CHAPITRE 1

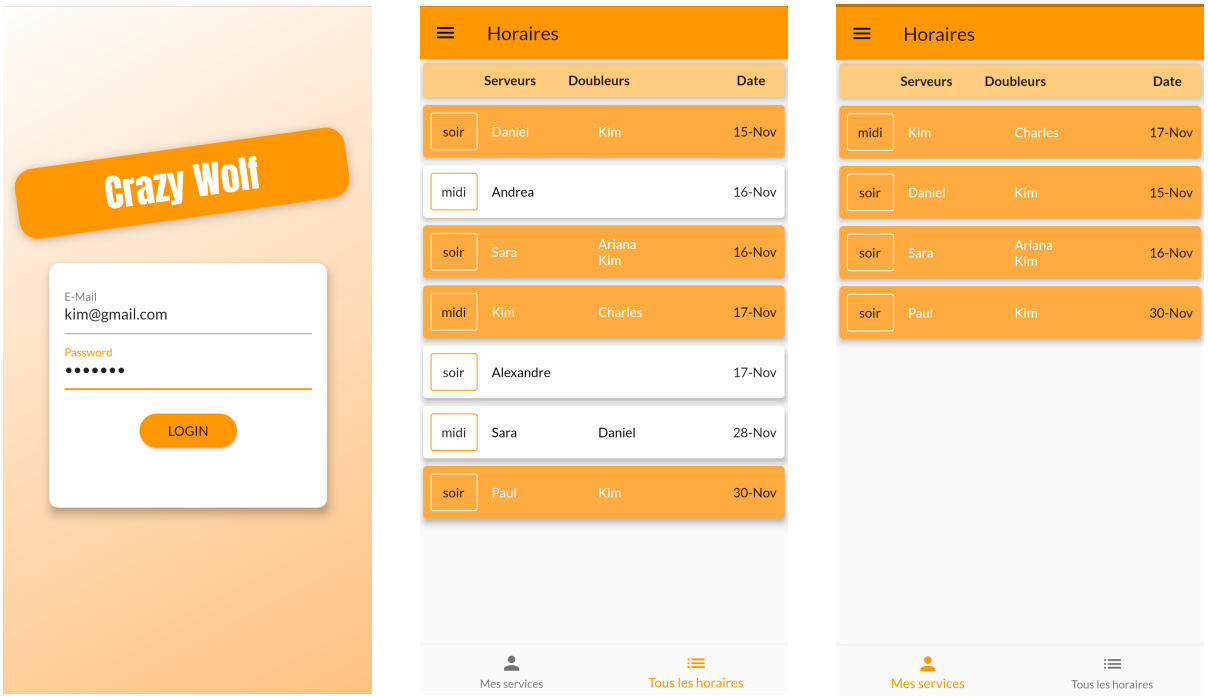
# Présentation de l'application

Dans ce chapitre, les scénario les plus communs d'utilisation de l'application Crazy Wolf sont présentés au travers d'images commentées.

## 1.1 Scénario I

### Authentification & écran d'accueil

Ici la serveuse Kim, qui n'a pas de privilèges, c'est-à-dire qu'elle n'est ni manager ni administratrice, se connecte et accède au premier écran possible. Puis elle, clic sur *Mes services*.




(a) login (b) horaires (c) mes horaires

FIGURE 1.1 – scénario I

L'utilisateur doit dans un premier temps se connecter à l'aide d'identifiants déjà existant dans l'écran 1.1a. Une fois l'adresse mail et le mot de passe saisis, l'écran *Horaires* 1.1b s'affiche.

On y voit l'ensemble des horaires de travail. Les services sont définis par
- le type : midi ou soir.
- un ou plusieurs serveurs.
- zéro, un ou plusieurs doubleurs.
- la date

Les services sont ordonnés par date, les jours précédents au moment de la connexion ne sont pas affichés. Les horaires correspondant à la personne authentifiée sont de couleurs orange.

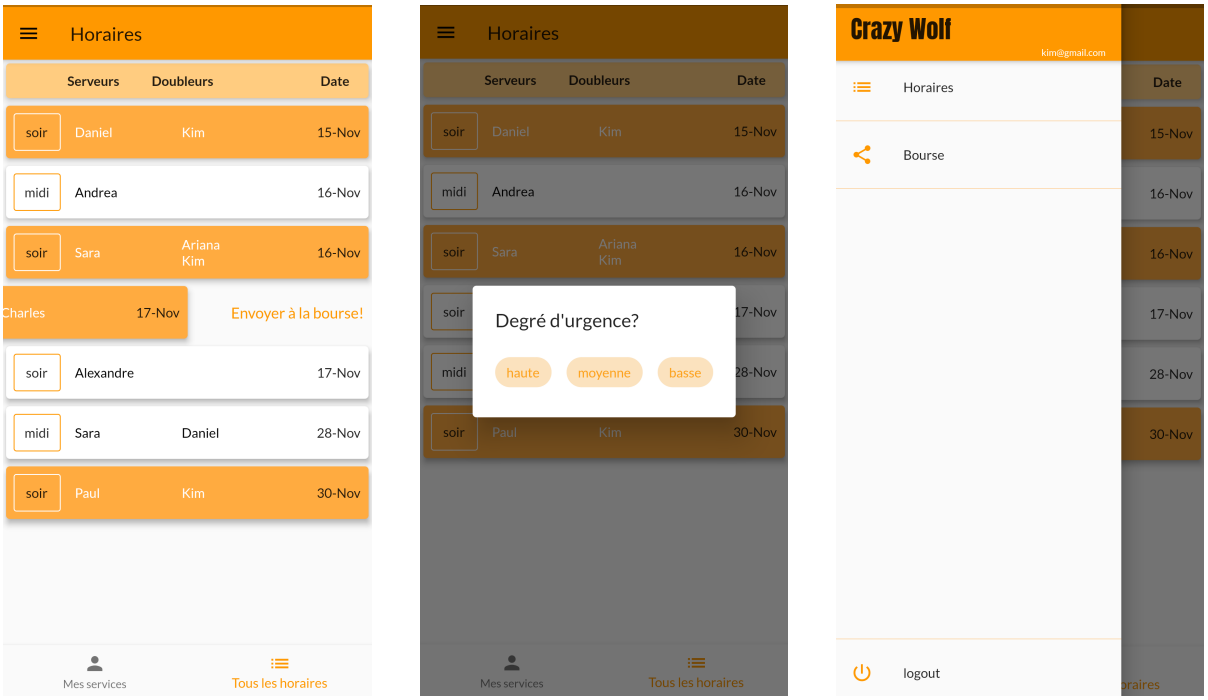
Les services sont affichés sous forme de liste scrollable.

On peut également naviguer à l'aide du menu inférieur à *Mes services* où seuls les horaires du serveur authentifié sont affichés. Comme on le voit dans 1.1c

## 1.2 Scénario II

### Mise en bourse d'un service

Supposons que Kim, la serveuse authentifiée ne puisse pas travailler le 17 novembre. Elle souhaite donc se faire remplacer. Pour se faire, dans l'onglet *Horaires*, elle peut mettre son service en bourse en glissant le service en question sur la gauche.




(a) mise en bourse (b) urgence (c) menu sans privilèges

FIGURE 1.2 – scénario II - a

Une fois le glissement effectué un popup est s'affiche 1.2b demandant à quel point le remplacement est urgent. Une fois que l'utilisateur à répondu, le service est mis en bourse. Un snakbar [1] s'affiche pour le notifier que l'action à réussi.

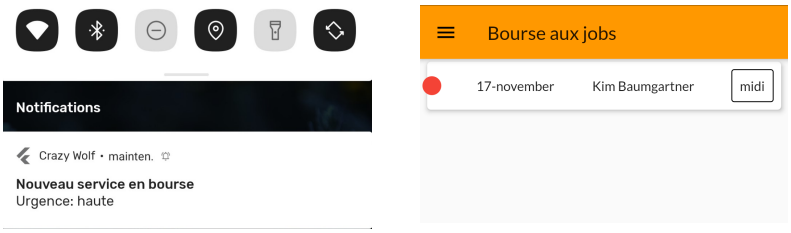
Si l'action est réussie, tous les utilisateurs de l'application sont notifiés 1.3a comme quoi un nouveau service est disponible. La notification informe sur la disponibilité d'un service ainsi que l'urgence requise à y répondre.

Suit à ça, l'utilisatrice peut naviguer à l'aide du menu latéral 1.2c où s'affichent les options de navigation suivante :

— Horaires : Pour aller à l'écran *Horaires* 1.1b
— Logout : Pour se déconnecter, qui renvoie à l'écran 1.1a d'authentification
— Bourse : Pour aller à l'écran *Bourse aux jobs* 1.3b

1. Petite notification informative qui émerge dans la partie inférieur d'un écran

Dans l'onglet *Bourse aux jobs* 1.3b son service est disponible à toute personne souhaitant y postuler. Cet onglet est partagé parmi tous les utilisateurs de l'application.




(a) notification (b) bourse

FIGURE 1.3 – scénario II - b

La couleur représente le degré d'urgence. Ainsi la convention suivante est appliqué :

— rouge : implique une urgence élevée.
— jaune : implique une urgence moyenne.
— vert : implique une urgence basse.

L'onglet *Bourse aux jobs* affiche tous les service actuellement en bourse sous forme de liste scrollable. L'utilisateur ayant mis son service en bourse doit patienter à ce que quelqu'un y postule.

Les utilisateurs sans privilèges sont autorisé à mettre un service en bourse uniquement s'ils y travaillent.

## 1.3 Scénario III

### Postuler pour un service

Dans la continuation du scénario précédent, après avoir été notifiés, deux serveurs vont postuler au nouveau service mis en bourse. Leurs parcours étant identiques, nous allons en montrer qu'un. Supposons qu'ils se soient authentifié comme vu en 1.1a et qu'ils se trouvent dans l'onglet *Bourse au jobs* 1.3b.

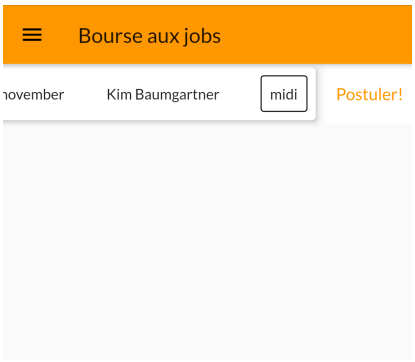
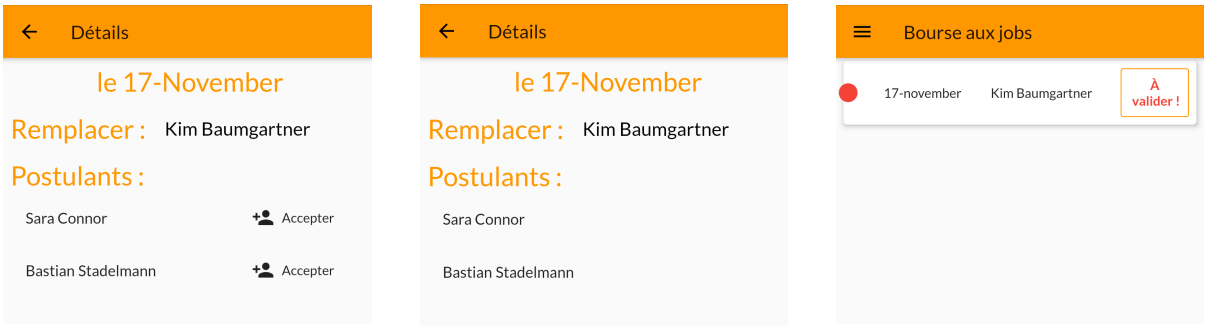



FIGURE 1.4 – postuler

Alors, ils peuvent glisser le service auquel ils souhaitent postuler sur la gauche 1.4. À nouveau, si l'opération réussi, un snackbar apparaît pour l'indiquer.

L'utilisateur ayant mis le service en bourse doit procéder à l'acceptation d'un seul postulant. Dans *Bourse aux jobs* 1.3b les éléments de la liste sont clicable. Lorsqu'un utilisateur clic deux options sont possibles :




(a) kim (b) autre (c) à valider

FIGURE 1.5 – scénario III

S'il s'agit de l'utilisateur ayant mis le service en bourse à l'occurrence Kim, l'écran *Détails* tel que 1.5a s'affiche.

S'il s'agit de n'importe quel autre utilisateur alors l'écran *Détails* s'affiche comme dans 1.5b

Ainsi, seul l'utilisateur ayant mis le service en bourse est en mesure d'accepter un ou une remplaçante.

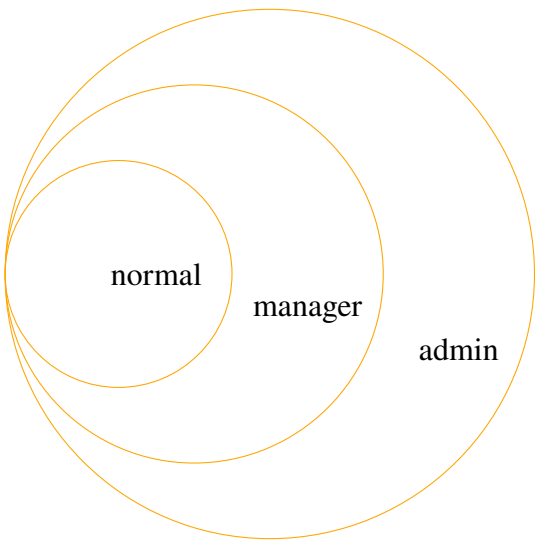
Pour se faire, Kim doit appuyer sur le bouton *accepter* de la personne de son choix. Elle accepte Sara. Tous les utilisateurs ayant des privilèges sont notifiés qu'un service nécéssite validation. En attendant, l'élément de la bourse dans *Bourse aux jobs* 1.5c s'affiche comme nécéssitant validation.

## 1.4 Scénario IV

### Valider un échange

Toujours dans la continuation de notre exemple, nous allons voir ici la validation d'un échange. En effet, même si un utilisateur a accepté un remplaçant pour son service. Il faut encore que cette transaction soit validée par un utilisateur avec des privilèges.

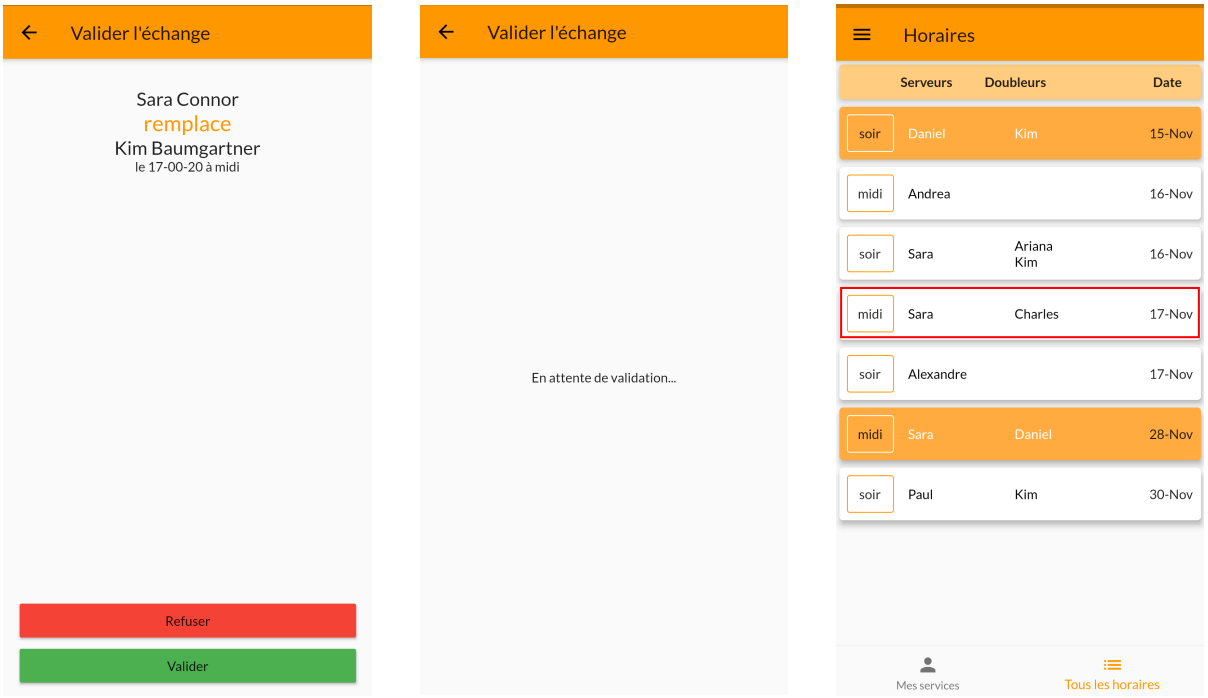
Il existe trois types d'utilisateurs :



Les utilisateurs autres que *normal* ont des privilèges.

Normal : Peut demander un échange et accepter des postulants.
Manager : Peut valider un échange, créer des services et demander des doubleurs en renfort.
Admin : Peut ajouter de nouveaux serveurs.

Lorsque l'on clic sur l'élément à valider 1.5c deux options sont possibles suivant si l'utilisateur authentifié 1.6a et *manager / admin* ou bien s'il est *normal* 1.6b.




(a) manager / admin (b) normal (c) Daniel : après validation

FIGURE 1.6 – scénario IV

S'il s'agit d'un utilisateur avec privilèges, alors il a l'option de valider l'échange 1.6a.

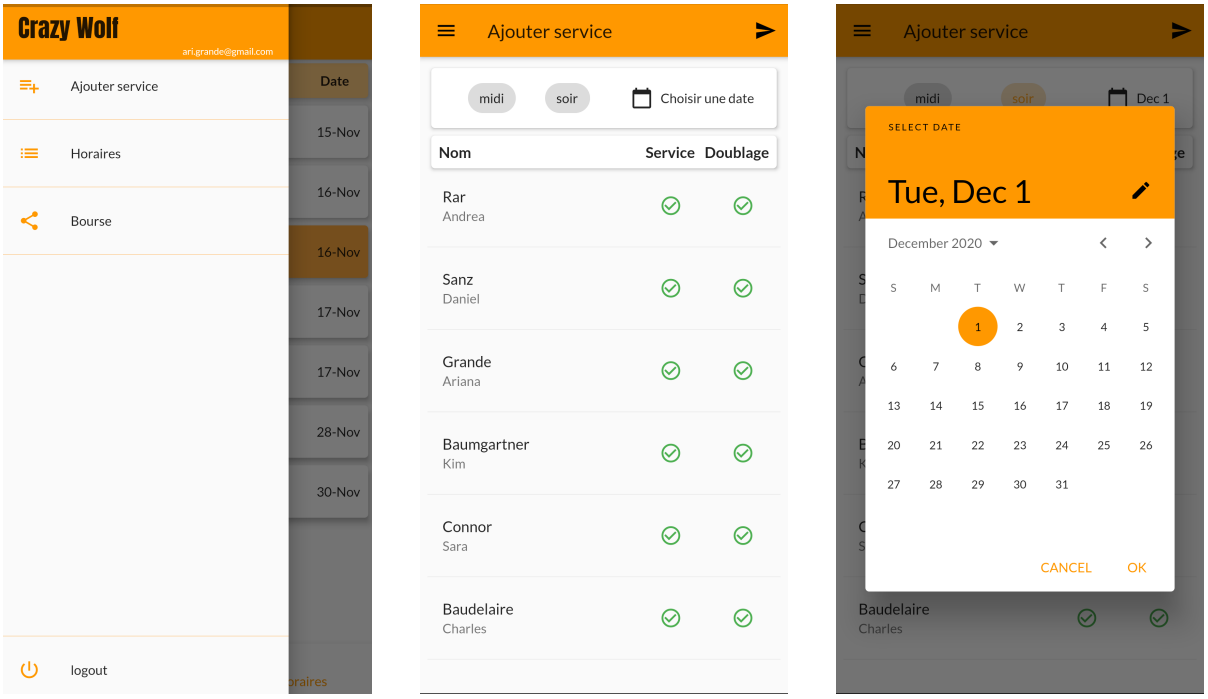
Supposons que Daniel, qui est administrateur valide l'échange. Alors, l'écran *Horaires* est modifié 1.6c pour tout le monde et affiche que c'est bien Sara qui travail le 17 novembre et non plus Kim.

De plus, l'élément dans *Bourse aux jobs* 1.3b est supprimé.

## 1.5 Scénario V

### Ajouter un service

Afin de créer les horaires pour le personnel, un manager ou administrateur doit créer des services. Supposons que c'est Ari, une manager qui le fait. Après s'être authentifiée, elle peut à l'aide du menu latéral 1.7a naviguer à *Ajouter service* 1.7b




(a) menu manager (b) ajouter service (c) choix date

FIGURE 1.7 – scénario V - a

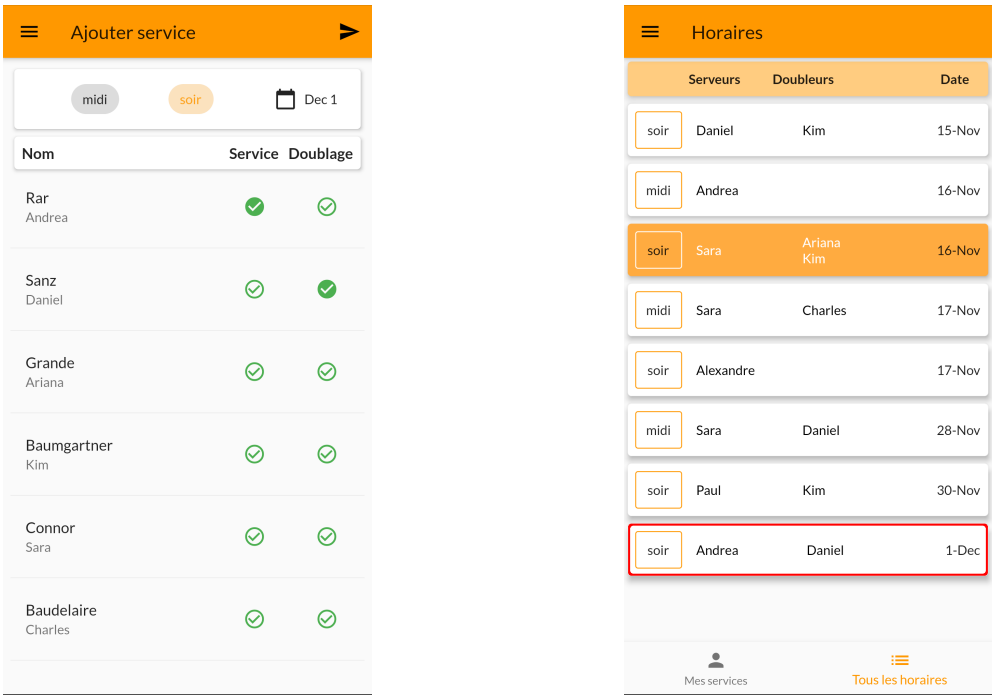
Pour créer le service, il faut choisir :

- une date en appuyant sur *Choisir une date* un popup avec le calendrier s'affiche 1.7c.
- un type en appuyant sur *midi* ou *soir* 1.8a.
- une ou plusieurs personnes au *Service* 1.8a.
- zéro, une ou plusieurs personnes au *Doublage* 1.8a.

Dans notre exemple Ari choisit le 1er décembre au soir. Andrea au service et Daniel au doublage.

Un service ne peut pas avoir la même personne au doublage et au service. Un service peut ne pas avoir de doubleurs.

De plus, l'opération ajouter le service aux horaires qui se fait au moyen du bouton envoyer dans le coin supérieur droit de la figure 1.8a retournera une erreur s'il existe un service du même type à la même date.




(a) choix du personnel et type

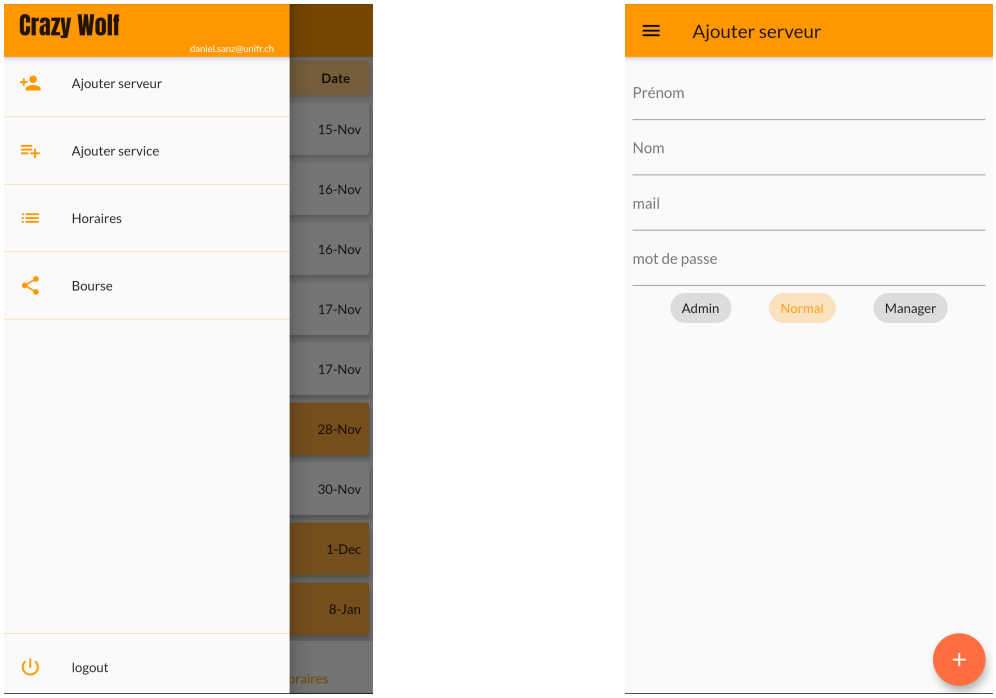
(b) nouveau service créé

FIGURE 1.8 – scénario V - b

## 1.6 Scénario VI

### Ajouter un serveur

Les utilisateurs ayant les privilèges administrateur ont la capacité d'ajouter de nouveaux serveurs.




(a) menu admin

(b) ajouter serveur

FIGURE 1.9 – scénario VI

Pour se faire, ils doivent après s'être authentifié, naviguer à l'aide du menu latéral 1.9a à l'onglet *Ajouter serveur*

Dans l'onglet *Ajouter serveur* il faut remplir les champs définissant un serveurs. I.e.

- Prénom
- Nom
- Adresse mail
- Mot de passe
- Niveau de privilèges : administrateur, normal, manager.

Le niveau de privilège le plus courant étant *normal* il est déjà pré-sélectionné.

Le mot de passe doit être strictement plus long que 6 caractères.

Utiliser la même adresse mail pour deux serveurs différents est interdit. Cependant, deux serveurs peuvent théoriquement avoir le même prénom et le même nom de famille.

Un fois tous les champs remplis, l'administrateur peut ajouter le serveur dans le système en appuyant sur le bouton + dans le coin inférieur gauche de l'écran *Ajouter serveur* 1.9b

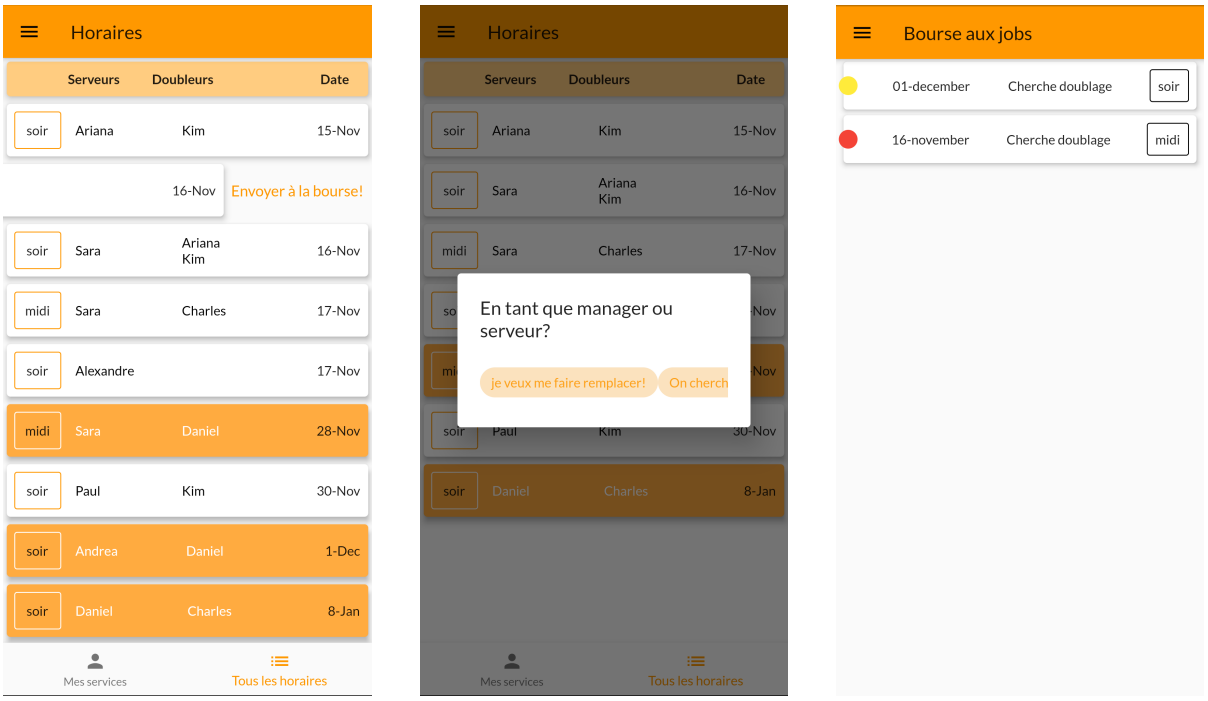
Ce sont les indetifiants nécéssaires pour s'authentifier dans l'écran 1.1a.

## 1.7 Scénario VII

### Doublages

Ici on va voir comment un administrateur ou un manager peut demander du personnel supplémentaire pour un service. Ces personnes supplémentaires sont dites doubleurs et font un doublage.

La démarche est similaire à celle d'un échange.




(a) demande doublage (b) remplacement ou doublage (c) bourse doublage

FIGURE 1.10 – scénario VII

Dans un premier temps un utilisateur avec privilèges fait glisser un service sur la gauche 1.10a. Deux options sont possibles suivant si l'utilisateur travaille dans ce service ou pas.

— S'il y travaille alors un popup apparaît 1.10b pour demander si c'est bien une demande de doublage ou bien se faire remplacer
— S'il n'y travaille pas alors la demande de doublage est directement envoyée en bourse.

Dans cet exemple, deux services sont mis en bourse. Un où l'administrateur n'y travaille pas 1.10a. Comme pour les échange, le degré d'urgence 1.2b est demandé. On peut voir le service du 16 novembre en bourse. Et l'autre où l'administrateur travaille et où il choisit 1.10b qu'il cherche un doublage.

À la différence du scénario III 1.5, la condition pour pouvoir accepter des postulants n'est plus d'être l'utilisater ayant mis le service en bourse mais d'avoir un niveau de privilèges supérieur à *normal*.

CHAPITRE 2

# Eléments de programmation

Dans ce chapitre sont présentés les técnologies, paradigmes et técniques liés à l'implémentation informatique de l'application. Notamment, lesquelles ont été utilisées, comment elles fonctionnent et, une à une, comment cela c'est traduit dans l'implémentation final de l'application. Les exemples de codes sont tirés de l'application.

## 2.1 Choix des technologies

Le développement d'applications pour smartphones est depuis un peu plus d'une décennie en pleine effervescence. Il existe, par conséquent, une multitude de frameworks, services, langages, méthodologies et paradigmes liés à leur développement.

Ces technologies aux noms exotiques, aux logos plus brillants les uns que les autres et aux conférences accrocheuses qui leur sont dédiées, font que leurs différences relèvent plus d'une stratégie marketing visant les informaticiens, réussie que des attributs intrinsèques de la technologie en question.

L'application étant d'une complexité modérée et ne demandant pas de ressources importantes comme tel pourrait être le cas pour une messagerie instantanée à grand échelle ou une application utilisant abondamment un domaine spécifique de connaissance comme le machine learning, le traitement d'images, les jeux vidéos, etc. Exclu *de facto* le choix d'une technologie basée exclusivement sur les performances ou sur le développement natif.

N'ayant jamais fait cela auparavant, il n'y a aucune préférence de ma part pour telle ou telle technologie.

Ces constats donnent lieu aux critères de sélection suivants :

- Développement cross-plateforme
- Simplicité
- Apprentissage d'un langage plutôt qu'une multitude
- Vaste documentation et ressources d'apprentissage

Le premier critère est celui qui réduit le plus la liste des possibilités. En effet, les frameworks permettant le développement d'applications pour Android et IOS ne se comptent pas en grand nombre. Il existe [1] :

- Xamarin - Microsoft
- React Native - Facebook
- Flutter - Google
- Adobe PhoneGap - Adobe
- Ionic - MIT

Le choix parmi ces possibilités découle essentiellement de l'arbitraire. Toutefois, Ionic a été exclu car il est nécessaire de maîtriser HTML5 et par conséquent CSS mais encore Angular JS. Ce qui contredit le 3ème critère.

React native a été exclu pour des raisons similaires. I.e. l'apprentissage de divers langages.

Finalement, suite à un cours de Academind d'une durée de 40 heures portant sur les aspects les plus basiques du développement jusqu'au déploiement de l'application en passant par le routage, la gestion de requêtes http, la connexion à tout un écosystème de bases de données, l'utilisation de caméra et géolocalisation, et même sur comment changer le logo de l'application, le choix s'est porté sur Flutter.

1. Liste non exhaustive

Flutter est un framework crée par Google. Ce dernier offre, pour les applications, le service Firebase qui englobe :

- Cloud Firestore
- Real time database
- Functions
- Machine learning
- Cloud messaging
- ...

Firebase s'intègre, par conception, particulièrement bien et facilement à Flutter. Même s'ils sont indépendants, leurs utilisation conjointe forme un seul écosystème plus facile à appréhender. Ainsi pour la base de données, le choix a été la Real time database. Car cette dernière fournit une API REST.

De plus, toujours dans cet ecosystème, *Cloud messaging* est utilisé pour l'envoi des notifications et *Functions* pour effectuer des actions côté serveur lorsque la base de données subit des modifications.

## 2.2 Flutter

Flutter est un *Software Developpement Kit* (SDK) développé par Google permettant de concevoir des applications pour plusieurs plateformes. Notamment, Android et IOS avec un seul code source.

Fluter est aussi et surtout un framework. Il se caractérise par le fait qu'il va principalement dessiner des éléments à l'écran en se basant sur les pixels. Ainsi, il n'utilise pas, de base, de libraries natives. Par exemple, pour dessiner un bouton, il ne va pas faire appel au primitives bouton dans Android ou IOS mais va dessiner pixel par pixel l'objet souhaité.

### 2.2.1 Dart

Dart est le langage de programmation avec lequel certains éléments du framework ont été développé mais il s'agit surtout du langage dans lequel on implémente une application en Flutter.

C'est un langage orienté objet avec un syntaxe de type C à l'image d'autres langages comme Java ou C++. Sa syntaxe est proche du Java.

Il partage certaines fonctionnalités propres au langages fonctionnels. Notamment, l'inférence de type ou encore certaines fonctionnalités comme les *map*, *functors* ...

En Dart, l'allocation de mémoire est gérée automatiquement et sa libération est faite par un garbage collector.

Voici un exemple minimal d'un *Hello world* en Dart :

```
1 void main() => print('Hello, World!');
```

Listing 1 – Hello world

### 2.2.2 Architecture

Pour mieux comprendre les différents éléments de programmation qui vont suivre, il est intérressant de s'attarder un peu sur l'architcture interne du framework. Flutter a un architecture en couches (*layers*).

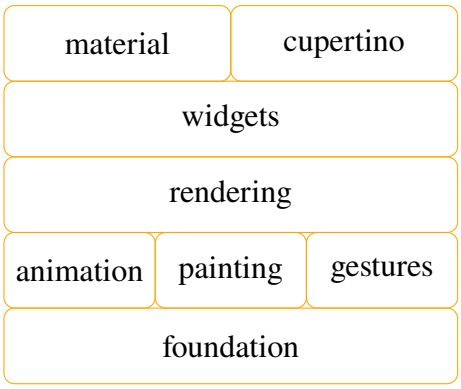



FIGURE 2.1 – Layers du Framework Flutter

Le *layer foundation* offre les clases de bases du framework sans avoir à tenir compte de l'*engine* écrit en C++ qui se trouve en dessous. Il offre aussi, au moyen de *animation*, *painting* et *gestures* les bases pour l'animation, le dessin et la detection de mouvement sur l'écran.

Le *layer rendering* offre une abstraction permettant de gérer l'affichage des éléments. Il va construire des objets (dérivés des widgets principalement) organisés sous forme d'arbre dynamique qui peuvent être affichés à l'écran.

Le *layer widgets* est une abstraction au dessus de *rendering* qui respecte le pattern de conception composite. Ainsi, chaque objet dans le *layer rendering* lui correspond une classe dans le *layer widgets*.

Le *layer material* et *cupertino* sont un ensemble de classes prédéfinies et personalisables qui implémentent tout un catalogue de widgets avec respectivement le style material design d'Android et le design propre à IOS.

### 2.2.3 Les widgets

Les widgets dans Flutter sont les éléments avec lesquels le programmeur à le plus de contact. Ils sont en effet, le composant principal d'une application Flutter. Voici un exemple :

```
1            Text(
2                  "Crazy Wolf",
3                  style: TextStyle(fontSize: 26, fontFamily: "Anton"),
4                ),
```

Listing 2 – Widget Text

Comme le code le sous-entend, les widgets peuvent s'encapsuler les uns dans les autres. Ainsi, Le widget *Text* accepte dans sont constructeur une chaine de caractères "Crazy Wolf" et un style qui est lui-même un widget *TextStyle*. À son tour, il accepte, entre autres, une taille et une police d'écriture en argument.

Cette encapsulation n'est pas un choix arbitraire mais trahi, comme vu auparavant, l'architecture du framework. En effet, Le *layer rendering* est un arbre dynamique de widgets (pas seulement) et le *layer widgets* respecte le pattern composite. Ainsi, il est naturel que les widgets se composent.

On en déduit que Flutter est un framework déclaratif plutôt qu'impératif. En effet, plutôt que de modifier des instances d'objets existants afin de varier l'affichage, Flutter construit de nouvelles instances de widgets selon les besoins.

La philosophie sous-jacente du framework est de composer de petits widgets aux fonctionnalités de bases afin de créer une fonctionalité complexe.

Le terme widget englobe une multitude de verbes. Un widget peut ne pas avoir de représentation en tant que tel mais peut offrire un positionement, une transformation, une taille, interaction ave l'utilisateur, animation, ...

Par exemple, le widget *Center* centre un autre widget en argument au centre de l'écran, dans la mesure de l'espace restant disponible.

```
1            Center(child: Text(...),),
```

Listing 3 – Widget Center

Les widget sont des classes immutables, c'est-à-dire qu'une fois l'objet instancié, il n'est plus possible d'en modifier le contenu. Il existe essentiellement deux types de widgets dans Flutter : ceux qui on un état : *StatefulWidget* et ceux qui n'ont en pas : les *StatelessWidget*.

### StatelessWidget

Ce sont les widgets dont les propriétés ne varient pas dans le temps. Par exemple :

```
1            import 'package:flutter/material.dart';
2            class ScreenSplash extends StatelessWidget {
3                @override
4                Widget build(BuildContext context) {
5                    return Scaffold(body: Center(child: CircularProgressIndicator(),),);}}
```

Listing 4 – Exemple *stateless*

Ici, *ScreenSplash* hérite de *StatelessWidget*. Afin, de determiner la représentation visuelle d'un widget, il faut *override* la méthode *build()* de *StatelessWidget*. Ainsi, *ScreenSplash* retourne un *Scaffold* qui est un écran d'affichage basique respectant le style material design. À son tour, *Scaffold* a une bare de progression circulaire (le *CircularProgressIndicator*) au centre (*Center*) de l'écran.

### StatefulWidget

Ce sont les widgets dont les propriétés peuvent varier dans le temps. Ces widgets ont un état, or ils sont immutable. C'est pourquoi ils sauvegardent l'état dans une classe séparée sous-classe de *State*.

Quand l'état change à l'aide de l'appel de fonction : *setState()*. L'interface est mise-à-jour. En guise d'exemple voici une partie du code source de l'écran *Horaires* 1.1b. Notamment, le petit menu inférieur permettant d'afficher tous les horaires où seulement ceux de l'utilisateur authentifié.

```
class ScreenHoraire extends StatefulWidget {
    static const routName = "/horaires";
    @override
    _ScreenHoraireState createState() => _ScreenHoraireState();}

class _ScreenHoraireState extends State<ScreenHoraire> {
    int _selectedIndex = 1;
    DateTime _selectedDate = DateTime.now();

    void _onItemTapped(int index) {setState(() {_selectedIndex = index;});}
```

Listing 5 – Screen Horaires

L'écran *Horaire* hérite d'un *StatefulWidget*. Dans la classe d'état il y a deux attributs : *selectedIndex* et *selectedDate*. Seul le premier est pertinent pour l'exemple. L'indice selectionné a pour valeur 1. Si la fonction *onItemTapped(int index)* est appellée avec un entier en argument alors, à l'aide de la fonction *setState* l'état est modifié et la fonction *build()* de *ScreenHoraireState* est appelée.

```
@override
Widget build(BuildContext context) {//...
    return Scaffold(//...
    bottomNavigationBar: BottomNavigationBar(
        onTap: _onItemTapped,
        currentIndex: _selectedIndex,
        items: const [
            BottomNavigationBarItem(icon: Icon(Icons.person), title: Text("Mes services"),),
            BottomNavigationBarItem(icon: Icon(Icons.list), title: Text("Tous les horaires"),)
            ]),
```

Listing 6 – Build dans horaires

Scaffold accepte un menu de navigation au bas de la page. Celui-ci est un widget : *BottomNavigationBar*. Ce widget prend en construction une fonction qui est appelée lorsque une tap a été detecté sur l'un des deux icons définis dans la liste *Items* ainsi que l'index actuellement selectionné. Le premier élément de la liste i.e. "Mes services" a pour index 0. Et "Tous les horaires" 1. Par défaut, *selectedIndex* vaut 1. C'est donc tous les horaires qui sont affiché. Lorsque l'utilisateur clic su l'icon de la personne, *BottomNavigationBar* va appeler la fonction *onItemTapped* définie dans le code 5 avec l'index de l'icon en argument. *onItemTapped* va changer l'état du widget et par extension la valeur de *selectedIndex*. L'état ayant changer la fonction *build* est appelée et donc *BottomNavigationBar* redessiné avec un *currentIndex* potentiellement différent.

### 2.2.4 Flux des données

**Par constructeur**

**Provider & Consumer**

## 2.3 La base de données

### 2.3.1 La méthodologie REST

### 2.3.2 REST appliqué à Dart

## 2.4 Notifications

## 2.5 Sécurité

### 2.5.1 Authentification

### 2.5.2 Firebase rules