L'application Crazy Wolf

Application smartphone pour la gestion des horaires et des services du personel du restaurant Crazy Wolf

TRAVAIL DE BACHELOR

DANIEL SANZ

Décembre 2020

Supérvisé par :

Prof. Dr. Jacques Pasquier

et

Pascal Gremaud

Software Engineering Group

UNIVERSITÉ DE FRIBOURG
UNIVERSITÄT FREIBURG

Groupe Génie Logiciel
Département d'informatique
Université de Fribourg (Suisse)

Table des matières

Table des figures

List of Listings

CHAPITRE 1

Introduction

La bonne gestion des ressources dans une entreprise est inaliénable à son bon fonctionnement. Le personnel et son horaire de travail l'est, par extension, également.

Un petit restaurant aurait du mal à investir dans un **ERP** [1] pour gérer l'attribution des horaires de son personnel. De plus, ceux-ci sont conçu pour une certaine stabilité et non pas pour des changements très fréquents.

En collaboration avec le restaurant du Crazy Wolf, à Fribourg, j'ai développé une application pour gérer les horaires et les échanges de services du personnel.

1.1 Motivation et Objectifs

Motivation

J'ai travaillé un an dans le restaurant du Crazy Wolf.

En ce qui concerne les serveurs, il y en a beaucoup. Environ une quinzaine. Pendant cette année de travail j'ai constaté que les serveurs, étant majoritairement étudiants, avaient, pour la plupart, de petits pourcentages d'occupation. Un ou deux services par semaine. Par service en entend la tranche horaire de travail de midi ou du soir. De par la jeunesse du personnel et des occupations annexes que les serveurs ou serveuses ont, les échanges sont très fréquents.

Actuellement le restaurant du Crazy Wolf utilise une messagerie instantanée pour gérer la distribution d'horaires, les échanges entre les serveurs, les imprévus, . . . Et ce, avec les inconvénients qui s'imposent : les demandes de remplacements se perdent dans l'historique de conversation, du personnel oublie de se présenter, le même message et répété plusieurs fois, entre autres aléas.

Ainsi, l'idée d'une gestion centralisée de l'horaire et des échanges de services m'est venu. En discutant avec plusieurs membres du personnel et avec les patrons, ils ont confirmé que cette idée répondait à une problématique réelle.

Objectifs

Créer une application disponible sur Android et IOS pour la gestion des horaires et des échanges. L'objectif est mesurable. En effet, si la messagerie instantanée n'est plus ou très peu utilisé alors la création d'une application se justifie.

1. Entreprise Resources Planning

1.2 Structure du rapport

Chapitre I : Introduction

L'introduction est le discours préliminaire de ce rapport.

Chapitre II : Aspects métier

Dans ce chapitre se trouve l'analyse de la problématique d'un point de vue métier. C'est-à-dire, les conditions d'utilisation, le type d'utilisateurs, définitions claires d'utilisation. Précisions sur le résultat recherché.

Chapitre III : Présentation de l'application

Dans ce chapitre l'application est présentée. Son mode de fonctionnement ainsi que des captures d'écran pour décrire plusieurs scénarios d'utilisations possibles.

Chapitre IV : Éléments de programmation

Dans ce chapitre se situe la partie technique en lien avec l'implémentation. On y retrouve la description du framework utilisé ainsi que les points clefs du développement, au travers d'extraits du code source commentés.

Chapitre V : Conclusion

Dans ce chapitre seront discutés les résultats du projet. De plus, j'y fais part de mon point de vue personnel.

CHAPITRE 2

Domaine métier

Dans le restaurant Crazy Wolf et dans la restauration en général, les employés sont susceptibles de ne pas travailler tous les jours ou de ne pas travailler toute la journée mais seulement certains "services".

Un service est une tranche horaire qui correspond aux heures pendant lesquelles le restaurant offre un service de restauration. Dans le cas concret du Crazy Wolf, il y a deux services : midi et soir. Qui englobent les heures de travail de 9h00 à 15h00 et de 17h00 à 23h00 respectivement. Dans ces tranches horaires sont incluses les heures nécessaires à la disposition du restaurant et d'autres préparations nécessaires avant l'arrivée de clients.

De plus, dans la restauration et surtout en ce qui concerne les serveurs, il est commun d'y trouver trois types d'acteurs avec différents statuts :

- serveur·euses
- manager
- patrons

La gestion, visualisation et organisation des services parmi ces trois acteurs est un élément vital pour un restaurant.

2.1 Analyse

D'après mon observation, la planification de l'horaire de travail des serveurs à un moment donné n'est pas invariante par rapport au temps. En d'autres termes, entre le moment où l'attribution de services aux serveurs est faite et le moment où un serveur y travaille effectivement il peut y avoir de grandes variations. J'ai constaté essentiellement trois facteurs susceptibles de perturber la planification initial :

- le désistement d'un·e serveur·euses
- les échanges
- la demande de renfort de la part du manager ou d'un patron.

Le premier peut être dû à diverses raisons : maladie, imprévu... De plus comme dans le Crazy Wolf, la majorité des serveur·euses sont étudiant·es les révisions et examens sont fréquemment cause de désistement.

Le deuxième est le fruit d'un mutuel accord entre deux serveurs pour échanger leurs services.

Le troisième est le fruit de l'analyse d'affluence des clients de la manager en accord avec les patrons. En effet, s'ils constatent que subitement l'afluence des clients augmente le jeudi soir et qu'il y a de fortes chances pour que ce soit aussi le cas le vendredi soir, il sera nécessaire de demander un ou des serveurs supplémentaires.

Les raisons de cette affluence augmentée peuvent être très diverses. Les conditions météo par exemple. Plus de gens vont au Crazy Wolf quand il pleut, ou en hiver. Ou encore des manifestations comme carnaval ou le marathon.

Dans le jargon propre au Crazy Wolf, ces serveurs supplémentaires sont appelés "doubleurs".

Ainsi, la gestion des horaires est dynamique et non statique.

2.2 Définition de la problématique

Actuellement, le Crazy Wolf utilise un groupe, contenant tout le personnel, d'une messagerie instantanée pour communiquer. Dans ce groupe sont traités toute sorte d'aspects. Depuis des félicitations d'anniversaires, jusqu'à l'envoi des horaires mensuels sous forme PDF en passant par des discutions d'échanges, de remplacements, de renforts ou encore sur des discutions relatives aux normes sanitaires dues à la pandémie du COVID-19.

Ainsi, lorsque la manager demande un renfort pour dans deux semaines et que personne ne se porte volontaire immédiatement, il y a de grandes chances que le message soit répété plusieurs fois.

Il est donc nécessaire de disposer d'une plateforme dynamique, standardisée, centralisée et intuitive dédiée exclusivement à la gestion des horaires.

Cette plateforme doit être accessible en tout temps depuis internet. Une application fonctionnant sur les deux principaux systèmes d'exploitation mobiles répond bien à ce besoin.

En somme, cette application doit fournir les fonctionnalités suivantes :

Login

L'application doit permettre aux utilisateurs de s'authentifier pour accéder à l'ensemble des fonctionnalités.

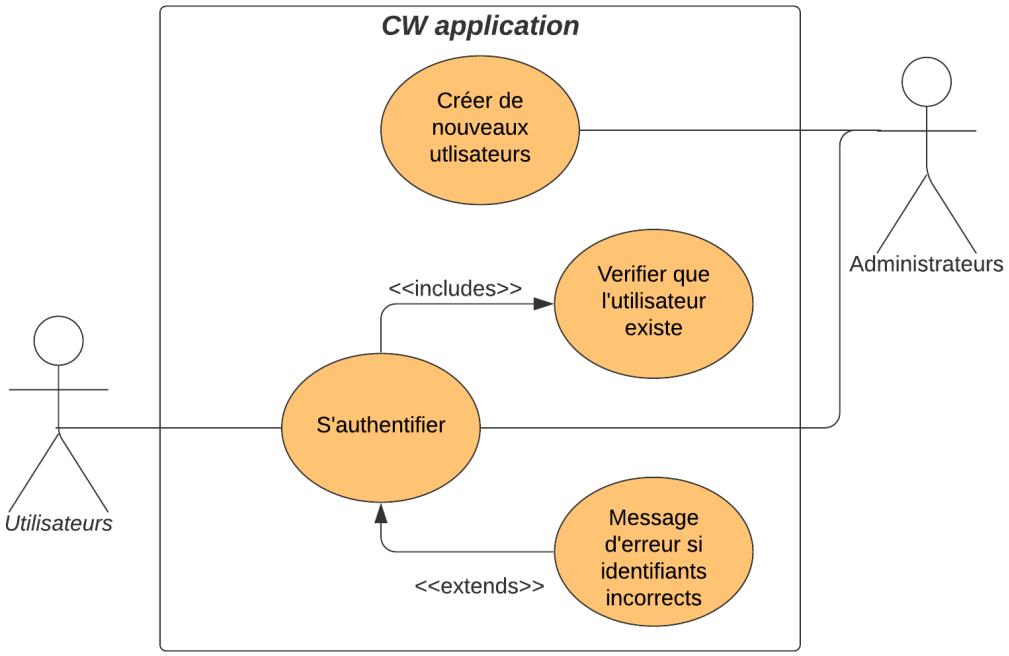


FIGURE 2.1 – use case : login

Pour que l'authentification soit validée, l'utilisateur doit posséder des identifiants qu'un administrateur à créer précédemment. Attention, administrateur doit aussi s'authentifier pour pouvoir créer de nouveaux utilisateurs.

De plus, les utilisateurs doivent être informés si leur saisie est incorrecte.

Visualiser les services

L'application doit permettre à tous les utilisateurs de consulter rapidement et facilement les services dans lesquels ils travaillent. De plus, ils doivent également pouvoir voir les services dans lesquels les autres utilisateurs travaillent. En effet, il n'y a là aucune information confidentielle.

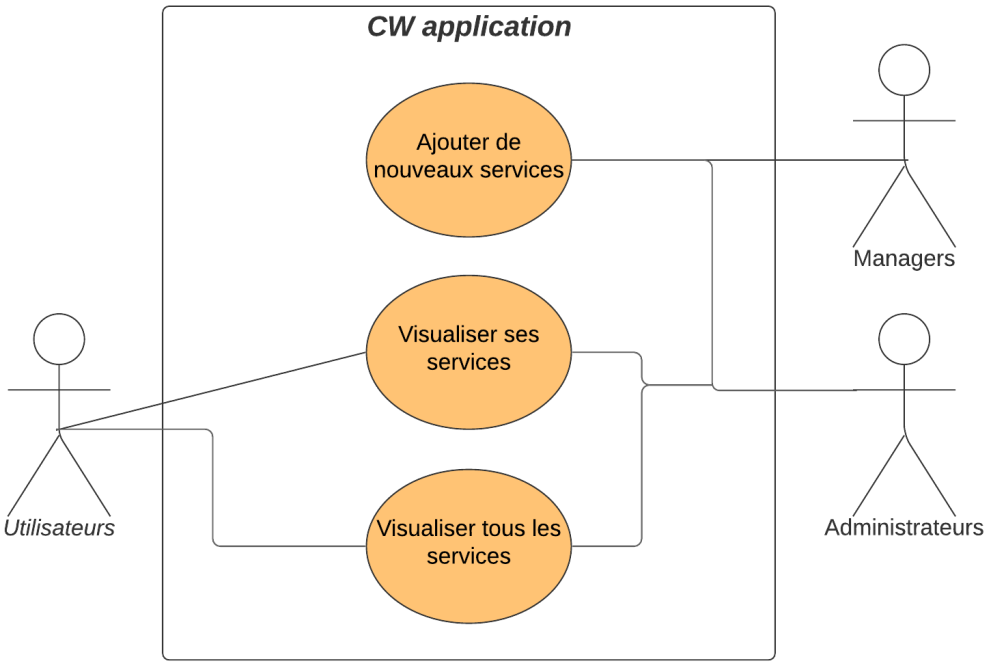


FIGURE 2.2 – use case : visualisation des services

Afin de pouvoir visualiser des services, il est nécessaire que préalablement, un manager ou un administrateur en ait ajouté dans le système. De plus, le niveau de permission n'exclut pas la possibilité qu'un manager ou administrateur aient des services dans lesquels ils travaillent. Ainsi, ils doivent également pouvoir les visualiser.

Mise en bourse

La mise en bourse d'un service est la fonctionnalité principale de l'application. L'application doit permettre à un·e serveur·euse de pouvoir mettre un service, où il ou elle ne peut pas travailler, en bourse afin que d'autres utilisateurs de l'application puissent le ou la remplacer.

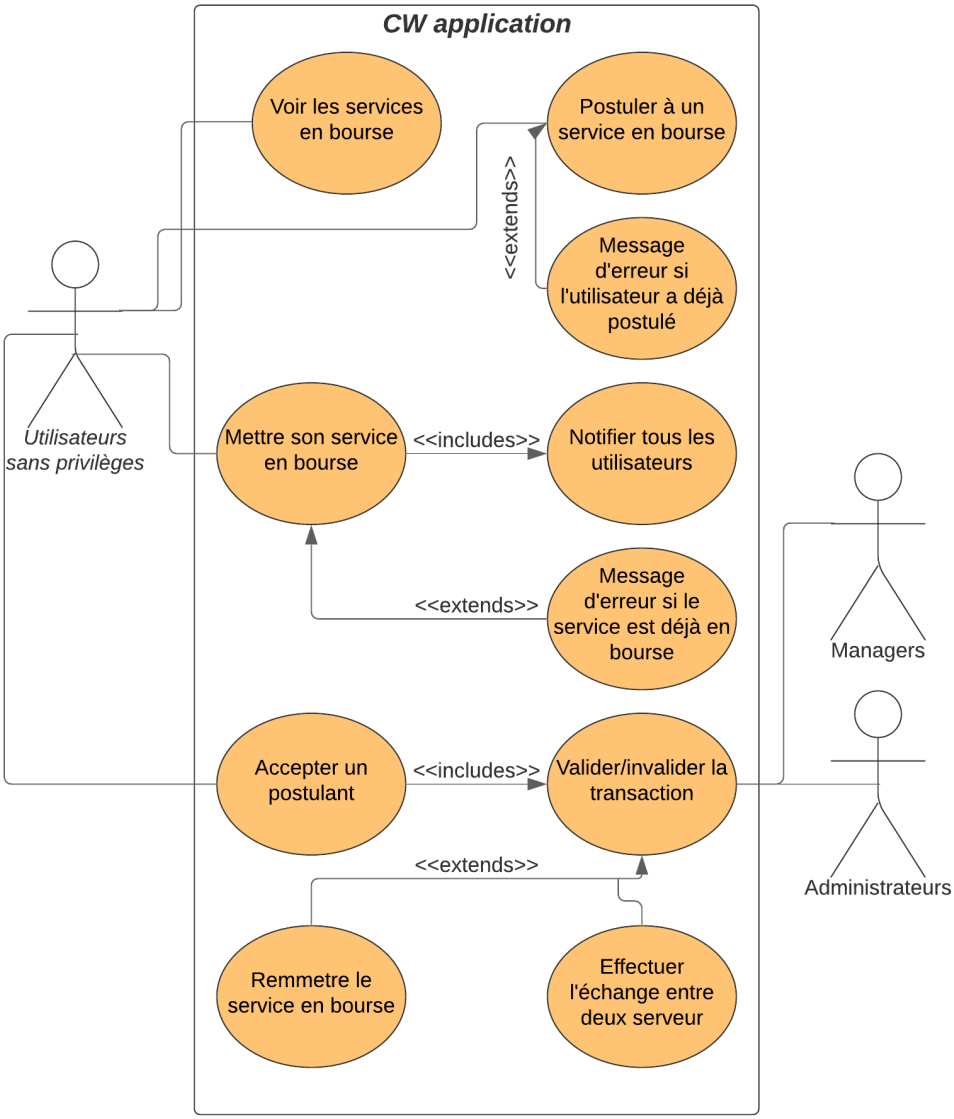


FIGURE 2.3 – use case : mise en bourse des services

Dans un premier temps, tout utilisateur (normal, manager ou admin) doit pouvoir visualiser les services en bourse. De plus, tout utilisateur doit pouvoir mettre son et uniquement son service en bourse. Ce qui engendre une notification à l'ensemble des utilisateurs de la disponibilité d'un nouveau service.

Tout utilisateur doit pouvoir postuler à un service. Cela même si le postulant et celui l'ayant mis en bourse sont la même personne.

Seul l'utilisateur, quel qu'il soit, ayant mis le service en bourse, peut accepter un postulant. Ceci fait, seul un manager ou un administrateur doit valider ou invalider la transaction. Si l'utilisateur ayant mis le service en bourse a des privilèges il peut s'auto-valider la transaction.

Afin de simplifier le diagramme, les restrictions d'un utilisateur normal ont été mise en exergue.

Demande de renfort

Pouvoir demander du renfort - des serveurs au doublage ou encore des doubleur - est la deuxième fonctionnalité *must be*.

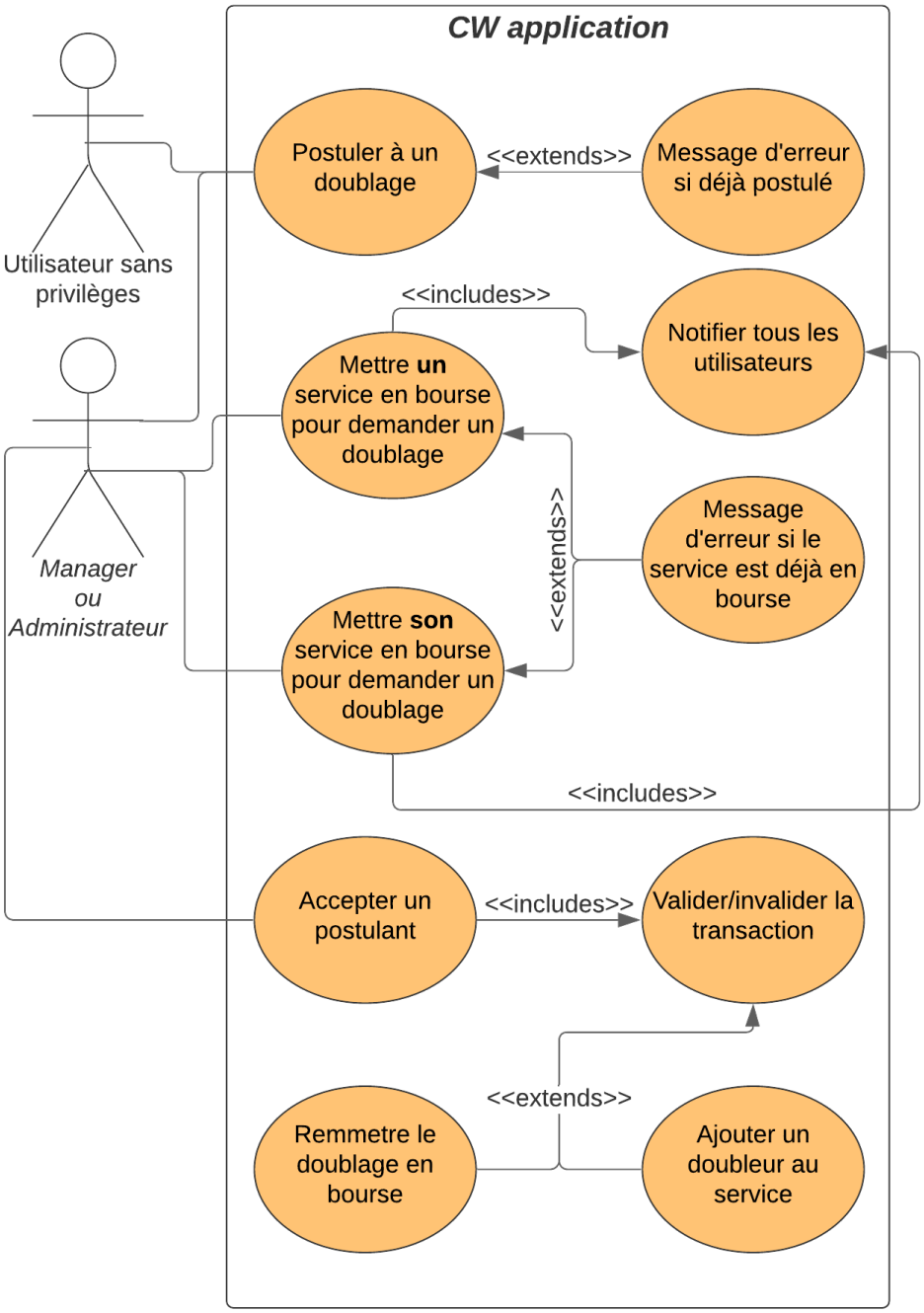


FIGURE 2.4 – use case : mise en bourse d'un doublage

Les serveur·euses sans privilèges ne peuvent que postuler pour des doublages déjà en bourse. Par contre, les utilisateurs avec privilèges peuvent compte à eux faire des demandes de doublage autant pour des services dans lesquels ils ne travaillent pas comme des services dans lesquels ils travaillent. Ils ont aussi la faculté d'accepter des postulants.

2.3 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.

Les fonctionnalités de l'application vues précédemment 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 vidéo, 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 et Xamarin ont été exclus pour des raisons similaires. I.e. l'apprentissage de divers langages.

Finalement, à la suite d'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.

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
- . . .

1. Liste non exhaustive

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 écosystème, *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.

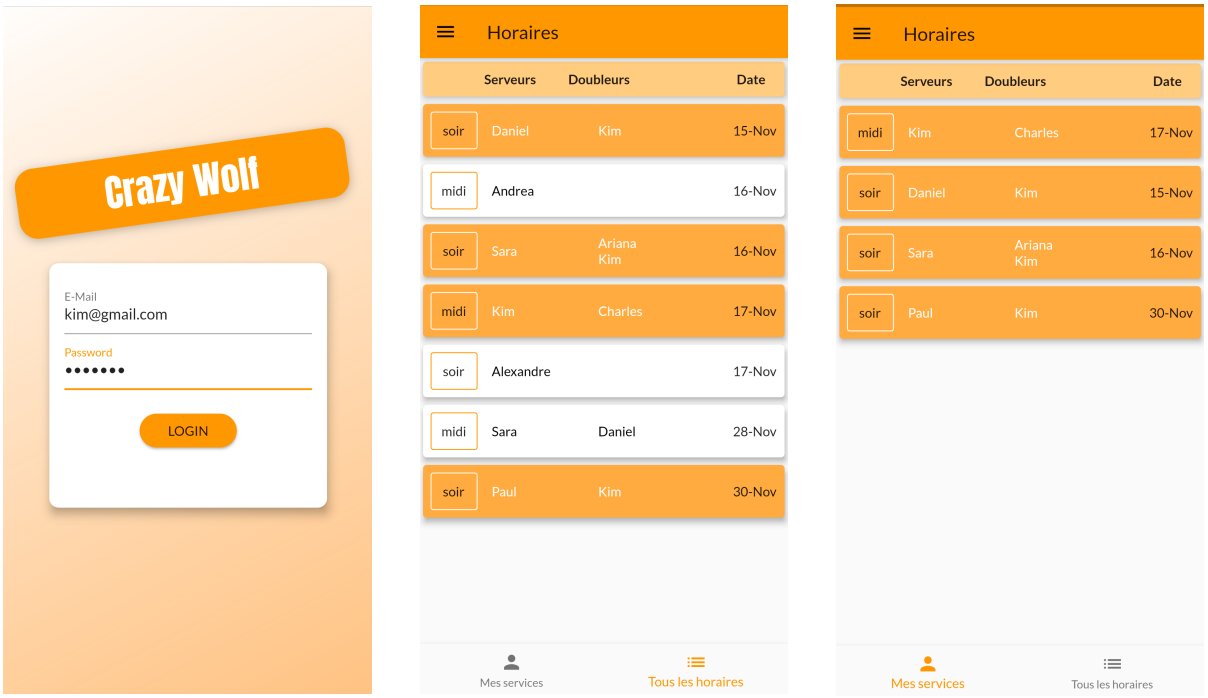
Présentation de l'application

Dans ce chapitre, les scénarios les plus communs d'utilisation de l'application Crazy Wolf sont présentés au travers d'images commentées. Ces explications ne demandent aucune connaissance en informatique. Elles servent ainsi comme mode d'emploi pour les utilisateurs finaux, par exemple.

3.1 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, clique sur *Mes services*.



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

FIGURE 3.1 – scénario I

L'utilisateur doit dans un premier temps se connecter à l'aide d'identifiants déjà existants dans l'écran 3.1a. Une fois l'adresse mail et le mot de passe saisis, l'écran *Mes horaires* 3.1c 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 correspondants à la personne authentifiée sont de couleur orange.

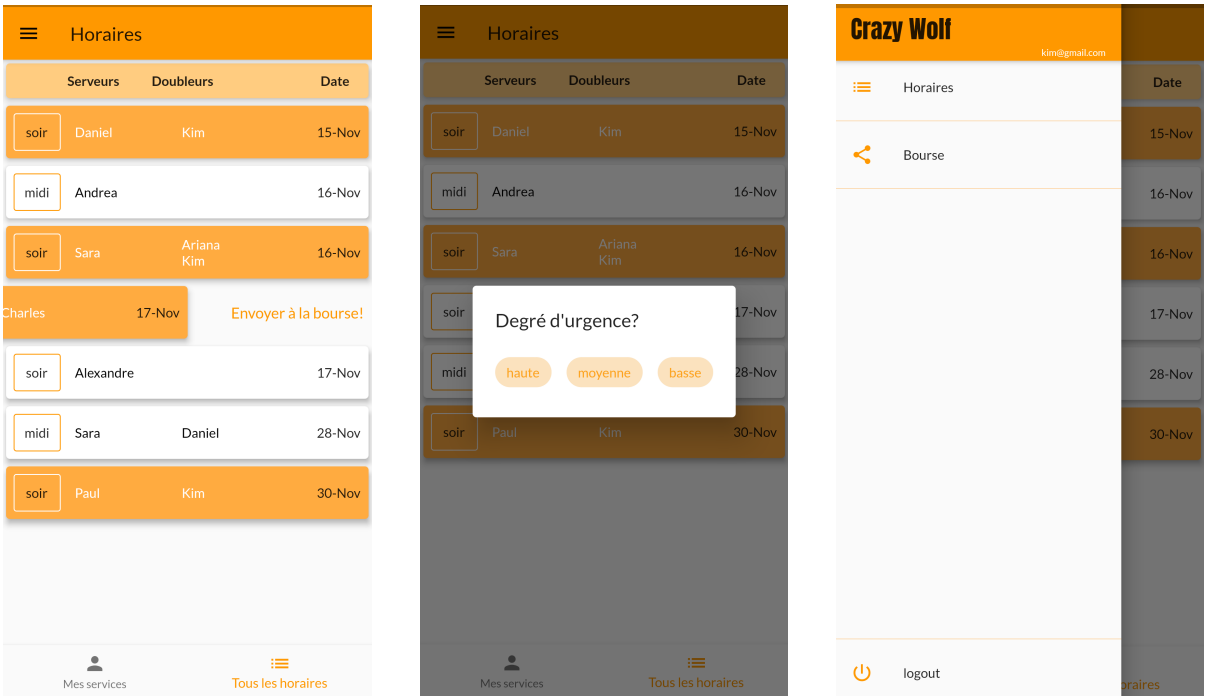
Les services sont affichés sous forme de liste qui peut être défilée avec le doigt.

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 3.1c

3.2 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 ce 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 3.2 – scénario II - a

Une fois le glissement effectué un *popup* s'affiche 3.2b demandant à quel point le remplacement est urgent. Une fois que l'utilisateur a répondu, le service est mis en bourse. Un snackbar [1] s'affiche pour notifier que l'action a réussi. Si l'action venait à échouer, par exemple à cause d'une déconnexion ou car le service se trouve déjà en bourse, un snackbar avec un message d'erreur pertinent s'afficherait également.

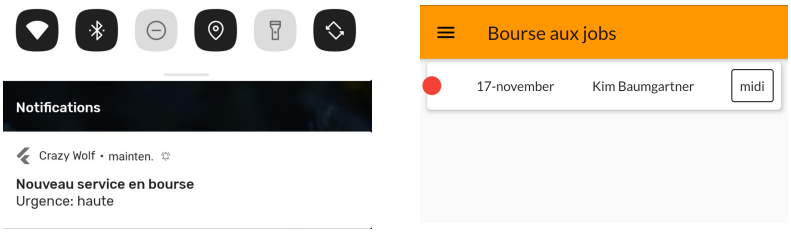
Si l'action réussit, tous les utilisateurs de l'application sont notifiés 3.3a qu'un nouveau service est en bourse. La notification informe sur la disponibilité d'un service ainsi que l'urgence requise à y répondre.

Suit à cela, l'utilisatrice peut naviguer à l'aide du menu latéral 3.2c où s'affichent les options de navigation suivantes :

— Horaires : pour aller à l'écran *Horaires* 3.1b
— Logout : pour se déconnecter, qui renvoie à l'écran 3.1a d'authentification
— Bourse : pour aller à l'écran *Bourse aux jobs* 3.3b

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

Dans l'onglet *Bourse aux jobs* 3.3b son service est disponible à toute personne souhaitant y postuler. Le contenu de cet onglet est partagé parmi tous les utilisateurs de l'application.



(a) notification (b) bourse

FIGURE 3.3 – scénario II - b

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

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

L'onglet *Bourse aux jobs* affiche tous les services actuellement en bourse sous forme de liste scrollable [2]. L'utilisateur ayant mis son service en bourse doit patienter à ce qu'un autre serveur y postule.

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

3.3 Scénario III

Postuler pour un service

Dans la continuation du scénario précédent, après avoir été notifiés, deux serveurs postulent au nouveau service mis en bourse. Leurs parcours sont identiques. Supposons qu'ils se soient authentifiés comme vu en 3.1a et qu'ils se trouvent dans l'onglet *Bourse au jobs* 3.3b.

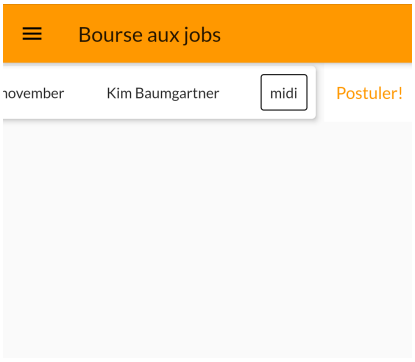
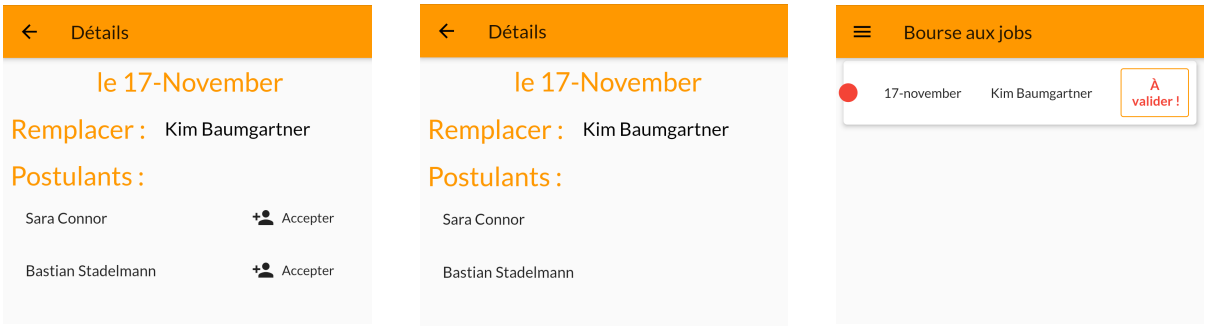


FIGURE 3.4 – postuler

Alors, ils peuvent faire glisser le service auquel ils souhaitent postuler sur la gauche 3.4. À nouveau, si l'opération réussit, un snackbar apparaît pour l'indiquer. Si au contraire, une erreur se produit, notamment car l'utilisateur a déjà postulé ou une erreur de connexion, un snackbar pertinent s'affiche aussi.

2. Scrollable : pouvant scroller, pouvant défiler

L'utilisateur ayant mis le service en bourse doit accepter un seul postulant. Dans *Bourse aux jobs* 3.3b les éléments de la liste sont cliquables. Lorsqu'un utilisateur clique, deux options sont possibles :



(a) kim (b) autre (c) à valider

FIGURE 3.5 – scénario III

S'il s'agit de l'utilisateur ayant mis le service en bourse en l'occurrence Kim, l'écran *Détails* s'affiche selon la figure 3.5a.

S'il s'agit de n'importe quel autre utilisateur, alors l'écran *Détails* s'affiche comme dans 3.5b

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

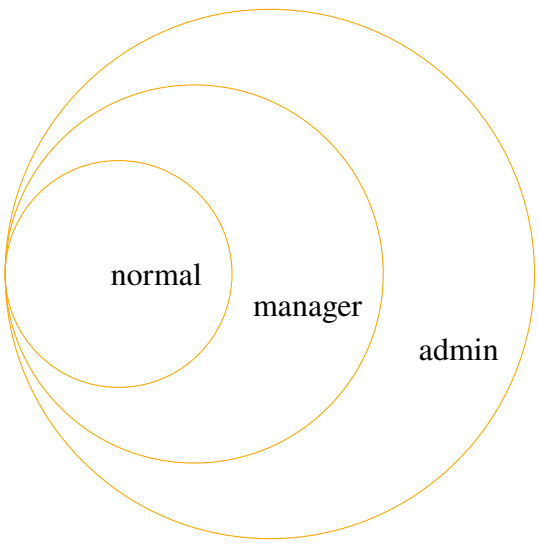
Pour ce faire, Kim doit appuyer sur le bouton *accepter* de la personne de son choix. Elle accepte Sara. Tous les utilisateurs ayant des privilèges sont notifiés qu'un service nécessite validation. En attendant, l'élément de la bourse dans *Bourse aux jobs* 3.5c s'affiche comme nécessitant validation.

3.4 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.

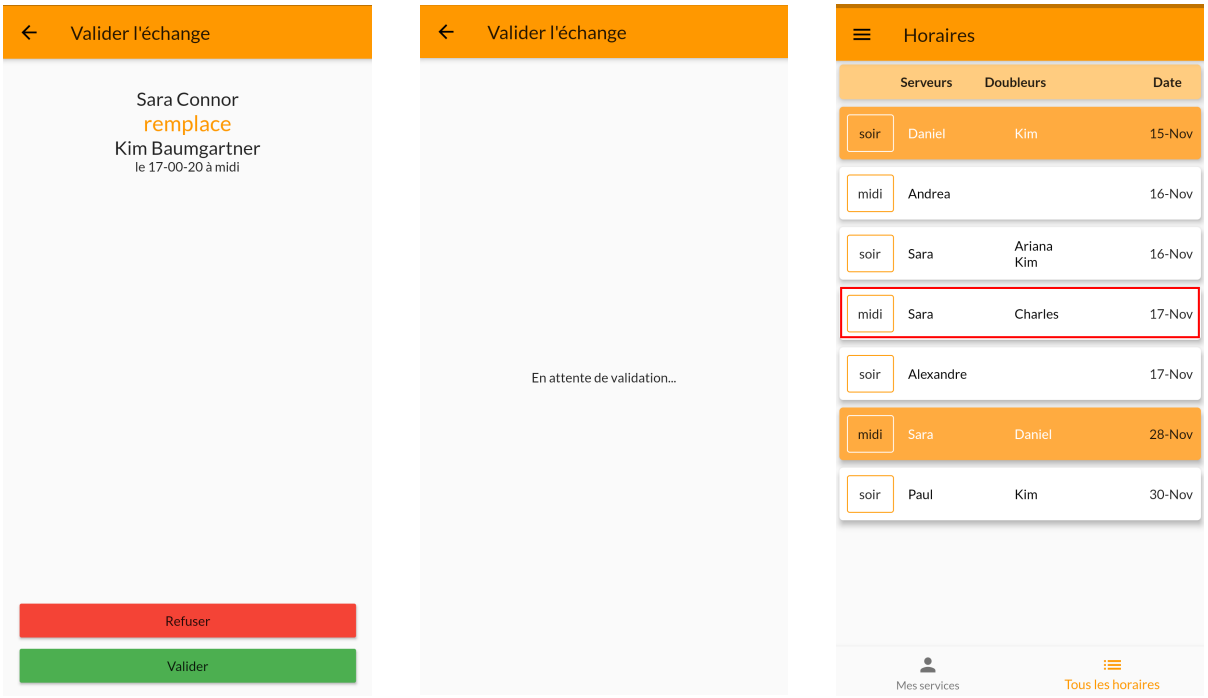
Il existe trois types d'utilisateurs :



Les utilisateurs dits *normaux* sont les seuls à ne pas avoir de 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 clique sur l'élément à valider 3.5c deux options sont possibles suivant si l'utilisateur authentifié 3.6a est *manager / admin* ou bien s'il est *normal* 3.6b.



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

FIGURE 3.6 – scénario IV

S'il s'agit d'un utilisateur avec privilèges, alors il a l'option de valider l'échange 3.6a.

Supposons que Daniel, qui est administrateur valide l'échange. Alors, l'écran *Horaires* est modifié 3.6c pour tout le monde et affiche que c'est bien Sara qui travaille le 17 novembre et non plus Kim.

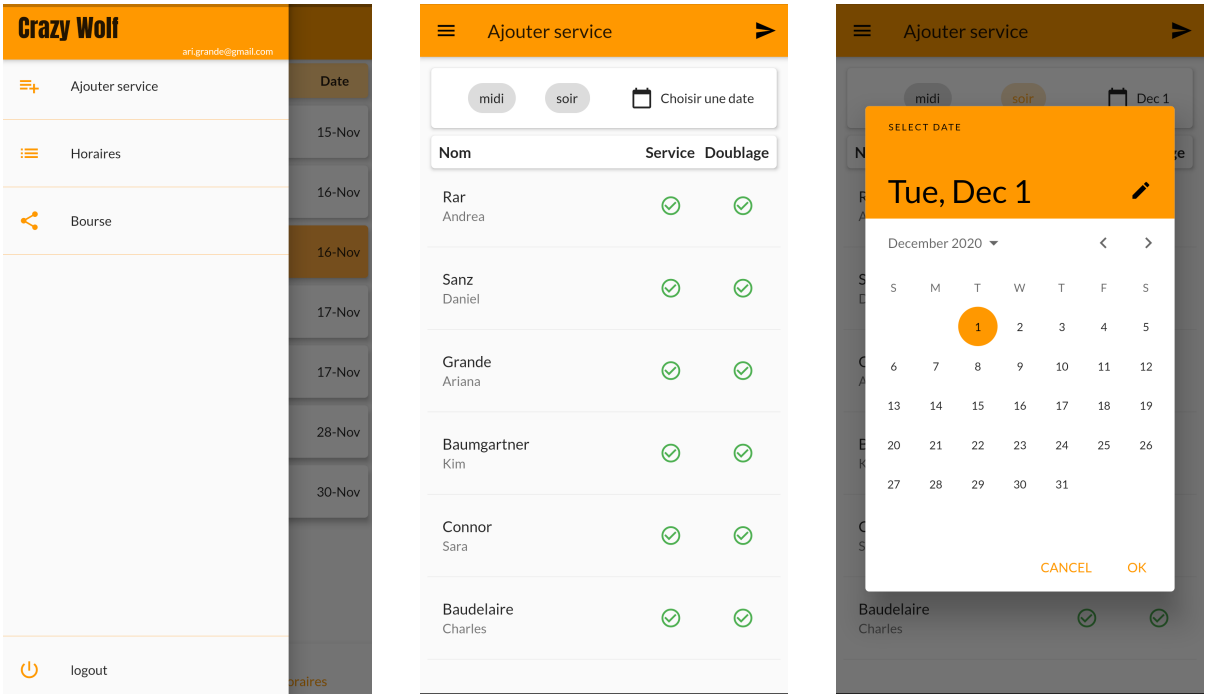
De plus, l'élément dans *Bourse aux jobs* 3.3b est supprimé.

Si au contraire, c'est un utilisateur sans privilèges, alors l'écran lui indique qu'une transaction est en cours de validation.

3.5 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, 3.7a naviguer à *Ajouter service* 3.7b. Notez que ce menu n'est pas disponible pour les utilisateurs sans privilèges.



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

FIGURE 3.7 – scénario V - a

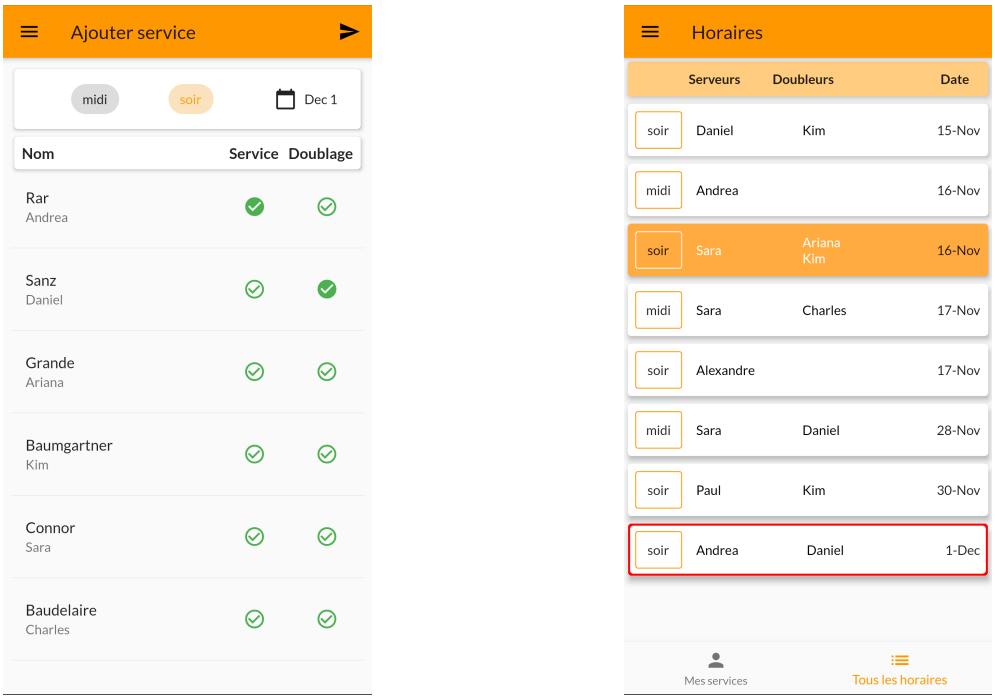
Pour créer le service, il faut choisir :

- une date en appuyant sur *Choisir une date*. Un *popup* avec le calendrier s'affiche 3.7c.
- un type en appuyant sur *midi* ou *soir* 3.8a.
- une ou plusieurs personnes au *Service* 3.8a.
- zéro, une ou plusieurs personnes au *Doublage* 3.8a.

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 3.8a, retournera une erreur s'il existe un service du même type à la même date.



(a) choix du personnel et type

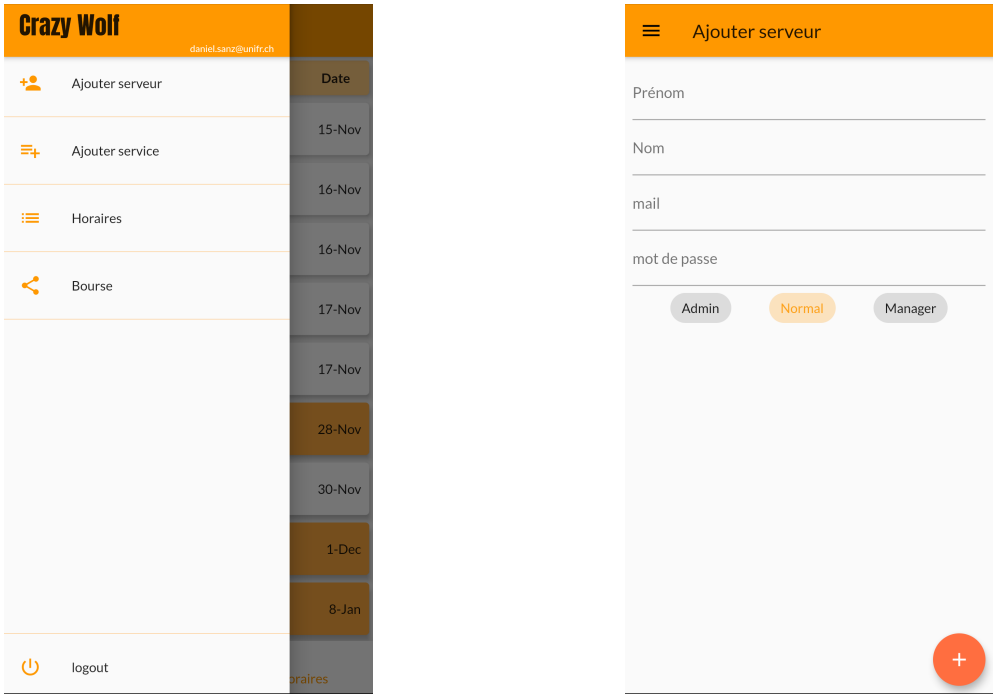
(b) nouveau service créé

FIGURE 3.8 – scénario V - b

3.6 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 3.9 – scénario VI

Pour ce faire, ils doivent après s'être authentifié, naviguer à l'aide du menu latéral 3.9a à l'onglet *Ajouter serveur*.

Dans l'onglet *Ajouter serveur*, il faut remplir les champs définissants un serveur. 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 présélectionné.

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.

Une 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* 3.9b

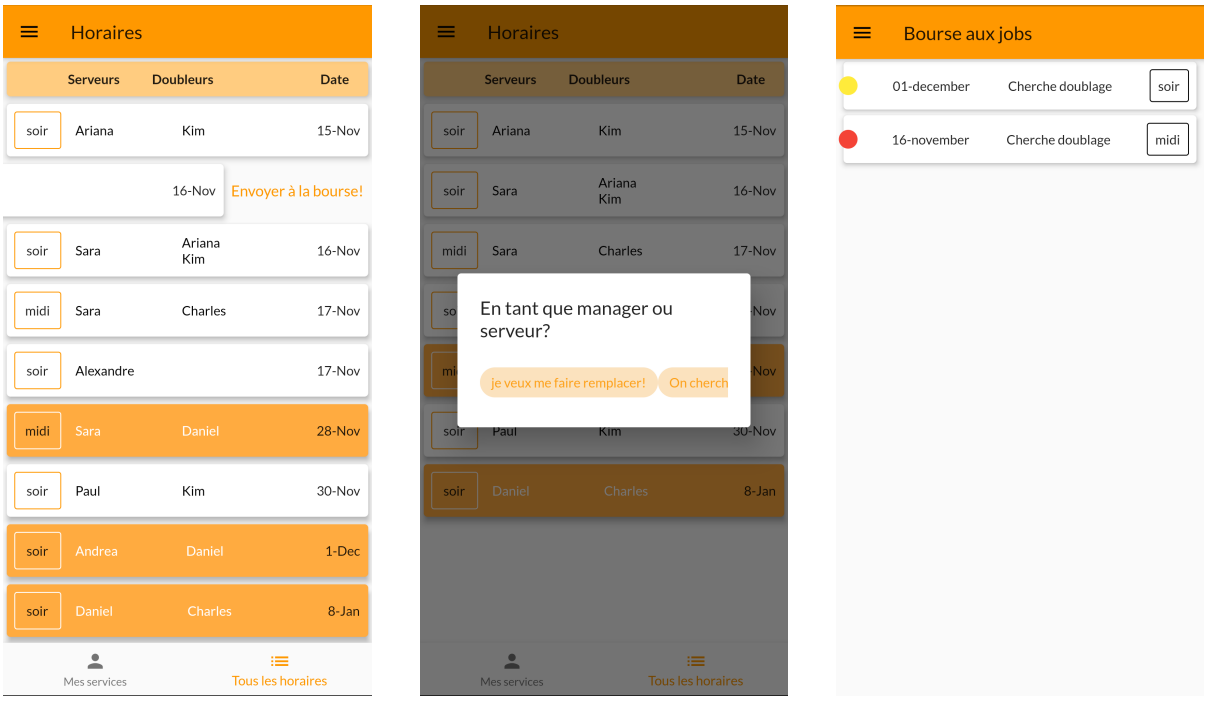
C'est ici que sont définis les identifiants nécessaires pour s'authentifier dans l'écran 3.1a.

3.7 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 3.10 – scénario VII

Dans un premier temps, un utilisateur avec privilèges fait glisser un service sur la gauche 3.10a. Deux options sont possibles suivant si l'utilisateur travaille dans ce service ou pas.

— S'il y travaille, alors un *popup* apparaît 3.10b pour demander s'il s'agit d'une demande de doublage ou de remplacement.
— 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. Dans le premier, l'administrateur n'y travaille pas 3.10a. Comme pour les échanges, le degré d'urgence 3.2b est demandé. On peut voir le service du 16 novembre en bourse.

Le deuxième, l'administrateur y travaille et il choisit 3.10b qu'il cherche un doublage.

À la différence du scénario III 3.5, la condition pour pouvoir accepter des postulants n'est plus d'être l'utilisateur ayant mis le service en bourse mais d'avoir un niveau de privilèges supérieur à *normal*.

CHAPITRE 4

Éléments de programmation

Dans ce chapitre sont présentées les technologies, paradigmes et techniques liées à l'implémentation informatique de l'application. Notamment, lesquelles ont été utilisées, comment elles fonctionnent et, une à une, comment cela s'est traduit dans l'implémentation final de l'application. Les exemples de codes sont tirés de l'application.

4.1 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 librairies 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.

4.1.1 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

4.1.2 Architecture

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

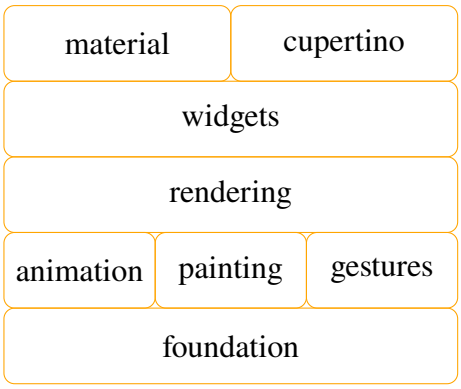


FIGURE 4.1 – Layers du Framework Flutter

Le *layer foundation* offre les classes 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 détection 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 personnalisables qui implémentent tout un catalogue de widgets avec respectivement le style material design d'Android et le design propre à IOS.

4.1.3 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 Listing: 2 le sous-entend, les widgets peuvent s'encapsuler les uns dans les autres. Ainsi, Le widget *Text* accepte dans son 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 fonctionnalité complexe.

Le terme widget englobe une multitude de verbes. Un widget peut ne pas avoir de représentation en tant que tel mais peut offrir un positionnement, une transformation, une taille, interaction avec 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 widgets 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 ont 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 déterminer 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 barre de progression circulaire (le *CircularProgressIndicator*) au centre (*Center*) de l'écran.

Cette classe représente l'écran affiché à chaque fois qu'une requête à la base de données se fait. Elle permet à l'utilisateur de savoir qu'une action est en cours et qu'il doit patienter.

StatefulWidget

Ce sont les widgets dont les propriétés peuvent varier dans le temps. Ces widgets ont un état, or ils sont immutables. 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()*, la méthode *build* est appelée et l'interface est mise-à-jour. En guise d'exemple, voici une partie du code source de l'écran *Horaires* 3.1b. Notamment, le petit menu inférieur permettant d'afficher tous les horaires ou seulement ceux de l'utilisateur authentifié.

```
class ScreenScheduleView extends StatefulWidget {
    @override
    _ScreenScheduleViewState createState() => _ScreenScheduleViewState();
    }
    class _ScreenScheduleViewState extends State<ScreenScheduleView> {
    var _selectedIndex = 0;

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

Listing 5 – Screen Horaires

La classe *ScheduleView* hérite d'un *StatefulWidget*. Dans la classe d'état il y a l'attribut : *selectedIndex*. L'indice sélectionné a pour valeur 0 à la construction de l'instance. Si la fonction *onItemTapped(int index)* est appelée avec un entier en argument alors, à l'aide de la fonction *setState* l'état est modifié et la fonction *build()* de *ScreenScheduleViewSate* est appelée. Ainsi, le *widget* est reconstruit en fonction de son nouvel état.

```
@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() de ScheduleView

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 lorsqu'une *tap* a été détectée sur l'un des deux icons définis dans la liste *Items*.
- un index qui définit lequel des éléments de la liste est actuellement sélectionné.
- une liste de *widgets* de type *BottomNavigationBarItem*.

Le premier élément de la liste i.e. "Mes services" a pour index 0. Et "Tous les horaires" 1. Par défaut, *selectedIndex* vaut 0. Dans un premier temps c'est "Mes services" qui est affiché.

Lorsque l'utilisateur clic sur un icône, *BottomNavigationBar* va appeler la fonction *onItemTapped* définie dans le code Listing: 5 avec l'index de l'icône en question en argument. *onItemTapped* va changer l'état du widget et par extension la valeur de *selectedIndex*. L'état ayant changé la fonction *build* est appelée et donc *BottomNavigationBar* redessiné avec un *currentIndex* potentiellement différent. Traduisant ainsi un affichage différent et l'icône sélectionné apparaîtrait en orange.

4.1.4 Flux de données

Dans la partie précédente, on constate que la gestion de l'état des widgets peut vite devenir encombrante. En effet, dans l'exemple, l'utilisation d'un *StatefulWidget* est naturelle, l'information étant un index n'est ni complexe ni lourde. Or comment, partager des données complexes entre les différents widgets ?

En général on fait la distinction entre l'état éphémère et l'état de l'application.

- L'état éphémère est celui qui peut entièrement être définit au sein d'un widget. Il n'a pas ou peu besoin d'être accédé à l'extérieur du widget et il ne varie pas de façon complexe.
- L'état de l'application est celui qui à l'inverse, peu se modifier de façon complexe, doit être préservé lorsque l'application est fermée, contient de vastes informations et doit être partagé parmi divers widgets. Comme par exemple, la liste des serveurs, la liste des services, les données d'authentification . . .

Il existe de nombreuses techniques pour la gestion du flux de données dans les applications et en informatique en général. Pour cette application, 3 façons ont été utilisées.

Pour transmettre les données d'un widget à ses enfants soit la transmission s'est faite par constructeur soit à l'aide des classes *ChangeNotifier*, *ChangeNotifierProvider* et *Consumer*.

Pour transmettre les données d'un enfant à un parent, les callbacks ont été utilisés.

Par constructeur

Il s'agit à la fois de la méthode la plus répandue comme la plus simple. Voici un exemple :

```
1 class ScheduleCard extends StatelessWidget {
2   final bool doesUserWorkIn;
3   final ServiceType stype;
4   final List<Waiter> waiters;
5   final List<Waiter> doublers;
6   final Service service;
7
8   const ScheduleCard({Key key, this.doesUserWorkIn, this.stype,
9     this.waiters, this.doublers, this.service,}) : super(key: key);
```

Listing 7 – ScheduleCard, par constructeur

Cette classe Listing: 7 est celle qui se charge de dessiner un élément de la liste des horaires Figure 3.1b. Son widget parent se charge de générer la liste dans son ensemble ainsi que d'autres implémentations de la logique de l'application. De plus, le parent transmet par constructeur à *ScheduleCard* les informations suivantes :

doesUserWorkIn : une variable booléenne indiquant si l'utilisateur actuellement authentifié travail dans le service en question. Ceci permet de changer la couleur de fond (orange ou blanc).
type : le type de service i.e. midi ou soir.
waiters : la liste des objets serveurs qui y travaillent.
doublers : la liste des objets serveurs qui doublent dans ce service.
service : l'objet service en question.

Ainsi chaque *Card* ou élément de la liste est une instance de l'objet *ScheduleCard*. On constat que les attributs sont *final* ce qui veut dire qu'il sont constant au *runtime*. Donc chaque instance de *card* est immutable. Ainsi, une nouvelle instance doit remplacer la précédente si l'on souhaite mettre-à-jour l'interface.

ChangeNotifier+Provider & Consumer

Théoriquement, rien n'empêche de transmettre toutes les informations par constructeur. Toutefois, l'arborescence des widgets devient rapidement profonde et dans la situation où une certaine information est nécessaire au niveau n mais aussi au niveau $n+i, i > 1$ il serait nécessaire de transmettre l'information dans tous les widgets intermédiaires alors qu'ils n'en n'ont pas besoin.

De plus, si une information n'est utile que tout en bas de l'arborescence et qu'elle a été transmise par constructeur, alors tous les widgets intermédiaires seront reconstruits inutilement.

ChangeNotifier
Il s'agit d'une classe dans Flutter qui implémente un objet de type *Listenable* du même framework. En somme, c'est une implémentation du pattern *Observer*[1]. Ainsi, il est possible d'utiliser *ChangeNotifier* lorsqu'une classe, susceptible de muter, doit notifier d'autres objets de ses modifications internes.

La classe *Pool*, celle qui gère la liste des *PoolItem* - objets représentant un élément en bourse - utilise *ChangeNotifier*.

```
1 class Pool with ChangeNotifier {
2   List<PoolItem> _pool;
3   bool isLoaded = false;
4   Pool(this._pool);
5   // ...
6 }
```

Listing 8 – ChangeNotifier - mixin

Le mot clef *with* veut dire qu'il s'agit d'un *mixin*. Les *mixin* permettent de combler l'absence d'héritage multiple. Ainsi, il est possible de réutiliser du code d'une classe dans de multiples hiérarchies de classes. Attention, ce n'est pas une interface dont il faut implémenter les méthodes.

1. https ://en.wikipedia.org/wiki/Observer_pattern

Voici, concrètement comment ça s'utilise. *deletePoolItem* est une méthode de la classe *Pool*. Lorsqu'un service mis en bourse est supprimé, cette méthode est appelée. Il s'agit d'une méthode asynchrone qui retourne, dans un futur, rien.

```
Future<void> deletePoolItem(String poolId, token) async {
   try {
     await http.delete(UrlManager.pool_path(poolId) + token);
     _pool.removeWhere((p) => p.id == poolId);
     notifyListeners();
     return;
   } catch (error) {
     print("[deletePoolItem]:: " + error.toString());
     throw error;
   }
 }
```

Listing 9 – ChangeNotifier - notify

Comme on peut le voir, trois actions ont lieu :

— Une requête de suppression dans la base de données.
— Supprime le *Poolitem* de la liste.
— Notifie tous les *observateurs* ou *listeners* d'une modification.

Ce type de comportement a été implémenté dans diverses méthodes des classes suivantes : *Pool*, *Services*, *Waiters* et *Auth*.

Pour pouvoir disposer de l'information il faut fournir ces classes un niveau au-dessus dans l'arborescence de là où leur contenu est utilisé. L'application n'est pas très complexe ainsi, la plupart des widgets qui utilisent ces diverses informations se situent directement en dessous du *MyApp*.

ChangeNotifierProvider

Voici une partie de la méthode *build()* de *MyApp* - le widget initial. L'objet *Multiprovider* est juste un *syntactic sugar* pour éviter l'imbrication des divers *Provider* dont dépend l'application.

```
@override
 Widget build(BuildContext context) {
   return MultiProvider(
     providers: [
       ChangeNotifierProvider(
         create: (context) => Auth()),
       ChangeNotifierProvider<Services>(
         create: (context) => Services([])),
       ChangeNotifierProvider<Waiters>(
         create: (context) => Waiters([])),
       ChangeNotifierProvider<Pool>(
         create: (context) => Pool([])),
     ],//...
```

Listing 10 – Multiprovider

Le contenu des classes : *Auth*, *Services*, *Waiters* et *Pool* est maintenant disponible dans l'ensemble de l'application au travers du *context*.

Le *context* est une instance de *BuildContext* qui se transmet de *widget* en *widget* leurs permettant de connaître leurs positions dans l'arborescence. Un usage courant du *context* dans les applications Flutter est :

```
Quelquechose.of(context);
```

permettant de retourner *quelquechose* que le *widget* le plus proche (relativement à celui qui fait l'appel) est en mesure de fournir.

Maintenant que l'information est disponible et qu'elle est mise-à-jour si on interagit avec elle, il faut en disposer.

Provider

C'est une des façons d'accéder aux différentes ressources mises à disposition. Par exemple, lorsqu'un remplaçant est accepté dans l'écran *poolDetails*, cette fonction ci-dessous est appelée au moment où le bouton est pressé.

```
Future<void> _accept(context, appId) async {
   var auth = Provider.of<Auth>(context, listen: false);
   try {
     await Provider.of<Pool>(context, listen: false)
         .selectApplicant(_poolItem.id, appId, auth.token);
     return;
   }//...
 }
```

Listing 11 – Provider of - example

Le premier *Provider* retourne une l'instance de la classe *Auth* contenant divers attributs et méthodes liés à l'authentification des utilisateurs. Entre autres, le *token* d'identification nécessaire pour les requêtes à la base de données. Cela sera discuté plus en détails dans la section authentification.

Le deuxième *Provider* donne accès à la ressource *Pool* ainsi qu'à ses méthodes. Dont une est utilisé. *selectApplicant* permet avec les identifiants d'un élément en bourse est d'un serveur y ayant postulé, de conclure la transaction en acceptant le postulant.

Cette distinction entre la logique relative à la base de données, l'affichage et la logique utilisateur permet un code réutilisable et plus performant. Tout widget/bouton permettant d'accepter un postulant par exemple, peut appeler le *Provider* et utiliser les méthodes de la ressource adéquate.

Consumer

Le widget *Consumer* permet de "consommer" les ressources. Voici un exemple où *Pool* est consommé.

```
//...
Consumer<Pool>(
  builder: (context, pData, _) {
    var pool = pData.pool;
    if (pool.isEmpty) {
      return Center(child: Text("Aucun service en bourse pour l'instant"),);
    }//...
```

Listing 12 – ScreenPool : Pool Consumer

Il est nécessaire de spécifier le type de la ressource, le *provider* en a besoin pour discriminer. L'argument *builder* est obligatoire. C'est la fonction qui sera appelé à chaque fois que le *ChangeNotifier* change i.e. la classe *Pool* qui contient des *PoolItem*.

L'écran *Bourse aux jobs* est intimement lié aux éléments *PoolItem*. De ce fait, certaines modifications de ceux-ci doivent traduire un changement dans l'interface.

Dans cet exemple, si la liste des *PoolItem* est vide, alors un écran indiquant qu'il n'y a aucun élément en bourse s'affiche. Ce *builder* sera appelé pour tout *notifyListeners()* présent dans le code de la classe *Pool*. Par exemple, accepter un postulant notifie les *listeners*, provoque l'appel de cette fonction et l'affichage se modifie pour indiquer que l'élément nécessite validation d'un administrateur.

Un autre exemple de *Consumer* est celui de *Auth*. En effet, l'affichage de l'application change radicalement suivant si l'utilisateur est authentifié ou pas. De plus, si au cours de l'utilisation cela vient à changer, l'entièreté de l'UI [2] doit changer. Voici un autre extrait de la fonction *build* de la classe *MyApp* :

```
//...
 child: Consumer<Auth>(
    builder: (ctx, auth, _) {//...
      return MaterialApp(
        title: 'Crazy Wolf',//...
        home: auth.isAuth
            ? ScreenSchedule()
            : FutureBuilder(future: auth.tryAutoLogin(),
                builder: (ctx, authResultSnapshot) =>
                    authResultSnapshot.connectionState == ConnectionState.waiting
                        ? ScreenSplash()
                        : ScreenAuth(),), //...
```

Listing 13 – ScreenPool : Auth Consumer

Auth est consommé. Si l'utilisateur est authentifié alors on affiche l'écran *Horaires*. Sinon, on essaye de s'auto-connecter. Si l'opération réussi, les *listeners* sont notifiés ce qui implique que cette fonction est appelée à nouveau et à la question *Auth.isAuth* la réponse sera vraie. Si l'opération d'auto-connexion échoue, alors l'écran de *login* est affiché.

2. User Interface - Interface Utilisateur

Callbacks

Un *callback* est par définition une fonction ou méthode qui est donnée en argument à une autre fonction ou méthode. Dans cette application les *callbacks* ont été utilisés pour transmettre une valeur d'un *widget* enfant à son parent sans en changer l'état.

Prenons, l'exemple de l'écran *Ajouter service* qui va se charger de faire la requête à la base de données avec toutes les information qui permettent d'y ajouter un service. Pour la construction de cet écran, il y a un parent *Statefull widget* : *ScreenAddService* et plusieurs enfants. Parmis ces enfants, seul le *widget TypeSelector* sera ilustré. En effet, ils implémentent tous le même mécanisme et les seules variations sont de type graphique.

Voici un extrait de la classe *ScreenAddService* :

```
1  class _ScreenAddServiceState extends State<ScreenAddService> {
2      ServiceType _selectedType;//...
3      TypeSelector(
4          (val) {
5              return _selectedType = val; },//...
6  }
```

Listing 14 – AddService callBack

Cette classe a un attribut de type *ServiceType*. A l'intérieur de cette classe, un *widget TypeSelector* dessine les boutons "midi" et "soir" est créé. L'argumemnt fournit au constructeur est une fonction. Celle-ci prend un paramètre *val* et retourne l'affectation de la valuer de *val* dans l'attribut *_selectedType*. En d'autres termes, *ScreenAddService* délègue l'affectation de son attribut *_selectedType* à l'objet de type *TypeSelector*.

```
1  typedef ServiceValue = ServiceType Function(ServiceType);
2  class TypeSelector extends StatefulWidget {
3    final ServiceValue select;
4    TypeSelector(this.select); //...}
5  class _TypeSelectorState extends State<TypeSelector> {
6    ServiceType currentSelectedType;
7    @override
8    Widget build(BuildContext context) {//...
9          (type) => ChoiceChip(//...
10             onSelected: (value) {
11             setState(() { currentSelectedType = ServiceType(type); });
12             widget.select(ServiceType(type));
13             }, ),//...
```

Listing 15 – AddService callBack

TypeSelctor quant à lui définit un type : *ServiceValue*. Ce type est une fonction qui prend en argument un *ServiceType* et qui retourne la même chose. En effet, si l'on veut pouvoir faire l'affectation dans le *widget* parent il faut que les types correspondent.

Finalement, cette fonction est assignée à la variable *select*. Lorsqu'un utilisateur appuie sur l'un des *ChoiceChip* - widgets chargés de dessiner les boutons - la fonction définie dans *onSelected* est appelée avec le type séléctionné en argument. A la ligne 12, l'appel de la fonction déléguée est fait avec le susdit type. On constat que dans le widget enfant, la fonction *setState* est aussi invoquée pour changer l'affichage des boutons.

4.2 La base de données

La base de données Real Time Database est une base de données non relationnelle. Les données sont stockées au format JSON. La base de données a trois objets : *Waiters*, *Services* et *Pool*. Pour serveurs, services, et éléments en bourse respectivement.

Nous allons parcourir chacun d'entre eux dans l'ordre.

Waiters

Il faut savoir que pour créer de nouveaux utilisateurs, il faut faire une requête *POST* avec le nom d'utilisateur, l'email et le mot de passe à une url spéciale dédiée aux interactions de type administrateur. Cette requête retourne un numéro d'identification (id). Ce sont ces identifiants qui vont permettre aux utilisateurs de se *logger* à l'écran Figure 3.1a. Ils sont créés uniquement à travers l'écran Figure 3.9b.

Dans cette implémentation, les utilisateurs et les serveurs (*waiters*) représentent la même entité.

Une fois que la création de l'utilisateur a réussie et que l'on dispose de son id, une deuxième requête *POST*, cette fois ci, à la base de données, est faite. Cette deuxième requête doit respecter le schéma suivant :

```
{
    "title": "Waiter",
    "type": "object",
    "required": [ "name", "sname", "role", "userId" ],
    "properties": {
        "name": {
            "type": "String",
            "description": "The user and waiter first name"
        },
        "sname":{
            "type": "String",
            "description": "The user and waiter second name"
        },
        "role": {
            "type": "String",
            "description": "credentials of the user. "
        },
        "userId": {
            "type": "String",
            "description": "id generated when signed"
        }
    }
}
```

Listing 16 – JSON Schema Waiters

Les propriétés correspondent aux champs dans 3.9b. Le *userId* et le numéro d'identification retourné par la première requête *POST*.

Comme il est uniquement possible d'ajouter des serveurs en étant administrateur et que l'on connait le rôle d'un utilisateur en regardant dans la base de données, il a fallu que le premier serveur soit ajouté manuellement depuis le navigateur.

Services

Représentent les objets définissant un service. C'est-à-dire, une date, un type, les serveurs et doubleurs qui y travaillent. Les requêtes *POST* pour populer la base de données doivent respecter le schéma suivant :

```
{
    "title": "Service",
    "type": "object",
    "required": [ "date", "id", "type", "waiterIds" ],
    "properties": {
        "date": {
            "type": "Iso: 8601 String",
            "description": "the date of the service"
        },
        "id":{
            "type": "String",
            "description": "autogenerated id of the service"
        },
        "type": {
            "type": "String",
            "description": "either midi or soir "
        },
        "waiterIds": {
            "type": "array",
            "items": { "$ref": "#/waiters/userId" },
            "description": "array of userIds field in waiters objects"
        },
        "doublerIds": {
            "type": "array",
            "items": { "$ref": "#/waiters/userId" },
            "description": "array of userIds field in waiters objects"
        }
    }
}
```

Listing 17 – JSON Schema Services

On constate qu'un service doit au moins avoir un serveur. L'objet *Waiter* n'est pas stocké en tant que tel mais juste le *userId* qui le représente.

En d'autres termes, les services "ont" des serveurs. Dans un premier temps la relation *many-to-many*, i.e. que les serveurs "aient" eux aussi des services fut envisagée. Cependant, cette pratique est déconseillée dans les bases de données non relationnelles. Ainsi la documentation de Real Time Database conseille d'aplatir les données.

Pool

Cet objet représente les éléments mis en bourse. Ainsi ils sont créés quand une demande de remplacement ou de doublage est faite. Chacun des élément respect le schéma suivant :

```
{
    "title": "Pool",
    "type": "object",
    "required": [ "needValidation", "type", "urgence", "serviceId" ],
    "properties": {
        "needValidation": {
            "type": "bool",
            "description": "true if poolItem needs to be validated by a priviledged user"
        },
        "id":{
            "type": "String",
            "description": "autogenerated id of the pool"
        },
        "type": {
            "type": "String",
            "description": "either service or doubler"
        },
        "urgence": {
            "type": "String",
            "description": "high or medium or low"
        },
        "applicantsIds": {
            "type": "array",
            "items": { "$ref": "#/waiters/userId" },
            "description": "array of userIds field in waiters objects"
        },
        "selectedAppId": {
            "type": "String",
            "description": "userId of the selected applicant"
        },
        "serviceId": {
            "type": "String",
            "description": "id of the service where someone needs to be replaced or looking for doublers"
        }
    }
}
```

Listing 18 – JSON Schema Pool

On voit que dans la base de données, il y a l'attribut *needValidation*. Celui-ci est modifié lorsqu'un remplaçant a été accepté. Le champs *applicantsIds* et *selectedAppId* ne sont pas obligatoires. Ils sont ajoutés seulement lorsque des utilisateurs postulent au service mis en bourse.

4.2.1 La méthodologie REST

En l'an 2000 Roy Thomas Fielding publie sa thèse *Architectural Styles and the Design of Network-based Software Architectures* qui entre autres définit le standard *Representational State Transfer* élégament abrégé *REST*. Dans cette thèse, M. Fielding fait le constat qu'il existe essentiellement deux façons de concevoir une architecture software :

- Tableau blanc où l'architecte donne libre cours à son imagination pour concevoir un système à base de composants qui lui sont familiers.
- À partir d'un système donné sans contraintes mais avec des besoins connus où l'architecte va graduellement cerner et appliquer des contraintes au système.

Concernant les services web, M.Fielding conçoit *REST* en identifiant les contraintes suivantes :

- **Client-Server** : l'interface utilisateur et séparée du stockage des données.
- **Stateless** : le serveur ne garde pas d'état concernant les clients. En d'autres termes, lorsqu'une requête est addréssée au serveur, elle contient toutes les informations nécéssaires pour y répondre.

- **Cache** : le serveur définit explicitement ou implicitement si les données d'une requête sont *cacheable*. Si elles le sont, alors le client en garde une copie temporaire suite à une requête. Évitant ainsi des répétitions de requêtes. L'avantage est que le système est plus performant car les requêtes sont réduites. L'inconvénient et que l'état des données dans le serveur et chez le client peut différer.
- **Uniform Interface** : le serveur et le client suivent un protocole pour intéragir avec les données. Deux composants de base forment cette contrainte :
  - **Identification** : toutes les ressources sont uniquement identifiées.
  - **Manipulation** : un ensemble d'opérations sur les données dont les résultats sont prévisibles.
- **Layered** : L'ensemble du service offert est composé de couches indépendantes et communicantes entre elles.

Dans ce travail, l'application respecte les contraintes *REST*. Concernant l'interface uniforme, le protocole *http* a été utilisé. Dans la *Real time database*, les verbes *http* on le sens suivant.

GET permet d'accéder à une ressource. Ne modifie pas les ressources.
POST création d'une ressource dans une liste de données. La base de données génère automatiquement un identifiant unique pour ce nouvel élément.
PUT permet d'écrire ou de remplacer une ressource pour une chemin donné.
PATCH permet de modifier partiellement une ressource.
DELETE supprime une ressource.

Dans ce contexte, le terme ressource représente toujours un objet JSON.

4.2.2 REST appliqué à Dart

Dans ce projet, plusieurs models ont été définit du côté client. Ces models sont extrêmement similaires aux formats JSON qui doivent être respectés du côté serveur. Ils fonctionnent par doublet, c'est-à-dire, qu'il y a le model qui représente une unité et celui qui représente une liste d'objets. Ces derniers, ont déjà été brièvement traités dans la partie *ChangeNotifier*.

Les models : *Waiter*, *Service* et *PoolItem* sont la partie unitaire de *Waiters*, *Services* et *Pool* respectivement.

Leur structure et fonctionnement sont similaires ainsi seul *Service* et *Services* vont être exposés dans ce rapport.

La classe *Service*, comme dit précédemment, n'est qu'une implémentation locale de l'objet JSON stocké dans la base de données. Ainsi elle a les attributs suivants :

```dart
class Service implements Comparable<Service> {
    final String _id;
    final DateTime _date;
    final ServiceType _type;
    final List<String> _waiterIds;
    final List<String> _doublerIds;

    Service(this._id, this._date, this._type, this._waiterIds, this._doublerIds);
}
```

Listing 19 – Service class

Les attributs sont tous finaux. Le trait du bas est la façon en Dart de faire des attributs privés. Chaque objet de cette classe est immutable. Ces objets sont chargés depuis la base de données ou s'ils sont créés en local, alors il seront envoyés à la base de données pour qu'elle soit mise-à-jour.

Il est donc nécessaires de disposer d'une traduction en JSON dans les deux sens. Voici le constructeur qui permet de construire une instance de Service avec un un JSON en argument.

```
1  Service.fromJson(Map<String, dynamic> json)
2  : this._id = json['id'],
3      this._date = DateTime.parse(json['date']),
4      this._type = ServiceType(json['type']),
5      this._waiterIds = json['waiterIds'] != null
6          ? (json['waiterIds'] as Map<String, dynamic>).keys.toList()
7          : [],
8      this._doublerIds = json['doublerIds'] != null
9          ? (json['doublerIds'] as Map<String, dynamic>).keys.toList()
10         : [];
```

Listing 20 – JSON to Service instance

Dans Listing: 18 il est dit qu'il y a une liste d'ids pour les serveurs et une autre pour les ids des doubleurs. On constate ici que ce n'est pas exactement une liste mais un *map*. En effet, la Real Time Database gère les listes comme des pairs {clef, valeur} où les clefs sont les indexes de la liste. Le problème est que si l'on modifie ou supprime un élément de la liste, les indexes ne sont pas mis à jour. Il s'est avéré beaucoup plus simple d'utiliser *map* car chaque id peut être uniquement accessible et l'ensemble des données reste consistent après modificiations ou suppressions. On voit dans les lignes 6 et 9 de Listing: 20 que seules les clefs, qui représentent les ids, du *map* sont extraites. Les valeurs ne portant aucune information.

Dans l'autre sens, la traduction d'une instance en objet JSON :

```
1  Map<String, dynamic> toJson() => {
2      'id': _id,
3      'date': _date.toIso8601String(),
4      'type': _type.toString(),
5      'waiterIds': {for (var w in _waiterIds) w: true},
6      'doublerIds': {for (var d in _doublerIds) d: true},
7      };
```

Listing 21 – Service instance to JSON

Comme on le voit dans les lignes 5 et 6, les valeurs du *map* sont juste des *true* qui doivent être castés en string en interne. Naturellement, JSON est un format qui n'accepte que du texte. Ainsi, la date est formaté en ISO 8601 [3] et on demande la version *toString()* du *_type*. *_type* est une instance de la classe *ServiceType* qui *override* a méthode *toString()* présente dans toutes les classes.

3. https ://www.iso.org/iso-8601-date-and-time-format.html

GET

Maintenant que chaque élément est facilement traduisible en JSON, on peut faire des requêtes à la base de données. Voici comment les services sont chargés en local :

```
1  Future<List<Service>> fetchService(String token) async {
2      try {
3          final response = await http.get(UrlManager.url_service + token);
4          final extractedData = json.decode(response.body) as Map<String, dynamic>;
5
6          if (extractedData == null) {
7              _services.clear();
8              return [];
9          }
10
11         final List<Service> loadedData = [];
12         extractedData.forEach((_, value) {
13             loadedData.add(Service.fromJson(value));
14         });
15
16         _services = loadedData..sort();
17         isLoaded = true;
18
19         notifyListeners();
20         return _services;
21     } catch (error) {
22         print("[fetchService]:: " + error.toString());
23         throw error;
24     }
25 }
```

Listing 22 – Fetch services

la méthode *fetchService* de la classe *Services* (attention au "s") prend un token en argument. Il s'agit d'un paramètre nécessaire pour l'authentification qui sera traîté plus en détails dans la section suivante. C'est une méthode asynchrone qui dans un futur retournera une liste de services. Au travers de *http* qui est un ensemble de classe disponibles dans le package http l'on fait appel à la méthode *http.get* avec l'url des services. L'objet retourné est de type JSON, il faut donc en extraire le corps et le *caster* en *map*.

Après s'être assuré que la réponse contient bel et bien des données, lignes 8 à 11, on extrait les données, en utilisant le constructeur définit dans Listing: 20, dans la variable loadedData.

La Real Time Database à cause du format JSON ne garantit pas l'ordre. Ainsi, les services sont triés par date en ordre croissant. D'où l'implémentation de l'interface *Comparable* que l'on voit dans Listing: 19.

4.3 Authentification

Afin d'authentifier les requêtes à la base de données, il faut disposer d'un token. Celui-ci s'obtient comme pour créer de nouveau utilisateurs à travers une requête POST à une URL administrateur. Cette requête doit contenir dans son corps, l'email et le mot de passe. Si ceux-ci sont correctes, un token et retourné. Du côté

```
Future<void> _authenticate(String email, String password, String url) async {
    try {
        final response = await http.post(
            url,
            body: json.encode(
             {
                'email': email,
                'password': password,
                'returnSecureToken': true,
             },
            ),
        ); ///...
    }
```

Listing 23 – Token request

serveur, il faut indiquer que seules les requêtes authentifiées sont permises. La Real Time Database offre un système de règles que l'on peut définir. Voici les règles qui empêchent de lire ou écrire dans la base de données sans s'être authentifié :

```
{
    "rules": {
        ".read": "auth != null",
        ".write":"auth != null",
    }
}
```

Listing 24 – Database rulest

Le token qui permet d'authentifier les requêtes est valable pendant une heure. Après quoi il expire.

Conclusion

En conclusion, les objectifs fixés ont été globalement atteints. L'application est fonctionnelle et réellement utilisable. Elle n'a pas de soucis de performance et les données transférées à la base de données sont suffisamment petites pour n'engendrer aucun coût pour le restaurant.

Malheureusement, depuis que j'ai finalisé le code de l'application, tous les restaurants sont fermé du à la pandémie du COVID-19. J'ai été et je suis actuellement encore dans l'incapacité de tester l'application dans une situation réelle.

Il y a toutefois, une multitude de points à améliorer. Notamment, au niveau de l'architecture. J'ai en effet fait la distinction entre le model et la *view*, or ça aurait été encore mieux de suivre un motif de type Model View Controller. Ou encore au niveau de la sécurité. Car, dans l'implémentation actuelle, les droits d'accès, c'est-à-dire, ce qui empêche un utilisateur normal d'être un administrateur sont uniquement gérés dans l'application elle-même en interdisant certains écrans d'être affichés. Il faudrait également ajouter des restrictions à certaines parties de la base de données, du côté serveur.

Finalement, pour une toute première fois dans le développement d'application je suis très fière de mon résultat. Malgré ses défauts. J'ai pu mettre en application toute sorte de concepts appris ces trois années de bachelor. Depuis l'héritage jusqu'à la programmation fonctionnelle (il n'y a pas une boucle for dans tout le projet), en passant par les bases de données, les documents semi-structurés (JSON), les design patterns (composite) ou encore les automates à états pour gérer les privilèges.