DineSum

Team: 0xFFFFFFFF

Karen Li, 204564235

Matthew Wong, 704569674

Uday Shankar Alla, 404428077

Stanley Hsu, 704338787

Susan Krkasharian, 104584663

Project Name: DineSum
Team Name: 0xFFFFFFFF
Team Members:
Karen Li, 204564235
Matthew Wong, 704569674
Uday Shankar Alla, 404428077
Stanley Hsu, 704338787
Susan Krkasharian, 104584663
Repository URL: https://github.com/codeKaren/DineSum
Demo Video URL: https://youtu.be/ALPsFErc8BA

Motivation

Many popular restaurants do not offer reservations, so diners must go in person and enter a waiting queue before being seated. However, a diner would be able to save a lot of waiting time if he or she was able to have someone else write their name down and make an informal “reservation” before going to the restaurant.

Thus, DineSum attempts to target this scenario by allowing users to make requests for “reservations”, which consists of specifying a restaurant, a time range for the customer to be entered into the waiting queue, and a party name and number. As further incentive, each request is worth a user-specified monetary value, which will be paid from the requester to the reserver after the requester is seated at the restaurant.

The home screen of the application displays a feed of currently pending requests made to restaurants geographically close to the user. As a reserver, the user can claim and complete pending requests. There is currently no other application on the market which uses crowdsourcing to allow diners to make informal “reservations” at restaurants that do not take real reservations. Thus, DineSum will be able to make the pre-dining experience less stressful by helping diners save precious waiting time when going to popular restaurants.

User Benefits

Scenario 1 (Stanley’s Perspective):

1. Stanley and his friends just finished their finals (it’s 6PM) and want to eat at Korea Pizza Company at 8PM in Koreatown but Koreatown does not take reservations.
2. It’s Friday night and anticipating the long waiting hours, Stanley logs into Dinesum and makes a request for a reservation at Korea Pizza Company for anywhere between 7:00 and 8:00 PM for $2.
3. At 7:05 PM he notices that Jim has claimed his request.
4. At 7:10 PM Jim has completed the request and mentions through Facebook that there’s an hour long wait and Stanley should get his table by 8:10 PM.
5. Stanley and his friends then leave their apartment at 7:15 PM to make it in time for their reservation.

6. Stanley reaches Korea Pizza Company by 8:10 PM and is immediately taken to his table.
7. Stanley then pays Jim and enjoys Spicy Kimchi Pizza with his friends.

User benefit from using app: Stanley, the requester, gets to make an informal “reservation” at Korea Pizza Company so he doesn’t need to wait in line for an extremely long time on a busy Friday night. He would rather pay a few dollars than wait for more than two hours just to get seated at a restaurant. Thus, Dinesum saves user’s waiting time if they want to go to a busy restaurant that doesn’t take reservations.

Scenario 2 (Jim’s Perspective):
1. Jim is in Koreatown and his wife Karen is shopping.
2. He is bored and decides to go on a stroll while his wife shops.
3. He realizes he is in a busy area with lots of restaurants around and so decides to open Dinesum to see if there are any requests so he can make easy money while his wife is busy shopping.
4. He notices Stanley’s request for Korea Pizza Company which is only 4 blocks away from him.
5. He claims and completes the request.
6. He goes on to complete Susan’s, Matt’s and Uday’s requests (who have all requested reservations in nearby restaurants) as well.
7. After conveying this information to each user, he goes backs to his wife to enjoy the rest of the night. He ended up making $10 that night. His wife was really impressed!

User benefit from using app: Jim, the reserver, is already in the area nearby the requests created by Stanley, Susan, Matt, and Uday. Thus, it is very easy for him to fulfill their requests and he gets to make some money by fulfilling the requests, a win win situation for Jim. Thus, Dinesum not only benefits users who want to avoid long wait times at restaurants, but also allows incentives other users to claim requests by rewarding them with monetary compensation.

Feature Description and Functional Requirements

Our application has five main features: the sent requests page (requests the user has made), the claimed requests page (requests the user has claimed), the nearby request feed (requests for restaurants in the same city as the user), the new request creation page, and the user scoring system.

We changed our UI layout by having four tabs on the bottom navigation bar for each of the main features. Previously in part A, we had three tabs on the bottom navigation bar for the new request creation page, nearby request feed, and the user profile request history page. The user profile request history page contained both the list of sent requests and the list of claimed requests, so we have now split that up into the sent requests page and the claimed requests page.

The user scoring system was not proposed in part A but was added after feedback from other groups stated that having a scoring system would help keep users accountable for their actions, such as discouraging claiming of requests and then unclaiming them.

The application can only be used after the user has logged in using his or her Facebook account. The picture used as proof of “reservation” and request payment will be coordinated between the requester and reserver externally through Facebook messenger, since users will have access to each other’s Facebook profiles after Facebook log-in.

Sent Requests Page

All of the requests that the user has made will be displayed in the sent requests page. The requests are sorted based on original request creation time, with the most recently created requests on the top. While on the sent requests page, the user is acting in the role of the “requester”. If a user clicks on a request, they will be able to see more information about the request and perform different actions on the request depending on the request’s state. If a request is pending, the user can remove the request. If a request is claimed, the reserver’s Facebook profile picture, which is also a link to their Facebook profile, will be displayed along with their user rating. The user can mark the claimed request as pending if they want to return if to the nearby request feed to be claimed by someone else or they can mark the request as completed if the reserver has successfully added them to the waitlist at the restaurant.

Claimed Requests Page

All of the requests that the user has claimed will be displayed in the claimed requests page. The requests are sorted based on the original request creation time, with the most recently created requests on the top. While on the claimed requests page, the user is acting in the role of the “reserver”. If a user clicks on a request, they will be able to see more information about the request and perform different actions on the request depending on the request’s state. If a request is claimed, the user can unclaim the request to set the request as pending again and return the request to the nearby request feed. If the request is completed, the reserver can mark the request as paid to indicate that they have received payment from the requester.

Nearby Request Feed

The nearby request feed contains the list of nearby pending requests, which are requests for today whose end times have not past the current time that are located in the same city as the user. On the request feed, the user can see the restaurant information and the amount the requester is willing to pay for a reservation. The user can then select one of these requests to see more detailed information about the restaurant, request, and the requester. If the user does not want to claim the request, the back button can be used to return to the list of nearby pending requests.

After the user chooses to claim the request, the user still has the option to unclaim the request in their claimed requests page. Otherwise, once the request is claimed, the reserver must make the reservation and send a picture of the restaurant's waitlist or a physical confirmation of the reservation to the requester through Facebook messenger. After the requester is seated at the restaurant, the user will receive payment from the requester and must confirm receipt of payment by marking the request as paid through the claimed requests page.

New Request Creation Page

As the requester, the user can make "reservation" requests for restaurants which don't take real reservations. Each "reservation" request must contain the following requester-specified information: a restaurant, a time range for the customer to be entered into the waiting queue, a party name, the number of party members, and the amount of money they will pay to whomever completes the request. After creating the request, the user will be able to see the request and its status and information on his or her sent requests page. A user can only create requests for the current day.

User Scoring System

The user scoring system is intended to reward users who claim and complete requests as well as users who create requests and mark them as paid at the end of the request flow. When a user first logs into the application through Facebook, their user points are initialized to 50. At the end of a request flow when a request is marked as paid, the reserver gets +10 points and the requester gets +5 points. To discourage bad behaviour such as claiming requests that you do not intend on completing, a reserver gets -5 points when they unclaim a request.

When a user clicks on a request for more information on the sent requests page, claimed requests page, or the request feed, the user score for the requester or the reserver will be displayed depending on if the request is an claimed or sent request. Thus, the user scoring system will keep users accountable for their actions since they will be able to see other user's scores before claiming pending requests or when checking to see who has accepted their claimed request. For example, if the user sees on their sent requests page that the person who claimed their request has a low user score, they can choose to mark that request as unclaimed so someone else can claim it from the nearby request feed.

Non-Functional Requirements Description

Non-Functional Requirement	Dinesum's Performance
Availability	Our database and data syncing across applications is always available since it is hosted on Firebase realtime database. When users go offline, the Realtime Database

	SDKs use local cache on the device to serve and store changes. When the device comes online, the local data is automatically synchronized.
Security	We used Facebook authentication for users in order to securely keep track of user information. Moreover, Firebase realtime database security rules make sure that read and writes can only be made by authenticated users, making sure that our database is secure.
Modifiability	Our use of the information hiding principle in our application makes it very easy to make changes to our application. For example, since we encapsulated all of our database interactions inside of the FirebaseManager class, it wouldn't be hard to change the underlying database for our application since we would just have to create a different database interaction class and would probably only take a few days to implement.
Testability	The back-end of our application is modular and has good encapsulation, which makes it easy to implement JUnit tests. Furthermore, our UI is straightforward, modular, and has a clean design, which makes it easy to implement instrumented integration tests that need to interact with the UI.
Useability	Our UI is simple with descriptive titles and labels for the UI elements that the user can interact with. Moreover, the flow of the application is intuitive and makes it easy for users to understand how to perform the different actions required in the request flow. The average user will be able to learn how to completely use our app in less than 5 minutes.
Performance	In terms of database response time, Firebase is able to update all of the data needed to be synced across all applications running Dinesum in real-time, within milliseconds, due to the properties of the Firebase database

	itself.

Design

We modified our UML domain-level and UI-level class diagrams, which are shown below:

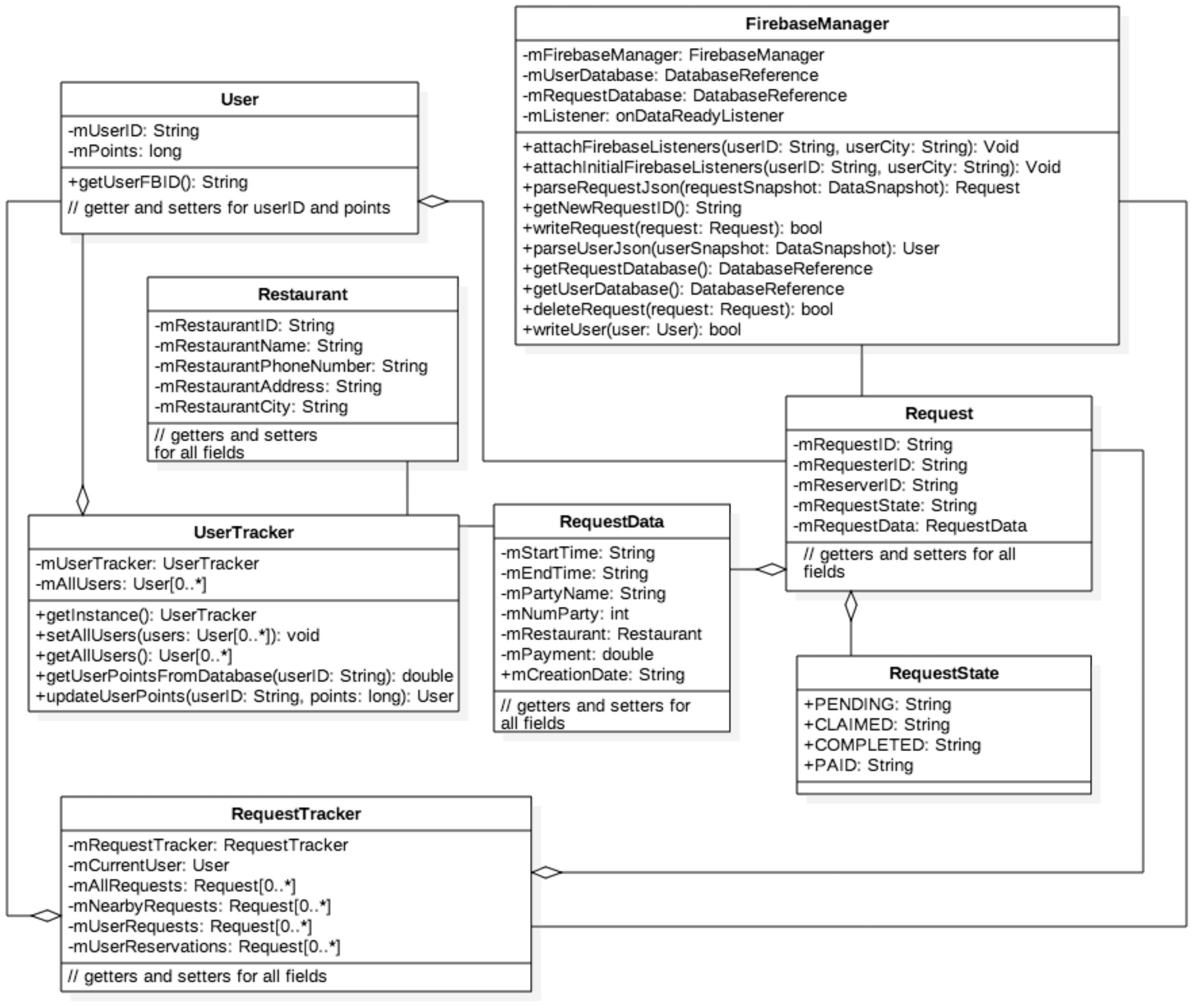

FirebaseManager
-mFirebaseManager: FirebaseManager
-mUserDatabase: DatabaseReference
-mRequestDatabase: DatabaseReference
-mListener: onDataReadyListener
+attachFirebaseListeners(userID: String, userCity: String): Void
+attachInitialFirebaseListeners(userID: String, userCity: String): Void
+parseRequestJson(requestSnapshot: DataSnapshot): Request
+getNewRequestID(): String
+writeRequest(request: Request): bool
+parseUserJson(userSnapshot: DataSnapshot): User
+getRequestDatabase(): DatabaseReference
+getUserDatabase(): DatabaseReference
+deleteRequest(request: Request): bool
+writeUser(user: User): bool
User
-mUserID: String
-mPoints: long
+getUserFBID(): String
// getter and setters for userID and points
Restaurant
-mRestaurantID: String
-mRestaurantName: String
-mRestaurantPhoneNumber: String
-mRestaurantAddress: String
-mRestaurantCity: String
// getters and setters for all fields
UserTracker
-mUserTracker: UserTracker
-mAllUsers: User[0..*]
+getInstance(): UserTracker
+setAllUsers(users: User[0..*]): void
+getAllUsers(): User[0..*]
+getUserPointsFromDatabase(userID: String): double
+updateUserPoints(userID: String, points: long): User
RequestData
-mStartTime: String
-mEndTime: String
-mPartyName: String
-mNumParty: int
-mRestaurant: Restaurant
-mPayment: double
+mCreationDate: String
// getters and setters for all fields
Request
-mRequestID: String
-mRequesterID: String
-mReserverID: String
-mRequestState: String
-mRequestData: RequestData
// getters and setters for all fields
RequestState
+PENDING: String
+CLAIMED: String
+COMPLETED: String
+PAID: String
RequestTracker
-mRequestTracker: RequestTracker
-mCurrentUser: User
-mAllRequests: Request[0..*]
-mNearbyRequests: Request[0..*]
-mUserRequests: Request[0..*]
-mUserReservations: Request[0..*]
// getters and setters for all fields

Figure 1: Updated UML Domain-Level Class Diagram

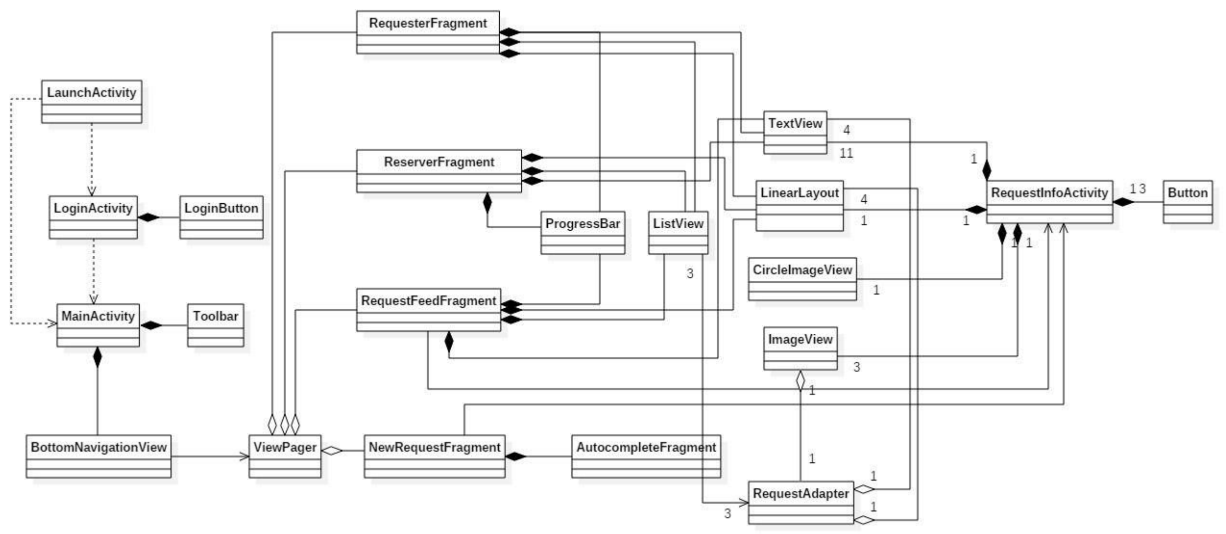


Figure 2: Updated UML UI-Level Class Diagram

In terms of external APIs, we continued to use Facebook Android SDK, Firebase Realtime Database, Google Places API, and JUnit Test Framework, which were described in our Part B Report.

For the final stages of our project, we also incorporated the Espresso test library for Android and the UI Automator testing framework when implementing our integration tests, which were instrumentation tests. Espresso is a testing library that can be used for both whitebox and blackbox testing. We used Espresso for white box testing to test all of the different request actions depending on the request state. Using Espresso, we were able to use the IDs of UI elements in our front-end to select UI elements and then perform actions on them such as clicking buttons or typing text into input boxes. For example, we created new requests and then made sure that deleting requests, claiming requests, unclaiming requests, completing requests, and marking requests as paid all worked correctly. Thus, we were able to test our whole application from the UI layer all the way down to the database since we had to verify that our requests were correctly written to the database and returned back from the database to the UI afterwards.

Similarly, UI Automator was also used to help with integration and UI testing. UI Automator was particularly useful in some cases since UI Automator doesn’t require you to have the ID of the UI element that you want to interact with. Instead, UI Automator scans the currently displayed view on the Android device to find UI elements to interact with. Therefore, we use the UI Automator to interact with UI elements such as widgets like the TimePicker and Google Places Autocomplete Search Fragment since we don’t have the IDs from the source code for those elements.

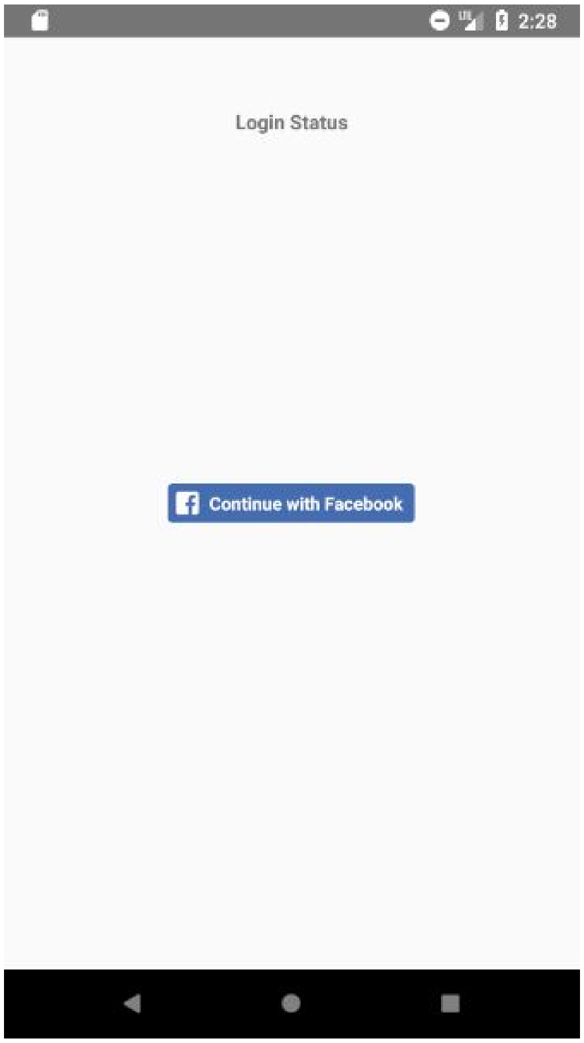
API Description

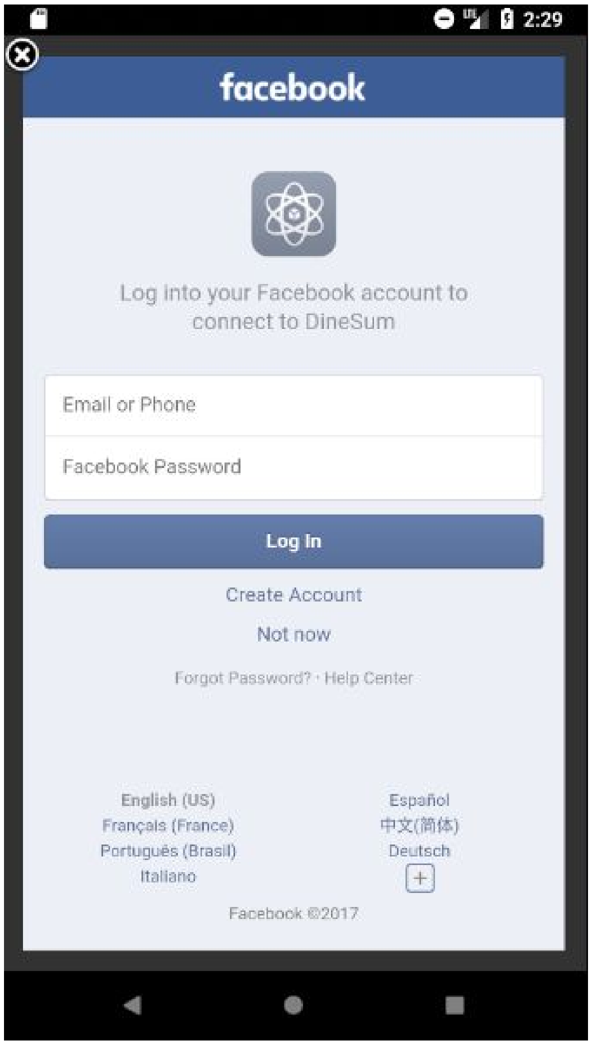
Since we are developing an Android application, we were able to use JavaDoc to automatically generate documentation for our project. You can view our JavaDoc by cloning our Github repository from the link below and opening the “DineSumJavaDoc/index.html” file.

JavaDoc repository link: https://github.com/skrkasharian/DineSumJavaDoc

User Interface Snapshots

When the user first opens the app, they will be initially shown the Facebook log-in page, seen on the left. After logging in through the Facebook authentication pop-up seen on the right, the user will not have to login again when starting up the app again next time because the account login is stored locally on the device.



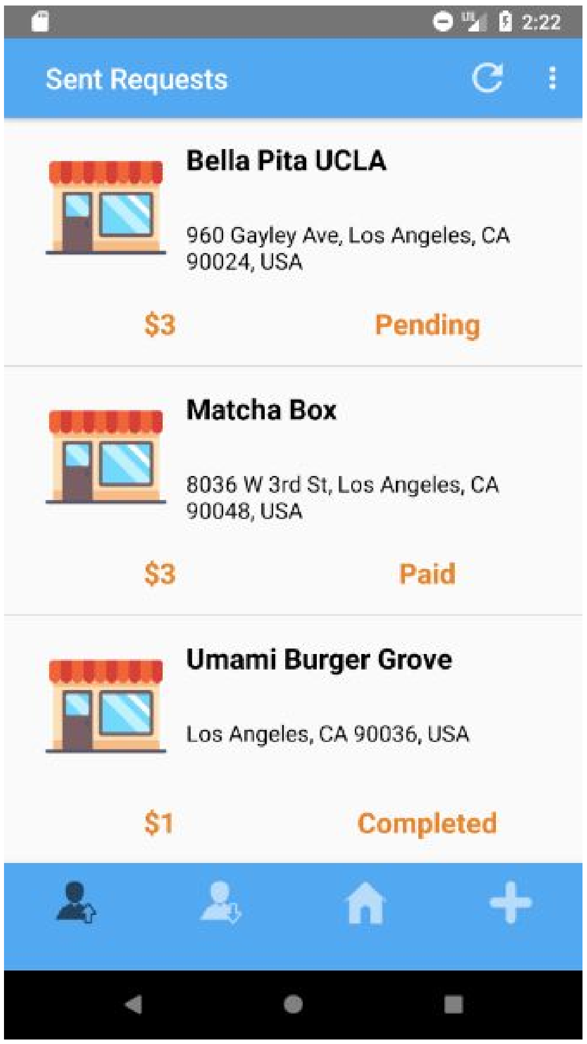


Figures 3 & 4: Initial Login Page and Facebook Authentication Page

As shown on the bottom of the screen, we use an extension of the BottomNavigationView to navigate around the app. The user can either swipe or tap on the icons to switch to different pages. The landing one is the sent requests page, seen on the left.

The sent requests page displays all of the requests that the user has created. The requests are sorted based on original request creation time, with the most recently created requests on the top. The user can press the refresh button to refresh the display of sent requests if the database has updated.

If a user clicks on a pending request from the sent requests page, they are shown the screen on the right. This screen shows more information regarding the restaurant and the request details. The user can delete this request by clicking the "remove request" button.

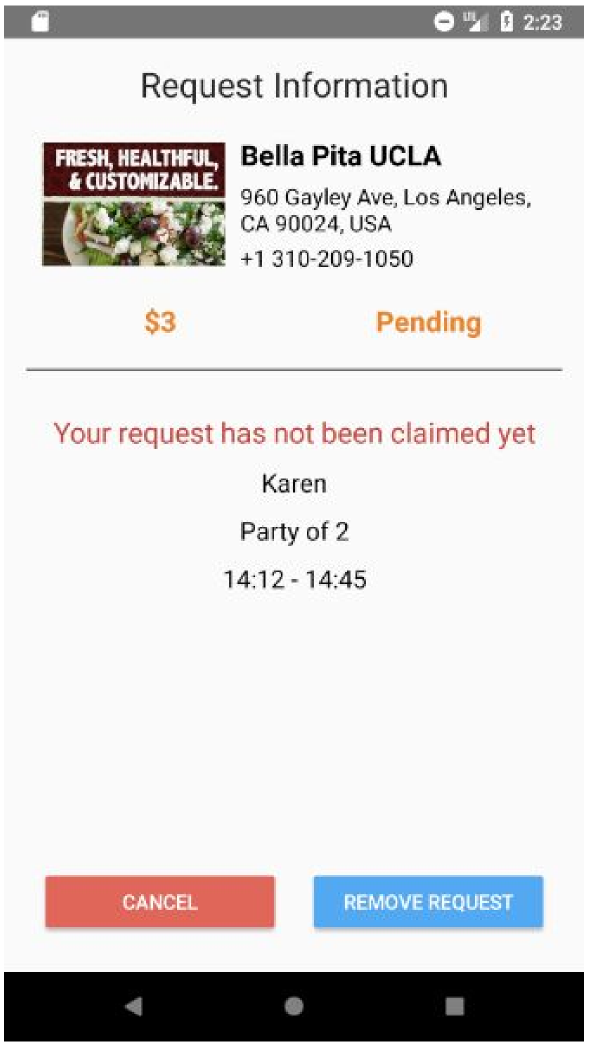

2:22

Sent Requests

Bella Pita UCLA
960 Gayley Ave, Los Angeles, CA 90024, USA
$3 Pending

Matcha Box
8036 W 3rd St, Los Angeles, CA 90048, USA
$3 Paid

Umami Burger Grove
Los Angeles, CA 90036, USA
$1 Completed


2:23

Request Information

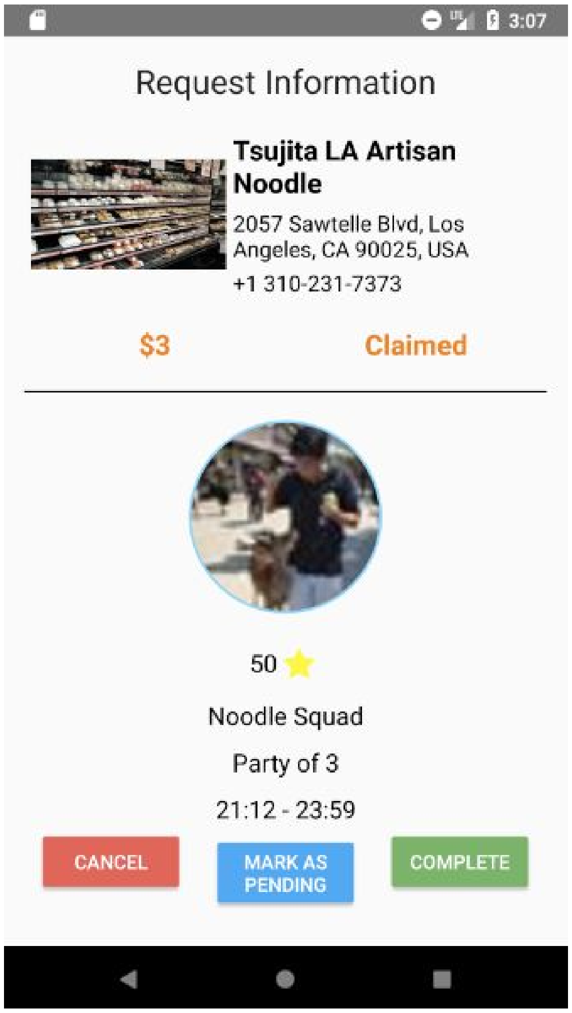
Bella Pita UCLA
960 Gayley Ave, Los Angeles, CA 90024, USA
+1 310-209-1050
$3 Pending

Your request has not been claimed yet
Karen
Party of 2
14:12 - 14:45

Figures 5 & 6: Sent Requests Page and Pending Request Info Page

If a user clicks on a claimed request from the sent requests page, they are shown the screen below. The page displays more information regarding the restaurant and the request details. Moreover, since the request has been claimed, the page contains the Facebook profile picture for the reserver, which is also a link to the reserver's Facebook profile. Thus, the user (requester) can click on the picture to communicate with the person who has claimed their request to coordinate reservation information and payments. The screen also displays the reserver's user score generated by our scoring system.

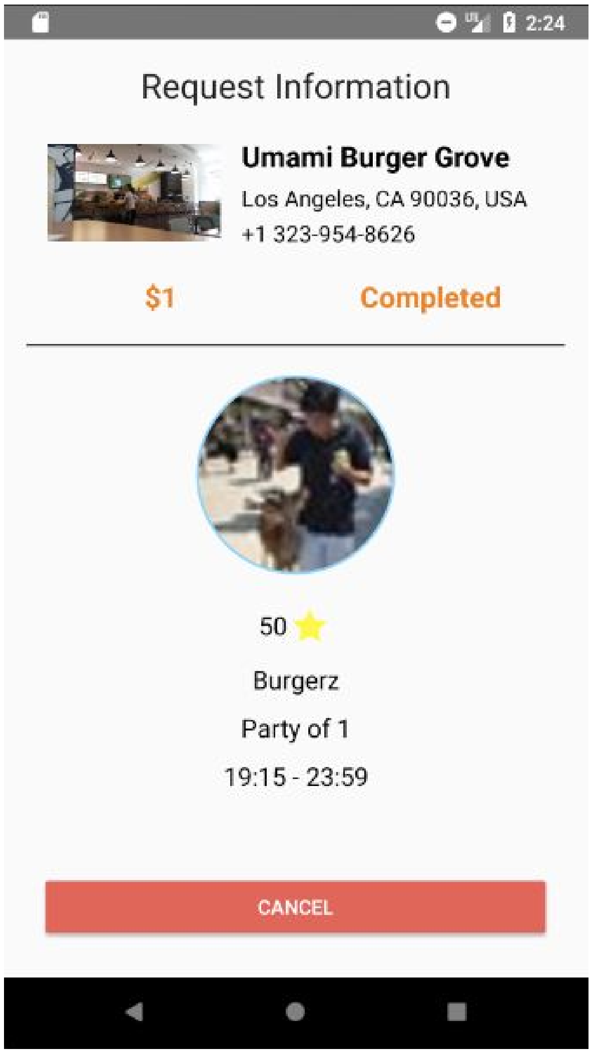
The user can change this request back to pending state and send it back to the nearby request feed to be claimed by someone else using the "mark as pending" button. Moreover, the user can use the "complete" button to mark the request as completed if they finished adding the requester to the waitlist for the restaurant.



Figures 7: Claimed Request Info Page

If a user clicks on a completed request from the sent requests page, they are shown the screen on the left. If a user clicks on a paid request from the sent requests page, they are shown the screen on the right.

Both pages show more information regarding the restaurant and the request details, the Facebook profile picture and link for the reserver, and the reserver's user score.



Request Information

Umami Burger Grove

Los Angeles, CA 90036, USA

+1 323-954-8626

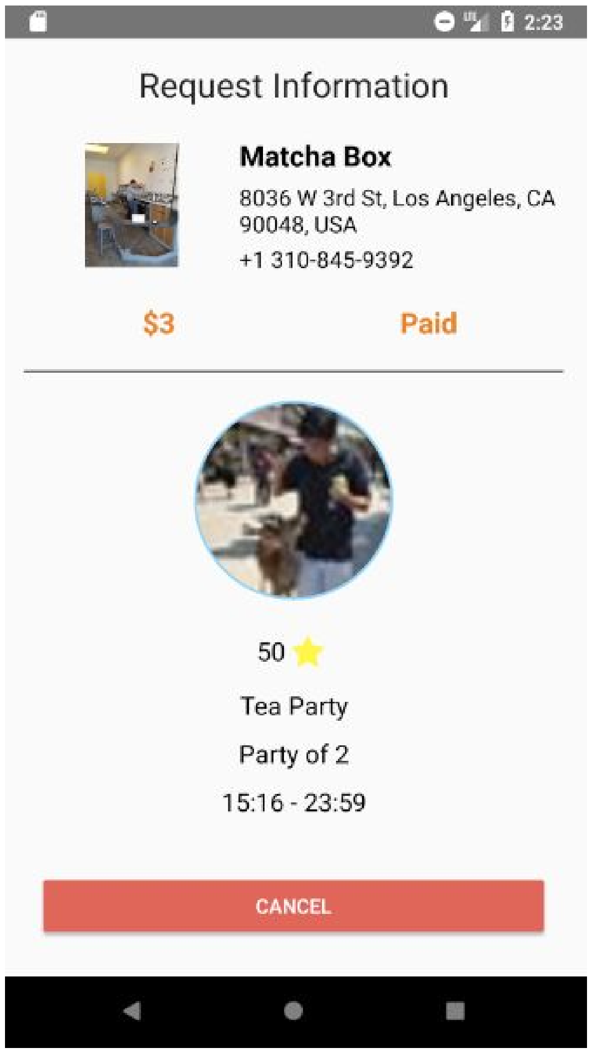
$1 Completed

50

Burgerz

Party of 1

19:15 - 23:59



Request Information

Matcha Box

8036 W 3rd St, Los Angeles, CA 90048, USA

+1 310-845-9392

$3 Paid

50

Tea Party

Party of 2

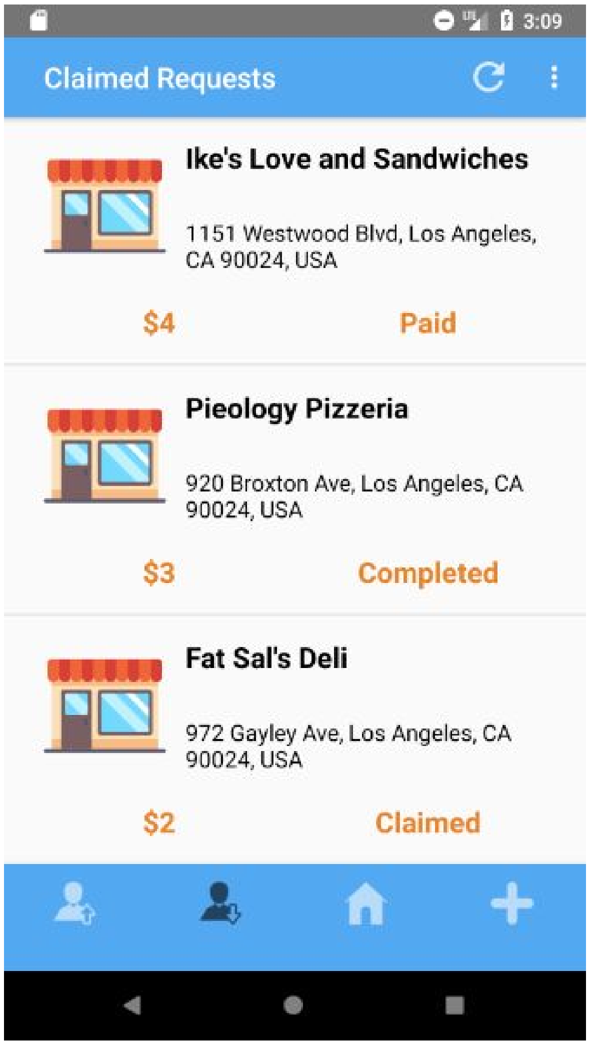
15:16 - 23:59

Figures 8 & 9: Completed Request Info Page and Paid Request Info Page

If the user clicks on the second icon from the left, the sent requests page displays all of the requests that the user has claimed. The requests are sorted based on original request creation time, with the most recently created requests on the top. The user can press the refresh button to refresh the display of claimed requests if the database has updated.

If a user clicks on a claimed request from the claimed requests page, they are shown the screen on the right. This screen shows more information regarding the restaurant and the request details. The user can unclaim this request and send it back to the nearby requests feed so someone else can claim it by clicking the “unclaim request” button.

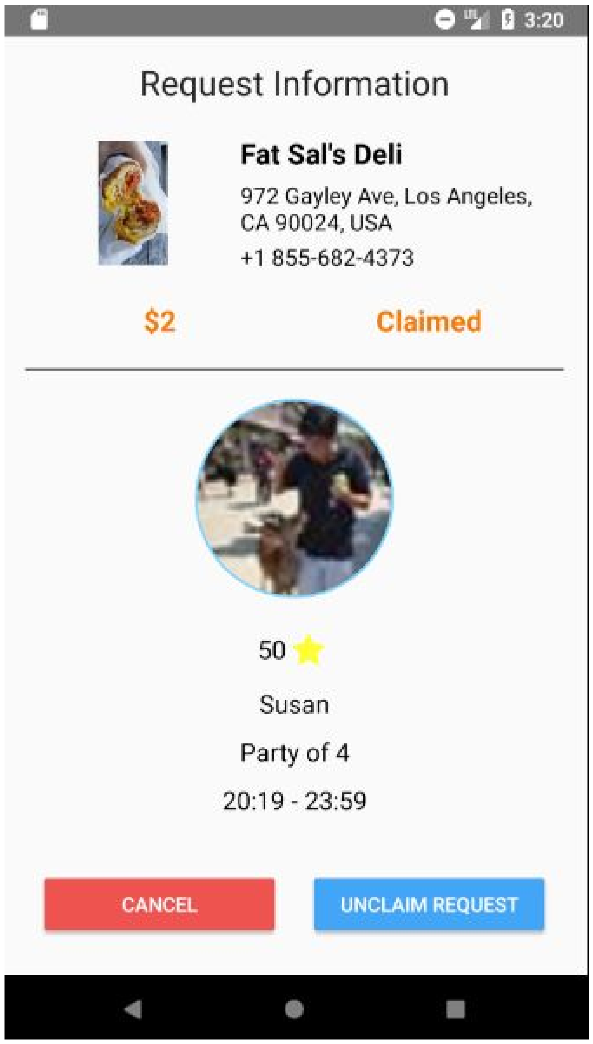
The Facebook profile picture and link is for the requester who originally created the request. Through clicking it, the user (reserver) can communicate with the person whose request they claimed. The screen also displays the requester’s user score generated by our scoring system.



Ike's Love and Sandwiches
1151 Westwood Blvd, Los Angeles, CA 90024, USA
$4 Paid

Pieology Pizzeria
920 Broxton Ave, Los Angeles, CA 90024, USA
$3 Completed

Fat Sal's Deli
972 Gayley Ave, Los Angeles, CA 90024, USA
$2 Claimed



Request Information

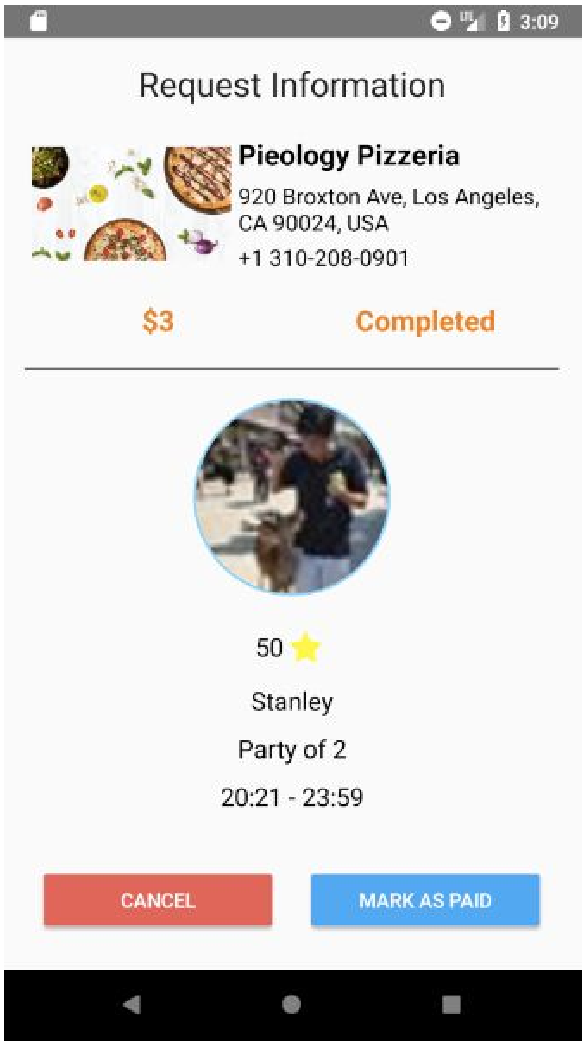
Fat Sal's Deli
972 Gayley Ave, Los Angeles, CA 90024, USA
+1 855-682-4373
$2 Claimed

50
Susan
Party of 4
20:19 - 23:59

Figures 10 & 11: Claimed Request Page and Claimed Request Info Page

If a user clicks on a completed request from the sent requests page, they are shown the screen on the left. If a user clicks on a paid request from the sent requests page, they are shown the screen on the right.

Both pages show more information regarding the restaurant and the request details, the Facebook profile picture and link for the requester, and the requester's user score.



3:09

Request Information

Pieology Pizzeria
920 Broxton Ave, Los Angeles, CA 90024, USA
+1 310-208-0901

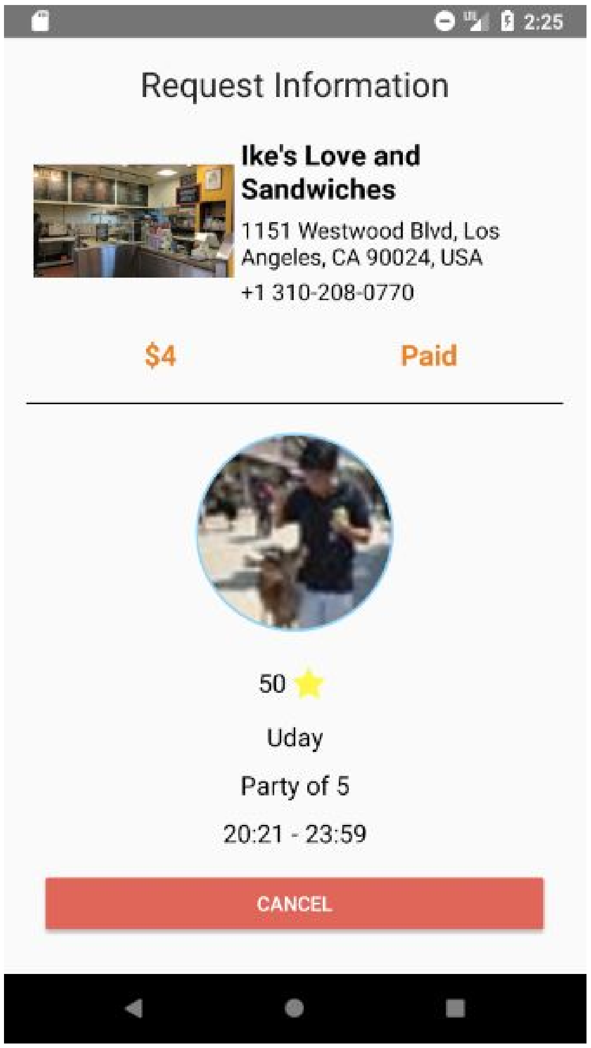
$3 Completed

50

Stanley

Party of 2

20:21 - 23:59



2:25

Request Information

Ike's Love and Sandwiches
1151 Westwood Blvd, Los Angeles, CA 90024, USA
+1 310-208-0770

$4 Paid

50

Uday

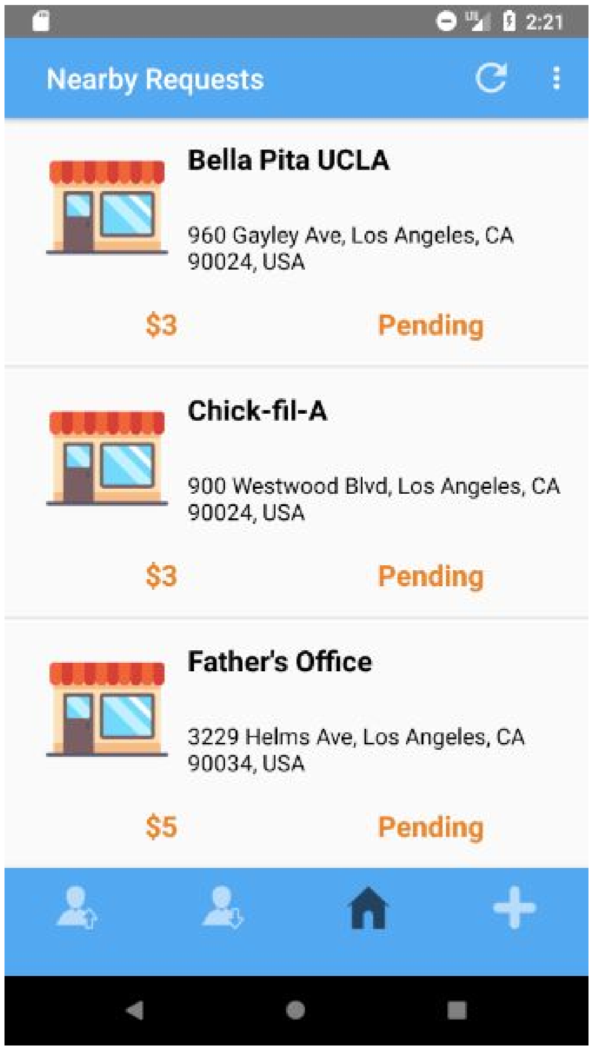
Party of 5

20:21 - 23:59

Figures 12 & 13: Completed Request Page and Paid Request Info Page

If the user clicks on the second icon from the right, the nearby request feed displays the list of nearby pending requests, which are requests for today whose end times have not past the current time that are located in the same city as the user. On the request feed, the user can see the restaurant information and the amount the requester is willing to pay for a reservation.

The requests are sorted based on original request creation time, with the most recently created requests on the top. The user can press the refresh button to refresh the display of nearby requests if the database has updated.



Bella Pita UCLA
960 Gayley Ave, Los Angeles, CA 90024, USA
$3 Pending

Chick-fil-A
900 Westwood Blvd, Los Angeles, CA 90024, USA
$3 Pending

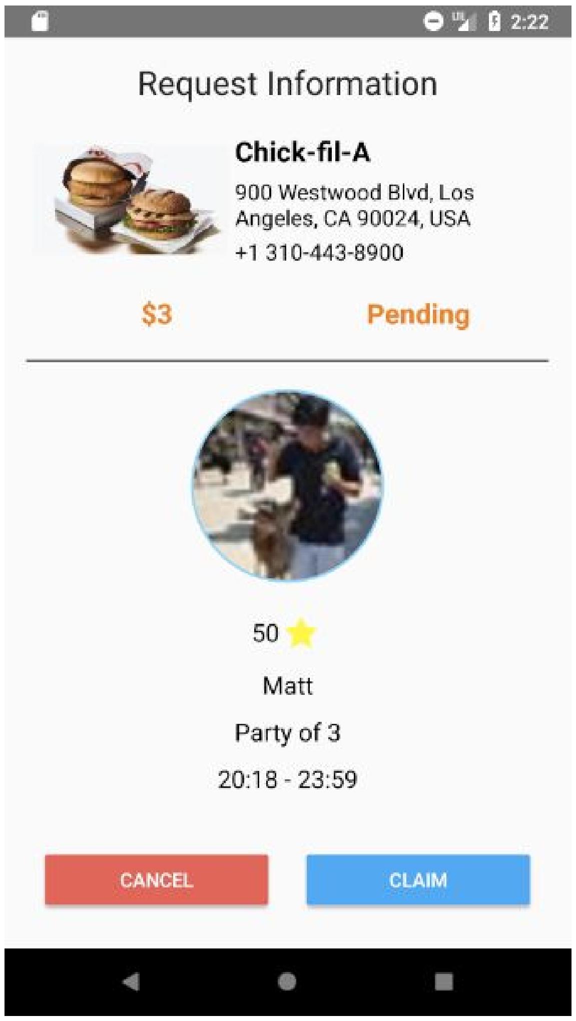
Father's Office
3229 Helms Ave, Los Angeles, CA 90034, USA
$5 Pending

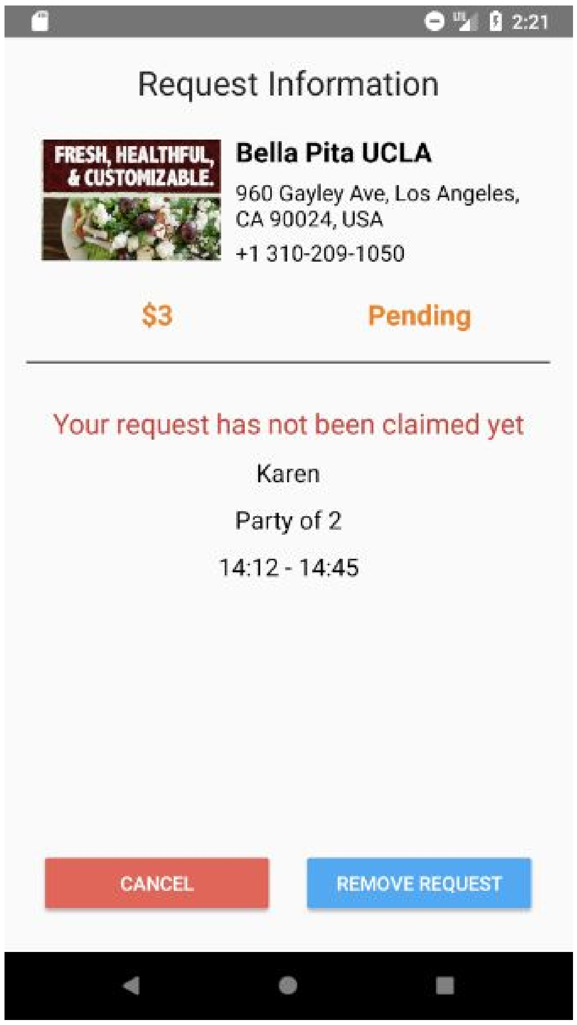
Figure 14: Nearby Request Feed

If a user clicks on a pending request that they did not create from the nearby requests page, they are shown the screen on the left. The Facebook profile picture and link is for the requester who originally created the request. The screen also displays the requester's user score generated by our scoring system.

If a user clicks on a pending request that they created from the nearby requests page, they are shown the screen on the right.

Both pages show more information regarding the restaurant and the request details.

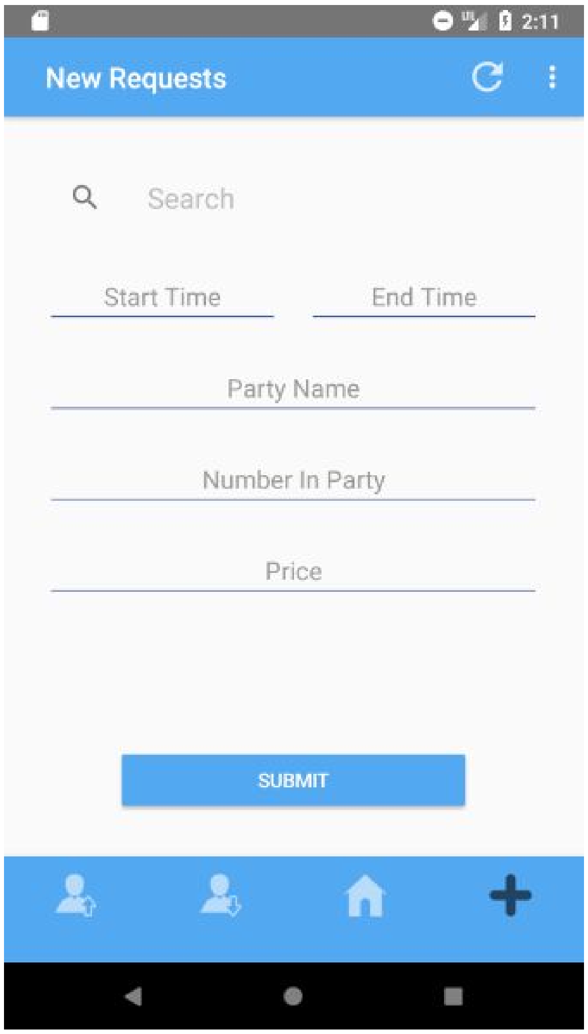


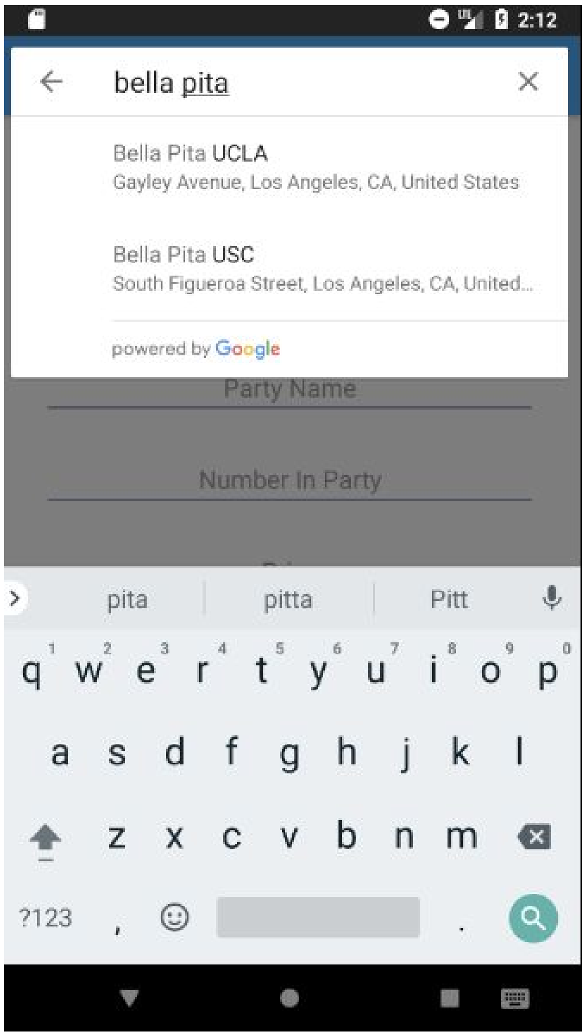


2:21
Request Information
FRESH, HEALTHFUL, & CUSTOMIZABLE.
Bella Pita UCLA
960 Gayley Ave, Los Angeles, CA 90024, USA
+1 310-209-1050
$3
Pending
Your request has not been claimed yet
Karen
Party of 2
14:12 - 14:45
CANCEL
REMOVE REQUEST

Figures 15 & 16: Pending Request Info Page (Other User Created and Self Created)

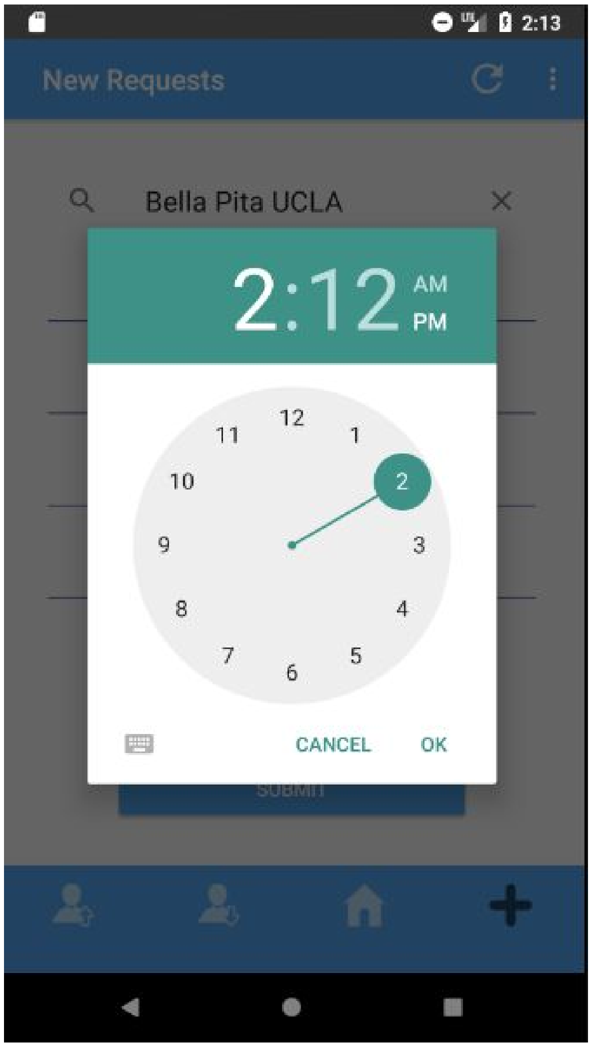
On the bottom tab of the screen, the user can click the “plus” button on the right to make a new request. The user needs to use the search box to find the restaurant, and fill out all the necessary information such as the time, the party name, and the number in the party. The search box uses Google Places Autocomplete API so the user can get typeahead suggestions as he types.

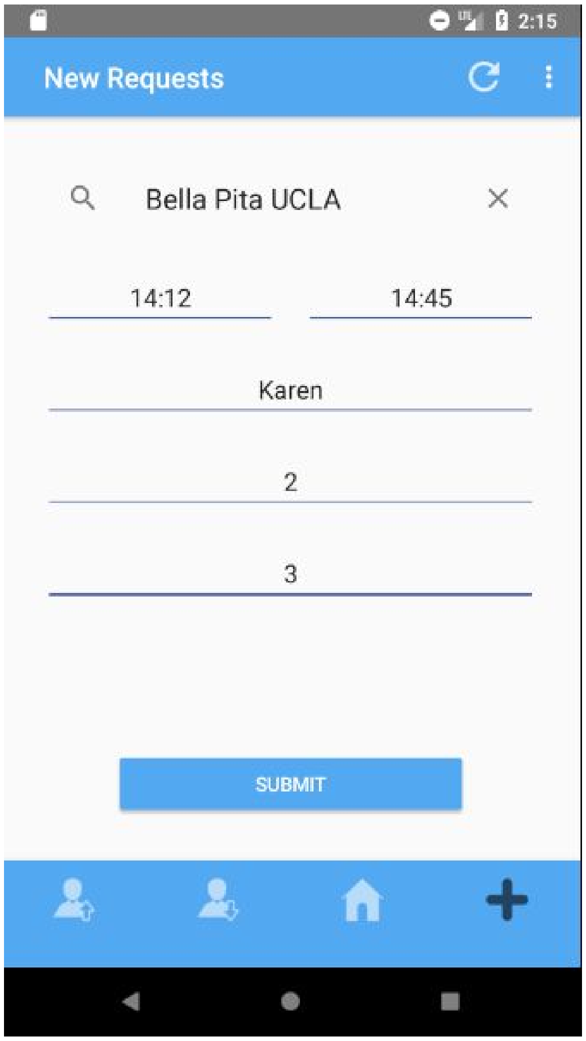




Figures 17 & 18: New Request Form and Google Places Autocomplete Search

The user can fill out the start and end time of the request using the TimePickerDialog, as seen on the left screen below. The user then fills out the rest of the form just using the keyboard, and presses the Submit button to submit the request.





Bella Pita UCLA

14:12 14:45

Karen

2

3

Figures 19 & 20: End-Time Selection Ticker and Completed Request Creation Form

Test Scenarios and Test Cases in JUnit

Integration Testing (Instrumentation Testing) Instructions

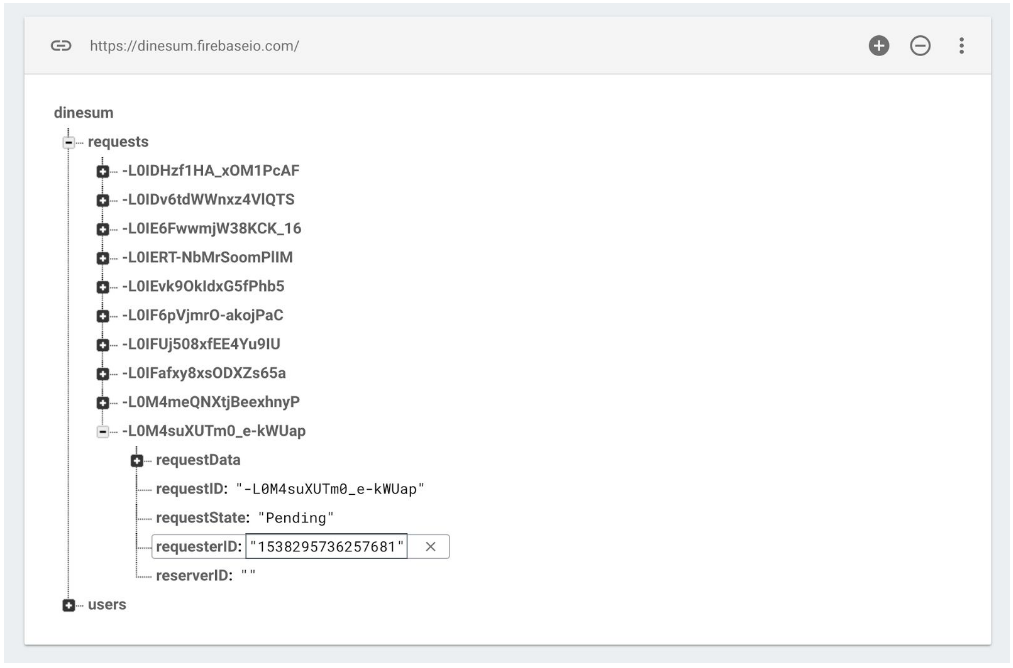
Integration Testing Repository URL:
https://github.com/codeKaren/DineSum/tree/master/app/src/androidTest/java/com/a0xffffffff/dinesum

Integration testing is complicated because our project is a crowd-sourced app. Thus, there are several steps that must be completed in order to properly run the integration tests. Our group has been using Android Studio 3.0 to develop and run the integration tests. One should download the whole project from Github and open it in Android Studio to run the tests and application. The best Android emulator to use is the Nexus 5X with Android 8.0 (Oreo).

First, one needs to be able to access the application's Firebase database through the web interface. The link to the database is:
https://console.firebase.google.com/u/1/project/dinesum/database/dinesum/data

One can log in to the Firebase database using a test account that has been given permission to alter data:
User Email: dinesumtest2@gmail.com
Password: dinesum2!

If one is able to access the Firebase database through the provided URL, one should see the database resembling something similar to the below figure.



```
https://dinesum.firebaseio.com/

dinesum
  requests
    -L0IDHzf1HA_xOM1PcAF
    -L0IDv6tdWWnxz4VlQTS
    -L0IE6FwwmjW38KCK_16
    -L0IERT-NbMrSoomPlIM
    -L0IEvk9OkIdxG5fPhb5
    -L0IF6pVjmrO-akojPaC
    -L0IFUj508xfEE4Yu9IU
    -L0IFafxy8xsODXZs65a
    -L0M4meQNXtjBeexhnyP
    -L0M4suXUTm0_e-kWUap
      requestData
      requestID: "-L0M4suXUTm0_e-kWUap"
      requestState: "Pending"
      requesterID: "1538295736257681"
      reserverID: ""
  users
```

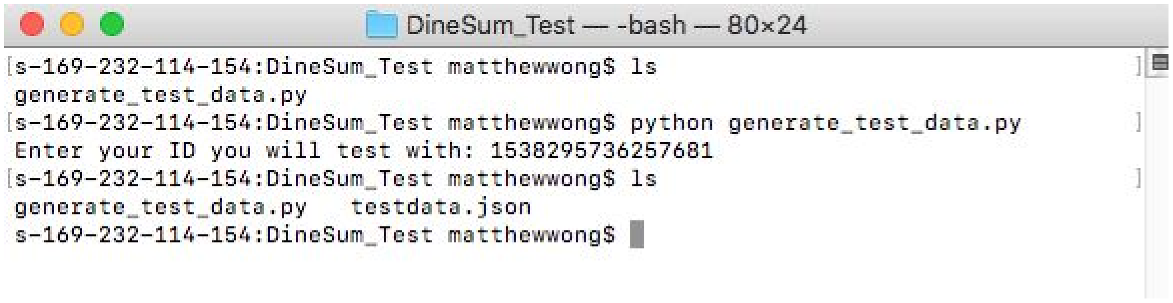
Figure 21: Firebase Database Example with Personal ID

The next step is to obtain a personal ID number. When one first logs in to the application through Facebook, this person's profile is assigned a random ID number that will be used to associate and filter all requests. After obtaining this ID number, one can run the provided Python script (generate_test_data.py) to create a JSON file that contains the appropriate test data to be imported to Firebase.

The easiest way to obtain this ID number is to go through the request creation process in the application and lookup the ID through the Firebase web interface. Once you have created a new request through your profile, the request should appear at the bottom of the requests table. Expand the last request and copy the requesterID field, as seen in the above figure.

(Originally, our group had tried to create test users with which the integration tests could be run. However, this meant that our group also had to create test Facebook profiles for the test users. Unfortunately, Facebook blocked every test Facebook profile our group tried to create.)

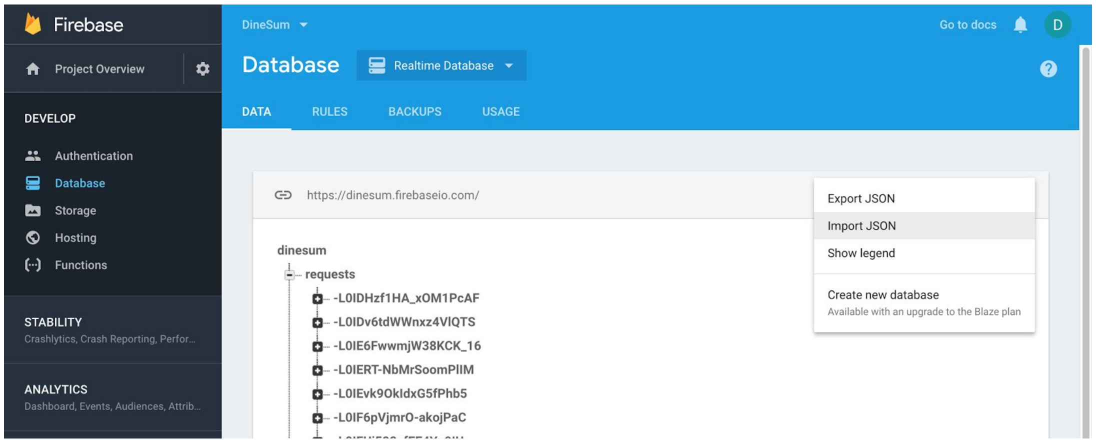
Now that one has obtained the personal ID number, one can generate the test data using the provided Python script. The Python script is called generate_test_data.py, and it is located within the androidTest folder:
https://github.com/codeKaren/DineSum/blob/master/app/src/androidTest/java/com/a0xffffffff/dinesum/generate_test_data.py
One should move the script to a new folder before running it to generate the test data. The script should work with Python 2 and Python 3. When one runs the script, the script will ask for an ID number. One should provide the ID number obtained in the previous step. When the script is finished, there should be a new file testdata.json which contains the test data to be imported to Firebase.



```
s-169-232-114-154:DineSum_Test matthewwong$ ls
generate_test_data.py
s-169-232-114-154:DineSum_Test matthewwong$ python generate_test_data.py
Enter your ID you will test with: 1538295736257681
s-169-232-114-154:DineSum_Test matthewwong$ ls
generate_test_data.py   testdata.json
s-169-232-114-154:DineSum_Test matthewwong$ 
```

Figure 22: Generating Test Data with Python Script and Personal ID

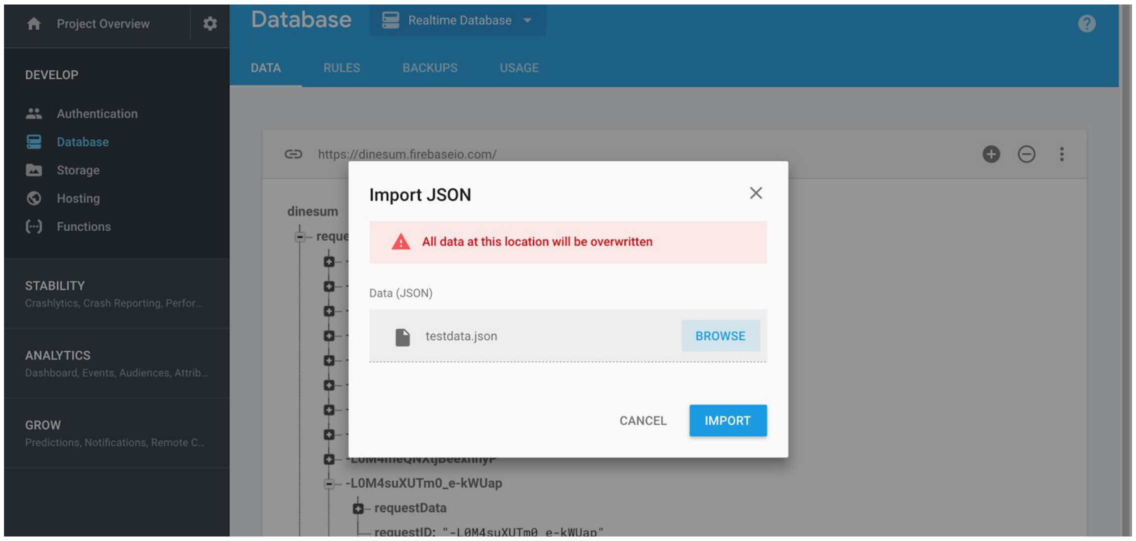
The next step is to import the newly created testdata.json file into the Firebase database through the web interface. This can be easily done by navigating back to the web interface and selecting the option to import a JSON file. Select the correct testdata.json file to import, and one should notice a successful update to the Firebase.



https://dinesum.firebaseio.com/

```
dinesum
  requests
    -L0IDHzf1HA_xOM1PcAF
    -L0IDv6tdWWnxz4VlQTS
    -L0IE6FwwmjW38KCK_16
    -L0IERT-NbMrSoomPlIM
    -L0IEvk9OkIdxG5fPhb5
    -L0IF6pVjmrO-akojPaC
```

Figure 23: Import JSON File into Firebase Database



https://dinesum.firebaseio.com/

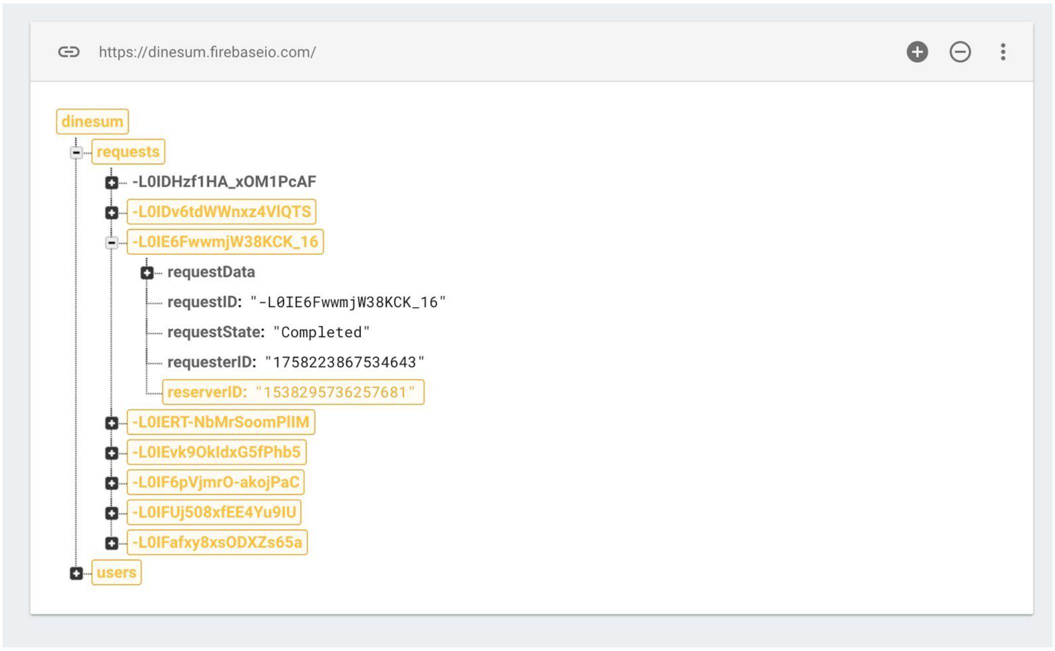
All data at this location will be overwritten

Data (JSON)

testdata.json

```
dinesum
  reque
    -L0M4suXUTm0_e-kWUap
      requestData
      requestID: "-L0M4suXUTm0_e-kWUap"
```

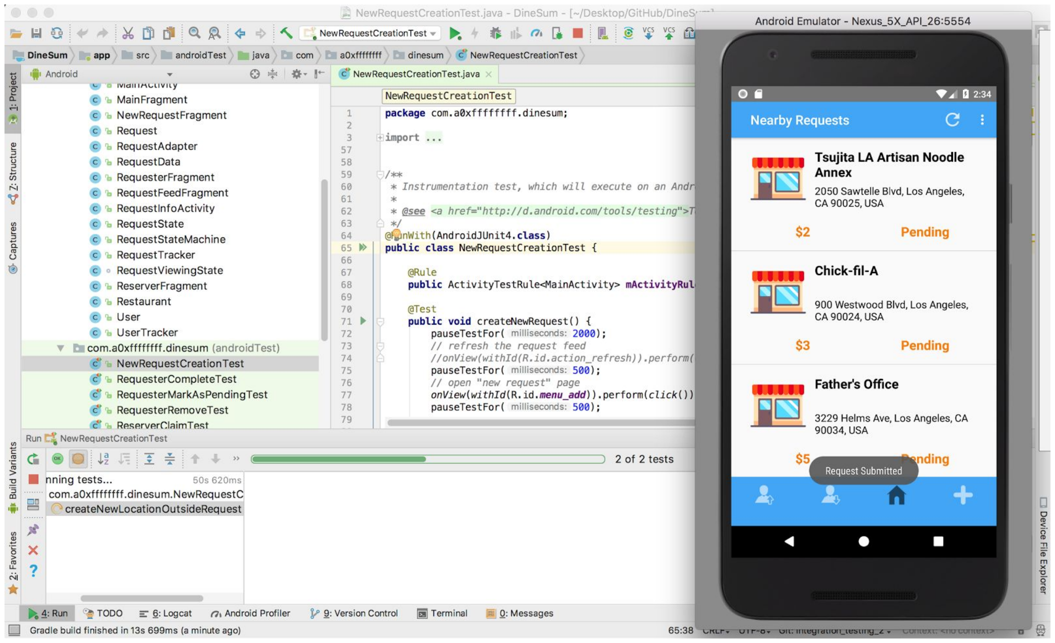
Figure 24: Select testdata.json to Import



```
https://dinesum.firebaseio.com/

dinesum
  requests
    -L0IDHzf1HA_xOM1PcAF
    -L0IDv6tdWWnxz4VIQTS
    -L0IE6FwwmjW38KCK_16
      requestData
      requestID: "-L0IE6FwwmjW38KCK_16"
      requestState: "Completed"
      requesterID: "1758223867534643"
      reserverID: "1538295736257681"
    -L0IERT-NbMrSoomPlIM
    -L0IEvk9OkldxG5fPhb5
    -L0IF6pVjmrO-akojPaC
    -L0IFUj508xfEE4Yu9IU
    -L0IFafxy8xsODXZs65a
  users
```

Figure 25: Updates to Firebase Database are Highlighted

Now that the test data has been imported, one should be able to run the integration tests through Android Studio. Within Android Studio, our project contains an androidTest folder, which holds the several classes containing the JUnit integration tests. One can run the integration tests through the same simulator used to log in to the application. It is important that the simulator being used to run the integration tests has already been logged in to the application, since the Facebook log in process could not be automated with the integration testing.

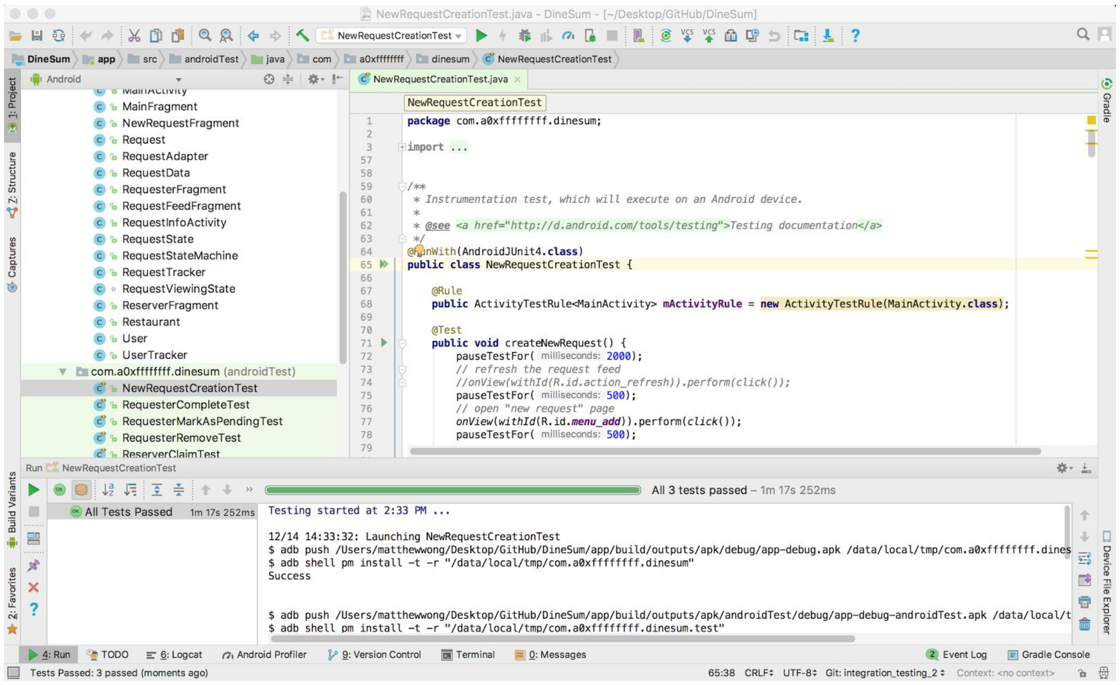


```java
package com.a0xffffffff.dinesum;

import ...


/**
 * Instrumentation test, which will execute on an Andr
 *
 * @see <a href="http://d.android.com/tools/testing">T
 */
@RunWith(AndroidJUnit4.class)
public class NewRequestCreationTest {

    @Rule
    public ActivityTestRule<MainActivity> mActivityRul

    @Test
    public void createNewRequest() {
        pauseTestFor( milliseconds: 2000);
        // refresh the request feed
        //onView(withId(R.id.action_refresh)).perform(
        pauseTestFor( milliseconds: 500);
        // open "new request" page
        onView(withId(R.id.menu_add)).perform(click())
        pauseTestFor( milliseconds: 500);
```

```
nning tests...                          50s 620ms
 com.a0xffffffff.dinesum.NewRequestC
  createNewLocationOutsideRequest
2 of 2 tests
Gradle build finished in 13s 699ms (a minute ago)
```

Android Emulator - Nexus_5X_API_26:5554
Nearby Requests
Tsujita LA Artisan Noodle Annex
2050 Sawtelle Blvd, Los Angeles, CA 90025, USA
$2
Pending
Chick-fil-A
900 Westwood Blvd, Los Angeles, CA 90024, USA
$3
Pending
Father's Office
3229 Helms Ave, Los Angeles, CA 90034, USA
$5
Pending
Request Submitted

Figure 26: Running Integration Tests through Android Studio and Simulator



```java
package com.a0xffffffff.dinesum;

import ...


/**
 * Instrumentation test, which will execute on an Android device.
 *
 * @see <a href="http://d.android.com/tools/testing">Testing documentation</a>
 */
@RunWith(AndroidJUnit4.class)
public class NewRequestCreationTest {

    @Rule
    public ActivityTestRule<MainActivity> mActivityRule = new ActivityTestRule(MainActivity.class);

    @Test
    public void createNewRequest() {
        pauseTestFor( milliseconds: 2000);
        // refresh the request feed
        //onView(withId(R.id.action_refresh)).perform(click());
        pauseTestFor( milliseconds: 500);
        // open "new request" page
        onView(withId(R.id.menu_add)).perform(click());
        pauseTestFor( milliseconds: 500);
```

All 3 tests passed – 1m 17s 252ms

All Tests Passed 1m 17s 252ms

```
Testing started at 2:33 PM ...

12/14 14:33:32: Launching NewRequestCreationTest
$ adb push /Users/matthewwong/Desktop/GitHub/DineSum/app/build/outputs/apk/debug/app-debug.apk /data/local/tmp/com.a0xffffffff.dines
$ adb shell pm install -t -r "/data/local/tmp/com.a0xffffffff.dinesum"
Success


$ adb push /Users/matthewwong/Desktop/GitHub/DineSum/app/build/outputs/apk/androidTest/debug/app-debug-androidTest.apk /data/local/t
$ adb shell pm install -t -r "/data/local/tmp/com.a0xffffffff.dinesum.test"
```

Tests Passed: 3 passed (moments ago)

Figure 27: Notification from Android Studio that Integration Tests Passed

Integration Tests (Instrumentation Tests)

We implemented integration (instrumentation) tests for all of the possible actions in the request flow and encapsulated each of them in a separate classes. We have described out test classes and specific test cases below. These test case descriptions can also be found inside of the README.md found inside our test folder.

Integration (instrumentation) test folder Github link:
https://github.com/codeKaren/DineSum/tree/master/app/src/androidTest/java/com/a0xffffffff/dinesum

NewRequestCreationTest

Instrumentation test for creating new requests.

Test Case	**Test Oracle**
createNewRequest()	Create a valid new request. A new request will be created with no errors and should appear in the Request Feed and User Requests List.
createNewEndTimePastRequest()	Create an invalid new request with an End Time that has already passed. A new request will be created with no errors

	but should not appear in the Request Feed. It should appear User Requests List, and will be deleted after confirming its presence.
createNewLocationOutsideRequest()	Create an invalid new request with a location outside of Los Angeles. A new request will be created with no errors but should not appear in the Request Feed. It should appear User Requests List, and will be deleted after confirming its presence.

RequesterCompleteTest

Instrumentation test for completing a request that you have created as a requester.

Test Case	Test Oracle
completeRequest()	Complete a claimed request. A claimed request will be marked as completed with no errors. The database should be updated and the changes should be reflected in the Requests List.

RequesterMarkAsPendingTest

Instrumentation test for marking requests as "pending" that you have created as a requester that have previously been claimed.

Test Case	Test Oracle
markRequestAsPending()	Mark a claimed request as pending, removing the reserver from the request. A claimed request will be marked as pending with no errors. The database should be updated and the changes should be reflected in the Requests List. The Request Feed should also contain the request again since it has been returned to the pending state.

RequesterRemoveTest

Instrumentation test for removing pending requests that you have created as a requester.

Test Case	Test Oracle
removeRequest()	Remove a pending request from the database. A pending request will be removed with no errors. The database should be updated and the changes should be reflected in the Requests List and Request Feed. The request should no longer appear in either of the two lists.

ReserverClaimTest

Instrumentation test for claiming new requests that are pending from the request feed.

Test Case	Test Oracle
claimRequest()	Claim a pending request from the Request Feed. A pending request will be claimed with no errors. The database should be updated and the changes should be reflected in the Reserver List. The Request Feed should not contain the request.

ReserverMarkAsPaidTest

Instrumentation test for marking completed requests as paid from the claimed requests page.

Test Case	Test Oracle
markRequestAsPaid()	Mark a completed request as paid. A completed request will be marked as paid with no errors. The database should be updated and the changes should be reflected in the Reserver List.

ReserverUnclaimTest

Instrumentation test for unclaiming requests that the user has previously claimed.

Test Case	Test Oracle
unclaimRequest()	Unclaim a request the user has previously claimed. A claimed request will be marked as pending with no errors. The database should be updated and the changes should be reflected in Request Feed. The Request Feed should contain the request again since it has been returned to the pending state.

JUnit Testing

We implemented JUnit test cases for our four main back-end classes: User, UserTracker, Request, and RequestTracker. We have described our test classes and specific test cases below. These test case descriptions can also be found inside of the README.md found inside of our test folder.

Unit test folder Github link:
https://github.com/codeKaren/DineSum/tree/master/app/src/test/java/com/a0xffffffff/dinesum

UserTest

The User class is responsible for containing the User’s ID and score. The functionality of the User class is important for the application’s UI to display scoring.

Test Case	Test Oracle

testUserCreatedSuccessfully()	Checks that the default user instance was created succesfully. Also ensures that the point value of the default user instance is correct
testUserIDSetSuccesfully()	Checks that the user ID of the user is set and retrieved succesfully.
testUserPointsSetSuccesfully()	Checks that the User Points of the user is set and retrieved succesfully.

UserTracker

The UserTracker class stores a list of the users received from the FirebaseManager and keeps track of scoring for users. It is important that UserTracker can be written and read for the application UI to properly display the scoring for each User in the request information page.

Test Case	**Test Oracle**
setUp()	Set up the test Users Lists with new users.
testAllUsersList()	Checks that the AllUsersList is set and retrieved successfully.
testGetUserPoints()	Checks that the points of a specific User ID can be retrieved successfully.
testUpdateUserPoints()	Checks that the points of a specific User ID is updated successfully.
teardown()	Reset the test usersList to an empty list.

RequestTest

Request objects contain all of the information associated with requests and reservations. Our ability to read, write, and copy Request objects is important to the functionality of the application's UI and database.

Test Case	**Test Oracle**
setup()	Initialize request objects.
testRequestCreatedSuccessfully()	This test checks if Request objects are successfully initialized with the expected information. The objects should not be null.
testRequestID()	This test checks if Request objects have been initialized with the correct Request IDs. It also checks if the Request ID string can be written and read in Request objects.

testRequesterID()	This test checks if Request objects have been initialized with the correct Requester IDs. Checks if the Requester ID (the ID of the user who created the Request) can be written and read.
testReserverID()	This test checks if Request objects have been initialized with Reserver IDs set to “” (empty string). Checks if the Reserver ID (the ID of the user who claimed the Request) can be written and read.
testRequestState()	This test checks if Request objects have been initialized with Request States set to PENDING. Checks if the Request State transitions properly upon being claimed, completed, and paid.
testRequestData()	Checks if the Request Data object can be written and read.

RequestTrackerTest

The RequestTracker class is responsible for storing all data received from the Firebase Realtime Database. It is important that RequestTracker can be written and read for the application UI to display Request objects in the request feed.

Test Case	Test Oracle
setUp()	Set up the test Requests Lists with new requests.
testNearbyRequestsList()	This test checks if Request objects can be added and removed from the Nearby Requests List if the location is the same as the user.
testAllRequestsList()	This test checks if Request objects can be added and removed from the All Requests List.
testUserRequestsList()	This test checks if Request objects belonging to the user can be added and removed from the RequestTracker.
testUserReservationsList()	This test checks if Request objects claimed by the user can be added and removed from the RequestTracker.
teardown()	Reset the test requestsList to an empty list.

Member Contributions

Karen worked on implementing the back-end for the project: the FirebaseManager class to interface with the Firebase realtime database by reading and writing requests as well as the RequestTracker to keep track of all requests and the UserTracker for the user scoring system. She also worked on writing the integration (implementation) tests for the request flow.

Matthew worked on the back-end of the project, assisting in the implementation of the UserTracker for the user scoring system. He also worked on writing the integration tests and worked with Susan in writing JUnit tests. Matthew also wrote the instructions for integration testing in the report and created the presentation slides.

Stanley worked on the front-end of the app, including all the visible UI elements and control flow. He implemented the mediator design pattern, making MainActivity a central piece to communicate with various fragments and activities. He also coordinated with the back-end engineers to update database elements on user actions, as well as selectively showing different request information screens based on the user role and the request state.

Susan worked on the project’s back-end development, implementing the User class. She also worked on the class diagram, contributed to writing the JUnit tests, and wrote documentation for the project, which was later generated as a Javadoc.

Uday worked on the project’s back-end development, implementing the Request class. He contributed in some of the JUnit tests. He helped with the class diagrams, as well as with the use case diagrams.