Table des matières

Liste des tableaux

Liste des figures

1. Glossaire

Abréviation	Signification
ADT	Android Development Tools
AES	Advanced Encryption Standard
AOT	Ahead-of-time
API	Application Programming Interface
ART	Android RunTime
BCL	Base Class Library
CSS	Cascading Style Sheets
GPS	Global Positioning System
HTML	HyperText Markup Language
IDE	Integrated Development Environment
JIT	Just in time
JNI	Java Native Interface
JSON	JavaScript Object Notation
JVM	Java virtual machine
NFC	Near field communication
OS	Operating System
PC	Personal Computer
PCL	Portable Class Libraries
RFID	Radio-frequency identification
SDK	Software development kit
SQL	Structured Query Language
URL	Uniform Resource Locator
UWP	Universal Windows Platform
WWDC	The Apple Worldwide Developers Conference
XAML	Extensible Application Markup Language

2. Introduction

En 1994, IBM sort un téléphone baptisé Simon. Il est considéré comme le 1er smartphone. Déjà équipé d'un écran tactile, il possédait des applications telles que la calculatrice, la gestion des contacts et un calendrier. Au fil des années, les constructeurs n'ont cessé de rendre leurs téléphones de plus en plus intelligents.

Ce fut en 2007 que le marché prit un véritable tournant avec la sortie de l'iPhone par Apple. Son succès commercial poussa tous les constructeurs à sortir des smartphones équipés d'écran tactile.

Dès lors, les smartphones sont en constante évolution et aujourd'hui, ils intègrent les technologies les plus avancées. Toujours plus performant, ils disposent de grande quantité d'espace de stockage et ils sont souvent équipés de fonctionnalités telles qu'un appareil photo, capteur d'empreinte, NFC …

Chaque smartphone est équipé d'un système d'exploitation qui peut être différent d'une marque à l'autre. Voici un tableau des ventes du 2ème trimestre 2016 par système d'exploitation.

Tableau 1 : Vente de smartphone dans le monde au 2T16 (en milliers d'unités)

OS	Unités 2T16	Part de marché (%) 2T16	Unités 2T15	Part de marché (%) 2T15
Android	296'912.8	86.2	271'647.0	82.2
iOS	44'395.0	12.9	48'085.5	14.6
Windows	1'971.0	0.6	8'198.2	2.5
Blackberry	400.4	0.1	1'153.2	0.3
Others	680.2	0.2	1'220.0	0.4
Total	**344'359.7**	**100**	**330'312.9**	**100**

(Gartner, 2016)

La popularité d'un système d'exploitation dépend en grande partie du nombre d'application disponible sa boutique d'application. Il est donc important lors d'un nouveau développement d'application de cibler les plateformes les plus populaires.

Si les dispositifs mobiles sont très utilisés par le grand public, les entreprises n'hésitent pas à équiper leurs employés avec des solutions mobiles. En effet, la mobilité et la flexibilité sont devenues des enjeux importants, car ils offrent des avantages non négligeables.

Tout d'abord, l'utilisation d'applications mobiles par les employés permet une plus grande réactivité, un gain de temps et d'efficacité. Elle permet aussi d'améliorer les processus métier d'une entreprise. Au lieu de remplir un formulaire papier, celui-ci est directement rempli dans une application sans devoir le saisir à nouveau. De plus, l'amélioration de l'interaction entre un employé et un client permet une augmentation de la satisfaction client.

Cependant, l'utilisation d'applications et d'appareils mobiles comporte quelques risques dont il faut tenir compte. L'élément le plus important est celui de la sécurité. Pour cela, il faut que le réseau, les appareils mobiles et les données soient sécurisés. Une bonne gestion des infrastructures et du parc d'appareil est nécessaire.

Le nombre d'applications mobiles ne cesse de croître. Néanmoins, le développement de celles-ci demande des moyens importants lorsqu'on veut qu'elles soient disponibles sur les différentes plateformes. Au fils de ce document, nous allons en apprendre plus sur ces différents systèmes et voir quelles sont les solutions possibles lorsqu'on veut concevoir une nouvelle application.

3. Système d'exploitation mobile

Un système d'exploitation mobile est conçu pour fonctionner sur un smartphone, une tablette ou tout autre dispositif mobile. Aujourd'hui, les trois systèmes les plus importants sont Android, iOS et Windows Phone qui sont détaillés ci-dessous.

3.1 Android

Android Inc. est une startup qui a été fondée en 2003. Son principal objectif était de développer un système d'exploitation pour les appareils photo. Cependant, le marché n'était pas assez large, elle décida de se concentrer sur le développement d'un OS pour smartphone afin de concurrencer Symbian et Windows Mobile. En juillet 2005, Android Inc. fut racheté par Google pour un montant de 50 millions de dollars.

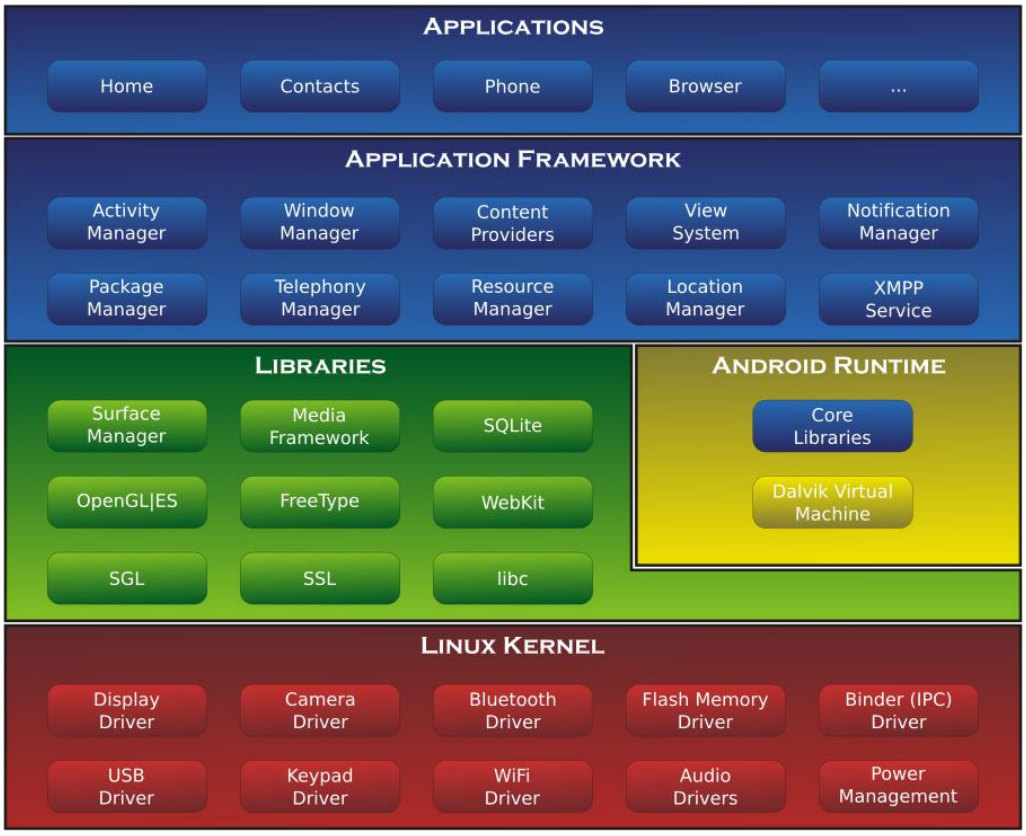
La première version d'Android fut lancée le 23 septembre 2008. À partir de la version 1.5, chaque version à un nom de code basé sur un dessert qui suit un ordre alphabétique. Actuellement, la dernière version est la 6.0.1 «Marshmallow », Google a déjà annoncé la sortie de la version 7.0 qui portera le nom de « Nougat ».

Le système d'exploitation d'Android fonctionne sur plusieurs appareils tels que les smartphones, tablettes, montres, téléviseurs et voitures.

L'App Store officiel des applications Android est le Google Play Store initialement Android Market. Le nombre d'applications disponibles est de 2.3 millions.

L'architecture d'Android est composée de plusieurs couches.

Figure 1 : Architecture Android



(Wikipedia, 2016)

Linux Kernel : Le kernel est basé sur Linux. Il permet de faire le lien entre la partie hardware et le logiciel. Il contient les différents drivers de la partie matérielle.

Libraries : La couche suivante contient plusieurs librairies telles que SQLite, SSL, OpenGL…

Android Runtime : La couche Android Runtime contient des librairies qui permettent aux développeurs d'utiliser un sous-ensemble des libraires Java ainsi qu'une machine virtuelle.

Applications Framework : Les applications Android interagissent avec la couche application Framework. Cette couche permet d'accéder au travers d'API aux fonctionnalités du dispositif.

Application : Toutes les applications installées se trouvent dans cette couche.

Les applications écrites en Java sont exécutées dans une machine virtuelle. Cependant, au lieu d'utiliser une JVM, Android utilise sa propre machine virtuelle.

3.1.1 Dalvik Machine

La Dalvik est une machine virtuelle utilisée pour pouvoir exécuter des programmes sur des appareils dont la puissance est limitée. Le compilateur Java va stocker le bytecode dans un fichier .dex. Lorsqu'une application est exécutée, le bytecode est compilé en instruction pour la machine grâce à la compilation JIT.

3.1.2 ART

Depuis la version Android 4.4 "KitKat", Google a introduit une nouvelle machine virtuelle qui se nomme ART mais elle n'est pas activée par défaut, car encore au stade expérimental. Ce n'est qu'à partir de la version Android 5.0 « Lollipop » qu'elle remplace la machine Dalvik.

ART utilise un compilateur AOT. Par conséquent, les applications utilisant ART s'exécutent plus rapidement, car le bytecode contenu dans le fichier .dex a été compilé en code machine durant l'installation de l'application. Cependant, comme le code a été précompilé, il est stocké dans la mémoire du dispositif ce qui implique que les applications prennent plus de place.

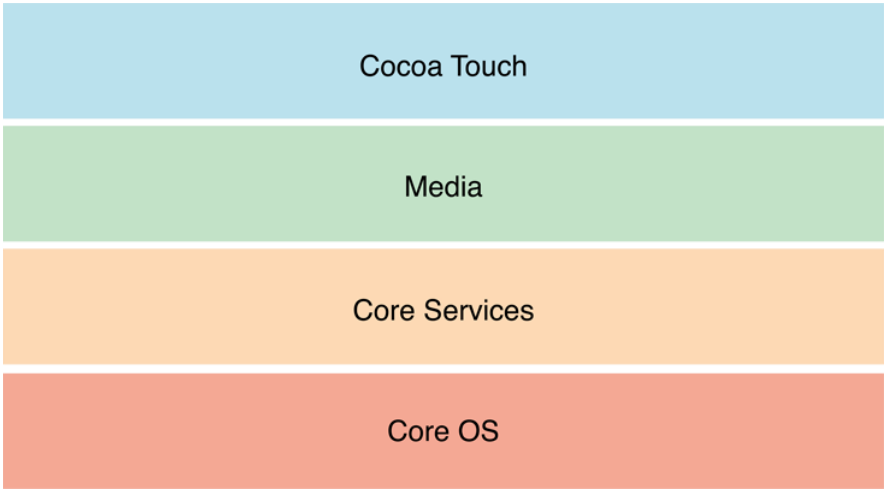
3.2 iOS

iOS est le système d'exploitation mobile développé par Apple et fonctionnant uniquement sur du matériel Apple. La première version de l'OS fut disponible sur le premier iPhone qui est sorti le 29 juin 2007. Précédemment, nommé iPhone OS son nom fut changé en iOS le 7 juin 2010 après avoir acquis auprès de Cisco une licence d'exploitation de la marque. La dernière version d'iOS est la 9.3.4. La version 10 est déjà disponible en version bêta pour les développeurs.

L'App Store est la boutique officielle d'applications d'Apple. Elle fut disponible à partir du 11 juillet 2008 avec la version d'iPhone OS 2.0. Tim Cook a annoncé lors de la WWDC de 2016 que l'App Store contenait 2 millions d'applications et que 130 milliards d'applications ont été téléchargées depuis son lancement.

L'architecture d'iOS est composée de 4 couches différentes.

Figure 2 : Couches système iOS



(Apple, 2014)

Core OS : Cette couche est responsable du système d'exploitation. Elle est en charge de la gestion de la mémoire ainsi que l'accès au matériel du dispositif.

Core Services : Elle contient les API qui permettent aux applications d'accéder à différents services tels que le réseau, les contacts, la base de données…

Media : La couche média permet d'implémenter des fonctionnalités audio, vidéo et graphiques.

Cocoa Touch : Cette couche définit les bases de l'application. Elle contient de nombreuses fonctionnalités comme la gestion des vues, la reconnaissance des gestes, le multitâche, les notifications…

3.3 Windows

Windows 10 Mobile est la dernière version de l'OS de Microsoft. Il est le successeur de la version Windows Phone 8.1.

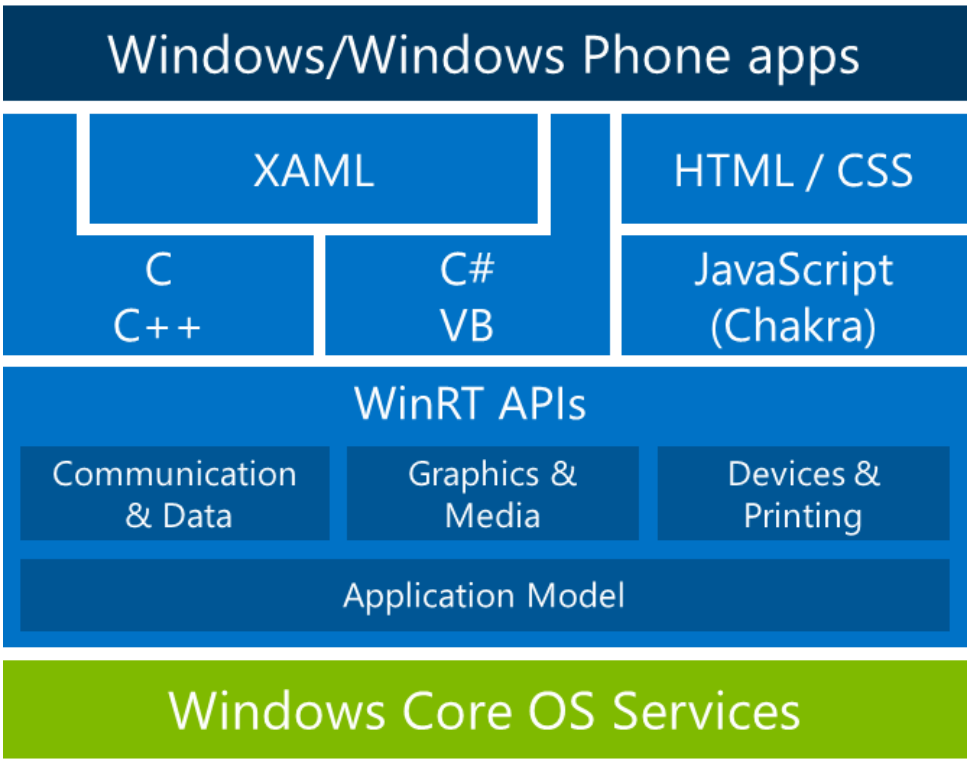
L'une des nouveautés de Windows 10 est le support des applications Universal Windows Platform. L'objectif étant qu'une application développée fonctionne sur PC, tablette, téléphone… L'idée est d'offrir la même expérience utilisateur peu importe le dispositif utilisé.

Aujourd'hui, plus de 300 millions d'appareils tournent sur Windows. L'objectif de Microsoft est d'atteindre 1 milliard en 2018.

La boutique d'applications de Microsoft se nomme Windows Store. Elle compte aujourd'hui 669'000 applications.

L’architecture d’une application Windows est composée de 4 couches.

Figure 3 : Architecture Windows



(Microsoft, 2014)

Core : La couche la plus basse dans la hiérarchie, c’est le kernel qui permet de gérer le matériel du dispositif.

System Services : Au-dessus, la couche contient les API qui permettent d’accéder aux fonctionnalités des appareils.

Model Controller : La couche suivante contient le code qui sera exécuté.

View : Pour finir, la dernière couche contient les vues avec lesquelles les utilisateurs vont interagir.

Les applications développées en UWP peuvent appeler les API WinRT communes à tous les appareils, mais aussi utiliser les API spécifiques (API Win32 et .NET) à chaque dispositif où l’application s’exécute.

5. Développement mobile

Plusieurs techniques existent afin de développer une application mobile. Nous allons voir ci-dessous qu'elles sont les solutions possibles.

5.1 Développement natif

Une application native est spécialement conçue pour une plateforme spécifique. Si l'on veut qu'elle soit disponible sur Android, iOS et Windows Phone, alors il faudra développer trois versions du logiciel. En effet, chaque plateforme possède ses propres outils, langages et spécificités.

Les différents IDE fournis par les trois grands systèmes sont très complets. Cependant, ils fonctionnent tous de manière différente. Une autre différence importante concerne les langages de programmation, chaque plateforme utilise un langage spécifique.

Voici un récapitulatif des langages et IDE utilisés :

Tableau 2 : Récapitulatif des langages et IDE

	Android	**iOS**	**Windows**
Langage	Java	Objective-C/ Swift	C#
IDE	Android Studio	Xcode	Visual Studio

Les API qui existent dans ces trois systèmes fonctionnent de manière différente. Prenons le cas d'un composant qui permet de basculer entre deux états.

Dans les trois plateformes, ce composant est nommé différemment. De plus, son implémentation n'est pas similaire.

- UISwitch est une vue dans iOS.
- Switch est un widget dans Android.
- ToggleSwitchButton est un contrôle dans Windows Phone.

On pourrait penser que les interfaces utilisateurs sont semblables entre les plateformes. Pourtant, elles sont différentes. En effet, chaque système a ses propres règles de conception et de navigation.

Les applications natives ont comme avantages d'offrir la meilleure expérience utilisateur possible. De plus, elles peuvent accéder aux fonctionnalités du dispositif à travers les API fournis.

Cependant, le natif a des désavantages qui ne sont pas négligeables. Comme évoqué au-dessus, le développement et la maintenance sont spécifiques à chaque plateforme.

Un développeur ne possède pas forcément toutes les compétences nécessaires. Tout ceci engendre une augmentation des coûts qui n'est pas forcement supportable par celui qui veut publier une application sur les différents stores.

5.2 Développement web

Les applications web sont conçues afin de tourner dans un navigateur web et elles sont généralement adaptées pour qu'elles puissent s'exécuter dans un navigateur mobile. Le principal avantage d'une application web est sa très grande compatibilité, il suffit d'un navigateur pour pouvoir l'utiliser. Contrairement aux applications natives qui sont installées sur le dispositif, le contenu est récupéré depuis un serveur web à l'aide d'une URL. Les performances de ce type d'application dépendent en partie du navigateur, de la vitesse de connexion et des performances du dispositif. Une connexion internet est souvent nécessaire pour pouvoir l'utiliser. Le plus gros désavantage de ce type d'application est l'accès aux fonctionnalités de l'appareil, même si avec le standard HTML5, il est possible de rendre les applications plus riches en accédant à quelques fonctionnalités.

5.3 Développement multiplateforme

Généralement, lorsqu'on désire développer une nouvelle application, on aimerait qu'elle soit disponible sur les principales plateformes. L'objectif du développement multiplateforme consiste donc dans l'élaboration d'une seule application qui fonctionnera sur les différents systèmes tout en résolvant les problèmes liés au développement natif.

Il y a plusieurs avantages à utiliser ces outils. Tout d'abord, il n'est plus nécessaire d'avoir des compétences pour chaque système d'exploitation, un seul langage de programmation suffit. Ensuite, le temps de développement est fortement réduit. Les coûts sont ainsi diminués, car une seule solution doit être développée et maintenue. Pour finir, en déployant une application sur tous les stores, cela permet de cibler un plus grand marché et de pouvoir engendrer des revenus en cas d'application monétisée.

6. Plateformes évaluées

Ils existent plusieurs solutions afin de procéder à un développement multiplateforme. Beaucoup d'outils sont disponibles, mais il ne m'est pas possible de tous les évaluer. C'est pourquoi après discussion avec Genève Aéroport le choix s'est porté sur les deux solutions les plus populaires qui sont Xamarin et Cordova.

6.1 Xamarin

Xamarin est une entreprise qui a été fondée en mai 2011 par les concepteurs du Projet Mono et rachetée par Microsoft en février 2016.

Mono a été lancé en 2001 par Miguel de Icaza au sein de la société Ximian. Elle fut rachetée par Novell en 2003. Mono est une plateforme de développement open source qui est basée sur .NET qui permet de développer des applications multiplateformes en C#.

Xamarin est un outil qui permet de développer des applications natives pour Android, iOS et Windows avec comme seul langage le C#. Il est unique, car il combine la puissance des plateformes natives et possède de nombreuses fonctionnalités :

- Xamarin contient des liaisons avec les SDK des plateformes iOS et Android. De plus, ses liens sont fortement typés ce qui signifie qu'en cours de développement, l'utilisation et la compilation sont plus faciles. Cela conduit à moins d'erreurs d'exécution et des applications de qualité supérieure.
- La possibilité d'appeler du code en : Objective-C, Java, ou C/C++. D'utiliser des librairies existantes.
- Le C# est un langage moderne, il possède de nombreuses améliorations par rapport à l'Objective-C et le Java.
- Les applications Xamarin peuvent utiliser la Base Class Library qui est une bibliothèque standard pour les langages basée sur le .NET. La BCL contient de nombreuses classes qui permettent d'utiliser des fonctions telles que l'interaction avec une base de données, le réseau, la sérialisation…De plus, il est facilement possible d'intégrer des composants existants, déjà développés pour l'utilisation de Xamarin.
- Pour pouvoir développer une application, la société fournit un IDE Xamarin Studio qui est compatible avec Mac et Windows. De plus, sur Windows, une deuxième possibilité est offerte qui est l'utilisation de Visual Studio. Ces deux IDE sont modernes et ils incluent plusieurs fonctionnalités.
- Comme on peut le voir, le but de Xamarin est de faire du développement multiplateforme. Certaines applications peuvent partager jusqu'à 90% du code ce qui a pour conséquence de réduire les coûts, le temps de développement et de pouvoir cibler les trois plateformes les plus populaires.

Le développement d'application Xamarin nécessite un environnement spécial.

Tableau 3 : Compatibilité des outils Xamarin

	Mac OS X	Windows
Environnement de développement	Xamarin Studio	Visual Studio
Xamarin.iOS	Oui	Oui (avec un Mac)
Xamarin.Android	Oui	Oui
Xamarin.Forms	iOS et Android	Android, Windows Phone et iOS avec un Mac

(Xamarin, 2016)

Plus de détails sont disponibles directement sur le site de Xamarin à l'adresse suivante : https://developer.xamarin.com/guides/cross-platform/getting_started/requirements/

6.1.1 Approches multiplateformes

Il y a deux approches pour le développement d'applications multiplateformes avec Xamarin qui sont détaillées ci-dessous :

6.1.1.1 Native

L'approche native permet au développeur d'élaborer les interfaces utilisateur spécifiques à chaque plateforme et de partager le code métier.

Figure 4 : Xamarin Native

(Xamarin, 2014)

Cependant, celle-ci nécessite d'avoir de l'expérience dans le domaine des applications natives.

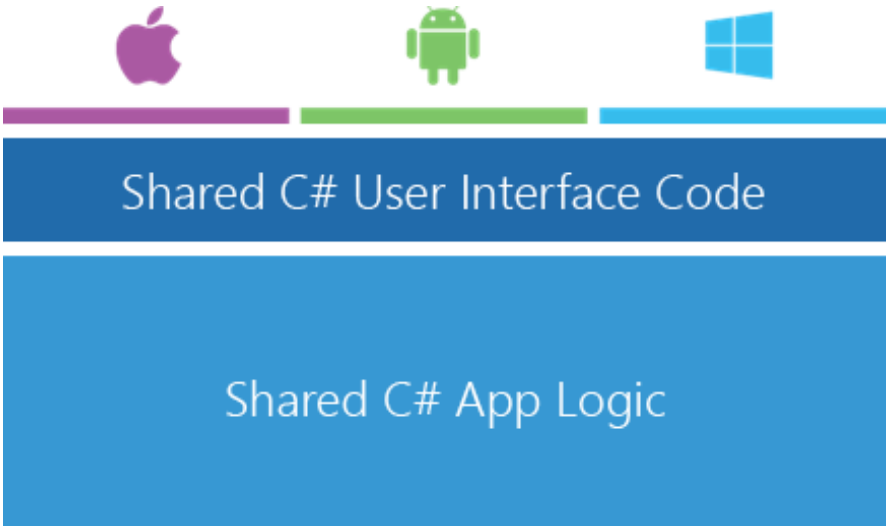
Cette approche devrait être utilisée pour :

- Les applications qui accèdent à du code spécifique à la plateforme.
- Quand l'expérience utilisateur est plus importante que le partage de code.
- Les interactions nécessitent un comportement natif.

6.1.1.2 Forms

Xamarin.Forms permet de partager le code de l'interface utilisateur ainsi que le code métier. Il y a deux techniques pour créer les interfaces : soit dans le code source en C#, soit en utilisant le langage XAML. De cette manière, l'interface utilisateur va utiliser les contrôles natifs de la plateforme cible. Xamarin.Forms permet le développement d'applications à partir de la version Android 4.0.3, iOS 6.1 et Windows Phone 8.1.

Figure 5 : Xamarin Forms



(Adapté de Xamarin, 2014)

D'après Xamarin, l'approche Forms est meilleure pour :

- Les applications qui requièrent peu de code spécifique à la plateforme
- Le cas où le partage de code est plus important que l'interface utilisateur.
- Les développeurs ayant une bonne connaissance du XAML.

6.1.2 Partage de code

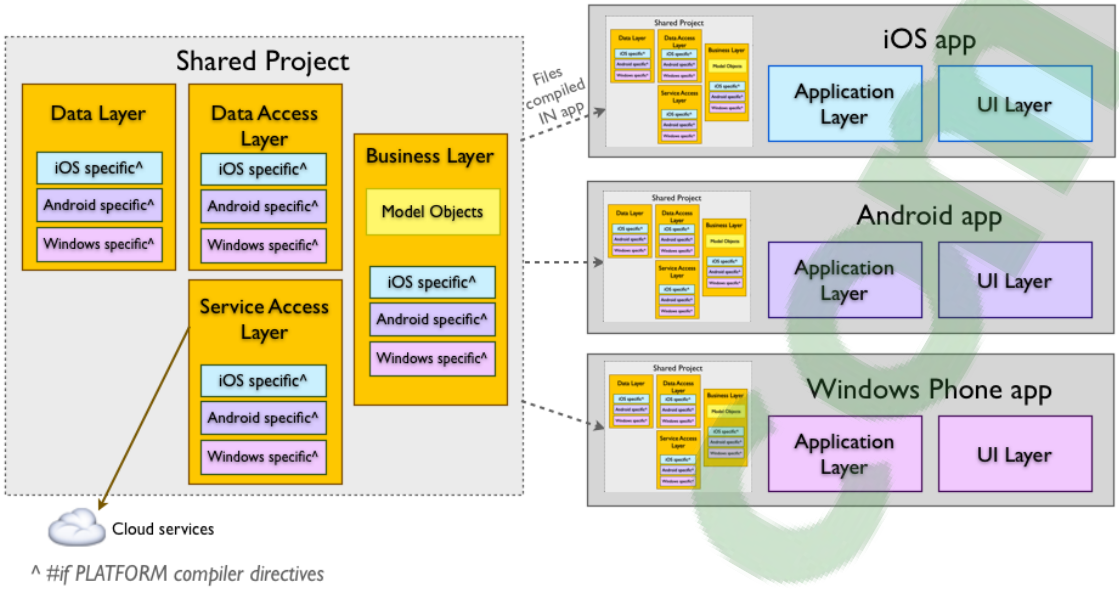
Après avoir choisi le type de développement, il faut choisir de quelle manière le code en commun entre les plateformes va être partagé. Pour cela, il existe deux possibilités Shared Project et Portable Class Librairies.

6.1.2.1 Shared Project

L'approche Shared Project permet le partage du code entre les différents projets en utilisant les directives du compilateur (`#if __ANDROID__`) pour gérer le code spécifique à chaque plateforme. De plus, ce type de projet permet la possibilité d'utiliser des références spécifiques à la plateforme.

Le diagramme suivant montre le fonctionnement.

Figure 6 : Shared Project



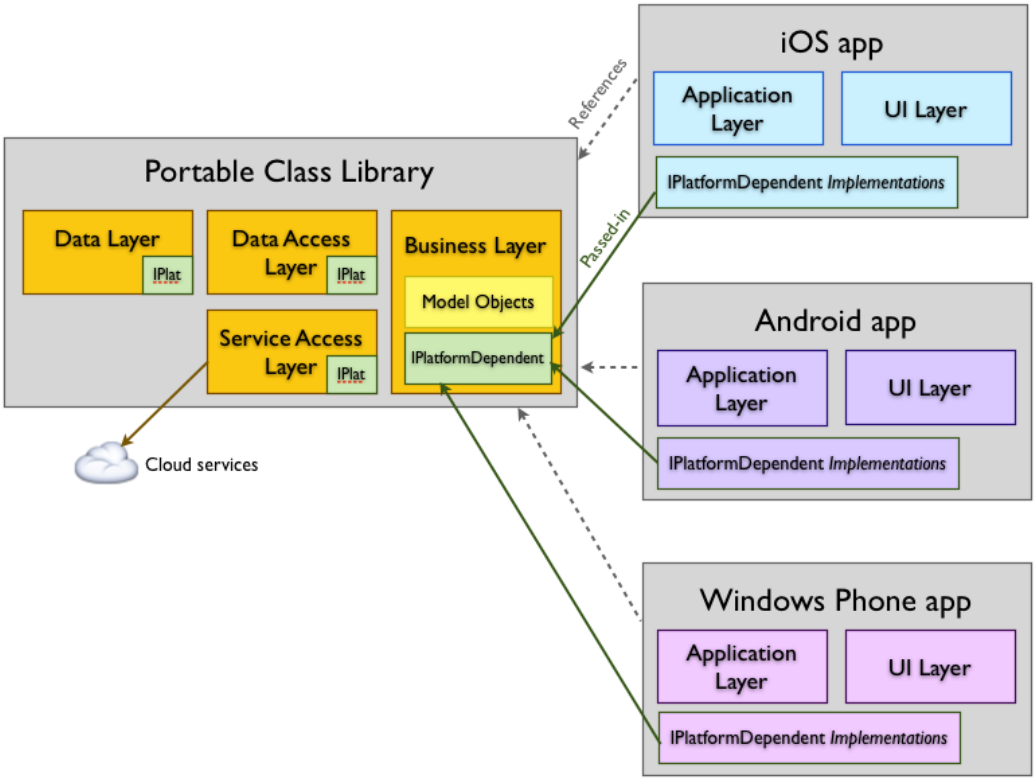
(Xamarin, 2016)

6.1.2.2 Portable Class Librairies

L'approche PCL permet d'écrire une libraire qui va pouvoir s'exécuter sur les différentes plateformes. Cela a comme avantage qu'un projet PCL peut être référencé par d'autres projets. Cependant, comme une PCL est partagée entre différentes plateformes, il n'est pas possible de référencer du code spécifique. Heureusement, il existe une solution pour contourner ce problème en utilisant le service de dépendance qui permettra de coder l'implémentation native.

Le diagramme ci-dessous montre l'architecture d'un projet utilisant la PCL et le service de dépendance pour les fonctionnalités spécifiques.

Figure 7 : Portable Class Library



(Xamarin, 2016)

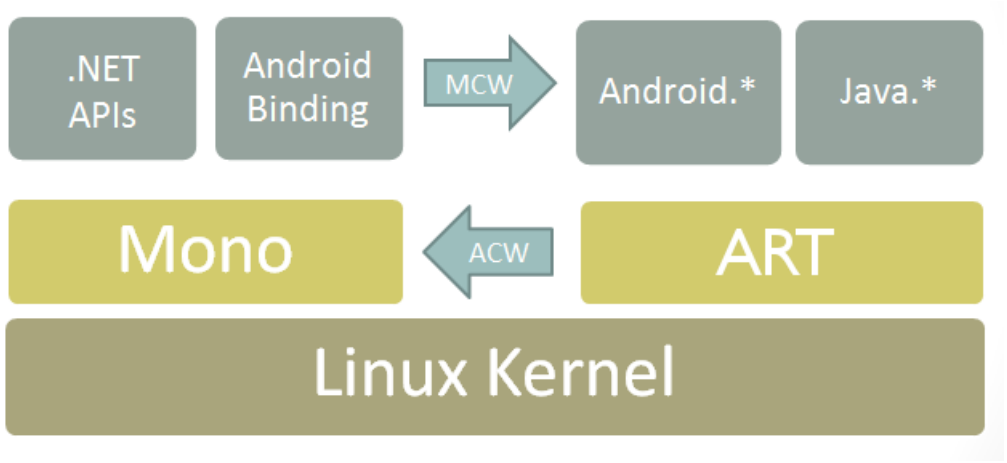
6.1.3 Compilation

Regardons de plus près comment fonctionne la compilation sur les différentes plateformes avec Xamarin.

6.1.3.1 Xamarin.Android

Lors de l'exécution d'une application sous Android, l'environnement Mono fonctionne en parallèle avec la machine virtuelle utilisée par le système. Lors de la compilation, le C# est compilé en langage intermédiaire (IL) et encapsulé avec la MonoVM. Les API .NET sont liées grâce au JNI qui permet au code Java qui tourne sur une machine virtuelle Java d'être appelé par du code non Java.

Figure 8 : Compilation Xamarin.Android

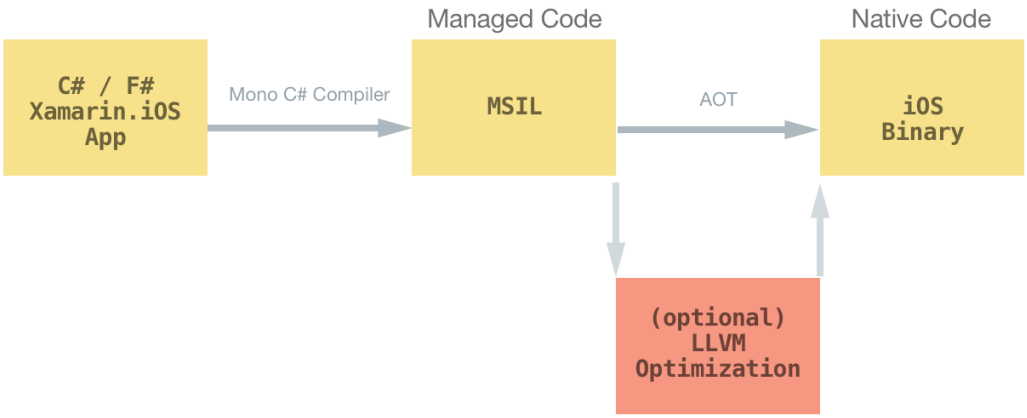


(Xamarin, 2016)

6.1.3.2 Xamarin.iOS

La compilation pour les applications iOS est faite en avance. C'est la compilation AOT. En effet, Apple a mis en place une sécurité dans iOS qui empêche l'exécution de code dynamique sur un dispositif. Pour être sûr de respecter le protocole imposé, le compilateur Mono va produire du code en MSIL (Microsoft Intermediate Language). Par la suite, la compilation AOT va produire le code natif pour iOS. Si l'application a été codée sous Visual Studio, alors la compilation devra être exécutée sur un Mac.

Figure 9 : Compilation Xamarin.iOS



(Xamarin, 2016)

6.1.3.3 Windows Phone

Contrairement aux plateformes citées ci-dessus, les applications Windows n'ont pas besoin des outils de Xamarin pour la compilation. Le C# est compilé en IL qui sera ensuite compilé par le dispositif.

6.2 Cordova

Apache Cordova est un Framework de développement mobile open source. Il permet de développer des applications multiplateformes à l'aide des technologies du web (HTML5, CSS et JavaScript)

En plus de supporter les trois principales plateformes, les systèmes suivants sont aussi supportés :

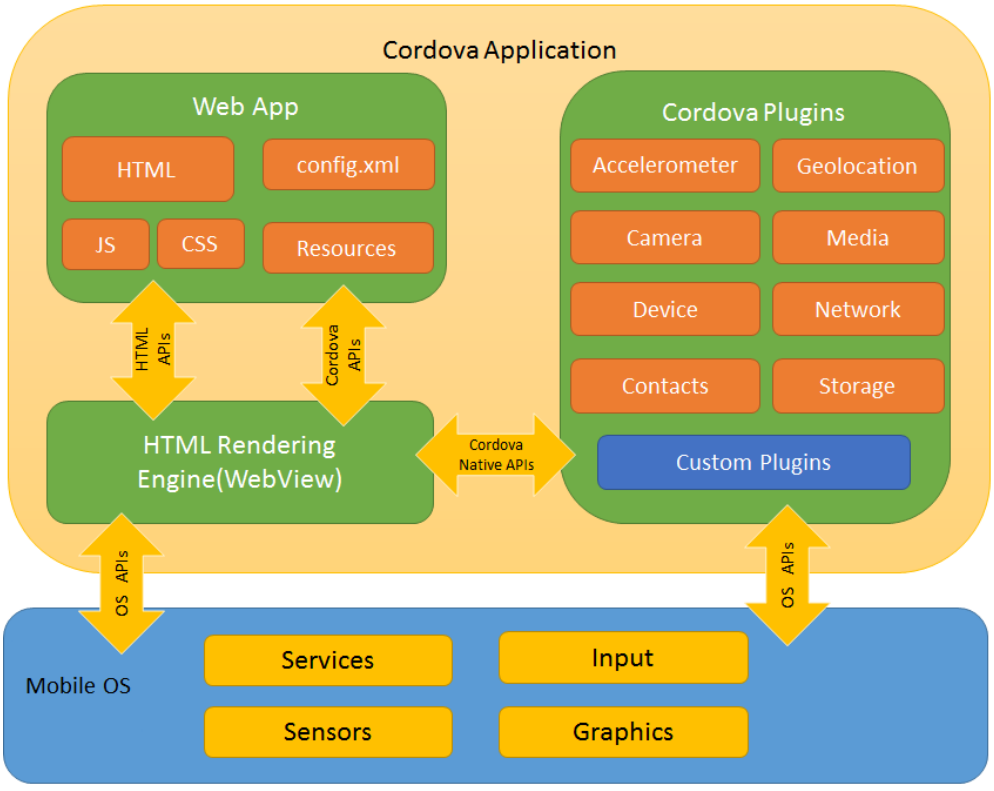
- Blackberry
- Ubuntu
- Firefox OS
- Fire Os
- Tizen

Les applications ne sont pas compilées dans le langage natif de la plateforme choisie. Elles sont exécutées dans un conteneur spécifique à chaque plateforme (WebView).

6.2.1 Architecture

Le fonctionnement d'une application développée avec Cordova est différent de Xamarin. Ci-dessous, une explication concernant l'architecture de Cordova.

Figure 10 : Architecture Cordova



(Cordova, 2016)

6.2.1.1 WebView

Une WebView est un composant qui est disponible nativement sur les appareils mobiles. Elle est basée sur le navigateur mobile de l'OS. Elle va permettre d'afficher l'interface utilisateur qui est une page HTML. C'est grâce à ce composant qu'on va pouvoir distribuer notre application sur les différents stores.

6.2.1.2 Web App

C'est la partie où le code de l'application réside. L'application est implémentée à travers des pages web qui vont s'exécuter dans la WebView.

6.2.1.3 Plugins

Les plugins sont des éléments importants dans Cordova. Ils permettent à la WebView de communiquer avec la plateforme native. Les plugins permettent l'accès aux fonctionnalités du dispositif qui ne sont généralement pas accessible aux applications web.

De base, Cordova fournit des plugins qui permettent d'accéder à des fonctionnalités du téléphone telles que, l'accéléromètre, le GPS, le niveau de batterie, etc... Si un plugin n'est pas fourni, il faudra le développer. Contrairement à l'application, les plugins sont codés dans le langage natif de la plateforme.

6.3 Ionic

Ionic est un framework HTML5 créé par Drifty Co. en 2013 pour créer des applications mobiles multiplateformes. Basé sur AngularJS pour la partie web, il doit être utilisé avec Cordova pour la partie native des applications. Son principal objectif est de faciliter la conception des interfaces utilisateur.

Pour pouvoir développer, il faut Node.js afin de pouvoir installer Apache Cordova. De plus, pour le développement Android, il est nécessaire d'installer le JDK de Java ainsi que le SDK Android. Pour iOS, Ionic va créer un projet Xcode qui devra être ouvert à l'aide d'un Mac.

Les plateformes supportées par Ionic sont iOS 7+ et Android 4.1+. Ionic a annoncé que la version 2 supporte le développement d'application UWP.

7. Critères d'évaluations

7.1 Accès aux fonctionnalités

Comme on a pu le voir, les applications natives sont conçues pour fonctionner sur une plateforme spécifique. L'un des avantages c'est qu'elles peuvent accéder aux fonctionnalités du dispositif. Ces fonctionnalités peuvent être de type hardware comme l'accès au GPS, l'accéléromètre, l'appareil photo, le Bluetooth etc... Mais elles peuvent aussi être de type software telles que l'accès au contact…

Pour pouvoir accéder à une de ces fonctionnalités, il faut que le constructeur l'ait autorisé. Prenons comme exemple un iPhone qui dispose d'une puce NFC. Il n'est pas possible pour un développeur d'accéder à ce composant, car celui-ci a été bridé par Apple.

Les applications codées à l'aide de Xamarin sont aussi natives. Elles peuvent par conséquent accéder aux mêmes fonctionnalités que les applications développées avec les outils constructeurs.

Comme vu dans la partie plugin de Cordova, ce type d'application a la possibilité d'accéder aux différentes fonctionnalités du dispositif en appelant du code natif via un plugin. Il n'y a donc aucune limitation.

Contrairement aux plateformes citées ci-dessus, les applications qui tournent dans un navigateur web sont celles qui sont le plus limitées en termes d'accès aux fonctionnalités. HTML5 intègrent des API qui permettent de rendre les applications web plus riches. La liste ci-dessous nous donne un aperçu des API utilisables dans les navigateurs mobiles.

Tableau 4 : API HTML5

API	**Safari iOS**	**Android Browser** **Navigateur Android jusqu'à la version 4.4**	**Google Chrome**	**Internet Explorer**
Plateforme	iPhone, iPad	Téléphones et tablettes	Appareils Android 4.0+	Windows Phone
Application Cache				
Web storage				
Web SQL storage				
IndexedDB				
Geolocation				

API	Safari iOS	Android Browser Navigateur Android jusqu'à la version 4.4	Google Chrome	Internet Explorer
Multimedia Lecteur audio et vidéo				
Web Workers Processus en arrière-plan				
Viewport definition Contrôle de la mise en page				
Canvas API Dessin 2D				
SVG				
Motions Sensors Accéléromètre, gyroscope, magnétomètre				
Form Virtual Keyboards Entrées de texte avec différents claviers				
Form New Controls				
Touch Events Touchstart, touchEnd…				
Pointer Events				
CSS 3 Basic				
CSS 3 Transforms 2D				
CSS 3 Transforms 3D				
CSS 3 Transitions				
CSS 3 Animations				
CSS 3 Regions				
Position : fixed support				
Position : sticky support				
WebGL Canvas 3D				
Navigation Timing API Mesure la vitesse de chargement d'une page				
File API Ouverture de fichier				

API	Safari iOS	Android Browser Navigateur Android jusqu'à la version 4.4	Google Chrome	Internet Explorer
FileSystem API Fichier système virtuel pour le stockage local				
HTML Media Capture Prendre une photo, vidéo ou de l'audio				
Web Speech API Reconnaissance vocale				
HomeScreen Webapp Icône dans le menu				
Network Information API Informations sur la connexion de l'appareil				
XMLHttpRequest 2.0				
CORS Cross-origin resource sharing				
Server-Sent Events Permets de recevoir des événements du serveur				
Web Sockets				
Media Capture Stream				
WebRTC Communication en temps réel				
Web Audio API				
Notification API				
Service Workers				
Animation Timing API Timer pour les animations				
Full Screen API				
Page Visibility API				
Battery Status API				
Ambient Light Events Informations sur la luminosité ambiante				
Vibrations API				
Remote Debugger				

(Adapté de Mobilehtml5, 2016)

7.1.1 Récapitulatif

Si dans le cas d'un développement d'application, celle-ci doit pouvoir accéder aux fonctionnalités de l'appareil alors les applications natives, Xamarin ou Cordova sont le meilleur choix contrairement au web qui n'offre que très peu de possibilités, avec des risques d'avoir des problèmes de compatibilité liés au support de HTML5 par les navigateurs.

7.2 Stockage local

La plupart des dispositifs mobiles sont souvent connectés à internet à l'aide du wifi ou des données cellulaires. Cela permet aux applications de pouvoir communiquer avec des services web afin de pouvoir traiter des données. Cependant, il arrive que dans des cas l'appareil ne puisse pas se connecter à internet pour diverses raisons.

Dans les cas où une connexion n'est pas établie, l'utilisateur s'attend à ce que son application continue de fonctionner de manière optimale. Il est donc important de pouvoir stocker des données du côté client.

Le fait de rendre son application utilisable hors ligne a aussi comme avantage un gain de performance. En effet, comme les données sont stockées localement, le temps de réponse est beaucoup plus rapide que s'il fallait passer par le réseau pour pouvoir consommer un service web.

Il existe plusieurs solutions différentes qui permettent d'implémenter la persistance des données que nous allons voir ci-dessous.

7.2.1 SQLite

SQLite est une librairie écrite en C qui permet d'implémenter une base de données relationnelle. Développée par le Dr. Richard Hipp, la première version fut publiée en août 2000. Actuellement, la dernière version est la 3.13.0. Le code source étant dans le domaine public, chacun est libre de l'utiliser, de le modifier ou de le publier. SQLite est la base la plus implémentée dans le monde, elle tourne sur la quasi-totalité des appareils mobiles et bon nombre de systèmes embarqués.

Contrairement à la plupart des bases de données, SQLite n'a pas besoin d'un processus serveur séparé (serverless). De plus, son fonctionnement ne nécessite aucune configuration. Une base de données est contenue dans un fichier unique. Très légère, la libraire ne pèse pas plus de 500 KB et la plupart des fonctionnalités de la norme SQL-92 sont implémentées.

SQLite est compatible avec toutes les plateformes natives ainsi que Xamarin et Cordova.

7.2.2 Realm

Realm est une base de données conçue pour être utilisée sur des appareils mobiles. Rapide, légère et très simple à intégrer à une application, elle a pour vocation de remplacer SQLite. Realm étant open source, son code est disponible sur Github. La documentation est disponible sur son site internet. En cas de problème, il est possible de poser des questions sur Github, Stackoverflow et Twitter.

Utilisé par les plus grandes entreprises telles qu'Amazon, Google, eBay… Realm tourne sur plusieurs millions d'applications utilisées tous les jours.

Facile d'utilisation, les données sont directement exposées en tant qu'objet et interrogeable par le code. Realm supporte l'encryption de la base de données avec le chiffrement AES-256.

Voici un exemple de code pour Xamarin :

Figure 11 : Exemple d'implémentation de Realm Xamarin

```
// Define your models like regular C# classes
public class Dog : RealmObject
{
    public string Name { get; set; }
    public int Age { get; set; }
    public Person Owner { get; set; }
}

public class Person : RealmObject
{
    public string Name { get; set; }
    public IList<Dog> Dogs { get; }
}

var realm = Realm.GetInstance();

// Use LINQ to query
var puppies = realm.All<Dog>().Where(d => d.Age < 2);

puppies.Count(); // => 0 because no dogs have been added yet

// Update and persist objects with a thread-safe transaction
realm.Write(() =>
{
    var mydog = realm.CreateObject<Dog>();
    mydog.Name = "Rex";
    mydog.Age = 1;
});

// Queries are updated in real-time
puppies.Count(); // => 1

// LINQ query syntax works as well
var oldDogs = from d in realm.All<Dog>() where d.Age > 8 select d;

// Query and update from any thread
new Thread(() =>
{
    var realm2 = Realm.GetInstance();

    var theDog = realm2.All<Dog>().Where(d => d.Age == 1).First();
    realm2.Write(() => theDog.Age = 3);
}).Start();
```

(Realm, 2016)

Realm est disponible pour les différentes plateformes suivantes :

Tableau 5 : Produit Realm

	Realm version	Plateforme
Realm Java	1.2.0	Android 2.3+
Realm Objective-C	1.1.0	iOS 7+
Realm Swift	1.1.0	iOS 8+
Realm Xamarin	0.78.0	Xamarin iOS 9.8.2.22 pour iOS 7+ Xamarin.Android 6.1.2.21 pour Android 2.3.3+

7.2.3 Web Storage

Web Storage permet de stocker des données dans un navigateur internet. Deux interfaces sont disponibles pour l'utilisation du Web Storage qui sont « sessionStorage » et « localStorage ». Les données sont stockées par paire clé/valeur.

Le sessionStorage permet d'enregistrer des données pour la durée de la session. Quand la session de la page est terminée, les données sont effacées. C'est donc une solution de stockage temporaire.

Le localStorage permet de mémoriser des données de façon permanente sans limites de durée de vie. La portée est bien plus grande que le sessionStorage. Les données sont accessibles entre plusieurs onglets / fenêtres du même domaine tant qu'elles sont dans le même navigateur.

Avantages :

- Utilisation simple
- Compatible avec la quasi-totalité des navigateurs

Inconvénients :

- Il n'est possible d'enregistrer que des strings. Les structures de données plus complexes doivent être sérialisées.
- Faible performance quand la quantité de données est importante.
- Pas d'indexation
- API synchrone
- Stockage limité

Tableau 6 : Stockage disponible pour Web Storage

	Chrome Android 51	**Safari iOS 9**
LocalStorage	10 MB	5 MB
SessionStorage	10 MB	Illimité

7.2.4 Web SQL Database

Web SQL Database est une autre solution qui permet le stockage de données du côté client. Cette API permet l'utilisation d'une base de données SQL dans le navigateur internet.

Avantages :

- Bonne performance. Les données peuvent être indexées.
- API asynchrone

Inconvénients :

- La structure doit être définie à l'avance
- API obsolète qui n'est plus supportée par Internet Explorer et Firefox

Tableau 7 : Stockage disponible pour Web SQL Database

	Chrome Android 51	**Safari iOS 9**
Web SQL DataBase	Géré par l'API Quota management.	50 MB

7.2.5 IndexedDB

Indexed Database API permet aussi le stockage de données dans un navigateur internet. Contrairement à Web SQL qui est une base de données relationnelle. IndexedDB est une base de données qui contient des objets au format JSON indexés par des clés.

Avantages :

- Bonne performance. Les données peuvent être indexées.
- API asynchrone

Inconvénients :

- API récente

Tableau 8 : Stockage disponible pour IndexedDB

	Chrome Android 51	**Safari iOS 9**
IndexedDB	Géré par l'API Quota management.	Géré par l'API Quota management.

7.2.6 Récapitulatif

Les solutions présentées ci-dessus nous offrent la possibilité de stocker des informations directement sur les différents appareils. SQLite propose une base de données de type relationnelle. Son principal avantage en plus d'utiliser la syntaxe SQL est sa grande compatibilité avec les différentes plateformes. Realm quant à lui propose une base de données orientée objet. Plus performante qu'une base de donnée SQLite et facile d'utilisation, c'est une bonne alternative. Cependant, celle-ci n'est actuellement pas compatible avec Windows ni Cordova.

En plus de pouvoir intégrer SQLite aux applications Cordova, il est possible d'utiliser les solutions de stockage des navigateurs. Concernant les applications web, il existe plusieurs solutions qui permettent de sauvegarder des données du côté client. Cependant, même si les données sont persistantes, l'utilisateur peut supprimer les données du navigateur. Ce n'est donc pas une solution optimale si on veut les garder dans le temps. Ce problème ne se pose pas avec les applications Cordova, car les données sont intégrées à l'application.

7.3 Communauté, documentation et version

Avant de commencer le développement d'une application, il est important de savoir si la technologie utilisée est populaire. L'utilisation d'une technologie récente peut avoir de nombreux problèmes. En effet, si elle n'est pas assez utilisée par les développeurs, la communauté risque d'être faible et en cas de soucis, il sera certainement plus difficile de trouver une solution. De plus, l'éditeur risque d'abandonner le suivi de sa technologie.

Un bon moyen de savoir si la technologie utilisée a une bonne communauté, c'est de se baser sur le nombre de questions posées sur le site Stackoverflow. Plus il y a de questions et de réponses, plus il sera facile de trouver une solution à un problème qui sera peut-être semblable au notre.

Le numéro de version peut aussi donner une indication par rapport au suivi de la technologie. Savoir qu'une prochaine version est prévue nous prouve que son développement est toujours en cours. Les bugs vont être corrigés et de nouvelles fonctionnalités vont être implémentées.

7.3.1 Android

Les premières applications Android étaient développées avec l'IDE Eclipse combiné avec le plugin ADT. En 2015, Google a mis fin au support du plugin et propose au développeur d'utiliser l'IDE officiel pour le développement d'application mobile qui est Android Studio.

La première version stable d'Android Studio est sortie le 8 décembre 2014. Android Studio est basé sur Intellij IDEA, il contient de nombreux outils pour le développement Android comme un constructeur d'interface graphique et un débogueur. La dernière version stable est la 2.1.3.

La dernière version de l'Android SDK est la 25.1.6. Android étant open source, chacun peut développer sa propre version de l'OS.

De même que Apple, la documentation pour Android est de très bonne qualité. On y retrouve les explications sur les API, des cours et des exemples.

Tableau 9 : Statistiques Android

	Tag	**Nombre de questions taguées**
Plateforme	Android	868'916
Langage	Java	1'108'759

7.3.2 iOS

XCode qui est l'environnement de développement natif pour les applications iOS permet aussi de développer des applications pour macOs, watchOS et tvOS. Il est actuellement disponible en version 7.3.1. Le 13 juin 2016, Apple a annoncé le lancement de la version 8.

XCode contient de nombreux outils qui permettent de faciliter le développement des applications. Il contient les différents simulateurs afin de tester son application, un debugger, un Framework de test ainsi que des instruments qui permettent de mesurer les performances.

La première version de l'iOS SDK est sortie le 6 février 2008. Il est actuellement en version 9.3.3, il suit la numérotation des versions d'iOS. Le développement d'une application iOS se fait en Objective-C ou en Swift.

Objective-C est un langage de programmation orienté Object. Il a été développé au début des années 1980. C'est le langage principal utilisé par Apple pour développer macOS et iOS.

Swift est un langage de programmation développé par Apple qui est sorti le 2 juin 2014. Actuellement, la dernière version de Swift est la 2.2. Le langage est très bien documenté, plusieurs guides sont disponibles gratuitement. Le 3 décembre 2015, le code source de Swift devient open source sous licence Apache 2.0 et est disponible sur GitHub. La version 3.0 de Swift est en cours de développement.

La documentation d'Apple concernant le développement iOS est très complète. On y trouve les explications sur les API, des guides, des exemples de codes ainsi que des vidéos.

Tableau 10 : Statistiques iOS

	Tag	**Nombre de questions taguées**
Plateforme	iOS	448'126
Langages	Objective-C	262'916
	Swift	104'352

7.3.3 Windows Phone

Le développement d'application Windows se fait sur l'IDE Visual Studio dont la dernière version s'appelle Visual Studio 2015 (14.0.23107). La première version de l'IDE fut publiée en 1997.

Windows 10 SDK permet le développement d'application Universal Windows App. La dernière version en date est la 10.0.10586.

Les langages utilisés pour le développement sont le C# et le VB.net. La 1re version du C# est sortie en 2002. Aujourd'hui, la dernière version stable est la 6 qui fut publiée en 2015. Quant au VB.net, il fut lancé en 2001 et est actuellement en version 10 qui n'a plus été mis à jour depuis 2010.

Comme les deux plateformes natives citées ci-dessus, Microsoft propose aux développeurs une très bonne documentation. De plus, un point non négligeable, c'est qu'elle est disponible en français.

Tableau 11 : Statistiques Windows Phone

	Tag	Nombre de questions taguées
Plateformes	Windows Phone	6539
	Windows Phone 7	20’076
	Windows Phone 8	18’546
	Windows Phone 8.1	7’813
	Windows 10 mobile	626
	Windows 10 Universal	1’034
	Win-universal-app	4’375
	UWP	3221
Langages	C#	980’531
	VB.net	110’195

7.3.4 Xamarin

Xamarin Studio 6.0.2.73 et Xamarin pour Visual Studio 4.1.2.18 sont les deux outils fournis par la société pour développer une application.

Les différents SDK sont open source disponibles sous licence MIT sur GitHub. Ils sont régulièrement mis à jour, les dernières versions sont Xamarin.iOS 9.8.2.22, Xamarin.Android 6.1.2.21 et Xamarin.Forms 2.3.1.

La documentation officielle disponible sur le site de Xamarin est très complète et de bonne qualité. Cependant, si l’on ne connait pas les différents produits, il peut y avoir de la confusion entre le développement Cross-Platform, Android, iOS et Forms.

Xamarin est une solution relativement récente et de nombreux bugs sont présents. Ils sont référencés sur la plateforme Bugzilla. Le 27 juillet 2016, 6232 bugs ont le statut d’ouverts.

Tableau 12 : Statistiques Xamarin

	Tag	Nombre de questions taguées
Frameworks	Xamarin	14’708
	Xamarin.iOS	8’059
	Xamarin.Android	5’518
	Xamarin.Forms	3’477
Langage	C#	980’531

7.3.5 Cordova

La dernière version en date de Cordova est la 6.3.1. Il n'y a pas d'IDE officiel pour développer une application. Utilisant les technologies du web, il est très facile de trouver de l'aide en cas de difficulté. Pour déboguer une application, il faut utiliser les navigateurs internet qui intègrent les outils nécessaires. La documentation est disponible sur le site internet de Cordova. De plus, les développeurs peuvent mettre des plugins à la disposition de la communauté.

Ionic étant open source, il est disponible sous licence MIT. Actuellement, la dernière version stable est la 1.3.1. Ionic est basé sur AngularJS qui est en version 1.5.8. Une version 2.0 est disponible pour Ionic qui est actuellement en bêta. Contrairement à la première version, celle-ci est basée sur Angular 2 qui est en Release Candidate. Ionic dispose d'une documentation complète sur son site internet qui permet d'installer l'environnement, de développer une application jusqu'à la publication sur les stores.

Tableau 13 : Statistiques Cordova

	Tag	**Nombre de questions taguées**
Frameworks	Cordova	45'882
	Ionic	5'035
	Ionic-framework	12'079
	Angularjs	187'029
Langage	Javascript	1'176'008

7.3.6 Récapitulatif

Les langages utilisés pour développer sur ces plateformes sont très utilisés. Les outils et les frameworks sont régulièrement mis à jour et les risques que les technologies soient abandonnées sont faibles. Chaque solution dispose d'une bonne documentation. Concernant la communauté des applications Windows, celle-ci est la moins importante à cause de l'intérêt des consommateurs pour ce système. De ce fait, les développeurs n'y trouvent pas leur compte. Xamarin est une solution récente, le nombre de questions posé sur stackoverflow est faible et en cas de soucis il sera certainement plus difficile de trouver une solution. De plus, une quantité importante de bugs existent. L'utilisation des technologies du web pour le développement d'une application web ou hybride ne devrait pas poser de problèmes, car les langages sont très utilisés. De plus, Cordova dispose d'une bonne communauté en cas de soucis.

7.4 Performance

La performance d'une application mobile est l'un des points les plus importants lorsqu'on la met à disposition des clients. En effet, les applications sont généralement la vitrine d'une entreprise, car les clients sont en contact direct avec celles-ci. Il est donc primordial qu'une application soit performante sous peine de ne pas être utilisée.

Les performances d'une application mobile dépendent de plusieurs facteurs tels que la puissance du dispositif, des serveurs, de la vitesse du réseau, de la qualité du code et des plateformes de développement utilisées. C'est ce dernier point que nous allons analyser ci-dessous en comparant les résultats obtenus par Magenic et Linus Oberg.

7.4.1 Temps de lancement

Nous allons voir combien de temps prend une application à s'exécuter. Pour cela, chaque lancement a été mesuré 10 fois.

Tableau 14 : Temps de lancement (en seconde)

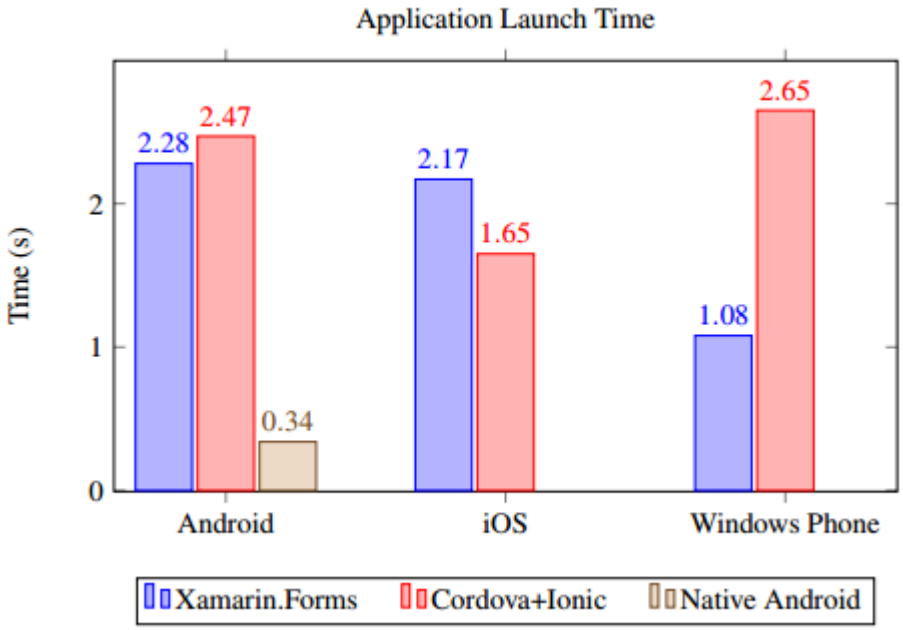
Plateforme	1	2	3	4	5	6	7	8	9	10	Moyenne
Android											
Java	1.47	0.81	0.9	1.24	1.22	1.06	1.05	0.98	1.07	1.05	1.085
Cordova	4.13	3.89	4.03	3.94	4.01	3.88	4.06	3.98	4	3.86	3.978
Xamarin Native	1.73	1.78	1.7	1.59	1.68	1.78	1.59	1.77	1.68	1.74	1.704
Xamarin.Forms	2.92	2.76	2.72	2.75	2.89	2.82	2.57	2.76	2.71	2.74	2.764
iOS											
Objective-C	1.19	1.23	1.16	1.28	1.4	1.35	1.13	1.18	1.16	1.13	1.221
Cordova	1.95	1.77	1.43	1.69	1.73	1.87	1.76	1.59	1.75	1.61	1.715
Xamarin Native	1.31	1.29	1.26	1.34	1.32	1.2	1.29	1.24	1.33	1.22	1.28
Xamarin.Forms	1.92	1.76	1.77	1.9	1.76	1.8	1.86	1.74	1.85	1.77	1.813

(Magenic, 2014)

Comme on peut le voir sur les tests réalisés par Magenic, la technologie native (Java / Objective-C) est la plus rapide. L'approche native de Xamarin, obtient aussi de très bons résultats. Par contre, Cordova et Xamarin.Forms sont les plus lentes, particulièrement Cordova sur Android.

Regardons les résultats obtenus par Linus Oberg s'ils sont similaires à ceux de Magenic.

Figure 12 : Temps de lancement (en seconde)



(Oberg, 2015)

Au vu des résultats, l'application native est la plus performante sur Android. Les autres solutions natives n'ont pas été testées sur les autres systèmes. Comme on a pu le voir précédemment, Cordova est plus performant que Xamarin.Forms sur iOS. En revanche, la différence est beaucoup plus faible qu'au-dessus pour le lancement de l'application Android entre Xamarin.Forms et Cordova même si l'avantage reste à Xamarin.

7.4.2 Calcul des nombres premiers

L'objectif de ce test est d'effectuer une opération intensive pour le processeur. Pour cela, un crible d'Ératosthène a été implémenté, qui permet de trouver tous les nombres premiers inférieurs à un entier. Dans ce cas, l'entier a été fixé à 5 millions. Comme précédemment, le test a été répété 10 fois.

Tableau 15 : Calcul des nombres premiers (en seconde)

Plateforme	1	2	3	4	5	6	7	8	9	10	Moyenne
Android											
Java	4.31	4.31	4.2	4.33	4.39	4.37	4.32	4.45	4.34	4.4	4.342
Cordova	91.7	95	94.3	94.4	94.7	94.1	94.1	91.8	93.6	97.8	94.1
Xamarin Native	4.27	4.25	4.15	4.32	4.51	4.41	4.22	4.12	4.14	4.19	4.258
Xamarin.Forms	4.21	4.17	4.31	4.3	4.2	4.34	4.29	4.36	4.22	4.19	4.259
iOS											
Objective-C	5.04	5.49	5.38	4.86	4.8	5.02	5.03	4.83	4.84	4.85	5.014
Cordova	66.7	67.4	67.2	67.3	67.2	67.4	67.1	67.1	67.6	67.6	67.3
Xamarin Native	4.41	4.42	4.35	4.34	4.49	4.37	4.17	4.27	4.39	4.28	4.349
Xamarin.Forms	4.51	4.33	4.31	4.31	4.33	4.4	4.41	4.4	4.33	4.46	4.379

(Magenic, 2014)

La première chose qu'on remarque est le temps nécessaire à l'application Cordova pour effectuer le calcul, elle n'est donc pas adaptée pour des tâches qui sollicitent fortement le processeur.

On peut voir que les résultats obtenus par Xamarin sont très bons et même légèrement mieux que ceux obtenus par l'application native.

7.4.3 Utilisation de la RAM et du CPU

Trois prototypes ont été réalisés pour Android par Linus Oberg afin d'évaluer l'impact sur la RAM et sur le CPU pour chaque plateforme.

7.4.3.1 RAM

La mesure de la RAM a été exécutée à l'aide de la commande « adb shell dumpsys meminfo <pid> ». Voici un exemple de sortie de son application développée avec Xamarin.

Figure 13 : Utilisation de la RAM

```
> adb shell dumpsys meminfo 30480

Applications Memory Usage (kB):
Uptime: 733602308 Realtime: 1459386764
** MEMINFO in pid 30480 [TekniskForvaltningXamarinForms.Droid] **
```

	Pss Total	Private Dirty	Private Clean	Heap Size	Heap Alloc	Heap Free
Native Heap	15913	15620	0	33280	29955	3324
Dalvik Heap	24966	24912	0	48346	32093	16253
Dalvik Other	360	360	0			
Stack	272	272	0			
Gfx dev	27800	27620	0			
Other dev	4	0	4			
.so mmap	2609	156	2072			
.apk mmap	6537	0	6380			
.ttf mmap	63	0	36			
.dex mmap	1312	4	1308			
.oat mmap	1335	0	364			
.art mmap	1140	776	12			
Other mmap	44	8	0			
EGL mtrack	39040	39040	0			
Unknown	8456	8456	0			
TOTAL	129851	117224	10176	81626	62048	19577

(Oberg, 2015)

D’après Google, la colonne PSS (Proportional Set Size) est une bonne mesure pour connaitre la quantité de RAM utilisée par un processus.

Tableau 16 : Utilisation de la RAM

	PSS Total (kB)
Java	66’567
Xamarin.Forms	129’851
Cordova	166’422

(Oberg, 2015)

Comme on pouvait s’y attendre, l’application native est celle qui consomme le moins de RAM, ensuite Xamarin et pour finir Cordova.

7.4.3.2 CPU

Comme pour la RAM une commande adb est utilisée. « adb shell top –m 10 » permet de voir l’utilisation du CPU par processus.

Figure 14 : Utilisation maximale du CPU

```
> adb shell top -m 10
User 3%, System 2%, IOW 0%, IRQ 0%
User 49 + Nice 13 + Sys 46 + Idle 1696 + IOW 11 + IRQ 3 + SIRQ 7 = 1825

  PID PR CPU% S  #THR     VSS     RSS PCY UID      Name
 3361  2   8% S    41 2241280K 121068K  fg u0_a108  com.ionicframework
 4885  1   0% S   109 2471776K 108696K  fg system   system_server
  369  0   0% S    14 148660K   8812K  fg system   /system/bin/surface
 7072  3   0% S    41 1647772K  48876K  fg u0_a13   com.google.gapps
 3461  5   0% R     1   6000K   1532K  fg shell    top
25593  0   0% S   112 1758096K  77832K  bg u0_a134  com.android.app
    8  3   0% S     1      0K      0K  fg root     rcu_preempt
  402  2   0% S     1      0K      0K  fg root     irq/215-fc38800
  273  1   0% S     1      0K      0K  fg root     mmcqd/0
 7582  0   0% S    91 2464536K  61832K  fg u0_a13   com.android.gms
```

(Oberg, 2015)

Les résultats obtenus sont les suivants :

Tableau 17 : Utilisation maximale du CPU

	Utilisation maximale du CPU
Java	8%
Xamarin.Forms	11%
Cordova	24%

(Oberg, 2015)

Comme on peut le voir, l'utilisation du processeur par l'application native et Xamarin est assez proche. Contrairement à Cordova qui sollicite le CPU deux à trois plus.

7.4.4 Récapitulatif

À travers ces résultats, nous avons pu constater sans surprise que les applications natives sont celles qui offrent les meilleures performances. Généralement, Xamarin est moins performant que le natif mais offre de meilleures performances par rapport à Cordova. De plus, Cordova n'est pas adapté à des applications qui exécutent de gros calculs.

7.5 Temps de développement

Le temps de développement est un des éléments les plus importants lorsqu'on désire sortir une nouvelle application. De plus, plus le temps est élevé plus celui-ci a un impact sur le prix de revient de l'application.

Différents prototypes ont été développés durant ce travail (voir la partie Prototype). La 1ère application a été développée avec Cordova + Ionic. Cependant, une application web étant déjà existante, une très grande partie du code source a pu être repris. Il m'a fallu environ 32 heures afin d'implémenter les spécificités voulues. En revanche, si une partie du code n'avait pas pu être repris dans ce cas, je pense qu'il aurait fallu environ 48 heures.

Quant à lui, le développement avec Xamarin a pris environ 64 heures. Tout comme Ionic, je ne connaissais pas ces différents Framework.

Une application native pour Android aurait pu être développée, cependant par manque de temps cela n'a pas été possible. Par contre, ayant quelques connaissances sur le développement Android, je pense qu'il aurait fallu environ 56 heures pour pouvoir réaliser un prototype.

Tableau 18 : Temps de développement par plateforme

Cordova + Ionic	6 j/h (48h)
Xamarin Forms	8 j/h (64h)
Android	7 j/h (56h)

Ces temps ont été réalisés et estimés en se basant sur mes compétences. Il est évident qu'une personne ayant de très bonnes connaissances dans une plateforme mettra beaucoup moins de temps.

Si on prend le cas d'une personne n'ayant jamais développé alors la solution la plus rapide sera certainement Cordova + Ionic. Par la suite, la version native. Cependant, il ne faut pas oublier que ce type de développement ne fonctionnera que sur une plateforme spécifique contrairement aux deux autres. Il faudrait donc multiplier le temps par deux voire trois pour obtenir le même résultat. Le développement avec Xamarin risque de prendre plus de temps que la solution Cordova. En cas de difficulté, on peut passer plus de temps à la résolution des problèmes du fait que la communauté soit plus faible.

7.6 Coûts

Les coûts de développement d'une application peuvent rapidement devenir importants. Les différents outils de développement pour le natif sont gratuitement fournis par les constructeurs (Xcode, Android Studio et Visual Studio Community). Cependant, concernant Visual Studio Community, celui-ci n'est pas utilisable si une entreprise possède plus de 250 PC ou génère plus de 1 million de dollars de recettes annuelles.

Depuis, le rachat de Xamarin par Microsoft, Xamarin est devenu gratuit. Pour finir, Cordova et Ionic sont également gratuits.

Les principaux coûts de développement sont liés au salaire des développeurs qui eux sont liés aux différents temps de développement. Afin de faciliter le calcul, il faut estimer qu'une personne travaillant sur un développement coûte 900 CHF par jour. En se basant sur les temps qui se trouvent au-dessus nous obtenons les résultats suivants :

Tableau 19 : Coût de revient des différentes plateformes

Cordova + Ionic	6 j/h x 900.-	5'400.-
Xamarin Forms	8 j/h x 900.-	7'200.-
Une application native	7 j/h x 900.-	6'300
Deux applications natives	2 x 7 j/h x 900.-	12'600.-
Trois applications natives	3 x 7 j/h x 900.-	18'900.-

7.6.1 Coûts de déploiements

Peu importe les outils utilisés pour le développement, une fois que l'application est prête, il faut pouvoir la distribuer.

La licence de développeur iOS coute 99 USD par année. En revanche, si une entreprise désire distribuer des applications au sein de son entreprise alors il faut acquérir une licence entreprise qui coûte 299 USD par année.

La distribution des applications Android coute 25 USD à payer une seule fois.

Pour qu'un développeur puisse publier sur le Windows Store la licence coûte 17 CHF à payer une seule fois. En revanche, pour une entreprise la licence coûte 90 CHF.

7.7 Interface graphique

Un client installant une nouvelle application sur son appareil mobile s'attend à ce que son fonctionnement soit semblable aux autres applications installées. Chaque système d'exploitation mobile possède ses propres règles de conception pour l'interface, la navigation et le fonctionnement.

Une mauvaise expérience utilisateur peut être le résultat d'une interface négligée. Cet élément est très important pour l'utilisateur final, il aura un impact sur l'utilisation de l'application. S'il est négatif, il n'hésitera pas à la désinstaller.

Il est donc normal qu'une application native offre la meilleure expérience possible, car celle-ci a été développée avec les composants de la plateforme.

Comme il est possible de voir dans la partie prototype de Xamarin.Forms, l'interface utilisateur s'adapte à la plateforme d'exécution. Cependant, il peut y avoir quelques légères différences par rapport à la native.

Ionic propose une multitude de composants afin de faciliter la conception d'interface graphique. Il est possible d'élaborer une interface de très bonne qualité. Cependant, elle reste différente de la native et ne s'adapte pas à la plateforme d'exécution.

8. Prototype

En collaboration avec Genève Aéroport, une application a été développée pour chaque plateforme à évaluer.

8.1 Scénario

Genève Aéroport est en constante évolution. Avec plus de 15 millions de passagers en 2015, plus de 18'000 personnes possèdent une carte d'identité aéroportuaire qui leur permet de travailler sur la plateforme.

L'objectif est d'avoir une application mobile qui permettra aux agents de sécurité de faire des contrôles d'identité. Actuellement, une application web existe. Certains agents sont équipés d'une tablette Panasonic FZ-X1 qui leur permet d'accéder à l'application et de lire des codes-barres. Pour pouvoir effectuer un contrôle, ils doivent saisir le numéro de badge dans l'application ou bien ils peuvent scanner le code-barre. Cependant, ils arrivent que le numéro de badge ou le code-barre soit difficile d'accès.

L'objectif de l'application à développer est d'utiliser la technologie sans contact pour récupérer le numéro de badge plus facilement, afin d'effectuer le contrôle d'identité.

8.1.1 RFID

Le RFID est une technologie qui permet la communication à distance entre un lecteur et un tag à l'aide d'ondes radio.

Un système RFID a trois composants :

- Une antenne
- Un émetteur / récepteur (contenu dans un lecteur)
- Un transpondeur (tag)

L'antenne qui est à l'intérieur du lecteur RFID va envoyer des ondes qui vont être reçues par le tag. Les tags sont généralement de types passifs. C'est-à-dire qu'ils vont recevoir de l'énergie par le lecteur qui va permettre d'alimenter le circuit intégré de la puce. Par la suite, le tag émet des ondes qui vont être reçues par le lecteur.

Pour pouvoir communiquer, les lecteurs et les tags doivent communiquer sur la même fréquence. Voici les différentes fréquences utilisées :

- Basses fréquences (LF) : 125 et 134.2 kHz
- Hautes fréquences (HF) : 13.56 MHz
- Ultra hautes fréquences (UHF) : 868 à 915 MHz
- Super hautes fréquences (SHF) 2.45 et 5.8 GHz

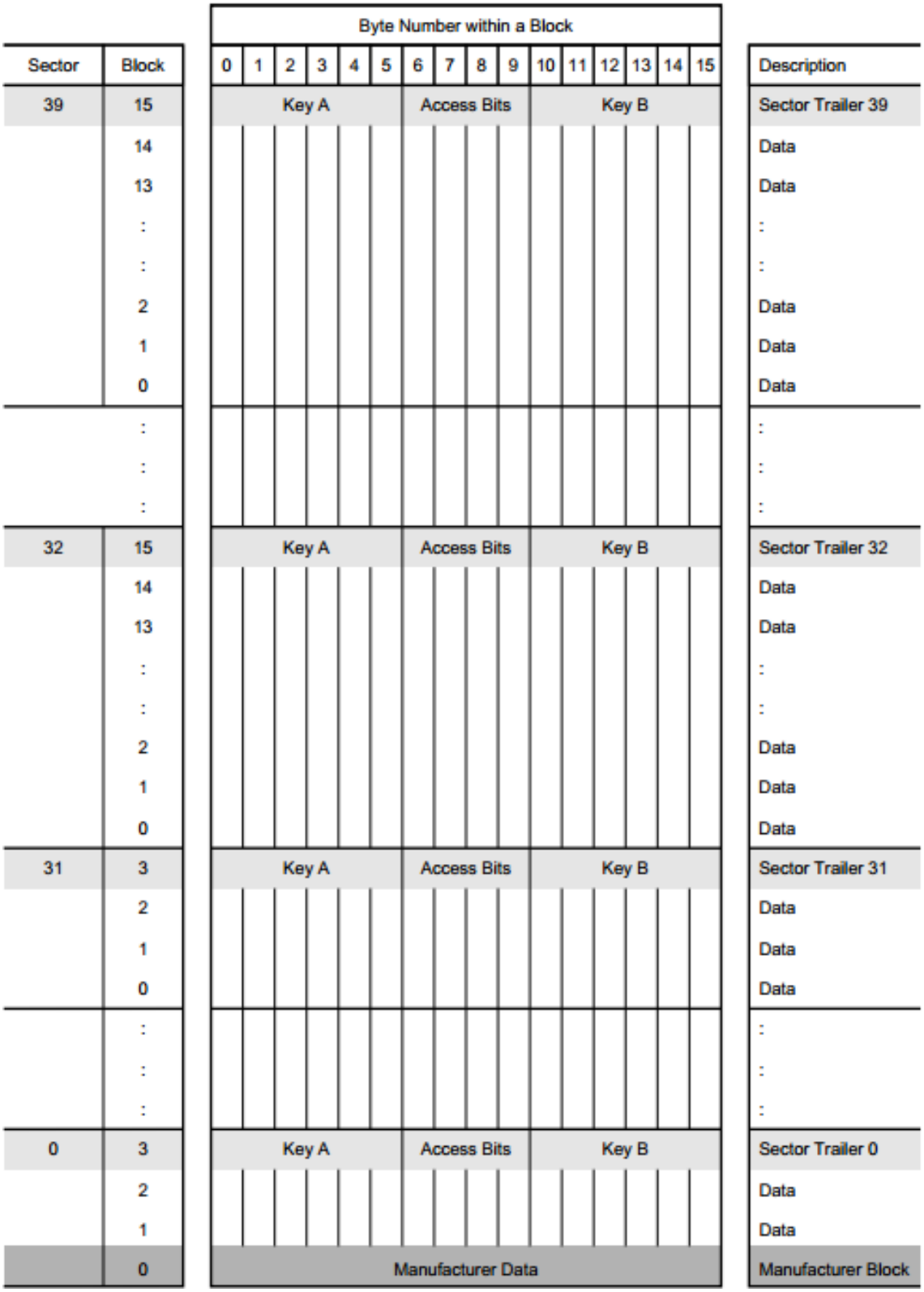
8.1.2 MIFARE

Les badges de Genève Aéroport intègrent deux technologies. Une des deux est la technologie MIFARE qui appartient à NXP Semiconductors qui est basée sur le standard ISO 14443 Type A fonctionnant à 13.56 MHz.

Les badges utilisés dans le cadre de Genève Aéroport sont des MIFARE Classic 4K. Les cartes Classic utilisent le protocole Crypto-1 pour l'authentification et le chiffrement. Cependant, ce type de carte n'est plus considéré comme sécurisé, car le protocole a été cassé en 2008.

Regardons de plus près comment fonctionne un badge Classic 4K. La mémoire de ce type de badge est de 4KB organisée en 40 secteurs. 32 secteurs de 4 blocs et 8 secteurs de 16 blocs.

Figure 15 : Structure de la mémoire d'un badge MIFARE 4K

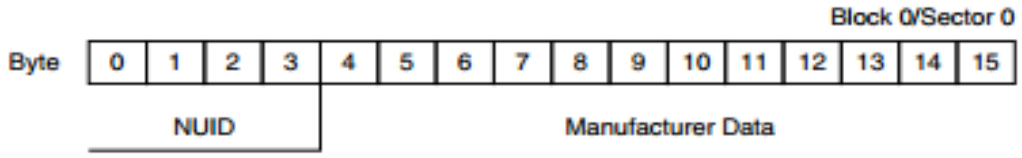


Sector	Block	Byte Number within a Block 0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	Description
39	15	Key A						Access Bits				Key B						Sector Trailer 39
	14																	Data
	13																	Data
	:																	:
	:																	:
	2																	Data
	1																	Data
	0																	Data
	:																	:
	:																	:
	:																	:
32	15	Key A						Access Bits				Key B						Sector Trailer 32
	14																	Data
	13																	Data
	:																	:
	:																	:
	2																	Data
	1																	Data
	0																	Data
31	3	Key A						Access Bits				Key B						Sector Trailer 31
	2																	Data
	1																	Data
	0																	Data
	:																	:
	:																	:
	:																	:
0	3	Key A						Access Bits				Key B						Sector Trailer 0
	2																	Data
	1																	Data
	0	Manufacturer Data																Manufacturer Block

(Fuzzysecurity, 2016)

Le premier bloc (bloc 0) du premier secteur (sector 0) est le Manufacturer Block. Il contient un identifiant et les informations du fabricant.

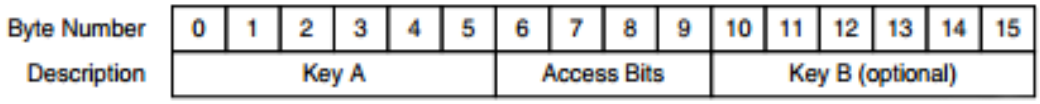
Figure 16 : Manufacturer Block



(Fuzzysecurity, 2016)

Le secteur d'annonce (sector trailer) qui est le dernier bloc de chaque secteur contient la clé A et la clé B qui permettent de décoder les données du secteur. Quant à lui, le contrôle d'accès (Access Bits) définit les droits d'accès (lecture, écriture…) des blocs du secteur.

Figure 17 : Sector Trailer



(Fuzzysecurity, 2016)

8.1.3 Problématique

Pour lire les badges, nous allons utiliser la technologie NFC des appareils mobiles. Le NFC respecte la norme ISO 14443 et fonctionne sur la fréquence 13.56 MHz. Cependant, tous les appareils équipés d'une puce NFC ne sont pas capables de lire les tags MIFARE.

Pour pouvoir lire le contenu d'une carte MIFARE, il faut obligatoirement un appareil équipé d'un contrôleur NFC de la marque NXP qui est propriétaire de la technologie MIFARE.

Certains appareils mobiles d'Apple sont équipés d'une puce NFC. Cependant, celle-ci ne peut être utilisée actuellement que pour le paiement mobile avec Apple Pay.

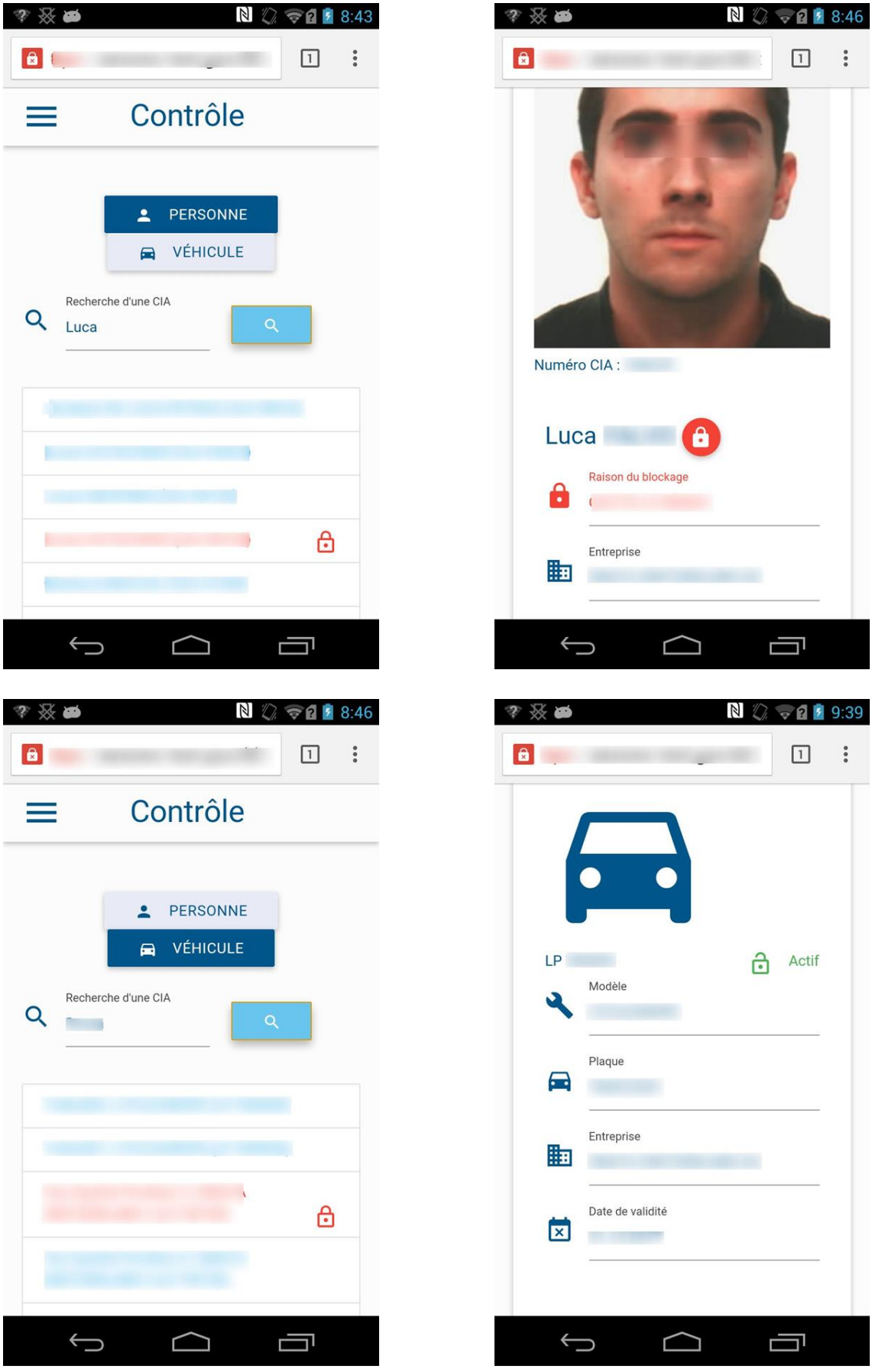
Seul Android propose des API qui permettent d'accéder au contenu d'un tag MIFARE.

8.2 Application existante

Une application web a déjà été développée en AngularJS qui permet de faire des contrôles d'identité, cependant l'accès à la puce NFC n'est pas possible. Pour pouvoir accéder à l'application, il faut obligatoirement être connecté au réseau de l'entreprise.

Une zone de recherche est disponible. L'utilisateur peut alors saisir le numéro de badge ou bien le nom d'une personne. Si la recherche renvoie plusieurs résultats, alors tous les résultats sont affichés dans une liste. En cas de clic sur un des résultats, le détail de la personne s'affiche. En revanche, si la recherche ne renvoie qu'un résultat, le détail est affiché directement. En plus de rechercher une personne, il est possible de rechercher un véhicule. Le fonctionnement reste le même.

Figure 18 : Application existante



8.3 Prototype Cordova / Ionic

La première application multiplateforme a été développée avec Cordova + Ionic afin de permettre l'utilisation du NFC pour lire les badges. Comme les applications Cordova utilisent les technologies du web, je n'ai pas eu la nécessité de redévelopper complètement l'application. En effet, la quasi-totalité du code JavaScript a pu être réutilisée. En revanche, les vues ont été modifiées en utilisant les composants CSS d' Ionic.

La partie la plus complexe fut d'implémenter l'interaction entre le dispositif et le badge. Pour cela, un plugin qui permet l'utilisation du NFC a été développé par Chariot Solution sous le nom de « PhoneGap NFC Plugin » disponible sur Github. Cependant, celui-ci n'est pas capable de lire le contenu des tags MIFARE. Pour remédier à ce problème, il a fallu apporter quelques modifications au plugin.

Le premier fichier à modifier est le fichier Javascript « phonegap-nfc.js ». C'est l'interface qui va permettre par la suite d'appeler du code natif.

Le code suivant a été ajouté :

Figure 19 : Interface JavaScript

```
readMifareSecBloc: function (sector,bloc,win, fail) {

    cordova.exec(win, fail, "NfcPlugin", "readMf_SB", [sector,bloc]);
}
```

La méthode « readMifareSecBloc » est la méthode qui va pouvoir être appelée depuis le code JavaScript. À l'intérieur de celle-ci, cordova.exec va communiquer avec la plateforme native. Voici comment chaque paramètre fonctionne :

- Win : C'est une fonction qui peut s'exécuter en cas de succès de cordova.exec
- Fail : La fonction s'exécute en cas d'échec.
- NfcPlugin : Le nom de la classe à appeler du côté natif.
- readMf_SB : Le nom de l'action à exécuter du côté natif.
- [sector, bloc] : Ce sont les arguments qu'il faut passer dans l'environnement natif.

Une fois l'interface JavaScript modifiée, il faut implémenter le code Java qui va permettre de lire le contenu du tag MIFARE.

Le code suivant a été ajouté :

Figure 20 : Implémentation native

```
private void readMifare_SB(JSONArray data, CallbackContext callbackContext) throws JSONException {
   int sector=Integer.parseInt(data.getString(0));
   int bloque=Integer.parseInt(data.getString(1));
   Tag tagFromIntent = savedIntent.getParcelableExtra(NfcAdapter.EXTRA_TAG);
   byte[] data_mf;
   data_nfc="";

   MifareClassic mfc = MifareClassic.get(tagFromIntent);
   try {
     mfc.connect();
     //boolean auth2 = mfc.authenticateSectorWithKeyA(sector, MifareClassic.KEY_DEFAULT);
     boolean authA = mfc.authenticateSectorWithKeyA(sector, custom_keyA);
     boolean authB = mfc.authenticateSectorWithKeyB(sector, custom_keyB);

     if(authA && authB)
     {
       int bIndex = 0;
       bIndex = mfc.sectorToBlock(sector);
       data_mf = mfc.readBlock(bIndex+bloque);
       data_nfc = new String(data_mf, "UTF-8");
     }
   } catch (IOException e) {
           //Log.e(TAG, "No Conecto", e);
   } finally {
     if (mfc != null) {
       try {
         mfc.close();
       }
       catch (IOException e) {
           //     Log.e(TAG, "Error closing tag...", e);
       }
     }
   }
   callbackContext.success(data_nfc);
}
```

Tout d'abord, il va falloir récupérer les paramètres passés depuis le JavaScript. Ils vont permettre de définir dans quel bloc et dans quel secteur sont les données. Ensuite, pour pouvoir les récupérer, il faut authentifier le secteur avec les différentes clés. En cas de succès, il sera possible de récupérer le contenu et de le renvoyer au JavaScript.

Une fois que le plugin a été modifié, il faut pouvoir l'utiliser dans l'application.

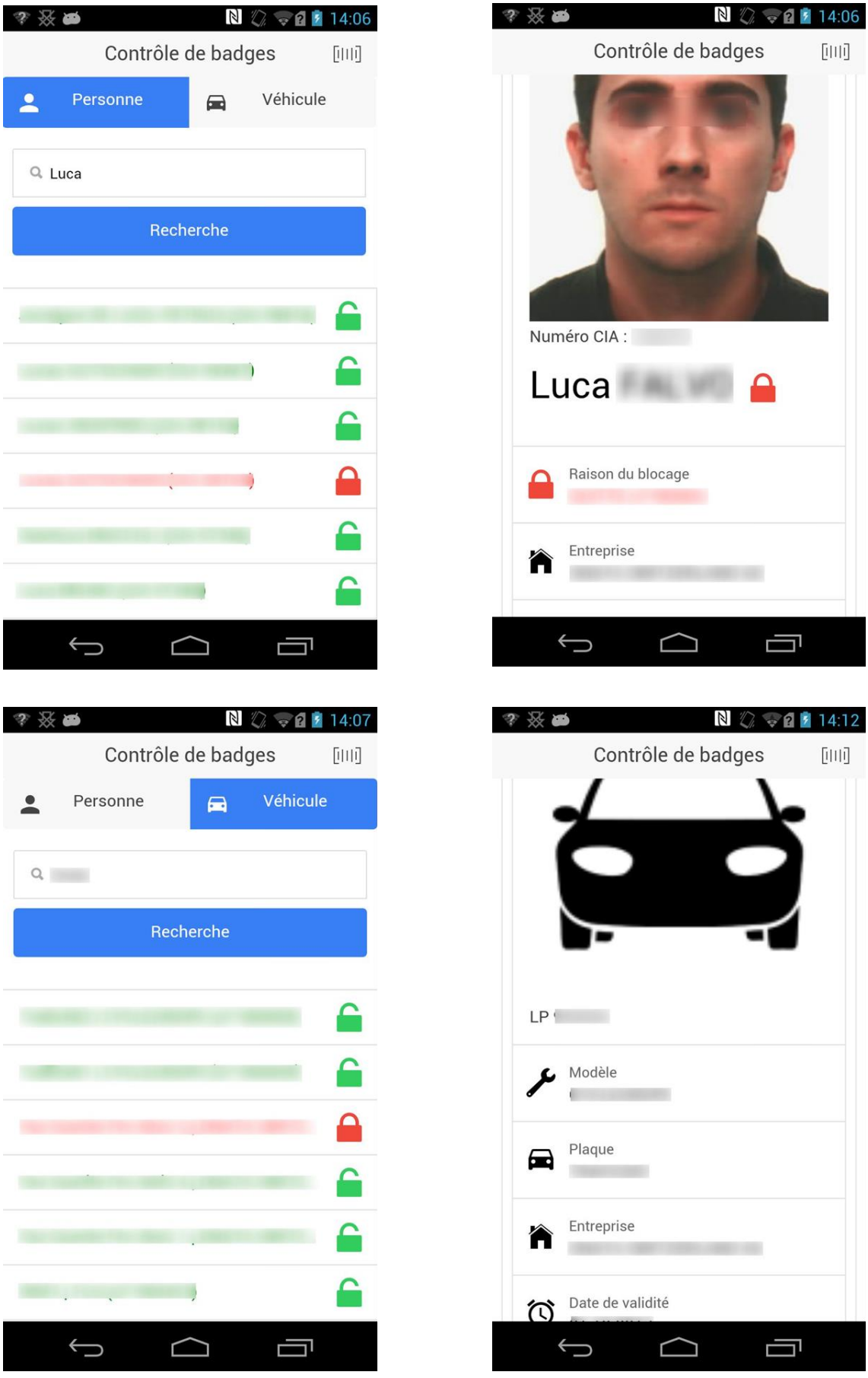
Figure 21 : Appel du plugin

```
.factory('nfcService', function ($rootScope, $ionicPlatform) {

    $ionicPlatform.ready(function () {
        nfc.addTagDiscoveredListener(function (nfcEvent) {
            $rootScope.recherche = "";
            $rootScope.$apply();

            nfc.readMifareSecBloc(5, 1, function (nfcdata) {
                if (nfcdata != ""){
                    $rootScope.recherche = nfcdata;
                    $rootScope.$apply();
                }else{
                    alert("Le numéro de badge n'est pas lisible.")
                }

            }, function () {
                console.log("mifare readed");
            },
                function (reason) {
                    alert("Error reading mifare " + reason);
                });
        }, function () {
            console.log("Listening for NDEF Tags.");
        }, function (reason) {
            alert("Error adding NFC Listener " + reason);
        });

    });

    return {

    };
});
```

Quand le dispositif est prêt, l'application va pouvoir déclencher un évènement quand un tag est découvert. Dans ce cas, on essayera de récupérer le numéro de badge MIFARE et de lancer la rechercher.

Le résultat final de l'application est le suivant :

Figure 22 : Prototype Ionic

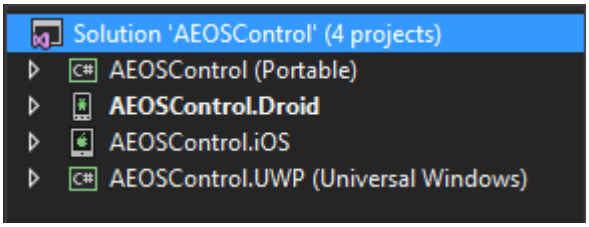


8.4 Prototype Xamarin.Forms

Après avoir développé l'application avec Cordova + Ionic, un prototype a été créé avec Xamarin. L'approche Forms a été utilisée, de cette manière l'interface a été codée une seule fois. Ayant déjà eu une expérience avec le XAML, mon choix s'est donc porté sur celui-ci pour la réalisation des vues. Concernant le partage de code, mon choix s'est porté sur un projet PCL. En effet, de cette manière le code est beaucoup plus facile à maintenir. Néanmoins, l'implémentation de fonctionnalité native est plus complexe.

La solution Visual Studio est composée des projets suivants :

Figure 23 : Arborescence de la solution Xamarin

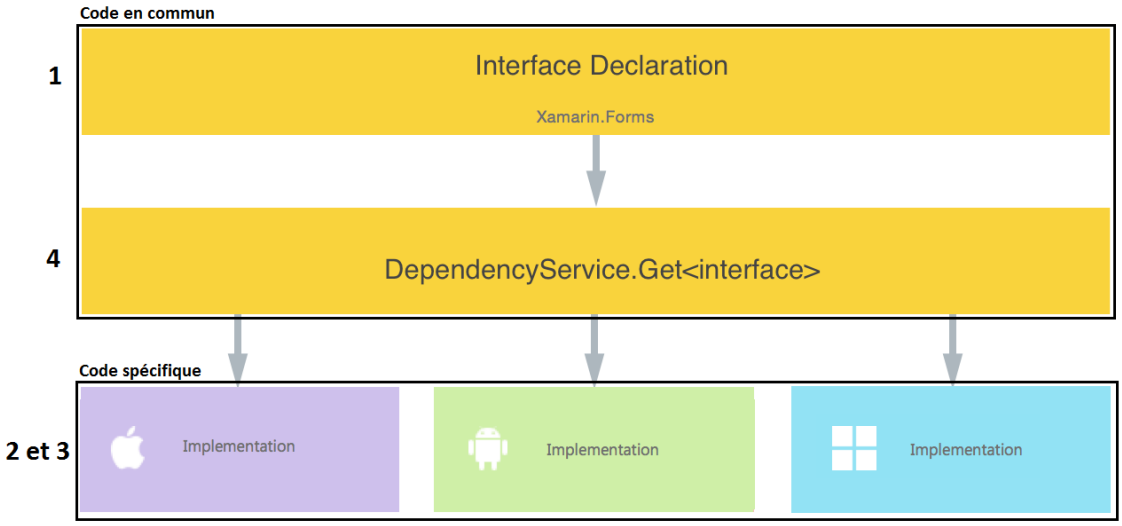


Je ne vais pas détailler le fonctionnement de l'application ainsi que le code pour lire le badge MIFARE, car ceci est semblable au prototype réalisé avec Ionic. Cependant, il est important de comprendre comment fonctionne le service de dépendance.

Le service de dépendance permet à l'application d'appeler du code spécifique à la plateforme depuis le code partagé.

Le diagramme suivant montre son implémentation :

Figure 24 : Implémentation du service de dépendance

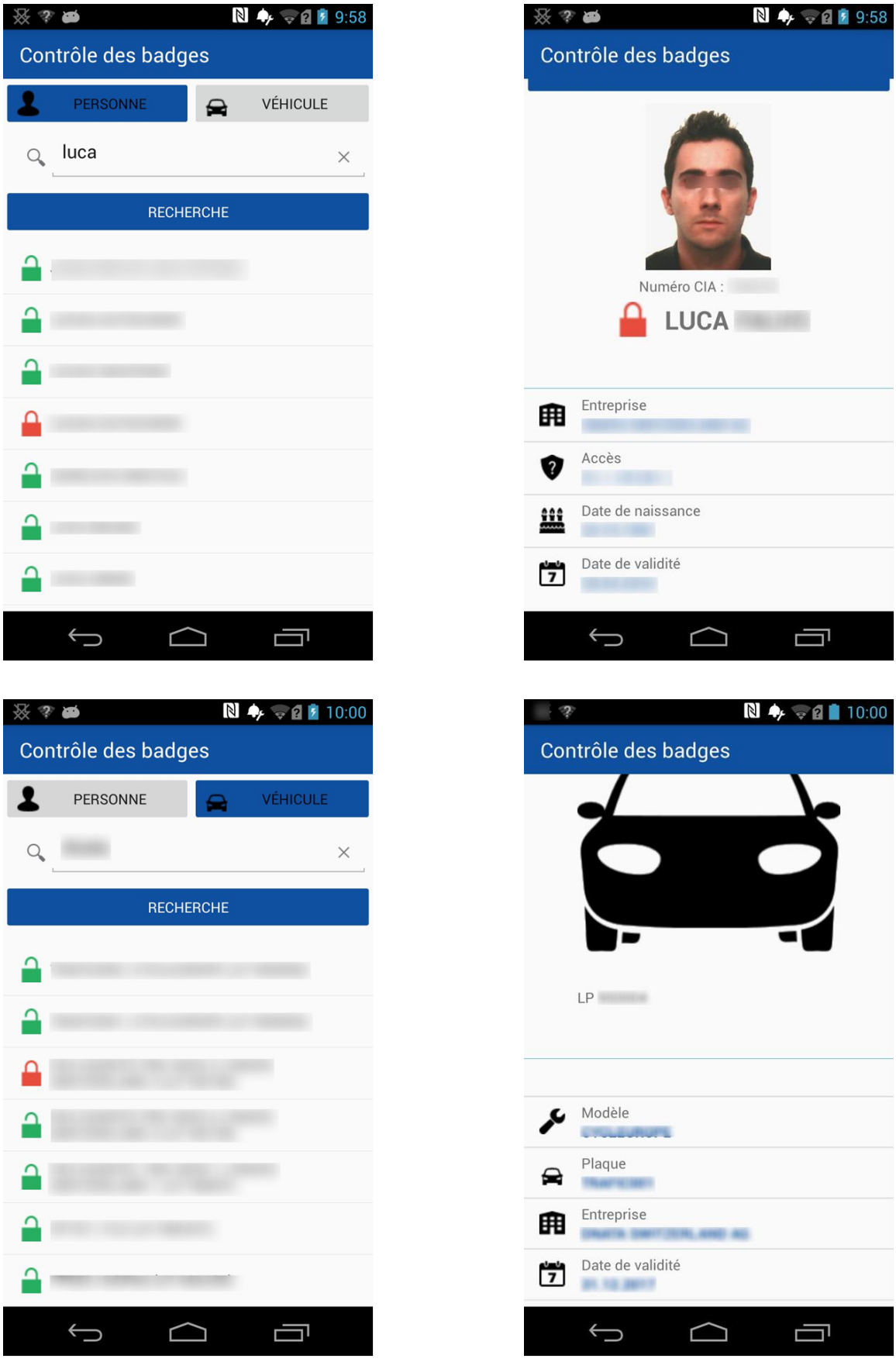


(Xamarin, 2016)

1. **Interface :** c'est la fonctionnalité qu'on veut implémenter, elle est définie dans le code partagé.
2. **Implémentation du code par la plateforme :** Il faut réaliser l'interface pour chaque plateforme où l'on veut implémenter le code spécifique.
3. **Inscription :** Pour chaque plateforme implémentée, il faut inscrire au démarrage de l'application le service de dépendance.
4. **Appel du service de dépendance :** Pour finir, la dépendance de service est appelée dans le code en commun.

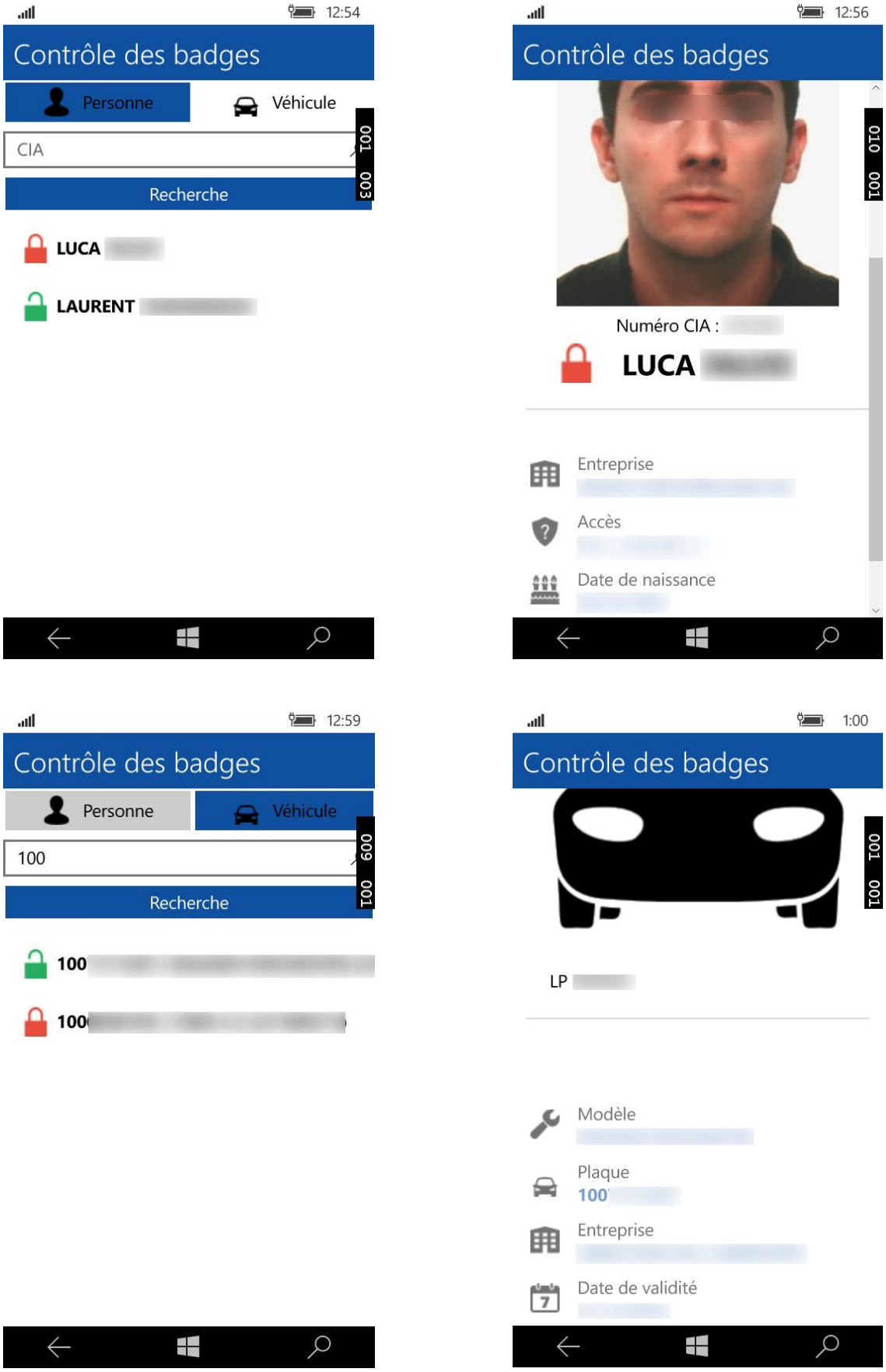
8.4.1 Prototype Android

Figure 25 : Prototype Android



8.4.2 Prototype Windows Phone

Figure 26 : Prototype Windows Phone



8.4.3 Prototype iOS

Figure 27 : Prototype iOS

9. Choix de la technologie

Dans le cadre de l'application développée ci-dessus, il faut définir quels sont les critères les plus importants. Pour cela, il faut les comparer entre eux. Par exemple, comparer A avec B et choisir lequel est le plus important.

Peu importe l'application que l'on veut développer, la démarche reste identique. Cependant, les résultats obtenus risquent d'être différents, car l'importance des critères dépendent de l'application.

Tableau 20 : Matrice préférentielle

	Accès aux fonctionnalités (A)	**Stockage local (B)**	**Communauté, documentation et version (C)**	**Performance (D)**	**Temps développement (E)**	**Coûts (F)**	**Interface graphique (G)**
Accès aux fonctionnalités (A)							
Stockage local (B)	A						
Communauté, documentation et version (C)	A	C					
Performance (D)	A	D	C				
Temps développement (E)	E	E	E	E			
Coûts (F)	F	F	F	F	F		
Interface graphique (G)	A	G	C	D	E	F	

Les poids suivants ont été définis :

- Accès aux fonctionnalités (A) : 4
- Stockage local (B) : 0
- Communauté, documentation et version (C) 3
- Performance (D) 2
- Temps de développement (E) 5
- Coûts (F) 6
- Interface graphique (G) 1

L'étape suivante consiste à définir quels sont les critères obligatoires que doit supporter la plateforme de développement.

Pour chaque plateforme, il faut donner une note à chaque critère en se basant sur l'analyse faite dans les points précédents.

La valeur des points attribués est la suivante :

- 1. Mauvais
- 2. Acceptable
- 3. Bien
- 4. Excellent

Tableau 21 : Matrice de décision

		Natif		Xamarin		Cordova		Web	
Critère obligatoire									
Accès à la puce NFC		Oui		Oui		Oui		Non	
Critère évalué	**poids**	**points**	**valeur pondérée**	**points**	**valeur pondérée**	**points**	**valeur pondérée**	**points**	**valeur pondérée**
Accès aux fonctionnalités (A)	4	4	16	3	12	3	12	1	
Stockage local (B)	0	4	0	4	0	4	0	2	
Communauté, documentation et version (C)	3	4	12	2	6	4	12	4	
Performance (D)	2	4	8	4	8	3	6	2	
Temps de développement (E)	5	1	5	3	15	4	20	4	
Coûts (F)	6	1	6	3	18	4	24	4	
Interface graphique (G)	1	4	4	3	3	2	2	2	
Total		22	**51**	22	**62**	24	**76**	19	**0**

La solution Cordova est celle qui obtient le plus grand nombre de points. Ensuite viennent Xamarin et la native. Le Web n'a pas été évalué, car il ne respecte pas le critère obligatoire.

La plateforme native est celle qui obtient le score le plus bas. Cependant, celle-ci a été évaluée dans une approche multiplateforme c'est pourquoi les critères « temps de développement » et « coûts » ont une note de 1.

Même si cette application peut fonctionner sur les différents systèmes, seule la version Android aura la possibilité de lire un tag MIFARE. Si le développement natif ne se fait que sur Android alors les points pour les critères, « temps de développement » et « coûts » ont une note de 3.

En refaisant le calcul, la solution native obtiendrait une note de 73. Dans ce cas, la différence avec Cordova est mince.

10. Conclusion

Au fil de ce document, nous avons pu observer les différentes possibilités qui existent pour le développement d'une application mobile. Toutes ces solutions ont leurs avantages et inconvénients. On ne peut pas dire qu'une approche est meilleure en tous points qu'une autre.

L'application native est celle qui offre la meilleure expérience utilisateur. Cependant, le développement pour les différentes plateformes nécessite plus de temps et un budget plus important.

Le développement d'application avec Xamarin permet de développer une application multiplateforme offrant de très bonnes performances. De plus, le design de l'application s'adapte à la plateforme d'exécution. Néanmoins, certaines faiblesses existent. Xamarin étant relativement récent, de nombreux bugs sont présents. De plus la communauté est faible ce qui peut être problématique lorsqu'on recherche une solution à un problème.

Cordova offre une solution pour développer une application multiplateforme à moindre coût. En effet, celle-ci est basée sur les technologies du web qui sont très répandues. Il est également possible de réaliser un portage d'une application web existante sans trop de difficultés. Malgré cela, ce genre d'application peut souffrir de problèmes de performance même si cela tend à s'améliorer. De plus, l'application a un design standard qui ne s'adapte pas à la plateforme cible.

Pour finir, le développement d'une application web a comme avantage de pouvoir s'exécuter sur n'importe quel dispositif ayant un navigateur internet. Pourtant, la plus grosse faiblesse est son accès aux fonctionnalités du dispositif.

Pour conclure, il est important de définir les spécifications de la future application à développer. De cette manière, en se basant sur les différents critères évalués ci-dessus, il sera plus facile de choisir quel type de solution est le plus adapté pour un développement mobile.

Bibliographie

APPLE INC, 2014. About the iOS Technologies [en ligne]. [Consulté le 24.08.2016]. Disponible à l'adresse : https://developer.apple.com/library/ios/documentation/Miscellaneous/Conceptual/iPhoneOSTechOverview/Introduction/Introduction.html

XAMARIN, 2014. Announcing Xamarin 3 [en ligne]. [Consulté le 24.08.2016]. Disponible à l'adresse : https://blog.xamarin.com/announcing-xamarin-3/

XAMARIN, 2016. Mobile App Development & App Creation Software [en ligne]. [Consulté le 24.08.2016]. Disponible à l'adresse : https://www.xamarin.com/

APACHE CORDOVA, 2016. Apache Cordova [en ligne]. [Consulté le 24.08.2016]. Disponible à l'adresse : https://cordova.apache.org/

MOBILE HTML, 2016. Mobile HTML5 compatibility on iPhone, Android, Windows Phone, BlackBerry, Firefox OS and other mobile devices [en ligne]. [Consulté le 24.08.2016]. Disponible à l'adresse : http://mobilehtml5.org/

REALM, 2016. A better mobile database means better apps [en ligne]. [Consulté le 02.09.2016]. Disponible à l'adresse : https://realm.io/

HTML5ROCKS, 2014. Working with quota on mobile browsers [en ligne]. [Consulté le 11.07.2016]. Disponible à l'adresse : http://www.html5rocks.com/en/tutorials/offline/quota-research/

STACKOVERFLOW, 2016. Tags [en ligne]. [Consulté le 25.07.2016]. Disponible à l'adresse : http://stackoverflow.com/tags

ANDROID, 2016. Android Developers [en ligne]. [Consulté le 01.09.2016]. Disponible à l'adresse : https://developer.android.com/index.html

BUGZILLA XAMARIN, 2016. Bugzilla [en ligne]. [Consulté le 27.07.2016]. Disponible à l'adresse : https://bugzilla.xamarin.com/

MICROSOFT DEVELOPER, 2014. Windows Phone 8.1 for Developers–Converging the App Models [en ligne]. [Consulté le 24.07.2016]. Disponible à l'adresse : https://blogs.msdn.microsoft.com/thunbrynt/2014/03/31/windows-phone-8-1-for-developersconverging-the-app-models/

TUTORIALSPOINT, 2014. SQLite Overview [en ligne]. [Consulté le 21.07.2016]. Disponible à l'adresse : http://www.tutorialspoint.com/sqlite/sqlite_overview.htm

MAGENIC, 2014. Mobile Development Platform Performance [en ligne]. [Consulté le 12.07.2016]. Disponible à l'adresse : http://magenic.com/Blog/Post/4/Mobile-Development-Platform-Performance

OBERG, Linus, 2015. Evaluation of Cross-Platform Mobile Development Tools [en ligne]. Umeå : Umeå University. Travail de master. [Consulté le 05.07.2016]. Disponible à l'adresse : http://umu.diva-portal.org/smash/get/diva2:898053/FULLTEXT01.pdf

DUPONT, Audrey, 2015. Analyse du développement mobile multiplateforme avec Xamarin. Travail de bachelor. [Consulté le 05.07.2016]. Disponible à l'adresse : https://doc.rero.ch/record/258928/files/Dupont_A_Analyse_du_d_veloppement_mobile_multiplateforme_avec_Xamarin.pdf

FUZZYSECURITY, 2016. Appendix A - Mifare Classic 101 [en ligne]. [Consulté le 05.07.2016]. Disponible à l'adresse : http://www.fuzzysecurity.com/tutorials/rfid/2.html

CNET, 2016. Apple by the numbers: 2 million apps, 15 million Apple Music subscribers [en ligne]. [Consulté le 11.07.2016]. Disponible à l'adresse :

http://www.cnet.com/news/apple-by-the-numbers-2-million-apps15-million-apple-music-subscribers10-billion-icloud-documents/

APPBRAIN, 2016. Number of Android applications [en ligne]. [Consulté le 07.09.2016]. Disponible à l'adresse : http://www.appbrain.com/stats/number-of-android-apps

GARTNER, 2016. Gartner Says Five of Top 10 Worldwide Mobile Phone Vendors Increased Sales in Second Quarter of 2016 [en ligne]. [Consulté le 12.09.2016]. Disponible à l'adresse : http://www.gartner.com/newsroom/id/3415117

WIKIPEDIA, 2016. Android (operating system) [en ligne]. [Consulté le 18.07.2016]. Disponible à l'adresse : https://en.wikipedia.org/wiki/Android_(operating_system)

WIKIPEDIA, 2016. Apple iOS [en ligne]. [Consulté le 18.07.2016]. Disponible à l'adresse : https://fr.wikipedia.org/wiki/Apple_iOS

WIKIPEDIA, 2016. Android (operating system) [en ligne]. [Consulté le 18.07.2016]. Disponible à l'adresse : https://en.wikipedia.org/wiki/Android_(operating_system)

MICROSFT DEVELOPER, 2014. Windows Phone 8.1 for Developers–Converging the App Models [en ligne]. [Consulté le 20.07.2016]. Disponible à l'adresse : https://blogs.msdn.microsoft.com/thunbrynt/2014/03/31/windows-phone-8-1-for-developersconverging-the-app-models/

WIKIPEDIA, 2016. MIFARE [en ligne]. [Consulté le 01.09.2016]. Disponible à l'adresse : https://en.wikipedia.org/wiki/MIFARE

GITHUB, 2016. realm/realm-java-benchmarks [en ligne]. [Consulté le 19.09.2016]. Disponible à l'adresse : https://github.com/realm/realm-java-benchmarks

CISCO BLOG, 2010. Cisco and Apple Agreement on IOS Trademark [en ligne]. [Consulté le 11.07.2016]. Disponible à l'adresse : http://blogs.cisco.com/news/cisco_and_apple_agreement_on_ios_trademark

ANGULARJS, 2016. Angularjs [en ligne]. [Consulté le 25.08.2016]. Disponible à l'adresse : https://angularjs.org/