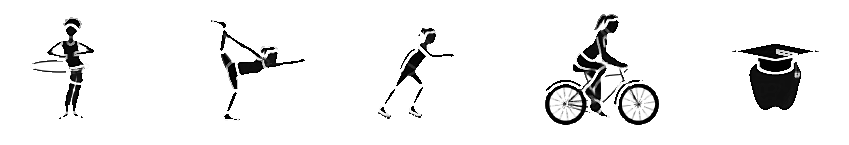
**INF 397 – Mobile Computing**

**Project 2:**

**User Activity Recognition App**

developed by **Katalina Dimitrova** and **Nikola Toshev**

1 May 2017

Our team has **successfully** developed the Android project and implemented all of the suggested features in the specifications (recognition of four user activities, displaying text for the appropriate activity, storing start time of each activity, displaying how long the previous activity has lasted).

*We hereby declare that the submitted project software is all our own work. We did not copy any software from anyone else or anywhere else. No one but us developed the project software.*

*We also declare that the contribution declared for each group member in this report is truthful and correct.*

Katalina Dimitrova \_\_\_\_\_\_\_\_\_\_\_\_\_

Nikola Toshev \_\_\_\_\_\_\_\_\_\_\_\_\_

1 May 2017

Functionalities:

* Activity Recognition Service – joint work
* Database interaction and helper classes – Nikola
* Last activity comparison and information – mostly Katalina
* UX and design (icons, images) – mostly Katalina

The application has one Main Activity which holds the GoogleApiClient and regulates the communication with the API. In the onCreate() method the client is initialized and connected to the Google Play Services, and the event listeners are associated with it.

**protected void** onCreate(Bundle savedInstanceState) {  
 **...**  
 **mApiClient** = **new** GoogleApiClient.Builder(**this**)  
 .addApi(ActivityRecognition.***API***)  
 .addConnectionCallbacks(**this**)  
 .addOnConnectionFailedListener(**this**)  
 .build();  
  
 **mApiClient**.connect();  
 **...**  
}

Once connected, a PendingIntent is created which wraps an Intent, which itself starts the ActivityRecognizedService. The idea behind using a PendingIntent is that the Activity Recognition API can start it when activities have been recognized, let the service associated with the PendingIntent process them, and then let the service close itself. This way battery is saved since the ActivityRecognizedService need not be running constantly, but only when necessary to process activities.

The PendingIntent is bound to the ActivityRecognitionApi in the following method:

**public void** onConnected(@Nullable Bundle bundle) {  
 Intent intent = **new** Intent(**this**, ActivityRecognizedService.**class**);  
 PendingIntent pendingIntent = PendingIntent.*getService*(**this**, 0, intent, PendingIntent.***FLAG\_UPDATE\_CURRENT***);  
 ActivityRecognition.***ActivityRecognitionApi***.requestActivityUpdates( **mApiClient**, 3000, pendingIntent);

getLoaderManager().initLoader(***ACTIVITY\_LOADER***, **null**, **this**);

}

In onConnected() we also initialize the loader associated with the activity. It takes the data from the database and it automatically tracks updates to the database.

The actual Loader is created in onCreateLoader:

@Override  
**public** android.content.Loader<Cursor> onCreateLoader(**int** id, Bundle args) {  
 Uri mUri = ActivityContract.ActivityEntry.***CONTENT\_URI\_FIRST\_TWO***;  
 **if** (**null** != mUri) {  
 *// Now create and return a CursorLoader that will take care of  
 // creating a Cursor for the data being displayed.* **return new** android.content.CursorLoader(  
 **this**,  
 mUri,  
 ActivityContract.ActivityEntry.***ACTIVITY\_COLUMNS***,  
 **null**,  
 **null**,  
 ActivityContract.ActivityEntry.***COLUMN\_ACTIVITY\_DATE*** + **"DESC"** );  
 }  
 **return null**;  
}

We create a CursorLoader which is associated with the Uri of our ContentProvider. Once created, the CursorLoader tracks changes to this Uri, which is taken care of by the ContentResolver API.

We take the data from the Loader in onLoadFinished():

@Override  
**public void** onLoadFinished(android.content.Loader<Cursor> loader, Cursor data) {  
 **if** (data.getCount() > 1) {  
 data.moveToPosition(0);  
 **long** currentActivityStartTime = data.getLong(ActivityContract.ActivityEntry.***INDEX\_ACTIVITY\_DATE***);  
 **int** currentActivityType = data.getInt(ActivityContract.ActivityEntry.***INDEX\_ACTIVITY\_TYPE***);  
 setImageAndDescription(currentActivityType);  
  
 data.moveToPosition(1);  
 **long** lastActivityStartTime = data.getLong(ActivityContract.ActivityEntry.***INDEX\_ACTIVITY\_DATE***);  
 **int** lastActivityType = data.getInt(ActivityContract.ActivityEntry.***INDEX\_ACTIVITY\_TYPE***);  
  
 **long** deltaTime = currentActivityStartTime - lastActivityStartTime;  
  
 **long** minutes = TimeUnit.***MILLISECONDS***.toMinutes(deltaTime);  
 **long** seconds = TimeUnit.***MILLISECONDS***.toSeconds(deltaTime) - TimeUnit.***MINUTES***.toSeconds(minutes);  
  
 String activityMessage = **"You "**;  
  
 **switch** (lastActivityType) {  
 **case** DetectedActivity.***IN\_VEHICLE***:  
 activityMessage += **"were in a vehicle for "**;  
 **break**;  
 **case** DetectedActivity.***WALKING***:  
 activityMessage += **"were walking for "**;  
 **break**;  
 **case** DetectedActivity.***RUNNING***:  
 activityMessage += **"were running for "**;  
 **break**;  
 **case** DetectedActivity.***STILL***:  
 activityMessage += **"were standing still for "**;  
 **break**;  
 }  
  
 Snackbar.*make*(findViewById(R.id.***root\_coordinatorlayout***),  
 String.*format*(activityMessage + **"%d minutes and %d seconds"**, minutes, seconds), Snackbar.***LENGTH\_LONG***).show();  
 }  
}

When the Loader is done loading, that means it has fetched the necessary data from the database. This method runs on the initial creation of the loader as well as after updates to the database. This means that when we add a new activity, the loader will reflect that immediately. Thus, we take the latest activity added, which is the current activity, subtract from its start time the start time of the activity before it in order to get the time that the previous activity had been done for and display it to the user. We also update the image and activity description to reflect the current activity.

**private void** setImageAndDescription(**int** activityType) {  
 **switch** (activityType) {  
 **case** DetectedActivity.***IN\_VEHICLE***:  
 **actionImage**.setImageResource(R.drawable.***vehicle***);  
 **actionDescription**.setText(**"In a vehicle"**);  
 **break**;  
 **case** DetectedActivity.***WALKING***:  
 **actionImage**.setImageResource(R.drawable.***walking***);  
 **actionDescription**.setText(**"Walking"**);  
 **break**;  
 **case** DetectedActivity.***RUNNING***:  
 **actionImage**.setImageResource(R.drawable.***running***);  
 **actionDescription**.setText(**"Running"**);  
 **break**;  
 **case** DetectedActivity.***STILL***:  
 **actionImage**.setImageResource(R.drawable.***still***);  
 **actionDescription**.setText(**"Standing still"**);  
 **break**;  
 }  
}

As for the actual database, we took what we had implemented for the London Tour Guide application and modified it for the needs of our application. The information we hold in the database is the activity type code (an integer number defined in the Activity Recognition API), as well as the start time of the activity as a number representing the time stamp.

**public static class** ActivityEntry **implements** BaseColumns {  
 **public static final** String ***TABLE\_NAME*** = **"activity"**;  
  
 **public static final** String ***COLUMN\_ACTIVITY\_TYPE*** = **"activity\_type"**;  
 **public static final** String ***COLUMN\_ACTIVITY\_DATE*** = **"activity\_date"**;  
  
 **public static final int *INDEX\_ACTIVITY\_ID*** = 0;  
 **public static final int *INDEX\_ACTIVITY\_TYPE*** = 1;  
 **public static final int *INDEX\_ACTIVITY\_DATE*** = 2;

...

}

The database schema is defined as:

**public static final** String ***DATABASE\_NAME*** = **"activities.db"**;  
  
**private static final** String ***SQL\_CREATE\_ENTRIES*** =  
 **"CREATE TABLE "** + ActivityContract.ActivityEntry.***TABLE\_NAME*** + **" ("** + ActivityContract.ActivityEntry.***\_ID*** + **" INTEGER PRIMARY KEY,"**

+ ActivityContract.ActivityEntry.***COLUMN\_ACTIVITY\_TYPE***

+ **" INTEGER NOT NULL,"** +  
ActivityContract.ActivityEntry.***COLUMN\_ACTIVITY\_DATE*** + **" INTEGER NOT NULL)"**;

The actual activity recognition processing happens in the ActivityRecognizedService:

@Override  
**protected void** onHandleIntent(Intent intent) {  
 **if** (ActivityRecognitionResult.*hasResult*(intent)) {  
 ActivityRecognitionResult result = ActivityRecognitionResult.*extractResult*(intent);  
 handleDetectedActivities(result.getProbableActivities());  
 }  
}  
  
**private void** handleDetectedActivities(List<DetectedActivity> probableActivities) {  
 **for** (DetectedActivity activity : probableActivities) {  
 Log.*d*(**"Got activity:"**, **""** + activity.getType() + **" with confidence: "** + activity.getConfidence());  
 **if** (relevantActivity(activity) && activity.getConfidence() > 66) {  
 **if** (**currentActivityType** == **null** || activity.getType() != **currentActivityType**) {  
 **currentActivityType** = activity.getType();  
 saveActivity(activity);  
 }  
 }  
 }  
}

As the service is started from an Intent, the onHandleIntent method is called first with an Intent which holds the recognized activities, which are then processed in the handleDetectedActivities method. What we do in order to process an activity is to first check if it is one of the activities of interest to us (standing still, walking, running, or in a vehicle) through the helper method relevantActivity and that it has a confidence of at least 67, where the confidence ranges from 0 to 100. If the recognized activity passes those checks, we make sure it is not the same as the currently ongoing activity, so that we do not have a duplicate entry in our database, and then we save it to the database using the saveActivity method. The code for the helper functions is:

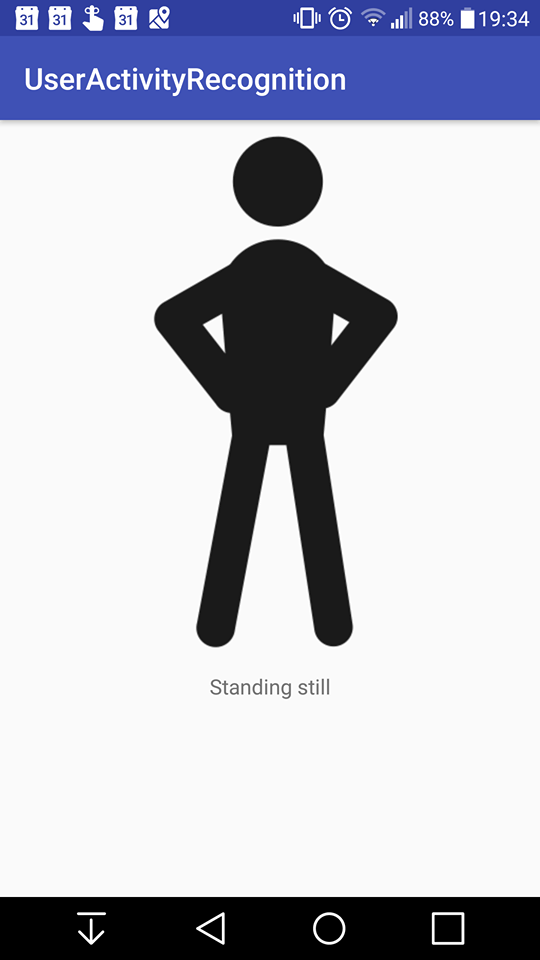
**private boolean** relevantActivity(DetectedActivity activity) {  
 **switch** (activity.getType()) {  
 **case** DetectedActivity.***IN\_VEHICLE***:  
 **return true**;  
 **case** DetectedActivity.***WALKING***:  
 **return true**;  
 **case** DetectedActivity.***RUNNING***:  
 **return true**;  
 **case** DetectedActivity.***STILL***:  
 **return true**;  
 **default**:  
 **return false**;  
 }  
}

**private void** saveActivity(DetectedActivity activity) {  
 ContentValues cv = **new** ContentValues();  
 cv.put(ActivityContract.ActivityEntry.***COLUMN\_ACTIVITY\_TYPE***, activity.getType());  
 cv.put(ActivityContract.ActivityEntry.***COLUMN\_ACTIVITY\_DATE***,

**new** Date().getTime());  
  
 getContentResolver().insert(ActivityContract.ActivityEntry.***CONTENT\_URI***, cv);  
}

The end result is an application that displays to the user the current activity he or she is doing as well as for how long they have been doing it once they start a new activity.

*The user is standing still The user is walking*





*The user is running The final project structure*

