Code Quality Review report

Architecture

- Flutter Framework divides folders based on programming languages. In the figure below , There is a folder for android files, iOS files, for web as well as test files. We also created a folder for other files such as images and custom fonts. Our application code lives in the lib folder

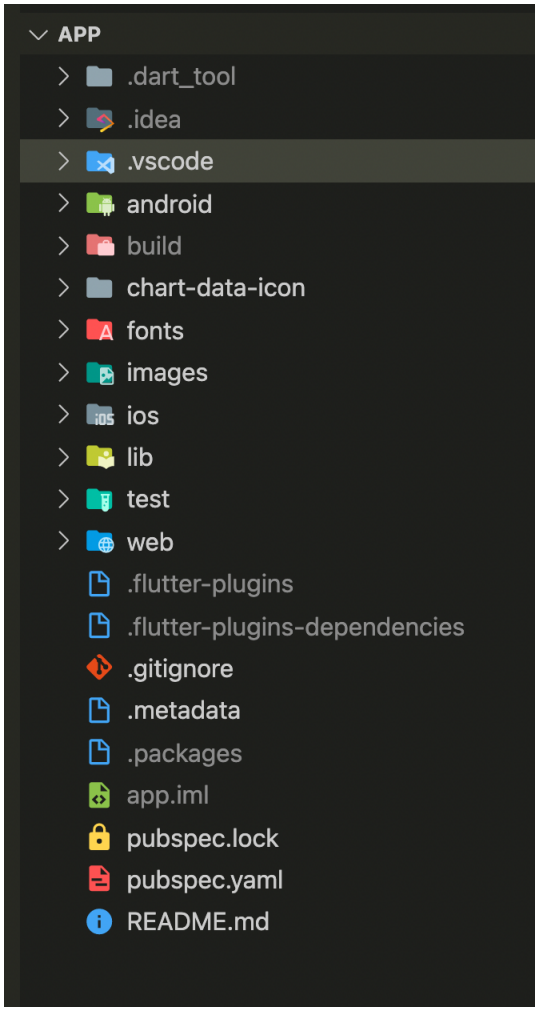
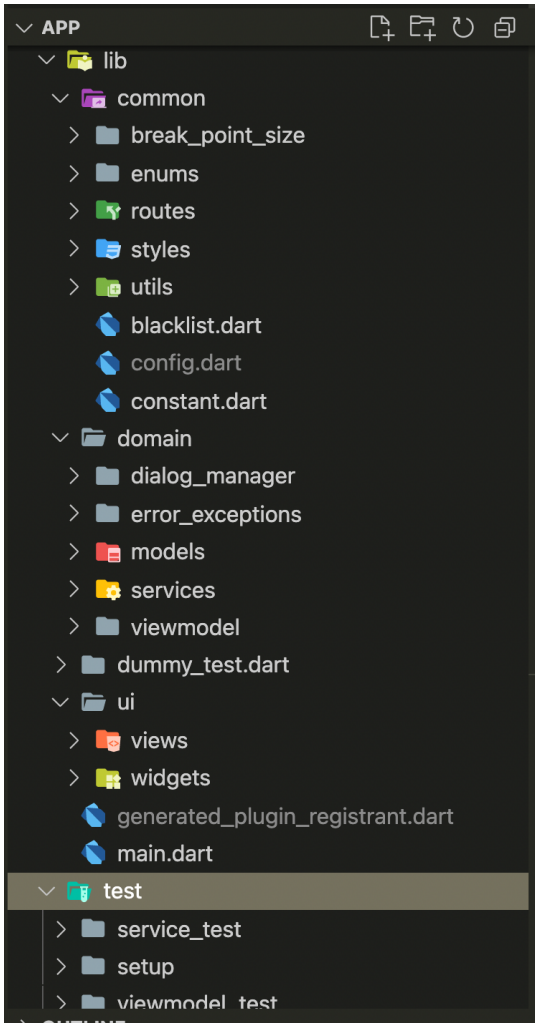


Figure 1: Overview figure

- Our application is split into layers and tiers. We Followed Model View Viewmodel (MVVM) architecture to build our application. MVVM is a

common architecture to build web apps and mobile applications and it was appropriate for us to use this architecture .

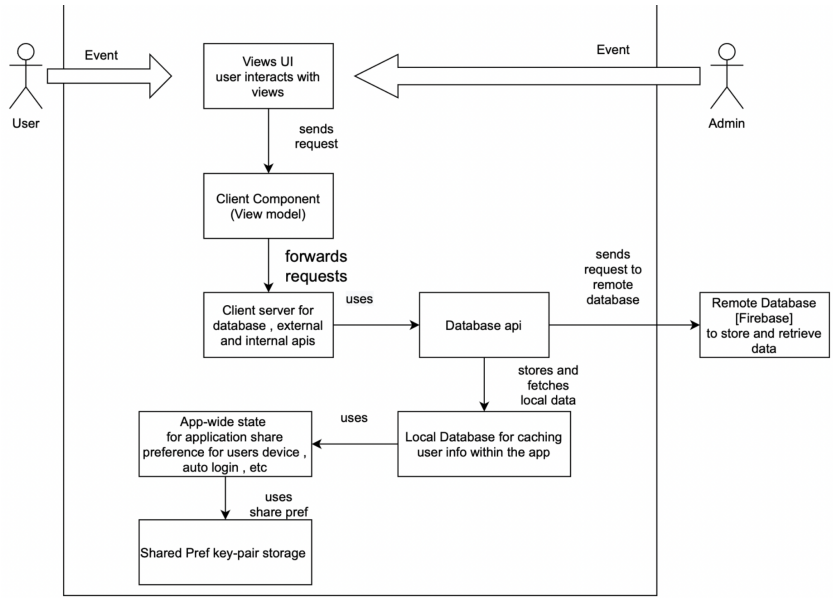
Figure2: MVVM



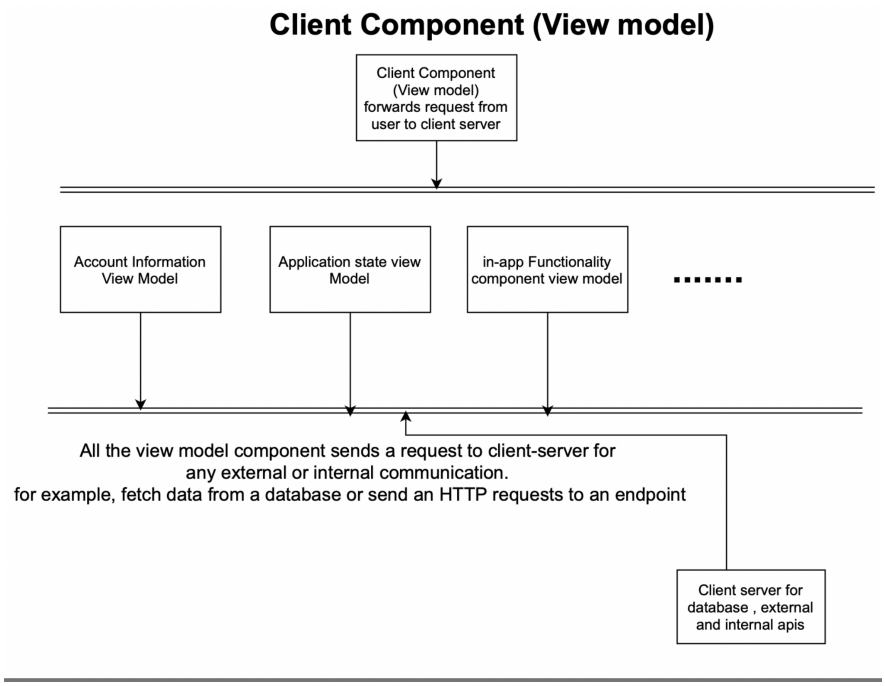
App Structure figure 3

In the diagram below , It shows the flow of our application.

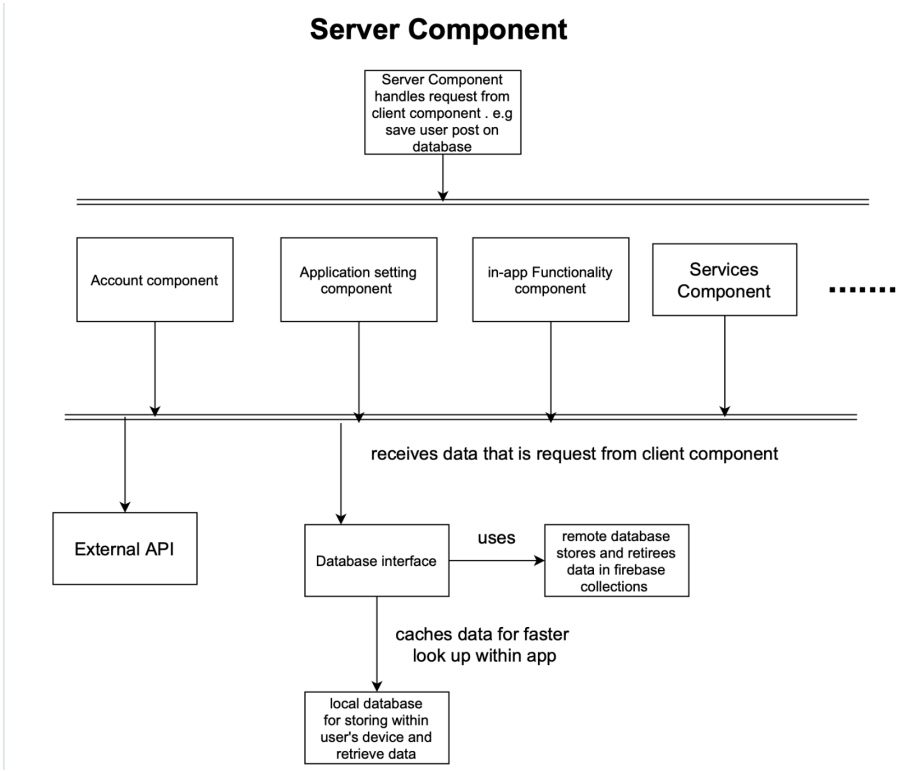
- User interacts with the view. For example, the user clicks on the add post button to share a useful link.
- In the middle , We have created a client component model which acts as a controller to forward requests to appropriate services or client servers to handle user requests. The client component communicates with external service and client server.
- The client server layer is responsible for the abstraction of the data sources. Model and ViewModel work together to get and save the data.



Mobile Application Figure 4



ViewModel Figure



Model Figure re

Code Formatting

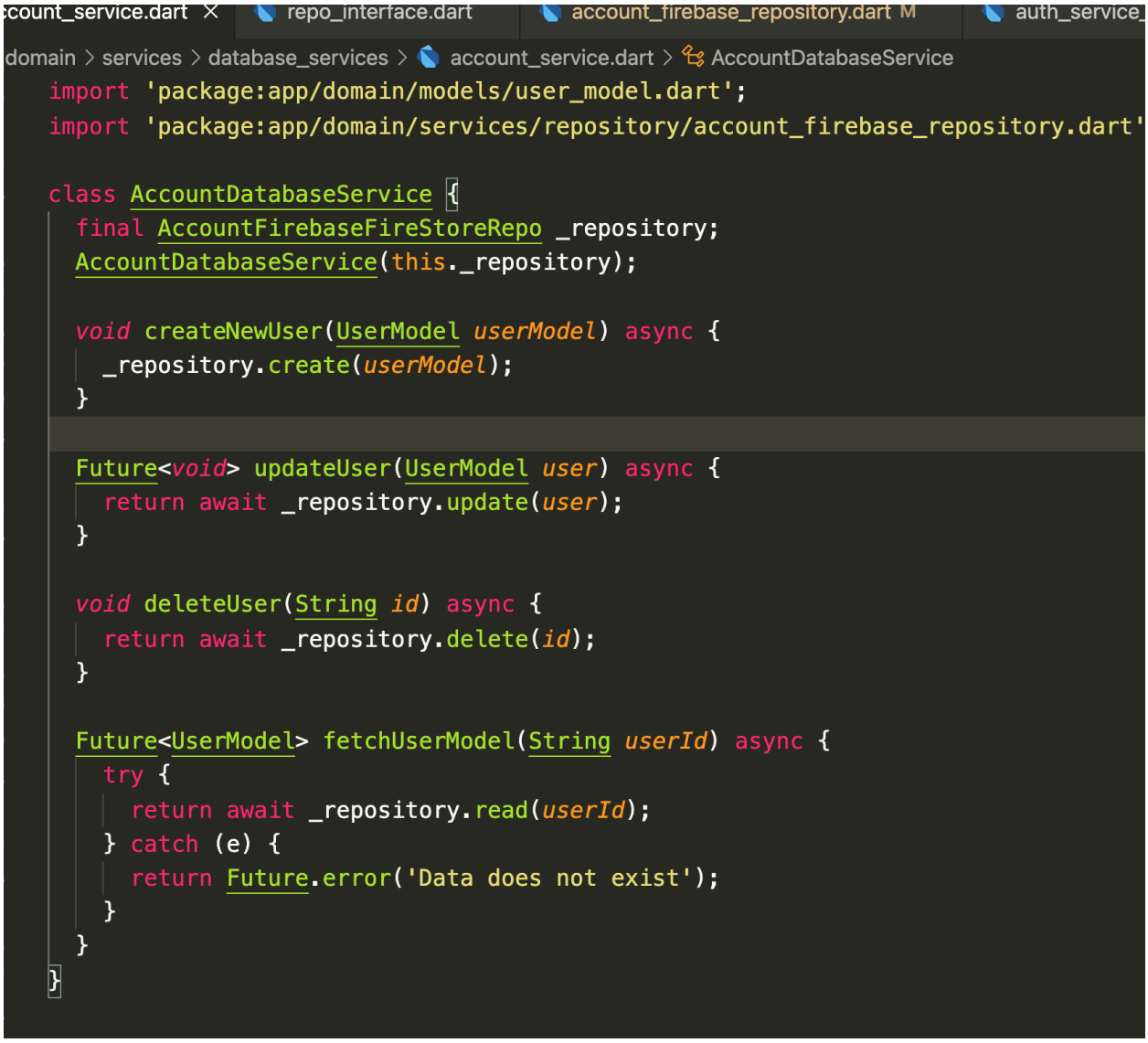
- For this project, We have installed prettier extensions to help us make our code consistent within the team. This includes proper indentation and functions line break. Prettier is a recommended extension for flutter framework. Therefore, we are meeting the latest standard for flutter projects.
- For our database design, We use a repository design pattern to help us design a well maintainable code. All of our modal classes would need to implement RepositoryInterface

```dart
// this class must be implemented by concrete classess
abstract class RepositoryInterface<T> {
  Future<void> delete(String id);
  Future<T> read(String id);
  Future<void> update(T data);
  Future<void> create(T data);
}
```

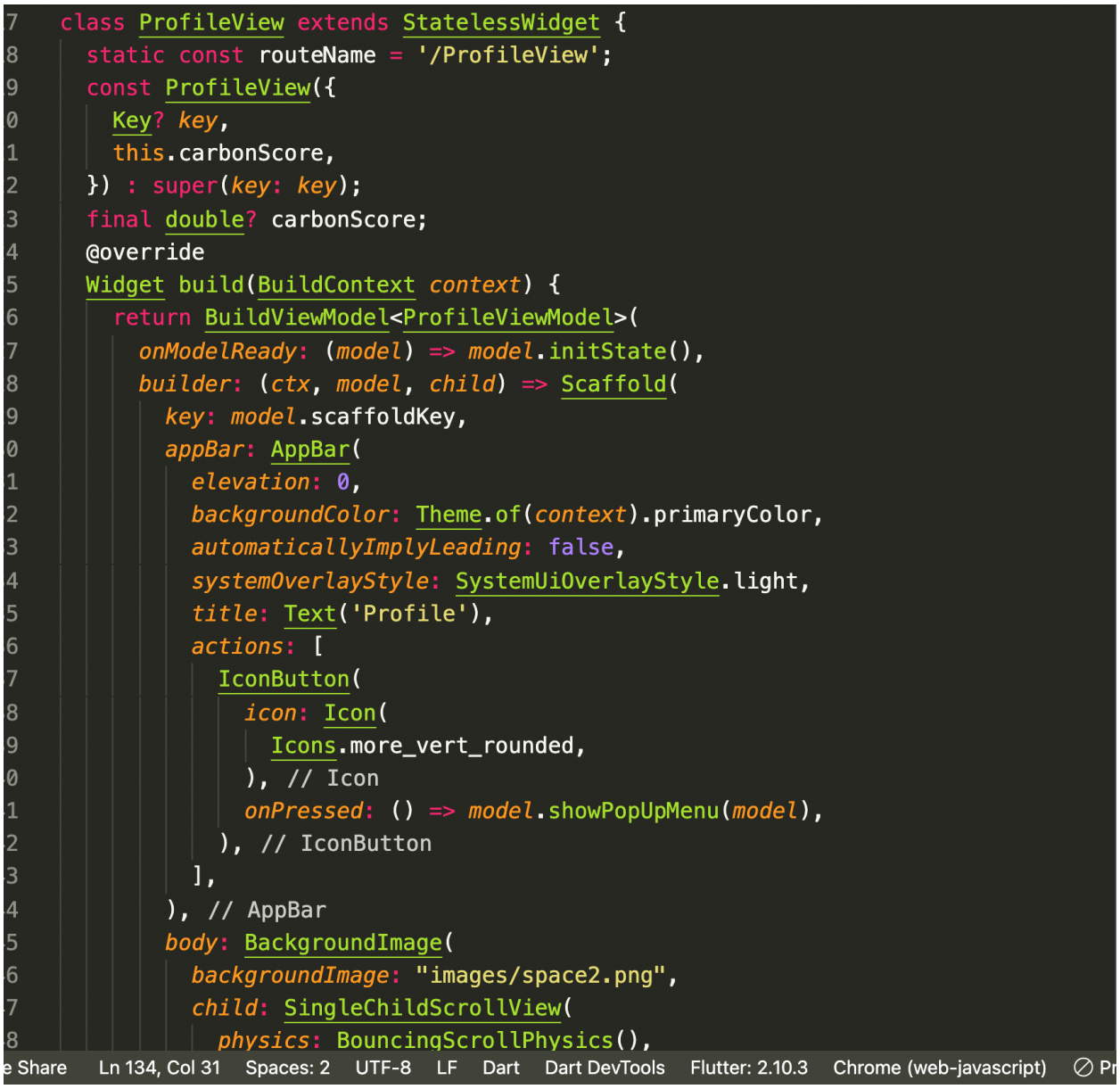
```dart
class AccountFirebaseFireStoreRepo implements RepositoryInterface<UserModel> {
  AccountFirebaseFireStoreRepo();
  @override
  Future<void> create(UserModel user) async {
    firestore.collection(USER_COLLECTION).doc(user.userId).set(
          user.toMap(),
        );
  }

  @override
  Future<UserModel> read(String id) async {
    final doc = await firestore.collection(USER_COLLECTION).doc(id).get();
    final data = Optional.ofNullable(doc.data());
    if (data.isPresent()) {
      return UserModel.fromMap(data.get()!);
    }
    return Future.error('Data does not exist');
  }

  @override
  Future<void> update(UserModel user) async {
    await firestore
        .collection(USER_COLLECTION)
        .doc(user.userId)
        .set(user.toMap());
  }


  @override
  Future<void> delete(String id) async {
    // TODO: implement delete
```

re Ln 18, Col 1 Spaces: 2 UTF-8 LF Dart Dart DevTools Flutter: 2.10.3 Chrome (web-javascript) ⊘ Pr

- Our Account firebase repo implements a repository interface for our userModel Object. This helps make our code clean and easier to understand
- Now in order to use this class in our database class, we need to pass it down in the constructor which will give us loosely coupled in our codebase



We have implemented a similar code structure to help us design clean code in our user interface using Model View ViewModel architecture. In our View is only responsible for displaying the User Interface (UI). E.g displaying a button to upload profile image.



```
class ProfileView extends StatelessWidget {
  static const routeName = '/ProfileView';
  const ProfileView({
    Key? key,
    this.carbonScore,
  }) : super(key: key);
  final double? carbonScore;
  @override
  Widget build(BuildContext context) {
    return BuildViewModel<ProfileViewModel>(
      onModelReady: (model) => model.initState(),
      builder: (ctx, model, child) => Scaffold(
        key: model.scaffoldKey,
        appBar: AppBar(
          elevation: 0,
          backgroundColor: Theme.of(context).primaryColor,
          automaticallyImplyLeading: false,
          systemOverlayStyle: SystemUiOverlayStyle.light,
          title: Text('Profile'),
          actions: [
            IconButton(
              icon: Icon(
                Icons.more_vert_rounded,
              ), // Icon
              onPressed: () => model.showPopUpMenu(model),
            ), // IconButton
          ],
        ), // AppBar
        body: BackgroundImage(
          backgroundImage: "images/space2.png",
          child: SingleChildScrollView(
            physics: BouncingScrollPhysics(),
```


View Figure

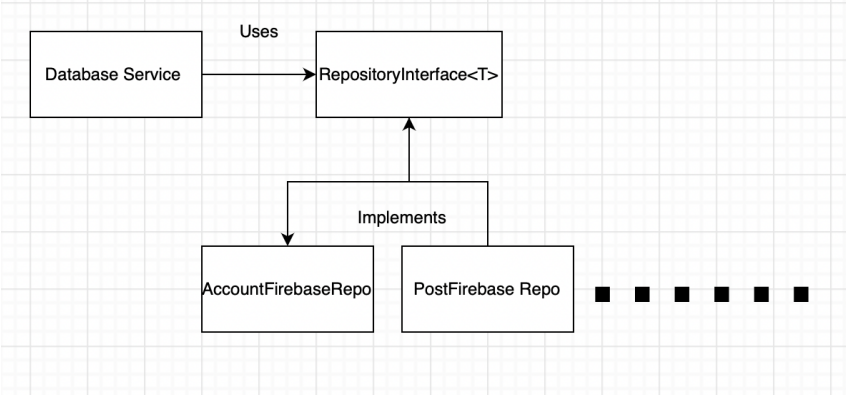
Our ViewModel handles the user request and forwards data to the appropriate service and shows the result on the view for the user to see. E.g allowing users to upload their image profile and save it on the local device of the user. Each property of our model classes are private (hence underscore variable indicates private) and final (meaning it can only be set once and cannot change). This makes our code more robust and easy to maintain.

```
class ProfileViewModel extends BaseViewModel {
  static const IMAGE_KEY = 'IMAGE_KEY';
  static const USER_INFO = 'USER_INFO ';
  final _authService = locator<AuthService>();
  final _navService = locator<NavigationService>();
  final _scaffoldKey = GlobalKey<ScaffoldState>();
  final _userAccountDb = locator<AccountDatabaseService>();
  final _auth = locator<AuthService>();

  UserModel? _userModel;
  File? _image;
  final imagePicker = ImagePicker();
  // final _questionaireBox = locator<Boxes<QuestionaireResult>>();
  // all getters
  GlobalKey<ScaffoldState> get scaffoldKey => _scaffoldKey;
  File? get image => _image;
  UserModel? get user => _userModel;

  // all methods

  void signOut() {
    _authService.signOut();
    notifyListeners();
    _navService.navigateAndReplce(UserRegisterationFormView.routeName);
  }

  void showPopUpMenu(model) {
    showModalBottomSheet(
      context: _scaffoldKey.currentState!.context,
      builder: (ctx) => ProfileMenuPopUp(
        command: model,
      ),
```

ViewModel figure

Coding best practice and design principles

- **Styling:** We have used good coding practices to build our project. If you look at Figure 3, We have created a style folder to keep our styling (fonts,colors , etc) consistent throughout the app. If we need to change a particular font size or color , We would only change it in one place.
- **Proper comments:** We have used comments to give credit and explain our code. We also used TODO comments for any sub-tasks that need to be done in the future.
- **Repository design pattern:** We have implemented a repository design pattern to hide data access implementation logic from the business logic.



- **Single Responsibility Principle (SRP)** - We utilized this principle in our application. As mentioned above, View is only responsible for only one thing which shows UI to the user(e.g button to read an article post). The viewmodel is responsible for forwarding requests to the appropriate services and transforming this data, which can be shown on the views. The model is responsible for encapsulating data.

- **program to an interface not implementation -** our code is written in a way that we can program to an interface instead of to implementation . An example for this is our database repository interface design where we can implement as many databases as we

can without breaking any code (e.g using repository interface to create test database or forum post database) .

- **Open-Close Principle -** our class objects and function as well as our widget UI are open for an extension but closed to modifications. An example of this are our services, we can add service or replace service without breaking any codes.
- **Dependency Injection -** We have used a package called get_it to help us inject dependency which is accessible anywhere in our app and allow us to decoupled interfaces from a concrete Implementation.

Reference:

Get_it: Dart Package https://pub.dev/packages/get_it.

Hussain, Maraj. “Flutter : MVVM Architecture Best Practice Using Provide & HTTP.” *Medium*, Medium, 8 Sept. 2021, https://medium.com/@ermarajhussain/flutter-mvvm-architecture-best-practice-using-provide-http-4939bdaae171.

Santos, Bruno. “Why to Use Repository Pattern in Your Application Flutter?” *Medium*, Medium, 13 June 2020, https://medium.com/@bruno.santos_/why-to-use-repository-pattern-in-your-application-flutter-549c0739a892.

Dane. “Stacked: Flutter Package.” *Dart Packages*, 19 Jan. 2022, https://pub.dev/packages/stacked.

MVVM Figure 2 Image:
https://encrypted-tbn0.gstatic.com/images?q=tbn:ANd9GcSV1OjBJLufmbf0QD6DZG7Oym9Rq3YKSI3KFA&usqp=CAU