions ;
  - › **Mettre à jour le store** suite à une mise à jour serveur ;
  - › **Mise en cache** de certaines informations ;
  - › Etc.
- ▷ Comme pour les Slice, Redux propose une nouvelle façon de gérer ces appels asynchrone avec **RTK Query**

# RTK Query

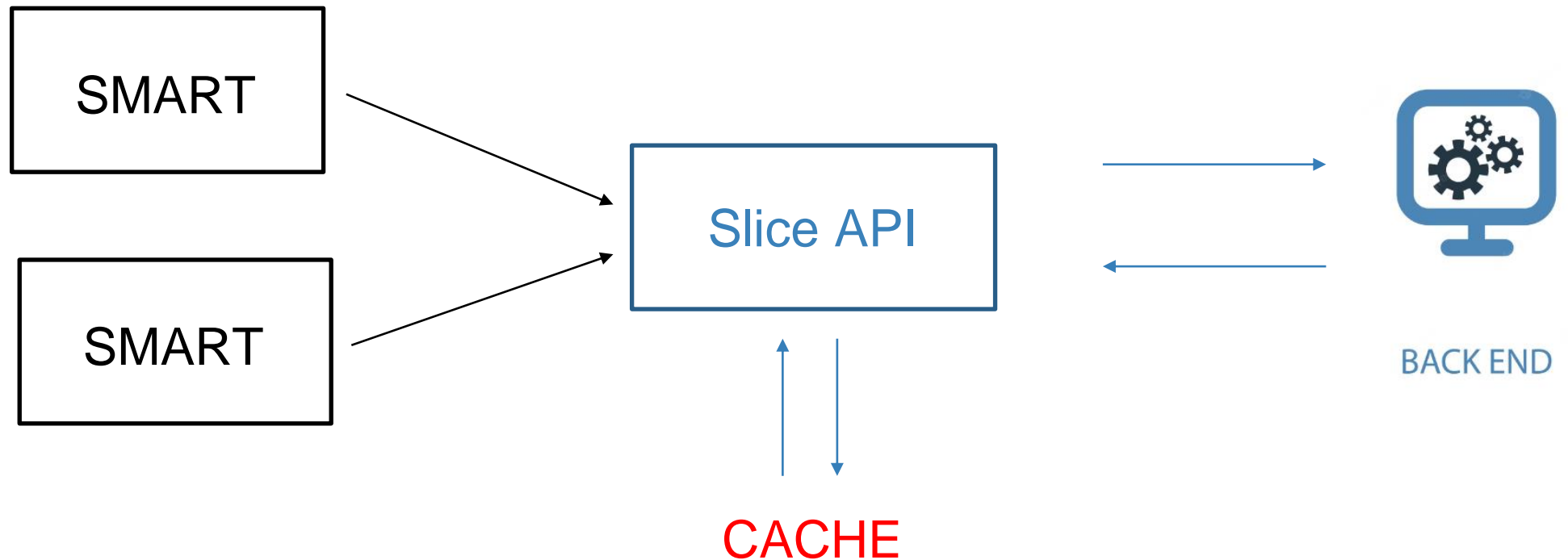
- ▷ Déjà inclus dans le package « [reduxjs/toolkit](#) »
- ▷ Outil de [récupération](#) de données et mise en [cache](#)
- ▷ Nous permettra par exemple :
  - › Le suivi de l'[état](#) de chargement des requêtes ;
  - › D'éviter les demandes en double pour les mêmes données ;
  - › Les [mises à jour optimistes](#) pour rendre l'interface utilisateur plus rapide ;
  - › La gestion du [temps de cache](#) lorsque l'utilisateur interagit avec l'interface utilisateur
  - › Etc.

# Fonctionnement de RTK Query

- ▷ Auparavant, avec redux nous cherchions à constamment **réagir** aux appels serveur pour **maintenir le store en phase**
- ▷ Avec RTK, nous chercherons à répondre aux questions suivantes :
  1. D'où viennent ces données ?
  2. Comment cette mise à jour doit-elle être envoyée ?
  3. Quand mettre en cache les données et quand aller les récupérer ?
  4. Dois-je mettre en cache cette donnée ou non ?
- ▷ Vous l'avez compris, c'est bien plus complexe !

# Mise en place de RTK

- Avec RTK, il est nécessaire de créer une « API » local qui centralisera tous nos futurs appels :



# Création de l'API

- ▷ Tentons de mettre en place notre premier « Slice API »
  - › On va y aller **étape par étape** !
- ▷ Commençons par créer l'entité API
  - › Toolkit fournit une méthode toute prête pour ça

```
import { createApi } from '@reduxjs/toolkit/query/react'

export const api = createApi({
  // ...
})
```

- ▷ Ecriture très similaire aux « createSlice » que l'on connaît !

# Création de l'API

▷ Voici les propriétés importantes que l'on va étudier :

```
export const api = createApi({
  reducerPath: ...,

  baseQuery: ...,

  endpoints: ...
})
```

services/api.ts

- ▷ « **reducerPath** » : Clé unique représentant l'API, utile quand il y en a plusieurs
- ▷ « **baseQuery** » : Représente les informations du serveur à consulter
- ▷ « **endpoints** » : Points de terminaison du serveur. Routes de l'API (avec leurs méthodes associées) que l'on souhaite appeler
  - › Il est d'usage de n'avoir qu'un et unique « **createAPI** » par application React



# Création de l'API

▷ Voyons en détail les 2 premières propriétés :

```
import { createApi, fetchBaseQuery } from '@reduxjs/toolkit/query/react';
```

services/api.ts

```
export const m2gApi = createApi({
  reducerPath: 'm2gApi',
  baseQuery: fetchBaseQuery({
    baseUrl: 'https://serveur-demo.m2g-intellect.fr',
  }),
  endpoints: ...
});
```

On met ce que l'on veut ici

Wrapper de fetch. Fonctionne un peu comme Axios !

Le wrapper permet notamment de configurer la route par défaut

▷ Une fois l'API configuré, nous allons pouvoir configurer nos **endpoints** un à un

# Création de l'API – Configuration endpoints

- ▷ RTK query nous fournit un builder permettant de configurer nos routes simplement

```
export const m2gApi = createApi({
  reducerPath: ...,
  baseQuery: ...,
  endpoints: (builder) => ({
    pingServer: builder.query({
      query: () => `/${`,
    }),
  }),
});
```

*services/api.ts* → C'est là que l'on va déclarer toutes nos méthodes pour joindre le serveur

→ Avec « query », nous indiquons que la requête est de type « GET »

→ Endpoint à appeler ! Ici on fait appel à la racine pour ping le serveur 😊

→ On peut passer certains paramètres (qui pourront être passés à la route en POST plus tard)

```
export const { usePingServerQuery } = m2gApi;
```

- ▷ Comme pour createSlice, RTK export automatiquement les hooks permettant de réaliser l'appel au serveur
  - › Ce hook permettra de récupérer certaines méthodes prédéfinies que l'on étudiera plus tard !

# Création du store

- ▷ Comme pour « `createSlice` », nous dépendant bien évidemment d'un store qu'il faut créer en amont
  - › C'est lui qui gèrera le `cache` etc.

store.ts

```
import { configureStore } from '@reduxjs/toolkit';
import { m2gApi } from '../services/api';
```

```
export const store = configureStore({
  reducer: {
    [m2gApi.reducerPath]: m2gApi.reducer,
  },
  middleware: (getDefaultMiddleware) =>
    getDefaultMiddleware().concat(m2gApi.middleware),
});
```

Correspond au reducerPath indiqué précédemment

Généré automatiquement par createApi

Il est obligatoire d'ajouter les middleware de notre API à ceux de redux, autrement une erreur est levée !

# Enclencher les requêtes

- ▷ C'est le grand moment ! Appelons nos **endpoints** précédemment déclarés

```
import { usePingServerQuery } from './services/user';
```

```
export default function App() {
```

```
  const obj = usePingServerQuery(undefined);
```

```
  console.log(obj.data); // { response: "Vide" } -> Le ping a fonctionné !!
```

```
  ...
```

L'appel est fait là ! Pas besoin de useEffect  
C'est déjà optimisé !

Paramètres à fournir à notre query (ici aucun)

- ▷ Voici les élément important de cet objet :

- › **data** : Donnée renvoyée
- › **error** : Erreur s'il y en a
- › **isLoading** : Booléen pour savoir si la requête serveur est en cours
- › **refresh** : Force la requête et rafraichit le cache
- › etc

<https://stackblitz.com/edit/react-create-api>

# Fonctionnement du cache

- ▷ Quand une requête est réalisée, RTK enregistre un **identifiant** pour cette requête (endpoint + paramètres) et stocke le **résultat en cache**
  - › Le cache dans notre cas correspond à notre **store Redux** !
  - › Ce cache a une durée de vie !
- ▷ Quand une autre requête est effectuée pour les mêmes données, RTK fournit les données en cache plutôt que d'envoyer une requête supplémentaire au serveur
  - › Pour mettre à jour le cache, il faudra appeler « **refresh** »
- ▷ Il est possible de configurer le **temps de cache** pour chacune de vos données !
  - › Par défaut, le temps de cache est de **60s** et se remet à 0 à chaque fois que la donnée est requêtée !
- ▷ **Tous les composant abonné à un endpoint hook seront modifié si le cache change !**



---

# Tests unitaires

# React & test

- ▶ En utilisant « create-react-app », React intègre un outil de testing :



- ▶ Chaque fichier de test est représenté par une extension « test.js »



- ▶ Et une ligne de commande simple pour enclencher les tests :

```
Test Suites: 1 failed, 1 total
Tests:      0 total
Snapshots:  0 total
Time:       2.464s
```



## Watch Usage

- › Press a to run all tests.
- › Press f to run only failed tests.
- › Press q to quit watch mode.
- › Press p to filter by a filename regex pattern.
- › Press t to filter by a test name regex pattern.
- › Press Enter to trigger a test run.

# App.test.js

▷ Voici le contenu généré par défaut :

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

App.test.js

```
it('renders without crashing', () => {  
  const div = document.createElement('div');  
  ReactDOM.render(<App />, div);  
  ReactDOM.unmountComponentAtNode(div);  
});
```

On crée une div dynamiquement

Retire un composant React monté du DOM et nettoie ses gestionnaires d'événements et son état local.

▷ « **it** » permet de déclarer un test unitaire  
› Fournit par JEST



# Plusieurs fonctions fournies

[Liste des fonctions](#)

Intitulé du test

```
it('Test de la fonction du meilleur cours', () => {  
  expect(quelEstLeMeilleurCours()).toBe('ReactJS');  
});
```

▷ **expect** prend une valeur, et en association à **toBe**, il permet de vérifier que la valeur est bien égale au résultat attendu

```
test('valeurs numériques', () => {  
  expect(100).toBeWithinRange(90, 110);  
  expect(101).not.toBeWithinRange(0, 100);  
  expect({ apples: 6, bananas: 3 }).toEqual({  
    apples: expect.toBeWithinRange(1, 10),  
    bananas: expect.not.toBeWithinRange(11, 20),  
  });  
});
```

# Thématique de test

## ▷ Il est possible de définir des thématiques de test

```
describe('Les différentes synthaxes en JEST', () => {  
  it('Premier test simple', () => {  
    expect(1 + 2).toEqual(3);  
    expect(2 + 2).toEqual(4);  
  });  
  
  it('Simple boolean', () => {  
    expect([1]).toBeTruthy();  
    expect(0).toBeFalsy();  
  });  
  
  it('Manipulation sur objet', () => {  
    const houseForSale = {  
      bath: true,  
      bedrooms: 4  
    };  
  
    expect(houseForSale).toHaveProperty('bath');  
    expect(houseForSale).toHaveProperty('bedrooms', 4);  
    expect(houseForSale).not.toHaveProperty('pool');  
  })  
});
```

# Gestion des tests asynchrones

▷ Soit le code suivant :

```
export default function asynchronousRequest() {  
  return new Promise((resolve, reject) => {  
    setTimeout(() => {  
      resolve(true);  
    }, 2000);  
  });  
}
```

Test.js

Test.spec.js

```
it('Tester le retour asynchrone', async () => {  
  expect.assertions(1); // S'attend à 1 appel asynchrone durant ce test  
  await expect(asynchronousRequest()).resolves.toBeTruthy();  
});
```

Problème

✓ Tester le retour asynchrone (2004ms)

# Manipuler le temps

- ▷ Un test doit être **F.I.R.S.T**
- ▷ Nos tests doivent être enclenchés à chaque modification du code
  - › Donc ils doivent être rapides !

```
jest.useFakeTimers();
```

```
it('Tester le retour asynchrone', async () => {  
  expect.assertions(1); // S'attend à 1 appel asynchrone durant ce test
```

```
  const promise = asynchronousRequest().then(resolved => {  
    expect(resolved).toBeTruthy();  
  });
```

```
  jest.advanceTimersByTime(2000);  
  return promise;  
});
```

✓ Tester le retour asynchrone (1ms)

# Mock

- ▷ Un appel au serveur étant coûteux en terme de temps, nous allons **bypasser** le comportement natif d'axios pour **simuler** l'appel

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

user.service.js

```
class Users {  
  static getAllUsers() {  
    return axios.get('/users').then(resp => resp.data);  
  }  
}
```

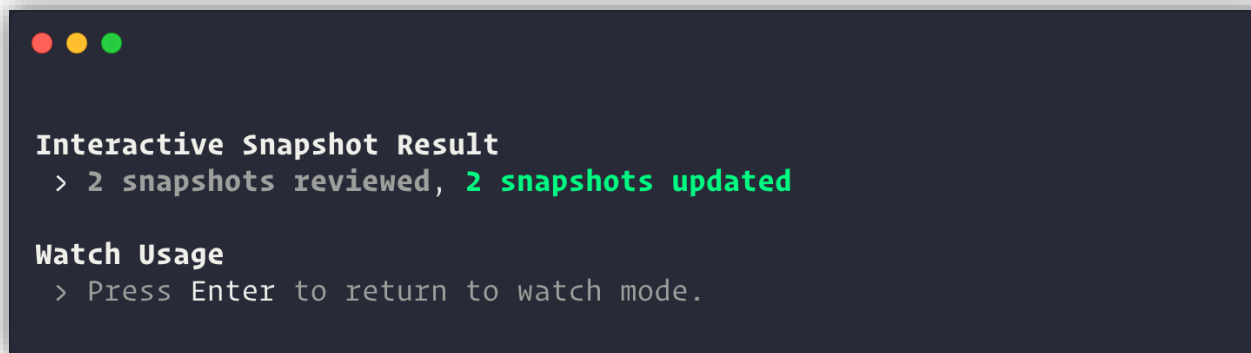
```
jest.mock('axios'); // SURCHARGE !
```

user.service.test.js

```
it('Récupère les utilisateurs du site', () => {  
  const users = [{name: 'Bob'}];  
  const resp = {data: users};  
  axios.get.mockResolvedValue(resp);  
  
  return Users.getAllUsers().then(data => expect(data).toEqual(users));  
});
```

# Snapshots

- ▷ Jest a introduit une notion de tests « **Snapshot** »
  - › Capture
- ▷ Permet de **figer l'état** d'un composant à un moment **t** et de comparer les différents « snapshot »
  - › Permet de repérer les éventuels différences



```
Interactive Snapshot Result
> 2 snapshots reviewed, 2 snapshots updated

Watch Usage
> Press Enter to return to watch mode.
```

# Snapshots

- ▷ Si le snapshot change et que l'on compare les différentes version, le « **test snapshot** » échouera ;
  - › Ce n'est pas forcément une mauvaise chose !
- ▷ Une fois le « Snapshot test » échoué, **vous décidez** si la différence doit provoquer un échec du test ou non.
  - › Très utile pour nous assurer que l'interface utilisateur ne change pas de manière inattendue
- ▷ Avec les snapshot, nos tests seront bien plus légers, mais pour l'utiliser il va être nécessaire d'installer « **react-test-renderer** »

```
> npm install react-test-renderer
```

# Test d'un composant

## Link.js

```
export default class Link extends React.Component {
  constructor(props) {
    super(props);

    this._onMouseEnter = this._onMouseEnter.bind(this);
    this._onMouseLeave = this._onMouseLeave.bind(this);

    this.state = {
      class: STATUS.NORMAL,
    };
  }

  _onMouseEnter() {
    this.setState({class: STATUS.HOVERED});
  }
}
```

## Link.js (suite)

```
  _onMouseLeave() {
    this.setState({class: STATUS.NORMAL});
  }

  render() {
    return (
      <a
        className={this.state.class}
        href={this.props.page || '#'}
        onMouseEnter={this._onMouseEnter}
        onMouseLeave={this._onMouseLeave}
      >
        {this.props.children}
      </a>
    );
  }
}
```



# Test d'un composant

```
import Link from './Link';
import renderer from 'react-test-renderer';

it('Link changes the class when hovered', () => {
  const component = renderer.create(
    <Link page="http://www.facebook.com">Facebook</Link>,
  );
  let tree = component.toJSON(); // Génère arbre
  expect(tree).toMatchSnapshot(); // Vérifie si le snapshot match la dernière version

  // Trigger manuellement la méthode onMouseEnter()
  tree.props.onMouseEnter();
  // Ré-affichage
  tree = component.toJSON(); // On reprend un snapshot
  expect(tree).toMatchSnapshot();

  // Trigger manuel
  tree.props.onMouseLeave();
  // Ré-affichage
  tree = component.toJSON();
  expect(tree).toMatchSnapshot();
});
```

Simule l'affichage du composant

Vérifie si le snapshot match la dernière version

On reprend un snapshot

# Fonctionnement

▷ A la première exécution du test précédent, Jest va créer le snapshot suivant :

```
exports[`Link changes the class when hovered 1`] = `<a className="normal"
  href= " http://www.facebook.com "
  onMouseEnter={{[Function]}}
  onMouseLeave={{[Function]}} >
  Facebook
</a> `;
```

▷ Aux tests suivants, Jest comparera les prochains snapshot avec ceux pris précédemment pour aller rapidement

- › S'il ne passe pas, le test échoue

# Fichier snapshot généré

Link.test.js.snap

```
1 // Jest Snapshot v1, https://goo.gl/fbAQLP
2
3 exports[`Link changes the class when hovered 1`] = `
4 <a
5   className="normal"
6   href="http://www.facebook.com"
7   onMouseEnter=[[Function]]
8   onMouseLeave=[[Function]]
9 >
10 | Facebook
11 </a>
12 `;
13
14 exports[`Link changes the class when hovered 2`] = `
15 <a
16   className="hovered"
17   href="http://www.facebook.com"
18   onMouseEnter=[[Function]]
19   onMouseLeave=[[Function]]
20 >
21 | Facebook
22 </a>
23 `;
24
25 exports[`Link changes the class when hovered 3`] = `
26 <a
27   className="normal"
28   href="http://www.facebook.com"
29   onMouseEnter=[[Function]]
30   onMouseLeave=[[Function]]
31 >
32 | Facebook
33 </a>
34 `;
35 |
```

# Mettons à jour notre test

▷ On fait le choix de modifier l'url :

```
const component = renderer.create(  
  <Link page="http://www.macademia.fr">Macademia</Link>,  
);
```

▷ Et puis l'erreur survient :

```
- Snapshot  
+ Received  
  
  <a  
    className="normal"  
-   href="http://www.facebook.com"  
+   href="http://www.macademia.fr"  
    onMouseEnter={[Function]}  
    onMouseLeave={[Function]}  
  >  
-   Facebook  
+   Macademia  
  </a>
```

# Mettre à jour les snapshot

- Comme indiqué précédemment, c'est à nous développeur d'indiquer si la différence entre les 2 snapshots est un « *bug ou une feature* »

## Watch Usage

- > Press a to run all tests.
- > Press f to run only failed tests.
- > Press p to filter by a filename regex pattern.
- > Press t to filter by a test name regex pattern.
- > Press u to update failing snapshots.
- > Press i to update failing snapshots interactively.
- > Press q to quit watch mode.
- > Press Enter to trigger a test run.

—

## TP – Tester c'est douter

- ▷ Créez un composant simple permettant d'afficher une liste d'éléments
- ▷ Le composant doit contenir un **input** et un **bouton** pour valider
- ▷ **Testez** ce composant en vérifiant que la fonction permettant d'ajouter un élément fonctionne
- ▷ Testez ce composant avec **différents snapshots**

# Merci