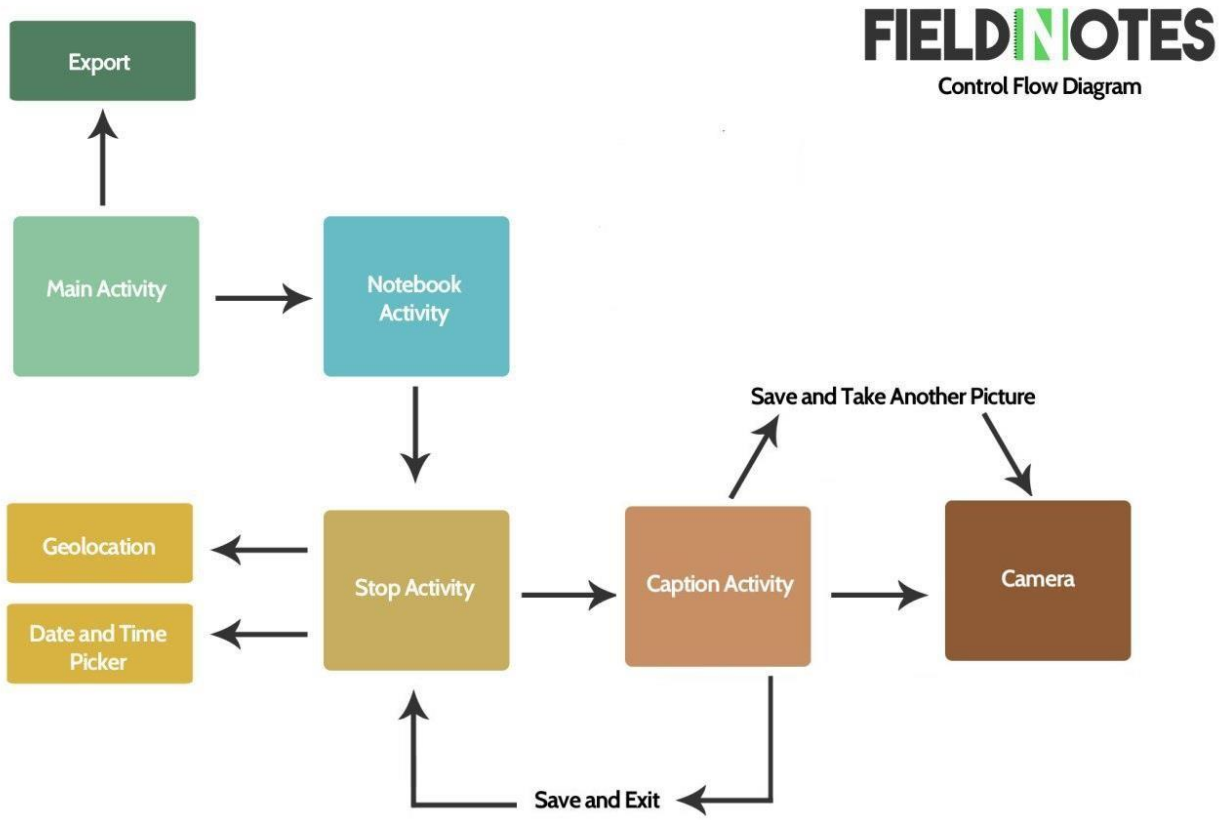
FIELDNOTES

Mobile Application Developer Guide

TABLE OF CONTENTS

Activities and Information Flow



Above is a convenient diagram that shows all the Activities used in Fieldnotes, and below is a quick description of how each one sends information.

Main Activity

From the Main Activity, the user can open/create a Notebook, in which case it simply sends the Unix Time (the Primary Key in the database) of the Notebook to the Notebook Activity through an intent. (Intents are how all information is sent between Activities in this app.)

The Main Activity can also open the Export Activity, where it again simply sends the Unix Time of the Notebook selected.

Notebook Activity

The only thing the user can do here is open the Stops Activity, where it sends the Unix Time of the Stop selected. When the Stop Activity is done, it returns the Unix Time of the Notebook it belongs to, so the Notebook Activity knows which Notebook to load again.

Stop Activity

The Stop Activity can open three other Activities: The Date and Time Picker, the Geolocation, and the Caption Activities. There is no information sent to the Date and Time Picker and the Geolocation Activities. However, the Caption Activity needs the Unix Time of the Stop so it knows which Stop to save to in the database, and also which value to return to the Stop Activity.

Date and Time Picker Activity

This Activity is only ever reached if the user's API version is less than 24. Otherwise, a popup will display in the Stops Activity. This Activity needs no information, and returns, simply enough, the date and time the user selects back to the Stop Activity.

Geolocation Activity

This Activity appears to the user as a loading screen. If it can find the location in ten seconds, it returns to the Stop Activity with the latitude and longitude found. If it cannot find the location, it returns with a failed status and the app simply toasts that the location could not be found.

Caption Activity and Camera Activity

The first time the Caption Activity is called, the screen does not load, but rather starts the Camera Activity, which saves a photo to the phone. The Caption Activity prepares a file in storage on the phone and passes that filepath to the Camera Activity, which saves its picture to that file. We did not implement our own camera, we simply call the Android camera.

Upon the return from the Camera Activity, the picture is displayed to the user. From there, they can choose "Save and Take another Picture," which sends the user back to the Camera Activity with the same functionality, or "Save and Exit," which returns the user back to the Stop Activity.

If the user cancels out of the Camera Activity, it returns a fail status, and the Caption Activity returns to the Stop Activity, again without loading to the screen.

Export

Similar to the Geolocation Activity, this is displayed simply as a loading screen. It's important that this Activity is not interrupted while it's generating the rtf file. While on this Activity, the screen will not time out. When the file is finished generating, it opens an intent to share, which is handled by Android, and then closes the Activity.

Room and SQLite Database

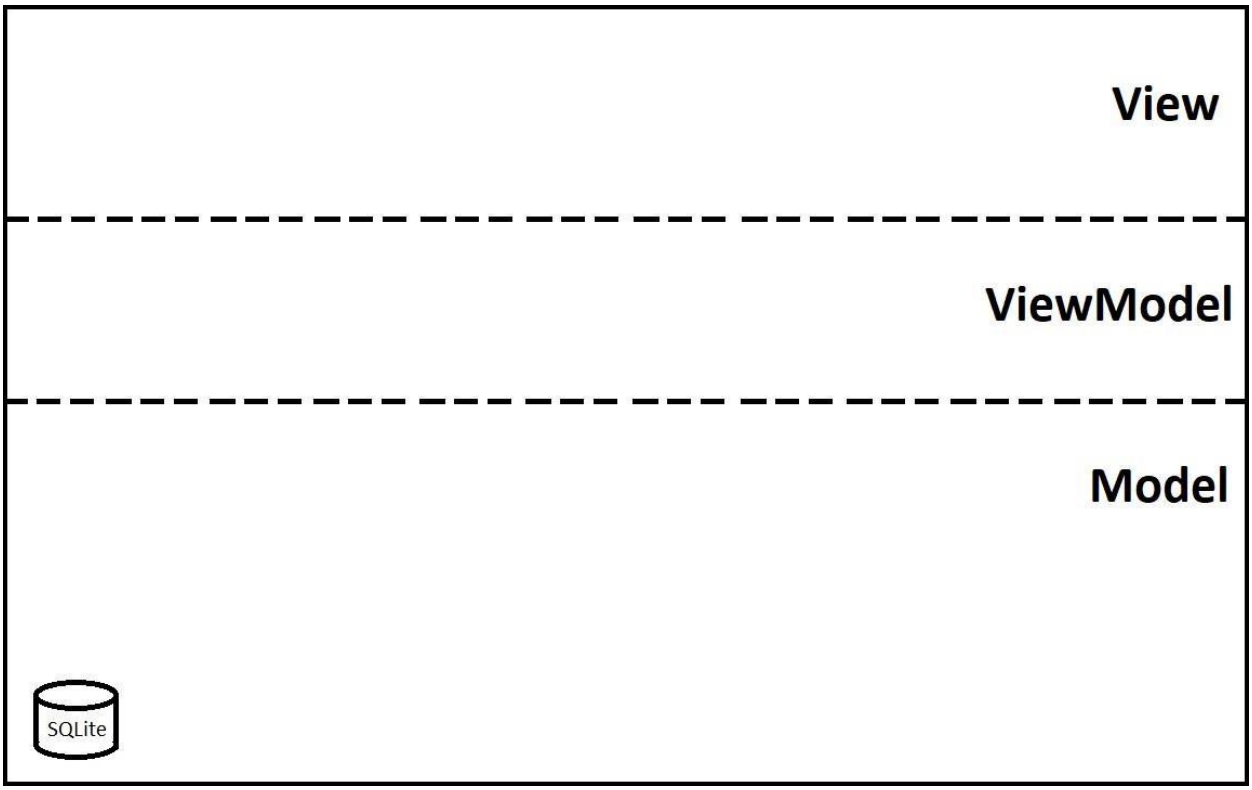
Contents of Database

The database used by Fieldnotes is small and simple. Pictures belong to a Stop, and Stops belong to a Notebook, so we have three tables, one for each. Since the only relationship between entities is a parent/child relationship, we simply connect each child to the parent via primary key, which in all instances in this app is the Unix Epoch Time that the object was created. The only non-intuitive piece of information that the database contains is the pictures. Since whole pictures cannot be saved to the database, we instead use a String variable for each picture's filepath on the user's phone.

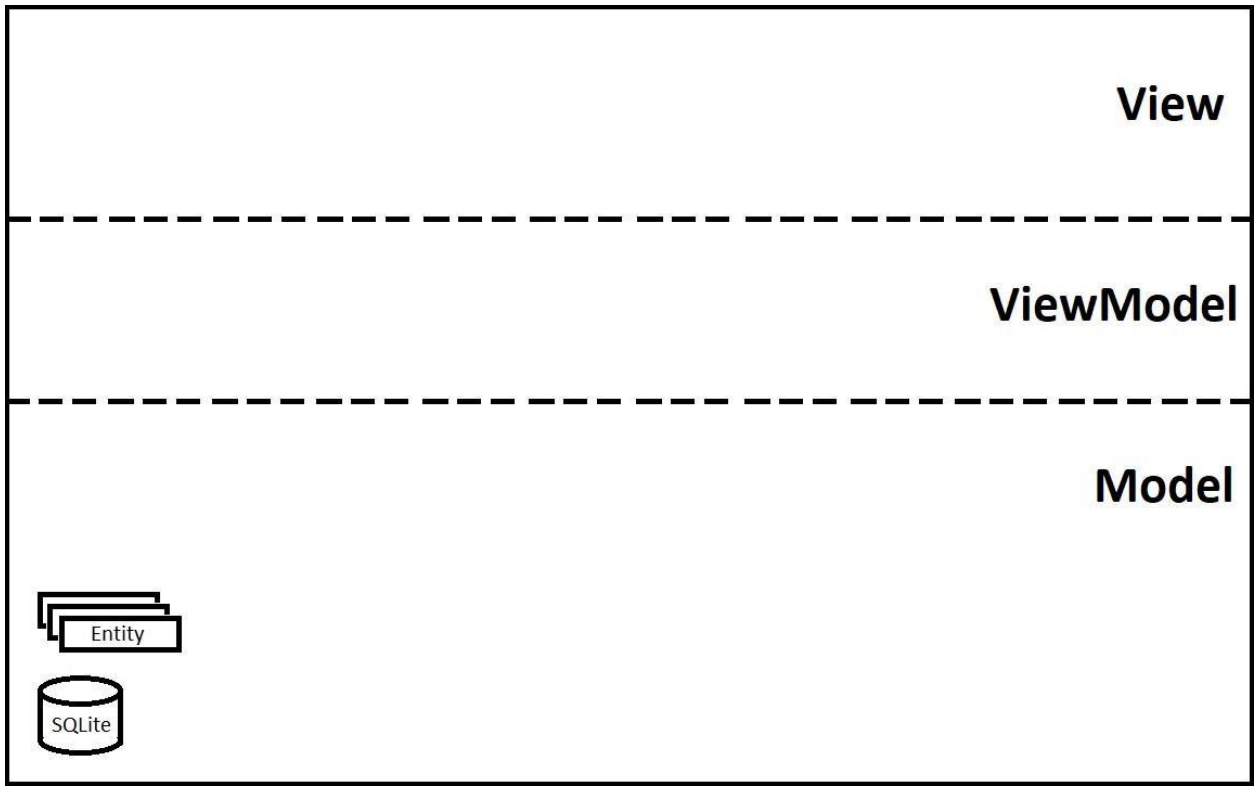
Application Structure

The Fieldnotes app follows the Model View ViewModel (MVVM) application architecture, which is currently Google's best-practice structure for Android app creation. This pattern of app creation is intended to work with Android architecture components to provide a robust design that is modular, contains persistent data, and is able to withstand configuration changes. The main components involved with this are the LiveData class, the ViewModel class, and the Room database library. Each of these will be explained in greater detail as needed.

At the base of this three-tier design is the Model, or data layer. This starts with the SQLite database.

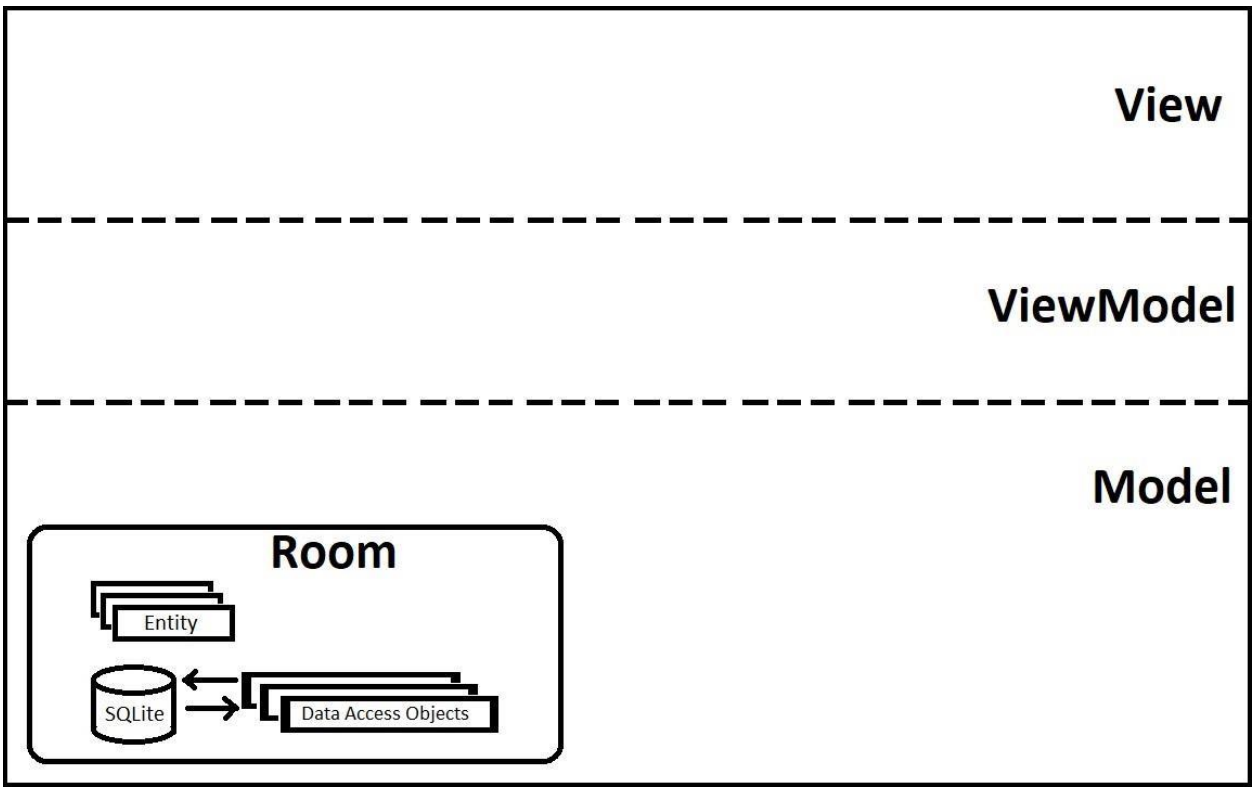


In the past, we would have used a SQLiteOpenHelper class to handle queries and database creation through use of prepared statements, and any errors within the SQL code wouldn't be caught until runtime. However, making use of the Room database library, Fieldnotes minimizes SQL boilerplate code, and even checks the SQL statements at compile time. The plain old Java objects (POJOs) which will be the model of a database table will become what Room refers to as **entities.** For the app, Notebooks, Stops, and pictures each have their own database tables, so the model class for each is tagged as an entity. Specifics of this tagging will be discussed in detail in the database portion of the manual. For now, we'll just continue to talk about MVVM structure.

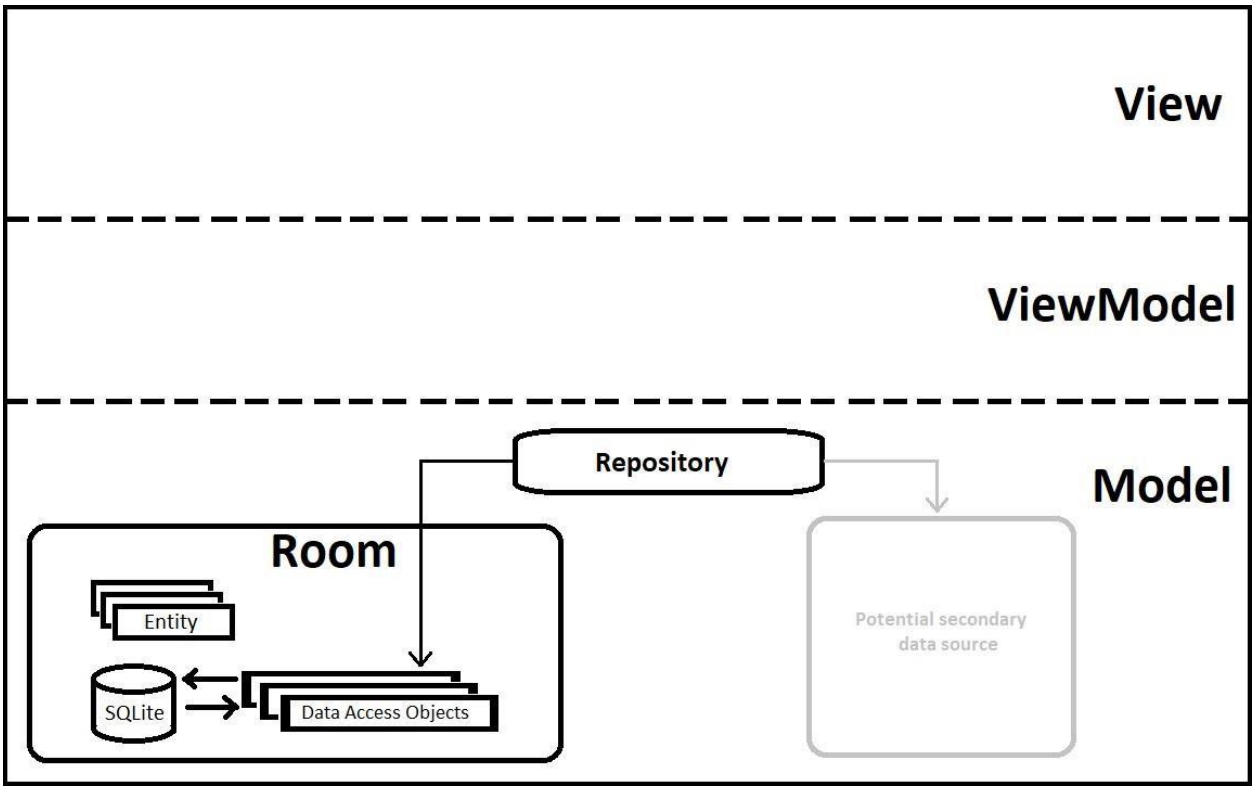


We have our database class, and we have our tables (entities), so now we just need our queries. To do this, Room has **database access objects** (DAOs). These classes will take Java method templates and map them with SQL queries using more of Rooms annotations. Only one Dao is

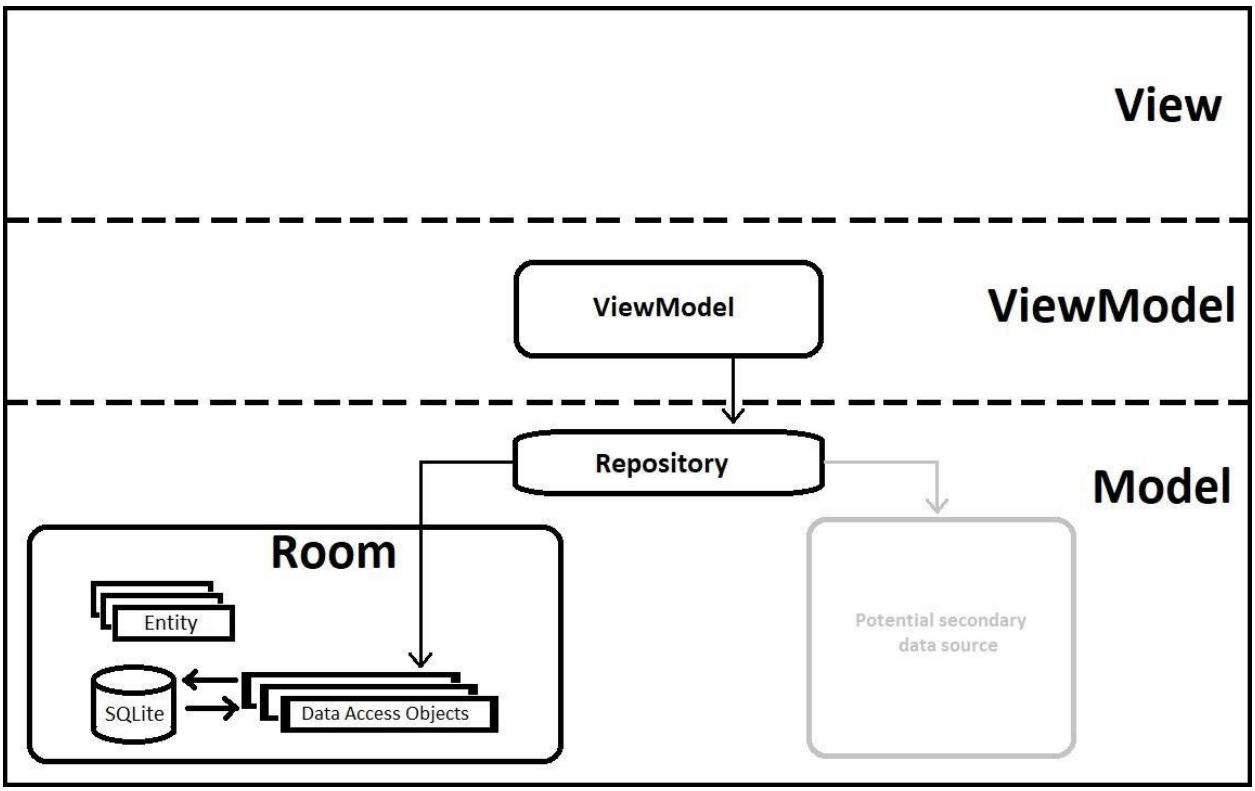
needed for a database, it is, however, a good rule of thumb to have a separate Dao for each type of object in the DB. For Fieldnotes, this meant the NotebookDao, StopDao, and PictureDao classes were created. Collectively, these pieces all come together to create the full Room database library.



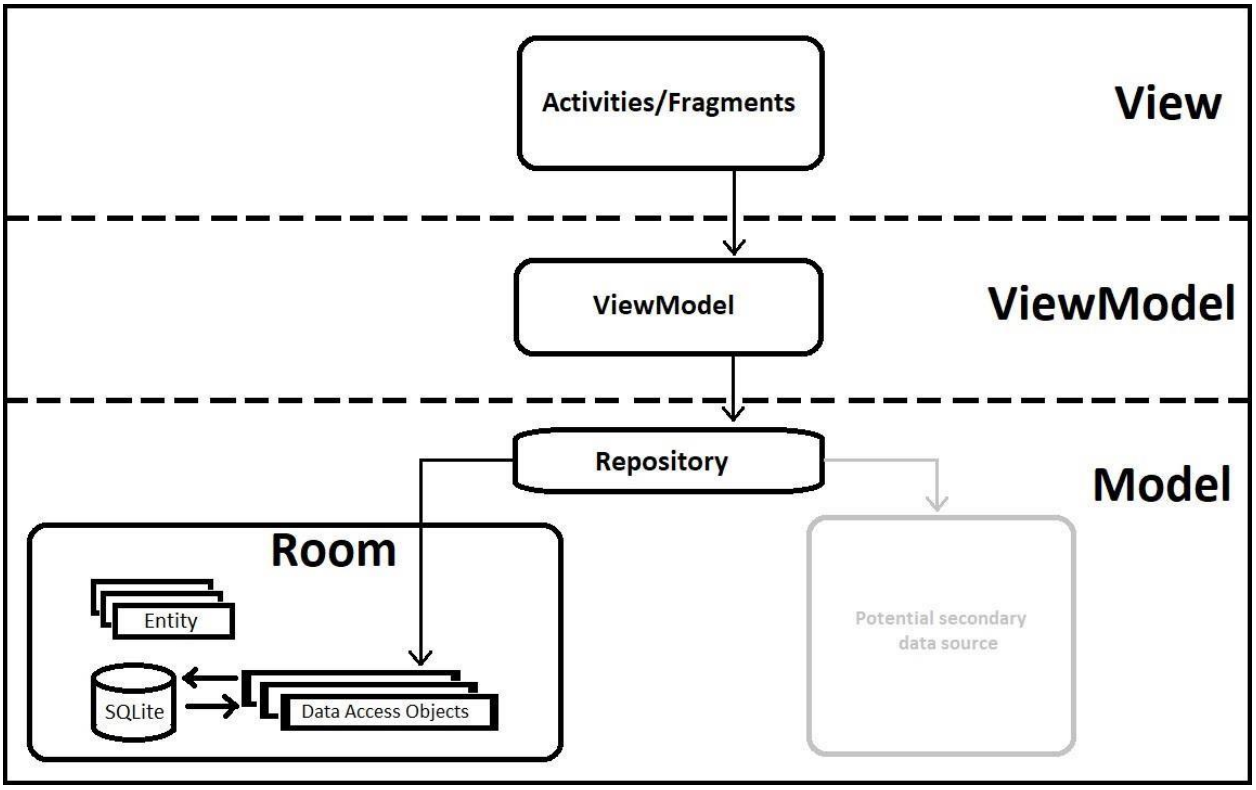
Although not necessary, it is best practice to include a repository class for handling access to external services, whether it be a SQlite database, the internet, or some other service. For Fieldnotes, only a SQLite database is accessed using the repository.



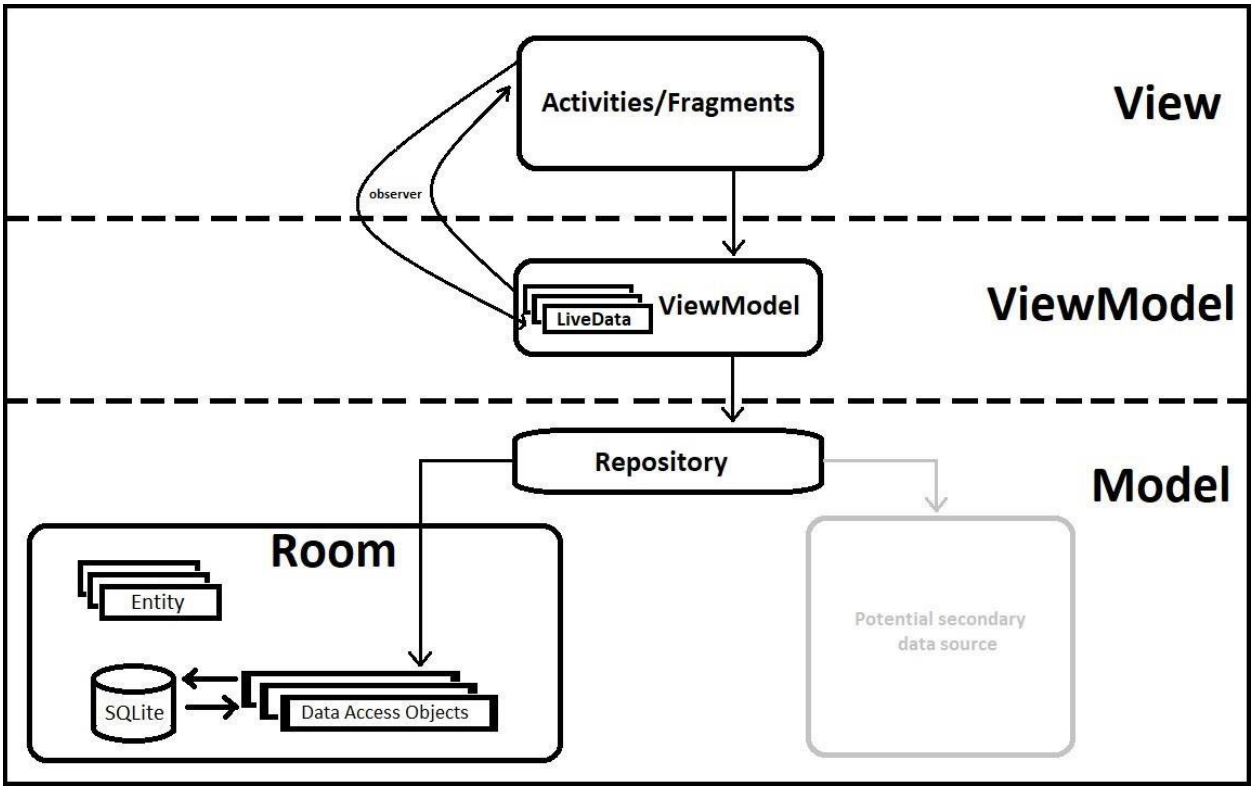
With the Model layer out of the way, next we'll talk about the ViewModel. The ViewModel is a class that is designed to hold and maintain UI-related data in a way that's lifecycle aware. This awareness will help data to persist through configuration changes, such as screen rotation, which would have otherwise required a complete rebuilding of the View data.



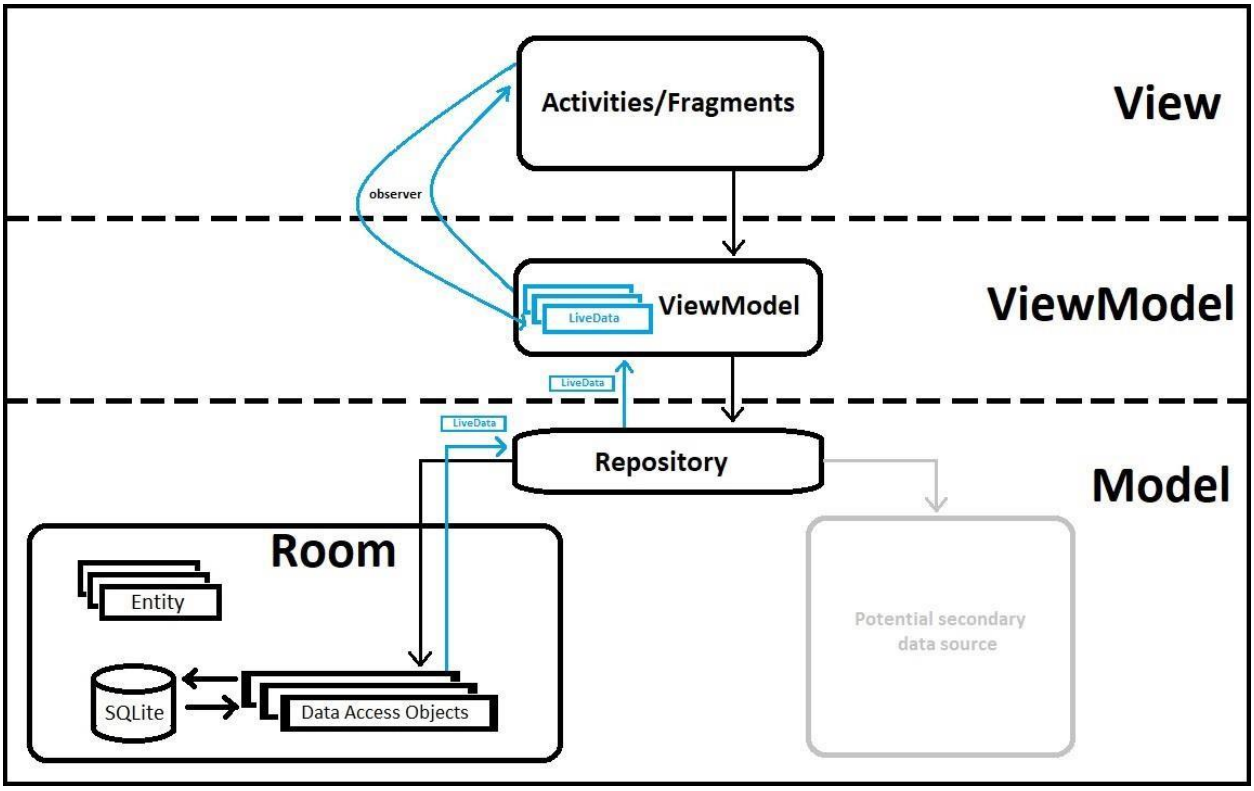
Lastly, and most familiarly to those who know Android programming, is the View layer of the app, which will house the Activities and Fragments of the project. However, the view of choice within the Activities for this app is the RecyclerView, which acts as a flexible, more efficient version of the common ListView. It is optimized to work with larger datasets, “recycling” data that gets scrolled off screen.



Now, the next of these architecture components that will be discussed is the LiveData class. This is a data wrapper class which allows data to be observable, and with that, updatable in real time. The LiveData itself is held within the ViewModel of the project, but it's the UI components which place what is called an observer on the data, since that is where data is displayed, and where changes need to be seen. Note that this LiveData is also lifecycle aware, so in the event that the containing Activity is stopped or paused, the LiveData observation will stop until the Activity/fragment is in the foreground again. Data will also persist through configuration changes, such as screen rotation.



Because LiveData is part of the same set of Architecture components that Room is, it already knows how to return observable LiveData.



The last architecture component to be discussed is the RecyclerView, which will be explained in more detail later, as this was just an overview of the app as a whole.

Room Database Library

In this section, we'll dive into the details of Room which weren't mentioned in the App Architecture section. To refresh, Room's three main components are the database class itself, the entity classes to be stored within the database, and the data access objects (or DAOs) which are used to perform the queries. Because of class reliance, we'll start with the entities.

For any class whose data you wish to store in a table, you will need to give the class an @Entity tag before its class declaration. For our example, we'll use the Stop class from the Fieldnotes app.

```
@Entity(tableName = "stops_table")
public class Stop {
```

As you can see, the Entity tag contains a parameter for the name of the table which the Stops will be stored in.

Next, you'll need to identify your entries in the database.

```
@PrimaryKey
@NonNull
@ColumnInfo(name = "stop_id")
private long unixTime;
```

Within your entity class, one of the class-level variables will be tagged with @PrimaryKey. If you should desire, the primary key does have a parameter option (autogenerate = true) for automatically incrementing primary key values as entities are added. However, for Fieldnotes, the Unix Epoch Time at which an entity was created was used for the primary key instead. In this case, the possibility of inserting an entity without a primary key could arise, so the variable also needs to be tagged with @NonNull. Any class variables with this tag, not just the primary key, must have a value, even if a filler, upon insertion. The @ColumnInfo tag is used to name the column of the table for the given value, denoted by the name parameter.

Since Stops are contained within Notebooks for this app, a foreign key is needed for when all the Stops of a particular Notebook need to be retrieved together.

```
@ColumnInfo(name = "stop_name")
@NonNull
private String stopName;
@ForeignKey(entity = Notebook.class,
        parentColumns = "notebook_id",
        childColumns = "parent_notebook_id",
        onDelete = CASCADE)
@ColumnInfo(name = "parent_notebook_id")
@NonNull
private long parentUnixTime;
```

As you can see, the @ForeignKey tag requires several parameters to function. In order they are: The class of the parent entity (in this case, Notebook.class), the name of the primary key column within the parent entity class, the name of the foreign key column within the child entity class (the same as given in the @ColumnInfo tag), and a parent deletion strategy. For the app, onDelete is set to CASCADE, which will remove child entries from the database, should they be found without a parent.

Now that we have our database tables, we'll need to perform queries, and the do that, we need Dao classes. While you can have just one Dao for accessing the entire database, best practice would have one Dao for each type of data. For Fieldnotes, this means one for Notebooks, one for Stops, and one for pictures.

```
@Dao
public interface StopDao {
```

As you can see, the Daos are interfaces which are tagged with @Dao. These interfaces will be implemented within the database upon its creation, as we'll see later. The Daos serve the purpose of mapping SQLite statements to Java methods.

```
@Insert()
void insert(Stop stop);
```

Above each method declaration, a tag for the corresponding type of statement @Insert, @Delete, or @Update will be given. More specific queries can be handled using the @Query tag.

```
@Query("DELETE FROM stops_table WHERE parent_notebook_id = :parent_unix_time")
void deleteAllStopsByNotebook(long parent_unix_time);

@Query("SELECT * FROM stops_table WHERE stop_id = :stop_unix_time")
Stop selectStop(long stop_unix_time);
```

Above are examples of specific deletion and retrieval queries from the database. Keep in mind that any Java parameters used within the query will have to be preceded with a colon.

Lastly, we'll take a look at the database class itself.

```
@Database(entities = {Stop.class, Notebook.class, Picture.class}, version = 3, exportSchema = false)
public abstract class FieldNotesRoomDatabase extends RoomDatabase {
```

With the @Database tag, you'll need to provide a list of all entities contained with the database and a version number, which will need to be incremented as changes are made to the database. The exportSchema parameter is set to false to suppress several warnings about unsafe version incrementation.

The only things that the database class will need are an instance of itself to make into a singleton, a Dao for each of its data types, and a getDatabase method.

```
static synchronized FieldNotesRoomDatabase getDatabase(final Context context) {

    if (INSTANCE == null) {
        //database gets created here if it doesn't exist
        INSTANCE = Room.databaseBuilder(context.getApplicationContext(),
                FieldNotesRoomDatabase.class,  name: "field_notes_database")
                .fallbackToDestructiveMigration()
                //UNCOMMENT LINE BELOW AND RUN TO POPULATE APP WITH TEST DATA
                //.addCallback(sRoomDatabaseCallback)
                .allowMainThreadQueries()
                .build();
    }


    return INSTANCE;
}
```

With the context of the application, the database class, and a string name to give the database, it will create the instance of itself if it has not already been created. Before it is built, methods can be run to set the behavior of the database. fallbackToDestructiveMigration is a method that will return to a working version of the database if a safe migration cannot be made when updating the database version. For the sake of testing, the commented-out line is a callback to an in-class method which will delete all database contents and repopulate the tables with pre-determined data for consistent testing. Lastly, although it is not recommended as part of best-practice, allowMainThreadQueries will allow queries to be made on the UI thread of the app. By default, Room only allows queries to be made asynchronously to prevent the main thread from stalling when queries are made. Allowing main thread queries won't pose a problem as the app is, because it is only retrieving single Notebook and Stop objects on the main thread, which will not take long due to the small datasets contained in the database.

The Repository

Although not necessary by Google as a part of app structure, the repository class serves an important purpose, and that is to compile access methods, and to modularize the app. As mentioned in the section on architecture, Fieldnotes makes no use of external data sources other than its own Room database, but should that change in the future, the repository would be the point at which to put the methods for that access. It also handles all of the asynchronous methods for database.

The bulk of the repository is comprised of private classes for performing the asynchronous queries to the database, as well as the public methods for using them.

```
private static class deleteAsyncNotebookTask extends AsyncTask<Notebook, Void, Void> {

    private NotebookDao asyncNotebookDao;
    private StopDao asyncStopDao;
    private PictureDao asyncPicDao;
    deleteAsyncNotebookTask(NotebookDao nbDao, StopDao sDao, PictureDao pDao) {
        asyncNotebookDao = nbDao;
        asyncStopDao = sDao;
        asyncPicDao = pDao;
    }

    /**...*/
    @Override
    protected Void doInBackground(final Notebook... params) {

        long parent_unix_time = params[0].getUnixTime();
        List<Stop> stops = asyncStopDao.getStopsByNotebook(parent_unix_time);

        for (Stop s: stops) {
            new deleteAsyncStopTask(asyncStopDao, asyncPicDao).execute(s);
        }
        asyncStopDao.deleteAllStopsByNotebook(parent_unix_time);
        asyncNotebookDao.deleteNotebook(parent_unix_time);

        return null;
    }
}
```

AsyncTask classes are made for each kind of query that can performed for each type of object within the database. Within the object, relevant Daos, a constructor for initializing those Daos, and an overridden version of the doInBackground method where the queries will be performed. The above example is one of the more complex tasks needed for the app, as three levels of data (Notebook > Stops > Pictures) will need to be deleted when a Notebook is deleted. To do this, AsyncTasks are created to delete each Stop within the Notebook, which will, in turn, delete the pictures of each of those Stops, before finally deleting the Notebook itself.

The View Model

In the past, Android apps would need to handle configuration changes by use of the proper Lifecycle methods, such as onCreate(), onDestroy(), etc. However, as part of the Android architecture components, the ViewModel is designed to store interface-related data in a lifecycleconscious way that will withstand configuration changes, such as screen rotation. This is achieved through the use of the LiveData class. Because we made use of a repository for handling database operations, much of the ViewModel class for Fieldnotes is just accessor and modifier methods.

One distinction which must be made is that the LiveData wrapper must only be used for information to be used on the app's UI. As such, objects like Notebooks and Stops cannot be "unwrapped" and used for update and delete queries. Attempting to retrieve a Notebook object using the getValue() method of a LiveData<Notebook> will only result in a null object. Thus, any object retrieved for purpose of creating a new Activity must be done through the included non-live methods.

Since ViewModels are created through use of a ViewModelProvider, one can't simply add more constructors to their ViewModels, so in the case of a NotebookActivity, where we need to know the id of the Notebook to display the contents of, we make use of ViewModelProviders to create this custom ViewModel for us.

```
viewModel = ViewModelProviders.of( activity: this, new FieldNotesViewModelFactory(
        this.getApplication(), unixTime)).get(FieldNotesViewModel.class);
```

As you can see in the call to the of method, along with the current Activity, we need a ViewModelFactory object to contain our parameters.

```
public class FieldNotesViewModelFactory implements ViewModelProvider.Factory {

    private Application mApplication;
    private long unixTime;

    public FieldNotesViewModelFactory(Application application, long param) {...}

    @Override
    public <T extends ViewModel> T create(Class<T> modelClass) {
        return (T) new FieldNotesViewModel(mApplication, unixTime);
    }
}
```

Within the ViewModelFactory, the only things that are required are the fields to be used as parameters, an initializing constructor, and an overridden version of the create method to return the ViewModel.

The RecyclerView

In keeping with the idea of Google standards for Android development, and the use of the architecture components, the Main and Notebook Activities of the Fieldnotes app make use of the RecyclerView for listing their data. RecyclerView is a more flexible, robust version of the ListView, but has several components working together to display the apps data. The main data container is the RecyclerView object itself, which populates with data provided by either a common ViewManager, like GridLayoutManager or LinearLayoutManager, or by a layout manager you implement yourself.

An item to be listed within the view is defined with a view holder, by extending RecyclerView.ViewHolder. Since the Main Activity of the app will be listing Notebooks, and the Notebook Activity lists Stops within a given Notebook, a ViewHolder is needed for each kind of visually represented item. What makes the RecyclerView more refined than the ListView is that it will only create as many ViewHolders as are needed to fill the display, plus only a couple that will be off-screen until the display is scrolled. This allows only visible or near-visible items to be kept in memory, which becomes valuable as data sets grow large.

These ViewHolder objects are then managed by an adapter, which is created by extending ViewModel.ViewAdapter. The adapter is responsible for the optimization mentioned above, whereby only a necessary number of ViewHolders are created and bound to the view itself. Because of the simplicity of the data presented in each view holder, the holder classes are implemented within the NotebookViewAdapter and StopViewAdapter classes.

Non-Standard Features

Using the Camera

Sending an intent to the default camera app on a phone and receiving a picture back is a tricky process in Android. This guide is meant to show the breakdown on just how to do that.

These are the steps we need in order to call the Camera Activity and return a picture:

- Add permission to android manifest ○ <uses-feature android:name="android.hardware.camera" /> ○ <uses-feature android:name="android.hardware.camera.autofocus" />
- Create a helper method that will call the Camera Activity in the appropriate class. ○ private void openCameraActivity()
- In the helper method, create a StrictMode.VmPolicy builder and set the vmPolicy to itself. This may seem arbitrary, but it is needed for Android to know we won't do anything dangerous to the user's files. We assure you **this is absolutely necessary and without it the Camera Activity will not work correctly.** ○ StrictMode.VmPolicy.Builder builder = new StrictMode.VmPolicy.Builder(); ○ StrictMode.setVmPolicy(builder.build());
- Next, you create a File (java.io) object with the name of picture. Adding the directory to the picture file is optional. Then use Uri.fromFile() to convert file to URI. Then add URI as an extra to an intent. The intent needs MediaStore.ACTION_IMAGE_CAPTURE, which uses the default camera app. ○ File img = new File(imageRoot, pictureName); ○ Uri outputFileUri = Uri.fromFile(img); ○ Intent intent = new Intent(MediaStore.ACTION_IMAGE_CAPTURE); ○ intent.putExtra(MediaStore.EXTRA_OUTPUT, outputFileUri);
- Then you call startActivityForResult() with an integer request code that you choose. Make it a global variable so you can access it later.
    - startActivityForResult(intent, TAKE_PICTURE_REQUEST_CODE);
- After calling startActivityForResult, you need to provide additional information to the camera, called ContentValues.
    - ContentValues values = new ContentValues(); ○ values.put(MediaStore.Images.Media.TITLE, pictureName); ○ values.put(MediaStore.Images.Media.MIME_TYPE, "image/jpeg"); ○ values.put(MediaStore.Images.ImageColumns.BUCKET_ID, img.toString().toLowerCase(Locale.US).hashCode());
    - values.put(MediaStore.Images.ImageColumns.BUCKET_DISPLAY_NAME, img.getName().toLowerCase(Locale.US).hashCode());
    - values.put("_data", img.getAbsolutePath());

-
  We then need to resolve this content, which is like loading the values we added, and pass them to the MediaStore URI. ○ ContentResolver contentResolver = getContentResolver();
    - contentResolver.insert(MediaStore.Images.Media.EXTERNAL_CONTENT_URI
      , values);
- If the picture was successfully created we will find the picture at the filepath we gave the intent.
    - String filePath = img.getAbsolutePath(); ○ File createdPicture = new File(filePath);
- In the class that will call the Camera Activity, we still need to override the onActivityResult method. This method is called when the startActivityForResult method returns. The requestCode will match the value you passed into startActivityForResult.
  Use that to test if the camera has returned with any errors. ○ protected void onActivityResult(int requestCode, int resultCode, Intent intent) { ○ switch (resultCode) { ○ case RESULT_CANCELED:
    - finish(); ○ …

Finding Geolocation

Finding the location of an android device is fairly straightforward, there is just a lot of setup, but all it easy to follow.

- First the class must implement LocationListener, ConnectionCallbacks, OnConnectionFailedListener to override methods
- We also need to instantialize Google's API at the end of the onCreate method.
    - GoogleApiClient googleApiClient = new GoogleApiClient.Builder(this).addApi(LocationServices.API).addConnectionCallbacks(this).addOnConnectionFailedListener(this).build();
- A helper method that sets up the timing of the location services is also needed.
    - locationRequest = new LocationRequest();
    - locationRequest.setPriority(LocationRequest.PRIORITY_HIGH_ACCURACY); ○ locationRequest.setInterval(100); //checks for changes in location every 0.1 seconds
    - locationRequest.setFastestInterval(100);
    - LocationServices.FusedLocationApi.requestLocationUpdates(googleApiClient, locationRequest, this);

-

  After this, we override the onStart method to immediately start the connection process.
  - ○ @Override
  - ○ protected void onStart() { ○ super.onStart();
  - ○
  - ○ if (googleApiClient != null) { ○ googleApiClient.connect(); ○ } else {
  - ○ exitActivity(API_ERROR); ○ }
  - ○ }
- Now all that is left is to receive the data of the location, which we do through another overridden method onLocationChanged(Location location) ○ @Override
  - ○ public void onLocationChanged(Location location) { ○ if (location != null) { ○ double longitude = location.getLongitude(); ○ double latitude = location.getLatitude(); ○ }
  - ○ }

RTF File Generation

In order to understand RTF files, please read https://www.oreilly.com/library/view/rtf-pocketguide/9781449302047/ch01.html

This class only has one public method: create RTF. This method will create an RTF Document with the Notebook's name passed to it as the title, so **NotebookName.rtf**. This file is saved in the default Downloads directory.

This method must be called in its own dedicated thread. If you attempt to call it on the main UI thread, **the app WILL crash**.

- The first step for RTF File Generation is to determine what settings you want the document to have. Font, styling, etc. and putting that in a global variable in the creation class.
  - ○ private static final String RTF_HEADER = "{\\rtf1\\ansi\\deff0 {\\fonttbl {\\f0 Times New Roman;}}";
- It is also important to get the systems newline character ○ private static final String RTF_NEWLINE = System.getProperty("line.separator");

  Next you must create a file and get an I/O stream to write to it. The easiest way is with a BufferedWriter.
  - ○ File file = new File(“document.rtf”);

-
    - BufferedWriter writer = new BufferedWriter(new FileWriter(file));
- If you want text at 12pt font, use the writer to write the rtf paragraph format ○ writer.append("{\\pard\\qc\\f0\\fs24 " + string_of_text + "\\par}");
- If you need a page break, use ○ writer.append("{\\page}");
- If you want pictures, they are slightly more complex. First we need to convert the picture into a byte stream by using the compress method in the bitmap class. Bitmap is the standard way to store pictures in android.
    - ByteArrayOutputStream stream = new ByteArrayOutputStream(); ○ bitmap.compress(Bitmap.CompressFormat.JPEG, 100, stream);
- Then we have to convert that back into a bytestream to output it into the rtf file.
    - byte[] imageInBytes = stream.toByteArray();
    - ByteArrayInputStream bis = new ByteArrayInputStream(imageInBytes);
- We still cannot add these bytes to our rtf file, as they are not formatted correctly. The hexadecimal bytes that are returned do not have padding, so we need to add padding with zeros.
    - int temp;
    - while ((temp = bis.read()) != -1) { //gives -1 when stream reaches EOF ○ String hexString = Integer.toHexString(temp); ○ if (hexString.length() == 1) ○ hexString = "0" + hexString; ○ writer.append(hexString);
    - }
- And once you are done, finally write an ending brace and close the writer to have your file.
    - writer.append("}"); ○ writer.close();