

Grail XRP Mobile Application

Overview of Grail code with screenshots:

The goal is to build the FanPay application by modifying the Grail app source code we developed during Wave 4. FanPay will allow users to store XRP/RLUSD in a virtual card, which can then be used for payments at shops in sporting events. Users will generate a QR code for payment, which can be scanned by an Ingenico terminal to complete the transaction.

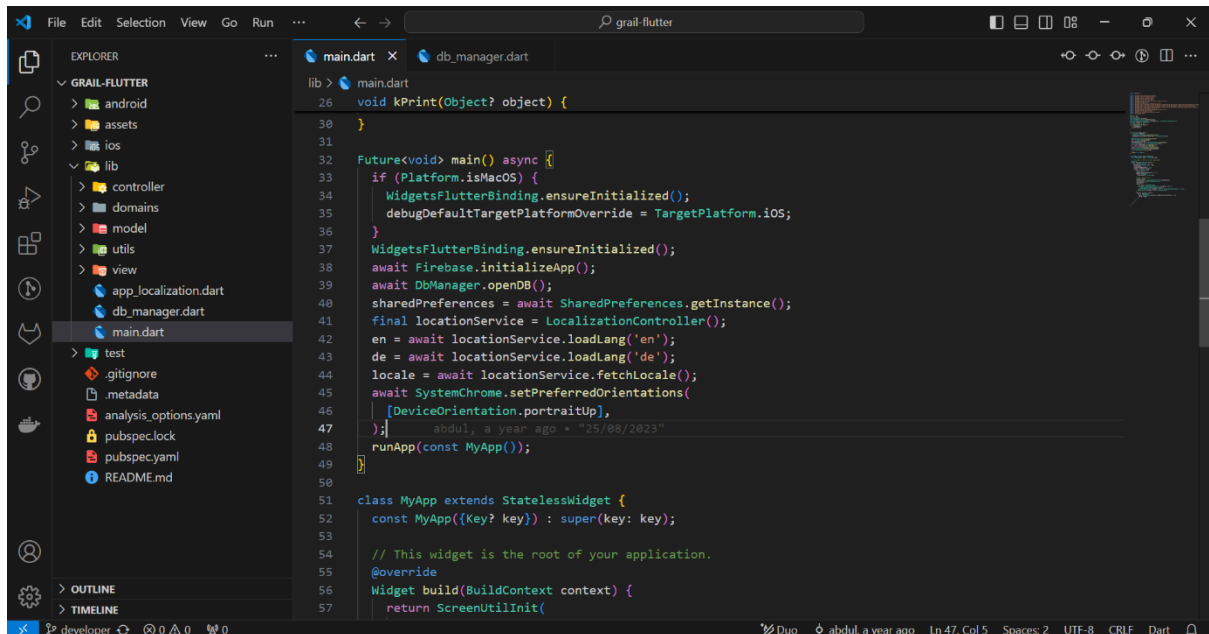
The Ingenico terminal will be connected to our Redimi API, which communicates with the EVM sidechain on XRPL to process the transaction and return the response to the terminal. Users will receive reward tokens against the transactions they make in the form of reward tokens that are mined on XRPL.

Grail application was built using Flutter utilizing GetX Architecture for state management. GetX is an extra-light and powerful solution for Flutter.

Following is the project structure:

```
Flutter Project Structure
├─ android/
├─ ios/
├─ assets/
├─ lib/
│   ├─ main.dart
│   ├─ db_manager.dart
│   ├─ app_localization.dart
│   ├─ controller/
│   ├─ domains/
│   ├─ model/
│   ├─ utils/
│   └─ views/
├─ test/
├─ pubspec.yaml
├─ README.md
└─ [other files...]
```

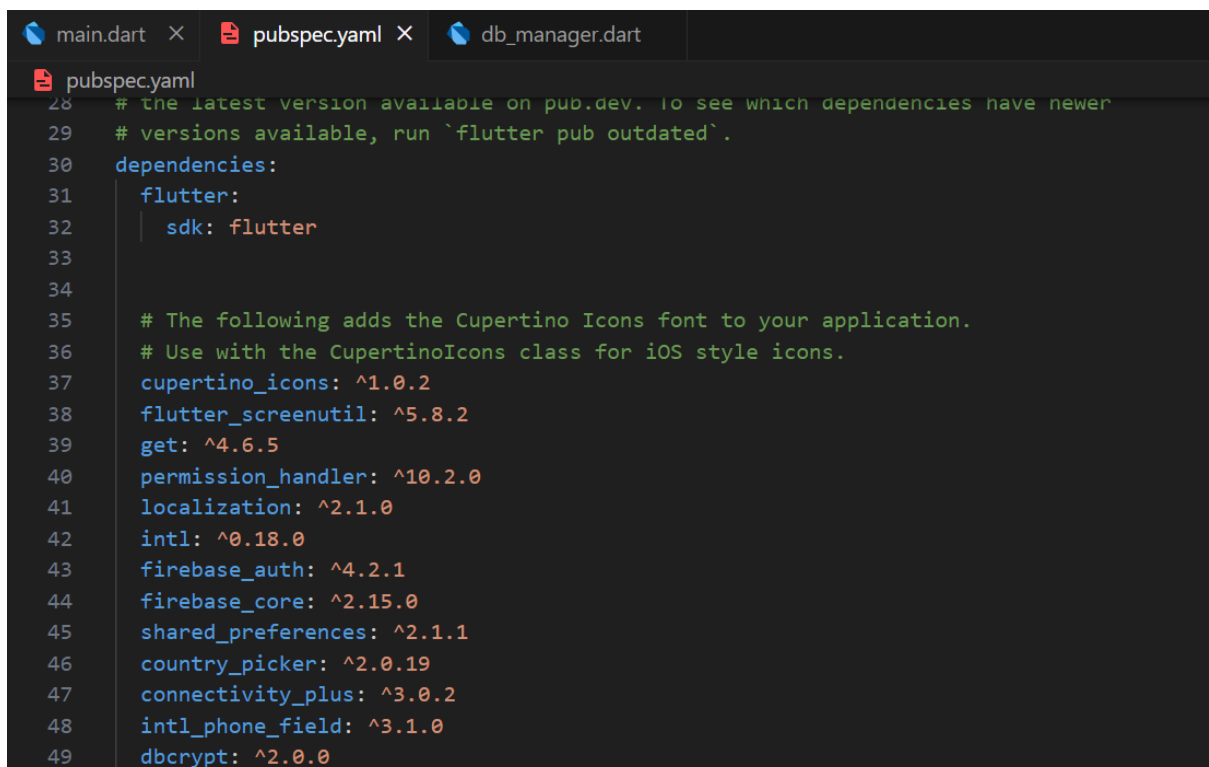
The main.dart is the entry function of the application from where the program begins its execution and it's within this function that the program logic is initiated.



The screenshot shows an IDE with the 'main.dart' file open. The Explorer panel on the left shows the project structure for 'grail-flutter', including folders like 'android', 'assets', 'ios', 'lib', 'controller', 'domains', 'model', 'utils', 'view', and files like 'app_localization.dart', 'db_manager.dart', 'main.dart', 'test', '.gitignore', '.metadata', 'analysis_options.yaml', 'pubspec.lock', 'pubspec.yaml', and 'README.md'. The main.dart file contains the following code:

```
lib > main.dart
26 void kPrint(Object? object) {
27
28 }
29
30
31
32 Future<void> main() async {
33   if (Platform.isMacOS) {
34     WidgetsFlutterBinding.ensureInitialized();
35     debugDefaultTargetPlatformOverride = TargetPlatform.iOS;
36   }
37   WidgetsFlutterBinding.ensureInitialized();
38   await Firebase.initializeApp();
39   await DbManager.openDB();
40   sharedPreferences = await SharedPreferences.getInstance();
41   final locationService = LocalizationController();
42   en = await locationService.loadLang('en');
43   de = await locationService.loadLang('de');
44   locale = await locationService.fetchLocale();
45   await SystemChrome.setPreferredOrientations(
46     [DeviceOrientation.portraitUp],
47   );
48   runApp(const MyApp());
49 }
50
51 class MyApp extends StatelessWidget {
52   const MyApp({Key? key}) : super(key: key);
53
54   // This widget is the root of your application.
55   @override
56   Widget build(BuildContext context) {
57     return ScreenUtilInit(
58       builder: (context) {
59         return kPrint('abdu, a year ago * "25/08/2023"');
60       },
61     );
62   }
63 }
```

Below is a screenshot of the pubspec.yaml file. This file is used to define all the packages and plugins required for the application.



The screenshot shows an IDE with the 'pubspec.yaml' file open. The file contains the following code:

```
main.dart x pubspec.yaml x db_manager.dart
pubspec.yaml
28 # the latest version available on pub.dev. To see which dependencies have newer
29 # versions available, run `flutter pub outdated`.
30 dependencies:
31   flutter:
32     sdk: flutter
33
34
35 # The following adds the Cupertino Icons font to your application.
36 # Use with the CupertinoIcons class for iOS style icons.
37 cupertino_icons: ^1.0.2
38 flutter_screenutil: ^5.8.2
39 get: ^4.6.5
40 permission_handler: ^10.2.0
41 localization: ^2.1.0
42 intl: ^0.18.0
43 firebase_auth: ^4.2.1
44 firebase_core: ^2.15.0
45 shared_preferences: ^2.1.1
46 country_picker: ^2.0.19
47 connectivity_plus: ^3.0.2
48 intl_phone_field: ^3.1.0
49 dbcrypt: ^2.0.0
```

The following is the class of registration controller, and we call them in registration screen. In this function we send http request to the backend API, which tells us that username is valid or not. And the strings with .tr means these are just keys and translations are in assets/translation file.

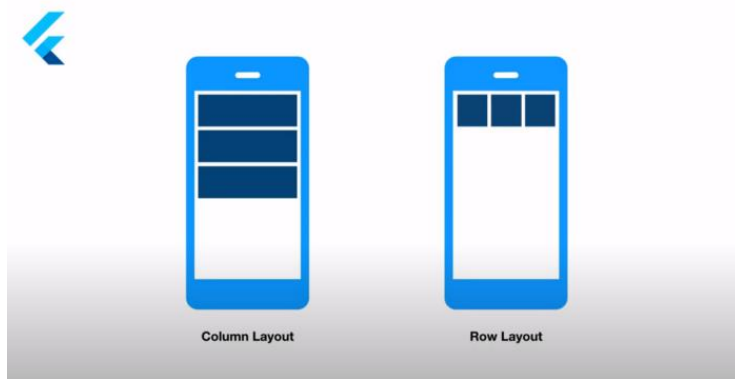
```
lib > controller > registration_controller.dart
16  class RegistrationController extends GetxController {
60
61
62  //Function to Verify Username
63  Future<bool> usernameVerification(String username) async {
64    String feedback = "";
65    final response =
66      await http.post(usernameVerificationURL, body: {"username": username});
67    final jsonBody = jsonEncode({"username": username});
68    if (response.statusCode == 200) {
69      return true;
70    } else {
71      final resultJson = jsonDecode(response.body);
72      if (resultJson['message'] == 'username already exists') {
73        feedback = 'user_name_already_exists'.tr;
74      } else if (resultJson['message'] == 'special characters not allowed') {
75        feedback = 'special_char_text'.tr;
76      } else {
77        feedback = 'invalid_username_text'.tr;
78      }
79      customSnackBar("error_text".tr, feedback, Colors.red);
80      return false;
81    }
82  }
83 }
```

Here's how to initialize a controller:

In the screenshot below, we use `GetBuilder` to initialize the registration controller within our widget.

```
lib > view > registration_screen.dart
10
11 class RegistrationScreen extends StatelessWidget {
12   const RegistrationScreen({Key? key}) : super(key: key);
13
14   static final GlobalKey<FormState> _form = GlobalKey<FormState>();
15
16   @override
17   Widget build(BuildContext context) {
18     double bottom = MediaQuery.of(context).viewInsets.bottom;
19     return GetBuilder<RegistrationController>(
20       init: RegistrationController(),
21       builder: (controller) {
22         return SingleChildScrollView(
23           child: Padding(
24             padding: EdgeInsets.fromLTRB(
25               40.w, 25.h, 40.w, bottom > 0 ? bottom : 20.h),
26             child: Column(
27               crossAxisAlignment: CrossAxisAlignment.center,
28               children: [
29                 Form(
30                   key: _form,
31                   child: Column(
32                     crossAxisAlignment: CrossAxisAlignment.start,
33                     children: [
34                       Text('username_min_char'.tr,
35                         style: fieldTitleTextStyle),
36                       SizedBox(
37                         height: 5.h,
38                       ),
39                     ],
40                   ),
41                 ],
42               ),
43             ),
44           ),
45         ),
46       ),
47     );
48   }
49 }
```

These are the built-in front-end functions of Flutter, which help us create and develop UI screens. We used `SingleChildScrollView` to make the screen scrollable, padding to center the content, and `Form` to define a form structure. This allows us to use a `TextField` validator for user input validation. The figure below illustrates the concepts of `Column` and `Row` layouts.



The following code represents the contact model. In this code, we create an instance of the contact model, which contains several fields and includes two functions: one for parsing JSON data into a variable instance and another for converting a variable instance back into JSON data.

```
class ContactModel {  
    ContactModel({  
        this.phone,  
        this.username,  
        this.name,  
    });  
  
    String? phone;  
    String? username;  
    String? name;  
  
    factory ContactModel.fromJson(Map<String, dynamic> json) => ContactModel(  
        phone: json["phone"] ?? "",  
        username: json["username"] ?? "",  
        name: json["name"] ?? "",  
    );  
  
    Map<String, dynamic> toJson() => {  
        "phone": phone,  
        "username": username,  
        "name": name,  
    };  
}
```