

# Path'Partout



## Table des matières

<b>Introduction</b>	3
<b>Technologies utilisées</b>	3
<b>Adaptations</b>	3
<b>React et VueJS</b>	3
Adaptation réductive	4
Adaptation dé-portative	5
Différences entre les applications	6
<b>iOS et ReactNative</b>	7
Choix de la randonnée	7
Carte	8
Approche d'un point d'intérêt	9
<b>Tutorial React</b>	10
Introduction	10
Structure	10
Découpage en composants	11
Utilisation de la technologie	13
Référencement des avantages et des inconvénients	16
Avantages	16
Inconvénients	16
<b>Tutorial VueJS</b>	17
Introduction	17
Les vues	17
Les évènements	18
Communication entre les composants	18
Adaptation au dispositif	19
Référencement des avantages et des inconvénients	20
Avantages	20
Inconvénients	20

<b>Tutoriel iOS</b> .....	21
Introduction.....	21
CocoaPods .....	21
Storyboard.....	22
Vues .....	22
Tab Bar Controller .....	23
Communication vue non reliées .....	24
Avantages et inconvénients .....	25
Avantages .....	25
Inconvénients .....	25
<b>Tutorial ReactNative</b> .....	26
Introduction.....	26
Structure.....	26
Utilisation de la technologie.....	27
Le problème « Expo » .....	27
Création de l'application sans « Expo » .....	27
Création d'un composant .....	27
Interactions entre les composants .....	27
Actualiser l'affichage d'un composant .....	27
Mise en place des adaptations .....	28
Référencement des avantages et des inconvénients .....	28
Avantages .....	28
Inconvénients .....	28
<b>Conclusion</b> .....	29
<b>React vs Vue.js</b> .....	29
<b>iOS vs ReactNative</b> .....	29
<b>Installation</b> .....	29
<b>React</b> .....	29
<b>VueJS</b> .....	30
<b>iOS</b> .....	30
<b>ReactNative</b> .....	30
<b>Table des figures</b> .....	30

## Introduction

L'application a pour objectif de planifier et d'accompagner l'utilisateur pour effectuer tous types de trajets ; qu'il s'agisse d'une randonnée, d'un tour d'une ville, etc... Il va pouvoir utiliser une carte interactive pour l'élaboration de son trajet en créant un ensemble de « point d'intérêt ». Une fois le trajet planifié, l'utilisateur va pouvoir se servir de l'application pour s'orienter et obtenir des informations sur les points d'intérêts qu'il va rencontrer.

## Technologies utilisées

Pour réaliser ce projet nous avons décidé d'utiliser 4 technologies différentes, deux principalement orientés pour créer des applications dites de « bureau » (ordinateur avec un écran plutôt large) :

- React : bibliothèque « JavaScript » libre développée par « Facebook » pour la réalisation d'applications web,
- VueJS : framework « JavaScript » pour la réalisation d'applications web.

Et deux uniquement tournées vers des dispositifs mobiles :

- iOS : langage de programmation développé par « Apple », destiné à la programmation d'applications sur les systèmes d'exploitation iOS et macOS,
- ReactNative : framework « JavaScript » libre développée par « Facebook » pour le développement mobile.

## Adaptations

### React et VueJS

Les deux premières applications vont subir des adaptations liés au « **dispositif d'affichage** ». Bien que des applications uniquement destinées aux mobiles soient développées en parallèle, énormément de navigation internet se fait désormais sur smartphone. Il est donc intéressant d'adapter sa présentation et le contenu présent en fonction du dispositif qui l'affiche. Le principe des deux applications réalisées est le suivant :

*« Un dispositif avec un large affichage<sup>1</sup> doit afficher toutes les informations nécessaires pour une action sans changer de page.*

*Un dispositif moyen<sup>2</sup> doit permettre autant de fonctionnalités par page mais les éléments composants la vue doivent être plus facilement identifiable.*

*Un petit dispositif<sup>3</sup> doit afficher au maximum un élément retenant toute l'attention de l'utilisateur »*

---

<sup>1</sup> Ordinateur, affichage utilisé : 1920x1080

<sup>2</sup> Tablette, type iPad, affichage utilisé : 780x1024

<sup>3</sup> Smartphone, type Samsung Galaxy9+, affichage utilisé : 360x740

Les deux applications répondent aux fonctionnalités suivantes :

- Présenter l'application
- Posséder un compte
- Créer des trajets à l'aide d'une carte interactive
- Afficher les trajets créés

Cela s'est traduit par manière d'adapter nos vues :

## Adaptation réductive

On va resserrer les éléments jusqu'à un certain point où il est nécessaire de les réagencer (et de retirer les éléments superflus comme le téléphone dans l'exemple ci-dessous) :

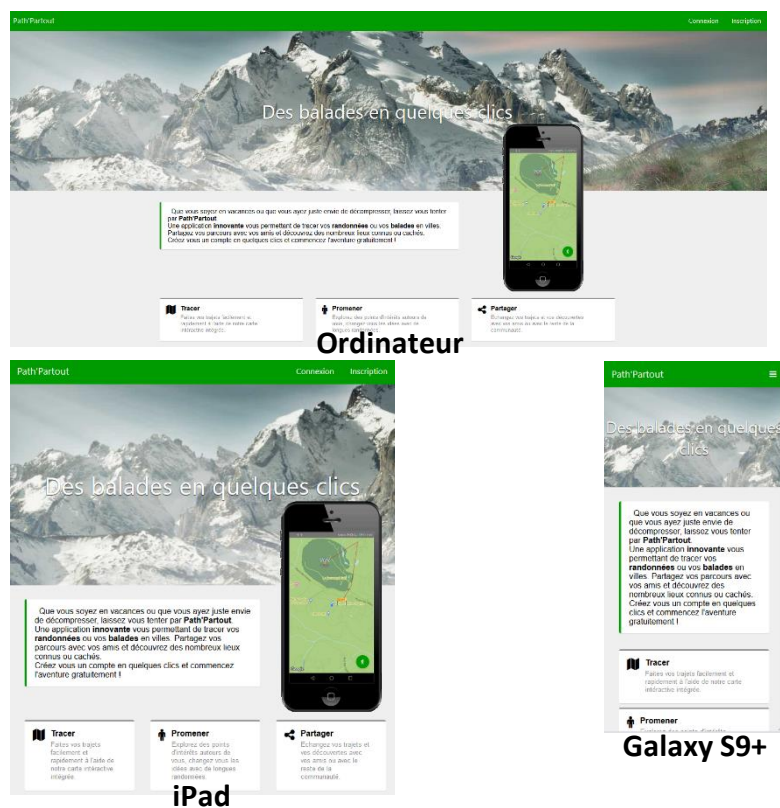


Figure 1 - Adaptation réductive sur la page d'accueil de l'application React

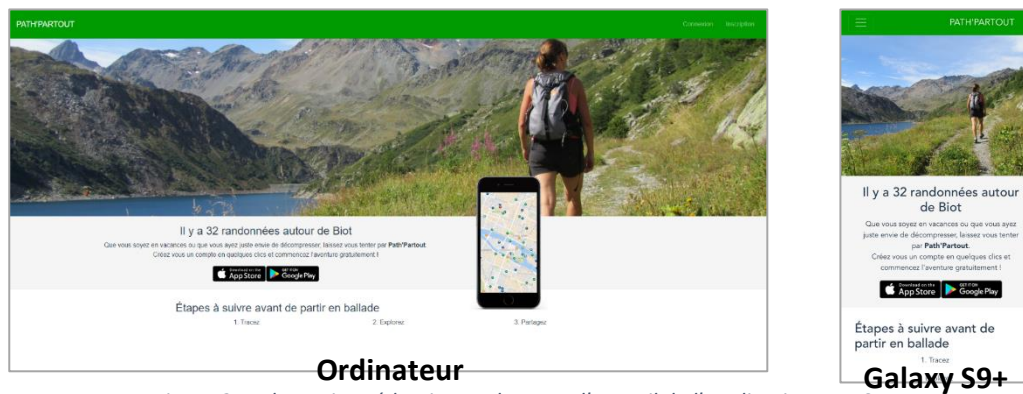


Figure 2 - Adaptation réductive sur la page d'accueil de l'application VueJS

A noter également le changement de la barre de navigation, qui devient un « hamburger menu », bien plus commun sur mobile et prenant moins d'espace.

### Adaptation dé-portative

D'après le postulat de base, une page complexe, comme la page des trajets, doit être divisée en plusieurs vues sur mobile mais pas sur des dispositifs plus larges. Si on prend en exemple les applications utilisant des cartes (Google Maps, Waze, Geaocaching, etc...) la carte est systématiquement seule ; cela pour une raison simple : elle affiche déjà à elle seule beaucoup d'informations (même trop si on n'adapte pas son contenu) et est l'élément centrale de l'application (l'utilisateur y concentre son attention). Il a donc fallu déporter la carte sur une vue isolée lorsque l'utilisateur se trouve sur un dispositif trop petit :

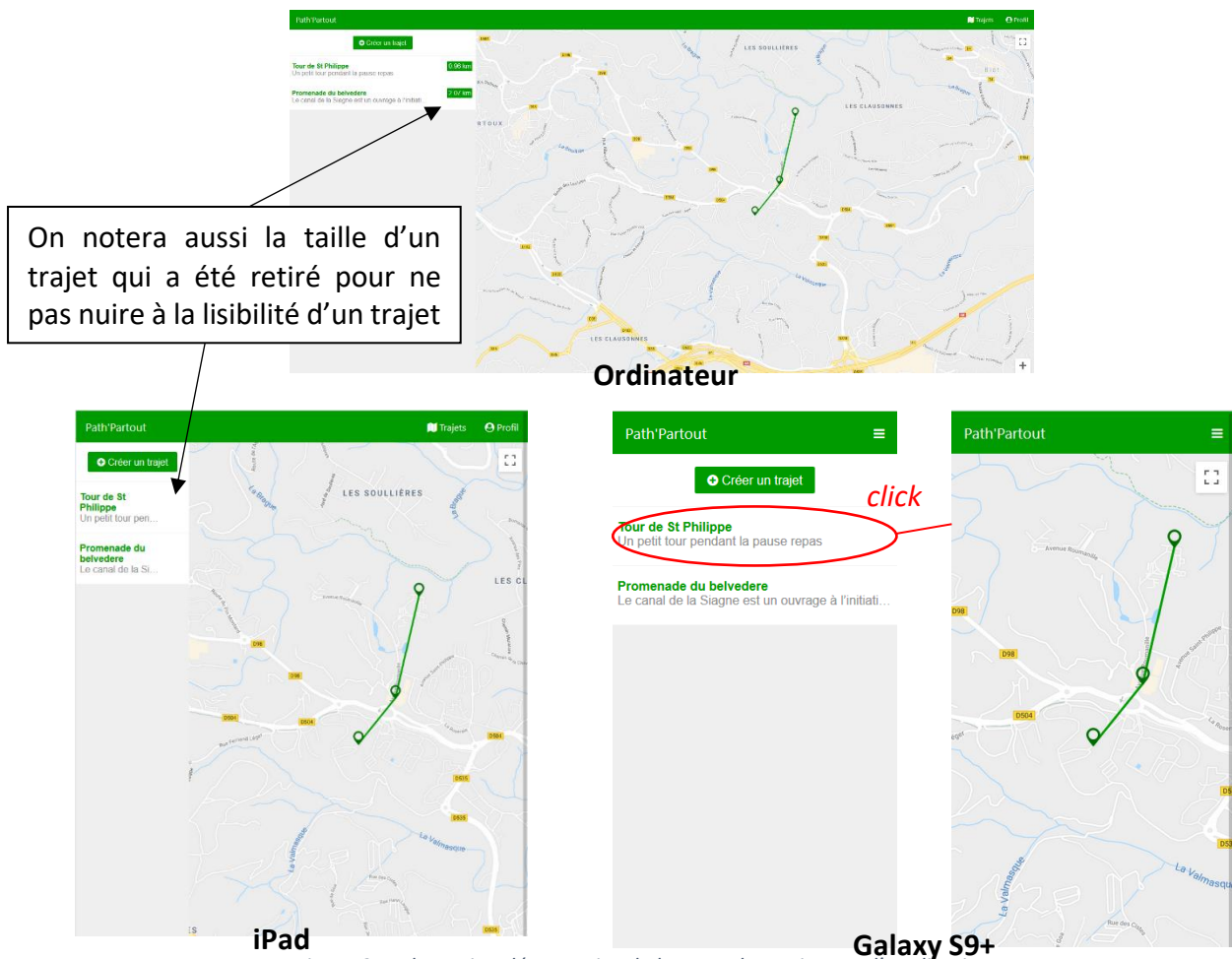
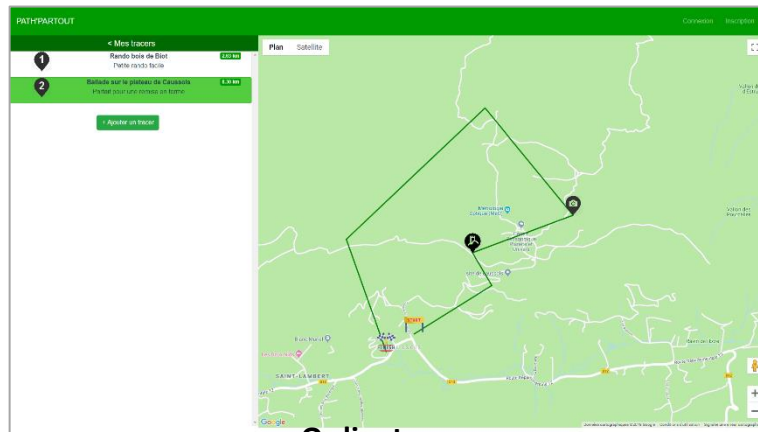
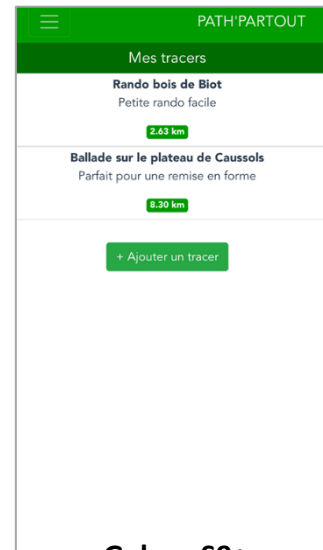


Figure 3 - Adaptation dé-portative de la page des trajets sur l'application React



Ordinateur

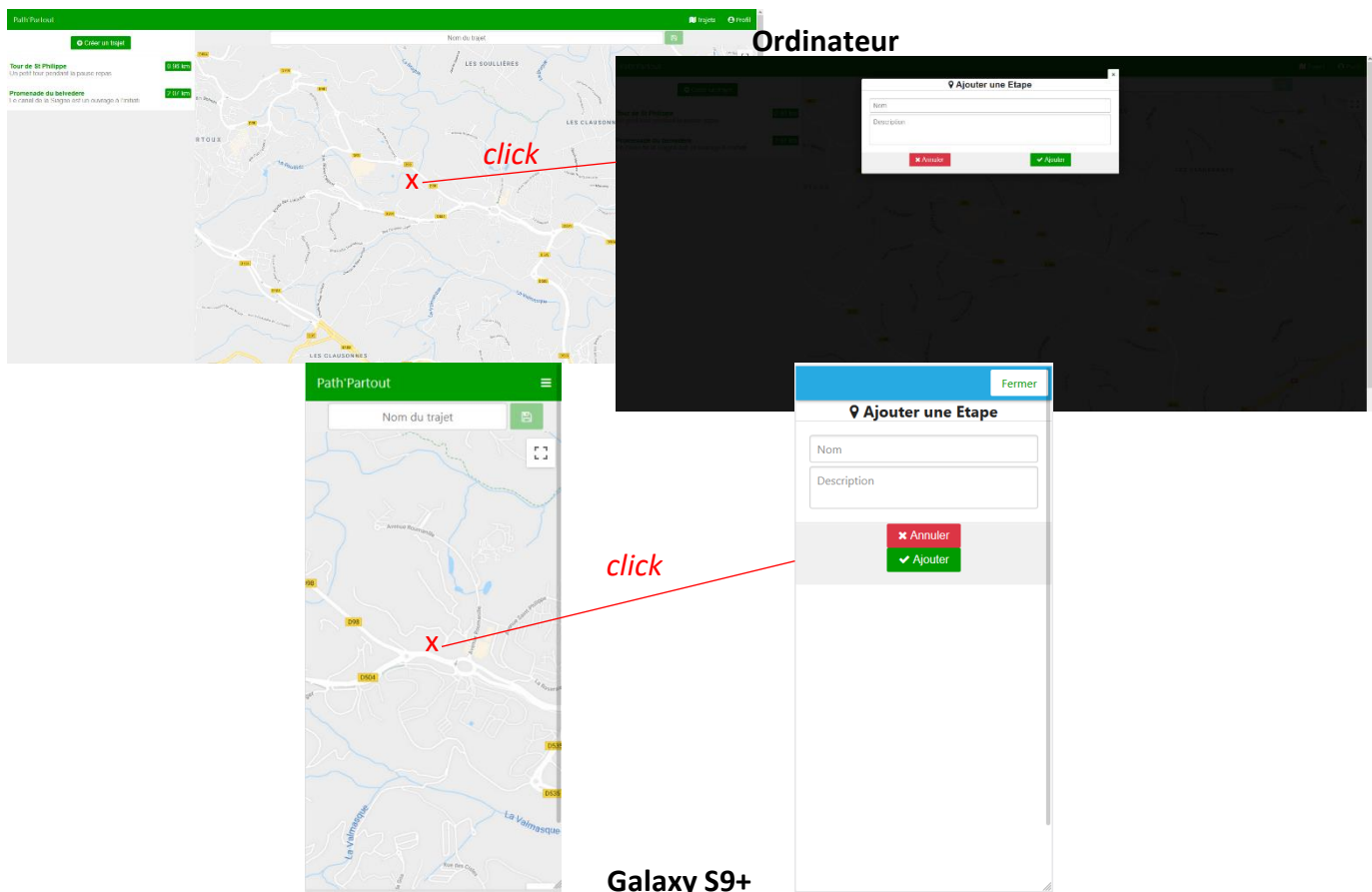


Galaxy S9+

Figure 4 - Adaptation dé-portative de la page des trajets sur l'application VueJS

## Différences entre les applications

Outre les modifications graphiques légères (et le code), les deux applications divergent sur un point : la création d'un trajet. Pour l'application « React » il a été choisi de capter des clicks sur la carte (sur large et petit dispositif) et d'y ajouter des points à l'aide d'une fenêtre demandant des informations sur l'étape. De cette manière on garde l'aspect interactif et on n'est pas bloqué par la recherche d'adresse lors de l'élaboration de trajet dans la nature :



Ordinateur

Galaxy S9+

Figure 5 - Création d'un trajet sur l'application React



Sur l'application « VueJS » on va simplement rentrer le nom d'un lieu dans la barre de recherche de lieu pour que cela ajoute un marker sur la carte. Alors que sur la vue pour large dispositif va avoir un comportement similaire à l'application « React » :

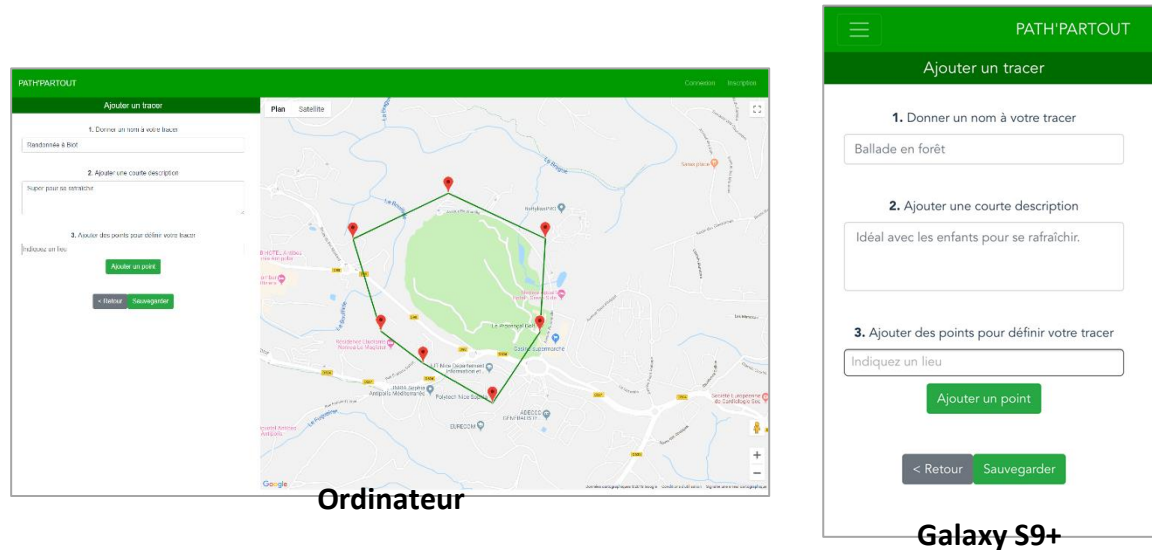


Figure 6 - Création d'un trajet sur l'application VueJS

## iOS et ReactNative

Les deux applications mobiles vont subir des adaptations liées au capteur GPS. Comme ce sont des applications utilisées exclusivement pour effectuer des trajets et se déplacer, il est donc essentiel que le GPS soit omniprésent dans l'application. Nous avons cherché à obtenir les caractéristiques suivantes :

« Un utilisateur doit pouvoir facilement sélectionner un circuit grâce à la distance qui le sépare de ce dernier.

Lorsqu'il trouve un point, il doit être notifié de sa découverte. »

Les deux applications répondent aux fonctionnalités ci-dessous :

- Permettre à l'utilisateur de s'orienter
- Afficher sur la carte un trajet créé au préalable
- Récupérer une liste de trajets triés par leurs distances par rapport à l'utilisateur

## Choix de la randonnée

La première adaptation mise en place à laquelle l'utilisateur va être confronté est présente sur le choix de la randonnée. En effet, avant de commencer une randonnée il doit aller la sélectionner parmi ses randonnées. Nous avons choisi de représenter les randonnées sous la forme d'une liste. L'adaptation résulte en l'organisation de cette liste de manière à simplifier la recherche des randonnées les plus proches. On va donc utiliser la position de l'utilisateur et calculer la distance entre les randonnées et sa position afin de trier la liste.

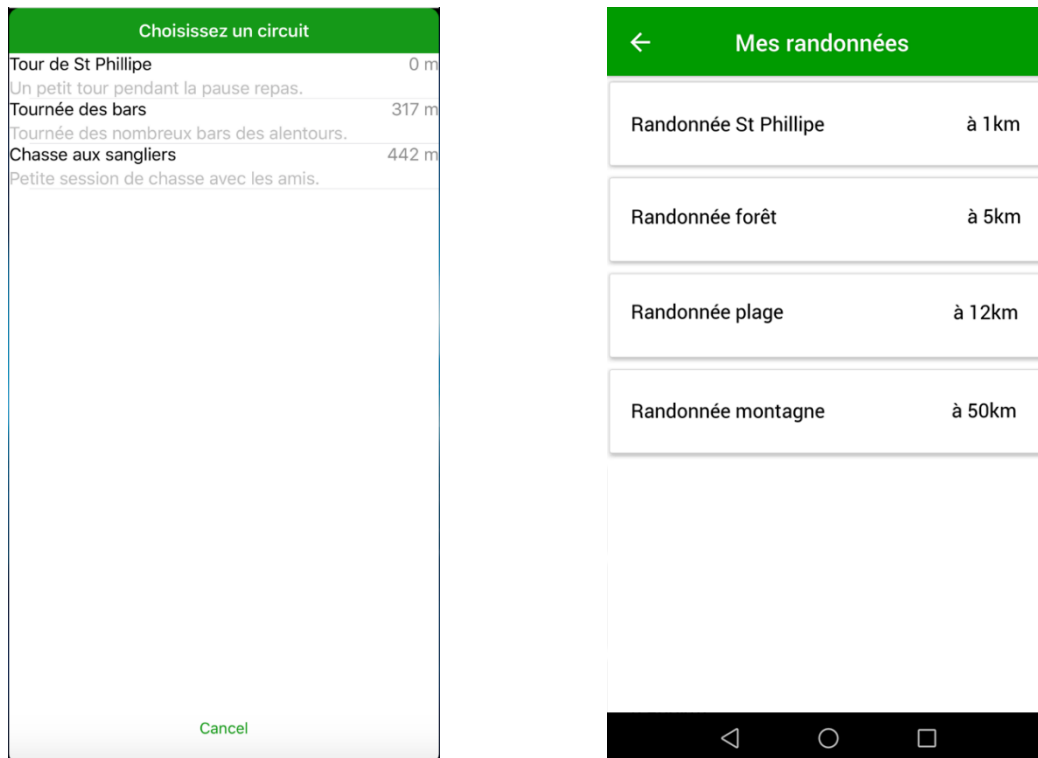


Figure 7 - Vue listant les trajets sur l'application iOS (à gauche) et ReactNative (à droite)

## Carte

A l'origine, nous n'affichons aucun circuit : la carte serait vite surchargée si l'utilisateur en rentre plusieurs. Une fois un parcours sélectionné, il est affiché sur la carte et l'utilisateur va pouvoir partir à la conquête des différents points d'intérêts :

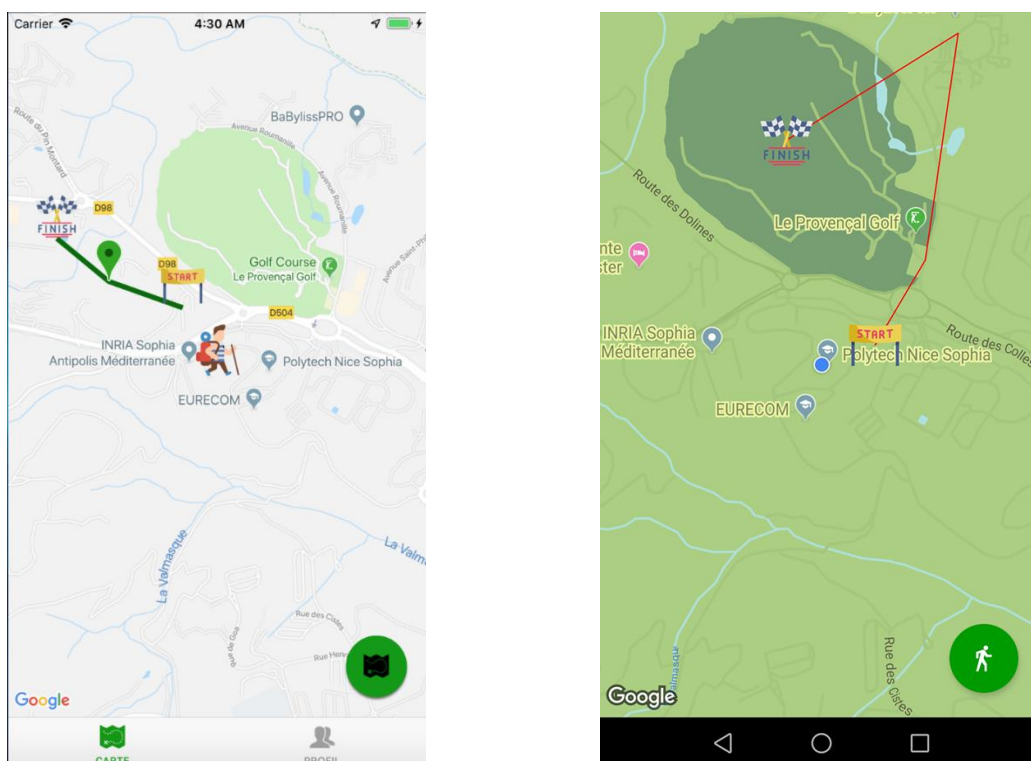


Figure 8 - Carte interactive de l'application iOS (à gauche) et ReactNative (à droite)



### Approche d'un point d'intérêt

La seconde adaptation concerne la découverte de point. Lorsqu'un utilisateur trouve un point sur la carte, il est notifié par un popup le lui indiquant. Pour cela, nous suivons la position de l'utilisateur tout au long de sa randonnée et lorsqu'il s'approche assez d'un point d'intérêt de la randonnée en cours, l'action se lance. Une fois le parcours terminé, l'utilisateur est aussi notifié. On voit ici une différence entre l'implémentation « iOS » et « ReactNative » puisque sur « iOS » l'utilisateur est juste notifié de la découverte du point d'intérêt alors que sur « ReactNative » une fenêtre s'affiche présentant le point d'intérêt.

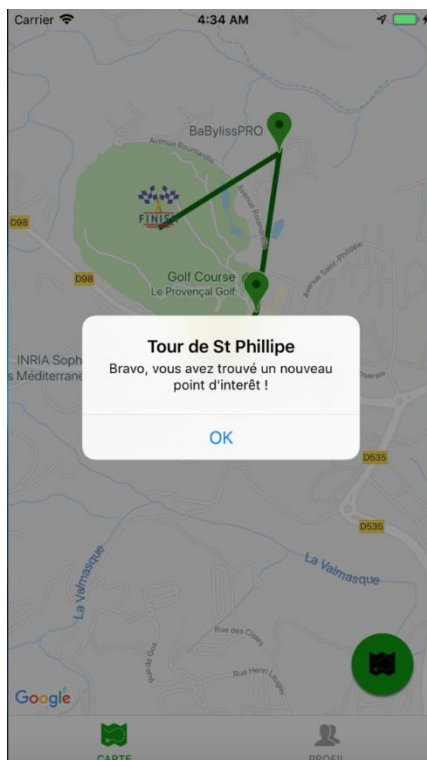


Figure 9 – Approche d'un point d'intérêt sur iOS (à gauche) et ReactNative (à droite)

## 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.

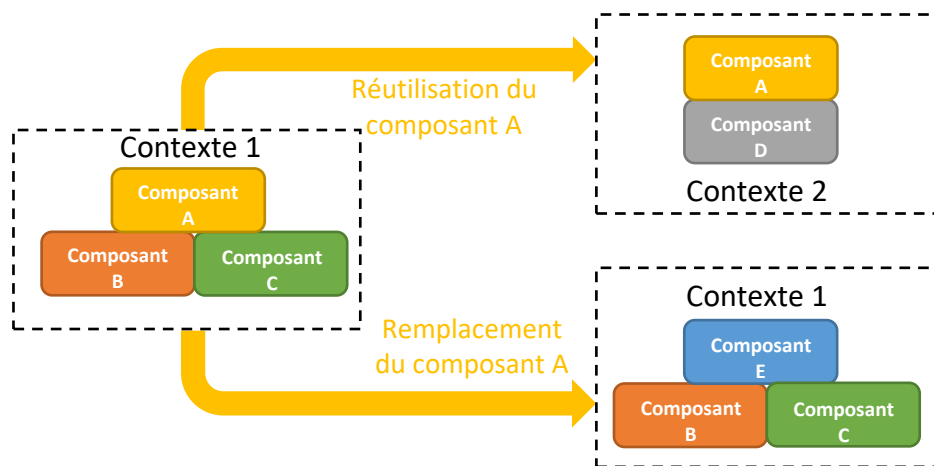


Figure 10 - Développement par composant

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 cette partie.*

### 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 :

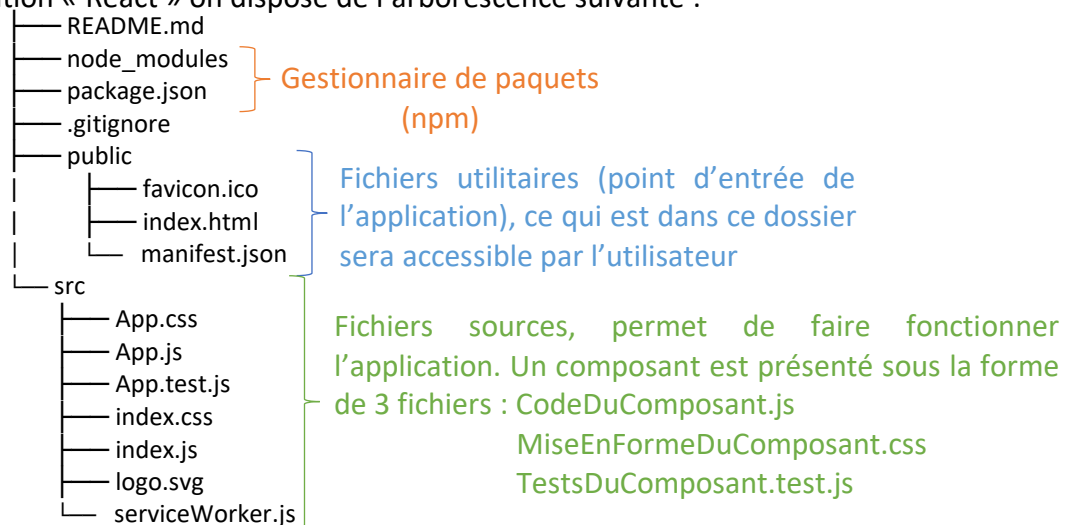


Figure 11 - Architecture proposée par "create-react-app"

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

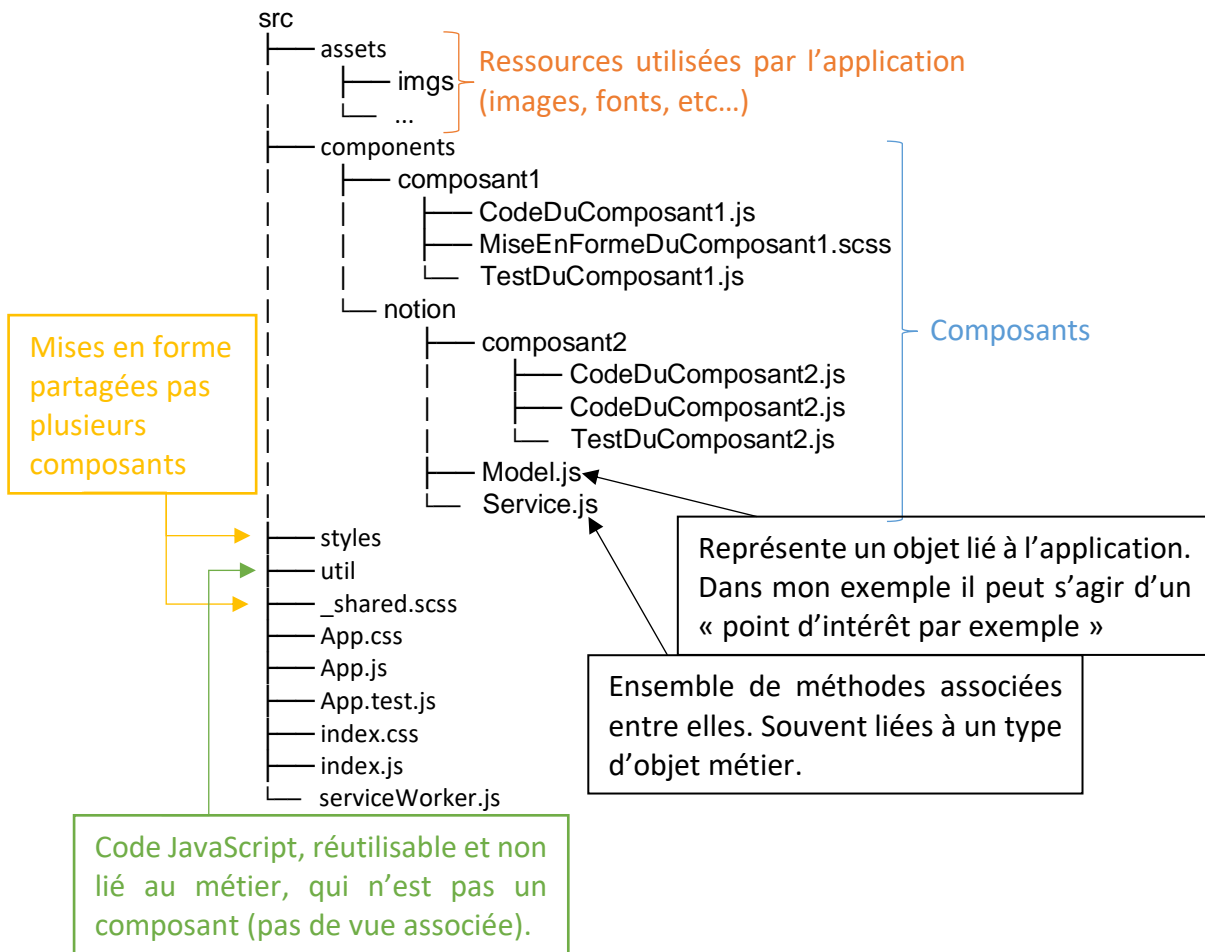


Figure 12 - Architecture adoptée

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.

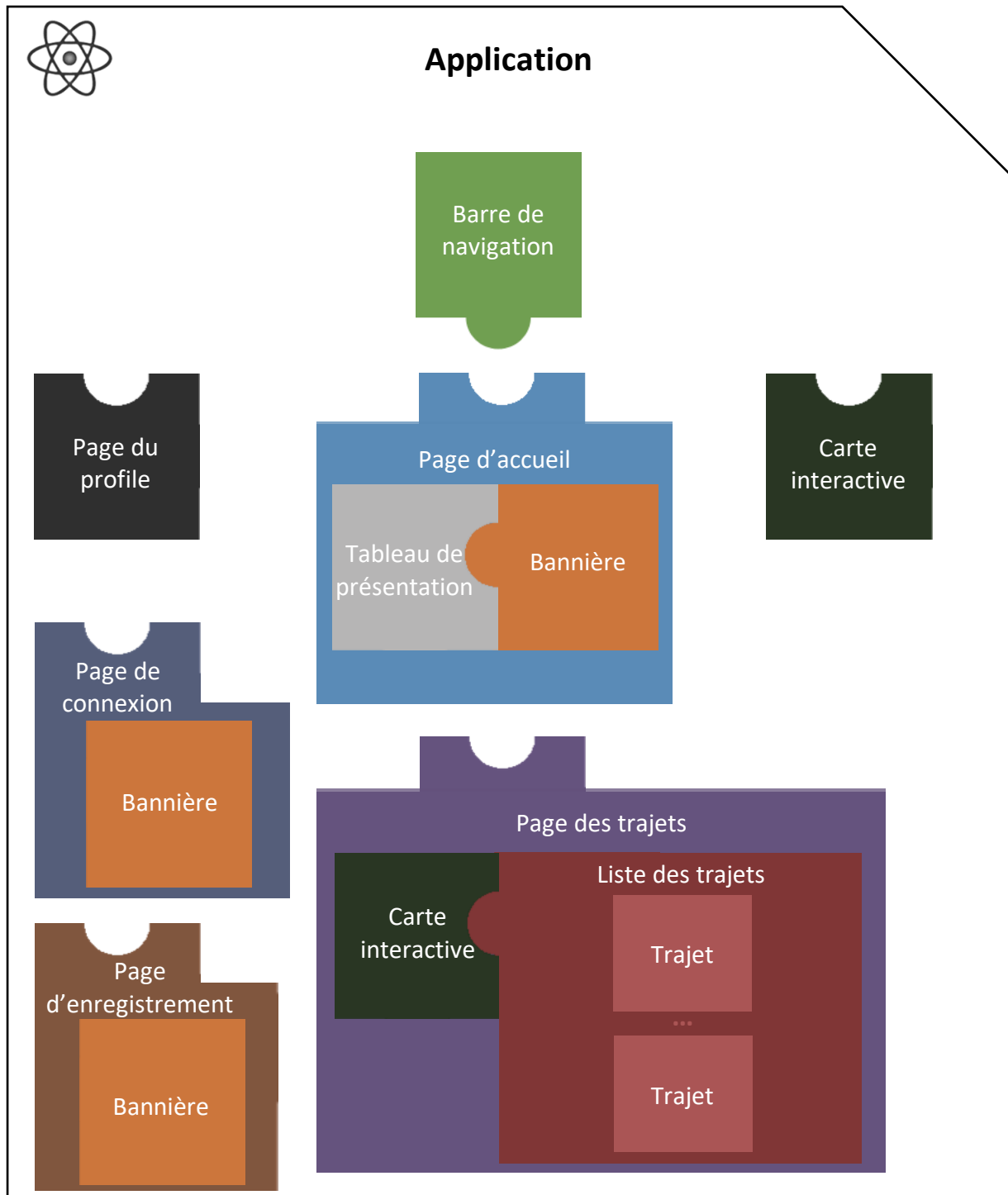


Figure 13 – Ensemble des composants réalisés

## 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 14 - 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.

Pour certaines adaptations ce comportement m'a été utile. D'une manière générale je me suis servi du modules fournis par « PureCSS », des « mediaquery CSS » et des variables « JavaScript » comme « windows.innerWidth » pour afficher/masquer/redimensionner des éléments, ou bien changer complètement de composant.

## 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.

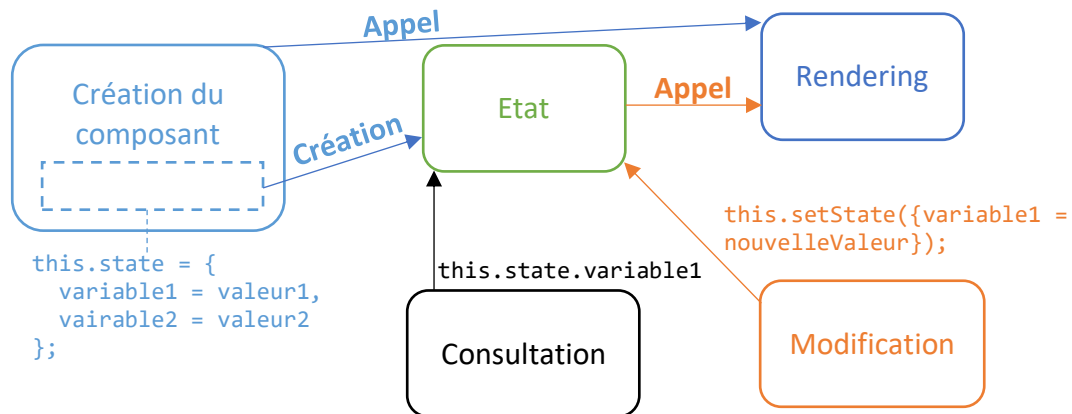


Figure 15 - Représentation simplifiée du mécanisme d'état

## 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.

## Tutorial VueJS

### Introduction

Vue est un framework javascript permettant de développer des interfaces utilisateurs. Comme ses principaux concurrents React et Angular, Vue se veut être un framework visant à simplifier et organiser le développement d'applications web en implémentant un bon nombre de fonctionnalités telles que les modèles, les composants, les transitions, le routage, etc...

La particularité de Vue.js est qu'il a été créé pour permettre des développements rapides tout en possédant une simplicité d'utilisation et de compréhension de la logique. Ainsi migrer depuis un projet sans framework est beaucoup plus facile. Un composant au sens Vue.js est représenté par un fichier qui a comme extension « .vue », dans ce fichier on peut retrouver le modèle du composant (son template html, son style css mais aussi sa logique métier grâce à un script javascript).

### Les vues

Tout comme ses concurrents React Native et Angular, on peut réutiliser un même composant en définissant lors de sa création les données qu'il contiendra. Un fichier vu est découpée en 3 parties correspondant au code html, js et css.

```
<template>
  <div>
    <div v-for="(run, index) in runs" :key="index" v-on:click="clickRun(index)">
      <Run :title=run.title :description=run.description :date=run.date :img=run.img />
    </div>
  </div>
</template>
```

```
<script>
import RunsJSON from "../resources/runs.json";
export default {
  name: "Runs",
  components: {
    Run
  },
  data: function() {
    return {
      runs: RunsJSON.runs,
    };
  },
  methods: {
    clickRun: function(index) {
      this.clickedItem = index;
      this.$emit("clicked", index);
    }
  }
};
</script>
```

```
<style scoped>
.navbar-brand {
  background: #006b00;
  width: 100%;
  color: white;
  padding-right: 0px;
  margin-right: 0px;
  cursor: pointer;
}
</style>
```

Figure 16 - Exemple type d'un fichier Vue.js

Le framework vue dispose de plusieurs fonctions et directives permettant de simplifier le développement des composants. Il existe par exemple la directive `v-if='isValid()'` qui est une structure conditionnelle d'affichage, l'élément sera affiché seulement si la condition est vérifiée. J'ai pu utiliser également `v-for='(run, index) in runs'` qui permet le rendu d'une liste d'éléments avec la notion d'élément courant et son index. Ainsi comme le montre la figure 1, il est très simple de définir le composant fils qui sera réutilisé dans le boucle du v-for.

### Les évènements

L'écriture d'évènement avec Vue.js est rendue très accessible et simple d'implémentation. En effet il est possible de définir sur n'importe quel élément une directive de type `'v-on:click=addMarker()'`

```
<div v-for="(run, index) in runs" :key="index" v-on:click="clickRun(index)">
  <Run :title=run.title :description=run.description :date=run.date :img=run.img />
</div>
```

```
methods: {
  clickRun: function(index) {
    this.clickedItem = index;
  }
}
```

### Communication entre les composants

Ma principale difficulté dans la prise en main de ce framework a été de bien gérer la communication entre les composants. Par exemple dans l'application, sur la vue de mes tracers, dès lors que l'on clique sur un tracer en particulier, la carte est mise à jour pour afficher le tracer. Ainsi je dois faire remonter l'information au parent comme quoi un des tracers a été cliqué et ensuite le parent doit informer le composant GoogleMap.vue pour que la carte se mette à jour. La documentation officielle de Vue.js préconise d'utiliser les \$emits et les évènements.

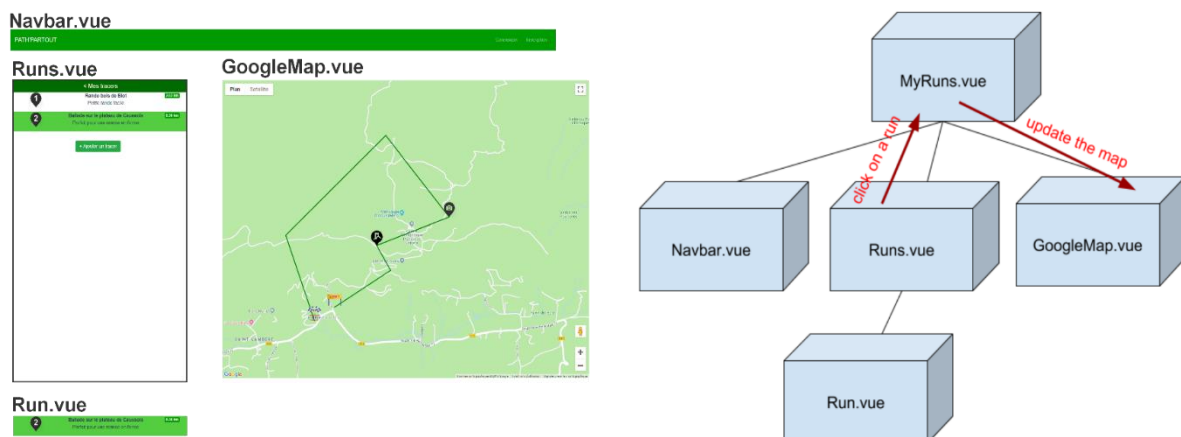


Figure 17 - Communication entre les composants

J'ai donc procédé comme suit :

Runs.vue

```
<div v-for="(run, index) in runs" :key="index" v-on:click="clickRun(index)">
  <Run :title=run.title :description=run.description :date=run.date />
</div>
```

```
clickRun: function(index) {
  this.clickedItem = index;
  this.$emit("clicked", index);
},
```

MyRuns.vue

```
<div class="col-md-4">
  <Runs @clicked="onClickChild"/>
</div>
<div class="col-md-8">
  <div>
    <GoogleMap ref="runIndex"/>
  </div>
</div>
```

```
onClickChild(value) {
  this.$refs.runIndex.setMarkers(value);
},
```

GoogleMap.vue

```
setMarkers(index) {
  this.path = this.runs[index].path;
  this.markers = this.runs[index].pois;
},
```

### Adaptation au dispositif

Comme le but de notre projet était d'adapter l'affichage en fonction du dispositif, il était nécessaire d'avoir un moyen de détecter la taille du dispositif de l'utilisateur. Pour cela plusieurs possibilités s'offraient à moi. Je pouvais soit utilisé les Media Queries en CSS, qui va charger un certain style en fonction de la taille de l'écran. Ou alors je pouvais aussi choisir de n'afficher un élément que si l'utilisateur était sur un certain dispositif avec du code javascript. J'ai décidé de procéder ainsi. J'ai donc créer un fichier DeviceHelper.js qui contient la fonction isMobileDevice(). Donc cette fonction, on vient détecter les meta name contenus dans la variable navigator. Et en fonction de ce qu'elle contient on peut estimer si l'utilisateur est sur mobile ou non.

Ainsi, dans chaque vue où j'affiche un élément de façon conditionnelle, je charge le fichier helper et j'utilise la directive **v-if** pour afficher ou non un composant. Ce moyen de faire est assez lourd et pas forcément le plus efficace, puisque sur le navigateur par exemple, si on ouvre la console et que l'on décide de montrer l'application comment elle serait sous mobile, la fonction isMobileDevice() ne détectera pas que l'on est sous mobile puisque le navigateur reste un navigateur desktop et donc les affichages conditionnels ne fonctionneront pas correctement. Mais c'est un choix que j'ai pris pour utiliser le plus possible les directives offertes par Vue.js.

```
<div class="col-md-4 myCol">
  <Runs v-if="this.isAddingRun == false" @clicked="onClickChild" @reset="reset" @addRun="addRun"/>
  <AddRun v-if="this.isAddingRun == true" @back="back" @newMarker="newMarker" />
</div>
<div v-if="!helper.isMobileDevice()" class="col-md-8 myCol">
  <div>
    <GoogleMap ref="runIndex"/>
  </div>
</div>
```

## Référencement des avantages et des inconvénients

### Avantages

- Architecture modulaire
- Facilité de migration d'un projet sans framework vers Vue.js
- Framework très flexible
- Contrairement à React, Vue.js est lié au DOM et utilise des attributs HTML spéciaux pour rendre le DOM réactif
- Poids du framework comparé à React et Angular

### Inconvénients

- Framework récent et peu documenté
- Fichiers « .vue » qui peuvent être rapidement énormes puisqu'ils embarquent le code HTML, CSS et JS
- Manque de ressources
- Communication entre les composants assez lourde (voir la partie correspondante ci-dessus)



## Tutoriel iOS

### Introduction

Swift est un langage de programmation développé par la société Apple, destiné à la programmation d'applications sur les systèmes d'exploitation iOS, macOS, ... Ce langage est donc propre à Apple, il est donc rare de s'en être servi si l'on n'a pas développé d'applications destinées à un dispositif Apple.

La particularité que j'ai tout de suite remarquée par rapport aux autres langages mobiles (que ce soit Frameworks multi-plateforme ou Android) est que l'interface de développement est très visuelle. L'on peut effectuer beaucoup de traitement directement depuis les "storyboard" qui sont des fenêtres contenant nos différentes vues, et les relations entre elles.

### CocoaPods

Suite à différents problèmes lors de l'import de bibliothèques telle que GoogleMaps, j'ai choisi d'utiliser un logiciel de gestion de dépendances afin de m'aider dans cette tâche. Ce logiciel permet d'importer « automatiquement » les librairies demandées, et requiert un fichier « Podfile » de la forme:

```
1 platform :ios, '12.0'
2
3 source 'https://github.com/CocoaPods/Specs.git'
4 target 'PathPartout' do
5   pod 'GoogleMaps'
6   pod 'GooglePlaces'
7 end
```

Pour lancer CocoaPods, il faut utiliser la commande `pod install` puis `open App.xcworkspace`. La chose à ne pas oublier est que par la suite, il faut toujours ouvrir le fichier `.xcworkspace` et non le `.xcodeproj` car les dépendances ne sont pas gérées sinon. Le problème est que de base, XCode (éditeur de code développé par Apple spécialement pour le swift) ouvre de base les `.xcodeproj`, il faut donc bien penser à l'ouvrir manuellement à chaque fois...

## Storyboard

### Vues

Les différentes vues et contrôleurs doivent être créés directement depuis le fichier « storyboard » :

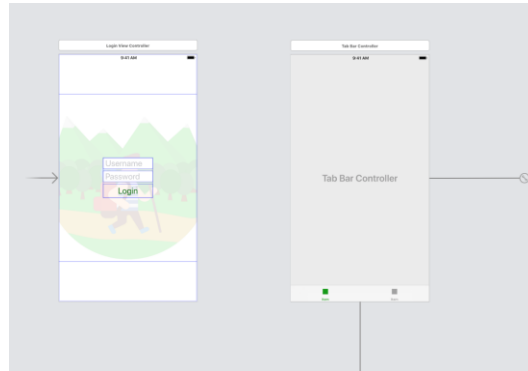


Figure 18 - Storyboard

L'on peut définir depuis cette page le point d'entrée de l'application, nos vues, et les relations entre elles. Par exemple, nous pouvons choisir quelle vue s'ouvrira lors d'un clic sur un bouton directement depuis l'interface graphique, en maintenant « control » en glissant vers la vue cible :

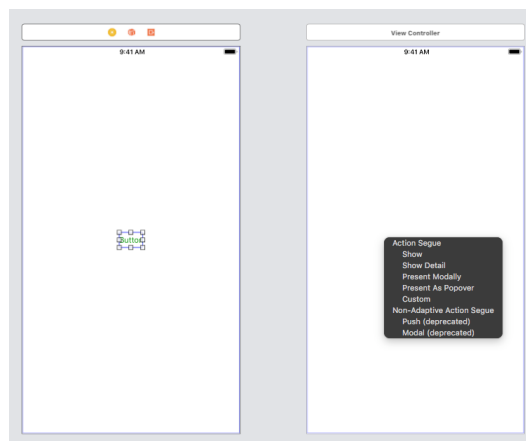


Figure 19 - Relation entre vues depuis storyboard

L'on peut ensuite choisir quel type d'actions, et même la transition directement depuis l'interface graphique. C'est un processus un peu surprenant au départ, car non initié mais finalement tout est bien pensé et on devient vite familier avec la méthode. Le processus est le même lorsqu'on souhaite faire un lien entre la vue et le contrôleur : l'on peut par exemple faire glisser un bouton depuis la vue vers le contrôleur pour créer la variable qui lui sera assignée, ou même pour le lier à une action (onclick, ... ) :

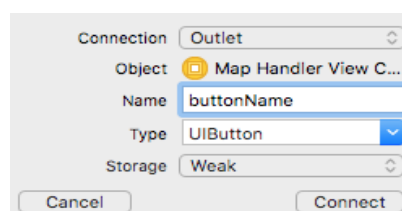


Figure 20 - Connexions vues - controller depuis storyboard

### Tab Bar Controller

J'ai trouvé l'approche pour créer un Tab Bar Controller différente de ce que j'ai eu l'occasion de faire habituellement. Encore une fois, l'on peut tout faire en interface graphique et lorsqu'on crée un Tab Bar Controller, nous obtenons le résultat suivant :

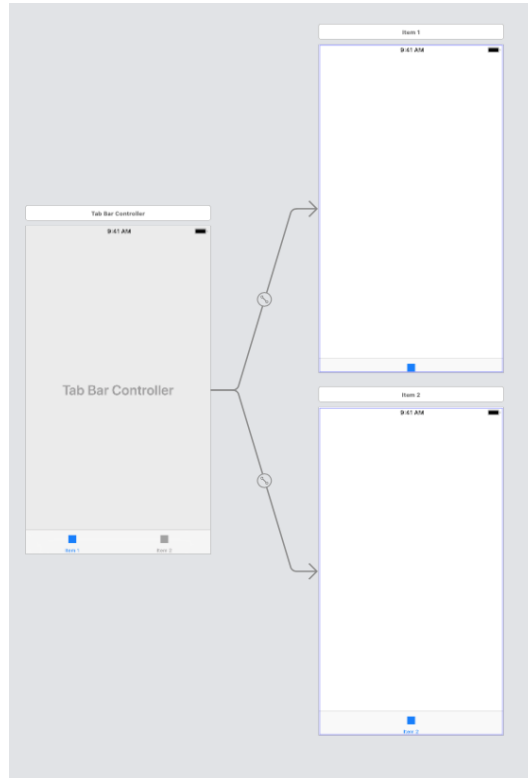


Figure 21 - Tab Bar Controller

En fait, lors de la création deux **Items** sont automatiquement créés ce qui correspond aux différentes pages affichées dans le controller. J'ai trouvé cette manière différente, et ai dans un premier temps essayé de créer une barre de navigation en bas de page, que j'aurais ensuite placée sur les différentes vues où elle doit apparaître. Ici en fait le processus est simplifié par rapport à d'autres environnements, mais différent ce qui cause parfois des pertes de temps.

## Communication vue non reliées

Lors du développement de l'application, j'ai dû faire communiquer des vues non reliées. C'était dans le cadre particulier de l'intégration d'une page de connexion : la page de connexion doit pouvoir être accessible à n'importe quel moment, si l'utilisateur souhaite se déconnecter. Il serait mauvais de placer sur chaque vue un bouton "se déconnecter" reliée à la vue de la connexion. C'est pourquoi, il est commun de créer une vue de connexion reliée à aucune autre vue et d'utiliser un fichier « Switcher ». C'est en fait un fichier global, qui pourra être accessible depuis n'importe quelle vue. La fonction ci-dessous montre un exemple d'utilisation du Switcher (dans cet exemple le mot de passe n'est vérifié, l'on supposera qu'il est bon) :

```
@IBAction func buttonActionLogin(_ sender: Any) {
    UserDefaults.standard.set(true, forKey: "status")
    Switcher.updateRootVC()
}
```

Et le « Switcher » :

```
static func updateRootVC(){

    let status = UserDefaults.standard.bool(forKey: "status")
    var rootVC : UIViewController?

    if(status == true){
        rootVC = UIStoryboard(name: "Main", bundle:
            nil).instantiateViewController(withIdentifier: "tabbarvc") as!
            UITabBarController
    }else{
        rootVC = UIStoryboard(name: "Main", bundle:
            nil).instantiateViewController(withIdentifier: "loginvc") as!
            LoginViewController
    }

    let appDelegate = UIApplication.shared.delegate as! AppDelegate
    appDelegate.window?.rootViewController = rootVC

}
```

Ainsi, uniquement dans le cas où le mot de passe est bon le contenu de l'application sera dévoilé, sinon l'on restera sur la page de connexion. Pour déconnecter l'utilisateur, c'est aussi simple : l'on met le status à false et l'on rappelle `Switcher.updateRootVC()`. Bien sûr, l'on ne prend pas en compte le backend ici, il y aurait sinon plus de traitements à prendre en compte. L'on remarque aussi que l'on peut appeler les ViewController avec leurs identifiants. Ce processus permet de réutiliser facilement les composants.

### Avantages et inconvénients

#### Avantages

- Très simple de vérifier le design de l'application sur tous les dispositifs (pour vérifier que les différentes tailles d'écran sont bien optimisées) car peu de modèles
- L'accès au GPS, et aux autres fonctionnalités est très facile
- XCode est vraiment optimisé pour la language : usage très plaisant
- Ensemble de tutoriels écrits par Apple bien détaillés
- Gestion des couleurs et images de manière intuitive (pas de chemins d'accès à utiliser, il suffit de les glisser dans un dossier)

#### Inconvénients

- Apprentissage plutôt long (pas le langage de programmation, mais tout ce qu'il y a autour)
- Utilisation de Swift qui est restreint à Apple
- Lors de recherches d'aide sur internet, beaucoup de méthodes sont inutilisables : beaucoup de modifications du langage lors des nouvelles versions
- Pas de documentation sur certaines erreurs faisant planter l'application

## Tutorial ReactNative

### Introduction

ReactNative est un framework JavaScript libre développée par Facebook. Il permet de faciliter le développement d'application mobiles natives multiplateformes en se basant sur le mécanisme de composants déjà mis en place par React.

### Structure

Concernant la structure du projet, j'ai décidé de garder la structure générée automatiquement lors de la création d'un projet ReactNative.

La structure se présente comme suit :

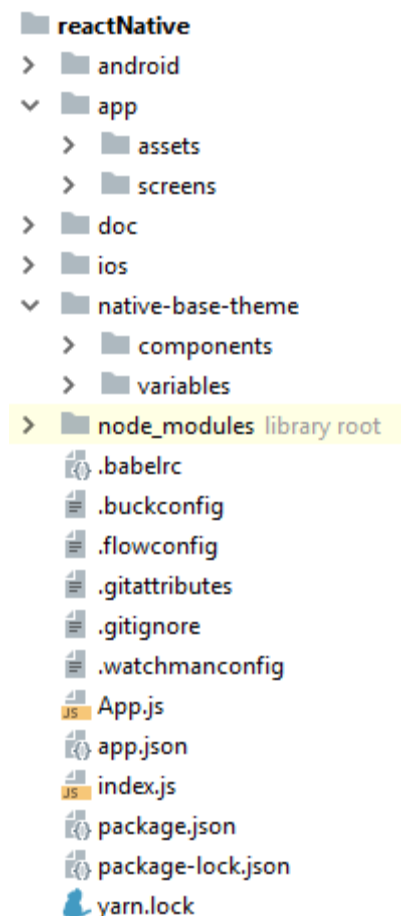


Figure 22 - Structure d'un projet ReactNative

Dans cette structure, on peut trouver deux dossiers « android » et « ios ». Ces dossiers contiennent les fichiers spécifiques aux deux plateformes et nécessaires au bon fonctionnement de l'application une fois compilée. On y trouve le plus souvent des fichiers de configuration, mais également les icônes de l'application ou du code natif servant à importer des modules.

On trouve ensuite un dossier « node\_modules » contenant les modules NodeJS installés dans le projet. Ces modules sont des bibliothèques qui vont permettre d'accéder à certaines fonctionnalités, permettant au développeur de gagner du temps en n'ayant pas à les développer.

Le dossier « app » est un dossier que j'ai créé afin d'y placer les éléments et le code qui serait commun aux différentes plateformes. On y trouve ainsi un dossier « assets » contenant



les ressources du projet, ici des images, et un dossier « screens » contenant le code des écrans de l'application et leurs composants respectifs.

Enfin, le dossier « native-base-theme » est un dossier créé à la suite de l'installation du module « native-base ». Ce module permet d'accéder à des composants plus poussés que les composants de base de ReactNative et dont le style visuel est plus acceptable. Ce dossier permet donc de modifier l'apparences des composants du module « native-base » et m'a notamment servi à modifier les couleurs de ces derniers afin de coller avec la charte graphique que nous nous étions fixés pour ce projet.

### Utilisation de la technologie

ReactNative est plutôt bien documenté et la mise en place d'un projet utilisant cette technologie n'est pas très complexe.

### Le problème « Expo »

Un des premiers problèmes auquel j'ai eu à faire face a été « Expo ». « Expo » est un outil construit autour de ReactNative et qui est sensé rendre le projet plus portable en permettant de compiler une application ReactNative sans passer par AndroidStudio et ses outils ou XCode. ReactNative a ainsi choisi d'intégrer « Expo » par défaut à ses projets. Le problème est que j'ai personnellement eu de nombreux échecs lors de la compilation de l'application dus à « Expo ». Après recherche sur Internet, il s'avère que cette technologie n'est pas complètement stable et que je n'étais pas le seul à avoir des problèmes avec la compilation, ce qui pousse de nombreuses personnes à choisir de développer en se passant d'« Expo ». Suivant leurs conseils et ayant eu moi-même des problèmes avec cette technologie j'ai donc choisi de développer l'application sans « Expo ».

### Création de l'application sans « Expo »

Afin de créer une application sans « Expo » il suffit d'utiliser la commande :

```
react-native init monApplication
```

### Création d'un composant

La création d'un composant est similaire à celle de « React » (voir ci-dessus).

### Interactions entre les composants

Une fois encore « ReactNative » est très proche de « React » dans sa syntaxe, et le principe d'interactions entre père et fils reste le même (voir ci-dessus).

### Actualiser l'affichage d'un composant

Comme React, ReactNative utilise un système d'état. L'état va être une variable contenant les différentes variables propres à la vue. Pour modifier cet état, on doit faire appel à la méthode « setState » qui va non seulement changer la variable d'état mais également lancer l'actualisation de la vue.

### Mise en place des adaptations

Afin de mettre en place les adaptations choisies au sein de l'application, j'ai dû avoir accès à la position de l'utilisateur. Pour cela ReactNative intègre une méthode permettant d'accéder aux coordonnées géographiques de l'utilisateur. Cette méthode se présente comme suit :

```
navigator.geolocation.getCurrentPosition(successCallback(), errorCallback());
```

### Référencement des avantages et des inconvénients

#### Avantages

- Génération d'une application native à partir d'un code commun,
- Application aussi fluide et performante qu'une application directement développée en natif,
- Liberté de la structure de l'application,
- Possibilité d'écrire du JavaScript librement (syntaxe ES6 supportée),
- « Interactions » (passage de données) entre les composants faciles,
- Application en kit, ce qui permet de n'importer que le strict nécessaire,
- Navigation entre les composants très rapide à prendre en main.

#### Inconvénients

- Utilisation de modules « communautaires » qui risquent d'amener des problèmes de compatibilités lors de futures mises à jour.

## Conclusion

### React vs Vue.js

Ce sont deux outils qui laissent aux développeurs une vraie liberté de développement que ce soit au niveau de l'architecture des projets que de la gestion des packages. Une des différences majeures que l'on peut noter entre ces deux outils et la simplicité de prise en main de la technologie. En effet, grâce à l'architecture des fichiers « .vue », le développement est très simplifié et la réutilisation de composants en devient logique. Alors que « React » manque cruellement de cadre pour aider à la maintenabilité de l'application (il est nécessaire de s'auto-cadrer). Cependant, « React » sera plus adaptée pour développer une application lourde, avec de nombreuses pages et composants, puisqu'il y a tout de même une distinction entre la mise en forme et le code du composant contrairement à « VueJS » qui mélange tout pelle-mêle.

### iOS vs ReactNative

La comparaison entre les deux technologies est difficile puisque leurs applications sont bien différentes. En effet, même si elles servent toutes les deux à développer des applications mobiles, le fait que « ReactNative » permette de développer des applications « cross-platform » la différencie énormément du développement natif « iOS ». Cette dimension « cross-platform » est un des grands avantages de « ReactNative » puisqu'il permet de faire aussi bien que le développement natif « iOS » et natif « Android » réunis. iOS natif a cependant pour avantage un accès plus simple à certaines fonctionnalités, notamment les capteurs. « ReactNative » est aussi dépendant pour certaines fonctionnalités de modules « communautaires », ce qui risque de poser des problèmes de compatibilité sur le long terme.

## Installation

Lien du dépôt git : <https://github.com/PierreRainero/PathPartout>

### React

Le projet « React » ne contient pas de clef d'API Google Maps il faut donc rajouter un fichier « .env » à la racine du projet la contenant :

```
// .env
REACT_APP_GOOGLE_MAPS_API_KEY='my API Key'
```

On peut alors lancer l'application en mode développement (déploiement non optimisé et recompilation dynamique lors d'une modification d'un fichier « .js ») avec la commande :

```
npm start
```

On peut générer une version *static* de l'application. Celle-ci est optimisée et doit être utilisée pour la production (il est nécessaire de disposer d'un serveur pour utiliser l'application ainsi déployée).

```
npm run build
```

## VueJS

Le projet « VueJS » utilise également « npm » (et contient une clef d'API valide), on dispose donc des deux mêmes commandes que pour React) :

```
npm start
```

```
npm run build
```

## iOS

Une clef d'API Google est inclus dans le projet, le déploiement avec « iOS » se fait avec la commande suivante :

```
pod install
open PathPartout.xcworkspace
```

## ReactNative

Une clef d'API Google Maps est fournie dans ce projet, vous pouvez la changer dans le fichier « /android/app/src/main/AndroidManifest.xml » :

```
<meta-data android:name="com.google.android.geo.API_KEY"
  android:value="[cléAPI]"/>
```

On peut ensuite déployer l'application dans un environnement de test avec la commande suivante, pour « Android » :

```
react-native run-android
```

Pour « iOS » :

```
react-native run-ios
```

Et dans un environnement de production pour « Android » :

```
cd android
./gradlew assembleRelease
```

## Table des figures

Figure 1 - Adaptation réductive sur la page d'accueil de l'application React .....	4
Figure 2 - Adaptation réductive sur la page d'accueil de l'application VueJS .....	4
Figure 3 - Adaptation dé-portative de la page des trajets sur l'application React.....	5
Figure 4 - Adaptation dé-portative de la page des trajets sur l'application VueJS .....	6
Figure 5 - Création d'un trajet sur l'application React.....	6
Figure 6 - Création d'un trajet sur l'application VueJS .....	7
Figure 7 - Vue listant les trajets sur l'application iOS (à gauche) et ReactNative (à droite) .....	8
Figure 8 - Carte interactive de l'application iOS (à gauche) et ReactNative (à droite) .....	8
Figure 9 – Approche d'un point d'intérêt sur iOS (à gauche) et ReactNative (à droite) .....	9
Figure 10 - Développement par composant.....	10
Figure 11 - Architecture proposée par "create-react-app" .....	10
Figure 12 - Architecture adoptée .....	11
Figure 13 – Ensemble des composants réalisés .....	12
Figure 14 - Illustration d'une navigation entre composants .....	15
Figure 15 - Représentation simplifiée du mécanisme d'état .....	16
Figure 16 - Exemple type d'un fichier Vue.js .....	17
Figure 17 - Communication entre les composants.....	18
Figure 18 - Storyboard .....	22
Figure 19 - Relation entre vues depuis storyboard .....	22
Figure 20 - Connexions vues - controller depuis storyboard .....	22
Figure 21 - Tab Bar Controller .....	23
Figure 22 - Structure d'un projet ReactNative .....	26