JedAI Android SDK- Reference Manual

Version 6.0.3

October, 2017

Table of Contents

1. Introduction

This document will guide you through the integration process of the JedAI SDK.

The JedAI SDK includes several groups of services:

1. **Mobility Status User State Changes**
    a. Activity in which the user is engaged (i.e. walking, running, driving etc.)
    b. Type of location in which the user is situated (i.e. restaurant, home, mall etc.)
2. **Driving Related Events**
    a. Parking/Driving events
    b. Approach to car event
    c. Automatic Find My Car
3. **Routing functions**
    a. Walking Route Time
    b. Multi-Route Walking Time
    c. Multi-Points Routing Engine
4. **Locations History**
5. **Geo Fence Functions**

The rest of the document lists the required integration steps for each of the above groups of services and any additional maintenance information.

For further assistance, please contact info@anagog.com.

While the JedAI SDK document is built to be self-explanatory, we recommend that you consider a training session with the Anagog team.

2. Environment Set Up

This chapter explains the steps needed to set up the JedAI SDK environment.
The new version of the JedAI SDK uses AAR instead of JAR and maven as a means of fetching it instead of a JAR you copy to the "libs" directory.
If you are already using the (older) JAR SDK, see section 2.2.
If you haven't been using our SDK and want to start using it see section 2.1 below.

2.1. First time integration with JedAI SDK

Update build.gradle file as is described below.

Add the Anagog repository:

```
repositories {
    …
    maven {
        url "https://repository.anagog.com/artifactory/gradle-release/"
    }
    …
}
```

The notation "…" refers to whatever code you already have in place. Just place the above 'maven' part anywhere in the '`repositories`' segment.

Add dependency to the JedAI SDK:

```
dependencies {
    …
    compile "anagog.pd.service:service:1.710.0"
    …
}
```

The notation "…" refers to whatever code you already have in place. Just place the above 'compile line anywhere in the "`dependencies`" segment.

2.2. Migrating from JAR to AAR

Remove the following from the manifest file (if you have them):

```
<service
    android:name="anagog.pd.service.MobilityService" android:process=":remote">
    <intent-filter>
        <action android:name="anagog.pd.service.MobilityService" />
    </intent-filter>
</service>
<receiver android:name="anagog.pd.service.AlarmReceiver"
android:exported="true" android:process=":remote">
</receiver>
<receiver android:name="anagog.pd.service.StartOnUser" android:process=":remote">
<intent-filter>
    <action android:name="android.intent.action.USER_PRESENT"/>
</intent-filter>
</receiver>
<receiver android:name="anagog.pd.service.StartOnBoot" android:process=":remote">
<intent-filter>
    <action android:name="android.intent.action.BOOT_COMPLETED"/>
    <action android:name="android.intent.action.QUICKBOOT_POWERON"/>
</intent-filter>
</receiver>
```

Follow the steps in section 2.1

3. Starting the Service

The Anagog mobility service should be set to start running when the user launches your application for the first time.

The recommended best practice is to put the following lines of code into your main activity onCreate() method:

```
Intent intent = new Intent ("anagog.pd.service.MobilityService");
intent.setClassName(getPackageName(), "anagog.pd.service.MobilityService");
startService(intent);
```

The service will now run in the background, even when your application is closed, and generate the mobility status events.

3.1. Requesting Permissions at Run Time

Starting from Android 6.0 (API level 23), users grant permissions to apps in run time. Please make sure the Anagog Service is run (using the code above) only after permissions are granted. The following is a code sample demonstrating how to do that:

```
//If permissions were not requested or granted:
if (ContextCompat.checkSelfPermission(PlacesAPIActivity.this,
    android.Manifest.permission.ACCESS_FINE_LOCATION) !=
    PackageManager.PERMISSION_GRANTED) {
       // Request the needed permissions
       ActivityCompat.requestPermissions(PlacesAPIActivity.this, new
       String[]{android.Manifest.permission.ACCESS_FINE_LOCATION,
       Manifest.permission.WRITE_EXTERNAL_STORAGE},
       PERMISSION_REQUEST_CODE);
}
else { //permissions granted
       startAngogService (); // this is a function that runs the service according
}


//Activity should override this function. Android will call this function after user
granted/denied permissions
@Override
public void onRequestPermissionsResult(int requestCode, String permissions[], int[]
grantResults) {
switch (requestCode) {
       case PERMISSION_REQUEST_CODE:
       if (grantResults.length > 0 && grantResults[0] ==
       PackageManager.PERMISSION_GRANTED) {
              startAngogService ();// this is a function that runs the service
              according to the instructions above
       }
       break;
}
```

4. Mobility Services User Events

This chapter explains how to interface with information concerning the user's current location and activity. The Anagog mobility service analyzes the various data coming from the user's smartphone sensors, such as GPS, Wifi, Bluetooth, accelerometer, and more. The Anagog proprietary algorithms then use this information to identify user mobility events, such as whether the user arrived at or left a point of interest (home, office, mall, etc.) or is engaged in some activity (walking, running, riding a bike, etc.).

4.1. Background

The user state mechanism will help you identify the user's current activity and location. The API is divided into two categories:

- Activity: contains various user activity types, such as running, walking, driving, etc.
- Location: contains various location types, such as office, home, mall, hospital, airport, etc.

Each user state (Running, Mall …) contains different values, depending on the state type and the confidence level (0 - 1) indicating the likelihood of the current user state.
The use state can be in Enter or Exit status, depending on your configuration. For example, you can decide to be notified that the user enters a running state when the confidence level is above 70% and exits when the confidence level is below 30%. (Note that Exit will only occur if the state was first Entered.)

*We recommend that you get occasional updates from Anagog to find out when support becomes available for new user state types.

4.1.1. Activity Types

There are several types of activities the service can detect when the user starts and ends them. The service sends a notification when it detects that the user has started a certain recognizable activity (i.e., the specific activity type state is *Entered*) and another notification when it detects that the user has stopped doing this activity (i.e., the specific activity type state is *Exited*).

Currently the service supports the following activity types:

```
DRIVING
CYCLING
WALKING
RUNNING
ASLEEP
```

4.1.2. Location Types

There are several types of locations the service can identify when the user enters or exits them. The service will send a notification when it detects that the user has *entered* a certain recognizable location type and another notification when it detects that the user has *exited* from this location.
Currently the service supports the following location types:

```
HOME
OFFICE
SCHOOL
MALL
DOG_PARK
PLACE_OF_WORSHIP
POLICE_STATION
PARKING_LOT
FOOTBALL_STADIUM
GYM
HOTEL
AIRPORT
SPORTS
RAILWAY_STATION
UNIVERSITY
HOSPITAL
GOLF
FUEL_STATION
```

Home and Office are special location types because the service needs to **learn** about them; this is a process that takes some time (days) as opposed to the other location types, which the service already knows about and can identify immediately.

4.1.3. Important Information About User States

- **Exclusivity** – Some of the user states are not mutually exclusive. A user can be in an office that also resides in a mall, and thus be in two locations at the same time.
- **Training** – To provide reliable results, detection of home and office locations require several days of initiation within the user device. Please take this into account when using these states in your app.
- **Confidence level** – This parameter describes how confident the service is that the user is actually in a specific state. The values are between 0 (not in this state) and 1 (100% sure the user is in this state). It is possible to request that updates for a specific state be sent from the service, starting from a specific confidence level. For example, you can request notification of an Asleep state only if the confidence level is 0.75 or above. In this case, notification will be sent every time the confidence level reaches 0.75.
- **Registering for notification** – To benefit from the user state information, you must either register to receive notification update events or initiate a request for the current user state.

4.2. Setting Up Notifications for User States

There are two initial steps you are required to take to use the user state mechanism: adding filters to your manifest and subscribing to user state notifications.

4.2.1. Adding Filters to Your Manifest

Add the following `receiver` section to your manifest:

```
<receiver android:name=".AnagogReceiver">
    <intent-filter>
      <!-- Activity based (Biking, Driving) updates -->
      <action android:name="anagog.pd.service.intent.USER_STATE_ACTIVITY"/>
      <!-- Location based (Home, Office, Hospital) updates -->
      <action android:name="anagog.pd.service.intent.USER_STATE_LOCATION_TYPE"/>
    </intent-filter>
</receiver>
```

4.2.2. Subscribing to User State Notifications

You can either subscribe to individual state changes or to all of them at once.

Define the State Types you wish to be notified of, along with their entry and exit thresholds. To do so you define and populate a UserStateConfig object as shown in the following code snippet:

```
// Create configuration for user state
UserStateConfig config = new UserStateConfig();
// Set up the interesting activity types. Entry level for each state is 60% and exit
is 30%
config.set(UserStateActivityType.ALL, 0.6f, 0.3f);
// Set up the interesting location types (Mall, Home)
config.set(UserStateLocationType.ALL, 0.6f, 0.3f);
```

You can then "tell" the service to register for the states you selected:

```
// Send the configuration to service
UserState.updateConfiguration(mContext, config);
```

See the above sections [Activity Types, Location Types] for the possible values for Activity and Location types. Keep in mind that the special type "All" that was used above is used to listen for any change of any type, without having to specify each state individually.

In the same way you used "All," you can instead specify individual types, as shown in the example below.
Subscribe to Driving, Mall and Home notification changes:

```
// Instantiate a new UserStateConfig object
UserStateConfig config = new UserStateConfig();
// Populate the configuration with the desired state types
config.set(UserStateLocationType.HOME, 0.6f, 0.4f);
config.set(UserStateLocationType.MALL, 0.62f, 0.3f);
config.set(UserStateActivityType.DRIVING, 0.7f, 0.3f);
// Inform the server about your choices
UserState.updateConfiguration(mContext, config);
```

4.3. Receiving and Processing User State Notifications

To receive and process the notifications for changes in user state, set your receiver as shown in the snippet below.

This will ensure you receive notifications about the activity types you selected, as described in Section 4.2.2 above.

```
Public class AnagogReceiver extends BroadcastReceiver
{
@Override
public void onReceive(Context context, Intent intent) {
   // The activity intent filter notifying of Activity State Changes
   if(UserState.ANAGOG_USER_STATE_ACTIVITY.equals(intent.getAction())) {
       // Get the activity data from the intent
       ActivityUserStateData data = UserState.extractActivityUserState(intent);
       // Get the activity type (Driving, Running)
       UserStateActivityType type = UserState.extractUserStateActivityType(intent);
       // Get the confidence level
       float confidence = data.getConfidenceLevel();
   }

   // The location intent filter notifying of Location State Changes
   if(UserState.ANAGOG_USER_STATE_LOCATION.equals(intent.getAction())) {
       // Get the location data from the intent
       LocationUserStateData data = UserState.extractLocationUserState(intent);
       // Get the location type (Hospital / Home)
       UserStateLocationType type = UserState.extractUserStateLocationType(intent);
       // Get the confidence level
       float confidence = data.getConfidenceLevel();
   }
}
}
```

4.3.1. Polling

You can also poll the current user state from the service, without waiting for a change notification to arrive.

The following code snippet will cause the service to send back a list of all user states and their confidence values.

```
// Define the data structure to hold the user states
private ArrayList<UserStateData> mUserState;
private DecimalFormat mDecimalFormat = new DecimalFormat("#0.00");

// This method will send a poll request to the service for the current user states
// of all supported states
public void requestCurrentActiveUserState() {
   // First register to the user state receiver
   getContext().registerReceiver(mUserStateReceiver, new
IntentFilter(UserState.ANAGOG_CURRENT_USER_STATE));

   // Send a command to the service to request the current user state for all
supported states
   UserState.getCurrentActiveUserState(mContext);
}
```

```
// This receiver will receive the response for the polling request and store it
// in the mUserState object defined above
private BroadcastReceiver mUserStateReceiver = new BroadcastReceiver() {
   @Override
   public void onReceive(Context context, Intent intent) {
       // Extract the user state data
       mUserState = UserState.extractCurrentUserState(intent);
   }
};

/**
* The following method is an example of how to build a string out of the
* user state sent back from the polling request
*/
private String userStateToString() {
  StringBuilder sb = new StringBuilder();

  for(UserStateData userStateData : mUserState) {
      switch (userStateData.getCategory()) {
          case Activity:
              sb.append(getActivityUserStateText((ActivityUserStateData)
userStateData) + "\n");
             break;
          case Location:
              sb.append(getLocationUserStateText((LocationUserStateData)
userStateData) + "\n");
             break;
      }
  }

  return sb.toString();
}


// This method builds a string from the activity type and its confidence level
private String getActivityUserStateText(ActivityUserStateData data) {
   return data.getType() + ": " + mDecimalFormat.format(data.getConfidenceLevel());
}

// This method builds a string from the location type and its confidence level
private String getLocationUserStateText(LocationUserStateData data) {
   return data.getType() + ": " + mDecimalFormat.format(data.getConfidenceLevel());
}
```

4.3.2. Polling the current user state for a single specific type

If you need to fetch the confidence level for a ***single*** specific activity or location type, use the following example:

```
// First register a receiver to receive the result of the poll request
mContext.registerReceiver(mReceiver, new
IntentFilter(UserState.ANAGOG_SPECIFIC_CURRENT_USER_STATE));
// Now request that the service send you the state (enter or exit and confidence
level)
// for the specific user state in which you are interested (in this example it is
DRIVING)
UserState.getCurrentSpecificUserState(mContext, UserStateActivityType.DRIVING);

// Define a receiver to receive the poll response
BroadcastReceiver mReceiver = new BroadcastReceiver() {
   @Override
   public void onReceive(Context context, Intent intent) {
      // Only a single UserStateData object is returned with the results for
       // the specific User State Type you asked for
      UserStateData data = UserState.extractCurrentSpecificUserState(intent);
   }
};
```

5. User Meta State

5.1. Background

The Meta State extends the User State with information on the user's activity, as opposed to locations. Although the User State responds relatively quickly to minor changes in the user's activity, the Meta State tries to provide a broader view of the state and ignore temporary interruptions.

For example, consider a user who is walking but stops at a pedestrian light. While waiting for the light to turn green she uses her phone to answer a chat. This could be considered a 'stop walking' activity since she indeed stopped walking, but in the bigger picture she is still in the middle of a walking activity. In this example, the User State would have shown that she exited the walking activity.

This is where Meta State comes in. It tries to ignore short interruptions and provide a high-level view of the activity. In the above example, if the light turns green within a normal timeframe for traffic lights, and the user resumes walking, the Meta State won't send an event that the user stopped walking and started again.

Due to its nature, the Meta State works slower than the User State when reporting that an activity started and ended.

The following sections explain how to subscribe to Meta State notification events and what information is received as part of the notification.

5.2. Activity Types Supported by Meta State

The Meta State supports notification for a user entering or exiting the following activities:

DRIVING
CYCLING
WALKING
RUNNING

5.3. Setting Up Notifications for Meta State

There are two initial steps needed to set up the Meta State mechanism: adding filters to your manifest and subscribing to Meta State notifications.

5.3.1. Adding Filters to Your Manifest

Add the following `receiver` section to your manifest:

```
<receiver android:name=".AnagogReceiver">
    <intent-filter>
    <action android:name="anagog.pd.service.intent.META_STATE_CHANGED" />
    </intent-filter>
</receiver>
```

5.3.2. Subscribing to Meta State Notifications

To receive and process the notifications for changes in Meta State, set your receiver as shown in the snippet below. It also contains an example of how to extract the data related to the event.

```
Public class AnagogReceiver extends BroadcastReceiver
{
  @Override
  public void onReceive(Context context, Intent intent) {
   if (intent.getAction().equals(MetaStateManager.ANAGOG_META_STATE_CHANGED)) {
      MetaStateData data = MetaStateManager.extractMetaState(intent);
      // Returns current Meta State Enum (e.g DRIVING, WALKING...)
      data.getMetaState();
      // Returns the event timestamp in milliseconds
      data.getTime();
      // Returns the duration the user was in this state in ms (Only for exit)
      data.getDuration();
      // Returns True if the event is Enter State, False for Exit state
      data.isEnter();
    }
  }
}
```

6. Driving Related Functions

The events and functionality in this chapter relate to driving and parking location. The following sections explain how to integrate Anagog's unique mobility services with an external application:

1. Parking / Depart (start driving) events
2. Approach the car event
3. Automatic Find My Car

6.1. Receiving Parking / Depart (start driving) events

To get Parking / Depart events in real time, add the following filters to your receiver. Note that *Depart* means that the user departed from her parking location and started ***Driving***.

```
<receiver android:name=".AnagogReceiver">
    <intent-filter>
    <action android:name="anagog.pd.service.intent.PARKING_EVENT " />
    <action android:name="anagog.pd.service.intent.DEPART_EVENT" />
    </intent-filter>
</receiver>
```

To handle the events in your receiver and extract the event information, add the following:

```
Public class AnagogReceiver extends BroadcastReceiver
{       @Override
        public void onReceive(Context context, Intent intent) {
            // Extract data included in the Intent
          Bundle extras = intent.getExtras();
                if(extras != null)
                {
              if (intent.getAction().equals("anagog.pd.service.intent.DEPART_EVENT"))
              {
                    double latitude = extras.getDouble("Lat");
                    double longitude = extras.getDouble("Long");
                    float accuracy = extras.getFloat("Accuracy");
                    long duration = extras.getLong("Duration");
                    }
              if (intent.getAction().equals("anagog.pd.service.intent.PARKING_EVENT"))
              {
                    double latitude = extras.getDouble("Lat");
                    double longitude = extras.getDouble("Long");
                    float accuracy = extras.getFloat("Accuracy");
                    long duration = extras.getLong("Duration");
                    long DriveDist = extras.getLong("DriveDistance");
                    }
          }
        };
}
```

6.2. Registering for Approach the car events

This event is fired when the service determines that a user is returning to his/her car and is (most likely) going to drive away. This event provides a confidence level (value between 1 – 3 1/2/3). The higher the level, the more likely it is that the user is approaching her car.
Note: This is not a certainty, rather it is an event occurring with a high probability.

To register for this event, add the following to the receiver section in the host's manifest:

```
<receiver  android:name=".AnagogReceiver">
    <intent-filter>
       <action android:name="anagog.pd.service.intent.APPROACH_LEVEL"/>
    </intent-filter>
</receiver>
```

To handle this event, you must add the following code into your receiver:

```
public class AnagogReceiver extends BroadcastReceiver
{
  @Override
  public void onReceive(Context context, Intent intent) {
    // Extract data included in the Intent
    Bundle extras = intent.getExtras();
    if(intent != null &&
       intent.getAction().equals("anagog.pd.service.intent.APPROACH_LEVEL"))
    {
       int level = 0;
       if (extra !=null) {
         level = extra.getInt("level");
       }
    }
  }
```

6.3. Automatic Find My Car

6.3.1. Overview

Automatic Find My Car is one of the fundamental functions provided by the Anagog mobility status service. By using the smartphone's built-in sensors, the mobility service can automatically identify when and where the smartphone owner parks her car, without any involvement from the user's side.

Testing operation of the function

To verify that you are able to invoke the Automatic Find My Car function correctly, do the following:

1. Install and invoke the app containing the JedAI SDK.
2. Make sure the device has a SIM card with a cellular data connection.
3. Enable GPS and Network Location.
4. Drive at least 1 or 2 kilometers.
5. Park the car. The car location will be logged after you walk about 40 meters away from the car.
6. Wait for a few minutes.
7. Request the estimated location of the parked car from your app.

6.3.2. Receiving the Location of Parked Cars

To request the estimated location of the parked car from within your app:

1. Register a receiver (i.e., "AnagogReceiver") in your manifest to listen to parking events.

```
<receiver  android:name=".AnagogReceiver">
   <intent-filter>
   <action android:name="anagog.pd.service.intent.PARKING_UPDATE " />
   </intent-filter>
</receiver>
```

2. Write a receiver function that receives the parked car location such as the following:

```
Public class AnagogReceiver extends BroadcastReceiver
{
   @Override
    Public void onReceive(Context context, Intent intent)
   {
          if
        (intent.getAction().equals("anagog.pd.service.intent.PARKING_UPDATE"))
         {
                Bundle extras = intent.getExtras();
                double latitude = extras.getDouble("Lat");
                double longitude = extras.getDouble("Long");
                float accuracy = extras.getFloat("Accuracy");
                long duration = extras.getLong("Duration");
         }
   }
}
```

3. Send the request using the following intent:

```
Intent i = new Intent("anagog.pd.service.GET_PARKING_UPDATE");
i.setClassName(getPackageName(),"anagog.pd.service.MobilityService");
startService(i);
```

The "anagog.pd.service.intent.PARKING_UPDATE" response with the parked car location is now sent from the service to the receiver you defined.

If the Lat and Long values are -1000, it means the car is not in a parked state according to the algorithm's state machine.

6.3.3. Testing with Fake Location

For testing purposes, it is possible to add an extra parameter and get back a fake location:

```
Intent i = new Intent("anagog.pd.service.GET_PARKING_UPDATE");
i.setClassName(getPackageName(),"anagog.pd.service.MobilityService");
i.putExtra("test", true);
startService(i);
```

7. Geo-Fencing

This chapter provides a detailed description of the Anagog geo-fencing functionality. This capability allows you to define a specific geographic area associated with alerts if the user enters or leaves this zone.

7.1. Registering for Geo-Fence Alerts

To register for geo-fence events in real time, add the following code to the receiver section in the host's manifest:

```
<receiver  android:name=".AnagogReceiver">
      <intent-filter>
            <action android:name="anagog.pd.service.intent.LOCATION_ALERT" />
      </intent-filter>
</receiver>
```

7.1.1. Defining a Geo-Fence Alert

To receive location alerts for a geo-fence around a pre-defined location, add the following code to the receiver section in the host's manifest:

```
Intent serviceIntent = new Intent("anagog.pd.service.ADD_LOCATION_ALERT");
serviceIntent.setClassName(getPackageName(),"anagog.pd.service.MobilityService");
serviceIntent.putExtra("latitude" , (double) Latitude);
serviceIntent.putExtra("longitude" , (double) Longitude);
serviceIntent.putExtra("radius" , (long) Radius);
serviceIntent.putExtra("id" , (long) Id);
serviceIntent.putExtra("exit" , (boolean) Enter);
serviceIntent.putExtra("enter" , (boolean) Exit);
startService(serviceIntent);
```

Description

- **Latitude** – Latitude of the alert zone center
- **Longitude** – Longitude of the alert zone center
- **Radius** – Radius in meters of the circle defining the alarm zone
- **Id** – An integer number associated with the alarm; must be unique
- **Enter** – Send an alert each time the zone is entered
- **Exit** – Send an alert each time the zone is exited

7.1.2. Removing a Geo-Fence Alert Location

To remove a specific geo-fence alert, use the following code in the receiver section in the host's manifest:

```
Intent serviceIntent = new Intent("anagog.pd.service.REMOVE_LOCATION_ALERT");
serviceIntent.setClassName(getPackageName(),"anagog.pd.service.MobilityService");
serviceIntent.putExtra("id" , (long) Id);
startService(serviceIntent);
```

Description

- **Id** – Unique ID number used when the alarm was added.

7.1.3. Removing All Alerts

To remove all geo-fence location alerts, use the following code in the receiver section in the host's manifest:

```
Intent serviceIntent = new Intent("anagog.pd.service.REMOVE_ALL_LOCATION_ALERT");
serviceIntent.setClassName(getPackageName(),"anagog.pd.service.MobilityService");
startService(serviceIntent);
```

7.1.4. Receiving Geo-Fence Location Alerts

To receive geo-fence location alerts in real time, use the following code in your app:

```
public class AnagogReceiver extends BroadcastReceiver {

public static final String ANAGOG_ALERT = "anagog.pd.service.intent.LOCATION_ALERT";
@Override
    public void onReceive(Context context, Intent intent) {

        if (ANAGOG_ALERT.equals(intent.getAction())) {
            Bundle extras = intent.getExtras();
            if (extras !=null) {
                double Latitude = extras.getDouble("latitude");
                double Longitude = extras.getDouble("longitude");
                float Accuracy = extras.getFloat("accuracy");
                double AlertLatitude = extras.getDouble("alertLatitude");
                double AlertLongitude = extras.getDouble("alertLongitude");
                long AlertRadius = extras.getLong("alertRadius");
                long AlertId = extras.getLong("id");
                boolean Exit = extras.getBoolean("exit");
                boolean Enter = extras.getBoolean("enter");

            }
        }
    }
}
```

Every time the user enters or exits the geo-fence alert zone, this intent will be sent by the service.

Description

- **Latitude** – Latitude of the phone's current location
- **Longitude** – Longitude of the phone's current location
- **Accuracy** – Accuracy in meters of the phone's current location
- **AlertLatitude** – Latitude of the alert zone center
- **AlertLongitude** – Longitude of the alert zone center
- **AlertRadius** – Radius of the circle that defines the geo-fence zone
- **AlertId** – ID of the geo-fence alarm triggered
- **Exit** – True if the user exited the zone
- **Enter** – True if the user entered the zone

8. Routing

This chapter describes the Anagog routing capabilities.

8.1. Walking Route Time

The service can provide the walking route time between two points.

8.1.1. Asking for Walking Route Time

To get the amount of time it took to walk from one point to a second point, use the following code in the receiver section in the host's manifest:

```
Intent WalkIntent = new Intent("anagog.pd.service.GET_WALKING_TIME");
WalkIntent.putExtra("fromLat",(double)fromLat);
WalkIntent.putExtra("fromLon",(double)fromLon);
WalkIntent.putExtra("toLat",(double)toLat);
WalkIntent.putExtra("toLon",(double)toLon);
WalkIntent.setClassName(getPackageName(),"anagog.pd.service.MobilityService");
startService(WalkIntent);
```

Description

- **fromLat** – Latitude of the origin
- **fromLon** – Longitude of the origin
- **toLat** – Latitude of the destination
- **toLon** – Longitude of the destination

8.1.2. Receiving the Walking Time

To get the total amount of time walked in seconds, use the following code in the receiver section in the host's manifest:

```
public String WALKING_TIME= "anagog.pd.service.intent.WALKING_TIME";
private BroadcastReceiver WalkingTimeBroadcastReceiver = new BroadcastReceiver() {
	@Override
	public void onReceive(Context context, Intent intent) {
		if (WALKING_TIME.equals(intent.getAction())) {
			int walkingTime = 0;
			Bundle extras = intent.getExtras();
			if (extras !=null) {
				walkingTime = extras.getInt("travelTime");
			}
		}
	}
};

IntentFilter intentFilter = new IntentFilter();
intentFilter.addAction(WALKING_TIME);
registerReceiver(WalkingTimeBroadcastReceiver, intentFilter);
```

8.2. Multiple Route Walking Time

The Anagog service provides the amount of walking time when the route includes several sources and destinations.

8.2.1. Asking for Walking Time Between Multiple Points

To get the walking time between several points, use the following code in the receiver section in the host's manifest:

```
Intent WalkIntent = new Intent("anagog.pd.service.GET_WALKING_TIME_MATRIX");
WalkIntent.putExtra("origins",);
WalkIntent.putExtra("destination",);
WalkIntent.putExtra("key",);
WalkIntent.setClassName(getPackageName(),"anagog.pd.service.MobilityService");
startService(WalkIntent);
```

Description

- **origins** – List of lat, lon points divided by |
  Example: 32.793967,34.990658|32.785443,34.987654
- **destinations** – List of lat, lon points divided by |
  Example:
  32.795013,34.996344|32.793967,35.000142|32.792957,34.997202|32.795013,34.996344|32.793967,35.000142|32.792957,34.997202|32.795013,34.996344|32.793967,35.000142|32.792957,34.997202|32.795013,34.996344|32.793967,35.000142|32.792957,34.997202
- **key** – Key for using this functionality. Please contact the Anagog support team to get this key

8.2.2. Receiving the Walking Time

To get the results of the walking time for multiple points, use the following code in the receiver section in the host's manifest:

```
Public String WALKING_TIME= "anagog.pd.service.intent.WALKING_TIME_MATRIX";
private BroadcastReceiver WalkingTimeBroadcastReceiver = new BroadcastReceiver() {
      @Override
      public void onReceive(Context context, Intent intent) {
            if (WALKING_TIME.equals(intent.getAction())) {
                  String walkingTime = 0;
                  Bundle extras = intent.getExtras();
                  if (extras != null) {
                        walkingTime = extras.getInt("WalkingTimes");
                  }
            }
      }
};

IntentFilter intentFilter = new IntentFilter();
intentFilter.addAction(WALKING_TIME);
registerReceiver(WalkingTimeBroadcastReceiver, intentFilter);
```

The information for all the walking routes is received as one JSON object, as illustrated in the code below. The result contains a row for each origin, and the list of destinations at which the user arrived. The JSON object provides the walking time from every origin to every destination.

```
{"status": "OK",
"origin_addresses": [ 32.094721,34.775959|32.084721,34.675959],
"destination_addresses": [ 32.085322,34.781817],
"rows": [
        {"elements": [
                {"walking_time":0.33442570456291}]},
        {"elements": [ {"walking_time":0}]}
]}
```

8.3. Multiple Points Routing Engine

The JedAI SDK provides routing from multiple points in a single query. The results are received from a proprietary routing engine that does the calculations in real time.

8.3.1. Registering Multiple Points Routing (Distance Matrix)

The Anagog service can provide the distances from one origin to several destinations in real time. This is done using a distance matrix.

To register for the distance matrix, add the following receiver code to your manifest. In this example the receiver name is "AnagogReceiver" but you are free to choose any name:

```
<receiver  android:name=".AnagogReceiver">
<intent-filter>
<action android:name="anagog.pd.service.intent.DISTANCE_MATRIX" />
</intent-filter>
<intent-filter>
```

This receiver gets the distance reports sent from the service. Since the service is running in the background, you will need to handle the events from the receiver and not from your app, which might be closed when the event is taking place.

8.3.2. Requesting Multi-Route Information

The following code allows you to actively request information for multi-point routes. In a single query, the service calculates a matrix of all routes between a set of origins and a set of subsequent destinations.

```
Intent intent = new Intent("anagog.pd.service.GET_DISTANCE_MATRIX");
intent.setClassName(getPackageName(), "anagog.pd.service.MobilityService");
intent.putExtra("origins", list of [lat, long] separated by | as String, e.g.
32.793967,34.990658|32.785443,34.987654);
intent.putExtra("destination", list of [lat, long] separated by | as String, e.g.
32.795013,34.996344|32.793967,35.000142|32.792957,34.997202|32.795013,34.996344|32.79
3967,35.000142|32.792957,34.997202|32.795013,34.996344|32.793967,35.000142|32.792957,
34.997202|32.795013,34.996344|32.793967,35.000142|32.792957,34.997202);
intent.putExtra("key", API Key as String);


startService(intent);
```

The service will send back the `"anagog.pd.service.intent.DISTANCE_MATRIX "` intent.

You can register your activity to receive this intent and handle it as follows:

```
public class AnagogReceiver extends BroadcastReceiver {

String ANAGOG_DISTANCE_MATRIX = "anagog.pd.service.intent.DISTANCE_MATRIX ";

@Override
public void onReceive(Context context, Intent intent) {
      if (ANAGOG_DISTANCE_MATRIX.equals(intent.getAction())) {
            Bundle extras = intent.getExtras();
            String distances = extras.getString("distances");

            }
      }

}}
```

The result returned from the service is in JSON format, including the origins and destinations requested. The distances are separated into rows, where each row includes the distances from that origin to all destinations. The API key defines how many origins and destinations are allowed in each request.

```
{
          "status": string,
          "origin_addresses": [],
          "destination_addresses": [],
          "rows": [{
                    “elements": [{
                              “distance”: double
                    }]
          }]
}
```

The example above was run with an API key that restricts the number of origins and destinations to 10. Therefore, a result is returned only for the first 10 routes in the list.

{"status": "OK",
"origin_addresses": [32.793967,34.990658|32.785443,34.987654],
"destination_addresses": [
32.795013,34.996344|32.793967,35.000142|32.792957,34.997202|32.795013,34.996344|32.793967,35.000142|32.792957,34.997202|32.795013,34.996344|32.793967,35.000142|32.792957,34.997202|32.795013,34.996344|32.793967,35.000142|32.792957,34.997202],
"rows": [
{"elements": [
{"distance":0.61478264412032},{"distance":1.0262837127521},{"distance":0.74908461384427},{"distance":0.61478264412032},{"distance":1.0262837127521},{"distance":0.74908461384427},{"distance":0.61478264412032},{"distance":1.0262837127521},{"distance":0.74908461384427},{"distance":0.61478264412032},]},
{"elements": [
{"distance":2.4289782878468},{"distance":2.8404793564786},{"distance":2.5632802575707},{"distance"

:2.4289782878468},{"distance":2.8404793564786},{"distance":2.5632802575707},{"distance":2.4289782878468},{"distance":2.8404793564786},{"distance":2.5632802575707},{"distance":2.4289782878468},]}]
}

9. Location History Retrieval

9.1. Overview

This function retrieves a list of the location points the user passed through in the previous 48 hours, or a maximum of 1500 locations (whichever is reached first).

During normal operation, the mobility service occasionally stores the location of the user's device. These locations are stored in 2 buffers: one for the current 24 hour period and the other for the previous 24 hours.

A 'day' starts the first time the service is installed and launched (via its hosting app). Every 24 hours, the older buffer is cleaned and the last 24 hours become the 'old' buffer's contents. The new day's locations are stored in a current 24 hour buffer.

The history records the user's locations during a period that varies from 47 hours, 59 minutes and 59 seconds—all the way to 24 hours. The history saved is based on how much time has passed since the 'new day' started.

9.2. Retrieving the Location History

To receive the list of location history, the hosting app must register an intent.

9.2.1. Registering for Location History

To register for the location history, add the following receiver code to your manifest:

```
<receiver  android:name=".AnagogReceiver">
   <intent-filter>
      <!-- any other actions you register to !-->
      <action android:name="anagog.pd.service.intent.LOCATION_HISTORY_RESPONSE_EVENT"/>
   </intent-filter>
<intent-filter>
```

9.2.2. Retrieving Location History

To retrieve the location history list, send an intent as shown below:

```
Intent locationHistoryIntent = new Intent("anagog.pd.service.LOCATION_HISTORY_REQUEST_EVENT");
locationHistoryIntent.putExtra("size", 1000);// Get 1000 last locations
locationHistoryIntent.setClassName(getPackageName(),"anagog.pd.service.MobilityService");
startService(locationHistoryIntent);
```

The **size** parameter depicts how many locations you want returned; the maximum number is 1500.

9.2.3. Location History Structure and Testing

For testing purposes, you can add an extra parameter that retrieves a list of fake locations. To do this, add the following line to your code:
`locationHistoryIntent.putExtra("test", true);`
The service will send back the `"anagog.pd.service.intent.LOCATION_HISTORY_RESPONSE_EVENT"` intent.

You can register your activity to get this intent and handle it using the following code:

```
public class AnagogReceiver extends BroadcastReceiver {
       String ANAGOG_LOCATION_HISTORY =
       "anagog.pd.service.intent.LOCATION_HISTORY_RESPONSE_EVENT";

       @Override
       public void onReceive(Context context, Intent intent) {
         if (ANAGOG_LOCATION_HISTORY.equals(intent.getAction())) {
           Bundle extras = intent.getExtras();
           if (extras !=null)
           {
              String[] latitude = extras.getString("latitude").split("_");
              String[] longitude = extras.getString("longitude").split("_");
              String[] accuracy = extras.getString("accuracy").split("_");
              String[] timestamp = extras.getString("locationTime").split("_");
              String[] clientTime = extras.getString("clientTime").split("_");
              String[] speed = extras.getString("speed").split("_");
              String[] bearing = extras.getString("bearing").split("_");
              String[] provider = extras.getString("provider").split("_");
           }
         }
       }
}
```

Description

- **clientTime** – Time in milliseconds at which the location was stored. Long.
- **latitude** – Latitude of the location. Double.
- **longitude** – Longitude of the location. Double.
- **accuracy** – Location's accuracy. Float.
- **locationTime -** UTC timestamp in milliseconds at which the server recorded the user's location. Long. This is not the same as the clientTime.
- **speed –** Speed in meters/second over ground, if it is available; otherwise return -1. Float.
- **bearing –** Bearing in degrees, if it is available; otherwise return -1. Float.
- **provider –** From where the location was received – "GPS" or "network". String

Each data item is a string of values separated by an underscore symbol "_" .

Example of history for three locations:
Client time: 1456141539939_1456141549939_1456146139939
Latitude: 33.333333_33.444455_33.555555
Longitude: 34.333333_34.444444_34.555555
Accuracy: 30.2_34.5_12.0
Location time: 1456141539939_1456141549939_1456146139939
Speed: 2.2_2.1_2.3
Bearing: 33.3_66.6_99.9
Provider: gps_network_gps

Each color above depicts information for the same location.

10. Stop Service

It may be necessary to stop the mobility service if the user chooses to opt out.

To stop the mobility status service, use the following intent:

```
Intent serviceInt = new Intent("anagog.pd.service.StopMobilityService");
serviceInt.setClassName(getPackageName(),"anagog.pd.service.MobilityService");
startService(serviceInt);
```

To start the service again, send the following intent:

```
Intent serviceInt = new Intent("anagog.pd.service.MobilityService");
serviceInt.setClassName(getPackageName(),"anagog.pd.service.MobilityService");
startService(serviceInt);
```