CHAPITRE 1

Éléments de programmation

Dans ce chapitre sont présentés les technologies, paradigmes et techniques liés 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.

1.1 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 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 a été exclu 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.

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 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.

1.2 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 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.

1.2.1 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

1.2.2 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*).

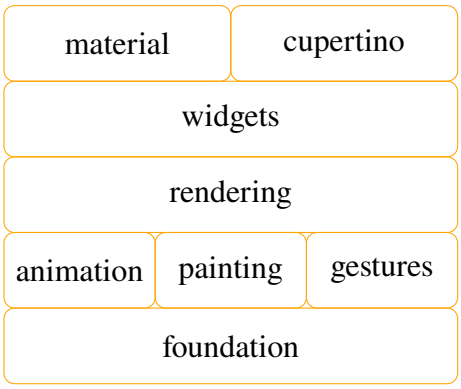

FIGURE 1.1 – Layers du Framework Flutter

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.

1.2.3 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 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 bare 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 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()*, 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* **??**. 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 selectionné a pour valeur 0 à la construction de l'instance. 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 *ScreenScheduleViewSate* est appelée. Ainis 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 lorsque une *tap* a été detecté sur l'un des deux icons définis dans la liste *Items*.
- un index qui définit lequel des éléments de la liste est actuellement séléctionné.
- 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 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 changer la fonction *build* est appelée et donc *BottomNavigationBar* redessiné avec un *currentIndex* potentiellement différent. Traduisant ainsi un affichage différent et l'ion sélectionné apparaîtrait en orange.

1.2.4 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 nombreuse techniques pour la gestion du flux de données dans les applications et en informatique en général. Pour cette application, 3 façons on 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 on 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 7 est celle qui se charge de dessiner un élément de la liste des horaires **??**. 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 un 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 reconstruit 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*[2]. 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.

2. 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 future, 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());
   }
 }
```

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, lorsque un remplaçant est accepté dans l'écran *poolDetails*, cette fonction ci-dessous est appelée au moment ou 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 et authentifié ou pas. De plus, si au cours de l'utilisation cela vient à changer, l'entièreté de l'UI [3] 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.

3. User Interface - Interface Utilisateur

1.3 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 donnée 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 intéractions 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 **??**. Ils sont crées uniquement à travers l'écran **??**.

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 dipose 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 14 – JSON Schema Waiters

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

Comme il est uniquement possible d'ajouter des serveurs en étant administrateur et que l'on connait le role 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 15 – 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 service "on" des serveurs. Dans un premier temps la relation *many-to-many*, i.e que les serveurs "aïent" eux aussi des services fut envisagée. Cependant, cette pratique est déconseillé dans les bases de données non relationnelles. Ainsi la documentation de Real Time Database conseille d'applatir les données.

Pool

Cet objet représente les éléments mis en bourse. Ainsi ils sont crées 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 16 – 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 accépté. Le champs *applicantsIds* et *selectedAppId* ne sont pas obligatoires. Ils sont ajouté seulement lorsque des utilisateurs postulent au service mis en bourse.

1.3.1 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'autre 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. Evitant 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 composant de base forment cette contrainte :
  - **Identification** : toutes les ressources sont uniquement identifiées.
  - **Manipulation** : un ensemble d'opérations sur les données dont les résultat sont prévisibles.
- **Layered** : L'enseble 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'acceder 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.

1.3.2 REST appliqué à Dart

Dans ce projet, plusieurs models on été définit du côté client. Ces models sont extrêmement similaires aux formats JSON qui doivent être respéctés 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'objects. Ces derniers, on 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 est fonctionnement sont similaires ainsi seul *Service* et *Services* vont être exposés dans ce rapport.

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

```
1  class Service implements Comparable<Service> {
2      final String _id;
3      final DateTime _date;
4      final ServiceType _type;
5      final List<String> _waiterIds;
6      final List<String> _doublerIds;
7
8      Service(this._id, this._date, this._type, this._waiterIds, this._doublerIds);
9  }
```

Listing 17 – Service class

Les attributs sont tous finaux. Le traît du bas est la façon en Dart de faire des attributs privés. Chque objet de cette classe est immutable. Ces objets sont chargé depuis la base de données ou s'ils sont crée en locale, alors il seront envoyés à la base de données pour qu'elle soit mise-à-jour. Il est donc nécéssaires de disposer d'une traduction en JSON dans les deux sens.

```
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 18 – JSON to Service instance

Dans 16 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 beacoup plus simple d'utiliser *map* car chaque id peut être uniquement accessible et tout reste consistent après modificiations ou suppression. On voit dans les lignes 6 et 9 de 18 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 19 – Service instance to JSON

Comme on le voit dans les lignes 5 et 6, les valeurs du *map* sont juste des *true*. 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 :

la méthode *fetchService* de la classe *Services* (attention au "s") prend un token en argument. Il s'agit d'un paramètre nécéssaire pour l'authentification qui sera traîté plus en détails dans la section suivante. C'est une méthode asynchrone qui dans un future 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 18, 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 la classe *Comparable* que l'on voit dans 17.

```dart
Future<List<Service>> fetchService(String token) async {
    try {
        final response = await http.get(UrlManager.url_service + token);
        final extractedData = json.decode(response.body) as Map<String, dynamic>;

        if (extractedData == null) {
        _services.clear();
        return [];
        }

        final List<Service> loadedData = [];
        extractedData.forEach((_, value) {
        loadedData.add(Service.fromJson(value));
        });

        _services = loadedData..sort();
        isLoaded = true;

        notifyListeners();
        return _services;
    } catch (error) {
        print("[fetchService]:: " + error.toString());
        throw error;
    }
}
```

Listing 20 – Fetch services

1.3.3 Authentification

1.4 Notifications

1.5 Sécurité

1.5.1 Firebase rules