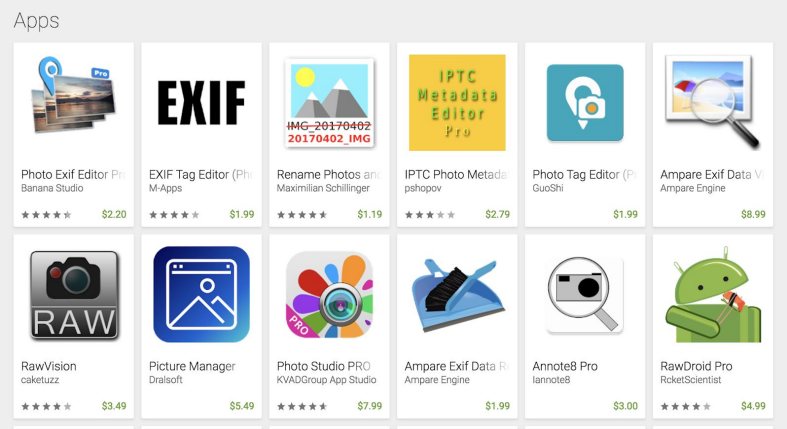
Final Report - EXIF Editor

By Doron Galambos and Vasko Nikolov

Project Statement

Exif Editor is a tool used to view and modify image metadata, known as EXIF. The idea was originally thought of by Doron Galambos. As a software engineer working for a photo sharing tech company, he frequently has to edit metadata for photos for unit testing and automation. There is one main issue with this: editing photos is a hassle, because the necessary tools are on a computer only.

There are some apps that do this, however they are locked behind paywalls.



To solve this issue, we decided to make an app that does exactly what we want – on a the device we will be using it with!

Our app does many things:

a. Allows users to import photos
b. Allows users to launch camera and take photos
c. Allows users to view and edit the metadata in their photos
d. Allows users to view their photos on a map

Though the usage for this app is quite niche, we still believe it's very useful to some.

One possible usage for it is to be able to physically add metadata to images. This can be needed for many reasons. One reason that we've needed this for is because metadata is often wiped when uploading photos to websites. This is due to trying to save space during compression.

Another usage is for people who want to see where they took their photos. Currently, using stock Android galleries, we can only view the location of one photo at a time. Using our app, we can view all the locations at once.

One requirement for our app is that you need a large screen – this means you either need a tablet or a phone with a large screen – for our testing, we used both a Samsung Galaxy Tab A and a Google Pixel XL 2. This is because we physically had these devices for testing. We couldn't use smaller devices, as there was too much information to fit on the screen at once when in editing mode.

To work around this issue, we embedded a ScrollView to allow users to scroll if their screen is too small.

Note: This is only a soft-requirement. The app ***does*** work on other, smaller devices, but the user experience is not as sleek as we'd like. Users can also use our app in landscape mode, too.

One hard requirement is that you must be using Android 8.1, API 27. This app does not work on API 28, as some of the API's used in the code were deprecated in that update. Unfortunately, this is not something we can change at the time.

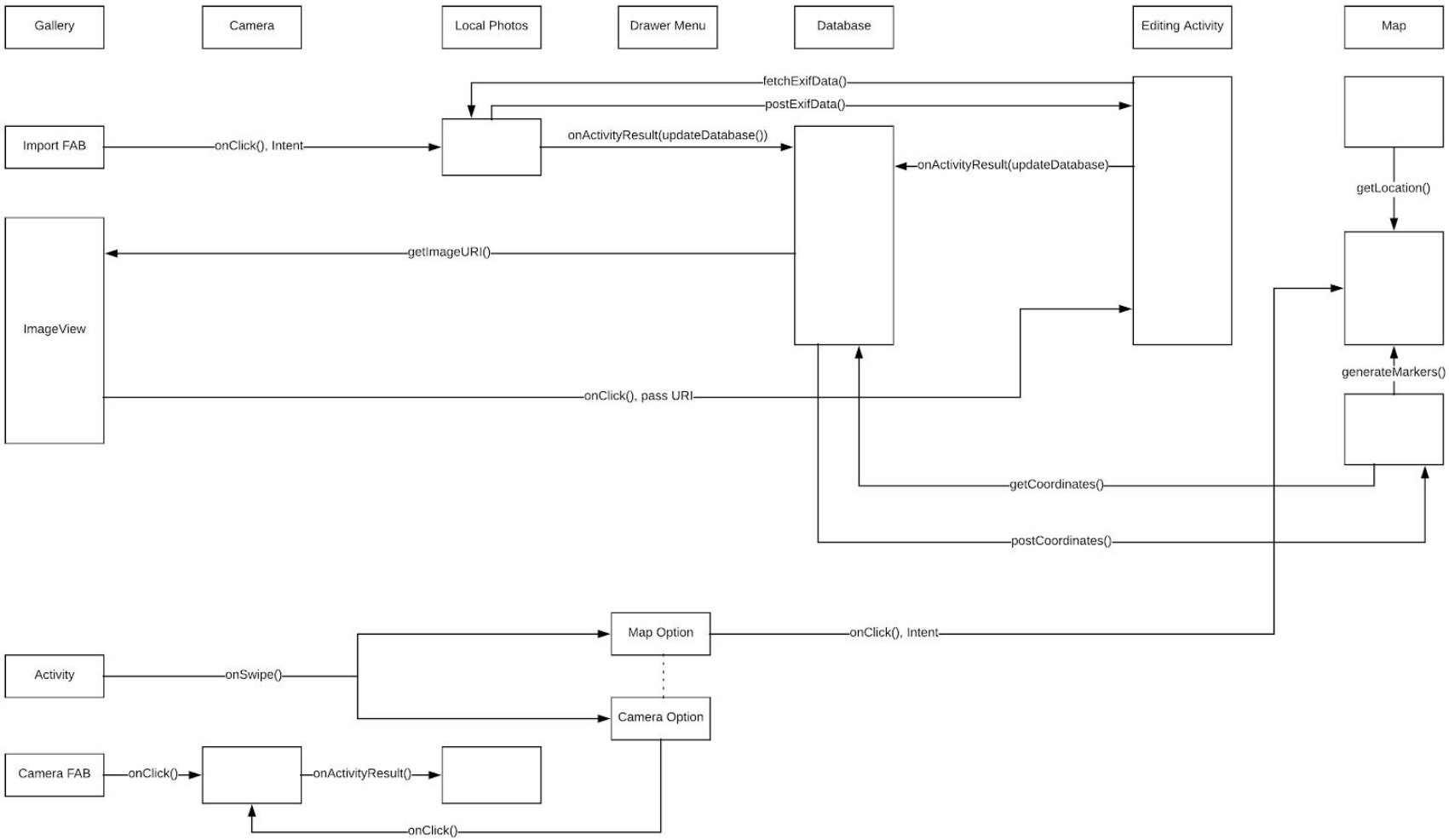
Application Design

Before starting to write our code, we decided to look at similar apps to see what we can learn from them. One thing we noted is that everything was thrown at the user at once. This made the UX (user experience) worse. Because of this, **we decided to go for a simple and neat design.** We didn't want to have redundant and fancy buttons everywhere. We didn't want a hard to see color scheme. We wanted to make a simple tool - that works.

The (high-level) structure of the app goes as follows:

a. Gallery (main screen)
    a. View imported images (redirecting to editing activity via onClick())
    b. Camera Floating Action Button
    c. Import Floating Action Button
    d. Scroll for Drawer Menu
b. Drawer Menu
    a. Camera Button
    b. Map Button
c. EXIF Editing Page
    a. GridView within a ScrollView
        i. Contains 4 columns
            1. 2 TextViews, 1 TextEdit, 1 Button

See next page for a low-level UML Sequence Diagram

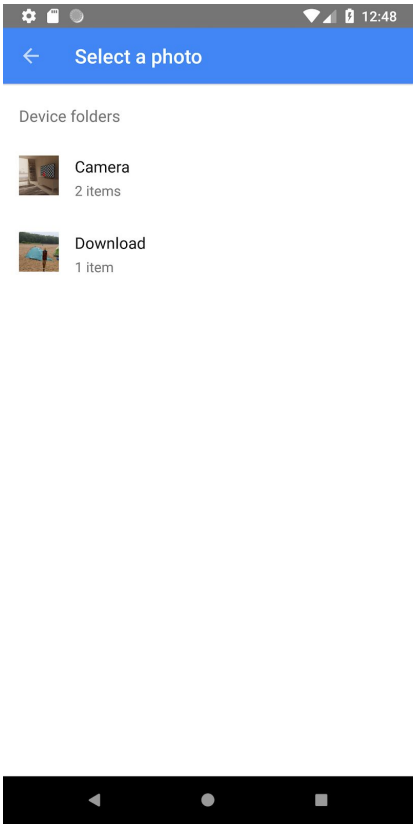

Gallery
Camera
Local Photos
Drawer Menu
Database
Editing Activity
Map
fetchExifData()
postExifData()
Import FAB
onClick(), Intent
onActivityResult(updateDatabase())
onActivityResult(updateDatabase)
getLocation()
getImageURI()
ImageView
generateMarkers()
onClick(), pass URI
getCoordinates()
postCoordinates()
Map Option
onClick(), Intent
Activity
onSwipe()
Camera Option
Camera FAB
onClick()
onActivityResult()
onClick()

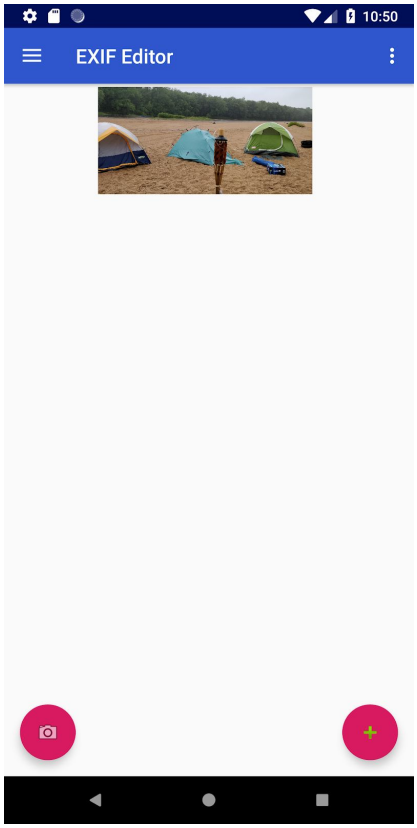
Application Implementation and Evaluation

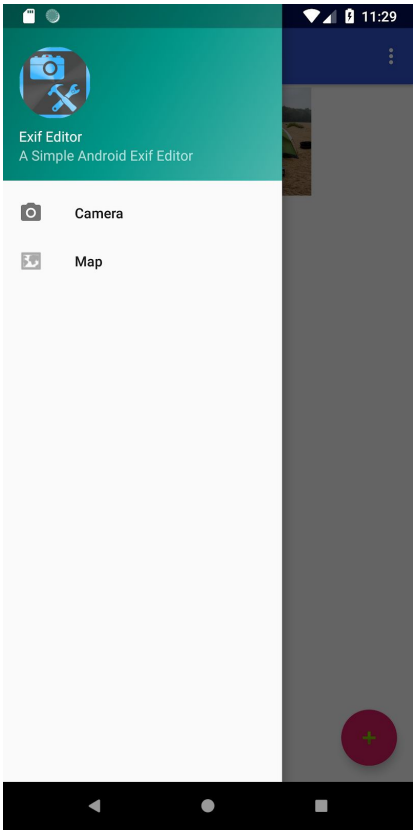
We defined and implemented 4 classes: ViewImgList, ViewImg, MapsActivity, and ImageDatabase. ViewImgList is the main page of the application. It is where the Gallery exists along with the Drawer Menu. In this page, we begin by asking for user permissions that we require. These range from reading/writing on external storage to camera and location.

On this page is where we give the user the ability to import images. To do this, the user must press the import FAB (floating action button) at the bottom right, which will process an intent to open the phone's local images (Intent.*ACTION_PICK*, MediaStore.Images.Media.*INTERNAL_CONTENT_URI*). The user can go through the available folders and choose a single image to import. To import an image, the user just have to click on it. When the user selects an image to import, we then have to fetch the EXIF data along with other info. One issue we ran into was that images were not displaying in their folders when the Import button was pressed. We would be able to import them, but not see them (not in Files nor in our app). **To fix this, we found out that if you open the Google Photos app, it will refresh the files in the phone and we can then view them.** We call insertData() from an ImageDatabase object we initialize and import the URI, filename, latitude and longitude, along with their references, date and time, the camera make and model. This gets appended to the database. The imported image is then displayed in the gallery activity in an ImageView.

We also had to define 2 new methods to get this data. We had to implement a method that takes a URI and returns its respective file path. Similarly, when we get the latitude and longitude - we get a string DMS (Degree Minute Second) format, but we need a double format. We then implemented a method to convert from DMS to decimal latitude and longitude.

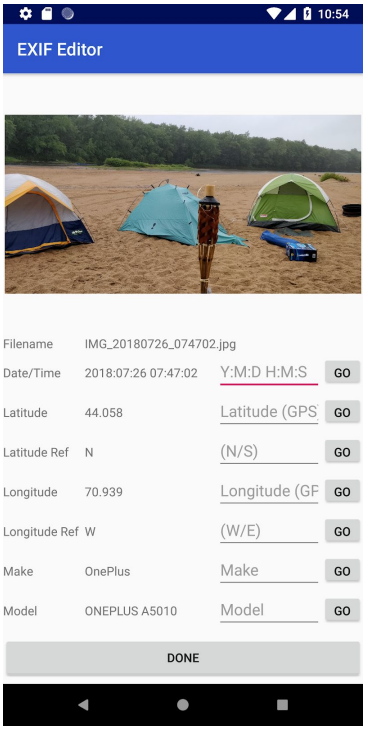






We also allow the user to open the camera. If they open the camera, the application will process an Intent to open the camera (MediaStore.ACTION_IMAGE_CAPTURE) and generate a random filename for it. We then save it wherever Environment.getExternalStorageDirectory() returns and put it inside the DCIM/Camera/ folder. Though it seems trivial, we had a lot of trouble implementing this. We had issues because we had permissions errors due to not being able to set a proper, universal content provider. We also couldn't get the EXIF data to save on this image, either. This is fine because the user can just edit it themselves if they want.

Upon clicking an imported image in the Gallery, we go to ViewImg with an intent to which we add the image URI as a string which is then used in ViewImg to display the image in the ImageView. ViewImg is where the metadata manipulation actually happens. We have a GridView and ImageView within a LinearLayout - all which is within a ScrollView. In here, we have the image displayed up top and all of the EXIF data we give the user ability to view and edit. The user can then input the values they want to change, and if they are valid, they will be updated both in the TextView as well as in the image itself.



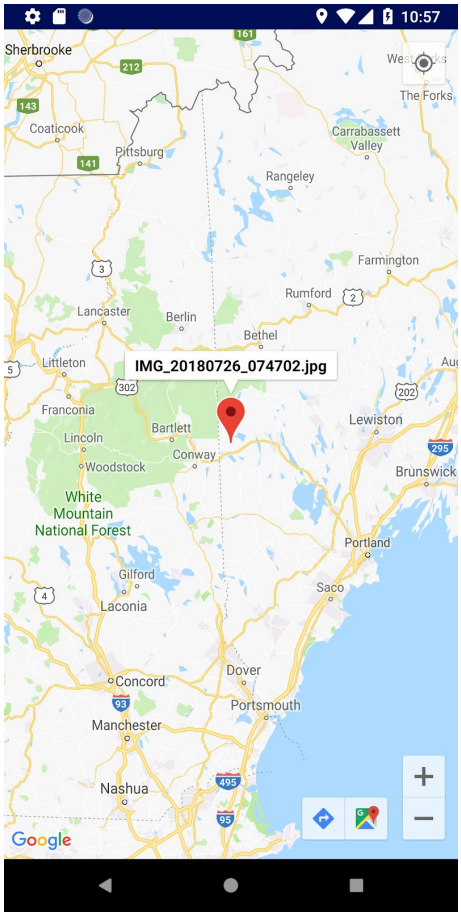
To get the EXIF data, we create an ExifInterface object given the filepath of the image. We then extract all of the information via getAttribute() methods. Similarly, we have a method for each "Go" button that's calls a setAttribute() for that specific tag. This is all quite trivial.

The only two things we had to worry about was GPS coordinates being strings and the filename length. To tackle the GPS issue, we used the same function we had implemented in ViewImgList to convert from DMS to decimal. Then we output that. When we set the GPS coordinates, we take the user input and call a method to convert from decimal to DMS. To tackle the filename issue, we decided to cap the filename

displayed at 40 characters. This way, if the filename length is > 40, then we get the last 37 characters and append those to "..." so we output "...long_filename_goes_here.jpg"

To implement the database, we originally had a JSON implementation. We had issues because reading and rewriting took too long. This was poorly thought out. To tackle this issue, we upgraded for a physical JSON file format to a SQLite implementation. This was a simple implementation that creates a database called "assets.db" to store all of the assets if it is not created. If it is, it simply appends to it. It has 2 main methods to it: insetData() and getAllData(). One inserts a whole new row, given all the 10 input fields. This is referenced in Gallery. The other returns all the data when called. This is referenced in the Map Activity.

To open the map, the user had to open up the Drawer Menu (from the top left corner of the main gallery page) and using an Intent, the app would open up the MapsActivity. When first opened, we iterate over the database to dynamically populate the map with markers based on the latitude and longitude coordinates of each image in the database by calling getAllData() from the database object. We set a base condition that the file must have a valid latitude, longitude and file name in order to generate a pin. Each marker is then displayed on the map as a red pin. When clicked on a marker, a small information bubble displays the image name. To go back to the Gallery screen, the user can press the Android back button. If the user imports another image and then opens the map again, the map would change accordingly reflecting the updated database.



References

We used a lot of built-in classes from Android. Here are the APIs that we referenced:

- https://developer.android.com/reference/android/media/ExifInterface
- https://developers.google.com/maps/documentation/android-sdk/marker
- https://developer.android.com/training/data-storage/sqlite

Experiences and Thoughts

We had high hopes when starting this project, however, since we had to take on 4 other courses, we weren't able to allocate enough time to learn how to and actually implement everything we had hoped. Originally, we wanted to have a whole list of images - this ended up being an issue. We couldn't get the app to dynamically display multiple images while still being stable. Because of this, we couldn't get a search feature to work. We wanted to give the user the ability to search through the Gallery to find images with specific tags: Whether date, or AI generated labels via Clarifai.

Another issue we had is that we couldn't get the database to update upon EXIF modification. Since we only implemented the database near the end, changing everything would take too long. We would probably be able to add this change if we had an extra week or so.

We also wanted to display more on the markers, such as more data than just the file name. We wanted to display a thumbnail of the image, as well as add the markers cluster when close together.

Overall, we believe we did a lot, considering the time we had to implement this was not so long. We are proud of what we accomplished.

If we could change one thing about the class, we would say that we would rather have more, smaller, hands-on group work than one-sided lectures. This would make the course more engaging.
