Final Report
Prepared for SEG 2105 B
By
Chenxuan Gao, Jack Snelgrove, Eric Kwak
Dec 4, 2022

Table of Contents

List of Figures

List of Tables

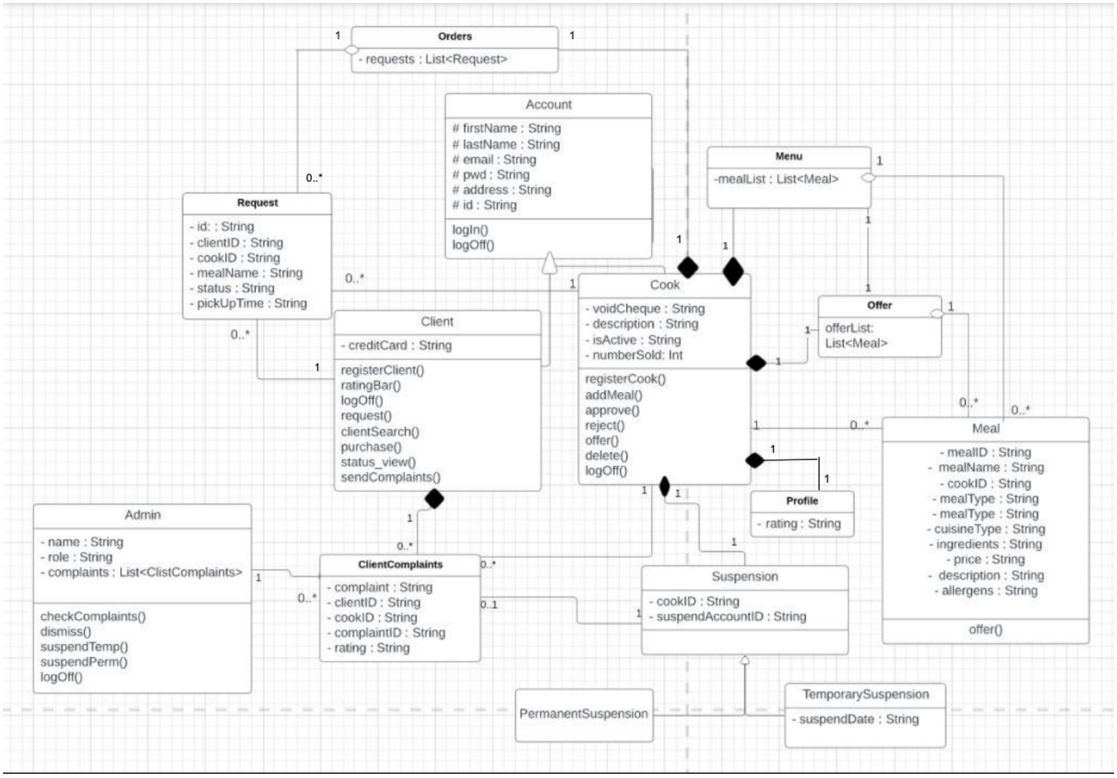
1.0 Introduction

Mealer project is an Ottawa-based sharing application where local cooks can sell meals to clients remotely and allow clients to purchase food from their phones.

This report is oriented towards helping any user to become familiar with the mealer app, who could register as a client, cook, or admin. The client has the following features: (a) purchase meal; (b) see the ordered meal. They can search for the meal by its name, real name, and cuisine type through the purchase interface. Then the user can order the meal by specifying a pickup time. The user can view their order's status and rate the cook on the Ordered interface. As for the cook, they will have a regular menu and an offer menu. The regular menu represents the meal they will be able to cook but not necessarily can be offered at the time, which means only the offered meal are available to the customer. Cook can also see a list of meals requested by the client and their ratings from the client. Lastly, the admin can only see a list of complaints from clients and have the authority to suspend a cook temporarily or permanently.

This report is separated into four sections. The first has the updated UML diagram. The second specifies the contributions of team members for each deliverable. The third section will include all the screenshots of the mealer app. The last sections will summarize our group's lesson from this project.

2.0 UML Diagram



3.0 Contributions of Team Members

Members / Deliverables	Chenxuan Gao (300229429)	Jack Snelgrove (300264568)	Eric Kwak (300264568)
Deliverable 1	Created the project files and finish the code for cook, client registration	Fixed bugs and optimized the visuals on both client and cook features	Created and fixed the domain model (UML), created admin account
Deliverable 2	Updated new account features, complete all admin functionalities	Updated the Account class, added suspend Temp and Perm class	Updated the UML. Clean up some visuals in the Admin activity
Deliverable 3	Fixed a bug in Cook/Client registration classes, optimized the if-else clause in MainActivity activity	Added the unit tests, created menu and menu item classes	Updated the UML, MainActivity to the latest version
Deliverable 4	Complete search function, fixed the client Ordered menu bugs, fixed complaints bugs, add the unit tests.	Updated admin class, corrected cook activity, and client activity, add log off buttons	Updated the final UML, created Request, Mela class

Table 1 Contribution Table

4.0 Screenshots of the App

This section will present all the screen shots of the app.

4.1 Client

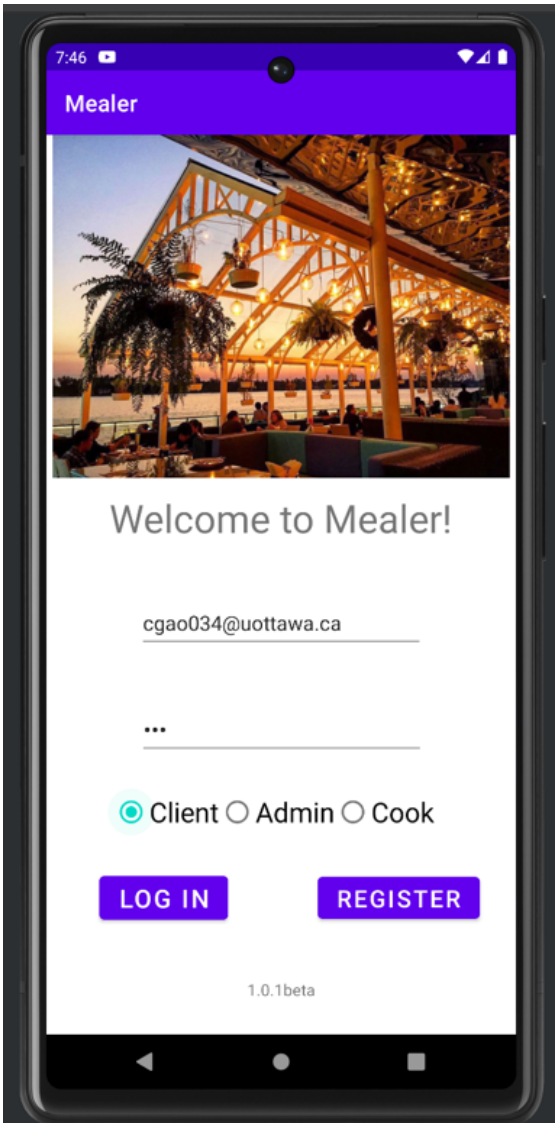


Figure 1 Client Log in Correct

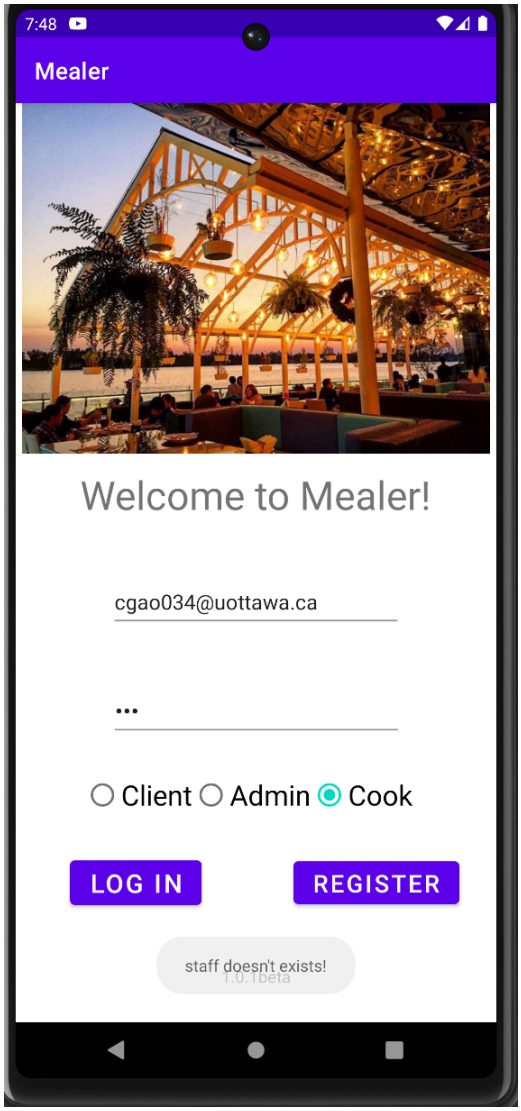


Figure 2 Client Log in Incorrect

The client will be able to log in as figure 1 shows, and must select the “Client” radio group, otherwise a toast shown in figure 2.

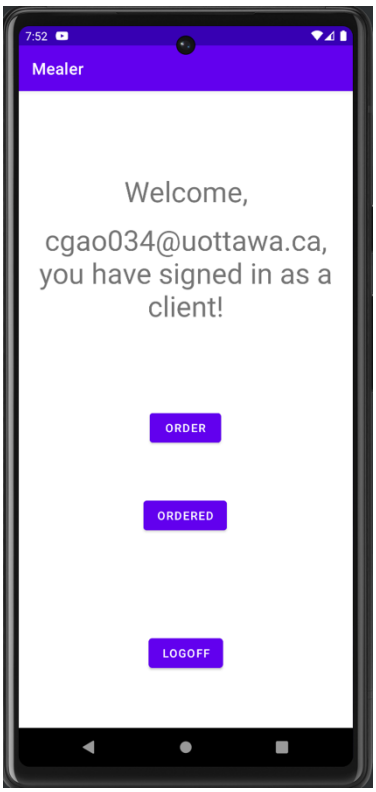


Figure 3 Client UI

Figure 3 Figure 3 shows the client interface, the order button will lead the user to the search activity, and the client can view his or her ordered meal through an ordered button. The log-off button will help the client quickly log off if he or she decides to do so.

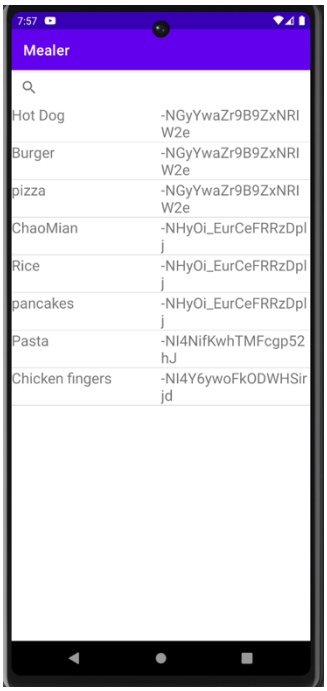


Figure 4 Client Search UI

Order

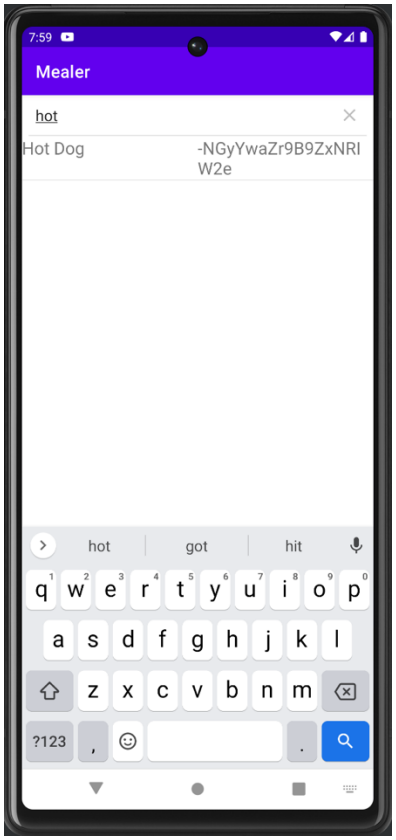


Figure 5 Search by Name

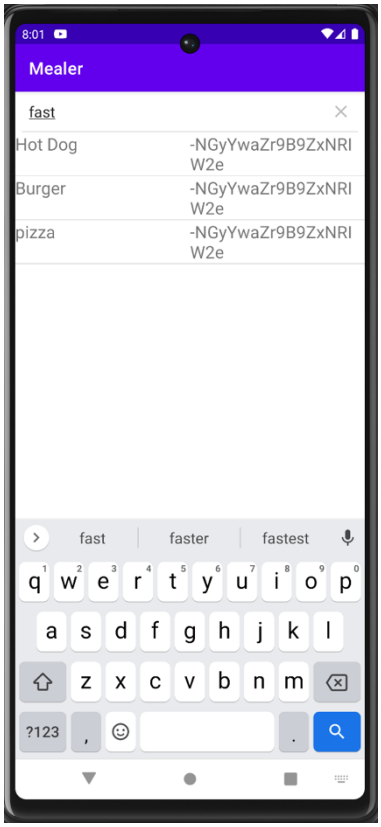


Figure 6 Search by Type

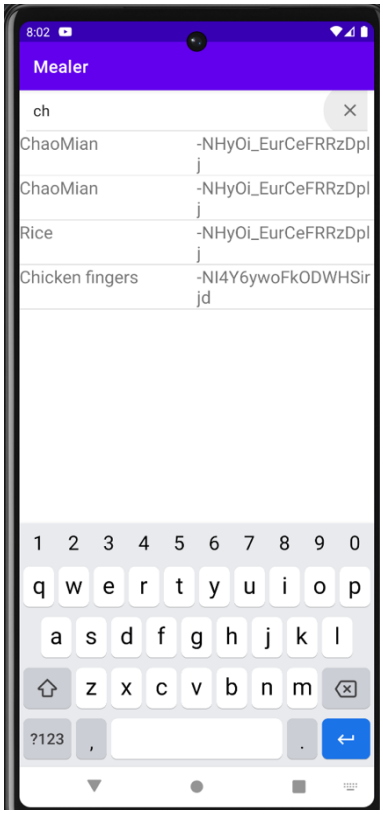


Figure 7 Search by Cuisine

After clicking the order button, the client can search by the meal name, meal type, and cuisine type. In figure 5, the search for the string “hot dog” will filter all meal name which does not contain the string “hot dog”; similarly, searching the meal by type “fast food” will give the results in figure 6; lastly, search by the cuisine type “Chinese” will show all the Chinese food, figure 7’s results are as expected.

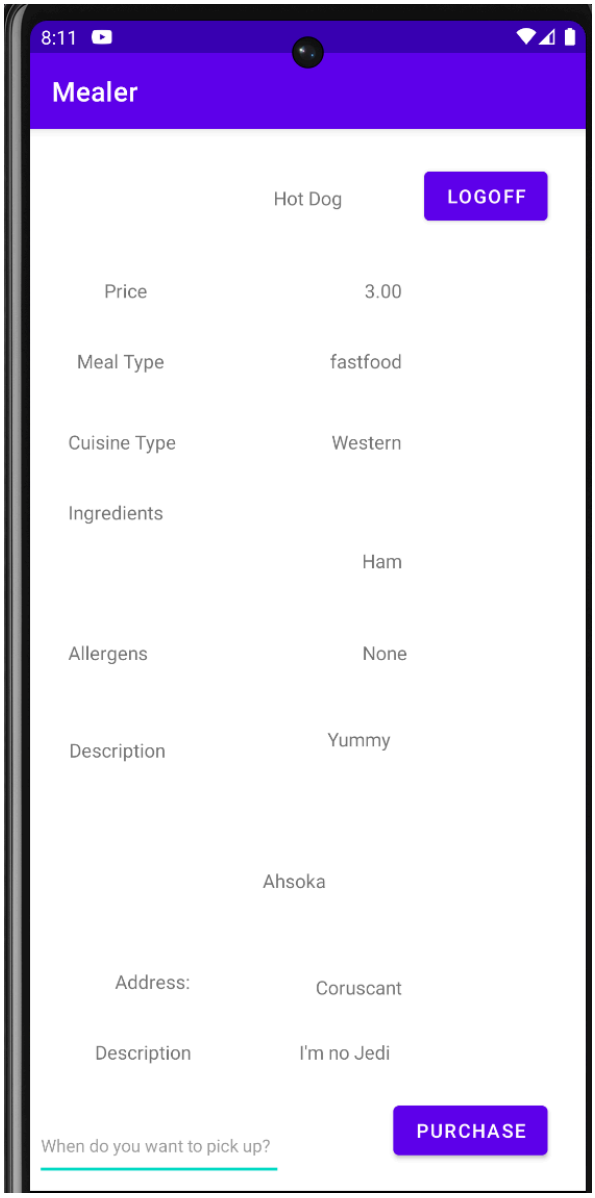


Figure 8 Purchase Menu

Clicking one of the meals in the search list view will display the activity where the client can specify a pick up time and purchase the meal. It can be seen from figure 8 that the client can see all the information about that particular meal and the cook who can offer the meal.

Ordered

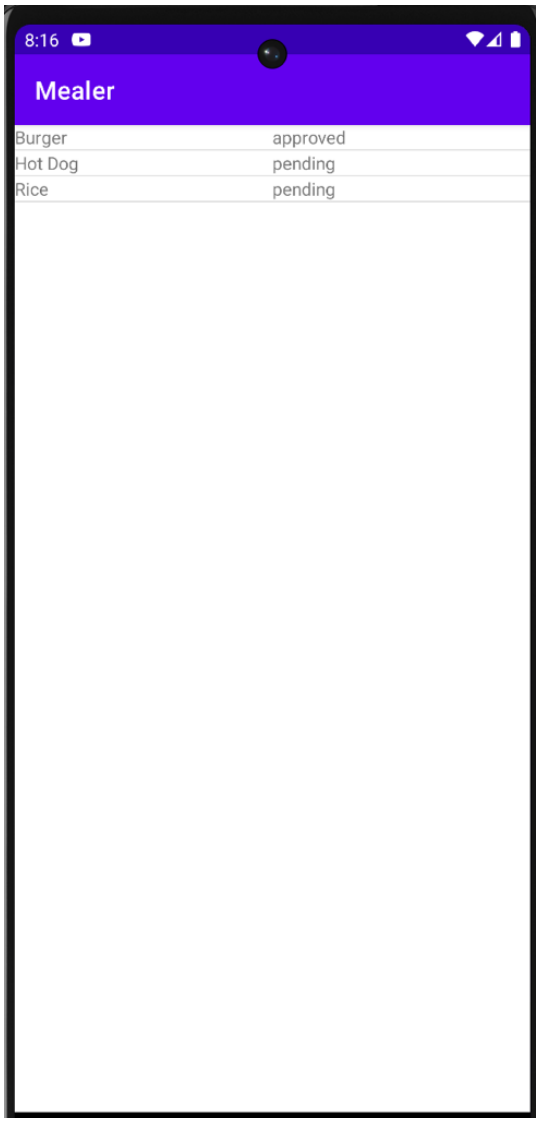


Figure 9 Ordered Meal

Figure 10 Order Status

The ordered button will lead the clients to the order they have purchased. If the cook has not yet approved the order request for the meal, the status, shown in figure 9 and figure 10 ‘s right side, will remain pending. Therefore, the client cannot click on that meal, as figure 10 shows; the client cannot click if the status is rejected. If the status is approved, then a client can click on it and rate the cook or send complaints to the cook.

Rating

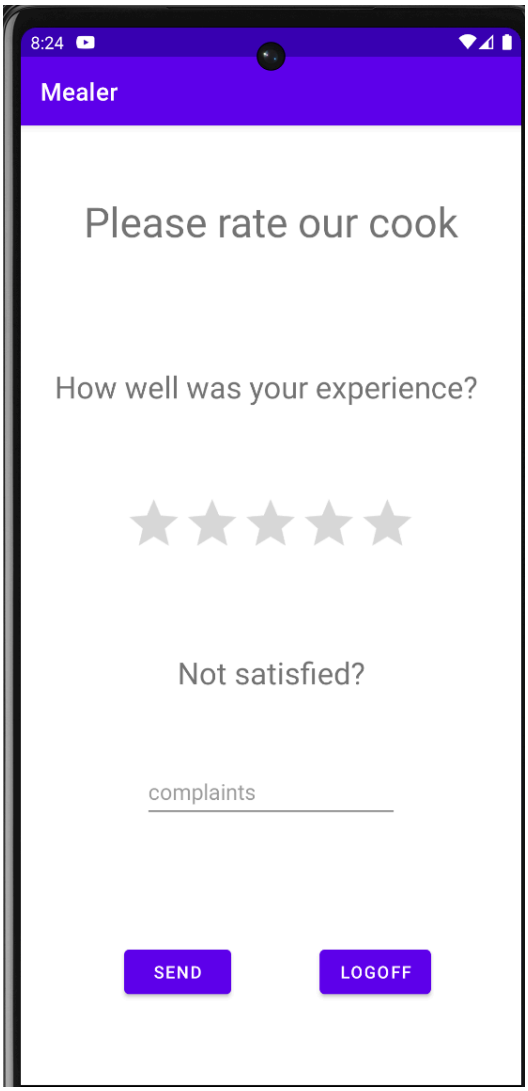


Figure 11 Rate Cook

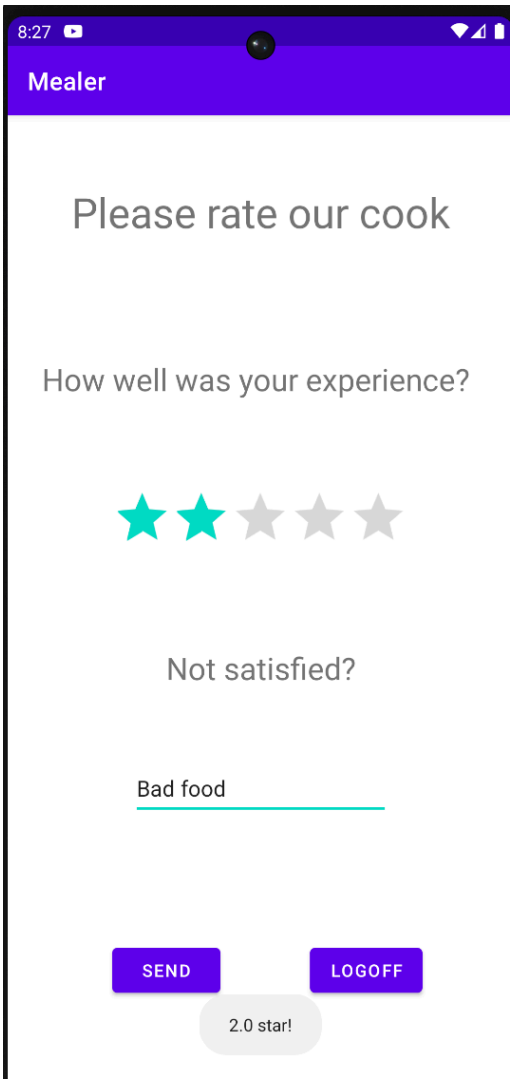


Figure 12 Complaint

After entering the rating activity, the client can use the rating bar to rate the cook then log off or can choose to send a complaint to the cook.

Cook

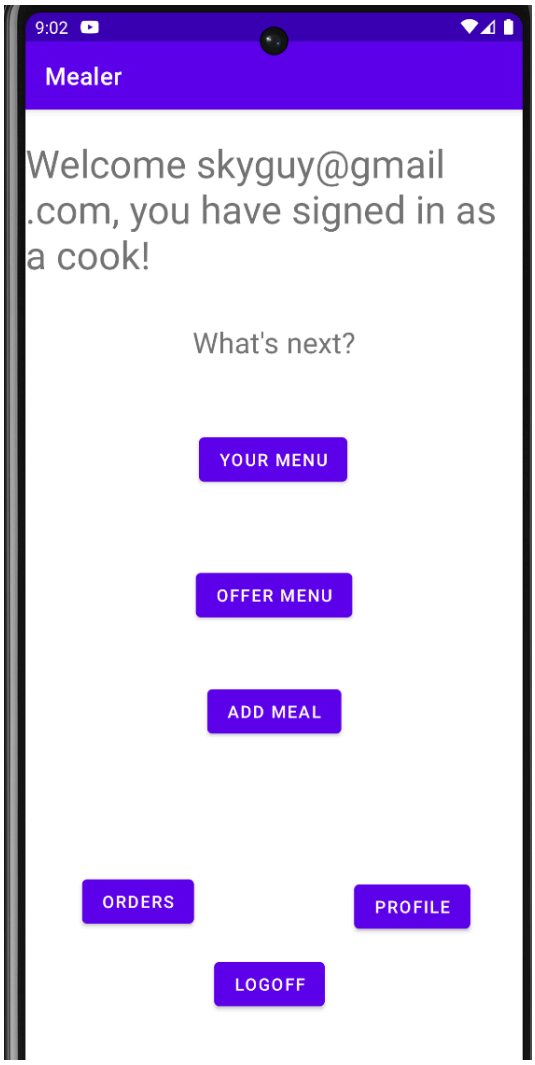


Figure 13 Cook UI

In the cook interface as figure 13 represents, a cook can check their regular menu (available to do but not necessarily offered), offer menu, order from clients, profile and add meal option.

Regular Menu

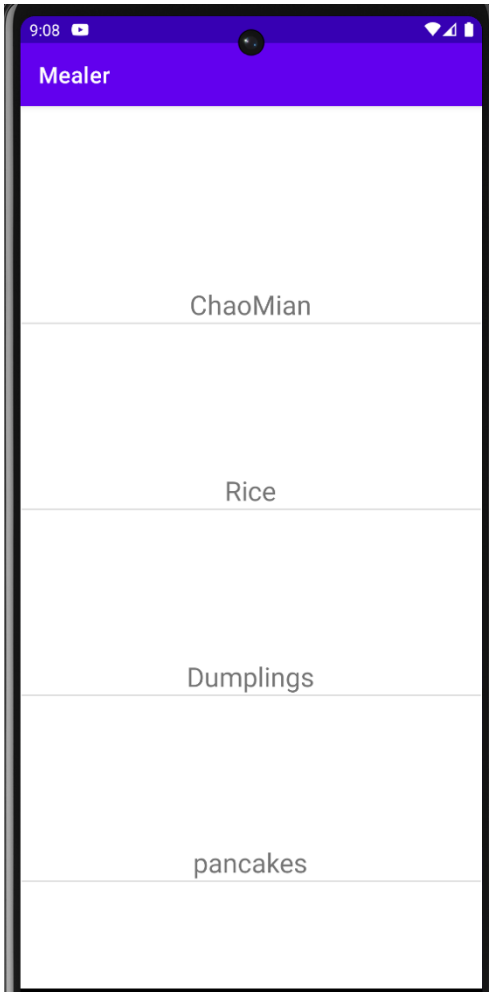


Figure 14 Cook Menu

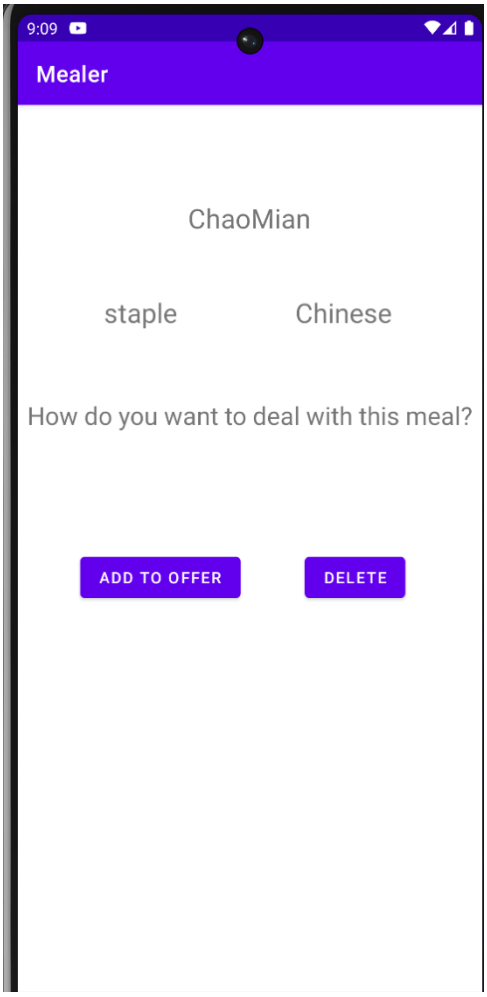


Figure 15 Menu Option

The regular menu contains a list view of available which the cook adds to the menu, by clicking one of the meals presented in figure 14 will show the meal's information, s figure 15 shows. The cook can view the meal's name, type, and cuisine type to decide whether offer this meal to the client or delete it from the menu.

Offered Menu

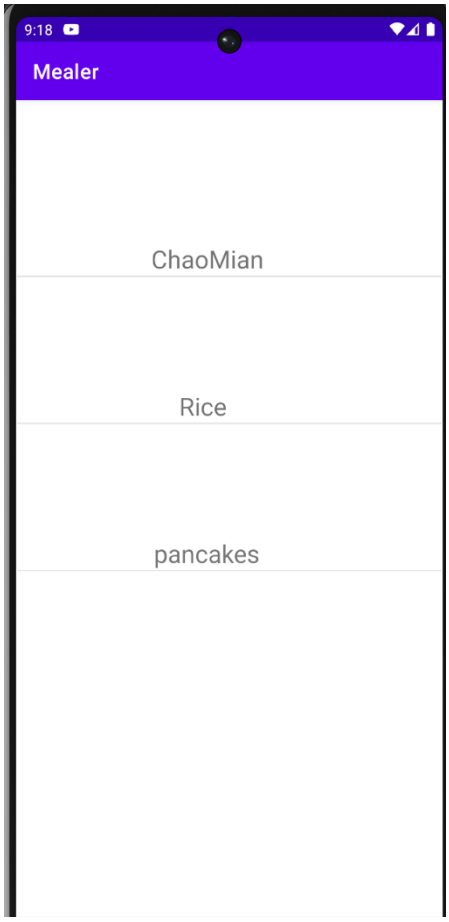


Figure 16 Offered Menu

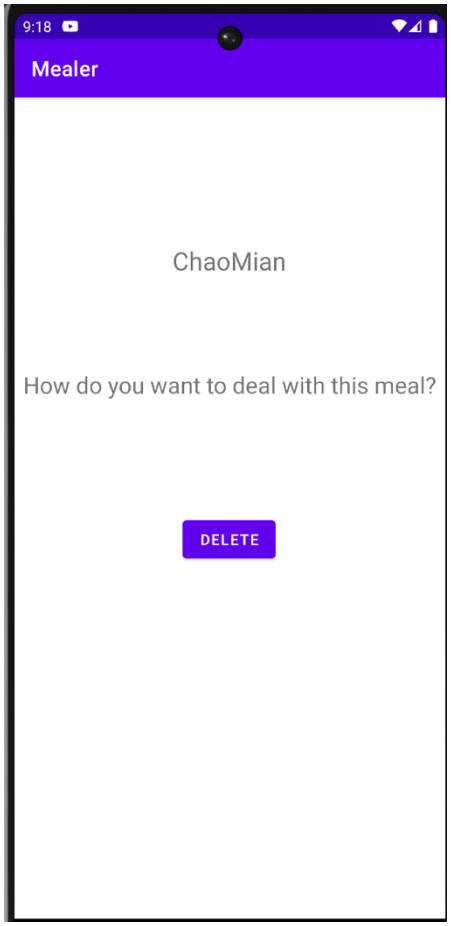


Figure 17 Offered Meal

In the offered menu, as figure 16 represents, the cook can choose to delete the meal, otherwise the clients can order the meal from their search bars.

Add Meal

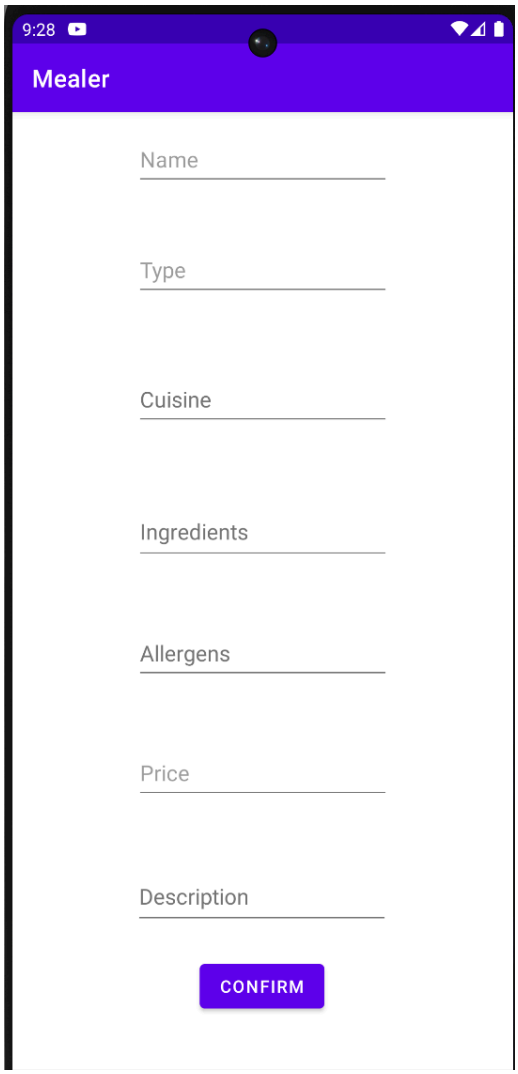


Figure 18 Add Meal Function

To add a meal, the cook needs to specify the meal name, type, cuisine, ingredients, allergens, price and a brief description, for the sake of convenient search by clients.

Order List

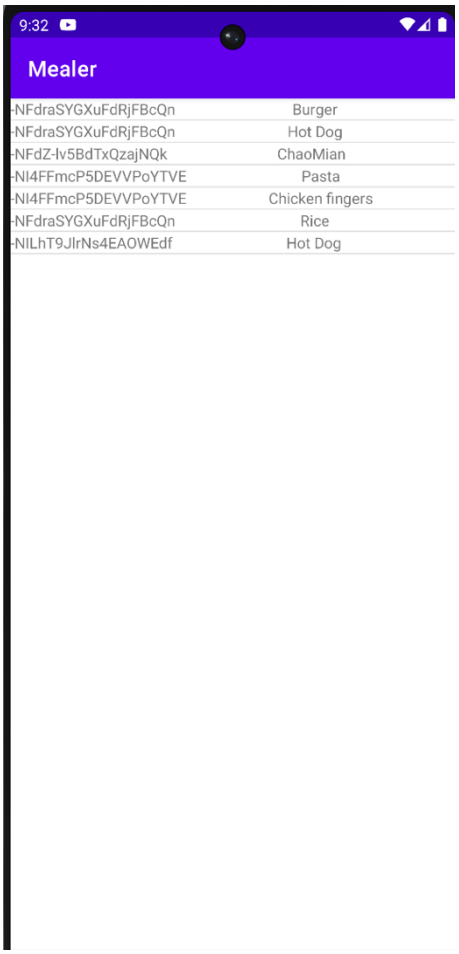


Figure 19 Client Orders

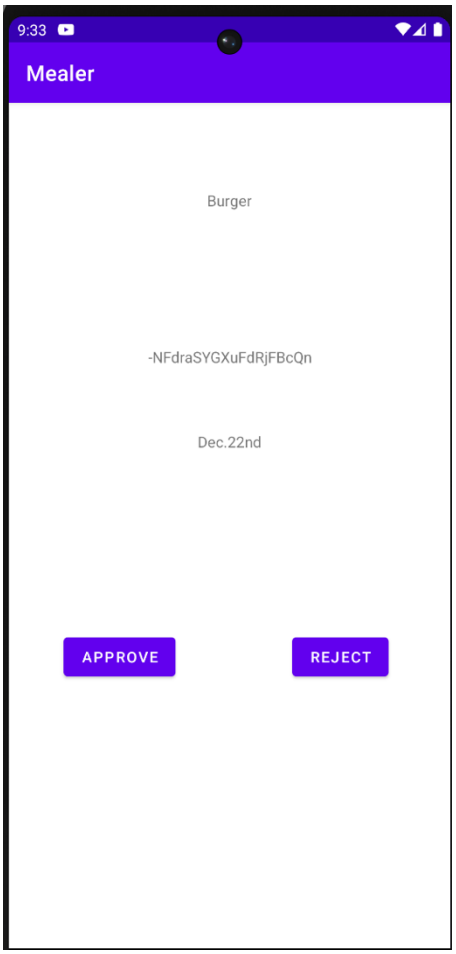


Figure 20 Handle Order

The cook can view all the orders purchased bought by the clients. Figure 15 represents the layout of each purchase request. The client id is on the left, and on the right is the real name. The cook has the option to approve or reject the order. The client can only rate the cook if the meal status is approved. Figure 20 also presents the reject button, which can cancel the meal order, which means that the client needs to repurchase the meal.

Rating

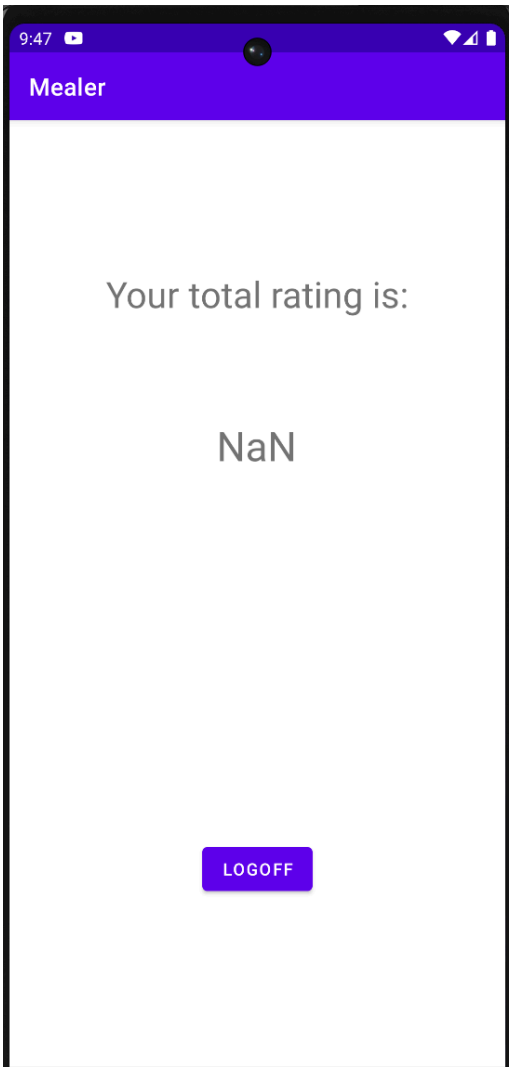


Figure 21 Rating Result 1

Figure 22 Rating Result 2

The cook can review his rating by clicking the profile button. If no clients rated the cook yet, the text view will display NaN as figure 20 presents. Otherwise, the number displayed will derived from the average of all ratings, as shown in figure 22.

Admin

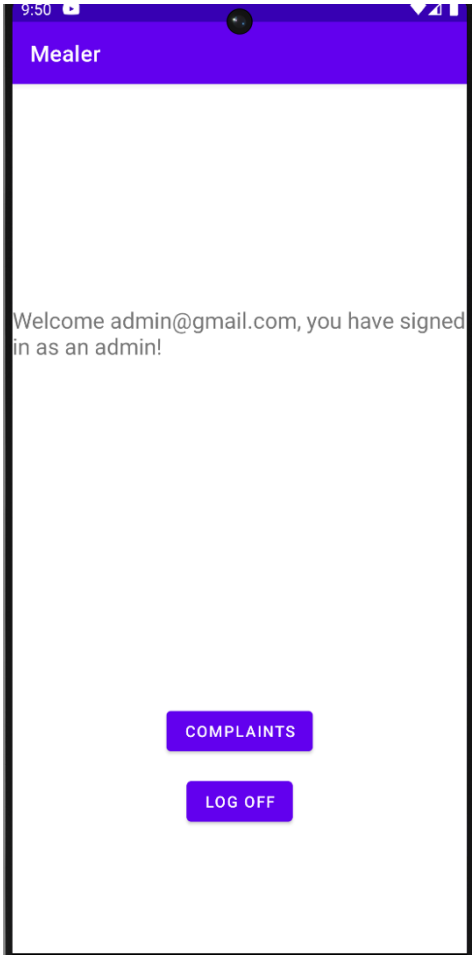


Figure 23 Admin UI

Figure 24 Complaints

Admin only has one function: suspend a cook temporarily or permanently, as shown in figure 22. Admin can view all the complaints sent by the clients. Figure 23 shows an example. In the list view, the client text view represents the client id to the cook id below.

Suspend Temp & Dismiss

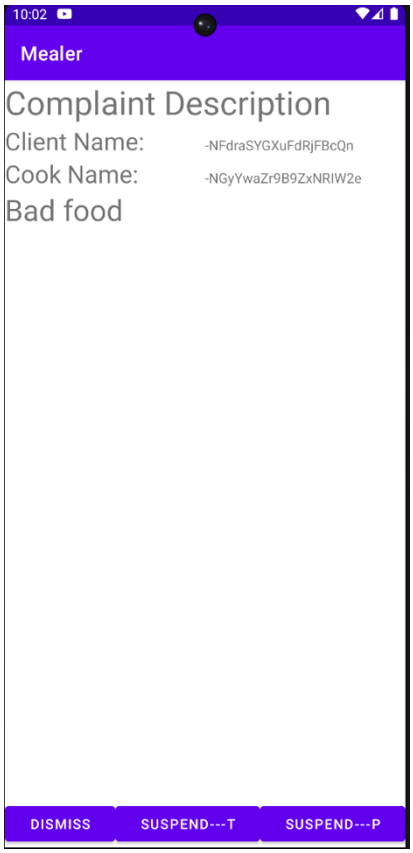


Figure 25 Suspend Options

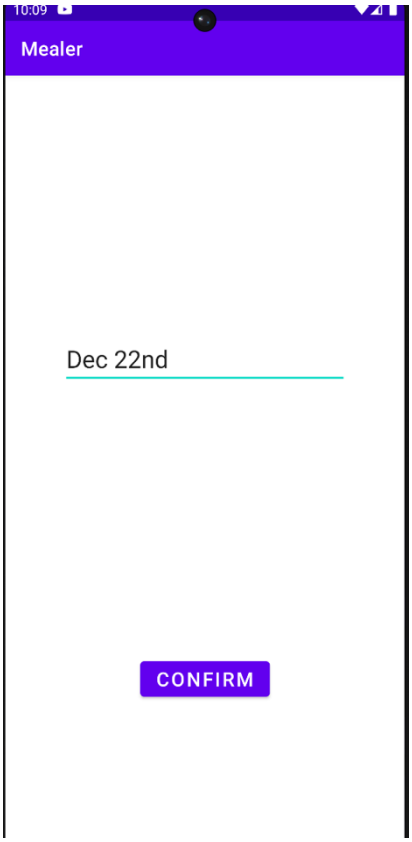


Figure 26 Suspend Date

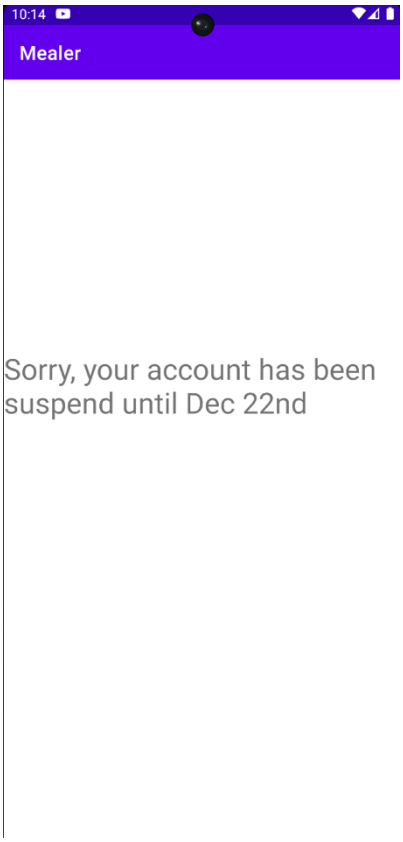


Figure 27 Suspend

In the complaint description section, shown in figure 25, the fourth row displays the complaint message sent by the client. If the admin clicks the dismiss button, this complaint will disappear from the database.

If the admin chooses to suspend the cook temporarily, this cook will be able to log in at the time specified by the cook in figure 26. The cook log-in results can be seen in figure 27.

Suspend Perm

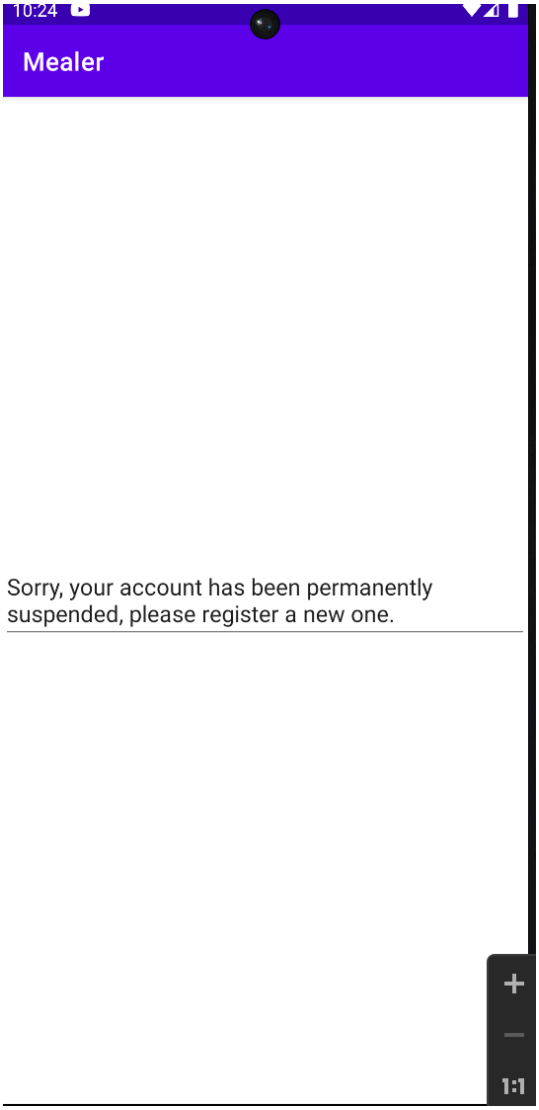


Figure 28 Suspend Perm

However, if the cook is banned permanently, a concrete message will display after the cook logged in, as shown in the figure 28.

5.0 Lessons Learned

Android Studio Technique. The experience where our group learned how to use IDE---Android Studio to build real-life useable software is priceless. The procedure of building an app, the brainstorming of a question, and the Ui design significantly helped us use the OOP concept learned from last semester. We value this experience and agree that this is an excellent practice and preparation for further programming and networking skills.

Firebase Real-Time Database. Using firebase's real-time database is the most helpful skill throughout the course. We will demonstrate our knowledge of the real-time database.

```
DatabaseReference userReference;
userReference = FirebaseDatabase.getInstance().getReference();
```

Figure 29 Database Initiation

```
clientReference = FirebaseDatabase.getInstance().getReference( path: "Clients");
```

Figure 30 Client Reference

Figure 29 represents the initiation of a firebase instance. Since the firebase does not allow a "new" operation, we must call the "getInstance()" method, then specify the reference path by calling "getReference()" method. So far, the userReference now refers to the entire database to which the current project is connected. Therefore, when we need to create a path, for instance, a registered user needs to specify a path so that the path can be created, as figure 30 shows.

Firebase Data Snapshot. After we get the reference path in the database, we can start creating the clients by a technique called data snapshot.

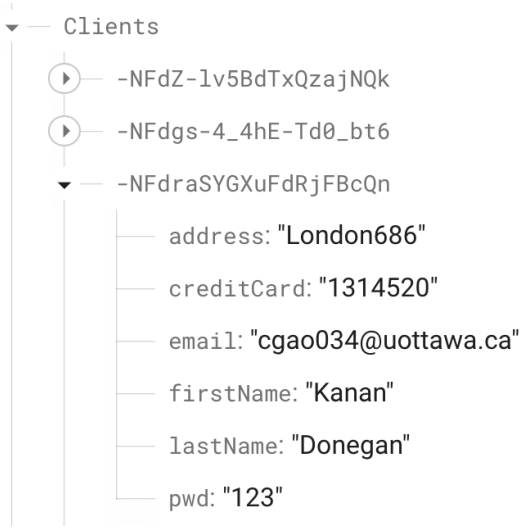


Figure 31 Data Snapshot

```
// listen for new change in Clients
clientReference.addValueEventListener(new ValueEventListener() {
    @Override
    public void onDataChange(@NonNull DataSnapshot snapshot) {
        clientAccounts.clear();
        for(DataSnapshot child : snapshot.getChildren()){
            Account client = child.getValue(Client.class);
            clientAccounts.add((Client) client);
        }
    }

    @Override
    public void onCancelled(@NonNull DatabaseError error) {

    }
});
```

Figure 32 onDataChange Method

A data snapshot is a key value pair with only one key but can have multiple values. From figure 31, the data snapshot is the key-value pair with key equals to “Clients,” value being “-NFdZ…” and the rest of the hash values.

In order to add the value to the data snapshot, we need to use the “addValueEventListner,” and use the “onDataChange()” methods. Figure 32 is an example. Since we are creating a reference Client, we need to wrap the database values into Java objects by calling the method “child.getValue()”. The child is the type “DataSnapshot,” which is created by using “snapshot.getChildren()”, thus, the child now refers to all the hash values (“-NFdZ…”), and each child is a data snapshot too.

Since we do not want the same account being created twice, we will create a list of Clients every time the “onDataChange()” gets called. We will clear the list so the database only listens to all the values once.

Client Search Function. One of the hardest problems is to let the client search by meal name, type and cuisine in the offered menu. In our case, the offer menu stores many cooks' id, and each cook has their offered meals as shown in figure 33.

Figure 33 Offered Meal

Figure 34 Offered Menu

The challenge is that we need to get the path of "cuisineType", "mealName", and "mealType" stored in the simple adapter and put onto the list view, how to achieve this?

```
offerReference = FirebaseDatabase.getInstance().getReference( path: "Offer");

offerReference.addValueEventListener(new ValueEventListener() {
    @Override
    public void onDataChange(@NonNull DataSnapshot snapshot) {
        offer.clear();
        List<Map<String, String>> list = new LinkedList<>();
        for(DataSnapshot child : snapshot.getChildren()){
            for(DataSnapshot c : child.getChildren()) {   RecklessSnips, 11/
                Map<String, String> map = new HashMap<>();
                Meal meal = c.getValue(Meal.class);
                Objects.requireNonNull(meal).setMealID(c.getKey());
                offer.add(meal);

                // This allow user to search either in name, cuisine, or type
                map.put( k: "mealID", meal.getMealID());
                map.put( k: "mealName", meal.getMealName());
                map.put( k: "mealType", meal.getMealType());
                map.put( k: "cuisineType", meal.getCuisineType());
                map.put( k: "cookID", meal.getCookID());
                list.add(map);
            }
        }
```

Figure 35 Offer Reference

The solution is given in figure 34. First, we locate the reference on "Offer", which is the data snapshot in figure 35, then loop through each child of the snapshot, then use an inner loop and use "child.getChildren()" to get all the meals of which each cook offered, then wrap the values into the Meal class.

Child Path to get Specific value. The last but no least lesson we learned, is to use child() method proficiently. The problem is that to show the rating value on the cook's profile page. We need to keep on track of which cook is logging in and show his or her rating value.

Figure 36 Complaint Reference

The solution is to store the rating information in the Complaints only when the client can only get through the complaint activity to rate. In addition, if the admin sees the "complaint" is not an actual complaint (could be a compliment), the admin can dismiss that message. Therefore, we only need to get the Complaints reference from the database and use the "child()" to specify the path we want to use. Thus, we can have the correct rating information from the client.

Figure 37 Solution

```
arrayList = new ArrayList<>();

complaintsReference = FirebaseDatabase.getInstance().getReference( path: "Complaints");

complaintsReference.addValueEventListener(new ValueEventListener() {
    @Override
    public void onDataChange(@NonNull DataSnapshot snapshot) {
        arrayList.clear();
        for(DataSnapshot child : snapshot.getChildren()){
              System.out.println(child);
            if(((String)child.child("cookID").getValue()).equals(cookID)){
                arrayList.add(Double.valueOf((String) child.child("rating").getValue()));
            }
        }

          add together
        for(int i = 0; i< arrayList.size(); i++){
            sum += arrayList.get(i);
        }
        rating = String.valueOf(sum / arrayList.size());
        rate.setText(rating);
    }
```

From figure 37, we use the path “cookID” to determine the current logging-in cook. Then we access the path “rating” to know the value of the cook’s rating.

In Conclusion, we have learned a lot from this project, and the experience of debugging the above problems are priceless lesson. We will value them throughout our software engineer career.