



PCNC 2000

Developing Flutter APP Manual.

26/7/2023

George Bannoura
Supervisor: Hamzeh Taradeh

Introduction:

This document explains the development flow of a Flutter App in our company, this document will explain in details the main concepts and procedures to work in flutter as our development department suggests.

This document can be used for Programmers, Project managers and team leaders.

Technology Stack

We use flutter with many tools to accomplish features implementation in a high UI and UX principles with secure and usable code to be used by other developers in present or future.

Important things to know as a developer in PCNC:

- 1- JavaScript: As dart language depends on JavaScript concepts, it is a must to know JavaScript main concepts and principles.
- 2- Advanced Knowledge in Flutter Framework and its components.
- 3- IDEs: We use “visual studio code”, “xcode for IOS”, Android Studio for Android.
- 4- Version Control: We use git to control our code and repositories, so please never upload the code to any drive, flash drive or GitHub.
Only push to Bitbucket using the provided email by PCNC.
Also delete the code in your devices when asked.
- 5- Styling: We provide a professional Figma project, you have to stick to the provided design unless you were asked otherwise.
- 6- Responsive Design: Understanding how to create web pages that adapt to different screen sizes and devices, because we implement the same code for IOS, Android and Web.
- 7- Always Debug and Test your code.
- 8- Frontend Testing: It is recommended to make Unit Testing for the work you have done.
- 9- Cross-Platform Compatibility: Understanding how to ensure web/Mobile applications work consistently across different platforms.
- 10- Performance Optimization: Knowledge of techniques to optimize frontend performance, like lazy loading, code splitting, and caching.
- 11- RESTful APIs: Understanding how to interact with backend services using RESTful APIs.

- 12- UX/UI Design: Basic understanding of user experience (UX) and user interface (UI) design principles.
- 13- Command Line: Proficiency in using the command line for various tasks like running scripts and managing projects.
- 14- Collaboration and Communication: Effective communication and collaboration skills to work with designers, backend developers, QA team, and other team members.
- 15- Security: Our apps must be secure unless it will make damage to stakeholders.

Development Environment Setup

- 1- Flutter SDK: Use stable and upgraded version of flutter sdk, and check with the team the exact version number they use.
- 2- Android Studio: Keep updated.
- 3- Version Control: Fork or SourceTree are recommended.
- 4- Xcode and CocoaPods: Must always be updated, you should know how to install, update and manage IOS pods, debug and create builds (.ipa build must be shared through diawi.com platform).
- 5- Project dependencies: Gradle version, Kotlin version and other dependencies must be upgraded by the team leader only.

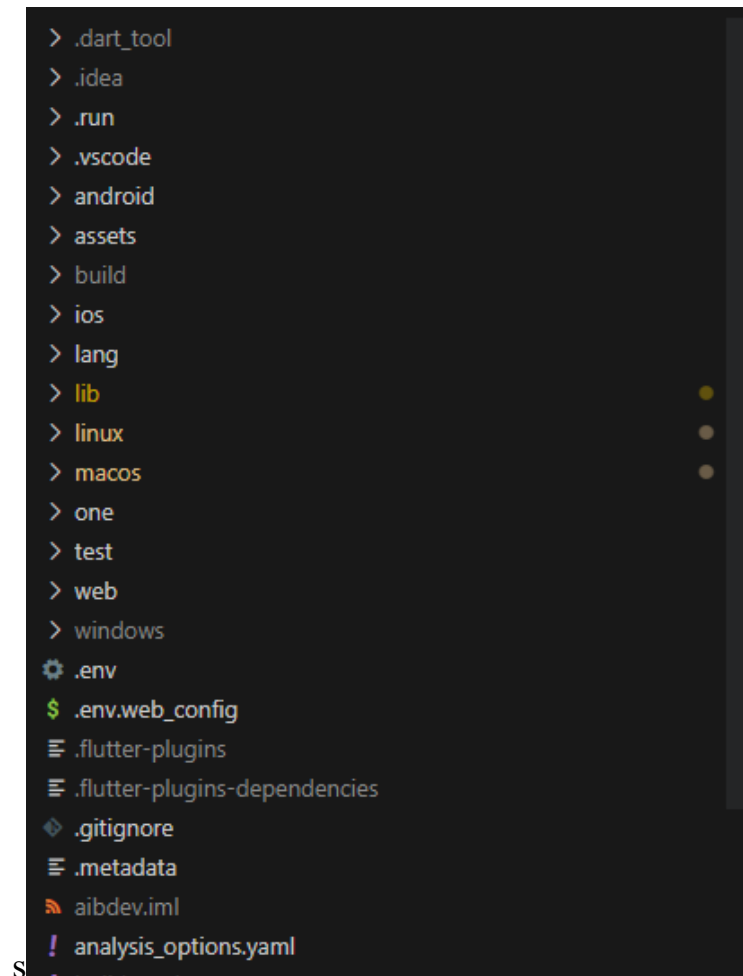
Project Structure

Now lets come to the advanced part of the document.

Clean code and Code Architecture are what makes the team work in a flexible way and features implemented and tested professionally.

- 1- Clean code: Variable naming, function naming and scope, separating UI from functionality.
- 2- Backend: The backend flow must be taken in notice while developing to reflect the flow of the backend in the frontend.
- 3- Clean Architecture and SOLID Principles: Flutter apps must follow the clean arch and SOLID Principles.

How are the folders divided?



Android: This folder contains the native Android-specific configuration files.

main.dart: The entry point of your app where execution begins.

assets: This folder is used to store asset files such as images, fonts, and other resources required by the app.

ios: This folder contains the native iOS-specific configuration files.

web: This folder contains web-specific configuration files.

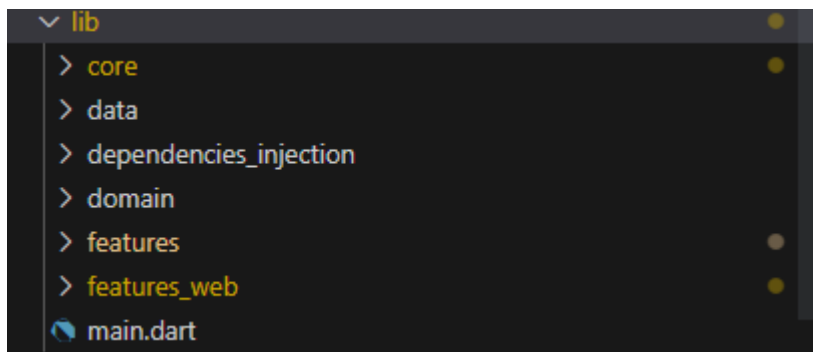
pubspec.yaml: The pubspec.yaml file is where you declare the app's dependencies, assets, and other configuration details.

.env: This file contains the most secure and critical configurations, as Base-URL, Certificates used for certificate pinning.

Base-URL: We have three different urls, Dev url used by the development team,

User Under Testing (UAT) URL: that is used by stakeholders, and ofc production url that is published in the store.

Lib Folder:



1. **lib**: The main folder that contains most of the Dart code for the app.
2. Presentation Layer: (Features + Features_web)
 - **ui**: Contains the user interface related components like screens, widgets, and presentation logic.
 - **Controllers**: Contains the Business Logic Components (Getx) responsible for managing the state and business logic of the application.
3. Domain Layer:
 - **entities**: Contains the business entities or models representing the core data of the application.
 - **Interactors_impl**: Defines the application's use cases or interactors, which are the application's business rules and operations.
 - **interactors**: Abstract classes defining the contract for data access and communication between the domain and data layers.
4. Data Layer:
 - **gateway**: Concrete implementations of the repositories defined in the domain layer. and responsible for data retrieval from APIs.
 - **services**: Shared services for main features as cards and accounts.
5. Dependency_injection:
 - **dependency_injection**: Contains configuration related to dependency injection (e.g., using packages like **getx**).

State Management:

We use mostly Getx for state management with stateless widgets, or stateful widget when needed in some cases.

Handling Data and APIs:

We use Dio package and its configuration already done in base-gateway for get, post, update, delete and its headers.

Localization and Internationalization

We use English and Arabic only, as we define one key for a text in LangKeys class and we call it by .tr to get en or ar text for the key.

For example:

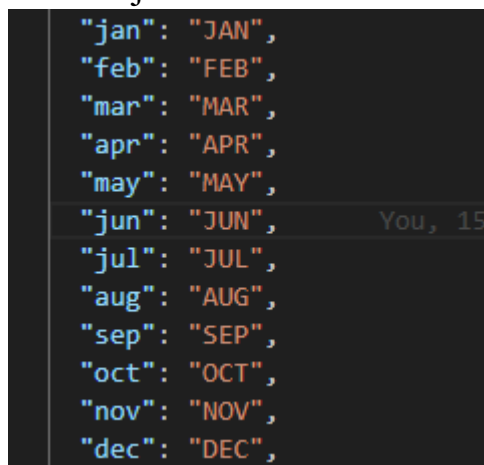
I want to use months in English and Arabic:

1- I define a key for each word/ sentence:

A screenshot of a code editor showing Dart code for defining month keys. The code is as follows:

```
//Months
static const String JAN = "jan";
static const String FEB = "feb";
static const String MAR = "mar";
static const String APR = "apr";
static const String MAY = "may";
static const String JUN = "jun";
static const String JUL = "jul";
static const String AUG = "aug";
static const String SEP = "sep";
static const String OCT = "oct";
static const String NOV = "nov";
static const String DEC = "dec";
```

2- In en.json file I define as so:

A screenshot of a code editor showing the content of an en.json file. The content is a JSON object with month keys and their English values. The code is as follows:

```
{
  "jan": "JAN",
  "feb": "FEB",
  "mar": "MAR",
  "apr": "APR",
  "may": "MAY",
  "jun": "JUN",
  "jul": "JUL",
  "aug": "AUG",
  "sep": "SEP",
  "oct": "OCT",
  "nov": "NOV",
  "dec": "DEC",
}
```

4- in ar.json file I define as so:

```
paymentHistory : "تقارير الدفعات",
"jan": "كانون الثاني",
"feb": "شباط",
"mar": "آذار",
"apr": "نيسان",
"may": "أيار",
"jun": "حزيران",
"jul": "تموز",
"aug": "آب",
"sep": "أيلول",
"oct": "تشرين الأول",
"nov": "تشرين الثاني",
"dec": "كانون الأول",
"minutes": "دقائق",
"delegatedPerson": "الشخص المفوض"
```

Security Considerations

Security is an important role in development:

- 1- Always use secure, tested and updated third party packages.
- 2- To connect dart side to native side, dart -> kotlin and dart -> swift, Use Pigeon channel instead of MethodChannel.
- 3- If you want to add a new package discuss with the team leader.
- 4- Apply security techniques given to you in the project implementation time.

Development Flow

I will explain how to develop a new feature in flutter app:

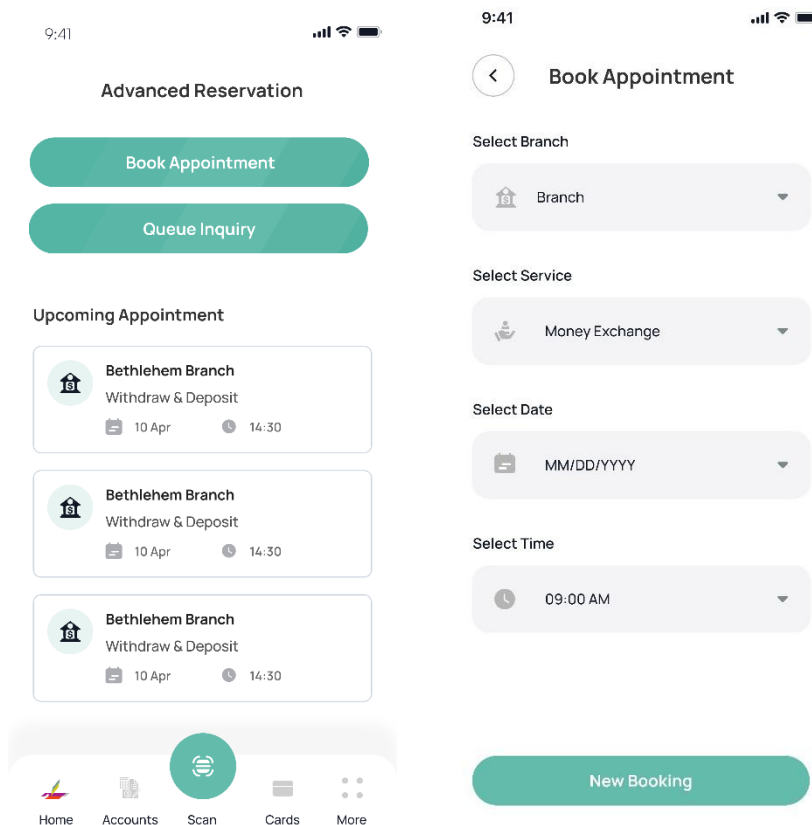
Let's say you have received a task as: Remote Booking Screen

this screen will have two functions:

1. View current appointments:
screen will have 2 sections:
 1. Book an appointment button
 2. View upcoming appointments section:
 1. Branch name
 2. service name
 3. Date
 4. time
2. Book an appointment:
this will open new window where user have to pick the following:
 1. Branch : call API to get Booking branches
 2. Services: call API to get selected branch available services
 3. date : call api to get available working days
 4. Time: call Api to get available times for the selected days
 5. new booking button will call add appointment

What should you do?

- 1- You should make sure that you have received UI design.



- 2- Make sure you have received the API integration with the backend.
It should contain: api route, API METHOD, Request Body , Response Body.

Base route: /api/v1.0/ticket

Appointments API:

METHOD: POST

Route: appointments

Request Body:

```
public class GetAppointmentsRequest
{
    public long BranchId { get; set; }
    public string MobileNumber { get; set; }
}
```

Response: List of AppointmentResponse

```
public class AppointmentResponse
{
    public string TicketNumber { get; set; }
    public string AppointmentDay { get; set; }
    public string AppointmentTime { get; set; }
    public bool ServiceAvailForAppt { get; set; }
    public bool ServiceAvailForWalkIn { get; set; }
    public long ServiceDepartmentId { get; set; }
    public long ServiceId { get; set; }
    public string ServiceName { get; set; }
    public long ServiceGroupId { get; set; }
    public BranchResponse BranchResponse { get; set; }
}

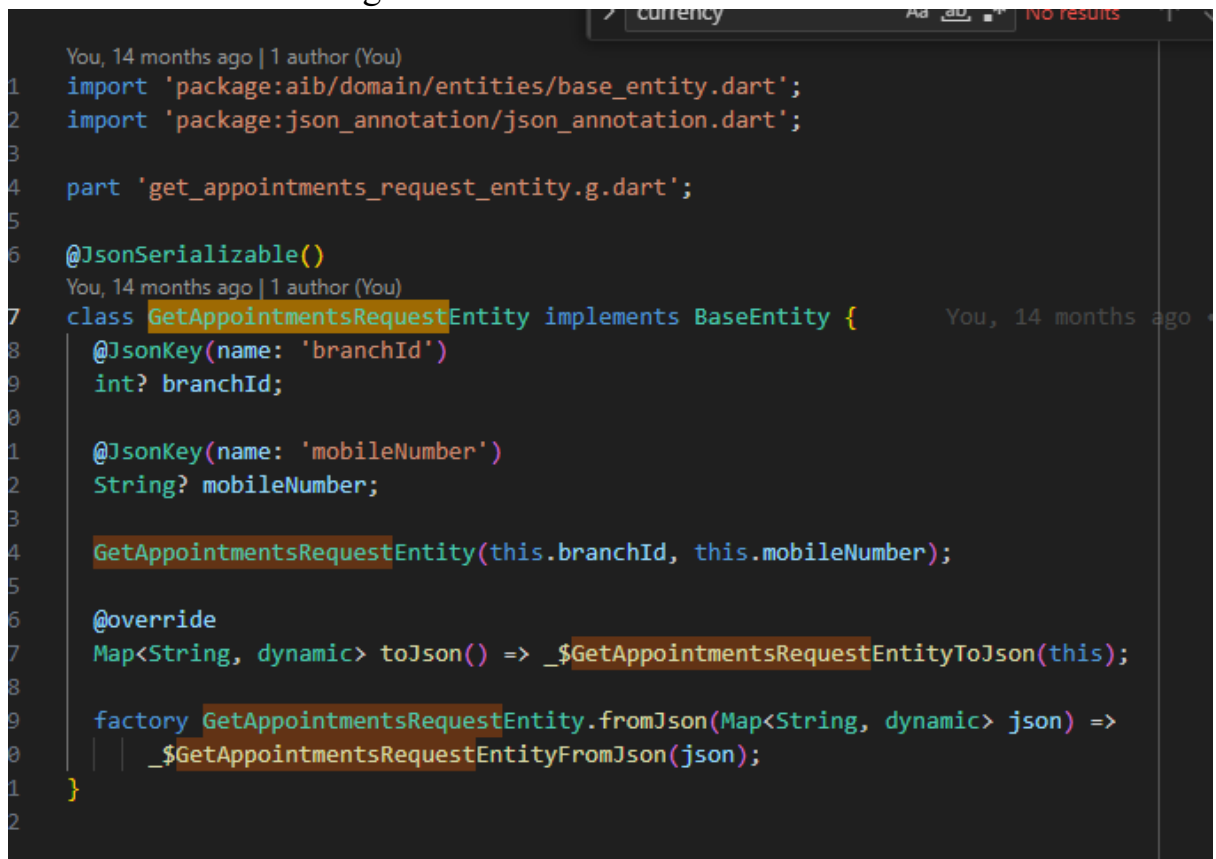
public class BranchResponse
{
    public long Id { get; set; }
    public long CityId { get; set; }
    public long DepartmentId { get; set; }
    public double Distance { get; set; }
    public string Identity { get; set; }
    public bool IsWorkingNow { get; set; }
    public double Latitude { get; set; }
    public double Longitude { get; set; }
    public string Name { get; set; }
}
```

```

public WorkingShiftHours[] WorkingShiftHours { get; set; }
}
public class WorkingShiftHours
{
public string From { get; set; }
public string To { get; set; }
}

```

3- You have to start making the entities:



```

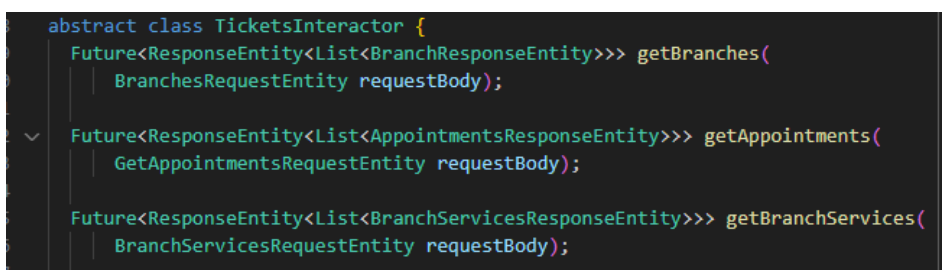
You, 14 months ago | 1 author (You)
1 import 'package:aib/domain/entities/base_entity.dart';
2 import 'package:json_annotation/json_annotation.dart';
3
4 part 'get_appointments_request_entity.g.dart';
5
6 @JsonSerializable()
7 You, 14 months ago | 1 author (You)
8 class GetAppointmentsRequestEntity implements BaseEntity {
9   @JsonKey(name: 'branchId')
10   int? branchId;
11
12   @JsonKey(name: 'mobileNumber')
13   String? mobileNumber;
14
15   GetAppointmentsRequestEntity(this.branchId, this.mobileNumber);
16
17   @override
18   Map<String, dynamic> toJson() => _$GetAppointmentsRequestEntityToJson(this);
19
20   factory GetAppointmentsRequestEntity.fromJson(Map<String, dynamic> json) =>
21     | _$GetAppointmentsRequestEntityFromJson(json);
22 }

```

Then run the command: flutter packages pub run build_runner build --delete-conflicting-outputs

4- Then go to the domain/interactors and create appointment (ticket) related .dart file if not exist.

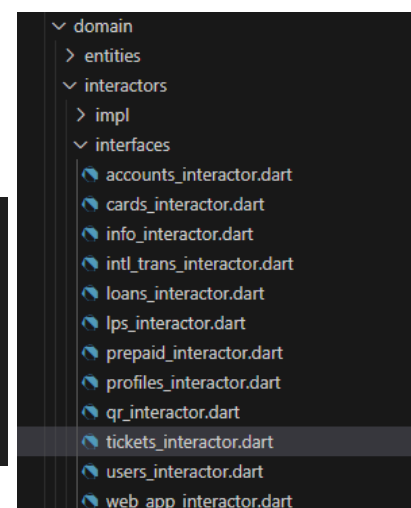
then add abstract function as so:



```

abstract class TicketsInteractor {
  Future<ResponseEntity<List<BranchResponseEntity>>> getBranches(
    BranchesRequestEntity requestBody);
  Future<ResponseEntity<List<AppointmentsResponseEntity>>> getAppointments(
    GetAppointmentsRequestEntity requestBody);
  Future<ResponseEntity<List<BranchServicesResponseEntity>>> getBranchServices(
    BranchServicesRequestEntity requestBody);
}

```



```

domain
├── entities
├── interactors
├── impl
├── interfaces
│   ├── accounts_interactor.dart
│   ├── cards_interactor.dart
│   ├── info_interactor.dart
│   ├── intl_trans_interactor.dart
│   ├── loans_interactor.dart
│   ├── lps_interactor.dart
│   ├── prepaid_interactor.dart
│   ├── profiles_interactor.dart
│   ├── qr_interactor.dart
│   └── tickets_interactor.dart
├── users_interactor.dart
└── web_app_interactor.dart

```

Lets understand that function:

This is an abstract function with no body, it defines the function name “getAppointments” and Request body entity and Response body entity.

5- Go to the domain/interactors_impl and create the same

```
class TicketsInteractorImpl implements TicketsInteractor {
    final TicketsGateway gateway;

    TicketsInteractorImpl({required this.gateway});

    @override
    Future<ResponseEntity<List<AppointmentsResponseEntity>>> getAppointments(
        GetAppointmentsRequestEntity requestBody) {
        return gateway.getAppointments(requestBody);
    }
}
```

The function is an implementation for the abstract function which works as a mediator to call gateway function.

6- The gateway function is missing: lets implement it.

```
Future<ResponseEntity<List<AppointmentsResponseEntity>>> getAppointments(
    GetAppointmentsRequestEntity requestBody) async {
    try {
        await postAsync(API_TICKET_APPOINTMENTS, requestBody);

        return isSuccess()
            ? ResponseEntity(data: toList(AppointmentsResponseEntity.fromJson))
            : ResponseEntity(error: getErrorBody());
    } catch (e) {
        return ResponseEntity(error: CommonError.defaultError());
    }
}
```

7- Now you are ready to call the function from the GetxController.

A- Call the interactor

```
//Interactors.  
final TicketsInteractor _ticketsInteractor =  
    Injector().get<TicketsInteractor>();
```

B- Handle the response:

```
Future<void> getAppointmentsList() async {  
    try {  
        startLoading();  
        appointmentsList.value = [];  
        String mobileNumber = await SecureStorage.getAsync(SSKeys.mobileNumber);  
        var requestBody = GetAppointmentsRequestEntity(null, mobileNumber);  
        var response = await _ticketsInteractor.getAppointments(requestBody);  
        if (!response.isSuccess()) {  
            stopLoading();  
            return;  
        }  
        stopLoading();  
        appointmentsList.addAll(response.data!);  
    } catch (e) {  
        loadingController.stopLoading();  
        _showErrorMsg(LangKeys.SOMETHING_WENT_WRONG);  
    }  
}
```