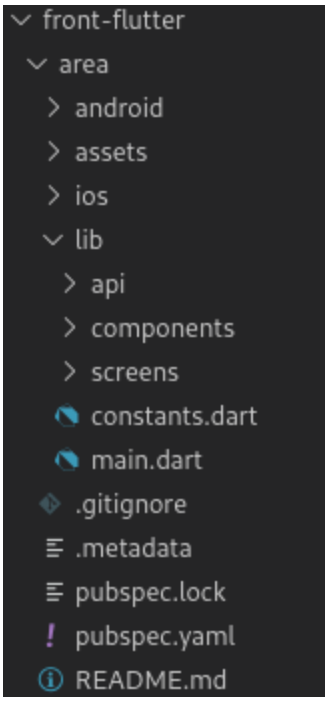
Documentation Front Mobile

1. Architecture du code



Le projet flutter se présente en différentes parties.

1.1 Le dossier “lib”

Le sous dossier API contient toutes les fonctions qui permettent à l’application de communiquer avec l’API du serveur. Les parties principales sont le profile, l’authentification, les services, et les areas.

Dans lib/main.dart, nous avons défini nos routes.

```
routes: <String, WidgetBuilder>{
  // Set routes for using the Navigator.
  '/home': (BuildContext context) => new WelcomeScreen(),
  '/app': (BuildContext context) => new NavBar(),
  '/web': (BuildContext contex) => new CustomWebView(),
  '/edit': (BuildContext context) => new EditArea(),
},
```

- “/home” correspond à la partie où l’utilisateur peux choisir comment se connecter et créer son compte.
- “/app” correspond à l’application principale.
- “/web” contient la Webview dans lequel l’utilisateur se connecte.
- “/edit” contient une page appart ou l’utilisateur peux éditer une Area.

1.2 Pubsec.yml

Contient toutes les dépendances auquel notre projet est confronté.

Au niveau version, on ne tourne pas sur la version 2.0 de Flutter. Celle-ci étant sortie trop récemment à l’heure où le document est disponible.

On tourne donc sous Flutter 1.22 pour le projet.

En enlevant les librairies par défaut, on a rajouté :

- “shared_preferences” qui permet de stocker des variables dans le téléphone.
- “webview_flutter” qui permet de créer des Webviews pour les authentifications.
- “webview_cookie_manager” qui permet de manager facilement les cookies sur les Webviews.

1.3 Docker et le building automatique

Un dockerfile est présent sur le projet. Ce dernier nous sert pour setup l'environnement Flutter et ainsi pouvoir build l'application Android dans un environnement de test commun.

```
# Set up Android SDK
RUN wget -O sdk-tools.zip https://dl.google.com/android/repository/sdk-tools-linux-4333796.zip
RUN unzip sdk-tools.zip && rm sdk-tools.zip
RUN mv tools Android/sdk/tools
RUN cd Android/sdk/tools/bin && yes | ./sdkmanager --licenses
RUN cd Android/sdk/tools/bin && ./sdkmanager "build-tools;28.0.3" "patcher;v4" "platform-tools" "platforms;android-28" "sources;android-28"
RUN cd Android/sdk/tools/bin && ./sdkmanager "build-tools;29.0.3" "patcher;v4" "platform-tools" "platforms;android-29" "sources;android-29"
ENV PATH "$PATH:/home/developer/Android/sdk/platform-tools"

# Download Flutter SDK
RUN git clone --depth 1 --branch 1.22.5 https://github.com/flutter/flutter.git
ENV PATH "$PATH:/home/developer/flutter/bin"
```

Après avoir testé notre projet, on choisit de passer notre projet sous un GitHub action.

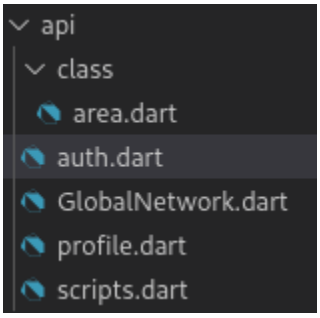
Celui-ci utilise le DockerFile précédent pour build le projet et créer un Artefact qui contient les différents APK. Cette artefact nous sert à vérifier que l'application tourne bien en mode RELEASE et peux nous servir à tester le projet en mode release.

2. Le code en détail :

2.1 API

2.1.1 Général

L'API contient différents fichiers :



- "GlobalNetwork.dart" contient le header, le userID de l'utilisateur ainsi que l'URL de l'API.
- "auth.dart" contient toutes les fonctions de base permettant à l'utilisateur de se connecter à l'API.
- "profile.dart" contient toutes les fonctions permettant de récupérer des informations pour la page "Profile" de notre application.
- "script.dart" contient toutes les fonctions permettant de récupérer ou d'éditer un Script/Area.

2.1.2 3 exemples de requêtes

LES REQUÊTES GET:

```
Future<Map<String, dynamic>> getReactionAvailable() async {
  final response =
      await http.get(urlArea + '/reaction/available', headers: headers);

  Map<String, dynamic> userinfo = jsonDecode(response.body);
  if (response.statusCode == 200) {
    return userinfo;
  } else {
    userinfo['error'] = true;
    return userinfo;
  }
}
```

Ici on génère l'URL de la requête et on envoi le header qui nous permet de nous identifier. On parse la requête sous un "Map<String, dynamic>" et on renvoi la requête à qui la demande dans l'application.

LES REQUÊTES POST:

```
Future<bool> postScriptCreate(ScriptCreation script) async {
  Map<String, dynamic> body = script.toJson();
  final msg = jsonEncode(body);
  final response =      You, 2 days ago • [ADD] Begin of SocialLogin and fix emp
      await http.post(urlArea + '/script/create', headers: headers, body: msg);

  if (response.statusCode == 200) {
    return true;
  } else {
    return false;
  }
}
```

Ici pour on demande un type défini de réponse afin de s'assurer que ce qu'on envoi à l'API est valide.
A partir de ce type en argument, la fonction s'occupe de transformer les arguments en JSON.

On génère l'URL de la requête et on envoi le header et le body qui nous permet de nous identifier.

A la fin, on renvoie "*true*" si la requête s'est bien passée ou "*false*" dans le cas contraire.

LES REQUÊTES PUT:

```
Future<bool> putScriptUpdate(ScriptEditing script) async {
  Map<String, dynamic> bod = script.toJson();
  final body = jsonEncode(bod);
  final response =
      await http.put(urlArea + '/script/update', headers: headers, body: body);

  if (response.statusCode == 200) {
    return true;
  } else {
    return false;
  }
}
```

Même fonctionnement qu'au-dessus pour POST mais ici on souhaite modifier la donnée.

2.1.3. La classe Area

Il arrive certaines fois, afin de s'assurer de l'intégrité de ce qu'on envoi qu'on utilise des classes

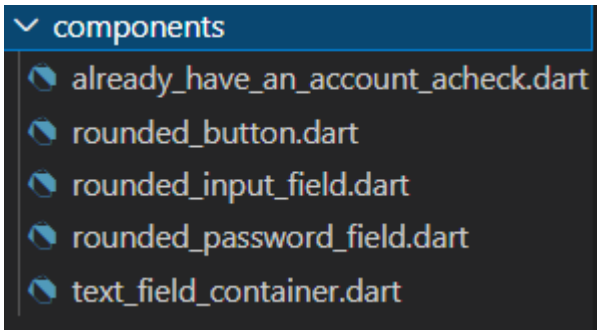
```
class ActionCreation {
  String actionId;
  Map<String, dynamic> parameters;
  ActionCreation(this.actionId, this.parameters);
  Map<String, dynamic> toJson() => {'action_id': actionId, 'parameters': parameters};
}
```

La classe ci-dessus par exemple nous permet d'avoir les bons arguments quel que soit le type et ainsi pouvoir générer un JSON.

Map<String, dynamic> toJson() nous permet de transformer notre classe en JSON par la suite.

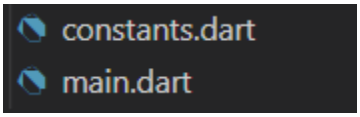
2.2 La Bar de navigation principale et les Screens

2.2.1. Les components



Les components sont des classes qui sont utilisées partout dans le code, notamment les champs de texte, les buttons à thème, et les textes de redirection.

2.2.1. Le main et les constantes

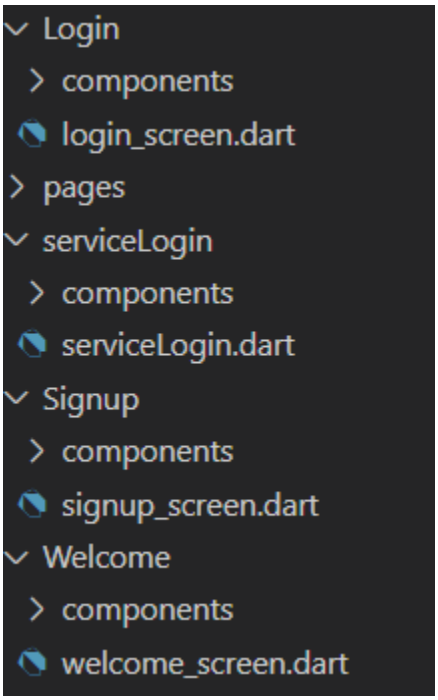



```dart
import 'package:flutter/material.dart';

const kPrimaryColor = Color(0xFF6F35A5);
const kPrimaryLightColor = Color(0xFFF1E6FF);
const kSecondayColor = Color(0xFF78C4D4);
const kLightGreyColor = Color(0xFFF6F6F6);

```

Le fichier main.dart contiens le root du projet. Il s'agit du fichier qui contient les routes de navigation et les paramètres principaux de l'application ; paramètres qui seront transmis par héritage a tous les widgets enfants de l'application.

2.2.2. Les screens

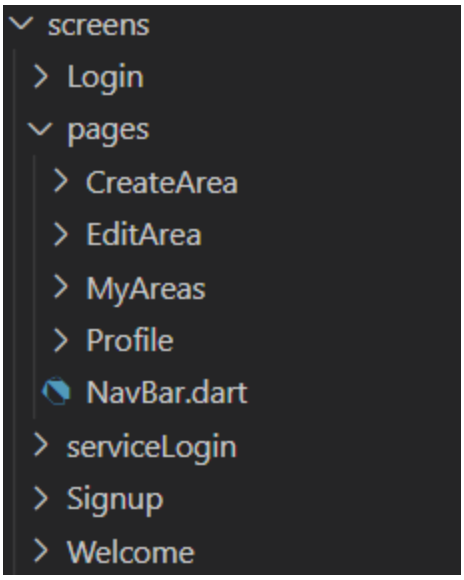
LOGIN, SIGN UP, ET SOCIAL LOGIN :



Le Welcome screen sert de page d'accueil pour rediriger vers le login, le sign up, ou la connexion avec un service. Les redirections se font grâce au Navigateur de flutter.

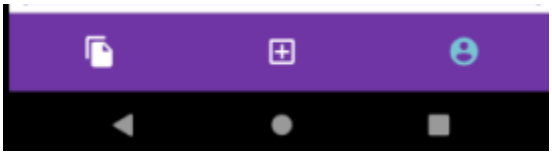
```
press: () {
  Navigator.push(
    context,
    MaterialPageRoute(
      builder: (context) {
        return SignUpScreen();
      },
    ), // MaterialPageRoute
  );
```

LA BAR DE NAVIGATION :

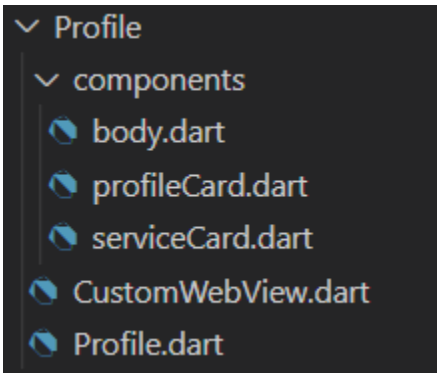


Les screens sont les sous dossiers qui contiennent tous les différents écrans et menus de l'application. La NavBar correspond à la base de la section principale. Elle contient 3 pages qui sont : mes areas, créer une area, et le profile.

```
List<Widget> _widgetOptions = <Widget>[MyAreas(), CreateArea(), MyProfile()];
void _onItemTapped(int index) {
  setState(() {
    _selectedIndex = index;
  });
}
```



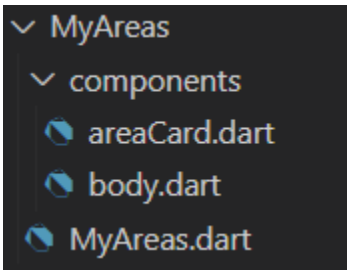
LA PAGE DE PROFILE :



Ce dossier contient un sous dossier components et les fichiers Profile.dart et CustomWebView.dart. Ils ont respectivement la base de la page de profile et la webView permettant de se connecter aux Services. La profileCard affiche les informations de l'utilisateur et lui permet de modifier ces informations et les serviceCards affichent les statuts des services et permettent à l'utilisateur de s'y connecter.

Le widget "FutureBuilder" permet de charger dynamiquement les informations sur le profile et le service en faisant un appel asynchrone à l'API.

LA PAGE MY AREAS, CREATE & EDIT AREAS :



Ces dossier contient les pages affichant les areas de l'utilisateur. Grace au Widget "FutureBuilder" elle sont dynamiquement chargés. Elles peuvent être activées ou désactivées, modifiées, et en créer de nouvelles. Le créateur d'area récupère toutes les actions et réactions disponible avec un appel de l'APK et génère le formulaire-.

```
@override
Widget build(BuildContext context) {
  return FutureBuilder(
    future: getActionAvailable(),
    builder: (context, snapshot) {
      if (snapshot.connectionState == ConnectionState.done) {
```