Design and Implementation of Mobile Applications

POLITECNICO
MILANO 1863

DatingFoss
Design Document

Authors: F C
F A

Professor: Luciano Baresi

Academic Year 2021-2022

Version 1.0

Contents

1 Introduction

DatingFoss is a classic dating app where user can share each other infromation, pictures and messages. Other users' profile are presented one after the other. When a user is interested to an other profile, he can swipe right to say to the application he likes it. Only when both users like each other we have a match, and a conversation can start.
Every user has a **profile**, that is the collection of information he wants to share with others. These can be textual information like a small biography, some pictures, the sex, or even some important dates.
Every type of information can be private or public:

- **Private**: Is visible only to the users he likes
- **Public**: Is visible to everyone

1.1 Definitions, Acronyms, Abbreviations

1.1.1 Definitions

- **User**: an account registered in the application with username and password (more precisely the username, public key pair).
- **Logged User**: fixed the application device, a user that is currently logged inside the application using that device.
- **Searching criteria**: referred to a user, some properties that another user should satisfy in order to be interesting for the referred user.
- **Possible Partner**: referred to a user, a user that matches the searching criteria of the referred user.
- **Like**: an action done by the logged user inside the application to communicate his interest to another user.
- **Match**: two users match if and only if they like each other.
- **User information**: every kind of data written by a user about himself. It can be either public or private.
- **Private**: a user information is private if the only way another user can access it is receiving a like from that user.
- **Public**: a user information is public if it is not private.
- **Profile**: referred to the user, represent the collection of all user's information.
- **Biography**: a type of user information; a small textual description.
- **Pictures**: a type of user information; it's an image.

1.1.2 Acronyms

- **DD**: Design Document
- **FOSS**: Free and Open Source Software
- **JWT**: JSON Web Tokens
- **API**: Application Programming Interface
- **REST**: Representational state transfer
- **DTO**: Data Transfer Object

1.1.3 Abbreviations

- **BIO**: Biography
- **APP**: Application

1.2 Revision history

- Version 1.0 — First Release

1.3 Document Structure

- Chapter 1 is the introduction.
- Chapter 2 provides details about the system architecture, its components and how they interact.
- Chapter 3 describes the UX and shows the user interface providing mock-ups.
- Chapter 4 presents the implementation, integration and test plan. Also the testing strategies are described.

2 Architectural design

2.1 Overview: High-level components and their interaction

The system is based on a typical client-server architecture. More precisely the client is a cross platform mobile application built in flutter (the main focus of this document), and the server is an Asp.Net 6.0 Web application. Communication is implemented through RESTful APIs. We devided the logic of the app in different macro components, each one in a separete package (apart from the core components that are in the main project).

- **Core Components**: Contains BLoCs, resonsible for all the business logic for the App functionality and all the Widgets responsible for the presentation layer
- **Models**: Classes without any business logic. Everything that is domain specific.
- **Services**: The interfaces (abstract classes) that contains only the contracts of the business, data and communication layers. Informally we will call services not only those interfaces but also theirs implementations.
- **Communication**: Small class that has the only goal to send and receive (raw) data through the application. This is the only point where the application communicates with the server. It handles also the session logic with JWT token and such.
- **Controllers**: Using an abstraction of the communication class, contains the logic on how to convert DTOs to json objects and vice versa. Contains all the contracts of the server endpoints divided by a semantical topic (for example we have the UserController, DiscoverController etc.).
  Note: They are a one-to-one map of the Asp.Net Web Application Controllers.
- **Repositories**: Map the interfaces of controllers in a different interface more suitable to be used for the application. For example, while the Notification controller exposes a method to (long) poll asynchronously the server for notification, the repository exposes a stream of notification. In that way the implementation detail of long polling is completely hidden to the rest of the application, that sees only a stream of notifications.

Note: The abstractions of communication, controller and repositories are all services. There are also other services such as the EncryptionService.

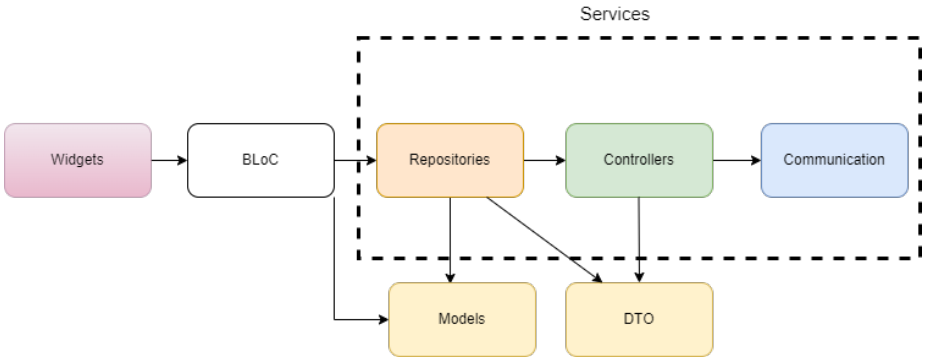


Figure 1: Components Dependency

2.2 Component view

In the following diagram only the application client is analyzed in detail

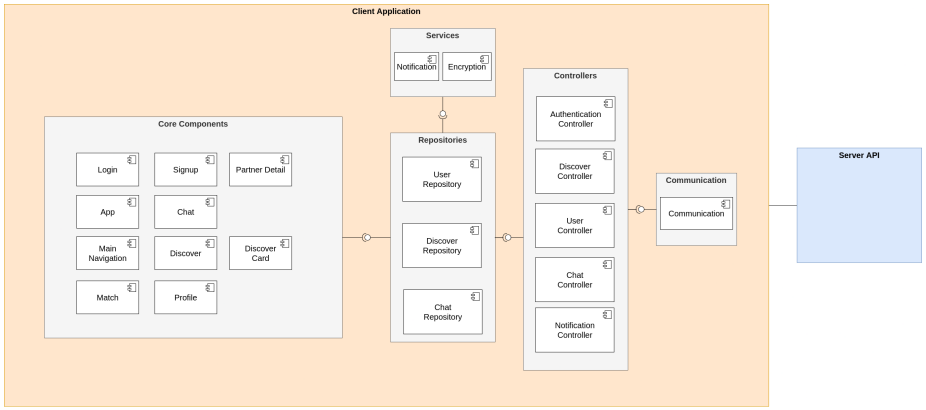


Figure 2: Exapnded Components Dependency

2.2.1 Core Components

- **AppBloc**: is responsible to show the login page (if the user is not authenticated) or the main screen (otherwhise), this BLoC also store the authenticated user.
- **LoginBloc**: is responsible for the validation of the login credential and the submission of those.
- **SignupBloc**: is responsible for the validation and submission of the user's data.

- **MainBloc**: is responsible for the navigation between the main pages.
- **MatchBloc**: is responsible for keeping the records of the partners liked by the user and the partners that like the user, find the matches.
- **ChatBloc**: is responsible for managing the chats with the messages.
- **DiscoverBloc**: get the possible partners from the repository and handle all the event in the Discover view.
- **DiscoverCardBloc**: is used to mange the single partner to fetch pictures and handle the action inside each discover card.
- **PartnerDetailBloc**: fetch the partner from the given username and put it in its state to be able to display it.
- **ProfileBloc**: handle the changes of the user data, saving them on the server.

2.2.2 Repositories

- **ChatRepository**: is used to get and send messages.
- **DiscoverRepository**: is used to fetch all the data reguarding the partners, unencrypt everything that can be unencrypted, put like (send add like message), put dislike (send remove like message), and receive like message.
- **UserRepository**: is used for the authentication, fetch and edit of all the data related to the user of the app.

2.2.3 Controllers

- **AuthenticationController**: Handles everything is needed in the signup, login phases.
  - *SignUp:* Register a user into the system using a username and a public key.
  - *GetLoginChallenge:* Returns a login challenge for a given username.
  - *LogIn:* Logs a user into the system using the username and the login challenge
  - *LogOut:* Logs a user out from the system.
- **ChatController**: Exposes a single API for send a message to a user.
  - *SendMessage:* Send a message (string) to a user using the username. It's assumed that the user is already logged in (i.e. in the current RESTful implementation, a valid JWT is sended with the request).
- **DiscoverController**: Exposes all APIs for fetching other users data.

  - *FetchPartners:* given some filtering options (for example the number of results), returns a list of possible partners for the logged user, according to his public searching criteria.
  - *FetchPartner:* Returns all user info from the username, with the exception of the images' content.
  - *SendSealedLikeMessage:* Sends a like message to a user.
  - *FetchPartnerPublicPicture:* Fetches a public image content of a user by the name.
  - *FetchPartnerEncryptedPrivatePicture:* Fetches a private image content of a user by the name.
  - *FetchPublicKey:* Gets the public key of a user by the username.

- **NotificationsController**: Exposes a single method to fetch a notification. A notification could be a like message or a chat message. Basically it's everything the server needs to to client:
  - *Fetch:* Waits asynchronously until a new notification is available, and returns it back.

- **UserController**: Handles everything is related to the logged user.
  - *FetchUser:* Fetch all the logged user data.
  - *PushUser:* Updates all the logged user data.
  - *PushPublicPic:* Adds a public picture to the user profile.
  - *FetchPublicPics:* Fetches all specified public images contents from the logged user.
  - *PushEncryptedPrivatePic:*: Adds a private picture to the user profile.
  - *FetchEncryptedPrivatePics:* Fetches all specified private images contents from the logged user.
  - *FetchSymmetricEncryptedKeys:*: Fetches all symmetric keys used by the user until his registration.
  - *PushSymmetricEncryptedKeys:* Adds a symmetric key to the keys collection of the user.

The controller package has also associated a models folder with the DTOs that are used for the transfer of data to and from the server.
Each controller has two implementation, one that exploit the Communication package to communicate with the server and one that simply mock those function for testing purpose.

2.2.4 Other Services

- **EncryptionService**
  It is compose by two separate services:
  - **SymmetricEncryption**
    Is a AES implementation that generates random keys and encrypt and decrypt data.
  - **AsymmetricEncryption**
    Is an RSA implementation that generates a pair of key and encrypt, decrypt, sign and verify data.

2.2.5 Communication

The communication interfaces exposes the following methods:

- **Post:** Post an object to and endopoint (with authentication if needed), and returns back the raw response.
- **Get:** Returns back an object from an endpoint specifying some query parameters (with authentication if needed).
- **Download:** Downloads a raw file (i.e. an iamge) from a specified endpoint (with authentication if needed).
- **SetToken:** Sets an access token for all next requests.
- **UnsetToken:**: Remove the access token for all next requests.

2.2.6 Server

The Server is based on the Clean Architecture of Jason Taylor together with CQRS pattern. It's composed of 4 projects:

- **Domain:** Contains all the domain models. In this project there is the very core of the application. There is no logic at all here.
- **Application:** This project references only the Domain project. Contains all the business logic of the application and defines all the interfaces of the services needed. However in this project there are not the implementation of these services.
- **Infrastructure:** References the Domain and the Application projects. Contains all the implementation of the services defined in the Application layer. For example, While the application layer contains an interface *IUserCrud* containing all the methods for create, update, delete and update a user, the implementation of this interface is contained here in the infrastructure project. The diea behind that is to be able to plug in and out different infrastructure layers for the same application projects without any modification

- **Server:** This is a Asp.Net 6.0 MVC Web Application, where controllers contains the REST endpoints. Authentication is implemented with Json Web Token together with the standard Claims-based authentication of ASP.NET Core. This project references all the other projects but the only reason why the infrastructure project is referenced is to bind Application level interfaces to Infrastructure implementations in the dependency injection container at the startup. In this project are defined all the DTOs, that are mapped to the other projects in domain models and/or queries/commands.

CQRS is implemented trough the NuGet package MediatR. This, together with the dependency injection and the Clean Architecture offers great decoupling and testability.

2.3 Runtime view

In this section are shown the sequence diagrams of some features. They are useful to clarify the runtime behaviour of the components involved for the main functionalies

2.3.1 Put like

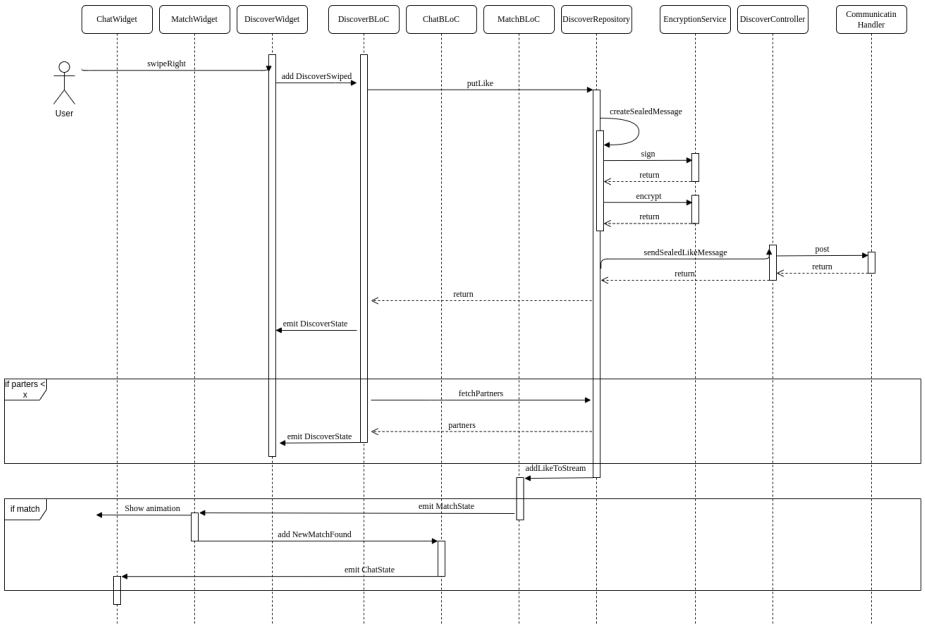


Figure 3: Put like sequence diagram

2.3.2 Signup

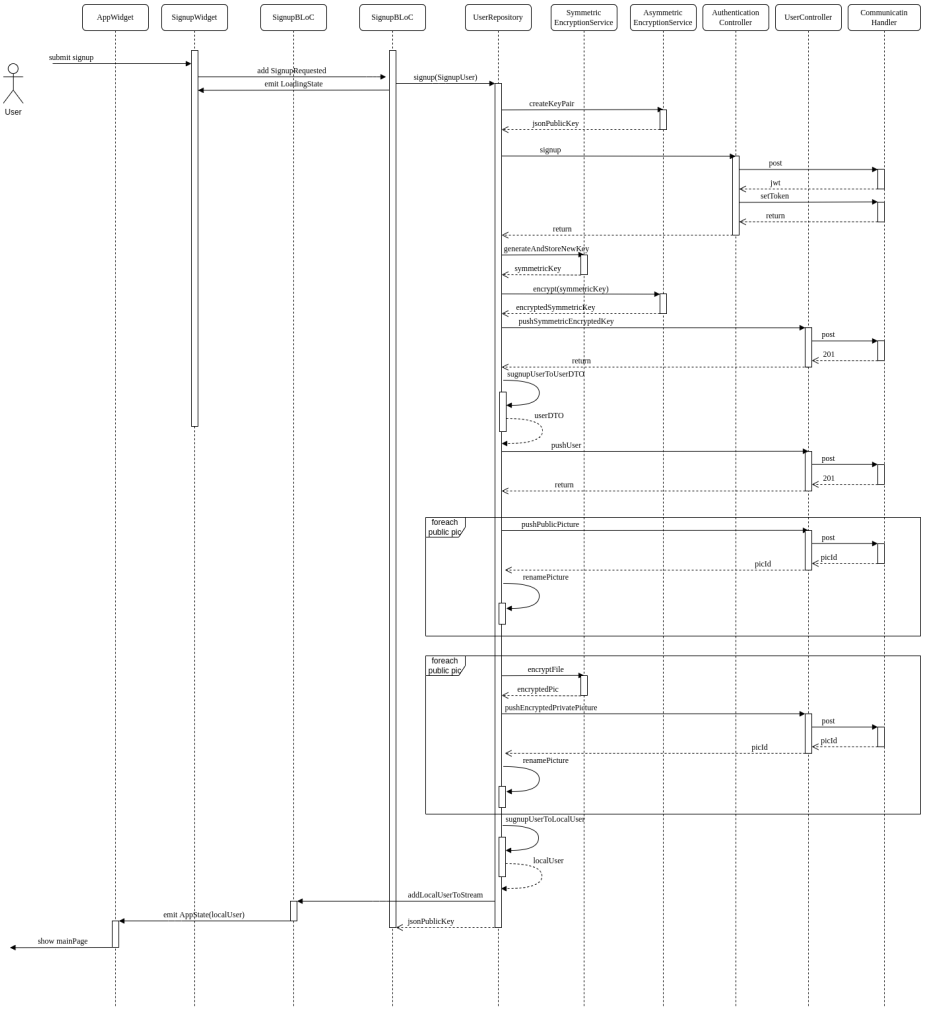


Figure 4: Signup sequence diagram

2.4 Selected architectural styles and patterns

- **BLoCs**: A state mangement solution.
  We organized the code in BLoCs (Business Logic Components) to decople the UI logic for presentation and all the business logic and state of the app.

- **DTOs Pattern**: Data Transfer Objects.
  The communication between the the server and the user application uses DTOs so that if the sever API will change the client has to update only the DTOs and not the model.

- **DI Pattern**: Dependency injection.
  We follow the DI Pattern to separate the concerns of constructing objects and using them. This way ensures that an object or function which wants to use a given service should not have to know how to construct those services. A great advantage is also noticable in writing tests because it's really easy to mock and inject a class dependecies.

- **RESTful architecture**: REST is a software architectural style.
  The communication between the server and the user application uses HTTP(S) requests which follows REST principles. One of the most important REST principles is that the interaction between the client and server is stateless between requests. Each request from the client to the server must contain all of the information necessary to understand the request. For instance the client wouldn't notice if the server were to be restarted at any point between the requests. This is achieved using a uniform and predefined set of stateless operations based on strict use of HTTP request types.

2.5 Other design decisions

2.5.1 Encryption Protocol

To garantee the privacy of the user we created a protocol to minimize the knowledge of the server.

- Every user have an asymmetric key pair generaetd from the username and password that garantee their identity.
- All the private data of a user is encrypted with a randomly generated symmetric key before been send to the server.
- When a user (Alice) want to put a like to another user (Bob), it create a sealed message to send to Bob in this way:
  - Sign Bob's username with her private key.
  - Create a PackedMessage with: the signed username, the list of symmetric key used for the private data and Alice's username.
  - Encrypt the PackedMessage with Bob's public key.
- When Bob receive a like message, he decrypt the message with his private key, verify that his own username is signed by Alice, store the symmetric keys that will use to decrypt the private data.

3 User interface design

3.1 UX Diagrams

In all the diagrams the actions to go back are omitted to simplify the diagrams. The UX is slightly different for tablet and smartphone to reach a better experience.

- **Smartphone**

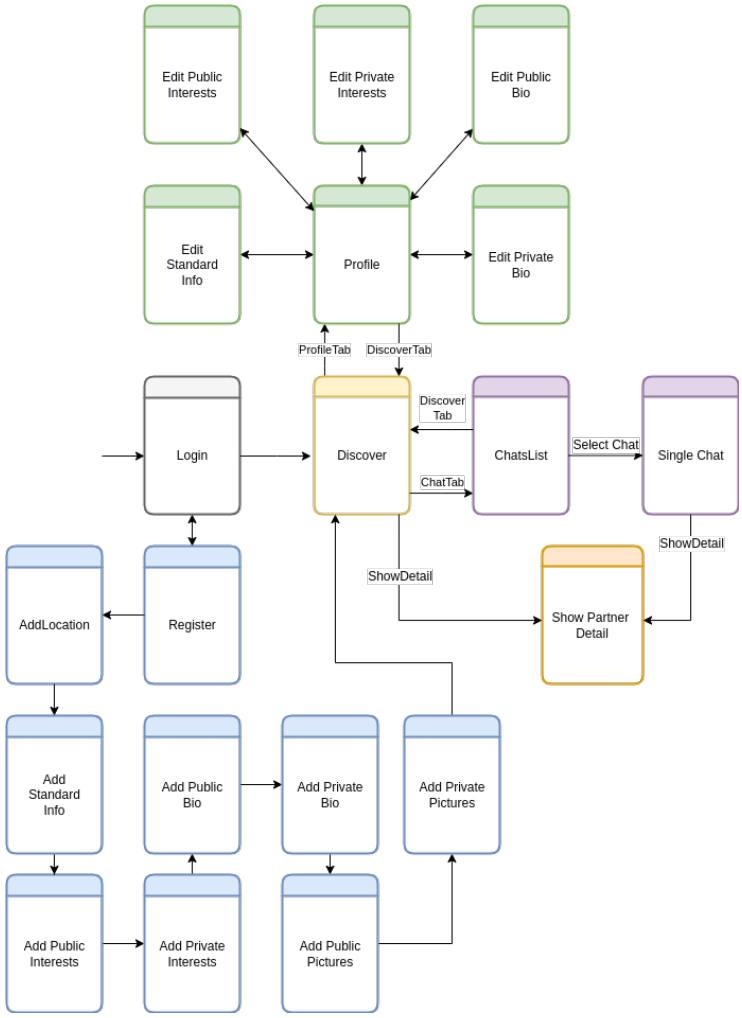


Figure 5: Phone UX diagram

- **Tablet**

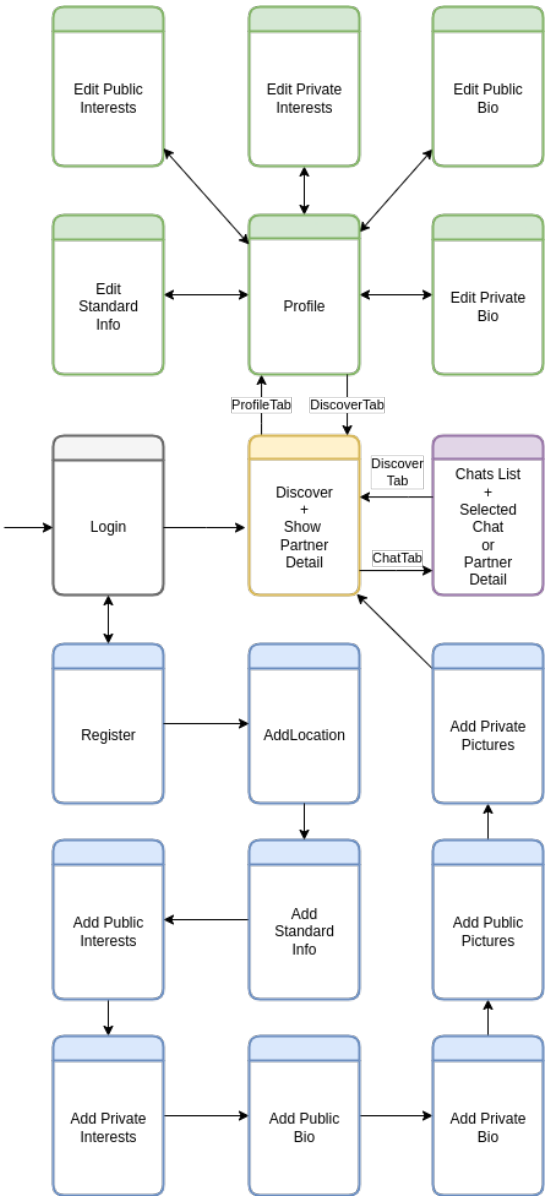


Figure 6: Tablet UX diagram

3.2 User interface

3.2.1 Authentication

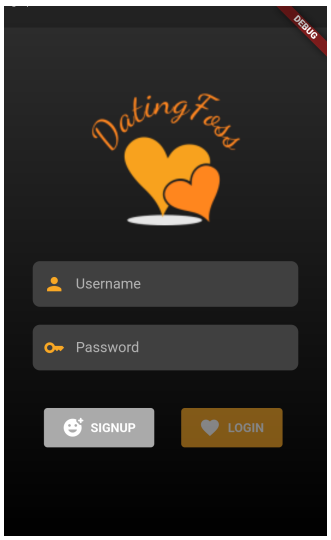


Figure 7: Login

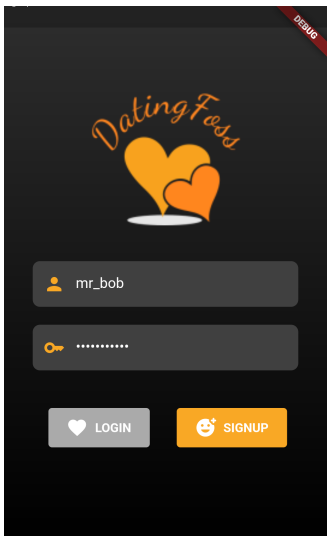


Figure 8: Signup

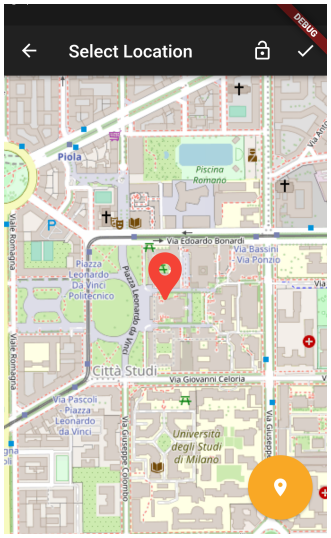


Figure 9: Location

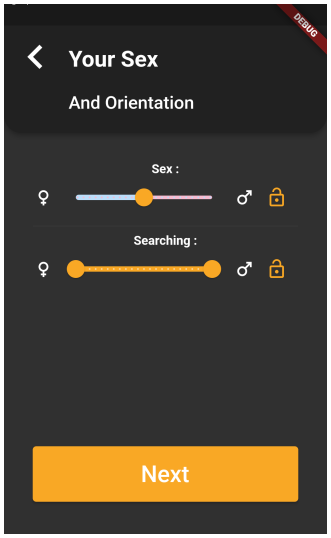


Figure 10: Sex and Orientation

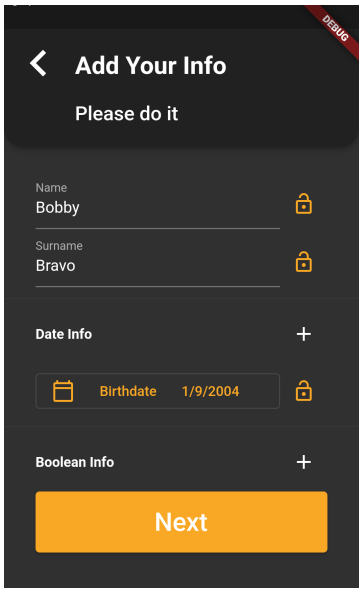


Figure 11: Standard Info Top

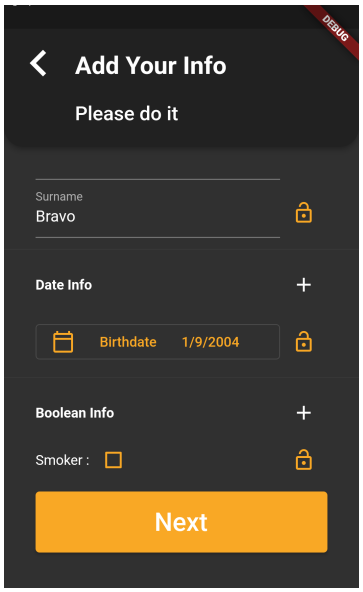


Figure 12: Standard Info Bottom

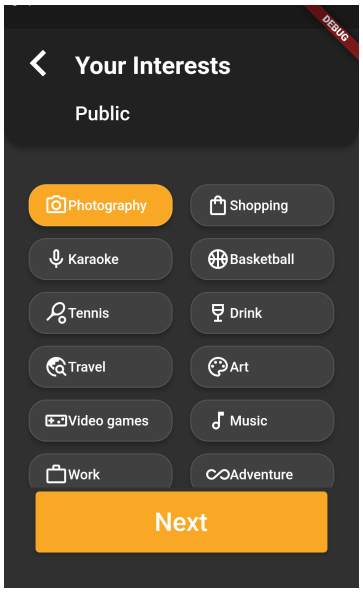


Figure 13: Public Interests

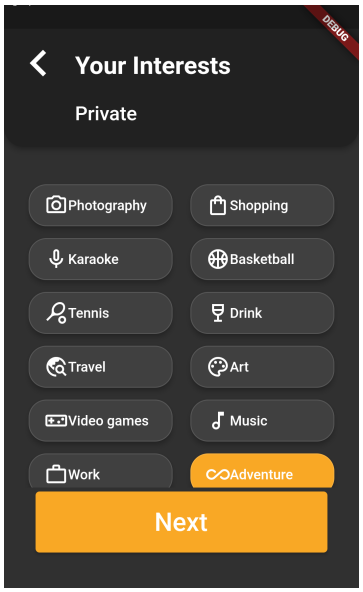


Figure 14: Private Interests

Figure 15: Public Bio

Figure 16: Private Bio

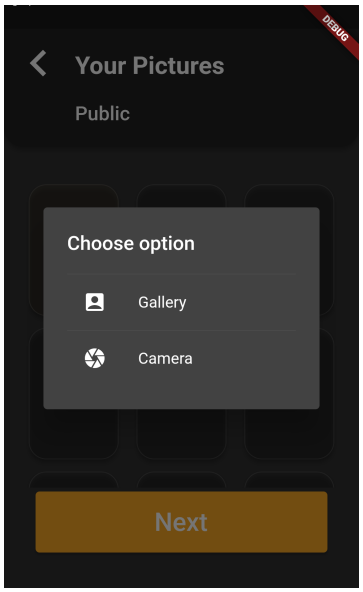


Figure 17: Public Picture Dialog

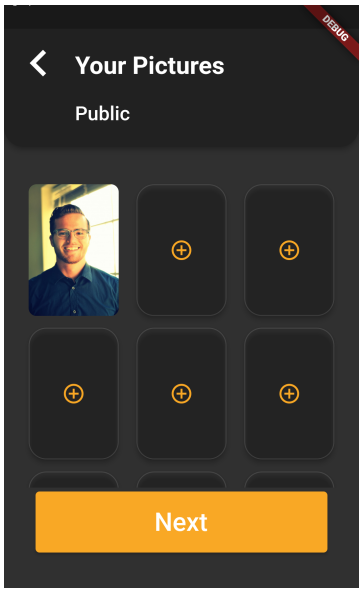


Figure 18: Public Picture

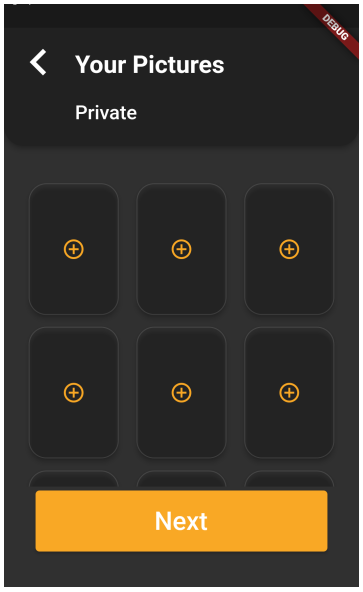


Figure 19: Private Picture Empty

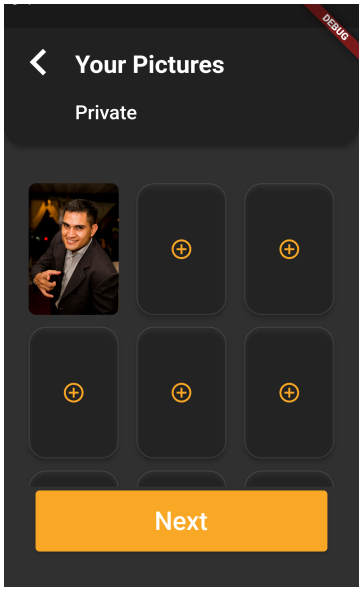


Figure 20: Private Picture

3.2.2 Discover

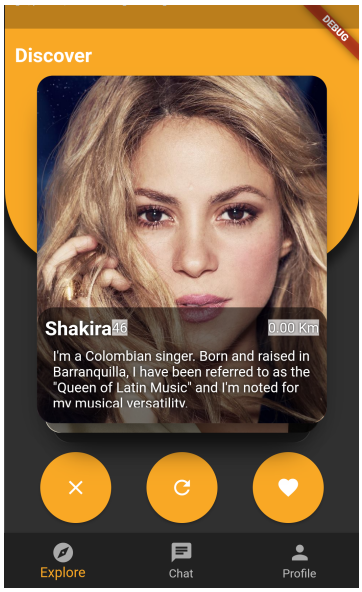


Figure 21: Discover

Figure 22: Discover Animation

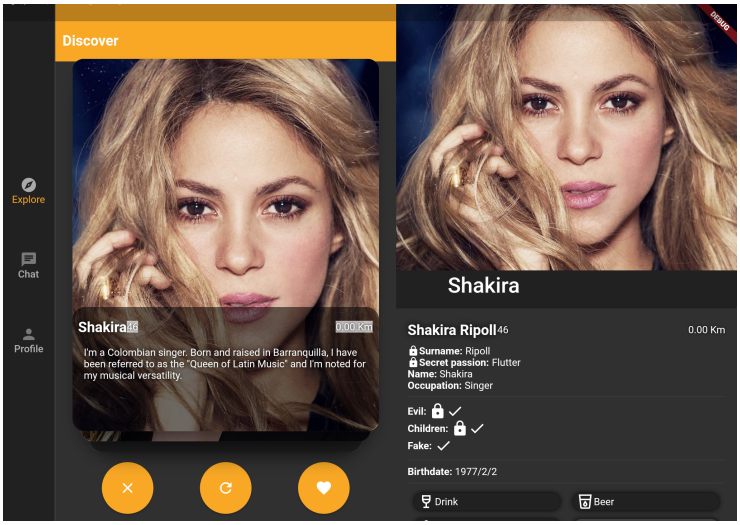


Figure 23: Tablet Discover

3.2.3 Partner Detail

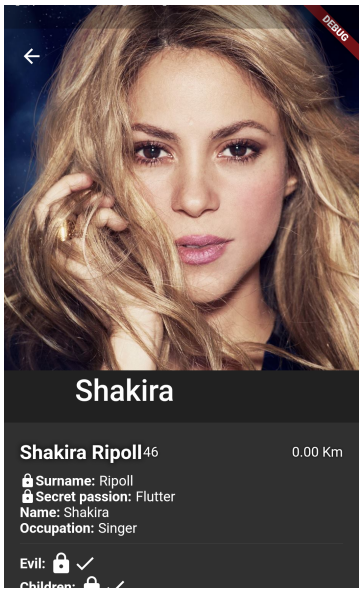


Figure 24: Partner Detail

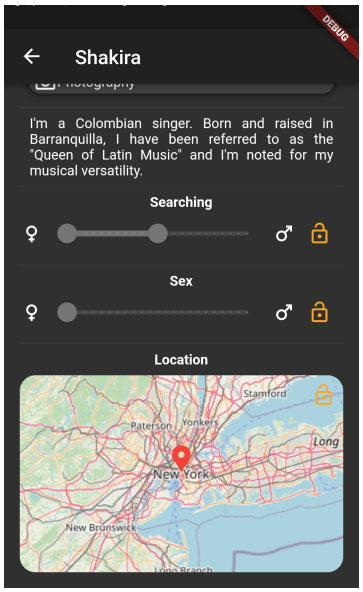


Figure 25: Partner Detail Bottom

3.2.4 Chat

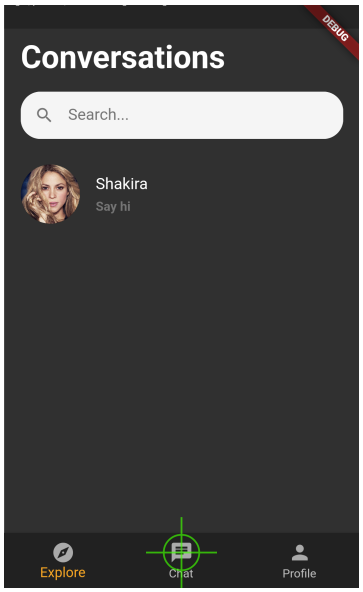


Figure 26: Chat List

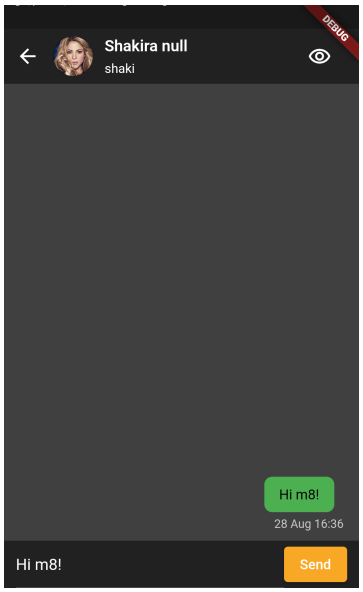


Figure 27: Single Chat

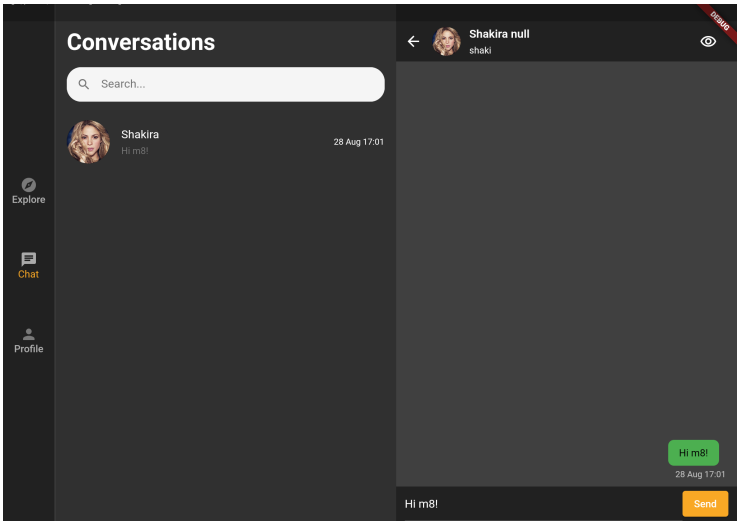


Figure 28: Tablet Chat

3.2.5 Profile

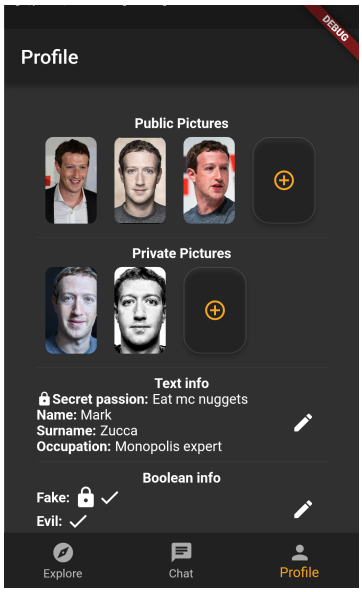


Figure 29: Profile

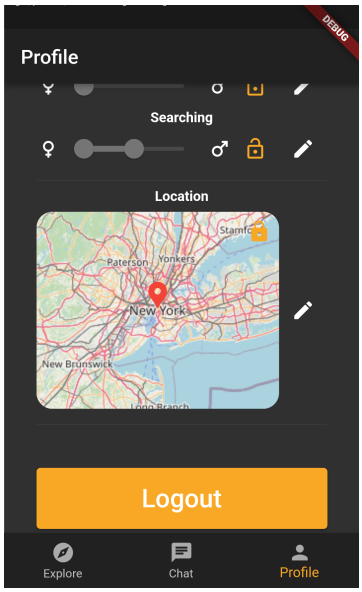


Figure 30: Profile Bottom

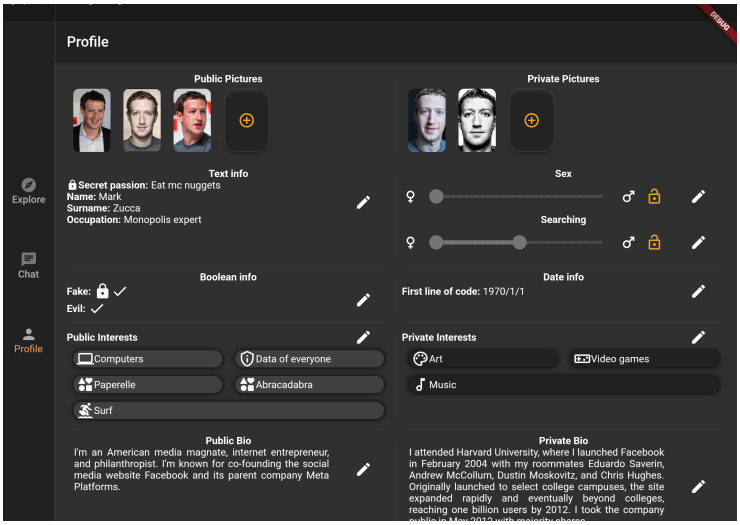


Figure 31: Tablet Profile

4 Implementation, integration and test plan

4.1 Integration Process

To develop the app we exploited **git** with **Gitlab** that gave us some really useful functionalies.
First of we organized the repository in branches:

- **main**
  Where only the releases will be merged.
- **development**
  Where there is the actual state of the developed app.
- **feature-branches**
  Any time we want to add a feature, we create a branch from development and merge in development when the feature is ready.

This organization make possible to use the CI of gitlab, we configured it to run all the tests and make a report every time we open a Merge Request. The gitlab UI doesn't let you merge untill the tests are all passed avoiding the merge of code that may contain bug.

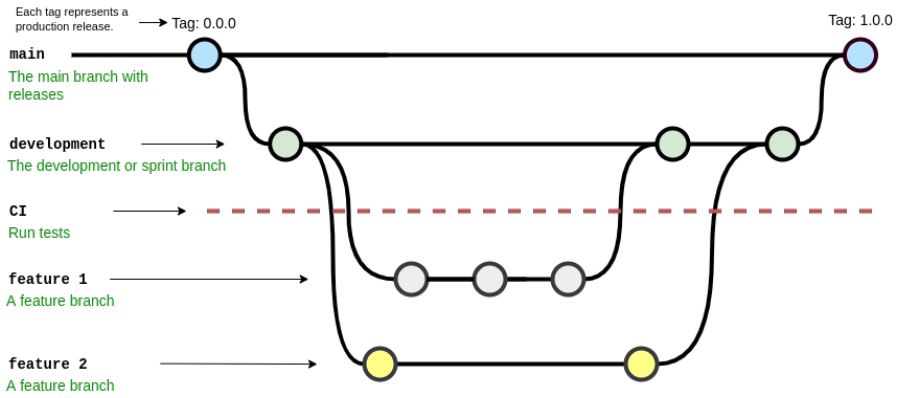


Figure 32: Git CI

We also exploited **Lefthook** a git hooks manager to:

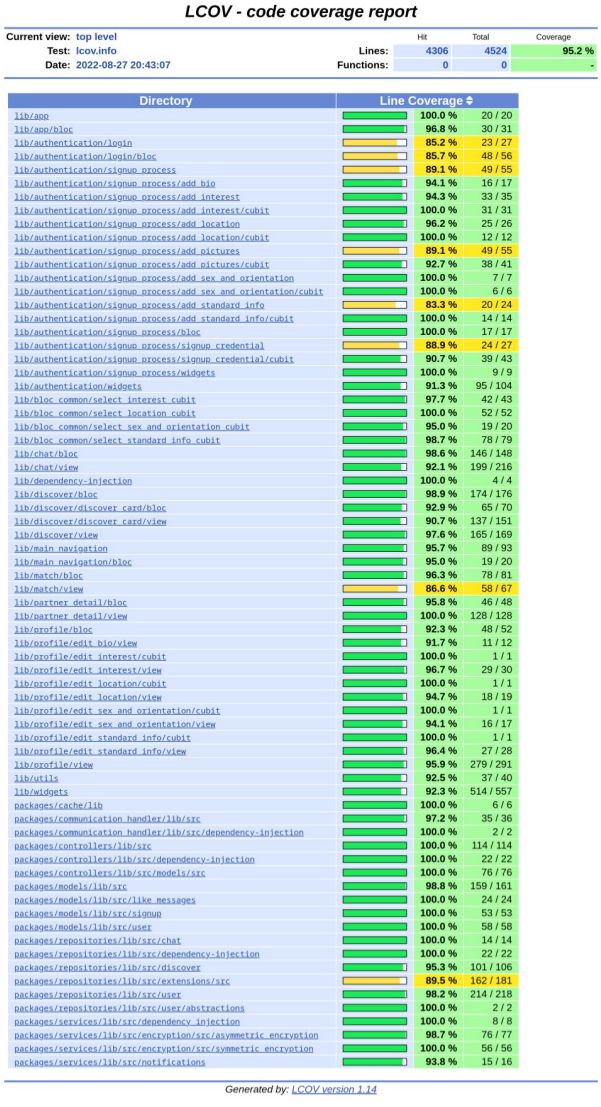
- Allow only commit that respect the Conventional Commits standard.
- Run ‘dart fix‘ to fix all the autamatically fixable lint problem and ‘flutter format‘ that format the code to follow the code style choosen.
- When try to push run the tests and the analyzer and block the push if one fail.

To help us write cleaner code we imported Very Good Analysis that provides with a lot of lint rules.

4.2 Testing

4.2.1 Unit testing

We have tested every component of the app with Unit Testing reaching a coverage of around **95%**.
We have around 400 tests and widget tests.



LCOV - code coverage report

		Hit	Total	Coverage
Current view: top level	Lines:	4306	4524	95.2 %
Test: lcov.info	Functions:	0	0	-
Date: 2022-08-27 20:43:07				

Directory	Line Coverage	
lib/app	100.0 %	20 / 20
lib/app/bloc	96.8 %	30 / 31
lib/authentication/login	85.2 %	23 / 27
lib/authentication/login/bloc	85.7 %	48 / 56
lib/authentication/signup_process	89.1 %	49 / 55
lib/authentication/signup_process/add_bio	94.1 %	16 / 17
lib/authentication/signup_process/add_interest	94.3 %	33 / 35
lib/authentication/signup_process/add_interest/cubit	100.0 %	31 / 31
lib/authentication/signup_process/add_location	96.2 %	25 / 26
lib/authentication/signup_process/add_location/cubit	100.0 %	12 / 12
lib/authentication/signup_process/add_pictures	89.1 %	49 / 55
lib/authentication/signup_process/add_pictures/cubit	92.7 %	38 / 41
lib/authentication/signup_process/add_sex_and_orientation	100.0 %	7 / 7
lib/authentication/signup_process/add_sex_and_orientation/cubit	100.0 %	6 / 6
lib/authentication/signup_process/add_standard_info	83.3 %	20 / 24
lib/authentication/signup_process/add_standard_info/cubit	100.0 %	14 / 14
lib/authentication/signup_process/bloc	100.0 %	17 / 17
lib/authentication/signup_process/signup_credential	88.9 %	24 / 27
lib/authentication/signup_process/signup_credential/cubit	90.7 %	39 / 43
lib/authentication/signup_process/widgets	100.0 %	9 / 9
lib/authentication/widgets	91.3 %	95 / 104
lib/bloc_common/select_interest_cubit	97.7 %	42 / 43
lib/bloc_common/select_location_cubit	100.0 %	52 / 52
lib/bloc_common/select_sex_and_orientation_cubit	95.0 %	19 / 20
lib/bloc_common/select_standard_info_cubit	98.7 %	78 / 79
lib/chat/bloc	98.6 %	146 / 148
lib/chat/view	92.1 %	199 / 216
lib/dependency-injection	100.0 %	4 / 4
lib/discover/bloc	98.9 %	174 / 176
lib/discover/discover_card/bloc	92.9 %	65 / 70
lib/discover/discover_card/view	90.7 %	137 / 151
lib/discover/view	97.6 %	165 / 169
lib/main_navigation	95.7 %	89 / 93
lib/main_navigation/bloc	95.0 %	19 / 20
lib/match/bloc	96.3 %	78 / 81
lib/match/view	86.6 %	58 / 67
lib/partner_detail/bloc	95.8 %	46 / 48
lib/partner_detail/view	100.0 %	128 / 128
lib/profile/bloc	92.3 %	48 / 52
lib/profile/edit_bio/view	91.7 %	11 / 12
lib/profile/edit_interest/cubit	100.0 %	1 / 1
lib/profile/edit_interest/view	96.7 %	29 / 30
lib/profile/edit_location/cubit	100.0 %	1 / 1
lib/profile/edit_location/view	94.7 %	18 / 19
lib/profile/edit_sex_and_orientation/cubit	100.0 %	1 / 1
lib/profile/edit_sex_and_orientation/view	94.1 %	16 / 17
lib/profile/edit_standard_info/cubit	100.0 %	1 / 1
lib/profile/edit_standard_info/view	96.4 %	27 / 28
lib/profile/view	95.9 %	279 / 291
lib/utils	92.5 %	37 / 40
lib/widgets	92.3 %	514 / 557
packages/cache/lib	100.0 %	6 / 6
packages/communication_handler/lib/src	97.2 %	35 / 36
packages/communication_handler/lib/src/dependency-injection	100.0 %	2 / 2
packages/controllers/lib/src	100.0 %	114 / 114
packages/controllers/lib/src/dependency-injection	100.0 %	22 / 22
packages/controllers/lib/src/models/src	100.0 %	76 / 76
packages/models/lib/src	98.8 %	159 / 161
packages/models/lib/src/like_messages	100.0 %	24 / 24
packages/models/lib/src/signup	100.0 %	53 / 53
packages/models/lib/src/user	100.0 %	58 / 58
packages/repositories/lib/src/chat	100.0 %	14 / 14
packages/repositories/lib/src/dependency-injection	100.0 %	22 / 22
packages/repositories/lib/src/discover	95.3 %	101 / 106
packages/repositories/lib/src/extensions/src	89.5 %	162 / 181
packages/repositories/lib/src/user	98.2 %	214 / 218
packages/repositories/lib/src/user/abstractions	100.0 %	2 / 2
packages/services/lib/src/dependency_injection	100.0 %	8 / 8
packages/services/lib/src/encryption/src/asymmetric_encryption	98.7 %	76 / 77
packages/services/lib/src/encryption/src/symmetric_encryption	100.0 %	56 / 56
packages/services/lib/src/notifications	93.8 %	15 / 16

Generated by: LCOV version 1.14

Figure 33: Coverage

To make possible to run the tests we explited some components:

- An adaptation this script that made possible to run all the tests from the packages and lib folder and than merge the lcov.info files generate from the different packages in a single one.
- `genhtml` to generate the html report from the lcov.info file
- `cobertura` to convert the lcov.info report to cobertura format readable by gitlab
- `Mocktail` to make it easy to mock the components not under test

4.2.2 Integration testing

We also implemented a couple of integretion test that help to identify regression bug when a new feature is introduced.

- **Signup**
  that make a complete signup process navigating through all the nine screen of the signup flow submitting fake data and landing in the Discover page. It also take screenshots to every screen that it pass through for later analysis.
- **Login, put like, send message and logout**
  Execute those steps:
  - Do the login landing in the Discover page.
  - Put a like to the first partner visible getting a match animation.
  - Navigate to the Chat screen.
  - Open the conversation with the matched partner.
  - Send a message.
  - Open the PartnerDetail page and scroll untill it reach the bottom.
  - Close the PartnerDetail page and conversation.
  - Navigate to the Profile page and scroll down.
  - Make a logout landing in the Login page.

4.2.3 Other types of testing

Other than that we spent some time to manually test the app. Trying to find edge cases where it could broke.

5 References

- Flutter
- Mocktail

5.1 Software used for this document

- Draw.io: Diagrams
- LaTeX: Document preparation system
- VSCodium: Text Editor with user-friendly live share plugin