POLITECNICO
MILANO 1863

Reservation

Design and Implementation of Mobile Applications course
Professor Luciano Baresi

WRITTEN BY RICCARDO FACCHINI
MAT: 899190

2018-2019

Deliverable specific information

Deliverable:	Design Document
Title:	Reservation Design Document
Authors:	Riccardo Facchini
Version:	2.0
Last revision:	July 16, 2019
Download page:	`https://github.com/Riccardo95Facchini/Reservation.git`
Copyright:	Copyright © 2019, Riccardo Facchini – All rights reserved

Contents

List of Figures

1 Introduction

1.1 Purpose

This document aims to detail the design of the software and of the architecture regarding the application Reservation. To do so it will be taken a more detailed approach for the description of each component and the overall architecture of the system.

1.2 Scope

Reservation is an appointment management application, designed to help the users with their daily routine by keeping track for them of their next appointments in registered shops and avoiding the hustle of queueing or trying to reach the shop managers on the phone.
The same application can be used by **customers** and **shop owners** to manage the reservations, for the former it will help with the actual reservation process and act as an agenda by keeping track of the next appointments taken, while for the latter it will display the next customers that reserved a spot while giving only minimum information to preserve the customer's privacy.
Both kinds of users have access to chat functionalities that allow them to communicate if needed and a back-end handled system of notifications that will notify them when a new reservation has been made (receivable for shops only as they can't make a reservation themselves) or a new chat message is available.
Customers are also able to leave a review score from 1 to 5 (stars) and in future update that value if they wish so.

1.3 Definitions, Acronyms, Abbreviation

- DB: DataBase
- DBMS: DataBase Management System
- GPS: Global Position System
- UID: Unique Identifier

1.4 Document Structure

This document is structured has:

1. **Introduction**, it provides an overview of the entire document.
2. **Architectural Design**, it describes different views of components and their interaction.
3. **Algorithm Design**, it describes the main algorithm and query methods.
4. **User Interface Design**, it provides an overview about the aspect of the user interfaces of the system.
5. **Implementation, Integration and Testing**, it describes the orders in which the components are implemented and the order of the integration with some testing.
6. **Appendix**, it contains software used.

2 Architectural Design

2.1 Overview

The system to be designed needs to help customers make reservations in their favourite shops in the least amount of time possible and also keep track of all the future appointments they have taken.
It also must allow shop owners to register their establishment to the platform and keep track of the next customers that have registered an appointment.
Since this interaction between user and system can be summarize as:

1. User request a service to the system.
2. System responds to the user with the requested service.

Based on this, a client-server architectural approach has been chosen.

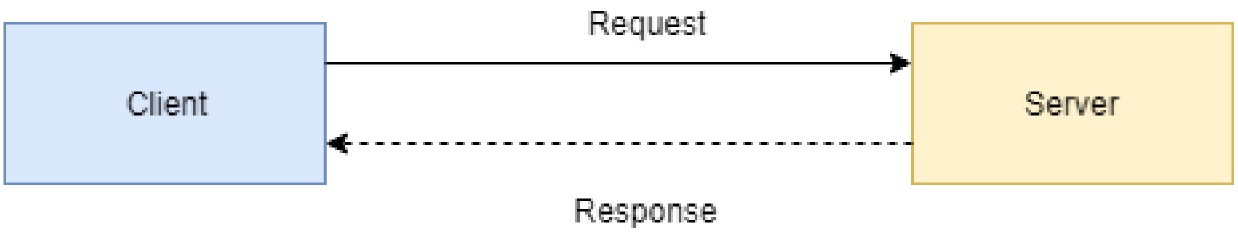


Figure 1: Client Server architecture

Furthermore, the system can be divided into three different subsystems: the presentation layer, the application layer and the data layer, where:

- The *Presentation Layer* provides the GUI of the system.
- The *Logic Layer* contains the logic of the application,that receives the requests from the user.
- The *Data Layer* stores and maintains the data needed from the system to works properly, i.e. user's & shop's information and reservations details.

2.2 Model–View–ViewModel (MVVM)

Since in the Android development it's quite easy to find conflicting arguments regarding the use of the well known Model-View-Controller pattern due to views being in part also controllers, the officially suggested architecture is the Model–View–ViewModel one, where the (graphical user interface) GUI is separated from the model by a ViewModel that handles the conversion of data and exposes to the View only the needed information.
The main advantage of using ViewModel classes is that they are part of the Android class system already, therefore the developer only needs to extend the ViewModel class to create a custom one for the needed purpose.
It should be noted how ViewModel classes are not destroyed when the View is recreated (for example when rotating the screen), meaning that data is actually preserved to allow for a better user experience.
A classic example of MVVM can be seen when using **Room** to locally store data in a MySQL database (with all the advantages of the latest Repository and DAO integration for easier use), but for this project there was no need to do so since Firebase already takes care of storing data when off-line.

2.3 Live Data

Live Data is a class of objects that allow to be observed, meaning that if a ViewModel returns a reference to a Live Data Object, the View can simply observe the given object and when a change happens it can update itself by executing the code in the call.This has been used in conjunction with FirebaseFirestore observers to update the reservation lists in real time even when the application is already opened, making therefore obsolete the use of a "refresh button".

2.4 Component View

2.4.1 Overview

In Figure 2 is possible to see the high level components of the system and the interfaces used to connect one to another, where:

- *Firebase* allows the use of many functionalities to aid development;
- *Data* is a macro category containing functionalities and services to store information:
  - *Firestore* is the cloud database system used to store information.
  - *Real-time DB* is the database used to store chat conversations.
  - *Storage* is Firebase system to store files such as user's profile pictures.
- *Firebase Auth* provides the authentication system;
- *Cloud Functions* allow to make modifications to the database when defined triggers are fired, moving computation to the back-end instead of relying on the user's device;
- *Cloud Messaging* notification system handled by the back-end once properly set-up;
- *Maps API* given by Google to use Maps functionalities such as address search and maps display;
- The *Mobile Application* is the mobile application used by a user with his/her smartphone.

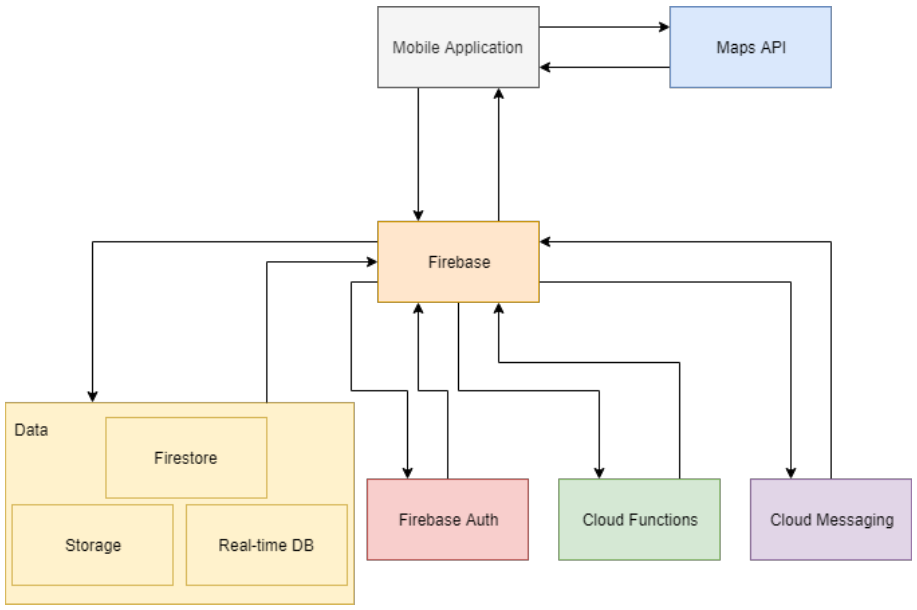


Figure 2: High level Component Diagram

2.4.2 Firestore View

Unlike traditional Entity-Relationship DB systems, Firestore is organized in Collections-Documents hierarchy that can be further nested, meaning a Document can contain one or more Collections containing other Documents and so on.
The main Collections used are shown in Figure 3, while the others shown in Figure 4 where used as support Collections in order to add functionalities.

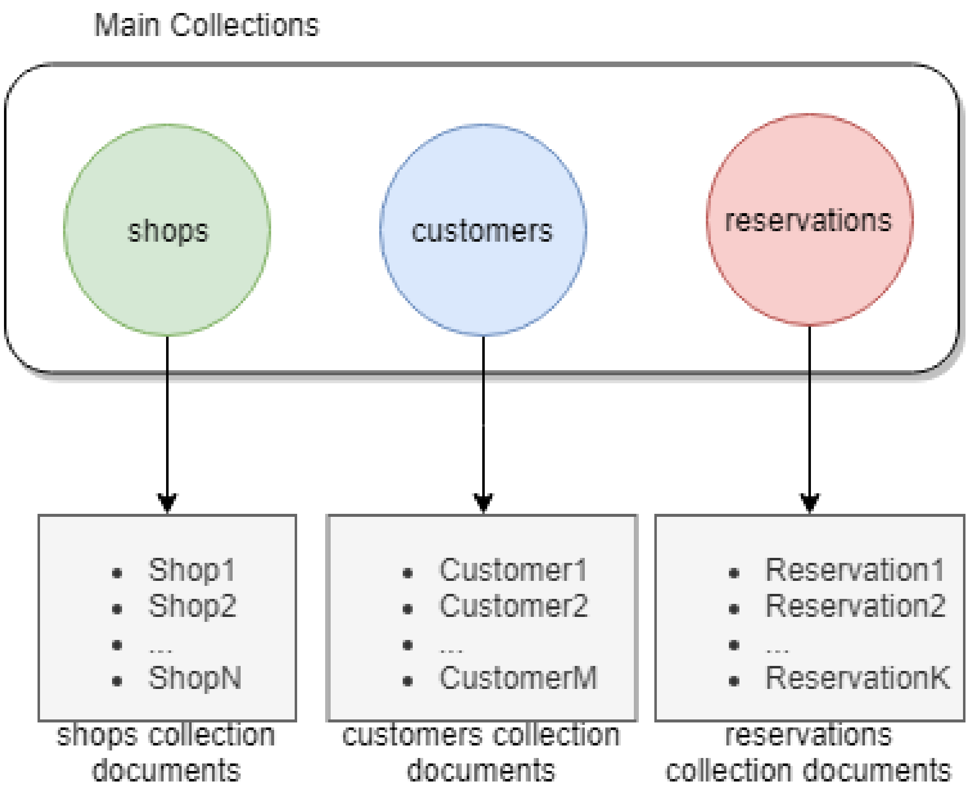


Figure 3: *Main* Collections

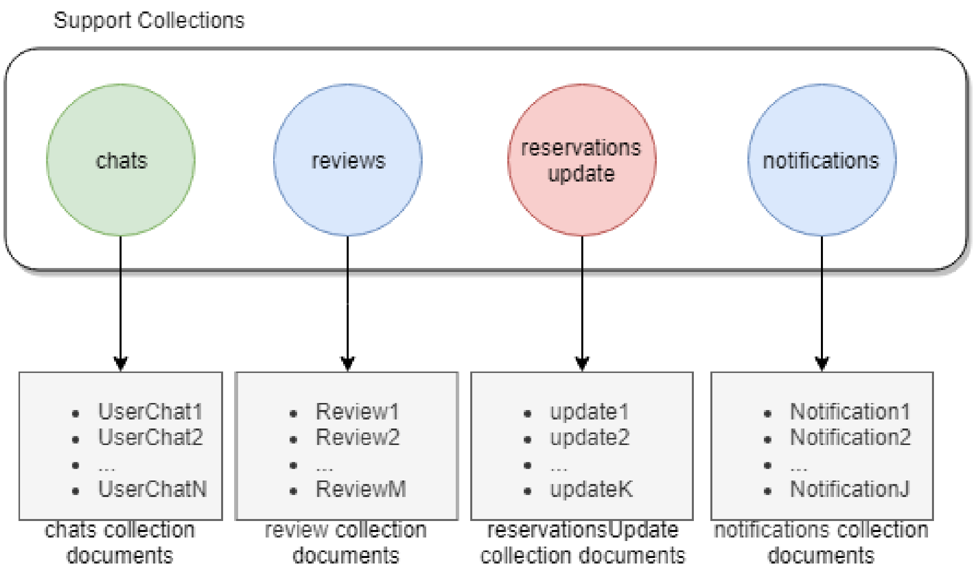


Figure 4: *Support* Collections

Each object is stored in its separate document with an unique ID (UID) given by Firestore itself, here are some examples of how each one looks like in the JSON representation:

Customer :

```
{
  "uid": "jcUgNG7wi4eZmQjakQRTQNSYWqk1",
  "name": "Bob Peterson",
  "phone": "3343200266",
  "mail": "bob.peterson@gmail.com",
  "profilePicUrl": "URL to storage" File storage is a different service
}
```

Shop :

```
{
  "uid": "0wiQ9FYr1YdL1aTPVgUe5ndCmSt2",
  "name": "Forbici & Capelli",
  "phone": "0523468159",
  "mail": "forbici.capelli@gmail.com",
  "profilePicUrl": "URL to storage", File storage is a different service
  "address": "Via Emilia Parmense 22/B",
  "city": "Piacenza",
  "zip": "29122",
  "latitude": 45.0391566,
  "longitude": 9.7204521,
  "intLongitude": 9,
  "averageReviews": 4.7,
  "numReviews": 12,
  "tags": ["barber", "parrucchiere", "forbici & capelli", "forbici", "
   capelli"],
  "hours": [
    {
      "Monday": [
        "08:30",
        "12:00",
        "Closed",
        "Closed"
      ],
      "Tuesday": [
        "07:30",
        "12:00",
        "14:00",
        "17:30"
      ],
      "Wednesday": [
        "07:30",
        "12:00",
        "14:00",
        "17:30"
      ]Cut for page length purposes
}
```

Reservation :

```
{
  "time": "1563199200000", Stored as long (time in milliseconds)
  "where": "Via Emilia Parmense 22/B",
  "shopUid": "0wiQ9FYr1YdL1aTPVgUe5ndCmSt2",
  "shopName": "Forbici & Capelli",
  "shopPic": "URL to storage", File storage is a different service
  "customerUid": "jcUgNG7wi4eZmQjakQRTQNSYWqk1",
  "customerName": "Bob Peterson",
  "customerPic": "URL to storage" File storage is a different service
}
```

Chat : Bob Peterson's

```
{
  "thisName": "Bob Peterson",
  "thisPhoto": "URL to storage", File storage is a different service
  "otherName": "Forbici & Capelli",
  "otherUid": "0wiQ9FYr1YdL1aTPVgUe5ndCmSt2",
  "otherPhoto": "URL to storage", File storage is a different service
  "lastText": "See you on Monday!",
  "isRead": true,
  "lastMsgDate": "2019-08-10 13:11:17 UTC+2"
}
```

Reservation Update : Needed only for listeners

```
{
  "reservations": 2
}
```

Review :

```
{
  "reviewScore": 3,
  "ShopUid": "DzSJgsPGfsPrDhAFwEnyK6VeRJH2",
  "customerUid": "jcUgNG7wi4eZmQjakQRTQNSYWqk1"
}
```

Notification : Once the notification is sent the entry will be automatically removed*

```
{
  "recipientUid": "0wiQ9FYr1YdL1aTPVgUe5ndCmSt2",
  "title": "Bob Peterson",
  "body": "See you on Monday!" Empty if it's a new reservation notification
}
```

2.4.3 Real-time Database View

For convenience the actual chat conversations are stored in the real-time database under the node **messages** in a series of sub-nodes where each conversation is identified by *smallerUID_biggerUID* (to be uniquely identifiable from both users without storing a new conversation UID) where the difference is given by the *compare* method that Java offers for Strings. Each message in a conversation is a new child node under the *smallerUID_biggerUID* generated UID with the following structure:

```
reservation-PRIVATE_PROJECT_UID
└─messages
  ├─15dQDKOmhKTImo4nZS9Yielor2m2_jcUgNG7wi4eZmQjakQRTQNSYWqk1
  │ └─-LjhD3f7Nzuy12g_q2LZ UID auto-generated
  │   ├─message:  "Hi, would you mind if I asked you a question?"
  │   └─user:  "Bob Peterson"
  └─DzSJgsPGfsPrDhAFwEnyK6VeRJH2_jcUgNG7wi4eZmQjakQRTQNSYWqk1
    ├─-MLDD3f7dxipo89s_23aq UID auto-generated
    │ ├─message:  "Hi, how much is it for a haircut?"
    │ └─user:  "Bob Peterson"
    └─-TjhDf5qNzuy12g_qongZ UID auto-generated
      ├─message:  "It's 15 euros"
      └─user:  "Forbici & Capelli"
```

2.4.4 Storage View

The Firebase Storage structure is simply organized in a single folder since it's only needed to store profile pictures.

```
profile_pics
├─profile_picture_user_1
├─profile_picture_user_2
├─profile_picture_user_3
└─profile_picture_user_4
```

2.4.5 Firebase Auth View

The Firebase Authentication system handles automatically the authentication of the connected user by storing:

- Identifier (for example email address)
- Provider (Email/Gmail account/Phone etc...)
- Creation date
- Last sign in
- User UID (which is then used in the DB as an identifier)

2.4.6 Firebase Cloud Functions View

Cloud functions are functions written in JavaScript that allow for actions to be executed by the Firebase server on specific triggers to relieve computation from the user's device.
For this project two functions have been written:

- **newShopReview**: triggered when a new *document.write* (which means it will trigger both on update and on create) occurs in the *reviews* collection, it is tasked to read the value of the review and update the average and total number of reviews of the shop that was reviewed. It also handles the case in which the review is just an update of an already existing one.
- **sendNotification**: triggered when a new *document.create* occurs in the *notification* collection, it is tasked with sending the content of the notification to the given recipient. It also handles the deletion of the created element in the database to avoid storing now useless data.

2.4.7 Firebase Cloud Messaging View

Built in system for messaging (notifications), since at login each user subscribes to an unique topic (identified by the UID of user itself), the Cloud Messaging system will send the notification content (by using the *sendNotification* cloud function) to only that user. The notification will be received as soon as the user's device is reachable.

2.4.8 Maps API

Google Maps API have been added to the project in order to use the **places** and the **maps SKD for Android** that are needed respectively to search an address from where the search will start and to show the map view when a shop is selected.

2.4.9 Mobile Application

The *Mobile Application* is used by the user via its own smart device. The *Mobile Application* communicates directly with the Firebase and Google Maps system using the provided API.

2.5 Implementation choices

The technology chosen for the implementation on the system are all based on Java and JavaScript since it is the most common way of developing Android applications using Firebase and the availability of documentation and other sources of learning materials are abundant, plus some experience was already obtained during past projects.
It also offers the possibility of adding new functionalities in future, making the system more scalable given how the platform is constantly evolving

To summarize the technologies used are:

Mobile application: entirely Android (Java) based with Firebase and Google Maps API added in order to use the external tools. Other modules such as RecycleViews and Cards have been added.

Main DBMS: Firestore was selected over the old real time database system since it has been developed as a successor to substitute it. I provides easy to use DB mechanisms, at the loss of complexity such as complex queries and it's not a relational DB system which is usually more familiar.

Chat DBMS: It has been used the real-time database because it's ideal to store simple structures like the one needed for chats and shown in subsubsection 2.4.3.

Storage: Firebase Storage system has been chosen since it's part of the ecosystem.

Cloud Functions: Just like Storage, Firebase Cloud Functions have been chosen since they are already part of the ecosystem and they were needed for some back-end actions.

Authentication: Firebase auth allows for easy and fast authentication methods already handled.

Cloud Messaging: Part of the Firebase ecosystem, handles the majority of the work needed to deliver notifications to users.

2.6 Runtime View

Here are represented the two most important runtime views by using diagrams that highlight the main actions the user has to follow in order to complete the given task.
In Figure 5 it's highlighted in red the interaction that the system has with the authentication system, then the two branches between choosing to register as a customer (green) or shop (blue) are available to be picked.
In Figure 6 are again highlighted in red the interactions of the application with the Firebase systems, the yellow one is referred to the use of Google Maps API (in this case the **places** API to be exact) while in blue is noted the optional choice to select a different distance than the default one.
It should be noted how none of the handled errors or problems are shown to keep the diagrams simple.

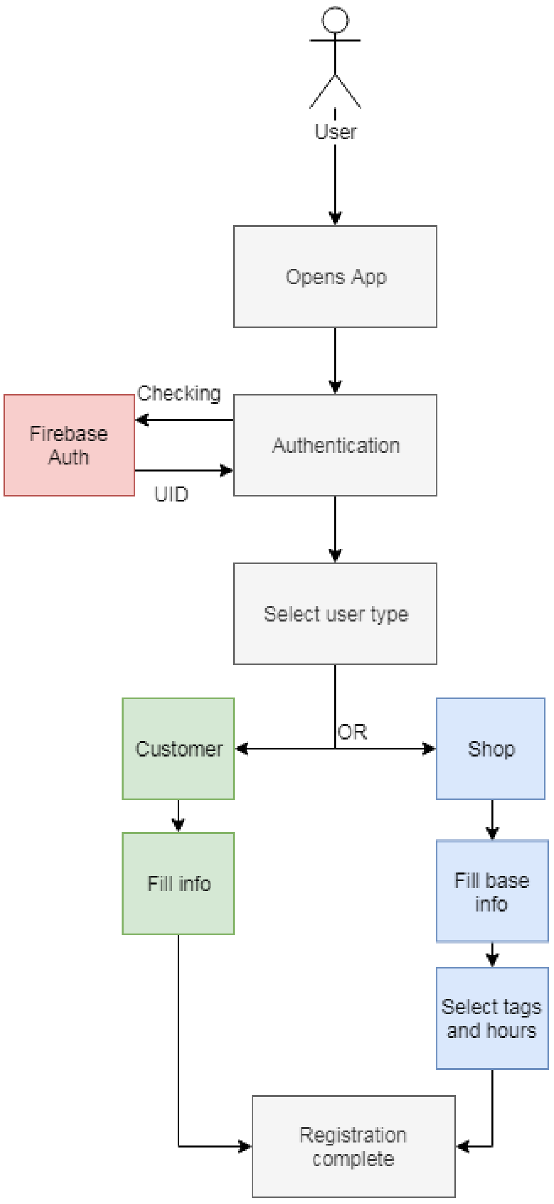


Figure 5: *User* registration diagram

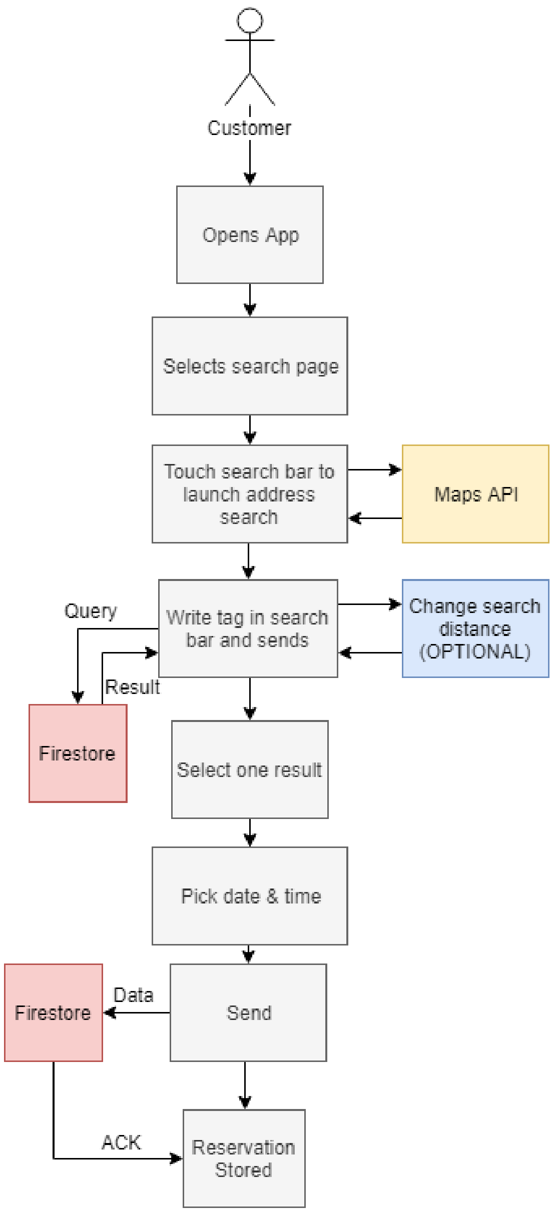


Figure 6: *Customer* reservation process

3 Algorithm Design

In this section will be described the main algorithms of the system, which are the search of a shop in the area and the next reservations query.

3.1 Distance Calculation

Since the GPS system is heavy on battery consumption, it's not always reliable (especially when just turned on) and the user may want to search in a different location than where he/she is right now, when the customer wants to start a search a dialog shown in Figure 14 asking for the address is shown, then the only needed values of **latitude** and the **longitude** are extracted.
Given the maximum **distance** the user wants the search to be done, the deltas for both the latitude and the longitude can be computed as follows:

$$deltaLat = (distance/earthRadius) * (180 * Math.PI)$$

$$deltaLng = Math.toDegrees((dist/(earthRadius*Math.cos(Math.toRadians(currentLat)))))$$

3.2 Queries

Firestore allows for some query functionalities, these can be used to retrieve data even when the the UID is not sufficient to identify the requested information.

3.2.1 Shop Search Query

Due to a limitation of Firestore, queries are made by searching for shops with the given tag, the latitude inside the min and max values (found respectively by subtracting and adding to the current latitude the value of deltaLat) and then a third parameter called *intLongitude* is used, corresponding to the longitude truncated.
This has to be done since Firestore doesn't allow for queries with different search parameters except for equals, so instead of doing two queries (one for latitude and one for longitude) and then merging the common values, one query is done limiting the search to only one longitude and then the values retrieved are checked locally if they are inside the range. In the edge cases where the search spans over more than one longitude integer value, then more than one query will be made with the different values set, each query will return a different set than the others obviously so it must only be checked if the shops are in the correct range and no control between queries result has to be done.
Here is the example of the first (and usually only) query:

```
shopsCollection
.whereArrayContains("tags", searchedText)
.whereGreaterThanOrEqualTo("latitude", minLat)
.whereLessThanOrEqualTo("latitude", maxLat)
.whereEqualTo("intLongitude", minIntLng)
.get();
```

3.2.2 Reservations Query

The next/past reservations are queried once when the application is opened and after that a listener is placed on the current user entry in the **reservationsUpdate** collection, when a change occurs the trigger will result in a new query of both future and past reservations, the ViewModel will then handle the update of the Live Data lists and the views will automatically update once the observed value changes and the trigger is fired.
Here for example is show the query that will find the next reservations for a customer:

```
reservationsCollection
.whereEqualTo("customerUid", thisUid)
.whereGreaterThan("time", now.getTime())
.orderBy("time", Query.Direction.ASCENDING)
.get();
```

And one for the past reservations of a shop:

```
reservationsCollection
.whereEqualTo("shopUid", thisUid)
.whereLessThan("time", now.getTime())
.orderBy("time", Query.Direction.DESCENDING)
.get();
```

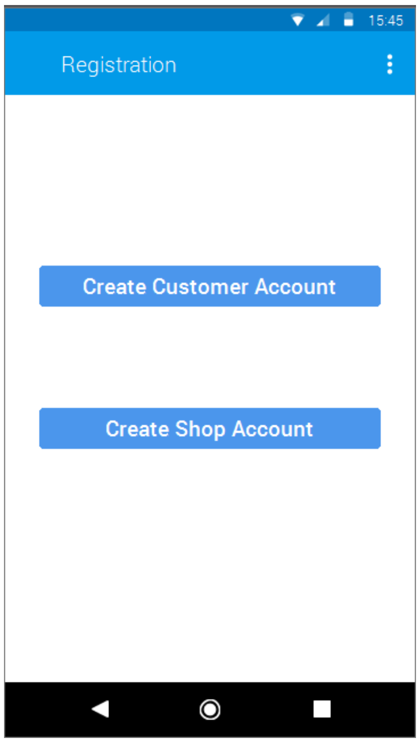
The duplicated elements in the *reservation* stored object allow for one single query and therefore (given the asynchronous nature of Firebase requests) there is no need to locally order the results, unlike when a get request for each shop/customer that has to be displayed in the reservation card has to be done in the case of non-duplicated data. It should be remembered that duplicated data when needed is actually recommended by the Firebase team since the JSON database is much faster than a relational one when reads are in the majority of the actions performed (also the lack of classic relational functionalities like foreign keys make almost impossible to avoid duplicating data).

4 User Interface Design

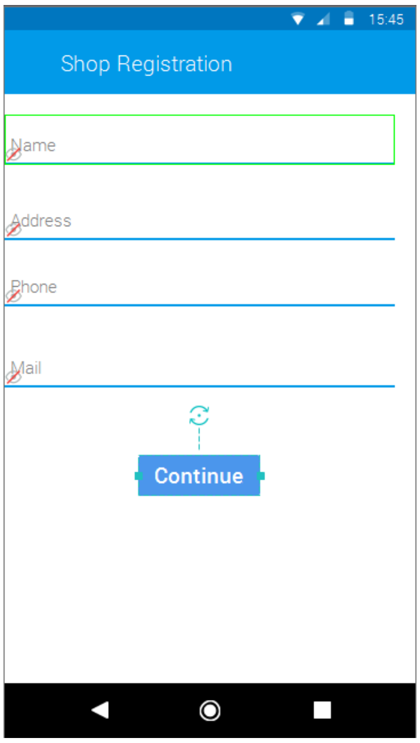
These are some representations of how the mobile application should look like.

4.1 Mockups

In this section some concept Mockups that were used to then develop the first iteration of the application are shown, it should be noted that the finished product must follow the concept of these designs and not use the actual graphical elements here showcased, but a professional artist should be instead hired to design the interface given these guidelines.

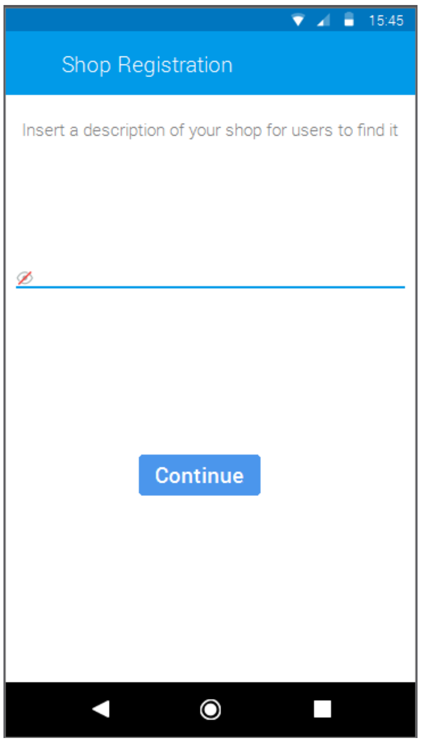


(a) User type choice

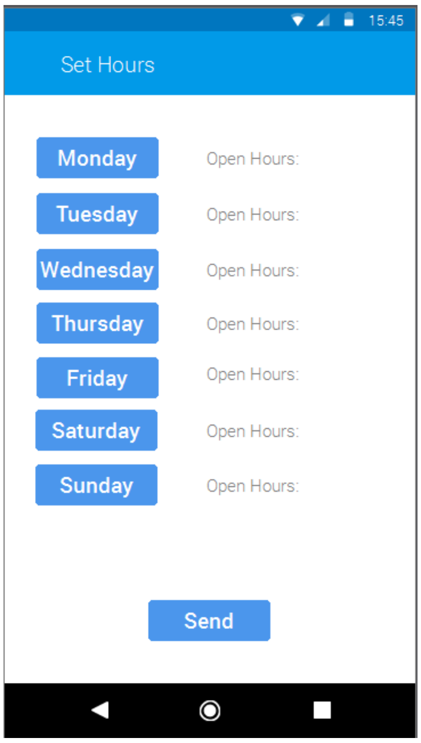


(b) Shop registration

Figure 7: Login on the left (a) and first step of Shop registration on the right (b)

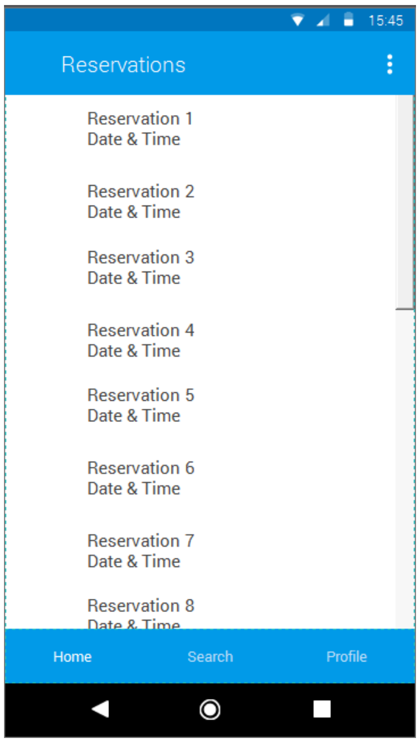


(a) Shop description request

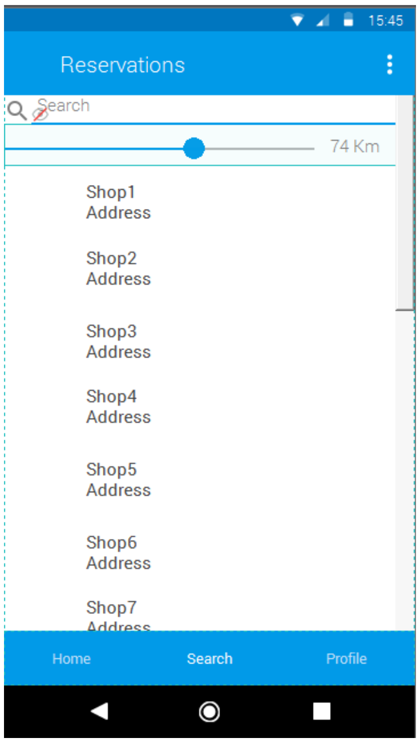


(b) Shop hours request

Figure 8: Second page for shop registration (a) and third one(b)



(a) Customer Home

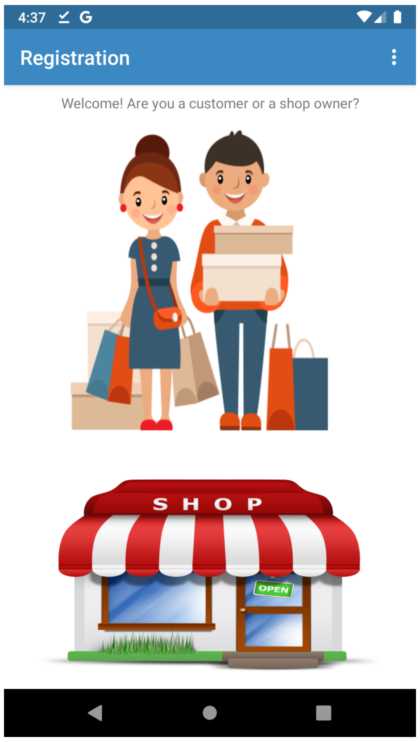


(b) Customer Search

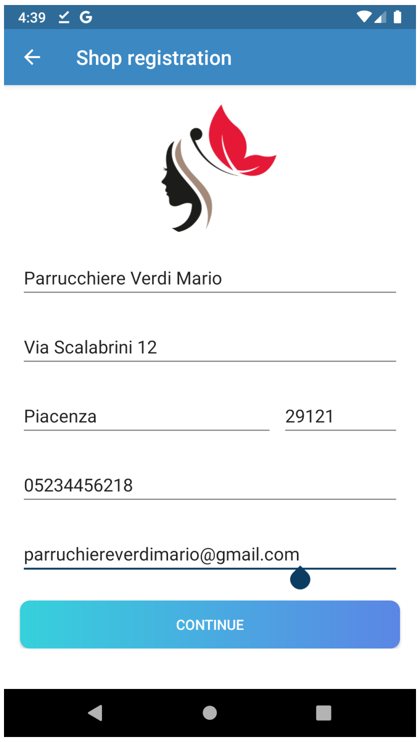
Figure 9: The customer’s home page (a) and the search screen (b)

4.2 Application Screenshots

Here are instead shown the screenshots of the implementation of the application from the emulator. Some pages are not shown since they would look the same as others (for example editing a shop's information opens the same screens of the registration with the fields already inserted).



(a) Registration Home



Parrucchiere Verdi Mario

Via Scalabrini 12

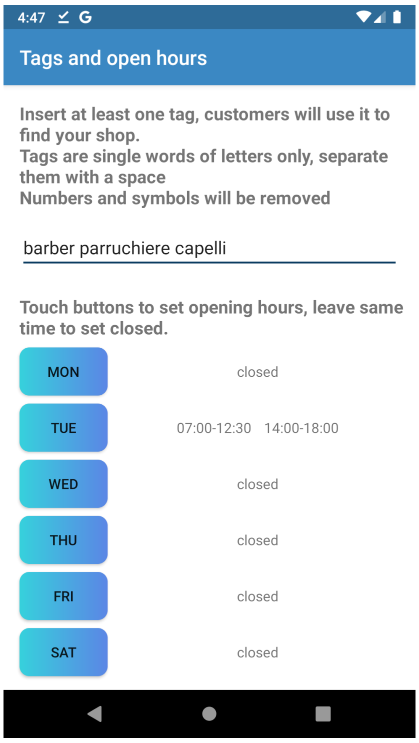
Piacenza 29121

05234456218

parruchiereverdimario@gmail.com

(b) First shop registration page

Figure 10: Registration home on the left(a) first of the two registration pages for the shop on the right(b)


4:47

Tags and open hours

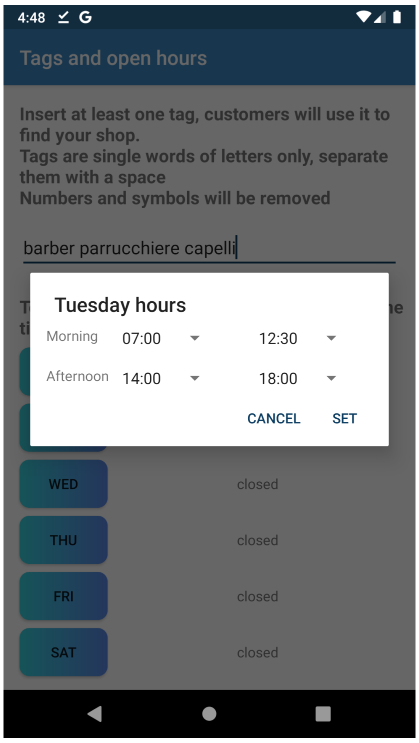
Insert at least one tag, customers will use it to find your shop.
Tags are single words of letters only, separate them with a space
Numbers and symbols will be removed

barber parruchiere capelli

Touch buttons to set opening hours, leave same time to set closed.

MON	closed
TUE	07:00-12:30 14:00-18:00
WED	closed
THU	closed
FRI	closed
SAT	closed

(a) Insertion of tags and selection of opening hours


4:48

Tags and open hours

Insert at least one tag, customers will use it to find your shop.
Tags are single words of letters only, separate them with a space
Numbers and symbols will be removed

barber parrucchiere capelli

Tuesday hours

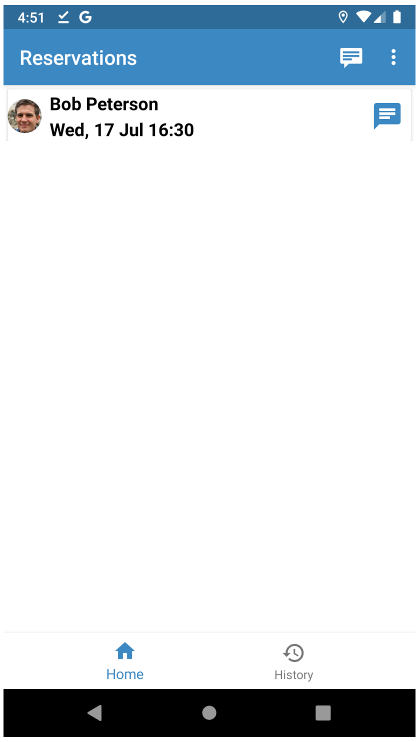
Morning	07:00	12:30
Afternoon	14:00	18:00

CANCEL SET

WED	closed
THU	closed
FRI	closed
SAT	closed

(b) Opening hours detail

Figure 11: Page for inserting tags and hours (a) with detail of the alert shown to pick times(b). Last buttons are below the view.

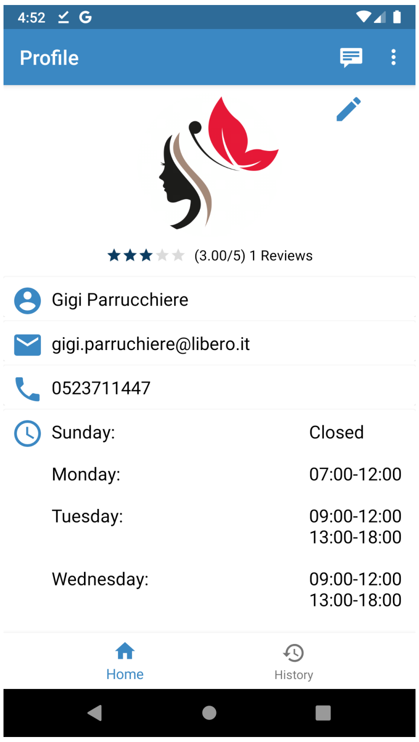

4:51

Reservations

Bob Peterson
Wed, 17 Jul 16:30

Home History

(a) Shop's home page


4:52

Profile

★★★☆☆ (3.00/5) 1 Reviews

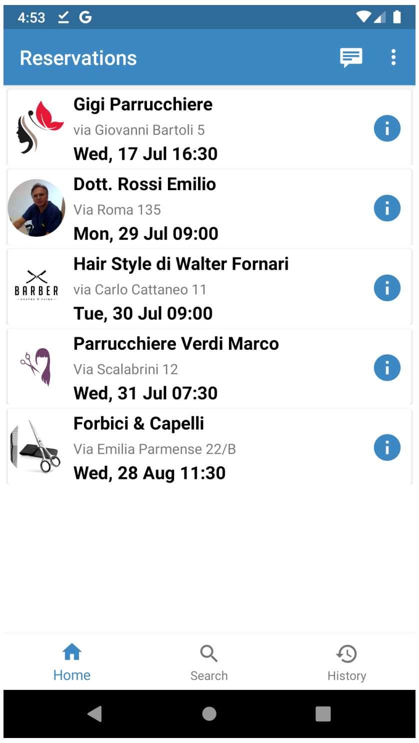
Gigi Parrucchiere
gigi.parruchiere@libero.it
0523711447

Sunday:	Closed
Monday:	07:00-12:00
Tuesday:	09:00-12:00 13:00-18:00
Wednesday:	09:00-12:00 13:00-18:00

Home History

(b) Shop's profile

Figure 12: Shop home page with next reservations (a) and shop profile recap with button to edit info(b).


4:53

Reservations

Gigi Parrucchiere
via Giovanni Bartoli 5
Wed, 17 Jul 16:30

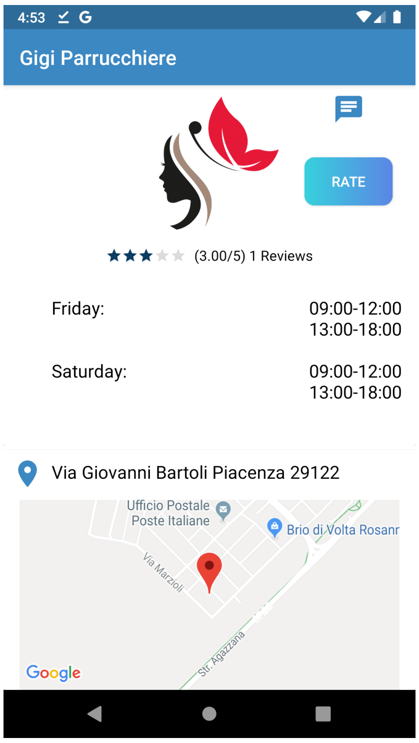
Dott. Rossi Emilio
Via Roma 135
Mon, 29 Jul 09:00

Hair Style di Walter Fornari
via Carlo Cattaneo 11
Tue, 30 Jul 09:00

Parrucchiere Verdi Marco
Via Scalabrini 12
Wed, 31 Jul 07:30

Forbici & Capelli
Via Emilia Parmense 22/B
Wed, 28 Aug 11:30

(a) Customer's home page


4:53

Gigi Parrucchiere

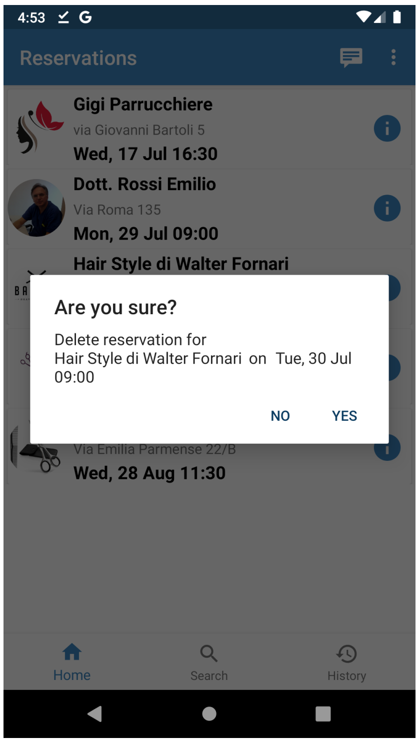
(3.00/5) 1 Reviews

Friday:	09:00-12:00 13:00-18:00
Saturday:	09:00-12:00 13:00-18:00

Via Giovanni Bartoli Piacenza 29122

(b) Details (scrolled down to see map)

Figure 13: Customer home page with next reservations (a) details shown when the (i) icon is pressed(b).


4:53

Reservations

Gigi Parrucchiere
via Giovanni Bartoli 5
Wed, 17 Jul 16:30

Dott. Rossi Emilio
Via Roma 135
Mon, 29 Jul 09:00

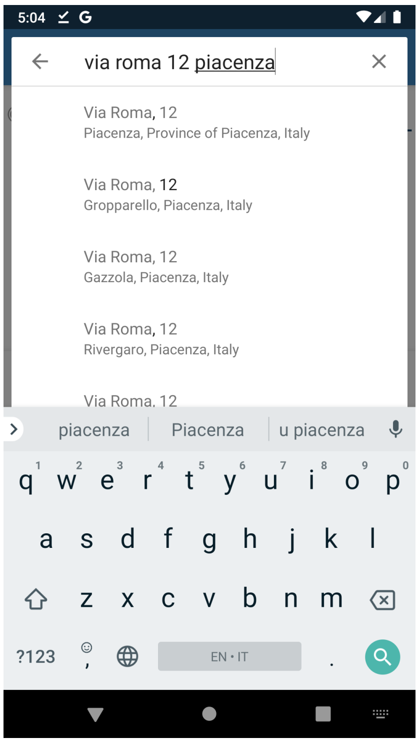
Hair Style di Walter Fornari

Are you sure?

Delete reservation for
Hair Style di Walter Fornari on Tue, 30 Jul 09:00

Via Emilia Parmense 22/B
Wed, 28 Aug 11:30

(a) Delete reservation


5:04

via roma 12 piacenza

Via Roma, 12
Piacenza, Province of Piacenza, Italy

Via Roma, 12
Gropparello, Piacenza, Italy

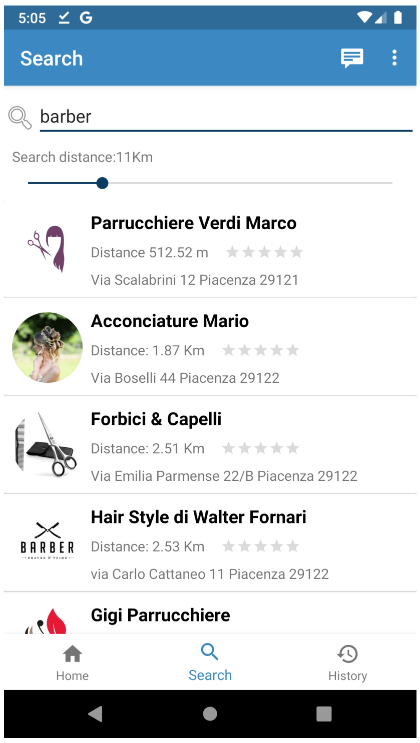
Via Roma, 12
Gazzola, Piacenza, Italy

Via Roma, 12
Rivergaro, Piacenza, Italy

Via Roma, 12

(b) Search address

Figure 14: Popup after long click on a reservation card (a) and search page when customer prompted for the search start address(b).



5:05

Search

barber

Search distance:11Km

Parrucchiere Verdi Marco
Distance 512.52 m
Via Scalabrini 12 Piacenza 29121

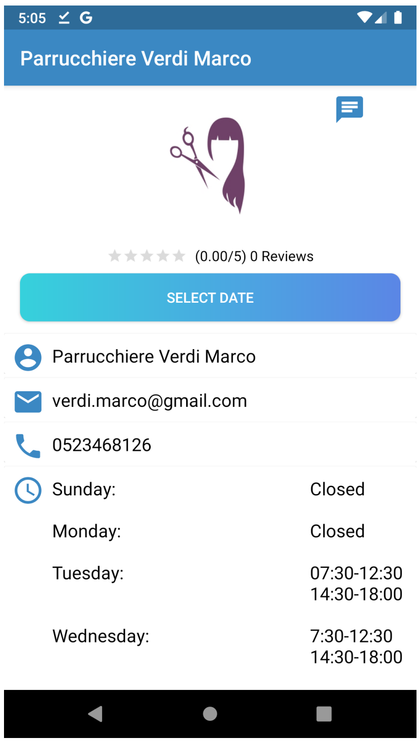
Acconciature Mario
Distance: 1.87 Km
Via Boselli 44 Piacenza 29122

Forbici & Capelli
Distance: 2.51 Km
Via Emilia Parmense 22/B Piacenza 29122

Hair Style di Walter Fornari
Distance: 2.53 Km
via Carlo Cattaneo 11 Piacenza 29122

Gigi Parrucchiere

(a) Search shop



5:05

Parrucchiere Verdi Marco

(0.00/5) 0 Reviews

SELECT DATE

Parrucchiere Verdi Marco

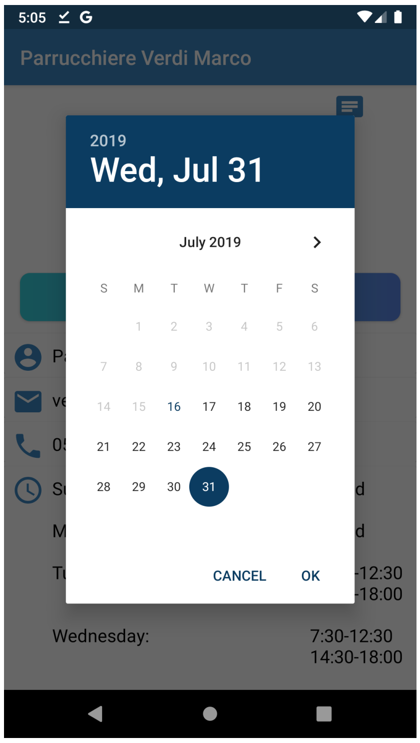
verdi.marco@gmail.com

0523468126

Sunday:	Closed
Monday:	Closed
Tuesday:	07:30-12:30 14:30-18:00
Wednesday:	7:30-12:30 14:30-18:00

(b) Search result selection

Figure 15: Page when search is successful (a) and page after a shop is selected (b).



5:05

Parrucchiere Verdi Marco

2019

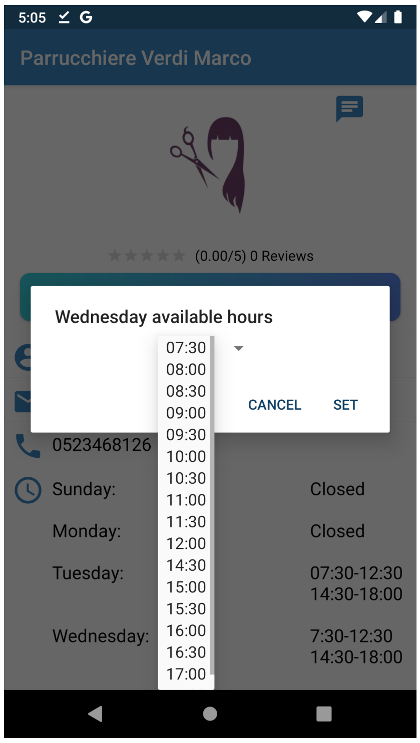
Wed, Jul 31

July 2019

S	M	T	W	T	F	S
	1	2	3	4	5	6
7	8	9	10	11	12	13
14	15	16	17	18	19	20
21	22	23	24	25	26	27
28	29	30	31			

Wednesday: 7:30-12:30 14:30-18:00

(a) Picking the day



5:05

Parrucchiere Verdi Marco

(0.00/5) 0 Reviews

Wednesday available hours

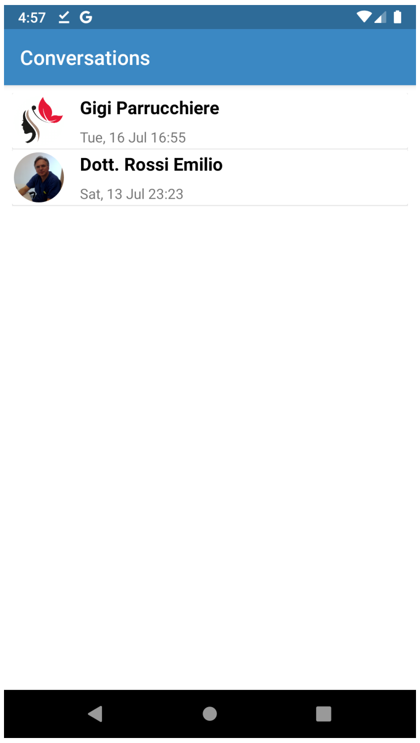
07:30
08:00
08:30
09:00
09:30
10:00
10:30
11:00
11:30
12:00
14:30
15:00
15:30
16:00
16:30
17:00

0523468126

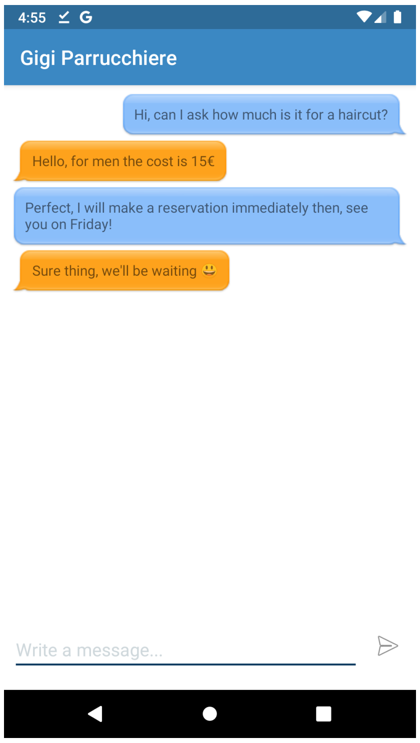
Sunday:	Closed
Monday:	Closed
Tuesday:	07:30-12:30 14:30-18:00
Wednesday:	7:30-12:30 14:30-18:00

(b) Hour selection popup

Figure 16: Day pick popup dialog (a) and selection of available hours (b).



(a) Chat Homepage



Hi, can I ask how much is it for a haircut?

Hello, for men the cost is 15€

Perfect, I will make a reservation immediately then, see you on Friday!

Sure thing, we'll be waiting 😃

(b) Opened chat with a shop owner

Figure 17: Chat Homepage with every active chat (a) and one opened (b).

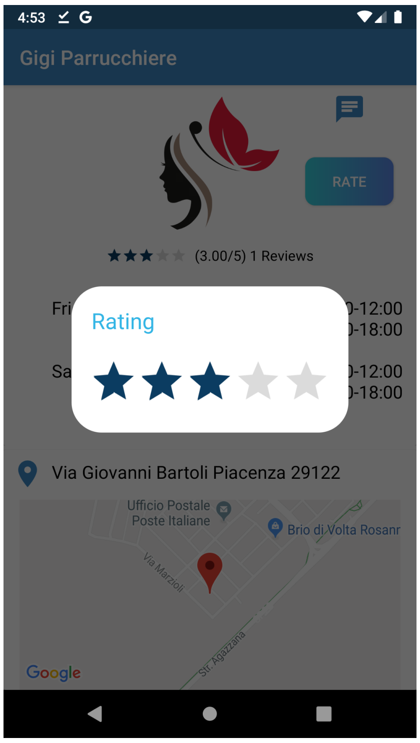


Figure 18: Customer rating a shop from info page

5 Implementation, Integration and Testing

In this section is present the documentation regarding the implementation order of the different components of the system and their integration with one another.

5.1 Entry Criteria

What follows is an illustration of the requisites that need to be fulfilled before the Integration phase can start.

Design Document:
This document and the initial mockups must be finished before starting with the development, it must be noted how it has to be updated after the development is finished in order to integrate it with what changed during development as the DD is not fixed and is subject to changes if needed.

Documentation:
Each method and class must be fully documented using JavaDoc in order to assure that the code is easily understandable, making it easier to extend and maintain even by different people that may end up working on the system in the future.
Names of classes, methods and variables must be chosen with the intent of communicating the reason of their existence and not be confusing or too similar to one another, also they should follow the standard Java naming conventions.
Also names should use a cascade system such as: Activity_TypeOfUser_Screen_Details.

5.2 Elements to be Integrated

1. *Database*: Both DB and DBMS are handled by the Firebase API, in particular Firestore, meaning that until a proper release is planned the free to use testing plan is more than enough.

2. *Notification system*: Using Cloud Messaging and trigger events from Cloud Functions.

3. *Maps*: Integration with Google Maps API to allow for maps to be displayed (not simple screenshot but actual widgets) and auto-complete for address searching.

4. *Client*: Composed by the *mobile application* it holds the logic of the system and the presentation system. The integration should focus on making sure that the communication doesn't imply high waiting time and that the graphical interface behaves as it should regarding the received data, displaying error when needed without crashing.

5.3 Testing

5.3.1 Firebase Test-Lab

For testing the Firebase test-lab functionality was used at first, but the Robo system wasn't able to progress further than the Firebase Auth login because it was not able to create a customer or shop account with the needed information, ending up in a "Passed" execution after 7 minutes because the application never crashed even though the inputs where quite unexpected.
The functionality was put aside for the moment given that there are very few uses before it becomes paid for each time it is run.

5.3.2 Beta Testing

12 people have been asked to try and later rate the application, the sample was composed by people between 20 and 25 years of age, 7 males and 5 females. They were told that the study was about the response they gave and not about the quality of the application in order to mitigate any bias they may have when expressing their opinion, all data was recorded using a Google Form later submitted to them, here are the results.

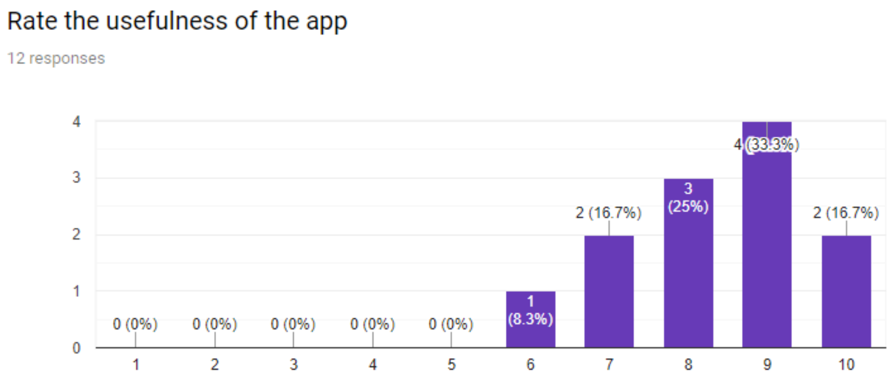


Figure 19: Response beta testing 1

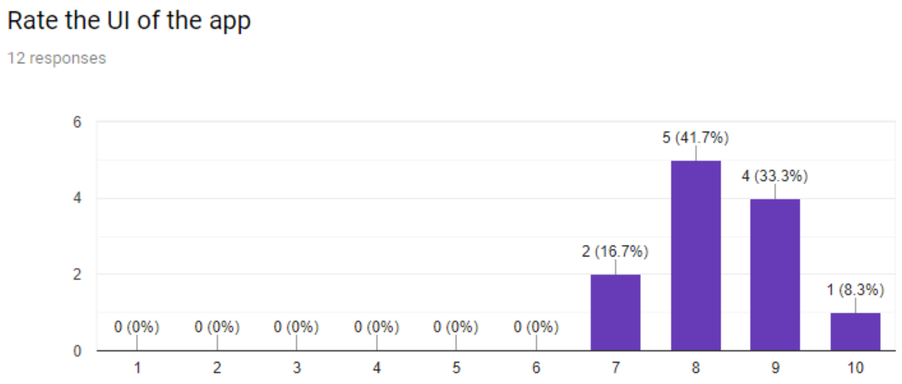


Figure 20: Response beta testing 2

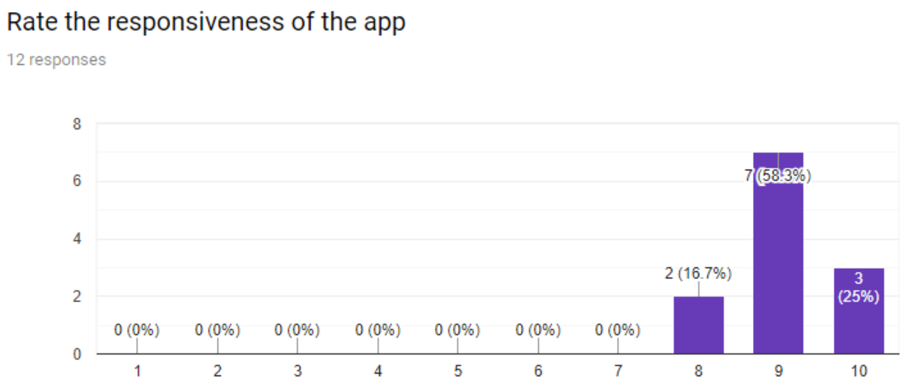


Figure 21: Response beta testing 3

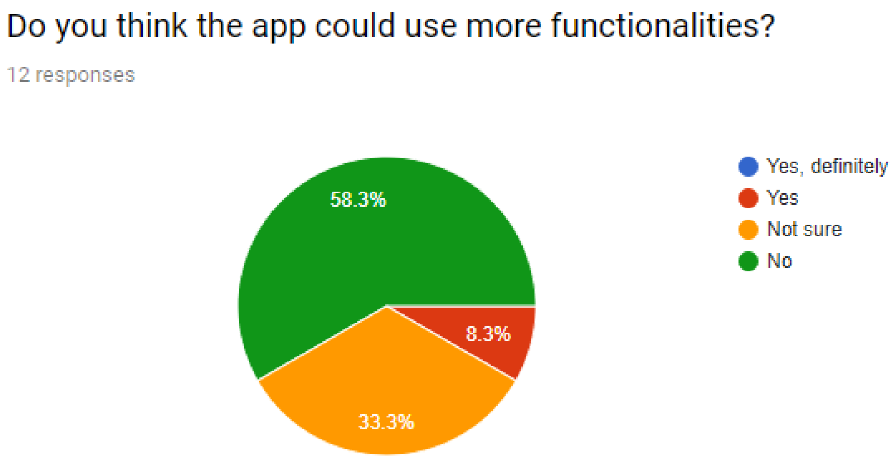


Figure 22: Response beta testing 4

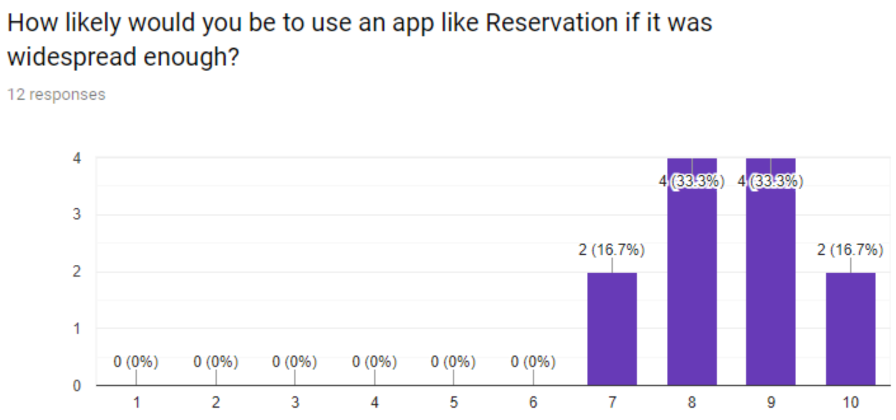


Figure 23: Response beta testing 5

Even though the sample size was quite small the results of the testing can be considered quite satisfactory as a first testing by people that aren't familiar with software development.

6 Appendix

6.1 Software & Hardware Used

1. Texmaker as an editor for LaTeX.
2. Draw.io for diagrams.
3. Justinmind Prototyper for Mockups.
4. Android Studio from 3.3.1 to 3.4.2 for developing.
5. Vegas Pro 16.0 Trial for elevator pitch video.
6. NodeJs to implement Cloud Functions.
7. Visual Studio Code to write JavaScript Cloud Functions.
8. Microsoft PowerPoint for presentation.
9. Physical Nexus 5X with Android Oreo 8.1.
10. Virtual Nexus 5X and Pixel 2 both with Android Pie 9.0.
11. Google Maps API.
12. Git & GitKraken for version control.