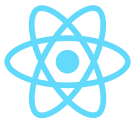
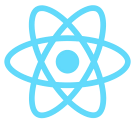
Introduction au développement d'UI avec ReactJS

Objectifs de cours



- Comprendre les notions de base d'une SPA
- Savoir créer des composants React complets
- Comprendre comment React met à jour le DOM
- Connaître les lifecycle Hooks React
- Apprendre comment gérer un state global avec Redux
- Savoir gérer des routes dans une application
- Savoir gérer des formulaires
- Mettre en place un petit projet



Ce cours est organnisé en 4 grandes parties

1. Introduction : Tout ce qu'il y a à savoir pour commencer à utiliser React
2. Hooks : Comment utiliser react de manière moderne et efficace
3. Optimisation : Comment optimiser React et écrire du code performant
4. Bibliothèques tierces : Vue d'ensemble des bibliothèques tierces utiles pour utiliser React pleinement

Rappels ES6

Histoire de commencer tous sur de bonnes bases ;)

Rappels ES6

EcmaScript

- Ensemble de normes sur les langages de type script (Javascript, Actionscript ...)
- Standardisée par Ecma International (European Computer Manufacturers Association) depuis 1994

Rappels ES6

Quelques version d'ES

- **ES5** (ou ES 2009) compatible avec la plupart des navigateurs
- **ES6** (ou ES 2015) qui nécessite un transcompilateur vers ES6 (type Babel) pour fonctionner avec les navigateurs plus anciens (qui a dit IE ?)
- **ES10** (ou ES 2019) la version la plus récente

https://babeljs.io/

https://caniuse.com/#search=es6

ES6 : let

Le mot clé let :

Contrairement à var qui donne une portée locale à une variable, **let** lui donne la portée du bloc dans lequel elle est déclarée :

```
if (true){
  let x = 42;
}
console.log (x); // génère une erreur
```

ES6 : const

Le mot clé const:

Permet de déclarer une référence qui ne change pas de valeur

```
const x = 42;
x = "Hello World"; // génère une erreur
```

ES6 : fonctions fléchées

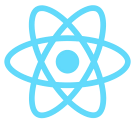
Les fonctions fléchées (ou expressions lambda)

Syntaxe abrégée pour les fonctions

Sans paramètres	Avec paramètres

`// es5`
`function sayHello() {`
`  return 'hello';`
`}`
 	
`// es6 explicite`
`const sayHello = () => {`
`  return 'hello';`
`}`
 	
`// es6 implicite mono-ligne`
`const sayHello = () => 'hello';`
	`// es5`
	`function add(a, b) {`
	`  return a + b;`
	`}`
	
	`// es6`
	`const add = (a, b) => a + b`

ES6 : fonctions fléchées



Les fonctions fléchées (ou expressions lambda)

La seule vraie différence avec une fonction classique, c'est la gestion du **this**

- Dans une fonction classique, **this** représente l'objet qui a appelé la fonction (window, button...)
- Dans une fonction fléchée, **this** représente l'objet au sein duquel est définie la fonction

ES6 : Classes

En Javascript, la POO est basée sur le prototypage, cependant ES6 rajoute des **classes**, plus intuitives car plus proches des autres langages. Il s'agit de sucre syntaxique (on reste sur du prototypage en réalité).

```
class Point {
  // constructeur appelé quand la classe est instanciée
  constructor(abs, ord) {
    this.abs = abs;
    this.ord = ord;
  }
}
const point = new Point(2,4);
```

ES6 : Classes et héritage

```
class Personne {
  constructor(nom, prenom) {
    this.prenom = prenom;
    this.nom = nom;
  }
}
var personne = new Personne("John", "Wick");

class Tueur extends Personne {
  constructor(nom, prenom, chien) {
    super(nom, prenom); // appelle le constructeur de la classe mère
    this.chien = chien;
  }
}
var tueur = new Tueur("John", "Wick", false);
```

ES6 : Destructuring

Le **destructuring** fonctionne avec les objets et les tableaux et permet d'affecter plusieurs variables avec les éléments du tableau ou les propriétés d'un objet.

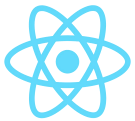
```
// destructuring : tableaux
const personnes = ['Pierre', 'Paul', 'Jacques'];

// sans destructuring
let pierre  = personnes[0];
let paul    = personnes[1];
let jacques = personnes[2];

// avec destructuring
let [pierre, paul, jacques] = personnes;
```

ES6 : Destructuring

Le **destructuring** fonctionne avec les objets et les tableaux et permet d'affecter plusieurs variables avec les éléments du tableau ou les propriétés d'un objet.

```
// destructuring : objets
const personne = {
  prenom : 'John',
  nom    : 'Wick'
};

// sans destructuring
let prenom  = personne.prenom;
let nom     = personne.nom;

// avec destructuring
let {prenom, nom} = personne;
```

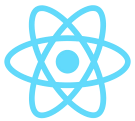
ES6 : Spread operator



En javascript, les itérables (objets, tableaux ...) sont passés par référence, et non par valeur. Le **spread operator** (ou décomposition) permet d'étendre un itérable dans des éléments individuels.

```
var params = [3, 4, 5];
var autres = [1, 2, ...params];
console.log(autres);

var a = { prop : 'val' };
var b = { ...a };
b.prop = false;console.log(a,b);
```

ES6 : Paramètre par défaut



ES6 introduit la possibilité d'avoir des **paramètres par défaut** dans les fonctions comme dans la plupart des autres langages. Ces paramètres sont bien entendu à placer après les paramètres obligatoires.

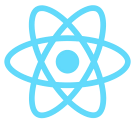
```
const bonjour = (nom = 'anonyme', message = 'Bonjour') => {
  return message+' '+nom;
}
console.log(bonjour());
console.log(bonjour('John'));
console.log(bonjour('frérot', 'Salut'));
```

ES6 : Rest parameter

Le **rest parameter**, représenté avec les «...» permet de représenter un nombre indéfini d'arguments sous forme d'un tableau.

```
function somme(...params) {
  return params.reduce((accumulateur, element) => {
    return accumulateur + element;
  });
}
console.log(somme(1)); // 1
console.log(somme(1, 2, 3, 4, 5)); // 15
```

ES6 : Template literals



Les **templates literals** permettent d'interpoler facilement des données dans des chaînes avec le backtick (Alt Gr + 7 sous PC). Pour interpoler des variables on utilise ${variable}. On peut également gérer des chaînes multilignes de cette façon.

```
const pers = {
  prenom : 'John',
  nom : 'Wick'
}

// ES5
const message = 'Bonjour '+ pers.prenom + ' ' + pers.nom + ' !';
// ES6
const message2 = `Bonjour ${pers.prenom} ${pers.nom} !`
```

ES6 : Promises

Une **promesse** est un objet permettant de lancer certaines opérations sous certaines conditions. Elle dispose de 3 états :

- **Pending** : La promesse a été faite, mais elle n'est pas encore réalisée
- **Fulfilled** : La promesse a été réalisée
- **Rejected** : La promesse ne sera pas réalisée

```
const myPromise = new Promise((resolve, reject) => {
   // code qui déclenche résolve ou reject
});
```

ES6 : Promises

Exemple de promesse : si tu es sage, maman te donnera une glace au goûter.

```
const estSage = false;
const avoirUneGlace = new Promise(
  (resolve, reject) => {
    if (estSage) {
      const glace = { parfum: 'chocolat' }
      resolve(glace); // fulfilled
    } else {
      var raison = new Error('Tu as fait de bêtises');
      reject(raison); // reject
    }
  }
);
```

ES6 : Promises

Le code précédent ne fait que créer notre promesse. Pour qu'elle se réalise, il fait la consommer. Cela se fait avec la méthode then :

```
const demanderMaman = () => {
  avoirUneGlace
  .then(fulfilled => console.log(fulfilled))
  // output: { parfum: 'chocolat' }
  .catch(error => console.log(error.message));
  // output: 'Tu as fait de bêtises'
};
demanderMaman();
```

ES6 : Promises

Les promesses peuvent s'enchaîner. Soit une nouvelle promesse :

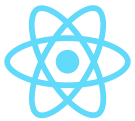
```
const mangerGlace = glace => {
  return new Promise(
    (resolve, reject) => {
      var message = 'J\'adore la glace ' + glace.parfum;
      resolve(message);
    }
  )
};

// syntaxe abrégée
const mangerGlace = glace => {
  var message = 'Hum, j\'adore la glace ' + glace.parfum;
  return Promise.resolve(message);
};
```

ES6 : Promises

Il est possible de chaîner cette nouvelle promesse

```
const demanderMaman = () => {
  avoirUneGlace
  .then(mangerGlace)
  .then((fulfilled) => {
    console.log(fulfilled);
    // output: 'Hum, j'adore la glace chocolat'
  })
  .catch((error) => {
    console.log(error.message);
    // output: 'Tu as fait de bêtises'
  });
};
demanderMaman();
```

ES6 : Import / Export



ES6 introduit les **modules**. Un module est simplement un fragment de code (classe, méthodes, fonctions, variables ...) qui est rendue réutilisable via **l'export** / **import** dans d'autres fichiers.

module.js	index.js

`export let nom1, nom2`
`export function nomFonction(){...}`
`export class NomClasse {...}`
`export { nom1, nom2 }`
`export { var1 as nom1 }`
	`import * as nom from "./module"`
	`import { nom1 } from "./module"`
	`import{ nom1 as alias } from "./module";`
	`import { nom1, nom2 } from "./module";`

`export default nom1`	`import nom1 from "./module"`

Rappel : Map, Filter, Reduce

Déjà présentes dans ES5, ces 3 fonctions sur les tableaux sont absolument essentielles: **map**, **filter**, **reduce**.

Elles permettent toutes de parcourir un tableau et d'effectuer plusieurs opérations sur ses éléments :

- **Map** : la plus générique, permet d'appliquer une fonction à chaque élément d'un tableau
- **Filter** : permet de renvoyer un tableau filtré selon une condition
- **Reduce** : permet d'effectuer des accumulations sur les éléments du tableau

Map

Map prend en argument une fonction dont le paramètre est l'élément en cours du tableau. La fonction doit retourner la nouvelle valeur.

L'indice de l'élément peut être fourni en 2ème argument.

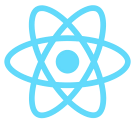
```
const array1 = [1, 4, 9, 16];
const map1 = array1.map(x => x * 2);
console.log(map1); // expected output: Array [2, 8, 18, 32]

const map2 = array1.map((x, i) => x + i);
console.log(map2); // expected output: Array [1, 5, 11, 19]
```

Filter

Filter prend en argument une fonction dont le paramètre est l'élément en cours du tableau. La fonction doit retourner la condition (booléenne) de filtre. L'indice de l'élément peut être fourni en 2ème argument.

```
const users = ['john', 'marc', 'matthew', 'peter', 'paul'];
const result = users.filter(user => user.length > 4);
console.log(result);// expected output: Array ["matthew", "peter"]

const result2 = users.filter((u, i) => i > 2);
console.log(result2); // expected output: Array ["peter", "paul"]
```

Reduce



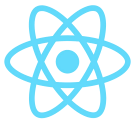
Reduce prend en argument une fonction dont le 1er paramètre est l'accumulateur (la valeur précédente retournée par la méthode) et en 2ème paramètre l'élément en cours du tableau. La fonction retourne l'opération à faire. L'indice de l'élément peut être fourni en 3ème argument.

```
const array1 = [1, 2, 3, 4];
const reducer = array1.reduce((accu, elt) => accu + elt);
console.log(reducer); // expected output : 10

const reducer2 = array1.reduce((accu, elt, i) => accu + elt + i);
console.log(reducer2); // expected output : 16
```

Partie I - Introduction

« React kesako ? »

Introduction

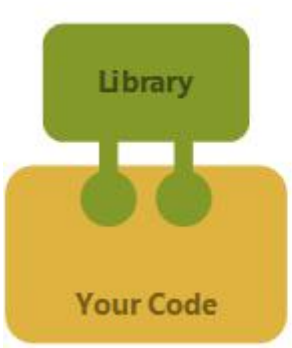


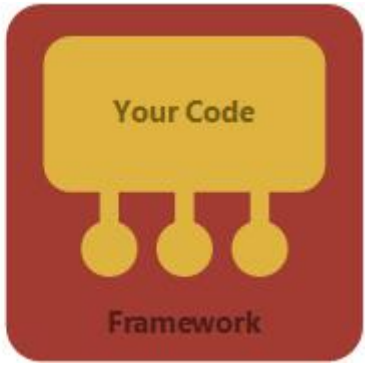
- Librairie javascript open source
- Pour des Single Page Application
- Créée par Facebook en 2011 (dans son newsfeed, puis pour Instagram en 2012)
- Open sourced en 2013
- Programmation déclarative
- Orienté composants
- Utilisant du JSX
- Basée sur un virtual DOM

Librairie ou Framework

- React est définie comme une librairie, mais dans les faits, elle se trouve un peu à cheval entre les deux mondes.
- Différence entre librairie et framework ?

  Inversion de contrôle :
  - **Libraire** : Votre code appelle la librairie
  - **Framework** : Le framework appelle le développeur

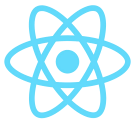




Différence avec Angular

- React offre plus de libertés qu'un framework comme Angular, ce qui peut être une bonne comme une mauvaise chose
- React est plus facile à maîtriser qu'Angular
- React contient moins de fonctionnalités pré-installées (routage, flux...)
- React propose un data binding uni directionnel

Programmation déclarative

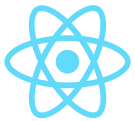


- Opposée du style impératif
- Déclaratif : « Ça devrait ressembler à ceci »
- Impératif : « CA doit faire cela »
- Plus facile à lire, moins d'abstraction (la librairie gère l'abstraction elle même)
- D'autres exemples : SQL, CSS, ...
- Dans React on créée des UI interractives en changeant le state (les données), React se charge de modifier le DOM en fonction.

Components

- React est basé sur l'utilisation de composants
- Des briques d'UI autonomes qui gèrent leurs propres données (state)
- Ces briques peuvent être assemblées pour créer des interfaces plus complexes
- Les composants peuvent communiquer et interagir entre eux

Prérequis



- Connaître JavaScript (ES6 et plus)
- Aisance à l'utilisation du terminal (bash ou cmd.exe)
- Installation
  - NodeJS
  - NPM
  - Visual Studio
  - Google Chrome 63+ ou Firefox Developer Edition 58+
  - Create React App (optionnel)
  - React Developper Tool

Hello World

On ne va pas y couper

Hello World

C'est le moment de coder. Nous allons créer une toute petite application from scratch qui affichera un « **Hello World** » (original...)

1. Créer un fichier index.html avec un squelette basique
2. Importer les fichiers js de react depuis un CDN : https://fr.reactjs.org/docs/cdn-links.html
3. Importer babel depuis un CDN : https://babeljs.io/setup#installation

Le fichier index.html

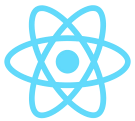
Intégrer dans le <body> du fichier index.html :

1. Une div avec un ID (souvent root ou app par convention)
2. Une balise script de type text/babel
3. Dans cette balise script, utiliser la méthode render de l'objet ReactDOM que l'on a importé plus haut, avec les paramètres suivants :
   - **Param 1** : Le html à afficher (Hello World dans notre cas, pourquoi pas dans une balise <h1>)
   - **Param 2** : L'élément html au sein duquel on veut l'afficher (avec un document.getElementById)

Le fichier index.html

```
<!DOCTYPE html>
<html lang="en">
 <head>
  <meta charset="UTF-8">
  <meta name="viewport" content="width=device-width, initial-scale=1.0">
  <title>Component test</title>
  <script
src="https://unpkg.com/@babel/standalone/babel.min.js"></script>
  <script crossorigin
src="https://unpkg.com/react@16/umd/react.development.js"></script>
  <script crossorigin src="https://unpkg.com/react-dom@16/umd/react-
dom.development.js"></script>
 </head>
 <body>
  <div id="root"></div>
  <script type="text/babel">
    ReactDOM.render(<h1>Mon titre</h1>, document.getElementById("root"));
  </script>
 </body>
</html>
```

Hello World



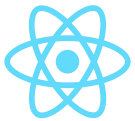
Pour écrire une première page avec React, nous avons ajouté trois scripts: React, ReactDOM et Babel.

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 plate-formes 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
- React Sketch pour générer des fichiers Sketch

Premier composant



- Dans notre script, avant le render(), créer une classe App qui étends React.Component (les majuscules sont importantes)
- Dans la classe App, ajouter une méthode render qui retourne notre <h1>Hello World</h1>
  N'utilisez pas de quotes. Retournez directement la balise.
- Modifier la méthode render de ReactDOM pour qu'elle ne prenne plus notre balise h1 en premier paramètre, mais la balise suivante : <App /> (sans quotes ici aussi)

Premier composant

Votre code devrait ressembler à ça :

```
<div id="root"></div>
<script type="text/babel">
    class App extends React.Component {
      render() {
        return <h1>Hello World</h1>
      }
    }
    ReactDOM.render(<App />, document.getElementById("root"))
</script>
```

Notez que la méthode render doit retourner qu'une seule balise (qui peut éventuellement en contenir d'autres...)

Félicitations ! Vous avez créé un composant.

Premier composant

Ce que fait notre code

- Nous avons d'une part une balise div vide. C'est l'élément de la page dans lequel nous allons injecter notre application react.
- Nous créons ensuite un composant classe App qui retourne du code (html?) au sein d'une méthode render
- Nous utilisons ensuite la méthode render de ReaactDOM pour ajouter notre composant sur la balise div.

JSX

« Mais que font ces balises html dans mon code js ? »

JSX

Javascript **S**yntax e**X**tension

- C'est extension syntaxique du js (... ne me remerciez pas) qui permet d'écrire du code proche du html/xml au sein du javascript.
- JSX permet de déterminer la future apparence de nos composants (style déclaratif)
- Ce code JSX sera ensuite transcompilé en Javascript
- Le composant complet (logique + rendu) est contenu dans un seul fichier
- Attention : **cela reste du Javascript !**

JSX

JSX nous permet d'interpoler directement des variables à l'intérieur entre accolades : { variable }

```
class App extends React.Component {
  render() {
    const name = "David"
    return <h1>Hello {name}</h1>
  }
}
```

- On peut utiliser n'importe quelle expression javascript entre accolades (2+2, user.name ...)
- On peut également utiliser JSX pour modifier dynamiquement des attributs html :

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

JSX

Conventions du JSX :

Utiliser les accolades pour utiliser des expressions JS à l'intérieur et uniquement des expressions.

JSX Valide	JSX Non Valide
{1+1} {true} {name} {user.name} {getUserName()}	{const user = 'David'} {while(posts){<Posts />}} {if(user.logged){<UserProfile />}}

JSX

Conventions du JSX :

Pas de guillemets autour du JSX, ni autour des expressions JS utilisées entre accolades.

Dans JSX, les guillemets représentent une chaine en dur.

```
return (
    <Card className='ml-5'>
      <Card.Body>
        {comment.content}
      </Card.Body>
    </Card>
)
```

JSX

Conventions du JSX :

camelCase pour les noms des attributs html classiques (ex : tabindex devient tabIndex)

Le terme class étant réservé par javascript, il faut utiliser l'attribut className pour l'attribut html class

```
return (
    <Card className='ml-5'>
      <Card.Body>
        {comment.content}
      </Card.Body>
    </Card>
)
```

JSX

Conventions du JSX :

Le code JSX retourné par un composant doit être contenu dans une seule balise (doit posséder un élément racine)

L'élément racine étant visible dans le DOM, il sera parfois préférable d'utilise un Fragment, ou sa syntaxe abrégée : <>...</>

```
return (
  <div>
    <NewPost />
    <NewsFeed />
  </div>
)
```

```
return (
  <React.Fragment>
    <NewPost />
    <NewsFeed />
  </React.Fragment>
)
```

```
return (
  <>
    <NewPost />
    <NewsFeed />
  </>
)
```

JSX

Conventions du JSX :

Les balises auto-fermantes doivent être correctement écrites :
 au lieu de

Toute balise commençant par une minuscule est réservée au html

Toute balise commençant par une majuscule doit être déclarée dans le scope courant et est réservée aux composants React (natifs, custom ou importés)

Conditions ?

Il faut bien comprendre que JSX produit avant tout une expression Javascript, et non pas une instruction.

Il n'est donc pas possible d'utiliser un if / for / while à l'intérieur d'un code JSX.

C'est cependant possible de conditionner le retour de notre composant :

```
if (isLoggedIn) {
  return <LogoutButton />;
} else {
  return <LoginForm />;
}
```

Conditions ?

Et si je dois tout de même conditionner mon JSX ? Je peux utiliser une expression javascript : l'opérateur ternaire

`booleen ? retour_si_vrai : retour_si_faux`

```
return <div>{isLoggedIn ? <LogoutButton /> : <LoginForm />}</div>
```

Pour une condition simple (sans else, donc) on utilise l'opérateur logique **&&**

`booleen && retour_si_vrai`

```
return <div>{isLoggedIn && <LogoutButton />}</div>
```

Boucles ?

JSX permet également d'intégrer des expressions quelconques entre { }

Il est donc possible d'utiliser .map() directement à l'intérieur :

```
return (
  <ul>
    {users.map((user) =>
      <ListUser key={user.id} avatar={user.avatar} />
    )}
  </ul>
);
```

Boucles ?

Notez la présence de l'attribut key, un attribut réservé par React qui doit être unique à chaque élément de la liste lorsque l'on affiche des liste.

C'est dû au fonctionnement du virtual DOM, sur lequel nous reviendrons en détail.

Cette clé doit correspondre à l'ID de l'élément en BDD mais pas à l'index du tableau.

```
<ul>{users.map((user) =>
   <ListUser key={user.id} avatar={user.avatar} />
 )}</ul>
```

DOM virtuel

Kesako ? Ou : Pourquoi React est si rapide ?

DOM virtuel

- Ici nous relançons la méthode render à chaque seconde. Méthode qui affiche notre composant App

```
class App extends React.Component {
  render() {
    return <h1>Hello World
     <div>Il est actuellement {new Date().toLocaleTimeString()}</div>
    </h1>;
  }
}
setInterval(() => {
  ReactDOM.render(<App />, document.getElementById("root"))
}, 1000);
```

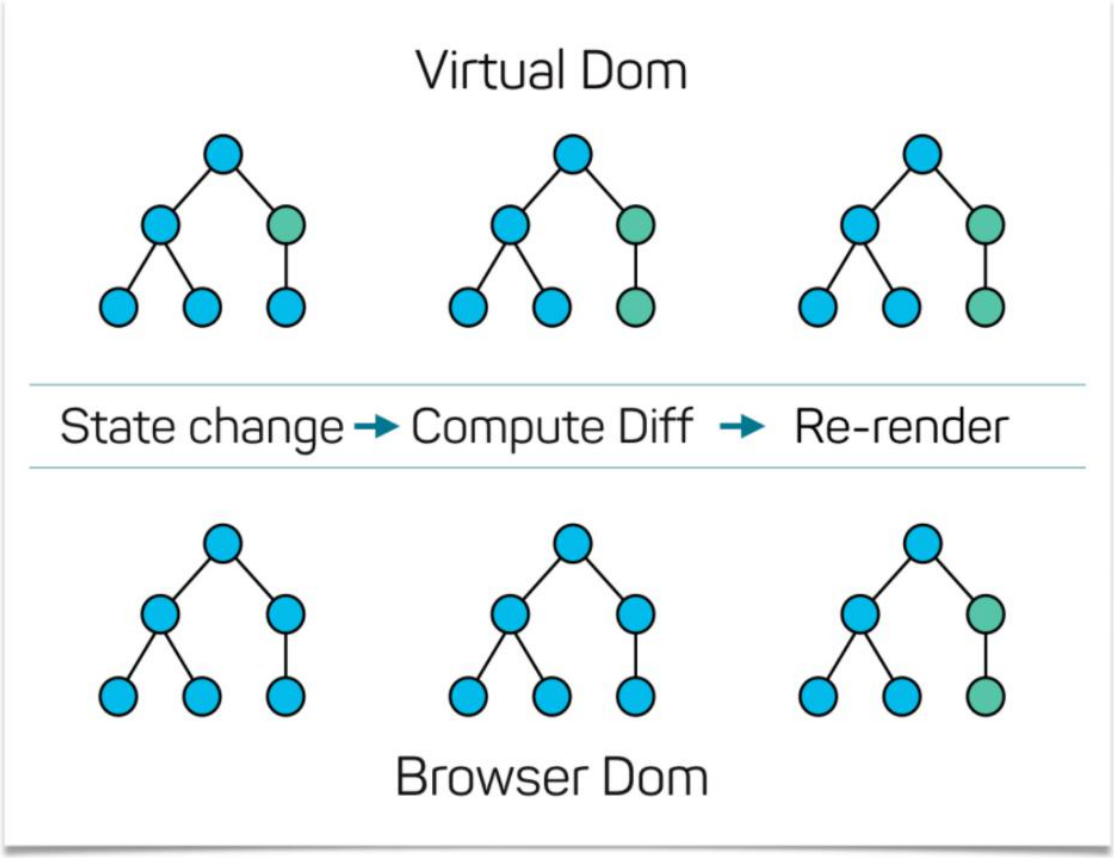
DOM virtuel

- En affichant la source html rendu avec les outils de développeurs, on se rend compte que seule est re-rendue la partie qui est réellement modifiée par notre code (à savoir l'heure).
- Le reste est conservé en mémoire et non modifié.
- C'est ce que React appelle le React DOM (aussi appelé Virtual DOM)

DOM virtuel

- React conserve l'intégralité du DOM de notre app en mémoire dans un DOM virtuel (géré par ReactDOM)
- A chaque appel d'une méthode render, React va :
  - modifier le DOM virtuel
  - impacter les modifications dans tous les composants qui doivent changer si nécessaire
  - comparer le nouveau DOM virtuel au précédent DOM virtuel au DOM réel et calculer les différences
  - mettre à jour le DOM réel en ne tenant compte que de ces différences

DOM virtuel


Virtual Dom
State change
Compute Diff
Re-render
Browser Dom

Create React App

On passe aux choses sérieuses !

Create React App

- Pour commencer à étudier les composants, nous allons utiliser create-react-app

```
npx create-react-app mon-app
cd mon-app
npm start
```

- Create react app ne prend pas en charge la logique coté serveur, les bdd ... Il construit uniquement la partie front
- Intègre Babel et Webpack

Create React App

- Une fois le script terminé, vous aurez un dossier mon-app avec l'arborescence suivante :

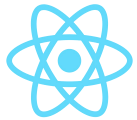
```
mon-app/
├── node_modules/ # Contient toutes les dépendances
├── public/ # Fichiers statiques servit par le serveur
├── src/ # Code source de l'application
├── .gitignore
├── package-lock.json # Liste des sous-dépendances
├── package.json # Liste des dépendances
└── README.md
```

Create React App

- Une fois le script terminé, vous aurez un dossier mon-app avec l'arborescence suivante :

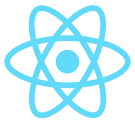
```
mon-app/src/
 ├── App.css # CSS du composant <App />
 ├── App.js # Définition du composant <App />
 ├── App.test.js
 ├── index.css # CSS global
 ├── index.js # Principal fichier JS
 ├── logo.svg
 ├── serviceWorker.js # JS lié au service worker
 └── setupTests.js # Fichier de test pour le DOM
```

Create React App



- La commande npm start va lancer Webpack en mode développement, avec le hot-reloading activé, rechargement automatiquement les onglets où votre application est lancée quand votre code est modifié.
- Familiarisez-vous avec le code généré et essayez de le modifier pour comprendre son fonctionnement.

Create React App



Tentons de recréer notre composant Hello World dans ce nouvel environnement.

Pour cela, créez un fichier HelloWorld.js dans le dossier src.

Par convention, les fichiers qui décrivent des composants ont le même nom que le composant, et ils doivent être écrit en PascalCase.

Comme notre composant est très simple, nous pouvons passer par une fonction plutôt qu'une classe.

Create React App

Dans ce fichier HelloWorld.js , nous allons d'abord importer la librairie ReactJS.

```
import React from 'react'
```

Un composant doit retourner son contenu (un élément React au format JSX)

```
function HelloWorld(props) {
  return <h1>Hello World</h1>
}
```

Enfin, il nous faut exporter ce composant (par convention, avec un export par défaut)

```
export default HelloWorld
```

Create React App

Enfin, il faut l'ajouter dans notre application.

Dans le composant racine (App.js), il faut importer notre composant :

```
import HelloWorld from './HelloWorld'
```

Et enfin ajouter le composant HelloWorld dans le JSX retourné par notre composant App

```
function App() {
  return <div><HelloWorld /></div>
}
```

Create React App

- Notez qu'il est possible d'intégrer des composants au sein d'autres composants de cette façon. Il est même possible d'en créer plusieurs instances si besoin :

```
function App() {
  return (<div>
    <HelloWorld />
    <HelloWorld />
    <HelloWorld />
  </div>)
}
```

- Dans notre exemple, App est appelé composant parent et HelloWorld, composant enfant

Composants

La brique de base d'une application React

Composants

- Les composants vous permettent de découper l'interface utilisateur en éléments indépendants et réutilisables
- Conceptuellement, les composants sont comme des fonctions JavaScript ou des classes ES6 :

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

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

Composants

Un composant sous forme de fonction :

- Peut prendre un argument (optionnel), nécessairement nommé props
- Doit retourner un élément React valide

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

Les props sont les propriétés de notre composant, à savoir, des données qui lui sont transmises (cf chapitre suivant)

L'élément React retourné doit être une (et une seule) balise jsx (html ou balise composant) qui peut à son tour contenir d'autres balises

Composants

Un composant sous forme de classe:

- Doit étendre React.Component
- Doit contenir une méthode render() au minimum
- Cette méthode doit retourner un élément React

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

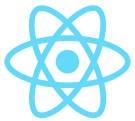
L'élément React retourné doit être une (et une seule) balise JSX (html ou balise composant) qui peut à son tour contenir d'autres balises

Composants

Tout l'intérêt des composants étant d'être modulaires, il convient également d'importer react et d'exporter notre composant quel que soit le type de composant.

```
import React from 'react';

 /* ... */

export default App;
```

Composant fonctionnel ou composant classe ?



Avant la version 16.8 (Février 2019) :

Seuls les composants classe pouvaient gérer le state et les lifecycle hooks tandis que les composants fonctionnels étaient dits stateless et ne servaient qu'à de l'affichage.

Depuis la version 16.8 :

L'arrivée des hooks rend les deux types de composants viables, et Facebook tend même à recommander davantage les composants fonctionnels de manière générale :

https://reactjs.org/docs/hooks-intro.html#motivation

C'est donc sur ces composants que nous allons nous attarder.

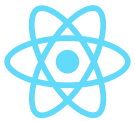
Mise en pratique

ToDoList

Events

Quelques petites spécificités de React sur les événements

Events



Les events vont nous permettre de rendre notre application dynamique. React nous permet de déclarer nos gestionnaires à même le code JSX de nos composants tout en gardant des performances optimales.

Quelques différences avec les événements classiques du DOM :

- les noms sont en camelCase
- en JSX on passe une fonction event handler plutôt qu'une chaîne de caractères

Différences avec Javascript

- event handler :

```
// Javascript
<button onclick="release()">Release the Kraken !</button>
```

```
// JSX
<button onClick={release}>Release the Kraken !</button>
```

- prevent default :

```
// Javascript
<a href="#" onclick="console.log('clic'); return false">Cliquer</a>
```

```
// JSX
function handleClick(e) {
  e.preventDefault();
  console.log('clic.');
}
return (<a href="#" onClick={handleClick}>Cliquer</a>);
```

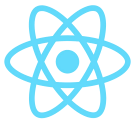
Mise en pratique

ToDoList

Props

Comment passer de la data aux composants enfants

Props



Les props sont, comme leur nom l'indiquent, des propriétés.

Elles permettent de transmettre des données d'un composant parent à un composant enfant.

Les props sont readonly. Elles sont logiquement faites pour transmettre des données qui doivent uniquement affichées telles quelles.

Les props peuvent être des variables, tableaux, objets, et même des fonctions

Props : transmission

Pour transmettre une propriété, il suffit de l'ajouter en attribut sur l'élément JSX. Lorsque c'est le cas, React transmet au composant enfant un objet nommé props.

```
function App() {
  return <div><HelloWorld name="John Wick" /></div>
}
```

Props : récupération

Pour récupérer une propriété dans une classe enfant, il suffit de faire appel au paramètre props si on est dans un composant fonction ou à l'attribut this.props si on est dans un composant classe.

```
function HelloWorld(props) {
  return <h1>Hello {props.name}</h1>
}
```

Props : fonctions

Il est également possible de transmettre des fonctions dans les props, d'un parent vers un enfant. Puis de lancer l'éxécution de la fonction du parent au sein de l'enfant :

```
const Parent = props => {
   const parentFunction = () => {
    console.log("fonction du parent déclenchée par l'enfant")
  }
   return <Enfant parentFunction={parentFunction} />
}
```

```
const Enfant = props => {
  const handleClick = () => props.parentFunction()
  return <button onClick="handleClick">click me</button>
}
```

Props

Il existe trois props réservées par React:

- **key**: qui sert à différencier les composants dans une liste à l'aide d'un identifiant unique (obligatoire dans ce cas). Les key vont aider ReactDOM à détecter les changements.
- **ref**: permet d’identifier et d'accéder directement à un élément via le DOM
- **children**: liste le contenu entre les balises ouvrantes et fermantes du composant.

Props : children

Dans props.children se trouve le contenu d'une balise composant :

```
<Welcome>Bonjour monde !</Welcome>
```

```
function Welcome(props) {
  return <p>{props.children}</p>;
}
```

Props par défaut, et PropTypes

Il est possible de définir des props par défaut, des props requis, et des types attendus pour chaque props. Cela se fait à l'aide du module props-type qu'il faut importer :

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

/* ... Code de la classe composant ... */

HelloWorld.defaultProps = {
  greeting: 'Hello',
  name: 'World',
}

HelloWorld.propTypes = {
  greeting: PropTypes.string.isRequired,
  name: PropTypes.string,
}
```

Props par défaut, et PropTypes

Les PropTypes ne sont pas obligatoires mais vivement recommandées.

Elles permettent une vérification qui facilite la collaboration entre différents développeurs en affichant directement dans la console si proptype définie n'est pas respectée.

De plus, elles n'impactent pas les performances de votre application en production vu qu'elles sont effacées lors de la création de la phase de build.

La liste complète des validateurs est disponible ici : https://fr.reactjs.org/docs/typechecking-with-proptypes.html

Mise en pratique

ToDoList

Style CSS

On fait du front après tout

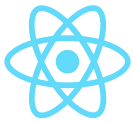
Style CSS

Une façon d'intégrer du style dans notre application va être d'ajouter directement le CSS dans le fichier style du dossier public, puis d'appliquer les classes dans nos composants.

Ca fonctionne et c'est très bien, lorsque nous avons des styles globaux sur notre application (couleurs, polices, marges générales ...)

Cependant React est fait pour nous permettre de rendre nos composants les plus autonomes possibles, et donc de lier le CSS au composant directement.

Une feuille de style par composant



Méthode 1 : une feuille de style par composant

- Créer un dossier par composant contenant le composant .js et la feuille de style .css correspondant
- Dans le composant, importer la feuille de style
- Utiliser les classes de notre feuille de style dans le JSX du composant

Une feuille de style par composant

Méthode 1 : une feuille de style par composant

- Avantages :
  - Facile à mettre en place
  - Facile de trouver et maintenir le style d'un composant
- Désavantages :
  - Le css n'est pas limité au composant
  - Pas si facile de trouver la feuille de style responsable du style d'un composant lorsqu'on déborde...

Style inline dans le composant

Méthode 2 : inline style

Une autre façon de faire est d'avoir une constant styles dans chaque composant qui définit une liste de propriétés / valeurs css

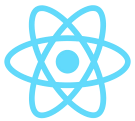
```
const styles = {
    task: {
        color: blue,
        fontWeight: bold
    }
}
const Task = ({ task }) =>
    <span style={styles.task}>{task.label}</span>
```

Style inline dans le composant

Méthode 2 : inline style

- Avantages :
  - Style hermétique
- Désavantages :
  - Impossible de définir un style qui s'applique aux enfants
  - Peu recommandé par React pour des raisons de performances

Bibliothèque tierce



Méthode 3 : Utiliser des bibliothèques tierces

Ex : StyledComponents

Partie II - React Hooks

Coder du React après 2019

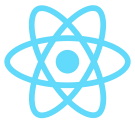
Hooks ?

Les Hooks sont une nouveauté introduite par React 16.8 (fin 2018). Ils permettent de bénéficier d'un état local et d'autres fonctionnalités de React sans avoir à écrire de composants classes.

On distingue trois grands types de Hooks :

- Les **hooks d'état (useState)** faits pour gérer des données propres à un composant
- Les **Hooks d'effet (useEffect)** faits pour générer des effets supplémentaires à certains moments
- Les **Hooks de contexte (useContext)** faits pour faire transiter des données à travers l'arborescence

Hooks

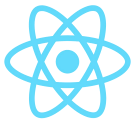


Les hooks doivent respecter certains principes dans leur utilisation :

- Être appelés au niveau racine (pas dans une boucle, une condition ou une fonction imbriquée)
- Être appelés dans des fonctions (composants fonctionnels ou dans des hooks personnalisés)
- Avoir un nom qui commence par «use»

Heureusement, un linter (eslint-plugin-react-hooks) est inclus automatiquement dans create-react-app et permet de vérifier le respect de ces règles.

Hooks

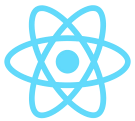


Globalement, par rapport aux précédentes versions de React, les hooks rendent les composants React plus simple à lire et à maintenir.

A noter tout de même que : les hooks sont encore relativement récents. Même si leur stabilité n'est pas à mettre en doute, toutes les bibliothèques React tierces ne sont pas nécessairement passées aux hooks.

State et Hook d'état

State



Le state d'un composant représente son état qui lui est propre : un objet contenant les données propres au composant.

Contrairement aux props, le state peut être modifié, mais pas transmis à un autre composant. Les autres composants n'y ont pas accès.

Étant donné que chaque modification du state demande à React de re-render le DOM, il faut les modifier d'une manière spécifique : via useState

State

- Pour utiliser le state **importer** useState

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

- Pour définir un state, on utilise le destructuring

```
const [name, setName] = useState('John Wick');
```

- Pour afficher le state, il suffit de l'interpoler

```
<p>{name}</p>
```

- Pour modifier le state on utilise le setter défini avec notre state (ici setName), par exemple sur un event

```
setName(newName)
```

State : useState

useState déclare une variable dans le state. Elle est à appelée autant de fois que l'on veut de variables dans notre state.

```
  const [name, setName] = useState('John Wick');
  const [age, setAge] = useState(1);
```

- Paramètre : useState prend en paramètre le state initial (chaîne, nombre, objet ...)
- Valeur de retour : useState renvoie une paire : le state courant et fonction setter de ce state

```
const [name, setName] = useState('John Wick');
```

State

- C'est grâce à l'utilisation de ce setter que React comprend ce qui a été modifié et peut ensuite mettre à jour uniquement cette partie du DOM.

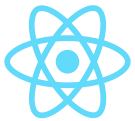
```
setName(newName)
```

- On va maintenant pouvoir l'utiliser au sein d'événements par exemple :

```
<button onClick={() => setName(newName)}>Change Name</button>
```

Formulaires

Formulaires



Maintenant qu'il est possible de gérer un state, et avant de continuer sur les hooks, nous allons voir comment modifier dynamiquement ce state à travers les formulaires

En HTML, les éléments de formulaire tels que <input>, <textarea>, et <select> maintiennent généralement leur propre état et se mettent à jour par rapport aux saisies de l'utilisateur. En React, l'état modifiable est généralement stocké dans le state des composants.

Formulaires

On peut combiner ces deux concepts en utilisant le state React comme « source unique de vérité ». Le composant React qui affiche le formulaire contrôle aussi son comportement par rapport aux saisies de l'utilisateur. Un composant dont l'état est contrôlé de cette façon est appelé «composant contrôlé».

Comme en JS, la fonciton appelée lorsqu'un événement est déclenché prend en paramètre l'objet événement. Il est alors possible d'utiliser event.target.value pour obtenir la valeur saisie.

Formulaires : inputs

```
const NewTask = props => {

  const [value, setValue] = useState('');

  const handleSubmit = (e) => {
    e.preventDefault();
    props.addTask(value)
    setValue('')
  }

  const handleChange = (e) => {
    setValue(e.target.value)
  }

  return (
    <form onSubmit={handleSubmit}>
     <input type="text" name="newTask" value={value} onChange={handleChange} />
     <button type="submit">+</button>
    </form>
  )
}
```

Formulaires : inputs

Ici, l'attribut value de notre input sera toujours le state value, faisant du state la source de vérité. Puisque handleChange est déclenchée à chaque frappe pour mettre à jour le state, la valeur affichée reste mise à jour au fil de la saisie.

Dans un composant contrôlé, la valeur du champ est pilotée par le state React.

A noter : Lorsque nous utilisons l'attribut value, l'attribut onChange est obligatoire (autrement React retourne un warning)

Formulaires : textarea et select

Pour les balises textarea, cela va fonctionner d'une manière très similaire aux inputs.

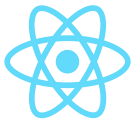
Pour un select, il suffit de mettre la value et l'event onChange directement sur la balise select.

```
<select value={this.state.value} onChange={this.handleChange} >
 <option value="grapefruit">Pamplemousse</option>
 <option value="lime">Citron vert</option>
 <option value="coconut">Noix de coco</option>
 <option value="mango">Mangue</option>
</select >
```

Hook d'Effet

Anciennement appelés « lifecycle hooks »

Effect Hooks



Pour appeler des API externes, React ne dispose pas de méthode qui lui soit propre. On utilise simplement la fonction native fetch de JS.

Mais il convient de savoir comment gérer cela au mieux avec un composant React.

La première question est de savoir où effectuer l'appel.

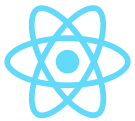
Effect Hooks

Evidemment, l'appel est à faire dans le composant de plus haut niveau qui a besoin de dispatcher les données à travers les composants enfant.

Mais où ? Un appel de la fonction setter de notre state ? Directement dans le code du composant ?

Aucune de ces deux solution ne semble convenir, puisque les appels à l'API seraient effectués à chaque render de notre composant.

Effect Hooks



Nos composants ont des cycle de vie, des phases, de leur montage à leur démontage, durant lesquelles on peut intervenir grâce à des Hooks d'effet

Si vous êtes familiers des versions antérieures de React, ***useEffect*** *est une combinaison de* ***componentDidMount****,* ***componentDidUpdate****, et* ***componentWillUnmount****.*

Effect Hooks : useEffect

Les Hooks d'effet (useEffect) permettent de gérer des effets de bord. Il en existe deux grands types :

- les effets qui ne nécessitent aucun nettoyage
- les effets qui nécessite un nettoyage

Dans les deux cas, ils sont gérés avec **useEffect**

Un **effet de bord** est simplement une fonction qui effectuera des actions supplémentaires et sera déclenchée après chaque mise à jour du DOM. Un effet peut être de nature très diverse (appel à une API, charger des données, console.log ...)

Effect Hooks : useEffect

useEffect prend :

- en premier paramètre : une **fonction** qui va être appelée à chaque fois que le composant va être render.
- en second paramètre : un **tableau optionnel** de valeurs dont l'effet dépend. Il s'exécutera de manière conditionnelle
  - Si ce tableau n'est pas fourni, useEffect s'effectuera à chaque render (montage + mise à jour)
  - Si ce tableau est fourni, useEffect ne s'effectuera qu'à la mise à jour (il ne dépend que des données passées en argument)
  - Si c'est un tableau vide qui est fourni, useEffect ne s'effectuera qu'au montage (il ne dépend d'aucune donnée)

Effect Hooks : sans nettoyage

Les effets «sans nettoyage» interviennent lorsque l'on veut exécuter du code supplémentaire après la mise à jour du DOM

***Exemples** : requête réseau, modification manuelle du dom, console.log ... autant d'effets qui ne nécessitent aucun nettoyage*

Grâce à useEffect, il est possible de faire intervenir ces effets après chaque render de notre composant (au montage et à la mise à jour). Par défaut, useEffect sera appelée après chaque affichage (possibilité d'optimiser cela)

Effect Hooks : sans nettoyage

Ici, le console.log va s'afficher à chaque fois que notre composant Compteur va être rendu. Donc à l'initialisation et au click du bouton.

Remarque : à chaque affichage, un nouvel effet succède au précédent (useEffect exécute une nouvelle fonction)

```
function Compteur() {
  const [count, setCount] = useState(0);
  useEffect(() => console.log(count));
  return (<>
    <h1>Vous avez cliqué : {count} fois</h1>
    <button onClick={()=>setCount(count+1)}>+1!</button>
  </>);
}
```

Effect Hooks : sans nettoyage

Ci dessous, le cas typique d'un appel à une API. Notre useEffect est conditionné sur l'ID de l'utilisateur courrant.

```
useEffect(() => {
  fetch(url)
    .then(res => res.json())
    .then(result => setUserProfile(result))
    .catch(err => console.log(err))
},[currentUser.id])
```

Attention cependant aux boucles infinies : mettre en condition d'arrêt un state qui est modifié durant l'appel de useEffect (et donc relance l'effet).

Effect Hooks : avec nettoyage

Les effets «avec nettoyage» interviennent lorsque l'on s'abonne à des observables avec RXJS par exemple.

Pour ce faire, la fonction paramètre de useEffect peut retourner une fonction de nettoyage.

La fonction de nettoyage est exécutée avant que le composant ne soit retiré de l'UI pour éviter les fuites mémoire.

```
useEffect(() => {
  const subscription = props.source.subscribe();
  return () => {
    // Nettoyage de l'abonnement
    subscription.unsubscribe();
  };
});
```

Effect Hooks : avec nettoyage

Si un composant est rendu plusieurs fois, l'effet de bord précédent est nettoyé avant l'exécution du prochain. Dans cet exemple cela veut dire qu'un nouvel abonnement est recréé à chaque mise à jour.

Il est cependant possible de fournir en 2ème argument de useEffect un tableau de valeur dont l'effet dépend. Dans ce cas l'abonnement n'est recréé que lorsque props.source change.

```
useEffect(() => {
  const subscription = props.source.subscribe();
  return () => {
    // Nettoyage de l'abonnement
    subscription.unsubscribe();
  };
},[props.souce]);
```

Effect Hooks : avec nettoyage

Il est possible d'obtenir un effet similaire aux fonctions lifecycle avec les hooks d'effet :

```
// Equivalent de componenentDidMount
useEffect(() => {
  /*...*/
},[]);
```

```
// Equivalent de componenentDidUpdate
useEffect(() => {
  /*...*/
},[dependance]);
```

```
// Equivalent de componenentWillUnmount
useEffect(() => {
  return () => { /* ... */ }
},[]);
```

Hooks de contexte

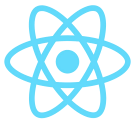
Application State vs UI State

Il y a plusieurs types de state dans une app :

- Application State: état général d'une application (stocké dans une base de données, par exemple)
- UI State: état propre à une partie de l'application (ex: formulaire), éphémère et qui peut être effacé

Note : il existe d'autres types de state qui ne sont pas évoqués ici (router state, transient client state ...)

Application State vs UI State



Une bonne pratique est de laisser l'UI State propre à un composant (ou à une petite arborescence de composants).

Mais l'Application State contient des data plus globales, dont de nombreux composans ont besoin à travers toute l'arborescence.

Dans une application React typique, les données sont passées de haut en bas via les props, mais cela devient vite lourd pour gérer l'App State

Contexte

Le Contexte offre un moyen de partager des valeurs comme celles-ci entre des composants sans avoir à explicitement passer une prop à chaque niveau de l'arborescence.

Le Contexte est conçu pour partager des données qui peuvent être considérées comme « globales » pour une arborescence de composants React :

- utilisateur authentifié
- thème choisi
- langue

Utiliser le contexte

Pour créer un contexte, en haut de l'arborescence, on utilise React.createContext :

```
const defaultValue = 'light'
const ColorTheme = React.createContext(defaultValue);
```

Pour transmettre le contexte aux composants déscendants :

```
<ColorTheme.Provider value={pickedColor} />
  <ChildComponent />
</ColorTheme.Provider>
```

Pour récupérer et consommer du contexte, n'importe où, plus bas dans l'arborescence :

```
const theme = useContext(ColorTheme);
```

Hooks de contexte

Ce qu'il est important de comprendre :

- useContext est une souscription à un contexte
- a chaque changement du contexte, l'arbre de composants qui l'utilisent va être re-render

Pour éviter de re-render trop souvent (et optimiser notre app) il vaut mieux gérer plusieurs contextes et les transmettre aux composants qui les requierent. Plutôt que d'utiliser un contexte global qui contiendrait plusieurs objets

Modifier le contexte

Il n'est pas rare qu'au sein d'un même contexte, on trouve des données et des fonctions pour modifier ces mêmes données au sein d'un même objet.

Ainsi on peut modifier les données du contexte n'importe où dans l'arborescence :

```
<ColorTheme.Provider value={{pickedColor, changeTheme}} />
  <ChildComponent />
</ColorTheme.Provider>
```

On utilise alors le destructuring dans le context consumer pour les récupérer

```
const { theme, changeTheme } = useContext(ColorTheme);
```

Custom Hooks

Custom Hooks

Un Hook personnalisé est une fonction JavaScript dont le nom commence par «use» et qui peut appeler d'autres Hooks.

Le but est de factoriser un maximum le code répétitif dans ces hooks personnalisés.

Les hooks personnalisés ne sont ni des hooks d'état, ni des hooks d'effet, bien qu'ils peuvent (doivent ?) appeler d'autres hooks.

Custom Hooks

Un exemple simple : nous voulons modifier le titre du document (affiché dans l'onglet), par exemple si le state évolue (notifications, nouveaux messages ...)

```
function Compteur() {
  const [count, setCount] = useState(0);
  const increment = () => setCount(count + 1);
  useEffect(
    () => {document.title = `Vous avez cliqué ${count} fois` },
    [count]
  )
   return <>
    <h1>{count} clicks</h1>
    <button onClick={increment}>+1</button>
  </>
}
```

Custom Hooks

On peut alors extraire la logique du hook useEffect de notre composant pour le rendre plus générique et réutilisable à travers un hook personnalisé :

```
const useTitle = title => {
  useEffect(
    () => {document.title = title},
    [title]
  )
}
```

```
function Compteur() {
  const [count, setCount] = useState(0);
  const increment = () => setCount(count + 1);
  useTitle(`Vous avez cliqué ${count} fois`);
  return <>
    <h1>{count} clicks</h1>
    <button onClick={increment}>+1</button>
  </>
}
```

Partie III - Optimisation

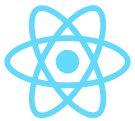
Une app qui marche... C'est bien.

Une app rapide... C'est mieux !

useCallback

Optimiser les performances

useCallback



Déclarer des fonctions dans le corps d'un composant pose problème d'un point de vue optimisation : à chaque render du composant, une nouvelle fonction est créée par JS. On ne va pas pouvoir faire grand chose contre ça.

En revanche, si cette fonction est passée en props à un compsant enfant, cela va déclencher le render du sous arbre de composant

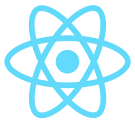
useCallback

```
const [name, setName] = useState('John')
const [age, setAge] = useState(42)
const logName = () => console.log(name)
```

Dans le composant ci-dessus, l'appel à setAge déclenche un nouveau rendu du composant. A ce moment, une nouvelle instance de logName est créée. Si cette fonction est passée en paramètre à un autre composant, celui-ci sera donc également rendu à nouveau, puisque l'une de ses propriétés aura changé.

Si l'âge change, le composant enfant sera render inutilement, puisque la fonction ne dépend que du nom.

useCallback



Pour optimiser cela, React met en place un hook spécifiquement conçu pour les callbacks passées en props à un autre composant :

useCallback

paramètre 1 : la fonction callback

paramètre 2 : un tableau de valeurs qui conditionnent la création de la nouvelle fonction (un peu comme pour useEffect vu plus haut). Si ces valeurs changent, une nouvelle callback va être transmise.

useCallback

```
const [name, setName] = useState('John')
const [age, setAge] = useState(42)
const logName = useCallback(() => console.log(name), [name])
```

En déclarant logName avec un useCallback, on optimise ce comportement.

D'un rendu à l'autre, logName ne sera modifiée que si name est effectivement modifié.

Il est conseillé comme bonne pratique de déclarer toute callback à l'aide de useCallback.

Attention à bien fournir le bon tableau en second argument : toute variable utilisée dans la callback doit être déclarée dans le tableau en second paramètre

Memo

Optimiser les performances

React.memo

React.memo est un composant d'ordre supérieur (nous verrons cette notion un peu plus tard). Il joue le rôle d'un PureComponennt pour des composants fonctionnels.

Lorsqu'une fonction composant est pure, c'est à dire qu'elle renvoie toujours le même résultat pour les mêmes propriétés en entrée, alors elle peut être englobée dans un React.memo, ce qui conserve en mémoire ses résultats et augmente les performances dans certains cas.

React.memo

React.memo ne se préoccupe que des modifications de props. Si la fonction englobée utilise useState, ses changements entraineront un nouveau rendu.

```
function MyComponent(props) {
  /* Faire le rendu en utilisant les props */
}
export default React.memo(MyComponent);
```

React.memo

Il est également possible d'obtenir un rendu similaire (quoi qu'inverse dans sa logique) avec en créant nous même la fonction de comparaison des props pour forcer ou non la mise à jour.

Cette fonction doit être passée en second argument de React.memo et prend en paramètre le précédent tableau de props (avant modif) et le nouveau tableau de props après modif.

Cette fonction renvoie true si les prevProps et les nextProps sont identiques. False dans le cas contraire

React.memo

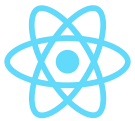
```
function MyComponent(props) {
  /* Faire le rendu en utilisant les props */
}

function areEqual(prevProps, nextProps) {
  /*  Renvoie `true` si passer l'objet nextProps à la fonction de
rendu  produira le même résultat que de lui passer l'objet prevProps.
Renvoie `false` dans le cas contraire.  */
}

export default React.memo(MyComponent, areEqual);
```

Attention cependant, cette méthode n’est qu’un outil d’optimisation des performances. Elle ne doit pas être utilisée pour empêcher un rendu, cela peut causer des bugs.

Ecrire des composants réutilisables

Optimiser les performances

Pourquoi écrire des composants réutilisables ?



L'intérêt de d'écrire des composants réutilisables est multiple :

Des composants :

- plus simples à comprendre
- plus simples à maintenir
- qui favorisent la collaboration
- avec une meilleure qualité du code
- qui font globalement gagner du temps de développement

Principes 1 : simplifier !

Développer une application complexe pourra vous inciter à écrire des composants complexes.

C’est pourtant dans ce cas qu’il sera plus pertinent d’écrire de petits composants, même si ce n’est pas pour les réutiliser.

Cela passe souvent par le fait de scinder les composants qui deviennent trop lourds, y compris de simples composants d'affichage, par exemple

Principes 1 : simplifier !

Considérons le composant suivant

```
const ContactView = ({ contact, editGeneralInfos, editAddress }) => (
  <div>
    <section>
      <h2>General info</h2>
      <p>First name: {contact.firstName}</p>
      <p>Last name: {contact.lastName}</p>
      <button onClick={() => editGeneralInfos()}>Edit</button>
    </section>
    <section>
      <h2>Address</h2>
      <p>
        {contact.address.addressLineOne}<br />
        {contact.address.addressLineTwo}<br />
      </p>
      <button onClick={() => editAddress()}>Edit</button>
    </section>
  </div>
)
```

Principes 1 : simplifier !

On se rend compte qu'au fur et à mesure que ce composant va s'agrandir il va également se complexifier alors qu'il ne sert qu'à afficher des données !

Une première étape serait de créer un composant pour chaque section à afficher

Principes 1 : simplifier !

```
const ContactGeneralInfos = ({ contact, editGeneralInfos }) => (
  <section>
    <h2>General info</h2>
    <p>First name: {contact.firstName}</p>
    <p>Last name: {contact.lastName}</p>
    <button onClick={() => editGeneralInfos()}>Edit</button>
  </section>
)

const ContactAddress = ({ contact, editAddress }) => (
  <section>
    <h2>Address</h2>
    <p>
      {contact.address.addressLineOne}<br />
      {contact.address.addressLineTwo}<br />
    </p>
    <button onClick={() => editAddress()}>Edit</button>
  </section>
)

const ContactView = ({ contact, editGeneralInfos, editAddress }) => (
  <div>
    <ContactGeneralInfos contact={contact} editGeneralInfos={editGeneralInfos} />
    <ContactAddress contact={contact} editAddress={editAddress} />
  </div>
)
```

Principes 2 : homogénéiser

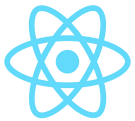
Une seconde idée est derrière le fait de rendre nos composants réutilisable est de rendre l'UI aussi homogène que possible.

Dans l’exemple précédent, on peut aller plus loin en remarquant que les deux sections (général et adresse) partagent du code en commun :

les deux sont affichées au sein d’une balise <section>, et comportent un titre <h2>.

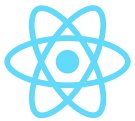
Ce peut donc être une bonne idée de créer un composant pour ces sections :

Principes 2 : homogénéiser



```
const ContactSection = ({ title, children, edit }) => (
  <section>
    <h2>{title}</h2>
    {children}
    <button onClick={() => edit()}>Edit</button>
  </section>
)
const ContactGeneralInfos = ({ contact, editGeneralInfos }) => (
  <ContactSection title="General info" edit={editGeneralInfos} >
    <p>First name: {contact.firstName}</p>
    <p>Last name: {contact.LastName}</p>
  </ContactSection>
)
const ContactAddress = ({ contact, editAddress }) => (
  <ContactSection title="Address" editAddress={editAddress}>
    <p>{contact.address.addressLineOne}</p>
    <p>{contact.address.addressLineTwo}</p>
  </ContactSection>
)
const ContactView = ({ contact, editGeneralInfos, editAddress }) => (
  <div>
    <ContactGeneralInfos contact={contact} editGeneralInfos={editGeneralInfos} />
    <ContactAddress contact={contact} editAddress={editAddress} />
  </div>
)
```

Principes 3 : limiter le state et les effets



Il est parfaitement acceptable qu'un composant gère un state local et des effets de bord.

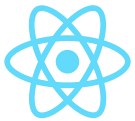
Cependant, toujours en supposant que l'application grossisse et se complexifie, il sera extrêmement bénéfique de limiter le nombre de composants qui en bénéficient.

Un composant sans state local (stateless) et sans effets de bords est beaucoup plus facile à utiliser, maintenir, tester et réutiliser.

Principes 3 : limiter le state et les effets

```
const Contact = (props) => {
  const [contact, setContact] = useState(null)
  useEffect(() => {
    fetch('/api/contacts/' + props.contactId)
      .then(res => res.json())
      .then(contact => setContact(contact))
  }, [props.contactId])
  editGeneralInfos = () => { /*...*/ }
  editAddress = () => { /*...*/ }
  if (contact) {
    return (
      <div>
        <ContactGeneralInfos contact={contact}
          editGeneralInfos={editGeneralInfos} />
        <ContactAddress contact={contact} editAddress={editAddress} />
      </div>
    )
  } else {
    return <span>Loading…</span>
  }
}
```

Principes 3 : limiter le state et les effets



Ce composant a en réalité deux responsabilités bien distinctes :

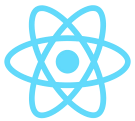
- Récupérer les informations du contact.
- Afficher les informations du contact, ou un placeholder le temps de les récupérer.

Ici, la première chose à faire pour simplifier ce composant est d'extraire l'affichage des informations dans un autre composant (cela tombe bien, c'est justement le rôle de ContactView)

Principes 3 : limiter le state et les effets

```
const Contact = (props) => {
  const [contact, setContact] = useState(null)
  useEffect(() => {
    fetch('/api/contacts/' + props.contactId)
      .then(res => res.json())
      .then(contact => setContact(contact))
  }, [props.contactId])
  editGeneralInfos = () => { /*...*/ }
  editAddress = () => { /*...*/ }
  if (contact) {
    return (
        <ContactView contact={contact}
          editGeneralInfos={editGeneralInfos}
          editAddress={editAddress}
        />
    )
  } else {
    return <span>Loading…</span>
  }
}
```

Principes 3 : limiter le state et les effets

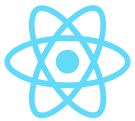


Ce composant comporte toujours un state local et fait toujours une requête HTTP, mais au moins nous avons extrait une partie de ses responsabilités (l'affichage des informations) dans un composant sans état ni effets de bord (ContactView).

Et ContactView peut être utilisé pour afficher des informations d'un contact indépendamment de la manière dont celui-ci a été récupéré.

Higher Order Component

Higher Order Component

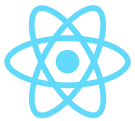


Les composants de haut niveau, plus couramment appelés higher-order components (HOC), sont un moyen particulièrement élégant de créer des composants réutilisables.

Le principe est simple : un HOC est une fonction qui prend en paramètre une définition de composant, et renvoie une nouvelle définition de composant, qui ajoute du comportement à la première.

Il s'agit en fait du pattern Décorateur appliqué aux composants React.

Higher Order Component



L’intérêt d’un HOC est d’extraire un comportement partagé par plusieurs composants dans des fonctions réutilisables.

Les composants de haut niveau sont utilisés par de nombreuses bibliothèques de composants, à commencer par la fameuse React Redux.

Exemple

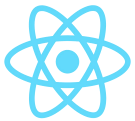
Un exemple simple de HOC

```
const HelloComponent = ({ name, ...otherProps }) =>
  (<div {...otherProps}>Hello {name}!</div>)

const withNameOverride = (BaseComponent) => {
  return ((props) => {
    return <BaseComponent {...props} name="New Name" />
  })
}

const withStyling = BaseComponent => props =>
  <BaseComponent {...props} style={{ color: 'green' }} />

const EnhancedHello1 = withNameOverride(HelloComponent);
const EnhancedHello2 = withStyling(HelloComponent);
```

Higher Order Component

```
const CommentList = () => {
  // `DataSource` est une source de données quelconque
  const [comments, setComments] = useState(null)
  useEffect(() => {
    DataSource.addChangeListener(handleChange);
    return () => {
      DataSource.removeChangeListener(handleChange);
    }
  }, [])
  const handleChange = () => {
    setComments(DataSource.getComments())
  }
  return (
    <>{comments.map((comment) => (
        <Comment comment={comment} key={comment.id} />))}</>
  );
}
```

Higher Order Component



```
const BlogPost = props => {
  const [blogPost, setBlogPosts] = useState(null)
  useEffect(() => {
    DataSource.addChangeListener(handleChange);
    return () => {
      DataSource.removeChangeListener(handleChange);
    }
  }, [])
  const handleChange = () => {
    setBlogPosts(DataSource.getBlogPost(props.id))
  }
  return <TextBlock text={this.state.blogPost} />;
}
```

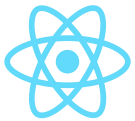
Higher Order Component

CommentList et BlogPost ne sont pas identiques : ils appellent des méthodes différentes sur DataSource, et ont des affichages distincts. Pourtant une grande partie de leur implémentation est la même :

- Au montage, ils ajoutent un écouteur d'événements à DataSource.
- Dans l'écouteur, ils appellent setState quand la source de données est modifiée.
- Au démontage, ils enlèvent l'écouteur d'événements.

Dans une appli importante, ce motif d'abonnement à une DataSource et d'appel à setState sera récurrent. Il nous faut une abstraction qui nous permette de définir cette logique en un seul endroit et de la réutiliser pour de nombreux composants. C'est là que les composants d'ordre supérieur sont particulièrement utiles.

Higher Order Component



Nous pouvons écrire une fonction qui crée des composants qui s'abonnent à une DataSource, comme CommentList et BlogPost.

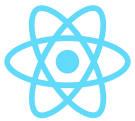
La fonction acceptera parmi ses arguments un composant initial, qui recevra les données suivies en props. Appelons cette fonction withSubscription :

```
const CommentListWithSubscription = withSubscription(
  CommentList,
  (DataSource) => DataSource.getComments()
);

const BlogPostWithSubscription = withSubscription(
  BlogPost,
  (DataSource, props) => DataSource.getBlogPost(props.id)
);
```

Higher Order Component

```
const withSubscription = (WrappedComponent, selectData) => {
  return ((props) => {
    const [data, setData] = useState(null)
    useEffect(() => {
      DataSource.addChangeListener(handleChange);
      return () => {
        DataSource.removeChangeListener(handleChange);
      }
    }, [])
    const handleChange = () => {
      setData(selectData(DataSource, props))
    }
    return <WrappedComponent data={data} {...props} />;
  })
}
```

Higher Order Component



Le premier paramètre est le composant initial. Le second charge les données qui nous intéressent, en fonction de la DataSource et des props existantes.

Lorsque CommentListWithSubscription et BlogPostWithSubscription s'affichent, CommentList et BlogPost reçoivent une prop data qui contient les données les plus récentes issues de la DataSource

Remarquez qu'un HOC ne modifie pas le composant qu'on lui passe. Il compose le composant initial en l'enrobant dans un composant conteneur. Il s'agit d'une fonction pure, sans effets de bord.

Higher Order Component

Le composant enrobé reçoit toutes les props du conteneur ainsi qu'une nouvelle prop, data, qu'il emploie pour produire son résultat.

Le HOC ne se préoccupe pas de savoir comment ou pourquoi les données sont utilisées, et le composant enrobé ne se préoccupe pas de savoir d'où viennent les données.

Higher Order Component

Puisque withSubscription est juste une fonction, vous pouvez lui définir autant ou aussi peu de paramètres que vous le souhaitez.

Comme pour les composants, le rapport entre withSubscription et le composant enrobé se base entièrement sur les props. Ça facilite l'échange d'un HOC pour un autre, du moment qu'ils fournissent les mêmes props au composant enrobé.

Render Props

Render Props

WIP

Conclusion

Conclusion

Nous en avons terminé avec l'exploration des possibilités offertes par React en standalone, c’est-à-dire sans autre bibliothèque ou presque.

Les connaissances acquises dans ces chapitres vous permettront déjà de développer vos premières applications web à l’aide de React.

Dans la prochaine partie, nous verrons comment router, structurer et tester React avec des librairies complémentaires.

Routing

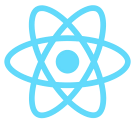
Avec React Router

Routing ?

Réact permet de créer des Single Page Applications : une seule page est chargée et l'URL ne change jamais. Mais cela ne veut pas dire que le routing n'y est pas possible pour autant :

- Le Routage de React va nous permettre de :
- mettre à jour l'URL en fonction de la navigation
- pouvoir accéder à un composant via une URL
- naviguer dans l'historique (précédent/suivant)
- pouvoir mettre des pages de notre app en favori
- ... tout en restant sur une SPA

Routing



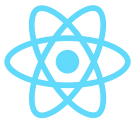
React n'intègre pas de fonctionnalités de routage par défaut. Cependant il existe des librairies tierces qui permettent de le gérer aisément. C'est le cas de React Router Dom, la biblothèque de routing la plus utilisée :

Pour l'installer :

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

Doc complète : https://reacttraining.com/react-router/web/guides/quick-start

Routing : Utilisation basique



React Router s'utilise de manière assez simple. On dispose de 3 grands types de composants :

- De routing <BrowserRouter> et <HashRouter>
- De définition de routes <Route> et <Switch>
- De navigation <Link> <NavLink> et <Redirect>

Nous allons les étudier en détails

Composants Routeurs

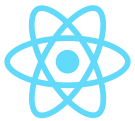
Souvent renommé «Router» lors de l'import, il permet de définir quelle portion de l'app est concernée par le routage. La plupart du temps, c'est toute l'application qui est englobée dans le Router.

- **<BrowserRouter>** utilise des chemins d'accès classiques : http://www.exemple.com/ma/page

Cependant cela nécessite que le serveur serve la même page systématiquement à toute les URLs

- **<HashRouter>** sauvegarde l'emplacement actuel dans une portion hashée de l'URL de type : http://www.exemple.com/#/ma/page

Routes



Un compsant Route (ou Route Matcher) permet de déclarer quel composant s'affiche pour quelle route. Il en existe deux, qui fonctionnent de concert

<Switch> qui, qui permet de déclarer que un bloc de routage

<Route> qui, au sein du bloc **<Switch>** permet de déclarer un composant sur un chemin en particulier

Routes

- Route prend un paramètre **«path»** qui est le chemin du composant en question.
- Pour déclarer le composant, il est possible de l'englober dans une **<Route>** ou de le déclarer en paramètre component
- Le Switch va s'arrêter sur la première Route qui matche avec le début de l'URL demandée. Par exemple **path='/'** matche toutes les URL. Les routes vont donc être organnisées de la plus spécifiques à la plus générale, ou avec un attribut **«exact»**

Navigation

La navigation est géré par des liens **<Link>** (pour un lien classique) et **<NavLink>** (pour un lien de navigation type menu). La seule différence est que NavLink possède une propriété «activeClassName» qui permet d'ajouter du CSS quand on matche la route

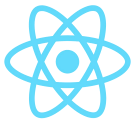
Et les redirections avec **<Redirect>** qui force une redirection lorsqu'il est rendu.

Tous les composants de navigation possèdent un attribut «to» qui indique la route vers laquelle ils mènent.

Assemblons tout ça !

```
import {
    BrowserRouter as Router,
    Route,
    Link } from "react-router-dom";
function App() {
  return (
    <Router>
      <Switch>
        <Route path="/" exact><NewsFeed /></Route>
        <Route path="/profile/:id" component={UserProfile} />
        <Route path="/profile" component={LoggedUserProfile} />
      </Switch>
      <ul>
        <li><Link to="/">Home</Link></li>
        <li><NavLink to="/profile">My profile</NavLink></li>
     </ul>
    </Router>
  )
}
```

Paramètres de navigation



Les paramètres de navigation sont des emplacements dans l'URL qui commencent avec :

exemple : /profile/:id

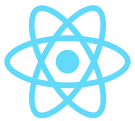
Pour accéder aux paramètres de l'URL dans le composant appelé ainsi, il faut utiliser le hook spécifique à React Router : useParams() avec du destructuring

exemple :

```
const { id } = useParams();
```

Redux

State Management

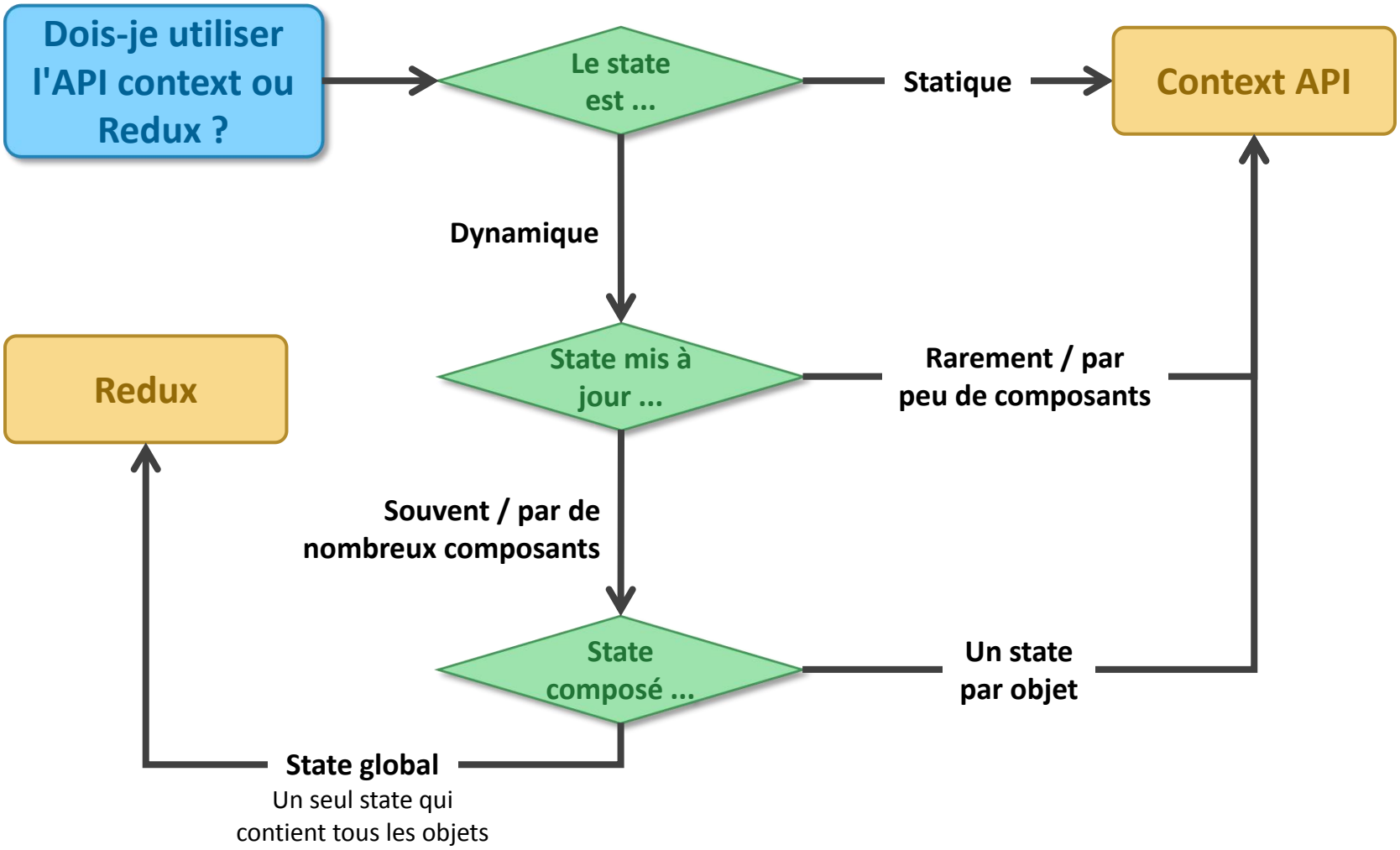


Nous avons vu les bases de React (composants, state, props...)

Nous avons également vu comment passer un state dans le contexte pour le rendre accessible partout, évitant ainsi de devoir passer toutes nos données dans les props de tout notre arbre de composant.

Cependant, l'API contexte, bien que fort pratique possède certaines limitations en termes de performances et reste assez récente (donc pas nécessairement utilisée par tous les projets).

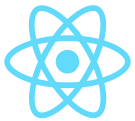
Contexte vs Redux



Redux est une librairie de gestion d'état de manière prévisible, créé par Dan Abramov pour les applications JavaScript.

Redux permet de stocker l’état d’une application, tout en donnant les moyens d’y accéder de manière globale, et de le mettre à jour.

- https://redux.js.org
- Créé en 2013 par Facebook, inspiré de XHP
- Focalisé sur la gestion des vues
- Orienté composant

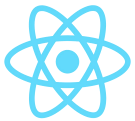


Elle est basée sur le concept de circulation unidirectionel de données, popularisé par l'équipe Facebook avec son architecture Flux.

Il n’est pas spécifique à React et donc peut être utilisée avec d'autres frameworks JS.

C’est la bibliothèque React-Redux qui fournit les éléments permettant de faire travailler React et Redux en collaboration.

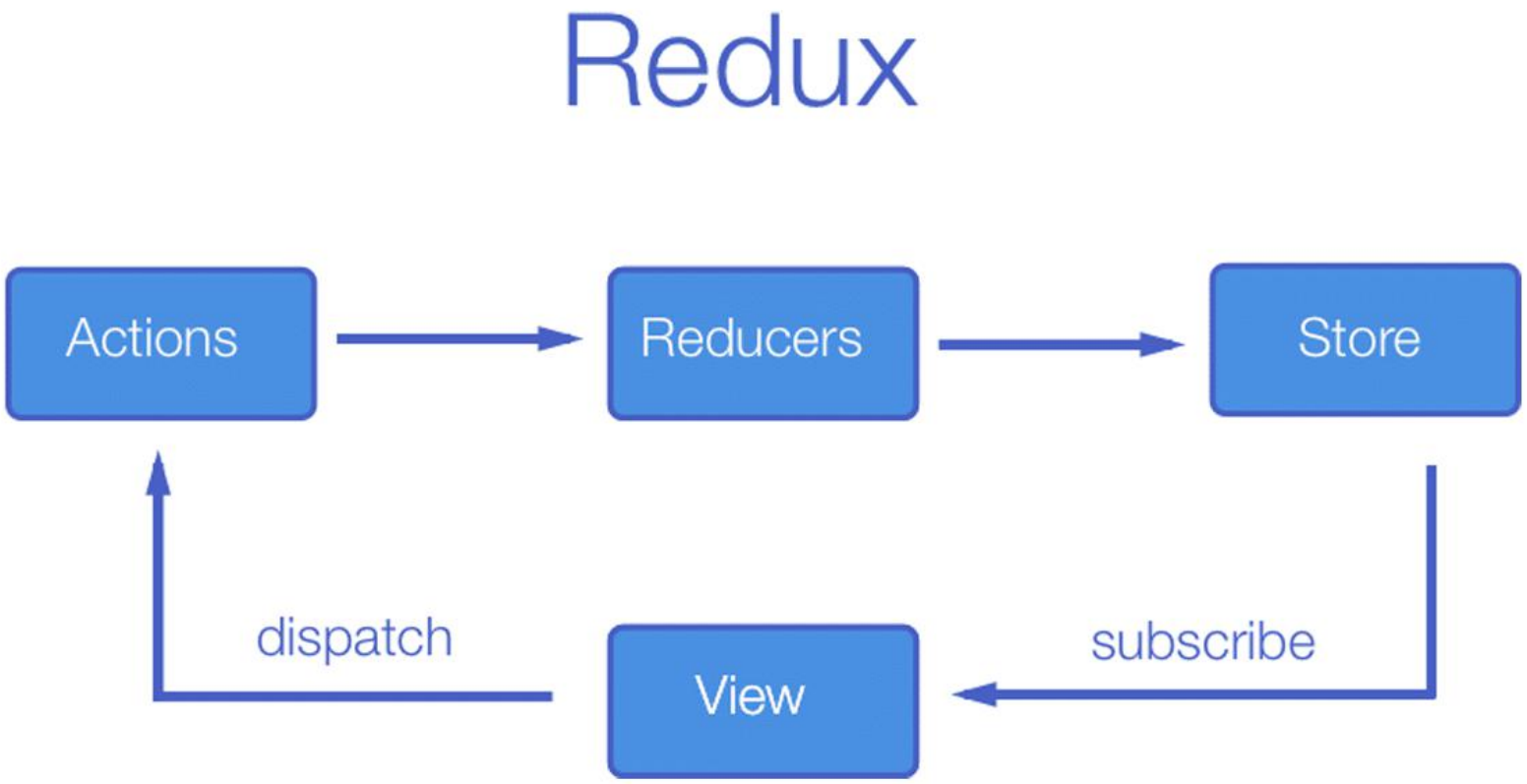
Unidirectional Data Flow



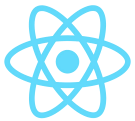
L'architecture Flux repose sur l'idée d'un flux de données unidirectionnel strict. On ne peut affecter les données qui y transitent qu'en suivant un sens précis (on ne le court-circuite pas).

Redux implémente cette architecture avec un vocabulaire qui est lui est propre mais le principe est bien le même.

Unidirectional Data Flow

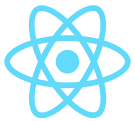


Unidirectional Data Flow



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

State



L’élément le plus important de Redux est le state. C’est en effet sa principale fonction : stocker l’état d’une application.

Les données qu’il stocke peuvent être de tout type : objets, nombres, tableaux, fonctions, etc.

Le state peut être lu (par un composant React ou autre); en revanche, il ne sera jamais modifié directement (readonly). Nous allons devoir indiquer à Redux comment en générer la version suivante du state.

Actions

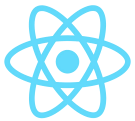
Afin de mettre à jour le state, la première étape sera de déclencher une action, par exemple au clic sur un bouton. On dira alors que l'action est **dispatchée**.

Une action est un **objet**, constitué :

- d'un type (le nom de l'action)
- éventuellement d'autres données (souvent appelées payload)

L'idéal est de concevoir une action comme un verbe, associé à des paramètres. Par exemple, si le state contient un compteur, on pourrait avoir les actions **incrémenter**, **décrémenter** et **définirValeur(valeur)**

Reducer

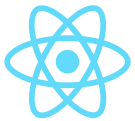


Le reducer est une **fonction**, prenant en paramètres un **state** et une **action**, et renvoyant un **nouveau state**.

Par exemple, pour un compteur, à partir d'un state valant 5 et d'une action incrémenter, nous renverrions comme nouveau state la valeur 6.

Le reducer ne change pas directement le state, mais indique à Redux la nouvelle version du state en fonction de l'action qui a été dispatchée.

Reducer

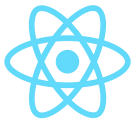


Le reducer doit être une **fonction pure**, au sens défini par la programmation fonctionnelle : Leurs logiques doivent être prédictibles.

Cela implique un certain nombre de limitations que nous allons voir.

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

Reducer



Contraintes des reducers :

Pour un ensemble de paramètres donnés (action, payload, state initial), on doit toujours retourner le même state

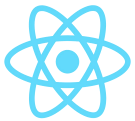
En pratique cela veut dire que, nous devrons pas avoir dans le reducer une logique qui dépend du moment présent. On écrira plutôt cette logique dans un action creator.

Reducer

Contraintes des reducers :

Un reducer ne doit pas avoir d’effets de bord modifier un état qui lui est extérieur : ne pas modifier d'autres variables, ne pas dispatcher d'autres actions, ne pas déclencher de traitement asynchrone ...

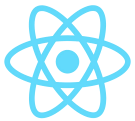
Reducers



Contraintes des reducers :

Le store retourné doit toujours être un nouvel objet car Redux effectue une vérification (comparaison d'égalité stricte) entre l'ancien et le nouveau state pour mettre à jour la vue seulement quand il y a une modification.

Store



Le store n’est autre que l’objet unifiant les notions que nous venons de voir.

Il est initialisé au démarrage, en lui fournissant un reducer et un state initial. Il est ensuite possible de

- lire le state courant avec **getState**
- dispatcher des actions avec **dispatch**
- souscrire aux changements du state avec **subscribe**

Note : un store ne prend qu'un seul reducer, mais il est possible de combiner plusieurs reducers en un seul, comme nous le verrons plus tard.

Redux

Installation

Redux : Installation

On installe la dépendance Redux

- npm install redux

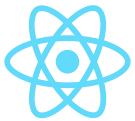
Malgré le fait que Redux n'a aucune dépendance avec ReactJS, il y a une librairie officielle (crée par l'équipe Redux) qui harmonise les deux et permettent ainsi d'écrire moins de code, que nous allons également utiliser:

- npm install react-redux

Redux

Premier exemple

Redux - Premier exemple



Pour notre premier exemple, nous allons commencer avec un compteur (évidemment ...)

L'idée ici est d'aborder et de mettre en oeuvre les concepts de base de Redux (store, actions, reducers...) sans se soucier réellement de l'implémentation dans l'architecture React, de react-redux.

Pour rester simple, nous allons centraliser tout le code au sein de notre App.js

Redux - Premier exemple

Commençons par importer redux, et plus particulièrement la méthode createStore qui nous permet de créer un store

```
import { createStore } from 'redux'
```

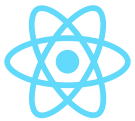
Nous allons également remplir notre composant App avec un state compteur, une fonction handler (qui ne fait rien pour l'instant), un compteur (qui affiche notre state compteur) et un bouton qui déclenche notre fonction handler.

Redux - Premier exemple

```
function App() {
  const [counter, setCounter] = useState()
  const handleIncrement = () => {}
  return (
    <>
      <span>{counter}</span>
      <button onClick={handleIncrement}>+</button>
    </>
  )
}
```

Et ce sera tout pour App pour l'instant. Le reste va se passer en dehors de notre composant (mais dans le même fichier)

Redux - Premier exemple



Nous allons maintenant mettre Redux en place à proprement parler :

Nous avons besoin de notre action creator (increment)

Comme nous l'avons vu plus haut, une action est un objet qui contient : son type et un payload optionnel

Pour nous aider, nous aurons besoin d'un **action créator**, une fonction qui retourne notre action.

```
const increment = () => ({type: 'INCREMENT'})
```

Redux - Premier exemple

Au tour du reducer. Pour rappel, il doit prendre en paramètre un state et une action pour retourner le nouveau state. Un reducer simple pourrait ressembler à ça :

```
const reducer = (state = 0, action) => {
  if (action.type === 'INCREMENT') {
    state.counter += 1
    return state
  }
}
```

Nous allons quand-même l'améliorer.

Redux - Premier exemple

Quelques bonnes pratiques :

1. Le state courant n'est pas défini au premier appel du reducer. Il faut donc utiliser une valeur par défaut pour initialiser le state à ce moment.

2. Il est donc intéressant de définir un state initial. Pour l'instant nous n'avons que notre compteur, mais nous aurons à stocker d'autres informations plus tard.

Redux - Premier exemple

Si l’action ne correspond à aucun type connu, nous devons tout de même renvoyer un state, nous renvoyons donc le state courant sans modification. En effet Redux commence par dispatcher des actions internes au démarrage.

Structurer un reducer avec un bloc switch est la manière de faire la plus élégante et la plus courante.

Redux - Premier exemple

Utiliser la décomposition pour le state permet d’une part de s’assurer qu’on ne modifie pas le state actuel, et d’autre part de ne mettre à jour que les attributs qui nous intéressent ici. (Nous n'avons qu'un seul attribut pour l'instant, mais ça ne sera pas toujous le cas, évidemment !)

Redux - Premier exemple

Ce qui nous donne donc :

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

Par ailleurs, remarquez que cette fonction est pure : nous ne modifions pas de valeur extérieure à la fonction (notamment le state), et elle n'entraîne pas d'effet de bord.

Redux - Premier exemple

À présent, nous pouvons créer notre store avec la fonction createStore de Redux en lui passant en paramètre notre reducer :

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

Il est également possible d'utiliser simplement Redux DevTools extension en lui passant un middleware en paramètre :

```
const store = createStore(
  reducer,
  window.__REDUX_DEVTOOLS_EXTENSION__ && window.__REDUX_DEVTOOLS_EXTENSION__()
)
```

Redux - Premier exemple

Le store étant créé, nous pouvons à présent faire trois choses :

- Lire son state grâce à store.getState();
- Mettre à jour son state en dispatchant des actions grâce à store.dispatch(...)
- Souscrire aux changements de state avec store.subscribe(...)

Nous allons voir comment mettre ces 3 actions en place dans notre exemple.

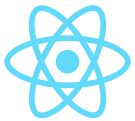
Redux - Premier exemple

Notre fonction handler est le meilleur endroit pour pouvoir dispatcher une action. Elle appelée dès que le bouton est cliqué. Dans cette fonction, nous dispatchons une action de type INCREMENT grâce à l'action creator increment()

```
handleIncrement = () => {
  store.dispatch(increment())
}
```

Notez que la mise à jour de notre compteur ne se fait pas ici

Redux - Premier exemple



Le principe de flux unidirectionnel fait que ça n'est plus à notre composant de décider quand mettre à jour son state. Cette tâche incombe désormais au store !

store.subscribe() nous permet de nous abonner aux changements du store. La callback passée en paramètre est déclenchée lorsque le store subit un changement.

En l'occurence, nous voulons récupérer le counter dans le sotre et l'affecter au state de notre composant

Redux - Premier exemple

Ce qui nous donne donc :

```
store.subscribe(() => {
    setCounter(store.getState().counter)
})
```

C'est un peu verbeux, il y a beaucoup de boilerplate, et c'est assez mal organnisé (pour l'instant). Mais ça a le mérite de fonctionner et d'être assez simple à comprendre.

Maintenant essayons d'améliorer cela.

Redux avec React

Découvert de React-Redux

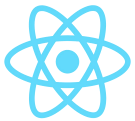
Redux avec React

L’exemple précédent était laborieux : beaucoup d’opérations pour un résultat modeste.

Mais c’est en le couplant avec React que tout son potentiel est visible.

Il va nous permettre d’extraire un maximum de logique métier (mise à jour du state, appels à une API, etc.) hors des composants, pour n’y conserver que leur responsabilité originale : le rendu.

Redux avec React



La bibliothèque React-Redux est là pour nous faciliter grandement la tâche. Nous n'aurons pas besoin de nous soucier du store, car elle nous permet de définir composant par composat :

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

Redux avec React

Créons cette fois-ci un composant spécifique pour notre compteur et appelons le dans App.

Pour l'instant notre composant ne connait ni counter, ni handleIncrement. Mais ça va venir.

```
const Counter = () => {
   return (
    <>
      <span>{counter}</span>
      <button onClick={handleIncrement}>+</button>
    </>
)}
```

Redux avec React

Et maintenant, un peu de rangement...

Pour ma part j'aime bien créer un dossier store avec l'arborescence suivante :

```
store/
├── actions.js
├── reducers.js
└── index.js
```

Redux avec React

Et le contenu de ces fichiers est le suivant :

```
// actions.js
const actions = {
  increment: () => ({ type: 'INCREMENT' })
}
export default actions
```

```
// reducers.js
const initialState = { counter: 0 }
const reducer = (state = initialState, action) => {
  switch (action.type) {
    /* la logique reste inchangée ici */
  }
}
export default reducer
```

```
// index.js
import { createStore } from 'redux'
import reducer from './reducers'
const store = createStore(reducer)
export default store
```

Redux avec React

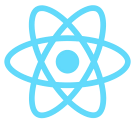
Notez que nous avons rendu notre action réutilisable en en faisant une propriété d'objet. Il nous sera désormais possible de créer d'autres actions.

Par ailleurs, nous exportons les actions pour les rendre disponnibles dans nos compsants.

Nous exportons également le store pour pouvoir le connecter à React-Redux.

Maintenant, où souhaitons-nous faire intervenir Redux ?

Redux avec React



Nous souhaiterions que le store soit accessible depuis le plus haut niveau de notre application pour être accessible de tous les composants qui l'appellent.

Ensuite, nous pouvoir faire appel à notre composant sans lui passer les propriétés counter ou increment. De cette façon :

<Counter />

Redux avec React

Pour cela, React-Redux nous offre un composant appelé Provider, auquel nous passons en propriété le store, et dans lequel nous englobons toute notre application

Il faut donc modifier index.js pour qu'il transmette notre store à toute notre application :

- Importer le store et Provider
- Englober notre app dans le Provider
- Passer le store au provider

Redux avec React

```
import { Provider } from 'react-redux'
import store from './store'

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

Cette notation semble très similaire aux contextes. C'est normal. En interne, React-Redux utilise bien l'API context de React

Redux avec React

Maintenant pour récupérer notre state et nos actions dans nos composants, React-Redux met à disposition deux hooks custom particulièrement utiles :

- **useSelector** prend en paramètre un sélecteur, (une fonction) qui, à partir du state, renvoie la valeur que l'on souhaite. Cette valeur est à son tour renvoyée par le hook.
- **useDispatch** renvoie simplement une fonction permettant de dispatcher une action.

Redux avec React

```
const Counter = () => {
  const counter = useSelector(state => state.counter)
  const dispatch = useDispatch()
  const handleIncrement = useCallback(
    () => dispatch(actions.increment()),
    [dispatch]
  )
   return (
    <>
      <span>{counter}</span>
      <button onClick={handleIncrement}>+</button>
    </>
  )
}
```

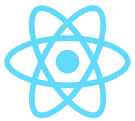
Redux avec React

- Tout d’abord, nous récupérons la valeur de l’attribut counter du state en appelant useSelector
- Puis nous créons une fonction dispatch en appelant useDispatch.
- Notez l'utilisation du hook useCallback, utilisé pour créer des fonctions qui seront passées en propriété à un composant, et éviter des nouveaux rendus inutiles.
- Dans le callback increment ainsi créé, nous dispatchons une action increment.

Redux Saga

Actions complexes et asynchrones

Redux saga



Jusque-là, notre manière d'utiliser Redux est restée relativement simple :

- dans le composant nous dispatchons une action ;
- l'action passe dans le reducer, qui met à jour le state ;
- le composant est réaffiché pour tenir compte du nouveau state.

Si ce processus convient déjà à beaucoup de besoins, certains vont nous poser quelques problèmes : comment peut-on indiquer qu'une action doit dispatcher une autre action (effets de bord) ? Comment peut-on faire en sorte qu'une action déclenche une requête HTTP (asynchrone) ?

Redux Saga

Redux Saga permet de palier à ces deux problématiques

https://github.com/redux-saga/redux-saga

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

Redux Saga

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

Voyons dans un premier temps ce qui se cache derrière les générateurs, après quoi nous verrons en quoi ils sont utiles à la compréhension et à l’utilisation des sagas.

Les générateurs

Dit simplement, un générateur est une fonction renvoyant un itérateur. Commençons donc par présenter ce qu'est un itérateur. Il s'agit d'un objet possédant une interface permettant de :

- connaître sa valeur courante ;
- se déplacer à la valeur suivante ;
- savoir si l'on a atteint la fin de l'itérateur

Plusieurs implémentations sont possibles ; voici par exemple une fonction permettant, à partir d'un tableau donné, d'obtenir un itérateur pour le parcourir :

```
const getArrayIterator = array => {
  let index = -1
  return {
    next() {
      index++
      return {
        value: array[index],
        done: index === array.length,
      }
    }
  }
}
```

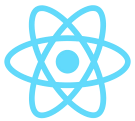
Ici, les trois actions propres à l'itérateur sont faites au sein d'une seule fonction next, qui renvoie un objet contenant l'attribut value avec la valeur courante et l'attribut done qui est à true si on est au bout.

La manière la plus naturelle d'utiliser cet itérateur est par une boucle while :

```
const it = getArrayIterator()
let res
do {
  res = it.next()
  console.log(res)
} while (!res.done)
```

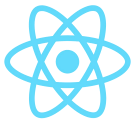
La fonction getArrayIterator renvoie un nouvel itérateur. JavaScript nous offre donc la possibilité de les définir d'une manière différente, sous forme de générateur, grâce au mot-clé function* :

```
function* getArrayIterator(array) {
  let index = 0
  while (index !== array.length) {
    yield array[index]
    index++
  }
}
```



Cette fonction génératrice a exactement le même comportement que celle définie précédemment. Elle renvoie des itérateurs qui peuvent être parcourus exactement de la même manière. Et pourtant sa syntaxe peut paraître déroutante.

Le fait d'utiliser une boucle while donne l'impression que l'on va parcourir tout le tableau. En réalité, il n'en est rien, car l'utilisation du mot-clé yield stoppe en quelque sorte l'itération courante, et se met en attente de la suivante



Pour mieux comprendre les générateurs, on peut donc imaginer qu'une instruction yield retourne une valeur, stoppe la fonction, et fait en sorte qu'au prochain appel on reprenne l'exécution à cet endroit exact, là où l'on s'était arrêté.

- WIP

Tester React

Que tester ?

Sur une application front-end, la question se pose régulièrement de savoir quoi tester unitairement.

En effet, si l'on prend par exemple un composant qui n'est responsable que de présentation en générant du HTML, il serait laborieux de tester précisément le rendu du composant dans le navigateur.

De plus, il est possible que le moindre changement dans le CSS associé au composant change le rendu au point de mettre le test en échec, ce qui n'est généralement pas le but.

Que tester ?

Par certains aspects, il est tout de même intéressant de tester quelques composants :

- pour tester les informations affichées ;
- pour tester le comportement du composant en réponse à des actions de l'utilisateur.

Par ailleurs, si votre application utilise Redux, il est possible de tester totalement le store en fonction des actions dispatchées, et ainsi tester la logique métier de l'application.

Tester React

Pour pouvoir découvrir les tests unitaires de composants React, commençons par créer le composant principal de notre application : un formulaire avec un seul champ et un bouton.

Le champ nom sera initialisé avec une propriété contact.name

S'il est vide, un message d'erreur devra s'afficher

Tester React

```
const ContactForm = ({ contact, updateContact }) => {
  const [name, setName] = useState(contact.name)
  const onChangeName = useCallback(
    event => setName(event.target.value),
    [setName]
  )
  const onSubmit = useCallback(event => {
    event.preventDefault()
    if (name !== '') {
      updateContact({ name })
    }
  })
  return (
    <form onSubmit={onSubmit}>
    <label>Name:
    <input name="name" value={name} onChange={onChangeName} /><br />
    {name === '' && <span className="error">Please enter a name.</span>}
    </label><br />
    <button type="submit">OK</button>
    </form>
  )
}
```

Tester React

Le nom en cours d'édition est stocké dans un état local grâce au hook useState, initialisé avec le contact donné en propriété, et la soumission du formulaire, après vérification que le nom saisi n'est pas vide, appelle la fonction updateContact.

Pour lancer l'application avec ce composant et le tester manuellement, faisons en sorte qu'il soit affiché au chargement de l'application (dans src/index.js) :

```
<ContactForm contact={{ name: 'John Doe' }} updateContact={console.log} />
```

Tester React

Pour lancer les tests, nous utiliserons Jest (https://jestjs.io), qui s'est imposé comme un standard pour tester des applications React, bien qu'il soit possible d'utiliser n'importe quel outil de test JavaScript.

Jest nous fournira plusieurs choses :

- Le moyen de lancer nos tests (via la commande jest) et de les organiser au sein de blocs describe et du cas de test it.
- Les fonctions pour effectuer les assertions, par exemple pour vérifier qu'une valeur obtenue est bien conforme à celle attendue
- Et enfin le moyen de créer facilement des mocks de fonctions, et de vérifier qu'ils ont été appelés et avec quels paramètres.

Tester React

Associé à Jest, nous aurons également besoin d'un outil permettant de faire le rendu d'un composant React (celui que nous souhaitons tester). Il existe plusieurs solutions pour cela, et plusieurs outils. Citons par exemple

- ReactTestUtils (https://reactjs.org/docs/test-utils.html), intégré à React mais désormais déprécié au profit des deux autres librairies
- React Testing Library (https://testing-library.com/docs/react-testing-library/intro#_docusaurus)
- Enzyme d'AirBnB (https://airbnb.io/enzyme/).

Tester React

Enzyme a été créée par AirBnb, et présente la particularité de proposer deux modes de fonctionnement :

- Le **shallow rendering** qui permet de monter un composant avec un seul niveau de hiérarchie. C'est-à-dire que s'il fait appel à des composants enfants ceux-ci ne seront pas rendus. Cela permet de vérifier leurs propriétés
- Le **full DOM rendering**, effectuant le rendu du composant dans un DOM virtuel de la même manière que dans un navigateur, c'est-à-dire en affichant toute l'arborescence du composant et en permettant l'accès aux API du DOM.

Tester React

Aucun mode n’est meilleur que l’autre dans l’absolu. Pour notre usage, c’est-à-dire tester unitairement un composant, en isolation totale des autres composants, le premier mode nous conviendra davantage.

Commençons par installer Jest, Enzyme, ainsi que quelques dépendances nécessaires :

WIP

Tester React

Par convention, les tests se trouvent dans des fichiers dont le nom se termine par « .test.js ». Ainsi pour tester notre composant ContactForm, créons un fichier ContactForm.test.js situé à côté.

Pour organiser nos tests, Jest nous permet d'utiliser deux types de blocs :

- it (ou test) permet de définir un cas de test.
- describe permet de grouper plusieurs cas de tests, ainsi que de définir des actions à effectuer avant chaque test

Tester React

Un fichier de test aura donc par exemple la structure suivante :

```
describe('maFonction', () => {
  it('renvoie 0 si aucun paramètre n'est passé', () => {/* ... */})
  it('renvoie 1 si un paramètre est passé', () => {/* ... */})
})
```

Au sein d'un bloc describe, il est aussi possible d'avoir des blocs beforeAll, beforeEach, afterAll et afterEach, permettant comme leur nom l'indique d'effectuer respectivement des actions avant tous les tests, avant chaque test, après tous les tests et après chaque test.

Tester React

Initialisons à présent notre fichier de test avec un premier cas de test que nous allons remplir au fur et à mesure :

```
describe('ContactForm', () => {
  it('displays the name in the input', () => {
    /* ... */
  })
})
```

La première étape de notre test va être d'effectuer le rendu (en mode shallow rendering) de notre composant. Pour cela, nous utiliserons la fonction shallow fournie par Enzyme.

Tester React

La première étape de notre test va être d'effectuer le rendu (en mode shallow rendering) de notre composant. Pour cela, nous utiliserons la fonction shallow fournie par Enzyme.

```
const wrapper = shallow(
  <ContactForm
    contact={{ name: 'John Doe' }}
    updateContact={() => { }}
  />
)
```

Tester React

L'objet renvoyé, appelé wrapper dans la terminologie d'Enzyme, propose plusieurs méthodes pour accéder au contenu du rendu, c'est-à-dire les composants générés. Pour notre test, nous souhaitons récupérer le champ de saisie (input) et vérifier que sa valeur est bien le nom du contact que l'on a passé en propriété :

```
expect(wrapper.find('input').get(0).props.value).toEqual('John Doe')
```

Tester React

La méthode wrapper.find prend en paramètre un sélecteur de type CSS semblable à ceux utilisés par document.querySelector par exemple ('tag', '.classe', '#id', etc...)

Elle peut aussi prendre directement une définition de composant comme nous le verrons plus loin.

Elle renvoie potentiellement plusieurs éléments, d'où l'utilisation de .get(0) pour récupérer dans notre cas l'unique champ input affiché.

Tester React

La prochaine étape est de vérifier que lorsque l'on tape un nouveau nom dans le champ, celui-ci est bien affiché.

Pour simuler une action de l'utilisateur, le wrapper renvoyé par shallow propose la méthode simulate, qui prend deux paramètres : le nom de l'évènement à déclencher (« change » dans notre cas) et l'évènement lui-même.

Tester React

```
it('updates the name when updating input value', () => {
  const wrapper = shallow(
    <ContactForm
      contact={{ name: 'John Doe' }}
      updateContact={() => { }}
    />
  )
  wrapper.find('input').simulate('change',
    { target: { value: 'Jane Doe' } }
  )
  expect(
      wrapper.find('input').get(0).props.value
  ).toEqual('Jane Doe')
})
```

Tester React

On commence à remarquer que ces deux premiers tests ont beaucoup de code en commun. Essayons déjà de mutualiser certaines parties. Par exemple, le wrapper sera toujours généré de la même manière. On peut donc le définir avant chaque test, dans un bloc beforeEach :

```
let wrapper
beforeEach(() => {
  wrapper = shallow(
    <ContactForm contact={{ name: 'John Doe' }}
    updateContact={() => { }} />
  )
})
```

Tester React

De même, le champ input sera toujours récupéré de la même manière, on peut donc créer une fonction pour simplifier cela :

```
const getInput = () => wrapper.find('input')
```

Tester React

Comme troisième cas de test, vérifions que l'erreur est affichée si et seulement si le champ de saisie est vide. On peut ensuite vérifier que l'erreur n'est pas affichée au chargement, mais affichée lorsqu'on vide le champ :

```
const getError = () => wrapper.find('.error')
it('displays the error message when input is empty', () => {
  expect(getError().length).toEqual(0)
  getInput().simulate('change', { target: { value: '' } })
  expect(getError().length).toEqual(1)
})
```

Tester React

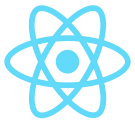
Afin de vérifier ensuite que la soumission du formulaire entraîne l'appel de la fonction updateContact, il nous faut définir cette fonction. Pour cela, Jest propose grâce à jest.fn() de créer des fonctions qui peuvent être espionnées, c'est-à-dire qu'il est possible de savoir si elles ont été appelées, et avec quels paramètres.

Définissons donc la fonction et modifions la création du composant dans le beforeEach pour lui injecter cette fonction en propriété updateContact :

Tester React

```
let updateContact
beforeEach(() => {
  updateContact = jest.fn()
  wrapper = shallow(
    <ContactForm
      contact={{ name: 'John Doe' }}
      updateContact={updateContact}
    />,
  )
})

const getForm = () => wrapper.find('form')
it('calls updateContact on form submit', () => {
  getInput().simulate('change', { target: { value: 'Jane Doe' } } )
  getForm().simulate('submit', { preventDefault() { } })
  expect(updateContact).toHaveBeenCalledWith({ name: 'Jane Doe' })
})
```

Tester React



Deux choses sont importantes à noter ici.

Premièrement, nous ne simulons pas un clic sur le bouton, mais directement la soumission du formulaire lui-même. En effet, le shallow rendering d'Enzyme ne propage pas les évènements comme ils le seraient dans le DOM. Un clic sur le bouton n'aurait donc dans notre cas aucun effet, puisque nous n'avons pas défini de propriété onClick dessus.

De plus, on fournit un objet évènement contenant une méthode preventDefault qui ne fait rien. Comme dans notre composant nous appelons event.preventDefault() à la soumission, une erreur serait déclenchée si nous ne fournissions pas cette méthode.

Tester React

Le dernier cas de test à implémenter est relativement similaire, puisqu'il s'agit de vérifier que la fonction updateContact n'est pas appelée si le champ de saisie est vide :

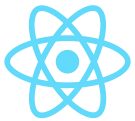
```
it('doesn\'t call updateContact if input is empty', () => {
  getInput().simulate('change', { target: { value: '' } })
  getForm().simulate('submit', { preventDefault() { } })
  expect(updateContact).not.toHaveBeenCalled()
})
```

Notre test est complet, et vérifie l'ensemble des spécifications imposées.

Bibliographie

C'est déjà fini ?

Ressources



- https://reactjs.org
- https://www.reddit.com/r/ReactJS
- Tutoriels : https://dev.to/t/react