4HYBD - Documentation technique

Supinfo Meng.1.

Corentin Hoareau

Tristan Charpentier

Sommaire

- 4HYBD - Documentation technique
  - Sommaire
  - Introduction
  - Lancer l'application
    - APK
    - Installation
  - Guide d'utilisation
    - Tests unitaires
  - Conception
    - Technologies choisies
    - Schéma de données
  - Challenges techniques
    - Synchronisation des utilisateurs
    - Les différentes versions de @angular/fire
    - Mocker les plugins Capacitor
    - Protection des collections Firestore
  - Conclusion

Introduction

Ce document a pour vocation de décrire l'organisation de notre groupe, les choix technologiques, ainsi que les défis rencontrés lors de la réalisation de l'application "BeUnreal" pour le projet 4HYBD.

Les demandes du sujet sont comme suit :

- Réalisation d'une application mobile type "Instagram/BeReal" sur Android avec Ionic ;
  - Choix du framework UI : libre ;
  - CRUD basique des utilisateurs ;
  - Gestion des images ;
    - Images publiques ;
    - Stories avec géolocalisation ;
  - Gestion de chats privés ;
    - Envoi d'images ;
    - Chats en groupe ;
- Réalisation du backend de l'application ;
  - Technologie libre.

Lancer l'application

APK

Une APK signée de l'application est fournie dans le dossier racine du rendu si vous souhaitez utiliser l'application sur un appareil mobile sans installer l'environnement de développement.

Installation

Installez le CLI ionic.

```
npm install -g @ionic/cli
```

Installez les dépendances de l'application.

```
npm i
```

Installez Android Studio (ou le SDK standalone Android) afin de pouvoir créer et lancer des émulateurs.

Guide Ionic

Android Studio

Lancez l'application sur votre émulateur avec la commande :

```
ionic cap run android
```

Cette commande se chargera automatiquement du build de l'application, de la synchronisation des plugins capacitor (si nécessaire), et du lancement de l'émulateur (vous devrez choisir un émulateur après avoir lancé la commande).

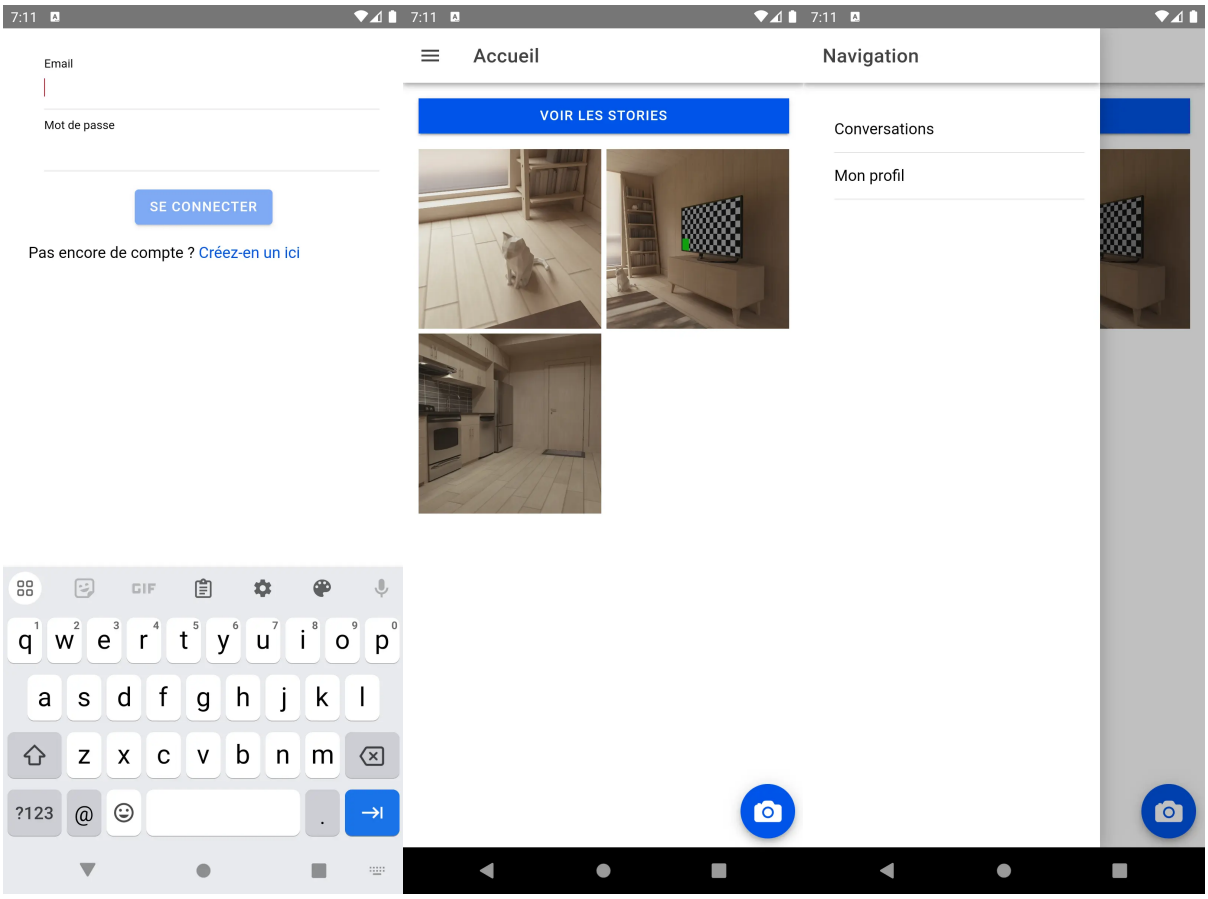
Alternativement, le dossier `android` se trouvant à la racine du projet peut être ouvert dans android studio et l'application peut être lancée de cette manière.

Il n'y a pas de backend à lancer, tout est hébergé sur Firebase.

Guide d'utilisation

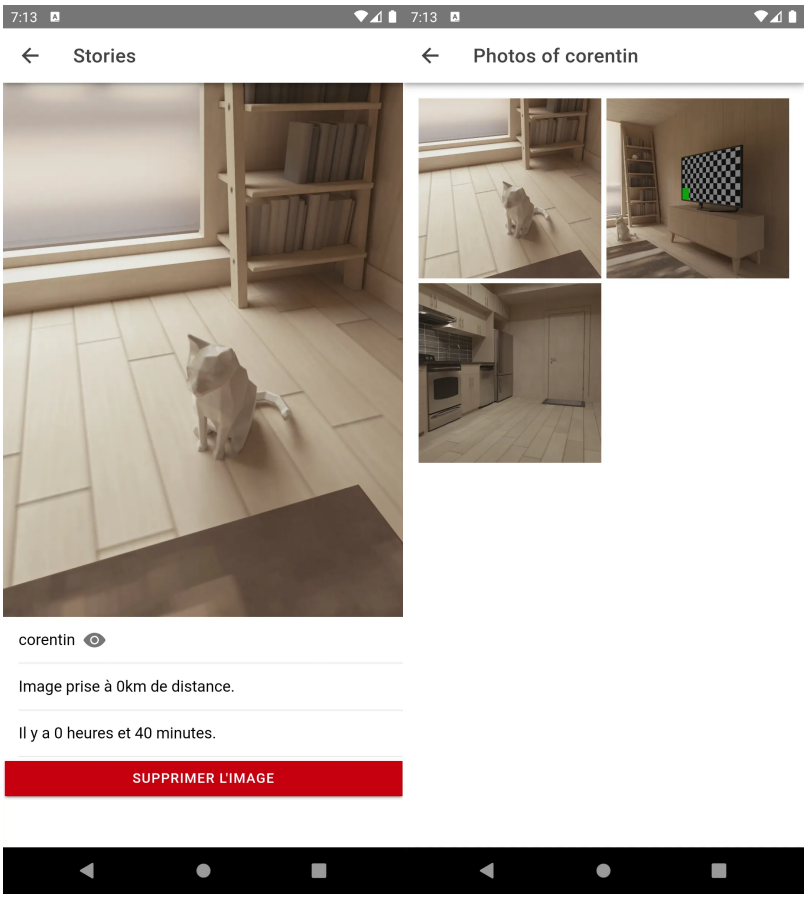
La première page de l'application est la page de connexion.

Une fois connecté, l'écran d'accueil affiche toutes les photos des plus récentes aux plus anciennes. Le volet de navigation nous permet d'accéder aux autres fonctionnalités.

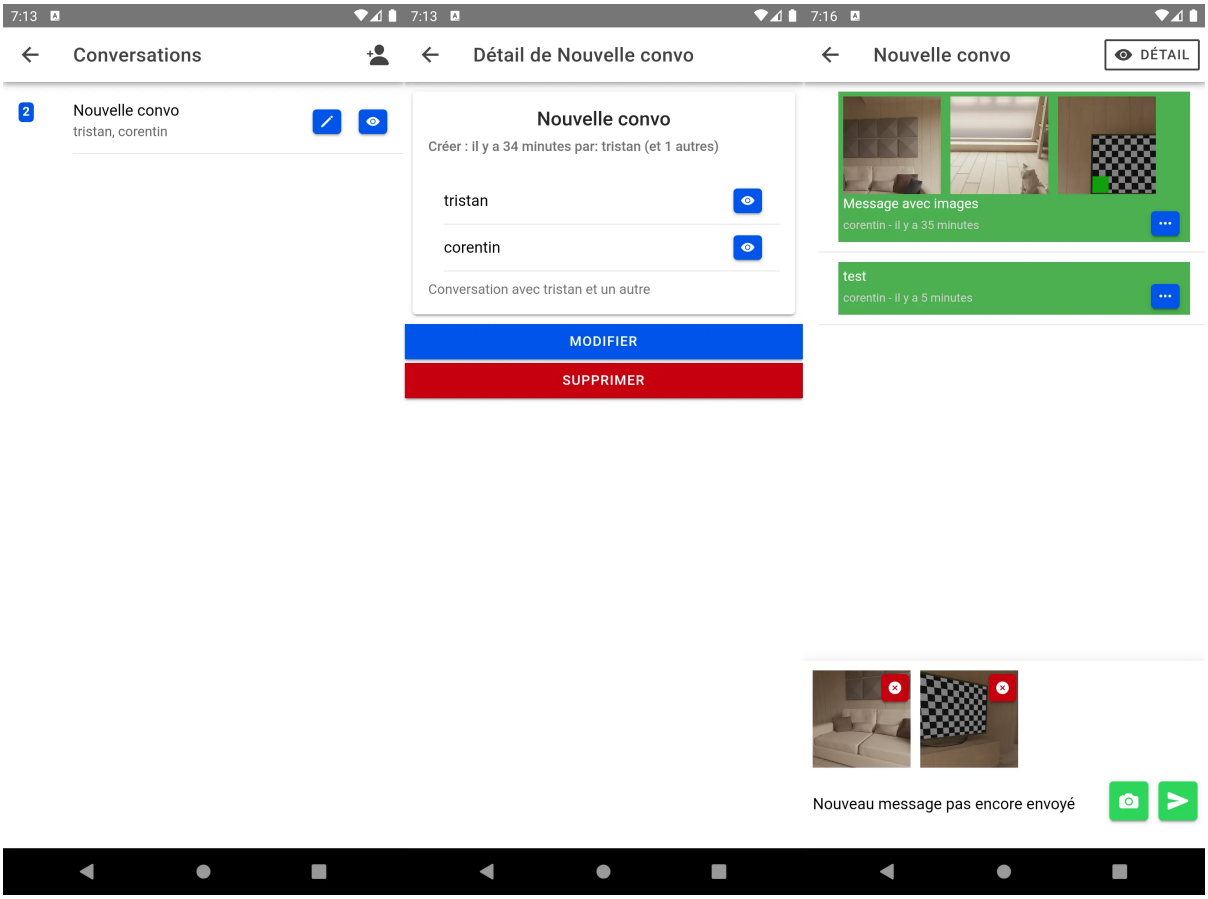


Le bouton d'affichage des stories permet de visionner toutes les images qui ont été prises à moins de cinq kilomètres il y a moins d'un jour.

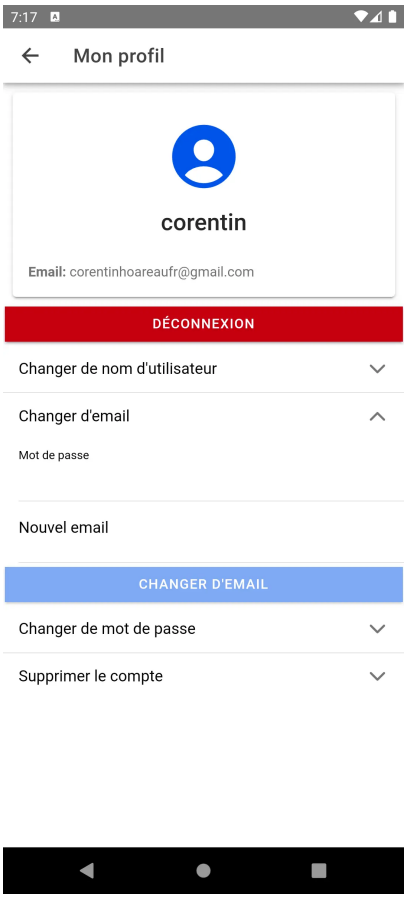
Lorsque l'on visionne une image, on peut appuyer sur le nom de l'uploadeur afin de visionner toutes les photos de son compte.



L'application permet de créer différentes conversations, dans lesquelles vous pouvez ajouter une ou plusieurs personnes, et envoyer messages comme images.



Enfin, la page de profil permet de voir vos informations de profil.



Tests unitaires

La commande `npm test` permet de lancer les tests unitaires, seuls les services sont testés.

Conception

Pour ce projet, nous avons réparti les tâches par fonctionnalité et non avec une séparation front-end, chose que nous faisons habituellement, cela pour deux raisons :

- Nous avons choisi une solution "backend as a service" au lieu de créer notre propre serveur backend ;
- Ionic étant la technologie phare du projet, nous ne souhaitions pas priver un des membres du groupe de l'opportunité de développer sur ce framework.

Technologies choisies

- Back end ;
  - Nous utilisons Firebase pour mettre en place notre back end ;
- Front end ;
  - Le framework UI utilisé est Angular ;
  - Les tests unitaires tournent sur Jasmine, et sont orchestrés avec Karma ;
  - Capacitor apporte les différentes fonctionnalités natives nécessaires.

Schéma de données

Nous utilisons la base de données noSQL `Firestore`, les schémas de données sont définis par des interfaces dans le code front-end.

`IImage` : Interface intégrée dans d'autres schémas de données.

Nom	Type
url	string
filename	string

`IPublicImage` : Images disponibles publiquement sur l'écran d'accueil ou les stories.

Nom	Type
id	string
ownerUid	string
geoPoint?	GeoPoint
createdAt	Timestamp
image	IImage
distance?	number

`IUser` : Données utilisateur.

Nom	Type
id	string
email	string
name	string

`IChat` : Interface pour un message individuel dans le chat.

Nom	Type
idUser	string
message	string
images	IImage[]
createdAt	Date

`IConversation` : Une conversation entre deux ou plusieurs personnes.

Nom	Type
id?	string
idUsers	string[]
name	string
createdAt	Date

Nom	Type
chats?	IChat[]

Challenges techniques

Plusieurs défis ont été rencontrés durant la réalisation du projet.

Synchronisation des utilisateurs

Toutes les fonctionnalités liées à Firebase sont implémentées grâce à la librairie `@angular/fire`.

Nous utilisons `AngularFireAuth` pour la gestion de l'identité. Cependant, si l'on veut récupérer des informations sur un autre utilisateur, il faut des droits d'administrateur. La solution `FireAuth` n'est pas conçue pour servir de "collection d'utilisateurs", c'est pour cela que nous avons mis en place une collection utilisateurs sur `FireStore`, cela implique que les opération CRUD doivent parfois être effectuées à la fois sur `FireAuth`, et `Firestore`.

```
async createUser(email: string, password: string, name: string)  {
    const result = await this.afAuth.createUserWithEmailAndPassword(email,
password);
    if (result.user != null) {
        const userData: IUser = {
        email: email,
        name: name
        };
        await this.firestore.doc('users/' + result.user.uid).set(userData);
        await result.user.sendEmailVerification();
        sessionStorage.setItem('idCurrentUser', result.user.uid);
        this.router.navigateByUrl('/home');
    }
}
```

Dans cette méthode, l'utilisateur est d'abord crée avec `AngularFireAuth`, puis avec `AngularFirestore`.

Il n'est cependant pas toujours possible d'effectuer les deux mises à jour simultanément.

Dans le cas d'un changement d'adresse email par exemple, un email de confirmation est envoyé avant que la mise à jour soit reconnue sur `AngularFireAuth`. Cela signifie que la mise à jour n'est pas "officielle" tant que le lien de confirmation n'a pas été cliqué.

Pour palier à ce problème, nous avons choisi de ne pas mettre à jour l'adresse email sur `Firestore` immédiatement. A chaque lancement de l'application, nous vérifions se l'email sur `FireAuth` et `Firestore` correspondent, sinon, l'email sur `Firestore` est mis à jour.

```
initializeApp() {
    this.unsubscribeAuthState = this.afAuth.authState.subscribe(auth => {
        if (auth?.uid) {
        sessionStorage.setItem('idCurrentUser', auth.uid ?? '')
        this.userService.getCurrentUser().subscribe((result) => {
        if (result != null) {
            if (result.email != auth.email) {
            if(auth.email)
            this.userService.updateEmailCollectionUser(auth.email)
            }
        }
        })
        this.router.navigateByUrl('/home');
        } else {
        this.router.navigateByUrl('/login');
        }
    });
}
```

Les différentes versions de @angular/fire

La librairie `@angular/fire` a connu en 2021 une mise à jour à la version `v7`. Cette version apporte une nouvelle manière d'importer les composants de la librairie.

Avant la `v7`. Les imports étaient effectués en injectant une unique classe, dont l'instance contenait toutes les fonctionnalités du service Firebase.

```
import { AngularFirestore } from '@angular/fire/compat/firestore';

getImagesForOwner(ownerUid: string) {
    return this.firestore.collection<IImage>(
        'images',
        ref => ref.orderBy('createdAt', 'desc').where('ownerUid', '==',
ownerUid)
    ).valueChanges({ idField: 'id' });
}
```

Depuis la `v7`. Il est possible d'importer individuellement chaque méthode que l'on souhaite utiliser (par exemple, si l'on ne manipule que des collections et pas des documents), la classe injectée n'apporte plus les fonctionnalités du service, mais sert uniquement de "clé" que l'on fait passer comme premier argument de chaque méthode.

```
import { Firestore, GeoPoint, addDoc, collection, collectionData, endAt,
getDocs, orderBy, query, startAt, where } from '@angular/fire/firestore';

getImagesForOwner(ownerUid: string) {
    const imagesCollection = collection(this.firestore, 'images');

    return collectionData(
        query(
            this.imagesCollection,
            orderBy('createdAt', 'desc'), where('ownerUid', '==', ownerUid)
        ), { idField: 'id' }
    ) as Observable<IImage>
}
```

Nous avons d'abord utilisé le nouveau mode d'import, afin de pouvoir profiter du tree-shaking sur les imports de `@angular/fire`. Cependant, lors de la rédaction des tests unitaires, nous avons réalisé que le système de tests, `jasmine`, n'est pas capable de créer des mock sur les imports de méthode, uniquement sur les providers.

Nous avons donc dû effectuer des refactors sur tous les services afin de pouvoir mocker les providers et rédiger les tests unitaires.

Mocker les plugins Capacitor

Un problème similaire a été constaté avec Capacitor. Comme jasmine utilise un navigateur web, Capacitor renvoie des erreurs car il ne fonctionne que sur une plateforme native.

Comme Capacitor n'est pas un élément injectable, il est impossible de mocker ses classes (`Geolocation`, `Camera`, etc...).

Le guide Ionic préconise de changer des paramètres dans les définitions typescript afin que tous les imports de `@capacitor/*` ciblent un différent chemin.

Nous avons tenté cette solution mais avons ensuite rencontré certains problèmes quant au lancement des tests. Nous avons finalement décidé de conteneuriser les appels natifs dans des fonctions séparées, nous créons ensuite des mocks de ces fonctions uniquement.

Protection des collections Firestore

`Firestore` permet de configurer un accès conditionnel aux ressources par l'intermédiaire de règles de sécurité, nous limitons par exemple les opérations de lecture d'une image aux utilisateurs authentifiés, et les opérations d'écriture aux utilisateurs ayant crée la ressource.

```
service cloud.firestore {
  match /databases/{database}/documents {

    match /images/{image} {
      function isSignedIn() {
        return request.auth != null;
      }

      function isOwner(rsc) {
        return request.auth != null && request.auth.uid ==
rsc.data.ownerUid;
      }

        allow read: if isSignedIn();
      allow create: if request.auth != null && isOwner(request.resource);
      allow update, delete: if request.auth != null && isOwner(resource);
    }

  }
}
```

L'authentification est vérifiée par la présence (ou l'absence) de la propriété `auth` de la requête.

Conclusion

Dans cette documentation technique, nous avons présenté les demandes du sujet, puis les instructions d'installation, avant de parler de la phase de conception et des challenge techniques rencontrés.

Ce projet nous a permis de solidifier nos compétences en développement Angular et Ionic. Cependant, le plus grand inconnu n'était pas l'implémentation du code natif/UI, mais plutôt l'intégration de Firebase.

Ce projet était notre premier projet de développement avec une solution "backend as a service". La facilité d'accès et de configuration nous a permis à tous les membres du groupe de contribuer au backend, même ceux spécialisés sur le front end. De plus, un backend partagé facilite le processus d'itération, et nous a permis de facilement tester les fonctionnalités telles que les stories et le chat, de plus, la latence additionnelle comparée à un backend on-premise rajoute un délai plus réaliste aux requêtes.

En somme, nous sommes très satisfaits des avantages que nous offre Firebase, cette solution a grandement facilité tous les aspects du développement backend (qui à ce stade n'est plus que de la configuration). Pour ce qui est du front-end, nous avons trouvé Angular plus agréable à utiliser que React, malgré une première prise en main plus complexe (à cause du style de développement favorisant les observables). Ce projet fut une expérience agréable et enrichissante.