# Individual Programming Project Report

*An analysis of project design and implementation*

**Prepared by:** Ryan Stillwell
**Tutor:** Arno Leist
**Paper:** 159.333
**Date:** Semester 3, 2015/2016

# Table of Contents

# 1. Abstract

For this project, I chose to create an Android application for use on mobile devices.

The purpose of the application was to provide users with the ability to tag locations with a name, message and image, then use geofences to display notifications when those locations are entered, as well as the ability to share this information with other users. By adapting this functionality I would also provide users with a unique method to engage with the application using trails of tags.

Using standard Google toolkits, I was able to develop and implement most of the functionality required. I was not able to complete the sharing component in time, however I did get data transferring across devices.

# 2. Introduction

As described in paper information provided by Massey University, 159.333 is an individual programming project which requires significant effort on the part of the student. Students are expected to formulate and specify a small programming project, develop an appropriate piece of software to carry out the stated purpose, conduct some experiments with the software and provide a report on the outcomes of their investigations.

This report aims to provide a detailed look at the resulting application and some analysis of the research conducted which influenced the design decisions made.

## 2.1 Background

At the beginning of the project, I had no previous experience with application development in an Android environment, so a large portion of time was dedicated to investigating, understanding and testing smaller bits of functionality, as well as looking at alternative implementations to figure out what I thought worked best.

Ultimately, I would end up having to teach myself Android application development, mostly relying on my ability to apply the knowledge I accumulated over the last two years at Massey University to different scenarios. As a result, a significant of the code in the application has been adopted from the Android developers guide. There is also a small amount of code that was adopted from other sources - mostly from answers provided in StackOverflow, as well as various other tutorial sites/videos. I have tried to keep a record for this in the code itself by including links to the appropriate documentation.

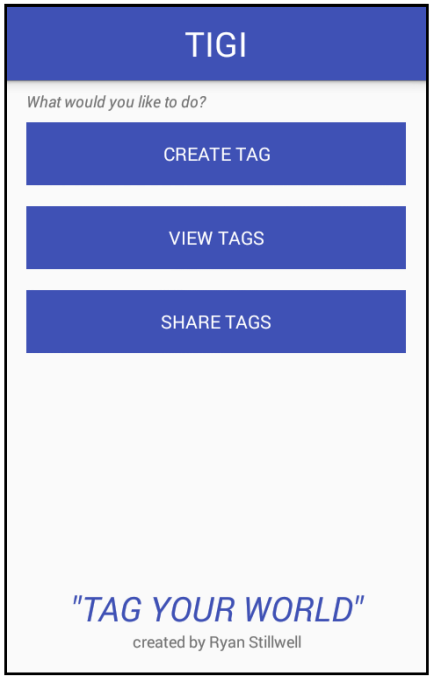
## 2.2 Project Brief

The main goal of this project was to create an application that would allow users to tag locations with some information which would trigger an alert whenever that location is re-entered, and share those tags with other users - essentially creating a proximity alert system with some aspects of social networking.

# 3. Application Details

## 3.1 Overview

The application provides users with three main functions (as seen in the figure below):




1. *Create Tag*
   User selects the type of tag they want to create, sets the location, and adds information to the tag

2. *View Tags*
   User can view information for any previously created tags

3. *Share Tags*
   Users can transfer tag data via Bluetooth to each other

## 3.2 GeoTag Objects

Before going into details in regards to the functionality of the application it's import to get an understanding of GeoTags (or 'tags') and how they are managed, as these are the primary objects on which this application is based.

GeoTags contain the data required to set up and monitor geofences, as well the information need to create relevant notifications and views. A detailed look at the composition of the class reveals more about what is contained in these objects and why that information is relevant.

### 3.2.1 Member Data Fields

```
private String  tagName;
```
The name of the tag. Essentially acts as a title, eg: "Home"

```
private TagType tagType;
```
There are three GeoTag types: SOLO, GROUP and TRAIL.

SOLO tags are standalone and exist individually, with no links to other tags. They are all monitored and are always visible.

GROUP tags exist as a collection of tags, all of which are visible and actively monitored.

TRAIL tags exist as a collection of tags, except only those which have been picked up are visible and monitored. See section XX for more details regarding trails.

This is an enumerated type, with the values SOLO(1), GROUP(2) and TRAIL(3). I set it up this way, as there are many times in the code where it is necessary to check what type of tag is being handled, and a simple call to this enum type in the class made it much easier to test for equality, eg:

if (geotagobject.getTagType() == GeoTag.TagType.SOLO) ...

```
private String  tagGroupName;
```
For GROUP and TRAIL tags, this is the name of the collection they are a part of, as defined by the user. For SOLO tags the value is null as it is not required.

```
private Double  tagLatitude;
private Double  tagLongitude;
```

Together, these values form a point on the Earth which marks the position of the GeoTag object.

```
private Float   tagRadius;
```
Provides value in metres for the geofence transition zone. Radius is centred on the tag position.

```
private Boolean tagActive;
```
State to confirm whether the GeoTag object should be actively monitored or not.

```
private Boolean tagVisible;
```
State to confirm whether the GeoTag object should be visible to display or not.

```
private String  tagMessage;
```
String containing a message, if provide by the user, eg: "16 Deep Creek Rd"

```
private String  tagImagePath;
```
String containing the absolute path on the device for an image captured during the tag creation process.

### 3.2.2 Other Features

One key feature of this class is that it implements the Serializable interface. This makes it really easy to add to and retrieve from intents as an extra.

This class also has a static strings associated with each member variable which act as labels, and are particularly important when tag data is being read from or saved to file.

### 3.2.3 GeoTag Trails

A trail is a collection of GeoTag objects which has special rules applied to it.

When a trail is created, all of the tags are made invisible and only the first tag is set as active. This means geofence transitions will only be monitored for the first tag, however no tags will be visible when trying to view the trail details (see Viewing Tags for more).

When the user enter a geofence that is being monitored they will receive a notification as usual, but there is additional logic that makes the associated tag visible and sets the new tag in the sequence to be actively monitored. This continues until all of the tags have been picked up and become visible.

A good example of what a trail could be used for is a treasure hunt. The message of the first tag could advise the user there is treasure somewhere in the area and provide a clue for the location of the next tag. The user will try to find the location of the next clue, and subsequent tags could continue to provide clues, until the final tag is located (ie: "X marks the spot").

# 4. Managing Data

## 4.1 ArrayLists

For the life of the application, all of the GeoTag objects are stored in three separate ArrayLists, based on their tag type, as follows:

```
private ArrayList<GeoTag> mAllSoloGeoTags;
private ArrayList<ArrayList<GeoTag>> mAllGroupGeoTags;
private ArrayList<ArrayList<GeoTag>> mAllTrailGeoTags;
```

The SOLO tags are all together in one list, whereas the GROUP and TRAIL tags are stored in list with the rest of the tags from the same collection, then those lists are stored in a list of lists.

To make it easier to pass this information around I created the GeoTagContainer class, which stores all three tag lists to a single GeoTagContainer object. Like the GeoTags themselves, this class also implements the Serializable interface, meaning it can be added to and pulled from an intent as a serializable extra.

## 4.2 XML Parser

All of the tag data is stored in an xml file ("tagdata.xml") locally on the device, in the data folder of the application directory. This is private to the application and the directory can be easily located by using the getApplicationContext().getFilesDir() method. Whenever the main activity of the application is created, all of the data is read from the file, and whenever tag data is updated, all of the data is written to the same file. To do this, I created a class that parses XML data (GeoTagXmlParser) to/from this file.

The basic format of the file is as follows:

```
<?xml version="1.0" encoding="utf-8"?>
<tags>
    <tag>
        <name></name>
        <type></type>
        <groupname></groupname>
        <latitude></latitude>
        <longitude></longitude>
        <radius></radius>
        <active></active>
        <visible></visible>
        <message></message>
        <imagepath></imagepath>
    </tag>
</tags>
```

Saving the data to file is relatively easy. A GeoTagContainer object is passed to the GeoTagXMLParser, the tags from all three lists are added to a single list sequentially (which maintains their order). Then, a XmlSerializer object is used to write the data to file, starting with the header and opening label ("<tags>") . After that, each GeoTag object is written in blocks following the above format and the file is closed off with the appropriate xml labels. Ensuring the xml document is well formed is as simple as making sure you are opening and closing labels correctly, for example:

```
xmlSerializer.startTag("", GeoTag.NAME);
xmlSerializer.text(tag.getTagName());
xmlSerializer.endTag("", GeoTag.NAME);
```

Reading the data and manipulating it into the correct structure is more difficult. We need to read all of the data from the file, create GeoTag objects,  then sort those GeoTag objects into the lists/sub-lists.

An XmlPullParser object is created to read the feed. Once the file has been correctly opened, the XmlPullParser is supplied with the opening label (in this case "<tags>") and a loop is initiated to continue pulling data until it reaches the closing label ("<\tags>"). During the loop, if it comes across a new opening label which matches our data (ie: <tag>) then it will pass the next portion of the feed to another method. This method starts a similar loop, which continues until it has found a new opening tag (such as <name>).

At this point, we have located data that needs to be parsed, however a new read method is required for each tag we are interested in, because the label name is be different (and it's likely the return value required will also be different). For example:

```
private String readTagName(XmlPullParser xmlPullParser) throws Exception {

    xmlPullParser.require(XmlPullParser.START_TAG, null, "name");
    String tagName = readText(xmlPullParser);
    xmlPullParser.require(XmlPullParser.END_TAG, null, "name");
    return tagName;

}

private Double readTagLatitude(XmlPullParser xmlPullParser) throws Exception {

    xmlPullParser.require(XmlPullParser.START_TAG, null, "latitude");
    Double tagLatitude = Double.parseDouble(readText(xmlPullParser));
    xmlPullParser.require(XmlPullParser.END_TAG, null, "latitude");
    return tagLatitude;

}
```

Eventually the XmlPullParser will reach the end labels and the stack will collapse, providing us with a single list of all the GeoTag objects that have been parsed. These are then sorted and returned to the calling activity as a GeoTagContainer object.

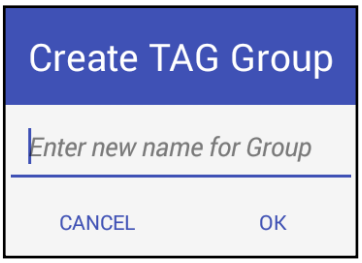
# 5. Functionality

## 5.1 Creating Tags

When a user presses "CREATE TAG" from within the main activity, a dialog appears asking them to select the type of tag they wish to create, as shown below:


This dialog was created using a custom DialogFragment (SelectTagTypeDialog), ArrayAdapter (SelectTagTypeArrayAdapter) & layouts. The main reason for this was to create a consistent style throughout the application. Another for doing this was to enable multiple lines of text associated with each choice in a more flexible way than the standard Android layouts allow.

If the user selects a group or trail tag, another custom dialog will appear asking them to enter a name for the collection, and the title of the dialog will be updated depending on which type of tag is requested:





Once a selection has been made and details have been entered, a new activity (SetGeoTagLocationActivity) will begin. This activity will build a list of tags (which will only contain one tag is SOLO was requested) and return that back to the main activity once it has successfully completed.

The layout for this activity contains several elements. The largest section is a SupportMapFragment, which can be used to navigate and set the location of a tag. This fragment requires an API key, which is obtain from the Google developers console. The key is registered to the application and enables the loading of map objects.

To set the location of a tag, this fragment implements two listeners - OnMapLongClickListener (tap and hold to create new marker) & OnMarkerDragListener (tap and hold/drag marker to change location and the activity implements the interface and provide override methods for both of these.

Users can also set the location by address using the "Find location by address" functionality. Once the user has entered an address and clicks "GO" the activity runs a geocoder task (getFromLocationName), provided by the android.location.Geocoder library, which returns a list of Address objects that can be used to obtain the latitude and longitude of the requested address. If a result is received, the map fragment is moved to the resulting position and a marker is also placed in that spot.

One other interesting feature is the marker information. Using the google.maps.android.ui.IconGenerator library, it is really easy to create a bitmap which can be used as an icon for the map marker. This is a really effective way to get relevant data to show on the map, and also makes things easier to control visually. The code for this implementation is here:

```
IconGenerator iconGenerator = new IconGenerator(mContext);
iconGenerator.setStyle(IconGenerator.STYLE_RED);
Bitmap iconBitmap = iconGenerator.makeIcon(getString(R.string.new_marker_title));

mCurrentMarkerOptions = new MarkerOptions()
        .icon(BitmapDescriptorFactory.fromBitmap(iconBitmap))
        .position(latlng)
        .draggable(true);
```

In the image below, you can clearly distinguish between the new marker and a marker for a tag that has already been created in this group.

The map fragment also implements an OnMarkerClickListener, so once the user has placed a new marker on the map in the desired location, tapping on the marker will produce a dialog to confirm the user does want to add a tag to that location. If so, the latitude, longitude and radius for the new tag will now be set and stored, and a new activity (AddNewGeoTagActivity) will begin to capture the rest of the information required to create a new tag.

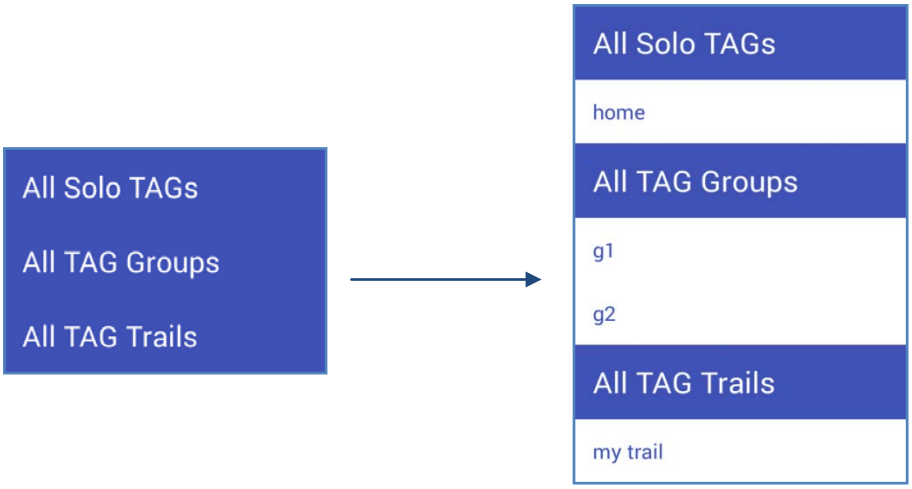
This new activity will capture the name for the tag, as well as a message and image the user wants to display in their notification. Getting the name and message details is fairly straightforward, but getting an image is slightly more complicated.

Handling images can be tricky (see "Working with images" for more details), but to capture an image a new intent (takePictureIntent) is created using the parameter "MediaStore.ACTION_IMAGE_CAPTURE". This is a standard intent action that opens the camera application of the device and allows the user to take a photo, which is then returned in the overridden onActivityResult method. However, before the intent is started, a unique filename is created for the photo, along with details for the path where to store it. To create the unique name, a timestamp is obtained and is added to the filename, which is added to the takePictureIntent as an extra.

Once the user has completed adding tags, tapping the "DONE" button in the top right corner will return the list of the newly created tags to the main activity, where it will be saved to file and monitoring will begin.

## 5.2 Viewing Tags

When a user taps the "VIEW TAGS" button from the main activity, a new activity (ViewAllGeoTagsActivity) is launched and a dialog with an expandable list appears in its collapsed sate. The expandable list view uses a custom adapter (ViewTagsExpandableListAdapter), and details for the expandable list are built from a GeoTagContainer passed to the ViewAllGeoTagsActivity.




When a parent item is clicked, the list expands to show details for either the tag names (for SOLO tags) or the group names (for GROUP and TRAIL tags). An onChildClick method is implemented for the expandable list view element, which builds an intent to launch the ViewGeoTagsOnMapActivity and passes across relevant tag data. This activity implements another map fragment which is populated with markers for each of the GeoTag objects in the list.

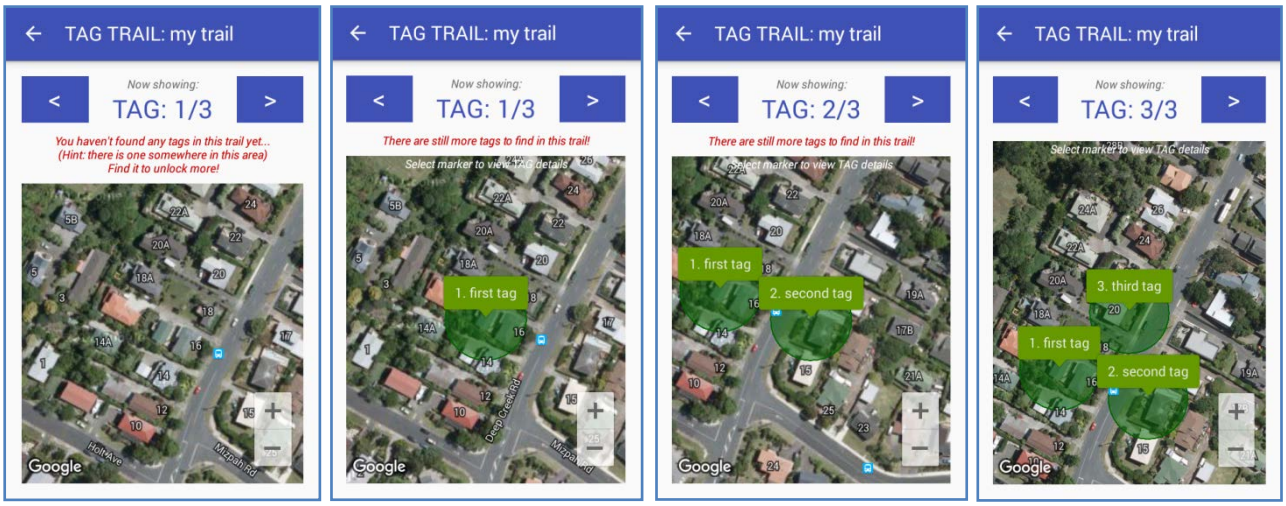
For trails, only the tags that are visible will appear, and notifications regarding the current state are also displayed. Below is an example of a trail with three tags, which shows the details at different stages, as follows:

1. No tags are visible, which is reflected in the text displayed. However, the view of the map will be centred on the first tag of the trail, providing a clue to its location.

2. The first tag is located and is now visible. The text indicates there are still more tags to locate.

3. The second tag is located and is now also visible, but there are still more tags to locate.

4. The final tag is located and all tags in the group are now visible.



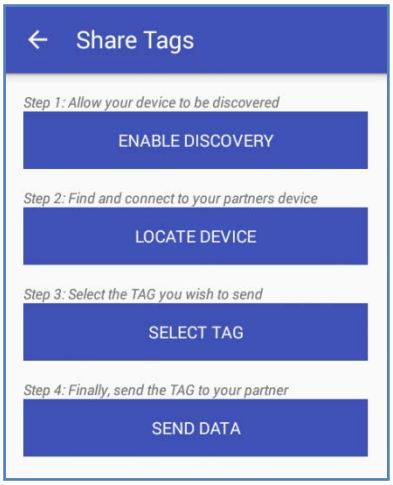

This activity also provides navigation buttons for scrolling between markers and information regarding the number of tags is also presented. When viewing groups and trails, these buttons will wrap around the view of the tag being displayed - ie: when the last tag is currently in view, pressing the next button will move the map back to the first tag. For trails, navigation only occurs between the tags which are visible.

## 5.3 Sharing Tags

When a user taps the "VIEW TAGS" button from the main activity, the ShareGeoTagsActivity is launched. This activity handles all of the events associated with Bluetooth transfers.

The activity immediately checks that the device has Bluetooth capability and also checks it is turned on, providing the user with a prompt before doing so. If Bluetooth is not available, or the user cancels the request to turn it on, the activity will end.

There are four steps associated with the transfer process, which are outlined in the layout of the activity:




### 5.3.1 Enabling Discovery

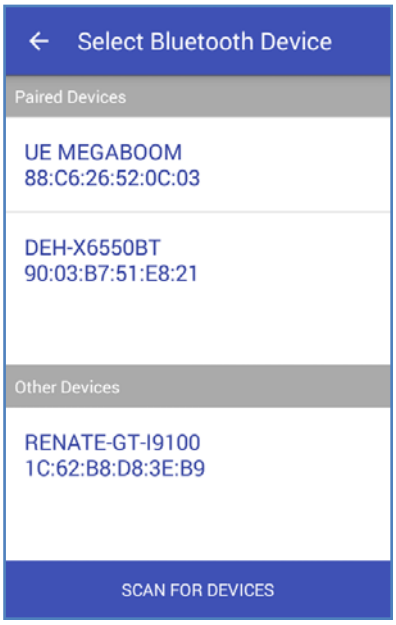
This action is only required for devices that aren't already paired.

For a Bluetooth device to be located, it must be made discoverable. Clicking "ENABLE DISCOVERY" starts an intent which sets makes the device discoverable for 60 seconds.

### 5.3.2 Locating Devices

Both users must locate and select their partners device before they will be able to transfer any data.

This launches the BluetoothDeviceListActivty which populate a list view with the details for all of the currently paired devices. Clicking "SCAN FOR DEVICES" will locate nearby discoverable devices and populate a list view with these details. To do  this, the activity implement a broadcast receiver which updates the ArrayAdapter with the details when it receives a result. This is demonstrated in the image below:




Selecting one of the devices from these lists will return a BluetoothDevice object to the calling activity (ShareGeoTagsActivity). When this result is received, a connection request is initiated. This creates an instance of the BluetoothConnectionService class, which handles connecting the devices and transmissions between devices. This requires a Handler object created from the calling activity, which the BluetoothConnectionService object will post back messages to.

### 5.3.3 Connecting Devices

For a Bluetooth transfer to occur, one device must act as a server (server the data) and the other acts as a client (receiving the data). One special feature of the BluetoothConnectionService is that it manages a client/server connection on both devices, meaning users can send/receive data from either device. The server and client are considered connected to each other when they each have a connected BluetoothSocket on the same RFCOMM channel. This is done by calling createRfcommSocketToServiceRecord on each device using the same UUID (Universally Unique Identifier), as follows:

```
private static final UUID MY_UUID = UUID.fromString("0f8d8aa5-c4b6-47ff-9853-
12e7668abdcb");
...
device.createRfcommSocketToServiceRecord(MY_UUID);
```

These connections are managed by Thread objects which are defined as inner classes in the BluetoothConnectionService class.

When the BluetoothConnectionService starts, it creates an AcceptThread and starts it. This opens a server socket which begins listening for connection requests:

```
public void run() {

    BluetoothSocket socket = null;

    // Keep listening to the server socket if we're not connected
```

```
    while (mState != STATE_CONNECTED) {

        try {  = mmServerSocket.accept();
...
```

A connection request is initiated when the connect() method is invoked on the BluetoothConnectionService object. This create a ConnectThread which attempts to connect to the same socket of the server, then starts a ConnectedThread. The devices are now connected.

### 5.3.4 Transferring Data

Using the socket details, a ConnectedThread creates an InputStream & OutputStream.

During the life of the ConnectedThread, it will listen to the InputStream for incoming byte data. Any information received is read into a byte array buffer, and then posted back to the calling activity (ShareGeoTagsActivity) through the Handler.

To send data, write is invoked on the BluetoothConnectionService, which takes a message (byte array) and passes it through the OutputStream of the ConnectedThread.This message is posted back to the device through the Handler.

When a message is received by a Handler, there is a value associated with it which allows the Handler to interpret whether the device has just read a message from or written a message to another device.

As data is being transferred to a client, the messages received by the Handler are decoded and added to a ByteArrayOutputStream. In my implementation the client device will take any further action until it recognises an "EOF" tag, so all calls to the write method of the BluetoothConnectionService must be followed by a second call to advise the data transfer is complete - for example:

```
mBluetoothConnectionService.write(outStream.toByteArray());
mBluetoothConnectionService.write("|EOF|".getBytes());
```

Since the buffer size for messages coming through is only 1024 bytes implementing it this way allows large files (such as images) to be transferred across devices, provided the file can be converted to a byte array.

# 6. Geofences

## 6.1 Getting Location Updates

Ensuring the appropriate geofences are being monitored accurately requires continuously tracking the location of the users device at regular intervals. As this is part of the main functionality of the applications, this is managed in the main activity. A LocationHandler is created and a connection is made by calling the connectGoogleAPIClient() method.

The LocationHandler class builds a GoogleApiClient and creates a pending location request. Once the connection is invoked the LocationHandler requests the location updates begin and starts listening for the location update callbacks provided by the GoogleAPIClient (handled by the overridden method onLocationChanged).

## 6.2 Geofence Management

The LocationHandler object also receives all of the GeoTag data currently stored in the application (passed through as a parameter by the main activity on instantiation). Each time the location is updated, the LocationHandler loops through all of the GeoTag objects, building a list of tags that are active and within 2km of the user. This is the list of GeoTag objects which need to be monitored.

For each tag in this list, a new geofence is built and added to a list of geofence objects. The code for that component looks like this:

```
mGeofenceList.add(new Geofence.Builder()
                .setRequestId(geoTag.getTagName())
                .setCircularRegion(geoTag.getTagLatitude(),
                                   geoTag.getTagLongitude(),
                                   geoTag.getTagRadius())
                .setExpirationDuration(Geofence.NEVER_EXPIRE)
                .setTransitionTypes(Geofence.GEOFENCE_TRANSITION_ENTER |
                                    Geofence.GEOFENCE_TRANSITION_EXIT)
                .build()
);
```

Important features to note:

- The geofence request ID is the name GeoTag object.
- The transition type being monitored are both entry and exit

## 6.3 Handling Geofence Transitions

Once the list of geofences has been built a PendingIntent is created. A PendingIntent specifies an action to take in the future. In this case, it will launch an IntentService (GeofenceTransitionsIntentService) which will send a notification related to the geofence transition event. A new request is then made to monitor the geofences in the newly created geofence list which will launch the PendingIntent on transition - as follows:

```
// Request to monitor geofences
PendingResult<Status> pendingResult =
LocationServices.GeofencingApi.addGeofences(mGoogleApiClient, mGeofencingRequest,
mPendingIntent);
```

When a geofence transition is received, the PendingIntent is triggered and the GeofenceTransitionsIntentService is launched. The GeofenceTransitionsIntentService identifies the geofence transition does the following:

- *Load all GeoTag data from file*

- *Search records to find GeoTag object associated with the trigger*
  The only information received by this service is the geofence triggering ID, which is associated with the name of one of the GeoTag objects.

- *if GEOFENCE_TRANSITION_ENTER*
  *> send notification(GeoTag object)*
  *> set tag active (false)*
  We only want a notification to be sent if the user is entering a location, but we don't want it to keep being sent, so we deactivate it. This stops notification loops.

- *if GEOFENCE_TRANSITION_EXIT*
  *> set tag active (true)*
  When we leave the area we want it to become active again.

## 6.4 Notifications

At this point, the GeofenceTransitionsIntentService has located the GeoTag object (or objects) which have triggered a geofence entry transition and a notification needs to be sent to the user to alert them.

The GeofenceTransitionsIntentService sendNotification method accepts a list of GeoTag objects as a parameter, creates a new intent (which will launch the DisplayTagDetailsActivity) and add the GeoTag objects as an extra to this intent. As the DisplayTagDetailsActivity is a child of the main activity, a TaskStackBuilder is required to provide proper back navigation. The new intent is added to the TaskStackBuilder, and will be accessible via the getIntent() method. A PendingIntent is then created which contains the entire back stack and will be triggered when the user clicks the notification after it has been received in the device's action bar.

# 7. Future Concerns

## 7.1 Errors/Bugs

When creating groups or trails there is a glitch when the marker for a newly created tag is clicked. This opens the same dialog used when creating a new tag, and if the users selects yes, it will return an invalid result and crash the application. A check needs to be implemented to disable this dialog for markers which are displaying created tags.

Alerts for geofence transitions may be late or not received at all. Depending on the size of the radius and when the device receives location updates, a transition event may not occur. If the application is open, the accuracy will be higher because it will be regularly requesting updates, but when the application closes the location request stops and geofence transitions are managed only by the Android geofence service.

Clashes will occur if the same names are used for the tags. This happens when a geofence transition is triggered, as the application searches through a complete list of GeoTag objects to match the triggering ID against the tag name. This could be fixed by implementing a unique ID for each tag.

It would also be advisable to review the way tag data is stored and accessed, and look at implementing a SQLite database, instead of the current method which is far less efficient.

## 7.2 Bluetooth Data Transfer

Unfortunately I was unable to complete this functionality before the end of the project timeframe. It took a significant amount of time trying to understand and implement the process for connecting devices, as well as figuring out how to transfer images. Eventually I did manage to get this working, but in its current state this application can only send/receive the data and doesn't actually do anything with it.

## 7.3 Edit/Remove GeoTags

Currently there is no functionality to enable tag data to be edited. I would like to implement this as part of the "View Tags" component to allow tag information to be updated, enable/disable monitoring manually, as well as adding tags to groups/trails.

For testing purposes, I created a button to remove all geofences and delete all the tag data. This can be found by swiping up in the main activity, under the tag data log (which was included for testing purposes also). This functionality could be modified to allow tags to be removed.

## 8. Conclusion

It was an exciting project to work on and there is a lot I learnt from it, above and beyond its original scope. I was able to study and train myself on development in an environment that was almost completely foreign to me, which I believe I have been reasonably successful with.

Although I was not able to complete some of the proposed functionality, the functionality related to the core purpose of the application is working as desired. I believe there is still a lot of potential for this application, and will continue development in the future.