

**VOTRE
SUPPORT DE COURS**

ReactJS, programmation avancée

31 - 02 févr. 2022



SÉMINAIRES - COURS DE SYNTHÈSE - STAGES PRATIQUES - CERTIFICATIONS

PARCOURS CERTIFIANTS - FORMATIONS À DISTANCE



E-LEARNING - COACHING



APRÈS VOTRE FORMATION... POUR CONTINUER LES ÉCHANGES



L'évaluation de la formation : elle doit être remplie le dernier jour de la formation avant 18h. Elle est accessible en ligne depuis votre poste, tablette ou smartphone à l'adresse <http://eval.orsys.fr>. Le mot de passe nécessaire sera fourni par votre formateur.



MyOrsys : espace web réservé aux participants qui ont suivi l'une de nos formations. Vous y trouverez vos supports de cours au format PDF. Les formateurs ont également un accès afin de pouvoir compléter, si besoin, avec des documents annexes : articles, études de cas, corrections d'exercices, etc.



Le blog : "Les carnets d'ORSYS" est alimenté plusieurs fois par semaine et rassemble tous nos articles d'actualité couvrant l'ensemble des domaines enseignés. Un travail réalisé en collaboration avec nos intervenants, spécialistes de leur domaine et parfaitement au fait de l'actualité.



Les conférences d'actualité et les webinars gratuits : pour vous tenir informés et échanger sur des thèmes d'actualité, nous vous proposons tout au long de l'année de venir rencontrer nos experts et d'échanger avec eux.



Les vidéos : avis d'expert, présentation des cours, description de méthodes pédagogiques... directement depuis notre site ou sur notre chaîne YouTube, nos vidéos seront pour vous une source d'informations appréciable !

Suivez notre activité sur les réseaux sociaux :



Notre vocation, votre réussite

LES DOMAINES ABORDÉS

TECHNOLOGIES NUMERIQUES

- Management des SI
- Gestion de projets, MOA
- Développement logiciel
- Big Data, BI, NoSQL, SGBD
- Technologies Web
- Réseaux et sécurité
- Systèmes d'exploitation
- Virtualisation, Cloud et DevOps
- Messagerie, travail collaboratif
- Bureautique
- Graphisme, multimédia, PAO, CAO

COMPETENCES METIERS

- Gestion, comptabilité et finance
- Ressources humaines
- Formation
- Office Manager, assistant(e), secrétaire
- Commercial et relation client
- Marketing digital
- Marketing et communication d'entreprise
- Amélioration continue, Lean, QSE
- Gestion de production, performance opérationnelle
- Achats, services généraux, logistique
- Droit et contrats
- Banque et assurance
- Secteur public
- Santé et action sociale

MANAGEMENT ET DEVELOPPEMENT PERSONNEL

- Management d'entreprise
- Management
- Management avancé et Leadership
- Développement personnel
- Office Manager, assistant(e), secrétaire
- Perfectionnement à l'Anglais
- Gestion de projets

QUELQUES CHIFFRES

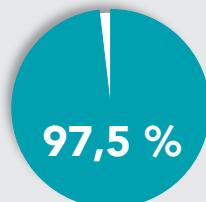


10 000
sessions par an.

Près de **70 000**



personnes formées en 2018.



de taux de satisfaction exprimé par nos clients.



d'expérience.



plus de
2 000
cours différents.

ORSYS

la méthode

DES SPÉCIALISTES RECONNUS



- Les animateurs des cours ORSYS sont des spécialistes reconnus dans leur domaine,
- Hommes et femmes de terrain, responsables de projets d'avant-garde dans l'industrie ou la recherche, consultants ou coachs,
- Concepteurs de leur cours, et notamment de leur documentation,
- Indépendants des constructeurs ou éditeurs de logiciels, de toute « école ».



UNE ORIENTATION OPÉRATIONNELLE



- Les cours ORSYS ont pour objectif de former d'une manière concrète et directement applicable,
- Centrés sur les méthodes et techniques de pointe et porteuses d'avenir,
- Les travaux de groupe, les exercices sont issus des projets réalisés par les animateurs,
- Les cas réels permettent d'aborder les difficultés techniques et aussi organisationnelles.



ORSYS

une offre structurée

LES FORMATIONS



LES SÉMINAIRES

- Maîtriser les concepts et les techniques,
- Faire le point sur l'Etat de l'Art,
- Choisir les solutions d'avenir.



LES COURS PRATIQUES

- Maîtriser les outils, les systèmes, les méthodes,
- Apprendre à les mettre en œuvre,
- Acquérir une spécialisation.



LES COURS DE SYNTHÈSE

Permettent de faire un point complet sur un thème précis : ils présentent les informations les plus récentes dans une optique opérationnelle, et sont accompagnés de démonstrations et d'études de cas.



LES MULTIMODALES

ORSYS vous propose différents types de formations qui associent le présentiel à des solutions innovantes mixant plusieurs modalités d'apprentissage : séquences e-learning, études de cas pratiques, ateliers collaboratifs, serious games, quiz, tests de validation des acquis...

UNE ORGANISATION PAR FILIÈRES



Pour faciliter vos choix de formation, nous avons privilégié une approche par filière. Organisée selon des thématiques technologiques, cette approche correspond à un enchaînement précis de cours, qui permet de passer en quelques semaines de l'initiation à une technologie à sa pleine maîtrise opérationnelle.

www.orsys.com - info@orsys.fr
tél +33 (0)1 49 07 73 73

VOTRE
SUPPORT DE COURS

FORMATIONS ORSYS



SÉMINAIRES - COURS DE SYNTHÈSE - STAGES PRATIQUES - CERTIFICATIONS

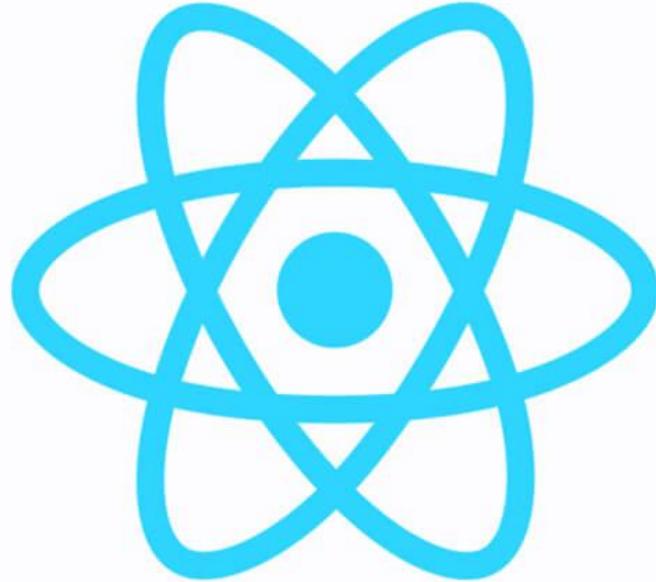
PARCOURS CERTIFIANTS - FORMATIONS À DISTANCE



E-LEARNING - COACHING



Ce support pédagogique vous est remis dans le cadre d'une formation organisée par ORSYS. Il est la propriété exclusive de son créateur et des personnes bénéficiant d'un droit d'usage. Sans autorisation explicite du propriétaire, il est interdit de diffuser ce support pédagogique, de le modifier, de l'utiliser dans un contexte professionnel ou à des fins commerciales. Il est strictement réservé à votre usage privé.



React avancé

Animé par Mazen Gharbi

Présentation



React avancé

```
constructor() {  
    let user = new User();  
    user.name = 'Mazen GHARBI';  
    user.addSkills(['ReactJS', 'Angular', 'NodeJS', 'PHP', 'Symfony', ...LIST_OTHERS]);  
    user.company = 'Macademia';  
    user.email = 'mazenGharbi@gmail.com';  
}
```



Présentation

Et vous?

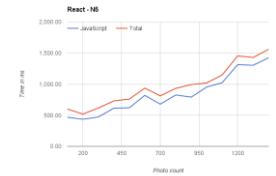
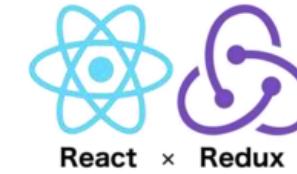
React

- ▷ React est une librairie (*et non un framework*)
- ▷ Créeé en 2013 par Facebook 
- ▷ Très simple à la base
- ▷ Des milliers de librairies pour React créées par la communauté
- ▷ Orienté composant !



Objectifs du cours

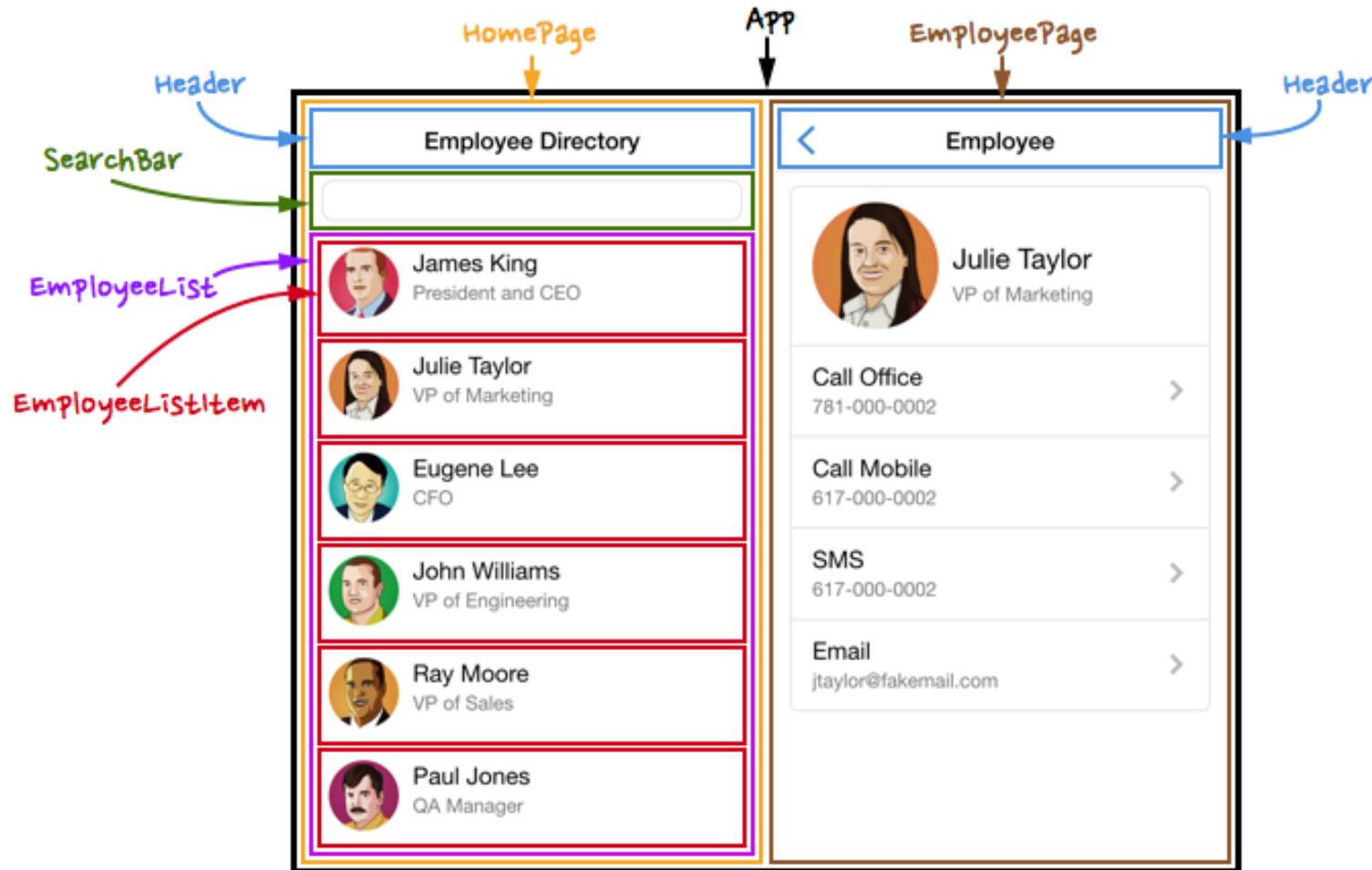
- ▷ Comprendre les concepts avancés de React
- ▷ Optimiser les performances des applications et l'expérience utilisateur
- ▷ Améliorer la qualité du code produit
- ▷ Intégrer les différentes librairies externes incontournables



FORMIK



Orienté composants



JavaScript XML

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

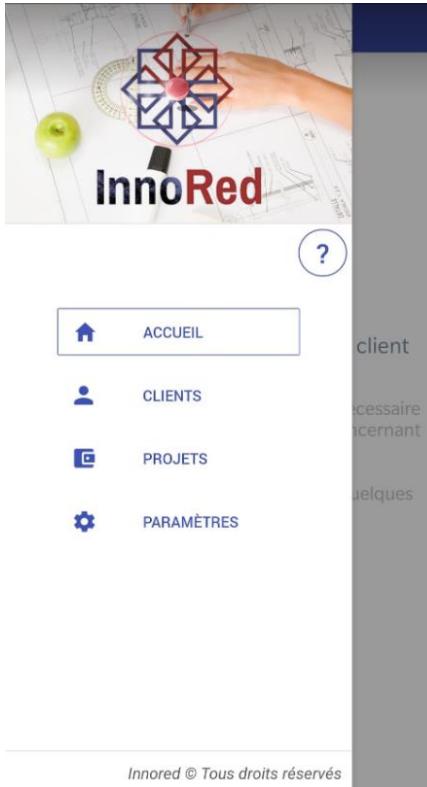
```
class Button extends Component {  
  
  handleClick = () => {  
    this.props.clickHandler(this.props.value);  
  };  
  
  render() {  
    return (  
      <div className={'Button ' + this.props.class} onClick={this.handleClick}>  
        <div>  
          {this.props.value}  
        </div>  
      </div>  
    );  
  }  
}
```

Single Page Application

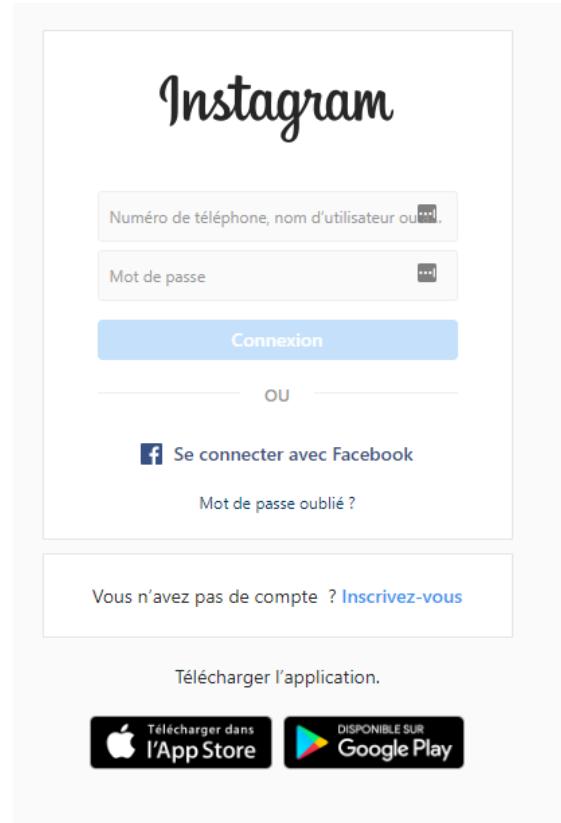
- ▷ React est parfaitement adaptée pour la mise en place de SPAs



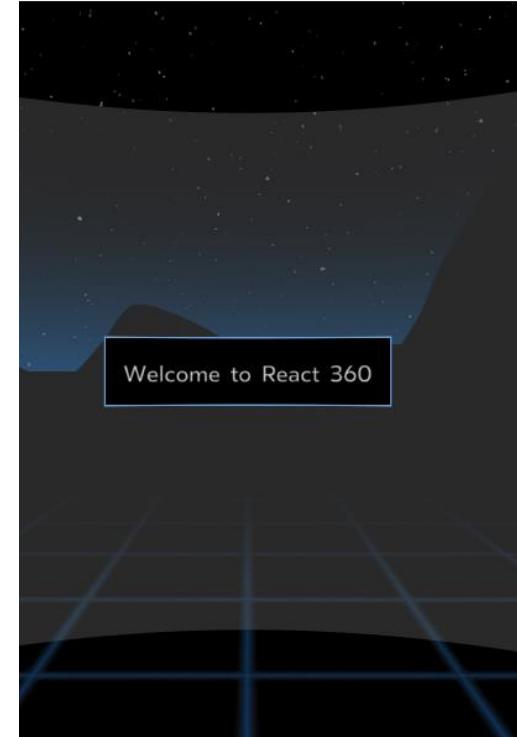
Ce qu'on peut faire avec React



Applications mobiles



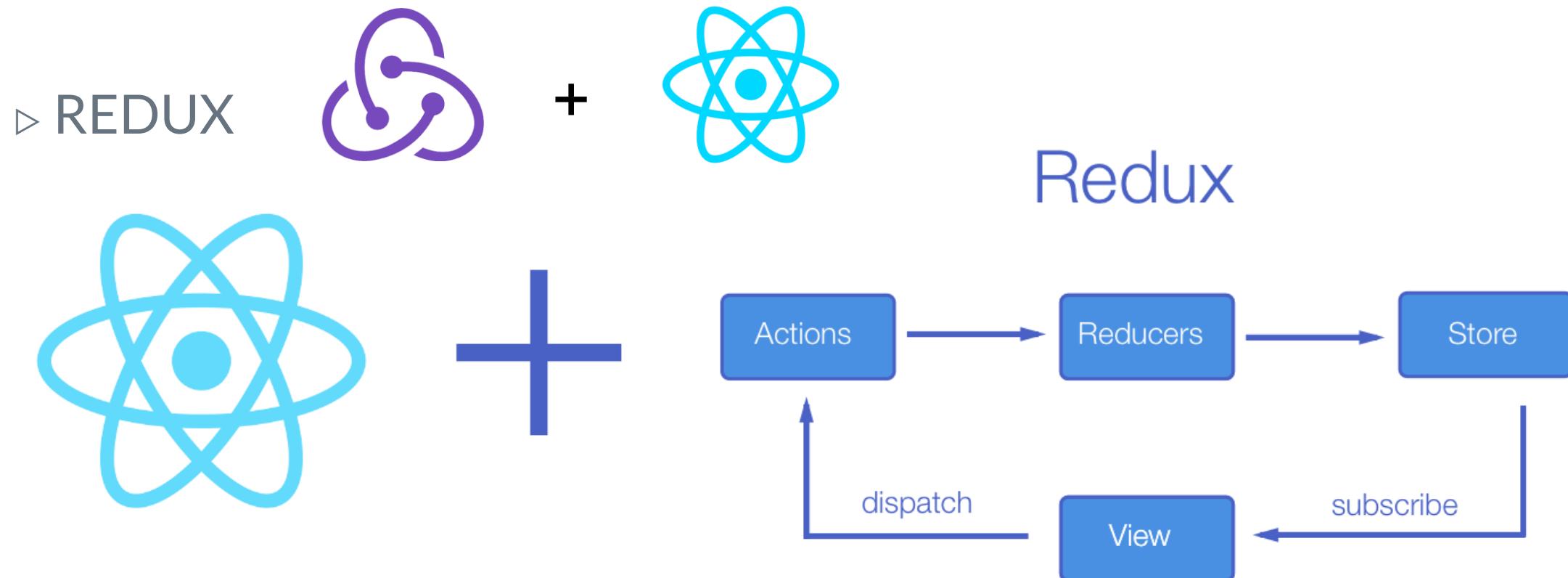
Sites webs



VR

Les limites

- ▷ Simple et peu adapté pour les grands projets par défaut
 - › Le code devient très vite brouillon et peu organisé dès que l'appli grandit





EcmaScript 2015 +

Animé par Mazen Gharbi

L'histoire de Javascript

- ▷ **Création du langage en 1995** ;
 - › Version initiale de Javascript créée en 10 jours seulement !
- ▷ **1997** : Javascript gagne la guerre et s'impose comme un standard « cross-browser » sous le nom officiel « EcmaScript » ;
- ▷ **2009** : Sortie de NodeJS ;
- ▷ **2015** : Finalisation du standard EcmaScript 6 ;

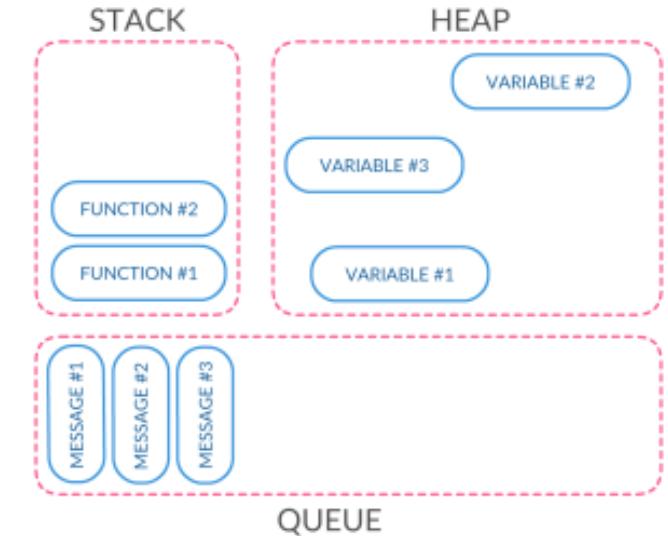
Propriétés du langage

- ▷ *Dynamiquement** typé
- ▷ *Faiblement** typé
- ▷ Multi-paradigme (Orienté Objet / Fonctionnel)
 - › Brendan Eich s'est inspiré de Self, Scheme, Java et C
- ▷ « Cross-browser » et « cross-platform »



Asynchrone et Single-Thread

- ▷ **Single-Threaded** = “Thread-Safe” & performant
 - › La fonction ne peut être interrompue de l'extérieur
 - › Pas limité par un nombre maximum de threads et les allocations mémoire associées.
- ▷ Gestion des évènements asynchrones avec le **Event-Loop** ;
- ▷ Un moteur web est constitué de la **Heap**, de la **Queue** et de la **Stack** ;



Valeur par défaut

```
var userName;  
  
console.log(userName) // ???
```

- ▷ Quelle différence entre **null** et **undefined** ?

Les objets et tableaux

```
/* Create user object. */
var user = { firstName: 'Foo' };

/* Add `lastName` attribute. */
user.lastName = 'BAR';

/* Remove `firstName` attribute. */
delete user.firstName;

/* Change value */
user.lastName = 'Foo';
```

Cloner un objet

- ▷ Parfois, il peut s'avérer nécessaire de cloner 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 cloner

- ▷ Il est également possible de cloner 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...
```

Les prototypes

```
var user = null;

var User = function User(firstName, lastName) {
    this.firstName = firstName;
    this.lastName = lastName;
};

User.prototype.name = function name() {
    return this.firstName + ' ' + this.lastName;
};

user = new User('Foo');

console.log(user.name());    -> Foo undefined
```

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)¹. 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;
        });
    });
});
```

```

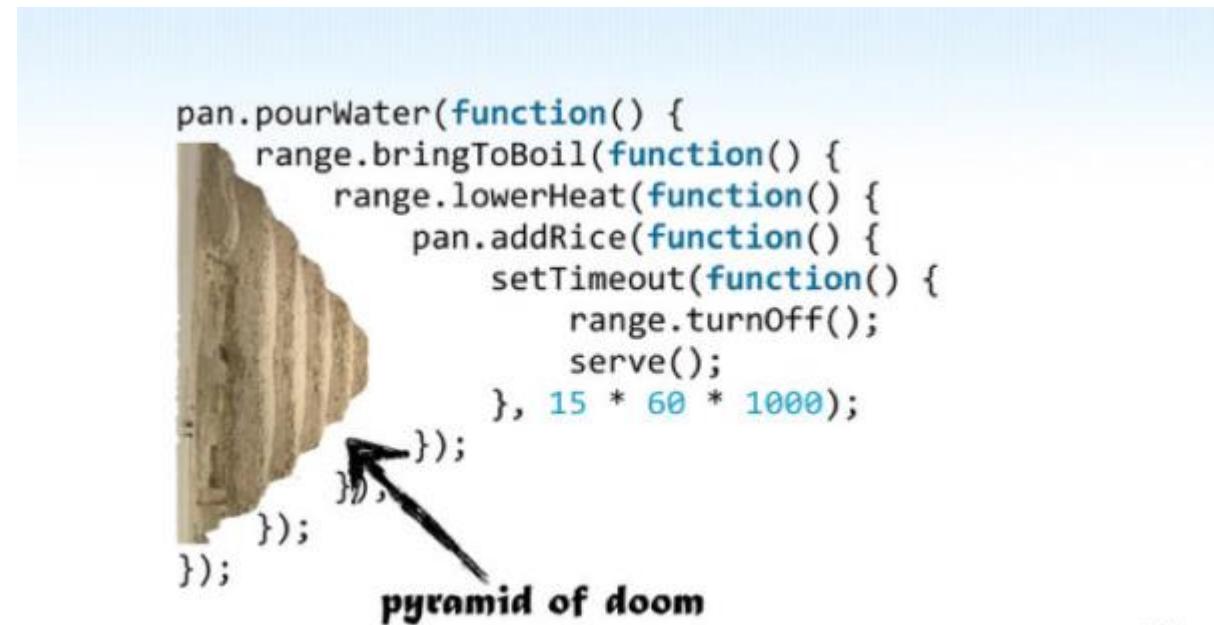
function register() {
    if (!empty($_POST)) {
        $msg = '';
        if ($_POST['user_name']) {
            if ($_POST['user_password_new']) {
                if ($_POST['user_password_new'] === $_POST['user_password_repeat']) {
                    if (strlen($_POST['user_password_new']) > 5) {
                        if (strlen($_POST['user_name']) < 65 && strlen($_POST['user_name']) > 1) {
                            if (preg_match('/^(a-zA-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



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 partie 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);
}) ;

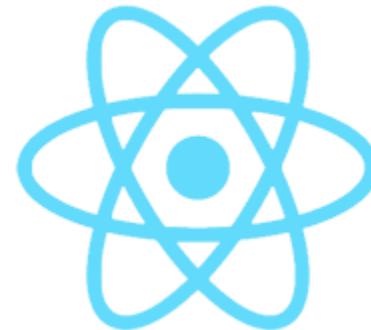
promise
    .then((res) => {
        console.log(res);

        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

Quelques bonnes pratiques

- ▷ Préférez toujours « `if (value === true)` » à « `if (value)` ».
- ▷ `camelCase` – `maVariable` – `mavariable`
- ▷ Indentation à `4 espaces` plutôt que `2` pour décourager les cascades de callback.
- ▷ Nommez toutes vos fonctions - déclarez et initialisez vos variables.
- ▷ Mettez toujours les « `;` » en fin de ligne, même si optionnel.
- ▷ Utilisez un maximum les nouvelles fonctionnalités `ES6` !

EcmaScript 6 / 2015

▷ Les classes !! Avec héritage

▷ Modules



▷ ➔ Arrow Functions

▷ { Template Strings }

▷ Spread & Rest

▷ Déstructuration (objet et array)

Les classes

EcmaScript 2015

```
class User {  
  constructor(firstName, lastName) {  
    this._firstName = firstName;  
    this._lastName = lastName;  
  }  
  
  /* Getter. */  
  firstName() {  
    return this._firstName;  
  }  
  /* Property. */  
  get lastName() {  
    return this._lastName;  
  }  
  set lastName(lastName) {  
    this._lastName = lastName;  
  }  
}
```

```
let user = new User('Foo');  
  
console.log(user.firstName());  
// => 'Foo'  
  
console.log(user.lastName());  
// => undefined  
  
user.lastName = 'BAR';  
console.log(user.lastName());  
// => 'BAR'  
  
console.log(user._lastName);  
// => 'BAR'
```

Les classes - Héritage

- ▷ La gestion de l'héritage a également été simplifiée

```
class Point {  
    constructor(x, y) {  
        this.x = x;  
        this.y = y;  
    }  
    toString() {  
        return this.x + ", " + this.y;  
    }  
}
```

```
class ColorPoint extends Point {  
    constructor(x, y, color) {  
        super(x, y);  
        this.color = color;  
    }  
    toString() {  
        return super.toString() \n  
        + " in " + this.color;  
    }  
}
```

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 pris 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@me.com'  
};
```

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

Spread & Rest

EcmaScript 2015

```
function add(...numbers) {  
    return numbers.reduce((lastSum, num) => lastSum + num);  
}  
const somme = add(1, 2, 3, 4);
```

Rest parameters

```
const arr1 = [1, 2, 3];  
const arr2 = [...arr1, 4]; // [1, 2, 3, 4] ← Spread !  
  
const obj1 = {a: 1, b: 2};  
const obj2 = {...obj1, a: 3, c: 1}; // {a: 3, b: 2, c: 1} ←
```

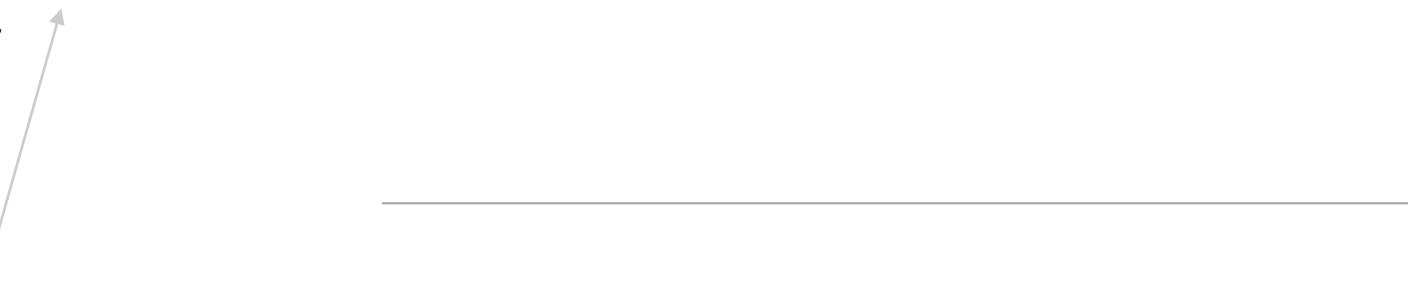
Depuis ES6, il est possible d'importer des fonctionnalités d'un fichier javascript vers un autre :

```
export class Personne {  
    firstname;  
}  
  
import { Personne } from './personne.class';  
  
let pers = new Personne();
```

personne.class.js

main.js

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

main.js



Quizz

```
var foo = 1;

function bar() {
    if(!foo) {
        var foo = 10;
    }
    console.log(foo);
}

bar();
```

> Que s'affiche-t-il dans la console?

```
var a = 1;
```

```
function b(a) {  
    console.log(a);  
    a = 10;  
}
```

```
b();  
console.log(a);
```

> Que s'affiche-t-il dans la console?

```
var a;  
var r2 = a || {name: 'toto'};  
  
console.log(r2);
```

> Que s'affiche-t-il dans la console?

```
var c = 10;  
var r2 = c  
    && function() { var b = 3 + 8; console.log(b); return b; }  
    && 'tt';  
  
console.log(r2);
```

> Que s'affiche-t-il dans la console?

```
var myObject = {};  
console.log(myObject.a);
```

```
var foo;  
console.log(foo);
```

```
console.log(bar);
```

> Que s'affiche-t-il dans la console?

```
var hi = function(name) {  
    return 'Hi ' + name;  
}  
  
var greeting1 = function(name) {  
    return hi(name);  
}  
  
var greeting2 = hi;  
  
console.log(greeting1('Abdel'));  
console.log(greeting2('Mazen'));
```

> Que s'affiche-t-il dans la console?

```
var obj = {  
    data: 'ma chaine'  
};
```

```
function myFunc() {  
    console.log(this);  
}
```

```
obj.myFunction = myFunc;
```

```
obj.myFunction();
```

> Que s'affiche-t-il dans la console?

```
var fact = function factorial(n) {  
    console.log(n);  
    return n === 0 ? 1 : (n * this.factorial(n - 1));  
}  
  
var r = fact(5);  
console.log(r);
```

> Ce code fonctionne-t-il ?

```
for(var i = 1; i < 4; i++) {  
    setTimeout(function() {  
        console.log(i);  
    }, 1000 * i)  
}
```

> Que s'affiche-t-il dans la console?

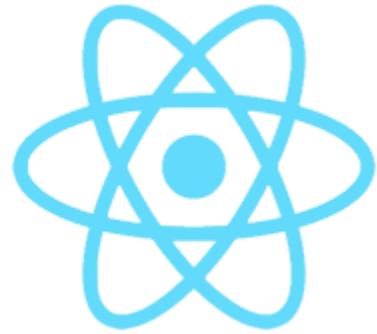
```
(function() {
    var createWorker = function() {
        var count = 0;
        var task1 = function() {};
        var task2 = function() {};

        return {
            job1: task1,
            job2: task2
        };
    };

    var worker = createWorker();
    worker.job2();
})();
```

> A quoi sert la syntaxe `(function () { ... })();` ?

Questions



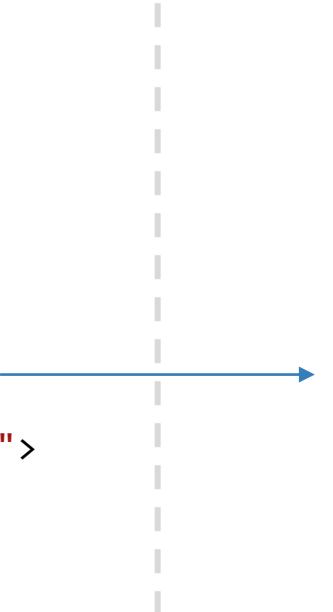
React

Développer avec ReactJS

Animé par Mazen Gharbi

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

Introduction

- ▷ Implémente la même logique côté front avec Javascript
- ▷ Voici un exemple de code React :

Vue d'un programme React

```
{  
  this.state.error &&  
  <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

Premiers pas

[Tester ce code](#)

```
<!DOCTYPE html>
<html>
<head>
  <meta charset="UTF-8"/>
  <script src="https://unpkg.com/react@16/umd/react.development.js"></script>
  <script src="https://unpkg.com/react-dom@16/umd/react-dom.development.js"></script>
  <script src="https://unpkg.com/babel-standalone@6.15.0/babel.min.js"></script>
</head>
<body>
<div id="root"></div>
<script type="text/babel">
  ReactDOM.render(
    <h1>Hello, world!</h1>, document.getElementById('root')
  );
</script>
</body>
</html>
```

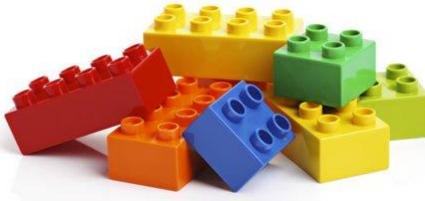
index.html

1
2
3

Code JSX

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 ;

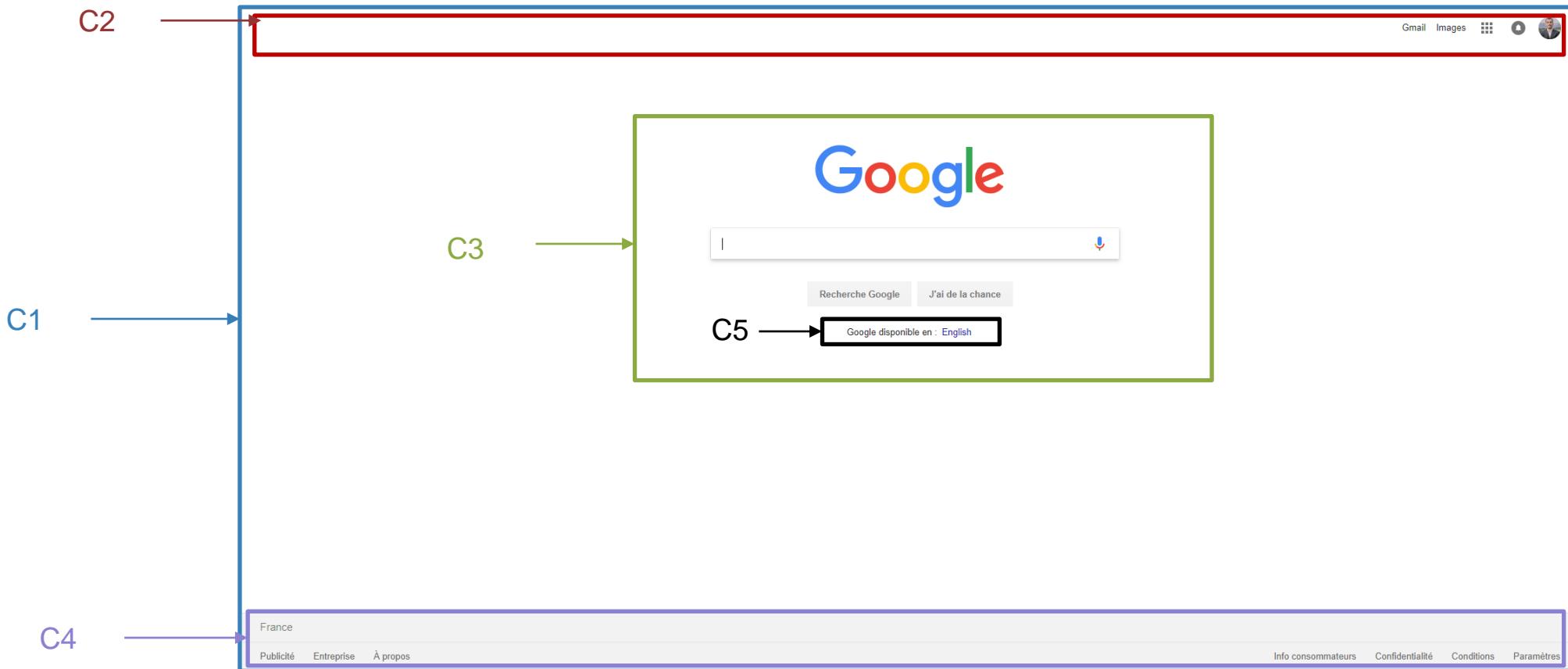


Les composants

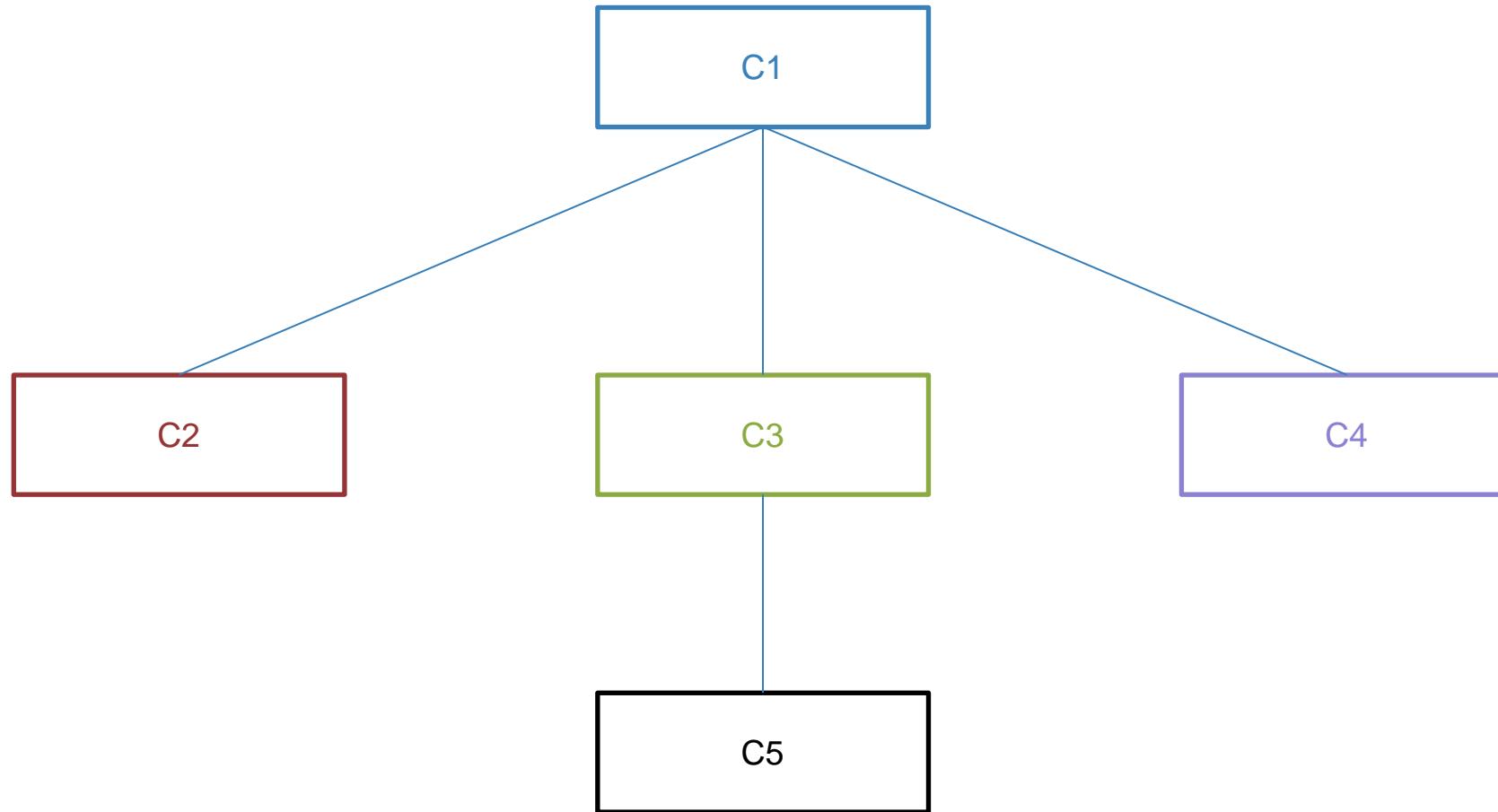
Pilier de la librairie

- ▷ 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



Composants React

- ▷ Un composant React peut être défini comme une classe ES6 qui étend la classe React :

```
import React from 'react';  
import ReactDOM from 'react-dom';  
  
export default class HelloScreen extends React.Component {  
  render() {  
    return (

Hello World!

);  
  }  
}  
  
ReactDOM.render(, document.getElementById('root'));
```

Importé au préalable

- ▷ Au minimum, un composant doit définir une méthode *render()* spécifiant le rendu du composant dans le DOM
 - › *Cette fonction doit obligatoirement être présente*
- ▷ La méthode de *render()* renvoie les noeuds React, qui peuvent être définis à l'aide de la syntaxe JSX en tant que balises de type HTML

Functionnal Components React

- ▷ C'est la nouvelle bonne pratique !

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

- ▷ Pas besoin de render cette fois !
- ▷ Il suffit de renvoyer la vue directement

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

Avec React, on peut...

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

App.js

```
class App extends Component {  
  render() {  
    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>  
    );  
  }  
}
```

JSX

Fourni par React. Définit un composant de base !

Et voici l'équivalent en Javascript...

Développer avec ReactJS

```
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."  
)  
) ;
```

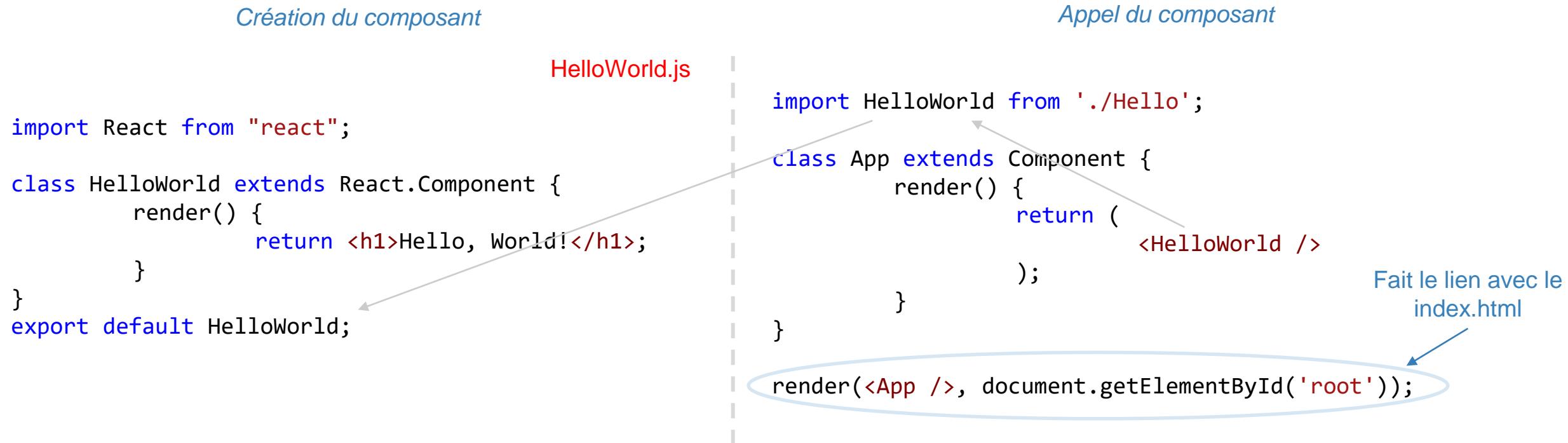
C'est quand même vachement mieux en JSX

JSX

- ▷ Comme vous avez pu le constater, la vue est directement intégrée dans le modèle ;
- ▷ Les développeurs React ont poussé jusqu'au bout leur vision d'avoir la logique intimement reliée à un élément.
- ▷ Le contenu a été intégré dans le même fichier afin de n'en avoir qu'un seul par composant.

JSX

- ▷ Le navigateur ne comprend pas le JSX. Cette syntaxe étant invalide.
- ▷ Certains outils permettent de « transformer » le JSX en code valide
 - › BabelJS



Avec React, on peut aussi...

- ▷ Intégrer aisément notre composant racine à notre page :

```
main.js  
Fournit par react  
import registerServiceWorker from './registerServiceWorker';  
  
ReactDOM.render(<App />, document.getElementById('root'));  
registerServiceWorker();
```

Permet la création d'un ServiceWorker pour:

- Faire tourner l'appli offline (gestion cache)
- La mise en place d'une PWA par la suite

JSX

- ▷ Si on veut ajouter des attributs HTML, ils doivent avoir les mêmes noms qu'en créant des éléments en JS et écrits en camelCase :



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

Les propriétés des composants

- ▷ 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 »

The diagram illustrates the flow of props from a Parent component to a Child component. A blue arrow points from the 'color' prop in the Parent's render method to the 'color' prop in the Child's style attribute.

```
render() {
  return (
    <div>
      <HelloWorld color='green' />
    </div>
  );
}

class HelloWorld extends React.Component {
  render() {
    return (
      <h1 style={{ color: this.props.color }}>React c'est le plus mieux</h1>
    );
  }
}
```

Parent

Enfant

Passage de propriétés - fonctions

- ▷ Rien ne change côté parent !

```
return (
  <div>
    <HelloWorld color='green' />
  </div>
);
```

Parent

```
function HelloWorld (props) {
  return (
    <h1 style={{ color: props.color }}>React c'est le plus mieux</h1>
  );
}
```

Enfant

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



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   },

```

✓ formData

```

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

```

A registration form

A simple form example.

First name*

Chuck

Last name*

Norris

Age

75

Bio

Roundhouse kicking asses since 1940

Password

Hint: Make it strong!

Submit

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
- ▷ Équivalent à la propriété data dans VueJS ou \$scope dans AngularJS.

State

- ▷ Pour commencer, on doit appeler le constructeur parent avec les mêmes paramètres donnés à notre constructeur.

```
class HelloWorld extends React.Component {  
  constructor(props) {  
    super(props); // Obligatoire si on souhaite accéder à this.props dans le constructeur  
    this.state = { counter: 0 };  
  }  
  
  render() {  
    return (  
      <h1>Hello World: {this.state.counter} times</h1>  
    );  
  }  
}
```

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](#)

State

▷ Note: Le binding dans le constructeur est nécessaire pour notre méthode increment car elle sera appelée dans une callback où **this** fait référence à window

```
class HelloWorld extends React.Component {
    constructor(props) {
        super(props);
        this.state = {
            value: 0
        };

        this.increment = this.increment.bind(this);
    }

    increment() {
        this.setState({
            value: this.state.value + 1
        });
    }

    ...
}
```



Mauvaise pratique

State

- ▷ `setState` est exécutée de manière asynchrone ;
- ▷ Pour éviter les erreurs, on préfèrera cette pratique :

```
class HelloWorld extends React.Component {  
    constructor(props) {  
        super(props);  
        this.state = {  
            value: 0  
        };  
  
        this.increment = this.increment.bind(this);  
    }  
  
    increment() {  
        // Modification dynamique  
        this.setState(oldState => ({  
            value: oldState.value + 1  
        }));  
    }  
  
    ...  
}
```



Callback attendu !

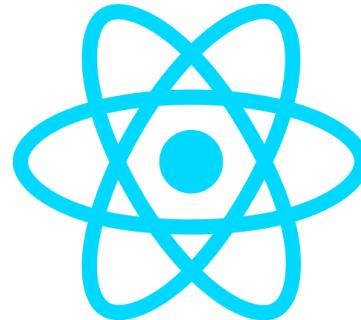
State - Affichage

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

Permet d'accéder aux variables de l'état actuel 😊

Peut être appliquée sur n'importe quel élément

Un composant enfant n'hérite pas du state père !



create-react-app

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 scripts 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 automatiseur 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
```

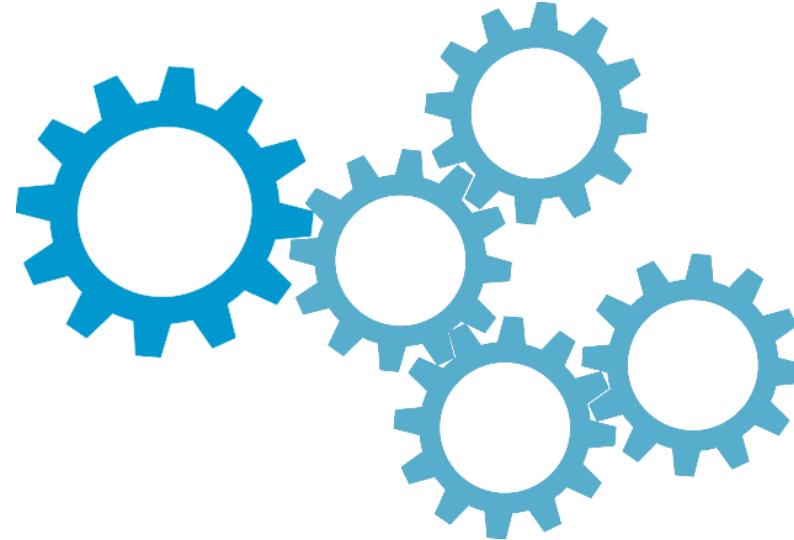
Configuration

- ▷ Par défaut, `create-react-app` est volontairement opaque ET non configurable ;
- ▷ Parfois, on souhaite appliquer d'autres configurations
 - › utiliser un langage CSS compilé comme Sass par exemple
- ▷ Pour y arriver, on lance la commande d'éjection :

```
> npm run eject
```

- ▷ Permet ainsi de modifier les fichiers de configuration
 - › C'est irréversible !

Question ?

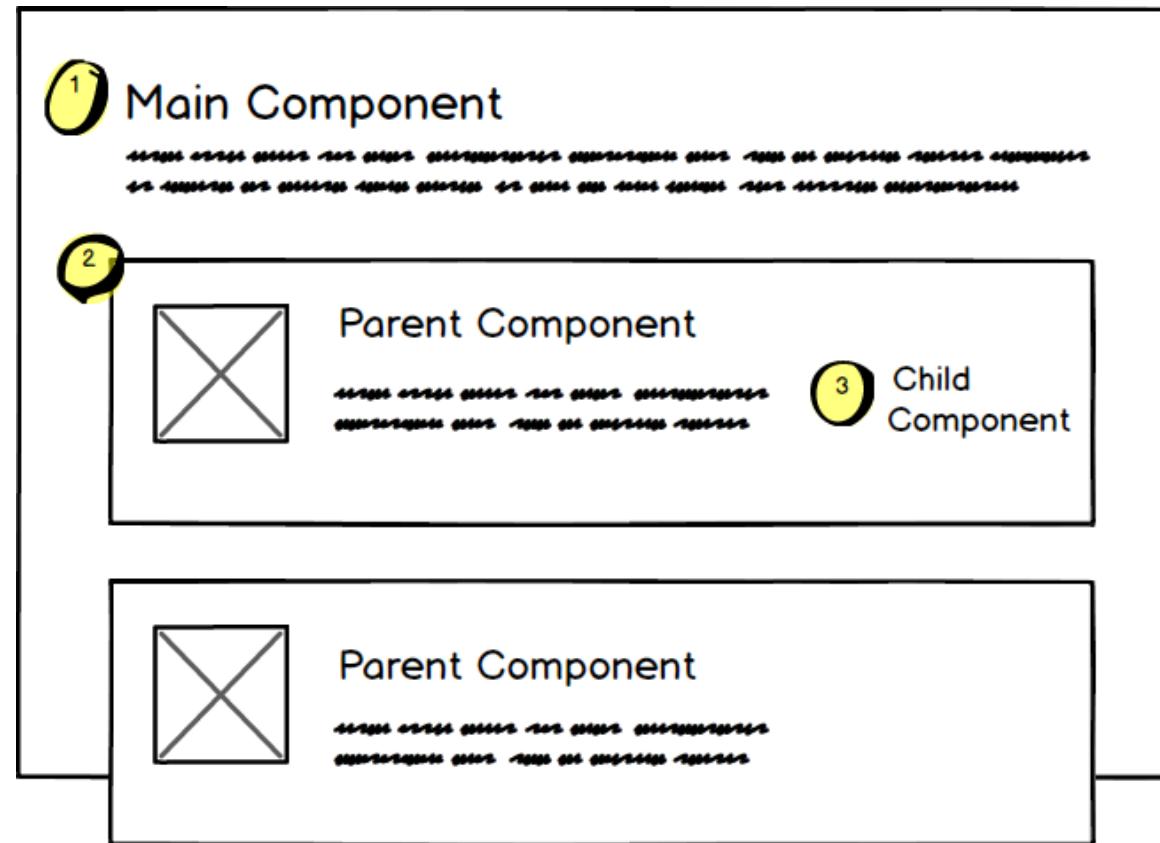


Manipuler les composants

Animé par Mazen Gharbi

Comprendre les composants

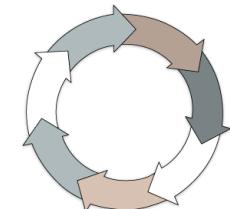
“Not only do React components map cleanly to UI components, but they are self-contained. The markup, view logic, and often component-specific style is all housed in one place. This feature makes React components reusable.”



Cycle de vie des composants

▷ Un composant bénéficie automatiquement de méthodes de gestion de son cycle de vie :

- › componentDidMount: le composant a été créé et son contenu a été ajouté dans le DOM
- › shouldComponentUpdate: permet d'éviter (ou pas) le ré-interpréter du composant. Pratique si les props ou le state sont modifiés de manière intensive
- › componentDidCatch: Un composant enfant a déclenché une erreur (et ne peut s'afficher)



[Plus d'infos ici](#)

Functional component

- ▷ Un composant fonctionnel est une fonction qui prend les props en paramètres et retourne du JSX ;
- ▷ Aucun state ;
- ▷ Ne bénéficie pas des méthodes de gestion de cycle de vie ;
- ▷ Très léger !

```
import React from 'react';

export default function Hello() {
    return <div>Bonjour tout le monde !</div>;
}
```

<Hello />

On l'appelle comme un
composant habituel

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.

Un functional component

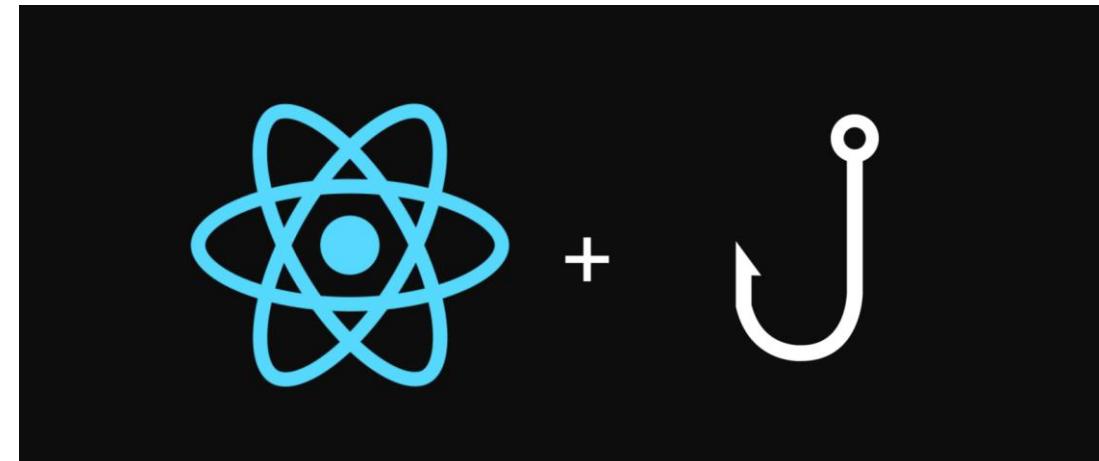
```
export default function MonComposant() {
    const b = 'Bonjour';

    function reagir() {
        console.log("b : " + b);
        console.log("Vous avez cliqué sur le bouton");
    }

    return (
        <div>
            <p>Ce composant est fonctionnel !</p>
            <p>Valeur de b = {b}</p>
            <button onClick={reagir}>Cliquez ici</button>
        </div>
    )
}
```

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>
    );
}
```

Composition

- ▷ Avec les **classes**, on réfléchissait notre application comme une **imbrication de composants**
- ▷ Avec les **fonctions**, on réfléchit notre application comme une **imbrication de fonctionnalités**
- ▷ Logiquement plus modulaire – L'objectif principal des hook a été de simplifier l'architecture d'un projet React conséquent
- ▷ Avec les **hooks**, même le state peut être extrait d'un composant !
 - › Permet donc la factorisation de bien plus de chose qu'auparavant

Code plus lisible

▷ Avec les hooks, le code suit un cheminement bien plus logique

```

import * as React from "react";
import { Card, Row, Input, Text } from "./components";
import ThemeContext from "./ThemeContext";

export default class Greeting extends React.Component {
  constructor(props) {
    super(props);
    this.state = {
      name: "Harry",
      surname: "Potter",
      width: window.innerWidth
    };
    this.handleNameChange = this.handleNameChange.bind(this);
    this.handleResize = this.handleResize.bind(this);
    this.handleSurnameChange = this.handleSurnameChange.bind(this);
  }

  componentDidMount() {
    window.addEventListener("resize", this.handleResize);
    document.title = this.state.name + ' ' + this.state.surname
  }

  componentDidUpdate() {
    document.title = this.state.name + ' ' + this.state.surname
  }

  componentWillUnmount() {
    window.removeEventListener("resize", this.handleResize);
  }

  handleNameChange(name) {
    this.setState({ name });
  }

  handleSurnameChange(surname) {
    this.setState({ surname });
  }

  handleResize() {
    this.setState({ width: window.innerWidth });
  }

  render() {
    let { name, surname, width } = this.state;
    return (
      <ThemeContext.Consumer>
        {theme =>
          <Card theme={theme}>
            <Row label="Name">
              <Input value={name} onChange={this.handleNameChange} />
            </Row>
            <Row label="Surname">
              <Input value={surname} onChange={this.handleSurnameChange} />
            </Row>
            <Row label="Width">
              <Text>{width}</Text>
            </Row>
          </Card>
        }
      </ThemeContext.Consumer>
    );
  }
}

```

```

import React, { useState, useContext, useEffect } from "react";
import { Card, Row, Input, Text } from "./components";
import ThemeContext from "./ThemeContext";

export default function Greeting(props) {
  let theme = useContext(ThemeContext);

  let [name, setName] = useState("Harry");
  let [surname, setSurname] = useState("Potter");
  useEffect(() => {
    document.title = name + " " + surname;
  });

  let [width,setWidth] = useState(window.innerWidth);
  useEffect(() => {
    let handleResize = () => setWidth(window.innerWidth);
    window.addEventListener("resize", handleResize);
    return () => {
      window.removeEventListener("resize", handleResize);
    };
  });

  return (
    <Card theme={theme}>
      <Row label="Name">
        <Input value={name} onChange={setName} />
      </Row>
      <Row label="Surname">
        <Input value={surname} onChange={setSurname} />
      </Row>
      <Row label="Width">
        <Text>{width}</Text>
      </Row>
    </Card>
  );
}

```

Class VS Hooks

useState

- ▷ « useState » est notre nouvelle méthode pour manipuler l'état du composant actuel
 - › Nous ne manipulons plus d'objet state global désormais et tant mieux !

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

- ▷ 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" });
```

Hooks d'effets

- ▷ Avec les functional components, plus les méthodes pour interagir avec le cycle de vie d'un composant

```
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 du composant
  };
});
```

- ▷ 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 réexecuté 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 propres Hooks

- ▷ 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`;  
}
```

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 pas la propType 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**

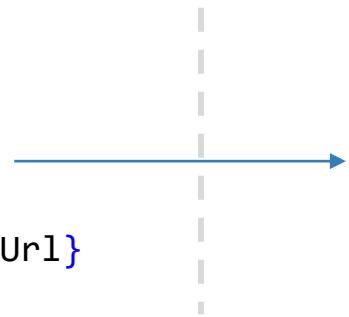
Children

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

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

```
<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}
/>;
```

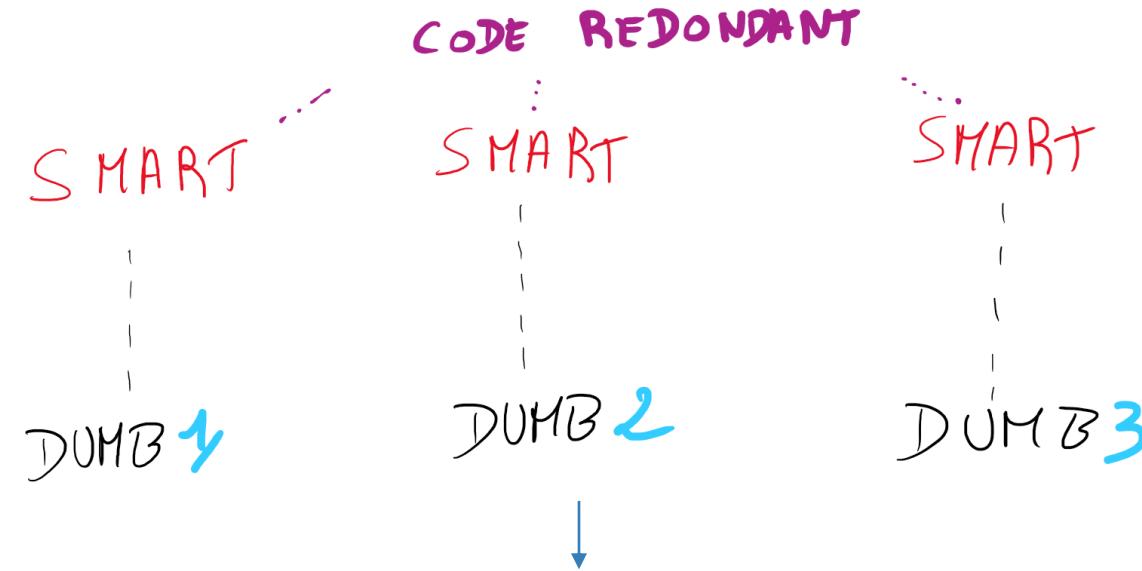


```
<Product
  key={"product-" + product.id}
  {...product}
  onVote={this.handleProductUpVote}
/>;
```

High Order Components

- ▷ Technique avancée pour optimiser la réutilisabilité
- ▷ Pour rappel, un composant permet de générer une vue réutilisable
- ▷ Un HOC est un composant **transforme un composant en... un autre composant !** Cela nous permettra de factoriser des fonctionnalités entre différents composants
- ▷ Il y a pas les mixins pour ça ?
 - › Mauvaise pratique ! En savoir plus [ici](#)

High Order Components



- ▷ Dans ce scénario, le H.O.C. permettrait d'automatiser la « génération » des **Smart** en prenant en paramètre leurs **Dumb** respectifs

Un exemple très simple

```
export default Component => (color, { ...props }) => {
  let colorToApply = color;
  if (color === 'red') {
    colorToApply = 'blue';
  }

  return <Component color={colorToApply} {...props} />;
};
```

```
export default function App() {
  const NewList = HOC(MaListe);
  const NewSayHi = HOC(SayHi);
  return (
    <div>
      <NewList />
      <SayHi />
    </div>
  );
}
```



Routing

Deep Linking

https://www.macademia.fr/admin/users.php

The URL is broken down into three parts: 'Protocole' (Protocol) points to 'https://', 'Nom de domaine' (Domain Name) points to 'www.macademia.fr', and 'Chemin vers la ressource' (Path to Resource) points to '/admin/users.php'.

- ▷ 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 « *mémorisateur d'état externe* »
 - › Il sert par exemple à ce qu'un utilisateur enregistre la page actuel 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
```

- ▷ Meilleur librairie du moment pour la gestion des routes

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

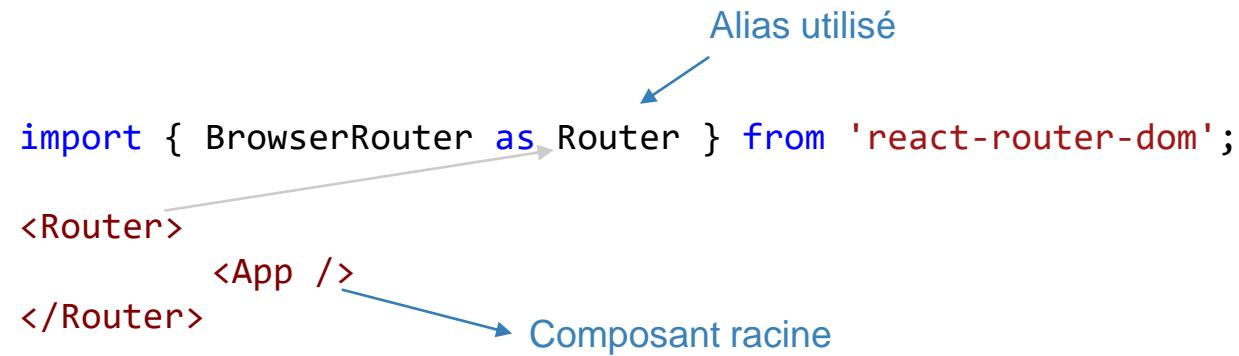
Mise en place

- ▶ 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

```
// On est actuellement dans le composant App
```

```
<Route path='/accueil' element={<HomeComponent />} />
<Route path='/utilisateurs' element={<UsersComponent />} />
<Route path='/details' element={<DetailsComponent />} />
```

Composants créés
au préalable

- ▷ Le mot-clé « **exact** » permet d'indiquer que l'on souhaite que la route soit EXACTEMENT celle-ci

- › Par défaut, le système de routing est très permissif, et une route comme « **/detailsblabla** » OU « **/details/1/2** » est valide pour afficher **DetailsComponent**

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 }  
}}  
/>;
```

The diagram illustrates the properties of a `Link` component object and their effects:

- `pathname: '/courses'` → Route visé (Targeted route)
- `search: '?sort=name'` → Paramètres GET (GET parameters)
- `hash: '#the-hash'` → Ancre (Anchor)
- `state: { fromDashboard: true }` → Mise à jour du state dans le composant de la route sur laquelle on arrive (Updating the state in the component of the route to which we arrive)

Mise en place du routing

- ▷ En React, chaque route est représentée par un Composant ;

<https://stackblitz.com/edit/macademia-react-router-basic>

- Route imbriquée : <https://stackblitz.com/edit/macademia-react-router-imbrique>

Routes

```
class App extends Component {  
    constructor(props) {  
        super(props);  
        this.state = { name: 'React' };  
    }  
  
    render() {  
        return (  
            <Router>  
                <Routes>  
                    <Menu />  
                    <Route path='/accueil' element={<Accueil />} />  
                    <Route path='/contact' element={<Contact />} />  
                    <Route path='/forum' element={<Forum />} />  
                </Routes>  
            </Router>  
        );  
    }  
}
```

Routes imbriquées

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

App.js

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

```
render() {
  return (
    <div>
      Contenu de la page Accueil -
      <Link to='/accueil/details'>
        <a> Détails</a>
      </Link>
      <Outlet /> Provoque l'affichage des routes enfants
    </div>
  );
}
```

Accueil.js

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, { Component } from 'react';
import { useParams } from 'react-router';

class Profile extends Component {
    constructor(props) {
        super(props);
        this.state = {
            user: null
        };
    }

    componentDidMount() {
        // Récupération du paramètre de route
        const { handle } = useParams();
        // fetch permet de requêter un serveur
        fetch(`https://api.twitter.com/user/${handle}`).then((user) => {
            this.setState((previousState) => {
                return {
                    user // remplacé par user: user
                }
            });
        });
    }

    this.render() {
        ...
    }
}
```

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>
```

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

app.js

contact.js

Tout est là

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();
  ...
}
```



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 apparaît :

```
class ControlledInput extends React.Component {  
  render() {  
    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

Formulaire

- ▷ 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'user ?

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>
  )
}
```

Les références avec les classes

- ▷ Comme vu précédemment, il existe un moyen pour relier les éléments du DOM avec le modèle grâce à la propriété « ref » ;

```
componentDidMount ()  {
  console.log(this.refs.element);
}

render ()  {
  return (<div ref="element"> Contenu de la div </div>);
}
```

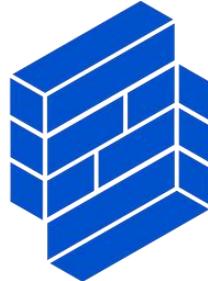
Et avec les functional components ?

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

```
export default function MiniForm() {  
  const firstnameRef = useRef(); Création de la référence  
  
  function onSubmitForm(event) {  
    event.preventDefault(); Annule le rechargement de la page  
    console.log(firstnameRef.current.value);  
  }  
  
  return (  
    <form onSubmit={onSubmitForm}>  
      <input ref={firstnameRef} /> On applique la référence à l'input  
      <button type="submit">Valider</button>  
    </form>  
  );  
}
```

Exemple

<https://stackblitz.com/edit/react-macademia-simple-form>



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 }) => (
    <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>}
```



ImmutableJS

ImmutableJS

- ▷ Un state doit être considéré comme **immutable** !
 - › Evite ainsi les erreurs d'affichage dues à l'accès concurrentiel des datas
- ▷ Soit un reducer gérant notre utilisateur :

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

Ce code n'est donc pas bon.

ImmutableJS

▷ Un code correct serait :

```
case UPDATE_DATA:  
  const { index, newFirstname } = action.payload;  
  
  return {  
    ...state,  
    users: this.state.users.map((user, indexIteration) => {  
      if(indexIteration === index) {  
        return {  
          ...user,  
          firstname: newFirstname  
        }  
      }  
      return user;  
    })  
  }  
}
```

Long et rébarbatif !

C'est un peu verbeux oui.

ImmutableJS

- ▷ Librairie qui nous simplifiera la vie
- ▷ Permet de mettre à jour un objet sans modifier la référence initiale
 - › Renvoie un clone
- ▷ Librairie à installer
 - › Crée et maintenue par Facebook

```
> npm i immutable
```

- ▷ Obligation de passer par des Objets Immutable désormais !

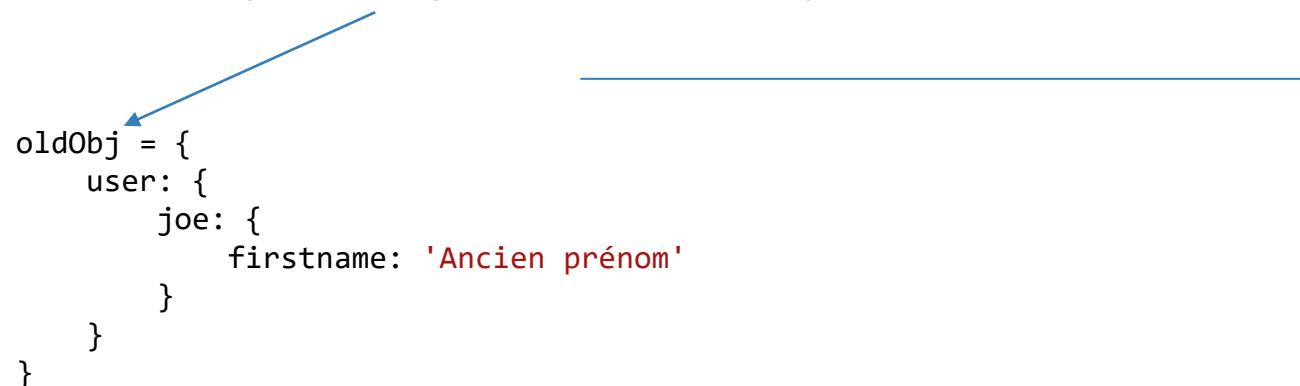
Quelques méthodes pratiques

```
setIn<C>(collection: C, keyPath: Iterable<any>, value: any): C
```

- ▷ Permet de naviguer dans une collection (Tableau / Objet) afin de mettre à jour une donnée selon un « keypath »

Exemple :

```
const newObj = oldObj.setIn(['user', 'joe', 'firstname'], valueNewFirstname);
```



Quelques méthodes pratiques

`fromJS(jsValue: any, [...]): any`

- ▷ Permet de convertir un objet Javascript vanilla en un objet Immutable

Exemple : `let newState = fromJS(this.state);`

`[immutableObject].toJS(): any`

- ▷ A l'inverse, convertit un objet Immutable en JS vanilla

Exemple : `this.setState(newState.toJS());`

ImmutableJS

Avant

```
increment = (i) => {
  this.setState({
    ...this.state, // Plus nécessaire depuis plusieurs mois
    objects: this.state.objects.map((info, index) => {
      if (index === i) {
        return {
          ...info,
          value: info.value + 1
        }
      }
      return info;
    })
  })
}
```

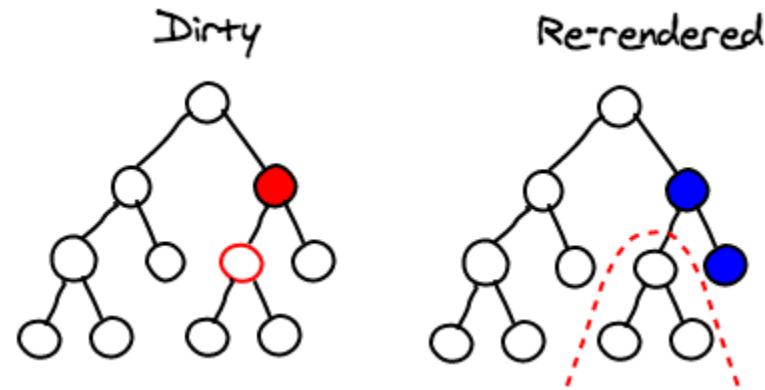
Après

```
let newState = fromJS(this.state);

newState = newState.setIn(
  ["objects", i, "value"],
  this.state.objects[i].value + 1
);

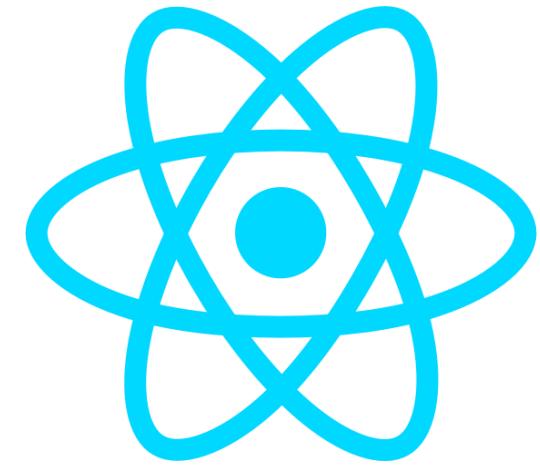
this.setState(newState.toJS());
```

Questions ?



Performances

Animé par Mazen Gharbi



1

Détection

2

Repérage

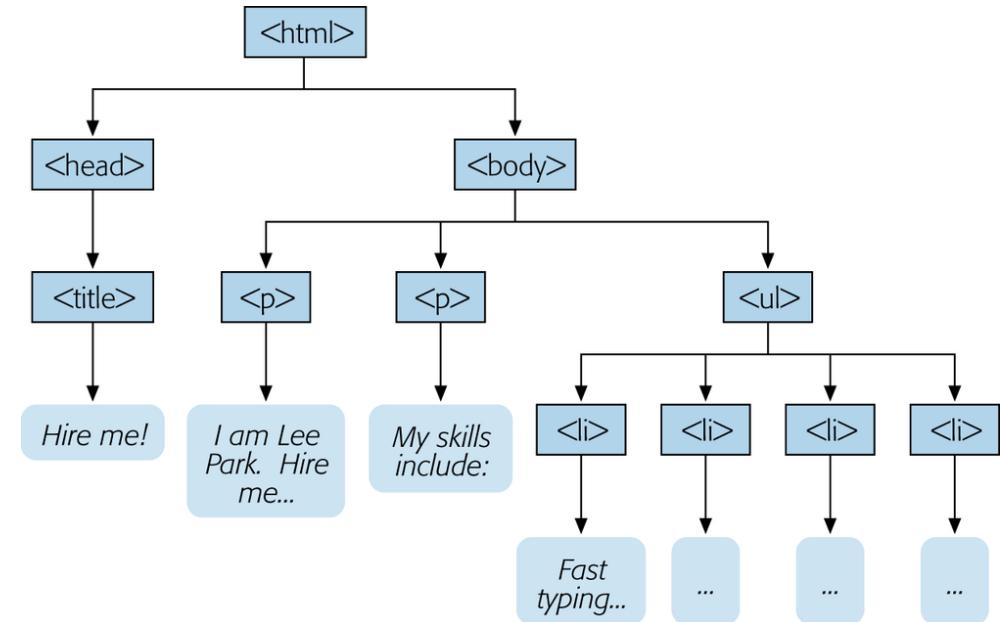
3

Changement



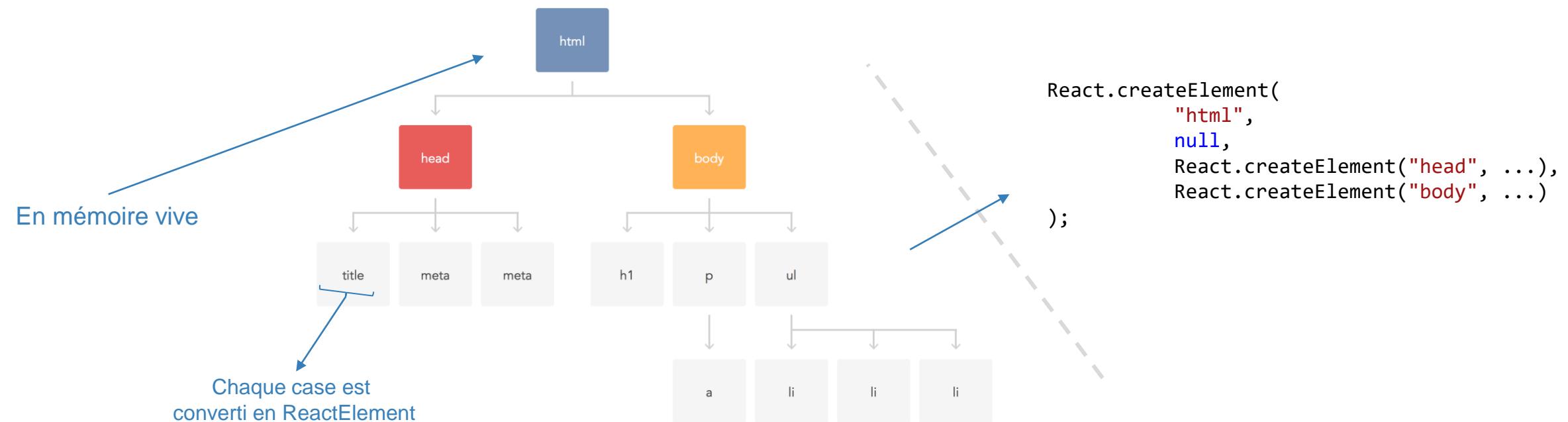
Document Object Model

- ▷ Chaque page Web est représentée en interne en tant qu'arbre d'objets (DOM)
- ▷ Les opérations de réaffichage du DOM sont extrêmement coûteuses .
- ▷ React va donc chercher à minimiser le nombre de fois où l'on va réafficher la page

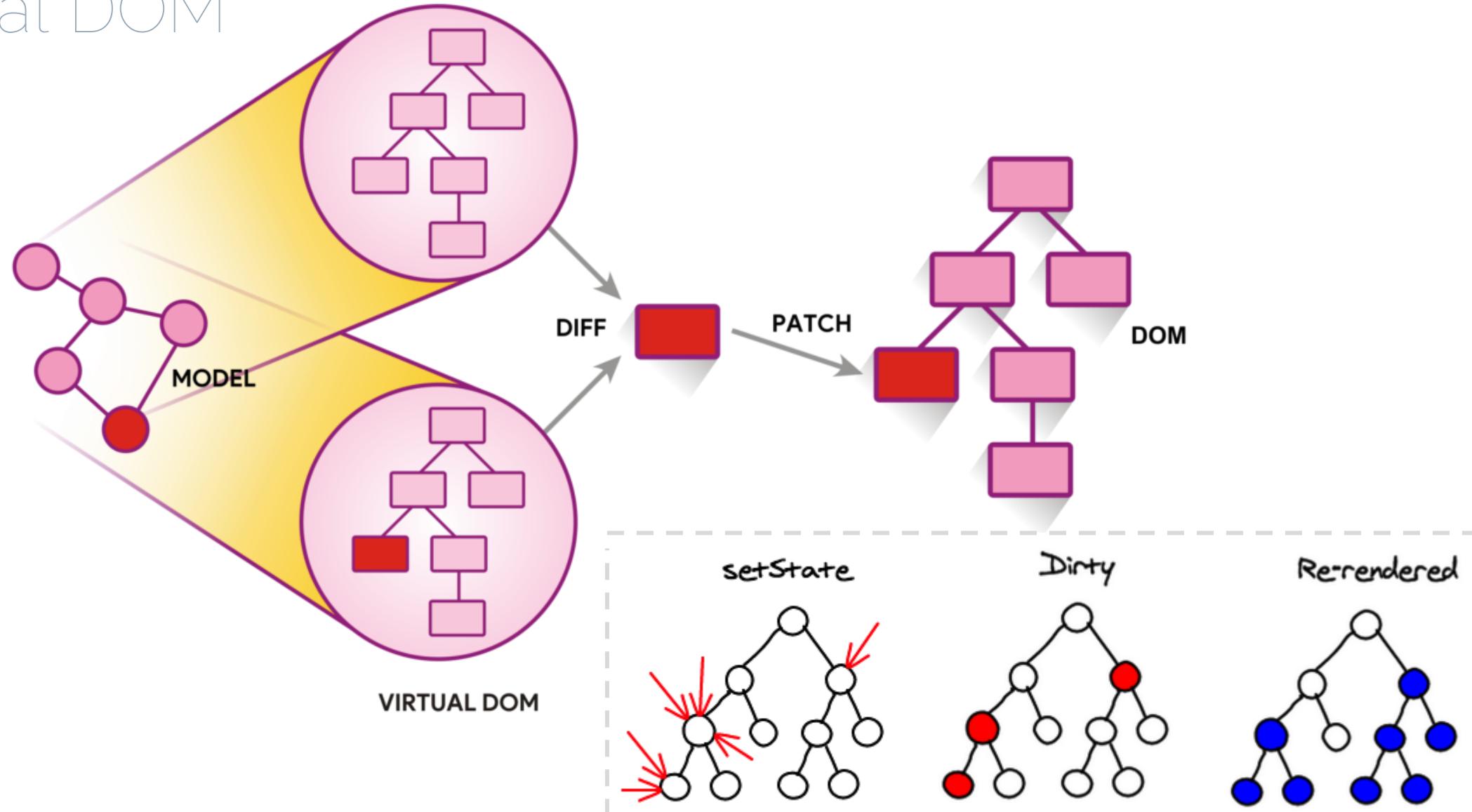


Virtual DOM

- ▷ React construit une représentation **virtuelle** du DOM actuel ;
 - › *Ne pas confondre avec le Shadow DOM*
- ▷ Chaque nœud de l'arbre est représenté par un **objet Javascript** ;

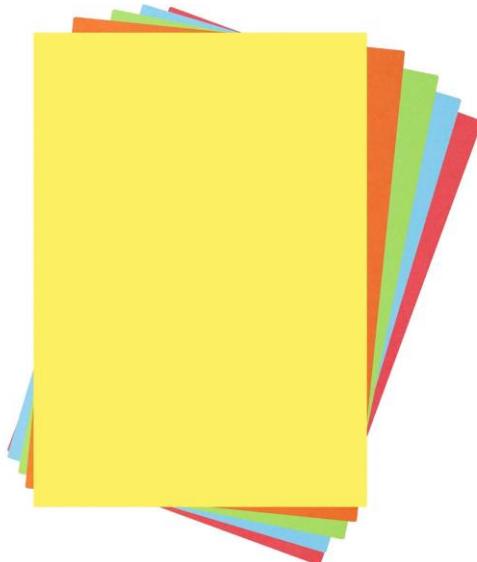


Virtual DOM



Diffing

▷ Une fois le DOM virtuel mis à jour, React compare le DOM virtuel avec un snapshot du DOM virtuel pris juste avant la mise à jour ;

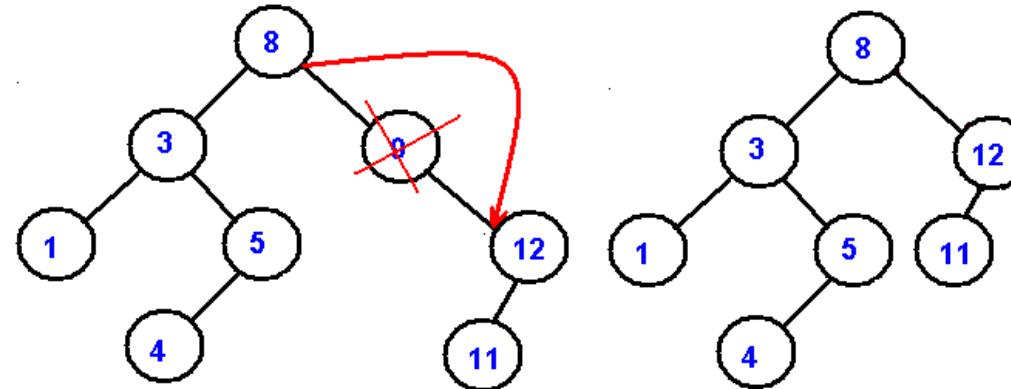


▷ En comparant le nouveau avec une version précédente, React détermine exactement quels objets DOM virtuels ont changés. Ce processus est appelé « diffing » ;

Diff

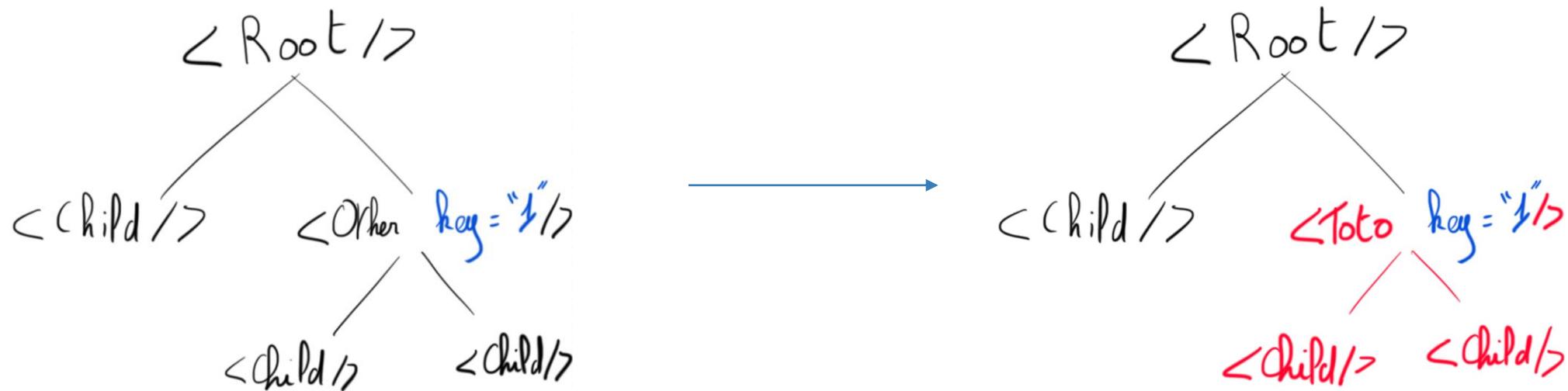
- ▷ Remplacer un arbre par un autre coûte cher en terme de performance
 - › $O(n^3)$
- ▷ React évite à tout prix de modifier l'entièreté de l'arbre à chaque fois
- ▷ Il se base sur 2 conditions pour établir si une partie de l'arbre doit changer :
 1. 2 composants de la même classe généreront des arbres similaires et deux composants de classes différentes généreront des arbres différents ;
 2. Il est possible de fournir une clé unique pour les éléments stables sur différents rendus.

$O(n^3) \Rightarrow O(n)$



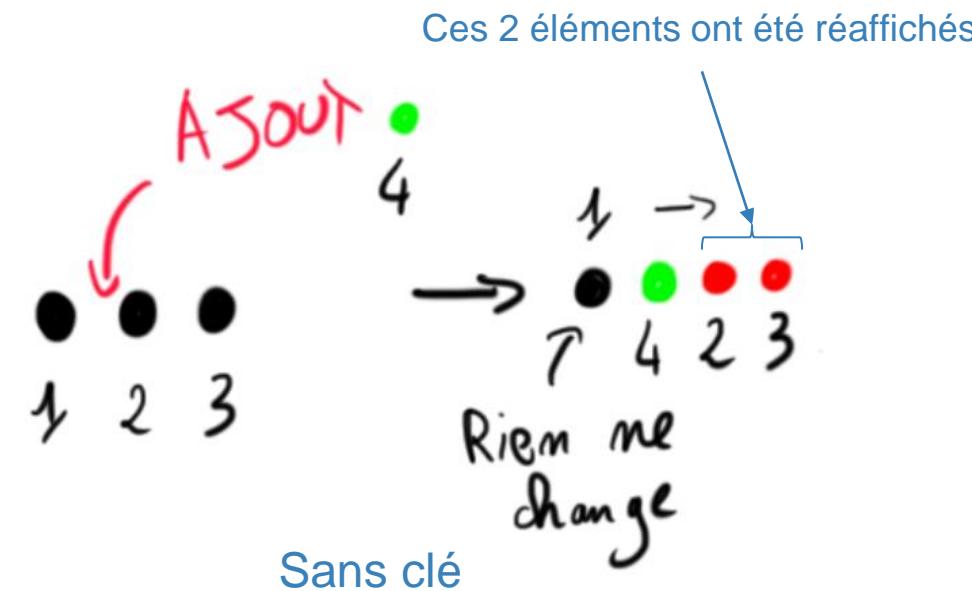
Diff – Différence de types

- ▷ Pour résumé :
- ▷ Un sous-arbre doit être réaffiché si les types ne sont pas les mêmes

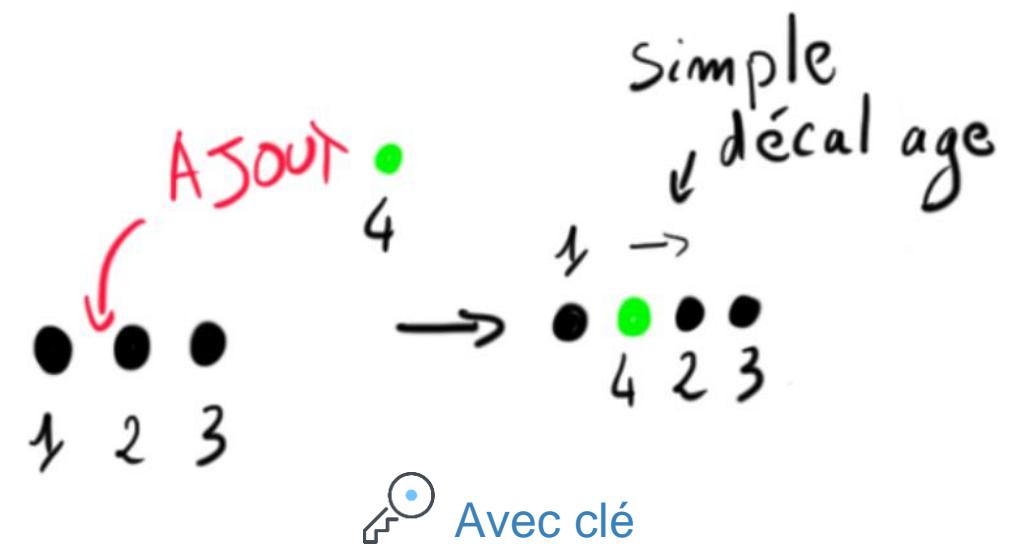


Diff – Différence de clés

- Le réaffichage se fait pour l'élément dont la clé a changé uniquement



```
<ul>
  <li>1</li>
  <li>2</li>
  <li>3</li>
</ul>
```



```
<ul>
  <li key="uniq1">1</li>
  <li key="uniq2">2</li>
  <li key="uniq3">3</li>
</ul>
```

Bonnes pratiques

- ▷ Parfois, 2 composants différents peuvent générer une sortie similaire
 - › React préconise dans ce cas de factoriser en 1 composant unique pour éviter le réaffichage suite au diffing ;
- ▷ Pensez à toujours appliquer une clé à vos éléments itérés ;

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

- ▷ *N'utilisez pas l'index de l'élément comme clé !*

Optimise les performances

- ▷ A chaque fois qu'une clé change, React considère que l'élément doit être réaffiché
- ▷ Or, il peut arriver qu'une propriété d'un composant change **sans que la vue ne soit impactée**

```
class MonComposant extends React.Component {  
    // Nous comparons nous même l'état des states,  
    // On renvoie true si on considère que la vue doit se mettre à jour  
    // false sinon  
    shouldComponentUpdate(nextProps, nextState) {  
        if (nextProps === 'toto') {  
            return false;  
        }  
  
        return true;  
    }  
}
```

Mesurer les performances

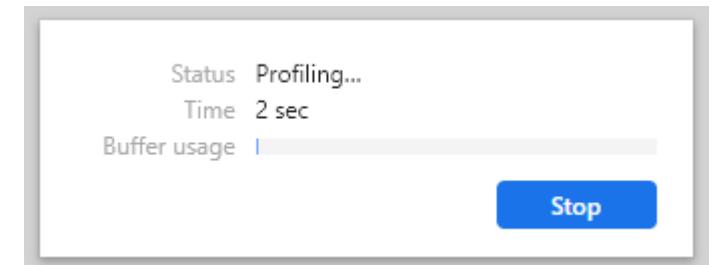
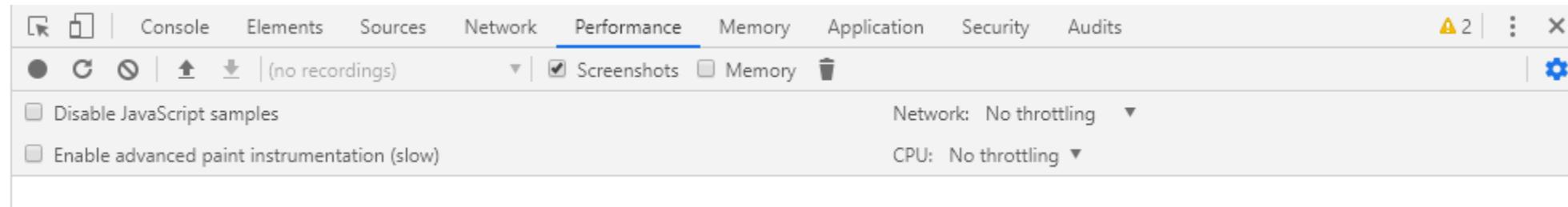
- ▷ Afin d'améliorer les performances de notre application, il va être nécessaire de les **mesurer** !
- ▷ Pour ce faire, React propose une librairie : **react-addons-perf**

```
> npm install --save react-addons-perf
```

- ▷ Mais obsolète depuis React 16 !!
- ▷ Désormais, l'équipe React préconise le profiler du navigateur

Mesurer les performances

- ▷ Ces manipulations doivent être faites avec toutes les extensions chrome désactivées ! (ou en navigation privée)
- ▷ Ouvrez votre console, et rendez vous sur l'onglet « Performance »

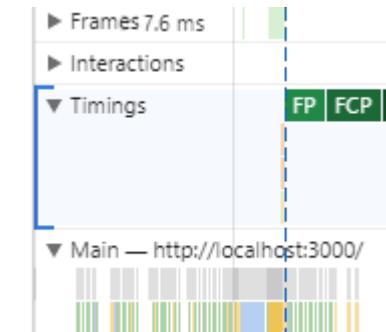


- ▷ Lancer le « record » puis rechargez la page

Mesurer les performances

[Plus d'infos](#)

- ▷ Tout va se passer dans la vue « Timings »



- ▷ Et voici le résultat :

Self Time		Total Time		Activity
5.9 ms	31.4 %	12.0 ms	64.4 %	▶ App [mount]
5.8 ms	30.9 %	5.8 ms	30.9 %	▶ Test [mount]
3.8 ms	20.2 %	15.8 ms	84.6 %	▶ (React Tree Reconciliation: Completed Root)
1.2 ms	6.2 %	1.2 ms	6.2 %	▶ (Committing Snapshot Effects: 0 Total)
0.9 ms	4.8 %	0.9 ms	4.8 %	▶ (Committing Host Effects: 1 Total)
0.5 ms	2.8 %	2.9 ms	15.4 %	▶ (Committing Changes)
0.4 ms	2.1 %	0.4 ms	2.1 %	▶ DireBonjour [mount]
0.3 ms	1.6 %	0.3 ms	1.6 %	▶ (Calling Lifecycle Methods: 0 Total)

Composant père

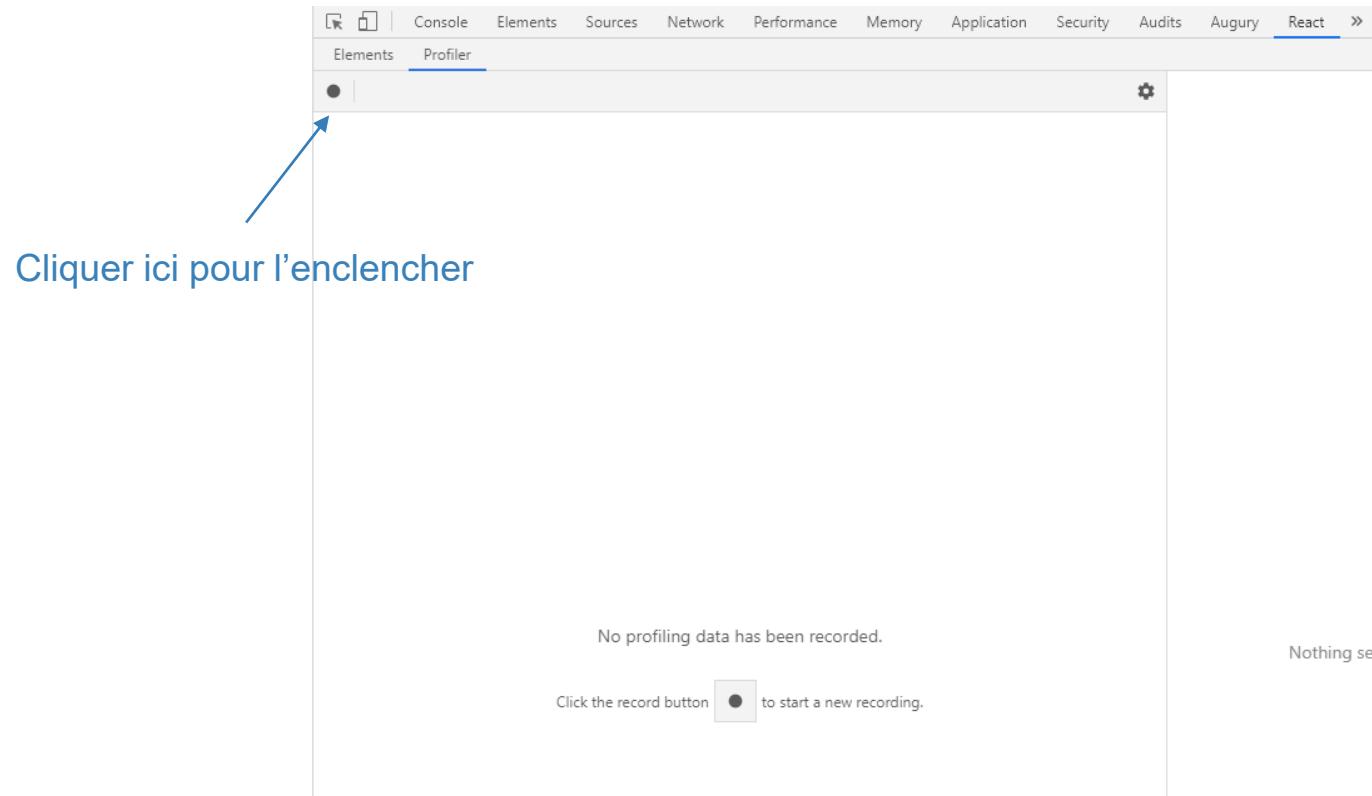
Mes 2 composants enfant

5.8 ms	30.9 %	▼ Test [mount]
5.8 ms	30.9 %	▼ App [mount]
5.8 ms	30.9 %	▶ (React Tree Reconciliation: Completed Root)

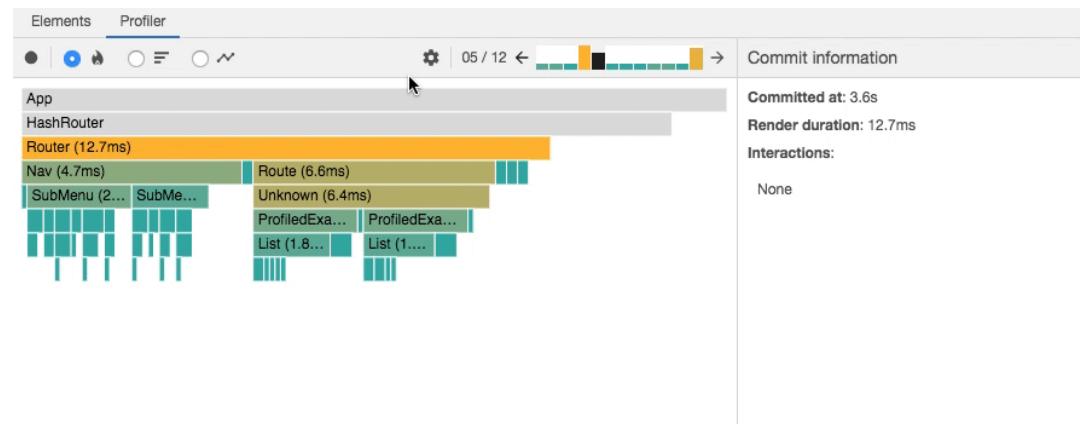
On voit ici que c'est principalement le ré-affichage du composant qui fut long

Mesurer les performances

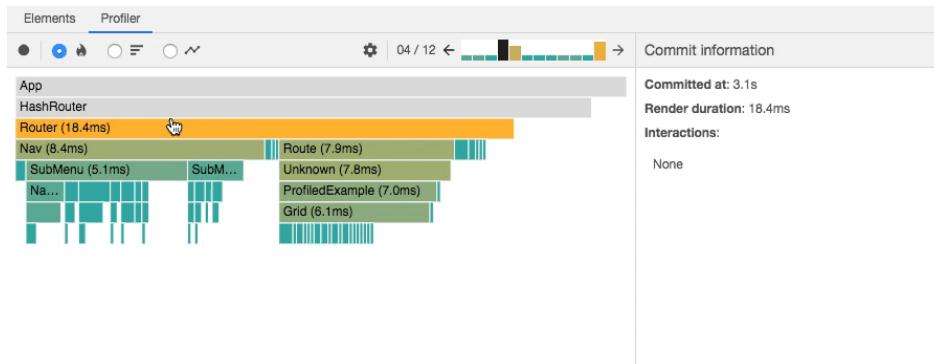
- ▷ Enfin, depuis peu, un profiler a été intégré directement dans le React Dev Tools :



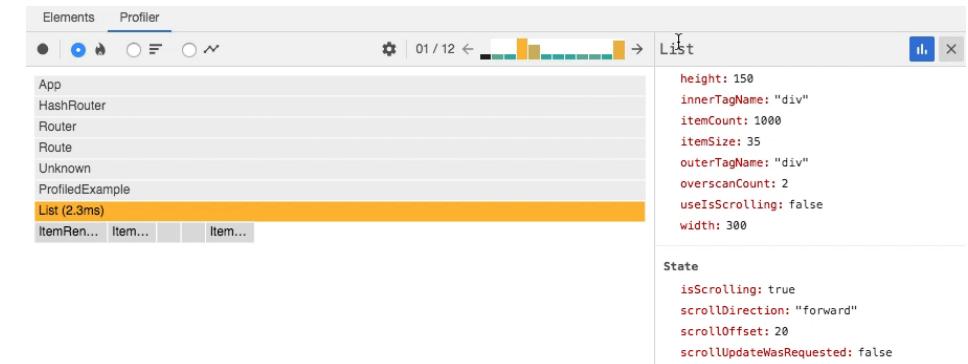
Mesurer les performances



Interface configurable

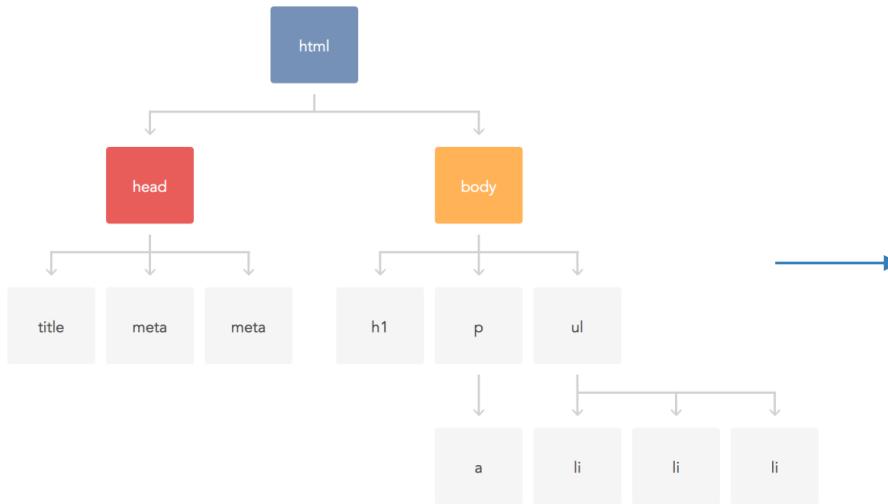


Consulter les valeurs des states



Naviguer entre les states successifs

Pour résumer



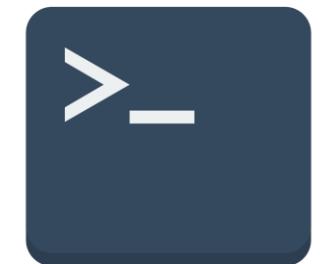
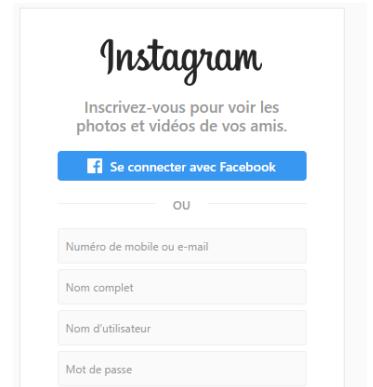
1. Votre code

```

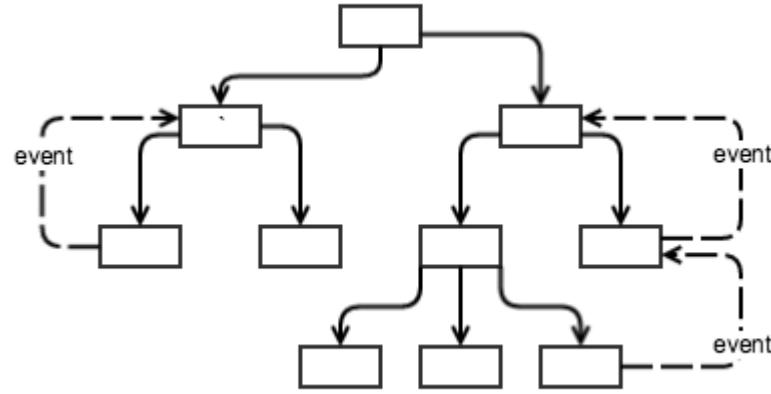
React.createElement(
  "html",
  null,
  React.createElement("head", ...),
  React.createElement("body", ...)
);
  
```

2. Converti en VirtualDOM

3. Puis transformé en HTML,
Swift ou même du bash



Questions ?



Redux

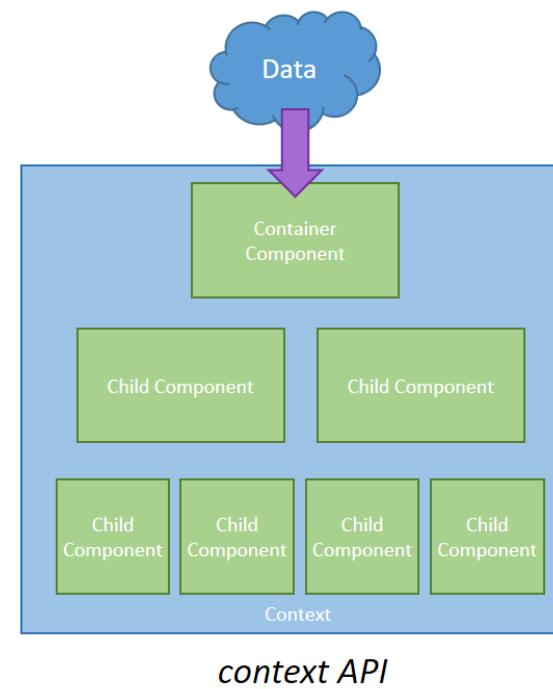
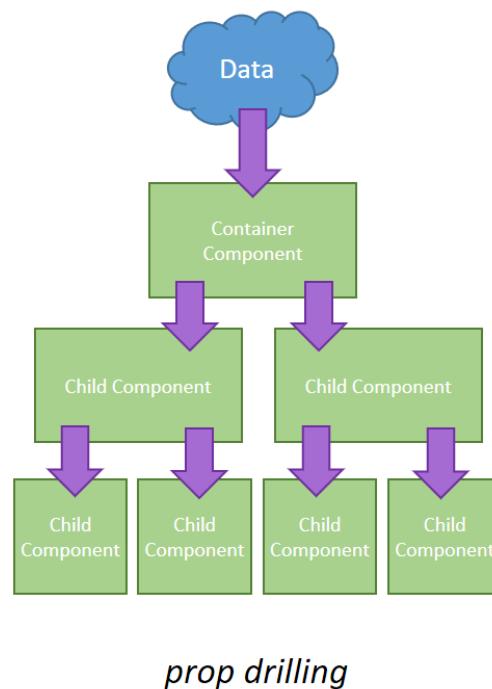
Animé par Mazen Gharbi

Sommaire

- ▷ Introduction
- ▷ Application State vs UI State
- ▷ Installation de l'environnement
- ▷ Unidirectional Data Flow
- ▷ Actions
- ▷ Reducers
- ▷ Store
- ▷ Context
- ▷ Créer un composant
- ▷ Connecter notre composant à Redux
- ▷ Dispatcher une action
- ▷ Middlewares

Context API

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



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

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

clients.screen.ts

```
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 (créeé par l'équipe Redux) qui harmonise les deux et permet ainsi d'écrire moins de code :

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

Redux - Installation de l'environnement

- ▷ Pour aider au debugging d'une application utilisant Redux, il y a un utilitaire qui log tous les changements d'état dans la console :

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

Unidirectional Data Flow

- ▷ L'architecture Flux repose sur l'idée d'un flux de données unidirectionnel strict. On ne peut affecter les données qui y transitent qu'en suivant un sens précis (on ne le court-circuite pas).
- ▷ Redux implémente cette architecture avec un vocabulaire qui est lui est propre mais le principe est bien le même.

Unidirectional Data Flow



Unidirectional Data Flow

- ▷ Les composants appellent les « actions creators » pour pouvoir modifier le store (Application State)
- ▷ Les « **actions creators** » contiennent la logique principale et dispatchent des actions (events) qui sont simplement des objets qui ont un type pour les différencier et une propriété

Actions

- ▷ Les actions sont créées à partir d'un action creator. C'est une fonction qui retourne un objet avec deux propriétés:
 - › type: type de l'action (pour pouvoir les différencier)
 - › payload: contient le contenu de l'action
- ▷ Les actions sont dispatchées au store

Actions

- ▷ Prenons par exemple un fichier orderActions.js dans le dossier src

```
// Les actions
export const ADD_ORDER = '[RESTAURANT] Add a new order';

// Nos action-creators
export function addOrderAction(food) {
    return {
        type: ADD_ORDER,
        payload: {
            order: {
                food,
                // Le + permet de caster la date en number (timestamp ici)
                orderAt: +new Date()
            }
        }
    }
}
```

Reducers

- ▷ Les **reducers** spécifient comment l'Application State doit changer en réponse aux actions dispatchées au store ;
- ▷ Leurs logiques doivent être **prédictibles**
 - › « *pour telle type action reçue avec tel payload et telle application state, je retourne systématiquement le même store* » ;
- ▷ En pratique, nous ne devons pas avoir une logique qui dépend du moment présent dans un reducer, sinon le nouveau store retourné ne sera pas le résultat voulu. On écrira plutôt cette logique dans un « *action creator* » ;
- ▷ Le store retourné doit toujours être un nouvel objet car Redux effectue une vérification === entre l'ancien et le nouvel state

Reducers

- ▷ Nous allons créer un fichier reducers.js dans le dossier src

```
import { ADD_ORDER } from './orderActions';
function ordersReducer(state = [], action) {
    switch (action.type) {
        // Ajouter à la liste des commandes
        case ADD_ORDER:
            // La fonction "concat" renvoie un nouveau tableau (donc nouvelle ref.)
            return state.concat([action.payload.order]);

        default:
            // Si type qu'on ne connaît pas, on retourne le state tel quel
            // D'ucoup, rien ne changera
            return state;
    }
}
```

Etat initial de l'application state

Reducers

- ▷ Nous allons maintenant faire combiner un fragment de notre Application State (nous l'appellerons ordersState) avec un reducer qui retournera un nouveau state à chaque fois qu'il traitera une action

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

const reducers = combineReducers({
    ordersState: ordersReducer // Reducer créé précédemment
    // On peut intégrer d'autres reducers ici
});

export default reducers;
```

Clé du reducer, on en aura besoin par la suite

- ▷ On peut avoir autant de reducers que l'on veut. Il faudra les combiner avec un fragment de notre Application State comme dans notre exemple.

Store

- ▷ Le store est finalement le **cerveau de notre application** qui contient l'Application State, les reducers et les méthodes qui permettent de dispatcher une action et de souscrire aux modifications de l'Application State
- ▷ La dernière étape sera de créer une instance du store et de la connecter aux composants de notre application.

Store

- ▷ Nous devons créer une instance de notre store, qui combine tous nos reducers

```
// index.js
import { createStore } from 'redux';
import { Provider } from 'react-redux';
import reducers from './reducers'; // resultat du combineReducers

const store = createStore(reducers);

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

Store

- ▷ Une instance d'un store Redux nous fournit trois méthodes essentielles à l'architecture que nous mettons en place:
 - › `dispatch` envoie les actions aux reducers
 - › `getState` retourne l'état de l'Application State
 - › `subscribe` notifie du changement d'état
- ▷ « <Provider> » et « connect » permettent l'accès à ces méthodes
- ▷ **Provider** définit ce que React appelle un contexte (Context)
 - › Tout comme BrowserRouter

Créer un composant

- ▷ Nous allons commencer par créer un composant qui affichera les commandes reçues pour notre restaurant **SANS LE STORE** pour l'instant

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

export default function OrdersList(props) {
    return (
        <ul>
            {props.orders.map((order) => (
                <li>{order.food}</li>
            )))
        </ul>
    );
}

OrdersList.propTypes = {
    orders: PropTypes.arrayOf(
        PropTypes.shape({
            orderAt: PropTypes.number,
            food: PropTypes.string
        })
    )
};
```

Créer un composant

- ▷ En ouvrant la console, vous aurez ce message:

```
Warning: Each child in an array or iterator should have a  
unique "key" prop.
```

- ▷ React a besoin d'une clé unique pour chaque élément venant d'un tableau (ici c'est props.orders) pour éviter de redessiner tous les éléments

```
<ul>  
  {props.orders.map((order) => (  
    <li key={order.orderAt}>{order.food}</li>  
  ))}  
</ul>
```

Créer un composant

- ▷ Nous allons créer une liste de commandes et les donner à App

```
// index.js
...
const initialOrders = [
    { orderAt: 1565965460, food: 'Pissaladière' },
    { orderAt: 1565965522, food: 'Salade niçoise' },
    { orderAt: 1565965839, food: 'Socca du chef' }
];

ReactDOM.render(
    <Provider store={store}>
        <App orders={initialOrders} />
    </Provider>,
    document.getElementById('root')
);
```

Créer un composant

- ▷ Puis dans le composant « App »

```
import React, { Component } from 'redux';
import OrdersList from './orders-list';

function App {
  return (
    <div>
      <OrdersList orders={this.props.orders} />
    </div>
  );
}

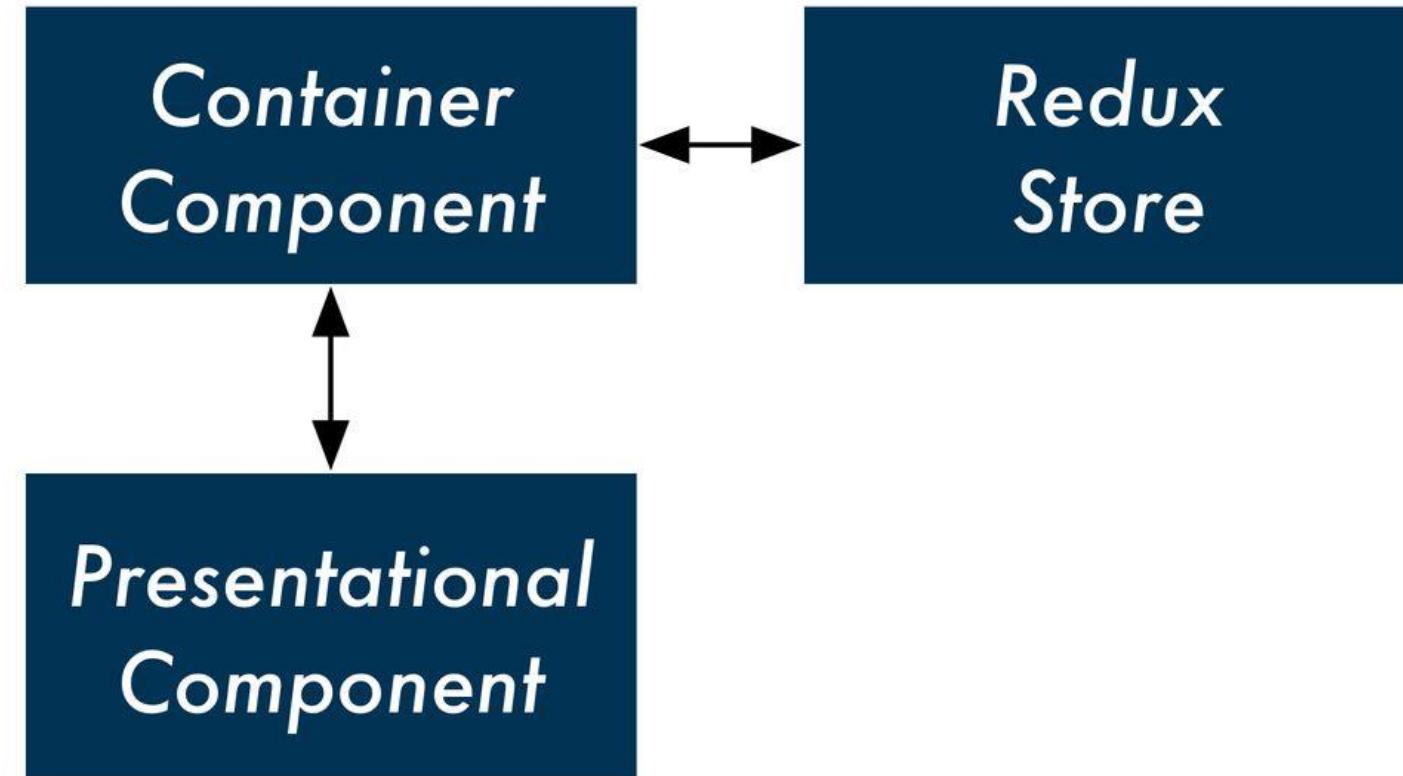
export default App;
```

- ▷ Super, ça fonctionne ! Mais nous n'utilisons pas REDUX du tout...

Connecter notre composant à Redux

- ▷ Il est temps de récupérer la liste des commandes directement au travers du store !
- ▷ Seuls les « élus » peuvent être attachés au store
 - › On les appellent les « Containers »
- ▷ On va encapsuler « <OrdersList /> » par un autre composant « <ConnectOrderList /> » qui lui sera connecté à Redux
 - › ConnectedOrdersList est ce qu'on appelle un « *Container component* »
 - › OrdersList est un « *Presentational component* »

Containers



Création du container

```
// connect-orders-list.js
import { connect } from 'react-redux';
import OrdersList from './orders-list';

// Cette fonction permettra de faire correspondre
// des reducers ou des variables des reducers au component
// auquel le container est relié
function mapStateToProps(store) {
  return {
    orders: store.ordersState
  }
}

const containerOrdersList = connect(
  mapStateToProps
)(OrdersList);

export default containerOrdersList;
```

Connecter notre composant à Redux

- ▷ Nous avons besoin maintenant de remplacer « <OrdersList /> » par « <ConnectOrdersList /> » dans App.js ;
- ▷ C'est l'occasion de simplifier notre composant App en le rendant fonctionnel ;
- ▷ Note: la propriété orders qui est transmise à App dans index.js n'est plus utile maintenant

```
import React from 'react';
import ConnectOrdersList from './connect-orders-list';

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

Connecter notre composant à Redux

- ▷ L'état initial de notre Application State est vide. Il ne nous reste plus qu'à le spécifier

```
const initialOrders = [
    { orderAt: 1565965460, food: 'Pissaladière' },
    { orderAt: 1565965522, food: 'Salade niçoise' },
    { orderAt: 1565965839, food: 'Socca du chef' }
];

const initialAppState = {
    ordersState: initialOrders
};

const store = createStore(reducers, initialAppState);
```

Dispatcher une action

- ▷ Notre liste de commandes n'est pas encore modifiable via notre interface utilisateur ;
- ▷ Nous allons créer un dernier composant qui se chargera de dispatcher une action **ADD_ORDER** avec le nom du menu commandé

Dispatcher une action

```
import React from 'react';
import { connect } from 'react-redux';
import { addOrderAction } from './orderActions';

function OrderComposer {
    return (
        <div>
            <input type='text' />
            <button>Nouvelle commande</button>
        </div>
    );
}

export default connect()(OrderComposer);
```

Dispatcher une action

- ▷ Nous avons besoin d'un état local qui enregistre la valeur de notre champ de texte.

```
// order-composer.js
...
function OrderComposer {
  const [order, setOrder] = useState('');

  return (
    <div>
      <input type='text' value={order}/>
      <button>Nouvelle commande</button>
    </div>
  );
}
```

Dispatcher une action

- ▷ Nous ajoutons maintenant deux méthodes qui mettent à jour notre état local et dispatch l'action d'ajout de commande

```
// order-composer.js
...
function OrderComposer {
    ...
    const updateOrder = (e) => {
        const orderValue = e.target.value;

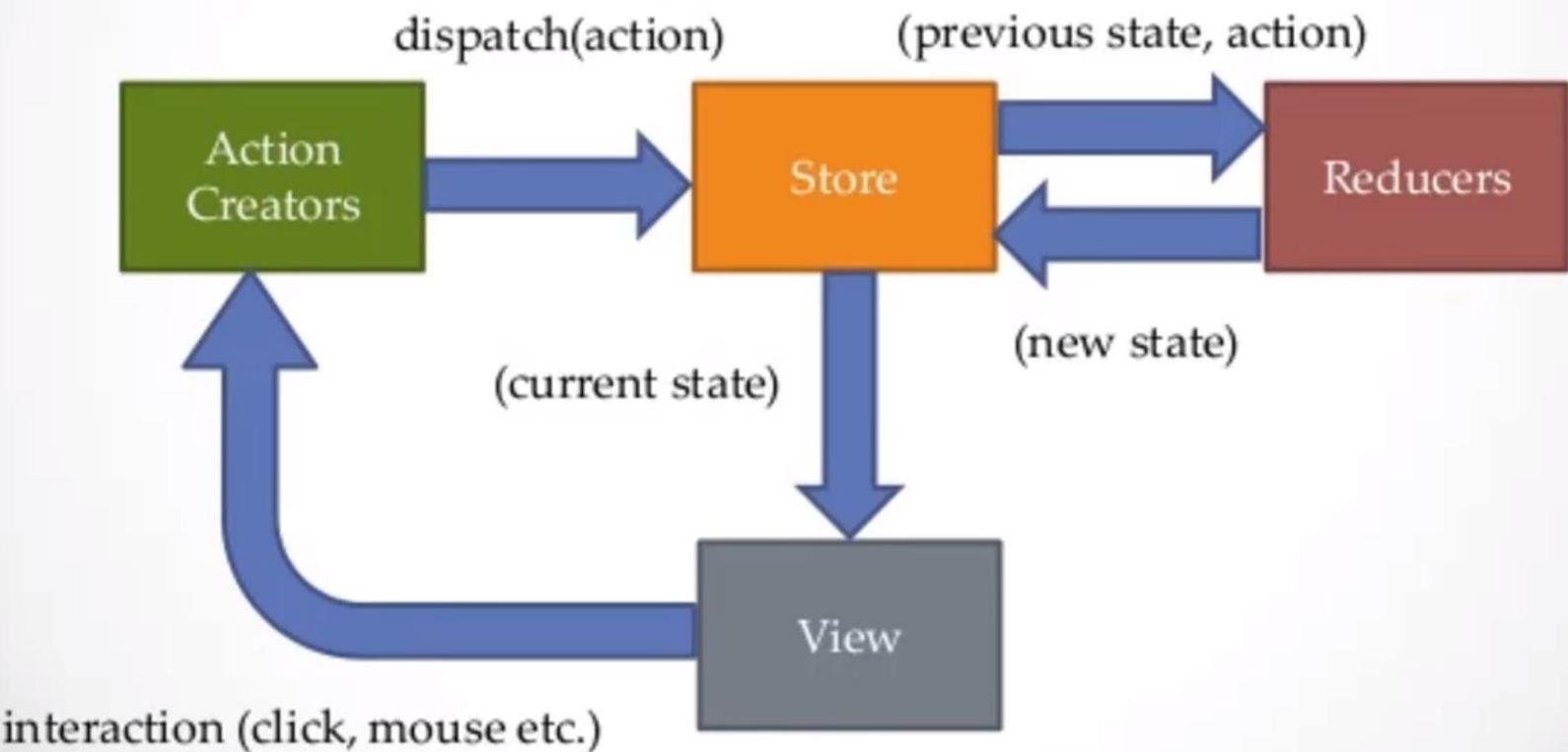
        setOrder(orderValue);
    }
    // La fonction dispatch est fournie par "connect"
    const addOrder = () => {
        if(order) {
            props.dispatch(
                addOrderAction(order)
            );
        }
    }
}
```

Dispatcher une action

- ▷ Et nous finissons par affecter les listeners des events DOM.

```
...
return (
  <div>
    <input
      type='text'
      value={order}
      onChange={updateOrder}
    />
    <button onClick={addOrder}>
      Nouvelle commande
    </button>
  </div>
);
...
```

Redux Data Flow



Un exemple moins élaboré

[Essayer ce code !](#)

Redux

```
const reducer = function(state, action) {  
  if(action.type === 'ADD') {  
    // On renvoie le nouveau state généré  
    return state + action.payload;  
  }  
  
  return state;  
};  
  
// Le deuxième paramètre correspond au state initial  
const store = createStore(reducer, 0);  
  
// On s'abonne au store pour être informé à chaque modification de celui-ci  
store.subscribe(() => {  
  console.log("Le store a été modifié :" + store.getState());  
});  
  
// On envoie un événement au reducer  
store.dispatch({type: 'ADD', payload: 1}); → Les actions sont émises  
// => Le store a été modifié :1  
store.dispatch({type: 'ADD', payload: 1});  
// => Le store a été modifié :2
```

C'est ici que notre store est modifié !

Un autre exemple

- ▶ Plus complexe cette fois. Nous allons tenter de mettre en place l'architecture Redux dans un contexte où un Container fournit les données à un « Presentational component ». Voici un visuel :



Un autre exemple (1)

▷ Une première étape importante va être de créer le store à partir des reducers. Puis d'entourer notre container principal avec un provider auquel on passe notre Store :

```
let reducers = combineReducers({  
    counter,  
    // On peut en mettre d'autres ici  
});  
  
let store = createStore(reducers);  
  
render(  
    <Provider store={store}>  
        <MainContainer /> ← On appelle le container qui se chargera d'appeler et d'afficher le composant  
    </Provider>,  
    document.getElementById('root')  
);
```

Un autre exemple (2)

[Essayer ce code !](#)

Redux

- ▷ Evidemment, on a besoin de définir les actions pour communiquer avec nos Reducers

```
export function increment() {  
  return {  
    type: 'ADD',  
    value: 1  
  };  
}  
  
export function decrement() {  
  return {  
    type: 'REMOVE',  
    value: 1  
  };  
}
```

actions/counter.actions.js

Un autre exemple (3)

- ▷ Le reducer ici permet de recevoir des actions et de modifier le store en conséquence

```
let initialState = {  
  counter: 0  
};  
  
let counterReducer = function (state = initialState, action) {  
  switch (action.type) {  
    case 'ADD':  
      return {  
        ...state,  
        counter: state.counter + action.value  
      };  
    case 'REMOVE':  
      return {  
        ...state,  
        counter: state.counter - action.value  
      };  
    default:  
      return state;  
  }  
};
```

reducers/counter.reducer.js

On retourne un objet avec une nouvelle référence pour forcer le rechargement

Un autre exemple (4)

[Essayer ce code !](#)

Redux

- ▷ Une fois le reducer et les actions créés, il va falloir mettre en place le container qui liera les informations du store au component

```
import * as CounterActions from '../actions/counter.actions'

// Données du store à envoyer au composant sous forme de props
let propsMapping = (state) => ({
  counter: state.counter.counter
});

// Fonctions que l'on souhaite mettre à disposition pour notre composant
let dispatchMapping = dispatch => ({
  actions: bindActionCreators(CounterActions, dispatch)
});

let MainContainer = connect(
  propsMapping,
  dispatchMapping
)(MainApp)

export default MainContainer;
```

containers/Main.container.js

L'arrow function renvoie directement un objet ici



Ici on relie les données du store et les actions au composant MainApp

Un autre exemple (5)

[Essayer ce code !](#)

Redux

▷ Et enfin, le chapitre final : Le composant !

```
export default function MainApp(props) {  
  return (  
    <div>  
      <p>Valeur actuelle : {props.counter}</p>  
      <button onClick={props.actions.increment}>Add</button>&ampnbsp;  
      <button onClick={props.actions.decrement}>Remove</button>  
    </div>  
  )  
}
```

components/MainApp.js

Les informations fournies par le container sont disponibles directement via les props

Pour les actions, il faut ajouter aller chercher dans le sous-objet « action »

Middlewares

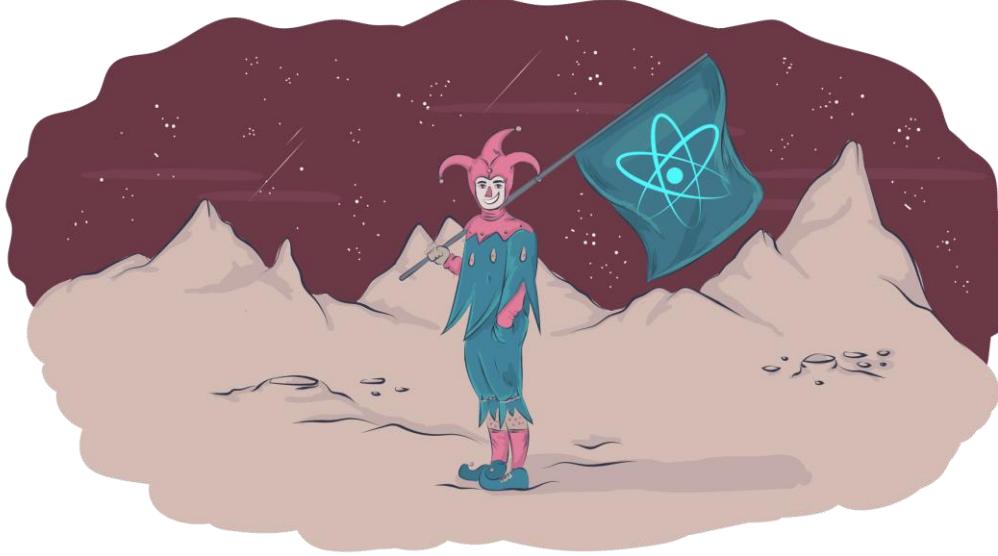
- ▷ Dans le contexte Redux, il est difficile d'avoir un hook dans le flux de données que nous venons de découvrir. Heureusement que les middlewares existent!
- ▷ Ils règlent tout simplement ce problème en ayant accès aux actions avant qu'elles soient traitées par les reducers mais aussi après qu'un nouveau store soit généré.

Middlewares

- ▷ « **redux-logger** » est un middleware. Pour nous aider lors du debugging, on va l'ajouter à notre store.

```
// index.js
...
import { applyMiddleware, createStore } from 'redux';
import { Provider } from 'react-redux';
import reducers from './reducers';
import logger from 'redux-logger';
...
const store = createStore({
  reducers,
  initialAppState,
  applyMiddleware(logger)
})
```

Questions ?



Tests unitaires

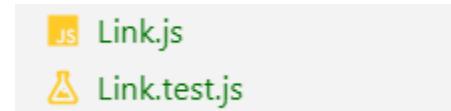
Animé par Mazen Gharbi

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';

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

App.test.js

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](#)

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

Intitulé du test

- ▷ **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

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

Test.spec.js

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);
  }
}

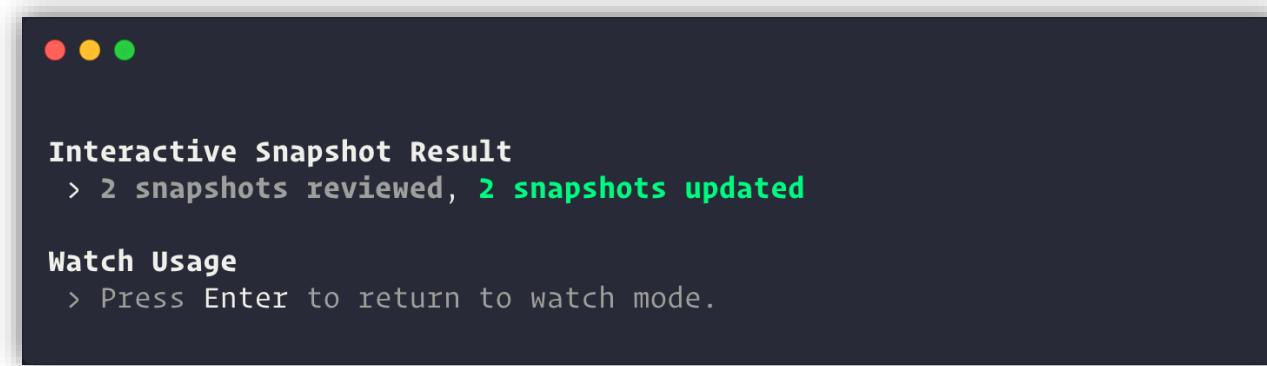
jest.mock('axios'); // SURCHARGE !                                user.service.test.js

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



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

```
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

```
_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>  
  );  
}
```

Link.js (suite)

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();

  // 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

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

```
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 `;
```

Link.test.js.snap

Mettions à jour notre test

- ▷ On fait le choix de modifier l'url :

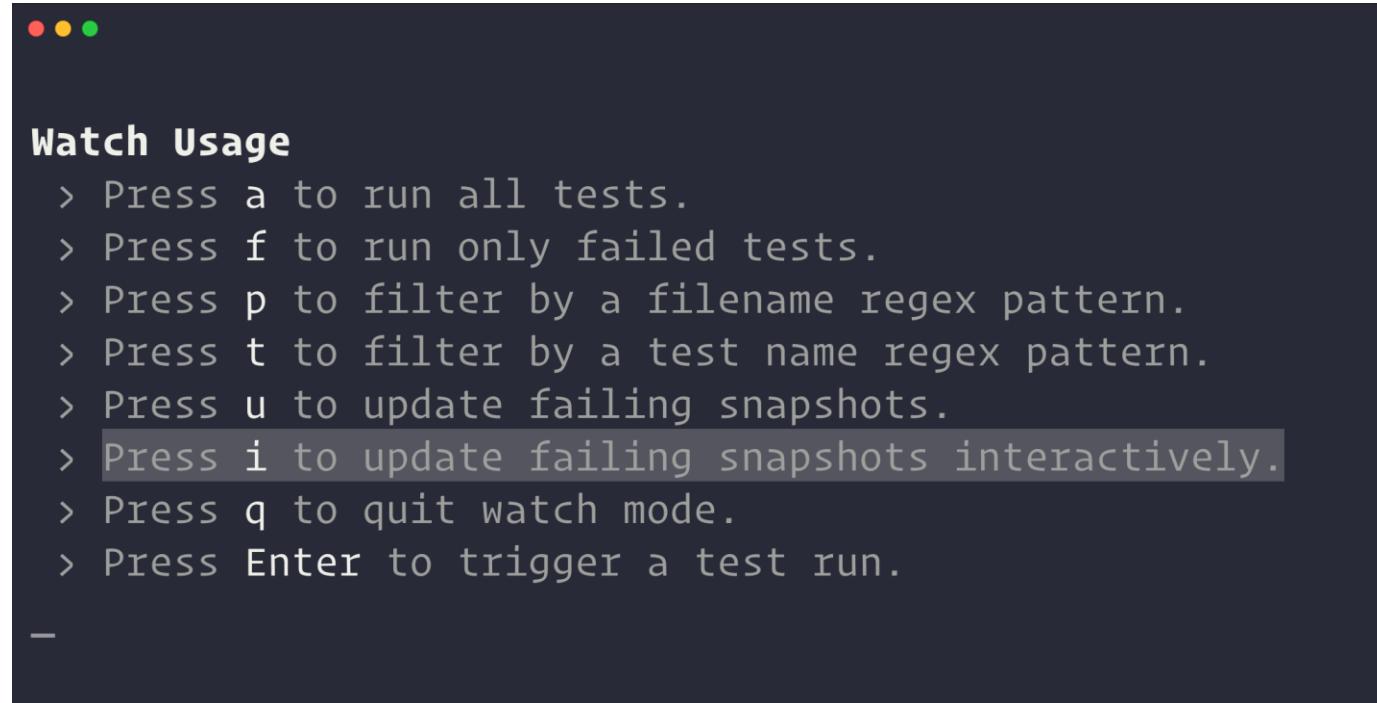
```
const component = renderer.create(  
  <Link page="http://www.macademia.fr">Macademia</Link>,  
);
```

- ▷ 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* »



Tester vos composants avec Enzyme

- ▷ Les snapshots sont bien pratiques, mais ils ne permettent pas de tester la logique de nos composants

```
↳ npm i --save-dev enzyme enzyme-adapter-react-16
```

- ▷ Nous allons donc utiliser Enzyme pour simuler la création d'un component



Enzyme

- ▷ Enzyme fournit une fonction « `mount` » qui nous permettra de simuler l'affichage d'un composant

```
import { configure, mount } from 'enzyme';
import Adapter from 'enzyme-adapter-react-16';

configure({ adapter: new Adapter() });

it('Devrait faire l\'affichage qu\'on lui demande', () => {
  const wrapper = mount(
    <Link>Mon texte</Link>
  );
  const p = wrapper.find('a');
  expect(p.text()).toBe('Mon texte');
});
```

Permet d'adapter la librairie à notre version de React

Simuler un événement

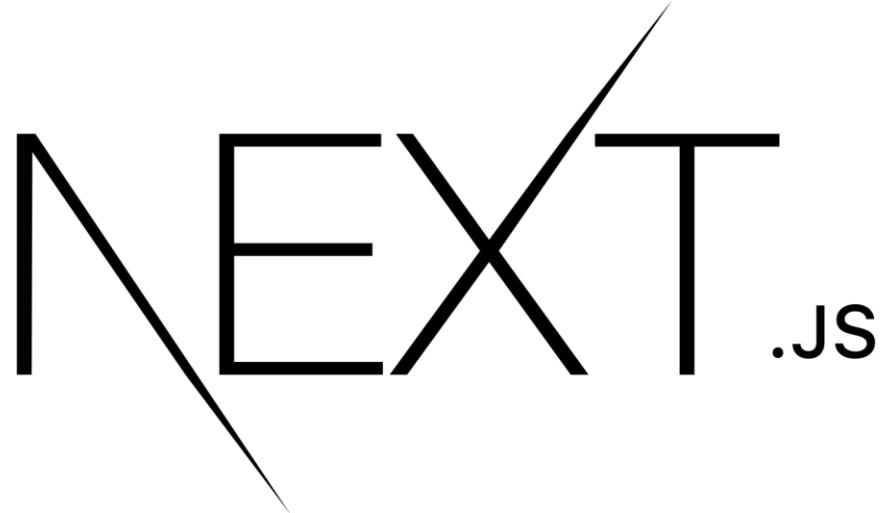
- ▷ Il est possible de simuler un click très simplement avec Jest

```
it('Simuler le click', () => {
  const wrapper = mount(
    <Link>Mon texte</Link>
  );
  const p = wrapper.find('a');
  p.simulate('click');
});
```

TP

- ▷ 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 **differents snapshots**

Questions ?



NE~~X~~T.JS

NextJS

Animé par Mazen Gharbi

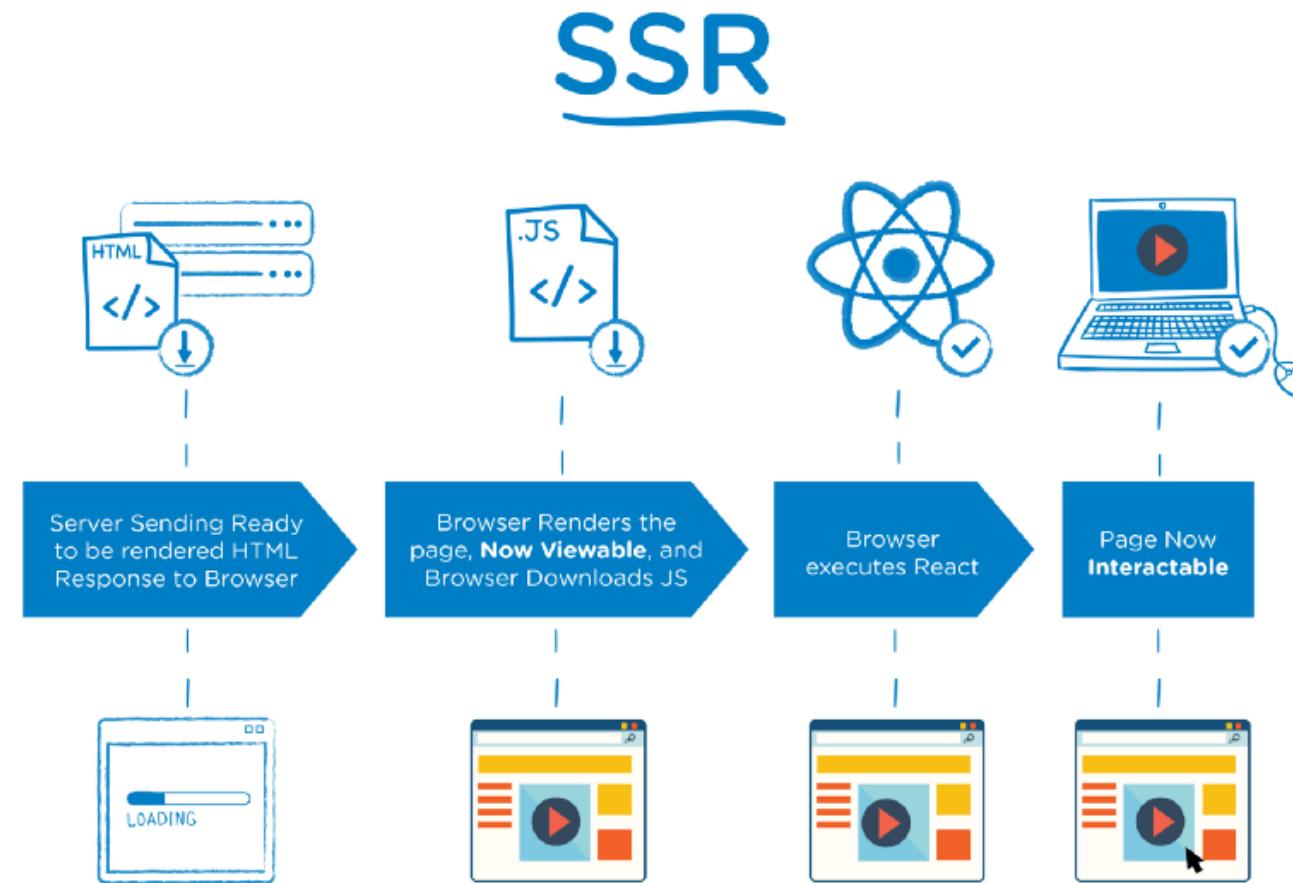
Qu'est-ce donc

- ▷ Outil développé par [Zeit](#)
- ▷ Framework permettant d'effectuer le rendu des applications web React par les serveurs
 - › Server Side Rendering
- ▷ Bâti sur [React](#), [Webpack](#) et [Babel](#)
- ▷ Objectif : Concevoir les applications [client-serveur](#)

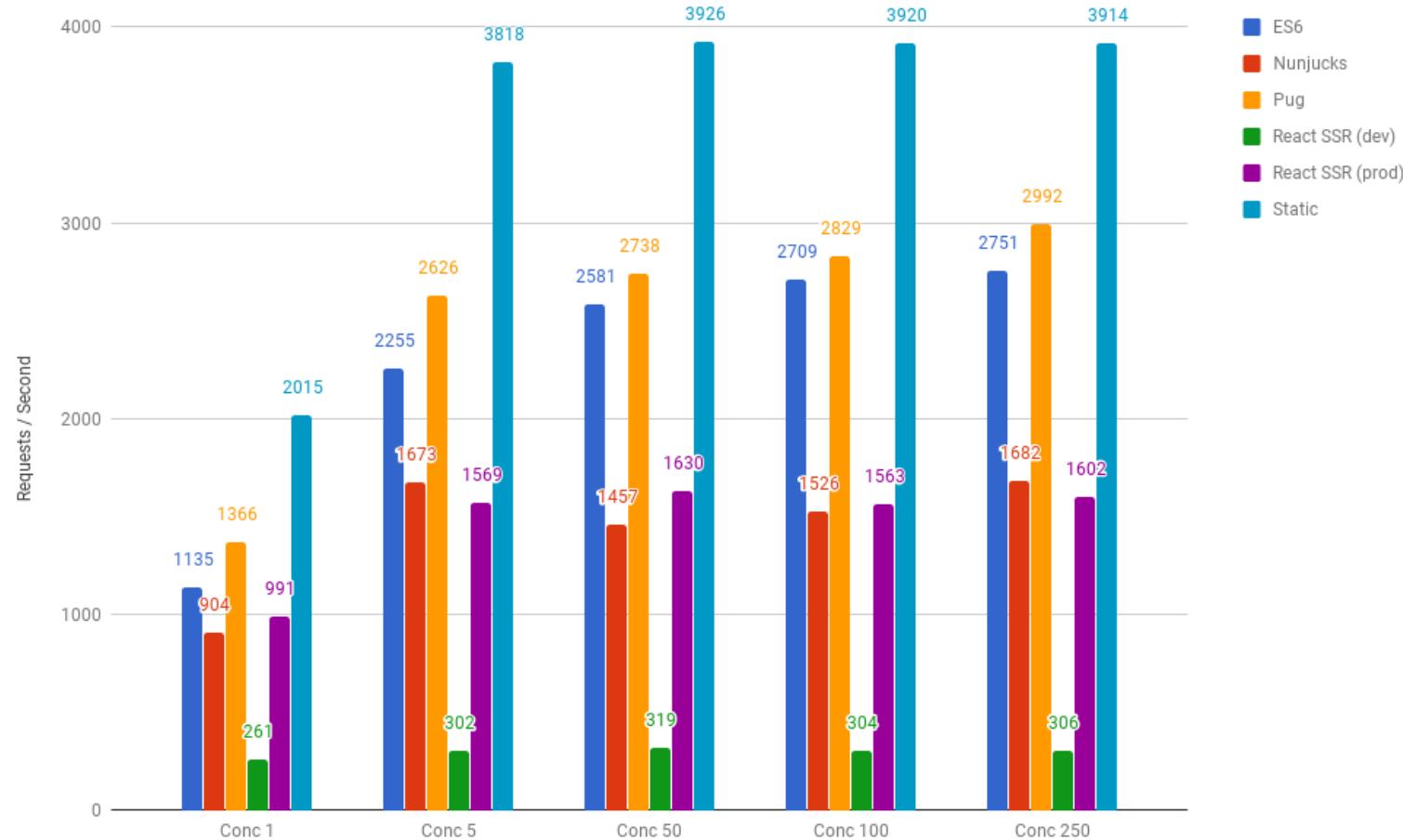
Server Side Rendering

- ▷ Une application Angular, React ou Vue doit préparer son contexte avant d'afficher la page
 - › Bootstrap time
- ▷ On va chercher à optimiser ce temps de chargement initial :
 - › Pour optimiser le SEO 
 - › Améliorer l'expérience utilisateur !
- ▷ 2 choses possibles avec le SSR !

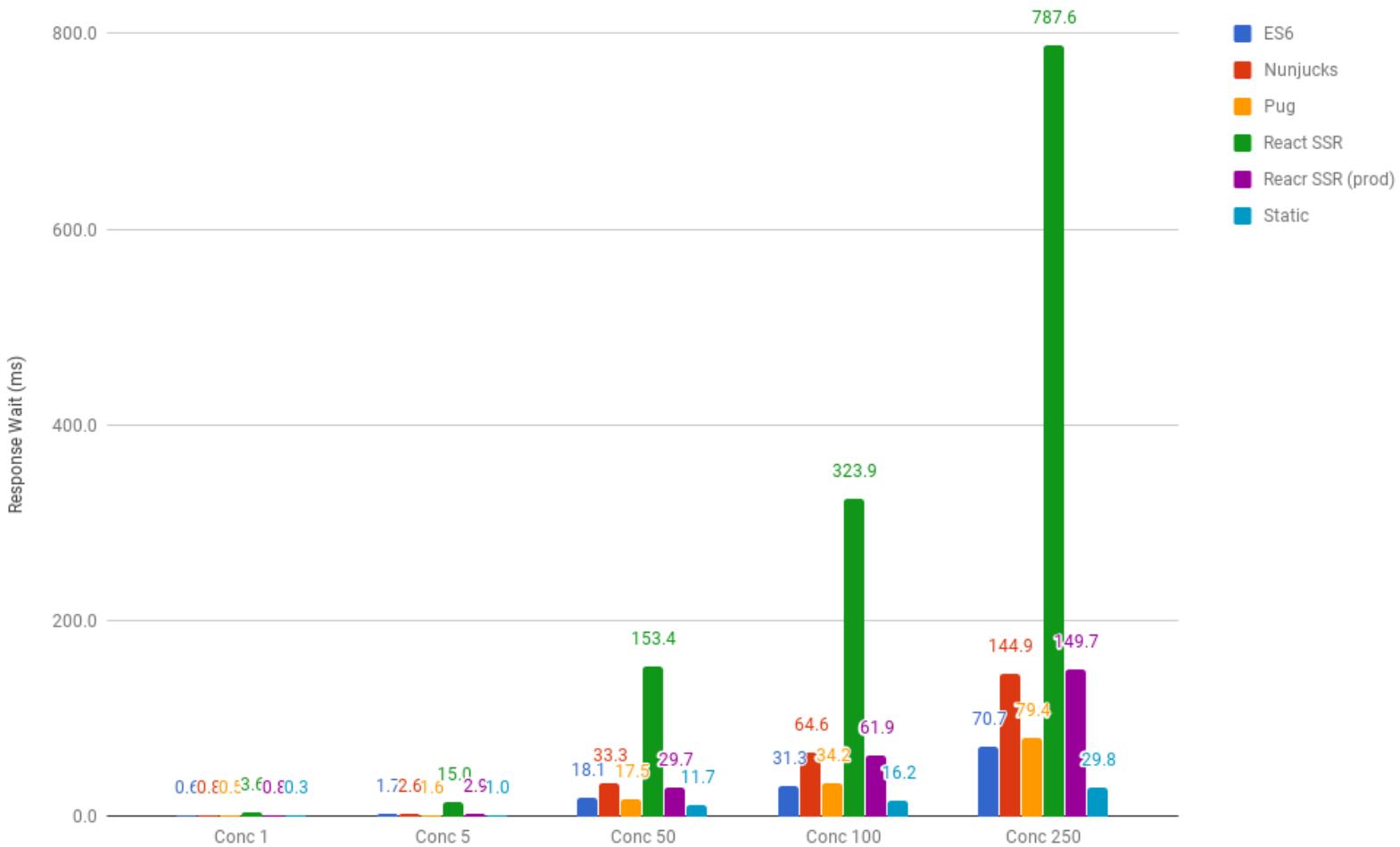
Server Side Rendering



Affichage bien plus rapide



Contrepartie



Mise en place

- ▷ Next est un framework, il importe donc une configuration bien spécifique

```
> npx create-next-app
```

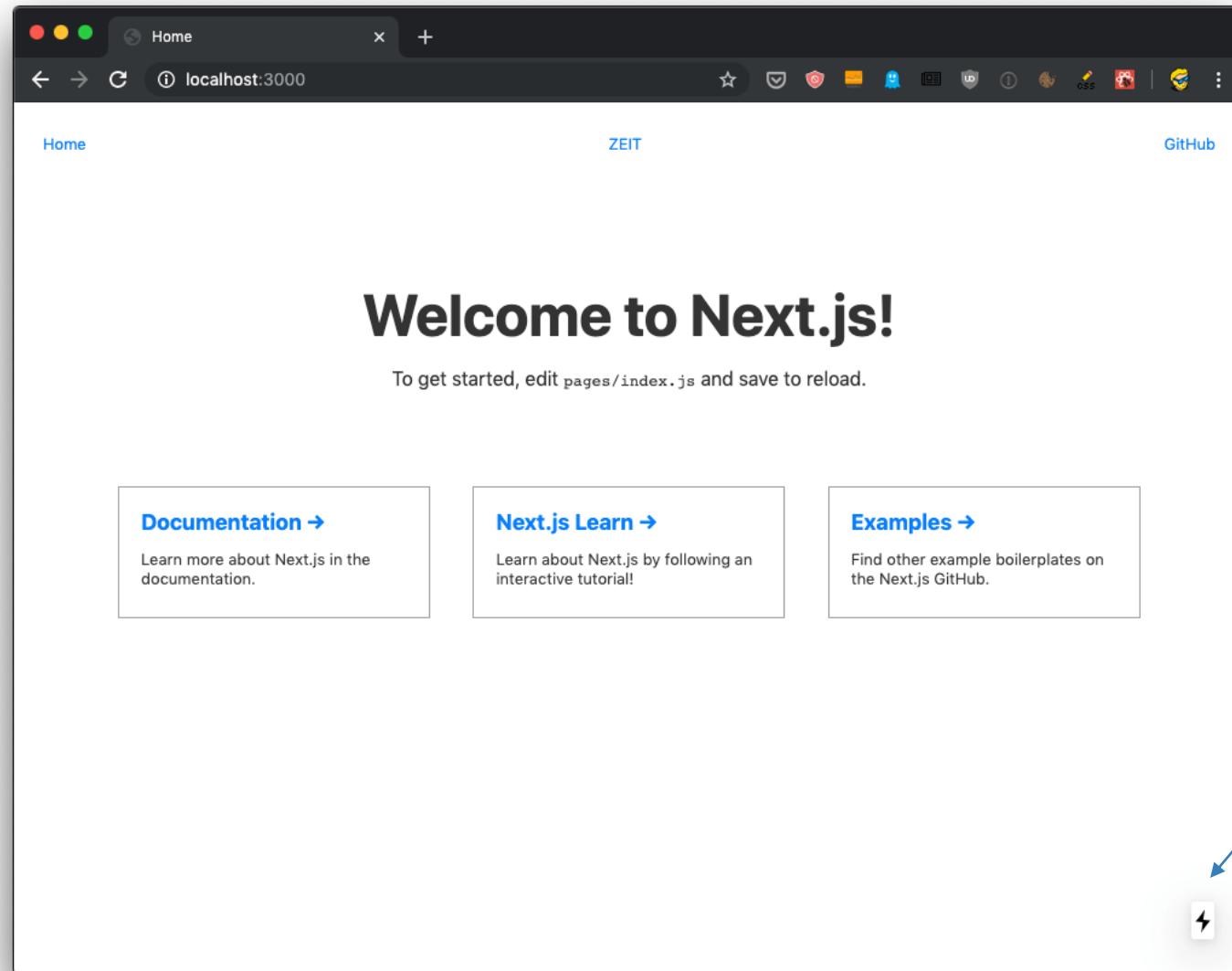
```
npx: installed 1 in 4.581s  
? What is your project named? » my-app
```



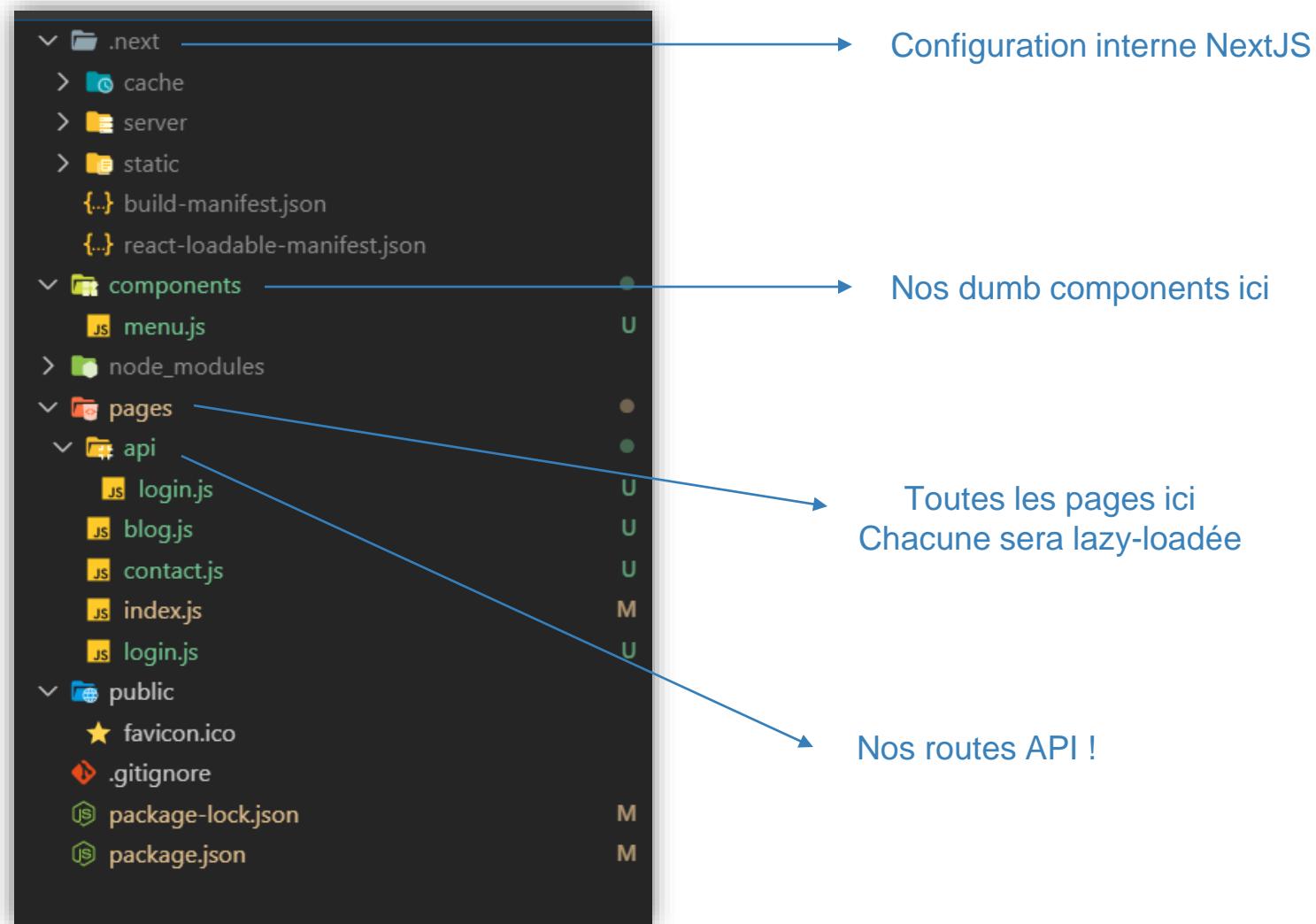
```
> npm run dev
```

- ▷ On entre le nom de notre application et le projet est créé

Résultat

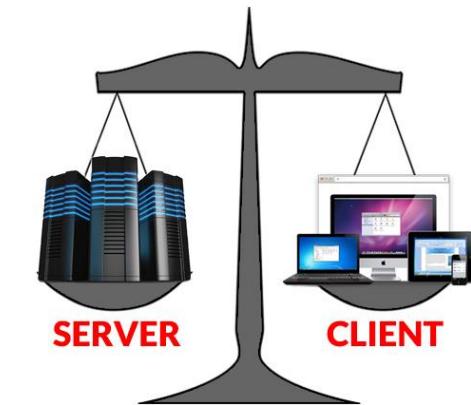
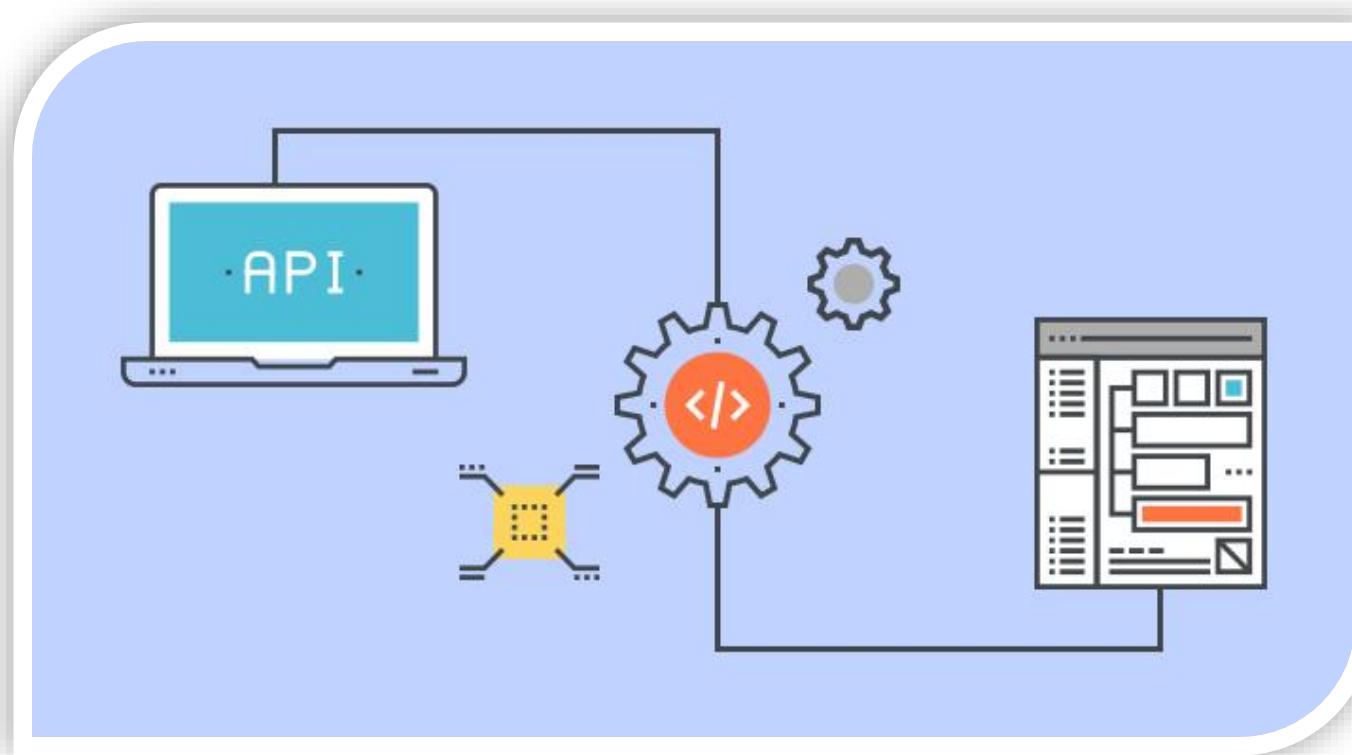


Architecture



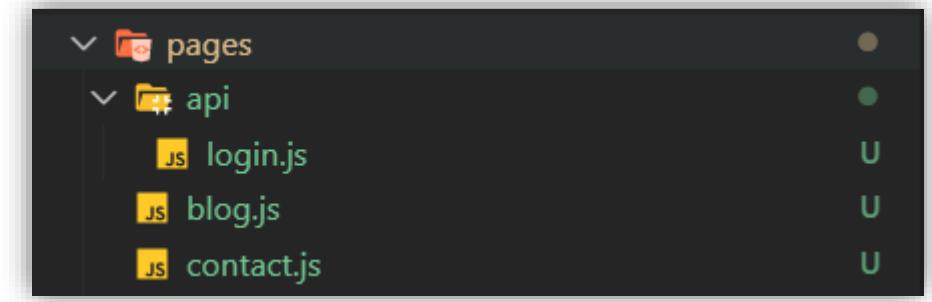
Front & Back

- ▷ Next nous permet de gérer des routes API avec lesquelles notre front pourra communiquer

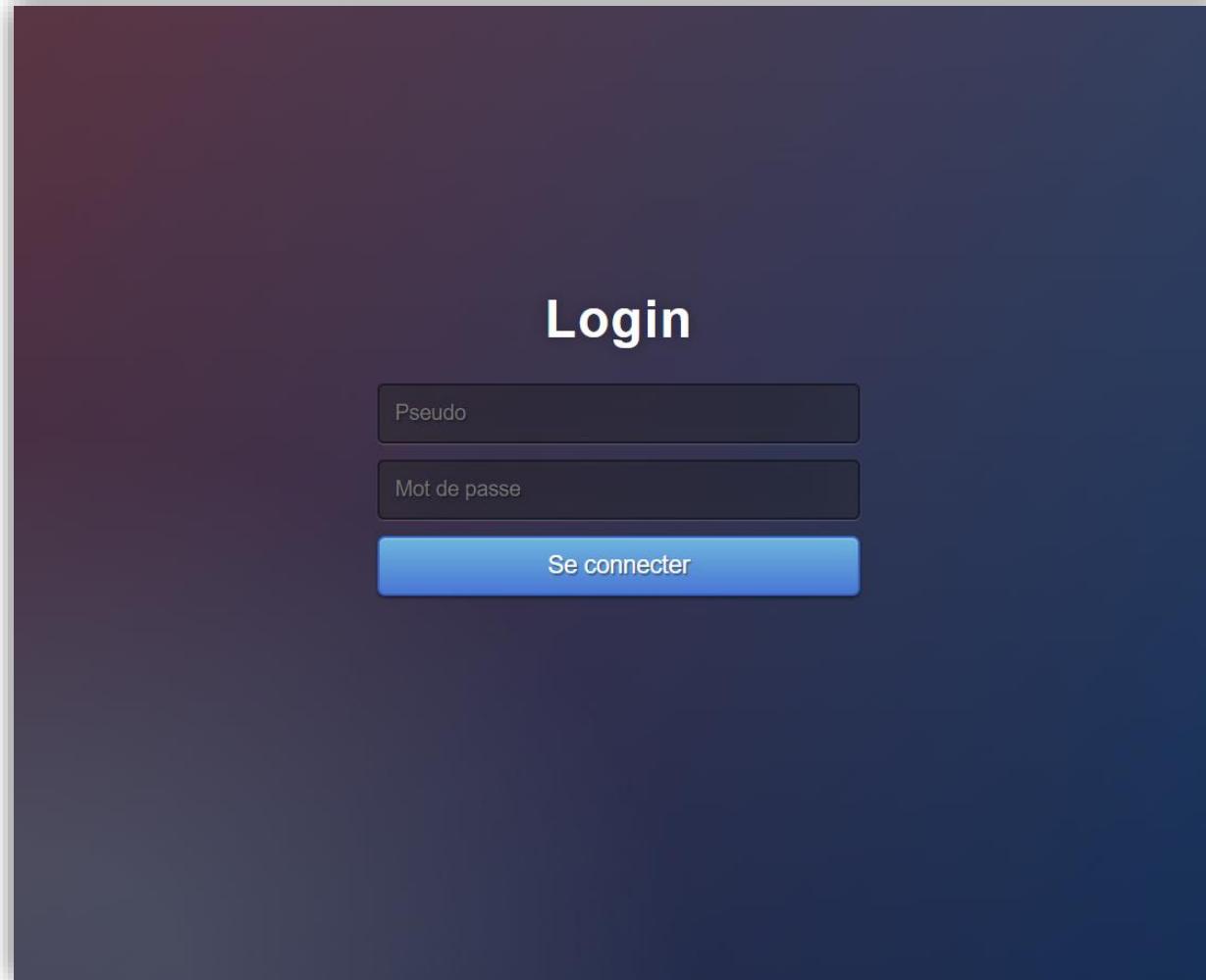


API

- ▷ On va mettre en place une route côté back-end qui permettra à notre application React de nous logger ;
- ▷ NextJS fournit une **solution simple** pour construire notre API
 - › Il faut commencer par créer un dossier **api/** dans le dossier **pages/**
- ▷ Chaque fichier dans **./pages/api** est mappé sur **/api/***. Par exemple, **./pages/api/login.js** est mappé sur la route **/api/login**.



Mise en place de la page de login



Mise en place de la page de login

```
let [form, setForm] = useState({ username: '', password: '' });
let [loading, setLoading] = useState(false);
...

<form method="post">
  <input
    type="text" value={form.username}
    onChange={(ev) => setForm({ ...form, username: ev.target.value })}
    name="username"
    placeholder="Pseudo"
    required="required" />
  <input
    type="password" value={form.password}
    onChange={(ev) => setForm({ ...form, password: ev.target.value })}
    name="password"
    placeholder="Mot de passe"
    required="required" />
  {
    loading ||
    <button type="submit" onClick={onLogin} disabled={!form.username || !form.password}>
      Se connecter
    </button>
  }
</form>
```

pages/login.js

Requête au serveur

```
function onLogin() {
  setLoading(true);

  axios.post('/api/login', { username: form.username, password: form.password })
    .then((res) => {
      Router.push('/');
    })
    .catch((err) => {
      console.error(err);
    })
    .finally(() => setLoading(false));
}
```

pages/login.js

```
function Login(req, res) {
  let { username, password } = req.body;
  if (username === 'admin' && password === 'password') {
    res.status(200).json({ log: true });
  } else {
    res.status(401).json({ error: 'Mauvais identifiants' });
  }
}
```

pages/api/login.js

Routing avec NextJS

- ▷ Contrairement au React natif, il n'y a aucun fichier permettant de déclarer les routes de notre application
- ▷ Chacune des routes est **représentée par un fichier**
- ▷ Pour une meilleure maintenabilité & performance, chaque route est **lazy-loadé**, donc le chargement de la page n'est fait qu'au moment de sa requête

Naviguer entre nos routes

- ▷ Comme pour le routing basique, un simple Link :

```
<Link href="/login">  
    <a>Login</a>  
</Link> -
```

- ▷ Mais l'import est différent :

```
import Link from "next/link";
```

- ▷ On peut également changer de route à partir du modèle :

```
import Router from 'next/router';  
Router.push('/');
```

Gestion des paramètres de route

- ▷ Ici malheureusement, c'est bien moins simple
- ▷ Il va être nécessaire de configurer le serveur ExpressJS pour rediriger l'url avec un ID vers cette même route avec un query

```
server.get('/blog/:id', (req, res) => {  
    return app.render(req, res, '/blog', { id: req.params.id })  
})  
/server.js
```

- ▷ Et au moment de l'envoit :

/pages/login.js

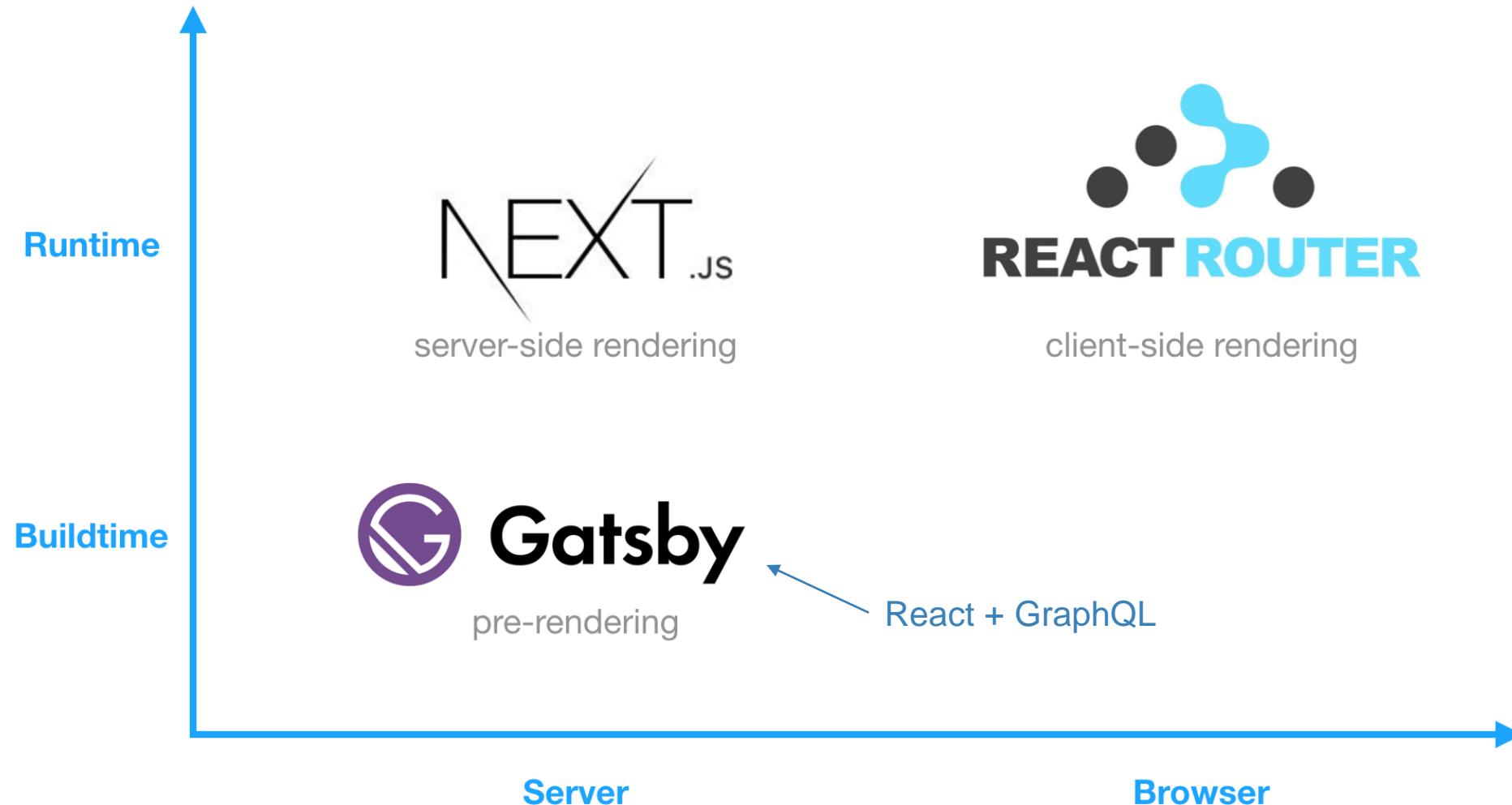
```
<input type="number" placeholder="Id du blog" value={id} onChange={(ev) => setId(ev.target.value)} />  
<Link href={"/blog?id=" + id} as={"/blog/" + id}>  
    <button>On y va</button>  
</Link>  
Donnée envoyé en tant que paramètre GET  
L'alias permet d'afficher l'URL sans le param GET
```

Pré-chargement des pages

- ▷ Lorsque vous appliquer la balise `<Link>` dans votre page, NextJS préchargera automatiquement le contenu de cette page en background
 - › Ce qui signifie qu'il téléchargera cette page pendant que vous naviguez
- ▷ Néanmoins, il existe certaines pages où vous savez pertinemment qu'elles ne seront que très rarement visitées, il est possible de désactiver le « prefetch » pour celles-ci :

```
<Link href="/ma-page" prefetch={false}>
  <a>Ma page</a>
</Link>
```

Pour finir



TP

- ▷ Créez une application NextJS avec 2 pages différentes :
 - › Page **Login** ;
 - › Page **Home**.
- ▷ Pour la connexion, créez une route API « **/login** » côté backend
- ▷ Créez également une route API « **/movies** » permettant de récupérer un tableau de film
- ▷ Affichez ce tableau de films dans la page **Home**

Questions ?



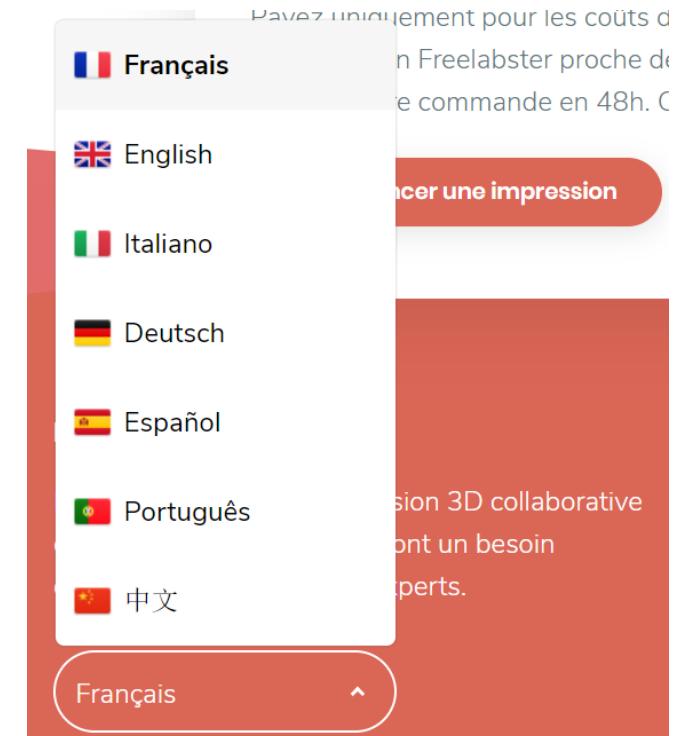
Internationalization

Animé par Mazen Gharbi

Principe

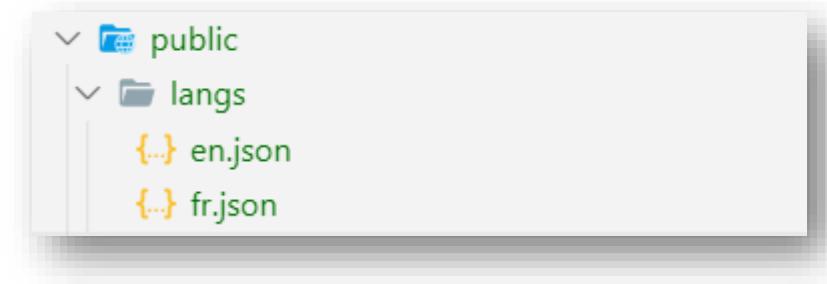
- ▷ Pour optimiser l'expérience utilisateur, il est souvent nécessaire de proposer différentes langues pour votre application WEB
- ▷ Il va être nécessaire de faire cela de façon fluide
- ▷ La communauté React propose plusieurs outils

```
npm install react-i18next@legacy i18next --save
```



Fonctionnement

- ▷ Nos traductions vont être stockées dans des fichiers .json différents
 - › Chacun représentant une langue différente



- ▷ On pourra importer ces fichiers par la suite

Fichiers de traduction

- ▷ Voici ce que contient nos .json :

```
{  
    "home.title": "ReactJS",  
    "home.description": "Bienvenue dans la meilleure formation du siècle"  
}
```

fr.json

```
{  
    "home.title": "ReactJS",  
    "home.description": "Welcome to the best course in the world !"  
}
```

en.json

Configuration

- ▷ La prochaine étape va consister à configurer notre multi-langue en implémentant le plugin

```
import i18n from "i18next";
import { reactI18nextModule } from "react-i18next"; config/i18n.js

import langFR from '../../public/langs/fr.json';
import langEN from '../../public/langs/en.json';

const resources = {
  fr: {
    translation: langFR
  },
  en: {
    translation: langEN
  }
};
```

Configuration

```
i18n
  .use(reactI18nextModule)
  .init({
    resources,
    lng: "fr",
    interpolation: {
      escapeValue: false // On échappe rien, React protège déjà contre le XSS
    }
  });
  export default i18n;
```

config/i18n.js (suite)

```
import './config/i18n';

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

index.js

Et enfin

```
import React from 'react';
import './App.css';
import { withNamespaces } from 'react-i18next';

function App(props) {
  const { t } = props;

  return (
    <div className='App'>
      <h1>{t('home.title')}</h1>

      <p>{t('home.description')}</p>
    </div>
  );
}

export default withNamespaces()(App);
```

Changement dynamique

- ▷ L'utilisateur peut évidemment changer la langue comme bon lui semble

Français Anglais

ReactJS

Welcome to the best course in the world !

```
<div>
  <button onClick={() => changeLanguage('fr')}>Français</button>
  <button onClick={() => changeLanguage('en')}>Anglais</button>
</div>
```

Français Anglais

ReactJS

Bienvenue dans la meilleure formation du siècle

```
const changeLanguage = (lng) => {
  i18n.changeLanguage(lng);
}
```

Gestion de catégories

- ▷ On peut définir des groupes de traduction :

```
{  
  "home": {  
    "title": "ReactJS",  
    "description": "Bienvenue dans la meilleure formation du siècle"  
  }  
}
```



```
<h1>{t('home.title')}</h1>  
  
<p>{t('home.description')}</p>
```

Pluralization

- ▷ Parfois, il peut s'avérer utile d'afficher une chaîne ou une autre en fonction d'une variable
 - › « Il y a 1 utilisateur connecté » / « Il y a 20 utilisateurs connectés »

```
{  
  "home": {  
    "title": "ReactJS",  
    "description": "Bienvenue dans la meilleure formation du siècle",  
    "nbUsers": "Il y a {{nb}} utilisateurs connectés"  
  }  
}
```

Variable attendue !

```
<p>{t('home.nbUsers', { nb: 10 })}</p> → Il y a 10 utilisateurs connectés
```

Plural

▷ Un petit problème :

Bienvenue dans la meilleure formation du siècle

Il y a 0 utilisateurs connectés

```
"home": {  
    "title": "ReactJS",  
    "description": "Bienvenue dans la meilleure formation du siècle",  
    "nbUsers_plural": "Il y a {{nb}} utilisateurs connectés",  
    "nbUsers": "Aucun utilisateur connecté"  
}
```

Bienvenue dans la meilleure formation du siècle

Aucun utilisateur connecté

Questions ?