Rapport sur le projet “Tradambar” - Adaptation des interfaces

8 novembre 2018

Renaud Costa

Alexis Deslandes

Josué Collombier

Antoine Lupiac

I - Introduction

Dans le cadre du projet d'Adaptation des interfaces, nous nous sommes intéressé à une communauté méconnue et sous-estimée : les amateurs de Carambar. Créée en 1954, cette confiserie a régalé les papilles de chaque génération, petits et grands.

Nous avons donc décidé de développer Tradambar, une application permettant aux amateurs de Carambar de partager leurs collections de sucreries et d'organiser des échanges avec d'autres utilisateurs. A travers le site internet, les utilisateurs pourront organiser des rendez-vous ou gérer leur profil. L'application mobile accompagnera les utilisateurs pour se rendre aux lieux de rencontres.

À travers ce cadre d'utilisation, notre objectif est de proposer des adaptations d'interfaces différentes selon le contexte d'usage de l'application. Nous nous intéresserons aux adaptations à l'utilisateur, aux dispositifs et à l'environnement, à travers 4 réalisations conçues avec des technologies différentes : une technologie native (Android), une technologie cross-platform (React), un framework Javascript (VueJS) et du Javascript Vanilla. Cette répartition nous permettra de développer 2 versions web et 2 versions mobiles de Tradambar. Nous pourrons ainsi comparer deux à deux les différences de pratique et d'implémentation entre les technologies.

L'objectif de ce projet est de comparer les différentes adaptations réalisées par chaque membre du groupe et de porter notre critique sur les limites et les atouts de chaque technologie. Dans un premier temps, nous présenterons les 4 réalisations, décrivant les choix techniques, les adaptations, et l'expérience que chacun aura eu avec sa technologie. Ensuite nous confronterons deux à deux les applications, afin de faire ressortir les avantages et les limites de chaque technologies, à travers l'expérience que nous avons eu lors du projet.

II - Technologies

II.1 - Javascript Vanilla

II.1.1 Expériences

Ayant eu des expériences avec les trois autres technologies que ce soit dans le cadre des cours ou de mon alternance, j'ai choisi de développer notre application avec la technologie JS basique. Avec l'utilisation des framework de développement mobile comme Ionic ou React Native, on dispose de moyens efficaces et rapides afin d'obtenir une interface agréable et fonctionnelle.

Pour cette partie, l'utilisation de Framework était interdite mais pas celle de librairies.

Par rapport à mon cursus universitaire, je n'ai pas eu beaucoup de cours en lien avec le graphisme ou même l'utilisation poussée des feuilles de style. Il était alors primordial pour moi d'utiliser une librairie m'aidant à avoir des composants déjà stylisés.

J'ai donc choisi la librairie bootstrap, laquelle permet également de disposer ses éléments efficacement dans l'espace de la page web.

II.1.2 Adaptation aux dispositifs

Très rapidement, nous nous sommes concertés avec mes collègues pour décider sur quelles adaptations nous concentrer en fonction de la technologie. JS Vanilla et Vue JS ne pouvant pas accéder aux capteurs du téléphones il a été décidé que nous nous occuperons de l'adaptation aux dispositifs tandis que les technologies Android et React Native seraient utilisées pour l'adaptation à l'environnement.

Lorsque l'on parle d'adaptation aux dispositifs, on pense souvent au concept de "responsive design". L'image ci-dessous représente bien ce concept.

Les différents composants d'une page se voient adapter en taille et en disposition en fonction du dispositif utilisé. Le problème de cette approche est qu'elle est bien trop simpliste. Elle peut être utilisé en partie, mais le problème réside dans l'utilisabilité de l'interface sur les différents environnements. En effet, nous n'avons pas les mêmes habitudes de navigations.

Nos composants changent donc en fonction du dispositif utilisé, nous verrons plus tard quelques exemples.

II.1.3 Début du projet

Au début du développement, comme une page web classique, j'ai créé une page HTML, une feuille de style et un fichier de script.

On récupère les données servant à remplir la page web dans un fichier contenant un objet JSON. Pour tester les capacités de JS il était important de ne pas tout initialiser directement dans la page html mais bien de se servir de javascript pour manipuler le DOM dynamiquement comme le font toutes les pages web de nos jours.

Les premières limites se sont fait remarqués dans ce cadre là. Avec l'utilisation d'Angular dans les autres technologies, on a accès à des instructions (*ngFor) permettant de directement inclure du html dynamiquement dans la page en fonction des données (issues par exemple de base de données).

Pour JS, on doit ajouter ces éléments au DOM grâce au script, ce qui est plus long à rédiger et où les erreurs sont plus courantes.

La deuxième principale contrainte de cette manière de procéder, c'est le volume que peut prendre nos fichiers. D'une part, on se retrouve avec des fichiers d'une taille trop importante qui vont devenir peu à peu inmaintenable et d'autre part on a du mal à naviguer entre deux pages html.

En effet, dans le cadre du web dynamique il nous faut échanger des données entre pages web et les manières de le faire ne sont pas aussi simple que sur les autres technologies.

Enfin, entre deux navigations, on perd l'ensemble des chargements de feuilles de style et de script ce qui ralenti considérablement l'apparition des pages.

En prenant en compte ces premières impressions et grâce aux remarques obtenues, j'ai repensé à ma manière de faire et je vous propose le tutoriel suivant.

II.1.4 Tutoriel

Pour ce tutoriel, nous nous appuierons sur l'utilisation des "Web components", cette technologie permet de créer nos propres balises HTML personnalisées et réutilisables.

A la manière des frameworks web, cela permet d'isoler tout le code d'un composant au sein d'un unique package contenant le fichier html, css et JS.

De cette manière, on perd les contraintes vues précédemment : Le code est bien découpé au sein de plusieurs packages. Nous n'avons plus à changer de page web, on se comporte comme une véritable web-app, possédant une unique page web contenant un ensemble de page que l'on rend visible ou non, contenant elles-mêmes des composants web.

Voyons un exemple du projet expliquant la démarche pour implémenter de nouvelles fonctionnalités.

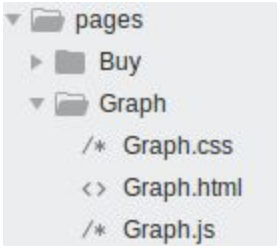
II.1.4.1 Ajout de page sur l'index

```
<body>
    <new-page data="..." id="..."></new-page>
    <link rel="import" href="/pages/new-page/new-page.html">
</body>
```

Au sein du <body> de l'index on insère notre nouvelle page avec son nom de balise customisé. On peut y insérer autant d'attributs que voulu (data,id,...) lesquels seraient nécessaire pour compléter la futur page. Dans le cas de données complexes, on peut utiliser la fonction JSON.stringify() afin de fournir ces données depuis un attribut.

Pour importer cette nouvelle page on utilise la balise link pointant vers le fichier html correspondant.

II.1.4.2 Structure de la page



Comme dit précédemment, pour un composant ou une page, on réunit les 3 sortes de fichier au sein d'un même package.

II.1.4.3 HTML du composant

Commençons par voir le fichier html.

```
<template id="new-page-template">
    <style>
        @import url("https://stackpath.bootstrapcdn.com/bootstrap/4.1.3/css/bootstrap.min.css");
        @import "/resources/styles/hover.css";
        @import "/index.css";
        @import "/pages/Graph/Graph.css";
    </style>

    <div class="d-none d-sm-block">

    </div>

    <div class="lighten-4 d-sm-none">

    </div>

</template>
<script src="new-page.js"></script>
```

Le fichier html d'un composant web se présente de la manière suivante :

- Une balise template qui englobe l'ensemble des composants basique de votre composant. La balise template permet en faites de conserver un ensemble de conteneur et de ne pas l'afficher immédiatement. On décidera par la suite grâce au script de l'afficher sur la page. Ce template possède une balise <style> permettant d'importer toutes les feuilles de style nécessaire au rendu du composant.
- Une balise script permettant d'appeler le script lié au composant.

Comme on peut voir dans l'exemple, il nous suffira d'utiliser les class boostrap (ici présent dans les 2 div) afin de personnaliser la vue du composant en fonction de la taille de l'écran de l'utilisateur.

II.1.4.4 Script du composant

```
currentDocument = document.currentScript.ownerDocument;

class NewPage extends HTMLElement {
    constructor() {
        super();
    }

    connectedCallback() {
        const template = currentDocument.querySelector('#new-page-template');
        const instance = template.content.cloneNode(true);
        this.appendChild(instance);
        /*
        this.attachShadow({mode: 'open'});
        const template = currentDocument.querySelector('#new-page-template');
        const instance = template.content.cloneNode(true);
        this.shadowRoot.appendChild(instance);
        this.addEventListeners();
        */
    }

    attributeChangedCallback(attr, oldValue, newValue) {
        if (attr === "id"){
            const id = newValue;
            //todo
        }
        if (attr === "data"){
            const data = JSON.parse(newValue);
            //todo
        }
    }
}

customElements.define('new-page', NewPage);
```

Première chose à effectuer, récupérer le document courant. En effet, on différencie le DOM général de la page (contenu dans l'index) du DOM de ce composant particulier.

On crée ensuite une classe héritant de HTMLElement en définissant un constructeur vide.

La méthode `connectedCallBack()` est appelé lorsque le composant est connecté au DOM principal. C'est à ce moment là que l'on va vouloir faire apparaître notre composant. Nous allons donc devoir récupérer le contenu du <template> vu précédemment pour l'afficher.

Nous allons avoir 2 cas de figure selon la volonté du programmeur :

- On sait que l'on va utiliser le composant plusieurs fois dans une seule page et on veut éviter les conflits d'interaction (les identifiants et classes de l'ensemble des balises d'un composant sont tous les mêmes).
- On souhaite que le composant qu'on développe interagissent avec le DOM principal.

Dans le premier cas, on définit `connectedCallBack()` avec le code mis en argument dans la capture d'écran. En effet, nous allons devoir utiliser le shadow DOM. Le shadow DOM est

en quelques sortes une balise permettant de cacher un ensemble de balise du reste du DOM. Cela permet d'éviter les soucis d'interaction de balise de mêmes attributs.

L'inconvénient est que le reste du DOM ne peut communiquer avec nous que par l'injection d'attributs directement depuis la balise customisée.

Pour pouvoir sélectionner depuis le script des éléments du DOM du composant on devra appeler `this.shadowRoot` et non pas le document.

Dans le deuxième cas on a une capacité d'échange avec le DOM totale. Comme dit juste avant, un simple appel au document pourra être fait pour manipuler le DOM du composant.

On est ensuite libre de manipuler le DOM comme on le souhaite au sein de cette fonction. Par exemple dans notre cas des articles, on va vouloir adapter le contenu du composant en fonction des données :

```
render() {
    const image = this.getAttribute("image");
    const description = this.getAttribute("description");
    const name = this.getAttribute("name");

    let img = this.shadowRoot.querySelector("#card__img");
    let img_phone = this.shadowRoot.querySelector("#card__img_phone");
    img.setAttribute("src", image);
    img_phone.setAttribute("src",image);

    let desc = this.shadowRoot.querySelector("#card__card-text");
    let desc_phone = this.shadowRoot.querySelector("#card__card-text_phone")
    desc.innerHTML = description;
    desc_phone.innerHTML = description;

    let header = this.shadowRoot.querySelector("#card__card-header");
    let header_phone = this.shadowRoot.querySelector("#card__card-header_phone");
    header.innerHTML = name;
    header_phone.innerHTML = name;
}
```

On a alors qu'à récupérer les éléments correspondants et à changer leur contenu/ les attributs voulus.

L'autre méthode principal à surcharger est `attributeChangedCallback()`, laquelle permet d'exécuter du code lorsqu'un attribut a changé depuis la balise customisée. Dans le cas de données complexes, on pourra appeler JSON.parse(data) afin de récupérer les données sous le bon format et ainsi initialiser le composant.

Enfin, après la déclaration de la classe, on définit la nouvelle balise customisée grâce à son appellation désirée et au nom de la classe.

II.1.5 Outils de développement/test

Ce qui a été pratique avec l'utilisation de Javascript Vanilla, c'est qu'on a besoin de peu d'outils pour le faire fonctionner.

Un simple éditeur de texte tient parfaitement la route, il suffit de prendre la documentation W3C pour connaître les quelques noms d'attributs/de méthodes et de balises dont on se sert et le tour est joué.

Dès qu'on a besoin d'une librairie en particulier, on insère le <link> correspondant dans l'index.html sans avoir besoin de télécharger quoi que ce soit. On a donc une application web extrèmement légère.

Pour les tests, en prenant Google chrome comme navigateur de test, on a accès à quelques fonctionnalités très utiles. D'une part la console pour les outputs, une analyse complète du DOM et des classes de style utilisées mais surtout la possibilité de faire des audits. Cela permet d'analyser les indices de performances pour le chargement de la page, des bonnes pratiques, du contenu hors ligne etc ...

II.1.6 Déploiement

Un des soucis récurrents que j'ai pu avoir au début était des problèmes de correspondance entre mon code et le résultat obtenu sur navigateur. En étudiant le soucis, le problème est une conservation du cache par les navigateurs. J'ai donc décidé d'utiliser un serveur http en ligne de commande. L'une des options permet de réinitialiser le cache à chaque changement des fichiers.

Pour le déploiement, on lance donc la commande "http-server . -c1" à la racine du projet. On obtient un url lequel nous permet de lancer la web-app depuis n'importe quel dispositif du même réseau local.

II.1.7 Capacités d'adaptations

Pour tester les capacités d'adaptations, il suffit d'utiliser un navigateur web ou d'utiliser d'autres dispositifs si l'on en a sous la main. Si l'on a uniquement un navigateur web, il suffit d'activer les options développeurs et de manipuler la taille de la fenêtre pour voir les changements opérés en fonction de la taille du dispositif.

L'un des problèmes qui pourra être observé, c'est lorsque l'on utilise un smartphone de petite taille. En effet, contrairement aux framework comme React Native ou Ionic qui vise à obtenir une vrai application, dans notre cas on souffre de l'utilisation de l'application dans un navigateur. A cause de cela on perd un précieux espace à cause de l'interface de navigation du navigateur.

II.2 - Android Natif

II.2.1 Description de l'environnement

Dans cette partie dédiée à la version Android Natif du projet, nous allons principalement comparer les performances et les différences d'implémentations avec la version React Native. Android Natif définit énormément de concepts propres à son environnement, et le coût de démarrage d'un tel projet est donc très élevé quand on n'a pas de connaissances initialement. Cependant, une fois les concepts compris, le développement de nouvelles fonctionnalités se fait assez aisément et c'est un des gros avantage de cette technologie. De plus, Android dispose d'une très grande communauté de développeurs, ce qui permet d'avoir facilement de l'aide. Enfin, un bon nombre de librairies sont disponibles, et utilisables très facilement.

L'application est composée de deux activités principales : une activité qui présente la liste des carambars disponibles à proximité, et l'activité de navigation GPS qui permet de se rendre au point où se trouve le vendeur du carambar sélectionné. On détaillera ces activités plus bas.

II.2.2 Adaptation aux dispositifs

Pour les parties qui s'adressent uniquement aux dispositifs portables (smartphones, tablettes), nous avons simplement prévu un affichage *responsive* capable de s'adapter aux différentes tailles d'écrans. En Android, j'ai utilisé des *ConstraintLayout*s et des *guidelines* qui permettent de définir la position et la taille des éléments en pourcentage d'écran plutôt qu'en valeurs absolues en *Density independent Pixels*.

II.2.3 Adaptation à l'environnement

Les parties Android Natif et React Native se démarquent des autres parties par leur grande possibilité d'adaptation à l'environnement, notamment grâce à l'utilisation de capteurs. Nous avons décidé d'utiliser deux capteurs : le **capteur de lumière** et le **capteur GPS** pour remplir au mieux la fonction "Guider l'utilisateur jusqu'au point de rendez-vous".

Premièrement, dans beaucoup de smartphones, la luminosité de l'écran dépend du niveau de lumière ambiante, afin d'offrir à l'utilisateur un meilleur confort visuel et d'éviter d'éventuels maux de tête. Nous avons décidé de pousser cette réflexion encore plus loin en proposant un mode nuit beaucoup plus sombre pour l'application, qui s'active automatiquement à l'arrivée de la nuit, et qui se désactive si la lumière ambiante le permet. Puis, pendant que l'utilisateur se rend au point de rendez-vous, si le mode nuit est activé, un bouton pour activer directement le flash du téléphone sera affiché sur la carte, afin de

permettre à l'utilisateur de rester sur l'application quand il active son flash s'il arrive dans un endroit trop sombre.

Ensuite, le capteur de GPS est utilisé dès l'écran qui liste les carambars. En effet, les carambars sont classés par distance par rapport à l'utilisateur afin de montrer en priorité ceux qui sont les plus accessibles. De plus, une barre de défilement permet de définir un rayon de recherche autour de soi et filtrer dynamiquement les carambars. Enfin, quand on clique sur un carambar, une activité qui implémente une carte Google Maps montre la position de l'utilisateur, ainsi que la position du vendeur de carambar. L'écran zoome alors automatiquement autant possible, tant que les point de départ et d'arrivée sont sur l'écran afin d'avoir la meilleure visibilité possible. De plus, le trajet le plus court se dessine alors entre les deux marqueurs. Comme sur l'écran précédent, quand la luminosité devient trop faible, l'affichage de la carte change automatiquement pour un mode sombre adapté à la nuit.

II.2.4 Tutoriel

Initialisation du projet

Pour démarrer le projet, j'ai créé un projet sur Android Studio, basé sur Android version 27, ce qui était compatible avec le smartphone que j'avais à disposition. Puis je suis parti d'une activité vide pour avoir plus de contrôle sur tout ce que j'allais faire. Une fois cela fait, tous les fichiers nécessaires à exécuter l'application sont créés :

- *AndroidManifest.xml* : définit tous les composants que l'on va utiliser dans l'application (activités, services, receivers…) ainsi que les permissions dont on va avoir besoin. On y définit aussi quelle est l'activité d'accueil de l'application.
- *build.gradle* : permet de définir tous les paramètres de construction de l'application, tels que la version d'Android, les *repositories* et les librairies utilisées.
- *MainActivity.java* : définit le comportement de l'activité principale à la résolution des événements classiques (au démarrage, à la pause, à la destruction etc.) ainsi que les fonctions qu'elle va appeler, c'est l'implémentation de l'activité.
- *activity_main.xml* : définit la structure visuelle de l'activité, les éléments qui en font partie, leurs positions et leurs attributs.

J'ai alors commencé à définir le modèle de données en définissant une classe Carambar qui contient les attributs d'un carambar : nom, description, *lien vers l'image* et coordonnées GPS. De plus, je l'ai définie comme implémentant l'interface *Serializable*, ce qui permettra de transporter plus facilement ses instances par la suite.

Dans le contexte de l'application, j'ai décidé d'entrer des données de test manuellement, alors pour pouvoir insérer des images il m'a fallu importer les images relatives aux carambars en tant que *Drawable*, ce qui permet d'y avoir accès dans l'environnement de développement, et aussi d'avoir à disposition plusieurs versions de l'image suivant la résolution du terminal. Le *lien vers l'image* mentionné plus haut est donc en fait un entier qui est l'id du drawable correspondant.

Affichage d'une liste

Une fois le modèle défini, j'ai pu commencer à implémenter la liste de carambars dans l'activité principale. Pour ce faire, j'avais d'abord implémenté une *ListView*, que j'ai rapidement transformée en *RecyclerView* qui présente plus d'avantages que la précédente comme par exemple un défilement plus fluide, une structure plus complète (utilisation de *LayoutManager*, *ItemDecoration et ItemAnimator*) ainsi qu'une manière plus simple de gérer les notifications et les modifications de sets de données. C'est cette dernière fonctionnalité qui m'a particulièrement intéressé, en prévision de la fonctionnalité de tri dynamique par rapport à la distance.

Pour implémenter la *RecyclerView*, il m'a fallu tout d'abord définir un nouveau *layout générique* qui représente une seule entité de la liste, tout en définissant les *id* de tous les composants afin de pouvoir y accéder depuis l'implémentation de la RecyclerView. J'ai aussi préparé un bouton pour chaque ligne, qui va servir par la suite à changer d'activité tout en gardant les informations relatives à sa ligne. Ensuite, il m'a fallu définir la classe CarambarAdapter qui hérite de la classe *Adapter* de RecyclerView. Cette classe contient la liste des données que l'on veut afficher et va définir comment remplir le layout générique avec les données de la liste. Enfin, dans l'activité principale, il faut définir l'Adapter du RecyclerView : une instance de CarambarAdapter, ainsi que son LayoutManager : un simple LinearLayoutManager fait l'affaire.

Mode nuit

Une fois cela fait je me suis penché sur la partie "mode nuit" de l'application. Pour pouvoir implémenter ce mode, j'ai utilisé les *thèmes*. Les thèmes sont définis dans le fichier *styles.xml* qui se trouve dans le package *res*, tout comme les *drawables*, les *strings* constantes etc. Ce fichier permet de définir les valeurs d'un ensemble d'attributs. A noter que tous les attributs que l'on souhaitera modifier de la sorte devront figurer dans le fichier *attrs.xml* dans une balise <*declare-stylable*>. Ainsi, j'ai pu définir la couleur de texte ainsi que la couleur de fond que je souhaitais pour chaque thème.

Je devais alors commencer à utiliser le capteur de luminosité du téléphone. La mise en place de l'utilisation de ce capteur a été très rapide à ma grande surprise. Il suffit d'utiliser un SensorManager qui permet de récupérer un objet Sensor de type *TYPE_LIGHT* à la

création de l'activité. Ensuite, il suffit de transformer la signature de la classe de l'activité pour qu'elle implémente l'interface *SensorEventListener* et de définir les 3 méthodes requises. La seule qui nous intéresse dans le cas présent est *onSensorChanged(SensorEvent event)* qui s'exécute en théorie dès qu'un capteur quelconque détecte un changement. En réalité cette fonction est appelée périodiquement et systématiquement à la condition qu'un *listener* de capteur soit enregistré. On utilise alors la méthode de SensorManager qui permet d'enregistrer le listener et il ne nous reste plus qu'à définir le comportement de la fonction onSensorChanged. Pour gérer le mode nuit j'ai décidé d'utiliser un booléen *static* qui indique si le mode nuit est activé ou pas. Rendre cette variable static permet de garder sa valeur même quand on relance l'activité. Alors, quand l'activité démarre, suivant la valeur du booléen, on appelle la fonction *setTheme(int)* de l'activité en passant en paramètre l'id du thème correspondant (par exemple *R.style.ActivityTheme_Primary_Base_Dark* pour le mode nuit).

Enfin, dans onSensorChanged, on définit une valeur en *lux* à partir de laquelle on passe en mode nuit (threshold bas : **10 lux**) et une valeur à partir de laquelle on repasse en mode jour (threshold haut : **30 lux**), et on change la valeur du booléen, puis on appelle la fonction *recreate()* de l'activité, qui va remplacer l'instance actuelle de l'activité par une nouvelle. La fonction *recreate()* est d'ailleurs la fonction qui est appelée automatiquement lorsqu'on change l'orientation du smartphone (mode paysage / portrait).

Utilisation de l'API Google Maps

Sur Android Studio, la création d'une nouvelle activité qui se base sur l'API de Google Maps est complètement automatisée, il suffit de cliquer sur "Nouvelle activité" puis de choisir le type Maps. Alors, l'IDE va ajouter les dépendances nécessaires dans le *build.gradle* et va mettre en place une activité simple qui affiche par défaut une carte en plein écran avec un marqueur sur Sydney. Cependant, afin d'avoir ce résultat il faut d'abord faire générer une clé API sur le site Google dédié aux développeurs, et activer les services que l'on souhaite utiliser. J'ai ai eu alors besoin d'activer *Maps SDK for Android* et *Directions API* qui permettent respectivement d'utiliser les services de cartographie de Google et de pouvoir demander le chemin le plus court entre deux positions GPS. Cette clé est à rentrer dans le fichier *google_maps_api.xml* généré au moment de la création de l'activité.

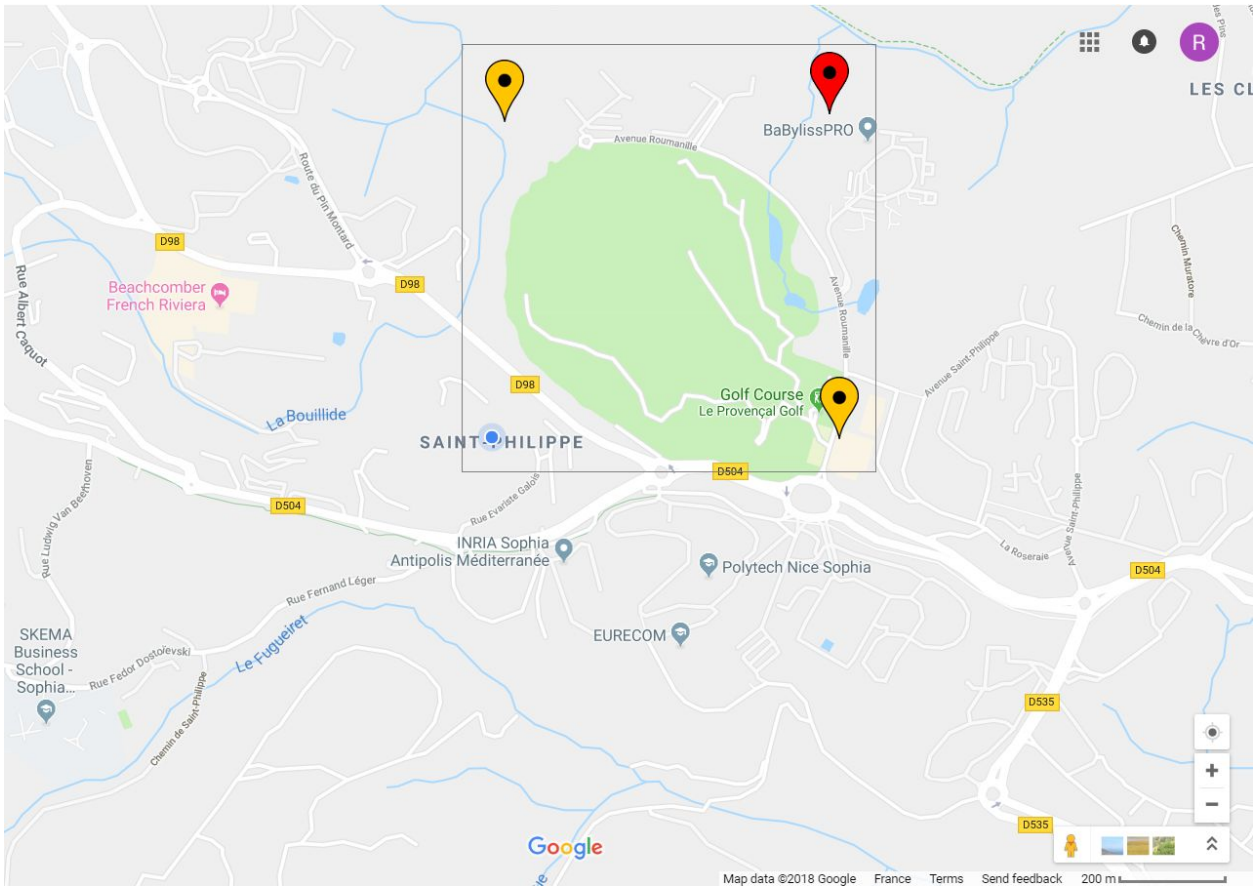
A présent il faut lier les deux activités ainsi présentes dans l'application. Pour cela, on utilisera les *Intents* explicites, qui sont des descriptions d'actions à effectuer et qui peuvent contenir des données grâce à la méthode *putExtra*. La plupart du temps ils sont utilisés pour charger une nouvelle activité, dans laquelle l'Intent pourra être accessible, on pourra donc accéder à ses données. Alors, on va éditer le CarambarAdapter pour ajouter un *onClickListener* aux boutons des lignes de la liste, qui va comme son nom l'indique, exécuter des actions lorsqu'on presse le bouton. C'est dans cette fonction que l'on va construire un Intent explicite dans lequel on va stocker une version *sérialisée* de l'objet Carambar

correspondant à la ligne du bouton. La sérialisation se fera automatiquement, en effet la méthode *putExtra* de la classe Intent attend un objet qui implémente l'interface *Serializable*. Il faut alors simplement définir la destination de l'intent comme étant la classe *MapsActivity* que je viens de créer, ajouter le carambar à l'intent et appeler *startActivity(intent)*. A présent quand on clique sur le bouton d'un carambar, on est amenés sur l'activité maps qui a accès aux données du seul carambar qui correspond au bouton cliqué. Maintenant, il s'agit de récupérer et utiliser ces informations.

Dans l'activité MapsActivity, on récupère l'intent avec la méthode *getIntent()* de l'activité, puis on en extrait le carambar avec la fonction *getSerializableExtra* de l'intent. On peut donc avoir accès à tous les attributs du carambar, et donc principalement ses coordonnées GPS qui nous intéressent particulièrement. Dès sa génération par Android Studio, l'activité implémente l'interface *OnMapReadyCallback*, qui permet entre-autres l'écrasement de la méthode *onMapReady(GoogleMap googleMap)* qui nous sera extrêmement utile par la suite car elle est appelée une fois que la carte est prête à être utilisée. Le paramètre googleMap représente l'objet GoogleMap sur lequel on va pouvoir appeler toutes nos méthodes pour en définir le comportement. Il est très simple d'afficher la position du carambar, puisqu'on a toutes les informations nécessaires, on appelle la fonction *addMarker* de l'objet googleMaps, en lui passant les coordonnées du carambar et le titre. Pour afficher la position de l'utilisateur, le procédé est un peu plus complexe mais reste très accessible.

L'utilisation de la position de l'utilisateur relève de l'utilisation de données personnelles, il y a donc une sécurité, représentée par une demande de permission. Premièrement on ajoute les permissions relatives à l'utilisation des données de géolocalisation dans le Manifest (*ACCESS_FINE_LOCATION* et *ACCESS_COARSE_LOCATION*), puis dans le *onCreate* de l'activité, on doit vérifier si la permission d'utiliser la géolocalisation a été déjà été donnée, et si non, on doit la demander. Une fois cela fait, on va pouvoir utiliser la position de l'utilisateur à condition de tester à chaque fois si on a bien la permission, sinon le compilateur renverra une erreur. Pour activer la géolocalisation il faut appeler *setMyLocationEnabled(true)* dans le *onMapReady* pour qu'elle soit activée dès le lancement de la carte à chaque fois, mais il faut bien penser à l'appeler également dans la fonction *onRequestPermissionsResult*, qui est appelée quand un utilisateur répond à une demande de permissions. Si on ne le fait pas, lors du premier lancement de l'application par l'utilisateur, la localisation ne sera activée car le *onCreate* aura déjà été résolu, et donc comme la permission n'aura pas déjà été donnée, on n'appellera pas *setMyLocationEnabled(true)*. Une fois tout cela fait, la position de l'utilisateur sera représentée par le rond bleu habituel et mis à jour automatiquement suivant ses déplacements.

A présent, nous avons les points de départ et d'arrivée. Cependant, la fenêtre de départ a un zoom par défaut qui n'est pas très pratique, et la fenêtre est centrée sur la position de l'utilisateur. Pour remédier à cela, il existe une fonction qui permet de faire le zoom et le centrage automatiquement en fonction de deux marqueurs "Sud-Ouest" et "Nord-Est". Le problème est qu'on ne sait jamais à l'avance dans quelle direction se trouve le carambar qu'on essaye de visualiser sur la carte, et si le point "Sud-Ouest" ne se trouve pas au Sud-Ouest de l'autre point, l'application crash. Alors j'ai dû utiliser des marqueurs fictifs invisibles qui représentent les deux autres côté du rectangle dont la diagonale est définie par le segment Position utilisateur → Position Carambar. Sur l'exemple ci-dessous, le marqueur rouge représente le carambar, le point bleu est la position de l'utilisateur et les marqueurs orange sont ceux qui sont calculés à partir des deux points précédents, mais ne sont pas affichés dans l'application.



Ainsi, même si l'utilisateur ne se trouve pas au Sud-Ouest du carambar, on pourra dans tous les cas utiliser le point des quatres qui est le plus au Sud-Ouest pour la borne inférieure, et le point le plus au Nord-Est pour la borne supérieure. A noter que cet exemple est la seule configuration pour laquelle on n'utilise pas les marqueurs fictifs. On peut alors appeler la méthode:

animateCamera(CameraUpdateFactory.newLatLngBounds(bounds, padding)) qui permet de placer la caméra comme montré par le rectangle gris précédemment, avec une animation de déplacement. *bounds* représente les deux bornes (SO et NE) et padding représente l'espacement entre le bord de l'écran et les marqueurs en *dp*. J'ai alors remarqué que quelques fois, l'application crashait car la fonction pour déplacer la caméra était appelée trop tôt : avant que le layout ait fini de charger complètement. J'ai donc dû placer le code qui recentre la caméra dans le *Runnable* de la fonction *Layout.post(Runnable)* afin que ce code ne soit exécuté seulement une fois que le layout est entièrement prêt.

L'utilisateur a donc directement une bonne vision de la position du vendeur de carambar. Il reste donc à afficher le trajet qu'il va devoir parcourir. On va donc se servir du deuxième service que j'avais activé sur ma clé API : **Directions**. J'ai défini une fonction qui permet de générer l'URL à appeler pour recevoir les informations sur le chemin de le plus court entre deux positions GPS. Cette URL est de la forme :

https://maps.googleapis.com/maps/api/directions/json?origin=40.1,40.6&destination=40.2,40.7&sensor=false&key=<key>

Ensuite il faut définir une classe qui hérite de *AsyncTask* qui va qui va se charger de faire une requête GET sur l'URL générée et accueillir les données reçues. Il faudra également des classes pour parser les données JSON envoyées par l'API pour en tirer des données utilisables par l'activité Maps. J'ai trouvé un set de 4 classes qui font ce travail complexe sur un tutoriel (liens dans le code). C'est dans l'une d'entre-elles (*ParserTask*) que le tracé du trajet s'effectue, dans la fonction *onPostExecute* qui va être appelée une fois que la tâche de réception et parsing de données est terminée.

Mode nuit dans maps

Dans cette activité, il a fallu redéfinir comme dans la précédente, la gestion du capteur de lumière pour changer le thème de l'activité. La différence majeure est que cette fois-ci, on ne peut pas se permettre de *recreate()* l'activité, puisqu'on va recharger la carte à chaque changement de mode, et donc revoir l'animation de zoom à chaque fois, ce n'est pas envisageable. Par chance, l'API Google Maps prévoit une fonction qui permet d'appliquer un style à la carte dynamiquement. Étant satisfait du thème par défaut de la carte pour le mode jour, il m'a juste fallu définir le style du mode nuit. Le style des cartes doit se trouver dans un fichier JSON dans le dossier res/*raw* et il a bien entendu un format exact attendu. Pour obtenir un fichier qui a le bon format, Google met à disposition un site web (*https://mapstyle.withgoogle.com/*) qui permet d'éditer en ligne toutes les propriétés de chaque élément d'une Google Map, et qui génère le JSON correspondant au style alors créé. Le principe est le suivant, quand on applique un style à une carte, chaque propriété a la valeur par défaut, à moins qu'elle soit écrasée. Alors, il suffit de définir seulement le style

des éléments que l'on veut changer. Le fichier qui permet de repasser au style par défaut contient donc juste un *JSON Array* vide. Une fois les deux styles définis, il faut se rendre dans la fonction *onSensorChanged* de l'activité Maps, et d'appliquer le style correspondant via la fonction *setMapStyle*.

Dès le début du projet, nous avions décidé d'implémenter une fonctionnalité qui permet d'activer le flash sans quitter l'application, c'est dans cette activité que nous allons le faire puisque quand l'utilisateur est sur cette activité, il est potentiellement en extérieur, en train de se déplacer, possiblement dans le noir. Alors, il faut définir premièrement un *FrameLayout* qui englobe le layout déjà existant, et qui va permettre de superposer des éléments par dessus la carte. Puis on va ajouter un bouton sur ce layout. On édite à nouveau *onSensorChanged* pour définir la visibilité du bouton suivant la luminosité (*setVisibility*). J'ai également posé une autre condition pour améliorer le confort utilisateur : si le flash est activé et que l'application passe en mode jour, le bouton reste alors disponible tant que le flash est allumé pour permettre à l'utilisateur de l'éteindre directement. Ensuite, avant de définir le comportement du bouton, on doit s'occuper des permissions, alors comme pour la géolocalisation, on ajoute dans le Manifest la ligne correspondante, en l'occurence : *<uses-feature android:name="android.hardware.camera" />*. Il faut garder en tête que cette balise, tout comme la ligne qui définit la version minimum de SDK requise, pose une condition nécessaire sur la présence de ladite *feature* au niveau du Google Play Store, ce qui réduit la part d'utilisateurs qui verra l'application apparaître sur le store. Il ne faut donc pas en abuser si ce n'est pas complètement bloquant. Dans notre cas, on pourrait se passer de la fonction flash, mais on considère que de nos jours, la quasi totalité des terminaux possèdent une caméra et un flash.

A présent, dans l'activité, on peut définir un *CameraManager* qui va, tout comme le SensorManager utilisé précédemment, utiliser la fonction *getSystemService* pour récupérer le *CAMERA_SERVICE* qui va nous permettre d'utiliser le flash du téléphone entre-autres. On va donc pouvoir définir le *onClickListener* du bouton en appelant dedans *CameraManager.setTorchMode* avec *true* ou *false* en paramètre pour activer ou désactiver le flash. Accessoirement, j'ai aussi changé légèrement la couleur du logo du bouton pour que l'utilisateur puisse savoir si son flash est allumé ou non, dans les cas où il ne fait pas extrêmement sombre.

Ressources

Afin de suivre les normes Android, j'ai défini toutes les chaînes de caractères constantes dans le fichier *strings.xml* du dossier *res*. Cela permet entre-autres de faciliter la traduction de l'application dans d'autres langues. J'ai également défini le valeur du threshold de niveau de lumière dans le fichier *integers.xml* comme ça il est très facile d'accès et modifiable plus rapidement et efficacement, puisque partout où la variable est utilisée, cette valeur est

référencée. Faute de logo, j'ai gardé le logo par défaut pour l'application, mais on aurait pu définir une image dans le dossier *mipmap* et la référencer dans le Manifest comme étant le logo de l'application, que l'utilisateur verra sur son smartphone dans sa liste d'applications.

Conclusion du tutoriel

Au final, je me suis rendu compte qu'implémenter une fonctionnalité impliquait souvent d'utiliser de nouveaux concepts et j'ai eu besoin de passer par de nombreuses phases d'apprentissage et de tests, mais je me sens maintenant très à l'aise avec beaucoup de ces concepts que j'ai eu l'occasion d'expérimenter. J'ai vraiment senti que mon efficacité a augmenté tout au long du projet, au fur et à mesure de mon apprentissage.

II.2.5 Outils de test

J'utilise personnellement un iPhone, et donc au début du développement j'ai dû utiliser Genymotion couplé avec une image de smartphone sous Android 26. Très rapidement et sans surprise, j'ai été limité par les possibilités offertes par l'émulateur. En effet, si l'émulateur permet bien de tester les fonctionnalités et l'affichage des *layouts*, il ne permet pas la simulation des capteurs du téléphone. Par la suite, j'ai donc uniquement utilisé un Samsung Galaxy S8 fourni par l'école, ce qui m'a permis de tester l'application en conditions réelles. Par exemple, afin de trouver une valeur optimale pour le threshold de lumière, je suis allé tester l'application en conditions réelles : en extérieur et en pleine nuit, pour faire en sorte que, par exemple, la lumière produite par un simple lampadaire ne suffise pas à passer en mode jour.

II.2.6 Exécution et déploiement

Pour exécuter le projet sur un smartphone, il faut tout d'abord activer le mode développeur dans les paramètres du téléphone. Sur la gamme des Galaxy S il faut aller dans paramètres > Système > A propos du téléphone et appuyer 7 fois sur le numéro de version. Les options développeur sont maintenant disponibles et il faut activer le débogage USB. A présent, le téléphone sera détecté par Android Studio et on pourra le choisir dans la liste des dispositifs quand on exécute l'application avec 

Pour le déploiement sur Android Studio, il faut tout d'abord enlever tous les Logs qu'on effectue, enlever l'attribut *android:debuggable* du manifest si on l'a ajouté et définir les attributs *android:versionCode* et *android:versionName*.

Ensuite il faut aller dans le menu *Build* > *Build Bundle(s) / APK(s)* > *Build APK(s)* pour construire la version exécutable du projet au format .apk, lisible par les smartphones. Enfin il faut télécharger l'apk sur le store visé.

II.2.7 Test des capacités d'adaptations

Premièrement, pour les adaptations aux dispositifs, il faut essayer d'exécuter l'application sur un autre smartphone pour voir si l'affichage est bien responsive. Pour les adaptations à l'environnement, on peut tester le mode nuit de l'application en l'utilisant le soir, ou tout simplement en cachant le capteur de lumière pour simuler une baisse de lumière et voir si l'application passe bien instantanément en mode nuit. Pour la géolocalisation, on peut lancer l'application à partir de plusieurs endroits différents pour voir si les carambars sont bien triés dans un ordre différent.

II.3 - React Native (Cross Platform)

II.3.1 Description de l'environnement

Le but de cette partie est de comparer l'utilisation et les performances du frameworks cross platform *React Native* avec celles d'*Android* natif.

Ayant plus souvent travaillé par le passé sur Angular et n'ayant jamais approché *React Native*, il m'a fallu un petit temps d'adaptation à la logique de React Native. Toutefois, ce framework est très utilisé ce qui permet d'obtenir aisément de l'aide sur internet, et possède énormément de librairies proposant des composants plus ou moins complexe ce qui fait gagner du temps, bien que l'on perde le contrôle du composant.

La logique derrière React Native repose sur une utilisation intensive des composants. Chaque composant possède une partie JSX (traduite en JS) et une feuille de style. Sa fonction *render()*, va s'occuper d'afficher le contenu de la page en retournant des balises HTML relative au composant qu'elle décrit.
Il faut savoir qu'un composant est Statefull, il possède un état, des propriétés et un cycle de vie. Il va d'abord être créé, puis l'on pourra travailler dessus une fois monté, à chaque modifications d'états il se réaffichera afin de prendre en compte les modifications d'états, et enfin sera démonté. On peut également lui passer des paramètres lors de son instantiation.

Sur l'exemple qui suit, on peut voir:

- les import des différents composants utilisés
- l'initialisation des variables d'états
- la fonction *render()* qui retourne les tags à afficher
- l'utilisation de la feuille de style du fichier

```
import React from 'react';
import { StyleSheet, Button, Text, View } from 'react-native';

export default class App extends React.Component {
  constructor(props) {
    super(props);
    this.state = {
      activated: this.props.activated
    }
  }

  render() {
    return (
      <View style={styles.container}>
        <Text>Open up App.js to start working on your app!</Text>
        <Text>Changes you make will automatically reload.</Text>
        <Text>Shake your phone to open the developer menu.</Text>
        <Button title="Activate" onPress={() => { this.setState({activated:
true})> </Button>
      </View>

    );
  }
}

const styles = StyleSheet.create({
  container: {
    flex: 1,
    backgroundColor: '#fff',
    alignItems: 'center',
    justifyContent: 'center',
  },
});
```

II.3.2 Adaptation aux dispositifs

Comme expliqué plus plus tôt, pour les dispositifs mobiles nous avons simplement prévu un design responsive s'adaptant aux dimensions des appareils. Pour React Native, j'ai utilisé la propriété flex de CSS afin de placer les différents éléments et aussi faire en sorte qu'un *container* prenne l'ensemble de la place disponible sur l'écran.
La carte *GoogleMaps* recouvre la globalité de l'écran de cette façon et il en est de même pour la liste des produits.

II.3.3 Adaptation à l'environnement

Les deux solutions mobiles *Android* et *React Native* se focalisent principalement sur l'adaptation à l'environnement, et par conséquent la possibilité d'utiliser les capteurs de **luminosité** et de **position.**

Le fil rouge de cette partie était de **"guider l'utilisateur jusqu'au point de rendez-vous"** et de lui "**faciliter la sélection de son produit"**. Deux variables en sont donc ressorties: la variation de lumière et sa variation de position.

La plupart des appareils proposent maintenant de gérer automatiquement la luminosité de l'écran mais très peu offrent un mode nuit. Ici, nous avons décidé de faire varier le style de l'application en fonction de la lumière ambiante. Ainsi, l'application claire en mode jour deviendra sombre en l'absence de lumière mais pourra également redevenir claire s'il y a assez de lumière. La liste des produits changera de thème selon le mode détecté et il en est de même avec la carte permettant d'aller au lieu de rendez-vous.
En cas de mode nuit lorsque la carte est affichée, un bouton permettant d'activer le flash du téléphone apparaît, puis disparaît en coupant automatiquement le flash lorsqu'il y a assez de lumière.

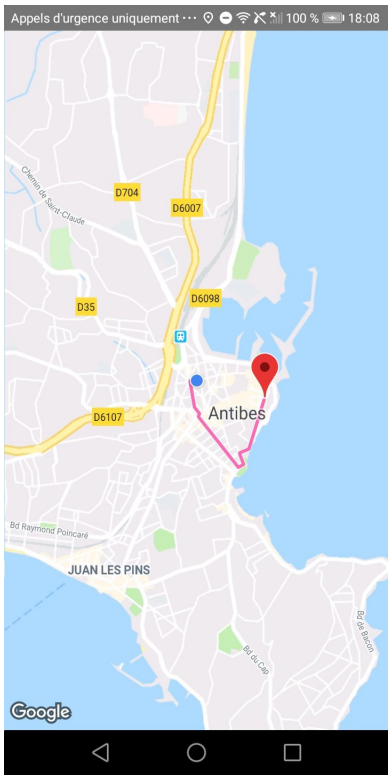
Concernant la variation de position, sur la page de catalogue les produits sont affichés en fonction de la distance entre ceux-ci et l'utilisateur. Un slider permet à l'utilisateur de préciser la distance maximale qu'il peut tolérer afin de filtrer les produits les plus susceptibles de l'intéresser. En appuyant sur le bouton *"Acheter"*, la carte apparaît en indiquant le trajet jusqu'au lieu de rendez-vous. Lorsque l'utilisateur se trouve à moins de 250m de la cible, la carte zoom automatiquement puis dézoom s'il s'en éloigne.

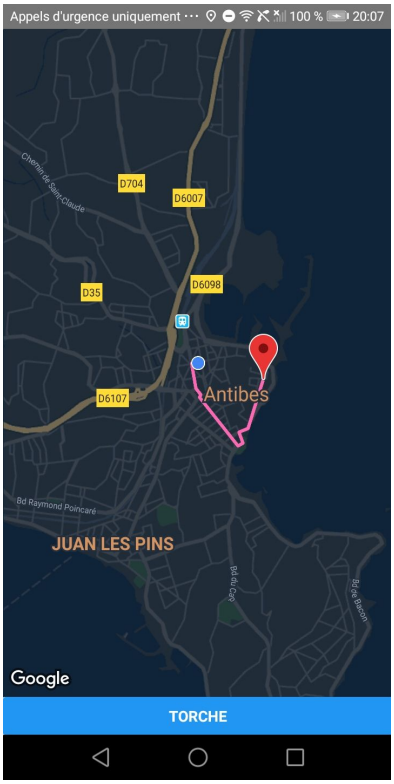
Aperçu de l'application


Appels d'urgence uniquement
100 %
18:08
Chercher dans un rayon de 87 km
Carambar Barbapapa
Antibes 12€/g
566 m
ACHETER
Carambar Magicolor
Biot 10€/g
5368 m
ACHETER
Carambar Magicolor
Biot 10€/g
5368 m
ACHETER
Carambar Mystery
Cannes 6.5€/g
9426 m
ACHETER
Carambar Atomic
Nice 8€/g
17486 m
ACHETER
Carambar Barbapapa
Antibes 12€/g
32855 m
ACHETER
Carambar Barbapapa
Antibes 12€/g
34044 m
ACHETER
Carambar Mystery
Cannes 6.5€/g
41886 m
ACHETER


Appels d'urgence uniquement
100 %
18:08
Chercher dans un rayon de 87 km
Carambar Barbapapa
Antibes 12€/g
566 m
ACHETER
Carambar Magicolor
Biot 10€/g
5368 m
ACHETER
Carambar Magicolor
Biot 10€/g
5368 m
ACHETER
Carambar Mystery
Cannes 6.5€/g
9426 m
ACHETER
Carambar Atomic
Nice 8€/g
17486 m
ACHETER
Carambar Barbapapa
Antibes 12€/g
32855 m
ACHETER
Carambar Barbapapa
Antibes 12€/g
34044 m
ACHETER
Carambar Mystery
Cannes 6.5€/g
41886 m
ACHETER


Appels d'urgence uniquement
100 %
18:09
Chercher dans un rayon de 16 km
Carambar Barbapapa
Antibes 12€/g
566 m
ACHETER
Carambar Magicolor
Biot 10€/g
5368 m
ACHETER
Carambar Magicolor
Biot 10€/g
5368 m
ACHETER
Carambar Mystery
Cannes 6.5€/g
9426 m
ACHETER


Appels d'urgence uniquement
100 %
18:08
D704
D6007
D35
D6098
D6107
Antibes
JUAN LES PINS
Google


Appels d'urgence uniquement
100 %
20:07
D704
D6007
D35
D6098
D6107
Antibes
JUAN LES PINS
Google
TORCHE

II.3.4 Tutoriel

Initialisation du projet

Afin de démarrer le projet, il faut créer un nouveau projet *React Native*. Pour cela, il faut avoir une version de node et npm à jour (sinon allez sur https://nodejs.org/en/download/). Possédant un Iphone 4 mais pas de Mac, j'avais donc en tête de déployer l'application sur *Android*, par conséquent il faut avoir installer une SDK.
A présent, il suffit d'installer React Native CLI avec cette ligne:

```
npm install -g react-native-cli
```

Créer toute la structure du projet *Tradambars* est très simple, il suffit d'exécuter la ligne de commande suivante :

```
react-native init Tradambars
```

On obtient ainsi l'arborescence du projet comportant notamment:

- un module *android*, où l'on pourra modifier les fichiers d'une application Android standard, mais surtout gérer les permissions dans le *AndroidManifest.xml*
- un module *ios*, que l'on pourra modifier de la même façon
- le fichier App.js étant le point de départ de notre application
- le fichier package.json incluant toutes nos dépendances

Suite à un bug également retrouvé dans d'autres groupes, il a fallu installer la dépendance de @babel/runtime et passer sur une version antérieure de React Native (la *55.2*).

La logique de React Native est d'utiliser des ***Composants*** réutilisable dans nos différentes vues, donc autant créer un répertoire *component* dans notre arborescence.
J'ai ainsi créé les composants ***AppNavigator, Catalog, Item*** et ***Map*** que je détaille plus bas.

Routing entre les vues

Afin de se déplacer parmi nos vues, j'ai dû créer un composant ***AppNavigator*** qui est un ***StackNavigator.*** Il s'agit simplement d'une pile, à chaque fois que l'on ira sur une vue elle sera ajoutée au sommet de la pile , et en appuyant sur la flèche de retour en viendra enlever cette vue de la pile.
Pour cela il faut importer **react-navigation**

```
npm install --save react-navigation
```

On définit nos vues en les ajoutant dans le StackNavigator, où l'on peut également préciser la page de démarrage. Ici, headerMode, sert à préciser que l'on ne veut pas avoir de header.

```
const AppNavigator = createStackNavigator({
 Catalog: { screen: Catalog },
 Map: { screen: Map },
},
 {
   initialRouteName: 'Catalog',
   headerMode: 'none'
 }
);
```

A présent il faut inclure ce composant dans la fonction *render()* du fichier App.js

```
 render() {
   return (
     <AppNavigator/>
   );
 }
```

On peut à ainsi naviguer entre notre catalogue et notre carte.

Affichage d'une liste

Une première version de la liste des produits consistait en un ensemble de *cards* qu'il fallait faire glisser pour voir le prochain produit selon le même principe que les cartes Tinder. Cela avait pour but de ne pas surcharger l'écran lorsqu'il y avait un grand nombre d'écran ainsi que d'améliorer l'expérience. Toutefois, pour comparer les performances de React Native avec celles d'Android, il fallait avoir une vue similaire, ce qui explique le changement de design vers une ListView plus traditionnelle.

Ainsi, cette vue est présente dans le fichier Catalog.js. Elle se découpe en deux blocs:

- le slider permettant de choisir le rayon de recherche des produits
- La liste contenant tous les items relatifs aux produits

Les produits sont mockés dans la liste *cards* afin de pouvoir simuler le rendu obtenu avec un web service. Une carte possède:

- le nom du produit
- la ville dans laquelle il se trouve
- son prix
- son image

Le slider est un composant de React Native qu'il faut importer. Il possède une valeur minimale, une maximale, un pas d'incrémentation, un appel de méthode lorsque la valeur du slider change et un autre appel de méthode lorsque la valeur du slider est choisie.

```
import {Slider} from 'react-native';
<Slider
         style={{ width: 300 }}
         step={1}
         minimumValue={0}
         maximumValue={150}
         value={this.state.radius}
         onValueChange={val => {
           console.log("Je change");
         }}
         onSlidingComplete={val => {
          console.log("J'ai été choisie");
         }}
       />
```

La *List* est aussi un composant de *React Native*, elle va itérer sur notre liste de produit pour afficher un composant *Item* (dont j'ai défini le modèle) pour chaque produit.

Le composant *Item* affiche :

- le nom du produit
- l'image du produit
- la ville dans laquelle il se trouve
- son prix
- sa distance depuis l'utilisateur en mètres.
- un bouton "*Acheter*" redirigeant vers le composant *Map* afin de pouvoir le récupérer

En faisant varier la valeur du slider, la liste va être actualisée de façon à filtrer les éléments trop loin. Pour se faire à chaque variation du slider on actualise la position de l'utilisateur puis on filtre notre liste de produit.

Afin d'accéder aux coordonnées GPS il n'y a pas besoin d'importer de librairie ou de composant. Il suffit d'appeler la méthode ***navigator.geolocation.getCurrentPosition()***, je stock ensuite la valeur de position dans une variable puis j'itère sur la liste des produits en calculant la distance grâce à la librairie ***geolib***, c'est à ce moment que je les filtre en ajoutant dans une liste vide la valeur du produit. A la fin du processus, je modifie la variable d'état ***this.state.filtered_list*** afin de réafficher la liste modifiée

Mode nuit

Afin d'implémenter le mode nuit, j'ai mis en place deux feuilles de style: une pour le jour une pour la nuit. elles se trouvent dans les composants les nécessitant. Elles contiennent l'ensemble des propriétés CSS de l'application concernant ces deux modes. On appellera donc ***styleNight.container*** pour afficher les propriété du mode nuit du container et ***styleNormal.container*** pour afficher celles du mode jour.

Maintenant que les styles sont définis il faut pouvoir les alterner. Pour cela, on doit avoir accès au capteur de luminosité du téléphone, ce qui ne fut pas évident et m'a fait perdre plusieurs heures. J'ai utilisé la librairie **react-native-sensor-manager** (https://github.com/kprimice/react-native-sensor-manager), la seule permettant d'accéder au capteur de lumière. La commande **rnpm link** permet de modifier automatiquement les fichiers ***android/settings.gradle, android/app/build.gradle, MainApplication.java et MainApplication.java***, afin de pouvoir accéder au capteur.

Une fois que tout est bien installé, l'utilisation est plutôt simple. On démarre le capteur, puis on ajoute un EventListener sur le capteur de lumière.
Je gère le mode nuit avec un booléen dans la variable d'état que je passe à *true* si la lumière est inférieure à 45 Lux et à *false* si elle est supérieure. Afin de ne pas constamment appeler la méthode *this.setState(),* je vérifie certaines conditions.

```
this.subscription = DeviceEventEmitter.addListener('LightSensor', function
(data) {
   lightdata = data.light;
   if (lightdata < 45 && self.state.nightMode == false) {
     self.setState({ nightMode: true });
     self.setState({style: stylesNight});
   }
   else if (lightdata >= 45 && self.state.nightMode == true) {
     self.setState({ nightMode: false });
     self.setState({style: stylesNormal});
   }

 });
```

J'ai appliqué ce principe sur les composant *App*, *Catalog* et *Item,* ce qui demande beaucoup de ressources puisque chaque item va avoir un EventListener. Sans ce listener, les items ne changeaient pas automatiquement de mode, ils avaient donc le bon mode d'afficher lors de leur création mais le mode ne variait pas. A présent oui, mais cela demande plus de ressources…

On peut maintenant automatiquement afficher le mode nuit en fonction de l'intensité lumineuse.

Utilisation d'une Google Maps

Pour utiliser une Google Map, j'ai encore une fois eu recours à des bibliothèques, à savoir **react-native-maps** (https://github.com/react-community/react-native-maps) pour afficher la carte et **react-native-maps-directions (**https://github.com/bramus/react-native-maps-directions**)** pour afficher le tracer de la direction à suivre. il est également nécessaire de posséder une GOOGLE_MAPS_APIKEY.

A présent il est très simple d'afficher une carte, il suffit d'importer le composant ***Mapview*** et de l'entourer d'une ***View***. Une *Mapview* nécessite quelques paramètres notamment la région à afficher. Pour cela, l'état de mon composant possède un *initialPosition* comprenant la latitude et la longitude de la position actuelle et deux autres variables permettant d'adapter le zoom de la caméra. Un fois que le composant est monté, je défini la fonction *componentDidMount()* dans laquelle je récupère la position actuelle de la même façon que précédemment (***navigator.geolocation.getCurrentPosition()*** *)* afin de l'actualiser, je mets ensuite en place une sorte d'EventListener afin d'actualiser continuellement la position de l'utilisateur avec ***navigator.geolocation.watchPosition()***. Il ne faut pas oublier de supprimer le listener lorsque l'on démonte le composant:

```
componentWillUnmount() {
   navigator.geolocation.clearWatch(this.watchID);
 }
```

C'est tout ce qu'il faut faire pour suivre la position de l'utilisateur

Maintenant pour afficher la direction à suivre il faut utiliser le composant *MapViewDirections*.

```
<MapViewDirections
             origin={{
latitude: this.state.initialPosition.latitude,
longitude: this.state.initialPosition.longitude
             }}
             destination={destination}
             apikey={GOOGLE_MAPS_APIKEY}
             strokeWidth={3}
             strokeColor="hotpink"
             mode="driving"
           />
```

Ici, on redessine constamment la ligne entre l'utilisateur et le point de rendez-vous pour ne pas conserver une traînée derrière son passage.

Lorsque l'utilisateur est à moins de 250m du lieu de rendez vous, l'application zoom automatiquement afin de lui permettre de mieux voir la zone, puis dezoom s'il s'en éloigne.

Mode nuit dans maps

Le composant MapView permet de définir un style particulier pour la carte, il suffit de rajouter l'attribut customMapStyle et de lui donner le JSON de la carte custom.
A notre niveau on sait si nous sommes en mode nuit ou en mode jour donc on pourrait penser qu'il est possible de changer le style de la carte de façon dynamique. Cependant la bibliothèque utilisée ne permet de faire un tel changement. La carte est donc dans le bon mode lors de son instanciation mais ne pourra pas être modifiée même en la redessinant à chaque changement de mode. Ce qui montre la limite de l'utilisation des bibliothèques étant données qu'on peut certes utiliser des composants sans avoir à les coder mais on perd le contrôle de leur comportement.

Toutefois, lorsque l'utilisateur est sur la carte en mode nuit un bouton permettant d'allumer le flash de l'appareil apparaît. Si l'application bascule en mode jour, le flash est automatiquement coupé et le bouton disparaît.
Pour implémenter cela, il faut installer la bibliothèque react-native-torch (https://github.com/ludo/react-native-torch) pour importer le composant Torch. Pour allumer ou éteindre le flash il ne reste plus qu'à appeler ***torch.switchState(boolean)*** avec le booléen à *true* ou *false*. Lorsque l'application détecte qu'on repasse en mode jour on appelle donc cette fonction avec false et on cache le bouton qui l'allumait.

Conclusion du tutoriel

React Native utilise des composants à état qui vont redessiner les parties dépendants de ses variables d'état à chaque fois que la variable est modifiée par la fonction **setState().**
L'utilisation de librairies peut permettre de gagner du temps si on ne compte pas être entièrement maître du composant, de plus des problèmes de dépendances peuvent apparaîtrent en plus du manque de maintenance de certaines librairies.
Cependant, en créant soi même ses composants on peut éviter des duplication de code pour des -futurs- composants ayant les mêmes fonctionnalités et comportements.

II.3.5 Outils de test

Etant l'heureux possesseur d'un iPhone 4 et pas de Mac, j'ai tout d'abord développé à l'aide de l'IDE Visual Studio Code et de l'émulateur Genymotion en visant Android. Cela m'a permis de mettre en place la structure du projet et d'implémenter la carte, mais dès lors qu'il a fallu utiliser le capteur de lumière je me suis retrouvé limité par l'émulateur. L'école a alors gentiment prêté un Samsung Galaxy S8 à Renaud et à moi. Ne pouvant pas toujours nous le partager j'ai donc acheté un téléphone sous Android afin de continuer d'avancer dans le projet. J'ai donc pû faire des tests en lumière basse et élevé pour ajuster le seuil du mode nuit du capteur de lumière.

II.3.6 Exécution et déploiement

Pour exécuter l'application, il faut tout d'abord activer le mode développeur sur son appareil puis télécharger le projet.

On se plaçant dans le dossier ReactNative/Tradambars, il faut exécuter les commande suivantes après avoir branché son téléphone à l'ordinateur.

```
adb reverse tcp:8081 tcp:8081
npm install
react-native run-android
```

II.3.7 Test des capacités d'adaptations

Pour tester les capacités d'adaptations, on peut dans un premier temps s'amuser avec l'application en cachant le capteur de lumière pour vérifier que le mode nuit s'active, puis remettre de la lumière pour voir que l'on repasse bien en mode jour. On peut sélectionner un produit et se déplacer pour voir si la carte zoom quand on est proche de la cible et dézoom quand on s'en éloigne. On peut également tester le tri des articles en fonction de la position en bougeant.

II.4 - VueJS

II.4.1 Choix techniques

Dans cette partie nous allons détailler le développement de l'application Tradambar avec le framework VueJS. Cette technologie réactive suscite beaucoup d'intérêt pour les développeurs front-end, en se revendiquant très accessible et performante. Adaptées aux Single Page Applications, VueJS sera donc expérimenté pour ce projet.

Au cours de stages et de projets, j'ai beaucoup été amené à utiliser Angular 2, ce qui est présenté comme un avantage pour la prise en main de VueJS. L'objectif de la réalisation est de produire un site web monopage dynamique permettant de répondre aux besoins d'adaptations.

II.4.2 Organisation du projet

Le framework VueJS propose une documentation et des tutoriels bien fournis. A partir du tutoriel de base, on a très vite entre les mains une application web monopage dynamique.

VueJS propose différentes solutions pour gérer les composants. Une architecture semblable à Angular est possible, en définissant chaque composant et sa logique dans un fichier séparé. Toutefois, cette pratique nécessite une *transpilation* et est plus adaptée aux projets à grosse échelle.

Les composants sont donc définis par des x-templates dans le HTML, que l'on peut réutiliser à souhait. Une fois que l'on déclare le template comme un composant Vue, on peut facilement l'utiliser dans le HTML. De plus, aucune logique n'est associée au template, ce qui permet de donner un comportement sur-mesure à chaque utilisation d'un composant.

L'affichage dynamique de boîtes de dialogues est facilité grâce au rendu conditionnel de VueJS. De plus, l'architecture Modèle-Vue-Vue modèle de ce framework permet de lier facilement les données et méthodes au modèle HTML.

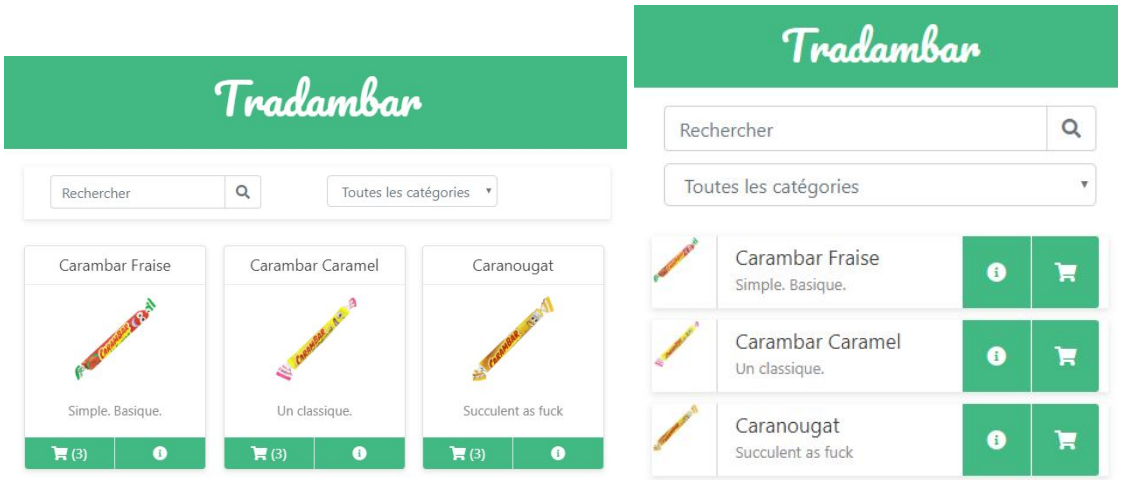
Afin de répondre aux contraintes du responsive design, j'ai choisi d'utiliser la librairie Bootstrap, par familiarité.

II.4.3 Adaptation aux dispositifs

Comme détaillé dans la partie "Javascript Vanilla", un site web peut difficilement s'adapter à l'environnement sans capteurs. Nous nous sommes donc essentiellement concentré sur des adaptations aux dispositifs, à travers des pratiques de responsive design.

Le responsive design opère de différentes façons sur la page. Premièrement, on retrouve un système de grille présentant les items de la boutique de Carambar. En fonction de la largeur de l'écran, et avec une habile utilisation de *media queries*, le nombre d'items par ligne s'adapte pour garder les informations centralisées sur la page.

Deuxièmement, le site web possède une toute autre interface sur mobile. Au-delà d'une simple adaptation de la taille et de la disposition des éléments, l'application propose un layout différent, optimisé pour une navigation mobile. En effet, les items sont présentés linéairement, avec plus de densité, et les boutons d'actions se positionnent sur le côté droit, pour être à proximité du pouce d'un utilisateur (droitier).



(1) Capture d'écran : interface Navigateur versus interface Mobile

La navigation se fait à travers des boîtes de dialogues (ou "modals"). Les propriétés réactives de VueJS facilitent cet affichage dynamique. Les interfaces des modals s'adaptent également au dispositif, permettant de simuler une navigation "native" sur mobile (à la manière d'une nouvelle fenêtre qui pop).

L'application propose aussi une fenêtre "Détails", qui permet d'obtenir plus d'informations autour du carambar. Cette interface propose des données variées : images, graphiques, textes, map... L'enjeu de l'adaptation de cette interface est de conserver lisibilité et navigation peu importe le dispositif. La disposition et la largeur des éléments change donc en fonction de la taille de l'écran, et une barre de boutons permet de filtrer les informations à afficher. Ainsi, la navigation sur mobile n'est pas polluée par la densité de contenu.

L'adaptation aux dispositifs est donc appliquée à différents niveaux de l'application : disposition des éléments, densité du contenu, optimisation de la navigation. Ces pratiques sont facilitées par le framework VueJS, avec l'utilisation de composants et le rendu conditionnel qu'il propose.

II.4.4 Adaptation à l'utilisateur

En plus des adaptations précédemment décrites, j'ai choisi d'implémenter une adaptation simple en fonction du type d'utilisateur. En effet, au moment où un utilisateur prend rendez-vous pour acheter des carambars, l'interface lui propose de se rendre sur l'application, si il est connecté. Sinon l'interface l'invite à télécharger l'application mobile et à se créer un compte.

L'adaptation à l'utilisateur proposée est certes minimaliste, mais permet de souligner qu'il est possible de distinguer les utilisateurs et ainsi de s'adapter.

II.4.5 Tutoriel

Nous allons maintenant décrire les différentes étapes de développement d'un morceau d'interface en utilisant le framework VueJS.

La mise en place d'un projet Vue est très simple, puisqu'il suffit d'un seul fichier. Ici j'ai choisi de séparer le script javascript de la partie HTML.

```
let app = new Vue({
   el: '#att',
   data: {...},
   methods: {...}
});
```

Cette déclaration permet d'être prêt à se lancer dans le grand bain. On définit ici les attributs et la logique de l'application. Nous aurons pour cet exemple une liste de Carambar en attribut.

La prochaine étape consiste à implémenter le template du composant que l'on souhaite afficher, comme suit :

```
<script type="text/template" id="item-template">
    <div class="card">
        <div class="card-title">
            <slot name="title">
                default name
            </slot>
        </div>
```

```
        <div class="card-body">
            <div class="card-text">
                <slot name="desc">
                    default desc
                </slot>
            </div>
        </div> ...
```

Les balises <slot> nous permettrons ainsi de *bind* les données aux emplacements que l'on souhaite. Le template ne définit que la structure HTML du composant, ce qui permet de ne pas avoir à lui associer de comportement directement.

Par la suite, nous allons utiliser la directive *v-for* de VueJS (équivalent au *ngFor* d'Angular) pour faire le rendu d'une liste d'éléments :

```
<ul class="items">
   <li v-for="carambar in carambars">
       <item>
           <div slot="img">
              <img class="card-img-top" v-bind:src="carambar.image"/>
           </div>
           <div slot="title">
               {{ carambar.name}}
           </div>
           <div slot="desc">
               {{carambar.desc}}
           </div>...
```

Nous obtenons ainsi l'affichage d'une liste d'items. Cette pratique de "*slot*" permet une utilisation très souple et modulaire des composants, et donc une bonne réutilisation. Ces avantages s'appliquent aussi aux feuilles de styles : on peut donner un style "global" au composant et styliser plus précisément les *slots* avec des attributs de classe.

Ce tutoriel permet de mettre en avant la simplicité de prise en main de la technologie, ainsi que la souplesse qu'elle offre. Avec de l'expérience en intégration web, on ne rencontre quasiment aucune difficulté pour implémenter un site web dynamique "simple".

II.4.6 Déploiement, tests et conclusion

Avec l'inclusion directe du framework dans une balise <script> dans le code HTML, toute notre application est contenue dans 3 fichiers : un fichier HTML, un fichier Javascript et un fichier de CSS. Il suffit d'un éditeur de texte pour développer, et d'un navigateur pour consulter le résultat. Un développeur VueJS n'a donc aucune contrainte de développement, ce qui est une force de ce framework.

Les outils de développement du navigateur Chrome permettent d'explorer toutes les facettes de l'applications, et de tester le rendu pour différents affichages (écrans grande taille, écrans réduits, écran smartphone). L'utilisation d'un serveur http sur réseau local permet d'accéder à l'application depuis un smartphone. Cette pratique permet d'expérimenter directement les adaptations pour mobiles.

En conclusion, VueJS est un framework plein de potentiel pour le développement d'applications réactives. La prise en main est simple et le développement est rapide. Les problèmes qui ont étés rencontrés ne sont pas propre à VueJS, mais à l'intégration web en générale. Un des points faible du framework est la délimitation floue entre "petit projet" et "gros projet". VueJS propose une solution idéale pour les applications monopage, et une architecture de fichier décomposée pour les grosses applications, mais aucun entre-deux.

III - Comparaisons

III.1 Vanilla vs. VueJS

III.1.1 Similarités

Pour les premières fonctionnalités, il nous a semblé primordial d'essayer au maximum d'obtenir deux interfaces le plus identique possible. Pour toute la partie de visualisation des carambars et des informations sur les vendeurs, nous avons opéré de cette manière.

On utilise en effet des "composants" issus de bootstrap en commun. Cela nous a permis d'obtenir un design plutôt ressemblant sans avoir à nous concerter souvent. VueJS étant un framework particulièrement léger, contrairement aux framework utilisant Angular, on utilise encore beaucoup de code natif avec Vue. Pour acquérir de l'expérience, nous avons quand même utilisé deux codes différents aboutissant à des divergences dans certains choix (par exemple au niveau des filtres).

Or, comme vu précédemment nous aurions pu partager une grande quantité de code métier.

La ressemblance primordiale pour l'adaptation aux dispositifs est l'utilisation des lignes/colonnes de la technologie bootstrap. En effet pour chaque composant, on peut définir le nombre de colonnes que le composant prendra en fonction de la taille de l'écran

du dispositif. Mais également changer complètement le design du composant en fonction de la taille d'écran, comme on peut le voir pour ce composant.

III.1.2 Utilisation de composants

Dans les deux technologies, nous utilisons des composants pour isoler certains éléments qui seront réutilisables. Comme on a pu voir auparavant, on ne les réalise pas de la même manière.

VueJS simplifie grandement l'échange de données entre parents et enfants, dans un sens comme dans l'autre. Cela est encore une fois, un exemple de la simplification apportée par VueJS.

III.1.3 Dynamisme du DOM

La différence est sur l'utilisation des directives (*v-for et *v-if) avec Vue.JS. Grâce à l'utilisation de ces directives, on réduit grandement l'utilisation des scripts dans le cadre de l'affichage multiple et des conditions d'affichage d'éléments sur la page.

En effet en JS vanilla, on devra récupérer le container des éléments que l'on souhaite ajouter puis créer les nouveaux éléments grâce au script. Cette manière est beaucoup moins rapide qu'avec des directives, chaque attributs de l'élément devant être rentré successivement grâce à des méthodes.

La directive v-if est également plus pratique, on peut l'assigner facilement à un booléen du script. Il suffira alors de changer ce booléen pour que le composant apparaissent ou non à volonté. En vanilla, on devra utiliser la propriété display pour afficher ou non un élément.

On pourrait penser également à supprimer purement les noeuds qui ne doivent pas être affichés, cependant, l'opération est lourde. On peut le voir sur l'utilisation des filtres de la version Vanilla.

III.1.4 Navigation implémentée pour le mobile

Deux axes de navigations différents ont été pris pour la partie mobile. Le premier, pour Vue JS a été de s'orienter sur une fenêtre plein écran sans navigation par header. Les modals sont ainsi affichées sur la totalité de l'écran et ne laissent plus apparaître le contenu initial. Les interactions possibles sont placées à portée du pouce pour obtenir une navigation agréable.

Pour la partie Vanilla, on obtient un header permettant de naviguer entre les différentes fonctionnalités de l'application. Le choix pour la consultation des articles a été d'étendre la zone portée par l'article pour voir les détails.

On a alors avec VueJS une meilleure visibilité mais on gagne la possibilité de comparer rapidement deux articles avec Vanilla.

En second point, on a voulu complexifier la navigation dans une page permettant d'afficher à la fois des images, des vidéos et des graphiques.

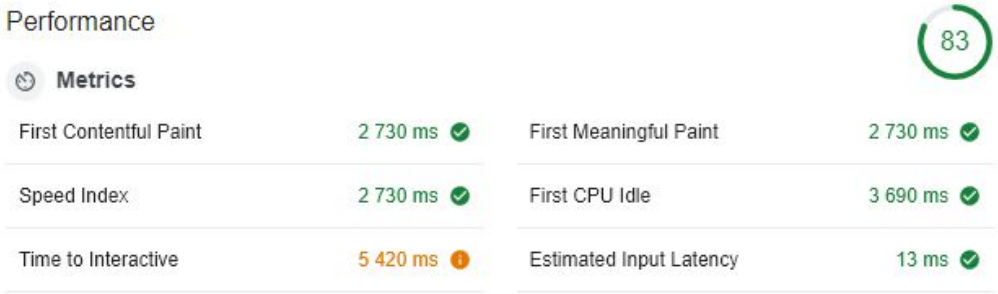
La solution pour afficher de manière agréable ces 3 éléments a été de fusionner trois possibilités de navigation :

- une première grâce à 2 options en bas de l'écran permettant de choisir si l'on veut afficher des médias ou des graphiques
- une deuxième utilisant le slide gauche/droite pour afficher photos ou vidéos avec une indication de la possibilité de slider
- une troisième utilisant des boutons gauche et droite pour naviguer entre les médias.

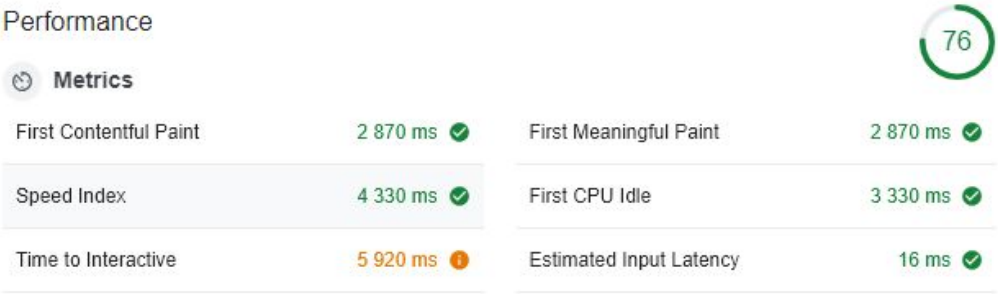
III.1.5 Performances

Nous avons effectué un l'outil d'Audit de Google Chrome, qui nous avait été présenté par Luc Marongui.

Audit pour VueJS



Audit pour Vanilla



L'utilisation d'un framework léger comme VueJS permet d'obtenir des performances web très satisfaisantes, car il encourage les bonnes pratiques grâce à ses directives.

Vanilla est par contre le plus optimisable. Des choix pris au début du projet ralentissent grandement les temps calculés par l'audit, comme le fait que certains composant sont ajoutés et supprimés du DOM. Ils auraient pu être seulement rendu visible/invisible.

III.2 Android vs. React

III.2.1 Différences de paradigmes

La notion clé derrière React Native est la composition de composants. Plusieurs développeur peuvent créer des composants que l'autre pourra utiliser. Cela permet de rajouter des fonctionnalités à notre application sans avoir à changer de manière drastique notre base de code. Les composants React Native possède un code à afficher, si ce code dépend de variables d'état le composant se ré-affichera automatiquement avec les nouvelles valeurs des variables sans avoir à appeler de fonction.

La proximité de la fonction de render() de React Native avec du HTML, l'utilisation de JSX proche de Javascript et les feuilles de style presque identiques aux feuilles CSS, permettent à des développeurs ayant des notions de web d'apprendre assez rapidement React Native.

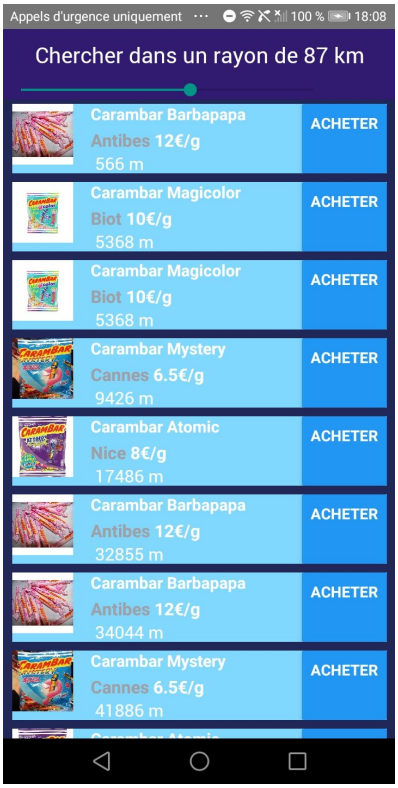
En Android, on peut également réutiliser des composants, le meilleur exemple étant les fragments que l'on peut implémenter dans plusieurs layouts différents sans avoir à réécrire du code.

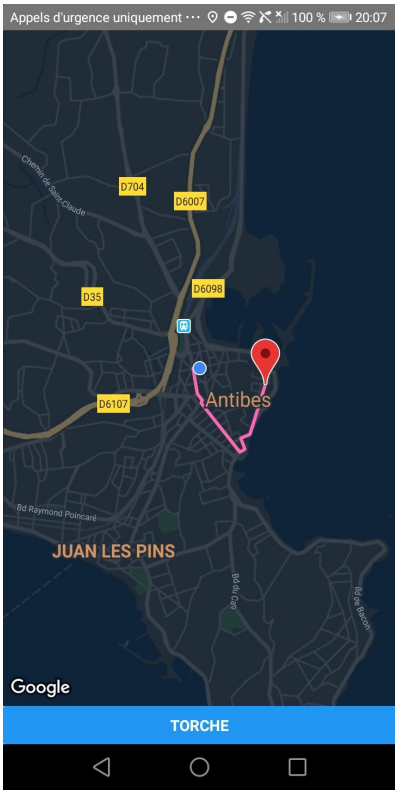
Android utilise un paradigme de programmation événementielle, qui est beaucoup basée sur la surcharge de méthode qui sont appelées automatiquement pendant l'exécution à la résolution d'événements particuliers. Ce paradigme présente l'avantage d'être très lisible et relativement facile à prendre en main.
Au niveau de l'implémentation des vues, les éléments des layouts sont principalement définis par rapport au parent ou aux éléments du même niveau, contrairement à React Native, qui se rapproche plus du développement web.

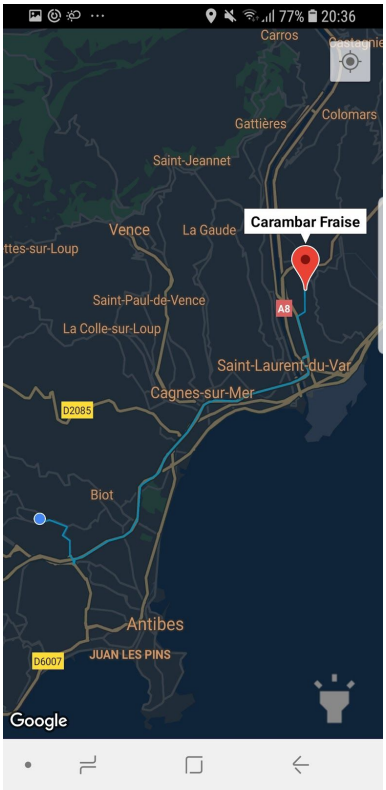
III.2.2 Comparaison des interfaces

Les différences de layout entre du natif et React Native dépendent grandement des compétences du développeur. Ainsi, un développeur connaissant bien ses composants natifs saura développer des composants React Native très ressemblants à ceux du natif. Nous avons implémenté deux vues légèrement différentes, mais qui restent dans le même état d'esprit, pour pouvoir comparer les deux versions. Les principales différences fonctionnelles concernent le passage au mode nuit, notamment dans l'activité de cartographie. Sur Android, le changement dynamique de mode d'affichage se fait bien et de manière transparente. En revanche, sur la partie React Native, nous ne sommes pas parvenu à obtenir un résultat semblable. En effet, la carte se charge bien dans le mode qu'il faut mais ne peut pas basculer dans l'autre. Cela provient de l'utilisation de composant externe, ce composant *MapView* ne permet pas de changer le style de carte une fois le composant monté. On trouve une limite à l'utilisation de librairie en React Native, puisque l'on perd le contrôle du comportement des composants. D'un point de vue fonctionnel, nous remarquons après réflexion que la mise en page réalisée avec React Native (Captures

d'écran à gauche sur la page suivante) semble plus adaptée au but de l'application. En effet, l'utilisateur a une bien meilleure vision de tout ce qui lui est proposé autour de lui, alors que sur la version Android, il ne voit que 4 propositions à la fois, ce qui va le pousser à beaucoup faire défiler son écran.







III.2.3 Utilisations des capteurs

Autant en Android qu'en React Native, l'utilisation des capteurs s'est révélée plutôt aisée. La différence étant qu'en Android, on utilise une classe présente de base dans Android alors que React Native nécessite l'utilisation d'une librairie externe. La demande de permissions également, se déclare d'une manière similaire avec les deux technologies.

III.2.4 Comparaison des performances

La première différence que nous avons remarquée a été au niveau du passage au mode nuit. En effet, si beaucoup de carambars sont affichés dans l'application, on observe une baisse de performances au niveau de la version React Native que l'on ne retrouve pas en Android Natif. En effet, la liste contient des composants *Item* qui possède chacun un *EventListener* pour connaître la valeur l'intensité lumineuse amiante. Un grand nombre d'item demande par conséquent plus de ressource. La fluidité du changement de mode en Android est dû à l'utilisation de la *RecyclerView*.

A l'échelle du projet, à part le problème cité ci-dessus, nous n'avons pas remarqué de différence notable entre les deux versions de l'application.

Les raisons nous poussant à développer avec React Native plutôt qu'en natif Android (et vice et versa) dépendent de ce qu'on attend de notre projet.

Développer en natif	Développer avec React Native
• Le projet ne nécessite de viser qu'une plateforme • Le projet nécessite beaucoup de portions de code complexe spécifique à la plateforme visée • L'application doit utiliser des nouvelles fonctionnalités de l'OS dès leurs sorties	• L'application aura la même apparence et le même comportement sur les deux plateformes • On dispose de peu de temps et de moyens tout en devant viser les ios et Android • Si les développeurs possèdent déjà des connaissances en React et en web

IV - Conclusion

De très nombreuses façons de réaliser des applications web ont vu le jour ces dernières années. La démultiplication des choix est une véritable aubaine pour les développeurs et plus encore pour les ergonomes. En effet, en fonction des adaptations désirées et de la complexité du projet, différentes solutions optimisées à chaque cas sont proposées.

A partir du moment où l'on devra s'adapter à l'environnement, le développement natif est le meilleur choix. L'utilisation des capteurs est primordial et elle est très accessible sur cette technologie. Cependant, de par la présence d' android et iOS sur le commerce, ne toucher qu'une seule des deux cibles pour un développement est dramatique dans le marché actuel.

Pour cela est apparu le développement hybride qui permet d'obtenir une application pour chacun des stores. Dans l'exemple de React Native, on peut obtenir ces deux applications, tout en profitant de composants développés par la communauté permettant d'accéder aux capteurs depuis les 2 plateformes.

Au vu de la conjoncture actuelle, le développement hybride natif comme react native semble prendre le dessus sur le développement purement natif (sauf dans des cas précis pour soucis d'optimisation). Le développement hybride est en effet plus pratique et moins contraignant qu' Android par exemple.

Enfin, pour de petits projets visant à être accessibles sur la majorité des dispositifs, la solution de framework léger comme VueJs semble parfaite. On pourrait envisager l'utilisation de Javascript Vanilla mais la fonctionnalité principale présentée dans ce document (les Web components) n'est pas encore totalement normalisée par le W3C. Ils ne sont pas accessibles depuis tous les navigateurs et c'est cela qui rend leur utilisation bien trop risquée.