

PICKERpals

Design Document

Matthew DeGenaro, Manoj George, Sean Spencer,
Gianluca Solari, Joseph Whittier, Ryan Beebe,
Joseph Mcilvaine, Joseph Antaki

Team Leader: Joseph Mcilvaine

<https://github.com/RyanBeebe1/SeniorProjectPICKERpals>

Slack: JoeCubed.slack.com

Table of Contents

High Level Description	3
Problem Solving Approaches	4
Design Mockups - Home Screen	5
Screen Navigation	10
Flow Diagrams	11
Technology Stack	14
Backend Information	15
Authentication and Security	16
Input and Output	17
Database Schema	18
Restful Endpoints	19
Team Responsibilities	22

High Level Description

The purpose of PickerPals is to help facilitate a fast and simple transaction between two groups of people; the person getting rid of their unwanted items, and the person searching for things being thrown away.

When the user opens PickerPals, they will be greeted with a feed page that displays listings containing nearby items for which people are throwing away. This listing will feature important information regarding the items, and will allow for communication between both parties involved. For a user to post new items to be 'picked', they will have to create an account by logging in through google.

The chat screen will start by displaying all previous people whom with the user has previously communicated. A user image and their name will be displayed, as well as the last message. Once a user is selected, the chat screen will be displayed, allowing a basic back and forth communication between two users.

Back on the feed screen, there will be a button in the lower right hand corner that allows users to add a listing for a new item. This will bring the user to another screen wherein they fill out the required fields for adding a new item, which will be added to the feed. A user can view the items they have added on their profile screen, as well as their own user information. A user can also be alerted from the app when listings that contain certain keywords appear on the feed, the user will have the ability to set what keywords that the app should look out for as well.

Problem Solving Approaches

Originally, we considered using Google Firebase for the backend database since we are also using it for Google OAuth. After the class lecture about RESTful API, we decided that interacting with these endpoints via Python with a Flask microframework was a better way to go since much of the team has familiarity with Python.

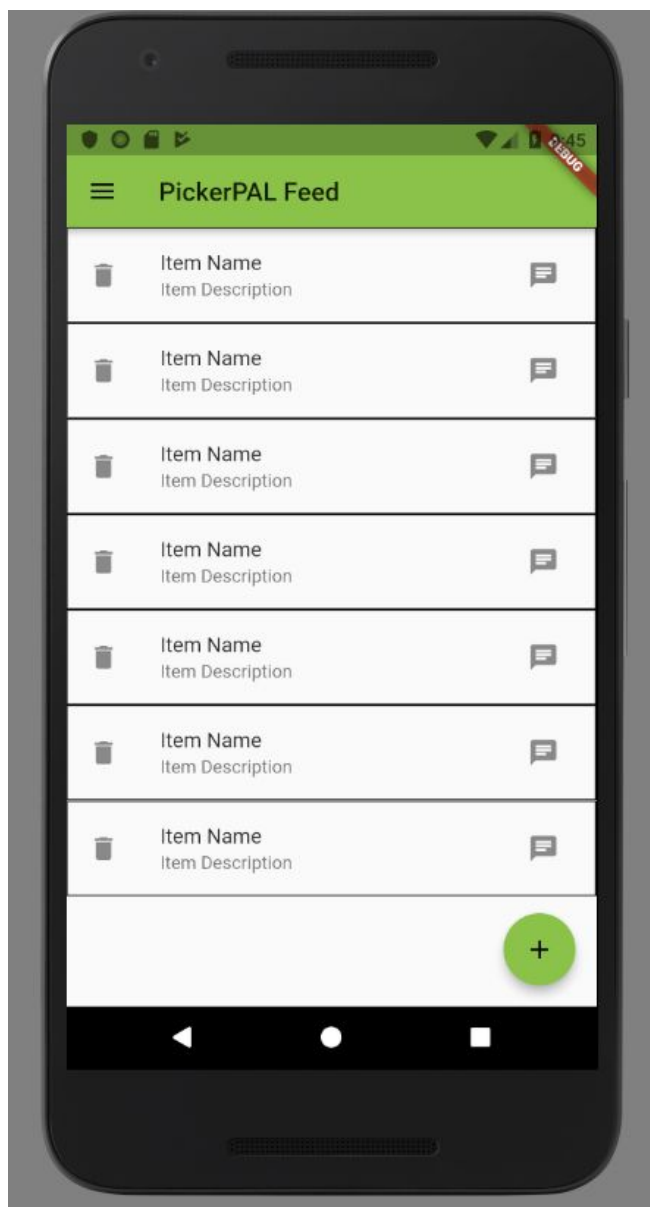
As for our web framework, we considered both Django and Flask, but ultimately chose Flask because it fit better for our purposes since it is considered much more bare-bones, keeping with the pattern of lightweight choices as we made for Amazon endpoints.

One of the problems that might be faced throughout development is the fetching/displaying of listing information on the front-end. For the dynamic feed to be viable, it needs to load only 20 listings at a time. Otherwise, we would end up with a bunch of cached data on the front-end that may never be viewed. So, we need to ensure that when querying the database, we are receiving listings in the correct order and not creating any duplicates. This will require configuring and caching of data on the back-end.

Another problem would be handling of data notifications from the backend to the frontend since that is new territory for us, the plan would be to use Firebase Cloud Messaging as a middleman from the front end to the backend as far as handling and sending notifications.

Design Mockups - Home Screen

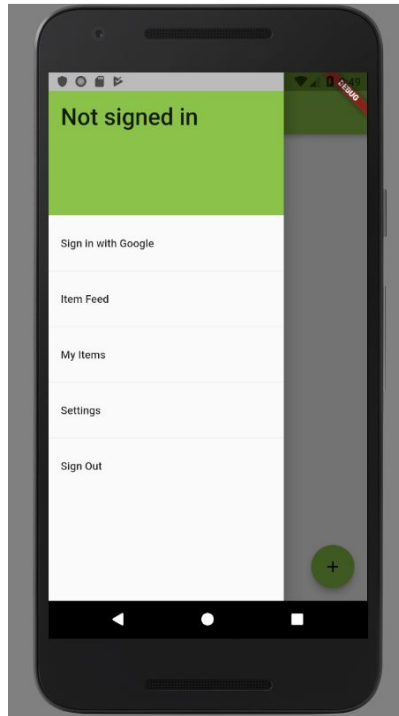
This is the first screen that users will be greeted with when they open the map. This is just conceptual at the time, so it contains no real data. From here, users will be able to navigate to any other screen. None of these designs are set in stone.



Design Mockups - Drawer

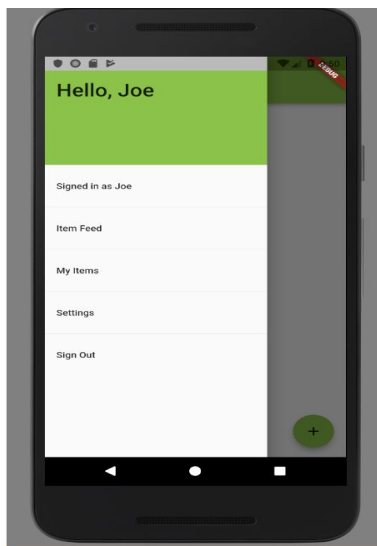
Not Logged in -

This is how the screen will appear when a user is not signed in.



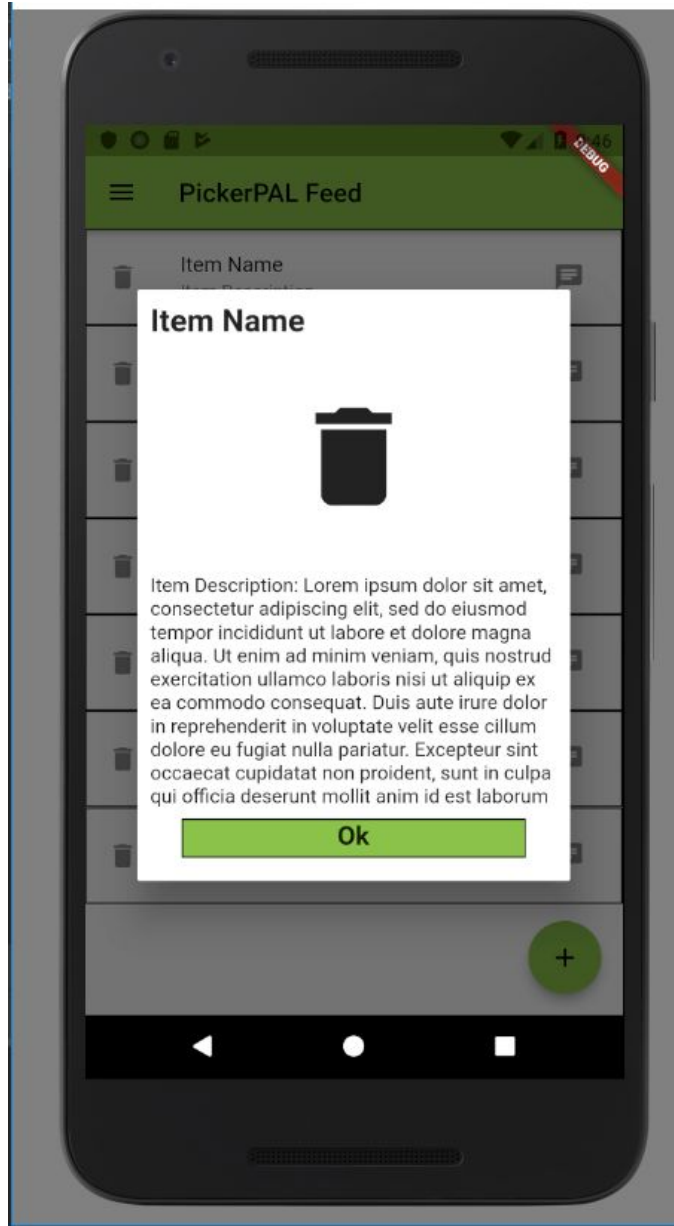
User Logged in -

This is how the screen will appear when a user is logged in through Google.



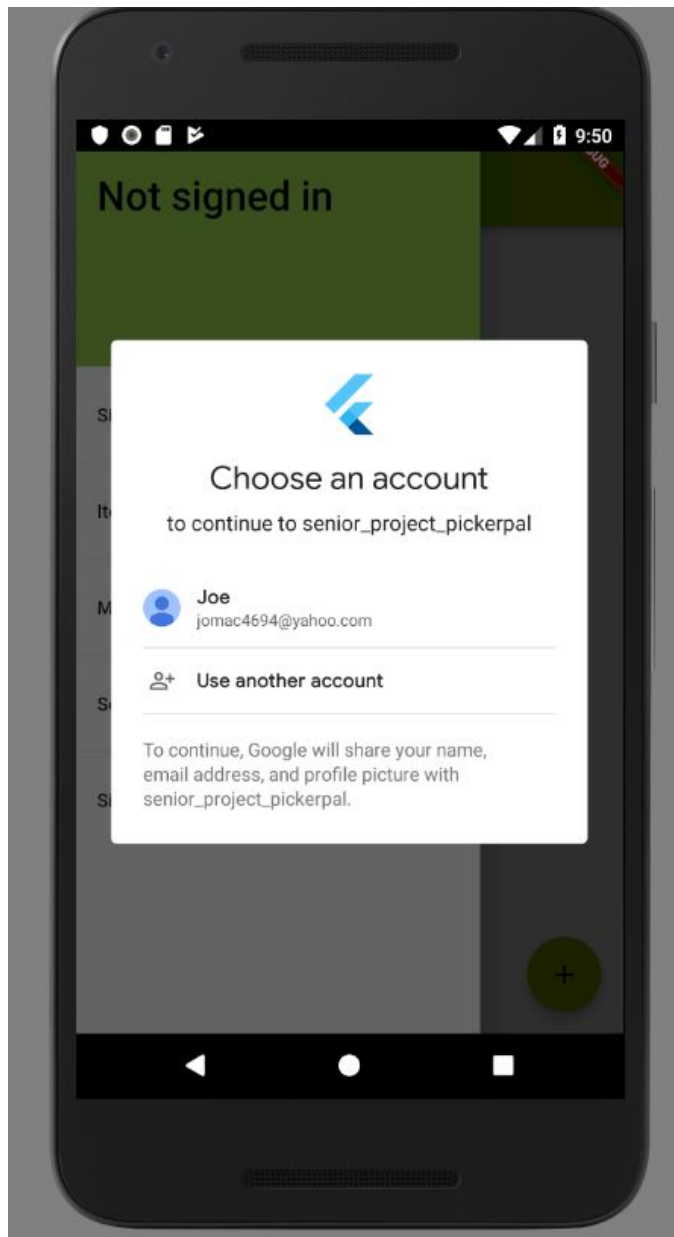
Design Mockups - Entry Description

The entry description will be accessible by clicking on one of the listing tiles. It will provide more detailed information about the listing including the user who posted it and its location.



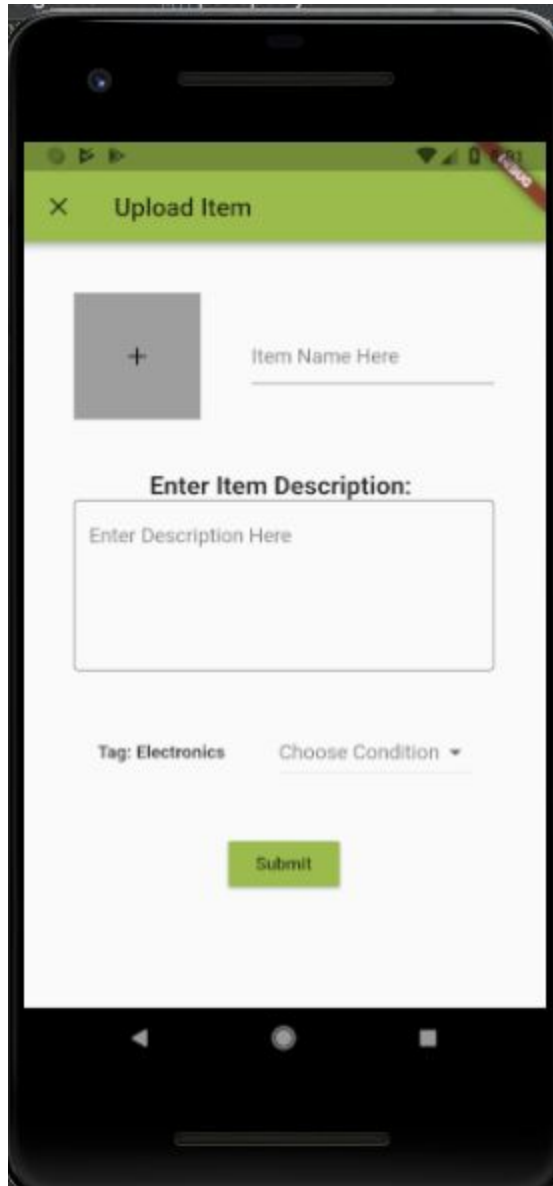
Design Mockups - Login

Users will be able to sign in with their Google account by accessing the right drawer and clicking “Sign in with google”. They will then be prompted with a screen similar to what is shown below.



Design Mockups - Upload Item

Users will be able to upload an item, if they are signed in, and depending on what category they pick a tag will be dynamically generated. This will require access to the camera so that they can upload or take a picture of the item they wish to put up.



The image shows a mobile application interface for uploading an item. The screen has a green header bar with a close button (X) and the title "Upload Item". Below the header, there is a grey square with a white plus sign for uploading a photo, followed by a text input field labeled "Item Name Here". Underneath is a section titled "Enter Item Description:" with a large text area labeled "Enter Description Here". At the bottom, there are two fields: "Tag: Electronics" and "Choose Condition" with a dropdown arrow. A green "Submit" button is located at the very bottom of the form area.

Screen Navigation

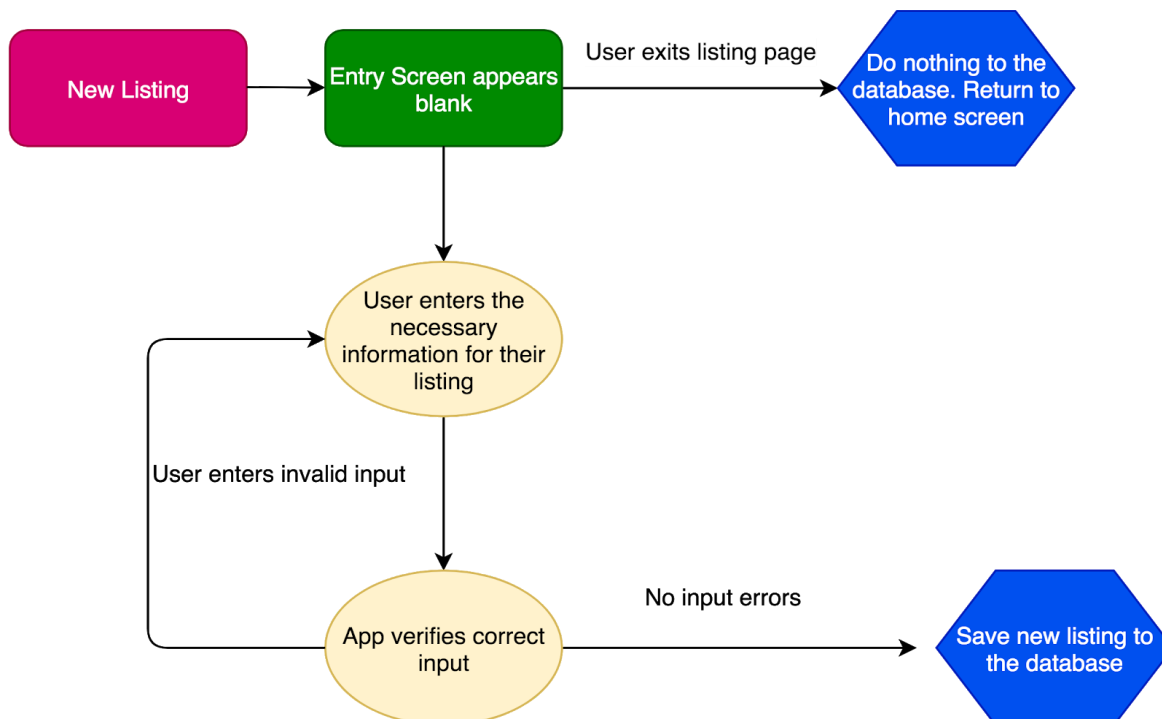
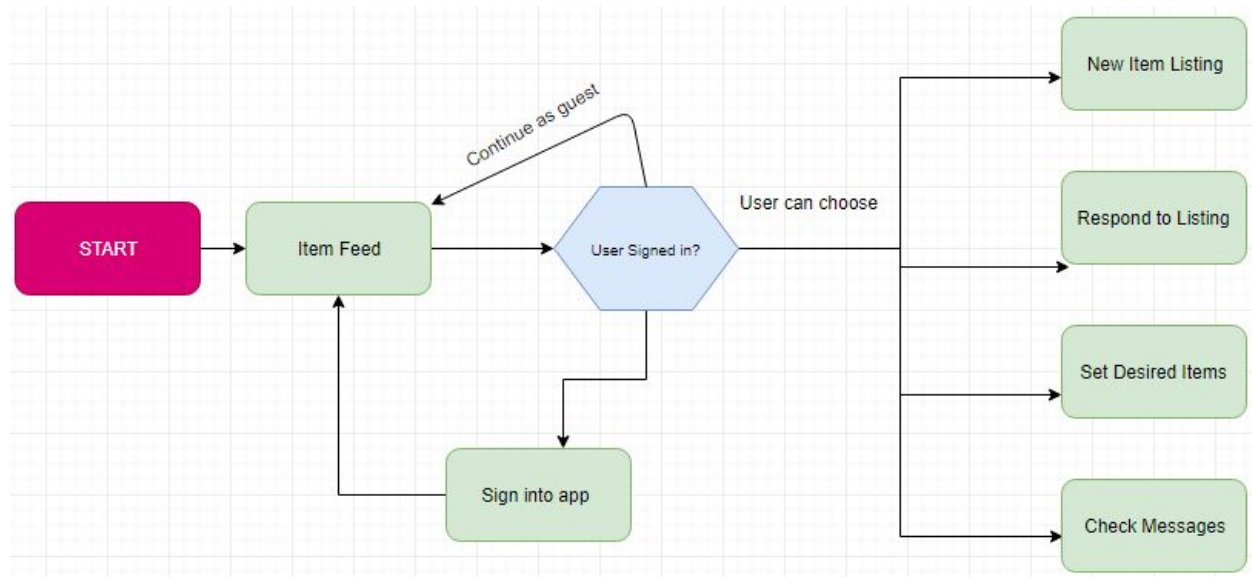
When a user initially opens the app, they will be presented with a dynamic feed of listings. From this screen, users will be able to do a number of tasks. Using the drawer on the left, users will be able to login in through Google, access account settings, view their own listings, and sign out of their Google account. A search icon will also appear in the top right corner to access search functionality. Also, there will be a floating action button that can be used to submit a new listing.

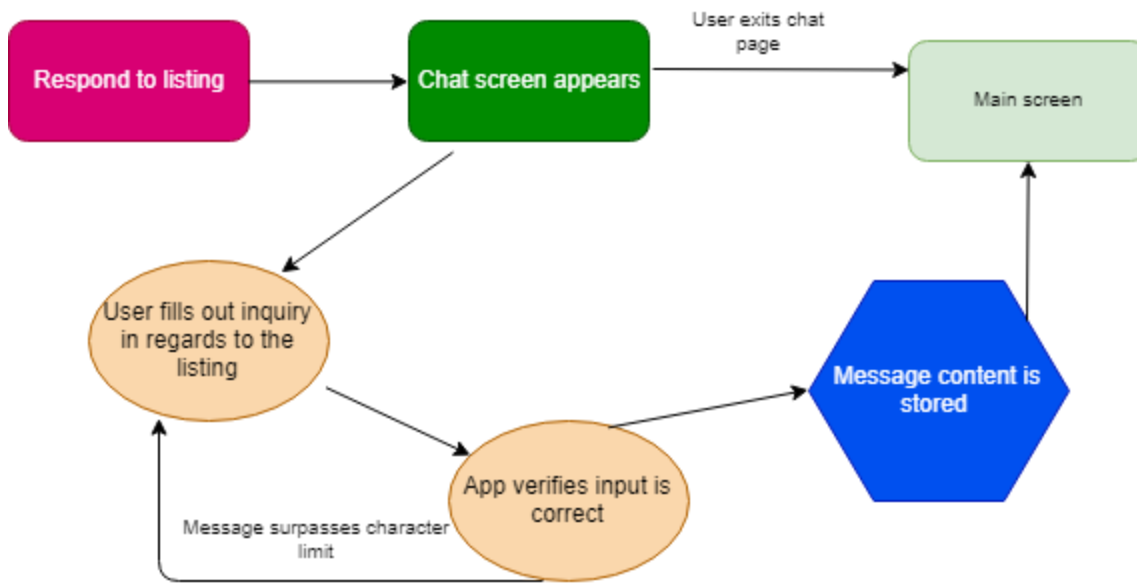
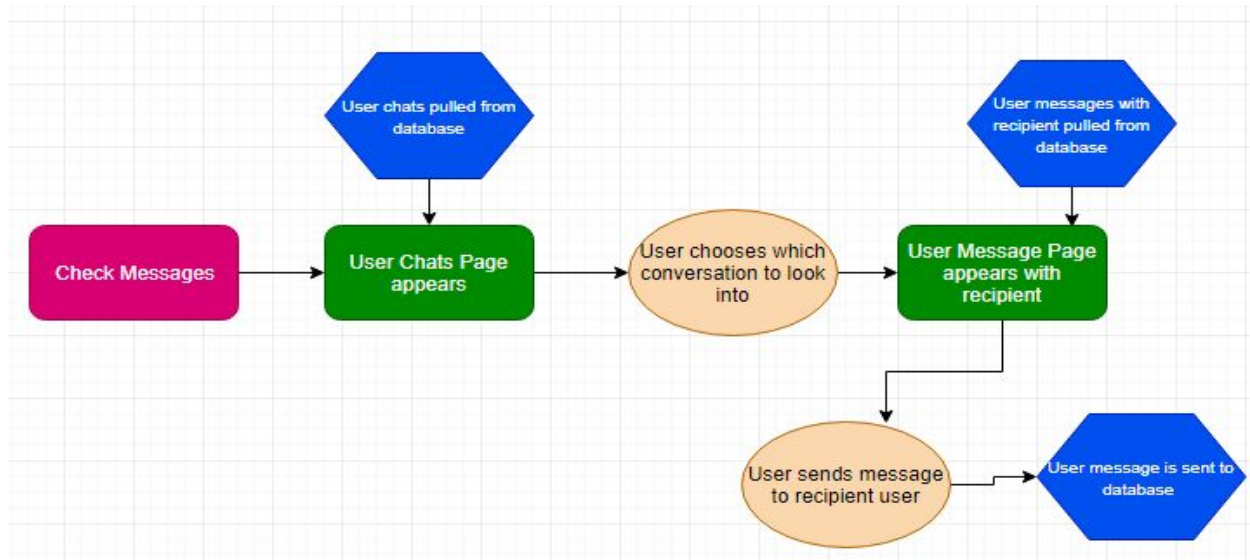
The dynamic feed will include listings submitted by other users. Each listing will contain an icon indicating the type of items, a small description, and a chat icon. If necessary, a user can click on the chat button to get any information about the items or setup a time to get them. When you click on a listing, it will present you with another screen that shows a picture of the items, and a more detailed description.

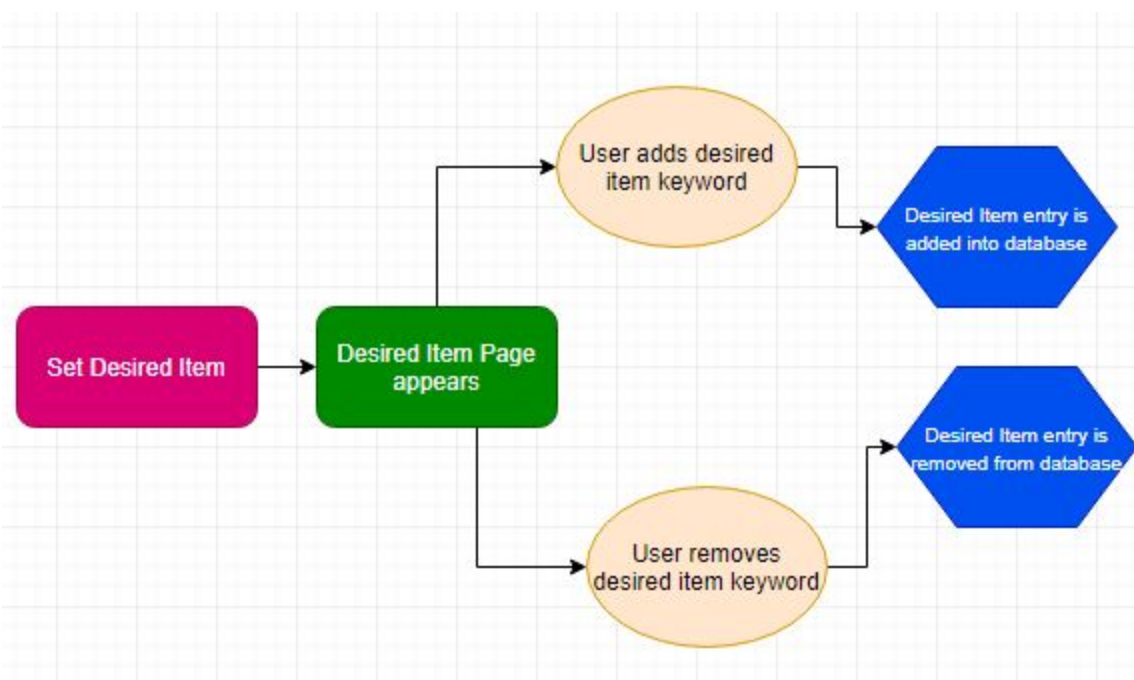
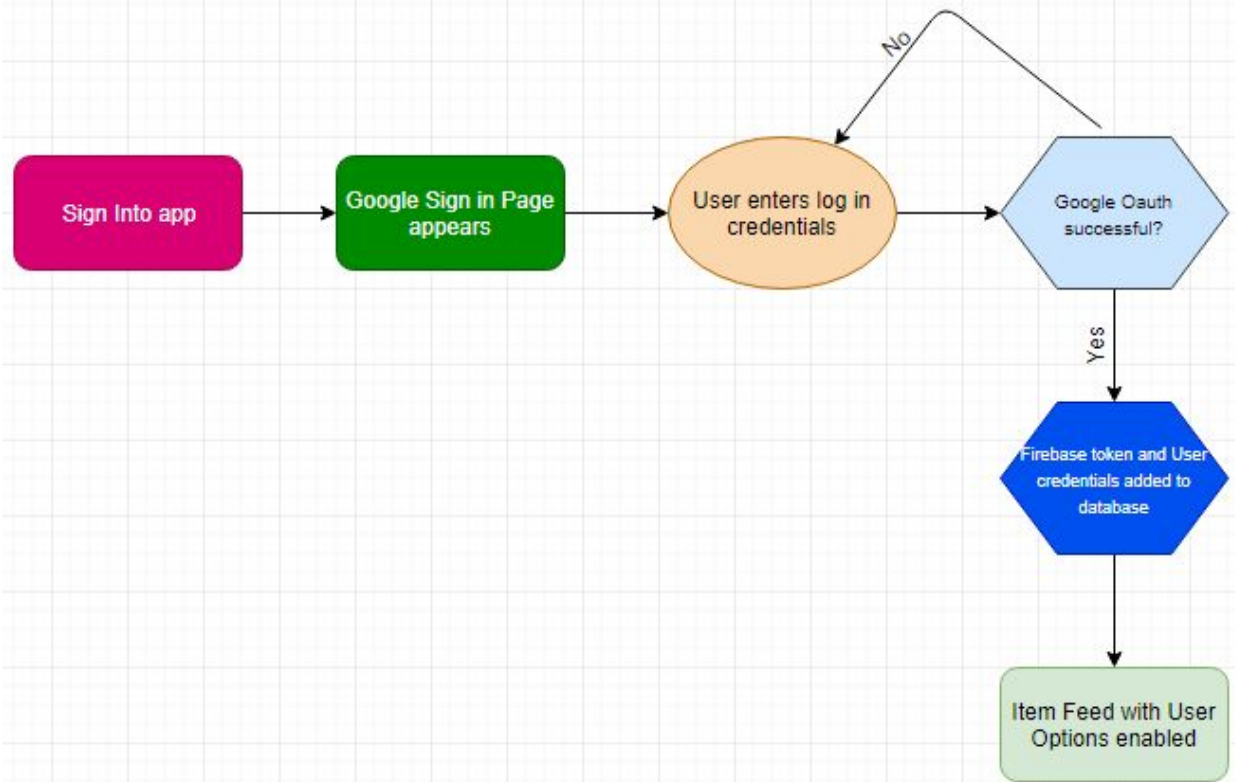
The listing entry page will appear when the floating action button is pressed. From here, users can enter information about the trash they are trying to get rid of. This includes a title, description, location, and a category. When a users submits an entry, they will be directed back the home page.

The user settings page will be accessible through the drawer on the home page. Users will be able to view/edit any information about their account on this page. Settings such as app theme and others will be included. Also, users will be able to manage settings for receiving alerts about listings based on any tags of interests that they specify to the app. Users will also be able to view any of their own listings on a separate feed by pressing the “my items” options in the drawer.

Flow Diagrams







Technology Stack

Our technology stack makes use of Flutter/Dart for the front-end and Flask/Python for the back-end. The app's front-end is being designed primarily in Android Studio. However, parts of the front-end and back-end are being developed in VSCode. Testing is done using both the Android emulator as well as Android devices.

We chose the Flutter framework for our front-end because of its simplicity and ability to quickly produce an interface that works with a RESTful API. Also, since the scale of the project is not large, Flutter was our best option. If the project was going to be used on a larger scale, React Native may have been more appropriate. Lastly, we chose Flutter because a few of our team members were already familiar with it.

We used Google Sign-in and OAuth for login because it's secure and easy to implement. The user's email is then used as a key into our database of profile entries. We also make use of Firebase Authentication on the front-end for the sake of token generation which allows the backend to send push notifications to that user with that token.

Our backend is done using Python and the Flask microframework, which implements a RESTful API. We then use HTTP requests to produce our data on the front-end as needed. Similarly to our front-end choice, we chose Python because of simplicity and familiarity, as well as it's vast library of third-party addons.

The database itself is MySQL and is hosted using AWS. We chose MySQL since we felt the design of our app lends itself best to be used with a relational database. The server itself requires less than a GB of RAM to run, and is running Ubuntu 18.04.1 LTS.

Backend Information

The backend is responsible for storing and managing all information on users and listings using a RESTful API. The backend is also responsible for querying and ordering listings based off various factors. This lets the front-end only have to worry about presenting the data without the need to do any extra sorting.

As listed in the previous section, our backend makes use of the Flask microframework and other Python libraries, while using a MySQL database. The backend is also hosted on an Amazon Web Services EC2 server where the Python App, the database, and user uploaded images are all stored.

As far as libraries go, we use the Flask-SQLAlchemy library which is an ORM (Object Relational Mapper) tool that helps facilitate communication between our Python app and the database. The Marshmallow library is used for the sake of easily generating JSON data from the database models SQLAlchemy uses. Python Image Library is also used to resize uploaded images for thumbnail generation.

In order for the backend to communicate with users via push-notifications we also make use of Firebase Cloud Messaging. We store device tokens of our users in the database and use the PyFCM library on the backend to directly send push notifications to that device with that token.

Authentication and Security

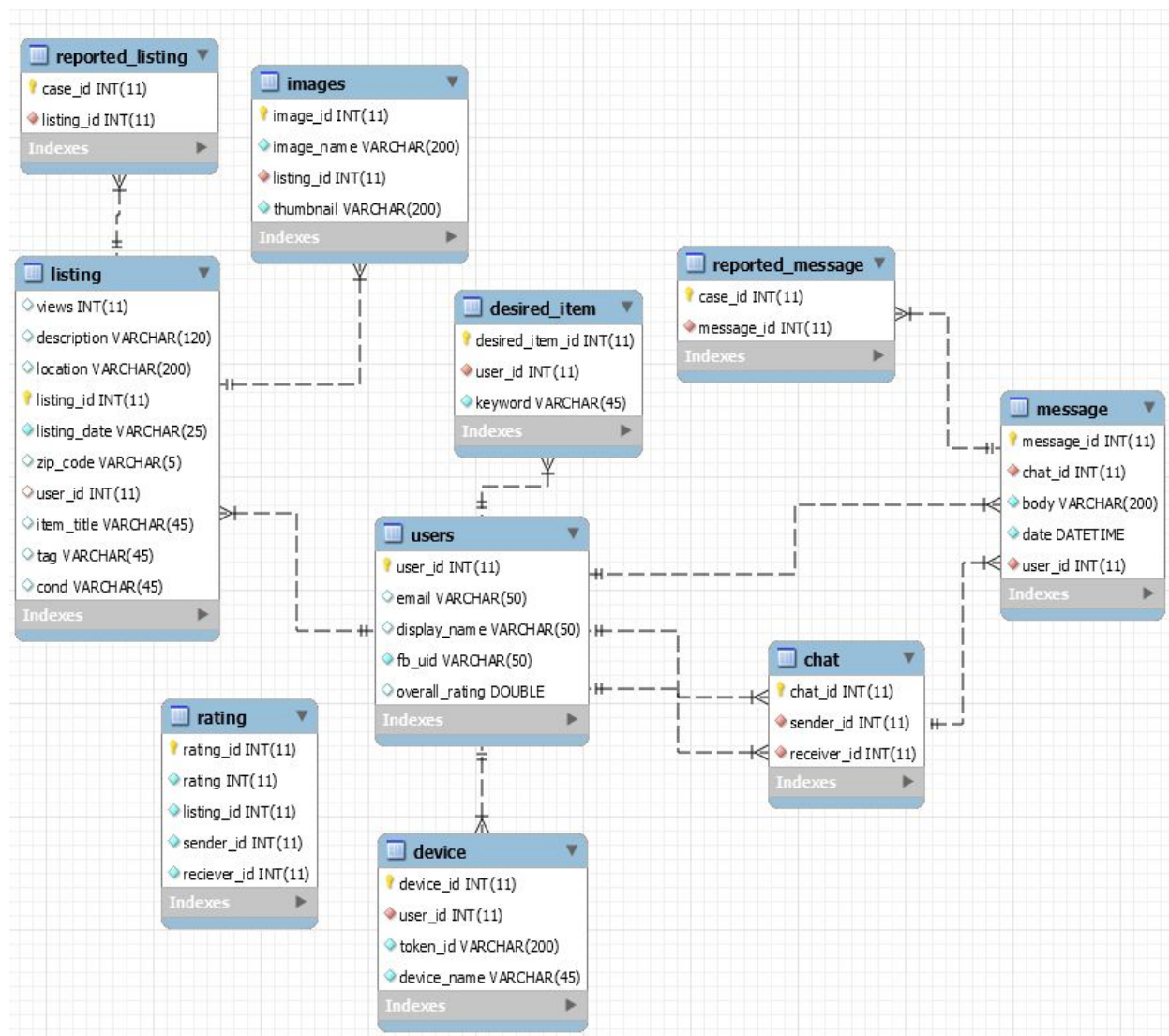
Our app's user authentication is done using Google's OAuth. The user simply needs to login using their Google email and password, the rest is then handled by Google. We chose this method for its convenience, since people don't like being forced to make a new account. On top of Google OAuth, Firebase Authentication is also used for the sake of generating storing user credentials such as their Device token and Google user ID.

Input and Output

Input to the app is primarily accomplished through virtual buttons on the touch screen, the device's physical buttons, and the device's virtual keyboard. The camera is also used for input to take a picture for a listing entry. Other input will be delivered through user interactions with push notifications and snack bars.

Output is provided as a dynamic feed of entries sorted by what is closest to the user's location. The app will also create push notifications when a chat message is received, or when a listing of a user-chosen category is posted in the user's area. Snack bars will notify the user when a listing could not be posted or a chat message could not be sent.

Database Schema



The database itself uses MySQL and consists of ten main tables: listing, users, rating, desired_item, images, chat, reported_message, device, reported_listing, and message. Most of the manipulation of the database is done through the backend with the aid of SQLAlchemy.

Restful Endpoints

POST /adduser

Add a user to the database if they do not already exist as well as adding the device token that the user is making the request from. Return a JSON of the newly created user.

POST /addlisting

Add a listing to the database. Return a JSON of the newly created listing.

POST /adddesireditem

Add a desired item to the database. Return a JSON of the newly created desired item.

POST /upload/{listing_id}

Add an image to the database for the specified listing_id. Return the name of the image.

POST /addrating

Add a rating to the database. Return a JSON of the newly created rating.

POST /changerating

Change the rating a user gave previously to a listing. Return a JSON of the updated rating.

GET /get_overall/{user_id}

Return a JSON containing the overall rating for the specified user_id.

GET /fetch_rating/{sender}/{listing}

Returns a JSON containing a rating that a user gave to a specified listing.

GET /inquire_rating/{sender}/{listing}

Return a JSON containing a true or false value indicating whether a user has rated a listing.

GET /images/{listing_id}

Return an image associated with the specified listing_id.

GET /thumbs/{listing_id}

Return a thumbnail from the image of a specified listing_id.

GET /listings

Return a list of JSONs of all the listings in the database.

GET /userbyid/{user_id}

Return a JSON containing user information associated with the specified user_id if it exists.

GET /listingbyid/{listing_id}

Return a JSON containing listing information associated with the specified listing_id if it exists.

GET /getdesireditem/{desired_item_id}

Return a JSON containing desired item information associated with the specified desired_item_id if it exists.

GET /listingsbyzipcode/{zipcode}

Return a list of JSONs containing listings associated with the specified zipcode if it exists.

GET /listingsbytag/{tag}

Return a list of JSONs containing listings associated with the specified tag if it exists.

GET /listingbyuser/{listing_id}

Return a list of JSONs containing listings associated with the specified user_id if it exists.

GET /deletelisting/{listing_id}

Delete the listing for the specified listing_id.

GET /deleteddesireditem/{desired_item_id}

Delete the desired item for the specified desired_item_id.

POST /sendmessage

Add a message to the database. Return a message indicative the message was sent successfully.

GET /getchats/{user_id}

Return a list of JSONs containing chats associated with the specified user_id if it exists.

GET /getmessages/{chat_id}

Return a list of JSONs containing messages associated with the specified chat_id if it exists.

GET /lastmessage/{chat_id}

Return a JSON of the last message associated with the specified chat_id if it exists.

GET /reportmessage/{message_id}

Add a report for the message associated with the specified message_id.

GET /reportlisting/{listing_id}

Add a report for the listing associated with the specified listing_id.

Team Responsibilities

Joseph Whittier: Back-end design/development

Ryan Beebe: Front-end design/development

Joseph Mcilvaine: Front-end design/development

Matt Degenaro: Back-end design/development

Joe Antaki: Back-end and front-end design/development

Gianluca Solari: Front-end design/development and logo development

Sean Spencer: Back-end design/development

Manoj George: Front-end design/development/morale booster