WikiWalks – Programmer Documentation

# Introduction

This document will serve to guide future developers wanting to take over, fork, or just contribute to the WikiWalks project. It will cover all components of the software and attempt to make understanding the codebase as quick and easy as possible. The project is made up of two parts – the Android app and the server. These will be covered in separate parts as they are not linked by code, just made to work with each other when deployed.

# Glossary

All words and phrases mentioned in this glossary can be referred to in either lowercase or.

Path – the parent object of everything else. The base of the app. Can contain a number of routes, points of interest, reviews, and photos. Used to refer to instances of the class in Path.java in the app, and the Path table and schema in the server.

Route – a specific walk within a path. As paths often branch out in multiple ways, there can be multiple routes on a path. Used to refer to instances of the class in Route.java in the app, and the Route table and schema in the server.

Point of interest / POI – a point within a path seen as notable. Can contain reviews and photos. Used to refer to instances of the class in PointOfInterest.java in the app, and the PointOfInterest table and schema in the server.

Photo – a photo of a path or point of interest. Used to refer to instances of the class in Photo.java, and the PathPicture and PointOfInterestPicture tables and schemas in the server.

Review – a review of a path or point of interest. Used to refer to instances of the class in Review.java, and the PathReview and PointOfInterestReview tables and schemas in the server.

Fragment – a native Android class for a portion of UI’s logic, different fragments are used to for different screens.

Activity – an Android application component that can be used for different functions. Only one activity as used, as most functionality that multiple activities used to provide can now be done with better performance by using fragments.

RecyclerView – an Android class that will use a template to create a list of UI elements using an iterable data set passed to it

Device ID – a unique ID generated by the app that effectively serves as a password, allowing a user to edit their contributions.

# Android App

## Basic Information

The app is coded entirely in Java, and was built using the Android Studio IDE. Although not required, Android Studio is recommended to continue development as it is the official software, completely free, and is integrated with a number of Android-specific features and can access Android-specific methods natively.

The app contains a number of external dependencies. These, along with versioning settings, can be found in the build.gradle file in the src folder. The main dependencies that need to be known about are:

* Play Services location and maps libraries – self-explanatory
* Gson, a JSON library used for creating JSON requests and parsing JSON responses
* Picasso, a library for easily loading images into ImageViews
* Retrofit, a library for easily making requests to the server.

There are also a number of AndroidX libraries, which extend native Android functionality, and some testing libraries.

Note: all global variables should be assumed to be private and all methods should be assumed to be public and not static unless otherwise noted.

## Non-UI Files

Within the src/main/java/com.wikiwalks.wikiwalks directory, there are a number of .java files. With the exception of those in the UI folder, which will be covered later, these are classes not UI-related. These will be covered in alphabetical order.

### CustomActivityResultContracts

This is a basic abstract class that contains two static custom ActivityResultContracts. ActivityResultContracts are a new feature from the AndroidX dependencies that make system functionality easier by handling result codes and callbacks easily. For most common functions there are already inbuilt ActivityResultContracts, however these were not sufficient for the app so custom ones had to be made.

The first is ExportSettings. This extends the built-in CreateDocument contract that was made for a creating a file, but narrows it down to creating specifically a JSON file, as that is the format settings are exported in.

The second is SelectPhoto. This extends the built-in GetContent contract that was made for selecting a file, but instead of opening a file picker, it opens the gallery so only photos can be selected, and in a way that makes it easier than navigating manually for photo files. This is used for selecting photos to upload.

### DataMap

The DataMap class is a singleton object for storing cached paths and points of interest. It is created with the getInstance method that will instantiate the class as a private static object if it has not already, and will return it. It cannot be created any other way to ensure that there is only one instance of the object at any time. The object contains 3 global variables – a LinkedHashMap for storing Paths, with each Path’s ID as their individual keys, a similar LinkedHashMap but for points of interest, and an ArrayList for storing DataMapListeners. DataMapListener is a public interface available in the class and will call either a success or a failure method when paths are updated. These are implemented in fragments, and as fragments are regularly created and destroyed, the ArrayList is needed to keep track of the active ones.

There are a number of methods in this class, most of which are simple. Methods for adding and removing paths and listeners, getters for the HashMaps, and methods to iterate through all listeners, calling either the success or failure method.

The primary method in this class is updatePaths(), which requires a LatLngBounds, which is essentially just a set of coordinate boundaries, and a context. The method creates a new JSON containing the device ID, then gets a Retrofit instance and creates a new getPaths request with the boundaries and JSON as arguments. It then runs the request. If the request succeeds, the server will return a JsonArray of Paths within the set boundaries, which will then be iterated through to create path objects and put them in the HashMap. The method to trigger all listeners’ success methods is then called. Otherwise, if the request failed and was not because the boundaries were too large (a maximum of 3 degrees worth can be requested at once), the listeners will have their failure method triggered.

### GroupWalk

Group walks are a function of the app that allow users to schedule walks that other users can attend by marking attendance and are accessed from the path screen. This class contains global variables for all of the information about group walks, which are the ID, the title, Unix time, a list of attendee names, the host name, whether the user is attending, whether the user created the group walk (named editable), and the path it’s associated with. The class is constructed with a JSON object containing its attributes, and the parent path. The JSON is parsed in a separate method named setAttributes() as the code is reused for updating the class.

The class has 3 interfaces for callbacks, GetGroupWalksCallback, which is used when requesting a path’s group walks, EditGroupWalkCallback, which is used when editing or deleting a group walk, and AttendGroupWalkCallback, which is used for updating the user’s attendance status. These each contain success and failure methods, with EditGroupWalkCallback containing separate success and failure methods for editing and deleting.

The important methods in this class are the methods used for constructing and sending JSON requests related to group walks. Each of these require a Context be passed in order to retrieve the device ID and get the Retrofit instance. First there is the static method submit(), which will take a path, Unix timestamp, title, and EditGroupWalkCallback along with the context. It constructs the path’s ID, time, and title, along with the device ID, into a request that is sent to the server. If this request succeeds, it will use the returned JsonObject to construct a new GroupWalk and add it to the path, then calling the callback’s success function. If it fails, the callback’s failure function will be called. The other methods are all relatively similar. There is edit(), which takes the new time and title and will send a request with a them, updating them locally upon success, delete(), which will send a request to delete the walk and will remove it from the path’s list upon success, and toggleAttendance, which sends a request to toggle whether the user is attending, and will update the walk details on success. These each also require and call their respective callbacks on success or failure. All other methods in the class are simple getters.

### LocationService

The LocationService class is an extension of Android’s Service class. Its purpose is to collect location information in the background while a path is being recorded, as a Service is needed to do this. Upon the app entering the background, it will show a notification and continue recording location data, passing it back and ending the service upon the app being brought back into the foreground. The class contains a number of public static final Strings, used as constants in the class. It also contains non-static variables for the last location, ArrayLists for coordinates, the distance walked since the service started, a location provider, and an instance of the native LocationCallback class.

The methods in this are pretty simple. onStartCommand() is what is run when an Intent is passed to either start or stop the service. It simply initialises the location provider, then checks if it is supposed to start or stop, and runs the respective commands for whichever it is. It returns START\_STICKY, which is an int and simply indicates the type of Service.

To start the service, createNotification() and then startLocationUpdates() will be run. createNotification() will check the Android version it is running on and build and display a notification accordingly, which is required for foreground services. startLocationUpdates() will build the location request and start a loop. Each time the location is retrieved, the callback will call addLocation() where the location’s coordinates will be added to the lists and the distance walked will be updated. To end the service, stopLocationUpdates() will be run, which will bundle all of the collected data, return it to the recording fragment, and stop the service.

### MainActivity

This class extends AppCompatActivity, and is the basis for the rest of the app. Android activities are basically application components. Only this one is needed for the app, as most different functionality can be put into fragments, which will be discussed in the UI section.

The main methods in this class are the static methods checkPermission(), which will check/request a permission passed to it, and getRetrofitRequests(), which will create a Retrofit instance and store it in a static variable. These really could be anywhere but were just put in MainActivity for convenience. Other methods simply initiate the first fragment and handle going back in a few basic lines of code.

### Path

This is the most important object class in the app. It contains a number of variables. First, the path’s direct attributes – its ID, name, walk count, and average rating. Then it contains a number of ArrayLists for objects it is the parent of – routes, points of interest, photos, and reviews, as well as another Review object for the user’s own review. It also contains internal variables for loading reviews and photos, as these are loaded in pages, it has variables for the next page to load and its current loading status. Finally, it has variables related to the map – a list of markers, its starting position, and its boundaries. Its constructor takes a JsonObject, and parses through it, assigning the variables, creating the route, point of interest, and group walk objects returned in the request, adding them to their respective lists.

The class contains 2 interfaces – GetPathCallback, and PathChangeCallback. The first is obviously for getting a path, and the latter is for editing it. They each contain success and failure methods.

Apart from getters and setters, the majority of methods in this class are request methods. All of these require contexts and most require callbacks as arguments, triggering their success methods on success and their failure methods on failure. The first is getPath(), which is a static method that takes a path ID and will request an individual path from the server. This is used when getting bookmarked paths that weren’t loaded by the map. Next is edit(), which will take a new title and send a request to the server to update it, updating it locally upon success. Then there is walk(), which will increase the walk count by one. Then there are getReviews() and getPhotos(). These are very similar methods, requesting a page of the path’s reviews and photos respectively. They will increase the local next page variable each time, and will stop calling if the last page has been reached. They may also be reset to the first page by passing true as for the refresh variable. Finally there is getGroupWalks(), which will request all group walks but does not do so in pages.

Finally there are a couple of minor methods, addPointOfInterest() and removeRoute() which are self-explanatory, and getAllAltitudes(), getAllLatitudes(), and getAllLongitudes(), which will combine the altitudes, latitudes, and longitudes of every route in the path, and makeMarker(), which will generate a Google Maps marker, store it in the list, and return it.

### Photo

This class serves as the class for photos for both paths and points of interest. It uses an enum, PhotoType, to store which it is for. The only differences between the 2 types are the URLs it sends requests to, and the lists it adds/removes itself to/from. It contains all the attributes you’d expect from a photo class – the ID, the parent (path, point of interest) ID, the URL, the width and height, the description (caption), the submitter name, whether it’s editable (true if the user created it), and the type. Its constructor requires a JsonObject to parse, a parent ID, and a type.

The class contains 2 interfaces – GetPhotosCallback, which is used in the Path and PointOfInterest classes for getting photos, and EditPhotoCallback, which contains methods for editing and for deleting photos. The editing methods are also used for submitting. Both of these contain success and failure methods.

The class mostly contains request methods. submit() is a static method for submitting new photos, but is different than other requests as it cannot use a JSON request. Instead it makes a multipart form that includes the parent object’s ID, the photo file taken from a Uri, and the caption, sending it to the URL for its type. On successful response it will add itself to the parent object’s photo list, calling the success callback, or the failure callback if the request failed. The edit() method will send a request to update the photo with a new caption, and the delete() method will send a request to delete it. These methods are only available for the original submitter.

### PointOfInterest

In many ways this class is similar to Path. It contains a number of the same attributes – an ID, name, average rating, ArrayLists for markers, photos and reviews, the user’s own review, as well as review and photo page variables and loading booleans. Additionally, it contains the point’s coordinates, the parent path, and a boolean named editable, indicating if the user created and can therefore delete it. Its constructor takes a JsonObject, which it parses, and a Path to set as the parent path.

The class contains 2 interfaces – PointOfInterestSubmitCallback, which is self-explanatory, and PointOfInterestEditCallback, which contains methods for editing and for deleting points. Both of these contain success and failure methods.

The methods in this class are mostly requests, with submit(), edit(), getPhotos(), and getReviews() being effectively identical to the ones described in the Path.java section, but with a few changed attributes and obviously pointing to the point of interest URLs on the server instead of the path ones. One extra request method not found in Path, however, is delete(), which will send a request to the server to delete the point if it was created by the person trying to delete it, removing all of the markers from the list and removing itself from the path upon success.

### PreferencesManager

This class is a singleton object for managing SharedPreferences, which is the way that Android handles saving settings. It has a private constructor and can only be retrieved by using getInstance(), which requires a context as the context is needed to initialse the SharedPrefs objects. There are 2 of these – preferences and statistics, that are initialised upon creation.

The class contains a number of methods for changing different preferences and statistics. Most of these are very basic and easy to understand – getting the current saved value, updating it, then saving it, or just simply returning a saved value. There are a few more complex methods though. First is toggleBookmark(), which takes a path ID and will toggle whether it’s saved as a bookmark. This is done by making the bookmarks into an ArrayList of strings, checking if the current ID is in it, then adding it if so or removing it if false, then turning the ArrayList back into a single string with comma separated values and saving it. It will return a boolean of whether or not it is now in the list. Secondly is getStatistics(). This method looks complicated but really is just getting all saved statistics, putting them into a string with their label, and returning them all as an array. There are slight variations for values depending on if the user is in a language that uses imperial measurements or one that uses metric. There are also the goal methods, which essentially take a start time, end time, and distance, and serialise them into a JsonObject, then saving the JSON as a string. Finally there are importPreferences() and exportPreferences(). These need to be passed a Uri from elsewhere in the app, but will export and import all preference and statistic data to and from .json files in the user’s location of choice. These look more complicated than they are, but effectively just run loops on all saved preference and statistics data to export it, nesting the statistics in their own section, and then loop once again to import it, making sure everything is imported as the correct type.

### RetrofitRequests

This is a simple interface that contains possible requests that can be made with Retrofit. These contain the method, the URL endpoint, and list the arguments required to send the request.

### Review

The review class is in many ways quite similar to the Photo class. It has an enum for whether it is a Path or PointOfInterestReview, has basically identical interfaces, and very similar methods. It contains variables for its ID, the parent object’s ID, the reviewer’s name, the review’s rating and message, and the review type. It’s constructor is also very similar to Photo’s, requiring a JsonObject, parent ID, and type.

As mentioned, it has interfaces basically identical to Photo’s – GetReviewsCallback and EditReviewCallback. These serve the same purpose as those found in the Photo class and have the same methods.

When it comes to methods, once again the majority are either getters/setters and request methods. submit() is a static method that takes a type to determine the URL, a parent ID, message, and rating to put into a request, and an EditReviewCallback to call on success or failure. Upon success, it will create a new Review and set it to the Path or PointOfInterest’s ownReview variable. edit() allows the submitting user to update the rating and message, and delete() will obviously delete the review.

### Route

This class is pretty simple. It contains variables for the route ID, the parent path, whether it’s editable by the user, ArrayLists for the 3 coordinate types and Google Maps Polylines, and the distance of the route. Its constructor requires a JsonObject, which it will parse into the variables and calculate the distance with, and the parent path.

It contains 1 interface – RouteModifyCallback, which has success and failure methods and is used for both submitting and deleting routes. The success method requires a Path argument.

For methods, the majority are getters. There are also 2 request methods. The first is submit(), which is a static method that takes a parent path for if it is a new route on an existing path, and a path title for if it is a new path, ArrayLists for latitudes, longitudes, and altitudes, and a callback. It builds a JSON request by converting the ArrayLists to JsonArrays and by checking whether the provided path is null, adding the title if it is or the path ID if it isn’t. Upon sending the request it will create a new Path as the result returns a whole path rather than just the route, adding it to the DataMap, which will overwrite the old path if it exists already. The other request method is delete(), which is self-explanatory and can only be used by the user who created the route. The last notable method is makePolyline(), which will generate a Google Maps Polyline, colour coded based on how much it’s been walked, add it to the ArrayList for polylines and then return it. The ArrayList is stored in the route for easily removing all of them if the route is deleted.

## UI Classes

There are 3 types of UI class – fragments, dialogs, and RecyclerView adapters. There are a lot of these, mostly with overlapping functionality and functions by nature of what they’re supposed to be, which is extensions of inbuilt classes. This section will primarily discuss highlights of each class as a lot of it is repetitive and easy to understand from just looking at it.

### Dialogs

#### GoalDialog

The GoalDialog exists to create and edit goals. The newInstance() method to create it requires a saved goal’s position in the goals array, or -1 if a new goal is being created. Although the looks a little complicated, it is mostly simple, just long. In the onCreateDialog() method, the parent GoalsFragment will be set as a listener. The unit system to use is checked by checking the system’s region and it displays either km or mi next to a distance input box. The select end date button is then set to open up a calendar, where the user will pick when the goal should end. The rest of the calendar’s values are then filled in to be at the last millisecond of the goal’s end date, and a preview of the date is displayed. The submit button then has its on click listener set to calculate the goal distance in metres, and depending on whether the goal’s was new, either run the PreferencesManager’s editGoal() or addGoal() with the information. The parent fragment’s goal RecyclerView is updated upon success, and the dialog dismisses itself. The delete button is then set up to show a yes or no prompt to confirm deletion, running PreferenceManager’s removeGoal() on confirmation.

Next, if it is a goal being edited, the goal’s data will be prefilled into the fields and the delete button will be made visible.

If the GoalDialog is destroyed, the current date and distance values will be saved in onSaveInstanceState(), and restored if savedInstanceState is not null in onCreateDialog().

#### GroupWalkDialog

This dialog is for creating or editing group walks. It requires a path ID and the target group walk’s position in the path’s GroupWalk array to be passed to newInstance() to create it. -1 can be passed if it is a new group walk.

In onCreateDialog(), the path and group walk will first be obtained through the DataMap. The select time button will have its click listener set to open calendar and clock dialogs, with the minimum date being the current date, and once these dialogs have had a date and time selected, this will be displayed and the submit button will be enabled.

The submit button will be set up to check if the walk title field is empty, and will auto-generate one if so, then checking whether the GroupWalk already exists, running the edit() method on it if so or the GroupWalk.submit() method if not, with the dialog’s fields being used to obtain information for both. The delete button then has its listener set to create a confirmation prompt, and run the delete() method on the group walk if confirmed. Both of these use the dialog itself as the EditGroupWalkCallback, as it implements its functions, updating the parent fragment’s RecyclerView on success, and showing a toast on failure.

Finally, if it is an existing group walk being edited, its values will be prefilled, and if the dialog has been destroyed and recreated, saved values from saveInstanceState() will be restored.

#### NameDialog

This class is a dialog with a single text box and a save and cancel button. It is reused across a number of different fragments, and uses an enum, NameDialogType, to determine whether it is being used to edit a path name, a point of interest name, or the user’s nickname. It also contains an interface, NameDialogListener, which has 2 methods – setNameDialog(), which takes a NameDialog instance (itself) as the argument for setting the parent fragment as the listener in cases where the dialog is reloaded (such as on screen rotation), and onEditName(), which takes a NameDialogType and the new name as arguments.

It is created through newInstance() which requires one of these types, as well as an object parent ID (-1 if for a new path or username). In the creation method, onCreateDialog(), the parent fragment is set as the NameDialogListener, and the setNameDialog() method is called in it. Then, the parent object’s current name or the user’s current nickname is obtained if it exists and prefilled into the text box.

If the dialog is destroyed and reloaded, the current name will be saved in onSaveInstanceState() and will be restored in onCreateDialog() if the saved state exists.

#### PhotoDialog

This is by far the most complicated of the dialogs. Its newInstance() method requires 4 arguments – a photo type, parent ID, position in the parent’s photo array, and a bitmap to preview the image. The position can be -1 and the bitmap can be null if it is a new photo, rather than one being edited. The class contains an interface – EditPhotoDialogListener, which contains 2 methods – onEditPhoto(), and onDeletePhoto() which takes a position.

In onCreateDialog(), the parent fragment is first set as the listener, and if it’s a photo being edited, the photo is retrieved from a path or POI from the DataMap. The buttons then have their click listeners set. The from camera button is set to first request external storage permissions, and if granted create a file in the phone’s Photos\WikiWalks folder using the correct method for the version, with MediaStore being used on Android Q and a simple file creation being used on older versions, then passing the file’s Uri onto takePhoto, an implementation of the native TakePicture() ActivityResultContract. This will launch the device’s native camera app and allow them to take a photo, returning said photo’s Uri and sending it to loadIntoImageView(), which will rotate the photo based on its EXIF tags and display it above the photo selector buttons. The from gallery button is set to also check external storage permissions, launching selectPhoto – an implementation of the CustomActivityResultContract SelectPhoto(), which open’s the phone’s gallery, allowing the user to select a photo from it. On the photo being selected, it will also be passed into loadIntoImageView(). Upon either of these finishing, the submit button will then be enabled.

The submit button will check if the photo exists and is being edited, or if it’s new. If it’s being edited it will take the text from the caption text box and send it to the photo’s edit() method, or if it’s new it will send the photo type, parent ID, the Uri of the selected photo, and the caption to Photo.submit(). If the photo is being edited, the delete button will be visible and will display a confirmation dialog to run the photo’s delete() method on click. These methods will all also pass the dialog itself as an EditPhotoCallback implementation, as it contains all required methods. On success of the photo being submitted, edited or deleted, it will call the parent fragment’s implementation of EditPhotoDialogListener. On failure, it will show an error toast.

Finally, once the on click listeners are set, it will prefill all fields if the photo is being edited or if a savedInstanceState was passed from onSaveInstanceState().

#### ReviewDialog

This dialog is used for creating and editing a review for a path or point of interest. It requires a ReviewType and a parent ID to be passed to newInstance() to create. It contains an interface – EditReviewDialogListener – which simply contains the single method onEditReview().

In onCreateDialog(), it will first set the parent fragment as a listener, then get the parent object and run getOwnReview() on to get the user’s own review. A rating bar then has its change listener set to enable the submit button if a value other than 0 is set. The button click listeners are then set, with the submit button checking if the own review is null, running Review.submit() with the type, parent ID, input box’s text, and rating, or running edit() on the review with the new message and rating if they’ve changed. The delete button, which will only show if it is an existing review being edited, is then set to show a confirmation dialog on click, with delete() being run on the review if confirmed. These all also take the dialog itself as an EditReviewCallback, as it implements the class. On success of these methods, the parent fragment’s implementation of EditReviewDialogListener’s success methods will be called, with a toast being shown on failure.

Finally, the values will be prefilled, first from the existing review if it is not null, then from a savedInstanceState if one was saved in onSaveInstanceState().

### RecyclerViewAdapters

#### BookmarkedListRecyclerViewAdapter

This class requires a parent fragment and an array of paths to create. For each entry, it will display the path’s title in a button, setting the button to open a PathFragment using the path on click. It will check if each path is null in case a bookmarked path is no longer available, and hide the entry if so.

#### GoalsListRecyclerViewAdapter

This class requires a parent GoalsFragment and an ArrayList of JsonObjects to create. For each entry, it will parse the start and end dates out, and display them formatted. It will then check the country, and convert the distance walked and distance goal to the user’s local units, displaying them in a fraction. It will then check if the distance walked is past the distance goal, and display it green if so. If the current date is past the end date and the distance target was not met, it will display it red. Finally, it will set a pencil icon next to each entry to open the goal in a GoalDialog.

#### GroupWalkListRecyclerViewAdapter

This class requires a parent GroupWalkListFragment and an ArrayList of GroupWalks to create. For each entry, it will first display the time and host, then check if there are any attendees and either display them or show “no attendees” if there aren’t any. It will then check if the GroupWalk is editable, and replace the attending checkbox with a pencil icon that will launch a GroupWalkDialog with the walk. It will then see if the user is attending the walk, set the attending checkbox to be filled in if so, and set the click listener of the attending to run toggleAttendance() in the parent fragment with the position of the GroupWalk in the ArrayList.

#### PhotoListRecyclerViewAdapter

This class requires a parent PhotoListFragment and an ArrayList of Photos to create. For each entry, it will run Picasso’s get() function to load the photo into an ImageView from a URL. It will then check if the photo is editable, italicise and set the name to “You”, and enable a pencil icon set to run the parentFragment’s launchEditDialog() method with the photo. Otherwise, it will set the name to the submitter’s name, then check if there is a caption and display it, or an italicised “no caption written” if there isn’t one.

#### PointOfInterestListRecyclerViewAdapter

This class requires a parent PointOfInterestListFragment and an ArrayList of PointOfInterests to create. For each entry, it will set a button’s label the point of interest’s name, set the click listener to open a PointOfInterestFragment with the point, and then display a different colour on the left of each button. This requires different methods for those on/above and under Android L, and is used for colour coding with markers in the PointOfInterestListFragment.

#### ReviewListRecyclerViewAdapter

This class requires a parent ReviewListFragment and an ArrayList of Reviews to create. For each entry, it will set the name and rating of the review. It will then check if the review has a message, displaying it if so, or displaying “no review written” in italics if not.

#### RouteListRecyclerViewAdapter

This class requires a parent RouteListFragment and an ArrayList of Routes to create. For each entry, it will first add itself to an ArrayList, buttons, then check the device’s country and show the route number followed by the path’s distance in localised units. It will then display a different colour next to each route label, and set a click listener to first remove any background highlights from other buttons, highlight its own background, then run the parent fragment’s selectRoute() method with the entry’s position. These require different methods for those on/above and under Android L, and the colour indicators are used for colour coding with polylines in the RouteListFragment.

#### StatisticsListRecyclerViewAdapter

This class requires a context and an array of strings to create. All of the processing for these strings is done earlier, so it simply displays each string.

### Fragments

#### BookmarksFragment

This fragment displays a list of bookmarked paths in a RecyclerView. It requires no arguments to create. Upon creation, it will request the list of bookmarks from PreferencesManager. If there are no bookmarks, it will show a message saying there aren’t any bookmarks. If there are, it will create an array of paths the size of the bookmarks list. It will check if each bookmarked path is already loaded and put them in the array in their position. For paths that aren’t loaded, it will call Path.getPath() with the ID of the path, adding results to the array, or null if the path is invalid or has been deleted. Once all of the array has been filled, a BookmarkedPathRecyclerViewAdapter will be shown with the array as the data.

#### GoalsFragment

This fragment displays a list of user goals, along with a button to create new ones. The fragment will request the list of goals from PreferencesManager, adding them to an ArrayList and passing that list to a GoalsListRecyclerViewAdapter.

The set goal button within the GoalsFragment will bring up a GoalDialog with -1 as the position to indicate a new goal.

#### GroupWalkListFragment

This fragment gets and displays a list of a path’s group walks. It requires a path’s ID be passed to the static method newInstance() to be created. Once created, it will use the DataMap to get the path with the passed ID. The fragment contains a SwipeRefreshLayout, which in turn contains a RecyclerView using the GroupWalkListRecyclerViewAdapter with the Path’s group walks ArrayList as the data set.

When the fragment is created, updateGroupWalksList() will be called, which will call the path’s getGroupWalks() method, using itself as the GetGroupWalksCallback required. Upon success, it will call updateRecyclerView() which will cause the RecyclerView to refresh to the latest version of the data set, or display a message saying there are no group walks if the data set is empty. The SwipeRefreshLayout is set to re-run this if swiped down on.

The fragment also has a button for scheduling group walks, which will bring up a GroupWalkDialog, using -1 as the position to indicate a new group walk.

#### MapsFragment

This is essentially the home screen of the app. It simply shows a map with markers for each path, a toolbar with buttons, and a button to create a new path. On creation in onCreateView(), it adds itself to the list of DataMapListeners in the DataMap (and removes itself upon destruction in onDestroy()) so that it will know to update or remove any paths that may be changed or removed. It then sets the toolbar buttons’ click listeners – making the gear icon open a new SettingsFragment, and making the bookmarks icon open a new BookmarksFragment. Next, it sets the create path button’s listener to open a new RecordingFragment. It then requests loading the map, continuing in the onMapReady() function overridden as an implementation of OnMapReadyCallback, where it sets the map type to hybrid (satellite and streets), calls setMapLocation(), which will check if the app has the location permission and centre the map on the user if it does, opening a new PermissionsFragment if it doesn’t. It then sets the marker click listener to itself, as it implements OnMarkerClickListener, setting the markers to open their respective PathFragment on click in onMarkerClick(). Finally, it sets the map’s OnCameraIdleListener, which is what will be called when the map stops moving, to call the DataMap’s updatePaths() method using the map’s current view boundaries if it encompasses less than 3 degrees in either coordinate. When the updatePaths() method finishes, it trigger’s the MapFragment’s implementation of DataMapListener, creating the lines and markers of the loaded paths if successful, and displaying a toast notification if failed.

#### PathFragment

This is the fragment that displays information on a path. It requires a path’s ID being passed to newInstance() to create. Upon creation, the DataMap will be searched for the path with the ID passed. In onCreateView(), the path will have its rating and name extracted and displayed. It will add itself as a DataMapListener so onDataMapUpdateSuccess() can redraw the route lines should one be deleted.

This fragment has a number of buttons – points of interest, group walks, reviews and photos in one menu, and select route, record route, and explore buttons at the bottom. All of these buttons have their click listeners set to open their respective fragments. It also has 2 buttons at the top – a bookmark button, which is set to send the path ID to the PreferenceManager’s toggleBookmark() method, and an edit button, which will bring up an EditNameDialog, passing through the type PATH, the path ID, and itself as the listener. Upon changing the name in the dialog, onEditName() will be triggered with the new name. If the new name is empty, a name will be autogenerated. It will then be sent to the path’s edit() function, using itself as the callback where onEditSuccess() will be called and update the toolbar title on success, and a onEditFailure() will be called and display an error on failure. Additionally, it sets itself up to listen for fragment results sent with the “update\_rating” tag, so that the rating bar can update when the user edits a review.

#### PermissionsFragment

This fragment displays when a permission required for functionality has been denied. It requires a string to be passed to newInstance() containing the Manifest.permission string, which is saved in the variable type. In onCreateView(), the type will be used to display the correct message depending on the permission, and will set the button to open the app’s settings, where the permissions menu can be found. If, upon exiting settings, the permission has been granted, the fragment will close itself.

#### PhotoListFragment

This fragment displays a list of photos and a button to submit a new one. It requires a PhotoType and a parent object ID to be created with newInstance(). In onCreateView(), the toolbar will have it’s name set to “Photos – [path name]”. It will then set up a PhotoListRecyclerViewAdapter in a RecyclerView. Depending on whether the device is portrait or landscape, it will set the RecyclerView to either one or two columns. It will then set the RecyclerView’s scroll listener to look call updatePhotosList() with the boolean false, indicating the next page rather than a full refresh, when the bottom is reached. The SwipeRefreshLayout containing the RecyclerView then has it’s refresh listener set to call updatePhotosList() with true, starting a full refresh. Next, the add photo button is configured to check external storage permissions. If the permission is granted, launchEditDialog(), which will open a new instance of PhotoDialog, will be called with the position -1 to indicate a new photo. If the permission isn’t granted a PermissionFragment will be created using the external storage permission string. Finally, if it is a new fragment (e.g. not destroyed and recreated from rotating), it will call updatePhotosList() with true.

updatePhotosList() will call the relevant object’s getPhotos() method, using the fragment as the callback as it implements GetPhotosCallback. It also implements EditPhotoDialogListener, so the PhotoDialog can call methods when needed. The success methods for these listeners will run updateRecyclerView(), which will simply refresh the content displayed. On failure to get photos, a toast will be shown.

#### PointOfInterestFragment

This fragment is very similar to PathFragment. It requires a point of interest’s ID to be passed to newInstance() to create. In onCreateView(), it will first check if the point is editable and enable the toolbar’s delete button if so. The toolbar then has its edit and delete button click listeners set.

The edit button will bring up a NameDialog with the type POINT\_OF\_INTEREST, and the point’s ID. The fragment implements NameDialogListener, and will have onEditName() called with the new name when the dialog’s submit button is pressed. It will then use itself as a callback to run edit() on the point of interest with the new name, as it also implements PointOfInterestEditCallback, and will trigger onEditPointOfInterestSuccess(), updating the toolbar on success, and trigger onEditPointOfInterestFailure(), showing a toast notification on failure.

The delete button will bring up a confirmation dialog with yes and no buttons. If yes is pressed, the point’s delete() function will be run, once again using itself as the PointOfInterestEditCallback. This time, the onDeletePointOfInterestSuccess() method is called on success, sending a fragment result to a fragment listening for a result with the key “update\_poi\_list”, then automatically closing the itself. On failure, it will call onDeletePointOfInterestFailure(), once again displaying an error message.

#### PointOfInterestListFragment

This fragment is created with a path ID being passed to newInstance(). In onCreateView(), it will get the list of points of interest from the path with the ID passed. It will initialise a RecyclerView with a PointOfInterestRecyclerViewAdapter, using the just retrieved list as the data set. If there are no points, it will display a message saying such. The fragment will then set itself up as the listener for the fragment result “update\_poi\_list”, running update() on a result being passed. update() will refresh the RecyclerView, remove any markers from the map, and recreate them. Next, in onStart(), the getMapAsync() will be called on the map. When the map has been retrieved, onMapReady() will be called with the new map. This method will set the map type to hybrid, draw polylines and markers for the path and points, then set the camera around the boundaries of the path.

#### RecordingFragment

This is by far the most complex of the fragments. It requires a path ID be passed to newInstance() to create. This can be -1 in the case of it being a new path. In onCreateView(), it first checks if it is an existing path or a new one, and gets the path if existing. It then checks if there is a savedInstanceState() and restores saved coordinates into their respective ArrayLists. The location provider is then initialised.

Next, the stop recording button has its click listener set. This makes the location provider stop getting updates, and sets the button text to “Resume Recording”. It then checks if the path is new. If it is, showSubmissionDialog() will be called with true, indicating a new path. If it isn’t, it will check if the recorded route is in range of the other routes in the path. If it is, it will call showSubmissionDialog() with false, indicating a new route on an existing path. If it isn’t in range, it will prompt the user if they want to create a new path instead. If they press yes, showSubmissionDialog() will be called with true. If they press no, nothing will happen and they are free to resume recording by pressing the button again, which will continue recording unless they are more than 25 metres away from the last point recorded. showSubmissionDialog() will bring up a confirmation dialog asking the user if they want to submit. If they press yes and it is a new path, a NameDialog will be created. onEditName() will receive the name entered in this dialog. This method will be used for both naming paths and points, so a check is done and if the type is PATH, it will call submitRoute() with the new name. If it is not a new path, submitRoute() will also be called without a new name. submitRoute() requires it be passed a RouteModifyCallback for testing purposes, so the fragment itself is passed through along with the name. If the name is empty, it will auto-generate one. It will then call Route.submit() with the context, path (null if new), title, latitudes, longitudes, altitudes, and callback (aka this fragment). On success, onRouteEditSuccess() will be called with the new path. It will then loop through submitting the saved points of interests, which will then call onSubmitPointOfInterestSuccess() on success. This will increase the number of points submitted for statistics purposes and close the fragment. If either submitting the path or points fail, their respective failure methods will be called to show a toast notification.

Then, the mark point button has its click listener set to open a NameDialog with the PointOfInterestType. onEditName() will be called with the type and name when the dialog is saved. This is used for both paths and points of interest, so there is an if statement to check the type. If it is a point of interest, it will check if the name is empty, auto-generate one if it is, then add the name and coordinates to ArrayLists to submit at the end.

After the mark point button’s listener has been set, the map will initialise, running getMapAsync() which will trigger onMapReady(). onMapReady() will configure the map, setting it to hybrid, disabling UI features, and enabling the user’s location. It will then create polylines for other routes if it is not a new path. Then it will initialise a new polyline to follow the user. Finally it will get the location to centre the map on the user and call startLocationUpdates().

startLocationUpdates() will create a new LocationRequest, setting it to high accuracy, and polling every 2 seconds. It will then check the location permission, and start requesting location updates using Looper.myLooper(), and locationCallback as the callback. locationCallback is an instance of LocationCallback initialised as a global variable which will call onLocationChanged() with the location. onLocationChanged() will call addLocation(), which will add the coordinates to their respective ArrayLists if it’s between 2 and 25 metres from the last point, and update the distance walked. If there have been more than 10 points marked, the stop recording button will be enabled. Back to onLocationChange(), it will recentre the camera on the user and update the polyline with the updated coordinate set.

This fragment is the only one in the app to implement onPause() and onResume() overrides. This is for the purpose of creating and ending the LocationService when the app is no longer in the foreground. onPause() will stop location updates in the fragment and create an intent to start the service. onResume() will register a BroadcastIntent to receive the coordinates from the LocationService and add them to the existing ArrayLists. It will then stop the service. It also implements onSaveInstanceState() to backup the coordinates should the fragment be destroyed and recreated. Finally, it also implements onDestroy() which will send the walked distance to the PreferenceManager for updating statistics.

#### ReviewListFragment

This fragment requires a review type and parent object ID to be passed to newInstance() to create. In onCreateView() it will first get the parent object, depending on whether the type is PATH or POINT\_OF\_INTEREST. It will get the review list from these respective objects and initialise a ReviewListRecyclerViewAdapter using it. It will then set the RecyclerView to use either one or two columns depending on if the phone is in portrait or landscape mode.

The RecyclerView will then have its scroll listener set so that when it reaches the bottom, the next page of reviews will be loaded by calling updateReviewsList() with false, indicating it is not a full refresh. updateReviewsList() will then call getReviews() on the parent object, using the fragment as a GetReviewsCallback. On getting the reviews successfully, updateRecyclerView() will be called. This will check if the user has their own review, and will show it if so. It also updates the data set within the RecyclerView with new entries. If neither the user’s own review nor any other review exists, a message will display as such. It also sends fragment results to listeners for “update\_rating” and “update\_point\_rating”, making them update their rating bars. The SwipeRefreshLayout containing the reviews will then have its refresh listener set to call updateReviewsList() with true, refreshing the data from page 1.

The write review button will then have its click listener set to open a ReviewDialog. When the ReviewDialog is saved, onEditReview() will be called, simply calling updateRecyclerView() again.

Finally, updateReviewsList() is called with true to load the initial data.

#### RouteListFragment

This fragment requires a path ID be passed to newInstance() to create. In onCreateView(), the path will be gotten from the DataMap, and the route list will be gotten from the path. A RouteListRecyclerViewAdapter will then be initialised with the route list. Then onStart() will be called, running getMapAsync() on the map. onMapReady() will be called when the map has been initialised, setting the map type to hybrid and drawing colour-coded polylines to show each route separately, coordinated with the colours of the indicators in the RecyclerView. The map will then centre around the path’s boundaries.

This fragment also contains selectRoute(), which is called from the RouteListRecyclerViewAdapter with a position. This will hide all polylines except for the one in said position. It will then enable the select route button, setting its listener to open a WalkFragment with the path and position. Additionally, if the route was created by the user, the delete route button will be made visible and have its click listener set to open a confirmation dialog, where when yes is pressed, the route’s delete() method will be called, using a new RouteModifyCallback that will update the RecyclerView on success, and automatically go back to the main screen if there are now 0 routes (as the path will no longer exist). On failure it will simply show an error toast.

#### SettingsFragment

This fragment is created with newInstance() without any arguments. In onCreateView(), it will initialise each of the buttons. First is the set name button, which will open a NameDialog with the type USERNAME. On saving the dialog, onEditName() will be called where a JsonObject request will be constructed to send the name and device ID to the server using Retrofit. On success, the dialog will be dismissed and a success toast will be shown. On failure, a failure toast will be shown.

Next are the statistics and goals buttons. These will have their click listeners set to open a StatisticsFragment and a GoalsFragment respectively.

Then come the export and import settings buttons. Both will first ask for external storage permissions, opening a PermissionsFragment if denied. The export settings button will use a CustomActivityResultContract - ExportSettings(), to select a location to save a new JSON file. Upon this file location being selected, the Uri will be passed to the exportPreferences() method in PreferencesManager. The import settings button will use the build in OpenDocument() ActivityResultContract. This will open a file browser for the use to select a backed up file, which will then have its Uri passed to importPreferences() in PreferencesManager().

#### StatisticsFragment

There is essentially nothing in this fragment. It is created with newInstance() using no arguments, then onCreateView() basically just sets up a StatisticsListRecyclerViewAdapter.

#### WalkFragment

This fragment requires a path ID and a route number be passed to newInstance() to create. -1 may be passed as a route number if the path’s explore button is used to show all routes rather than a specific one. In onCreateView(), a timer will first be created for 30 seconds. Upon the timer finishing, the path’s walk() function will be called, increasing the number of times it’s been walked.

It will then set the mark point button to open a NameDialog, returning the entered name to onEditName() where a name will be auto-generated if it is empty, and then will submit a point of interest from the current coordinates with the entered name, using this fragment as the callback as it implements PointOfInterestSubmitCallback. Upon success, a marker will be added to the map. On failure, an error toast will be shown. The add photo button will then have its click listener set to open a PhotoDialog. Upon submission of a photo from the dialog, a success toast will be shown.

The map will then be initialised with getMapAsync(), calling onMapReady() when done. This will configure the map type and UI settings, then draw either the route number’s polyline or polylines for all routes on the map if -1 was submitted. Markers will also be generated for points of interest. The marker click listener will be set to the fragment, as it implements OnMarkerClickListener, and will call onMarkerClick(), opening a PointOfInterestFragment for the marker’s point. The map will be centred on the user, and then startLocationUpdates() will be called where a location request will be generated and the location provider will be started in a loop. When the location is retrieved, it will call use the global variable locationCallback as a callback, which in turn will call onLocationChanged(). This will add to the distance walked, recentre the map on the user, then calculate the distance away from the nearest marked point on the route. If this distance is over 10 metres, a banner will be displayed at the top, pointing an arrow towards the nearest point and giving the distance away.

# Server

## Basic Information

This server is written in Python utilises the Flask framework to handle requests. It uses a number of pip dependencies, which can be found in requirements.txt, but the most important ones are SQLAlchemy, for handling SQLite queries, Marshmallow, for serialising and deserialising SQLite entries, and Pillow, which is used for image processing.

## app.py

This file is very simple. It will simply configure the database, initialise SQLAlchemy and Marshmallow, register the endpoints, ensure the images directory exists, then launch the server.

## gets.py

This file contains a number of methods for handling requests intended for retrieving information. Despite the name, these requests can be POST requests, but do not save anything to the server.

First, get\_submitter() is defined. This will search the database for a user with a device ID, create it if it doesn’t exist, and return it. This is used in a number of methods.

Onto the request methods, get\_path() will search for a path in the table with the ID specified in the URL. If it doesn’t exist, it will return a 404. If it is a POST request with a device ID, it will check if the user is the creator of any of the routes or points of interest in the path, setting their editable attributes to true. Finally it will use the PathSchema to dump it to a JSON format and return it.

get\_paths() will use boundaries provided in query string parameters to search paths for those within the boundaries. If the boundaries are greater than 3 degrees in either latitude or longitude, it will return a 500 rather than searching for performance reasons. Once it has the in range paths, it will repeat what is done in get\_path() for each of them then dump and return them in JSON format.

get\_poi() will search for a point of interest with the ID in the URL. If it does not exist, it will return a 404. It will then check if the user making the request submitted the point, and set its editable attribute to true if so. It then dumps and returns it as JSON.

get\_pois() will first check for points within provided boundaries, similar to get\_paths(), then set the editable attribute for each one before dumping and returning it.

path\_review\_list() and poi\_review\_list() are 2 almost identical methods, with the only difference being the URL and whether the paths or points of interest are queried. These will get the path or point of interest with the ID provided, then get either PathReviews or PointOfInterestReviews filtered by that ID, and ordered from newest to oldest. It requested with a POST, it will then check if the requested user has submitted a review, and if they have will set it to the variable own\_review. It will then replace the digit submitter attribute in the reviews with the corresponding user nicknames, before paginating it based on the query string page and dumping it, returning the review list, the own review, and the average rating. path\_pictures\_list() and poi\_pictures\_list() are again 2 almost identical methods, both extremely similar to the review\_list methods, except querying the pictures tables instead of the review tables and without any own\_picture type thing, returning just the paginated list.

get\_group\_walks() will use the path ID in the URL to find group walks with that ID. It will then check the group walk times and filter out any that have already passed, replacing the digit submitter with the nickname on ones that haven’t passed. If requested with POST, it will then check if the user is the creator of the group walk or attending it, setting the editable and attending attributes respectively. Finally it will dump the list and return it as JSON.

The last method in this file is picture(), which will return the picture with the filename provided in the URL.

## posts.py

This file contains a number of methods for saving data to the server from POST requests. It will first define a few reused functions – allowed\_file(), which will check if a submitted file is an image, process\_picture(), which will correct rotation and downscale images, get\_submitter(), which does the same as it does in gets.py, and get\_time(), which will get the current timestamp.

Onto request methods, set\_name() will get a nickname from a JSON request and save it to the name field in the user’s table entry. It will then return the user’s info, or a 500 error is it failed.

add\_route() will take arrays of latitudes, longitudes, and altitudes from a request JSON and make sure there are more than 10 points. If there are, it will get the minimum and maximum latitudes and longitudes to calculate the boundaries. If the request contains a path ID, it will update the path’s boundaries. Otherwise, it will create a new path. It will then create a new route with the submitter ID, time created, path ID, and the 3 coordinate arrays. After committing to the database, it will then set the editable attributes on the path’s points and routes before dumping and returning the path.

delete\_route() will get the submitter from a JSON request, then if the route with the ID in the URL was created by the submitter, delete it. If there are still routes left in the path. it will update the path’s boundaries by looping through each remaining route to find the minimum and maximum latitudes and longitudes. If there are no longer any routes, the path will also be deleted. On success it will return a simple success message.

edit\_path() will use the PathSchema to load the request JSON into an update() call on the path with the ID in the URL. It will return a simple success message on success, or an error 500 on failure.

walk\_path() will get the path with the ID in the URL, increase its walk count by 1, then return the new walk count on success.

add\_poi() will get a user with the ID in the JSON request, then use the PointOfInterestSchema to load other data from the request, before creating a new PointOfInterest with said data. After saving, it will set the editable attribute on the new point to true and return a JSON dump of the point.

edit\_poi() will use the PointOfInterestSchema to load the request JSON into an update() call on the point with the ID in the URL. It will return a simple success message on success, or an error 500 on failure.

delete\_point\_of\_interest() will check if the user from the device ID sent in the JSON request is the creator of the point of interest with the ID in the URL. If so, it will delete the point and return a simple success message.

add\_group\_walk() will create a new group walk with the time and title from the request, along with the path ID in the URL. It will return a dump of the group walk on success.

edit\_group\_walk() will use the group walk ID in the URL to retrieve the group walk with said ID. If the user from the device ID in the request is the creator of the group walk, it will call update() on the group walk with the request data. It will return a simple success message on success.

delete\_group\_walk() will use the group walk ID in the URL to retrieve the group walk with said ID. If the user from the device ID in the request is the creator of the group walk, it will delete the group walk. It will return a simple success message on success.

toggle\_group\_walk\_attendance() will use the group walk ID in the URL to retrieve the group walk with said ID. If the user from the device ID is currently in the attendees list, it will remove them. Otherwise, it will add them. It will commit the changes then return a dump of the group walk.

add\_path\_review() and add\_poi\_review() will first check if the JSON request contains a number between 1 and 5 inclusive and fail if not. If will then check if the parent object (path or point of interest depending on the function) has any ratings already, setting the average rating to the rating in the JSON and rating count to 1 if not, and calculating the average rating and incrementing the rating count by 1 if so. It will then create a new PathReview or PointOfInterestReview with the JSON data and return a dump of the review and the new average rating.

edit\_path\_review() and edit\_poi\_review() will check if the request contains a number between 1 and 5 inclusive, and that the device ID in the request is that of the user who created the review with the ID in the URL. If will then load the JSON request into the update() function on the review, and calculate the new average rating of the parent object. It will return the average rating on success.

delete\_path\_review() and delete\_poi\_review() will check if the device ID in the request is that of the user who created the review with the ID in the URL. If it is, it will delete the review, decrement the parent’s rating count, then, if the count is now 0, set the average rating to 0, or if it’s not 0, recalculate the average rating. It will return the average rating on success.

add\_path\_picture() and add\_poi\_picture() will first check if there is a file in the request, returning 500 if there isn’t. If there is, it will then check if allowed\_file() returns true with the file’s filename. If it does, it will pass the image to process\_picture() to compress and rotate it. It will then create a new database entry, either PathPicture or PointOfInterestPicture, and save the image to the images folder. It will then return a dump of the picture’s attributes.

edit\_path\_picture() and edit\_poi\_picture() will first check if the device ID in the request belongs to the user who submitted the picture with the ID in the URL. If it does, it will load the request into the update function(), commit the changes, then return a simple success message.

delete\_path\_picture() and delete\_poi\_picture() will first check if the device ID in the request belongs to the user who submitted the picture with the ID in the URL. If it is, it will remove the image from the images folder and delete the picture’s SQLite entry. It will return a simple success message on success.

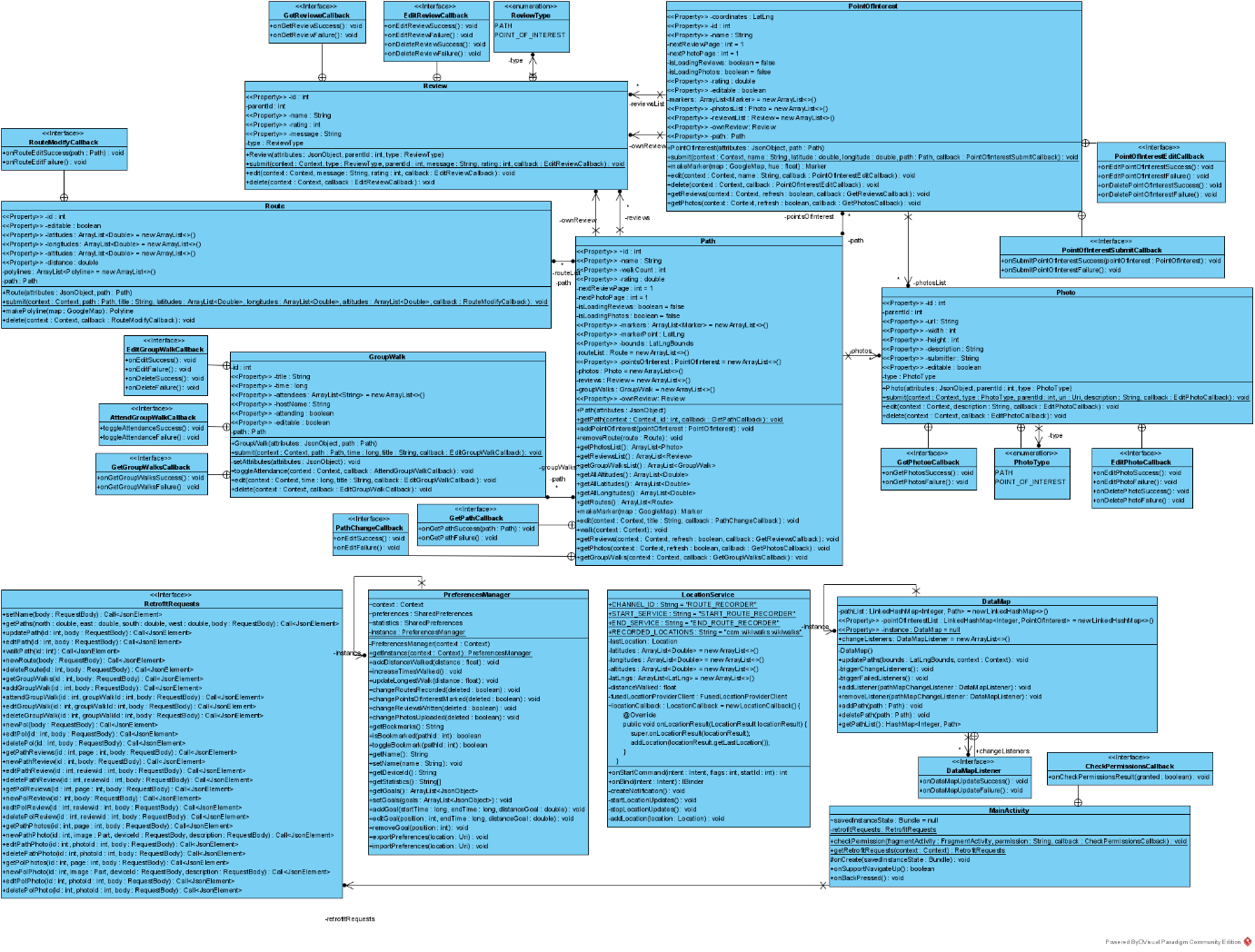
## schemas.py and tables.py

The content in these files are self-explanatory. tables.py will define SQLite tables with their columns, types and relationships. schemas.py defines Marshmallow schema classes that automatically are created using table data, along with a few additional fields that are edited when returning data.

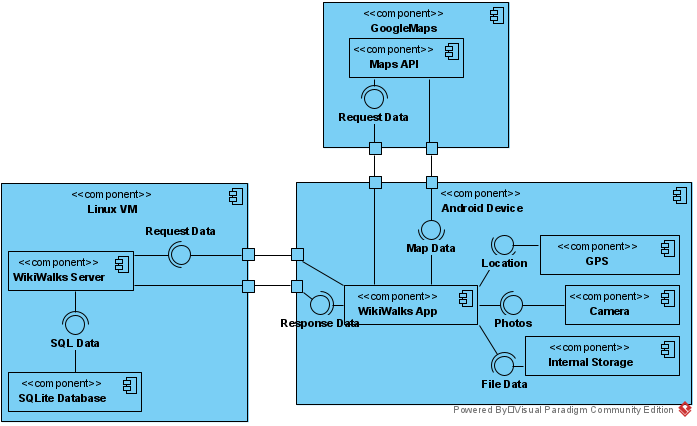
# UML Diagrams

These can be viewed at full size by right clicking the image and saving it.

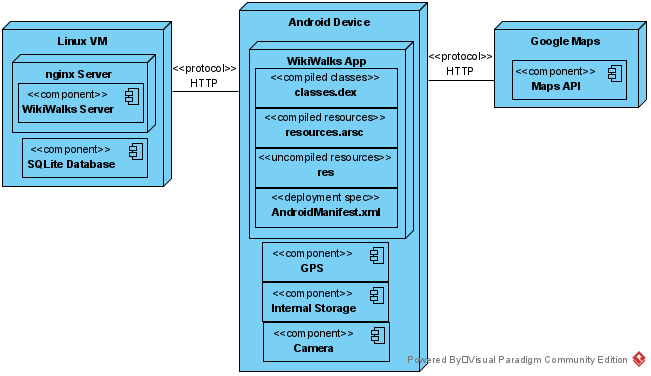
## Class Diagram



## Component Diagram



## Deployment Diagram



## Sequence Diagram

