Code Quality Review Report

Humane Transport

Date last edited: Apr 9, 2021

Team Members:
Kelly Holtzman, Sana Khan, Mansi Patel, Clark Inocalla

—

Code Formatting

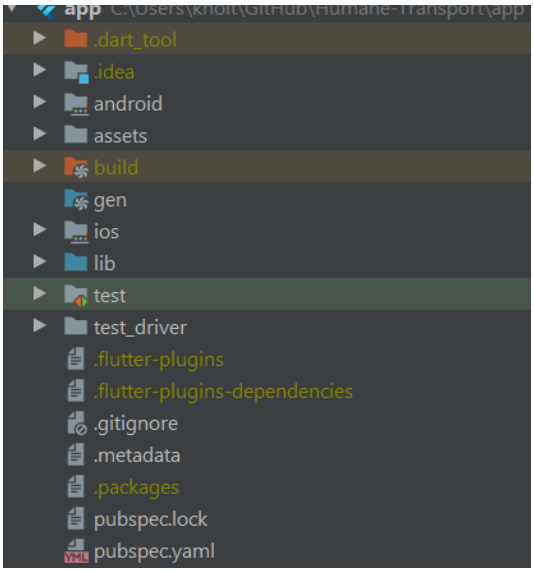
For this project we installed and used a code formatting script that ran every time a team member would commit changes - called Pre-Commit (2021). The script controls the alignment, white space, indentation, and line breaks of all the programming languages we use: kotlin, Java, Dart, and Swift; and configuration files YAML, YML, and XML. The formatting was checked not only by the pre-commit hook, but again by the GitHub Actions that ran after every code push and merge.

For things not caught by the pre-commit hook, they are instead caught during code review/pull request reviews by the reviewers. This includes naming conventions.

Our code formatter is set to the standard for the Flutter framework, so we are sure that we are meeting the latest standards. You can review our pre-commit configuration file and the formatting rules we used here.

—

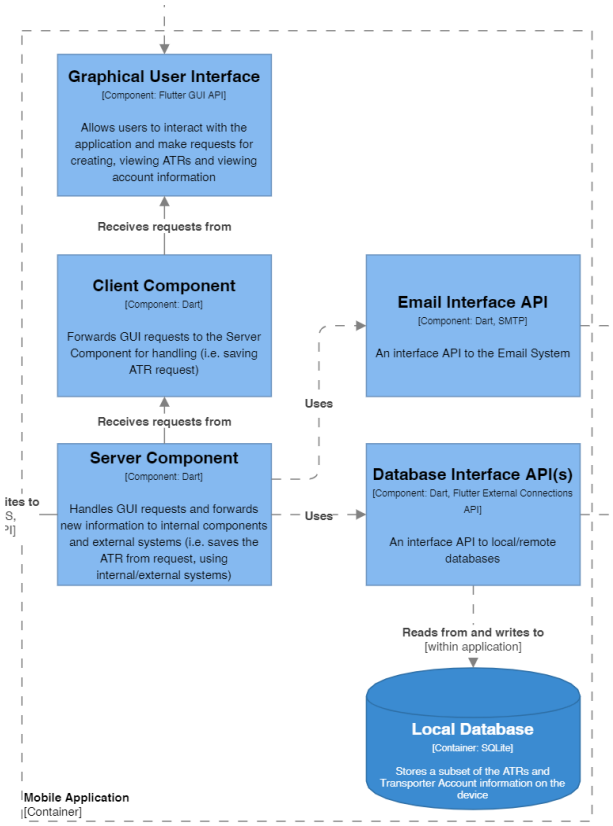
Architecture



Split into respective files (HTML, JavaScript and CSS)

The standard Flutter framework is divided into folders by programming language (for example, the `android` folder contains kotlin and Java code files).

Figure 1: Project Overview



Split into multiple layers and tiers as per requirements (Presentation, Business and Data layers)

Our application used the client-server architecture as is common in internet applications. We further divided the architecture into components we felt were logically separate.

Figure 2: Mobile Application

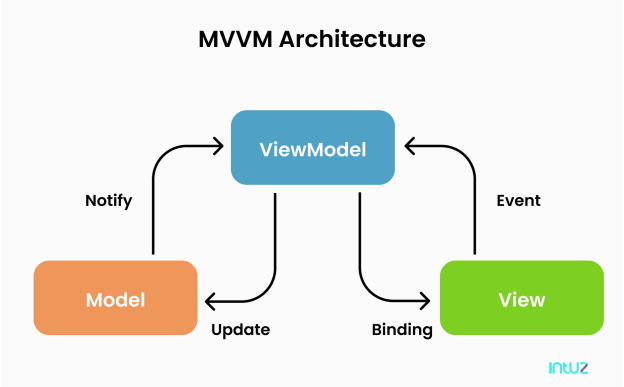


Figure 3: MVVM Architecture (D, 2020)

Design patterns: Use appropriate design pattern (if it helps), after completely understanding the problem and context

Further division was done using a model architecture similar to the Model-View-Controller, called the Model-View-ViewModel (MVVM). We decided to use MVVM as it fit the reactive nature of mobile applications.

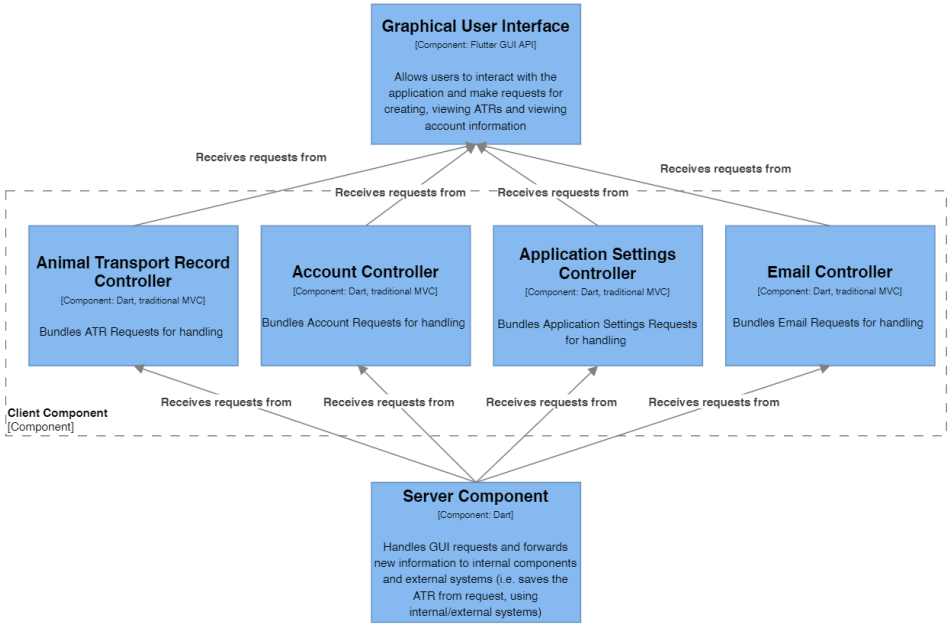


Figure 4: Client Component

Code is in sync with existing code patterns/technologies.

Within the client (and server, not shown) architecture, we further integrated the MVVM view models (called controllers in the above image) so as to separate application experiences logically.

For further information on our chosen architecture, see our client-server architecture documents.

Coding best practices

We prevented most interface appearance/CSS hardcoding with the provision of a styling file with defined constant: `style.dart`. Further, we defined a global application theme in `humane_transport_app.dart` which applies to the entire application except where otherwise overridden.

All comments in our application fall under three types: 1. TODOs with assigned tasks, 2. documentation comments for classes or widgets with particularly interesting uses, and 3. credit comments for patterns or widgets we have used for interesting purposes. All TODOs with assigned tasks have a matching task in our

GitHub Issues. Any code that is not our own has both the comment in the source file, and credit has been given in the application licenses as available in the deployed application.

Our application follows functional programming concepts as much as possible (consider Moutafis, 2020), with separation of interface and business logic as per the MVVM architecture. For example, see our services files which exemplify common functionality directly injected throughout the application (improving testability and more).

Further coding practices not considered commonplace were derived from experience and generic Flutter best practices (2021).

Non-Functional requirements

Our application code speaks for itself in terms of readability, maintainability (even for custom portions of the app), testability, reusability, and extensibility. The reader may consider the following files which exemplify the above concepts:

1. the family of database files which were designed with the expectation of being replaced and extended
2. the services locator which contains classes (services) which may be directly injected anywhere (testability, reusability, maintainability)
3. the family of form files, all similar but with enough differences that further reduction of duplication is difficult without sacrificing readability. The forms can be replaced at the lowest level (a single field) to an entire section without disruption of other forms, widgets, services, etc.

Our application is secured by client-side rules (within the source code of the app) which prevent actions without corresponding authentication. These rules are baked into application navigation and accessible through the `auth_service.dart`. Further security is provided with database rules which ignore requests that do not meet authentication requirements and requests not of the expected format (these rules are not shared publicly).

Better performance outside of Flutter's recommended practices (2021) is made possible with the use of asynchronous transactions throughout the entire codebase. The family of database files exemplifies the infectious nature of asynchronous transactions, forcing handling through the entire application as needed.

Object-Oriented Analysis and Design (OOAD) Principles

Our application is based on MVVM Architecture which allows us to have a code base that follows single responsibility principle (SRP), open-close principle, Liskov substitutability principle, interface segregation and dependency injection.

SRP. As shown in Figure 3, our code base is divided into three major categories: View - ViewModel - Model. The View is responsible for one thing only, which is defining the structure of what the user sees on the screen (e.g. buttons). The ViewModel is responsible for transforming model information into values that can be displayed on a view. The model is only responsible for encapsulating the applications data (e.g. data from database).

Open-Close Principle. The software elements (classes/functions) in our code are written in such a way where it is open for extensions, but closed for modifications (e.g. services and the database interface). Our Services are a great example that satisfies Liskov substitutability principle and Interface segregation as they can be extended without breaking changes: we can replace a Service with another one satisfying the same abstract functions.

Dependency Injection. To implement dependency injection to our application, we are using a dart package called `getIt` (Flutter Community, 2021). This dart package allows us to decouple interfaces from a concrete implementation and able us to access it everywhere in the app. This is especially useful when testing as we can replace instances with mocks very simply. The use of Services in injection is a much better method compared to singletons. for example.

References

Flutter. (2021, April 4). *Performance best practices.* https://flutter.dev/docs/perf/rendering/best-practices

Flutter Community. (2021, April 9). *get_it Dart package.* https://pub.dev/packages/get_it

D. N. (2020, November 19). *A Complete Guide And Comparison Of MVC and MVVM.* Intuz. https://www.intuz.com/blog/guide-on-mvc-vs-mvvm

Mackier, D. (2020, April 26). *Flutter and Provider Architecture using Stacked.* FilledStacks.

https://www.filledstacks.com/post/flutter-and-provider-architecture-using-stacked/#how-does-stacked-work

Moutafis, R. (2020, August 5). *Why developers are falling in love with functional programming*. Towards data science. https://towardsdatascience.com/why-developers-are-falling-in-love-with-functional-programming-13514df4048e

pre-commit. (2021, April 4). *A framework for managing and maintaining multi-language pre-commit hooks*. https://pre-commit.com/