Dokumentation

Mobile-Sensing Library

Dennis Lawo

dennis.lawo@uni-siegen.de

Inhaltsverzeichnis

[Kurzbeschreibung 2](#_Toc512788198)

[Verwendete Libraries 2](#_Toc512788199)

[Sensoren 2](#_Toc512788200)

[Speicherung 2](#_Toc512788201)

[Events 2](#_Toc512788202)

[Funktionsbeschreibung 2](#_Toc512788203)

[Sensoren 2](#_Toc512788204)

[Location 2](#_Toc512788205)

[Google Activity 3](#_Toc512788206)

[Google Activity Transition 3](#_Toc512788207)

[Network 3](#_Toc512788208)

[Clustering 3](#_Toc512788209)

[Track 4](#_Toc512788210)

[RunningApplication 4](#_Toc512788211)

[ScreenOn 4](#_Toc512788212)

[EventBus 4](#_Toc512788213)

[Speicherung 5](#_Toc512788214)

[Upload 5](#_Toc512788215)

[Nutzung der Library 5](#_Toc512788216)

[SensingManager 5](#_Toc512788217)

[UploadManager 5](#_Toc512788218)

[Upload Manager 5](#_Toc512788219)

[Datenadapter Beispiel 5](#_Toc512788220)

# Kurzbeschreibung

Die im Folgenden vorgestellte Android Library enthält verschiedene Sensoren und verwaltet die Handhabung der von diesen gewonnen Messdaten von der Messung bis zur Speicherung. Dabei stellt die SensingManager-Klasse eine einheitliche Schnittstelle zur eigentlichen App dar. Des Weiteren wird über Events kommuniziert, welche die App abfangen kann, um jederzeit die aufgezeichneten Daten Live anzuzeigen oder hochzuladen.

# Verwendete Libraries

## Sensoren

* Intel Context Sensing SDK (https://software.intel.com/en-us/documentation/context-sensing-sdk-for-android-states-datasheet)
* Google Location (https://developers.google.com/android/reference/com/google/android/gms/location/package-summary)
* Google Activity Recognition & Google Activity Transition (https://developers.google.com/android/reference/com/google/android/gms/location/ActivityRecognitionClient)
* TNME / Clustering (https://github.com/martinste1n/TNME/tree/ProximityRideSharing)

## Speicherung

* ObjectBox (http://objectbox.io/)

## Events

* GreenRobot Event Bus (http://greenrobot.org/eventbus/)

# Funktionsbeschreibung

## Sensoren

Im Folgenden werden die implementierten Sensoren kurz beschrieben und deren resultierende Objekte vorgestellt.

### Location

Dieser Sensor ist mittels der GoogleLocationAPI implementiert. In einem gegebenen Interval wird regelmäßig die aktuelle GPS-Position des Gerätes erfasst.

|  |  |  |
| --- | --- | --- |
| **Parameter Name** | **Datentyp** | **Beschreibung** |
| Id | Long | ID in Datenbank |
| Timestamp | Long | Aufnahmezeitpunkt in Millisekunden |
| Lat | Double | Latitude Koordinate |
| Lng | Double | Longitude Koordinate |
| Speed | Float | Geschwindigkeit |
| isClustered | Boolean | Benutzt im Clustering Ja/Nein? |
| parentCluster | Long | ID des ClusterObjects |

Tabelle 1 Beschreibung des GLocationObjects

### Google Activity

Dieser Sensor ist mittels der Google Activity Recongition implementiert. In einem gegebenen Interval wird regelmäßig die aktuelle Activity(IN\_VEHICLE, ON\_BICYCLE, ON\_FOOT, STILL, UNKNOWN, TILTING, WALKING, RUNNING) aufgezeichnet.

|  |  |  |
| --- | --- | --- |
| **Parameter Name** | **Datentyp** | **Beschreibung** |
| Id | Long | ID in Datenbank |
| Timestamp | Long | Aufnahmezeitpunkt in Millisekunden |
| Activity | String | Name der Activity |
| Probability | int | Wahrscheinlichkeit der Activity |

Tabelle 2 Beschreibung des GoogleActivityObjects

### Google Activity Transition

Dieser Sensor ist mittels der Google Activity Recongition implementiert. Als Erweiterung zur regelmäßigen Speicherung der Activities (IN\_VEHICLE, ON\_BICYCLE, ON\_FOOT, STILL, UNKNOWN, TILTING, WALKING, RUNNING), gibt dieser Sensor einen Wert aus, wenn ein Übergang zwischen zwei Aktivitäten stattfindet.

|  |  |  |
| --- | --- | --- |
| **Parameter Name** | **Datentyp** | **Beschreibung** |
| Id | Long | ID in Datenbank |
| Timestamp | Long | Start der Activitiy |
| Endtime | Long | Ende der Activity |
| Activity | String | Name der Activity |

Tabelle 3 Beschreibung des GoogleActivityTransitionObjects

### Network

Dieser Sensor ist mittels der Intel Context Sensing SDK implementiert. In einem gegebenen Interval wird regelmäßig die der aktuelle Netzwerk-Status abgefragt. Wenn der Netzwerktyp (z.B. LTE oder WIFI) vom vorherigen Typ abweicht, wird ein neues NetworkObject erzeugt.

|  |  |  |
| --- | --- | --- |
| **Parameter Name** | **Datentyp** | **Beschreibung** |
| Id | Long | ID in Datenbank |
| Timestamp | Long | Aufnahmezeitpunkt in Millisekunden |
| NetworkType | String | Name des NetworkTypes |

Tabelle 4 Beschreibung des NetworkObjects

### Clustering

Dieser Sensor (aus https://github.com/martinste1n/TNME/tree/ProximityRideSharing) berechnet alle 24 Stunden für die vorhandenen Locations ClusterLocations.

|  |  |  |
| --- | --- | --- |
| **Parameter Name** | **Datentyp** | **Beschreibung** |
| Id | Long | ID in Datenbank |
| date | Long | Datum |
| firstseen | Long | Datum des ersten Betreten des Clusters |
| count | int | Anzahl der Locations |
| lat | double | Latitude Koordinate |
| lng | double | Longitude Koordinate |
| Metajson | String | Cluster Meta Daten als JSON |

Tabelle 5 Beschreibung des ClusterObjects

Der MetaJson beschreibt die Metadaten des Clusters, diese werden für die internen Berechnungen des Sensors benötigt und beinhalten unter anderem die konvexe Hülle von Locations um das Cluster sowie den Cluster Type.

### Track

Mit jeder neu aufgezeichneten Location prüft dieser Sensor, ob eine Überschneidung mit einem Cluster vorliegt, mit dem Ergebnis „Im Cluster“ oder „Außerhalb des Cluster“. Anhand dieser Ergebnisse wird eine Strecke als Weg („Außerhalb des Cluster“) zwischen zwei „Im Cluster“ Ergebnissen interpretiert.

|  |  |  |
| --- | --- | --- |
| **Parameter Name** | **Datentyp** | **Beschreibung** |
| Id | Long | ID in Datenbank |
| startTimestamp | Long | Startzeitpunkt |
| endTimestamp | Long | Endzeitpunkt |

Tabelle 6 Beschreibung des TrackObjects

### RunningApplication

Dieser Sensor prüft in einem gegebenen Interval, im Wachzustand des Smartphones, die aktuelle Vordergrund App und speichert diese.

|  |  |  |
| --- | --- | --- |
| **Parameter Name** | **Datentyp** | **Beschreibung** |
| Id | Long | ID in Datenbank |
| timestamp | Long | Zeitpunkt |
| ApplicationName | String | Name der Application |

Tabelle 7 Beschreibung des RunningApplicationObjects

### ScreenOn

Dieser Sensor prüft in einem gegebenen Interval, im Wachzustand des Smartphones, den aktuellen Status des Bildschirms

|  |  |  |
| --- | --- | --- |
| **Parameter Name** | **Datentyp** | **Beschreibung** |
| Id | Long | ID in Datenbank |
| timestamp | Long | Zeitpunkt |
| screenOn | boolean | Bildschirm An/Aus? |

Tabelle 8 Beschreibung des ScreenOnObjects

## EventBus

Innerhalb der App wird zur Kommunikation der EventBus von GreenRobot genutzt.

Das SensorDataEvent wird von den Sensoren gefeuert, wenn diese einen neuen Wert gemessen haben. Das Event enthält das SensorObjekt.

Das UploadEvent wird vom UploadManager in einem gegebenen Interval gefeuert und enthält jeweils eine SensorTimeseries, welche bereit zum Upload ist.

Das TrackEndEvent wird gefeuert vom TrackSensor, wenn dieser einen neuen Track hat und kann z.B. genutzt werden um dies dem User anzuzeigen bzw. die RatingActivity zu öffnen.

## Speicherung

Die Speicherung ist mittels der bereits oben beschrieben SensorObjects und der Library „ObjectBox“ implementiert. Die beschriebenen Objekte werden tageweise zusammengefasst in den so gennannten „SensorTimeseries“-Klassen.

## Upload

Die App, welche die SensingLibrary nutzt, muss für den Upload ein Datenadapter implementieren. Dies ist eine Klasse, welche das UploadEvent fängt und die darin gewünschte Timeseries im gewünschten Format hochlädt.

# Nutzung der Library

## SensingManager

Der SensingManager wird benötigt um die Sensoren zu starten. Dies kann wie folgt genutzt werden.

Module.*init*(*context*, "USERNAME");  
*sensMang* = Module.*getSensingManager*();  
*sensMang*.setSensingSetting(SensingManager.SensorNames.*Activity*,true);  
*sensMang*.setSensingSetting(SensingManager.SensorNames.*GPS*,true);  
*sensMang*.setSensingSetting(SensingManager.SensorNames.*WLANUpload*,true);  
*sensMang*.setSensingSetting(SensingManager.SensorNames.*ScreenOn*,true);  
*sensMang*.setSensingSetting(SensingManager.SensorNames.*Apps*,true);  
*sensMang*.setSensingSetting(SensingManager.SensorNames.*Network*,true);  
*sensMang*.setSensingSetting(SensingManager.SensorNames.*Track*,true);  
*sensMang*.setSensingSetting(SensingManager.SensorNames.*Cluster*,true);  
*startSensing*();

Zusätzlich zu den On/Off Settings gibt es noch die Möglichkeit das TrackingInterval sowie weitere Optionen mittels der „AdvancedSettings“ einzustellen.

## UploadManager

### Upload Manager

uplMang = Module.*getUploadManager*();

uplMang.setDailyUpload(*context*);  
*uploader* = new ParseUploader();

### Datenadapter Beispiel

public class ParseUploader {  
  
 public ParseUploader()  
 {  
 EventBus.*getDefault*().register(this);  
 }  
  
 // This method will be called when a MessageEvent is posted (in the UI thread for Toast)  
 @Subscribe(threadMode = ThreadMode.*BACKGROUND*)  
 public void onMessageEvent(UploadEvent event) {  
 uploadToParse(event.data);  
 }  
  
 public void uploadToParse(SensorTimeseries st)  
 {  
 try {  
 ObjectBoxAdapter oba = new ObjectBoxAdapter();  
 //GLocationif (st.getClass().getName().equals(GLocationTimeseries.class.getName())) {  
 GLocationTimeseries glocTimeseries = (GLocationTimeseries) st;  
 ParseObject po = new ParseObject("Location");  
 po.put("basetime",glocTimeseries.getTimestamp\_day());  
 po.put("meta", new JSONObject());  
 po.put("name", "Location");  
 po.put("icon", "Icons/smarthome/default\_18.png");  
 JSONArray ja = new JSONArray();  
 JSONObject jo = new JSONObject();  
 jo.put("name", "GeoJSON");  
 jo.put("type","Geo");  
 ja.put(jo);  
 JSONObject jo2 = new JSONObject();  
 jo2.put("name","Timestamp");  
 jo2.put("type","Number");  
 ja.put(jo2);  
 po.put("valueTypes",ja );  
 po.put("user", ParseUser.*getCurrentUser*().getUsername());  
 JSONArray values = new JSONArray();  
 for(GLocationsObject gLocationsObject: glocTimeseries.getValues()){  
   
 JSONObject value = new JSONObject();  
 value.put("type","Feature");  
 JSONObject geo = new JSONObject();  
 geo.put("type","Point");  
 JSONArray coords = new JSONArray();  
 coords.put(gLocationsObject.getLat());  
 coords.put(gLocationsObject.getLng());  
 geo.put("coordinates",coords);  
 value.put("geometry",geo);  
 JSONObject props = new JSONObject();  
 props.put("timestamp",gLocationsObject.getTimestamp());  
 value.put("properties",props);  
 values.put(value);  
 }  
 po.put("values",values);  
 glocTimeseries.setUploaded(true);  
 oba.updateSensorTimeseries(glocTimeseries);  
 po.saveInBackground(updateSensorTimeseriesUpdated(glocTimeseries));  
 }  
 }catch(Exception e){  
 undoUpdated(st);  
 }  
 }  
  
 public SaveCallback updateSensorTimeseriesUpdated(final SensorTimeseries st){  
 return new SaveCallback() {  
 @Override  
 public void done(ParseException e) {  
 if(e == null){  
   
 }else{undoUpdated(st);  
 }  
 }  
 };  
 }  
 public void undoUpdated(SensorTimeseries st){  
 ObjectBoxAdapter oba = new ObjectBoxAdapter();  
 //GLocation  
 if (st.getClass().getName().equals(GLocationTimeseries.class.getName())) {  
 GLocationTimeseries glt = (GLocationTimeseries) st;  
 glt.setUploaded(false);  
 oba.updateSensorTimeseries(glt);  
 }  
}