RyFy Design Document

Gabriele Aldeghi - 899733

17 February 2020

Contents

1 Introduction

1.1 Scope

Nowadays, many companies offers the possibility to their employees to suggest candidates for working positions inside the company. If a candidate is successfully hired by the company, the person who recommended it obtains a money reward. RyFy is a smartphone application that focuses on the connection between two categories of people. The first category, called Mentors, are people currently hired inside a company and would like to propose qualified candidates. The second category is called the Mentees, and are people how are searching for a job. RyFy aims to:

- Let mentees find and contact mentors through the platform.
- It allows mentors to define questions for whichever mentee that wants to connect with the selected mentor. This allows mentors to test the capacities of the mentee, and therefore narrow discard any mentee they don't consider qualified enough, through their answers.
- Once the connection has been established, it allows a Mentor and a Mentee to chat in real time.

The platform can be accessed through a standard registration procedure, or by signing in with a Google account. Once logged in, the user will receive push notifications, such as a new received message.

1.2 Overview

- In section 1,

2 System Architecture

2.1 Application architecture

RyFy is an application developed with Flutter, which allows the application to be both available for Android and iOS smartphone platforms. The application connects to an application server, that expose the needed API for retrieving, saving and managing the data that will be shown in the application. The application's architecture is layered as it follows:

- Data Layer: The application possess a data layer, where data are persisted by using a SQLite database. The data stored are relative to the current user information, and their active chats.
- Logic layer: The logic layer is the one responsible for managing all the interactions between the background server, the external services (Such as the registration and authentication of the user through a Google account), and the user interaction of the app. The state management of the app has been implemented with the Business Logic Component design pattern, in short BLoC. The decision for such an approach for the state management of the app stems from the need to listen for asynchronous updates. In fact, the app relies on a socket connection for allowing users to chat. Moreover, push notifications may be triggered whenever a mentee contacts a mentor, or when a mentor accepts or refuse a contact request. All these events are intrinsically asynchronous, and the BLoC pattern perfectly fits in this scenario, allowing us to update the UI accordingly on the events received by the logic layer. In order to offer the BLoC elements throughout the application, the Provider package has been used.
- User Interface Layer: The user interface has been customized for both mentors and mentees. The main differences are presented in the home page of the application. In fact, while a mentee can search and send contact requests to a Mentor, the mentor on the other hand is presented with the pending requests of the mentees.

To implement the socket connection, the SocketIO library has been used. A set of custom events has been defined between the application and the server in order to communicate. On the other hand, the push notifications are implemented with the use of the Firebase Cloud Messaging.

The application uses:

- Camera: The photocamera on the smarthphone is used to add new images, such as custom user profile images.
- Microphone: The application allows mentees to send audio messages for answering the mentor's questions. These messages can be listen to by the mentors during the approval phase of the mentee.

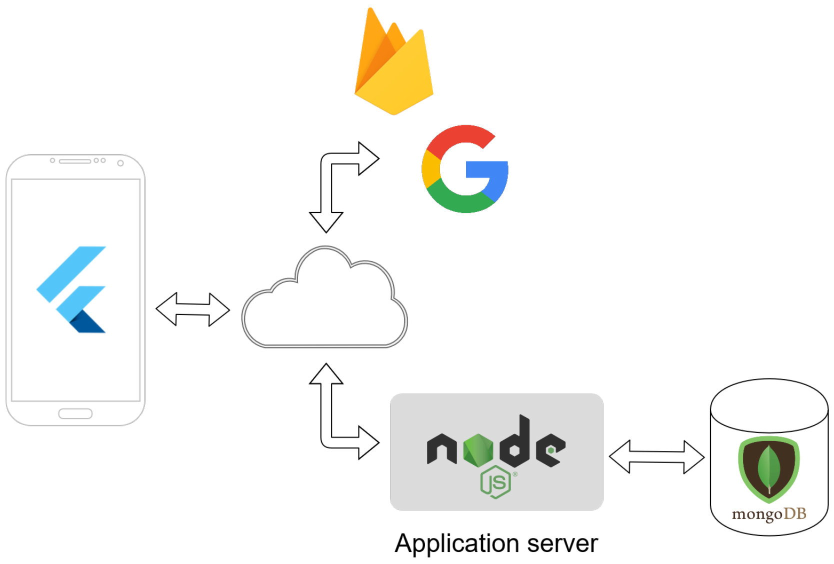


Figure 1: Overview of the RyFy system.

2.2 System overview

The system relies on the internet to connect the application, the server and the external services. The server is implemented in NodeJS, and stores the application data on a MongoDB database. In order to use both REST API calls and a socket connection, the server allows both connections via HTTPS. The external services used are:

- Google Services: As the application allows to log in with a Google account, an apposite procedure must be followed in order to correctly authenticate the user. Firstly, the user is asked to login with a certain account. After a Google account has been selected, the Google's API will return a token, associated with the selected account and the RyFy application. This token will be sent to the application server, where it will be validated again with the Google's API in order to verify its truthfulness.
- Firebase Cloud Messaging: This service allows to send push notifications to applications who correctly register to the service. Every device has its own token, that univocally identify it.

2.3 Implementation details - Push notifications

In order to be useful, push notifications must be shown on the user's device even if the application is in background or closed. In order to offer a more customized notification experience, a background thread is used. This thread will listen to every push notification, check with the internal database whether

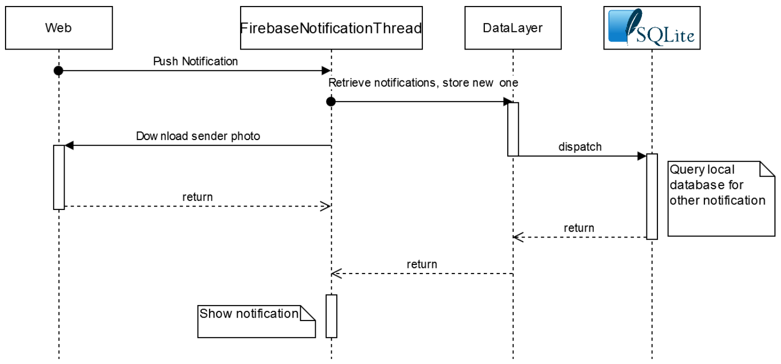

Web
FirebaseNotificationThread
DataLayer
SQLite
Push Notification
Retrieve notifications, store new one
Dow nload sender photo
dispatch
Query local database for other notification
return
return
return
Show notification

Figure 2: Sequence diagram for a push notification when the app is closed.

there are other unread notifications, style the notification and then show it to the user. Due to the limited amount of computation that can be done on these threads, the code has been optimized to perform as fast as possible.

2.4 Implementation details - Audio Files

As it will be shown here, the application interacts with the microphone of the smartphone in order to send audio messages. These audio messages have been implemented by using native code for both Android and iOS, where this native code would be invoked by the dart code inside the logic layer of the application. Whenever an audio message is registered, a custom waveform of the audio is shown. This waveform represents the whole audio message, where each bar corresponds to the average decibel volume in that instant of time. When the message is compressed and sent to the server, the registered decibel peaks are not sent to the server. Instead, implementation of FFmpeg on Flutter is used to analyze the received audio file, extract the average decibel peaks and compute the waveform.

3 Data Design

In figure 1 we can see how the entities of our systems have been mapped to an ER schema. The main entities we want to store in our database are the User (which can be either a Mentor or Mentee), and any active request between the two. A mentor can decide to ask the mentee several questions before accepting the request. A request between a Mentor and Mentee can be ether:

- Pending: The mentee has sent the request to the mentor.
- Accepted: The mentor has reviewed the request of the mentee, and decided to connect. This allows the two of them to chat together.
- Refused: The mentor has decided to refuse the connect request. The mentee cannot send a contact request to the mentor anymore.

The messages that Mentor and Mentee will exchange are linked to their Accepted ContactRequest.

The local database inside the applications is used to keep an offline copy of messages. Therefore, even without an internet connection, the user will be able to read the messages exchanged. Therefore, a reduced version of the database is maintained on the local device, where only ContactRequests and its user information are stored. These information can be actively updated by the application whenever an application makes a REST HTTPS request, or passively through the use of a socket or push notification events.

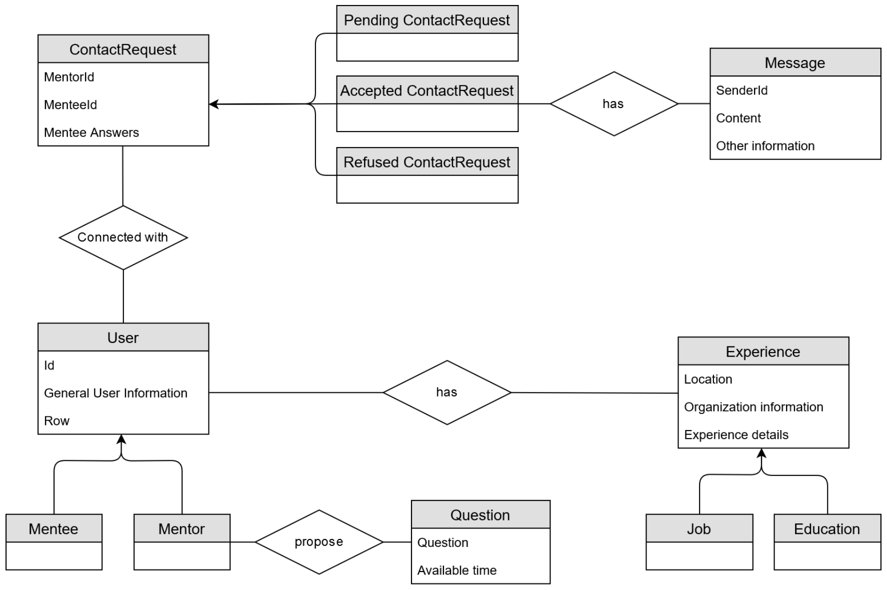


Figure 3: Simplified view of the database.

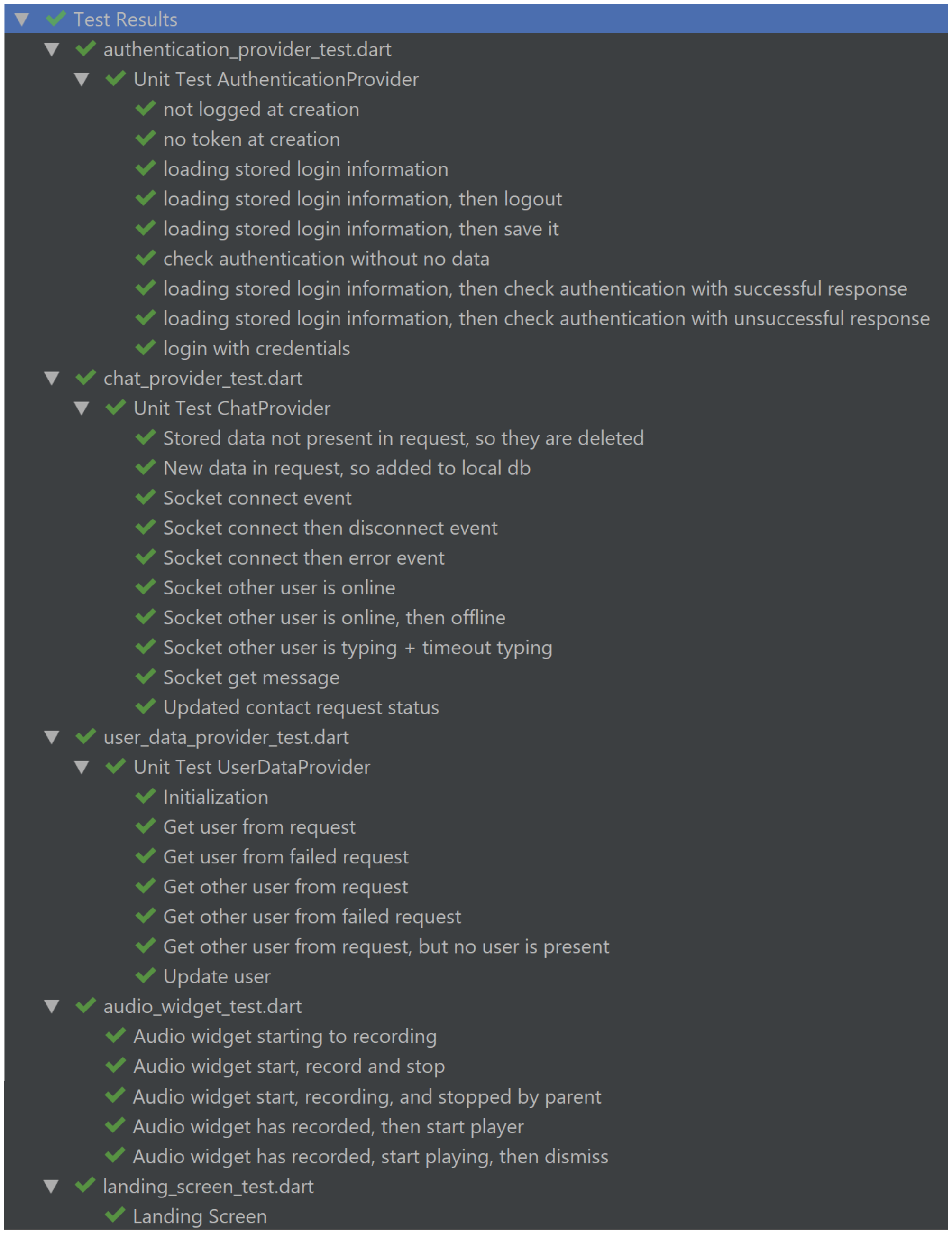
4 User Interface

5 Testing

In order to test the application, I've decided to use two approaches.

- Unit Test: The great majority of the logic layer of the app has been tested through the use of Unit Tests. In order to simulate any external call, such as REST requests or socket events, the Mockito library has been used to stub such events with actual responses from the server. There is no direct testing of the data layer, but instead its functionalities are checked during the interaction that the application layer has with it.
- Widget Test: Flutter creates the User Interface by composing Widget elements. Therefore, it allows developers to test Widgets independently, in order to test their correct functionality. Widget Testing has been used on the most complex widgets present in the app, such as the widget that allows a user to record and listen to vocal audios.

In 4, we can see all the tests executed on the application. Furthermore, extensive integration tests have been executed on different Android devices. Unfortunately, given the limitations of how Flutter is allowed to build and compile iOS applications, I could not test the application on an iOS environment. Therefore, I cannot guarantee that the iOS application would work flawlessly.



```
▼ ✓ Test Results
  ▼ ✓ authentication_provider_test.dart
    ▼ ✓ Unit Test AuthenticationProvider
        ✓ not logged at creation
        ✓ no token at creation
        ✓ loading stored login information
        ✓ loading stored login information, then logout
        ✓ loading stored login information, then save it
        ✓ check authentication without no data
        ✓ loading stored login information, then check authentication with successful response
        ✓ loading stored login information, then check authentication with unsuccessful response
        ✓ login with credentials
  ▼ ✓ chat_provider_test.dart
    ▼ ✓ Unit Test ChatProvider
        ✓ Stored data not present in request, so they are deleted
        ✓ New data in request, so added to local db
        ✓ Socket connect event
        ✓ Socket connect then disconnect event
        ✓ Socket connect then error event
        ✓ Socket other user is online
        ✓ Socket other user is online, then offline
        ✓ Socket other user is typing + timeout typing
        ✓ Socket get message
        ✓ Updated contact request status
  ▼ ✓ user_data_provider_test.dart
    ▼ ✓ Unit Test UserDataProvider
        ✓ Initialization
        ✓ Get user from request
        ✓ Get user from failed request
        ✓ Get other user from request
        ✓ Get other user from failed request
        ✓ Get other user from request, but no user is present
        ✓ Update user
  ▼ ✓ audio_widget_test.dart
      ✓ Audio widget starting to recording
      ✓ Audio widget start, record and stop
      ✓ Audio widget start, recording, and stopped by parent
      ✓ Audio widget has recorded, then start player
      ✓ Audio widget has recorded, start playing, then dismiss
  ▼ ✓ landing_screen_test.dart
      ✓ Landing Screen
```

Figure 4: List of all test executed.