République Algérienne Démocratique et Populaire
Ministère de l'Enseignement Supérieur et de la Recherche Scientifique
Université d'Abbès Laghrour Khenchela
Faculté des Sciences et de la Technologie
Département des Mathématiques et d'Informatique

Mémoire de fin d'études

pour l'obtention du diplôme de Master en Informatique
Spécialité : Génie logiciel et systèmes distribués.

Conception et réalisation d'une application mobile

Présenté par :

Allaoua Houssem Eddine .

Dirigé par :

Dr Toufik Messaoud Maarouk

Promotion : 2019/2020

Remerciement

Au nom de Dieu clement et miséricordieux.

Avant tout je tiens à Remercier Dieu et me prosterner devant lui, en guise de remerciment pour la force et la volonté qui a mit en moi, qui m'ont permit d'arriver au bout de ce travail.

J'adresse mes sincères remerciements de reconnaissances et gratitudes à tous les enseignants, et en particulier mon encadreur Docteur Maarouk Toufik, pour ses précieux conseils, et sa disponibilité, il c'est toujours montrés à l'écoute et très disponible tout au long de la réalisation de ce mémoire, ainsi pour l'inspiration, l'aide et le temps qu'il a bien voulu me consacrer et sans qui ce mémoire n'aurait jamais vu le jour.

Aussi je tiens à remercier ma famille qui m'ont épaulé durant toute cette phase notament mes parents pour leur contribution, leur soutien et leur patience.

Enfin, j'adresse mes plus sincères remerciements à tous mes proches et amis qui ont contribué de loin ou de près à la réalisation de ce travail.

Résumé

Dans ce document, nous examinerons d'abord les deux principales plates-formes d'applications mobiles (android, ios) exprimant les différences et les similitudes entre deux, puis nous parlerons un peu des différents types d'applications mobiles et des avantages et inconvénients de chaque type d'application mobile. Nous présentons Firebase, une solution sans serveur géniale pour écrire du code côté serveur. Nous examinerons l'architecture "MVVM" et le langage de programmation "Kotlin". Enfin nous développons une application de Foxy flexy pour les fournissuers de crédits de téléphonie.
Mots clé : Androif, IOS, Firebase, Kotlin, MVVM.

Abstract

In this document we will first take a look to the two major mobile app platform (android , ios) expriming the differences and similarities between them , then we will talk little bit about the diffrent types of mobile app and which advetages and downsides of each type of mobile application. We will be introduced to Firebase which is an awesome serverless solution to write server side code. We will examine the "MVVM" architecture and the "Kotlin" programming language. Finally we are developing a Foxy flexy application for telephone credit providers.

Mots clé : Androif, IOS, Firebase, Kotlin, MVVM.

Table des matières

Table des figures

Chapitre 1

Introduction générale

Chaque jour, les nouveaux appareils arrivent sur le marché avec des options innovantes grâce à une technologie croissante. L'évolution de la technologie de développement d'applications mobiles avec de nouveaux appareils a rendu nos vies beaucoup plus faciles.

Dans le monde des smartphones, il ne suffit pas d'avoir un site Web en cours d'exécution. Une étude récente, a montré qu'environ 45% [mobileAppIntro] et plus de la recherche Google se fait à l'aide des smartphones.

Le nombre est spectaculaire et il y a une croissance au sein de l'activité mobile. Être disponible sur un appareil compatible Internet est nécessaire pour chaque entreprise qui a donné le coup d'envoi au développement d'applications mobiles.

1.1 Qu'est-ce que le développement mobile ?

Le développement mobile ne consiste pas à créer des applications pour téléphones, bien qu'il en soit une grande partie. En effet, il fait tout développement pour tout type d'appareils mobiles, comme le développement d'applications pour téléphones, tablettes, smartwatches et tout types d'appareils portables exécutant les differents systèmes d'exploitation mobile.

Le développement mobile présente une chance raisonnablement distinctive pour une équipe de développement composée d'une seule personne de créer une application réelle, utilisable et significative de bout en bout pendant une période relativement courte. Cependant, le développement d'applications mobiles représente plus qu'une simple chance pour le développeur solo de créer son propre projet car il s'agit sans doute du développement à plus long terme, car les appareils mobiles prennent de plus en plus de place dans notre vie.

1.2 Principales plates-formes de développement mobile

IOS

C'est la plate-forme qui a introduit le développement mobile dans les temps modernes en changeant entièrement le concept d'appareil mobile et de système logiciel mobile . IOS est bien sûr développé par Apple et fonctionne exclusivement sur les produits Apple.

Apple fournit aux développeurs IOS de nombreux outils et des bibliothèques natives pour développer des applications IOS et sans être obligé d'utiliser les outils de développement d'Apple pour créer vos applications, il vous suffit d'avoir un Mac exécutant OS X pour créer votre propre application.

Android

Android est l'autre acteur dominant dans cet espace, sorti pour la première fois en septembre 2008, pratiquement un an plus tard qu'IOS, mais il a réussi à atteindre une part relativement massive du marché mobile.

Techniquement, Android est le système d'exploitation mobile qui détient la part la plus importante du marché avec environ 80% de part contre 18% d'IOS. Ces chiffres sont un peu trompeurs, car Android peut être un marché fragmenté composé des nombreux appareils différents créés par de divers fabricants, exécutant des versions complètement différentes du système d'exploitation Android.[mobileAppIntro]

1.3 Quelles sont les principales différences entre IOS et Android ?

— Android est soutenu par Google,
— IOS est soutenu par Apple,
— Tout le monde peut créer un appareil Android, et il est conçu pour fonctionner sur une variété de plates-formes matérielles et d'appareils avec des facteurs de forme et des capacités très différents,
— IOS est conçu pour fonctionner uniquement sur un ensemble spécifique d'appareils Apple,
— Android est basé sur le noyau Linux et Google publie le code source d'Android en open source,

— Comme Apple, Google fournit des outils natifs pour le développement Android.

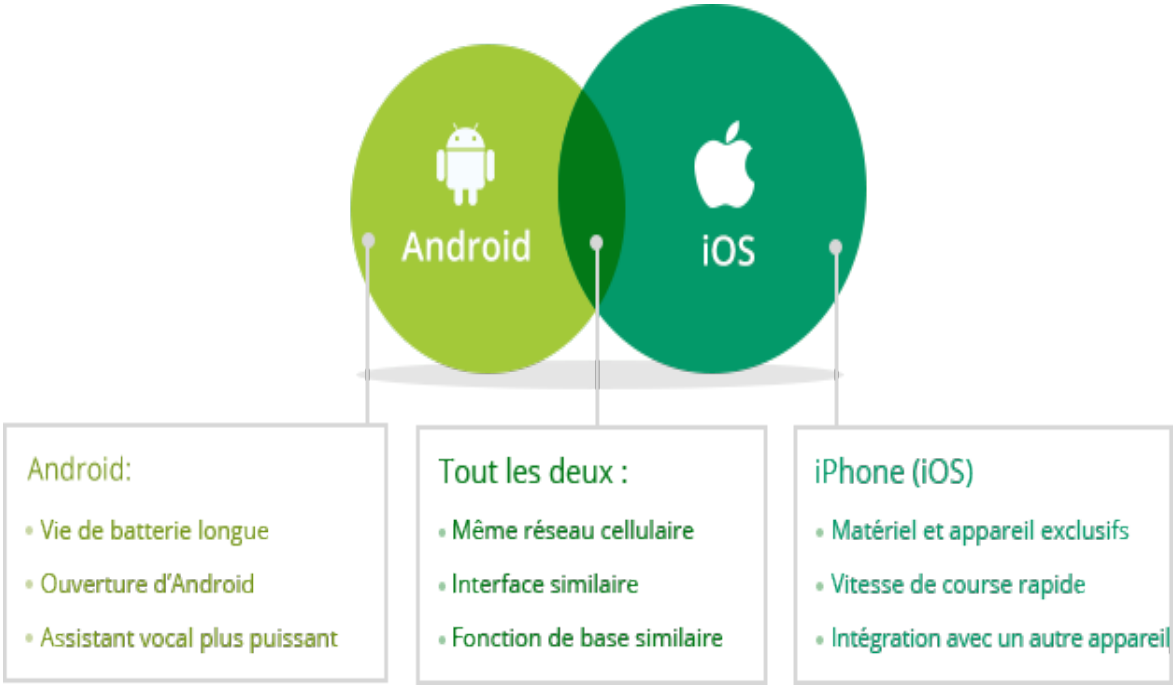


FIGURE 1.1 – Android vs IOS

1.4 Type de applications mobile

Dans cette section, nous procédons à une comparaison des différents types d'applications mobiles qui existent du point de vue d'une entreprise qui cherche à créer ou à embaucher quelqu'un pour dévelloper une application mobile et c'est un énorme choix qu'il pourrait faire ou défaire le succès d'une entreprise, dans le monde des applications mobiles, nous allons donc comparer les applications Web, natives et les applications hybrides et nous parlerons des différences que nous examinerons également avantages et inconvénients de chacune :
Il y a essentiellement trois types principaux d'applications mobiles, il y a une zone grise, mais pour la plupart, nous avons des applications Web, natives et hybrides. Considérons ces trois derniéres, les comparer entre elles et déterminer les avantages et les inconvénients de chacune .

1.4.1 Web Apps

Les applications Web ou les sites Web mobiles sont donc créés à l'aide du technologies Web standard tel que HTML, CSS et JavaScript, elles s'exécutent dans un

navigateur Web standard comme Chrome, Firefox, Safari ; elles sont construites et hébergées comme n'importe quelle application Web ou site Web sur Internet, la seule vraie différence est qu'elles sont creées pour bien paraître et fonctionner correctement sur les appareils mobiles, cela signifie généralement qu'elles sont réactives et conçues avec une approche mobile d'abord, ce qui signifie qu'elles commençent par se concentrer sur la vue mobile, comme elles sont adaptables aux ordinateurs de bureau.

Avantages

Avantages de la création d'une application Web :

— d'abord, elles sont construites en utilisant uniquement des normes Web standard tel que HTML, CSS et JavaScript,
— un autre avantage, vous pouvez utiliser n'importe quel type de technologie Web,
— les applications Web sont également de loin l'option la moins chère, comme si vous n'êtes pas un développeur Web, embaucher un développeur Web revient beaucoup moins cher que d'embaucher, par exemple, un programmeur Swift,
— le prix entre également dans cet avantage qui consiste à créer une application pour toutes les plates-formes, donc IOS, Android pour les personnes qui utilisent BlackBerry et Windows Phones, votre application fonctionnera sur n'importe quel appareil tant qu'elle peut fonctionner avec un navigateur c'est évidemment beaucoup moins cher que les applications natives et hybrides,
— non seulement cela, mais votre application est accessible aux ordinateurs de bureau ainsi qu'aux ordinateurs portables à condition qu'ils aient une connexion Internet.

Inconvénients

— Besoin d'un navigateur pour fonctionner, cela signifie que l'utilisateur doit réellement s'effectuer dans un navigateur sur son téléphone et taper l'URL de l'application et c'est une expérience utilisateur vraiment très médiocre,
— Les applications Web sont généralement beaucoup plus lentes que les applications natives, car les applications natives sont créées spécifiquement pour cet appareil, elles sont optimisées pour fonctionner aussi bien que possible sur cet appareil,
— les applications Web sont moins interactives et les boutons moins intuitifs n'ont pas le même type d'effet d'interaction qu'une application native,
— Vous devez exécuter l'application dans le navigateur ce qui la cause de ne pas avoir d'icône sur votre bureau mobile, comme vous le feriez si vous la téléchargiez

depuis l'App Store ou le Play Store qui, donne pas de très bonne expérience utilisateur,
— Les applications Web ne peuvent pas non plus être soumises aux magasins d'applications, vous êtes dans le besoin de transformer votre application Web en une sorte d'application hybride,
— les applications Web ne puissent pas intéragir avec les utilitaires de l'appareil, il n'y a pas d'API, par exemple pour la caméra , pour une géolocalisation ou l'une de ces fonctionnalités.

1.4.2 Native Apps

Une application native est le type le plus courant d'application mobile lorsque vous recherchez sur l'App Store et que vous téléchargez une application, elle est probablement native, mais les applications hybrides gagnent en popularité et peuvent également être ajoutées à l'App Store, mais une application native est conçue pour une plate-forme spécifique, une application Android est codée en Java ou kotlin et elle utilise le SDK pour cette plate-forme même chose pour une Application IOS, elle est écrite en Swift ou parfois en Objective-c et elle est écrite pour la plate-forme IOS.

Avantages

— Les applications natives sont très rapides et c'est parce que, elles sont conçues pour cette plate-forme spécifique,
— Les applications natives sont également très facilement distribuées dans les magasins d'applications, que ce soit le Apple Store, Google Play ou Windows Store,
— Les applications natives peuvent être approuvées très facilement et sont également beaucoup plus intéractives et les choses intuitives fonctionnent beaucoup plus facilement en ce qui en concerne les entrées et les sorties de l'utilisateur,
— Elles peuvent également intéragir facilement avec prèsque toutes les fonctionnalités du téléphone, que ce soit la géolocalisation, la caméra, le stockage ou la boussole, il existe une API étendue pour travailler avec à peu près n'importe quelle partie du matériel et les applications natives peuvent vraiment entrer dans cela et faire des choses vraiment impressionnantes.

Inconvénients

— Une application Android ne fonctionnera que sur Android, et une application Swift ne fonctionnera cependant que sur iPhone,

— Une application pour Android et iPhone ou même Windows ou BlackBerry risque la faillité parce que il faut traiter tout cela comme des projets séparés qui sont vraiment coûteux,
— Apprendre Swift ou Java est beaucoup plus difficile que d'apprendre JavaScript, HTML et CSS ou même certains stock-Web back-end, Swift et Java sont simplement des embauches beaucoup plus complexes les développeurs pour ces langages sont très chers par rapport à votre développeur Web standard en plus d'être une plate-forme unique plus difficile à créer et revient beaucoup plus cher,
— ils sont également plus difficiles à maintenir.

1.4.3 Hybrid Apps

Une application hybride est essentiellement une combinaison d'une application Web et d'une application native. Elle utilise du HTML, CSS et JavaScript, mais elle s'exécute également à l'intérieur d'un type de conteneur ou de vue Web, généralement via un cadre, de sorte qu'en surface elle peut être perçue comme un application native.

Avantages

— les application hybride, sont basés sur une technologie Web tel que HTML, CSS et JavaScript, ce qui les rend beaucoup plus faciles à créer,
— elles sont également beaucoup moins cheres qu'une application native si vous embaucher des développeurs ; un développeur Web est bien plus meilleur marché qu'un développeur Swift,
— vous n'avez besoin que d'une application unique pour toutes les plates-formes si vous utilisez une technologie comme Cordova PhoneGap qui est comme un contenaire pour les applications hybrides, vous pouvez en effet créer une base de code qui est une même application pour IOS et Android,
— donc vous en avez essentiellement une base de code à maintenir en opposition à plusieurs applications bien sûr, car les applications hybrides s'exécutent dans une vue Web, aucun navigateur n'est nécessaire, comme avec une application Web,
— elles peuvent être publiées sur tous les magasins d'applications, il est vraiment impossible pour l'utilisateur de savoir qu'il s'agit d'une application hybride et non d'une application native,
— elles peuvent être installées exactement de la même manière que les applications natives ont également l'accès à l'appareil interne API et elles peuvent accéder à des fonctionnalités tel que le stockage, la géolocalisation, la caméra,

— donc au lieu d'avoir une application Android et une application IOS, une application Windows et maintenir les trois séparément, c'est une seule base de code et cela accélère vraiment les choses.

Inconvénients

— elles sont généralement plus lentes que les applications natives, les applications natives sont spécifiquement conçues pour certaines plates-formes dans certains langages,
— les applications hybrides ont une sorte d'intermédiaire à traverser, elles peuvent se sentir un peu plus maladroites, mais cela s'améliore à mesure que ces frameworks progressent,
— les applications hybrides bien que beaucoup moins chères que les applications natives, elles sont plus chères que les applications Web standard car vous avez ce framework contenaire à utiliser et à maintenir,
— elles peuvent aussi être un peu moins interactives avec les pressions sur les boutons, les cadres hybrides sont des technologies relativement nouvelles et elles s'améliorent chaque jour.

Maintenant, il y a une exception ou même un quatrième type d'application mobile et c'est une plate-forme construite avec des plates-formes qui utilisent des technologies Web mais convertissent l'application en composants natifs plutôt que de simplement la produire en une vue Web, c'est ce que les frameworks comme React-Native et Xamarin, contrairement à Ionic et PhoneGap où ce ne sont essentiellement que des applications Web enveloppées dans une vue Web mais, une application native React-Native est très proche d'une vraie application native en termes de performances, donc ces technologies s'améliorent à fur et à mesure.

Dans ce travail nous développons une application mobile pour automatiser la fuidité du crédit mobile du fournisseur au consommateur

L'objectif de ce travail est d'aider les fourniseurs en mettant à leur disposition une nouvelle maniére pour assurer la distribution du crédit mobile aux consommateurs, suivant une certaine hiéarchie qui leurs permertra la répartition du crédit mobile sur une grande échelle couvrant le plus grand nombre possible de consommateurs

Ce mémoire est structuré en trois chapitres :

Dans le premier chapitre, nous découvrirons Firebase qui est une solution sans serveur géniale pour écrire du code côté serveur, nous prendrons un peu plus de détails sur les services Firebase tels que (Firebase Authontification, Firestore database et Firebase Functions) donnant un exemple pratique de implémentation de chaque service.

Dans le chapitre deux, nous examinerons l'architecture "MVVM" et le langage de programmation "Kotlin" et comment ils peuvent rendre la vie d'un développeur Android beaucoup plus facile, quand ils sont utilisés correctement, nous allons expliquer certains modèles de conception tels que "Lifecycle-Aware Components", "ViewModel", "LiveData" et le "Navigation Component" et voir la puissance de chaque modèle de conception.

Enfin, dans le chapitre trois, nous ferons un tour pour voir la logique derrière l'application Foxy Flexy, conçue pour aider les fournisseurs de crédit de téléphonie mobile, rendant la distribution de leur produit aussi simple s'effectuant par quelques clics de bouton.

Chapitre 2

Firebase

Introduction

Dans ce chapitre, on présente Firebase, qu'est ce que c'est Firebase, où il peut être utilisé, quelles sont ses approches...

FIGURE 2.1 – Firebase LOGO

Lorsque nous concevons des applications mobiles comme IOS ou Android ou peut-être une application web, la base de données pose un gros problème non pas parce qu'elle est difficile à concevoir, mais le probleme se pose parfois par la consommation abusive du trafic de la bande passante.

Entre une base de données et le front-end de l'application peuvent surgir des problèmes du fait que la page hôte peut provoquer d'autres difficultés.
En plus de la gestion de votre propre système d'authentification est également un peu délicat, soit parce que tout le monde a besoin d'une authentification Facebook, Twitter, peut-être Google login, ou même un simple système de authentification n'est pas facile à concevoir à partir de zéro.

Firebase vous donne une solution complète sur la façon dont les choses peuvent être resolues, c'est une très bonne solution back-end, complète, avec Firebase on peut faire toutes sortes d'authentification, les plus courantes sont par nom d'utilisateur, e-mail et mot de passe, mais on peut bien sûr utiliser un identifiant avec Twitter, Facebook ou vous connecter avec des comptes Google et github et toutes les autres applications populaires. En dehors de l'authentification, Firebase resoud également un très grand problème de base de données, il donne une base de données en temps réel, le débogage et le déploiement d'un niveau accru dans l'application, il peut faire beaucoup d'autres choses aussi bien que la messagerie cloud, les notifications d'applications, les programmes publicitaires et il y a des choses que qu'on peut voir dans la page de fonctionnalités de Firebase.

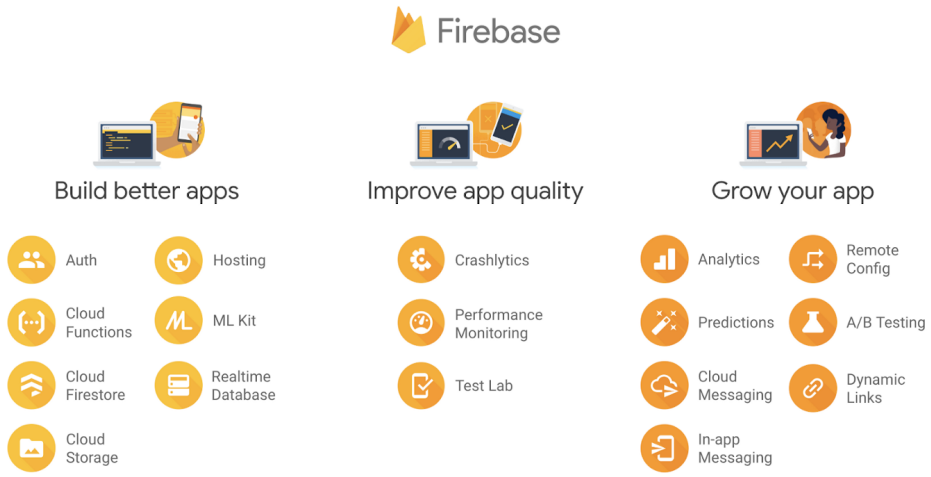


FIGURE 2.2 – Firebase Products

Parse est un concurrent pour Firebase (Open Source Backend Platform d'une société a été rachetée par Facebook en 2013 et fermée en janvier 2017) qui était en pleine croissance, et maintenant pour utiliser Parse, on devez y installer nos propres serveurs AWS (Amazon Web Services) et hébergez Parse nous-même, et on devez le maintenir.

La bonne partie de Firebase c'est que la base de données et la gestion de la bande passante n'ont pas besoin d'une attention particuliaire, donc tout vous est donné pour

les applications à petite échelle, Firebase prend un avantage, car pour les applications à petite échelle, il peut les gérer parfaitement.

Firebase peut être utiliser dans le langage Swift, ainsi qu'il dispose d'une API complète qui fonctionne là dedans. On peut aussi l'utiliser dans Android, il a également des directives spécifiques pour Android, si on utilise une application Web basée sur JavaScript, on peut également utiliser l'intégralité de Firebase en JavaScript.

2.1 Qu'est que c'est Firebase

> Firebase is Google's mobile application development platform that helps you build, improve, and grow your app.

Firebase est un ensemble d'outils pour «créer, améliorer et développer une application mobile», et les outils qu'il vous donne couvrent une grande partie des services que les développeurs devraient normalement construire eux-mêmes, mais ils veulent eviter cette tâche, car ils préfèrent se concentrer sur l'expérience de l'application elle-même. Cela inclut des choses comme l'analyse, authentification, bases de données, configuration, stockage de fichiers, push messagerie...Les services sont hébergés dans le cloud et évoluent avec peu ou pas d'effort de la part du développeur [stevenson2018firebase].

"Hébergé dans le cloud", c'est à dire les produits ont des composants backend entièrement gérés et exploités par Google. Les SDK clients fournis par Firebase interagissent avec ces services backend directement, sans avoir besoin d'établir un middleware entre l'application et le service. Ceci est différent du développement d'applications traditionnelles, qui implique généralement l'écriture de logiciels front-end et back-end. Le code frontal appelle simplement les points de terminaison d'API exposés par le backend, et le code backend fait réellement le travail. Cependant, avec les produits Firebase, le backend traditionnel est contourné, plaçant le travail dans le client. L'accès administratif à chacun de ces produits est fourni par la console Firebase.

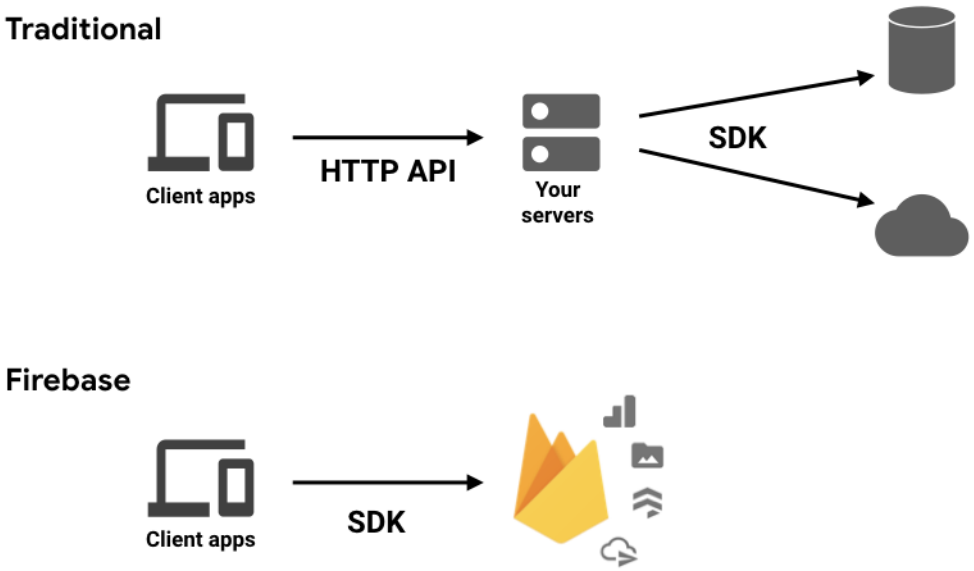


FIGURE 2.3 – Traditional Apps vs Firebase Apps

2.2 À quel type d'applications Firebase est-il bon ?

Il n'y a vraiment aucune limite aux types d'applications qui peuvent être aidées par les produits Firebase. il y a que des limites aux plates-formes sur lesquelles ils peuvent être utilisés. IOS et Android sont les principales cibles pour les SDK Firebase, et il existe une prise en charge croissante pour le Web, Flutter, Unity et C ++.
En plus de ces SDK, il existe une bibliothèque appelée FirebaseUI (Android, IOS, Web) qui fournit un tas d'utilitaires utiles à rendre le développement avec Firebase encore plus facile. Il y a également des projets tels que AngularFire qui encapsulent les SDK Web à utiliser avec Angular. Ceux-ci sont Open source.

2.3 Les Produits Firebase [stevenson2018firebase]

Pour avoir une idée de ce que les produits Firebase font réellement dans une application, on passe en revue les produits individuels de la Figure 1.2, trois principaux groupes de produits : build «construire», improve «améliorer» et grow «développer» (mais cette répartition n'est pas stricte).

2.3.1 Créez votre application

Le groupe de produits **build** «construction» est le suivant :

- **Authentication** : connexion et identification des utilisateurs,
- **Realtime Database** : base de données NoSQL en temps réel, hébergée dans le cloud,
- **Cloud Firestore** : base de données NoSQL en temps réel, hébergée dans le cloud,
- **Cloud Storage** : stockage de fichiers massivement évolutif,
- **Cloud Functions** : backend «sans serveur», piloté par les événements "serverless event driven backend",
- **Firebase Hosting** : Hébergement Web Global,
- **ML Kit** : SDK pour les tâches " Machin Learning communes.

2.3.2 Améliorez votre application - (stabilité et performances)

Le groupe de produits **improve** «améliorer» comprend :

- **Test Lab** : laboratoire de tests des applications évolutives et automatisées sur des appareils hébergés dans le cloud,
- **Crashlytics** : obtenir des informations claires et exploitables sur les crashes de votre application,
- **Performance Monitoring** : obtenir un aperçu des problèmes de performances de votre application.

2.3.3 Développez votre application

Le groupe de produits **Grow** est le suivant :

- **Analytics** : comprendre vos utilisateurs et comment utilisent-ils votre application,

— **Predictions** : appliquer l'apprentissage automatique à " **Analytics** " pour prédire le comportement des utilisateurs,
— **Cloud Messaging** : envoyer des messages et des notifications aux utilisateurs,
— **Remote Config** : personnalisez votre application sans déployer une nouvelle version et surveiller les changements,
— **A/B Testing** : mener des expériences de marketing et d'utilisabilité pour voir ce qui fonctionne le mieux,
— **Dynamic Links** : activer les conversions d'applications natives, le partage d'utilisateurs et les campagnes marketing,
— **App Indexing** : réengager les utilisateurs avec l'intégration de la recherche Google,
— **In-App Messaging** : engager les utilisateurs actifs avec des messages ciblés.

2.4 Firebase Authentification

Firebase facilite l'authentification pour les utilisateurs finaux et les développeurs. Les applications doivent connaître l'identité d'un utilisateur afin de pouvoir fournir une expérience personnalisée et sécurisée leurs données. La prise en charge de Firebase de nombreuses façons différentes pour les utilisateurs de s'authentifier. Firebase Authentification a une fonctionnalité intégrée pour les tiers fournisseurs, tels que Facebook, Twitter, GitHub et Google.
On peut créer notre propre interface ou profiter de leur open source FirebaseUI, qui est entièrement personnalisable et intègre des années d'expérience de Google dans la création d'une UX de l'authentification simple. Une fois qu'un utilisateur s'authentifie, trois choses se produisent :

1. Les informations sur l'utilisateur sont renvoyées à l'appareil via des callBacks. Cela nous permet de personnaliser l'expérience utilisateur de notre application pour cet utilisater,
2. Les informations utilisateur contiennent un identifiant unique qui est garanti distinct pour tous les fournisseurs (Facebook, Twitter, GitHub et Google), ne changeant jamais pour une authentification spécifique utilisateur,
3. Cet identifiant unique est utilisé pour identifier votre utilisateur et quelles parties de votre système back-end auquel ils sont autorisés à accéder.

Firebase gére également les sessions utilisateurs afin que les utilisateurs restent connectés après le redémarrage du navigateur ou de l'application, elle fonctionne sur Android, IOS et le Web.

2.4.1 Authentification FirebaseUI

FirebaseUI fournit une solution d'authentification directe qui gère les flux d'interface utilisateur pour l'authentification des utilisateurs avec leurs adresses e-mail et mots de passe, numéros de téléphone ; avec des fournisseurs d'identité fédérés populaires, notamment Google Sign-In et Facebook Login.
Le composant FirebaseUI Auth met en œuvre les meilleures pratiques pour l'authentification sur les appareils mobiles et les sites Web, Il gère également les cas extrêmes tels que la récupération de compte et la liaison de comptes qui peuvent être sensibles à la sécurité et susceptibles d'être traités correctement.

2.4.2 Firebase Authentification SDK

Firebase supporte plusieurs types d'authentification qu'on resume en ce qui suit :

1. Authentification par e-mail et mot de passe

Authentification des utilisateurs avec leurs adresses e-mail et mots de passe. Firebase Authentication gère également l'envoi d'e-mails de réinitialisation de mot de passe.

2. Intégration du fournisseur d'identité fédéré

Authentification des utilisateurs en intégrant des fournisseurs d'identité fédérés. Le SDK Firebase Authentication fournit des méthodes qui permettent aux utilisateurs de se connecter avec leurs comptes Google, Facebook, Twitter et GitHub.

3. Authentification avec numéro de téléphone

Authentification des utilisateurs en envoyant des SMS(OTP) sur leurs téléphones.

4. Intégration de système d'authentification personnalisée

Connectez le système de connexion existant de l'application au SDK d'authentification Firebase et accédez à Firebase Realtime Database et à d'autres services Firebase.

5. Auth anonyme

Utilisez des fonctionnalités qui nécessitent une authentification sans obliger les utilisateurs à se connecter d'abord en créant des comptes anonymes temporaires. Si l'utilisateur choisit ultérieurement de s'inscrire, on peut mettre à niveau le compte anonyme vers un compte normal, afin que l'utilisateur puisse continuer là où il s'est arrêté.

2.4.3 Comment ça marche ?

Pour connecter un utilisateur à l'application, il faut obtenir d'abord les informations d'authentification de l'utilisateur. Ces informations d'identification peuvent être l'adresse e-mail et le mot de passe de l'utilisateur, ou un jeton Auth d'un fournisseur d'identité fédéré. Ensuite, on transmet ces informations d'identification au SDK d'authentification Firebase. Les services backend vérifieront ensuite ces informations d'identification et renverront une réponse au client.

Après une connexion réussie, on peut accéder aux informations de profil de base de l'utilisateur et contrôler l'accès de l'utilisateur aux données stockées dans d'autres produits Firebase. Egalement utiliser le jeton d'authentification fourni pour vérifier l'identité des utilisateurs dans les services de backend[FirebaseDocs/auth].

2.5 Exemple d'implantation

Dans cet exemple, on utilise l'Authentification avec numéro de téléphone dans une application Android :

Nous devons ajouter Firebase à notre projet Android [FirebaseDocs/android/setup].

1. Ajoutez la dépendance de la bibliothèque Firebase Authentication Android

Au fichier Gradle (niveau de l'application) ajouter :

```
implementation 'com.google.firebase:firebase-auth:19.3.2'
```

19.3.2 est la derniére version au moment de la rédaction de ce document.

2. Activer la connexion par numéro de téléphone dans Firebase console

2.5.1 Envoyer un code de vérification sur le téléphone de l'utilisateur

Pour lancer la connexion par numéro de téléphone, on présente à l'utilisateur une interface qui l'invite à taper son numéro de téléphone. Ensuite, transmettez le numéro de téléphone à la méthode

`PhoneAuthProvider.verifyPhoneNumber( )`

pour demander à Firebase de vérifier le numéro de téléphone de l'utilisateur. Par exemple :

Lorsque `PhoneAuthProvider.verifyPhoneNumber( )` est appelée, on devez également fournir une instance de `OnVerificationStateChangedCallbacks`, qui contient

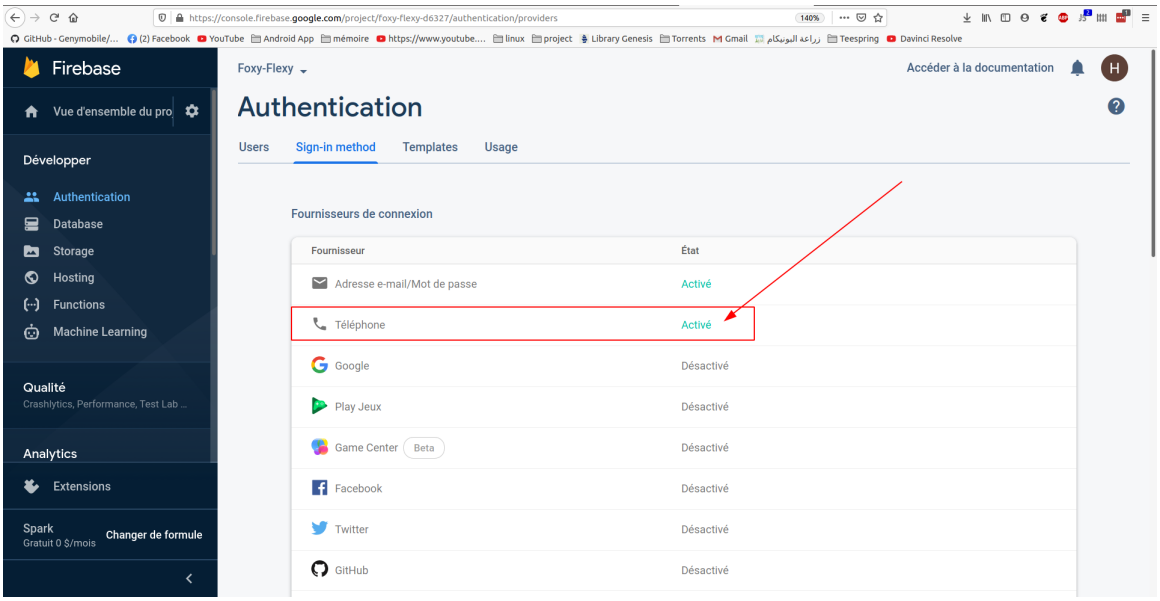


FIGURE 2.4 – Activation de FirebaseAuth par numéro de téléphone

```
PhoneAuthProvider.getInstance().verifyPhoneNumber(
                phoneNumber, // Phone number to verify
                60, // Timeout duration
                TimeUnit.SECONDS, // Unit of timeout
                this, // Activity (for callback binding)
                 callbacks) // OnVerificationStateChangedCallbacks
```

des implémentations des fonctions de rappel qui gèrent les résultats de la demande. Par exemple :

```
callbacks = object : PhoneAuthProvider.OnVerificationStateChangedCallbacks() {

    override fun onVerificationCompleted(
                            credential: PhoneAuthCredential) {

        signInWithPhoneAuthCredential(credential)
    }

    override fun onVerificationFailed(e: FirebaseException) {

        if (e is FirebaseAuthInvalidCredentialsException) {
            //handel Invalid request

        } else if (e is FirebaseTooManyRequestsException) {
            //handel case of SMS quota for the project has
            //been exceeded

        }

        // Show a message and update the UI
    }


    override fun onCodeSent(
        verificationId: String,
        token: PhoneAuthProvider.ForceResendingToken
    ) {

        // Save verification ID and resending
        //        token so we can use them later

        storedVerificationId = verificationId
        resendToken = token
    }
}
```

Rappels de vérification

Dans la plupart des applications, on implémente `onVerificationCompleted( )` `onVerificationFailed( )` et `onCodeSent( )`. On peut également implémenter `onCodeAutoRetr` `)`, en fonction des exigences de l'application.

`onVerificationCompleted (PhoneAuthCredential)`

Cette méthode est appelée dans deux situations :

1. Vérification instantanée : dans certains cas, le numéro de téléphone peut être vérifié instantanément sans qu'il soit nécessaire d'envoyer ou de saisir un code de vérification,
2. Récupération automatique : sur certains appareils, les services Google Play peuvent détecter automatiquement le SMS de vérification entrant et effectuer une vérification sans action de l'utilisateur.

Dans les deux cas, le numéro de téléphone de l'utilisateur a été vérifié avec succès et on peut utiliser l'objet `PhoneAuthCredential` qui passe au callback pour le connecter.

`onVerificationFailed (FirebaseException)`

Cette méthode est appelée quand une réponse à une demande de vérification non valide, telle qu'une demande qui spécifie un numéro de téléphone ou un code de vérification non valide.

`onCodeSent (String verificationId, PhoneAuthProvider.ForceResendingToken)`

Optionnel. Cette méthode est appelée après l'envoi du code de vérification par SMS au numéro de téléphone fourni.

Lorsque cette méthode est appelée, la plupart des applications affichent une interface utilisateur qui invite l'utilisateur à taper le code de vérification à partir du message SMS. Ensuite, une fois que l'utilisateur a tapé le code de vérification, on peut utiliser le code de vérification et l'ID de vérification qui ont été transmis à la méthode pour créer un objet `PhoneAuthCredential`, avec lequel vous pouvez connecter l'utilisateur.

Créer un objet PhoneAuthCredential

Une fois que l'utilisateur a entré le code de vérification que Firebase a envoyé au téléphone de l'utilisateur, il faut créer un objet `PhoneAuthCredential`, en utilisant le code et l'ID de vérification qui ont été transmis au rappel `onCodeSent` ou `onCodeAutoRetrievalTimeOut`.

Connectez-vous l'utilisateur

Après avoir obtenu un objet `PhoneAuthCredential`, que ce soit dans le rappel `onVerificationCompleted( )` ou en appelant`PhoneAuthProvider.getCredential( )`, terminez le `PhoneAuthProvider.getCredential( )` connexion en transmettant l'objet `PhoneAuthCredential` à `FirebaseAuth.signInWithCredential( )`

2.6 Firebase Firestore

Firestore est une base de données hébergée dans le cloud, elle facilite la synchronisation des données en temps réel entre plusieurs applications clientes connectées sur des différentes plates-formes, c'est donc un moyen simple et agréable de mettre en œuvre des fonctionnalités en ligne qui fonctionnent sur différents appareils sans être obliger de gérer nos propres serveurs[FirebaseDocs/irestore/data-model].

La base de données Firestore est une base de données dite NoSQL, ce qui signifie qu'au lieu de sauvegarder nos données dans des tables avec des lignes et des colonnes comme dans SQLite. Par exemple, nous stockons les données dans Firestore dans des éléments appelés **documents**, qui contiennent chacun un ensemble des **paires (clé/-valeur)** , ces **paires (clé/valeur)** sont appelées *champs* (figure 1.5) qui sont assez similaires au format **JSON**, et c'est fondamentalement du **JSON**, en plus il supporte des types de données supplémentaires.

FIGURE 2.5 – Firestore document

2.6.1 Document Firestore

Qu'est-ce qu'un document Firestore ?

— Un document peut stocker des types de données simples comme "Strings", "Integers" et "Booleans",

— Un document peut stocker aussi par exemple des points géographiques exprimés en longitude et latitude, et des valeurs binaires brûtes,

— Un document peut également stocker des

tableaux et des objets imbriqués qui sont ici appelés "maps", prenons par exemple cet objet imbriqué " **name** " de la figure 1.5 qui se compose d'un **"first"** et d'un **"last"**,
- Les documents sont organisés en collections qui servent essentiellement de dossiers,
- Chaque document dans une collection a un nom unique, nous pouvons soit le définir nous-mêmes, soit laisser Firebase générer un ID aléatoire,
- La racine des bases de données est toujours une collection même si elle n'en contient qu'un seul un document.

2.6.2 "Firestore is schema less"

La base de données Firestore est sans schéma, ce qui signifie que nous avons la liberté d'écrire les champs et les types de données dans un document, et nous ne sommes pas nécessairement obligés d'écrire les mêmes champs dans des documents au sein de la même collection. Donc si nous avons par exemple une collection **d'utilisateurs**, nous pouvons plus tard ajouter plus de champs à un document **utilisateur** sans rien casser, nous n'avons pas la même liberté en SQL où nous avons nos tables avec des colonnes clairement définies pour chaque ligne et un type de données prédefiner que nous pouvons mettre là dedans, mais à des fins de requête, il est généralement préférable d'avoir les mêmes champs sur plusieurs documents.

Firebase Real Time Database, l'ancienne base de données Firebase est aussi une base de données NoSQL, mais pas avec cette structure des documents et collections, à la place tout est organisé dans un seul arbre.

2.6.3 Les règles de Firestore :

1. un document est limité à un mégaoctet de taille,
2. une collection ne peut contenir autre chose que des documents ; ni champs ni autres collections ; juste des documents,
3. les documents ne peuvent pas contenir d'autres documents, cependant un document peut contenir des sous-collections.

Donc il est souvent le cas d'avoir une collection qui contient des documents, qui contiennent des sous-collections, qui contient des documents et ainsi de suite (vior la

figure 1.7).

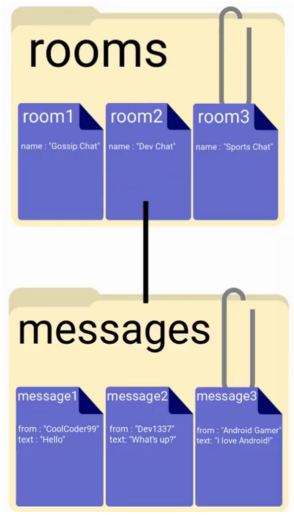


FIGURE 2.6 – subCollection

Par exemple, vous pourriez avoir une application de "chat Rooms" qui enregistre ses salles de discussion dans une collection appelée **rooms**, où chaque salle est représentée par un document, et avec Firestore optimisé pour un grand nombre de petits documents, comme des dizaines de millions voir des milliards, nous pourrions stocker chaque message de discussion dans un document séparé, de sorte que chaque salle aurait une sous-collection appelée messages qui contient tous les **messages** sous forme de documents et bien sûr, chaque salle pourrait avoir plus de sous-collections supplémentaires (Voir figure 1.6, figure 1.7).

Les documents et les collections sont créés implicitement, nous créons simplement une référence pour laquelle nous définissons une valeur, si elle n'existe pas, elle sera créée et si elle existe, elle sera mise à jour. Lorsque nous supprimons tous les documents d'une collection, la collection sera supprimée également.

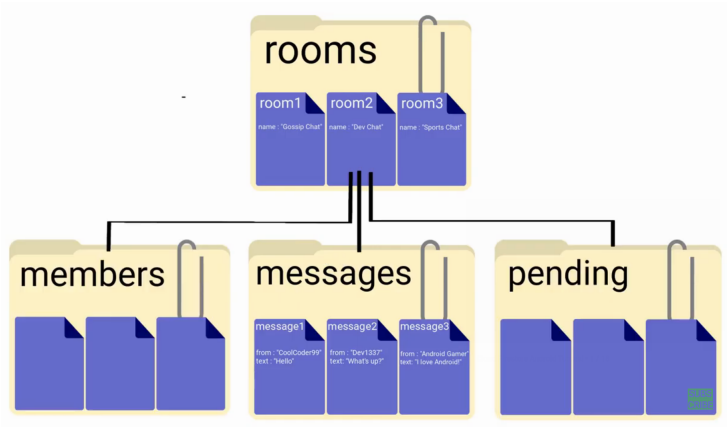


FIGURE 2.7 – Firestore Collections Documents and subCollections

Il convient également de mentionner que les requêtes Firestore sont superficielles par défaut, ce qui signifie que si nous voulons obtenir un document, nous n'avons pas à récupérer toutes ses sous-collections avec lui. Tandis que dans la base de données Firebase RealTime lorsque nous récupérons un élément de l'arborescence, nous devons également récupérer toutes les données en dessous, de sorte que les performances dans Firestore sont proportionnelles à la taille de l'ensemble de résultats et non à la taille

de l'ensemble de données.

La base de données Firestore fonctionne également hors ligne, bien sûr, nous ne pouvons rien mettre à jour dans le cloud sans connexion Internet, mais nous avons une copie des données Firestore que notre application utilise actuellement en cache sur l'appareil et vous pouvez l'interroger, l'écouter et y apporter des modifications dès que nous serons de retour en ligne, il sera synchronisé avec le cloud. Cette fonctionnalité de persistance hors ligne est activée par défaut sur Android.

2.6.4 Firestore vs SQl

Généralement les bases de données relationnellles comme Postgres ou MySQL, sont normalisées. Une ligne dans la table SQL peut être considérée comme un document dans une collection Firestore. L'ID de document serait comme la clé primaire sur une table et nous pourrions référencer d'autres ID à l'intérieur de ce document, tout comme nous le ferions avec des clés étrangères dans une table SQL.

Il n'y a pas de jointure dans Firestore, où en envoyant une requête conjointe à la base de données elle-même. Nous pouvons toujours joindre efficacement les données en faisant des requettes coté client, ce que nous examinerons plus loin dans cette partie. Et la raison pour laquelle nous n'avons pas de jointures dans Firestore, ou aucune base de données SQL, d'ailleurs, c'est parce qu'elles sont extrêmement inefficaces car la création de données utilisateur est devenue de plus en plus prolifique avec des applications comme Twitter, Facebook, Snapchat, etc.

Il a été difficile d'exploiter des bases de données SQL à grande échelle car elles nécessitent beaucoup de ressources CPU pour effectuer ces jointures. L'une des forces motrices derrière NoSQL ou Firestore est donc de concevoir vos données de manière à ne pas nécessiter de joint. Mais cela nous amène au concept le plus important de Firestore, à savoir modéliser vos données en fonction de la façon dont elles seront consommées par l'application.

Firestore a également des types de données spéciaux que nous nous ne trouverons pas dans la plupart des autres bases de données (coordonnées géographique, Arrys, Maps).

2.6.5 Modélisation de données relationnelles dans Firestore

Cette section présente un aperçu des quatre techniques principales que nous allons combiner de différentes manières pour la modélisation de données relationnelles dans Firestore.

Technique N°1 (l'Incorporation)

La première technique est l'Incorporation : ce qui signifie ajouter des données directement à un document.
Exemple : nous pourrions avoir une collection `post`. Et puis à l'intérieur de ce document, nous aurons un champ appelé `tags`. Et nous allons mettre tous nos `tags` directement dans ce document au lieu de les normaliser dans leur propre collection.

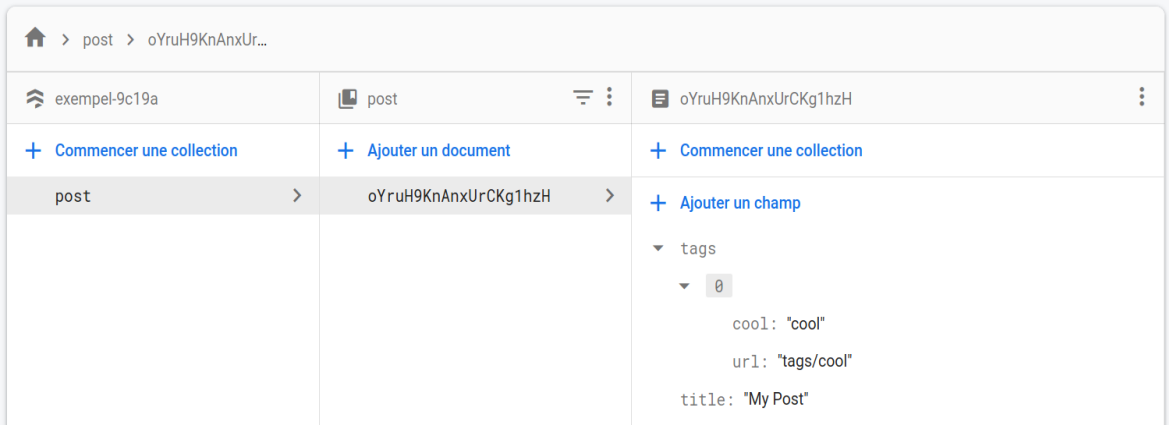


FIGURE 2.8 – Technique N°1 (l'Incorporation)

C'est généralement la première technique à considérer car c'est la plus performante. Et aussi la plus rentable, car vous n'aurez besoin que d'une seule lecture du document pour obtenir toutes les données. Mais ce n'est vraiment pratique que si on dispose d'un ensemble de données suffisamment petit pour tenir dans un mégaoctet.

Et vous n'avez pas non plus la possibilité d'interroger partiellement les données intégrées dans ce document. vous ne pouvez que lire l'intégralité du document à chaque fois que vous souhaitez y accéder.

Technique N°2 (Créer une collection racine)

La technique numéro deux consiste à créer une collection racine. Au lieu d'incorporer les données, nous donnerons à chaque `tag` son propre document dans une autre collection racine, puis référencerons son ID dans le document de `post`.
Exemple : Nous créons donc un document de `tag` appelé "cool", puis nous fournissons les mêmes données que celles que nous avions dans la `Map` intégrée dans l'exemple précédent.

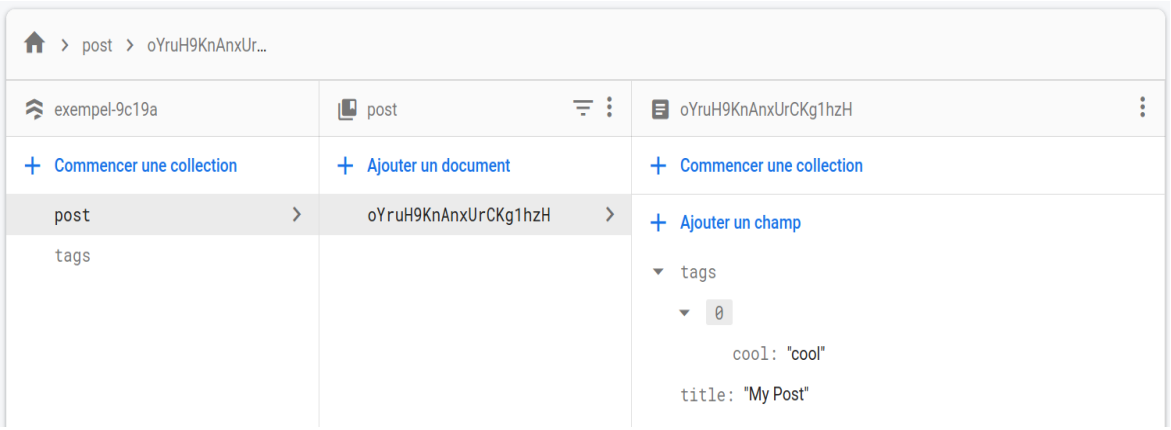


FIGURE 2.9 – Technique N°2 (root collections "posts")

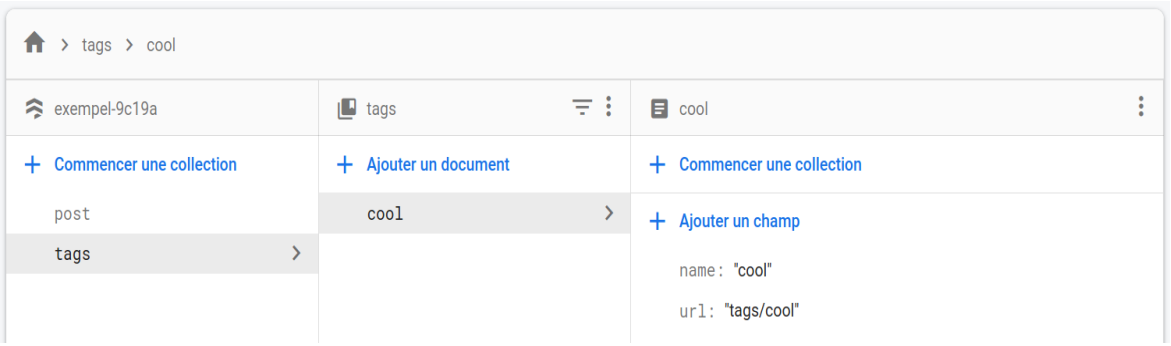


FIGURE 2.10 – Technique N°2 (root collections "tags")

Avec la collection `tags`, nous avons une (relation plusieurs à plusieurs). Lorsqu'un `tag` associé à de nombreux `posts` et un `post` est lié à de nombreux `tags`, les `posts` individuels n'auront peut-être qu'un à cinq `tags` si on pense à quelque chose comme Twitter, mais un `tag` peut avoir des milliers ou des millions de `posts` associés à une collection, nous pouvons maintenir la relation en référençant simplement les identifiants `tag` sur le document `post`, puisque nous avons plusieurs `tags` nous avons utilisé un tableau. Mais si on n'avez qu'une seule `tag`, ou si on avez juste une (relation un à un), nous pouvons simplement utiliser une propriété sur le document.
Alors, nous savons que le nom ou l'ID des `tags` que nous voulions utiliser. Et nous pouvons extraire ces documents individuellement si nous en avons besoin pour une vue spécifique.

Technique N°3 (imbriquer ces données autant que sous-collection)

Les collections racine sont vraiment intéressantes car elles offrent une grande flexibilité lors de l'interrogation sur plusieurs propriétés. Mais une autre technique consiste

à imbriquer ces données autant que sous-collection. Cela permettra d'étendre les données dans un document individuel, mais nous ne lirons réellement les données que sur une demande spécifique.

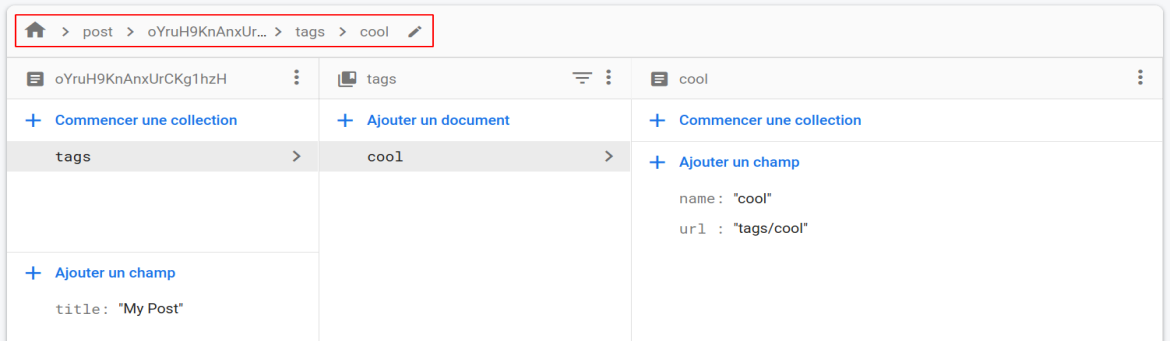


FIGURE 2.11 – Technique N°3 (imbriquer ces données autant que sous-collection

Nous ajoutons donc une collection à l'intérieur d'un document, puis créons une nouvelle collection appelée `tags`. Et nous obtiendrons un chemin qui commence à `posts` et va à l'identifiant de `post`, puis à la collection `tags` sous ce document.
La bonne chose à propos d'une sous-collection que c'est une relation (un à plusieurs) entre les `posts` et les `tags`, elle est simplement mise en place implicitement, car nous pouvons simplement référencer cette collection sous le document. Nous n'avons donc pas besoin de configurer le tableau d'identifiants de `tags` comme nous l'avons fait avec la collection racine .

Technique N°4 (bucketing)

La technique suivante est appeler "Bucketing". Avec des données incorporées, nous effectuons une seule lecture du document, mais nous pourrions extraire des données dont nous n'avons pas réellement besoin. Avec une collection racine, il se peut que nous devions effectuer plusieurs lectures de documents pour une petite quantité de données. Donc, un Bucket est une sorte de compromis entre ces deux où nous créons un nouveau document dans une collection différente. Mais ce document contient toutes les données relationnelles intégrées.

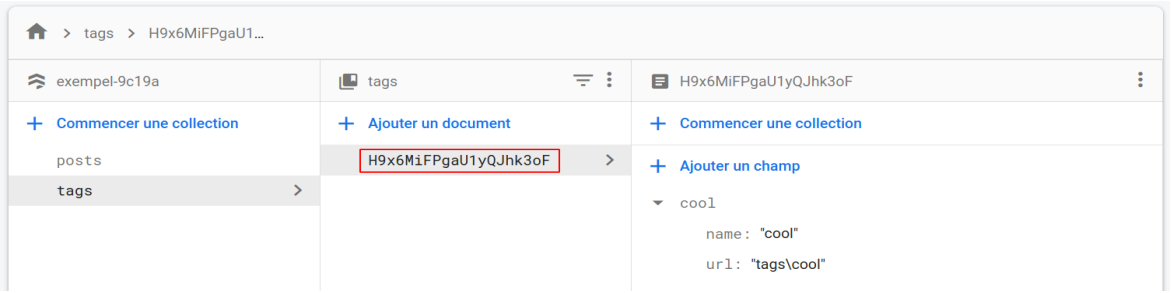


FIGURE 2.12 – Technique N°4 (bucketing "posts")

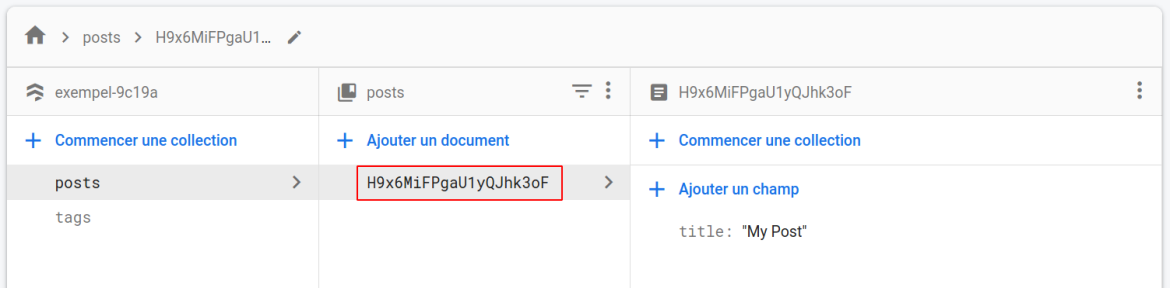


FIGURE 2.13 – Technique N°4 (bucketing "tags")

Nous avons donc deux documents avec le même ID, dont l'un dans la collection `tags` et l'autre dans la collection `posts`. Donc, si nous voulions lire les données d'un `post`, nous saisissons le `post` par cet identifiant, puis si nous voulions également afficher les données `tag` dans cet écran ou cette vue, alors nous pourrions lire le document `tag` aussi bien avec le même ID.

En d'autres termes, nous avons un document pour afficher les principales données du `post`, et un deuxième document appelé `tag`, qui est juste là pour contenir les données relationnelles. Donc, si nous avions un `post` avec 50 `tag`, nous pourrions afficher toutes les données avec seulement deux lectures de document au lieu de 51 lectures de document.

L'intégration de collections racine, des sous-collections et du compartimentage sont donc les quatre techniques principales que nous allions combiner pour créer des modèles de données de plus en plus complexes. Mais avant d'en arriver là, nous devons comprendre le fonctionnement des requêtes Firestore et leurs limites.

2.6.6 Requêtes dans Firestore

Lire les données à partir de Firestore

Nous allons maintenant examiner le code utilisé pour lire les données appartir d'un document ou d'une collection Firestore.

Si nous avons affaire à des **données incorporées**, la seule chose que nous devons savoir c'est l'ID du document, puis nous pouvons exécuter une lecture de ce document. Ce code est vraiment aussi simple que possible, nous référençons simplement la collection que nous appellerons collection posts. Et nous référençons le document par son ID.

```
const ref = db.colection("posts").document("postID")
```

Dans les extraits de code suivants, nous montrerons seulement le modèle d'accès réel pour référencer les données. La règle générale de la modélisation des données est

```
const ref = db.colection("posts").document("postID").get()
```

```
const ref =db.colection("posts").document("postID").onSnapshot()
```

que "si mon code de requête est très abrégé et lisible. Mais si mon code de requête est vraiment complexe, lorsque on fait toutes sortes de jointures et toutes sortes d'autres logiques complexes, cela signifie que je pourrais refléchir à ce modèle de données.

Les **sous-collections** sont également simples et agréables, car si nous connaissions déjà l'ID du document, tout ce que nous avons à faire est de référencer la collection de cette façon sous ce documents en appelant `.collection("collection-name")`.

```
const ref = db.collection("posts").document("postID").collection("tags")
```

Maintenant, si nous utilisions la stratégie **Bucketing** que nous avons examinée dans la section précédente nous pourrions éxècuter des requêtes très simples. Au lieu de lire un seul document, nous faisons deux lectures de document. Et nous pouvons les exécuter simultanément en utilisant le même ID du document correspondant. Dans ce cas, nous avons un document supplémentaire lu. Mais si nous affichions uniquement des `tags` sur un petit sous-ensemble des vues réelles de l'application, il est probablement logique de séparer ces données dans leur propre document dans une collection différente.

Firestore requêtes :interroger, filtre et trier des collections

Dans cette section, nous examinerons ce que nous pouvons et ce que ne pouvons pas faire avec les requêtes Firestore.

Pour faire une requête, nous commençons par faire une référence à une collection comme la collection `post`, l'une des méthodes les plus utiles que vous trouverez c'est `ordersBy()`, elle renvoie la collection dans un ordre spécifié et fait tout ce tri pour nous coté serveur, vous n'avez donc pas à vous en soucier côté client.

Nous pouvons toujours passer un deuxième argument de `"desc"` pour l'ordre décroissant car la valeur par défaut est ascendante. Il est également important de souligner que `ordersBy()` également peut être utilisé comme filtre.

```
const query = db.collection("posts").orderBy("date","desc")
```

Par exemple, si vous triez par une propriété qui n'existe que sur un sous-ensemble de documents, il filtrera tous les documents qui ne contiennent pas cette propriété. Et lorsque vous effectuions des requêtes dans Firestore.

```
const query = db.collection("posts").orderBy("date","desc").limit(20)
```

Et nous n'effectuerons que 20 lectures de documents et récupérerons 20 documents à partir de notre requête. En plus de limiter nos documents, nous pouvons également les paginer.
Nous classons d'abord les documents par date, puis nous pouvons commencer après la date de la semaine dernière. `startAfter()` est exclusif, ce qui signifie qu'il va tout nous donner après la semaine dernière, mais pas la semaine dernière. Si nous voulions inclure la semaine dernière, nous pourrions faire `startAt()` qui rendrait cette requête inclusive en nous donnant tout ce qui date de la semaine dernière et au-delà.

```
const query = db.collection("posts").orderBy("date","desc")
                        .startAfter(lastWeek)
```

```
const query = db.collection("posts").orderBy("date","desc")
                        .startAt(lastWeek)
```

Nous pouvons également paginer dans le sens inverse en utilisant `endAt()`, qui est la version inclusive et `endBefore()` pour la version exclusive.

L'autre méthode de requête de Firestore est `where()`. La méthode elle-même prend trois arguments. Le premier argument est le champ avec lequel nous souhaitons filtrer nos données, le second argument est un opérateur qui peut être `"=="`, `"<="`, `">="` , `"<"` etc.
Ces opérateurs fonctionnent sur des nombres, bien sûr, mais nous pouvons également les utiliser sur des Strings, des Date/temps et d'autres valeurs. Enfin, le troisième argument est la valeur à laquelle vous souhaitez comparer. Donc, si nous voulions

obtenir tous les `post` qui ont été publiés aujourd'hui, nous pourrions faire une requête pour savoir où la date est égale à aujourd'hui.

```
const query = db.collection("posts").where("date","==",today)
                        .where("name","==","houssem")
```

Nous pouvons également enchaîner plusieurs instructions `where()` ensembles, ce qui nous donnera une requête logique `&`. Si nous filtrons par plusieurs valeurs, nous aurons besoin d'un index composé, et gardons à l'esprit que nous ne pouvons utiliser au plus un seul opérateur d'intervalle par requête. Une autre limitation c'est qu'il n'y a pas opérateur d'inégalité. Cependant, nous pouvons simuler une requête d'inégalité en effectuant simplement deux requêtes combinées.
Par exemple, si nous voulions tous les documents dont le score différent de 5, nous pourrions simplement créer deux requêtes distinctes utilisant des opérateurs d'intervalle, puis les combiner après la lecture de ces documents.

```
const query = db.collection("posts").where("score","<", 5)

const query = db.collection("posts").where("score",">", 5)
```

Cela nous donne tout ce qui est au-dessus de **5** et tout ce qui est au dessous de **5**. Autre chose qui n'est pas directement prise en charge est l'operateur `or`. Cependant, nous pouvons toujours créer plusieurs requêtes et les combiner coté client, les deux requêtes que nous venons de faire sont essentiellement identiques à une requête `or` car nous avons demandé tous les documents qui sont au-dessus de **5** ou en dessous de **5**.

Nous voulons également montrer l'opérateur `array-contains`, c'est un opérateur spécial qui ne fonctionnera que sur le type de données tableau, et il vérifiera ce tableau pour voir si une certaine valeur existe et ne retournera que les documents qui ont une correspondance. `array-contains()` a la limitation de ne pouvoir vérifier qu'une seule valeur à la fois, mais c'est un outil très utile.

```
const query = db.collection("posts").where("tags","array-contains","cool")
```

2.7 Firebase Cloud Functions

Firebase est un back-end autant que service, nous permet de faire quelque chose appelée **"serverLess computing"** et cela signifie que nous n'avons pas à configurer notre propre serveur pour créer des applications et nous pouvons utiliser Firebase Service à cette fin.

Nous pouvons utiliser des services tels que **Database**, **Authentification**, **Hosting** ou **Cloud-Storage**, et nous pouvons utiliser tous ces différents services Firebase pour nos applications, et nous ne sommes pas obligés de configurer notre propre serveur pour gérer tout ça. Nous voulons parfois écrire notre propre code coté serveur pour peut-être ajouter des rôles aux utilisateurs, valider des données ou réagir aux modifications de notre base de données. C'est un peu difficile à faire si nous utilisons une approche sans serveur parce que nous n'avons pas de serveur pour le faire.

Firebase a créé Cloud Functions pour nous permettre de faire cela, cela signifie essentiellement que nous pouvons écrire du code qui s'exécute coté serveur sur les serveurs Firebase ; et ce code peut interagir avec d'autres services Firebase tels que **Database**, **Aauthentication** ou **Storage**, etc. Chaque bit de code que nous écrivons est conditionné dans une fonction et déployé sur Firebase. Chaque fonction peut faire quelque chose de différent, par exemple :

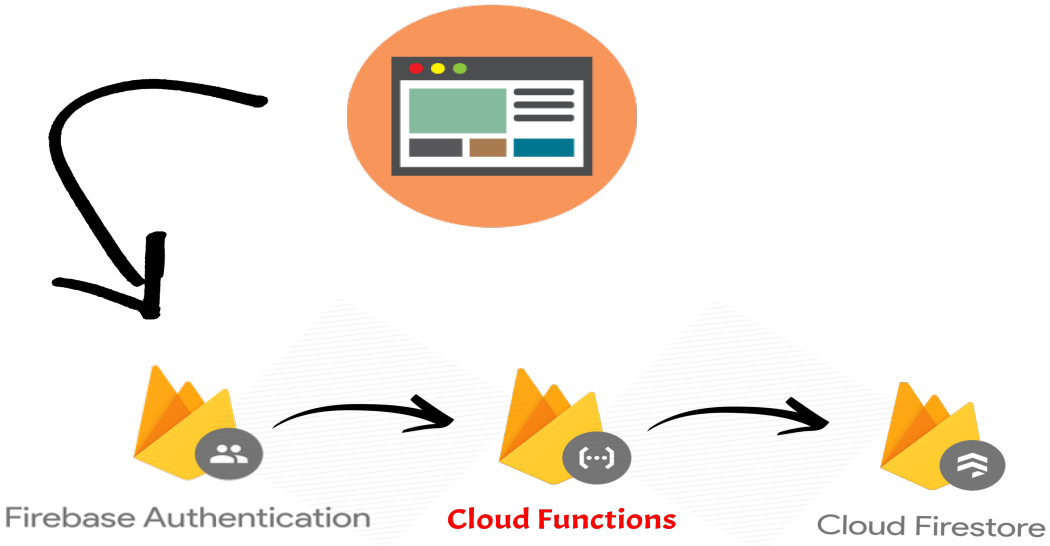


FIGURE 2.14 – Functions use case exemple

Nous pourrons avoir un nouvel utilisateur qui s'inscrit à notre application en utilisant l'authentification Firebase et nous voulons créer un enregistrement dans la base de données pour cet utilisateur, mais parfois nous ne voulons pas autoriser l'accès en écriture à la base de données depuis le front-end, nous souhaitons faire cela à partir du serveur, nous pourrions donc écrire et déployer une fonction cloud à cette fin.
La fonction écoute les inscriptions de nouveaux utilisateurs et se déclenche chaque fois que cet événement se produit pour créer un enregistrement de base de données pour cet utilisateur, il existe de nombreux cas d'utilisation différents pour Cloud Functions.

Firebase Cloud Functions s'exécute dans un environnement "Node.js" sur le service Firebase et cela signifie que nous pouvons écrire nos fonctions en JavaScript ou TypeScript.

2.7.1 Sources d'événements

Cloud Functions est un service Google Cloud Platform (GCP), mais il comporte une série d'événements spécifiques à Firebase, c'est pourquoi nous appelons souvent ces événements Cloud Functions pour Firebase.
Voici une liste rapide de certaines sources d'événements Cloud Functions :

— HTTP
— Firebase Realtime Database
— Firestore
— Firebase Authentication
— Firebase Storage.

HTTP triggers

Cloud Functions nous permet d'utiliser le framework **"Express"** pour **"Node.js"** pour gérer les appels HTTP. Nous pouvons monter une application Express entière sur un point de terminaison Cloud Functions, ou nous pouvons attacher un seul gestionnaire de requêtes Express à chaque point de terminaison Cloud Function. C'est assez flexible et nous permet de créer tout type d'API basée sur HTTP dont nous pourrions avoir besoin.

Firebase Realtime Database triggers

Nous pouvons écouter n'importe quelle partie de notre arborescence JSON Real-Time Database avec les déclencheurs suivants :

— onWrite()
— onCreate()
— onUpdate()
— onDelete()

Les événements font exactement ce que nous pensons. Le seul problème est que `onWrite()` s'exécute pour toutes les modifications apportées à l'arborescence de données, y compris les suppressions.

Firestore triggers

Firestore prend en charge les mêmes déclencheurs Cloud Functions que la RealTime Database :

— onWrite()
— onCreate()
— onUpdate()
— onDelete()

`onWrite()` est déclenché pour toutes les modifications, y compris les suppressions. [Full-Stack Firebase]

Firebase Authentication triggers

L'authentification n'a que deux déclencheurs :

— onCreate()
— onDelete()

Nous pouvons utiliser l'événement `onCreate()` pour définir les attributs personnalisés sur les jetons d'authentification de nos utilisateurs. Cette méthode est connue sous le nom de **Custom Claims** est extrêmement utile pour accorder des privilèges élevés dans les règles de sécurité Firebase, Firestore et Storage. [Full-Stack Firebase]

Exemple :

Firebase Storage triggers

Firebase Storage a un seul déclencheur avec deux façons de l'écouter :
La fonction facultative `.bucket('bucketName')` permet un filtrage basé sur des **storage buckets**

Exemple :

```
const admin = require('firebase-admin');
const serviceAccount = require('path/to/serviceAccountKey.json');
admin.initializeApp({
  credential: admin.credential.cert(serviceAccount),
  databaseURL: 'https://databaseName.firebaseio.com',
});

const emails = new Set(['housssem111@gmail.com', ''housssem222@gmail.com']);

exports.setCustomClaims = functions.auth.user().onCreate(event => {
  let promise = Promise.resolve();
  if (emails.has(user.email) {
    promise = admin
      .auth()
      .setCustomUserClaims(user.uid, { admin: true })
      .then(() => true);


  }

  return promise;
});

exports.generateThumbnail = functions.storage.object().onChange(event => {
  // ...
});

exports.generateThumbnail = functions.storage
  .bucket('bucketName')
  .object()
  .onChange(event => {
    // ...
  });
```

Firebase Storage est soutenu par Google Cloud Storage, qui traite chaque bucket comme un seul magasin.
Cloud Storage nous permet de mettre des slashes dans les noms de fichiers, c'est ainsi que Firebase Storage simule les dossiers. C'est pourquoi on ne peut pas écouter les

événements Cloud Storage qu'au niveau du bucket. On doit effectuer notre propre filtrage en fonction des noms de fichiers de nos objets.Full-Stack Firebase

2.8 Conclusion

Firebase est une puissante plate-forme d'applications Web et mobiles pour résoudre les principaux défis du développement d'applications. Les développeurs peuvent utiliser Firebase pour créer rapidement des applications mobiles sophistiquées.

Chapitre 3

Architecture et Langage de Programmation

Créer une application structurée, maintenable et testable n'est pas si simple. Pour écrire un code maintenable, compréhensible et testable, nous suivons des design patterns. Dans ce chapitre, on présente une démarche de design pattern la plus populaire pour développer des applications Android. On présente le design pattern **MVVM** ou **(Model View View Model)**. **MVVM** est basé sur **"Android Architecture Components"** (**Android Jetpack**).

Ce modèle est recommandé par Google. Avant de passer au **MVVM**, donnons d'abord des généralités sur les design patterns. Comme l'utilité des design patterns (modèle de conception). Nous développions les applications et cela fonctionnait bien, alors quel est le besoin de rendre les choses plus compliquées.
Supposons un scénario dans lequel on travaille sur une application. Cela fonctionnait bien, mais après un certain temps, soit 3 ou 4 mois, on aura du mal à comprendre ce qu'on a écrit. Mais si on suit les directives appropriées, les design patterns et les meilleures pratiques pour écrire le code, il sera facile à comprendre.

- L'utilisation de Design Pattern est donc de rendre notre code plus compréhensible. Si le code est compréhensible, il est facile de travailler dessus. Ainsi, suivre un modèle de conception approprié rend notre projet maintenable à long terme,
- Mettre le code dans les classes d'activités ou des fragments. Comme un "Network-Call" ou une "lecture de Database" ...Cela rend le code fortement couplé, ce qui signifie que mes classes dépendent les unes des autres, ce que pose des difficultés lors de l'ajout ou la modification du code. Opté pour le modèle de conception (MVVM), permis de réduire le couplage, et cela rend le projet faiblement couplé,

L'utilisation d'un modèle de conception est nécessaire pour créer une bonne appli-

cation. Google suggère quelques principes architecturaux communs que presque tous les modèles de conception suivent.

— En premier lieu, la séparation des préoccupations. L'activité ou fragment doit contenir les logiques qui sont responsables des fonctionnalités liées à l'interface utilisateur uniquement. L'un des problèmes courants qu'on pourra rencontrer, lorsque on fait un appel réseau dans un fragment, et l'essai de mettre à jour l'interface utilisateur en récupérant des données, l'application crashe. Donc,la tentative de mettre à jour un fragment qui est déjà détruit, cela conduit au crashe. On doit donc rendre ces classes d'interface utilisateur indépendantes des appels réseau, ce qui permet d'éviter les problèmes liés au cycle de vie,

— Le principe suivant, est qu'on doit dériver les données de l'interface utilisateur à partir d'un modèle. Le modèle est le composant responsable de la gestion des données de l'application et il est indépendant des vues. Et comme il est indépendant des vues, il n'est pas affecté par les problèmes de cycle de vie.

3.1 Pourquoi MVVM ?

On a les autres modèles de conception comme **MVC**, **MVP**. La sélection d'une architecture dépend entièrement de nos besoins. Il n'existe pas une architecture parfaite qui résout tous les problèmes. Chaque architecture a ses propres avantages et inconvénients, et elles sont choisies en fonction du problème. Google recommande d'utiliser MVVM. Le diagramme suivant explique l'architecture MVVM.

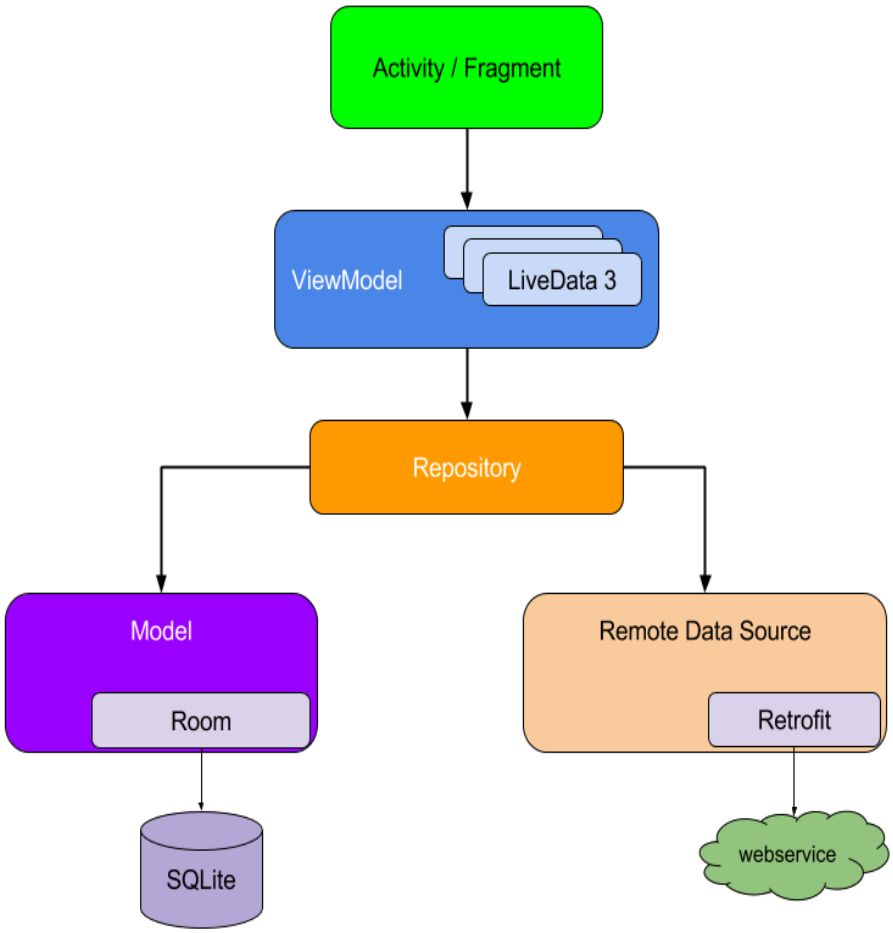


FIGURE 3.1 – MVVM Architecture

Nous avons d'abord les classes **Activity ou Fragment**, c'est essentiellement l'interface utilisateur de notre application. Notre activité dépend simplement de **ViewModel** pour obtenir les données. L'avantage est que **ViewModel** ne mettra pas à jour les données si la vue est détruite afin que le problème des modifications du cycle de vie soit résolu. **ViewModel** récupérera les données du **Repository**, donc encore une fois **ViewModel** dépend simplement du **Repository** pour obtenir les données. Le **Repository** peut utiliser la base de données locale qui est **SQLite** ou notre serveur backend utilisant **l'API Restful** ou dans notre cas **Firebase**. Ainsi, seul **Repository** dépend de plusieurs sources de données pour obtenir les données. Notre diagramme sera comme suit :

FIGURE 3.2 – MVVM with Firebase

Ce diagramme vient directement de **Android Architecture Components**, sauf que nous n'utiliserons pas **Room** pour la persitence locale puisque Firestore a déjà son propre mécanisme de mise en cache, ni un service Web comme source de données distante, nous serons en utilisant simplement Firebase.

3.2 Qu'est-ce que Android Architecture Components (Android Jetpack)

Dans cette section, nous parlerons de ce qu'est **Android Architecture Components** donnant un aperçu de ce qu'est **Android Jetpack**. Les composants de **Android**

Architecture Components sont un guide de l'architecture des applications Android avec des bibliothèques pour différentes tâches. Ils nous aident à créer une application robuste, testable et maintenable.
Android Architecture Components est une partie du **Android Jetpack**. La figure 2.3 donne un aperçu du **Android Jetpack**.

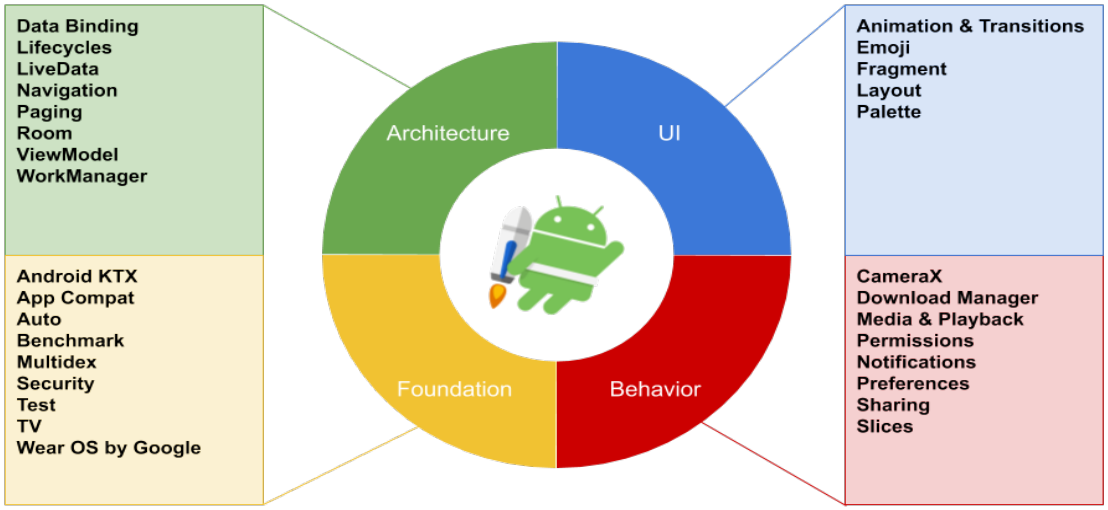


FIGURE 3.3 – Android-Jetpack

Android Jetpack est divisé en :
— **Architecture**
— **UI**
— **Fondation,**
— **Behavior**

Sous **Android Jetpack** il y a une collection de composants logiciels Android. Les quatre parties dans lesquelles ce **Android Jetpack** est divisé. **Android Jetpack** facilite le développement d'applications Android, car il nous donne une structure et une architecture pour construire notre application, et il nous aide à suivre les meilleures pratiques, ce qui rend l'application robuste et maintenable.

Android Architecture Components comprennent :
— **data binding**
— **Lifecycle Aware Components**
— **Live Data**
— **Navigation**
— **Paging**
— **Room**
— **ViewModel**

— **Work Manager**

Nous nous concentrerons sur **lifecycle Aware Components, Live Ddata, View-Model** et **Navigation**

3.3 Lifecycle Aware Components

Lifecycle Aware Component est un composant qui est conscient du cycle de vie d'autres composants, comme une **activité** ou un **fragment,** (life cycle owner) et effectue une action en réponse au changement de l'état du cycle de vie de ce composant (life cycle owner).

3.3.1 Pourquoi Lifecycle Aware Component ?

Dans les applications de lecteur vidéo simple, où nous avons une activité nommée `VideoActivity` qui contient l'interface utilisateur pour lire la vidéo et nous avons une classe nommée `VideoPlayer` qui contient toute la logique et le mécanisme pour lire une vidéo. Notre `VideoActivity` crée une instance de cette classe `VideoPlayer` dans la méthode `onCreate()`.

```
public void onCreate(...){
 //some UI setup
 mVideoPlayer = new VideoPlayer();
 }
```

Comme pour tout lecteur vidéo, nous aimerions qu'il lise la vidéo lorsque VideoActivity est au premier plan (n **resumed state**) et met en pause la vidéo lorsqu'elle passe en arrière-plan ((in the **paused state**)). Nous aurons donc le code suivant dans nos méthodes `onResume()` et `onPause()` de VideoActivity.

```
public void onResume(){
 mVideoPlayer.play();
 }

 public void onPause(){
 mVideoPlayer.pause();
 }
```

De plus, nous aimerions qu'il arrête complètement de jouer et libère les ressources lorsque l'activité est détruite. Ainsi, nous aurons le code suivant dans la méthode `onDestroy()` de VideoActivity.

```
public void onDestroy(){
mVideoPlayer.stop();
}
```

Utilisation de "arch.lifecycle"

Avec l'introduction de **Lifecycle Aware Component** dans la bibliothèque `'android.arch.li` nous pouvons déplacer tout ce code vers les composants individuels. Nos activités ou fragments n'ont plus besoin de jouer avec ces logiques de composants et peuvent se concentrer sur leur propre travail principal, c'est-à-dire maintenir l'interface utilisateur. Ainsi, le code devient propre, maintenable et testable.

Le package `'android.arch.lifecycle'` fournit des classes et des interfaces qui s'avèrent utiles pour résoudre ces problèmes de manière isolée.

3.4 Lifecycle Aware Component exemple

Nous pouvons ajouter les lignes ci-dessous à notre fichier (app gradle) pour ajouter des composants de cycle de vie à partir de la bibliothèque `'android.arch'`.

```
dependencies {
    // Lifecycles only (no ViewModel or LiveData)
    implementation "android.arch.lifecycle:runtime:1.1.0"
    annotationProcessor "android.arch.lifecycle:compiler:1.1.0"
}
```

Une fois que nous avons intégré les composants `'arch'`, nous pouvons faire en sorte que notre classe VideoPlayer implémente `LifecycleObserver`, qui est une interface vide avec des annotations. En utilisant les annotations spécifiques avec les méthodes de informé des changements d'état du cycle de vie dans VideoActivity. Donc, notre classe VideoPlayer sera comme :

Nous avons besoin aussi d'une liaison entre cette classe **VideoPlayer** et **VideoActivity** afin que notre objet **VideoPlayer** soit informé des changements d'état du cycle de vie dans **VideoActivity**.

Cette liaison est assez simple, **VideoActivity** est une instance de `'android.support.v7.app.` qui implémente l'interface **Lifecycleowner**. L'interface **Lifecycleowne** est une interface à méthode unique qui contient une méthode, `getLifecycle()`, pour obtenir l'objet **Lifecycle** correspondant à sa classe d'implémentation qui assure le suivi des changements d'état du cycle de vie de l'activité / fragment ou de tout autre composant ayant

```
class VideoPlayer implements LifecycleObserver{

 @OnLifecycleEvent(Lifecycle.Event.ON_RESUME)
 public void play(){
 //play logic
 }
 @OnLifecycleEvent(Lifecycle.Event.ON_PAUSE)
 public void pause(){
 //pause logic
 }
 @OnLifecycleEvent(Lifecycle.Event.ON_STOP)
 public void stop(){
 //stop logic
 }
 }
```

un cycle de vie. Cet objet **Lifecycle** est observable et informe ses observateurs du changement d'état.

Nous avons donc notre **VideoPlayer**, instance de **LifecycleObserver**, et nous devons l'ajouter autant qu'observateur à l'objet **Lifecycle** de **VideoActivity**. Nous allons donc modifier **VideoActivity** comme :

```
public void onCreate(...){
 //some UI setup
 mVideoPlayer = new VideoPlayer();
 getLifecycle().addObserver(mVideoPlayer);
 }

 public void onDestroy(){
 getLifecycle().removeObserver(mVideoPlayer);
 }
```

Cela rend les choses assez résistantes et isolées. Notre logique de classe **VideoPlayer** est séparée de **VideoActivity**. Notre **VideoActivity** n'a plus besoin de se soucier d'appeler ses méthodes de composants dépendants pour mettre en pause ou lire ses méthodes de rappel de cycle de vie, ce qui rend le code propre, gérable et testable.

3.5 ViewModel

Dans cette section, nous voulons présenter la classe **ViewModel** et nous montrerons un exemple pratique sur la façon de l'utiliser avec vos activités et fragments. La classe **ViewModel** aide les développeurs à concevoir des applications propres, robustes et maintenables.
La classe **ViewModel** est conçue pour stocker et gérer les données liées à l'interface utilisateur de manière consciente du cycle de vie, la classe **ViewModel** permet aux données de survivre à des changements de configuration tels qu'une rotation d'écran, donc lorsque nous parlons d'activité ou d'un fragment, nous faisons en fait référence aux contrôleurs d'interface utilisateur, et la classe **ViewModel** doit être créée pour chaque UI contrôleur afin de séparer des éléments tels que les données et différents calculs d'un UI contrôleur, dans un UI contrôleur, il ne devrait y avoir que du code nécessaire pour gérer notre interface utilisateur et tout le reste pourrait être placé dans la classe **ViewModel**.

3.5.1 Utilisez le ViewModel dans votre contrôleur d'interface utilisateur(activity/fragment)

Il y a trois étapes pour configurer et utiliser un **ViewModel** :

1. Séparez les données du contrôleur d'interface utilisateur en créant une classe qui étend ViewModel,
2. Configurer les communications entre ViewModel et le contrôleur d'interface utilisateur,
3. Utilisez ViewModel dans le contrôleur d'interface utilisateur.

Étape 1 : créer une classe ViewModel

Pour créer un ViewModel, nous devons d'abord ajouter la dépendance du cycle de vie. En général, nous créons une classe **ViewModel** pour chaque écran de l'application. Cette classe **ViewModel** contiendra toutes les données associées à l'écran et aura des getters et des setters pour les données stockées. Cela sépare le code pour afficher l'interface utilisateur, qui est implémentée dans les activités et les fragments, des données, qui résident désormais dans le **ViewModel**. Alors, créons une classe **ViewModel** :

```
dependencies {
    def lifecycle_version = "2.2.0"
    def arch_version = "2.1.0"

    // ViewModel
    implementation "androidx.lifecycle:lifecycle-viewmodel-ktx:\$lifecycle_versi
}
```

```
public class ScoreViewModel extends ViewModel {
   // Suit le score1
   public int score1 = 0;

  // Suit le score2
   public int score2 = 0;
}
```

Étape 2 : associer le contrôleur d'interface utilisateur et ViewModel

Notre contrôleur d'interface utilisateur (Activity / Fragment) doit connaître notre ViewModel. Cela permet à notre contrôleur d'interface utilisateur d'afficher les données et de mettre à jour les données lorsque des interactions d'interface utilisateur se produisent, par exemple en appuyant sur un bouton pour augmenter `score1`.

Créons cette association contrôleur d'interface utilisateur / ViewModel. Nous souhaitons créer une variable membre pour votre ViewModel dans le contrôleur d'interface utilisateur. Ensuite, dans `onCreate ()`, nous faisons :

```
@Override
protected void onCreate(Bundle savedInstanceState) {
   super.onCreate(savedInstanceState);
   setContentView(R.layout.activity_main);
   mViewModel = ViewModelProviders.of(this).get(ScoreViewModel.class);
   // Other setup code below...
}
```

Étape 3 : Utilisez le ViewModel dans le UI contrôleur

Pour accéder ou modifier les données de l'interface utilisateur, nous pouvons désormais utiliser les données de le ViewModel. Voici un exemple de la nouvelle méthode

onCreate et une méthode de mise à jour du score1 en ajoutant un :

```
// La méthode onCreate terminée
@Override
protected void onCreate(Bundle savedInstanceState) {
   super.onCreate(savedInstanceState);
   setContentView(R.layout.activity_main);
   mViewModel = ViewModelProviders.of(this).get(ScoreViewModel.class);
   displayScore1(mViewModel.score1);
   displayScore2(mViewModel.score2);
}

// Un exemple de lecture et d'écriture dans le ViewModel
public void addOneForTeamA(View v) {
   mViewModel.score1 = mViewModel.score1 + 1;
   displayScore1(mViewModel.score1);
}
```

Dans le diagramme ci-dessous, illustre le cycle de vie d'une activité qui subit une rotation puis est finalement terminée. La durée de vie du ViewModel est affichée à côté du cycle de vie de l'activité associé. Notez que ViewModels peut être facilement utilisé avec les fragments et les activités. Cet exemple se concentre sur les activités.

Le ViewModel existe à partir du moment où vous demandez pour la première fois un ViewModel (généralement dans onCreate the Activity) jusqu'à ce que l'activité soit terminée et détruite. onCreate peut être appelé plusieurs fois au cours de la vie d'une activité, par exemple lorsque l'application est tournée, mais le ViewModel survit tout au long de la vie.

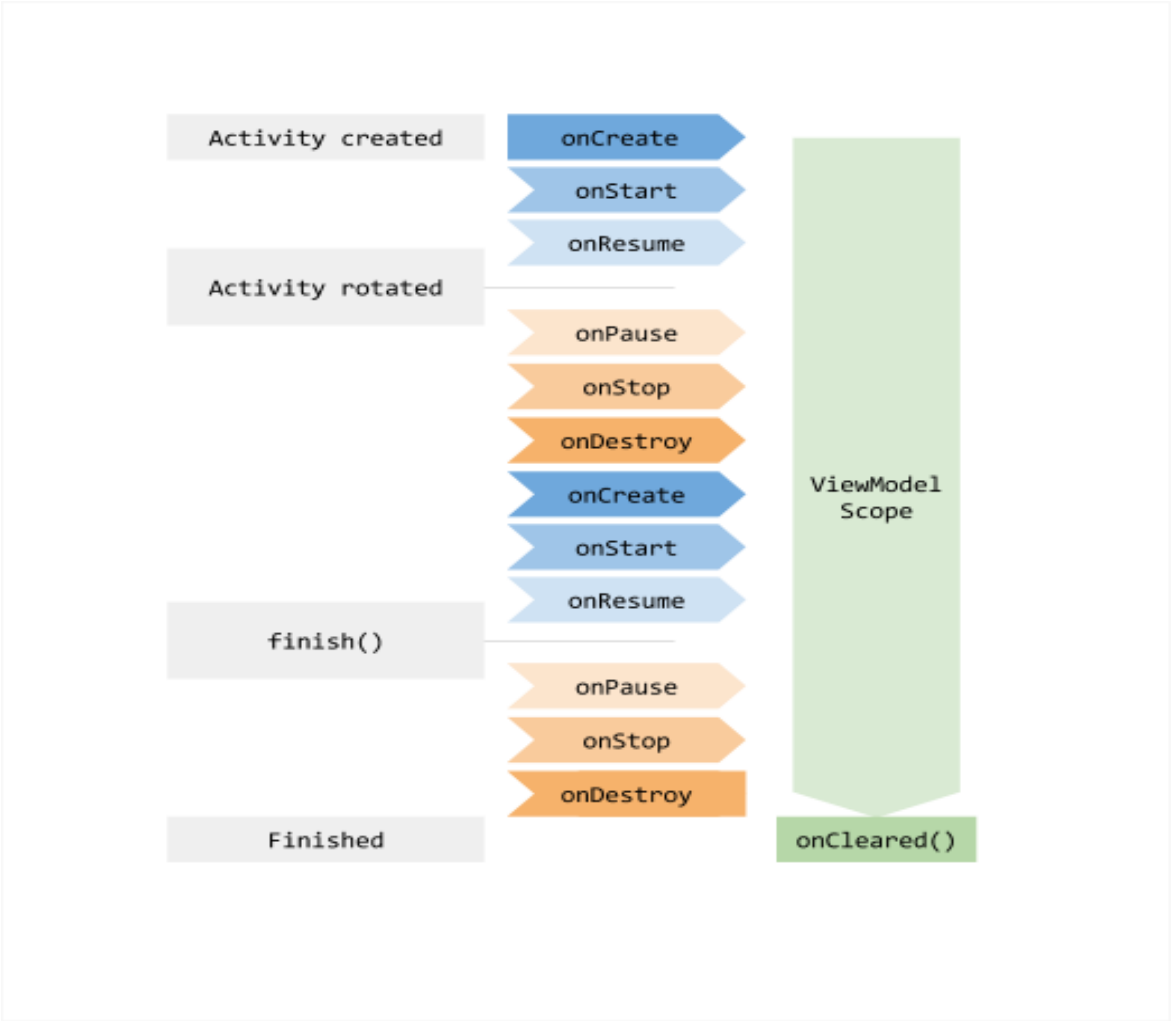


FIGURE 3.4 – ViewModele LifeCycle

3.6 LiveData

LiveData est une classe support de données observables. Contrairement à un observable ordinaire, LiveData est conscient au cycle de vie, ce qui signifie qu'il respecte le cycle de vie des autres composants de l'application, tels que les activités, les fragments ou les services. Pour mieux illustrer, mettons LiveData au centre comme le diagramme ci-dessous

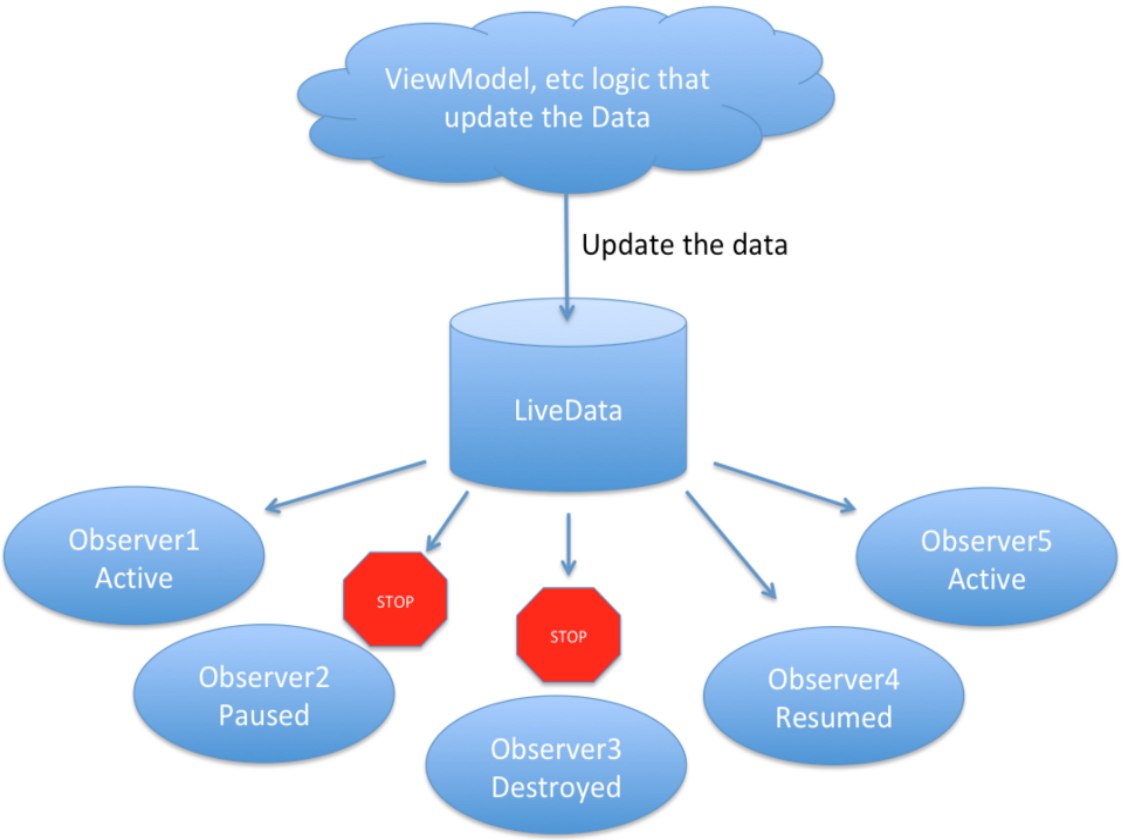


FIGURE 3.5 – Live Data illustretion

Lors de la mise à jour, elle notifierait alors tous ses observateurs (activités, fragment, service, etc.). Cependant, contrairement à toute autre approche (par exemple, Rxjava), il ne les avertit pas tous aveuglément, mais vérifie d'abord leur état en direct. Si l'observateur est actif, il peut être informé du changement de données dans LiveData. Cependant, si l'observateur est mis en pause ou détruit, il ne sera alors pas notifié. Une fois que l'Observer est repris, il serait immédiatement informé des dernières données de LiveData.

3.6.1 MutableLiveData

LiveData n'est en fait qu'une classe abstraite. Il ne peut donc pas être utilisé comme tel. Google a implémenté une classe concrète simple que nous pourrions utiliser. MutableLiveData est la plus simple LiveData, où elle servait tout simplement à obtenir les mises à jours et informer ces l'observateurs.

Le code suivant montre comment déclarer MutableLiveData

Soit un fragment qui observe un LiveData comme ci-dessous

```
 // Declaring it
  public MutableLiveData<String> liveDataA = MutableLiveData<String>()

 // Trigger the value change
  liveDataA.value = someValue

 // Optionally, one could use
  liveDataA.postValue(value)
 // to get it set on the UI thread


class MutableLiveDataFragment extends Fragment {

  private  Observer<String>  changeObserver = Observer<String> { value->
        txt-fragment.text =  value
    }

  public void onAttach(context: Context?) {
        super.onAttach(context)
        getLiveDataA().observe(this, changeObserver)
    }

    // .. some other Fragment specific code ..
}
```

À partir du code, nous pouvons voir

```
getLiveDataA().observe(this, changeObserver)
```

mais il n'y a pas de code pour le désabonner lorsque le fragment est en pause ou se termine. Même sans vous désabonner, cela ne posera aucun problème.

3.7 Navigation

Navigation permet à l'utilisateur de passer d'un écran à un autre. **Navigation component** fonctionne mieux avec une architecture à une seul activité et peut fournir une représentation visuelle du flux de navigation. De plus, nous pouvons bénéficier de ViewModel et Lifecycle, y compris la gestion de la complexité de la transition

des fragments. Le composant de navigation est une API simple qui fournit plusieurs fonctionnalités telles que :

— Configuration simplifiée pour un modèle de navigation commun
— Gère le **Back Stack**
— Traitement automatique des transactions de fragments
— **Safe Args**
— Gère les animations de transition
— simplifié l'utilisation de **Deep Linking**

3.7.1 Les bases lors de la mise en œuvre de Navigation

Navigation Graph

Le Navigation Graph est un type de ressource XML. Il comprend un éditeur visuel qui permet une représentation par glisser et déposer des fragments. Il contient également toutes les informations relatives à la navigation.

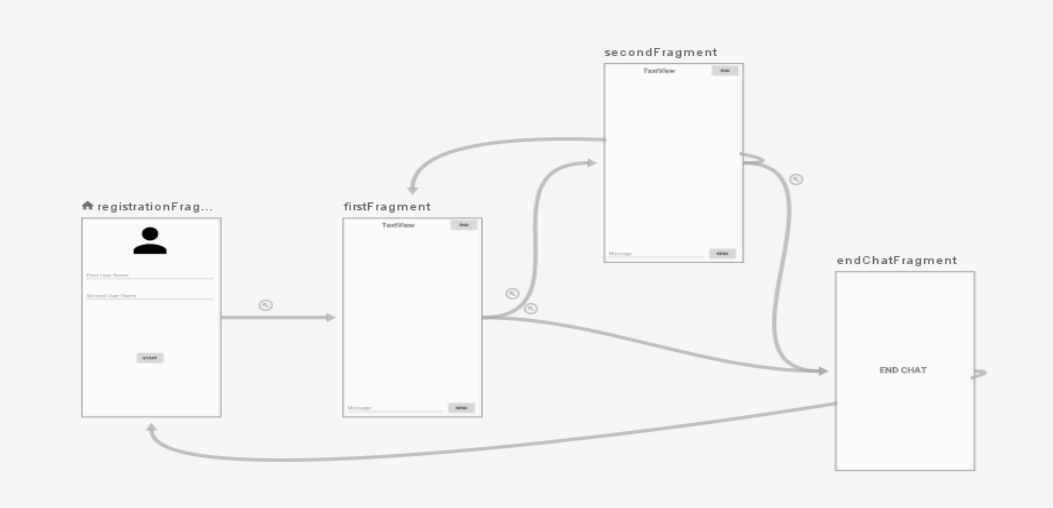


FIGURE 3.6 – Navigation Graph

Les écrans sont appelés **destinations**. Les flèches représentent l'**action** qui définit l'itinéraire de navigation. En cliquant sur la flèche, nous obtiendrons plusieurs options, notamment *animation de transition*, *définition de la destination*, *manipulation de backstack* et *transfert de données entre les destinations*. nous pouvons également définir un point d'entrée [HOME] directement dans ce graphique.
Toutes les visualisations ci-dessus sont également disponibles en XML :

```
<navigation xmlns:android="http://schemas.android.com/apk/res/android"
    xmlns:app="http://schemas.android.com/apk/res-auto"
    xmlns:tools="http://schemas.android.com/tools"
    android:id="@+id/nav_graph"
    app:startDestination="@id/registrationFragment">

    <fragment
        android:id="@+id/registrationFragment"
        android:name="com.example.navigationsample.fragments.RegistrationFragment"
        android:label="fragment_registration"
        tools:layout="@layout/fragment_registration">
        <action
            android:id="@+id/action_registrationFragment_to_firstFragment"
            app:destination="@id/firstFragment"
            app:popUpTo="@id/nav_graph"
            app:popUpToInclusive="true" />
    </fragment>

</navigation>
```

FIGURE 3.7 – Navigation Graph (XML)

Navigation Host Fragment

Navigation Host fragment est un widget ou un écran qui héberge votre **Navigation Graph**. Cela signifie que toute la navigation définie dans le **Navigation Graph** est exécutée dessus.

Nous devons définir Navigation Host Fragment dans une activité de la manière suivante :

```
<fragment
    android:id="@+id/nav_host_fragment"
    ...........
    android:name="androidx.navigation.fragment.NavHostFragment"
    app:defaultNavHost="true"
    app:navGraph="@navigation/nav_graph" />
```

Navigation Controller

Navigation Controlle est un objet qui commande l'écoulement entier du graphe de navigation. Il garde également une trace de la position actuelle et il prend également des instructions sur où naviguer et quelles actions doivent être suivies à partir de celles

définies dans le graphique. Pour naviguer d'une façon programmable, nous exécutons le code suivant :

```
findNavController()
        .navigate(R.id.action_registrationFragment_to_firstFragment)
```

`'R.id.action_registrationFragment_to_firstFragment'` est l'ID d'action défini dans le graphe de navigation [Figure 2.7].

Différentes actions disponibles

- **app : destination** : définit l'endroit vers lequel il se dirigera.
- **app : popUpTo** : dans les composants de navigation, nous ne pouvons pas accéder directement à la transition de fragment ou **addToBackStack** car elle est abstraite. **popUpTo** définit où naviguer après avoir appuyé sur le bouton retour.
- **app : popUpToInclusive** : il nécessite une valeur abooléenne, true signifie qu'il sortira la destination précédente après avoir navigué vers une autre.

Le contrôleur de navigation utilisera le **Navigation Host Fragment** pour afficher l'écran ou la destination appropriée.

3.8 Kotlin le langage de programmation de choix

Deux ans après avoir exploité Kotlin pour une utilisation dans le développement mobile Android - longtemps dominé par Java - Google fait de Kotlin l'option n ° 1. Google a promu Kotlin au statut de première classe en mai 2019 et a ensuite signalé qu'il avait été un succès auprès des ingénieurs Android.

"Le développement Android deviendra de plus en plus Kotlin-first", a déclaré Google dans [**?**] (7 mai 2019). "De nombreuses nouvelles API et fonctionnalités Jetpack seront d'abord proposées dans Kotlin.

3.8.1 Pourquoi Kotlin et non pas JAVA

Voici les principales raisons pour lesquelles, autant que développeur Android, nous devrions commencer à voir ce langage de programmation comme une alternative de Java :

1. **Il est totalement interopérable avec Java**

- L'une des plus grandes commodités avec l'utilisation de Kotlin est qu'il est compatible avec Java,
- Avec tous ses outils et frameworks, nous pouvons simplement les ajouter à nos projets Kotlin sans avoir besoin de changer le projet en Java,
- Migration d'un projet Java est simple.

2. **Kotlin est plus bref que Java**

   Nous sommes en mesure de résoudre les mêmes problèmes en utilisant moins de lignes de code qui ne peuvent se traduire qu'en un code plus fiable avec moins de bugs et de plantages côté UX. Kotlin est :
   - Plus facile à maintenir
   - Plus facile à lire
   - Plus facile à appliquer des modifications en cas de besoin.

   Certaines des fonctionnalités de Kotlin qui sont responsables de la brievté de son code sont :
   - Classes de données
   - Moulages intelligents
   - Interface de type
   - Propriétés

3. **kotlin offre un meilleur support pour la programmation fonctionnelle**

   Cela signifie que :
   - nous pouvons améliorer les performances des applications mobiles via l'inlining
   - nous pouvons "jogger avec" les concepts fonctionnels d'une manière plus explicite et concise
   - En effet, Kotlin nous permet de disposer de types de fonctions appropriés à utiliser à cet égard

4. **Il a Null-Safety dans son système de type**

   Les problèmes de nullité ont été l'un des points sensibles bien connus de Java. Comme il est courant dans Android que l'absence de certaines valeurs soit représentée comme «null», Kotlin vient résoudre ces problèmes en plaçant null directement dans son système de type.

La raison primordiale pour laquelle nous avons préfèré Kotlin à JAVA est **Coroutines**

3.8.2 Coroutine

Les coroutines sont d'excellentes fonctionnalités disponibles dans le langage Kotlin **Définition :** Les Coroutines sont une nouvelle façon d'écrire du code asynchrone et non bloquant. La première question qui se pose lors de la lecture de cette définition est en quoi les **Coroutines** diffèrent-elles des Threads ?.

Les coroutines sont des Threads allégés. Un thread allégé signifie qu'il n'est pas lié à un thread en particulier, son avantage étant qu'elle peut être suspendue et reprise plus tard. Une coroutine peut être suspendue dans un Thread et être reprise dans un autre. Il ne nécessite donc pas de changement de contexte vers le processeur, il est donc plus rapide. Il existe des coroutines dans de nombreux langages de programmation. Fondamentalement, il existe deux types de Coroutines :

— stackless Coroutines
— stackful Coroutines

Kotlin implémente Stackless Coroutines - cela signifie que les Coroutines n'ont pas leur propre pile, donc elles ne se lient pas à un thread en particulier.
Les coroutines et les threads sont multitâches. Mais la différence est que les threads sont contrôlés par le système d'exploitation, tandis que les Coroutines sont contrôlées par les utilisateurs.
Du fait qu'une coroutine est plus légère qu'un Thread, il est possible d'en créer des centaines de milliers en parallèle sur un poste classique sans avoir de problème de "out of memory", et donc de réaliser plusieurs traitements en même temps.

3.8.3 Comment ça marche ?

Généralement, lorsqu'une fonction appelle une deuxième fonction, la première ne peut pas continuer jusqu'à ce que la deuxième fonction se termine et retourne là où elle a été appelée. Le contrôle reste avec la deuxième fonction jusqu'à ce qu'il s'exécute complètement, puis le contrôle peut revenir à la première.

Comment les fonctions peuvent-elles coopérer entre elles ?

Les coroutines sont des fonctions dans lesquelles le contrôle est transféré d'une fonction à l'autre fonction de manière à ce que le point de sortie de la première fonction et le point d'entrée de la seconde fonction soient mémorisés - sans changer le contexte.

Ainsi, chaque fonction se souvient d'où elle a quitté l'exécution et d'où elle doit reprendre.

Transmettez les valeurs !

En utilisant ce concept, vous pouvez transmettre des valeurs entre des fonctions au milieu de l'exécution sans interrompre le contexte de chaque fonction.

3.8.4 Démarrer une coroutine

Il existe deux fonctions pour exécuter une coroutine :

1. `launch{}`
2. `async{}`

La différence est `launch{}` ne renvoie rien, mais `async{}` renvoie une instance Deferred qui a une fonction `await()` qui renvoie le résultat d'une coroutine, tout comme Futureen Java, où nous faisons `future.get()` pour obtenir le résultat.

L'utilisation de launch{}

```
fun main(args: Array<String>) {
    println("Kotlin Start")
    launch(CommonPool) {
        delay(2000)
        println("Kotlin Inside")
    }
    println("Kotlin End")
}\\
```

Cela donne :

```
// Kotlin Start
// Kotlin End
// Kotlin Inside
```

Ce code démarrera une nouvelle coroutine dans le pool de threads donné. Dans ce cas, nous utilisons CommonPool qui utilise ForkJoinPool.commonPool(). Les threads existent toujours dans un programme basé sur la coroutine, mais un seul thread peut exécuter plusieurs coroutines, il n'est donc pas nécessaire d'avoir trop de threads.

Suspending functions

Suspending functions sont au centre de toutes les coroutines. Une **Suspending functions** est simplement une fonction qui peut être mise en pause et reprise ulté-

rieurement. Ils peuvent exécuter une opération de longue durée et attendre qu'elle se termine sans blocage.

La syntaxe d'une **Suspending function** est similaire à celle d'une fonction régulière à l'exception de l'ajout du mot-clé `suspend`. Il peut prendre un paramètre et avoir un type de retour. Cependant, les **Suspending functions** ne peuvent être appelées que par une autre **Suspending function** ou dans une **coroutine**.
Exemple :

```
suspend fun doWorkFor1Seconds(): String {
    delay(1000)
    return "doWorkFor1Seconds"
}

suspend fun doWorkFor2Seconds(): String {
    delay(2000)
    return "doWorkFor2Seconds"
}

// Serial execution
private fun doWorksInSeries() {
    launch(CommonPool) {
        val one = doWorkFor1Seconds()
        val two = doWorkFor2Seconds()
        println("Kotlin One : " + one)
        println("Kotlin Two : " + two)
    }
}
```

Cela donne :

```
// Kotlin One : doWorkFor1Seconds
// Kotlin Two : doWorkFor2Seconds
```

Voyons maintenant comment utiliser async{}

cela nous donne :

Car nous utilisons async alors nous pouvons appeler await() pour obtenir le résultat d'une manière asynchrone.

```
// Parallel execution
private fun doWorksInParallel() {
    val one = async(CommonPool) {
        doWorkFor1Seconds()
    }
    val two = async(CommonPool) {
        doWorkFor2Seconds()
    }
    launch(CommonPool) {
        val combined = one.await() + "_" + two.await()
        println("Kotlin Combined : " + combined)
    }
}
// The output is

// Kotlin Combined : doWorkFor1Seconds_doWorkFor2Seconds
```

3.8.5 Coroutine Dispatchers

Coroutine Dispatchers sont des intercepteurs de continuation et un élément de contexte de coroutine qui garantissent que l'exécution et la poursuite d'une coroutine sont distribuées sur le bon thread. Ils peuvent être utilisés pour limiter une coroutine à un seul thread, un pool de threads ou permettre à la coroutine de s'exécuter sans restriction. Le `'kotlinx.coroutines'` est livré avec les implémentations standard suivantes :

Dispatchers.Default :

Il s'agit du dispatchers par défaut utilisé par tous les générateurs de coroutine si aucun répartiteur n'est spécifié. C'est le choix approprié pour exécuter des tâches gourmandes en CPU.

Dispatchers.IO

Utilisé pour bloquer les opérations d' Enter/Sortie qui ne sont pas gourmandes en CPU.

Dispatchers.Unconfined

Exécute une coroutine non limitée à un thread spécifique. La coroutine est immédiatement exécutée dans le thread courant mais reprend dans n'importe quel thread.

Dispatchers.Main

Ceci est fourni par l'artefact `'kotlinx-coroutines-android'` et limite l'exécution d'une coroutine au Main thread.

Exemple :

```
fun main(args: Array<String>) = runBlocking<Unit> {
    launch(Dispatchers.Default) {
        println("current thread:${Thread.currentThread().name}")
        delay(1_000)

        withContext(Dispatchers.Main){
        println("current thread:${Thread.currentThread().name}")
                }
    }

}
```

Cela nous donne

```
current thread: kotlinx.coroutines.DefaultExecutor
current thread: main
```

Dans l'exemple ci-dessus, nous pouvons voir que la même fonction s'exécute dans deux threads différents

3.9 Conclusion

Lors du développement d'une application Android, il est important de planifier l'architecture du projet. Cela nous permettra de créer des applications complexes, robustes, de bonne qualité et faciles à entretenir.

Chapitre 4

Foxy Flexy

Dans ce chapitre en premier lieu nous donnons un aperçu génèral sur le fonctionnement de l'application. Cette application à pour but la distribution de la fonctionnalité d'une puce Flexy pour plusieurs utilisateurs en temps réel en adoptant les service cloud de Firebase.

4.1 Modélisation

Les fournisseurs de credits de téléphone mobile nous ont exprimé leurs besoins de faciliter le travail en automatisant une partie de leur logique métier. Après discussion on a abouti au modèle suivant :
Le modèle consiste à faire acheminer le credit du téléphone mobile du **Fournisseur** au **Client** en passant par un **Administrateur** et un **Grossiste**.
FoxySoled représente le montant du crédit de téléphonie mobile qui peut être envoyé depuis les serveurs du fournisseur
Le **SuperUser** qui est le fournisseur, vend le FoxSold à l'administrateur, chaque Administrateur peut ajouter des Grossistes, lesquels il chapote afin qu'ils distribuent le FoxySoled aux Clients
Alors les Grossistes achètent le FoxySoled à l'Administrateur et le revendent aux Clients

Le Client comme tous les autres profils (SuperUser, Admin, Grossiste) peut exécuter des opérations Flexy en envoyant du crédit de téléphonie mobile depuis les serveurs du Fournisseur (SuperUser)

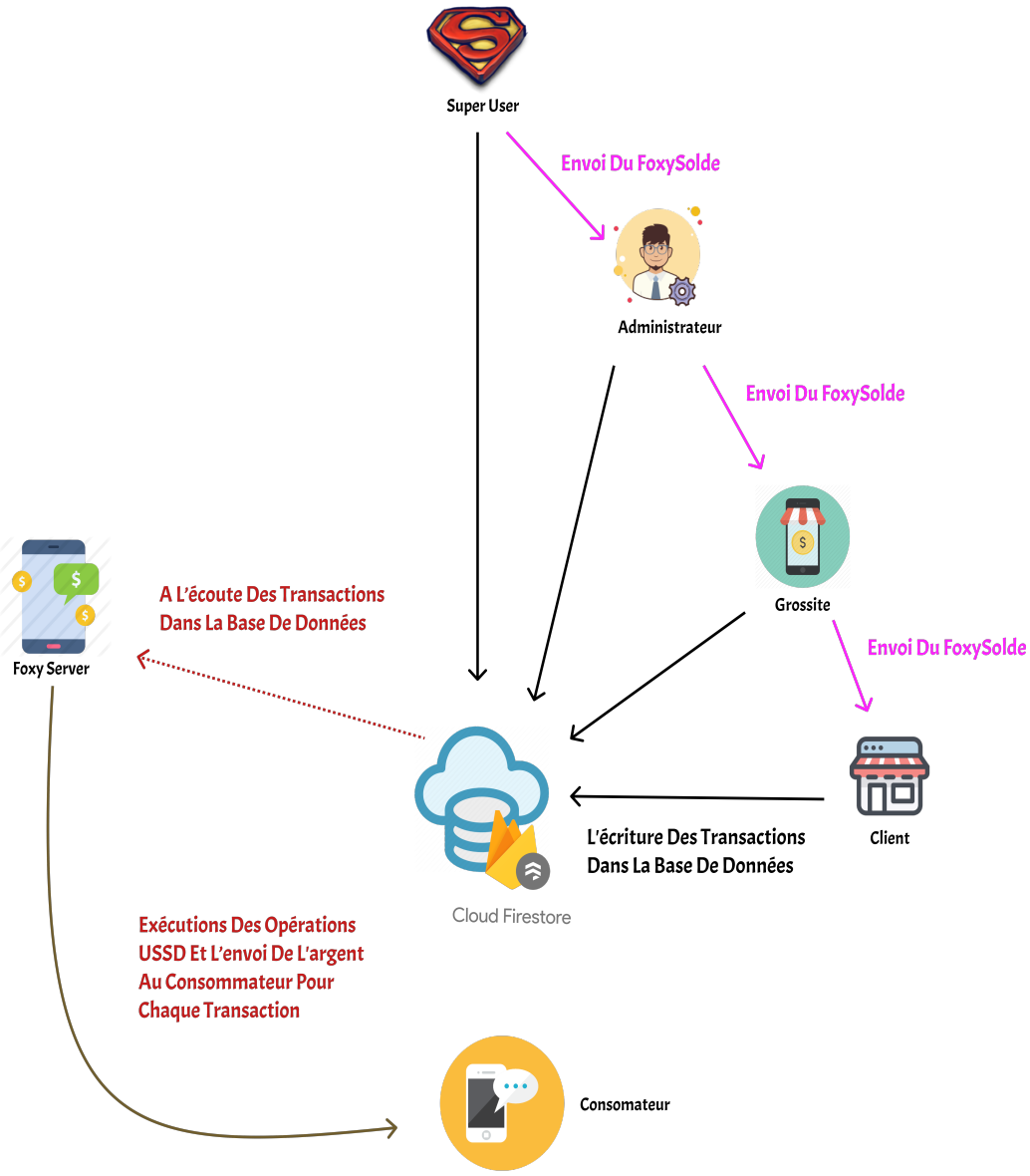


FIGURE 4.1 – Flux de Credit dans l'Application

4.2 Autheentification

Lorsqu'on ouvre l'application pour la première fois, on sera présenté à Login Activity qui nous invite à entrer notre numéro de téléphone comme il est mantré dans la Figure 3.2. Une fois qu'on a entré un numéro de téléphone valide, on appuye sur le bouton de vérification on reçoit un SMS avec un code de confirmation (OTP) à 6

chiffres. La plupart du temps, l'opération de saisie du code de confirmation (OTP) se fait automatiquement, sinon un écran vous demandera de le saisir pour le vérifier, voir Figure 3.3.

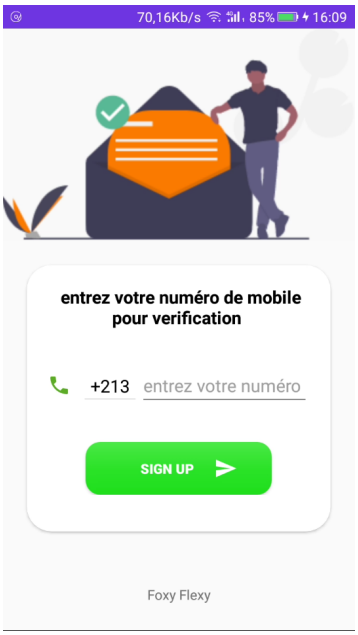


FIGURE 4.2 – Logn Activity

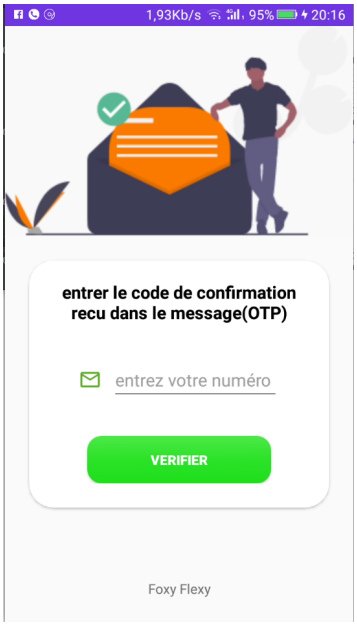


FIGURE 4.3 – Logn Activity (OTP)

Si le numéro de téléphone n'a pas de compte rattaché (première fois) on passe à la deuxième activité qui est Register Activité (figure 3.4), dans cette activité vous êtes invité via un écran où vous devez entrer un nom d'utilisateur, une wilaya et une commune afin de s'inscrire dans la base de données.

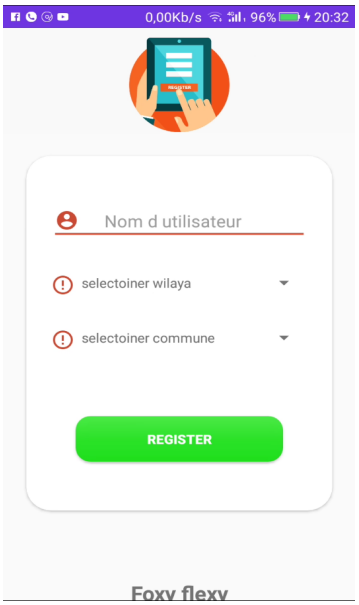


FIGURE 4.4 – Regester Activity

Et si le numéro de téléphone a déjà un compte, l'application récupérera les données du compte, puis elle les lancera avec. Ceci est fait avec l'aide de l'authentification Firebase un exemple d'implémentation est mentionné dans le chapitre 1(section 1.5 page 16).

Pour chaque nouvel enregistrement d'utilisateur, l'application prend un enregistrement dans la base de données avec les informations fournies par l'utilisateur, sous la forme d'un document dans une collection appelée `users`.
La Classe d'utilisateurs est indiquée dans la figure 3.5.
FireStore a besoin d'un constructeur sans paramètre, c'est pourquoi on a inclus un, le rôle de cette classe est simplement de contenir les données utilisateur, afin que nous puissions les écrire et les lire facilement à partir de la base de données FireStore.

```
6   data class User(
7       var admin :Boolean = false,
8       var phoneNumber: String,
9       var name: String,
10      var wilaya: String,
11      var commune:String,
12      var soled_foxy : Long,
13      var soled_mobilis : Long,
14      var soled_djezzy : Long,
15      var soled_ooredoo : Long,
16      var myAdmin :String,
17      var myGros :String,
18      var gros:Boolean = false ) {
19
20      constructor() : this( admin: false ,
21                            phoneNumber: "",
22                            name: "",
23                            wilaya: "",
24                            commune: "",
25                            soled_foxy: 0,
26                            soled_mobilis: 0,
27                            soled_djezzy: 0,
28                            soled_ooredoo: 0,
29                            myAdmin: "",
30                            myGros: "",
31                            gros: false)
32  }
```

FIGURE 4.5 – User Class

4.3 Profils

A partir de maintenant, toute l'application est une seule activité, cela est possible avec l'aide du composant de navigation de (Jetpack) toutes les captures d'écran qui suivront sont des fragments montés sur la même activité.
Quand un utilisateur authentifié ouvre l'application il est présenté au **MainFragment** qui, en fonction de son objet d'authentification, l'envoie à l'un des **HomeFragments**.

FIGURE 4.6 – MainFragment

Il y a un **HomeFragment** pour chaque profil

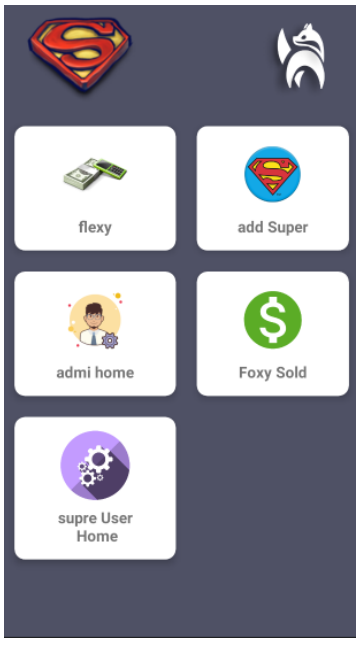


FIGURE 4.7 – SuperUser HomeFragment

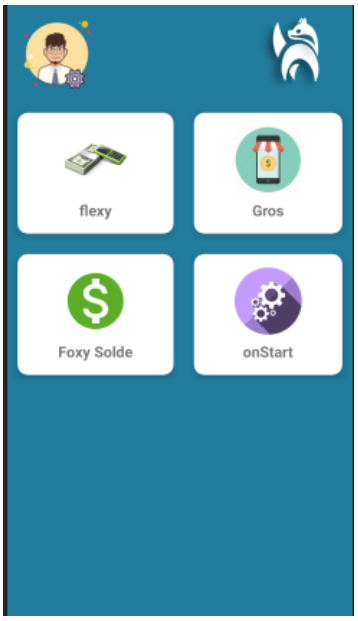


FIGURE 4.8 – Admin HomeFragment

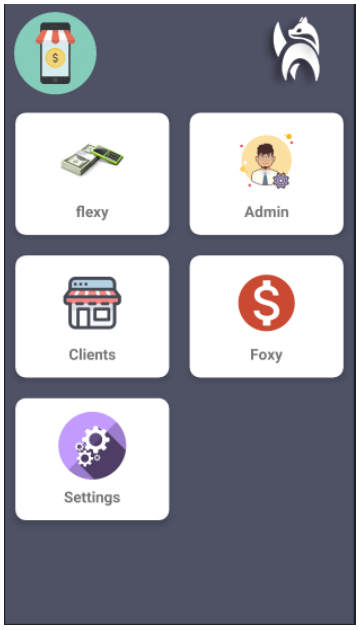


FIGURE 4.9 – Grossite HomeFragment

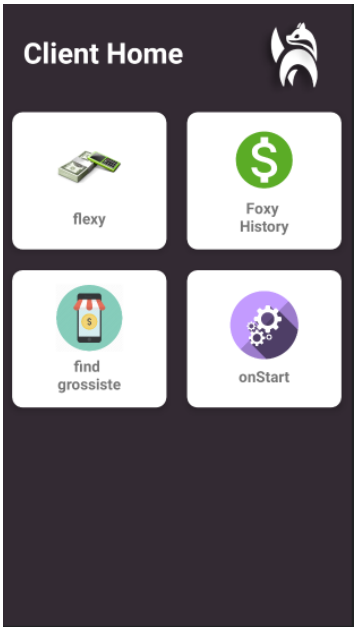


FIGURE 4.10 – Client HomeFragment

Afin de contrôler l'accès, on a utilisé **Firebase Custom Claims** qui nous a donné la possibilité d'attacher un objet au **jeton d'authentification** de l'utilisateur comme il est expliqué dans la figure 3.11.

```
exports.addAdminRole = functions.https.onCall((data, context) => {

  // cette fonction prend un Numéro De Téléphone comme entrée
  return admin.auth().getUserByPhoneNumber(data.PhoneNumber).then(user => {
    // attache l'objet {admin: true} au jeton d'authentification de l'utilisateur
    return admin.auth().setCustomUserClaims(user.uid, {
      admin: true
    }).then(()=>{
    // mettre à jour la base de données avec les modifications
     return admin.firestore().collection("users").doc(user.uid).update({
       admin:true
     })
    })
  }).then(() => {
    // renvoie un message si tout fonctionne bien
    return {
      message: `Success! ${data.PhoneNumber} has been made an admin.`
    }
  }).catch(err => {
    // ou une erreur en cas de problème
    return err;
  });
});
```

FIGURE 4.11 – addAdminRole Function

La figure 3.11 est une **Firebase Function** écrite en JavaScript. Ensuite on peut vérifier ces objets pour envoyer chaque utilisateur à son propre HomeFragment. L'idée est que le **SuperUser** puisse ajouter des **Administrateurs** qui achèteront le **Foxy-Soled** afin de le revendre aux **Grossistes**.

Les **Administrateurs** peuvent également ajouter des **Grossistes** subordonnés afin qu'ils puissent acheter FoxySold chez eux et le revendre aux Clients.

— le grossiste ne peut acheter FoxySold qu'auprès de l'administrateur qui l'a ajouté,
— un administrateur peut avoir de nombreux grossistes subaltarnes,
— un grossiste ne peut avoir qu'un seul administrateur.

Lorsque les grossistes ont le FoxySoled, tous les Clients peuvent acheter FoxySoled de n'importe quel Grossiste.

Chaque profil peut envoyer un crédit de téléphone mobile en écrivant un document contenant les informations de transaction dans une collection appelée "**TransactionsHolder**", où le Serveur peut écouter les changements dans la collection "**TransactionsHolder**" et exécuter l'opération de transfert de crédit pour chaque transaction. La figure 3.12 est un modèle de transaction.

```
data class Transaction(

    @ServerTimestamp var created: Timestamp? = Timestamp.now(),

    var sendTo: String,
    var senderUid:String? = FirebaseAuth.getInstance().currentUser?.uid,
    var senderPhoneNumber:String? = FirebaseAuth.getInstance()
                                    .currentUser?.phoneNumber?.displayPhoneNumber(),
    var old_soled: Long,
    var new_soled: Long?,
    var amount: Long,
    var operator:String,
    var type: String,
    var option:String    ) {
constructor(): this( created: null,
    sendTo: "",
    senderUid: "",
    senderPhoneNumber: "",
    old_soled: 0,
    new_soled: null,
    amount: 0,
    operator: "",
    type: "",
    option: "")

}
```

FIGURE 4.12 – Transaction Class

La figure 3.13 montre la relation entre les profils.

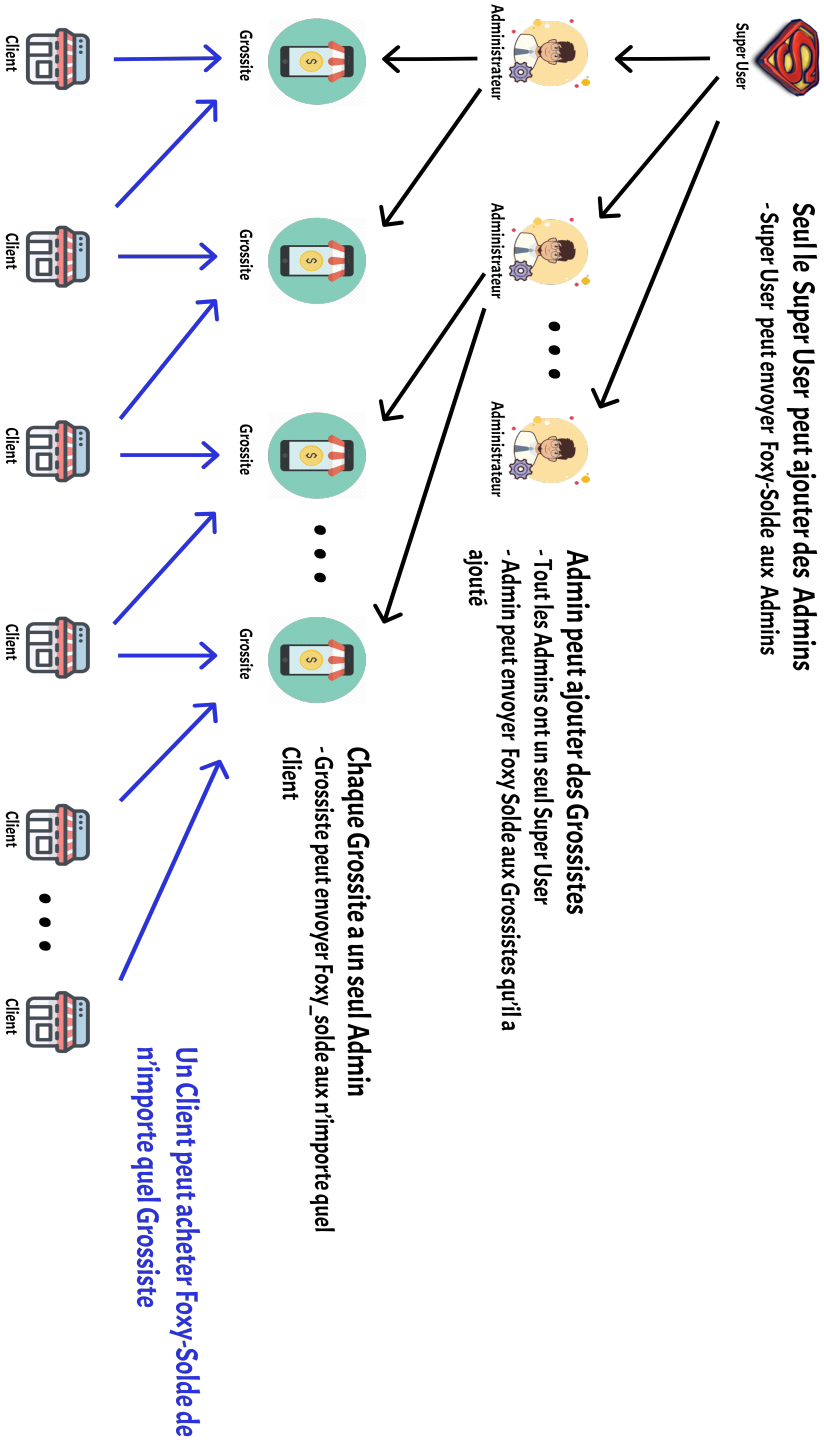


FIGURE 4.13 – Relation Entre Les Profils

Jetons maintenant un coup d'œil à chaque profil pour voir la fonctionnalité de chacun d'entre-eux.

4.3.1 Client

Le Client est le premier palier de l'application. Tout compte nouvellement créé est un **Client par defaut, authentifié par :**

— Son numéro de télephone,
— Nom d'utilisteur,
— Wilaya de résidence,
— Commune de résidence,
— Une propriété `isAdmin` initialisée à `false`
— Une propriété `isGros` initialisée à `false`
— Une propriété `soled_foxy` initialisée à `0.0`

Il peut effectuer les opérations suivantes :

— Executer des opérations de *"Flexy"*
— Consulter la liste des **Grossistes**,
— Consulter l'historique des opérations *"Flexy"* qu'il a éxecutées,
— Consulter l'historique des transactions *"Foxy"*reçues des différents grossistes.

1. éxecuter et consulter l'historique des opérations de *"Flexy"*

Premièrement, *Flexy Componet* est un composant séparé auquel tous les profils peuvent accéder, d'où ils peuvent envoyer du crédit de téléphone mobile, il est divisé en trois fragments

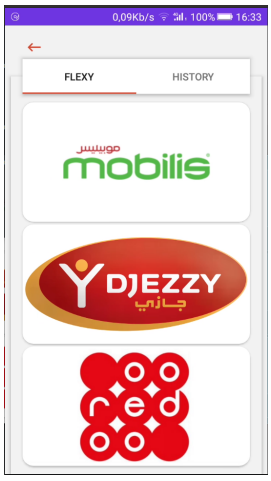


FIGURE 4.14 – Flexy Dashboard Fragment

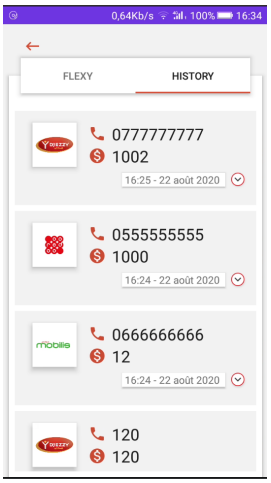


FIGURE 4.15 – Flexy History Fragment

A partir du DashboardFragment (Figure 3.14) l'utilisateur peut accéder à OperatorFragment (Figure 3.16 ; 3.17 ; 3.18) en fonction du bouton préssé l'utilisateur est

envoyé à l'opérateur correspondante.

FIGURE 4.16 – djezzy

FIGURE 4.17 – mobilis

FIGURE 4.18 – ooredoo

Où il peut entrer un numéro de téléphone et un montant et envoyer ce montant sous forme de crédit de téléphone mobile. L'opération *Flexy* n'est qu'un simple document écrit dans la collection **Transactions** (Figure 3.20).

```
fun flexy( consumerPhoneNumber : String,
           amount : Long,
           old_soled :Long,
           option : String) {

    viewModelScope.launch { this: CoroutineScope
            withContext(Dispatchers.IO){ this: CoroutineScope
                addTransaction(consumerPhoneNumber,
                    amount,
                    old_soled,
                    Operator.operator,
                    option)
            }

    }
}
```

FIGURE 4.19 – Flexy Function

```
private suspend fun addTransaction (consumerPhoneNumber : String,
                                    amount : Long,
                                    old_soled :Long,
                                    operator : String,
                                    option : String) {

    repository.db.collection( collectionPath: "transaction")
        .add(Transaction(
            sendTo = consumerPhoneNumber,
                        amount = amount,
                        old_soled = old_soled,
                        new_soled = null,
                        operator = operator,
                        type = "flexy",
                        option = option
        ))

}
```

FIGURE 4.20 – addTransaction Function.png

2. Consulter la liste des Grossistes

Un **Client** peut trouver des **Grossistes** à proximité (dans sa propre Wilaya). L'application fournit le nom et le numéro de téléphone du**Grossiste** dans le but de le contacter.

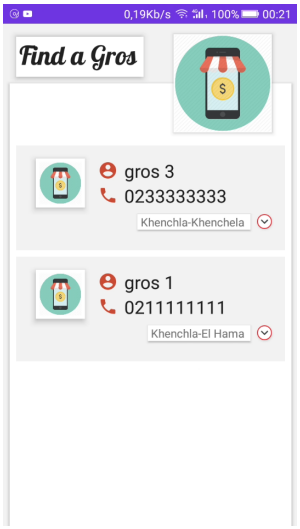


FIGURE 4.21 – List des Grossite

A cette fin, on a un RecyclerView avec les documents de la Collection **users** où le

champ "`isGros` = true" et le champ "`Wilaya` = la Wilaya de l'utilisateur actuel".

```
fun fetchGosInClientList(recyclerView: RecyclerView) {
    viewModelScope.launch { this: CoroutineScope
        val usersRef = repository.db.collection( collectionPath: "/users")

        withContext(Dispatchers.IO) { this: CoroutineScope
            usersRef
                .whereEqualTo( field: "gros", value: true)
                .whereEqualTo( field: "wilaya", getCurrentUserWilaya())
                .addSnapshotListener { querySnapshot, firebaseFirestoreException ->
```

FIGURE 4.22 – Gors query

3. consulter l'historique des transactions *"Foxy"*reçues des grossistes

Les transactions Foxy sont un peu compliquées que la transaction Flexy, car nous devons garder une trace à la fois de l'expéditeur et du destinataire afin de rendre la requête plus simple, de sorte que la transaction Foxy doit contenir l'ID ainsi que les nouveaux et anciens soldes de l'expéditeur et du destinataire plus le montant, l'heure et la date de création de la transaction.

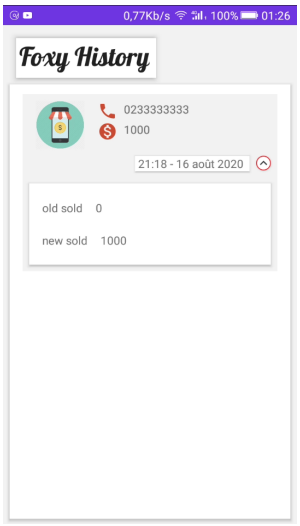


FIGURE 4.23 – Historique de réception de FoxySoled

Une fois cela est fait, on pourrait récupérer l'historique dans le profil du Client car on peut filtrer les transactions qui contiennent son ID, et de même fonctionne avec le profil de grossiste.

4.3.2 Grossiste

L'Admin peut transformer un compte **Client** en compte **Grossiste**, en ajoutant :
- une propriété `myAdmin` contenant l'ID de l'**Admin** qu'il l'a converti en **Grossiste**
- mise à jour de la propriété `isGros` en `true`

Le **Grossiste** est le maillon intermédiaire entre le **Client** et l'**Admin**, il peut effectuer les opérations suivantes :
- éxecuter des opérations de *"Flexy"*,
- envoyer du Solde Foxy à n'importe quel Client,
- consulter l'historique des opération *"Flexy"* qu'il a éxecutées(envoyer vers les **Consommateurs**),
- consulter l'historique des transactions *"Foxy"* qu'il a éxecutées(envoyer vers les **Clients**),
- consulter l'historique des transaction Foxy reçues de son propre **Admin**.

1. éxecuter et consulter l'historique des opérations de *"Flexy"*

Le Grossiste peut accéder au même composant *Flexy Componet* déjà mentionné dans le profil Client, il peut bénéficier de sa fonctionnalité de la même manière que le **Client**.

2. éxecuter et consulter l'historique des Transactions *"FoxySoled"*

```
private suspend fun isClient(docRef: DocumentReference): Boolean {

    val isClient = viewModelScope.async { this: CoroutineScope
        val isAdmin = async { this: CoroutineScope
            val isAdmin = docRef.get().await().data?.get("admin")
            isAdmin as Boolean ^async
        }

        val isGros = async { this: CoroutineScope
            val isGros = docRef.get().await().data?.get("gros")
            isGros as Boolean ^async
        }
        isGros.await() || isAdmin.await() ^async
    }
```

FIGURE 4.24 – isClient function

Le **Grossiste** peut envoyer FoxySoled uniquement aux **Clients**, cette fonction peut vérifier si le numéro fourni est rattaché à un compte **Client** (Figure 3.24). Une fois ce problème est résolu , il s'agit simplement de mettre à jour le `foxy_soled` du **Client** et du **Grossiste** en soustrayant le montant saisi du **Grossiste** et de l'ajouter au **Client**. Puis conserver une trace de la transaction afin de la récupérer dans l'historique du **Grossiste** et du **Client**.

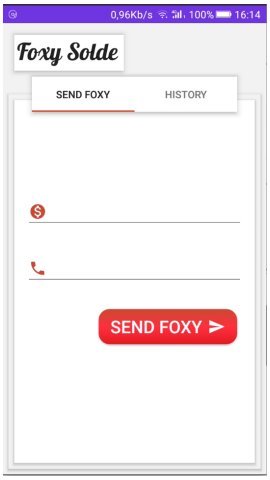


FIGURE 4.25 – Flexy Dashboard Fragment

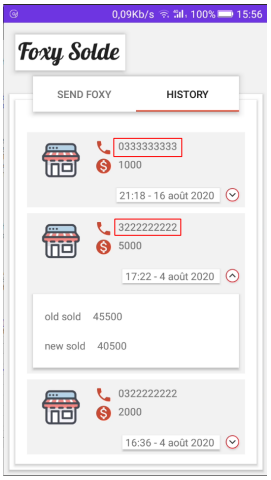


FIGURE 4.26 – Flexy History Fragment

3. Consulter l'historique des transaction *"Foxy"* reçues des grossistes

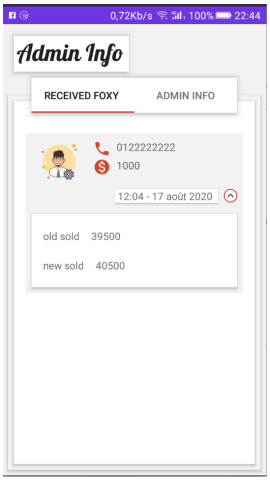


FIGURE 4.27 – Historique de Foxy reçu

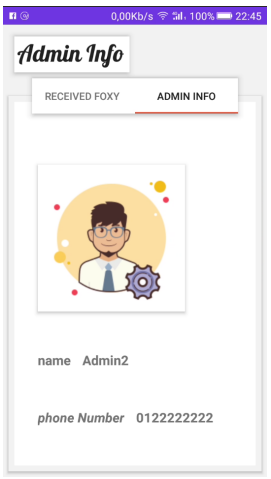


FIGURE 4.28 – Admin Info in Grossite

Avec le même concept, on garde une trace des transactions entre **l'Administrateur** et le **Grossiste** pour pouvoir obtenir l'historique des transactions Foxy dans les deux profils (Admin et Grossiste).

4.3.3 Administrateur (Admin)

Seul le **SuperUser** peut designé des **Admins**, en attachant un **Custom Claime** comme il est montré dans la **figure 3.11** qui ajoute l'objet `{admin: true}` au jeton d'authentification et met à jour le document de l'utilisateur dans la base de données avec les modifications.

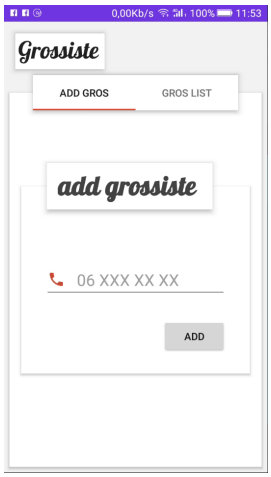
Le rôle **d'Administrateur** est de distribuer le FoxySoled aux **Grossistes**, il peut effectuer les opérations suivantes :

- éxecuter et consulter l'historique des opérations de *"Flexy"* qu'il a éxecutées(envoyer vers les **Consommateurs**),
- ajouter des **Grossiste** auxquels il vendra le FoxySoled (les convertir de **Clients** en **Grossiste**),
- envoyer du FoxySold à ses **Grossistes** et consulter l'historique transactions *"Foxy"* qu'il a éxecutées(envoyer vers ses **Grossistes**).

1. Executer et consulter l'historique des opérations de *"Flexy"*

L'Administrateur peut accéder au même composant *Flexy Componet* déjà mentionné dans le profil **Client**, il peut bénéficier de sa fonctionnalité de la même manière que le **Client**.

2. Ajouter des Grossiste et consulter leur liste



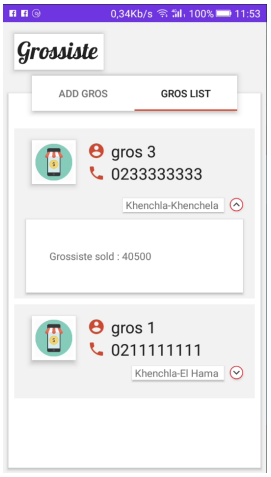


FIGURE 4.29 – Ajouter des Grossiste FIGURE 4.30 – Liste des Gorssistes

Pour obtenir la liste des grossistes de la figure 3.30 on remplit un RecyclerView avec les utilisateurs qui ont le champ `myAdmin`= ID de l'utilisateur actuel (l'Administrateur).

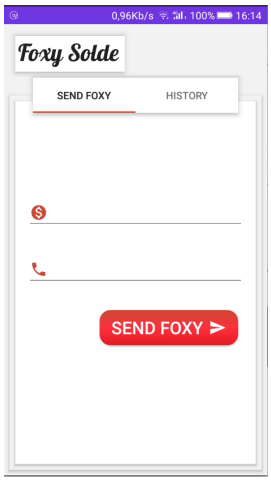
```
val usersRef = repository.db.collection( collectionPath: "/users")

    usersRef
        .whereEqualTo( field: "gros", value: true)
        .whereEqualTo( field: "myAdmin",repository.currentUserUid)
        .addSnapshotListener { querySnapshot, firebaseFirestoreException ->
```

FIGURE 4.31 – fetch gors in admin query

La fonction ci-dessus [Figure 3.31] permet de filtrer les documents dans la collection **"users"**

3. Envoyer du Solde Foxy à des Grossistes

En suivant la même logique expliquée dans le profil **Grossiste**, l'écran de la figure 3.33 peut afficher l'historique de la transaction Foxy effectuée par **l'Administrateur**.



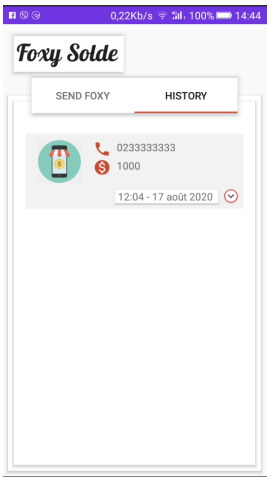


FIGURE 4.32 – Admin Foxy Dashboard FIGURE 4.33 – Admin Foxy History

La seule différence est au lieu de s'assurer que le compte associé au numéro saisi est un Client, on verifie qu'il s'agit d'un Grossiste que l'administrateur actuel l'a ajouté.

```
// vérifier si l'utilisateur est un grossiste ajouté par l'administrateur actuel
private suspend fun userIsGrosOfThisAdmin(uid: String): Boolean {
    println("uid :${uid}")
    val usersRef = repository.firestoreUtil.db.collection( collectionPath: "users").document(uid)

    val isGrosOfTHisAdmin = viewModelScope.async(Dispatchers.IO) { this: CoroutineScope
        val user = async { usersRef.get().await() }
        user.await().data?.get("myAdmin") == repository.firebaseAuth.currentUser?.uid ^async
    }
    return isGrosOfTHisAdmin.await()
}
```

FIGURE 4.34 – userIsGrosOfThisAdmin function

Cela peut être fait à l'aide de la fonction ci-dessus [Figure 3.31].

4.3.4 Fournisseur (SuperUser)

Il n'y a qu'un seul compte SuperUser, destiné à être utilisé par le fournisseur de crédit de téléphonie mobile, il a une propriété `isSuper = true`, il peut effectuer les opérations suivantes :

- éxecuter et consulter l'historique des opérations de *"Flexy"* qu'il a éxecutées(envoyer vers les **Consommateurs**),
- ajouter des **Administrateurs** auxquels il vendra le FoxySoled (les convertir de **Clients** ou **Grossistes** en **Administrateurs**)

— envoyer du FoxySold à ses **Administrateurs** et consulter l'historique transactions *"Foxy"* qu'il a éxecutées (envoyer vers ses **Administrateurs**).

1. Executer et consulter l'historique des opérations de *"Flexy"*

Le Fournisseur peut accéder au même composant *Flexy Componet* déjà mentionné dans le profil **Client**, il peut bénéficier de sa fonctionnalité de la même manière que le **Client**.

2. Ajouter des Grossiste auxquels il vendra le FoxySoled

Pour obtenir la liste des Admin (Figure 3.36) on remplit un RecyclerView avec les utilisateurs qui ont le champ `isAdmin = true`. Et pour ajouter le AdminRole dans la figure 3.35, on utilise la fonction Firebase montrée dans la figure 3.11.

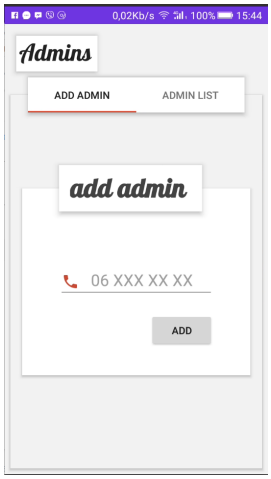


FIGURE 4.35 – Ajouter des Admin

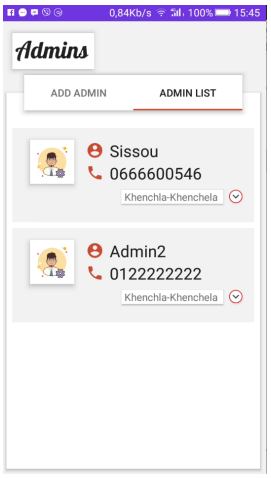


FIGURE 4.36 – List des Admins

3. Envoyer du Solde Foxy à des Administrateurs

En suivant la même logique expliquée dans le profil **Grossiste**, l'écran de la figure 3.36 peut afficher l'historique de la transaction Foxy effectuée par **le Fournisseur**.

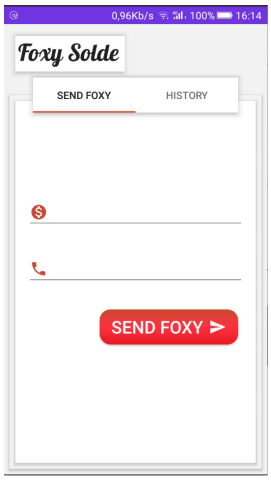


FIGURE 4.37 – Foxy Dashboard

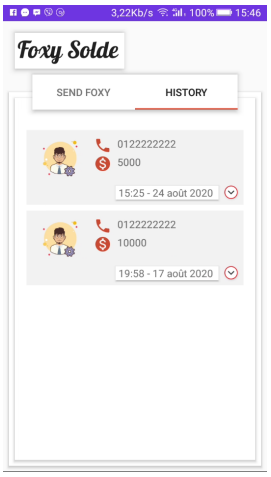


FIGURE 4.38 – SuperUserFoxy History

La seule différence est au lieu de s'assurer que le compte associé au numéro saisi est un client, on verifie qu'il s'agit d'un'Administrateur

```
private suspend fun isAdmin(docRef: DocumentReference): Boolean {

    val isAdmin = viewModelScope.async { this: CoroutineScope

            val isAdmin = docRef.get().await().data?.get("admin")
            isAdmin as Boolean ^async
        }

    return isAdmin.await()
}
```

FIGURE 4.39 – isAdmin function

Cela peut être fait à l'aide de la fonction ci-dessus [Figure 3.31].

4.4 Foxy Server

Foxy Server est une application distincte avec une seule activité, qui peut exécuter des sessions USSD. Pour exécuter l'appel USSD, nous avons besoin de la (permission CALL-PHONE) que nous pouvons obtenir en tapant la ligne ci-dessous dans le fichier Manifest de notre projet.

```
<uses-permission android:name="android.permission.CALL_PHONE"/>
```

FIGURE 4.40 – CALL PHONE permission

Pour communiquer les deux applications, on les a attachées au même projet Firebase, afin qu'elles accèdent à la même base de données FireStore, on a utilisé une collection appelée "TransactionHolder" comme canal de communication, de manière à ce que la première application (Foxy Flexy) écrive des documents contenant les informations sur l'opération Flexy (numéro de téléphone, montant, opérateur et option) et chaque écriture de document déclenche un SnapshotListener dans l'application Foxy-Server, qui génère le code USSD correspondant, exécute l'appel USSD et imprime les informations d'opération sous forme de texte sur l'ecran.

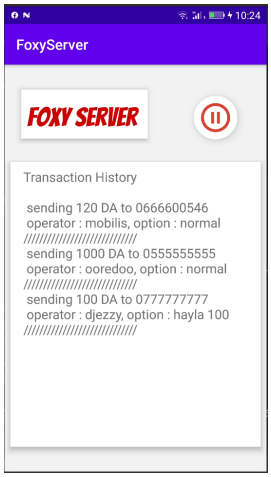


FIGURE 4.41 – Foxy Server

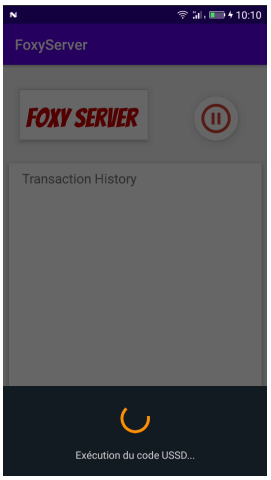


FIGURE 4.42 – run USSD call

4.5 Conclusion

Les fournisseurs avec des applications mobiles professionnelles ont atteint la croissance commerciale souhaitée et augmenté leurs revenus en raison des nombreux avantages concurrentiels dont vous bénéficiez lorsque vous lancez votre propre application mobile professionnelle permettant aux clients actuels et potentiels de s'engager davantage avec votre entreprise.

La plate-forme de marketing, de publicité et de service client est combinée en une seule plate-forme «votre application mobile professionnelle» qui ajoutera une valeur

exceptionnelle lorsque vous concevez, développez et lancez votre application mobile fonctionnelle répondant à vos besoins actuels et potentiels.

Conclusion et perspectives

Le marché national de la telephonie mobile se situe dans une transition digitale qui pousse les fournisseurs de credit mobile à établir des stratégies en ligne ultra concurrentielles et ultra flexibles, une application en mesure de gérer et de faciliter les transaction de tout les opérateurs éxistants est une exigence du marché nationale. La fluidité du crédit mobile du founisseur au consomateur est en perpetuelle amélireation.

Dans notre projet de fin d'étude nous avons conçu et réalisé une application baptisé FOXY FLEXY : l'application assure la fuidilé du credit mobile avec la fiabilité et dans les délais exigés par le consomateur ; l'application offre à ces utilisateurs la possibilité d'acheter de vendre le crédit mobile en tout temps seulement on appuyant sur une seule touche.

Le mémoire mentionne toutes les étapes traversées pour arriver au résultat attendu. Il a fallu dans un premier temps recenser les différents besoins éxistants, nous avons pu aussi donner un contexte général à notre projet et identifier les différentes exigences de la future application. Ce travail nous a été très formateur, puisqu'il nous a permis de découvrir une nouvelle technologie innovante, et nous a permis également relever plusieurs défis et remedier aux contraintes rencontrées : contraintes de temps, contraintes d'expérience et de technologie, surtout sur le plan developpement. En outre, ce projet nous a permis d'approfondir nos connaissances dans les bonnes pratiques de l'ingénierie génie logiciel.

Le présent travail peut être étendu pour la sécurisation des données, optimisation du code pour reduire le coût des services Firebase, et l'automatisation des calculs des revenus.

Bibliographie

[mobileAppsTypes] Traversy Media. (2017). Mobile Apps- Web vs. Native vs. Hybrid. https ://www.youtube.com/watch ?v=ZikVtdopsfY.

[mobileAppIntro] Sendian Creations. (2020). The Best introduction to Mobile Application Development. https ://www.sendiancreations.com/mobile-app-development/.

[FirebaseDocs/auth] Firebase team. (2020). Authentification Firebase . https ://firebase.google.com/docs/auth/.

[FirebaseDocs/irestore/data-model] Firebase team. (2020). Cloud Firestore Data model . https ://firebase.google.com/docs/android/setup.

[FirebaseDocs/android/setup] Firebase team. (2020). Ajoutez Firebase à votre projet Android .https ://firebase.google.com/docs/firestore/data-model

[Full-Stack Firebase] Chris Esplin a Google Developer Expert : Firebase. (7/2018). Full-Stack Firebase. https ://www.udemy.com/course/full-stack-firebase/.

[stevenson2018firebase] Stevenson, Doug. (2018). What is Firebase. The complete story, abridged. Diambil kembali dari https ://medium. com/firebasedevelopers/what-is-firebase-the-complete-story-abridged-bcc730c5f2c0.