Haute école d'ingénierie et d'architecture Fribourg
Hochschule für Technik und Architektur Freiburg

# Behind Food

Amélioration de l'application mobile sur la face cachée des aliments industriels

## Rapport

Semestre 5
2021-2022

Étudiant : Grégory Geinoz
gregory.geinoz@edu.hefr.ch

Professeur : Pascal Bruegger
Conseiller/Mandant : Samuel Fringeli

Hes·so // FRIBOURG FREIBURG
Haute Ecole Spécialisée de Suisse occidentale
Fachhochschule Westschweiz
University of Applied Sciences and Arts Western Switzerland

Version : 2.2
Fribourg, 27.01.2022

# Table des matières

# 1 Introduction

Behind food est une application mobile utilisée dans le cadre d'une exposition sur le développement durable. Elle permet aux visiteurs d'explorer les faces cachées de différents aliments du quotidien au moyen d'un parcours de divers thèmes reliés à ces aliments. Ce parcours donne accès à des images, à des vidéos et à des textes pour illustrer les caractéristiques des aliments concernés. Ces éléments sont mis à jour par l'équipe qui gère l'exposition, au moyen d'une interface backend, et sont accessibles depuis l'application grâce à une API. Dans la version actuelle de l'application, c'est une WebView qui est chargée et qui affiche les images et vidéos au fur et à mesure du parcours de l'utilisateur dans la structure, mais les médias ne sont pas sauvegardés dans le stockage local de l'application, ce qui rend impossible l'utilisation de celle-ci hors-ligne. Comme l'exposition a pour objectif de fonctionner entièrement hors-ligne, il serait nécessaire de faire en sorte que les données affichées soient téléchargées localement, avec un système permettant d'actualiser les dernières modifications effectuées sur le backend par l'utilisateur de l'application.

## 1.1 Contexte

L'application de base [1] est une application web qui utilise la bibliothèque ZircleUI [2], une interface utilisateur simple mais intelligente avec une navigation zoomable intégrée à travers des cercles, affichée dans une application iOS avec un WebView de SwiftUI. Tout le contenu affiché, y compris le texte, les images et les vidéos, est obtenu à partir d'une API créée par le mandant.

## 1.2 Objectifs

Le projet est constitué d'objectifs principaux et secondaires. Les objectifs principaux sont à réaliser en priorité.

### 1.2.1 Objectifs principaux

1. **Application multiplateforme**
   L'idée est d'adapter l'application iOS de base en une application multiplateforme. La technologie utilisée pour ce projet est documentée dans la section "Analyse" de ce document entre Cordova, Flutter, React Native, Ionic ou Xamarin.

2. **Compatibilité hors-ligne**
   L'application doit devenir compatible hors-ligne. Cela signifie qu'elle est capable de vérifier la connexion de l'appareil et de s'adapter selon cette dernière. Elle doit être capable de stocker son contenu, de le récupérer si l'appareil est hors-ligne, et de mettre à jour le contenu si l'API a été mise à jour quand l'appareil est en ligne.

### 1.2.2 Objectifs secondaires

1. **Mises à jour**
   Ajouter les applications Android et iOS respectivement sur le Play Store et l'App Store et gérer les futures éventuelles mises à jour de ces applications mobiles.

## 1.3 Outils de collaboration

Le code et les documents officiels de ce projet sont postés sur le GitLab de l'école (autorisations à demander) et les réunions entre les différents acteurs du projet se font soit en présentiel au bureau de M.Bruegger (D20.17) soit avec Microsoft Teams.

### 1.4 Structure du rapport

Le projet est organisé selon un modèle hybride cascade et agile avec la structure suivante :

1. Analyse
2. Conception
3. Implémentations
4. Tests
5. Conclusion
6. Documentation

L'analyse permet de trouver les directions avec lesquelles les implémentations sont codées. Puisqu'il s'agit d'une amélioration d'une application déjà existante, la conception a déjà été étudiée avec le mandant et ajoute simplement les objectifs à la conception existante. Des diagrammes d'explication du fonctionnement de l'application de base et de l'application améliorée sont tout de même disponible dans la section "Conception" de ce document. Les implémentations sont en fait des sprints référençant chaque objectif du projet. Chaque objectif est implémenté, testé et présenté au professeur et au mandant afin d'être confirmé pour pouvoir passer au sprint suivant. Une fois ces sprints réalisés, la phase de test confirme les implémentations ou permet de les corriger. La phase de documentation se déroule sur l'entier du projet, en commençant par ce document qui est écrit au fur et à mesure de l'avancement du projet.

## 2 Analyse

### 2.1 Les solutions multiplateformes

Il s'agit pour ce premier objectif de convertir l'application de base iOS en une version cross-plateforme en utilisant une des technologies décrites dans les chapitres ci-dessous. L'application de base affiche simplement l'application web dans une WebView du WebKit de iOS [3]. Afin de s'orienter au mieux pour le deuxième objectif, il faut que cette technologie supporte non seulement les WebViews mais également une solution de stockage local en cache, comme l'API IndexedDB [4], qui est nécessaire pour la persistance des données au sein d'un navigateur internet. Une autre approche est de tout adapter en code natif pour chaque plateforme pour simplifier l'accès au stockage local des appareils.

#### 2.1.1 Natif

Il s'agit de la façon la plus pragmatique de développer une application sur plusieurs plateformes : simplement créer une application spécialement implémentée pour chaque plateforme. Une application iOS en Swift, une application Android en Java et une application web pour ordinateurs en HTML5, CSS3 et JavaScript. Dans notre cas, les applications web et iOS existent déjà, il suffirait donc de simplement implémenter une version similaire en Android puis d'améliorer l'application web dans l'Objectif 2. Cette approche amène plusieurs avantages :

- Une disponibilité simplifiée et direct au matériel de chaque appareil (GPS, micro, caméra, stockage interne,...)
- Une UX toujours adaptée à chaque appareil
- Moins besoin de librairie externe open-source pour ajouter des fonctionnalités qui sont déjà disponibles en natif

Évidemment, cette méthode demande autant de code qu'il y a de plateforme à disposition, mais pour ce travail, cela ne devrait pas être un obstacle à prendre en compte. Il y a néanmoins une difficulté à relever, les restrictions de JavaScript. En effet, pour des raisons de sécurité, JavaScript a toujours eu des accès très limités au matériel des plateformes, c'est pourquoi un script ne pourra jamais accéder directement aux disques durs d'un ordinateur, il est entièrement encapsulé dans le navigateur. Il est donc nécessaire d'adapter l'application web pour qu'elle puisse stocker des fichiers dans le cache des navigateurs, car c'est l'unique zone de stockage qu'un script JavaScript pourra accéder. Le côté application web pour ordinateurs ne devraient pas poser de problème de ce côté, mais il s'agit

maintenant de savoir s'il est possible à des WebViews d'accéder au stockage local des téléphones, et si non s'il est possible de stocker des fichiers dans le cache des WebViews. Si ces choses sont impossibles, cela veut dire qu'il faut adapter entièrement en natif l'application web, sans utiliser de WebView. Il est tout à fait possible d'utiliser du JavaScript avec Swift ou Java (avec JavaScriptCore et AndroidJSCore respectivement), l'intégration du framework ZircleUI et la grande majorité de la logique de l'application web existante ne sera donc pas un problème, mais cela demandera beaucoup plus de codage que les solutions suivantes.

### 2.1.2 Xamarin

Xamarin est une plateforme open-source qui permet de créer des applications mobiles sur iOS, Android et Windows Phone. Le code partagé de l'application s'écrit en C# [5]. Un des frameworks de Xamarin est Xamarin.Forms [6], qui fournit les WebView [7]. Lors de la rédaction de ce document et durant une bonne partie du semestre 5, Xamarin est un sujet majeur du cours "Développement Mobile Cross-Plateforme".
Par exemple, UPS utilise Xamarin pour son application mobile.

### 2.1.3 Cordova

Apache Cordova est un framework de développement mobile open-source. Il permet de créer des applications multi-plateformes en utilisant les technologies web standards actuelles HTML5, CSS3 et JavaScript. Cordova utilise de base des WebViews afin d'afficher tout l'interface graphique des applications, ce framework est donc un bon choix pour s'implifier l'utilisation des WebViews pour les applications multiplateformes. Le plus gros avantage de Cordova est qu'il met à disposition une grande quantité de plugins (les Core Plugins notamment) permettant l'accès à des composants natifs des plateformes, ce qui est très rassurant pour le bon fonctionnement des deux Objectifs de ce projet.
Il n'y a plus beaucoup de grosses compagnies qui utilisent Cordova pour leurs applications mobiles. Adobe utilisait Cordova comme base pour leur solution de développement cross-plateforme Adobe PhoneGap, qu'ils ont décidé d'abandonner. Cordova, lui, continue d'exister et reste une référence du cross-plateforme.

### 2.1.4 Flutter

Flutter est un kit de développement logiciel (SDK) open-source créé par Google. Il permet de créer des applications multiplateformes sur Android, iOS, mais aussi Windows, Linux, Mac et le web à partir d'une seule base de code, le Dart. Le package webview_flutter permet d'importer les WebViews dans une application Flutter. Malheureusement, Flutter bloque IndexedDB et le seul moyen de contourner cette restriction est d'utiliser un plugin qui est en fait un simple wrapper Flutter pour IndexedDB, mais il semble que Flutter ne permette pas de mettre en cache des gros fichiers [8].
Par exemple, eBay Motors, BMW ou encore Google Ads utilisent Flutter pour leurs applications mobiles.

### 2.1.5 React Native

React Native est un framework de développement mobile open-source créé par Facebook. Le code est écrit en JavaScript puis est rendu en natif. C'est une évolution de React, une librairie JavaScript également créée par Facebook qui permet de créer des interfaces utilisateurs mais uniquement ciblé pour les navigateurs. Mais il s'agit d'un projet relativement récent, ce qui rend la recherche d'information compliquée et la documentation doit encore s'étoffer.
Puisqu'il s'agit d'une invention de Facebook, la grande majorité de leurs applications sont en React Native (Facebook, Instagram,...), mais d'autres tout autant connues (UberEats, Oculus, Discord, Skype, ...) sont également implémentées en React Native.

### 2.1.6 Ionic

Ionic est un framework en partie open-source de développement cross-plateforme utilisant les technologies web. Initialement, Ionic était basé sur AngularJS et Apache Cordova. Dans sa version 2.0, Ionic utilise Angular 2 qui permet de convertir du TypeScript vers du JavaScript, il est donc nécessaire de coder en TypeScript.
McDonald's Turquie, McLaren, Diesel, Sworkit sont des exemples de grosses applications utilisant Ionic.

### 2.1.7 Tableau comparatif

Le tableau suivant compare chacune des technologies précédemment présentées avec des critères pondérés afin de choisir au mieux la solution idéale pour ce projet.

| Objectif 1 | | | | | | | |
|---|---|---|---|---|---|---|---|
| Critères et pondération | | Native | Xamarin | Cordova | Flutter | React Native | Ionic |
| Connaissance personnelle | 4 | 9 (cours) | 10 (cours) | 10 (cours) | 1 | 1 | 1 |
| Réutilisabilité du code fourni | 5 | 1 | 5 | 10 | 5 | 5 | 5 |
| Communauté et documentation | 3 | 10 | 8 | 8 | 8 | 3 | 3 |
| Temps de développement gagné | 5 | 3 | 5 | 10 | 2 | 2 | 2 |
| **Totaux** | **17** | **86** | **114** | **164** | **63** | **48** | **48** |

### 2.1.8 Conclusion

Selon le tableau précédent, Cordova est la solution favorite pour les besoins de ce projet. Même s'il s'agit du doyen de toutes ces solutions, Cordova reste une référence dans le domaine, possède une communauté gigantesque, est particulièrement simple à adapter pour le projet car il s'agit de code d'application web qui crée une application mobile. Je pense donc que cette approche sera la mieux adaptée. Ce choix est également avantagé par ma connaissance de cette technologie et le fait que l'environnement de développement est déjà prêt sur mes machines. En effet, puisque Cordova était un sujet du cours de M.Bruegger "Développement mobile cross-plateforme", nous avons déjà eu l'occasion d'installer et d'utiliser Cordova, ce qui est un gros avantage comparé aux autres technologies. À l'heure à laquelle cette partie du document est rédigée, seul Xamarin est en train d'être étudié dans le cadre du cours, mais nous venons de commencer.

## 2.2 Les solutions hors-ligne

Afin de rendre l'application existante utilisable sans connexion, il faut que les données récupérées depuis l'API du mandant puissent persister d'une manière ou d'une autre. Pour ce faire, deux possibilités se présentent :

### 2.2.1 Convertir l'application web en application mobile native

Convertir entièrement l'application web avec une solution multiplateforme vue plus haut, afin d'accéder facilement au stockage de l'appareil mobile utilisé.

### 2.2.2 Convertir l'application web en application web multiplateforme

Améliorer l'application web actuelle afin qu'elle puisse accéder au cache des navigateurs sur lesquels elle tourne pour y enregistrer les données qu'elle utilise, afin de les récupérer facilement sans connexion dans le cas où le réseau tomberait ou qu'aucune connexion n'est disponible. Tout dépend donc à quel endroit les données sont stockées.
La première solution permet d'utiliser le stockage interne de l'appareil mobile et de sortir du champ d'application d'une WebView, mais demande une bien plus grande quantité de codage et de prise en main, quelque soit la technologie cross-plateforme choisie.
La seconde solution ne demande qu'une adaptation cross-plateforme de l'application mobile actuelle utilisant simplement une WebView afin d'afficher l'application web convertie en application "web-mobile" lancée dans une WebView, mais demande d'utiliser les caches des navigateurs car il n'est pas possible de sortir du champ d'application de ces derniers.
L'idée de la seconde solution est de faire comme cette application web d'exemple qui, lors de la première visite avec une connexion, stocke toutes les vidéos (quelques Mo par vidéos, un peu comme l'application web actuelle)

ainsi que les fichiers statiques comme le CSS et le JavaScript dans une IndexedDB. Grâce à ce processus, mettre son appareil en mode avion (ou dans un scénario plus réaliste, perdre sa connexion) n'empêche pas l'utilisation du site. Les vidéos s'affichent et peuvent toujours être lues sans aucun problème. Comme mentionné plus haut, les dernières versions de Safari et Chrome sont les versions 15 et 94, IndexedDB est parfaitement supporté sur ces deux versions récentes qui se mettent pour la grande majorité des appareils automatiquement à jour [9].
En contrôlant la connexion de l'appareil dès le démarrage de l'application [10] [11], il est possible dans le cas où il y a une connexion de vider toute la IndexedDB et de télécharger tous les fichiers de l'API afin de mettre leurs versions à jour dans la IndexedDB. Dans le cas où il n'y a pas de connexion, l'application va directement chercher dans sa IndexedDB les fichiers afin de créer l'application web sans avoir à télécharger quoi que ce soit du net. **Ceci inclut que la toute première utilisation de l'application se fasse avec une connexion internet.** Mais puisque l'objectif secondaire est de déployer cette application sur les stores, les utilisateurs désireux de l'acquérir n'auront pas le choix d'avoir une connexion internet, ce qui rend cette précondition plutôt triviale.

### 2.2.3 Tableau comparatif

Le tableau suivant compare quelques technologies afin de stocker des données autant du côté natif que du côté web, avec des critères pondérés afin de choisir au mieux la solution idéale pour ce projet. Puisqu'il a décidé dans la section précédente que Cordova est l'approche à utiliser pour ce projet, toutes les solutions présentées ci-dessous sont utilisables avec Cordova.

| Objectif 2 | | | | |
|---|---|---|---|---|
| Critères et pondération | | Plugin File | IndexedDB | Service Workers |
| Taille maximale des fichiers stockés | 5 | 10 (même que le stockage) | 10 (1GB à 50% du stockage) | 1 (52MB) |
| Taille maximale du stockage mis à disposition | 5 | 10 | 10 | 1 |
| Persistance en cas de stockage libre bas | 5 | 10 | 10 | 10 |
| Communauté et documentation | 3 | 8 | 8 | 5 |
| **Totaux** | **18** | **174** | **174** | **75** |

### 2.2.4 Conclusion

Il y a donc deux solutions qui sont à égalité : Plugin File et IndexedDB. Il faut tester ces deux solutions dans les extrêmes, notamment pour la taille maximale allouée pour l'application et la taille maximale par fichier stocké. Ces tests sont réalisés après le premier sprint d'implémentation, une fois le Objectif 1 terminé et validé puisque Cordova sera utilisé dans tous les cas.

# 3 Conception

## 3.1 Application de base

Ci-dessous un schéma simplifié de l'application web actuelle dont voici une explication :

1. Une WebView SwiftUI sur un iPhone ou un iPad lance un URL (https://adelente-admin.samf.me/app), ceci est la WebApp hébergée sur le serveur Strapi des mandants
2. La WebApp va ensuite faire un XMLHTTPRequest pour acquérir le fichier JSON fourni par l'API des mandants (également du côté de Strapi)
3. Ce dernier est ensuite décortiqué :
    - Si le média est un texte, il est immédiatement trouvé dans le JSON et est directement placé dans une balise div
    - Si le média est une image ou une vidéo, le JSON fournit une URL qui pointe vers le serveur Strapi qui héberge cette image ou vidéo, puis cet URL est placé dans une balise image ou video qui permet de les afficher

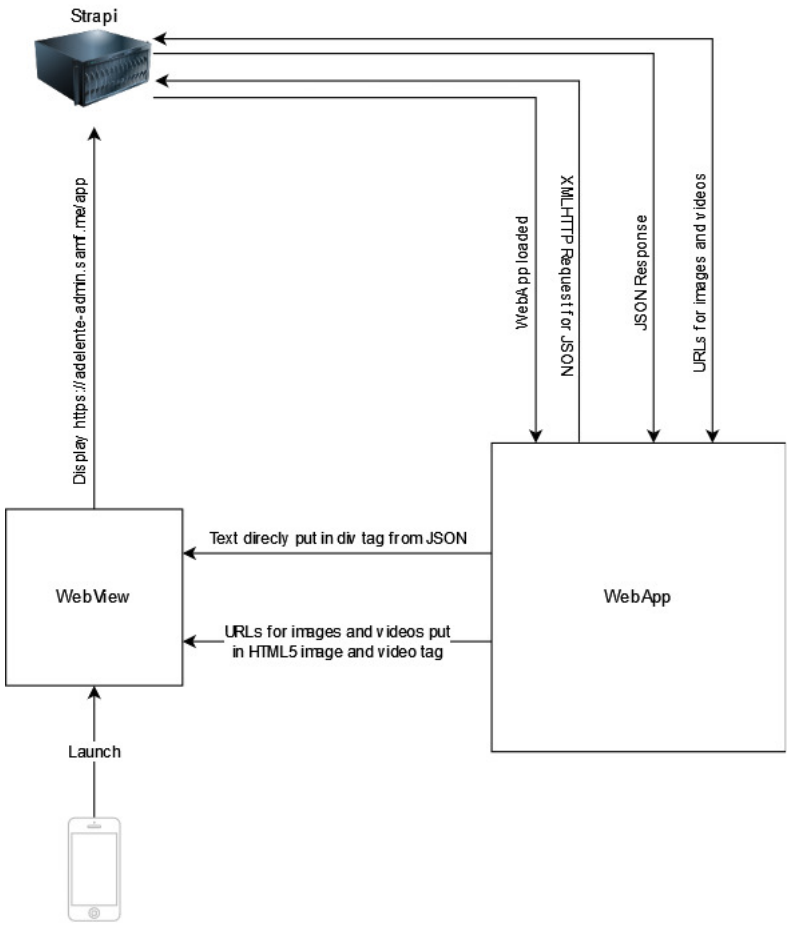


Figure 1 – Application web de base

## 3.2 Application modifiée

Ci-dessous un schéma du fonctionnement de l'application web cross-plateforme prévue dont voici une explication :

1. Une application web cross-plateforme Cordova nativement affichée dans une WebView, demande le fichier JSON à l'API Strapi
2. Une fois le JSON récupéré, il est comparé au fichier JSON mis en cache.
   - Dans le cas où il n'y aurait pas de JSON dans le cache, il serait stocké immédiatement et passerait à la suite (première utilisation), tout le contenu est téléchargé et stocké localement.
   - Dans le cas où un JSON est stocké, il est comparé au JSON fraîchement téléchargé. Si une ou plusieurs différences sont observées, elles sont traitées afin de récupérer le nouveau contenu et/ou remplacer le contenu mise à jour et/ou supprimer le contenu retiré.
   - Dans tous les cas, le JSON téléchargé écrase l'ancien JSON dans le cache.
3. Une fois ces contrôles terminés, la WebApp récupère et affiche tout le contenu mise en cache (connexion internet ou non).

Remarque : ce schéma ne couvre pas les points suivants :

1. Si dès le début il n'y a pas de connexion internet disponible, la WebApp récupère immédiatement le JSON et le contenu stockés. Si rien n'est stocké, c'est qu'il s'agit d'une première utilisation ou que la plateforme ait effectué un vidage de cache indésirable, il faudra donc afficher un message d'erreur et de demande de connexion internet.
2. Le framework ainsi que les polices utilisés par la WebApp sont également cachés et ce avant le JSON. Ce sont également les premiers éléments récupérés dans le cache afin de pouvoir afficher correctement le contenu.
3. Durant tout ce processus, l'utilisateur ne peut pas utiliser l'application. En effet, une barre de progression ainsi qu'un texte informe l'utilisateur qu'un contrôle du contenu est en cours.
4. Une option est disponible dans le menu de la barre d'outil pour forcer un contrôle à tout moment.

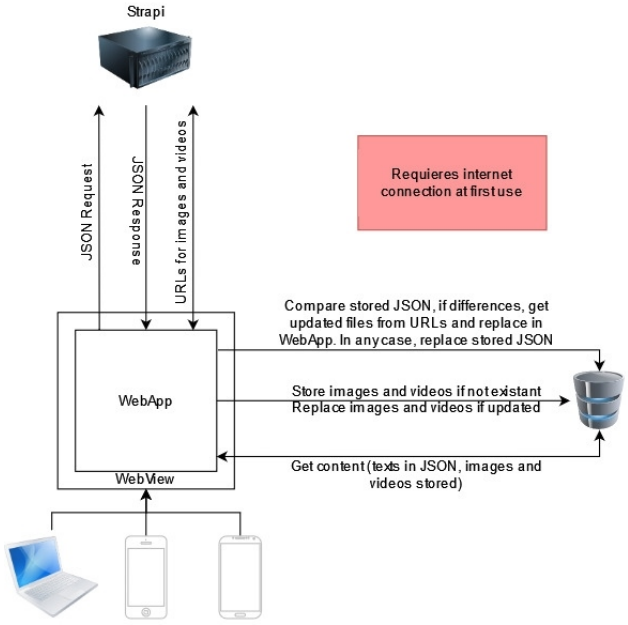


Figure 2 – Application web modifiée

# 4 Implémentations

## 4.1 Version 0

Afin de se réhabituer à l'utilisation de Cordova, l'exercice est de réaliser une version 0 qui reprend le même fonctionnement que l'application iOS actuelle, c'est-à-dire une application web hébergée sur internet et chargée dans une simple WebView. Cordova met à disposition le plugin InAppBrowser, qui permet de charger une instance d'un navigateur directement dans l'application. Le code est encore plus léger que pour l'application originale.

```
1            document.addEventListener('deviceready', onDeviceReady, false);
2
3            function onDeviceReady() {
4                openBrowser();
5            }
6
7            function openBrowser() {
8               var url = 'https://adelente-admin.samf.me/app';
9               var target = '_blank';
10              var target2 = '_system';
11              var options = "location = no, toolbar = no";
12              if (device.platform === 'Mac OS X') {
13                   var ref = cordova.InAppBrowser.open(url, target2, 'location=no,toolbar=no'
                         );
14                }else{
15                   var ref = cordova.InAppBrowser.open(url, target, 'location=no,toolbar=no')
                         ;
16               }
17           }
```

Notez que le plugin Device permet de connaître sur quel dispositif est lancée l'application Cordova, un plugin très utile qui servira dans les versions futures. Ici, il est utile pour savoir si l'application est lancée sur un système Mac OS X, car ce dernier ne supporte que le target "_system".

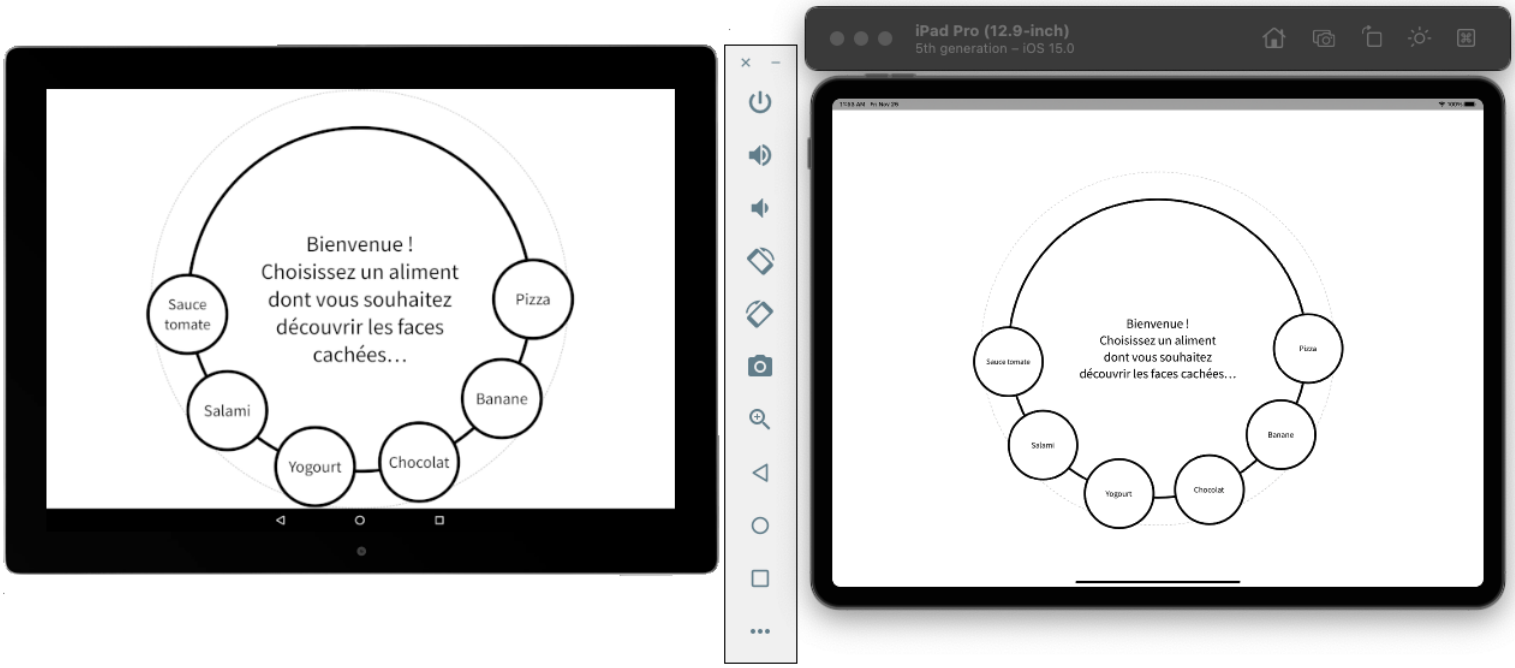


Figure 3 – Résultats de la Version 0 sur tablettes

Puisque l'application web de base a été prévue pour s'afficher sur des iPad grâce à un CSS spécifique, l'application s'affiche de façon adaptée sur simulateur (Figure 3 à droite). Sur cette même figure à gauche, l'application est lancée sur une tablette Android et il est clair que l'affichage n'est pas encore adapté pour cet environnement. Ceci met en évidence le travail d'adaptation cross-plateforme pour le reste de l'Objectif 1. Ceci est réalisé dans la version ci-après.

## 4.2 Version 1

L'intérêt de la version 1 est que l'application web est locale et seul le contenu est téléchargé depuis l'API Strapi du mandant. Une chose en moins à réfléchir pour l'Objectif 2. C'est pour ce résultat rapidement atteint que Cordova a été choisi. Le code de l'application web est en fait injecté directement dans le JavaScript de l'application Cordova. Le plus gros travail est de préparer un CSS spécifique pour que l'application soit lisible et utilisable sur une multitude de plateforme. À cette version s'ajoute les logos de l'application, le blocage de l'affichage en mode paysage et des media queries dans le CSS pour gérer l'échelle de chaque plateforme.
Malheureusement à ce stade, un problème de taille s'est présenté pour Android. En effet, l'image des vidéos ne s'affichent pas, mais le son et la lecture fonctionne. C'est en effet un problème connu avec Cordova Android, et il est résolu de la manière suivante :

```
1            if (window.cordova.platformId === 'android'){
2                let url = 'https://adelente-admin.samf.me' + data.media.url;
3                return `<div height="350" width="350" onClick="(function(){
4                    VideoPlayer.play('${url}');
5                    return false;
6                    })();return false;"><video></video></div>`;
7            }else{
8                return `<video height="280" width="280" src="https://adelente-admin.samf.me${
                     data.media.url}" controls></video>`;
9            }
```

Il faut savoir que ce problème n'existe que sur Android, il faut donc d'abord reconnaître la plateforme. Ceci grâce à window de Cordova, une version build-in du plugin Device, donc une meilleure solution et un plugin en moins. Puis une variable récupère l'entier de l'url construit avec la variable data.media.url, puis une div est créée avec un gestionnaire d'évènement et sa fonction inline afin d'utiliser le plugin VideoPlayer qui permet d'utiliser un lecteur natif pour lire des vidéos avec un url (ou un url local pour l'Objectif 2). Une vidéo vide est ajoutée à l'intérieur de la div afin d'afficher l'icône de démarrage de vidéo, c'est plus clair ainsi pour l'utilisateur. La vidéo démarre immédiatement et en plein écran, mais aucun contrôle n'est fourni. Malheureusement, aucune option pour afficher des contrôles n'est disponible pour ce plugin. Il est possible de quitter la vidéo en utilisant le bouton retour de l'appareil.

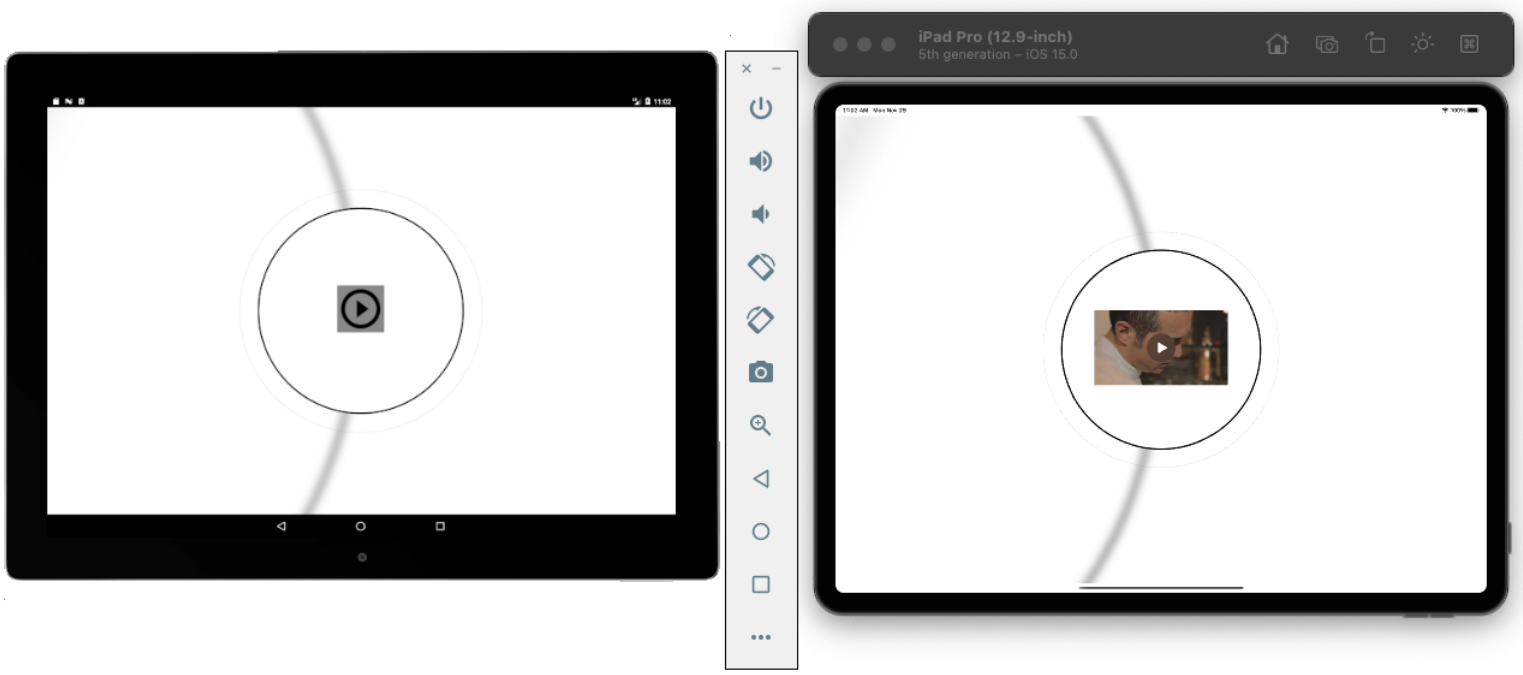


Figure 4 – Résultats de la Version 1 sur tablettes pour une vidéo

## 4.3 Version 2

La version 2 implémente entièrement l'objectif 2. Afin de simplifier la compréhension du fonctionnement du code, voici un résumé appuyé par un schéma, découpé par détail. Ce dernier peut être trouvé au complet en annexe.
Au final, l'application utilisera toujours son contenu local pour afficher la Vue.js, que le contenu soit complet ou non, et les accès à l'API ne se feront que lors de la première utilisation de l'application où il faudra télécharger l'entier du contenu multimédia nécessaire au bon fonctionnement de l'application, et si l'utilisateur souhaite mettre à jour le contenu local de l'application. Il y a donc plusieurs étapes à implémenter afin que tout ceci soit respecté.

### 4.3.1 Démarrage et contrôle de connexion

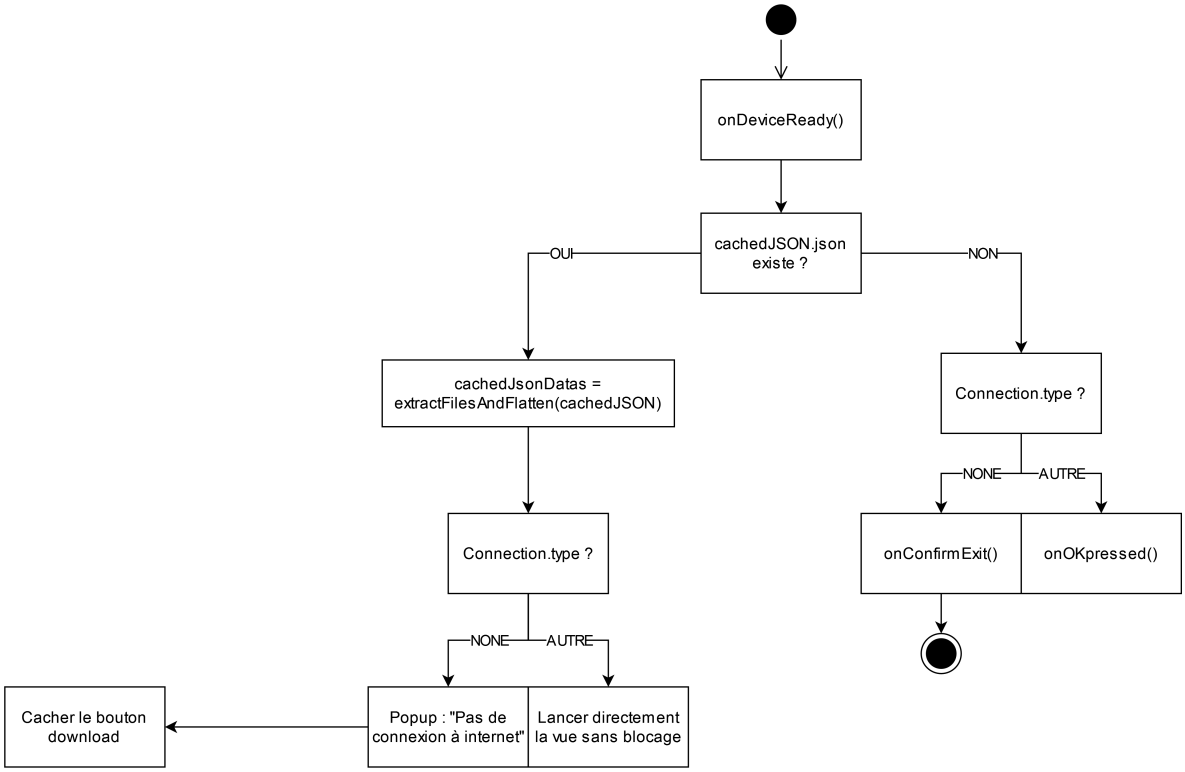


Figure 5 – Démarrage de l'application

Le démarrage de l'application attend que tous les composants Cordova soient prêts. Cet événement se nomme «onDeviceReady()» et nous pouvons avoir la certitude que tout soit correctement chargé après ce dernier. C'est pourquoi il est important que cet événement soit la racine de toute application Cordova.
La première étape de l'application est de contrôler si dans les fichiers locaux de l'application se trouve un fichier nommé «cachedJSON.json».

- Si un tel fichier n'existe pas, c'est que l'application a été démarrée pour la première fois. Il faut donc ensuite contrôler le type de connexion que l'appareil possède.
  - Si l'appareil ne possède aucune connexion à internet, il est impossible d'utiliser l'application correctement, car la première utilisation devra télécharger tout le contenu à afficher, un pop-up invite donc l'utilisateur à trouver une connexion à internet, puis ferme l'application.
  - Si l'appareil possède une connexion à internet, tel qu'une connexion wifi ou même ethernet, alors l'application peut correctement démarrer pour la première fois (cf. 4.3.2)

- Si un tel fichier existe, c'est que l'application a déjà été démarrée précédemment. Puisque ce contrôle ouvre le fichier s'il existe, l'application profite de préparer immédiatement avec le contenu du fichier l'objet «cachedJsonDatas» qui contient le contenu JSON préparé pour la Vue.js ainsi qu'un tableau d'URLs pour chaque média à afficher. Ces derniers concentrent la date de la dernière mise à jour du média, son URL vers l'hébergeur et donc le nom du fichier ainsi que son type (.mp4, .png, ...). Il faut donc ensuite contrôler le type de connexion que l'appareil possède.
    - Si l'appareil ne possède aucune connexion à internet, l'application cache le bouton qui permet de mettre à jour le contenu de l'application. Cette décision a été prise car les clients n'utiliseront pas les tablettes avec une connexion à internet lors de leurs manifestations, et n'auront besoin de cette option que lorsqu'ils auront une connexion à internet, c'est-à-dire quand ils utiliseront eux-mêmes l'application dans l'optique de mettre à jour le contenu de l'application. Un pop-up informe l'utilisateur qu'il lui faudra trouver une connexion à internet s'il souhaite mettre à jour l'application.
    - Si l'appareil possède une connexion à internet, l'application affiche le bouton qui permet de mettre à jour le contenu de l'application.
    - Dans les deux cas, l'application est lancée avec le contenu local, elle passe donc à l'étape suivante (cf. 4.3.3)

### 4.3.2 Première utilisation détectée

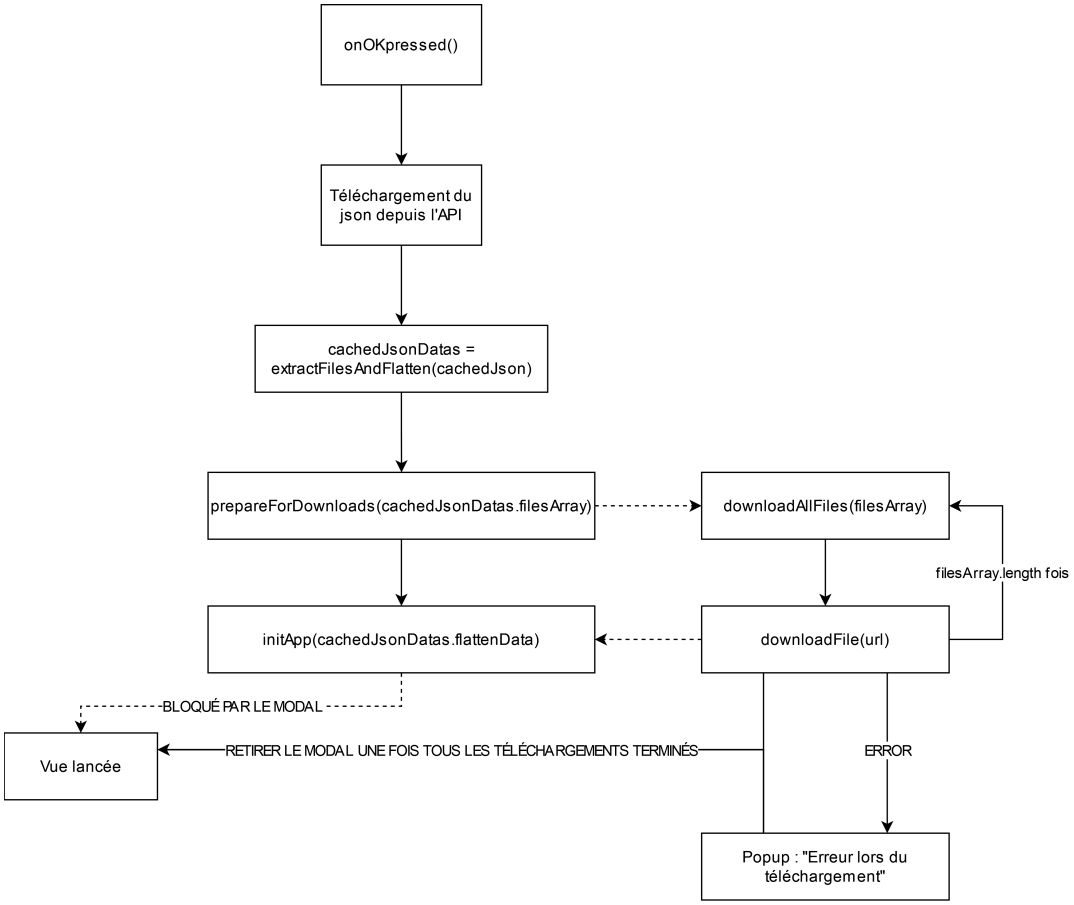


Figure 6 – Première utilisation détectée

Grâce à l'étape précédente, il a été déterminé que l'application est lancée pour la première fois. Puisque l'application utilise uniquement son contenu local pour fonctionner, il faut télécharger l'entier des médias nécessaires pour afficher et utiliser l'application grâce aux étapes suivantes :

1. L'application commence par télécharger le contenu JSON de l'appel à l'API. Le résultat de cet appel est d'abord enregistré dans un nouveau fichier nommé «cachedJSON.json».
2. Une fois ce fichier enregistré, l'application utilise encore une fois le résultat de l'appel afin de préparer l'objet «cachedJsonDatas» qui contient le contenu JSON préparé pour la Vue.js (cachedJsonDatas.flattenData) ainsi qu'un tableau d'URLs pour chaque média à télécharger (cachedJsonDatas.filesArray).
3. Ce tableau est ensuite donné à la fonction «prepareForDownloads(cachedJsonDatas.filesArray)» qui crée à partir de ce dernier une série de promesses jQuery qui seront exécutées les unes après les autres (cf. 4.3.5). Tant que toutes les promesses n'ont pas donné de résultat (réussite ou échec), un modal avec une barre de progression empêche l'utilisation de l'application, car la Vue.js sera prête avant la fin des téléchargements.
4. Une fois cette fonction lancée, l'application va, en parallèle des téléchargements, préparer et lancer la Vue.js avec la fonction «prepareForDownloads(cachedJsonDatas.flattenData)».
5. Une fois que toutes les promesses ont donné un résultat, le modal est retiré et l'utilisateur peut utiliser l'application. Non-représenté sur ce schéma, l'application profite du tableau d'URLs pour également contrôler l'entier du contenu de l'application, afin de déterminer s'il sera nécessaire à l'utilisateur de lancer une mise à jour car un ou plusieurs téléchargements ont échoué. Si tel est le cas, un pop-up informe l'utilisateur de la situation.
6. Puisque l'appareil est connecté à internet, le bouton de mise à jour est disponible. Si l'utilisateur souhaite mettre à jour le contenu de l'application, il passera à l'étape suivante (cf. 4.3.4).

### 4.3.3 Contenu local détecté

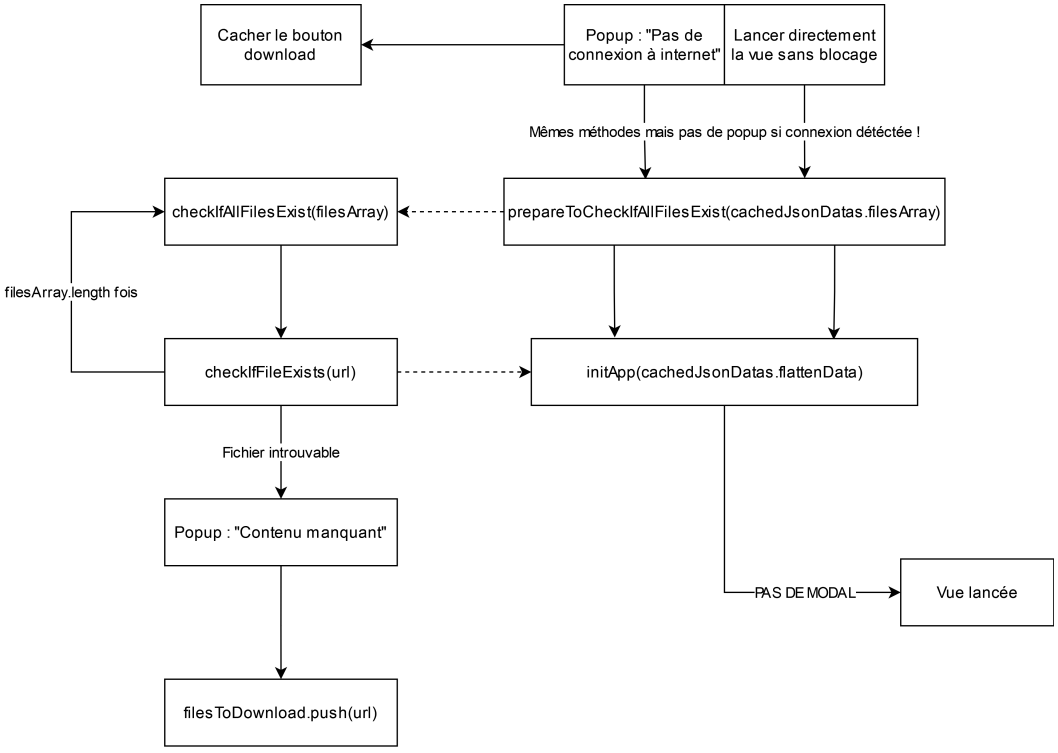


Figure 7 – Contenu local détecté

Grâce à l'étape précédente, il a été déterminé que l'application n'est pas lancée pour la première fois. Quelque soit la connexion de l'appareil, l'application lance les étapes suivantes :

1. À l'étape précédente lors du contrôle d'existence du fichier «cachedJSON.json», l'application a déjà préparé l'objet «cachedJsonDatas» à partir du contenu du fichier. Grâce à ce dernier, l'application contrôle tout d'abord si tous les fichiers que «cachedJsonDatas» a besoin sont bien présents dans les fichiers locaux avec la fonction «checkIfAllFilesExist(cachedJsonDatas.filesArray)».
    - Si un ou plusieurs fichiers sont introuvables, un pop-up informe l'utilisateur que du contenu manque pour utiliser correctement l'application. Il lui est recommandé de trouver une connexion à internet (si aucune connexion n'est détectée) ou de mettre à jour son contenu (si une connexion est détectée). Un tableau de fichier manquant est peuplé et sera utilisé si l'utilisateur souhaite mettre à jour le contenu de l'application. Ce dernier se nomme «filesToDownload» et l'application y met les URLs correspondantes aux fichiers manquants.
    - Si tous les fichiers sont trouvés, aucun besoin d'en informer l'utilisateur.
    - Un modal empêche l'utilisation de l'application lors de ce contrôle, mais cette étape est tellement rapide que le modal ne reste que quelques secondes voir quelques millisecondes.
2. Une fois cette fonction lancée, l'application va, en parallèle des contrôles, préparer et lancer la Vue.js avec la fonction «prepareForDownloads(cachedJsonDatas.flattenData)».

Au final, si l'appareil est connecté à internet, le bouton de mise à jour est disponible. Si l'utilisateur souhaite mettre à jour le contenu de l'application, il passera à l'étape suivante (cf. 4.3.4). Si l'appareil n'est pas connecté à internet, l'utilisateur devra se contenter du contenu local disponible. En effet, puisque la Vue.js est construite uniquement avec des tags HTML dont les attributs sources (src) sont des URLs locaux, si le fichier de cet URL n'existerait pas, l'application affichera un icone "fichier introuvable" mais continuera de fonctionner correctement.

### 4.3.4 Bouton de mise à jour appuyé

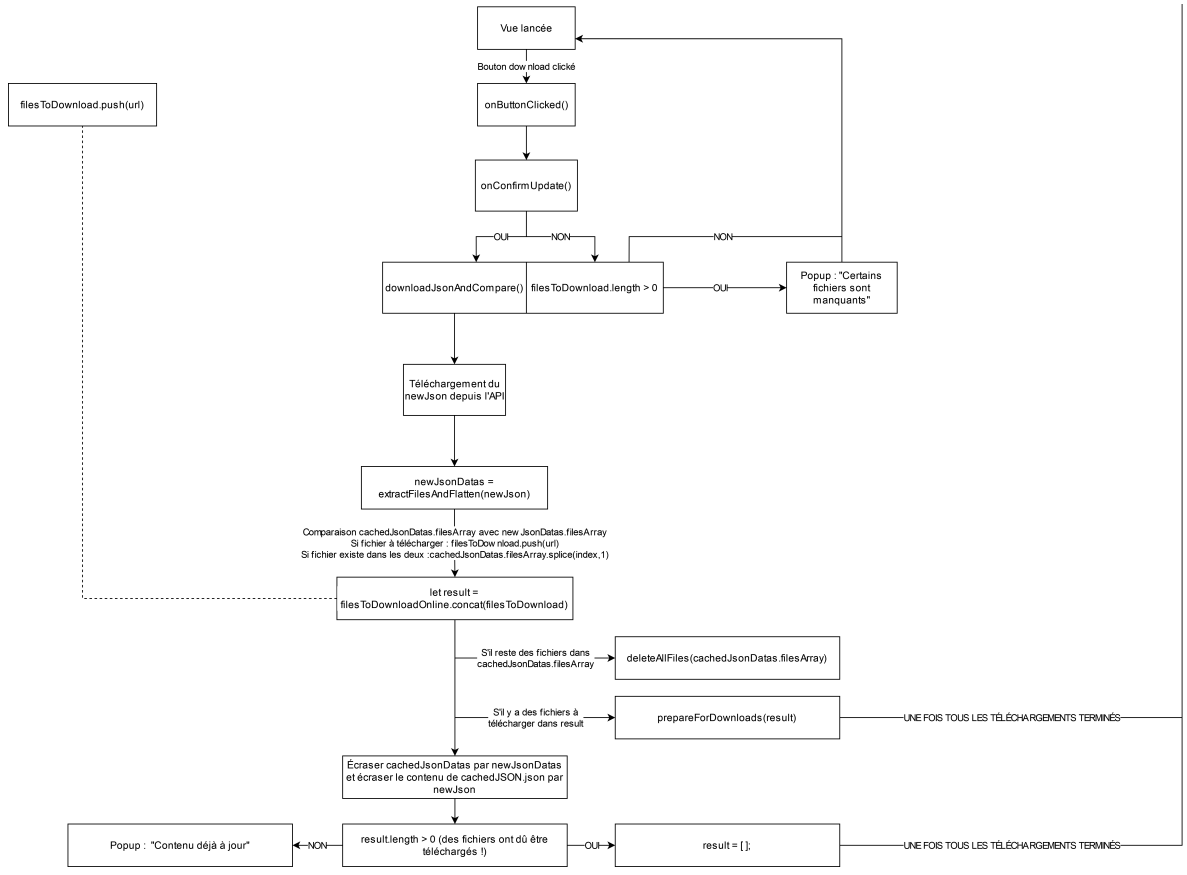


Figure 8 – Bouton de mise à jour appuyé

Grâce aux étapes précédentes, il a été déterminé que l'appareil possède une connexion à internet, sinon le bouton de mise à jour ne serait pas visible. Voici les étapes qui décrivent le fonctionnement de la mise à jour du contenu. Un pop-up demande à l'utilisateur s'il souhaite procéder à une mise à jour du contenu de l'application.

- Si l'utilisateur décline et que du contenu est trouvé dans le tableau «filesToDownload», un pop-up rappelle à l'utilisateur que du contenu manque à l'application et qu'il serait bon de mettre à jour l'application.
- Si l'utilisateur décline et que rien n'est trouvé dans le tableau «filesToDownload», les étapes suivantes sont ignorées et la Vue actuelle est toujours utilisable.
- Si l'utilisateur accepte, la fonction «downloadJsonAndCompare()» est lancée et procède aux étapes suivantes

  1. De la même manière que lors de la première utilisation, l'application commence par télécharger un nouveau contenu JSON de l'appel à l'API. Le résultat «newJson» de cet appel est utilisé pour préparer l'objet «newJsonDatas» qui contient le contenu JSON préparé pour la Vue.js (newJsonDatas.flattenData) ainsi qu'un tableau d'URLs pour chaque média à télécharger (newJsonDatas.filesArray).
  2. L'application procède ensuite à une comparaison entre les objets «newJsonDatas» et «cachedJsonDatas». Elle cherche pour chaque URL dans «newJsonDatas.filesArray» un élément similaire dans «cachedJsonDatas.filesArray».
     - Si un tel élément est trouvé dans «cachedJsonDatas», cela veut dire que «newJsonDatas» utilise également ce média et nous pouvons émettre l'hypothèse que ce dernier est enregistré dans les fichiers locaux. L'application peut donc retirer ce fichier du tableau de «cachedJsonDatas» avec un «splice(...)».
     - Si un tel élément n'est pas trouvé dans «cachedJsonDatas», cela veut dire que ce média a été récemment ajouté à l'API et que «cachedJsonDatas» ne connaît pas son existence, il faut donc le télécharger, son URL est ajouté à un second tableau «filesToDownloadOnline».
  3. L'application concatène le contenu des tableaux «filesToDownloadOnline» et «filesToDownload» (créé et rempli si besoin au démarrage de l'application, cf. 4.3.3) dans un tableau de résultat «result». Si les deux tableaux s'avèrent être vides, cela produit un tableau vide.
  4. S'il reste des fichiers dans le tableau de «cachedJsonDatas», c'est que des fichiers qu'utilisent «cachedJsonDatas» ne sont pas utilisés par «newJsonDatas», ils sont donc en trop et puisque nous avons précédemment émis l'hypothèse que ce qui se trouve dans «cachedJsonDatas» est effectivement enregistré dans les fichiers locaux, il faut donc supprimer les fichiers restants dans «cachedJsonDatas» grâce à la fonction «deleteAllFiles(cachedJsonDatas.filesArray)».
  5. En parallèle, si du contenu est trouvé dans le tableau «result», c'est qu'il faut télécharger ce contenu. L'application utilise donc la fonction «prepareForDownloads(...)» décrite en 4.3.2, mais cette fois-ci avec «result» et en mode mise à jour (cela modifie légèrement la finalité de la chaîne de promesses).
  6. En parallèle, l'application remplace l'objet «cachedJsonDatas» par «newJsonDatas» et remplace le contenu du fichier local «cachedJSON.json» par «newJson» et enregistre le fichier.
  7. Enfin, si aucun de ces contrôles n'est passé, cela veut dire que le tableau «result» est vide et qu'il n'y a donc rien à télécharger, et donc que le contenu est déjà à jour. L'utilisation de l'application est rendue à l'utilisateur.
  8. Si en revanche, un de ces contrôles est passé, cela veut dire que du contenu a été téléchargé ou supprimé, il faut donc recharger l'application. En rechargeant l'application, cette dernière se retrouvera au point 4.3.1 suivi logiquement du point 4.3.3 car «newJson» aura écrasé «cachedJson», la Vue sera donc créée à partir de «newJson» et un contrôle de contenu complet sera également effectué, permettant ainsi de confirmer les hypothèses émises aux étapes précédentes.

### 4.3.5 Système de promesses chaînées

Lors de la réalisation de l'Objectif 2, plusieurs problèmes sont survenus. Il est difficile de dire si ces derniers sont apparus à cause des limitations de Cordova ou des limitations des simulateurs pour les tests, ou s'ils étaient tout simplement dûs à une erreur de codage, un oubli ou encore une faute de frappe. La recherche du coupable n'étant pas importante, il est tout de même intéressant de décrire dans ce document les problèmes les plus intéressants et notamment les solutions trouvées.

Initialement lors du téléchargement de tout le contenu, j'utilisais des XMLHttpRequests afin de récupérer les différents fichiers sont formes de blobs que j'enregistrais finalement en local. Le problème est que ces objets mettent le stream complet du résultat de leurs appels en mémoire vive. C'est pourquoi, quand un gros fichier devait être téléchargé, l'application plante à cause de Memory Leaks. Au début, je pensais que ce problème était dû au fait que je lançais tous les téléchargements en parallèle, remplissant ainsi trop vite la RAM. C'est pourquoi j'ai essayé de chaîner des promesses jQuery afin qu'un prochain téléchargement attende que la promesse actuelle donne un résultat, et ainsi de suite. Cette solution était fonctionnelle mais ne résolvait pas le problème. J'ai donc essayé d'utiliser un plugin qui n'est plus supporté mais qui est encore utilisé par de nombreux développeurs Cordova, le plugin FileTransfer. Ce dernier gère facilement les gros fichiers et évite les problèmes liés au CORS (qui était également un problème avec les XHRs). Les deux soucis avec ce plugin, c'est qu'il n'est plus supporté, il est donc risqué de l'utiliser, et qu'il ne génère pas d'événement de chargement. Il n'est pas possible de créer un barre de progression avec les données actuellement téléchargées sur les données totales à télécharger. Heureusement, grâce au chaînage de promesses, il m'a été possible de créer un barre de progression avec les fichiers téléchargés sur les fichiers à télécharger. L'information est moins précise mais tout de même appréciable par l'utilisateur. Il aurait été possible de segmenter le stream des XHRs en morceaux qui peuvent être finalement reconstitués et enregistrés dans un fichier, mais vu que j'avais trouvé cette double solution, je ne me suis plus repenché dessus. Si d'aventure le plugin FileTransfer ne serait plus utilisable, cette solution serait tout à fait faisable. Voici le code important pour le téléchargement de fichier à partir d'un tableau d'URLs :

```
1  function downloadFile(url){
2      ...
3      var promise = new $.Deferred();
4      window.requestFileSystem(PERSISTENT, 0, function(fs){
5          ...
6          fs.root.getFile(fileName, { create : true, exclusive : false }, function(fileEntry){
7              ...
8              fileTransfer.download(
9                  ...,
10                 function(entry){
11                     promise.resolve();
12                 },
13                 function(err){
14                     promise.reject();
15                 }
16             );
17         }, onErrorCreateFile);
18     }, onErrorLoadFs);
19     return promise;
20 }
21
22 function downloadAllFiles(filesArray){
23     var fileIndex = 0;
24     var current = Promise.resolve();
25     filesArray.forEach(function(url){
26         current = current.then(function(){
27             document.getElementById("myProgress").value = ++fileIndex;
28             document.getElementById("myProgressText").innerHTML = "Fichier " + fileIndex + "
                   sur " + (filesArraySize-1);
29             return downloadFile(url);
30         });
31     });
32     return current;
33 }
34
35 function prepareForDownloads(filesArray, fromUpdate){
36     ...
37     downloadAllFiles(filesArray).then(function(){
38         //Tous les telechargements ont donc ete effectues avec succes
39         ...
40     }).catch(function(e){
41         navigator.notification.alert("Une erreur est survenue lors du telechargement ...");
42     })
43 }
```

Un tableau d'URLs est passé à une fonction «downloadAllFiles(filesArray)». Nous nous intéresserons plus tard à ce «.then(...)». Dans la fonction «downloadAllFiles(...)», je commence par résoudre une première promesse vide que je mets dans une variable «current», ceci me permet d'utiliser son «.then(...)» (= then) plus tard. Je démarre une boucle «forEach» sur le tableau d'URLs en paramètre. Puisque «current» est une promesse vide, je peux utiliser son then avec le premier élément de la boucle. Dans ce premier then, je demande de retourner à partir d'une fonction «downloadFile(url)» une nouvelle promesse qui va remplacer «current» et qui va à son tour pouvoir utiliser son then avec le prochain élément de la boucle une fois sa promesse résolue. Ainsi, je chaîne des thens, quelque soit leurs résultats. L'avantage de ce système est que la promesse n'est démarrée que quand elle est retournée, ce qui fait que la prochaine promesse dans la chaîne devra attendre que le then actuel soit terminé, puis elle sera retournée et remplacera «current» ce qui la démarrera, et ainsi de suite. Dans la fonction «downloadFile(url)», je crée une nouvelle promesse jQuery (qui au contraire des promesses JavaScript classiques ne démarre pas au moment de sa création), et selon si le téléchargement a correctement fonctionné, se résout ou se rejette. Elle est finalement retournée. Le premier then laissé de côté au début de l'explication permet en fait de déterminer si toutes les promesses ont été tenues ou si une ou plusieurs promesses ont été rejetées. Cela me permet d'informer l'utilisateur en conséquence.
En parallèle, j'utilise ce système pour mettre à jour un index qui me permet d'informer l'utilisateur à quel fichier on se trouve dans le téléchargement total à travers un modal.

#### 4.3.6 Serveur local

Un second problème est apparu en utilisant les liens locaux des fichiers enregistrés avec iOS. En effet, Cordova iOS utilisant les WKWebViews, il est interdit d'utiliser des liens locaux de type «cvdfile:///» (raccourci fourni par le plugin File de Cordova) et «file:///» pour accéder au fichier, cela génère des erreur CORS, car les liens accédant à des domaines différents de http ou https sont bloqués. Problème inexistant du côté Android, les WebViews y étant plus permissives.
Afin de contourner cette limitation, le plugin HTTPd permet de créer un tout petit serveur web interne à l'application qui va permettre de leurrer la WKWebView qui va croire aller chercher des fichiers sur un serveur local avec des URLs de type «http://127.0.0.1:8080/XXX.png» alors qu'en vérité, les URLs derrière ceux-ci sont de type «file:///var/mobile/Applications/<UUID de l'app>/Documents/XXX.png» pour iOS.
Détail intéressant, le plugin HTTPd devrait fonctionner avec Android, mais des erreurs empêchent de l'utiliser correctement et fait planter l'application. Puisque je n'ai pas besoin de contourner ce problème sur Android, je n'utilise ce serveur interne que pour iOS.

# 5 Tests

# 6 Conclusion

# 7 Figures et Tables

## 7.1 Liste des figures



## 7.2 Liste des tables

# 8 Bibliographie

[1] Samuel Fringeli. Behind food. https://github.com/samuel-fringeli/behind-food, 2021. [En ligne; Page disponible le 11-octobre-2021].

[2] Juan Martin. Zircleui. https://zircleui.github.io/docs/, 2019. [En ligne; Page disponible le 11-octobre-2021].

[3] Apple Inc. ios wkwebview. https://developer.apple.com/documentation/webkit/wkwebview, 2020. [En ligne; Page disponible le 11-octobre-2021].

[4] W3C. Indexed database api 3.0. https://www.w3.org/TR/IndexedDB/, 2021. [En ligne; Page disponible le 11-octobre-2021].

[5] Microsoft. Qu'est-ce que xamarin ? https://docs.microsoft.com/fr-fr/xamarin/get-started/what-is-xamarin, 2021. [En ligne; Page disponible le 11-octobre-2021].

[6] Microsoft. Qu'est-ce que xamarin.forms ? https://docs.microsoft.com/fr-fr/xamarin/get-started/what-is-xamarin-forms, 2021. [En ligne; Page disponible le 11-octobre-2021].

[7] Microsoft. Xamarin.forms webview. https://docs.microsoft.com/fr-fr/xamarin/xamarin-forms/user-interface/webview?tabs=windows, 2021. [En ligne; Page disponible le 11-octobre-2021].

[8] Hossam Tarek. How do i import the dart indexeddb library in flutter? https://stackoverflow.com/questions/67490597/how-do-i-import-the-dart-indexeddb-library-in-flutter, 2021. [En ligne; Page disponible le 11-octobre-2021].

[9] Alexis Deveria. Can i use... indexeddb. https://caniuse.com/indexeddb, 2021. [En ligne; Page disponible le 11-octobre-2021].

[10] Microsoft. Xamarin.essentials: Connectivity. https://docs.microsoft.com/en-us/xamarin/essentials/connectivity?tabs=android, 2019. [En ligne; Page disponible le 11-octobre-2021].

[11] Apache. cordova-plugin-network-information. https://cordova.apache.org/docs/en/10.x/reference/cordova-plugin-network-information/, 2021. [En ligne; Page disponible le 11-octobre-2021].

[12] WhatIsMyBrowser.com. What's the latest version of chrome? https://www.whatismybrowser.com/guides/the-latest-version/chrome, 2021. [En ligne; Page disponible le 11-octobre-2021].

[13] WhatIsMyBrowser.com. What's the latest version of safari? https://www.whatismybrowser.com/guides/the-latest-version/safari, 2021. [En ligne; Page disponible le 11-octobre-2021].

[14] Apache. Store data - indexeddb. https://cordova.apache.org/docs/en/10.x/cordova/storage/storage.html#indexeddb, 2021. [En ligne; Page disponible le 11-octobre-2021].

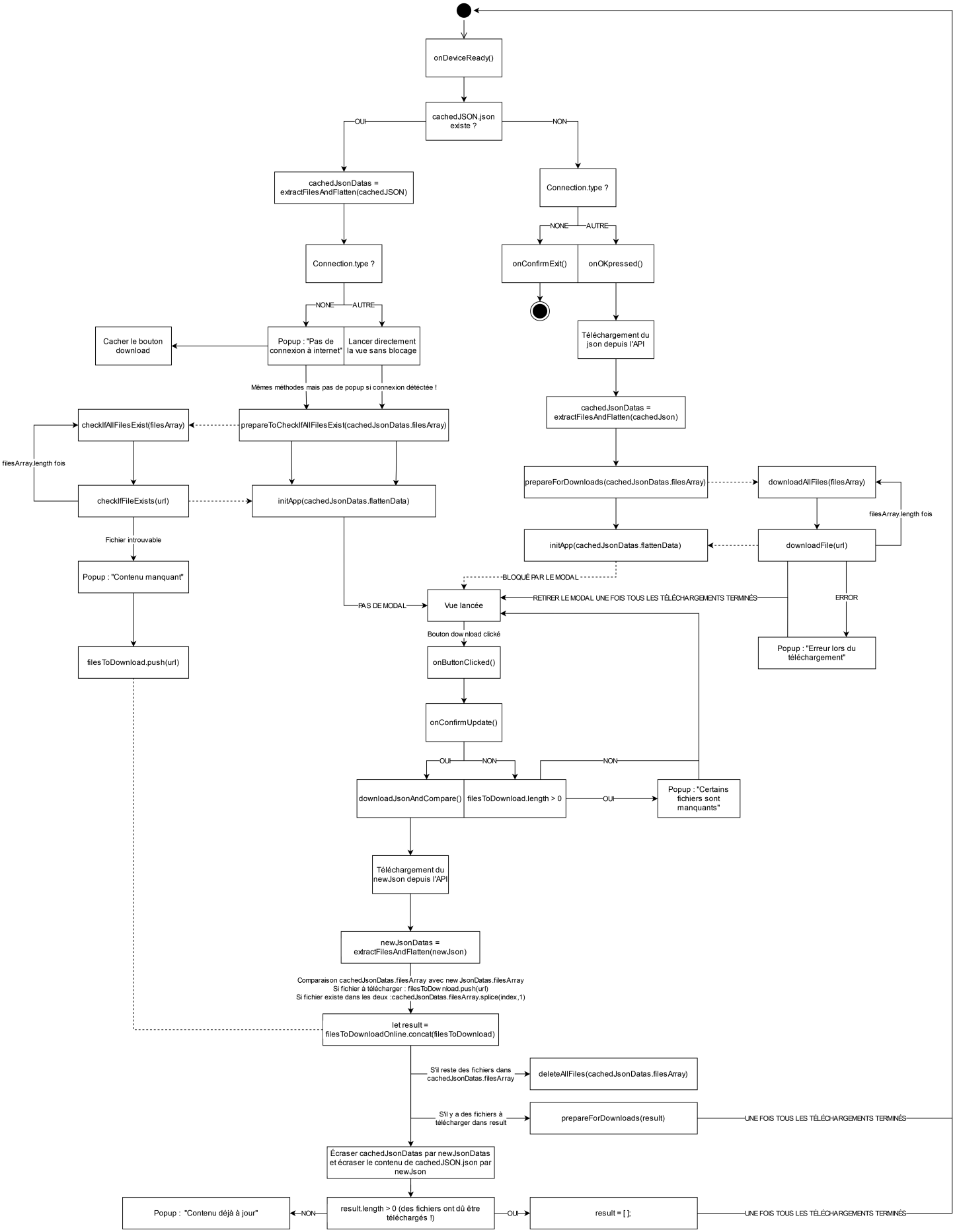


Figure 9 – Le schéma de fonctionnement complet