

Tutorial React

Introduction

« React » (appelée couramment « ReactJS » pour la différencier de « ReactNative ») est une bibliothèque « JavaScript » libre développée par « Facebook ». L'objectif est de faciliter le développement d'application web en se basant sur le mécanisme de composants. Un composant est une *brique logicielle*, l'idée est qu'il est (autant que possible) indépendant du reste et peut être réutilisé dans un autre contexte.

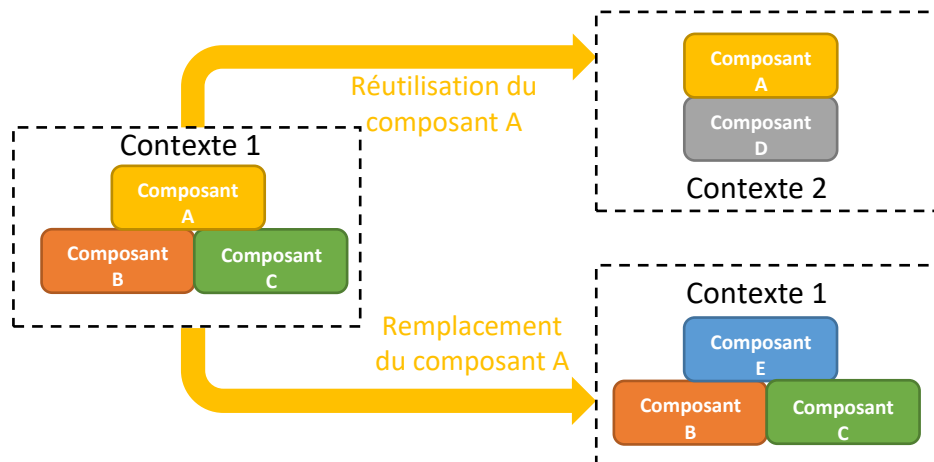


Figure 1 - Développement par composant

Durant ce tutorial je ferais régulièrement des comparaisons avec « Angular2+ » (au niveau des notions) qui est un « framework » « JavaScript » que j'ai plus fréquemment utilisé. Un *Référencement des avantages et des inconvénients* sert de sommaire à ce rapport.

Structure

« React » est une bibliothèque, elle offre donc un ensemble d'outils pour faciliter et améliorer le développement mais n'impose pas de structure. C'est un point essentiel à prendre en compte lors du choix des technologies, cela nous offre une bien plus grande liberté mais en contrepartie le développeur est bien plus livré à lui-même. Cela dit à la création d'une nouvelle application « React » on dispose de l'arborescence suivante :

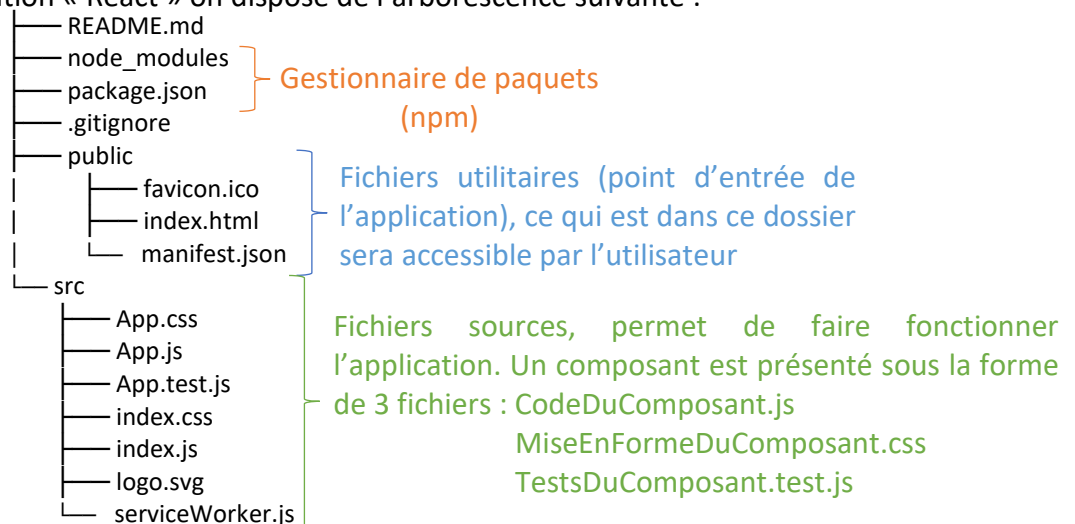


Figure 2 - Architecture proposée par "create-react-app"

J'ai décidé d'organiser mon application de la manière suivante, de sorte à la garder la plus maintenable possible :

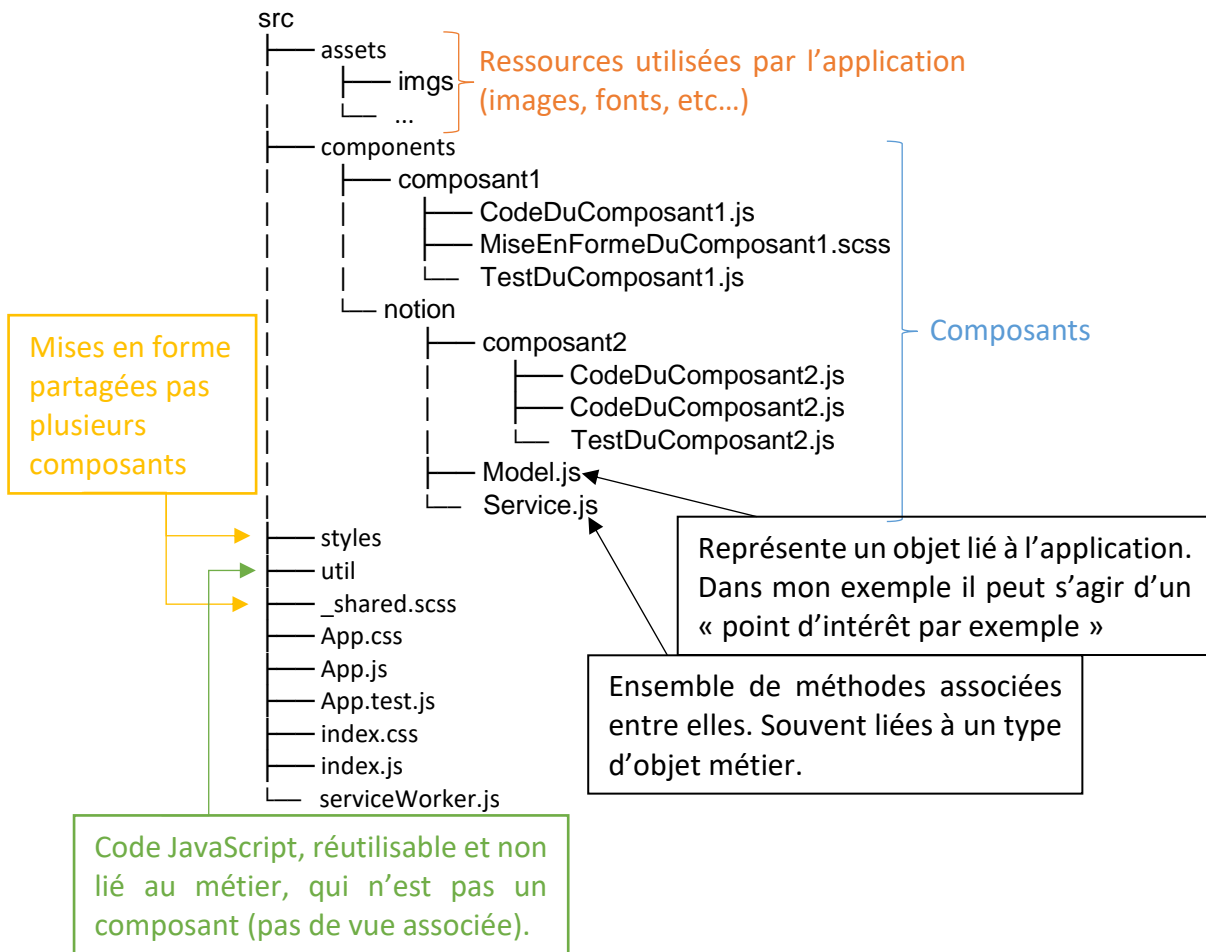


Figure 3 - Architecture adoptée

Je n'ai pas sorti cette architecture de nulle part, elle est très proche de ce qu'on rencontre sur « Android » ou celle d'un projet « Angular2+ ». Je pense que, ne pas freiner l'utilisateur en lui laissant le choix de la manière d'organiser son projet, est un bon point de « React » mais seulement si ce n'est pas la première technologie fréquentée par ce dernier. Sans convention élaborée par les membres du projet il y a de fortes chances pour que l'application devienne un véritable plat de spaghettis impossible à maintenir (et même que cela puisse nuire aux performances).

Découpage en composants

De manière à bien illustrer ce qu'il est possible de faire avec les composants « React » j'ai créé un total de 11 composants. Ils ont été pensés pour être utilisable par d'autres composants mais aussi de sorte qu'ils puissent fonctionner individuellement et sur différents types de visualisation.

Légende :

Chaque « boîte » correspond à un composant. Si deux boîtes sont de la même couleur il s'agit du même composant utilisé à plusieurs endroits. Les « creux » et « formes » qui s'emboîtent montrent que les composants coexistent sur la même vue.

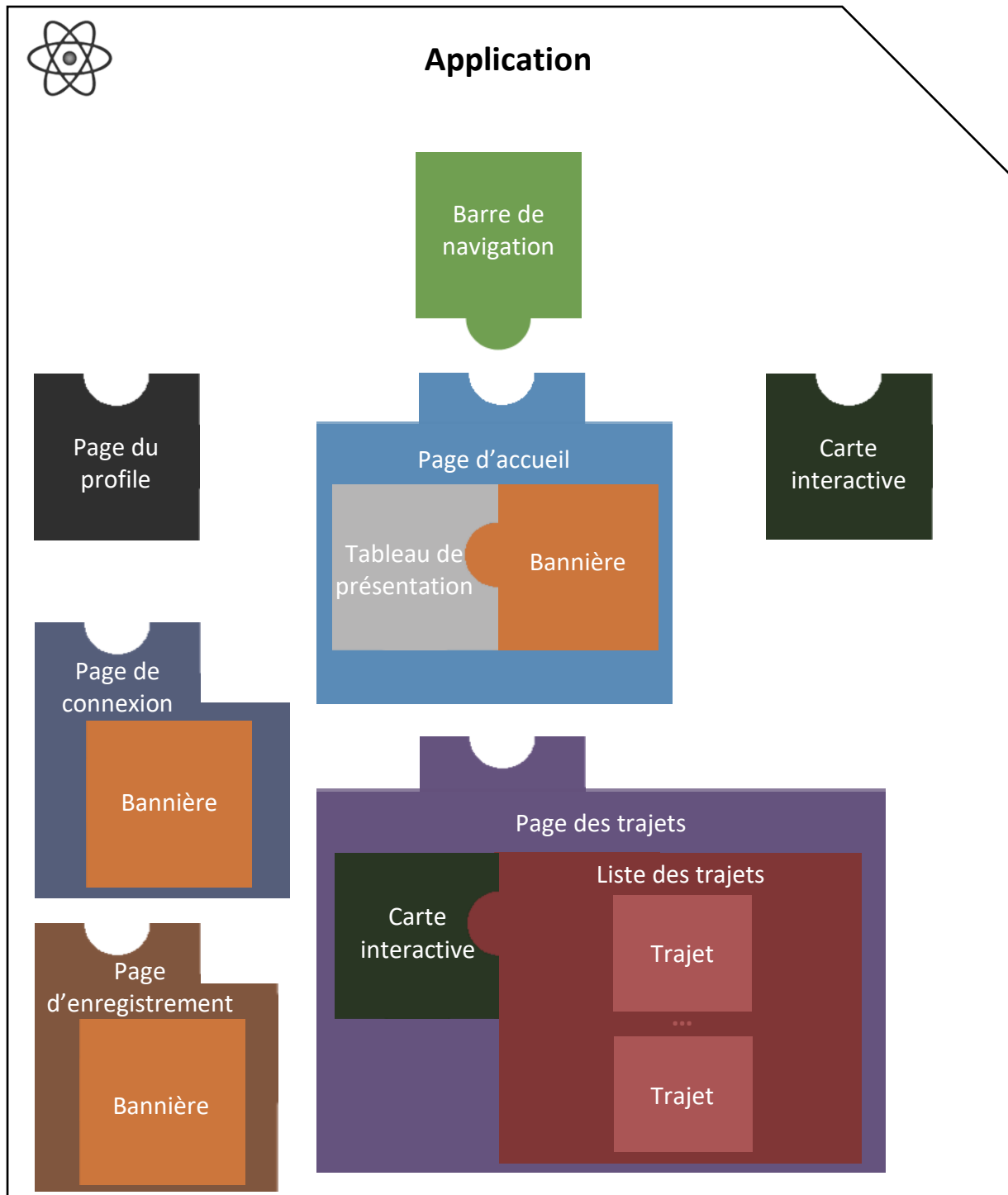


Figure 4 – Ensemble des composants réalisés

Utilisation de la technologie

Une fois les contraintes de conception réglées, l'implémentation d'une application « React » se fait plutôt rapidement à l'aide de la documentation de la page [create-react-app](#) (nécessite des éléments externes comme « npm », et donc « NodeJs »).

I. Création de l'application

Une simple utilisation de la commande :

```
npx create-react-app my-app
```

Suffit à créer l'arborescence minimale du projet (voir Structure).

II. Création d'un composant

Comme expliqué auparavant, l'organisation d'une application « React » dépend de l'équipe la développant. Je me suis longuement attardé sur la définition d'un composant et l'utilisation à en faire. Une fois l'application créée, un composant se définit de la sorte :

```
import React from "react";
import AutreComposantReact from './AutreComposantReact';
import './ComposantStyle.scss';

class Composant extends React.Component {
  render() {
    return (
      <div className="classCss">
        <p> Un élément HTML quelconque</p>
        <AutreComposantReact />
      </div>
    );
  }
}

export default Composant;
```

Import de la super classe « Component »

Import des autres composants à utiliser et de tous éléments nécessaires (styles, models, etc..)

La politique de « React » est de ne modifier le DOM que lorsque cela est absolument nécessaire. La modification du DOM se fait dans la méthode « render »

Comme on peut le remarquer avec l'attribut « className », le « code » renvoyé dans la méthode « render » n'est pas du HTML. Il s'agit d'une syntaxe particulière pourtant très proche de ce dernier, ce qui pourrait sembler être un bon point. Au contraire, il s'agit plus d'un désavantage car on risque souvent d'être tenté d'écrire du HTML et de faire une erreur.

III. Interactions entre les composants

Il est possible de passer des données à un composant fils à l'aide des « props ».

```
// Composant père :
const donnee = 'valeur quelconque';
render() {
  return (<div><ComposantFils proprieteDuFils={this.donnee} /></div>);
}
```

On passe les données nécessaires comme des attributs HTML, cela peut correspondre à des variables ou directement à des valeur (syntaxe ES6)

```
// Composant fils :
class ComposantFils extends React.Component {
  constructor(props) {
    super(props);
    this.variable = props.proprieteDuFils;
  }
}
```

Le constructeur va par défaut récupérer ces données dans la variable « props ». Elles seront accessibles avec l'appel suivant : « props.nomPasseLorsDeLaCreation »

Ce mécanisme est similaire à celui-ci des « @Input » en « Angular2+ », il est très utile car la plupart des composants vont mettre en forme des données issus d'ailleurs (en plus du fait de réutiliser un composant dans différents contextes). Par exemple la carte peut être en mode « création » ou « affichage » ; il est alors nécessaire de passer cette information au composant pour ajuster son comportement. Cependant, contrairement à « Angular2+ », la modification d'un objet passé en « props » ne modifie pas la vue (voir ci-dessous).

A l'inverse on peut vouloir envoyer des données d'un composant fils à son père (« @Output » en « Angular2+ »), n'ayant pas de mécanisme directement défini par « React » (et en prenant en compte que le fils ne connaît pas son père), on peut réaliser ce mécanisme par le biais de « callback » (appel retour) :

```
// Composant père :
fonctionDuPere= (donneeDuFils) => {
  // Traitement quelconque
}
render() {
  return (<div>
    <ComposantFils proprieteDuFils={this.fonctionDuPere} />
  </div>);
}
```

Similaire au passage de données précédent mise à part qu'on passe une fonction cette fois

```
// Composant fils :
constructor(props) {
  super(props);
  this.callbackFunction = props.proprieteDuFils;
}
fonctionDuFils= (donneeDuFils) => {
  this.callbackFunction(donneeDuFils);
}
render() {
  return (<div>
    <div onClick={() =>
      this.fonctionDuFils({this.donneeAEnvoyerAuPere})}>
      Elément quelconque déclenchant l'output
    </div>
  </div>);
}
```

Lors d'un click on appelle une fonction du fils qui pointe vers celle du père

A noter que les « props » ne peuvent pas être modifiés dans un composant fils, c'est pourquoi il est fréquent de les stocker dans des variables.

IV. Naviguer entre les composants (URL)

Outre le fait d'échanger entre les composants (et malgré le fait que « React » soit pensé de manière à faire des sites « one page »), il peut être utile de changer complètement de composant. Pour cela on utilise une URL différente :


http://monsite/composant1	http://monsite/composant2
	

Figure 5 - Illustration d'une navigation entre composants

Pour cela on va utiliser « BrowserRouter ». L'idée est d'encapsuler tout ce qui peut avoir vocation à changer dans une balise « <BrowserRouter> » et de définir quelle partie doit changer en fonction de quelle URL. Exemple :

```
<BrowserRouter>
  <ComposantToujoursPresent />
  <Route exact path="/" component={ ComposantParDefaut } />
  <Route exact path="/composant2" component={ Composant2 } />
</BrowserRouter>
```

Une barre de navigation ou un header par exemple

On peut alors changer de composant via des liens (génère des balises « <a> » pointant vers la bonne URL) :

```
<Link to="/composant2">
  Connexion
</Link>
```

Ou bien dans le code d'un composant :

```
this.props.history.push({
  pathname: '/composant2'
});
```

Change l'URL vers « /composant2 », ce qui affiche le « Composant2 » (défini dans le « BrowserRouter »)

Il est possible de définir son propre « guard » permettant de protéger des routes sous certaines conditions et de passer des données via des « pathParameter » et des « queryParameter » (variables dans l'URL). Je ne vais pas détailler ces comportements, des exemples existent dans le code du projet.

Ce qui est intéressant est de noter que « BrowserRouter » est une dépendance distincte de « React ». Une fois encore on voit bien la distinction entre une bibliothèque et un « framework », et une fois encore il s'agit d'un avantage et d'un inconvénient. On a un projet plus léger avec cet import non existant si on n'en a pas l'utilité, mais il est si fréquent d'y avoir recours qu'il est plus ou moins systématiquement dans les applications « React ». C'est lourd pour les développeurs d'aller chercher des petits morceaux partout.

V. Re-render un composant (principe d'état)

Comme expliqué précédemment, « React » ne change la vue que lorsque cela est strictement nécessaire. Il est donc primordial de savoir comment indiquer à notre application qu'il est nécessaire de réexécuter la fonction « render ». Pour cela il existe ce que l'on appelle « l'état » (« state »). Dans le constructeur d'un composant on va créer une variable nommée « state » un peu particulière. En effet on modifiera cette variable uniquement à l'aide de la fonction « setState » qui va, après modification des objets présents dans « state », appeler la méthode « render » du composant.

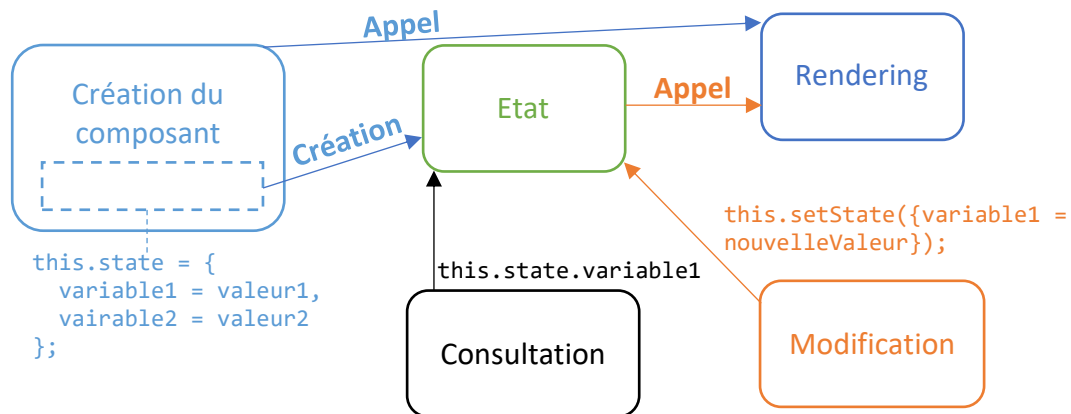


Figure 6 - Représentation simplifiée du mécanisme d'état

Référencement des avantages et des inconvénients

Avantages

- Liberté de la structure de l'application (voir ci-dessus),
- Possibilité d'écrire du JavaScript librement (syntaxe ES6 supportée),
- « Interactions » (passage de données) entre les composants faciles (voir ci-dessus),
- Application en kit, ce qui permet de n'importer que le strict nécessaire (voir ci-dessus),
- Navigation entre les composants très rapide à prendre en main (voir ci-dessus).

Inconvénients

- Manque de cadre pour aider à la maintenabilité de l'application (voir ci-dessus),
- Syntaxe non habituelle dans la méthode « render » (voir ci-dessus),
- Application en kit, ce qui force d'aller chercher régulièrement des packages qui ne sont pas de base dans « React » (voir ci-dessus),
- Mécanisme de « rendering » améliorant les performances mais n'est pas naturel et oblige la surcharge de state (voir ci-dessus),
- Lors de création de fonction j'ai heurté un problème de manière récurrente qui est : ce me retourne le « this ». Ce problème s'est particulièrement montré gênant lorsque que j'ai implémenté des fonctions « callback » entre un composant père et un composant fils. Le plus simple est de noter systématiquement une fonction d'un composant (qui n'est pas une fonction de « React ») de la manière suivante :

```

nomDeLaFonction= (parametre) => {
  code de la fonction ;
}

```

De cette manière le « this » correspond bien toujours à notre composant support.