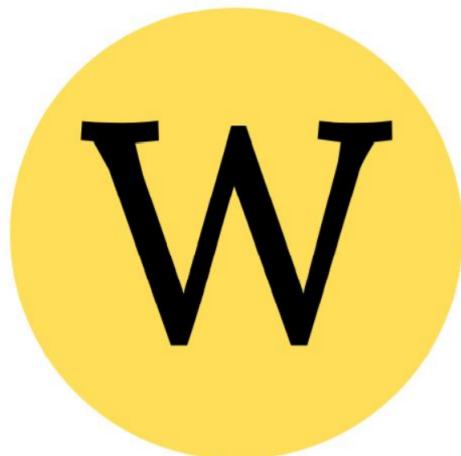


APPLICATION MOBILE

-

“ WHERE ”



CLAIN FLORIAN

-

GRONDIN LUDOVIC

L3 Info

Année scolaire 2019-2020



# SOMMAIRE

<b>1</b>	<b>Introduction</b>	<b>3</b>
1.1	Technologies et outils utilisés . . . . .	3
<b>2</b>	<b>Présentation de l'application</b>	<b>4</b>
2.1	Fonctionnalités et Interface graphique . . . . .	4
<b>3</b>	<b>Quelques points intéressants</b>	<b>11</b>
3.1	Le stockage avec Cloud Firestore et Firebase Storage [4] . . . . .	11
3.2	La persistance des données . . . . .	16
<b>4</b>	<b>Les principaux challenges rencontrés</b>	<b>21</b>
4.1	Les Custom Cell . . . . .	21
4.2	Code asynchrone . . . . .	22
<b>5</b>	<b>Partie ANDROID</b>	<b>24</b>
5.1	Présentation . . . . .	24
5.2	Stockage des données . . . . .	27
5.3	Difficulté rencontrée . . . . .	28
<b>6</b>	<b>Conclusion</b>	<b>29</b>
6.1	Les points à améliorer . . . . .	29
<b>7</b>	<b>Ressources</b>	<b>29</b>
<b>8</b>	<b>Annexes</b>	<b>30</b>

Avant de commencer : Toutes les images utilisées (sauf celles des villes de la Réunion) sont des images libre de droits téléchargées sur Pixabay.

# Résumé

## Vue globale et but de l'application

En tenant compte des différentes exigences requises et notées dans le document de CCTP, nous avons décidé de développer une application dont le but principal serait de recenser les inventaires des magasins puis d'afficher ces informations via une interface graphique.

C'est un sujet qui nous a naturellement inspirés car nous nous sommes rendu compte qu'à la Réunion beaucoup de magasins n'avaient pas de visibilité sur Internet (petits artisans, petits magasins en ville par exemple...). Or dans le monde d'aujourd'hui une présence digitale devient quasi indispensable pour la survie et la pérennité d'un business.

L'application permettra à un utilisateur de :

- **rechercher un magasin** : Avoir accès à son adresse, pouvoir visualiser tous ses articles comme s'il était sur place.
- **rechercher directement un article** : Avoir accès aux informations importantes comme son prix, une photo de l'objet, ainsi que le magasin qui le propose à la vente.
- **sauvegarder en favoris** : Magasins et articles pourront être ajoutés aux favoris afin d'avoir un accès direct à la page dans le menu 'favoris' de l'application.  
Ces informations seront **renseignées via une interface web** par les gérants des magasins, les artisans, etc... puis seront **stockées** avant d'être **récupérées et utilisées dans l'application mobile WHERE**

**WHERE** assurera donc une première présence digitale aux entreprises

# 1 Introduction

Notre application sera réalisée sous les plateformes Android et iOS (la principale version se fera sous iOS). Comme annoncé dans le Résumé, le but de l'application est de présenter à l'utilisateur les informations concernant un magasin et ses produits. Ces informations étant renseignées via une interface web, nous avons dû mettre en place différents outils afin d'assurer le transfert des données depuis le web vers l'application mobile.

## 1.1 Technologies et outils utilisés

Nous avons utilisé respectivement les langages Swift [1] et Java pour développer **WHERE** sous iOS et Android.

Pour satisfaire toutes les fonctionnalités de stockage des différentes informations dans l'application, nous avons décidé de travailler avec Firebase. D'une part, **Cloud Firestore** : Une base de données hébergée dans le Cloud, nous permettra de stocker **toutes les informations** nécessaires (explicitées dans la partie Présentation de l'application 2). Ces données seront ensuite récupérées et traitées sur l'appareil iOS.

En se renseignant sur l'enregistrement des images avec CloudFirestore, nous avons lu que les images ne devraient généralement pas être stockées au sein de bases de données.

D'autre part, nous avons donc suivi ces conseils et avons décidé d'utiliser Firebase Storage : Un espace de stockage en ligne dédié qui nous permettra d'enregistrer nos images et autres fichiers si besoin. C'est donc uniquement les download URL de nos images qui seront stockées dans notre base de données Cloud Firestore.

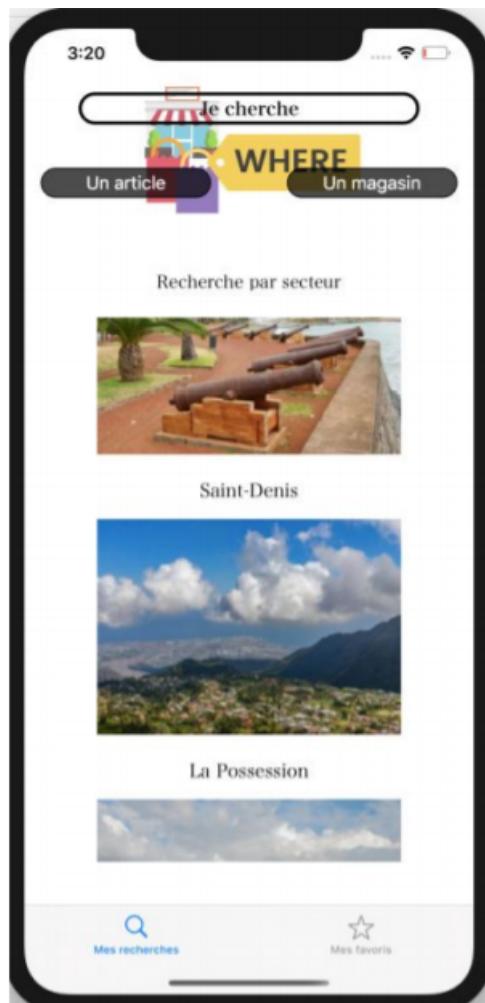
Pour finir, c'est le framework CocoaPods sous ios qui nous ont permis d'obtenir les librairies nécessaires pour une connexion aux outils Firebase (Cloud Firestore et Firebase Storage).(Solution recommandée sur la documentation Firebase)

## 2 Présentation de l'application

### 2.1 Fonctionnalités et Interface graphique

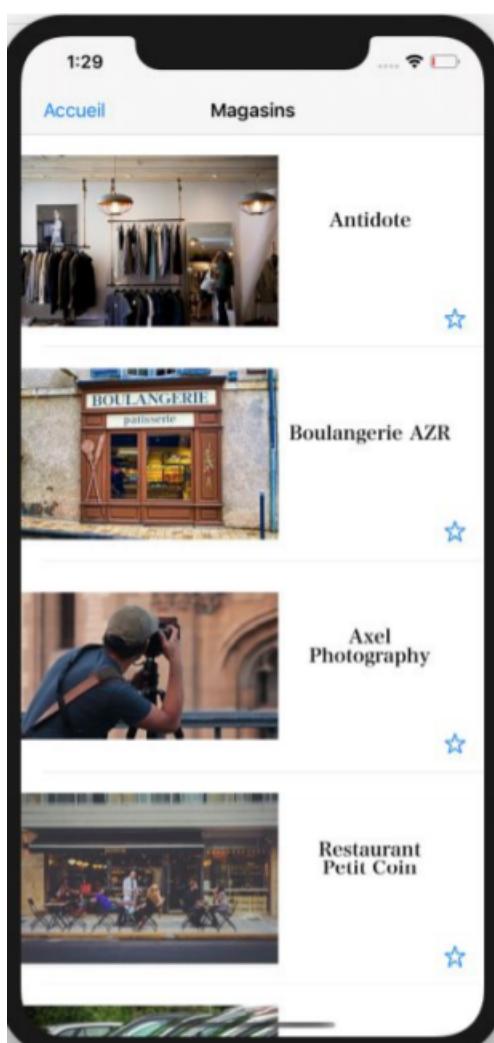
Nous détaillerons ici toute la partie utilisateur : Quelles seront les informations accessibles et comment l'utilisateur pourra y accéder.

Depuis le menu d'accueil, l'utilisateur aura le choix entre deux modes de recherche : une recherche parmi tous les magasins (avec tri par lieux directement possible), ou une recherche parmi tous les articles.



L'utilisateur de **WHERE** se retrouvera ensuite sur une seconde page proposant différentes fonctionnalités en fonction du mode de recherche choisi :

Clic sur ‘Magasin’ depuis l’écran d’accueil : La liste de tous les magasins enregistrés dans l’application est affichée. Si on clique sur un magasin, on obtient des informations détaillées sur celui-ci.

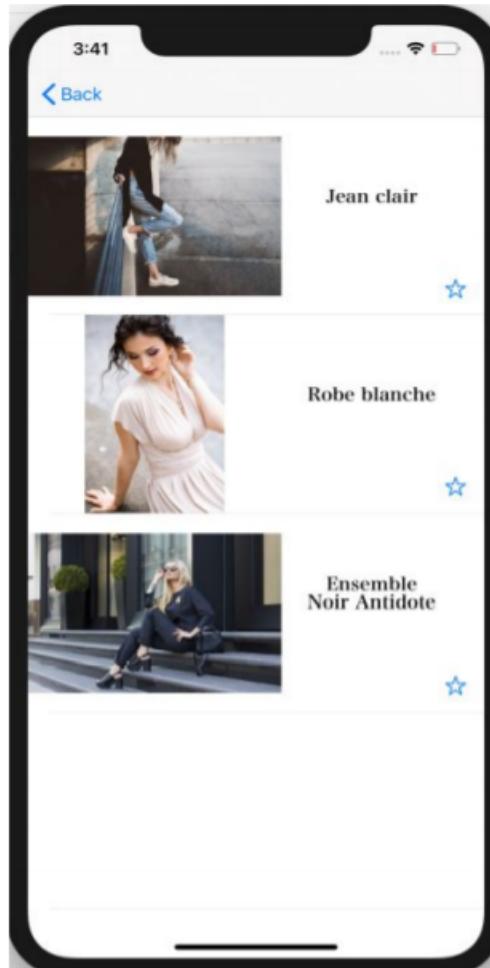


(Page Magasins)



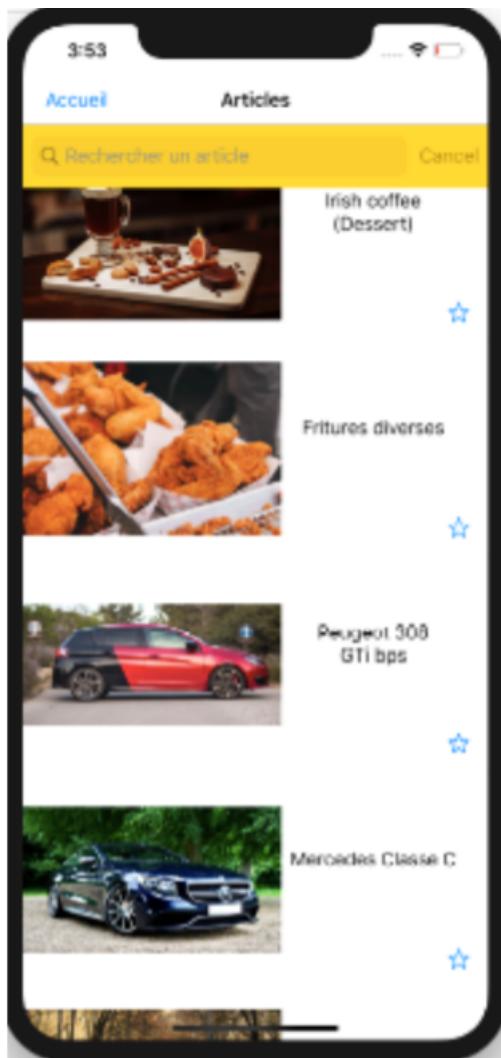
(Clic sur Antidote)

Sur la page affichant les informations du magasin, on peut consulter tous les articles du magasin en appuyant sur le bouton ‘Voir les articles de ce magasin’ en bas à droite de l’écran.



*(Résultat: Articles vendus par Antidote)*

Clic sur ‘Article’ depuis l’écran d’accueil : De la même manière, une liste de tous les articles (tous magasin confondus) est affiché. On peut par la suite obtenir les informations détaillées d’un produit (son prix, ...) en le sélectionnant.



(Page Articles)



(Clic sur Mercedes Classe C)

Une fois sur la page affichant les détails de l'article, on peut accéder à son magasin distributeur en cliquant sur ‘Voir le magasin référent’ en bas à droite.

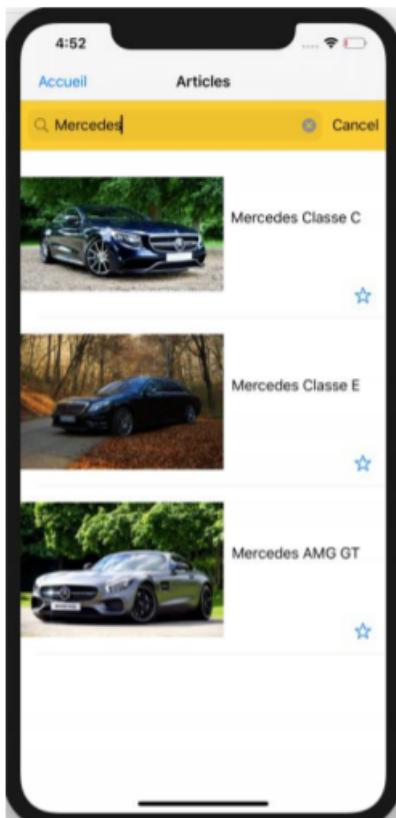


*(Résultat: Magasin/Société qui propose la Mercedes Classe C)*

L'utilisateur peut également ajouter les articles ou magasins en favoris en cliquant sur la petite étoile en bas à gauche. Lorsqu'un article ou magasin fait partie des favoris de l'utilisateur, l'étoile est remplie, sinon, elle est vide (ex : Sur le screen ci-dessus, le magasin CarLoc est en favoris).

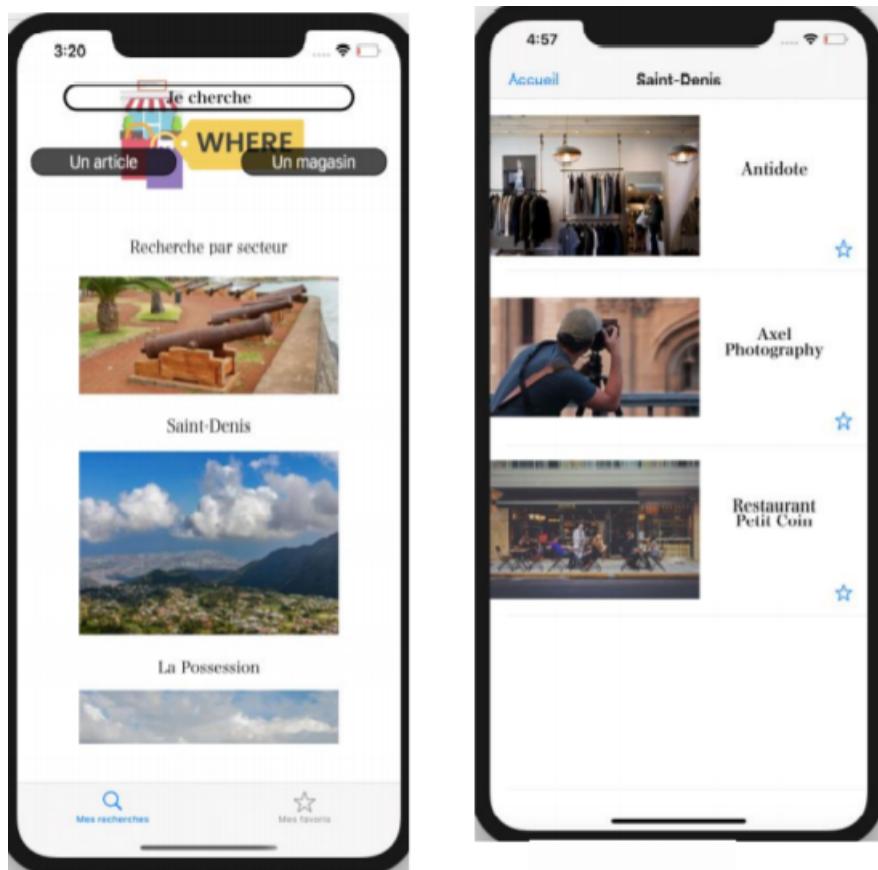
Les Favoris sont enregistrés et peuvent directement être consultés dans l'onglet ‘Favoris’ depuis l'écran d'accueil.

Pour finir, l'utilisateur peut également trier articles et magasins afin de faciliter sa recherche.



Pour les articles, une barre de recherche apparaît sur la page ‘Articles’. C’est une recherche stricte qui tient compte des majuscules et minuscules (ex : sur le screen ci-contre, si la recherche avait été ‘mercedes’ sans M majuscule, aucun produit n’aurait été trouvé)

Pour les magasins, une sélection de lieux est possible directement sur la page d'accueil via le menu 'Recherche par secteur'. Ce menu contient à ce jour les 24 communes de l'île.

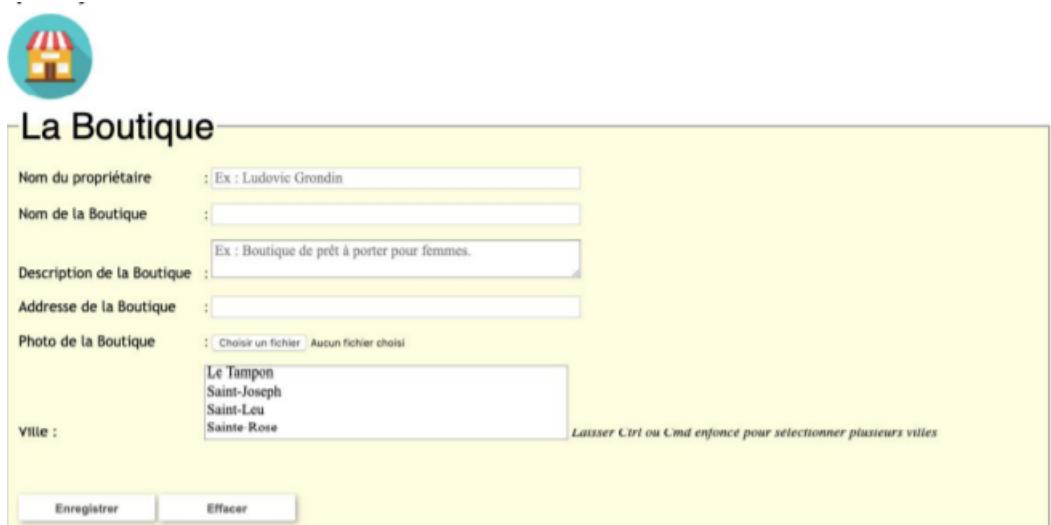


## 3 Quelques points intéressants

### 3.1 Le stockage avec Cloud Firestore et Firebase Storage [4]

Tous les propriétaires peuvent enregistrer leur Magasin via l'interface Web **WHERE**. Afin de rester focaliser sur l'application mobile, l'interface Web sera détaillée au strict minimum : Celle-ci permet l'envoi de toutes les informations dans la base de données Cloud Firestore à travers plusieurs formulaires (Magasin et Produits). Toutes les images envoyées via les formulaires sont stockées dans Firebase Storage, puis c'est leur URL qui est insérée dans la base de données.

Aperçu de l'interface :



**La Boutique**

Nom du propriétaire : Ex : Ludovic Grondin

Nom de la Boutique :

Description de la Boutique : Ex : Boutique de prêt à porter pour femmes.

Adresse de la Boutique :

Photo de la Boutique : Choisir un fichier Aucun fichier choisi

Ville : Le Tampon  
Saint-Joseph  
Saint-Leu  
Sainte Rose Laisser Ctrl ou Cmd enfonce pour sélectionner plusieurs villes

Enregistrer      Annuler



**Les produits**

Nom du produit : Ex : Jantes VW 15 pouces

Description du produit : Ex : Jantes tôles de base 15 pouces avec garantie ...

Prix :

Photo du produit : Choisir un fichier Aucun fichier choisi

Le produit est-il actuellement en stock ET disponible en magasin ?

Oui  
 Non

Ajouter un produit      Annuler

Depuis notre application mobile, nous récupérerons donc toutes les informations directement dans la base de données Cloud Firestore. (info : Cette base de données NoSql utilise des Collections qui contiennent des Documents. Ces Documents contiennent les informations enregistrées sous des paires “clé” :valeur)

Au sein de notre base de données, nous avons décidé d’organiser simplement les choses afin de pouvoir prendre en main Cloud Firestore le plus rapidement possible.

Lorsqu’un utilisateur envoie ses informations depuis notre interface Web, ses données sont dynamiquement enregistrées :

- **Une Collection** est créée. Celle-ci porte le nom de l’utilisateur
- **Deux Documents** : ‘shop’ et ‘products’ sont créés contenant les informations du magasin. Le Document ‘products’ contient **une liste de tous les produits**.  
(voir 8, Schéma d’une Collection utilisateur)

Il nous a ensuite fallu un moyen pour accéder à chacune des Collections utilisateur séparément. Pour ce faire nous avons décidé de créer un tableau contenant le nom de chaque Collection : Dès qu’une Collection est créée, son nom est instantanément ajouté à notre tableau.

(Voir 8, Schéma complet de la base de données)

Dans notre code, nous pouvons donc boucler sur ce tableau afin d’obtenir le nom de chaque Collection. Nous effectuerons ainsi à chaque tour de boucle :

- Une requête afin d'accéder à une Collection
- - Une requête permettant de récupérer les informations du Document ‘shop’ de la Collection. Nous créons ensuite notre objet Shop à partir de ces informations
- Une requête permettant de récupérer les informations du Document ‘products’ de la Collection. En bouclant sur la liste des produits, nous créons nos Objets Product et les associons au Shop précédemment créé.

Voici comment nous avons mis cela en oeuvre en Swift :

Tout d’abord comme annoncé précédemment, Cloud Firestore stock les informations sous forme de paires “clé” :valeur. En Swift, la clé reste une chaîne de caractères et la valeur

est de type ‘Any’. Il nous faut donc transformer chaque variable pour avoir le type attendu avant de l’utiliser. C’est ce que nous avons fait avec l’utilisation de ‘as ? type’ lors de nos récupérations de valeurs dans la base de données.

```
self.db!.collection("usersname").document("names").getDocument { (snapshot, error) in

    guard error == nil,
        let snapshot = snapshot, snapshot.exists,
        let usersname = snapshot.get("names") as? [String]
    else { print("error setupDb"); return }

        for user in usersname {
            //get the current user's shop
            self.db!.collection(user).document("shop").getDocument { (snapshot, error) in
                /*
                    * Récupération des informations du shop de l'utilisateur
                */

                //get the current user's products
                self.db!.collection(user).document("products").getDocument { (snapshot, error) in
                    /*
                        * Récupération des produits
                    */
                }
            }
        }
    }
}
```

Ici nous récupérons le tableau contenant le nom de chaque Collection et l’enregistrons dans la variable ‘usersname’ Ici nous récupérons le tableau contenant le nom de chaque Collection et l’enregistrons dans la variable ‘usersname’

Dans notre base de données, nous avons déclaré la paire “names” :[tableau] (cf : 8). Nous souhaitons récupérer la valeur sous forme de tableau contenant des chaînes de caractères et effectuons donc le cast avec [String] en utilisant snapshot.get(“names”) as ? [String].

Nous utilisons ensuite la boucle ‘for … in’ afin d’accéder à chaque élément du tableau via la variable ‘user’ self.db ! est une référence à notre base de données que nous avons mise en place auparavant.

```
//get the current user's shop
self.db!.collection(user).document("shop").getDocument { (snapshot, error) in
    guard error == nil, let snapshot = snapshot, snapshot.exists,
        let address = snapshot.get("address") as? String,
        let city = snapshot.get("city") as? [String],
        let description = snapshot.get("description") as? String,
        let name = snapshot.get("name") as? String,
        let imageUrl = snapshot.get("image") as? String,
        let downloadUrl = URL(string: imageUrl)
    else { print("error 2nd connexion database - getShop"); return }
    //create shop with empty product list
    let productList = [Product]()
    let shopDatabase = Shop(name: name, description: description, city: city, address: address, isFav: false, productList: productList)
    //Use the downloadUrl to get the image
    self.downloadImage(dlUrl: downloadUrl, object: shopDatabase)
}
```

Ici, nous récupérons toutes les informations du Shop et créons notre objet. Un tableau de produits vide est également créé car nous y associerons les produits à la prochaine requête.

```
//get the current user's products
self.db!.collection(user).document("products").getDocument { (snapshot, error) in
    guard error == nil,
        let snapshot = snapshot, snapshot.exists,
        let productsData = snapshot.get("products") as? Dictionary<String, NSDictionary>
    else { print("error get Products db"); return }

    for secondDictionaryLevel in productsData {
        // for each Product of the document
        var productName = ""
        var productDescription = ""
        var productPrice = 0.0
        var productStock = false
        var productImageUrl = ""
    }
}
```

Nous accédons désormais au Document ‘products’. Cependant sa composition n'est pas aussi simple que le document ‘shop’ qui contenait des paires “clé” : valeur directement accessibles. En effet dans notre Document ‘products’, nous devrons aller chercher les paires “clé” : valeur de nos produits, eux-mêmes dans une liste. En castant ‘productsData’ de la sorte, nous accédons à ce niveau dans notre base de données :



```

for finalValuesDictionary in secondDictionaryLevel.value as! Dictionary<String, Any> {
    // for each [key:value] attribute of the Product
    //print("clé: \(finalValuesDictionary.key), valeur: \(finalValuesDictionary.value)")
    switch(finalValuesDictionary.key) {
        case "name":
            productName = finalValuesDictionary.value as! String

        case "description":
            productDescription = finalValuesDictionary.value as! String

        case "price":
            productPrice = finalValuesDictionary.value as! Double

        case "addedTime":
            print("for sorting")

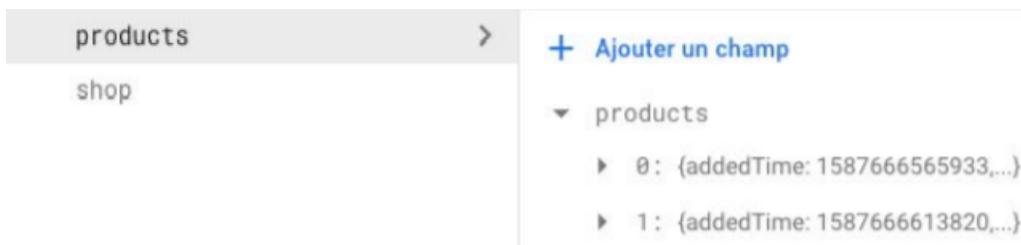
        case "stockInfo":
            productStock = finalValuesDictionary.value as! Bool

        case "image":
            productImageUrl = finalValuesDictionary.value as! String

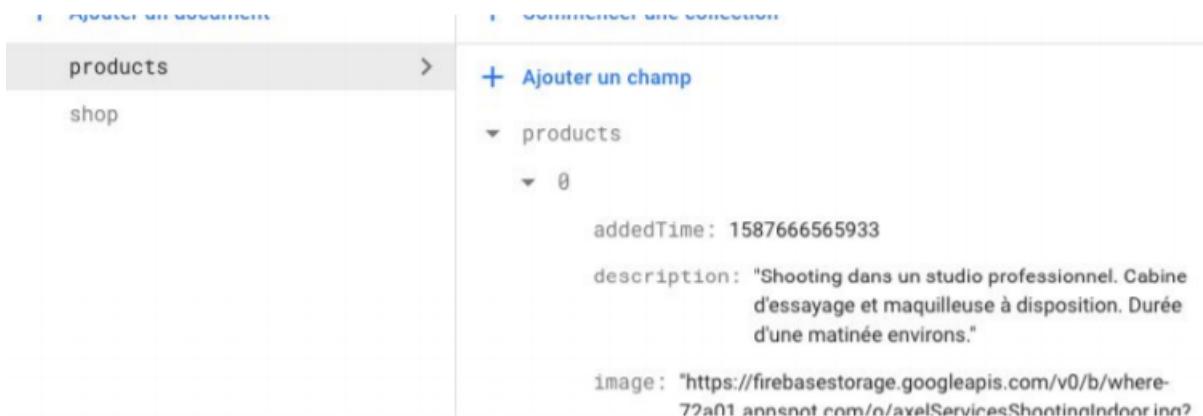
        default:
            printf("error in switch case to product association")
    }
}
//create product & associate to shop
let product = Product(name: productName, description: productDescription, isFav: false, inStock: productStock, price: Float(productPrice), shop: shopDatabase)

```

Le code ci-dessus nous permet d'accéder directement aux paires “clé” :valeur des produits.  
En effet nous parcourons le ‘secondDictionaryLevel’ et accédons à ce niveau :



Comme la “clé” des produits nous importe peu, nous accédons directement à leur valeur avec ‘secondDictionaryLevel.value’



Ainsi nous bouclons sur la liste de produits et leurs valeurs afin de créer nos objets Produit.

```
//create product & associate to shop
let product = Product(name: productName, description: productDescription, isFav: false, inStock: productStock, price:
    Float(productPrice), shop: shopDatabase)
//Set product Image
if productImageUrl != "" {
    self.downloadImage(dlUrl: URL(string: productImageUrl)!, object: product)
} else {
    //set default image if not defined by user
    //product.setImage(newImage: UIImage)
}

shopDatabase.productList.append(product)
}
//append the complete shop to the tab
self.tabOfShops.append(shopDatabase)
//print(self.tabOfShops)
```

Pour finir, lorsque tous les Produits créés sont ajoutés au tableau de produits de l'objet Shop, nous ajoutons ce magasin complet à notre tableau principal que nous utiliserons pour l'application. Le code termine la première boucle et recommence ainsi jusqu'à avoir parcouru toutes les Collections.

Nous avons donc converti chaque élément “clé” :valeur de notre base de données en objet cohérent utilisable dans notre application.

### 3.2 La persistance des données

Lorsque l'utilisateur de l'application enregistre un produit ou un magasin en favoris, celui-ci est sauvegardé en tant que donnée persistante et restera disponible dans la partie ‘Favoris’ même après une fermeture de l'application.

Pour que cela puisse se faire nous avons décidé de créer comme données persistantes de **nouveaux objets ‘Favorites’** servant de **référence** à nos objets **Shop** et **Product** utilisés.

Voici le procédé mis en place pour ajouter un magasin avec CoreData (appui sur l'étoile mettre en favoris) :

```
// MARK: - CoreData

func addCoreData(shop: Shop) {
    let appDelegate = UIApplication.shared.delegate as! AppDelegate
    let context = appDelegate.persistentContainer.viewContext

    let newObject = Favorites(context: context)
    newObject.name = shop.getName()
    newObject.whichType = WhichType.shop.rawValue

    appDelegate.saveContext()
}
```

Pour un magasin, l'objet ‘Favorites’ contient son nom (En effet 2 magasins/sociétés ne peuvent pas avoir le même nom d'après la loi Française, nous pouvons donc l'utiliser comme critère unique)

Ajouter un produit avec CoreData :

```
// MARK: - CoreData

func addCoreData(product: Product) {
    let appDelegate = UIApplication.shared.delegate as! AppDelegate
    let context = appDelegate.persistentContainer.viewContext

    let newObject = Favorites(context: context)
    newObject.name = product.getName()
    newObject.productParent = product.getShop().getName()
    newObject.whichType = WhichType.product.rawValue

    appDelegate.saveContext()
}
```

Pour un produit, l'objet ‘Favorites’ contiendra son nom, mais aussi le nom du magasin référent (car 2 produits peuvent avoir le même nom, nous déterminons donc de quel produit il s’agit en comparant le nom du magasin distributeur).

Afin de savoir si l’objet ‘Favorites’ est un Shop ou Product lorsque nous le re associerons, nous avons également ajouté un attribut **whichType** qui contiendra soit “shop” soit “product”. Celui-ci est décrit dans une **enum** à part.

```
import Foundation

enum WhichType: String {
    case product = "product"
    case shop = "shop"
}
```

Ainsi, pour re associer nos objets ‘Favorites’ nous vérifions de quel type il s’agit grâce à **whichType**. S’il s’agit d’un ‘Shop’, nous parcourons notre tableau et comparons les ‘Shop’ par nom jusqu’à retrouver le ‘Shop’ portant le même nom. S’il s’agit d’un objet, nous parcourons la liste de ‘Product’ de tous nos ‘Shop’ et comparons le nom et le ‘Shop’ référent :

```

func tableView(_ tableView: UITableView, cellForRowAt indexPath: IndexPath) -> UITableViewCell {
    let cell = tableView.dequeueReusableCell(withIdentifier: "favoritesCell", for: indexPath) as! FavoriteTableViewCell
    let object = favoritesTab[indexPath.section]

    cell.nameLabel.text = object.name
    if object.whichType == WhichType.product.rawValue {
        for shop in tabOfShops {
            for product in shop.getProductList() {
                if object.name == product.getName() && object.productParent == shop.getName() {
                    cell.favImgView.image = product.getImage()
                }
            }
        }
    } else if object.whichType == WhichType.shop.rawValue {
        for shop in tabOfShops {
            if object.name == shop.getName() {
                cell.favImgView.image = shop.getImage()
            }
        }
    }
    return cell
}

```

Ici cela nous permet d'afficher sur notre page Favoris les images et les textes correspondants aux objets enregistrés avec CoreData.

Le tableau ‘favoritesTab’ est un tableau d’objets ‘Favorites’ obtenu en récupérant tous les objets enregistrés avec CoreData et en les ajoutant à un tableau. Il nous sert donc de référence dans nos fonctions de tableView (grâce à ce tableau on connaît le nombre de sections à afficher, ...).

```

func fetchCoreData() {
    let appDelegate = UIApplication.shared.delegate as! AppDelegate
    let context = appDelegate.persistentContainer.viewContext

    let request = NSFetchedResultsController(entityName: "Favorites")
    request.fetchBatchSize = 20
    request.returnsObjectsAsFaults = false

    do {
        let results = try context.fetch(request)
        if results.count > 0 {
            // Get saved favorites
            for result in results as! [NSManagedObject] {
                favoritesTab.append(result as! Favorites)
            }
        }
    } catch {
        print("error while fetching CoreData")
    }
    tableView.reloadData()
}

```

Enfin pour supprimer les objets de CoreData (2nd appui sur l'étoile en bas à gauche de la page d'un produit ou d'un magasin), on procède de la manière inverse à lorsqu'ont re associe : En effet lorsqu'ont re associe, on a un 'Favorites' et on souhaite obtenir un 'Shop' ou 'Product'. Lorsqu'on supprime, nous avons le 'Shop' ou 'Product' à supprimer et nous souhaitons supprimer le 'Favorites' correspondant. Nous récupérons donc tous nos CoreData en tant que 'Favorites' et nous comparons pour un Shop son nom, et pour un Product son nom et son magasin référent.

Code pour supprimer un 'Shop' des favoris :

```
func deleteCoreData(shop: Shop) {
    let appDelegate = UIApplication.shared.delegate as! AppDelegate
    let context = appDelegate.persistentContainer.viewContext

    let request = NSFetchedResultsController<NSFetchRequestResult>(entityName: "Favorites")
    request.returnsObjectsAsFaults = false
    do {
        let results = try context.fetch(request)
        if results.count > 0 {
            for result in results as! [NSManagedObject] {
                let favoriteToDelete = result as! Favorites
                if favoriteToDelete.name == shop.getName() {
                    appDelegate.delete(favoriteToDelete)
                    appDelegate.saveContext()
                }
            }
        }
    } catch {
        print("----- Error deleteCoreData function -----")
    }
}
```

Nous effectuons le même code mais avec les comparaisons adaptées pour supprimer un produit.

## 4 Les principaux challenges rencontrés

### 4.1 Les Custom Cell

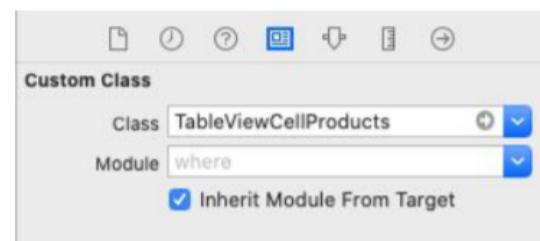
Nous avons eu besoin de créer nos cellules de tableau personnalisées afin de pouvoir obtenir le résultat visuel attendu. Pour ce faire nous avons créé notre cellule via le StoryBoard, puis nous avons relié tous ses éléments aux Outlets de notre classe personnalisée. Nous avons également fait hériter la cellule de notre classe personnalisée.

```
//Table View cell for TableView in SearchProductsTa
import UIKit

class TableViewCellProducts: UITableViewCell {

    @IBOutlet weak var productImg: UIImageView!
    @IBOutlet weak var nameOfProduct: UILabel!
    @IBOutlet weak var addFavButton: UIButton!

    override func awakeFromNib() {
        super.awakeFromNib()
        // Initialization code
    }
}
```



Afin de pouvoir utiliser ces cellules, nous avons caster la cellule de base retournée par la fonction tableView (let cell = ..... as! CustomCellClass).

Nous avons ainsi une référence à notre cellule personnalisée et pouvons l'utiliser.

```
func tableView(_ tableView: UITableView, cellForRowAt indexPath: IndexPath) -> UITableViewCell {
    let cell = tableView.dequeueReusableCell(withIdentifier: "cellProductTableView", for: indexPath) as!
        TableViewCellProducts

    if !isSearching {
        cell.nameOfProduct.text = tabOfShops[indexPath.section].productList[indexPath.row].getName()
        //Set product image or default image
        if let img = tabOfShops[indexPath.section].productList[indexPath.row].image {
            cell.productImg.image = img
        } else {
            cell.productImg.image = #imageLiteral(resourceName: "DefaultImage")
        }
    }
}
```

Ici nous définissons le texte grâce à l'outlet ‘nameOfProduct’ que nous avons défini au moment de la création de notre classe plus haut.

Pendant la création de nos cellules personnalisées, nous avons rencontré un bug de XCode qui s'avérait être connu : Les images ajoutées avec des contraintes width et height fixent occasionnent des messages perturbants dans le terminal.

```

2020-04-20 13:01:22.223512+0400 where[4357:166602]
  [LayoutConstraints] Unable to simultaneously satisfy
  constraints.
  Probably at least one of the constraints in the
  following list is one you don't want.
  Try this:
    (1) look at each constraint and try to figure out
        which you don't expect;
    (2) find the code that added the unwanted constraint
        or constraints and fix it.
(
  "<NSLayoutConstraint:0x600000001f520
    UIImageView:0x7f9e7cdb8c40.height == 190
    (active)>",
  "<NSLayoutConstraint:0x600000001fs70
    V:[UIImageView:0x7f9e7cdb8c40]-(-)-| (active,
    names: '|':UITableViewCellContentView:0x7f9e7cda9ff0
  )>",
  "<NSLayoutConstraint:0x600000001fb60
    V:|-(-)[UIImageView:0x7f9e7cdb8c40] (active,
    names: '|':UITableViewCellContentView:0x7f9e7cda9ff0
  )>",
  "<NSLayoutConstraint:0x6000000020b90
    'UIView-Encapsulated-Layout-Height'
    UITableViewContentView:0x7f9e7cda9ff0.height ==
    190.333 (active)>"
)
Will attempt to recover by breaking constraint
<NSLayoutConstraint:0x600000001f520
  UIImageView:0x7f9e7cdb8c40.height == 190 (active)>

```

Pour résoudre cela, nous avons vu qu'il fallait modifier la priorité de la contrainte à 999 car il y avait un conflit entre les contraintes automatiques des tableView et les nôtres.



(voir8)

## 4.2 Code asynchrone

Lors de notre phase de développement nous avons également été gêné par du code asynchrone : lorsque nous récupérons nos données dans Firestore, nous utilisons le réseau internet, les données n'étaient donc pas directement associées à nos variables et il y avait un petit délai d'attente avant que ces informations arrivent.

Le temps d'attente était quasi négligeable ( 1 seconde), cependant ne pas traiter cette situation aurait pu causer des bugs :

- En effet dans le cas où l'utilisateur, pressé, appuie sur ‘Magasin’ alors que les informations ne sont pas encore arrivées, rien n'est affiché.

- En plus, nos fonctions sont appelées dans la méthode viewDidLoad() et sont donc exécutées qu'une seule fois car nous souhaitons récupérer toutes ces données uniquement au démarrage de l'application. L'utilisateur est donc obligé de redémarrer l'application s'il ne laisse pas le temps aux données d'arriver.
- Enfin, nous avons pensé que le débit d'une connexion internet pouvant varier, ce temps d'attente qui était d'environ 1 seconde au moment des tests pourrait bien augmenter et passer à 5 ou 6 secondes avec un débit très faible ou coupé temporairement.

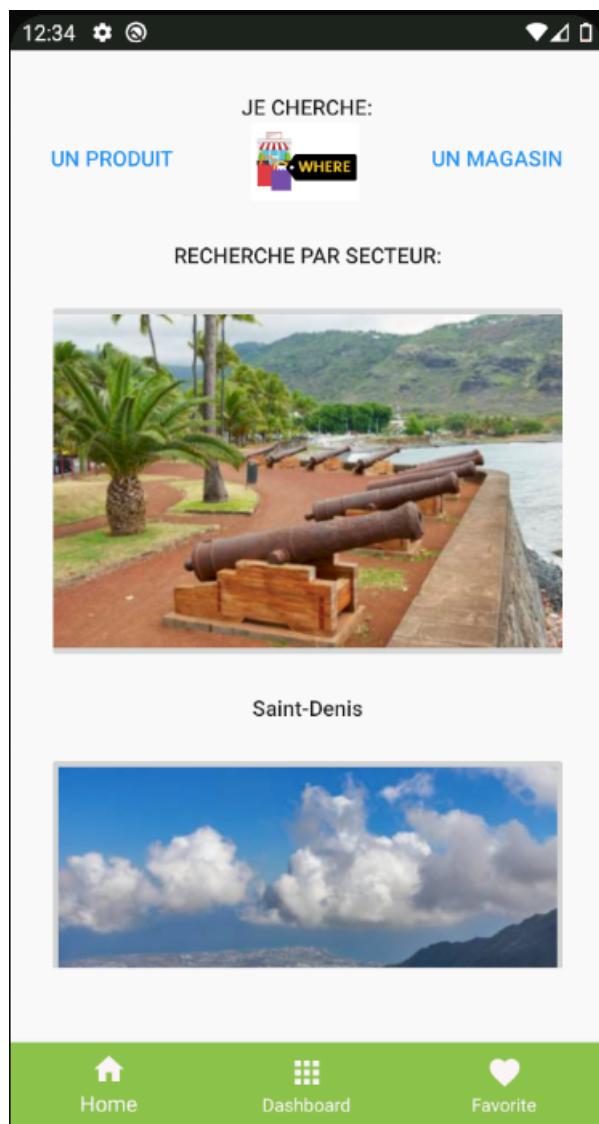
Pour pallier cette faille nous avons simplement ajouté une fonction vérifiant chaque seconde notre tableau principal. Tant que celui-ci est vide une spinner loading est affiché à l'écran. Dès que notre tableau se remplit, la spinner loading est effacée.

```
func fetchingDataLoading() {  
    // Reference to loading spinner  
    let tabBar = tabBarController as! TabBarController  
    spinner = tabBar.spinner  
  
    if tabOfShops.isEmpty {  
        //add loading spinner to VC  
  
        //check when data are available in tab  
        let _ = Timer.scheduledTimer(withTimeInterval: 1.0, repeats: true, block: { timer in  
            if !(self.tabOfShops.isEmpty) {  
                //Remove spinner  
                //print("every sec")  
                self.spinner!.willMove(toParent: nil)  
                self.spinner!.view.removeFromSuperview()  
                self.spinner!.removeFromSuperview()  
  
                self.fetchCoreData()  
                self.setFavAttributesWhenLaunch()  
  
                timer.invalidate()  
            }  
        })  
    }  
}
```

## 5 Partie ANDROID

La version ANDROID est ici une version simplifiée, elle n'a pas de base de données connecter comme IOS. Les informations affichées dans cette application sont à titre d'exemple pour illustrer le fonctionnement de l'application . IOS étant la version principale nous ne détaillerons au minimum ANDROID pour éviter de rendre le rapport trop long d'autant plus que les fonctionnalités sont similaires bien que cette version soit simplifiée.

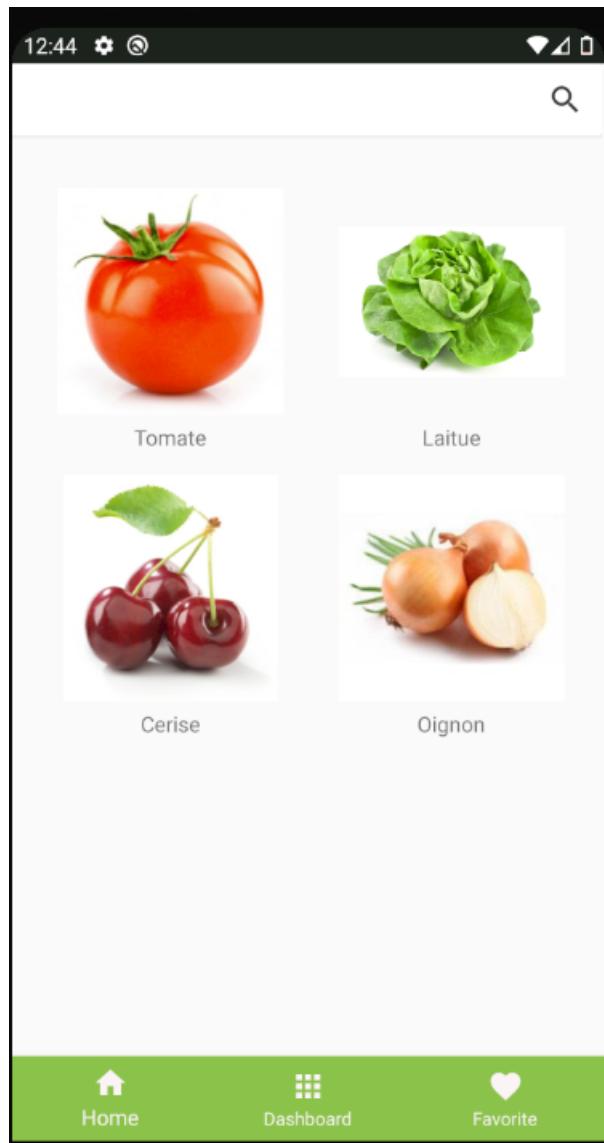
### 5.1 Présentation



Lors de l'ouverture un splash screen comportant le logo de l'application est affiché pendant le chargement qui nous conduit ensuite vers notre écran d'accueil.

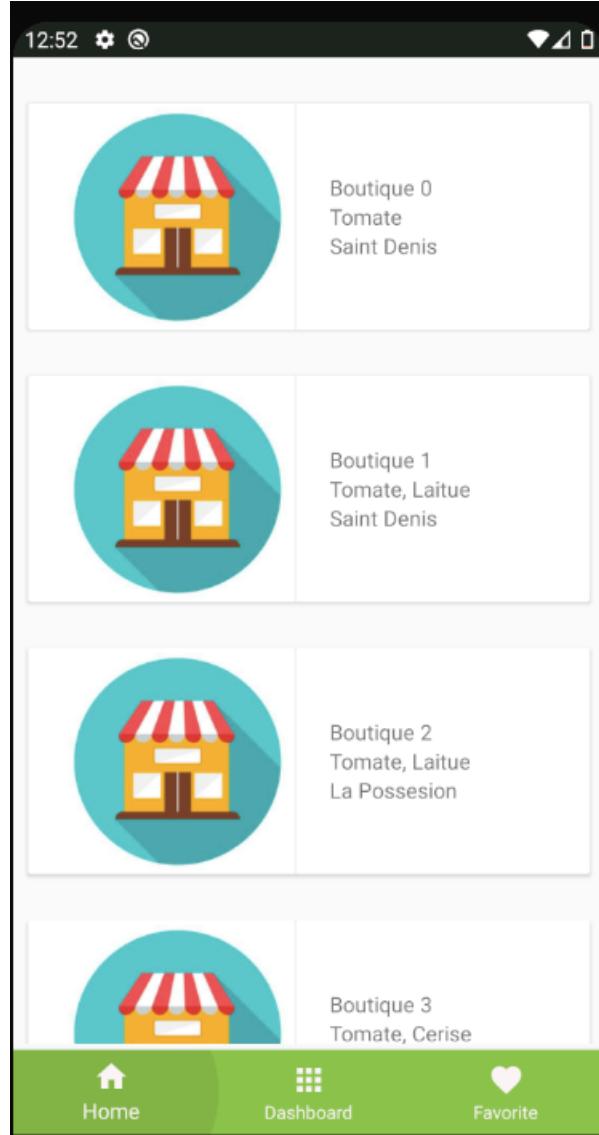
Ici plusieurs choix s'offre à l'utilisateur il peut visualiser tous les produits, visualiser toutes les boutiques disponibles ou encore choisir de voir les boutiques d'une ville particulière selon celle disponible .

Voyons ici l'exemple lors du choix de visualiser les produits, il atterrit donc sur cette vue :



Ici l'utilisateur peut choisir un produit afin de voir les ou le magasin proposant ce produit et ainsi avoir des informations ou le trouver.

La liste des magasins est représentée sous cette forme :



L'utilisateur peut ici ajouter une boutique en favori en effectuant un "long clique" ou voire les produits proposer en cliquant simplement sur le magasin de son choix . Pour accéder aux favoris il suffit de sélectionner l'icône coeur dans la barre de navigation.

## 5.2 Stockage des données

Les données telles que les images de produits et autres information sont stockée dans une base de données Sql local dans l'application.

Ici pour la création de la Base de Données nous avons utilisé DB Browser for SQLite [3]. La Base de Données se compose de trois tables qui sont FAVORI,lboutique et produit.

Nom	Type	Schéma
Tables (4)		
> FAVORI		CREATE TABLE "FAVORI" ( "ID" INTEGER)
> lboutique		CREATE TABLE "lboutique" ( "ID" INTEGER)
> produit		CREATE TABLE "produit" ( "ID" INTEGER)

Voici un exemple d'appel à des informations de l'une des tables :

```
//get all boutique
public List<Boutique> getBoutique(){
    SQLiteDatabase db = getReadableDatabase();
    SQLiteQueryBuilder qb = new SQLiteQueryBuilder();

    String[] sqlSelect = {"ID","NOM","PRODUIT","VILLE"};
    String tableName="lboutique";

    qb.setTables(tableName);
    Cursor c = qb.query(db,sqlSelect, selection: null, selectionArgs: null, groupBy: null, having: null, sortOrder: null);
    List<Boutique> result = new ArrayList<>();
    if (c.moveToFirst()){
        do {
            Boutique boutique = new Boutique();
            boutique.setId(c.getInt(c.getColumnIndex( columnName: "ID")));
            boutique.setName(c.getString(c.getColumnIndex( columnName: "NOM")));
            boutique.setArticle(c.getString(c.getColumnIndex( columnName: "PRODUIT")));
            boutique.setCity(c.getString(c.getColumnIndex( columnName: "VILLE")));
            result.add(boutique);
        }while (c.moveToNext());
    }
    return result;
}
```

### 5.3 Difficulté rencontrée

La principale difficulté rencontrée et qui n'est toujours pas résolue à ce jour est la material search bar [2] pour rechercher les produits par nom. En effet la recherche s'effectue cependant il y avait pas mal de bug. Cependant après quelques tests le problème a pu être résolu, la barre de recherche fonctionne son principe et le même que celui de la version ios (voir 2).

## 6 Conclusion

Avec son choix de sujets libre, ce projet nous a permis de faire preuve de créativité et d'imagination, de découvrir différentes étapes d'un processus de développement d'une application et surtout de pouvoir les réaliser soi-même.

### 6.1 Les points à améliorer

Pour conclure, nous avons réussi à développer notre application avec les fonctionnalités de base que nous attendions. Nous pouvons citer quelques points à améliorer comme par exemple :

- Un visuel plus attractif
- Une fonction de recherche par localisation (trouver des magasins autour de soi)
- Un système de notation pour les utilisateurs de l'application
- L'ajout de Firebase à la version ANDROID

## 7 Ressources

Nous nous sommes inspirés de plusieurs exemples de code sur des sites de tutoriel ou sur youtube, de solutions trouvées sur StackOverflow, etc... qui dans la plupart des cas ont dû être re adaptées afin de convenir à notre contexte.

Citées ci-dessous, quelques aides qui nous ont beaucoup aidé :

Bug XCode :

[Accéder à l'URL](#)

Barre de recherche par nom :

[Accéder à l'URL](#)

Télécharger des images :

[Accéder à l'URL](#)

## Références

- [1] Swift. <https://www.apple.com/fr/swift/>.
- [2] Materialsearchbar. <https://github.com/mancj/MaterialSearchBar>.
- [3] Sqlite. <https://sqlitebrowser.org/>.
- [4] Firebase. <https://firebase.google.com/docs/storage>.

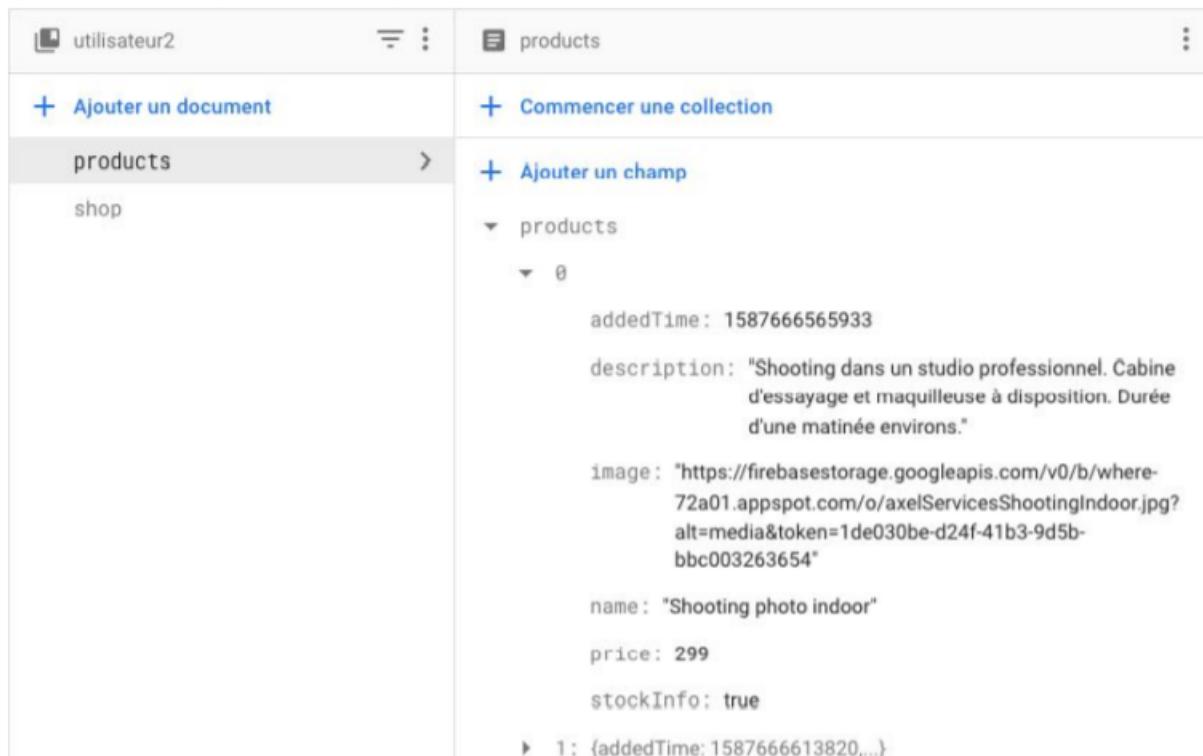
## 8 Annexes

Schéma d'une Collection utilisateur :

(document 'shop')

utilisateur2	shop
+ Ajouter un document	+ Commencer une collection
products	+ Ajouter un champ
shop >	address : "12 rue des Fleurs 97400 Sait-Denis,"
	- city
	0 "Saint-Denis"
	description: "Passionné de photographie, prise de rendez-vous par téléphone au 09010"
	image : "https://firebasestorage.googleapis.com/v0/b/where-72a01.appspot.com/o/AxelArtisanPhotographeShopImg.jpg?alt=media&token=6136998f-8b43-41bf-9407-41cb904df01a"
	name : "Axel Photography"

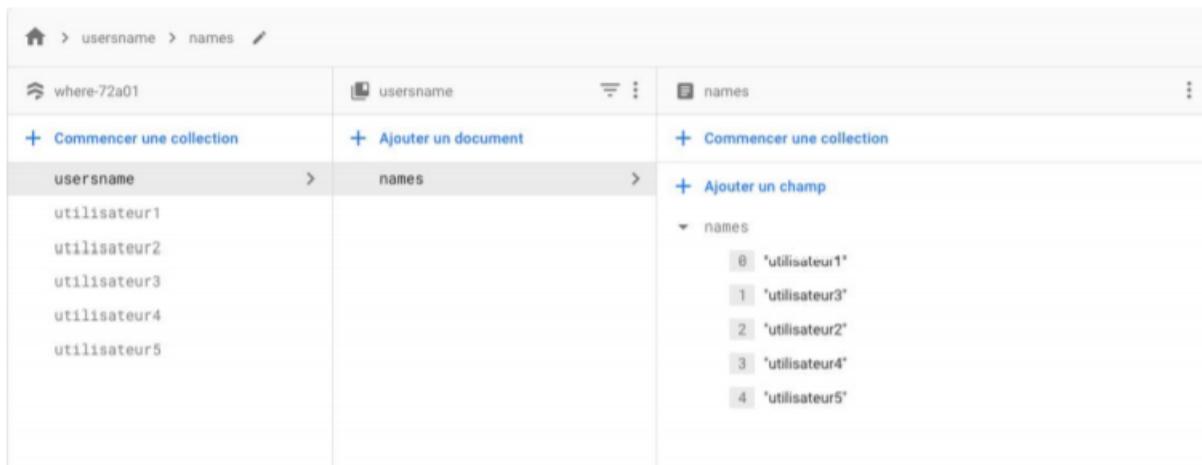
## (document 'products')



```

addedTime: 1587666565933
description: "Shooting dans un studio professionnel. Cabine d'essayage et maquilleuse à disposition. Durée d'une matinée environ."
image: "https://firebasestorage.googleapis.com/v0/b/where-72a01.appspot.com/o/axelServicesShootingIndoor.jpg?alt=media&token=1de030be-d24f-41b3-9d5bbc003263654"
name: "Shooting photo indoor"
price: 299
stockInfo: true
  
```

▶ 1: {addedTime: 1587666613820,...}

Schéma complet de la base de données :


username	names
utilisateur1	[{"0": "utilisateur1"}, {"1": "utilisateur3"}, {"2": "utilisateur2"}, {"3": "utilisateur4"}, {"4": "utilisateur5"}]
utilisateur2	[{"0": "utilisateur1"}, {"1": "utilisateur3"}, {"2": "utilisateur2"}, {"3": "utilisateur4"}, {"4": "utilisateur5"}]
utilisateur3	[{"0": "utilisateur1"}, {"1": "utilisateur3"}, {"2": "utilisateur2"}, {"3": "utilisateur4"}, {"4": "utilisateur5"}]
utilisateur4	[{"0": "utilisateur1"}, {"1": "utilisateur3"}, {"2": "utilisateur2"}, {"3": "utilisateur4"}, {"4": "utilisateur5"}]
utilisateur5	[{"0": "utilisateur1"}, {"1": "utilisateur3"}, {"2": "utilisateur2"}, {"3": "utilisateur4"}, {"4": "utilisateur5"}]

Loading Spinner :

