CHAPITRE 1

Eléments de programmation

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

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 vidéos, etc. Exclu *de facto* le choix d'une technologie basée exclusivement sur les performances ou sur le développement natif.

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

Ces constats donnent lieu aux critères de sélection suivants :

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

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

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

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

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

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

1. Liste non exhaustive

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

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

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

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

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 libraries natives. Par exemple, pour dessiner un bouton, il ne va pas faire appel au primitives bouton dans Android ou IOS mais va dessiner pixel par pixel l'objet souhaité.

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*).

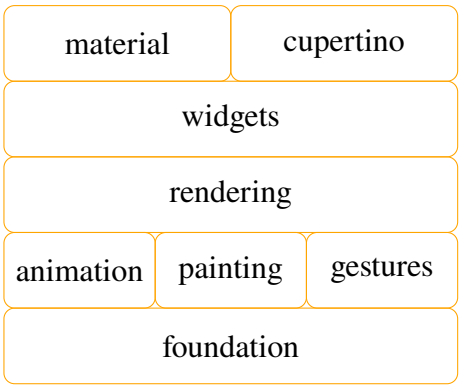

FIGURE 1.1 – Layers du Framework Flutter

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

StatefulWidget

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

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

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

class _ScreenHoraireState extends State<ScreenHoraire> {
    int _selectedIndex = 1; //...
    void _onItemTapped(int index) {setState(() {_selectedIndex = index;});}
```

Listing 5 – Screen Horaires

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

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

Listing 6 – Build dans horaires

Scaffold accepte un menu de navigation au bas de la page. Celui-ci est un widget : *BottomNavigationBar*. Ce widget prend en construction une fonction qui est appelée lorsque une tap a été detecté sur l'un des deux icons définis dans la liste *Items* ainsi que l'index actuellement selectionné. Le premier élément de la liste i.e. "Mes services" a pour index 0. Et "Tous les horaires" 1. Par défaut, *selectedIndex* vaut 1. C'est donc tous les horaires qui sont affiché.

Lorsque l'utilisateur clic su l'icon de la personne, *BottomNavigationBar* va appeler la fonction *onItemTapped* définie dans le code 5 avec l'index de l'icon en argument. *onItemTapped* va changer l'état du widget et par extension la valeur de *selectedIndex*. L'état ayant changer la fonction *build* est appelée et donc *BottomNavigationBar* redessiné avec un *currentIndex* potentiellement différent. Traduisant ainsi un affichage différent et l'icon séléctionné apparaitrai 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'exterieur 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 donnée d'authentification . . .

Il existe de nombreuse técniques pour la gestion du flux de données dans les applications et en informatique en générale. 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 qui est classique dans les langages orientés objets soit à l'aide des patterns Provider et Consumer.

Pour transmettre les données d'un enfant à un parent, les callback on été utilisés.

Par constructeur

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

```
1  class HoraireSubItem extends StatelessWidget {
2      const HoraireSubItem({//...,
3       @required this.isUserIn,}): super(key: key);
4      //...
5      final bool isUserIn;
6
7      @override
8      Widget build(BuildContext context) {
9          return Card(
10         color: isUserIn ? Colors.orangeAccent : Colors.white,
11         //...
```

Listing 7 – par constructeur

Provider & Consumer

1.3 La base de données

1.3.1 La méthodologie REST

1.3.2 REST appliqué à Dart

1.4 Notifications

1.5 Sécurité

1.5.1 Authentification

1.5.2 Firebase rules