CS 496 Final Project

Kelvin Watson
ONID: watsokel
OSU ID: 932540242

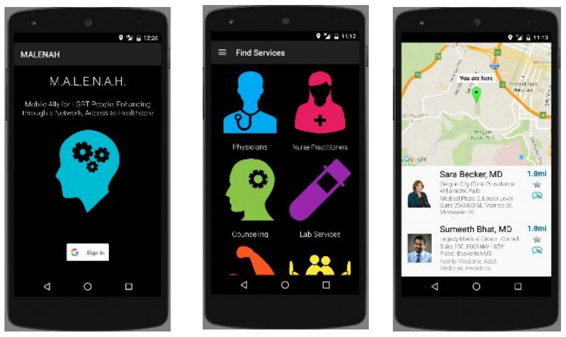
FINAL PROJECT
MALENAH Native Android Application Documentation

Demo Video URL:
http://web.engr.oregonstate.edu/~watsokel/cs496/watsokel496finalvideo.mp4

Documentation Contents

The following aspects of my final project are discussed in this document:

Introduction



My native Android application is called M.A.L.E.N.A.H., which stands for Mobile Ally for LGBT People, Enhancing, through a Network, Access to Healthcare. Briefly, LGBT people face significant barriers to healthcare access and have a host of unique health-related needs. For many in this population, this leads to avoidance and/or delay of necessary medical care. [1] Challenges to health and well-being include anti-gay physical or psychological violence, increased suicide risk and homelessness of LGBT youth, greater risk of HIV and other sexually-transmitted infections, substance abuse, and mental illness. [2] [3] Contributing to these issues are a general lack of healthcare professionals equipped to handle LGBT health-related issues, as well as denial of access to insurance coverage for same-sex domestic partners and transgendered people. [4]

As such, my Android application allows users to access information about culturally-competent healthcare providers who understand medical issues faced by LGBT people in the hopes of closing the LGBT health disparities gap. The Android client retrieves data from the non-relational database (NDB datastore) by communicating with a cloud backend API (malenah-android.appspot.com) which I built on Google App Engine in Python.

Development Platform

I used Android Studio to develop my native application. The application is written in Java. The cloud backend API is built using Google App Engine in Python. Data is stored in the NDB datastore, a non-relational database.

RESTful API

Cloud API and RESTful Constraints

The malenah-android API I built is hosted on Google App Engine, at http://malenah-android.appspot.com/. This API is based on the API I wrote for assignment 3 part 2 and assignment 4, with several important additions, described below.

While the MALENAH API meets some of the RESTful constraints, it does not meet others. Furthermore, some aspects of a given constraints were met while some aspects were not. In the following discussion, the client refers to my native Android application, and the server refers to my running malenah-android cloud API at the URL provided above.

The following addresses some of the REST requirements that were met by the MALENAH API:

- **client-server**: The client remains separate from the server. In the case of my API, the client is my native Android application, and the server consists of the running Google App Engine hosted Python API. The Android app makes use of the cloud API without needing to store data on the Android app itself as the server takes care of data storage. On the other hand, the API is not concerned with issues regarding user interface. [5][6]
- **stateless**: The Android client sends HTTP requests to my API. These HTTP requests contain information in the body and header that communicates state to the server. The server also sends back information in the HTTP request status, header and body, communicating state to client. Although the Android client can update JSON representations of user information (add favorite providers, remove favorites, modify user's display name and email), this changed information is tied to the resource — the user entity in the datastore — and not to the Android client. In this respect, the server does not maintain state about a client. Instead, state is maintained in the resource.
- **layered system**: The Android client connects directly to the malenah-android API. Although there are no other opaque layers, the client is unable to distinguish whether it is connected directly to my API or through some intermediary, so this RESTful constraint is met. [7]
- **uniform interface**:
  - resource identifiers: Each resource (User, Provider, Specialization, Review and Reply entity) is identified by URI's [6].
  - representations: The resources are represented as JSON objects. These JSON object representations of resources (e.g. user profile information, user favorites, reviews) can be modified by the Android client and transmitted back to the malenah-android API. An example of this would be the Android user modifying his/her display name and/or email, or adding/removing a favorited Provider from his/her favorites list. These changes to the representation are submitted as a POST

call to the server at the resource identifier http://malenah-android.appspot.com/user/<userKey>. The server then uses that modified representation to change the original resource in the datastore.
  - self-descriptive messages: The MIME type (JSON) is specified. Client applications can manipulate the resource based on the information in these representations. For example, every User, Provider, Specialization, Review and Reply resource includes its id (key) in the JSON representation. The Provider entity's key can be used in get, post, put and delete operations for Provider, Specialization, Review and Reply entities. The User entity's id (key) can be used to make changes to a user's favorites, and profile information (display name and email). [6][7]
  - HATEOAS: The header and body of the HTTP messages exchanged between the API and the client communicate state. The state is handled by hyperlinks. In my application, these hyperlinks are:
    - /user
    - /user/userID
    - /provider
    - /provider/providerID
    - /provider/providerID/review
    - /provider/providerID/review/reviewID
    - /provider/providerID/review/reviewID/reply
    - /provider/providerID/review/reviewID/reply/replyID
    - /review
    - /review/reviewID
    - /review/reviewID/reply
    - /review/reviewID/reply/replyID
    - /reply
    - /reply/replyID

Some REST requirements that were not met include the following:
- **cacheable**: The server does not send caching information in a Cache-Control header back to the Android client, so there is a possibility that the client may store stale data. [7]
- **uniform interface**:
  - self-descriptive messages: The cacheability of the resource is not specified in a Cache-Control header.
  - HATEOAS: While the resources themselves do not provide explicit links to other resources, it is not difficult to determine how to find other resources. [6]

URIs and Supported HTTP Verbs for each URI
The malenah-android cloud API supports HTTP GET, POST, PUT and DELETE for Provider, Specialization, Review and Reply routes, and HTTP GET and POST for the User routes. The following provides all of the URI's that the API exposes for public consumption (ending forward slashes are optional). An explanation of what calls to each of these URI's do using the 4 major HTTP verbs is also provided.

To access JSON representations of **Provider** entities (and its associated Review entities, and Review entities' associated Reply entities), use the following URL's:

URI and Explanation of what calls to these URI's do using the major HTTP verbs	**HTTP Verbs Supported**			
	GET	**POST**	**PUT**	**DELETE**
http://malenah-android.appspot.com/provider **GET**: Performing a GET request to this URI retrieves demographic information for all healthcare Provider entities in the datastore. **POST**: Performing a POST to this URI adds a new healthcare Provider entity to the datastore.	✓	✓		
http://malenah-android.appspot.com/provider/<providerID>/ **GET**: Performing a GET request to this URI retrieves demographic information for the healthcare provider identified by the providerID (key) in the URI.	✓		✓	✓

PUT: Performing a PUT to this URI allows modification of demographics of the healthcare provider entity with the specified provderID. **DELETE**: Performing a DELETE to this URI deletes from the datastore the healthcare Provider entity with the specified provderID.				
http://malenah-android.appspot.com/provider/<providerID>/review/ **GET**: Performing a GET request to this URI retrieves all of the reviews written for the Provider with the specified providerID. **POST**: Performing a POST to this URI adds a new Review entity to the datastore for the Provider with the specified providerID.	✓	✓		
http://malenah-android.appspot.com/provider/<providerID>/review/<reviewID>/ **GET**: Performing a GET request to this URI retrieves a single review specified by the reviewID for the Provider entity with the specified providerID.	✓			
http://malenah-android.appspot.com/provider/<providerID>/review/<reviewID>/reply **GET**: Performing a GET request to this URI retrieves all of the replies to the review of the specified reviewID for the healthcare provider with the specified providerID. **POST**: Performing a POST request to this URI adds a new Reply entity for the review with the specified reviewID for the healthcare provider with the specified providerID.	✓	✓		
http://malenah-android.appspot.com/provider/<providerID>/review/<reviewID>/reply/<replyID>/ **GET**: Performing a GET request to this URI retrieves the reply with the specified replyID written to the review of the specified reviewID for the healthcare provider with the specified providerID.	✓			

To access JSON representations of **Specialization** entities:

URI and Explanation of what calls to these URI's do using the major HTTP verbs	**HTTP Verbs Supported**			
	GET	**POST**	**PUT**	**DELETE**
http://malenah-android.appspot.com/specialization **GET**: Performing a GET request to this URI retrieves all medical specializations currently in the datastore. **POST**: Performing a POST to this URI adds a new specialization (if it does not already exist) to the datastore.	✓	✓		
http://malenah-android.appspot.com/specialization/<specializationID>/ **GET**: Performing a GET request to this URI retrieves a single medical specialization specified by the specializationID.	✓		✓	✓

To access JSON representations of **Review** entities (and its associated Reply entities):

URI and Explanation of what calls to these URI's do using the major HTTP verbs	**HTTP Verbs Supported**			
	GET	**POST**	**PUT**	**DELETE**
http://malenah-android.appspot.com/review **GET**: Performing a GET request to this URI retrieves all reviews currently in the database.	✓			
http://malenah-android.appspot.com/review/<reviewID>/ **GET**: Performing a GET request to this URI retrieves the review specified by the reviewID. **POST**: Performing a POST to this URI adds a new review to the datastore. **PUT**: Performing a PUT to this URI allows modification of the existing review specified by the reviewID in the URI. **DELETE**: Performing a DELETE to this URI deletes the existing review specified by the reviewID in the URI.	✓	✓	✓	✓
http://malenah-android.appspot.com/review/<reviewID>/reply/ **GET**: Performing a GET request to this URI retrieves all replies to the review specified by the reviewID.	✓			
http://malenah-android.appspot.com/review/<reviewID>/reply/<replyID>/ **GET**: Performing a GET request to this URI retrieves the reply specified by the replyID that is associated with the review specified by the reviewID.	✓			

To access **Reply** entities:

URI and Explanation of what calls to these URI's do using the major HTTP verbs	**HTTP Verbs Supported**			
	GET	**POST**	**PUT**	**DELETE**
http://malenah-android.appspot.com/reply **GET**: Performing a GET request to this URI retrieves all replies currently in the database. **POST**: Performing a POST request to this URI adds a new reply to the database.	✓	✓		
http://malenah-android.appspot.com/reply/<replyID>/ **GET**: Performing a GET request to this URI retrieves the reply specified by the replyID. **PUT**: Performing a PUT to this URI modifies an existing reply in the datastore. **DELETE**: Performing a DELETE to this URI deletes an existing reply in the datastore.	✓		✓	✓

To access **User** entities:

URI and Explanation of what calls to these URI's do using the major HTTP verbs.	**HTTP Verbs Supported**			
	GET	**POST**	**PUT**	**DELETE**
http://malenah-api.appspot.com/user **GET**: Performing a GET request to this URI retrieves demographics for all users in the datastore. **POST**: Performing a POST to this URI can accomplish several different tasks. If the user-action post parameter is set to "edit-user", then the User entity's demographics (display name, email) are modified and stored. If the user-action post parameter is set to "add-favorite", then a provider entity key is added to the User entity's favorites list. If the user-action post parameter is set to "remove-favorite", then a provider entity key is removed from the User entity's favorites list.	✓	✓ Multi-purpose, see explanation		
http://malenah-api.appspot.com/user/<userID>/ *Like the other ID's in the routes above, the userID is actually the id of the NDB key (kind,id) pair. Even though each user entity has a user_id property, the key is used instead of the user_id property because the user_id property is actually the identification number assigned by the Google Sign-In for Android API.* **GET**: Performing a GET request to this URI retrieves demographic information for the particular user with the specified userID.	✓	✓		

A code excerpt showing the routes in the malenah-android backend API from main.py is provided in Fig. 1 below.

```
routes = [
  webapp2.Route(r'/<:specialization>/<sid:[0-9]+><:/?>', handler=S.SpecializationHandler, name='specialization'),
  webapp2.Route(r'/<:specialization><:/?>', handler=S.SpecializationHandler, name='specialization-list'),
  webapp2.Route(r'/<:provider>/<pid:[0-9]+>/<:review>/<revid:[0-9]+>/<:reply>/<repid:[0-9]+><:/?>',
handler=r.ReplyHandler, name='provider-review-reply'),
  webapp2.Route(r'/<:provider>/<pid:[0-9]+>/<:review>/<revid:[0-9]+>/<:reply><:/?>', handler=r.ReplyHandler,
name='provider-review-reply'),
  webapp2.Route(r'/<:provider>/<pid:[0-9]+>/<:review>/<revid:[0-9]+><:/?>', handler=R.ReviewHandler,
name='provider-review'),
  webapp2.Route(r'/<:provider>/<pid:[0-9]+>/<:review><:/?>', handler=R.ReviewHandler, name='provider-review-list'),
  webapp2.Route(r'/<:provider>/<pid:[0-9]+><:/?>', handler=P.ProviderHandler, name='provider'),
  webapp2.Route(r'/<:provider><:/?>', handler=P.ProviderHandler, name='provider-list'),
  webapp2.Route(r'/<:review>/<revid:[0-9]+>/<:reply>/<repid:[0-9]+><:/?>', handler=r.ReplyHandler, name='reply'),
  webapp2.Route(r'/<:review>/<revid:[0-9]+>/<:reply><:/?>', handler=r.ReplyHandler, name='reply-list'),
  webapp2.Route(r'/<:review>/<revid:[0-9]+><:/?>', handler=R.ReviewHandler, name='review'),
  webapp2.Route(r'/<:review><:/?>', handler=R.ReviewHandler, name='review-list'),
  webapp2.Route(r'/<:reply>/<repid:[0-9]+><:/?>', handler=r.ReplyHandler, name='reply'),
  webapp2.Route(r'/<:reply><:/?>', handler=r.ReplyHandler, name='reply-list'),
  webapp2.Route(r'/<:user>/<uid:[0-9]+><:/?>', handler=U.UserHandler, name='user'),
  webapp2.Route(r'/<:user><:/?>', handler=U.UserHandler, name='user-list'),
  webapp2.Route(r'/', handler=P.ProviderHandler, name='provider-list'),
]
application = webapp2.WSGIApplication(routes, debug=True, config=config)
```

Fig. 1. Code excerpt from cloud backend API's main.py, written in Python and hosted on Google App Engine at malenah-android.appspot.com.

Non-Relational Database Design and Implementation

In my non-relational database design (depicted in Fig. 2. below), contention during writes of a review are minimized. For example, suppose two users are writing a review for the same healthcare provider. Each time a user writes a review, the key to the review is added to a list property in the Provider entity group, and a new Review entity (and entity group) is created. Review entities are always stored as the sole entity (root entity) of an entity group. Suppose that this process is repeated and there are now several reviews for a particular provider. At some later point in time, suppose two users write replies to two separate reviews. In this case, the writes could be performed as transactions on each individual Review entity group without concern about contention between the two users for the writes. In contrast, if Review entities are in the same entity groups as Provider entities, if two users try to write a reply (or even another review) for a given Provider entity group, there is a greater risk of contention as the two users compete to write to the same entity group. [12][13]

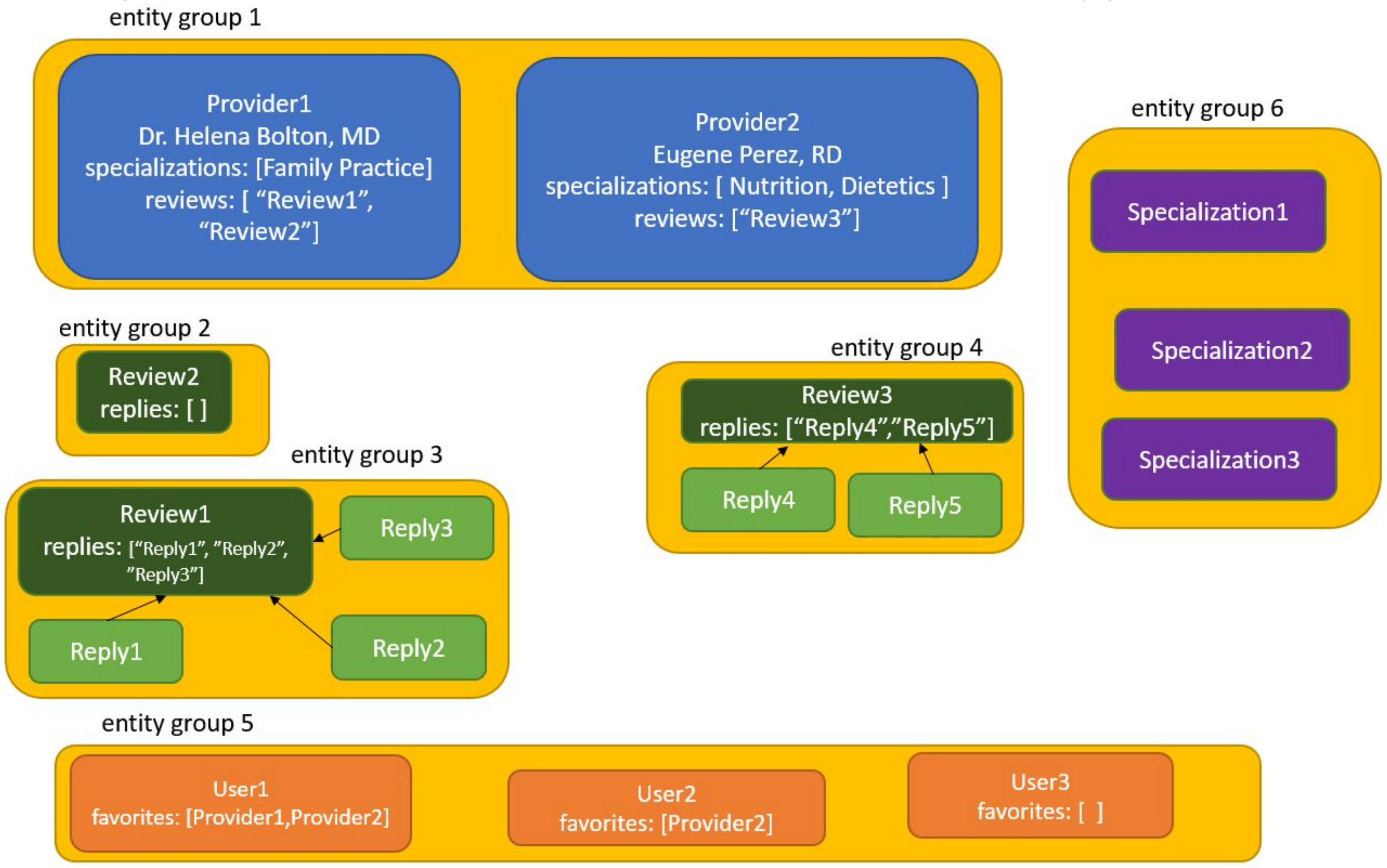


Fig. 2. Non-relational database design for the NDB datastore.

Another functionality of my cloud API is the ability of a registered user (saved as a User entity in the datastore) to view all of his/her favorited Provider entities. Because all Provider entities are members of the same entity group, a query for all Provider entities would not be a global query against multiple entity groups, but instead, an ancestor query. This ensures strong consistency when querying favorited Providers. Similarly, queries for Reply entities within a single Review entity group are always strongly consistent, meaning that if a goal is to find all of the replies for a particular review, all of the most current replies would be returned. [12][13]

However, this design is not without its drawbacks. One use case would be that when reading a particular Provider entity, all of the Review (and Reply) entities associated with the Provider entity need to be retrieved. This would be done by going through the list of keys in the Provider entity's review property and retrieving individual Review entity groups. This type of read cannot be completed as a transaction if there are more than 5 entity groups storing Review entities. In this particular use case, the query would yield an eventually consistent, but not strongly consistent result, meaning that it cannot be guaranteed that the most up-to-date

Reply entities will be returned by the query if during the query, a write of a new Reply entity by another user is being performed.

In order to retrieve all Specialization entities associated with a Provider, an ancestor query could be performed on the single Specialization entity group. This is because all Specialization entities are created using the same ancestor. Thus, reads of Specialization entities are strongly consistent. The disadvantage of this is there is a possibility of contention if there are multiple writes to the Specialization entity group. [12][13]

If we consider an alternative design where all Review and Reply entities for a Provider had that Provider entity as its ancestor, each read would involve reading all data within a single entity group, so reads will be set up as transactions (transactional reads) since a transactional read is a strongly-consistent read because it is within one entity group. Under this alternative design, when a user wants to view all of the reviews and replies for a single provider, an ancestor query could be performed. Because this query uses the entity group’s local indexes, the result is a strongly-consistent read. In other words, placing all of the reviews and replies in the same entity group as the healthcare provider would make a query result on a single Provider entity group strongly consistent. However, this alternative design could only allow one user at a time to write to an entity group. Therefore, if one user attempted to write a review (or reply) to a Provider entity group, the entire Provider entity group would be unavailable to other users to write either reviews or replies. If the write was defined in a transaction, it could be retried. However, tying up an entire Provider entity and blocking all writes to that Provider during a single write is less efficient than the current design, where a single write would only occupy one Review entity group, rather than all other Review entity groups. [12][13]

In the current design, there is still the possibility of contention when multiple users try to reply to a review. However, the frequency of contention in this design is likely to be less than if all Reviews associated with a Provider are part of the same entity group as that Provider entity.

My non-relational database is implemented in NDB in Google App Engine (Python). Provider entities are created under the same ancestor. Similarly, all Review and User entities are also created using the same Review and User ancestor respectively. In contrast, Review entities are not stored under the same ancestor and are always created as separate entity groups.

```
parent_key = ndb.Key(E.Provider, self.app.config.get('M-P'))
e = E.Provider(parent=parent_key)
e.populate(**properties)
e.put()
```

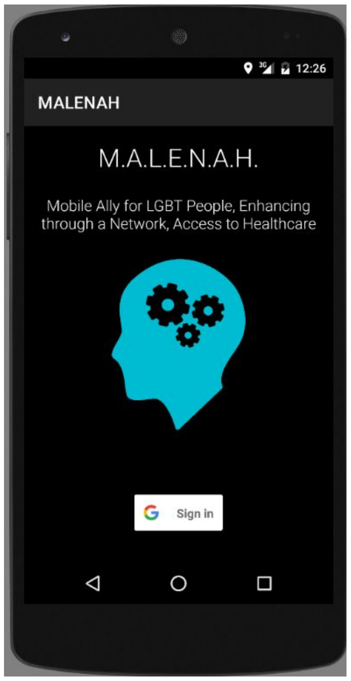
Fig. 3. Code showing Provider entity being created under the same ancestor. The result is that all Provider entities are in the same entity group.

Account System

My native Android application implements Google’s Sign-In for Android, and my cloud API backend implements OAuth 2.0 for user authentication.

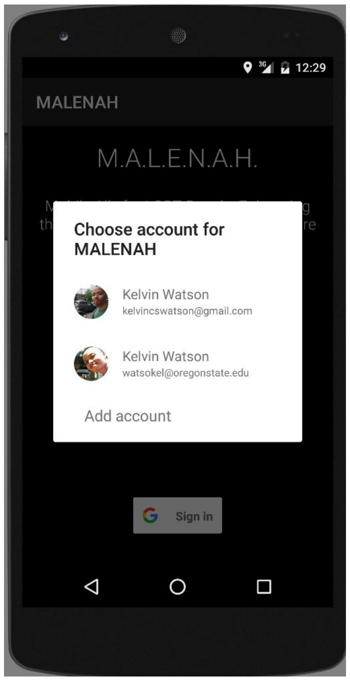
Creating an account and logging in

When the user taps on the Google “Sign In” button on the home screen (MainActivity.java), he/she is presented with two options: log in with one of the device’s existing Google accounts, or add a new account to log in to the application. If the user already has existing Google accounts in the physical Android device, he/she can use those accounts to log in directly. If the user does not have a Google account, he/she can sign up for one as part of the sign in process for the Android application. These scenarios are depicted in Fig. 4. Note that if there are no active Google accounts currently in the Android device, the user will be directed to the screen shown in Fig. 4c when he/she taps on the “Sign In” button.

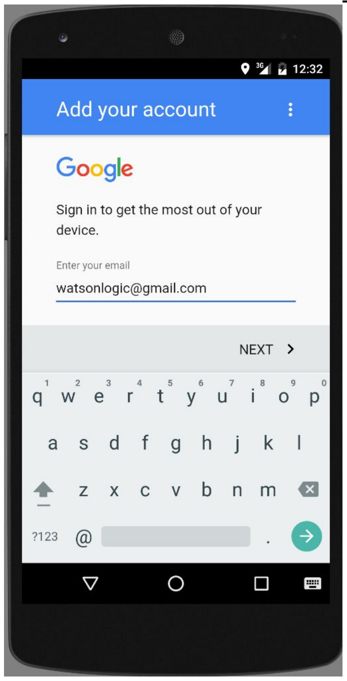


Mobile Ally for LGBT People, Enhancing through a Network, Access to Healthcare

4a. Home screen

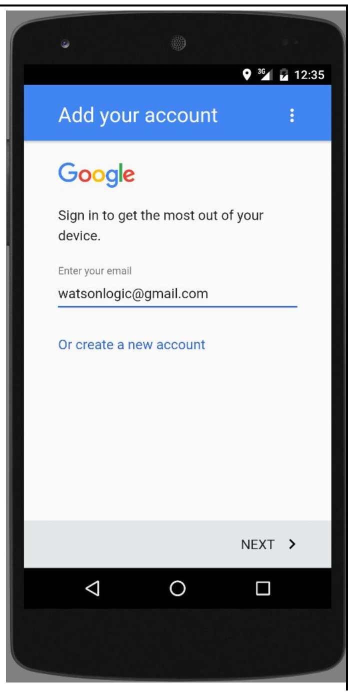


4b. Choosing an account



Sign in to get the most out of your device.

4c. If the user chooses "Add account", then he/she is directed to the Add your account screen.

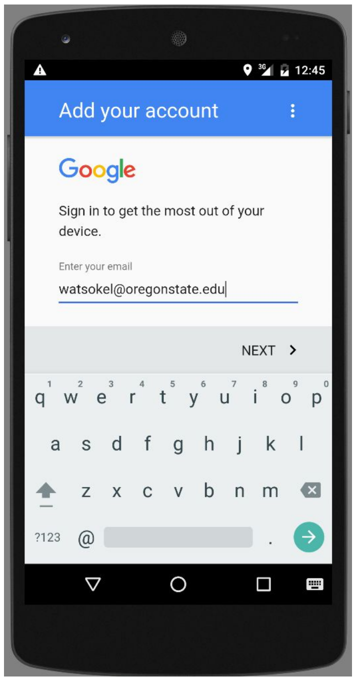


Sign in to get the most out of your device.

4d. The user could also create a Google account if he/she does not have an existing one.

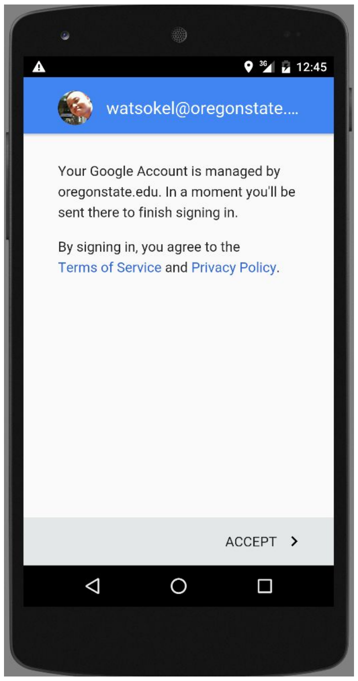
Fig. 4. Choosing an account for login, or adding a different account to log in to the MALENAH Android app.

If the user provides a valid Google account, then he/she will be logged in, the Google Sign-In button will disappear, and the "Launch" button will be enabled to allow the user to proceed (Fig. 6).



Sign in to get the most out of your device.

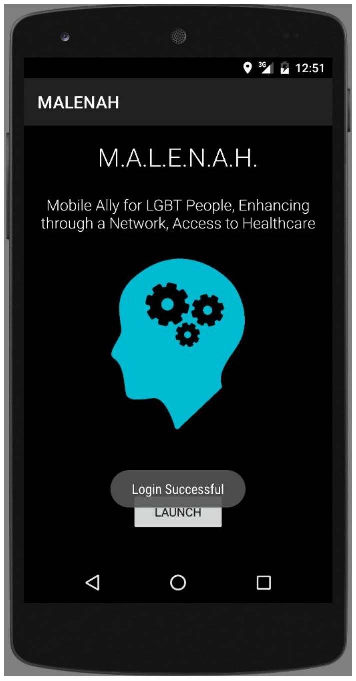
5a. Entering Google email and password



Your Google Account is managed by oregonstate.edu. In a moment you'll be sent there to finish signing in.

By signing in, you agree to the Terms of Service and Privacy Policy.

5b. Agreeing to terms



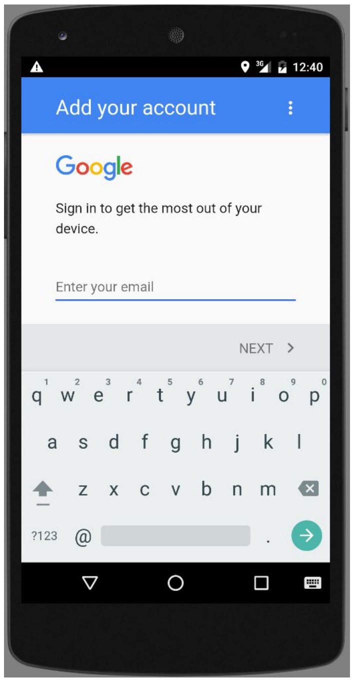
Mobile Ally for LGBT People, Enhancing through a Network, Access to Healthcare

5c. When valid Google account credentials are provided, login is successful.

Fig. 5. Successful sign-in sequence.

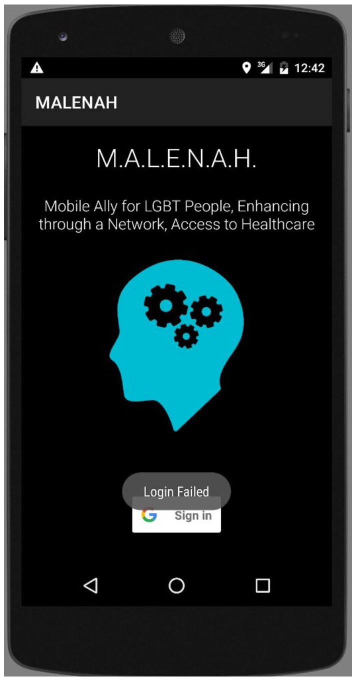
Error handling

If the user does not have any Google accounts installed on the Android device, he/she is immediately directed to the “Add your account” screen. However, if the user does not provide an account and instead presses the back button at the “Add your account” screen, the login will fail unless the user provides a valid Google account. (Fig. 6)



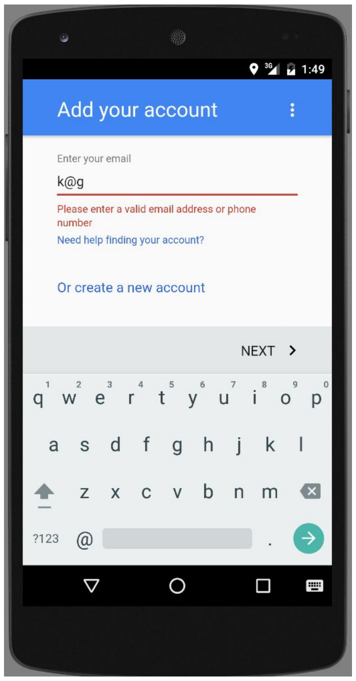
Sign in to get the most out of your device.

6a. Prompt to sign in with an existing Google account (or to create a new Google account). Pressing the back button here causes login to fail.



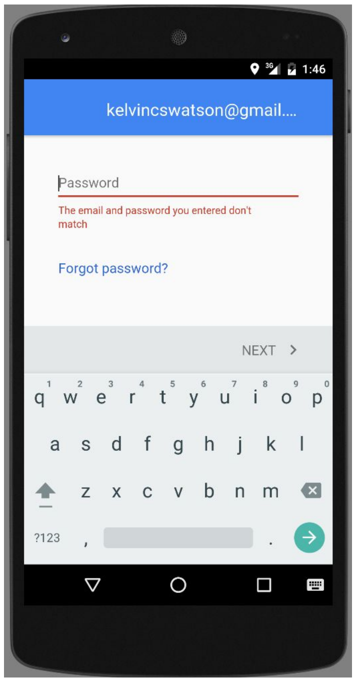
Mobile Ally for LGBT People, Enhancing through a Network, Access to Healthcare

6b. Pressing the back button results in a login failed message.



Please enter a valid email address or phone number

6c. Entering an invalid email address causes results in an error message



The email and password you entered don't match

6d. Entering an incorrect password causes results in an error message.

Fig. 6. Scenarios showing error handling on log in. If the user enters an invalid email, password, or presses the back button before submitting credentials, login will fail.

Explanation of Android Code for Google Sign-In

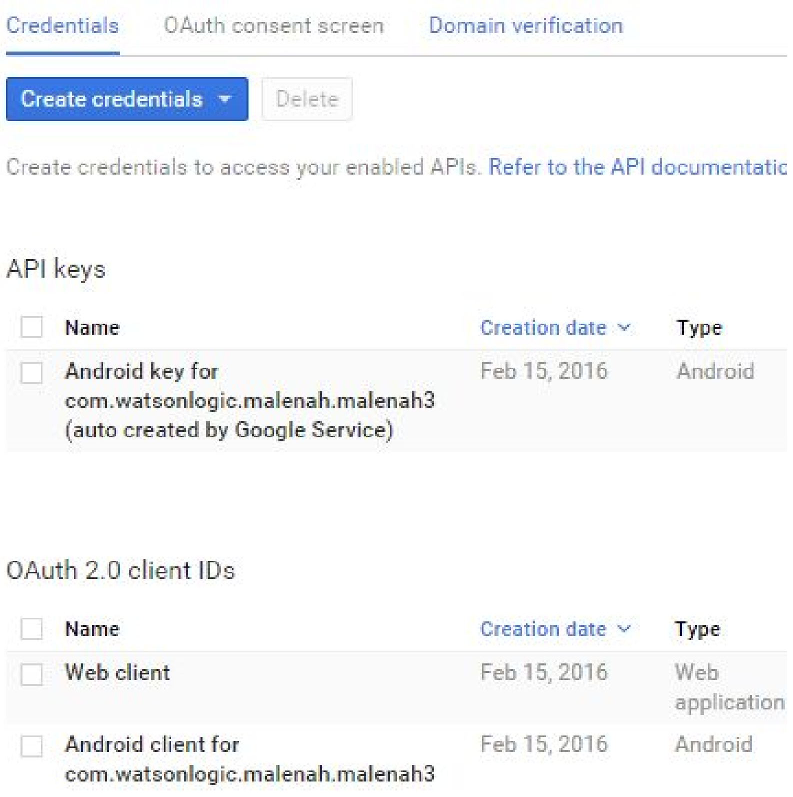
In MainActivity.java (home screen)’s onCreate() method, the configureGoogleLogin() method is called. This method creates a new GoogleSignInOptions object to request an ID Token. This token will be passed to the the Google App Engine Python server for user authentication as part of OAuth 2.0, described in the next section. [8] When the user clicks on the sign in button and undergoes the Google sign in process (depicted in the screen captures in Fig. 6), a result is returned, indicating whether or not the user has signed in successfully. The result object’s isSuccess() method will return true (login success) or false (login failed). If the user has successfully logged in, the Google Sign-In button disappears, a Toast message indicates that the attempt to login succeeded, and the launch button is enabled. If login fails (for example, if an invalid Google account username and/or password is/are provided, or if one or both the username and password is/are not provided), the Google Sign-In button will remain, the launch button will be disabled and a Toast message will indicate to the user that the attempt to login has failed. A relevant code excerpt is shown in Fig. 7.

```
public void configureGoogleLogin() {
    GoogleSignInOptions gso = new GoogleSignInOptions.Builder(GoogleSignInOptions.DEFAULT_SIGN_IN)
            .requestIdToken(getString(R.string.server_client_id))
            .requestEmail()
            .build();
    mGoogleApiClient = new GoogleApiClient.Builder(this)
            .enableAutoManage(this, this)
            .addApi(Auth.GOOGLE_SIGN_IN_API, gso)
            .build();
```

```
        signInButton = (SignInButton) findViewById(R.id.sign_in_button);
        signInButton.setSize(SignInButton.SIZE_STANDARD);
        signInButton.setScopes(gso.getScopeArray());
        findViewById(R.id.sign_in_button).setOnClickListener(this);
    }

    @Override public void onClick(View v) {
        switch (v.getId()) {
            case R.id.sign_in_button:
                signIn();
                break;
            // ...
        }
    }

    private void signIn() {
        Intent signInIntent = Auth.GoogleSignInApi.getSignInIntent(mGoogleApiClient);
        startActivityForResult(signInIntent, RC_SIGN_IN);
    }

    @Override protected void onActivityResult(int requestCode, int resultCode, Intent data) {
        super.onActivityResult(requestCode, resultCode, data);
        if (resultCode==SET_LOCATION_REQUEST) {
            switch (requestCode) {
                case 1:
                    break;
            }
        }
        if (requestCode == RC_SIGN_IN) {
            GoogleSignInResult result = Auth.GoogleSignInApi.getSignInResultFromIntent(data);
            handleSignInResult(result);
        }
    }

    private void handleSignInResult(GoogleSignInResult result) {
        Log.d("SignInActivity", "handleSignInResult:" + result.isSuccess());
        if (result.isSuccess()) {
            // Signed in successfully, show authenticated UI.
            signInButton.setVisibility(View.GONE);
            GoogleSignInAccount acct = result.getSignInAccount();
            Map<String,String> postParams = new LinkedHashMap<>();
            postParams.put("id_token", acct.getIdToken());
            postParams.put("email",acct.getEmail());
            postParams.put("name", acct.getDisplayName());
            new VerifyTokenAsyncTask(MainActivity.this, postParams).execute();
            Toast.makeText(getApplicationContext(),"Login Successful",Toast.LENGTH_LONG).show();
            startButton.setVisibility(View.VISIBLE);
            startButton.setEnabled(true); //disable start button until all data retrieved
        } else {
            startButton.setEnabled(false);
            Toast.makeText(getApplicationContext(), "Login Failed", Toast.LENGTH_LONG).show();
        }
    }
```
Fig. 7. Code showing implementation of Google Sign-In for Android on the MALENAH native Android application.

Implementing OAuth 2.0 in the backend server for User authentication in the malenah-android cloud backend API

This section describes how I implement user authentication using OAuth 2.0 in my Google App Engine malenah-android backend API written in Python.



Create credentials to access your enabled APIs. Refer to the API documentatic

API keys

Name	Creation date	Type
Android key for com.watsonlogic.malenah.malenah3 (auto created by Google Service)	Feb 15, 2016	Android

OAuth 2.0 client IDs

Name	Creation date	Type
Web client	Feb 15, 2016	Web application
Android client for com.watsonlogic.malenah.malenah3	Feb 15, 2016	Android

For user authentication, the Android application (front end) implements Google Sign-In for Android and my cloud API backend implements OAuth 2.0 to verify the integrity of the id token.

In order to authenticate the Android user on my backend server, I first needed to retrieve the OAuth 2.0 client ID that was created for my server, by logging into the Google Developer Console and selecting my malenah-android application (cloud API backend). This is illustrated in the screen capture in Fig. 8. (OAuth client ID's are not included in the screen capture).

Fig. 8. Retrieving the OAuth 2.0 client ID from the Google App Engine Developer Console in order to implement OAuth in the malenah-android API backend server.

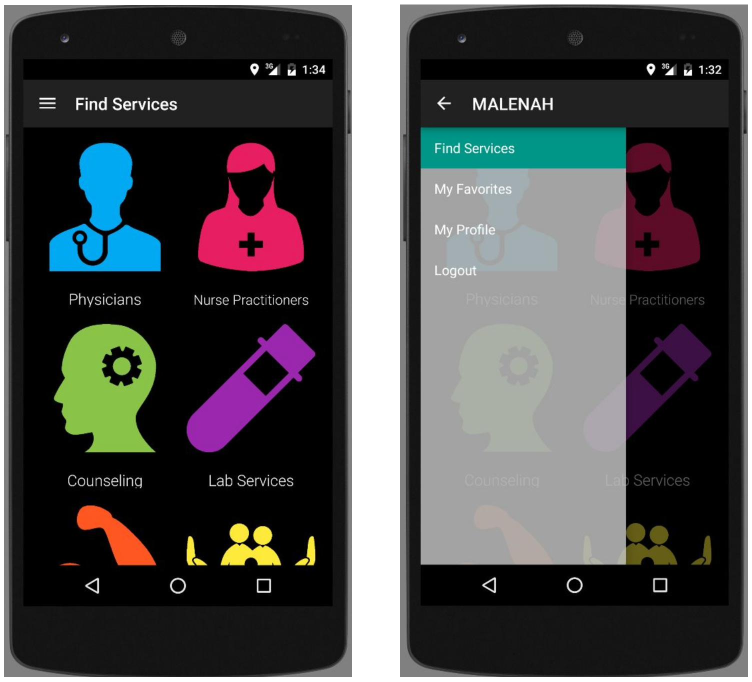
To confirm the identity of the user that is currently signed in, I need to verify that the ID token obtained by the GoogleSignInOptions object is authentic and valid. Once the user has signed in, I send the token via HTTPS to my API asynchronously using an AsyncTask in the Android client. Sending the token via HTTPS is important because this encrypts the transmitted token and renders it unreadable if intercepted. [8][9][10] The code for the VerifyTokenAsyncTask is shown in Fig. 9.

```
public class VerifyTokenAsyncTask extends AsyncTask<Void,Void,String> {
    @Override protected String doInBackground(Void... params) {
        /* Set params */
        byte[] postDataBytes = generatePostData();

        /* Set headers and write */
        URL url = null;
        HttpURLConnection conn = null;
        try {
            url = new URL("https://malenah-android.appspot.com/user");
            conn = (HttpURLConnection) url.openConnection();
            conn.setRequestMethod("POST");
            conn.setRequestProperty("Content-Type", "application/x-www-form-urlencoded");
            conn.setRequestProperty("Content-Length", String.valueOf(postDataBytes.length));
            conn.setDoOutput(true);
            conn.getOutputStream().write(postDataBytes);
        } catch (MalformedURLException e) {
            e.printStackTrace();
        } catch (IOException e) {
            e.printStackTrace();
        }

        /* Retrieve response */
        Reader in = null;
        String response = new String();
        try {
            in = new BufferedReader(new InputStreamReader(conn.getInputStream(), "UTF-8"));
            response = "";
            for (int c = in.read(); c != -1; c = in.read()) {
                response += (char) c;
            }
            Log.d(TAG+" response", response);
        } catch (IOException e) {
            e.printStackTrace();
        }
        return response;
    }
}
```
Fig. 9. Code excerpt from VerifyTokenAsyncTask.java

Once the token is sent by the AsyncTask in the Android client, it is received by the malenah-android API and handled by UserHandler.py. This is because the token was sent via the POST method to the https://malenah-android.appspot.com/user URI. Within UserHandler.py, the post method retrieves the token and verifies its integrity. Recall that we retrieved the OAuth client ID from the Google Developer Console. This client ID is passed to the Google tokeninfo endpoint located at https://www.googleapis.com/oauth2/v3/tokeninfo?id_token=XYZ123 (where XYZ123 is the id token obtained by the Android client). This endpoint checks to see if the token is signed, has not expired, and that the issuer (iss) is accounts.google.com or https://accounts.google.com. If these criteria are satisfied, we pass the token into the oauth2client.client.verify_id_token method to check if the audience (aud) is the app's client ID (obtained previously in the Google Developer Console for malenah-android.appspot.com). If the token is determined to be inauthentic, then a crypt.AppIdentityError is thrown. [5][6][7] Otherwise, the token is authentic and valid, and the user is authenticated (and stored in the NDB datastore if not already in the datastore). This process is illustrated in the code below (Fig. 10).

```
    def post(self, *args, **kwargs):
        #Verify token and store new user
        if self.request.get('id_token'):
            url = 'https://www.googleapis.com/oauth2/v3/tokeninfo?id_token='
            userid = self.verify_token(url, self.request.get('id_token'))
            if userid is not None:
                obj={}
                self.response.set_status(200,'- token verified, user logged in')
                obj['userid']=userid
                obj['status']=self.response.status
                name=self.request.get('name')
                email=self.request.get('email')
                exists_user = self.check_user(userid,name,email)
            else: self.error_status(400,'Invalid token.')
            return
        #rest of code not shown

    def verify_token(self, url, token):
        response = idinfo = userid = None
        try:
            response = urllib2.urlopen(url+token).read()
            idinfo = client.verify_id_token(token, self.CLIENT_ID)
            if idinfo['aud'] not in [self.ANDROID_CLIENT_ID, self.WEB_CLIENT_ID]:
                 raise crypt.AppIdentityError("Unrecognized client.")
            if idinfo['iss'] not in ['accounts.google.com', 'https://accounts.google.com']:
                raise crypt.AppIdentityError("Wrong issuer.")
            userid = idinfo['sub']
        except Exception as e:
            return None
            self.error_status(400,'Invalid token.')
        #token verified
        return userid

    def check_user(self, userid, name, email):
        match = next((eu for eu in self.existing_users if eu['user_id']==userid), None)
        if match: self.response.write(json.dumps(match))
        else:
            properties = {
                'user_id': userid,
                'email': email,
                'name': name,
                'favorites': [],
              }
            self.store_user(properties)
```
Fig. 10. Verification of id token and storing of user if not already in the datastore.

Logging Out



To log out, the user selects “Logout” from the navigation drawer menu.

Fig. 11. In the navigation drawer, tapping on the “Logout” button signs the user out of the application.

Logging out on the Android application involves calling the signOut method on the GoogleApiClient object that was created when the user signed in. Once the user is logged out, the finish() and killProcess() methods are called to close the Android application. (Fig. 12)

```
    @Override public void onNavigationDrawerItemSelected(int position) {
        switch(position){
            case 0: {
                //... code for other fragments not shown
            }case 3: {
                mTitle = "Logout";
                googleSignOut();
                break;
            }
        }
    }
    private void googleSignOut(){
        Auth.GoogleSignInApi.signOut(mGoogleApiClient);
        finish();
        android.os.Process.killProcess(android.os.Process.myPid());
    }
```
Fig 12. Code on the Android application showing implementation of Google sign out. Note that some of the code for the switch statement which loads the other Android fragments are not shown.

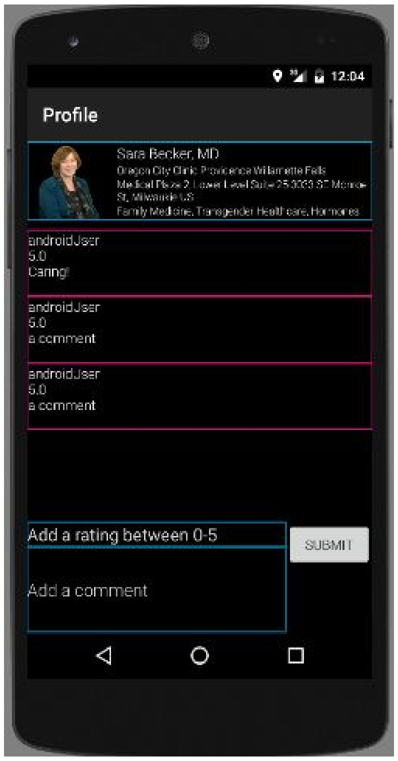
Native Android Front End

Code used to make POSTs to the API

The code used to make HTTP POSTs involves using an AsyncTask, which performs a background operation [15]. This is because network calls, such as POSTs cannot be performed in the main UI thread in an Android application.

The user can write reviews about a healthcare provider by posting a rating and a comment. The Android application then executes an AsyncTask to perform a POST request to the API. Below is the working code for the HTTP POST and an explanation of the code with corresponding UI screen captures.

```
public class PostReviewAsyncTask extends AsyncTask<Void,Void,String> {
    Context context;
    Map<String, String> postParams;

    /**
     * Overloaded constructor
     */
    public PostReviewAsyncTask(Context context, Map<String,String> postParams) {
        this.context = context;
        this.postParams = new LinkedHashMap<String,String>(postParams);
    }

    public PostReviewAsyncTask() {
    }

    public byte[] generatePostData(){
        StringBuilder postData = new StringBuilder();
        for (Map.Entry<String, String> dParam : postParams.entrySet()) {
            if (postData.length() != 0) postData.append('&');
            try {
                postData.append(URLEncoder.encode(dParam.getKey(), "UTF-8"));
                postData.append('=');
                postData.append(URLEncoder.encode(String.valueOf(dParam.getValue()), "UTF-8"));
            } catch (UnsupportedEncodingException e) {
                e.printStackTrace();
            }
        }
        byte[] postDataBytes = null;
        try{
            postDataBytes = postData.toString().getBytes("UTF-8");
        } catch (UnsupportedEncodingException e){
            e.printStackTrace();
        }
        return postDataBytes;
    }

    @Override
    protected String doInBackground(Void... params) {
        /* Set params */
        byte[] postDataBytes = generatePostData();

        /* Set headers and write */
        URL url = null;
        HttpURLConnection conn = null;
        try {
            url = new URL("http://malenah-android.appspot.com/review");
            conn = (HttpURLConnection)url.openConnection();
            conn.setRequestMethod("POST");
            conn.setRequestProperty("Content-Type", "application/x-www-form-urlencoded");
            conn.setRequestProperty("Content-Length", String.valueOf(postDataBytes.length));
            conn.setDoOutput(true);
            conn.getOutputStream().write(postDataBytes);
        } catch (MalformedURLException e) {
            e.printStackTrace();
        } catch(IOException e){
            e.printStackTrace();
        }
        /* Retrieve response */
        Reader in = null;
        String response = new String();
        try {
            in = new BufferedReader(new InputStreamReader(conn.getInputStream(), "UTF-8"));
            response = "";
            for (int c = in.read(); c != -1; c = in.read()){
                response += (char) c;
            }
        } catch(IOException e){
            e.printStackTrace();
        }
        return response;
    }

    protected void onPostExecute(String resp){
        super.onPostExecute(resp);
        JSONObject jResp=null; //parse the string
        try {
            jResp = new JSONObject(resp);
            if(jResp.has("key")) {
                ((ProfileActivity)this.context).postReviewDone(jResp, true);
            } else{
                ((ProfileActivity)this.context).postReviewDone(null, false);
            }
        } catch (JSONException e) {
            e.printStackTrace();
        }
    }
}
```
Fig. 13. Asynchronous task to perform POSTs to the cloud API.



Sara Becker, MD
Oregon City Clinic: Providence Willamette Falls Medical Plaza 2, Lower Level Suite 25 3093 SE Monroe St, Milwaukie US
Family Medicine, Transgender Healthcare, Hormones

androidUser
5.0
Caring!

androidUser
5.0
a comment

androidUser
5.0
a comment

When a user makes a comment, the comment is immediately posted to the Scrollable view above the EditText view. The PostReviewAsyncTask extends the AsyncTask base class. I overloaded the constructor in order to pass in the application's context and POST parameters to post to the API. The output stream writes data as bytes, so the post parameters that were passed into the AsyncTask (as a LinkedHashMap) must be converted to an array of bytes.

doInBackground() is a overridden method of the AsyncTask base class, which performs the main operation of the AsyncTask. In this case, this method opens a URL connection to my API (malenah-android.appspot.com), sets the request headers by specifying the method (POST) and content type (application/x-www.form-urlencoded") and content length, and POSTs to the review URL (malenah-android.appspot.com/review). doInBackground() also reads in a response from the API.

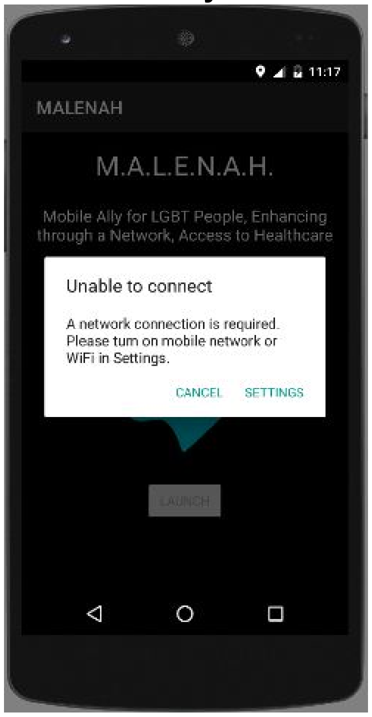
onPostExecute() is another method inherited from the AsyncTask base class. It checks to see if the response contains a 'key' property. If so, the Review entity was successfully created and POSTed to the API.

Mobile Features of the Native Android Application

The MALENAH native Android application uses the following mobile features, which are explained in further detail below:

- Web Connectivity
- Geolocation
- Android AsyncTasks
- Android Services
- Google Maps API
- Drawer Activity
- Toggles

Web Connectivity


11:17
MALENAH
M.A.L.E.N.A.H.
Mobile Ally for LGBT People, Enhancing through a Network, Access to Healthcare
LAUNCH

Unable to connect

A network connection is required. Please turn on mobile network or WiFi in Settings.

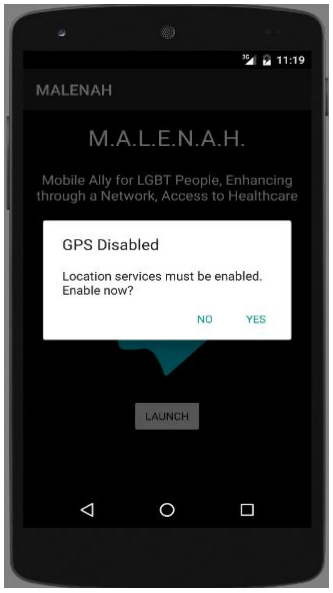
CANCEL SETTINGS

The Android app requires data retrieved from my API at malenah-android.appspot.com. As such, it requires an internet connection to retrieve this data. If the user is not connected to the internet, the user is prompted to activate WiFi or cellular data to allow network access. The code in Fig. 14 shows how the Android application determines whether or not the user is connected to the internet, and how it prompts the user to enable WiFi or cellular data if it is determined that the device is not connected to the internet.

```java
public boolean isFullyConnected(Context context){ //Check if internet is enabled
    if (isNetworkAvailable(context)){
        if(isConnectedMobile(context)){
            return true;
        }
        if(isConnectedWifi(context)){
            return true;
        }
    }
    return false;
}

protected void enableInternet(){ //if internet is not enabled, prompt user to enable
    AlertDialog.Builder builder = new AlertDialog.Builder(this);
    builder.setMessage("A network connection is required. Please turn on mobile network or WiFi in Settings.")
            .setTitle("Unable to connect")
            .setCancelable(false)
            .setPositiveButton("Settings",
                    new DialogInterface.OnClickListener() {
                        public void onClick(DialogInterface dialog, int id) {
                            startActivity(new Intent(Settings.ACTION_SETTINGS));
                        }
                    }
            )
            .setNegativeButton("Cancel",
                    new DialogInterface.OnClickListener() {
                        public void onClick(DialogInterface dialog, int id) {
                            MainActivity.this.finish();
                        }
                    }
            );
    AlertDialog alert = builder.create();
    alert.show();
}
```

Fig. 14. Android Code that checks if the user is connected to the Internet. If not, he/she is prompted to enable WiFi or cellular data.

Geolocation


11:19
MALENAH
M.A.L.E.N.A.H.
Mobile Ally for LGBT People, Enhancing through a Network, Access to Healthcare
LAUNCH

GPS Disabled

Location services must be enabled. Enable now?

NO YES

The Android app also requests the user's geolocation. This geolocation is used to plot the user's location on a Google Map (explained below under Google Map API). I have implemented several contingencies for obtaining the user's location, in the event that the last known location from the user's geolocation cannot be obtained. These methods are described below:

1. **Geolocation**: If the user has not activated geolocation services on his/her phone, he/she will be prompted to do so on start of the app (see code in Fig. 15). If the device's geolocation services is active, but the latitude and longitude cannot be extracted from the geolocation, the user's location will be obtained using one of the contingencies listed below (2 or 3).
2. **ip-api.com API**: If location services is active but the application was unable to obtain the user's last known location (as is the case if an emulator is used, or if an Android device is newly activated), the application calls the ip-api.com API in an AsyncTask (asynchronous task) as a background operation to obtain a location based on his/her current IP address. [14] See code in Fig. 16.
3. **Default location**: If both of the above fail to obtain the user's geolocation, a default location (in downtown Portland) is set. See code in Fig. 16.

```
protected boolean enableLocation(){
    if(ContextCompat.checkSelfPermission(this, Manifest.permission.ACCESS_FINE_LOCATION) !=
PackageManager.PERMISSION_GRANTED) {
        ActivityCompat.requestPermissions(this, new String[]{Manifest.permission.ACCESS_FINE_LOCATION}, 0);
    }
    if (checkLocationPermission()) {
        Log.d("LOCATION (permission)", "user granted permission");
    } else {
        Log.d("LOCATION (permission)", "permission denied");
        return false;
    }
    if  ((Build.VERSION.SDK_INT >= 23) &&
            (ContextCompat.checkSelfPermission(context, Manifest.permission.ACCESS_FINE_LOCATION) !=
PackageManager.PERMISSION_GRANTED) &&
            (ContextCompat.checkSelfPermission(context, Manifest.permission.ACCESS_COARSE_LOCATION) !=
PackageManager.PERMISSION_GRANTED)) {
        return false;
    }
    locationManager = (LocationManager)getSystemService(Context.LOCATION_SERVICE);
    gpsEnabled = locationManager.isProviderEnabled(LocationManager.GPS_PROVIDER); //check GPS status
    networkEnabled = locationManager.isProviderEnabled(LocationManager.NETWORK_PROVIDER); // check network status
    if (!gpsEnabled && !networkEnabled){
        Log.d("LOCATION (GPS)", "disabled, ask user to enable");
        //show dialog to allow user to enable location settings
        AlertDialog.Builder dialog = new AlertDialog.Builder(context);
        dialog.setTitle("GPS Disabled");
        dialog.setMessage("Location services must be enabled. Enable now?").setCancelable(false);
        dialog.setPositiveButton("Yes", new DialogInterface.OnClickListener() {
            public void onClick(DialogInterface dialog, int which) {
                startActivityForResult(new Intent(android.provider.Settings.ACTION_LOCATION_SOURCE_SETTINGS),
SET_LOCATION_REQUEST);
            }
        });
        dialog.setNegativeButton("No", new DialogInterface.OnClickListener() {
            public void onClick(DialogInterface dialog, int which) { //do nothing if user selects no
            }
        });
        dialog.show();
        return false;
    } else if (gpsEnabled) {
        locationManager.requestLocationUpdates(LocationManager.GPS_PROVIDER, 1000, 10, this);
    }else if (networkEnabled){
        locationManager.requestLocationUpdates(LocationManager.NETWORK_PROVIDER,1000,10,this);
    }
    return true;
}
```

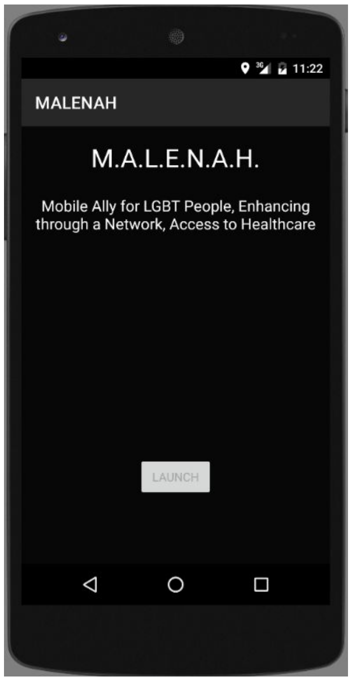
Fig. 15. Code excerpt showing how a prompt is generated if GPS or a Network Connection is unavailable to determine the user's geolocation. If location has not been enabled, the user is prompted to enable it in his/her Android device settings.

```
public class SetLocationFromIPAsyncTask extends AsyncTask<Void, Void, String> {
    private double portlandORLat = 45.52;
    private double portlandORLng = -122.6819;

    private Boolean parseJSONString(String jsonStr) {
        try {
            if (jsonStr != null) {
                JSONObject jsonObj = new JSONObject(jsonStr);
                if (jsonObj != null) {
                    Iterator<String> itr = jsonObj.keys();
                    while (itr.hasNext()) {
                        String key = itr.next();
                        if (key.equals("lat")) {
                            setLat(Double.parseDouble(jsonObj.get(key).toString()));
                        } else if (key.equals("lon")) {
                            setLng(Double.parseDouble(jsonObj.get(key).toString()));
                        }
                    }
                    if (getLat() > Double.NEGATIVE_INFINITY && getLat() > Double.NEGATIVE_INFINITY) {
                        return true;
                    }
                } else {
                    return false; //jsonObj is null
                }
            }
        } catch (JSONException e) {
            Log.e("parseJSONString()", "error");
        }
        return false;
    }

    private HttpURLConnection connectToURL(){
        try {
            urlConnection = (HttpURLConnection)url.openConnection();
            return urlConnection;
        } catch (IOException e) {
            Log.e("LOCATION (error)", "error opening connection");
            return null;
        }
    }

    private String retrieveJSON() throws IOException {
        try {
            BufferedReader inputStream = new BufferedReader(new
InputStreamReader(urlConnection.getInputStream()));
            String jsonStr = inputStream.readLine();
            return jsonStr;
        } catch (IOException e) {
            Log.e("LOCATION (JSON)", "error reading stream, internet connection maybe lost");
        }
        return null;
    }

    private void setFailSafeLocation(){
        setLat(portlandORLat);
        setLng(portlandORLng);
    }

    @Override
    protected String doInBackground(Void... params) {
        urlConnection = connectToURL();
        if(urlConnection != null) {
            String jsonStr = null;
            try {
                jsonStr = retrieveJSON();
                if(!parseJSONString(jsonStr)) setFailSafeLocation();
            } catch (IOException e) {
                e.printStackTrace();
                setFailSafeLocation();
            }
        } else setFailSafeLocation();
        urlConnection.disconnect();
        return "done";
    }
}
```
Fig 16. This AsyncTask executes only if the user's last known geolocation is unknown. This is possible if there was no last known location, even if location services are activated on the Android device (e.g. emulator). In that case, the application will attempt to retrieve the user's location using a GET request to ip-api.com in this AsyncTask.

```
{"as":"AS7922 Comcast Cable Communications, Inc.","city":"Portland","country":"United
States","countryCode":"US","isp":"Comcast Cable","lat":45.5073,"lon":-122.6932,"org":"Comcast
Cable","query":"73.25.153.201","region":"OR","regionName":"Oregon","status":"success","timezone":"America/
Los_Angeles","zip":"97201"}
```

Fig. 17. Example API response from http://ip-api.com/json

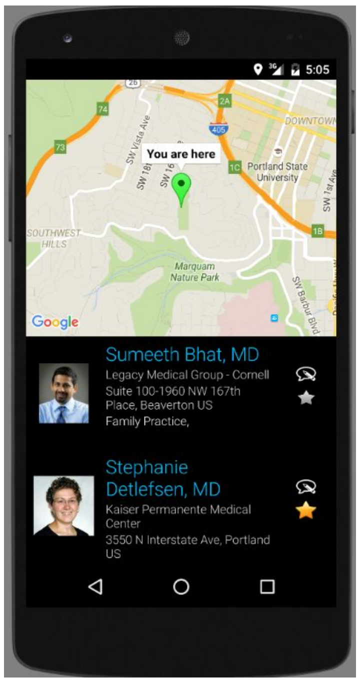
Android AsyncTasks
Asynchronous Tasks allows background operations to be performed without threads/handlers. Operations such as network calls (e.g. HTTP POST and HTTP GET) cannot be performed on the main UI thread, so they must be either performed as an Android AsyncTask or an Android Service [15]. My Android application performs several AsyncTasks, to verify the id token for user authentication (POST), to retrieve healthcare provider demographics (GET), to post reviews (POST), to update a user's name and/or email (POST), and to update a user's favorites (POST).

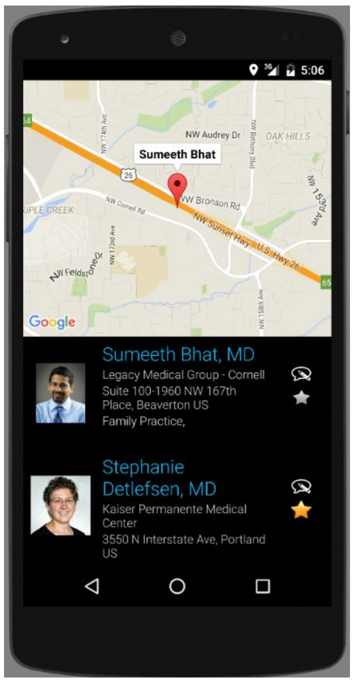
Android Services

11:22
MALENAH
M.A.L.E.N.A.H.
Mobile Ally for LGBT People, Enhancing through a Network, Access to Healthcare
LAUNCH

An Android Service performs operations continuously in the background without a user interface. [16] I use an Android service to obtain all of the provider information using a GET request to malenah-android.appspot.com/provider API prior to allowing the user to proceed to the next screen. Note that the "launch" button is inactive until the user has granted network and location access, all of the healthcare provider data from the malenah-android.appspot.com has been retrieved and the user has been successfully logged in. Once the data has been retrieved from the API, it is broadcasted back to MainActivity (the home screen). [16]

```
public class FetchAllDataService extends Service {
    private String filterStr = "com.watsonlogic.malenah.malenah3.providers";
    private String jsonResponse = null;
    private URL url;
    private HttpURLConnection urlConnection;

    public FetchAllDataService() {
    }

    /* Retreive data from database and broadcast it back to MainActivity */
    @Override
    public int onStartCommand(final Intent intent, int flags, int startId){
        try {
            Runnable r = new Runnable() {
                @Override
                public void run() {
                    try {
                        url = new URL("http://malenah-android.appspot.com/provider");
                        urlConnection = (HttpURLConnection)url.openConnection();
                    } catch (IOException e) {
                        e.printStackTrace();
                    }
                    try {
                        BufferedReader inputStream = new BufferedReader(new
InputStreamReader(urlConnection.getInputStream()));
                        String jsonResponse = inputStream.readLine();
                        Intent intent = new Intent();
                        intent.setAction(filterStr);
                        intent.putExtra("providers", jsonResponse);
                        sendBroadcast(intent);
                    } catch (IOException e) {
                        Log.e("FETCH (JSON)", "error reading stream, internet connection maybe lost");
                    } finally {
                        urlConnection.disconnect();
                    }
                }
            };
            Thread getDefaultEventAllBeersThread = new Thread(r);
            getDefaultEventAllBeersThread.start();
            return Service.START_STICKY;
        } catch(Exception e){
            e.printStackTrace();
            return Service.START_NOT_STICKY;
        }
    }
}
```
Fig. 18. Android Service to retrieve provider data from MainActivity (the home screen).

Google Maps API



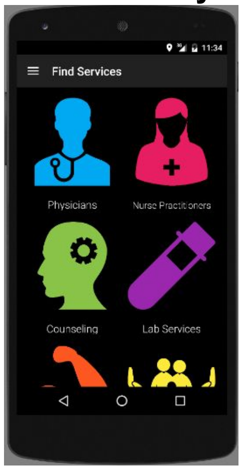


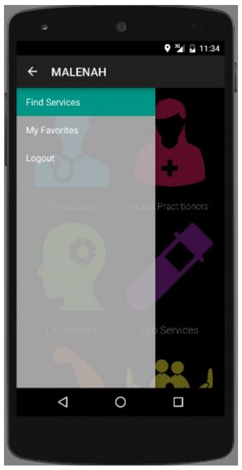
My application makes use of the Google Maps API to show the user's location, as well as location of healthcare providers. As mentioned before, the application obtains the user's geolocation. This geolocation is then used to plot the user's location on the Google map, using the location object's latitude and longitude. In addition to showing the user's location on the map, clicking on a healthcare provider in the list view animates the map camera and centers the camera on the particular provider's marker on the map. The relevant code excerpt for this functionality is provided in Fig 19.

```
public void placeUserMarker() {
    if (location != null && userLatLng != null) {
        map.moveCamera(CameraUpdateFactory.newLatLngZoom(userLatLng, 14));
        userMarker = map.addMarker(new MarkerOptions()
                .position(userLatLng)
                .icon(BitmapDescriptorFactory.defaultMarker(BitmapDescriptorFactory.HUE_GREEN))
                .title("You are here"));
        updateMapCenter(userLatLng,userMarker);
    } else {
        map.moveCamera(CameraUpdateFactory.newLatLngZoom(defaultLatLng, 14));
        userMarker = map.addMarker(new MarkerOptions()
                .position(defaultLatLng)
                .icon(BitmapDescriptorFactory.defaultMarker(BitmapDescriptorFactory.HUE_GREEN))
                .title("You are here"));
        updateMapCenter(defaultLatLng,userMarker);
    }
}

public void placeItemMarkers() {
    if (rowItems != null && rowItems.size() > 0)
        for (RowItem ri : rowItems) {
            ri.setMapMarker(map.addMarker(new MarkerOptions()
                    .icon(BitmapDescriptorFactory.defaultMarker(BitmapDescriptorFactory.HUE_RED))
                    .position(new LatLng(ri.getLatitude(), ri.getLongitude()))
                    .title(ri.getFirstName()+ " "+ri.getLastName())));
        }
}

protected void updateMapCenter(LatLng l, Marker marker) {
    map.animateCamera(CameraUpdateFactory.newLatLngZoom(l, 14));
    marker.showInfoWindow();
}
```

Fig. 19. This code excerpt shows how the user marker and the healthcare providers' markers are placed on the map. If the geolocation was obtained via the Android Location Manager (meaning that user's last known location was obtained successfully via GPS or Network connection or via ip-api.com/json), then the user's marker is placed based on that location. However, if the user's location is null (unsuccessfully obtained through GPS, network or ip-api.com/json), then a default location in downtown Portland is set. In addition, the updateMapCenter() method centers the map when a user clicks on a healthcare provider in the list view.

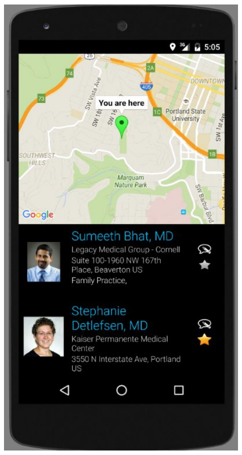
Drawer Activity





The drawer is a UI element that acts like a navigation sleeve/shelf. It is implemented using Android fragments, which are reusable modules that each have their own lifecycle. Each option in the navigation sleeve is a separate fragment. [17] [18]

Toggles



Sumeeth Bhat, MD
Legacy Medical Group - Cornell
Suite 100-1960 NW 167th Place, Beaverton US
Family Practice,

Stephanie Detlefsen, MD
Kaiser Permanente Medical Center
3550 N Interstate Ave, Portland US

Toggle buttons are included to add a provider to the user's favorites list, or remove a provider from a user's favorite list. If the star icon beside the provider in the listview is yellow, this means that the provider is currently in the user's favorites list. If the star icon is grey, this means that the provider is not in the user's favorites list. The code in Fig. 20 shows how the toggle is set based on the user's favorites ArrayList. An AsyncTask is execute each time the toggle is checked or unchecked. If the toggle is checked, then the AsyncTask calls the API with the "post-action" parameter set to "add-favorite". Conversely, if the toggle is unchecked, then the AsyncTask calls the API with the "post-action" parameter set to "remove-favorite."

```java
ToggleButton favoriteToggle = (ToggleButton)customRow.findViewById(R.id.favoriteToggle);
favoriteToggle.setChecked(false);
if(user != null) {
    favoriteToggle.setVisibility(View.VISIBLE);
    if (userFavorites != null || userFavorites.size() > 0) {
        for (RowItem f : userFavorites) {
            if (f.getId() == item.getId()) {
                favoriteToggle.setChecked(true);
            }
        }
    }
favoriteToggle.setOnCheckedChangeListener(new CompoundButton.OnCheckedChangeListener() {
    @Override public void onCheckedChanged(CompoundButton buttonView, boolean isChecked) {
        Log.d("CustomAdapter isChecked", item.getId() + " " + item.getFirstName() + " " + isChecked);
        Map<String, String> postParams = new LinkedHashMap<>();
        if (isChecked) { //user favorited the provider
            postParams.put("post_action", "add_favorite");
        } else { //user unfavorited the provider
            postParams.put("post_action", "remove_favorite");
        }
        postParams.put("favorites[]", String.valueOf(item.getId()));
        postParams.put("user_id", user.getUserId());
        new UpdateUserAsyncTask(postParams).execute();
        }
    });
} else {
    favoriteToggle.setVisibility(View.INVISIBLE);
}
```

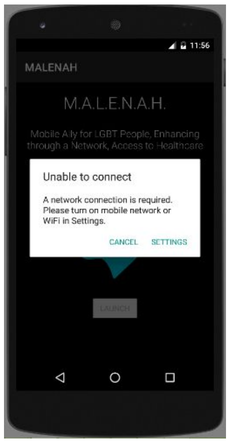
Fig. 20. Code showing how the toggles affect the post parameters sent to the API.

Error Handling

The application implements several mechanisms for error checking:

1. Checking if an internet connection is enabled on the Android device
2. Checking if location services is enabled on the Android device
3. Checking for invalid user input (empty or incorrect input)

1. **Check if internet connectivity (WiFi or cell data) is enabled on the Android device**

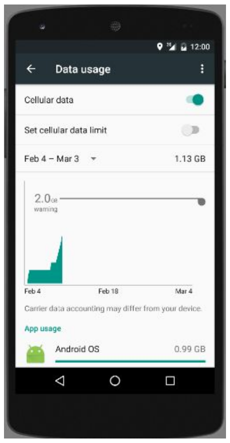

MALENAH

M.A.L.E.N.A.H.

Mobile Ally for LGBT People, Enhancing through a Network, Access to Healthcare

Unable to connect

A network connection is required. Please turn on mobile network or WiFi in Settings.



Cellular data

Set cellular data limit

Feb 4 – Mar 3 1.13 GB

Carrier data accounting may differ from your device.

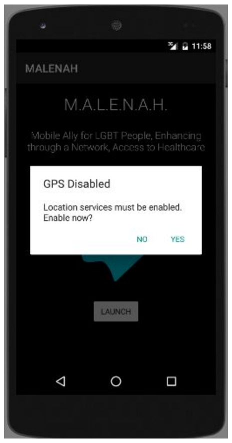
App usage

Android OS 0.99 GB

In the Android code, isFullyConnected() calls several submethods (isNetworkAvailable()), isConnectedMobile(), isConnectedWiFi()) to check for cell data or WiFi connection. If the user's mobile network or WiFi is not enabled, the user will be prompted to activate it in the device's settings menu. enableInternet() displays an alert to the user to enable their internet connection. Either WiFi or cellular data is accepted by this application. Once internet connectivity is enabled, the user is able to proceed. The code excerpt in Fig. 21 shows how this functionality is implemented.

```
if(!isFullyConnected(getApplicationContext())) { //check connection
    Log.d("NETWORK","not connected");
    enableInternet();
    return;
}else{ //internet connected
    getData();
    Log.d("NETWORK","connected");
    if (!enableLocation()) { //check Location
        return;
    }
}

public boolean isFullyConnected(Context context){
    if (isNetworkAvailable(context)){
        Log.d("NETWORK","available");
        if(isConnectedMobile(context)){
            Log.d("NETWORK","mobile connected");
            return true;
        }
        if(isConnectedWifi(context)){
            Log.d("NETWORK","wifi connected");
            return true;
        }
    }
    return false;
}

//check if network connection is available
public boolean isNetworkAvailable(Context context) {
    ConnectivityManager cm = (ConnectivityManager) context.getSystemService(Context.CONNECTIVITY_SERVICE);
    NetworkInfo activeNetwork = cm.getActiveNetworkInfo();
    if (activeNetwork != null && activeNetwork.isConnected()) {
        return true;
    } else{
        return false;
    }
}

//check if cell data is enabled
public boolean isConnectedMobile(Context context){
    ConnectivityManager cm = (ConnectivityManager) context.getSystemService(Context.CONNECTIVITY_SERVICE);
    NetworkInfo info = cm.getActiveNetworkInfo();
    if(info != null && info.isConnected() && info.getType() == ConnectivityManager.TYPE_MOBILE) {
        return true;
    }
    return false;
}

//check if wireless connection is enabled
public boolean isConnectedWifi(Context context){
    ConnectivityManager cm = (ConnectivityManager) context.getSystemService(Context.CONNECTIVITY_SERVICE);
    NetworkInfo info = cm.getActiveNetworkInfo();
    if (info != null && info.isConnected() && info.getType() == ConnectivityManager.TYPE_WIFI) return true;
    return false;
}

protected void enableInternet(){
    AlertDialog.Builder builder = new AlertDialog.Builder(this);
    builder.setMessage("A network connection is required. Please turn on mobile network or WiFi in Settings.")
            .setTitle("Unable to connect")
            .setCancelable(false)
            .setPositiveButton("Settings",
                    new DialogInterface.OnClickListener() {
                        public void onClick(DialogInterface dialog, int id) {
                            //Intent i = new Intent(Settings.ACTION_SETTINGS);
                            startActivity(new Intent(Settings.ACTION_SETTINGS));
                        }
                    }
            )
            .setNegativeButton("Cancel",
                    new DialogInterface.OnClickListener() {
                        public void onClick(DialogInterface dialog, int id) {
                            MainActivity.this.finish();
                        }
                    }
            );
    AlertDialog alert = builder.create();
    alert.show();
}
```
Fig. 21. In MainActivity.java (the home screen), I check to see if Internet is enabled on the Android device. If not, the user is prompted to enable their internet connection (either WiFi or cellular data).

2. **Check if the user has granted permission to use location services on their Android device**

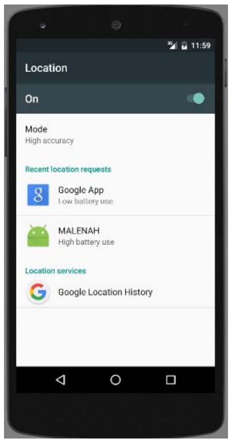

MALENAH

M.A.L.E.N.A.H.

Mobile Ally for LGBT People, Enhancing through a Network, Access to Healthcare

GPS Disabled

Location services must be enabled. Enable now?


Location

On

Mode
High accuracy

Recent location requests

Google App
Low battery use

MALENAH
High battery use

Location services

Google Location History

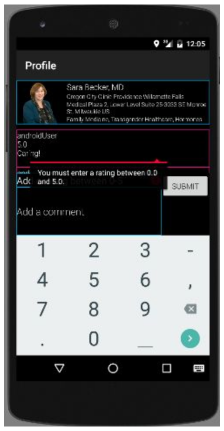
If the user's location services has not been enabled, the user will be prompted to enable it in the location settings.

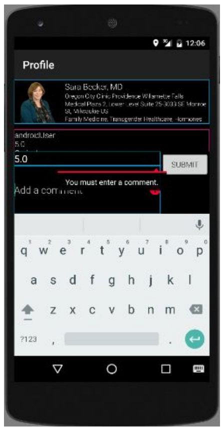
Once the user has enabled location services, he/she can proceed with using the app.

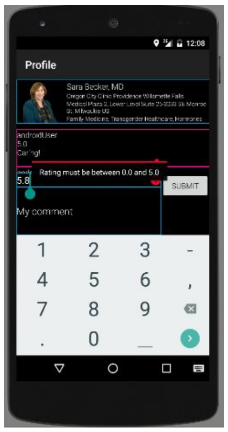
Fig. 22 shows how the user is prompted to enable their location services if it is not already enabled.

```
protected boolean enableLocation(){
    if(ContextCompat.checkSelfPermission(this, Manifest.permission.ACCESS_FINE_LOCATION) !=
PackageManager.PERMISSION_GRANTED) {
        ActivityCompat.requestPermissions(this, new String[]{Manifest.permission.ACCESS_FINE_LOCATION}, 0);
    }
    if (checkLocationPermission()) {
        Log.d("LOCATION (permission)", "user granted permission!");
    } else {
        return false;
    }
    if  ((Build.VERSION.SDK_INT >= 23) &&
            (ContextCompat.checkSelfPermission(context, Manifest.permission.ACCESS_FINE_LOCATION) !=
PackageManager.PERMISSION_GRANTED) &&
            (ContextCompat.checkSelfPermission(context, Manifest.permission.ACCESS_COARSE_LOCATION) !=
PackageManager.PERMISSION_GRANTED)) {
        return false;
    }
    locationManager = (LocationManager)getSystemService(Context.LOCATION_SERVICE);
    gpsEnabled = locationManager.isProviderEnabled(LocationManager.GPS_PROVIDER); //check GPS status
    networkEnabled = locationManager.isProviderEnabled(LocationManager.NETWORK_PROVIDER); // check network
status
    if (!gpsEnabled && !networkEnabled){
        Log.d("LOCATION (GPS)", "disabled, ask user to enable!");
        //show dialog to allow user to enable location settings
        AlertDialog.Builder dialog = new AlertDialog.Builder(context);
        dialog.setTitle("GPS Disabled");
        dialog.setMessage("Location services must be enabled. Enable now?").setCancelable(false);

        dialog.setPositiveButton("Yes", new DialogInterface.OnClickListener() {
            public void onClick(DialogInterface dialog, int which) {
                startActivityForResult(new Intent(android.provider.Settings.ACTION_LOCATION_SOURCE_SETTINGS),
SET_LOCATION_REQUEST);
            }
        });

        dialog.setNegativeButton("No", new DialogInterface.OnClickListener() {
            public void onClick(DialogInterface dialog, int which) { //do nothing
            }
        });
        dialog.show();
        return false;
    } else if (gpsEnabled) {
        locationManager.requestLocationUpdates(LocationManager.GPS_PROVIDER, 1000, 10, this);
        Log.d("LOCATION (GPS)", LocationManager.GPS_PROVIDER);
    }else if (networkEnabled){
        locationManager.requestLocationUpdates(LocationManager.NETWORK_PROVIDER,1000,10,this);
        Log.d("LOCATION (provider)", LocationManager.NETWORK_PROVIDER);
    }
    return true;
}
```
Fig. 22. In MainActivity.java, I check to see if location services (either GPS, network) is enabled on the Android device. If not, I display an alert to prompt the user to enable their location services.

4. Input Validation for POST request


You must enter a rating between 0.0 and 5.0.


You must enter a comment.


Rating must be between 0.0 and 5.0

When the user clicks “Submit” with an empty rating field, an error is displayed to the user. An empty rating is considered invalid and cannot be submitted. An empty comment field is also considered invalid. The user cannot submit an empty comment field. When the users clicked “Submit” with an empty comment field, an error is displayed. A rating of less than 0.0 or greater than 5.0 is considered invalid and cannot be submitted. When the users clicked “Submit” with an invalid rating field, an error is displayed. Fig. 23 shows how this functionality is implemented.

```
public void onClick(View view) {
    /*Check for empty input fields */
    String ratingStr = ratingEditText.getText().toString();
    String commentStr = commentEditText.getText().toString();
    if(TextUtils.isEmpty(ratingStr)){
        ratingEditText.setError("You must enter a rating between 0.0 and 5.0.");
        return;
    }

    if(TextUtils.isEmpty(commentStr)){
        commentEditText.setError("You must enter a comment.");
        return;
    }

    /*Check for invalid input */
    double rating = Double.valueOf(ratingEditText.getText().toString());
    if (rating<0.0 || rating>5.0){
        ratingEditText.setError("Rating must be between 0.0 and 5.0");
        return;
    }

    //POST THIS COMMENT
    Map<String,String> postParams = new LinkedHashMap<>();
    postParams.put("username", "androidUser");
    postParams.put("rating", ratingEditText.getText().toString());
    postParams.put("comment", commentEditText.getText().toString());
    postParams.put("provider", String.valueOf(profile.getId()));
    Log.d("POSTREVIEW","execute async task from ProfileActivity()");
    new PostReviewAsyncTask(ProfileActivity.this, postParams).execute();
    InputMethodManager imm = (InputMethodManager)getSystemService(Context.INPUT_METHOD_SERVICE);
    imm.hideSoftInputFromWindow((null == getCurrentFocus()) ? null : getCurrentFocus().getWindowToken(),
InputMethodManager.HIDE_NOT_ALWAYS);
    commentEditText.setText("");
}
```
Fig. 23. When the user provides empty input, or invalid input (rating < 0.0 or > 5.0), error handling is performed.

Sources

[1] Gay and Lesbian Medical Association. Guidelines for Care of Lesbian, Gay, Bisexual, and Transgender Patients. Gay and Lesbian Medical Association. Web. 23 June 2015.
http://www.rainbowwelcome.org/uploads/pdfs/GLMA%20guidelines%202006%20FINAL.pdf
[2] World Health Organization. "Improving the Health and Well-being of Lesbian, Gay, Bisexual and Transgender Persons." EXECUTIVE BOARD 133rd Session Provisional Agenda Item 6.3 6th ser. EB.133 (2013): Improving the Health and Well-being of Lesbian, Gay, Bisexual and Transgender Persons. WHO, 14 May 2013. Web. 23 June 2015.
http://www.ghwatch.org/sites/www.ghwatch.org/files/B133-6_LGBT.pdf
[3] GlobalHealth.gov. "Lesbian, Gay, Bisexual, and Transgender Health." GlobalHealth.gov. Web. 23 June 2015.
http://www.globalhealth.gov/global-health-topics/lgbt/
[4] HealthPeople.gov. "Lesbian, Gay, Bisexual, and Transgender Health." Lesbian, Gay, Bisexual, and Transgender Health. HealthyPeople.gov, n.d. Web. 23 June 2015.
http://www.healthypeople.gov/2020/topics-objectives/topic/lesbian-gay-bisexual-and-transgender-health
[5] Wolford, Justin. CS496 Winter 2016, Notes
[6] http://www.restapitutorial.com/lessons/whatisrest.html#
[7] https://en.wikipedia.org/wiki/Representational_state_transfer#Architectural_properties
[8] https://developers.google.com/identity/sign-in/android/sign-in#configure_google_sign-in_and_the_googleapiclient_object
[9] https://developers.google.com/identity/sign-in/android/start-integrating#get_your_backend_servers_oauth_20_client_id
[10] https://developers.google.com/identity/sign-in/android/backend-auth
[11] http://oauth.net/articles/authentication/
[12] Sanderson, Dan. Programming Google App Engine with Python. O'Reilly Media, 2015.
[13] https://cloud.google.com/appengine/docs/python/ndb/
[14] http://ip-api.com/
[15] http://developer.android.com/reference/android/os/AsyncTask.html
[16] http://developer.android.com/guide/components/services.html
[17] http://developer.android.com/guide/components/fragments.html
[18] http://developer.android.com/training/implementing-navigation/nav-drawer.html