

Rapport développement pour mobiles M1 Info

Guillaume NIRLO, Emmanuel NATIVEL, Gilles MAILLOT & Erwan GUICHARD

23 novembre 2019

Résumé

Dans ce rapport, nous allons vous présenter nos deux applications mobiles, chacune développée sous iOS et Android. L'application de gestion des employés, destinée aux gérants des entreprises, a été développée par Guillaume et Erwan, et l'application de pointage, destinée aux employés, par Emmanuel et Gilles. Nous avons ensuite décidé d'assigner une plateforme de développement par personne. Au final, pour l'application des gérants, la version Android a été faite par Erwan et la version iOS par Guillaume. Pour l'application des employés, c'est Gilles qui s'est occupé de la version Android et Emmanuel de la version iOS.

1 Introduction

Virtual Time Clock est une application destinée aux entreprises souhaitant améliorer la gestion de leurs employés et notamment le contrôle de leur assiduité au travail grâce à la géolocalisation.

En effet, le principal atout de cette application est la mise en place d'un système de pointage des employés automatique, à distance et en temps réel, et cela tout au long de la journée. Par conséquent, un gérant d'une entreprise peut savoir quels employés sont présents sur les lieux d'une mission à n'importe quel moment.

De plus, cette application met à disposition d'autres fonctionnalités supplémentaires telles que la gestion simultanées de plusieurs missions pour une même entreprise ou encore la possibilité pour l'employé d'enregistrer un rapport, avec ou sans image, directement via l'application.

2 La plateforme Firebase [1]

Nous avons dû réfléchir à la technologie que nous allions utiliser pour gérer tout le côté back-end de notre application, aussi bien pour le stockage de nos données que pour la gestion des comptes utilisateurs. Alors nous avons finalement choisi d'utiliser la plateforme Firebase, de Google.

Nous avons utilisé 3 modules de Firebase [1] dans notre application. Le module **Authentication** pour gérer la création et la connexion des utilisateurs, le module **Firestore**, une base de données NoSQL orientée document, pour stocker les données des utilisateurs et de l'entreprise, et enfin le module **Storage**, pour stocker les images de notre application.

Chacun de ces modules nous fournit des fonctions java et swift que nous avons utilisé pour interagir avec notre back-end.

Vous pourrez vous connecter à notre projet Firebase à l'adresse suivante : <https://firebase.google.com/?pli=1>.

L'identifiant est : *virtualtimeclock@gmail.com* et le mot de passe : *entreprise974*



3 Choix communs

Les deux applications sont traduites en français et en anglais. Au niveau de l'orientation des écrans, nous avons autorisé uniquement le mode portrait.

3.1 Les Sons

Plusieurs sons sont intégrés dans nos applications, notamment au niveau de l'écran de connexion, au moment de pointer ou encore en parcourant les vues dans l'application des gérants. Voici les exemples de code nous permettant d'insérer des sons :

— Sous iOS :

```
if let soundFilePath = Bundle.main.path(forResource: "soundOn", ofType: "mp3") {
    let fileURL = URL(fileURLWithPath: soundFilePath)
    do {
        try self.player = AVAudioPlayer(contentsOf: fileURL)
        self.player.delegate = self
        self.player.play()
    }catch { print("Erreur lors de la lecture du son") }}
```

— Sous Android :

```
public void mediaPlayer(MediaPlayer m){ m.start(); }
mediaPlayer(on);
```

3.2 Ressources

Les sons de notre application sont libres de droits et ont été pris sur *orange free sounds* [3]. Quant aux images, elles sont également libres de droits et proviennent de *Pixabay* [4].

3.3 Dépendances

Pour l'application des employé, nous avons utilisé la librairie externe *Toast-Swift* [6] pour l'affichage des toasts. Dans l'application des gérants, la librairie Ucrop[7] a été utilisée pour rogner les images prises par l'appareil photo, et la librairie HorizontalScrollMenu[2] a elle été utilisée pour gérer la navigation du TabBar (menu scrollable).

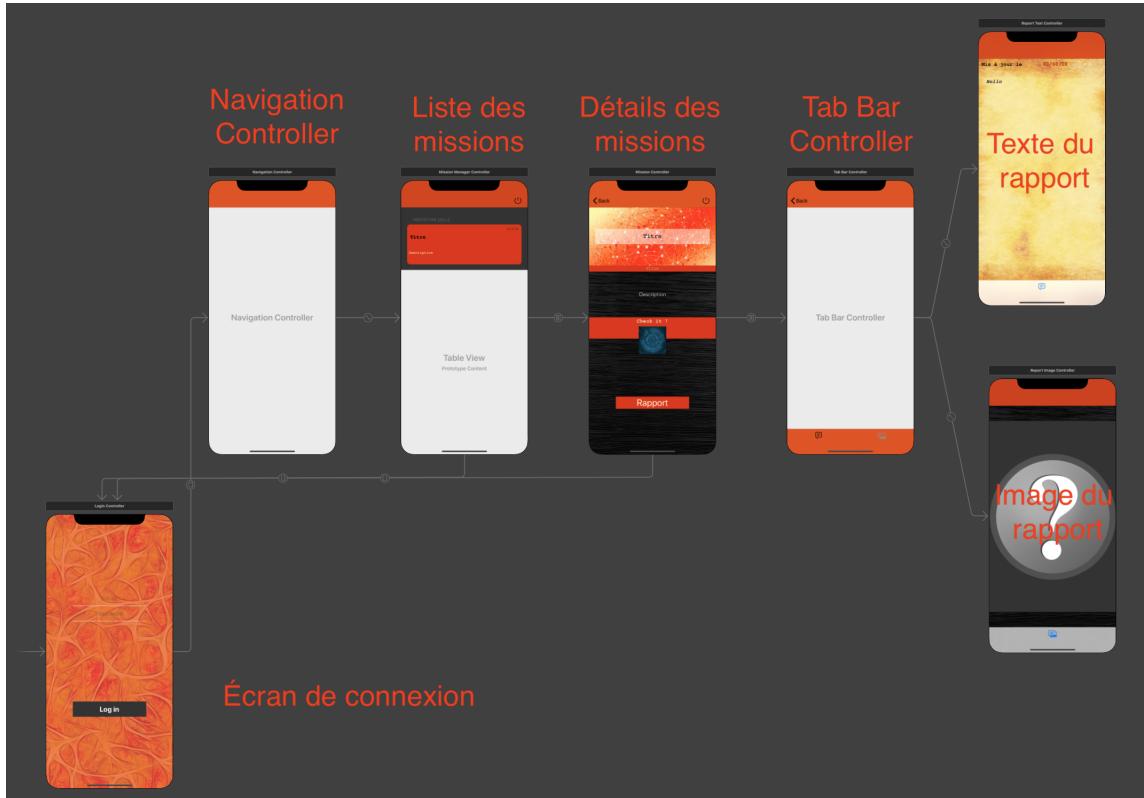
3.4 Ecran de lancement



4 Application des employés

Pour un employé, l'application permet de consulter les différentes missions de l'entreprise et de les consulter. Celui-ci peut ensuite indiquer qu'il est sur les lieux de la mission, ce qui va enclencher un suivi automatique de ses entrées et sorties dans la zone de la mission. De plus, il pourra faire un rapport de l'avancé des travaux pour la ou les mission(s) qu'il a traité durant la journée.

4.1 Architecture de l'application



4.2 Connexion d'un employé



Pour se connecter, un employé a besoin de son adresse email et de son mot de passe. Ces identifiants sont créés et fournis par le gérant de l'entreprise à chacun de ses employés. Si les identifiants rentrés sont ceux d'un gérant, alors la connexion échoue. De plus, une fois connecté, l'employé reste connecté jusqu'à ce qu'il se déconnecte, même si il quitte l'application. Si il est déjà connecté, il sera directement redirigé vers la liste des missions grâce à cette portion de code :

Dans ViewDidLoad :

```
// On va tester si un utilisateur est déjà connecté. Si c'est le cas,  
// on le redirige vers la liste des missions.  
if let user = Auth.auth().currentUser {  
    print("un utilisateur est déjà connecté : \(user.email ?? "")")  
    perform(#selector(presentMissionManagerController), with: nil, afterDelay: 0)  
    // Ici, on utilise un sélecteur pour s'assurer que la vue vers laquelle  
    // on veut rediriger l'utilisateur soit belle et bien chargée.  
} else {  
    print("Aucun utilisateur n'est connecté.")  
}  
  
// Fonction appelée si un utilisateur est déjà connecté  
@objc private func presentMissionManagerController() {  
    self.performSegue(withIdentifier: "loginToMissionManager", sender: self)  
}
```

De plus, dans cette vue, le clavier peut être fermé en cliquant n'importe où sur l'écran grâce à un **TapGesture**.

4.3 Liste des missions



Cet écran permet à l'employé de consulter les différentes missions de l'entreprise et de choisir celle sur laquelle il doit travailler. C'est ici que chaque mission est chargée dans l'application depuis la base de données. Pour chacune des missions, une instance de la classe **Mission** est créée et est stockée dans la liste "**missions**" avant d'être affichée dans la *TableView*. Ensuite, lorsque l'employé clique sur une mission, l'instance de la classe **Mission** correspondante est passée à la vue suivante comme cela :

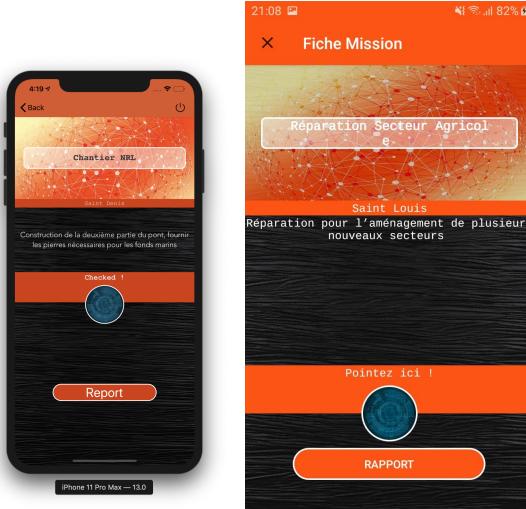
```
// Fonction permettant de faire des actions avant l'envoi du segue
override func prepare(for segue: UIStoryboardSegue, sender: Any?) {

    // On test si le segue est bien celui qu'on espère
    if segue.identifier == "MissionsManagerToMission" {

        // Récupération de la destination du segue
        let destination = segue.destination as! MissionController

        let mission = sender as! Mission
        destination.mission = mission
    }
}
```

4.4 Détails d'une mission



Lorsqu'un employé clique sur une mission, le détail de celle-ci s'affiche. Cette vue permet également à l'employé de pointer et de se diriger vers les écrans dédiés au rapport de cette mission. C'est également dans cette vue que le rapport de la mission est récupéré depuis la base de données, si il existe, puis celui-ci est transformé en une instance de la classe **Rapport**. Voici le code de récupération du rapport stocké dans la base de données :

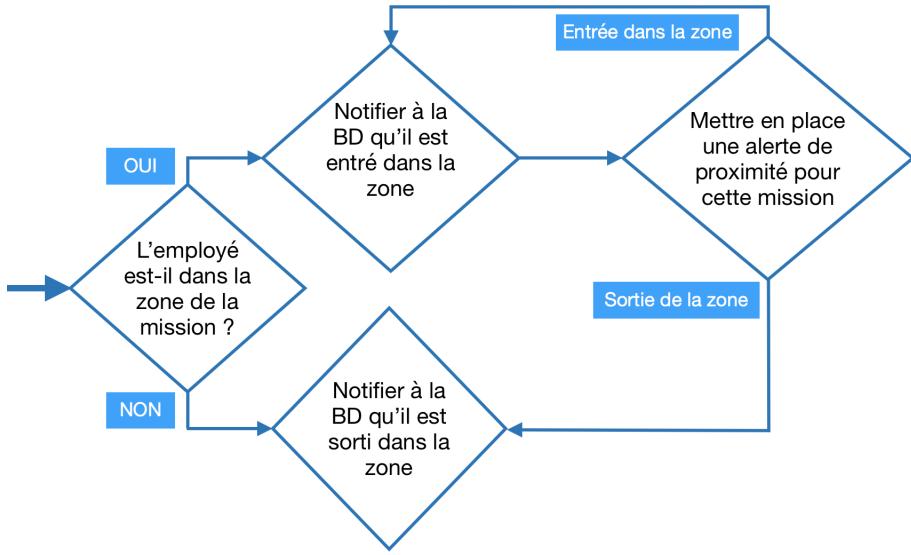
```
// Fonction permettant de récupérer le rapport de la mission courante dans la BD
private func getReportFromDB(forMissionId: String) {
    if missionID != "" {
        let missionsRef = database.collection("missions").document(missionID)

        missionsRef.getDocument { (document, error) in
            // On test si le document lié à cette mission existe bien
            if let document = document, document.exists {
                // On récupère le rapport stocké dans le document. C'est un dictionnaire.
                let rapportFromDB: [String: Any]? = document.get("rapport") as? [String: Any]

                if rapportFromDB != nil { // Si un rapport existe sur la base de données,
                    // On va récupérer les données de ce rapport :
                    let texte: String = rapportFromDB!["texte"] as! String
                    // ... Récupération des autres informations ...
                    // On récupère ce rapport sous forme d'objet
                    self.rapport = Rapport(texte: texte, imagePath: imagePath, date: date)
                } else { print("Il n'existe pas de rapport pour cette mission.") }
            } else {
                print("Erreur : Le document demandé pour cette mission n'existe pas !")
                self.view.makeToast(NSLocalizedString("missionNotExist", comment: "Alert message"), duration: 1.5) {}
            }
        }
    }
}
```

4.5 Pointage et contrôle d'assiduité automatique

Cette partie est la fonctionnalité principale de l'application, et permet au gérant de savoir si un employé se trouve sur les lieux de la mission à n'importe quel moment. Lorsqu'un employé clique sur le bouton de pointage le processus suivant s'enclenche et continue même si l'application fonctionne en **arrière plan** :



Dans la version iOS, toute la partie gestion des alerte de proximité est faite par la classe **LocationAreaManager** qui est un *Singleton Design Pattern* [5]. Le singleton nous permet de nous assurer que cette classe n'est instanciée qu'une seule et unique fois dans le code, grâce notamment à une constante *static* contenant l'instance de la classe au sein même de la classe.

```

class LocationAreaManager{

    static let shared = LocationAreaManager()

    let manager = CLLocationManager()
    let database = Firestore.firestore()
    var locationGranted: Bool?
    var currentArea: CLCircularRegion?

    private init(){}
    ...
  
```

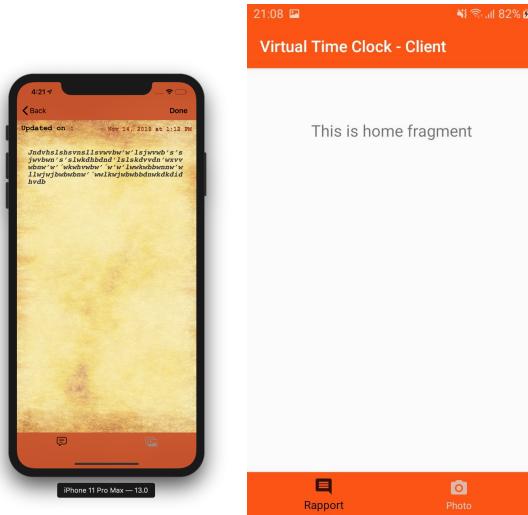
Grâce à **LocationAreaManager**, il est maintenant possible de stocker les différentes zones d'alerte de proximité activées, ce qui nous a permis de faire en sorte d'avoir toujours une seule et unique zone en monitoring (Car nous considérons que l'employé ne travail que sur une seule mission à la fois). Cela est géré par la fonction suivante, de la classe **LocationAreaManager** :

```

func startMonitoringForNewArea(newArea:CLCircularRegion, userID:String, mission: Mission){
    if currentArea != nil {
        manager.stopMonitoring(for: currentArea!)
        notifyExitToDB(userID: userID, mission: mission)
    }
    manager.startMonitoring(for: newArea)
    currentArea = newArea
}
  
```

4.6 Rapport de mission

4.6.1 Texte du rapport

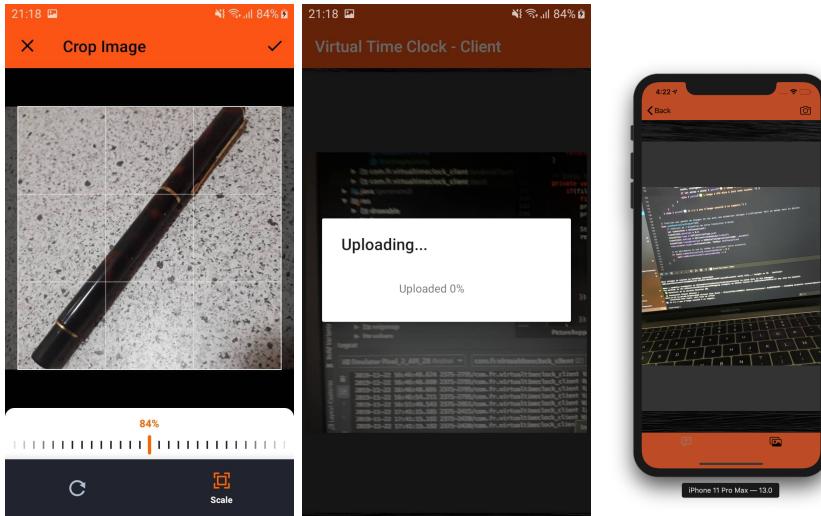


Dans cette vue, l'employé peut consulter et modifier le texte du rapport de la mission courante. En secouant le téléphone pendant plus de 0.5 secondes, il peut supprimer tout le texte d'un coup. Voici le code (condensé) de l'utilisation de l'accéléromètre :

```
private func listenDeviceMovement(){
    // On va utiliser les données pré-traitées de l'accéléromètre
    if motionManager.isDeviceMotionAvailable {
        motionManager.deviceMotionUpdateInterval = 0.1 // Interval de prélèvement des valeurs
        motionManager.startDeviceMotionUpdates(to: .main) { (dm, error) in
            let accelerationX = dm?.userAcceleration.x // Accélération sur l'axe X
            let accelerationY = dm?.userAcceleration.y // Accélération sur l'axe Y
            let accelerationZ = dm?.userAcceleration.z // Accélération sur l'axe Z
            let now = Date() // Date actuelle

            // Si on détecte une secousse
            if abs(accelerationX!) > 0.5 || abs(accelerationY!) > 0.5 || abs(accelerationZ!) > 0.5 {
                if !self.isMovingPhone { // Si le téléphone n'est pas en état "moving"
                    self.lastTimeCheck = now // On lance le chrono, en sauvegardant l'heure actuelle
                    self.isMovingPhone = true // Le téléphone est dans l'état "moving"
                } else if now.timeIntervalSince(self.lastTimeCheck!) >= 0.5 { // Secousse >0.5 secondes
                    self.reportTextView.text = "" // On efface le texte
                    self.isMovingPhone = false // Le téléphone n'est plus dans l'état "moving"
                }
            } else {
                // Le téléphone ne bouge plus, il n'est plus dans l'état "moving"
                self.isMovingPhone = false
            }
        }
    }
}
```

4.6.2 Image du rapport



En complément du texte, l’employé peut consulter/ajouter une image au rapport. Pour cela, il utilise l’appareil photo du téléphone. Voici le code permettant d’ouvrir la l’appareil photo :

```
@objc func openCamera(){
    // On vérifie si le téléphone a un appareil photo
    if UIImagePickerController.isSourceTypeAvailable(UIImagePickerController.SourceType.camera) {
        let imagePicker = UIImagePickerController() //Utilisation de l'appareil photo iOS
        imagePicker.delegate = self
        imagePicker.sourceType = .camera //Utilisation de la caméra uniquement
        imagePicker.allowsEditing = true //On autorise l'édition de l'image
        self.present(imagePicker, animated: true, completion: nil) //On affiche la vue
    }
}
```

4.6.3 Navigation dans le TabBar grâce aux geste courants

Pour naviguer entre le texte et l’image de rapport, l’employé peut "swiper". Lorsque le *Swipe-Gesture* est détecté de la manière suivante ...

```
let swipeLeft = UISwipeGestureRecognizer(target: self, action: #selector(onSwipeGesture))
swipeLeft.direction = .left
self.view.addGestureRecognizer(swipeLeft)
```

... l’écran est "poussé" grâce à une transition. Voici le code :

```
func pushControllerFromRight(){
    let transition = CATransition() //Création de l'animation de transition d'écran
    transition.duration = 0.5
    transition.type = CATransitionType.push
    transition.subtype = CATransitionSubtype.fromRight
    transition.timingFunction = CAMediaTimingFunction(name: .easeOut)
    view.window!.layer.add(transition, forKey: kCATransition)
    // On incrémente la vue du tabBar en utilisant notre animation
    if (self.tabBarController?.selectedIndex)! < 2 {
        self.tabBarController?.selectedIndex += 1
    }
}
```

4.7 Déconnexion

L'utilisateur peut se déconnecter grâce au bouton se trouvant dans la barre de navigation. Si il se déconnecte, il sera redirigé vers l'écran de connexion et devra s'authentifier pour accéder à l'application.



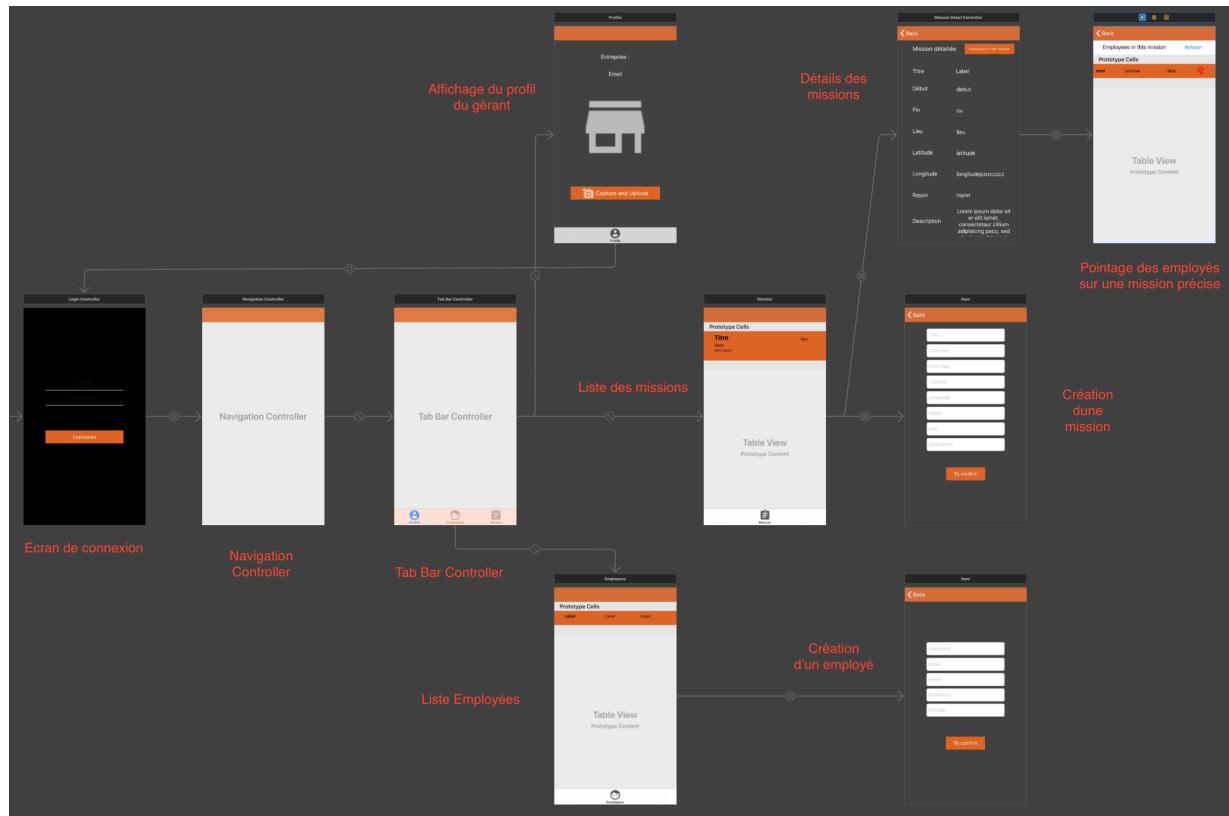
La déconnexion de l'utilisateur se fait grâce à Firebase, via le code suivant en java :

```
private void userLogout() {  
    mAuth.getInstance().signOut();  
    Intent intent = new Intent(MissionManagerActivity.this, LoginActivity.class);  
    intent.addFlags(Intent.FLAG_ACTIVITY_CLEAR_TOP);  
    intent.addFlags(Intent.FLAG_ACTIVITY_CLEAR_TASK);  
    startActivity(intent);  
}  
  
userLogout();
```

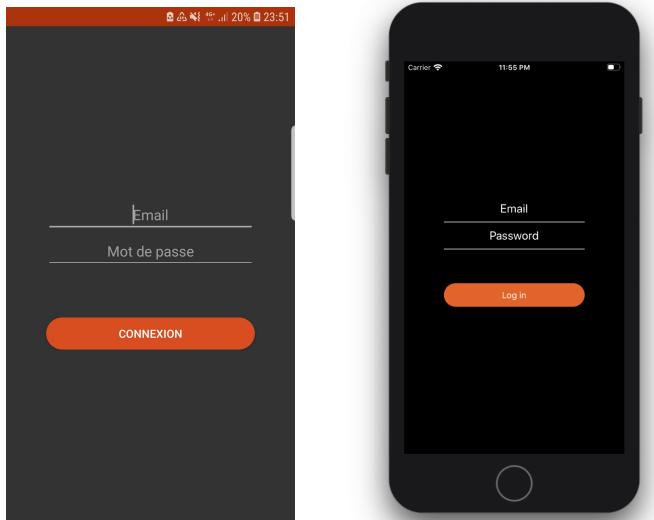
5 Application des gérants

Pour un gérant de l'entreprise, l'application lui permettra de gérer ses employés et ses différentes missions. En effet, il pourra consulter la liste des missions et l'avancement de celle-ci grâce au rapport rédigé par les employés mais aussi ajouter de nouvelles missions ou les supprimer. Il pourra également consulter la liste de ses employés, en ajouter de nouveaux.

5.1 Architecture de l'application



5.2 Connexion du gérant



Tout comme les employés, les gérants doivent se connecter avec un email et un mot de passe et la connexion se fera seulement si les identifiants sont bien ceux d'un compte gérant. La connexion est aussi maintenue tant qu'il ne se déconnecte pas et une fois connecté, ils sera redirigé sur le menu principal de l'application grâce à ce code swift :

```
override func viewDidLoad() {
    super.viewDidLoad()

    // On va tester si un gérant est déjà connecté.
    // Si c'est le cas, on le redirige vers son profil
    if let user = Auth.auth().currentUser {
        print("Un gérant est déjà connecté : \(user.email ?? "")")
        perform(#selector(loginagain), with: nil, afterDelay: 0)
        // Ici, on utilise un sélecteur pour s'assurer que la vue vers laquelle
        // on veut rediriger le gérant soit belle et bien chargée.
    } else { print("Aucun gérant n'est connecté.") }
}

@objc private func loginagain() {
    self.performSegue(withIdentifier: "loginToHome", sender: self)
}
```

Le clavier se réduit aussi en cliquant n'importe où sur l'écran avec la fonction **onKeyboard** :

```
//Tap gesture pour fermer le clavier quand on clique dans le vide
let tapGesture = UITapGestureRecognizer(target: self, action: #selector(hideKeyboard))
view.addGestureRecognizer(tapGesture)

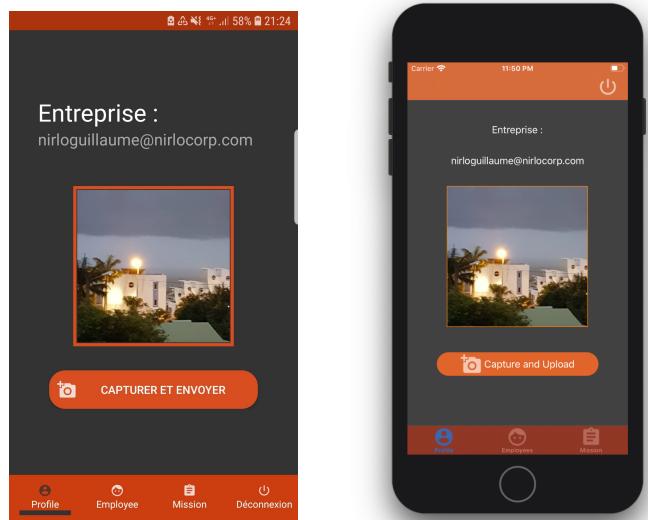
//Fonction appelée par le TapGesture : permet de fermer le clavier
@objc private func hideKeyboard(){
    mailTF.resignFirstResponder()
    mdpTF.resignFirstResponder()
}
```

5.3 Menu de navigation

Afin de naviguer dans les différents onglets, un menu de navigation tiré d'une librairie externe [2]. Elle a été choisie car elle permet d'avoir un menu extensible pour le futur de l'application lors de l'ajout de nouvelles fonctionnalités. En effet, en ajoutant des onglets au menu, celui-ci aura la possibilité de coulisser horizontalement.

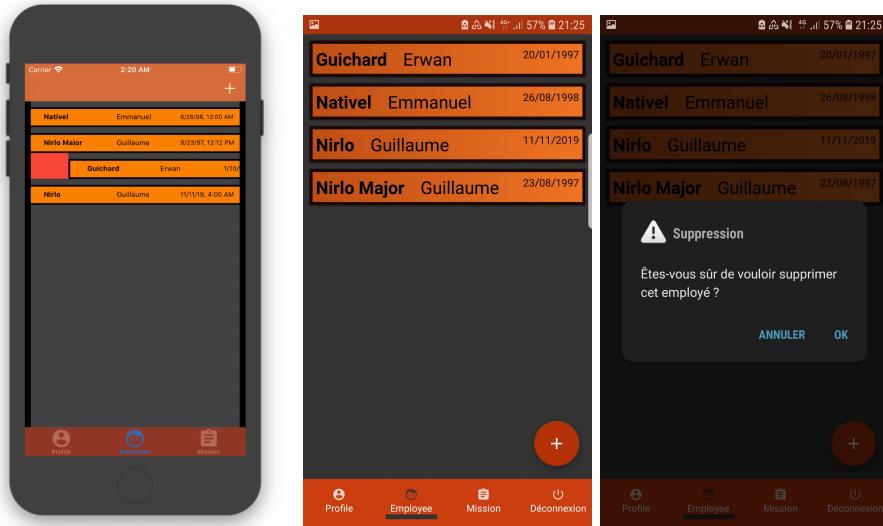
5.4 Profil de l'entreprise

Dans l'onglet Profil, le gérant a la possibilité de changer la photo de profil de l'entreprise en utilisant son appareil photo. Une fois la photo prise, il va devoir rogner l'image pour qu'elle soit adaptée au format du cadre de la photo de profil. Le redimensionnement est fait grâce à une librairie externe [7] qui permet de faire un redimensionnement simple avec la possibilité de faire une rotation de l'image.



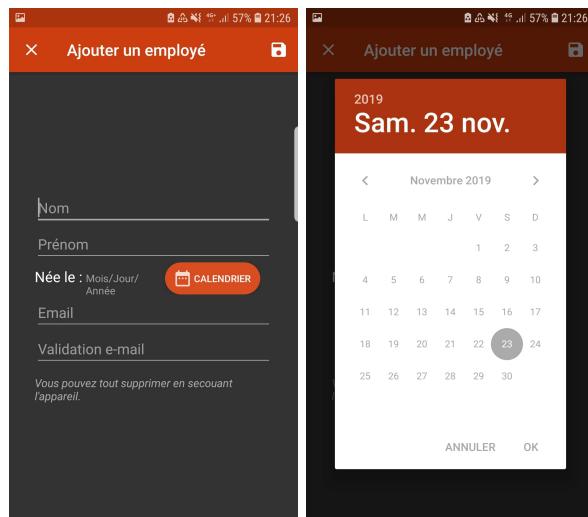
5.5 Liste d'employés

Dans l'onglet Employé, le gérant a juste accès à une liste en temps réel de tous les utilisateurs ayant un compte et il a la possibilité de supprimer un utilisateur en faisant glisser le cadre vers la droite ou la gauche Dans la version Android, il y aura par la suite une boîte de dialogue qui souvrira pour confirmer la suppression de cet utilisateur.



5.6 Crédation d'un employé

Dans ce même onglet, un bouton permet d'accéder à l'interface de création d'un nouvel employé. Le bouton calendrier permet d'ouvrir une petite interface qui permet de sélectionner la date facilement comme on peut le voir sur ces images :



La création d'un employé s'effectue en 2 parties. Il faut d'abord ajouter l'employé dans le module « Authentification » de Firebase, puis créer un document pour stocker les données de cet utilisateur dans la base de données. Le mot de passe lui est attribué aléatoirement. Voici le code de création d'un employé sous iOS :

```
let len = 8 //taille du mot de passe
let pswdChars = "abcdefghijklmnopqrstuvwxyzABCDEFGHIJKLMNOPQRSTUVWXYZ1234567890"
let rndPswd = String((0..

```

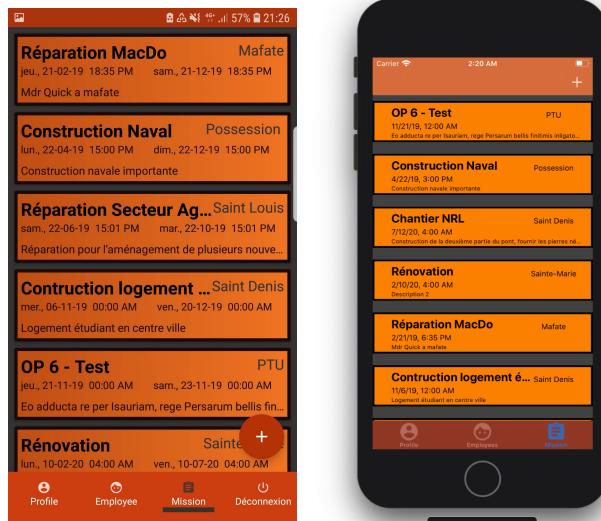
Si l'utilisateur souhaite vider rapidement le contenu des champs, il a la possibilité de secouer le téléphone afin d'ouvrir une boîte de dialogue pour confirmer la suppression des zones de textes.

Pour valider la création de l'utilisateur le gérant devra cliquer sur la disquette en haut à droite dans la barre du menu. Une fois la création de l'employé validée, Firebase va lui envoyer un mail contenant un lien pour réinitialiser son mot de passe. Voici un exemple du mail que recevra chaque nouvel employé avant leur première connexion :

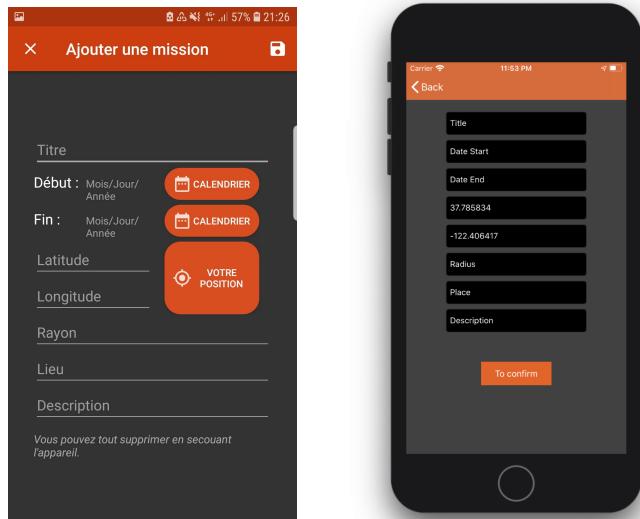


5.7 Liste des missions

Le gérant possède une vue sur l'ensemble des missions lui permettant d'ajouter ou de supprimer les missions actuelles. Chaque cellule est composée d'un bref affichage permettant de distinguer rapidement les différents enjeux.



5.8 Crédation de mission



L'onglet Mission contient aussi un bouton flottant pour accéder à l'interface de création de mission. Dans la version Android, la localisation de la zone de la mission peut être saisie manuellement ou être récupérée automatiquement en cliquant sur le bouton localisation. Dans la version iOS, dès l'affichage du formulaire, l'application demande l'autorisation de récupérer les coordonnées actuelles afin de remplir automatiquement les champs. Pour sauvegarder il faut que toutes les données soient saisies et que la date de départ soit plus petite que la date de fin. Voici un exemple d'écriture dans la base, il faut avant tout ouvrir la collection que l'on veut éditer puis écrire tous les champs manuellement et convertir les types de langage en types firebase si nécessaire.

```

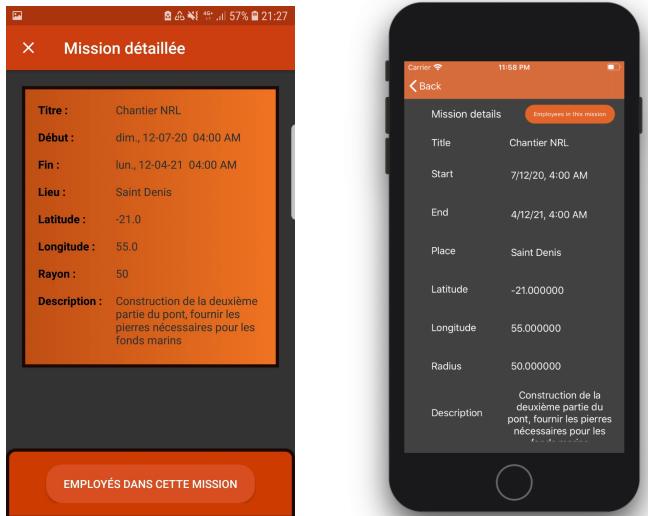
let db = Firestore.firestore()
let userDoc = db.collection("missions")
let dateCourante: Timestamp = Timestamp(date: datePicker!.date);
let dateCourante2: Timestamp = Timestamp(date: datePicker2!.date);
let getrayon: String = ""+rayonTF.text!
let rayonConvertDouble = Double(getrayon)
let rayonFinal: Double = Double(rayonConvertDouble!)
let pos: GeoPoint = GeoPoint(latitude: latitude, longitude: longitude )

userDoc.document().setData([
    "debut": dateCourante,
    "description": descTF.text!,
    "fin": dateCourante2,
    "lieu": lieuTF.text ?? "",
    "localisation": pos,
    "rayon": rayonFinal,
    "titre": titleTF.text ?? "",
])

```

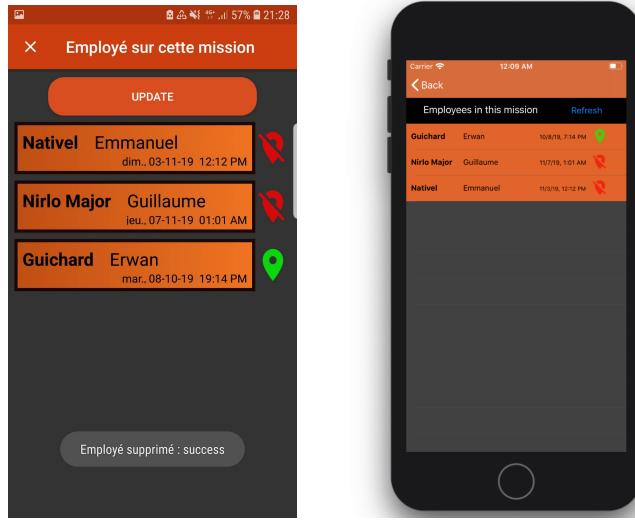
5.9 Détail d'une mission

Dans l'onglet Mission, le gérant a la possibilité de consulter les détails de chaque mission en cliquant dessus. Si la description est trop longue le gérant à la possibilité de scroll.



5.10 Consultation de la présence des employé

Dans la vue de détail des missions, le gérant peut accéder à la liste des employés présents acutellment sur les lieux de la missions en cliquant sur le bouton "Employés dans cette mission". Pour chacun des employés ayant pointé sur cette missions, le gérant peut voir son nom, prénom, la date de dernière mise à jour et si il est présent ou absent des lieux de la mission en temps réel. Si l'employé est sur les lieux, l'image d'un pointeur vert s'affiche, sinon c'est un pointeur rouge barré qui est affiché. Les données peuvent être actualisées en appuyant sur le bouton "Update".



Cette partie a été une des plus difficile à réaliser à cause de la gestion des requêtes asynchrones. En effet, nous avons du lire dans deux collections différentes en même, tout en synchronisant les lectures dans les deux collections et l'affichage des données. Tout d'abord, nous avons du récupérer l'id de la mission choisie par le gérant par le biais de la fonction prepare sous iOS :

```
override func prepare(for segue: UIStoryboardSegue, sender: Any?) {
    if segue.identifier == "goToPointEmp" {
        let destination = segue.destination as! ListPointEmpController
        destination.missionID = mission!.id
    }
}
```

Puis, il a fallut lire dans la collection "pointage" afin de récupérer le document associé à cette mission, grâce à son id. A partir de là, nous avions accès à la liste des id des employés qui ont pointés sur cette mission, la date de la dernière information de pointage et si l'employé a été présent ou non à cette date. Cependant, arrivé à ce niveau, il nous manquait encore le nom et le prénom de cet employé. Pour cela, nous avons du lire dans la collection "utilisateurs", pour chacun des id récupérés auparavant. Voici le code iOS correspondant :

5.11 Déconnexion

Le bouton Déconnexion ouvre simplement une boîte de dialogue qui demande au gérant si il souhaite vraiment être déconnecté et si il valide il sera ramené sur l'interface de connexion ou il devra se réauthentifier.



6 Limites rencontrées

Nous avons rencontré plusieurs limites pendant le développement de l'application. Premièrement, par manque de temps, il n'est pas possible de modifier des missions ou des employés, seules les fonctionnalités d'ajout et de suppression ont été implémentées. De plus, nous n'avons pas eu le temps de gérer l'affichage de la présence des employé en temps réel. Pour combler cela, nous avons ajouté un bouton d'actualisation. La création d'employés a aussi été délicate. En effet, après avoir créer un employé, ce dernier se connectait automatiquement en écrasant l'ancienne session connectée, ce qui pouvait causer des problèmes notamment à cause du fait qu'un client ne peut se connecter sur l'application d'un gérant.

7 Difficultés rencontrées

La principale difficulté que nous avons rencontré a été l'utilisation de Firebase, car aucun d'entre nous n'avait utilisé cette plateforme auparavant. Notamment pour gérer les requêtes asynchrones de firebase. En effet, il arrivait que les données étaient chargées après l'affichage de la vue. Au niveau de l'application des employés, la partie la plus difficile a été la gestion des alertes de proximité. Ce problème a été réglé grâce au singleton design pattern.

8 Conclusion

Pour conclure, nous avons réussi à réaliser l'objectif principal qui était de faire communiquer les deux applications afin de contrôler l'assiduité des employés au travail. De plus, nous avons découvert la plateforme Firebase qui s'avère très utile lorsque l'on veut se concentrer uniquement sur le front de notre application. Certaines fonctionnalités n'ont pas pu être implémentées et pourraient faire l'objet de futures évolutions.

Références

- [1] Firebase website. <https://firebase.google.com>.
- [2] Librairie horizontalscrollmenu. <https://github.com/darwin-morocho/HorizontalScrollMenu>.
- [3] Orange free sounds. <http://www.orangefreesounds.com>.
- [4] Pixabay. <https://pixabay.com/fr/>.
- [5] Singleton design pattern wiki. [https://fr.wikipedia.org/wiki/Singleton_\(patron_de_conception\)](https://fr.wikipedia.org/wiki/Singleton_(patron_de_conception)).
- [6] Toast-swift reference. <https://github.com/scalessec/Toast-Swift>.
- [7] Librairie ucrop. <https://github.com/Yalantis/uCrop>.