



Norwegian University of Science and Technology (NTNU)

ThirreMjeshtin

IMT3672 MOBILE DEVELOPMENT PRACTICE

KOSOVARS

PROJECT WORK
NOVEMBER - DECEMBER 2016
MARIUSZ NOWOSTAWSKI

<https://github.com/florim14/ThirreMjeshtin.git>
Weekly tasks: https://docs.google.com/spreadsheets/d/1Paz-UezNhM6qfkp4MoK_c17lucJSGpycpnv15FcFQSk/edit?usp=sharing
Much of the code taken as a reference from: <https://developers.android.com>

Contents

1	Introduction	1
1.1	Brief application description	1
2	Application core	2
2.1	Application architecture	2
2.1.1	Tools used	2
2.1.2	General Client-server discussion	3
2.2	Basic activity layouts and workflow	4
3	Conclusion, Issues, Difficulties, and what comes next?	18

1 Introduction

1.1 Brief application description

Our idea was based on a personal experience where the oven stopped working one day and we had to get through the struggle of googling for possible reasons why the oven didn't work anymore, finding and calling a repairman in the area of Gjøvik through Norwegian websites and then to finally setting an appointment with said repairman. After which event we thought that all of this information and work could be done in one place to make our and the repairman's life easier.

ThirreMjeshtrin (translated from Albanian *Call a repairman*) is a complex Android and Server Side application which does exactly what it says. It allows the user to both register as a user and receive services from other users who are registered as repairman in their area, and to register as a repairman and receive request on behalf of other users who require their services in their own set working radius. The application also makes it possible to instantly call a repairman, ask a repairman for advice through the application online chat, discover repairman in their surrounding, rate those repairman and give them feedback after having received their work.

2 Application core

2.1 Application architecture

2.1.1 Tools used

Well, because it is Android, it is more than understandable that we've user **Android Studio** all the time, and of course some other technologies to implement the client-server communication architecture.

On the server side, we did use **php** with **MySQL**, for processing the requests and getting responses from the server. The server database is in MySQL, and the main reason behind this choice is that, we all had a course on Web Application Programming at our home university, which was mainly about **php** and **MySQL** working together for bulding web apps.

We've also used **Google Firebase** real-time database for implementing the chat aspect of our project, and we did this mainly because, Firebase itself is an ideal platform for building and integrating chats in Mobile apps.



(a) *Android Studio*



(b) *php*



(c) *MySQL*



(d) *Google Firebase*

Figure 1: Technologies used

2.1.2 General Client-server discussion

Communication with the server is essential to the work of the app, considering that users can request immediate service from active repairmen.

The server hosts the *mySQL* database needed for data storage (user list, repairmen information, request registry, feedback history) and alerts single users when a change has happened on the database which they should be notified on. This service is realised by the use of downstream messaging (Server to Users) of Firebase Cloud Messaging. When a change happens in the database through php, the server generates http requests to the aforementioned Firebase service to send a message (push notification) to a user identified by a token. Tokens are unique for devices, therefore a user cannot be authenticated on a device while still being logged in on another. This is easily



Figure 2: Client-server architecture in mobile devices

checked by the value of the Token for each user, since when the user is not active in any device, the Token is null. These push notifications sent by Firebase Cloud Messaging are received by the `MyFirebaseMEssagingService` on the device, from where notifications are built based on the contents of the message, and needed changes are done in the app. Besides handling authentication, the server also notifies in the event when a service is required for a certain repairman, a service request has been accepted or refused, or when it has timed out.

The app contacts the server to realise events of:

- User registration and authentication
- Querying for repairmen of a chosen category in the area
- Sending requests to repairmen and handling the state of those requests
- Keeping track of the chat rooms that have been opened and chat logs (handled by the remote realtime Firebase database)
- Saving Feedback history on accepted service requests.

Contact to the server is realised by the `ConnectToServer` class, which contains:

- Variables of all the needed URLs for communication

- `sendRequest` method that takes a URL, parameters and a boolean indicator of whether the Network Connection can be asynchronously executed (in the background) or a result of this communication is needed.
- `results` List of HashMaps of Strings which contains the parsed response from the server.

The server always sends formatted responses in JSON, which are parsed and the extracted information is put in the `results` variable of the `ConnectToServer` object. If the connection is required to be synchronous, the UI thread waits until a response is returned as a result, otherwise the network communication is only initiated in the UI thread, while the actual execution happens in the background. The network communication is done in a `NetworkTask` object. `NetworkTask` is a subclass of `AsyncTask` where the response is also parsed from JSON.

Before a request is made to the server, the connectivity state of the device is checked, so that the user is notified if a network communication cannot be realised.

2.2 Basic activity layouts and workflow

Our application consist of several well connected activities, each of them for a different purpose. When a user opens the app for the first time, the user will be sent at the *login* activity, where they are asked to set the login credentials, of course, if there exist a user account for them. If not, down below there is a clickable piece of text, and, if the user clicks on that, they would be sent at the Registration activity.

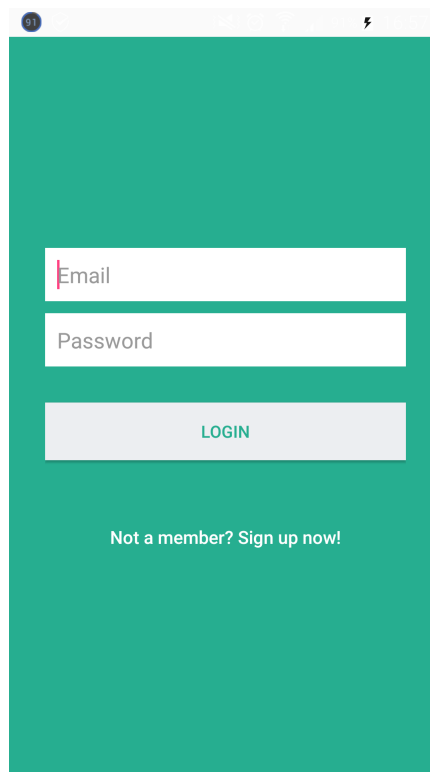


Figure 3: Login activity

If we suppose that the user enters our app for the first time, they'd need to be registered, and the *Registration* activity looks like this:

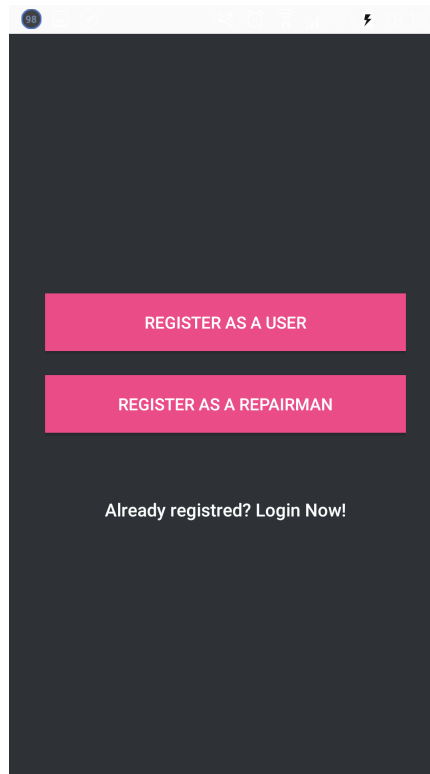


Figure 4: Register activity

Of course, if they have an account, and they sent themselves in this activity, there is a way to get back to the Login activity by clicking the *Already registered? Login now!*.

This activity consists of two buttons, and each of them is used depending on how you'd want to register in the app, as a **Repairman** (the Register as a Repairman button) or as a **User** (Register as a User button).

If a user wants to register in the app as a Repairman, they'd be asked to give their info as shown in the activity:

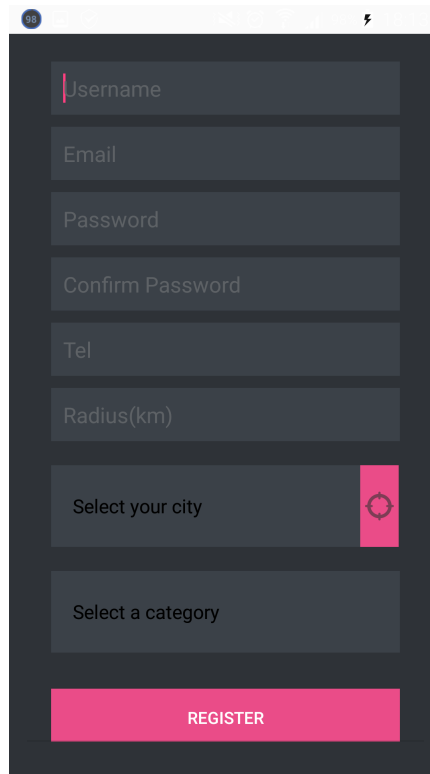
The image shows a mobile application interface for registering as a repairman. The form consists of several text input fields stacked vertically: 'Username', 'Email', 'Password', 'Confirm Password', 'Tel', and 'Radius(km)'. Below these is a 'Select your city' field with a red circular spinner icon to its right. Underneath that is a 'Select a category' field. At the bottom of the form is a prominent red button with the word 'REGISTER' in white capital letters. The entire form is set against a dark gray background.

Figure 5: Register as a Repairman

Each of information given in those boxes will be validated, and if not met the requirements, the user will see an error right at the box where the info could not be validated.

The **City** value represents the city where the user would operate from. At the moment, the alternatives on the spinner are only the cities in Norway, and a another alternative is to get their current location while registering.

The **Radius** value simply determines the circle area where they're able to operate.

The **Category** chosen, is related to what kind of Repairman is the user. There alternatives there are a Plumber, Electrician, and a Mechanic.

The registration activity for a simple user is much more simpler, requiring only the username, the email, and the password.

Here is what it looks like:

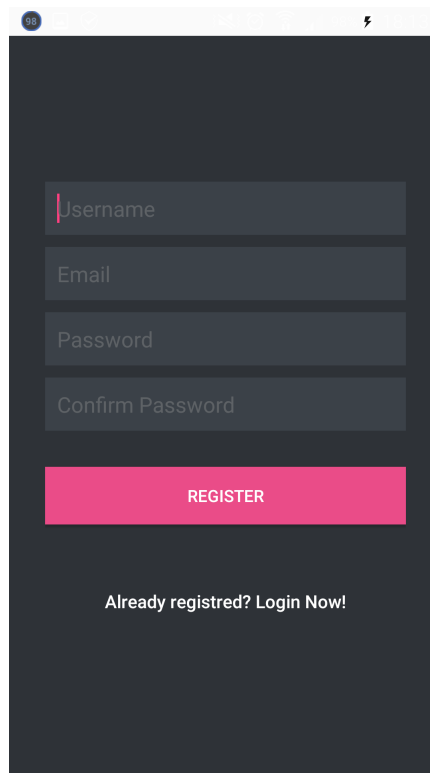
The image shows a mobile application interface for user registration. The background is a dark, solid color. At the top, there is a status bar with a blue circle icon on the left and a lightning bolt icon on the right. Below the status bar, there are four text input fields stacked vertically. The first field is labeled 'Username' and has a red cursor at the beginning. The second field is labeled 'Email'. The third field is labeled 'Password'. The fourth field is labeled 'Confirm Password'. Below these fields is a large, rectangular pink button with the word 'REGISTER' in white, uppercase letters. At the bottom of the screen, there is a line of text that reads 'Already registred? Login Now!'.

Figure 6: Register as a simple user

After clicking the Register button, the user will be created and they will be sent right at the Login activity to login themselves for the first time!

After logging in successfully, the user would be sent to an activity where they could choose for what kind of Repairman they're searching, and this by clicking on the clickable image views, where each of them represents a different category of Repairman.

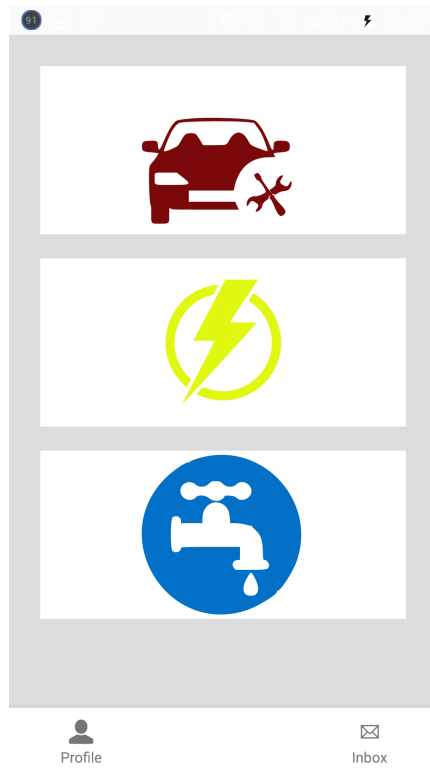


Figure 7: Search for a repairman

Down there is a bottom bar, helping the user to go to their profile and inbox.

Immediately after clicking on of the imageviews, a user will be sent at a map, telling their location and some other markers (with different color) representing the locations of the repairmans of that category with the operation area including the users location where the search comes from. Note that, if the repairman is not online, meaning, logged in the app, there won't be a marker telling their location, even if they are able to handle a service.

Those markers are clickable, so after clicking on them, the repairman profile would open.

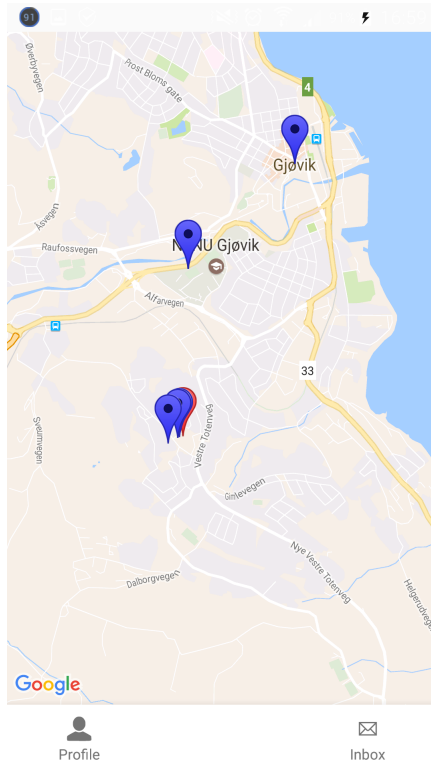


Figure 8: The map view

This is what a repairman profile would look like.

This layout is organized with tabs, so that the first tab presents the repairmans profile, the second tab is the Reviews tab, where you can find the rating from different users on that repairman, and the third tab is the Send Feedback tab, where you can send a feedback to the repairman, but only if there is at least one deal between the user and the repairman, in which deal the user can give feedback.

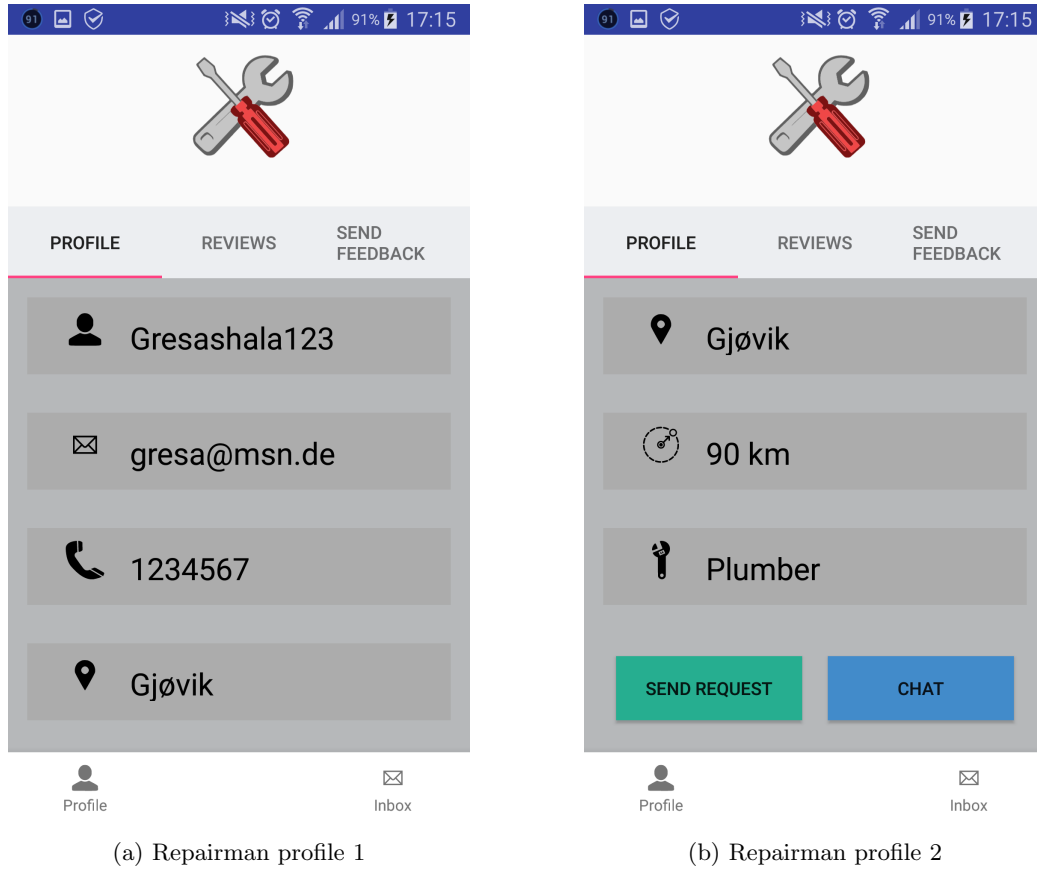


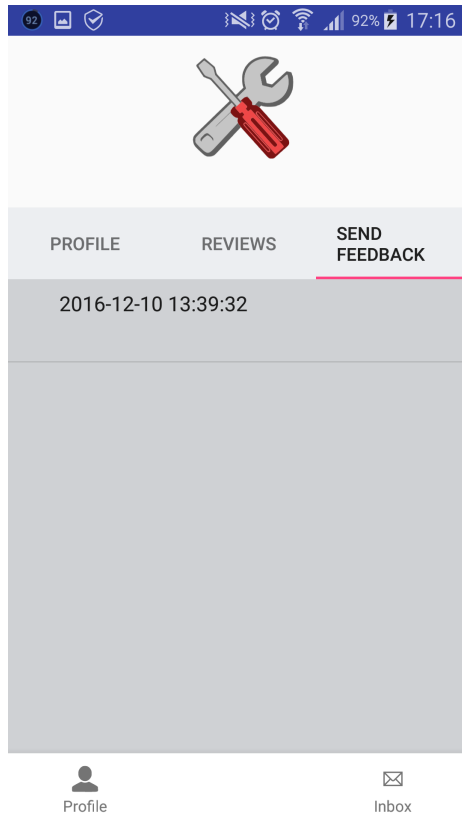
Figure 9: Repairman profile

If there are no accepted requests between the repairman and the user, the Send Feedback tab is showing no data. This is absolutely reasonable, since a simple user cannot rate a repairman without having an accepted request from the repairman.

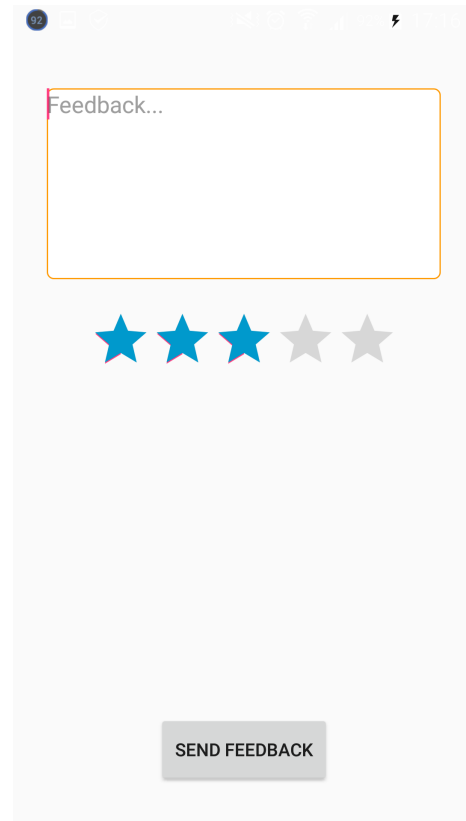
To get from one tab to another, you have to click the tab name, so, there is no swipe option for this.

If the user goes to the Send Feedback tab, there will be a list of the accepted requests between that user and the repairman, meaning that, the user is able to give feedback to the repairman when clicking on one of those list items, signed with the timestamp when the request was accepted.

After clicking on one of the items, the Feedback activity will open, where the user can write a feedback comment, a rating via a rating bar, and all these data after being processed and sent successfully will be shown in the Reviews tab of the Repairman profile and all this would look somewhat like:



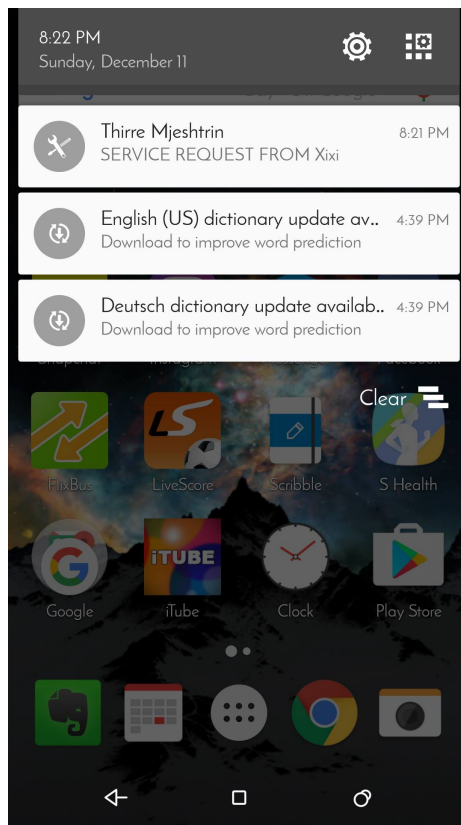
(a) Repairman feedback tab



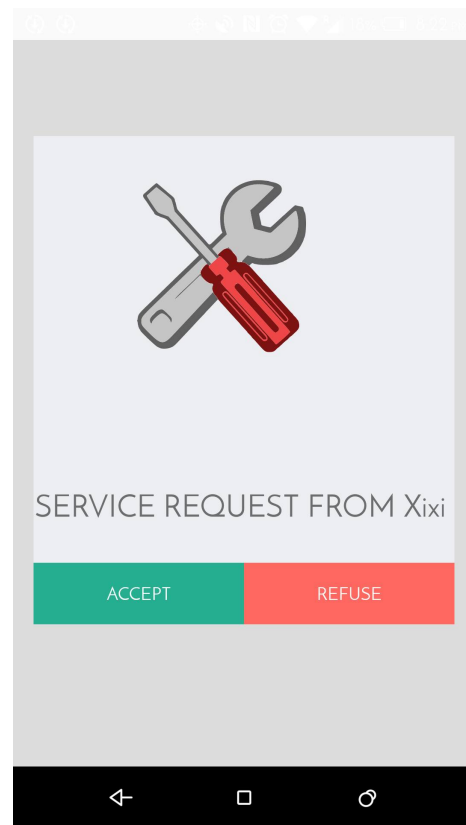
(b) Giving a feedback to a Repairman

Figure 10: Repairman feedback tab

A *Send requests* button is there if the user wants to send a request to the chosen repairman, and from there will be a push notification in the repairmans side app, notifying them that there is a request from a user for a service.



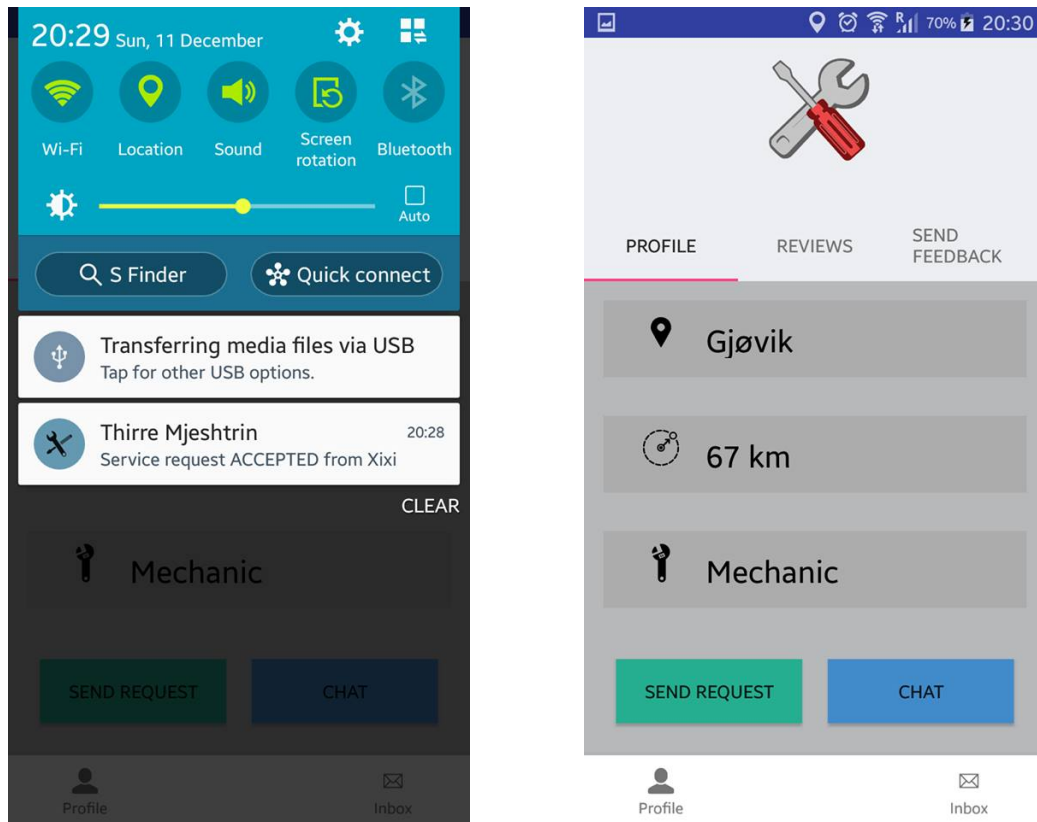
(a) Repairman getting a push notification



(b) Repairman request prompt

Figure 11: Repairman side while getting a request for a service

The repairman can accept or deny the request. If the repairman accepts the request, there will also be a push notification sent at the user from where the request came from, telling him that the repairman has accepted the request. Now, if the user clicks in the push notification they'll be sent at the repairmans profile, with the opportunity to chat with the repairman. Definitely, a user cannot chat with a repairman without having an accepted request from that repairman.



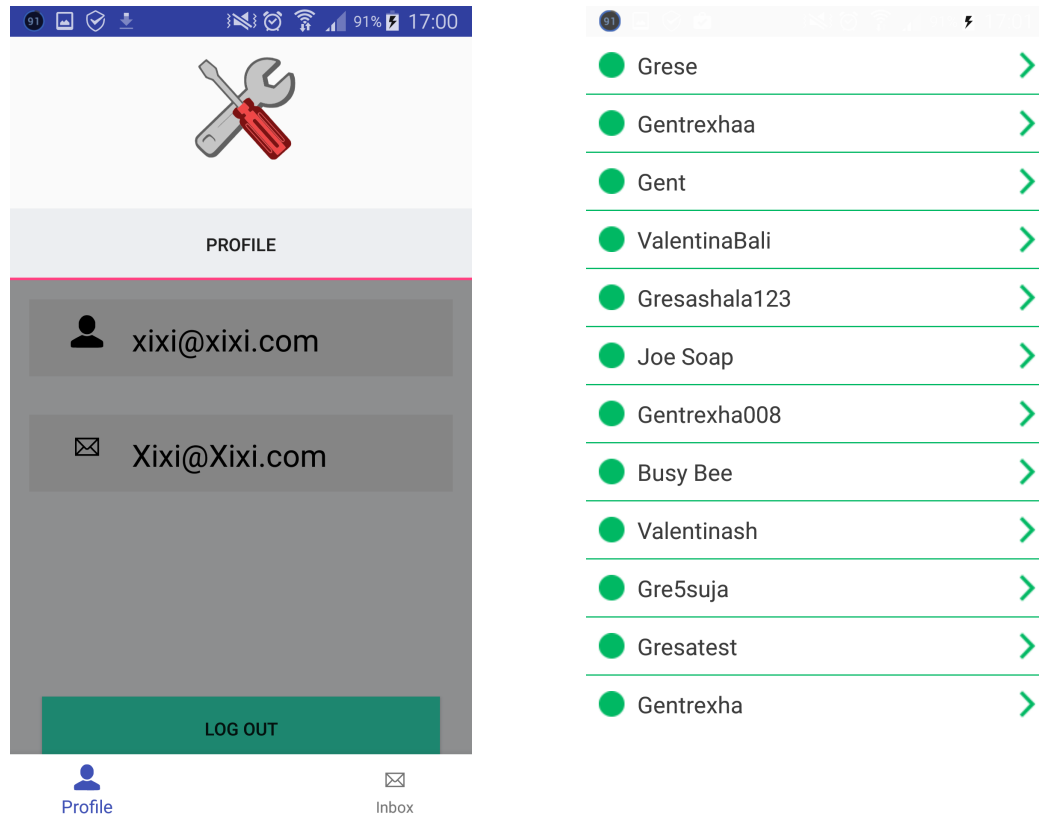
(a) Client getting a push notification

(b) The repairman profile with the chat option enabled

Figure 12: Client side while getting a positive request for a service

As we can see, from the client side, immediately after a positive request confirmation, the user can start chatting with the repairman.

Now back at the bottom bar, being shown in almost each activity, with the profile and the inbox options to select. As shown in the figure, the profile button in the bottom bar sends the user to their profile (so does to the repairman too), and the inbox button will just open the inbox of the user with the chatrooms (name of the repairman) that this particular user has opened with different repairmans. The same chatroom is used for all requests between a particular user and a repairman.



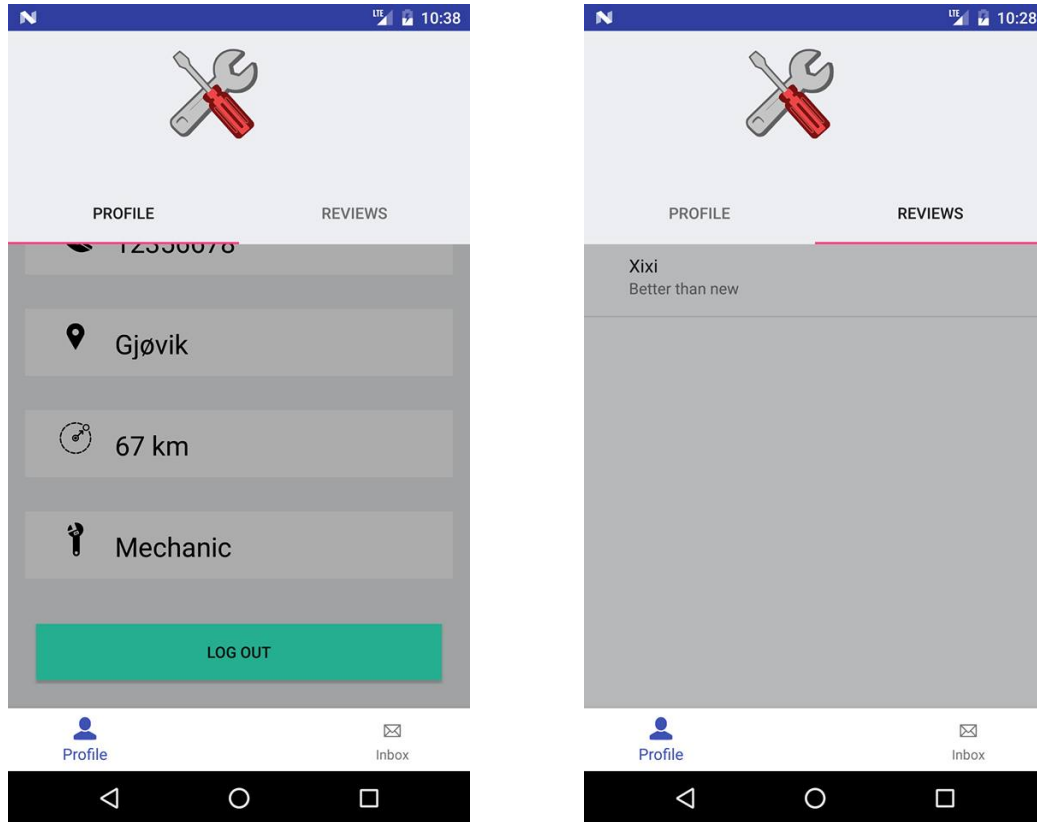
(a) Client profile view

(b) Users inbox

Figure 13: Bottom bar functionality from the Client side

From the repairmans aspect of view, things seem to be a bit different, since if a repairman clicks on the profile button in the bottom bar, their profile is being opened with the profile tab, and the review tab.

NOTE! The send feedback tab here will be missing because, as simple as it is, a repairman cannot send feedback to themselves.



(a) Repairman profile view

(b) Repairman review

Figure 14: Bottom bar functionality from the Repairman side

And the Inbox layout will just list all the roomchats available for that particular repairman, so, there are listed all the clients that created chatrooms and whose requests have been accepted by the repairman.



Figure 15: Repairman inbox

If the user or the repairman clicks on one of those chatrooms, the chat activity will be opened, linking that specific user with the repairman. Same thing is generated, if a user, after accepting a positive reply from a repairman, clicks on the chat button in the repairmans profile.

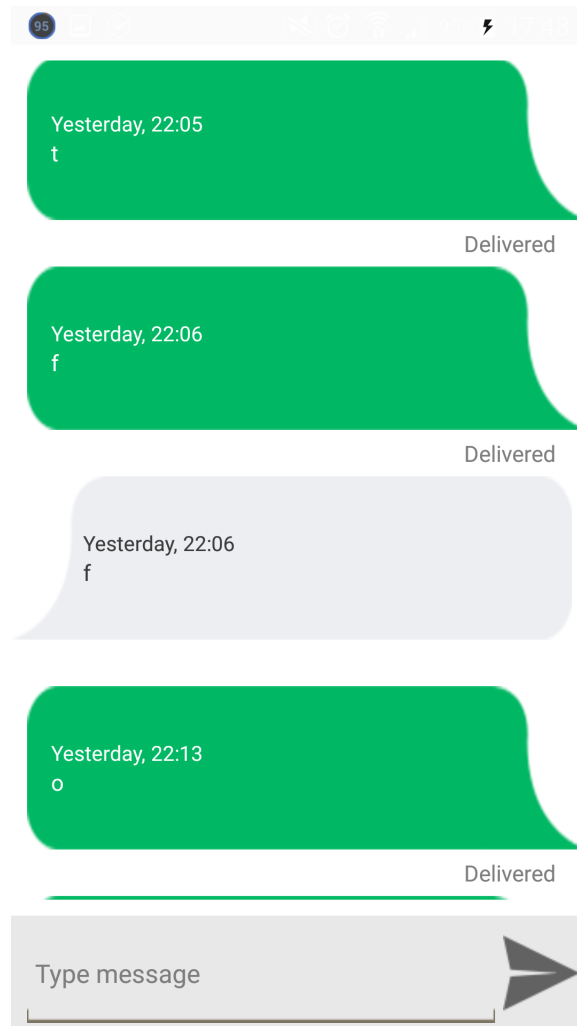


Figure 16: Inside chatroom

3 Conclusion, Issues, Difficulties, and what comes next?

After having worked for the first time in an Android project of this size we're very proud of our project and what we achieved with it. Coordinated group work has lead to the actual implementation being quite close to the initial idea for the app. The challenged presented by this project during our work have definitely increased our skills in Mobile Application Development.

Having to work for the first time with a real time database and implementing an application sided online chat, our approach for this solution was a little diffucult. Based on our research we decided to base our solution on the Firebase-powered backend because we wanted to focus on coding in Android and less on the server side of things, which firebase allowed us to do¹². The only problem was that our backend was already implemented in a SQL Server solution for authentication, feedback registry and general user information, which lead to our application connecting, authenticating and talking to two different databases. We considering migrating one of the databases to the other but decided we didn't have enough time to make such architectural changes in our system. We ended up using firebase for online chat and notification purposes and MySQL for the others.

Having to work with fragments made the layouts more dynamic but understanding how they work was more time consuming than we thought since we wanted our application to run on tablets as well. Fragments do not behave as static layouts, which is a gift and a curse for the developers that work with them.

Another challenged was providing the user with just the right amount of data in order for the app to complete its main task and not flood the user with unneccasary information.

Furthermore, making the application flexible to provide differenent functionaly and GUI for our two groups of users, regular users and repairman, required going deep into applications work flow and understanding what content needs to be shown at what time.

Android has definitely made sure to make work more difficult for developers with their new updated runtime permissions. As our application requires two dangerous permissions we had to make sure that in every part of the code where access to sensitive content is needed, the required permissions are present.

Since this is only a proto-app, and of course a university project, it is more than understandable that it can be extended on its functionality. Our application offers only a few Repairman category, and of course, adding more Repairman categories would make the app more complete, and more attractive to the users.

We might also think on changing the way data flow from the client to the server and backwards, because as there are more data, there is a risk of a overload for the server and the connection generally.

We've manipulated with two different databases, and, another improvement would be a single database for the application.

¹<http://myapptemplates.com/simple-android-chat-app-tutorial-firebase-integration/>

²<http://softwareengineering.stackexchange.com/a/262504>