Lab 07: SQLite, Shared Preferences, Alarms and Notifications

This lab will show how to save to the SQLite database the user's location and heart rate during an exercise, and sending them to the tablet once a connection is established. Moreover, you will learn how to schedule an Android alarm to later display a notification if the user is not training enough. Finally, you will learn how to add the info related to the notification as part of the user settings of our sport tracker app.

1 Structure app for saving and sending data

As shown in Figure 1, the first step is to save the heart-rate (HR) and location data in SQLite database in the watch, specifically in the **RecordingActivity**. In this case, we don't need to stream each value of the HR and location anymore, but we will send it all at once later, as shown in Figure 1. We will send the HR and location arrays of data to the **ExerciseLiveActivity**, so that we can send them to Firebase as already done in *Lab 6* in the specific branch of the recording.

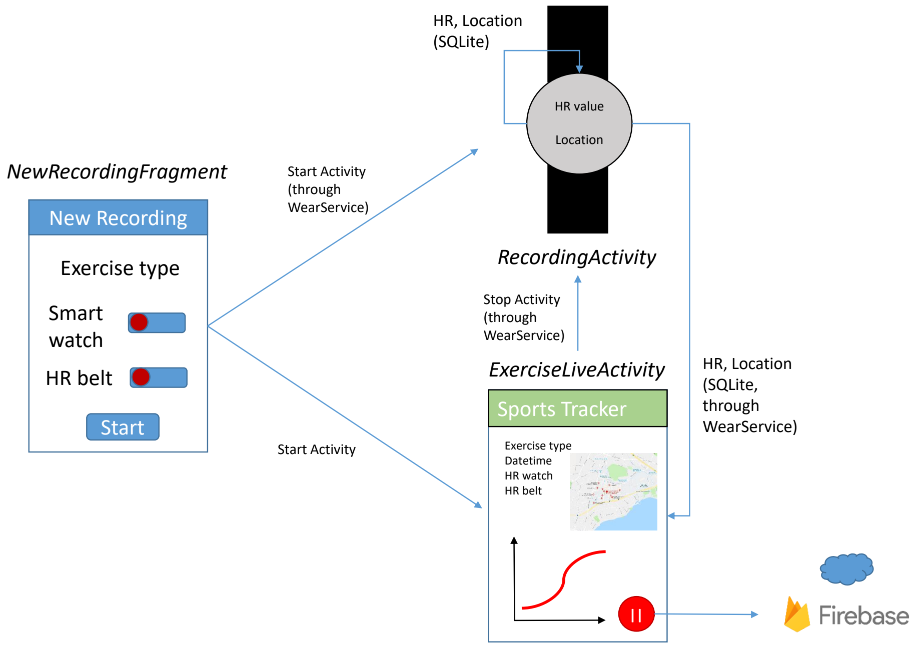


Figure 1: Lab structure for SQLite saving HR and location and sending to tablet

2 SQLite and the Room persistence library

Room provides an abstraction layer over *SQLite* to allow fluent database access while harnessing the full power of *SQLite*. Apps that handle non-trivial amounts of structured data can benefit greatly from persisting that data locally. The most common use case is to cache relevant pieces of data. That way, when the device cannot access the network, the user can still browse that content while they are offline. Any user-initiated content changes are then synced to the server after the device is back online. The core framework provides built-in support for working with raw *SQL* content. Although these *APIs* are powerful, they are fairly low-level and require a great deal of time and effort to use:

- There is no compile-time verification of raw *SQL* queries. As your data graph changes, you need to update the affected *SQL* queries manually. This process can be time consuming and error prone.
- You need to use lots of boilerplate code to convert between *SQL* queries and *Java* data objects. Room takes care of these concerns for you while providing an abstraction layer over *SQLite*.

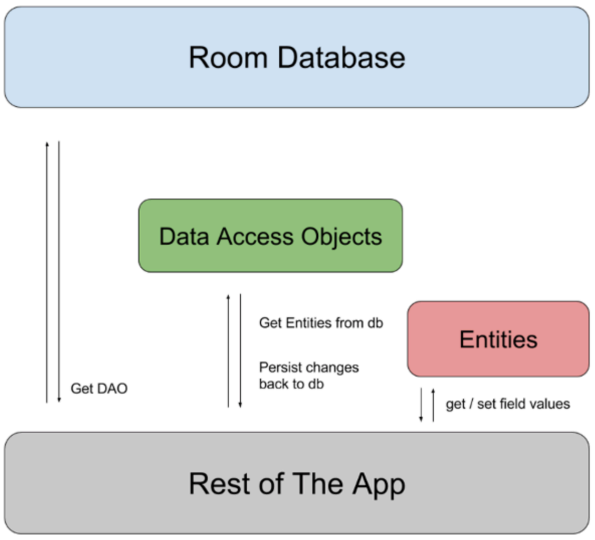


Figure 2: Room persistence library architecture

There are 3 major components in Room whose architecture is shown in Figure 2:

- **Database**: You can use this component to create a database holder. The annotation defines the list of entities, and the class's content defines the list of data access objects (DAOs) in the database. It is also the main access point for the underlying connection. The annotated class should be an abstract class that extends **`RoomDatabase`**. At runtime, you can acquire an instance of it by calling `Room.databaseBuilder()` or

`Room.inMemoryDatabaseBuilder()`.

- **Entity**: This component represents a class that holds a database row. For each entity, a database table is created to hold the items. You must reference the entity class through the entities array in the **Database** class. Entities are annotated with `@Entity`. Each field of the entity is persisted in the database unless you annotate it with `@Ignore`.
- **DAO**: This component represents a class or interface as a Data Access Object (DAO). DAOs are the main component of *Room* and are responsible for defining the methods that access the database. The class that is annotated with `@Database` must contain an abstract method that has 0 arguments and returns the class that is annotated with `@Dao`. When generating the code at compile time, *Room* creates an implementation of this class.

In order to use *Room*, we need to add the following dependencies in `build.gradle` file of the module wear (or app, depending on your project).

```
implementation ”android.arch.persistence.room:runtime:1.1.1”
annotationProcessor ”android.arch.persistence.room:compiler:1.1.1”
```

3 Saving HR and location in Room database

Let’s start by saving the HR and location in a local database in the watch. In order to do that, we need to create the 3 *Room* components that we call: **`SensorDataEntity`**, **`SensorDataDao`** and **`SportTrackerRoomDatabase`**.

3.1 SensorDataEntity

This class contains the data we want to save in our database, in particular HR, longitude and latitude. Note that the class needs to be annotated as **`Entity`** to be recognized by the *Room* library. Create the class in the wear module in the app package as **New → Java Class**.

```
@Entity
public class SensorDataEntity {
    // Different types of sensors
    public final static int HEART_RATE = 0;
    public final static int LATITUDE = 1;
    public final static int LONGITUDE = 2;
    // Primary key to access the row of SQLite table for the entity SensorData
    @PrimaryKey(autoGenerate = true)
    public int uid;

    // Different coloumn for different attributes of the entity SensorData
    @ColumnInfo
    public long timestamp;
    @ColumnInfo
    public int type;
    @ColumnInfo
    public double value;
}
```

Note that we are considering a primary unique key which gives access to the row of the SQLite tablet for the entity we are adding. We also add different attributes to the table of **SensorData** type objects.

3.2 SensorDataDao

This class is an **interface** object annotated as **Dao** to be recognized by *Room* as such. This interface is used to implement the queries of writing and reading into and from the SQLite database. Create the class in the wear module in the app package as **New → Java Class → Kind:Interface**. We define three methods to implement the queries:

- **getAllValues(...)** to get the all the values of a specific sensor type sorted by timestamp in descending order.
- **insertSensorData(...)** to insert a new data list into the database. We will insert data in chunks to avoid writing in the database each time we have a new sensor value.
- **deleteAll()** to delete all the data in the database.

Note the annotations for each method are queries in SQL, where two are manually written and one is already implemented by *Room* (**@Insert**).

```
@Dao
public interface SensorDataDao {
    // Implementation of the queries to use to access the database

    @Query("SELECT * FROM SensorDataEntity WHERE type = :sensorType ORDER BY " +
            "timestamp DESC")
    List<SensorDataEntity> getAllValues(int sensorType);
    @Insert
    void insertSensorDataEntityList(List<SensorDataEntity>
                                        sensorDataEntityList);

    @Query("DELETE FROM SensorDataEntity")
    void deleteAll();
}
```

3.3 SportTrackerRoomDatabase

This class is defined as **Database** in the *Room* annotations. Create this class in the wear module in the app package as **New → Java Class → Superclass:RoomDatabase**, **Modifiers:Abstract**. In this class, we define the type of entities that should be in the database and we call the **Dao** object to be able to do operations in the database. Moreover, we create two methods to instantiate our database that we call *SportTrackerDB* and to destroy this instance.

```
@Database(entities = {SensorDataEntity.class}, version = 1)
public abstract class SportTrackerRoomDatabase extends RoomDatabase {
    // Abstract class for inheritance: you don't implement the methods but you
    // can extend this class and implement them and add other features

    // Dao to associate to the database and use the queries implemented
    public abstract SensorDataDao sensorDataDao();

    // Instance of the database that will be used later
    private static SportTrackerRoomDatabase INSTANCE;

    // Constructor of the class. It's "synchronized" to avoid that concurrent
    // threads corrupts the instance.
    public static synchronized SportTrackerRoomDatabase getDatabase(Context context) {
        if (INSTANCE == null) {
            INSTANCE = Room.databaseBuilder(context, SportTrackerRoomDatabase
                    .class, "SportTrackerDB").build();
        }
        return INSTANCE;
    }

    // Method to destroy the instance of the database
    public static void destroyInstance() {
        INSTANCE = null;
    }
}
```

Notice that the database instance is a singleton to prevent having multiple instances of the database opened at the same time and the method to get the instance is declare as **synchronized** to avoid corruption of the instance by concurrent threads.

3.4 Writing HR and location in Room database

In the **RecordingActivity** of the wear module we are getting the HR and location in the methods **onSensorChanged(...)** and **onLocationChanged(...)**. In order to save these values in the database, we need to follow these steps:

- In the **onCreate(...)** method get the database instance with the following code

  ```
  sportTrackerDB = SportTrackerRoomDatabase.getDatabase
                      (getApplicationContext());
  ```

- Create a **List<Integer>** for the HR data and a **List<Location>** for the location data to collect the data and then save it in the database in chunks

- Create asynchronous tasks to perform operations on the database. Room requires data to be queried on a background thread and will throw an error if accessed on the main thread. **AsyncTask** enables proper and easy use of the UI thread. This class allows you to perform background operations and publish results on the UI thread without having to manipulate threads and/or handlers. We will create our own **SavingHeartRateAsyncTask** and **SavingLocationAsyncTask** that will extend **AsyncTask** and override the methods **doInBackground(...)** and **onPostExecute(...)**. The result of the method **doInBackground(...)** will be sent straight to the method **onPostExecute(...)**. Therefore, writing into the database and retrieving from it will be done within an asynchronous task. Notice that the **AsyncTask** takes three inputs: *params*, the type of the parameters sent to the task upon executio; *progress*, the type of the progress units published during the background computation; *result*, the type of the result of the background computation. So, if you want to do something with the result of the query the last parameter should not be **void**. Same for the input parameter, meaning that it won't be **void** only if you need to pass something to the **AsyncTask**. In our case, writing in the database will extend **AsyncTask<List<Integer>, Void, Void>**, while reading will extend **AsyncTask<Void, Void, List<SensorDataEntity>>**.

```java
public class SavingHeartRateAsyncTask extends AsyncTask<List<Integer>, Void,
    Void> {


private SportTrackerRoomDatabase db;

SavingHeartRateAsyncTask(SportTrackerRoomDatabase db) {
    this.db = db;
}

@SafeVarargs
@Override
protected final Void doInBackground(List<Integer>... lists) {
    // Insert the sensor data taken as input from the activity where the
    // task is executed
    List<Integer> hrValueList = lists[0];
    List<SensorDataEntity> sensorDataEntityList = new
            ArrayList<SensorDataEntity>();
    for (Integer hrValue : hrValueList) {
        SensorDataEntity sensorData = new SensorDataEntity();
        sensorData.timestamp = System.nanoTime();
        sensorData.type = SensorDataEntity.HEART_RATE;
        sensorData.value = hrValue;
        sensorDataEntityList.add(sensorData);
    }
    db.sensorDataDao().insertSensorDataEntityList(sensorDataEntityList);
    return null;
}
}
```

Try to do the **SavingLocationAsyncTask** in the same way as the HR!

- The final step is saving the HR values (and location) from the **onSensorChanged(...)** (or **onLocationChanged(...)** for location) of the **RecordingActivity** into a **List<↲ Integer>** (and List) and then performing the **AsyncTask** created.

```java
private List<Integer> hrList = new ArrayList<Integer>();
private int sizeListToSave = 10;

@Override
public void onSensorChanged(SensorEvent event) {
    int heartRate = (int) event.values[0];
    hrList.add(heartRate);
    // Save data when you have a multiple of sizeListToSave
    if (hrList.size() % sizeListToSave == 0) {
        SavingHeartRateAsyncTask hrAsyncTask = new
                SavingHeartRateAsyncTask(sportTrackerDB);
        hrAsyncTask.execute(hrList);
    }
}
```

4 Sending HR and location to tablet to save in Firebase

We want our data locally saved in the watch to be sent to the tablet, which will send it to Firebase. This means we need to:

- Read the data from the database through an **AsyncTask** in the receiver of **STOP_ACTIVITY** intent filter before finishing the activity (execute the task here). The **AsyncTask** should look like this

```
public class ReadingHeartRateAndLocationAsyncTask extends
    AsyncTask<Void, Void,
            List<SensorDataEntity>> {
// Room database and listener to check for task completed to avoid
// memory leaks
private final SportTrackerRoomDatabase db;
private final OnTaskCompletedListener onTaskCompletedListener;


// Constructor
ReadingHeartRateAndLocationAsyncTask(OnTaskCompletedListener
                                          onTaskCompletedListener,
                                  SportTrackerRoomDatabase db) {
    this.onTaskCompletedListener = onTaskCompletedListener;
    this.db = db;
}


@Override
protected List<SensorDataEntity> doInBackground(Void... voids) {
    List<SensorDataEntity> sensorDataEntityList = db.sensorDataDao()
            .getAllValues
                    (SensorDataEntity.HEART_RATE);
    db.sensorDataDao().deleteAll();
    return sensorDataEntityList;
}

@Override
protected void onPostExecute(List<SensorDataEntity> hrValues) {
    super.onPostExecute(hrValues);
    // Save heart rate in float array which we can send through an intent
    RecordingActivity.hrArray = new float[hrValues.size()];
    for (int i = 0; i < hrValues.size(); i++) {
        RecordingActivity.hrArray[i] = (float) hrValues.get(i).value;
    }
    onTaskCompletedListener.onTaskCompleted();
}
}
```

Notice that in this case we have only HR data, you can add the location if you already added it in the *Room* database in the previous section. Also, as soon as we read the data (and save them in a variable to send through an intent), we want to clear the database. The **AsyncTask** for reading data has a constructor with a listener (an interface). This listener is used to run **Activity** related things once the task is completed, since we cannot pass the **Activity** or **Context** as input to the **AsyncTask**, which will cause memory leaks. The interface you need to implement is the following (create a **New → Java Class → Kind:Interface**):

```
public interface OnTaskCompletedListener {
    void onTaskCompleted();
}
```

- In the **RecordingActivity** in the receiver of **STOP_ACTIVITY** after unregistering the sensor listener, you need to create the **OnTaskCompletedListener** and implement the method **onTaskCompleted()**. Then, you have to call and execute the reading data task.

```
OnTaskCompletedListener onTaskCompletedListener = new
    OnTaskCompletedListener() {
    @Override
    public void onTaskCompleted() {
        // Send HR array of float in an intent to WearService
        finish(); // finish activity
    }
};

// Execute the AsyncTask to read the data from Room database
ReadingHeartRateAndLocationAsyncTask hrAsyncTask = new
    ReadingHeartRateAndLocationAsyncTask
    (onTaskCompletedListener, sportTrackerDB);
hrAsyncTask.execute();
```

Notice that only when the task is completed we send an intent to the **WearService** and we finish the activity.

- Send the intent mentioned (in the **onTaskCompleted()** method) with the list of data (arrays of float) to the **WearService** in the wear module. You can retrieve both HR and location and send them in the same intent or send them separately.
- Retrieve the data in the mobile module as usual through the **WearService**. You can add a broadcast receiver in the **ExerciseLiveActivity** in the callback button to stop the exercise **stopRecordingOnWear(...)**. In the **onReceive(...)** you can retrieve the array of floats and fill the array **hrDataArrayList** that we were filling in *Lab 6* for each HR data arriving (remember to remove this line of code from the **HeartRateBroadCastReceiver** and do the same for the location).
- Then, send the list of values to Firebase in the specific recording branch (as we did in *Lab 6*) always in the **onReceive(...)** method of the newly created receiver.

5 Preferences

In Android, the preferences can be seen as a lightweight way to save data. It is used for saving the user settings, as well as other kind of data which is not controlled by the user.

It can store basic Java types, such as numbers (*bool*, *float*, *int*, *long*) and Strings. It is once again a key-value storage.

Having a settings page is a common pattern, hence Android provides for us a convenient way to define the user preferences as an XML file and will build an activity accordingly, displaying everything required.

5.1 Adding the menu button

First things first, we need to create a way to open the Settings page. In **MainActivity.java** of the mobile module, load a menu file for opening the preferences:

```
@Override
public boolean onCreateOptionsMenu(Menu menu) {
    getMenuInflater().inflate(R.menu.menu_main_activity, menu);
    return super.onCreateOptionsMenu(menu);
}
```

For the **menu_main_activity.xml** menu file itself, it only needs one item:

```
<?xml version="1.0" encoding="utf-8"?>
<menu xmlns:app="http://schemas.android.com/apk/res-auto"
    xmlns:android="http://schemas.android.com/apk/res/android">

    <item
        android:id="@+id/settings"
        android:title="@string/settings"
        app:showAsAction="never" />
</menu>
```

We now have the menu appearing in the activity ActionBar, but nothing happens yet as we haven't defined the listener in **MainActivity**:

```
@Override
public boolean onOptionsItemSelected(MenuItem item) {
    switch (item.getItemId()) {
        case R.id.settings:
            startActivity(new Intent(this, PrefsActivity.class));
            break;
    }
    return super.onOptionsItemSelected(item);
}
```

5.2 The PrefsActivity

The **Intent** we defined previously is looking for **PrefsActivity**, which we have to create accordingly. Do not forget to add it to the manifest (*Android Studio* will warn you about it):

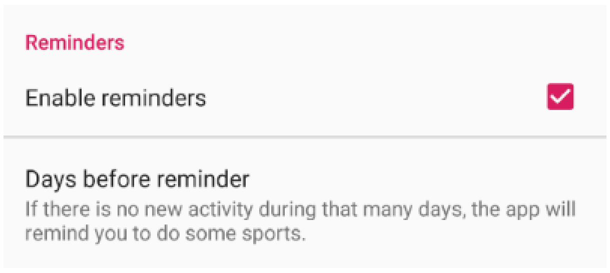
```
public class PrefsActivity extends PreferenceActivity {

    @Override
    protected void onCreate(Bundle savedInstanceState) {
        super.onCreate(savedInstanceState);

        addPreferencesFromResource(R.xml.preferences);
    }

}
```

This reads a description of preferences in an XML resource, which lives in an **xml** folder we have to create (**res** → Right-click → New → Android Resource Directory → Directory name and Resource type are *xml*). Add inside the required **preferences.xml** file with the following content:

```
<?xml version="1.0" encoding="utf-8"?>
<PreferenceScreen xmlns:android="http://schemas.android.com/apk/res/android">

    <PreferenceCategory android:title="@string/reminders">

        <CheckBoxPreference
            android:defaultValue="true"
            android:key="@string/key_enable_reminders"
            android:title="Enable reminders" />

        <EditTextPreference
            android:defaultValue="3"
            android:dependency="@string/key_enable_reminders"
            android:key="@string/key_reminder_days"
            android:selectAllOnFocus="true"
            android:inputType="number"
            android:singleLine="true"
            android:summary="@string/days_reminder_summary"
            android:title="@string/days_reminder_title" />

    </PreferenceCategory>
</PreferenceScreen>
```

This **preferences.xml** file has multiple interesting features:

- Preferences can be grouped in categories, which will appear with the given **title**,
- Multiple kind of preferences exist (we use **CheckBoxPreference** and **EditTextPreference** in our case),
- Preferences can have a **dependency** on one-another: in our example, disabling the checkbox (with a specific key) will disable the **EditTextPreference** (as it has a dependency on the checkbox's key),
- Input type can be added, to restrict typing to numbers rather than free-text.



Reminders

Enable reminders ✓

Days before reminder

If there is no new activity during that many days, the app will remind you to do some sports.

Figure 3: Screenshot of the preferences activity defined from XML input

Note: If you want or require to design your very own settings screen, it is possible, as we will soon learn how to save your own preferences. It is also interesting to have a look at the generated code when creating a *Settings Activity* from the Android wizard, as it is a multi-pane layout which presents alternative layouts on phones and tablets.

If you try the app now, the settings activity looks like Figure 3 and works well... most of the time. We have a crash if we can type in a huge number. Therefore, let's check that the user is giving a proper value for the reminder, by relying on the interface **SharedPreferences.OnSharedPreferenceChangeListener** which triggers each time the user changes one setting:

```
public class PrefsActivity extends PreferenceActivity implements
        SharedPreferences.OnSharedPreferenceChangeListener {

    @Override
    protected void onCreate(Bundle savedInstanceState) {
        super.onCreate(savedInstanceState);

        addPreferencesFromResource(R.xml.preferences);

        PreferenceManager.getDefaultSharedPreferences(this)
                .registerOnSharedPreferenceChangeListener(this);
    }
    @Override
    public void onSharedPreferenceChanged(SharedPreferences sharedPreferences,
                                          String key) {

    if (key.equals(getString(R.string.key_enable_reminders))) {
        // The reminders checkbox has changed
    } else if (key.equals(getString(R.string.key_reminder_days))) {
        // The reminders delay has changed
    }
}
```

From this initial code, we want to check the validity only when the reminder delay is updated, trying to parse the value as an integer. We use the **updateIntervalSetting** boolean as a flag to know if we need to overwrite bogus input from the user:

```
// Flag to correct (or not) the user input
boolean updateIntervalSetting = false;
int newIntervalDaysInt = 3; // Default number of days
String newIntervalDaysStr = sharedPreferences.getString(key,
        String.valueOf(newIntervalDaysInt));

// Try to parse the value as an integer
try {
    newIntervalDaysInt = Integer.parseInt(newIntervalDaysStr);
} catch (NumberFormatException e) {
    newIntervalDaysStr = null;
}

// Check if the result is valid
if (newIntervalDaysStr == null || newIntervalDaysStr.isEmpty() ||
        newIntervalDaysStr.matches("\\D")) {
    updateIntervalSetting = true;
    Toast.makeText(this, String.format(getString(R.string
            .invalid_days_value_defaulting_to_initial),
            newIntervalDaysInt), Toast.LENGTH_SHORT).show();
}
```

Note: We use `String.format(...)` in a similar way you can use formatting strings in C, where we can define placeholders for variables. In the previous code excerpt, the string is defined as follows:

```
<string name=”invalid_days_value_defaulting_to_initial”>
    Invalid value, defaulting to %d days.
</string>
```

Now we know if we have a valid value or not, we can further check if it sits in reasonable bounds (we don’t want a reminder for exercising on the same day, not too far in the future):

```
if (/* User input is invalid */) {
    // We already have this code
} else {
    // If valid, check restrict to acceptable range
    if (newIntervalDaysInt > MAX_UPDATE_INTERVAL) {
        updateIntervalSetting = true;
        newIntervalDaysInt = MAX_UPDATE_INTERVAL;
    } else if (newIntervalDaysInt < MIN_UPDATE_INTERVAL) {
        updateIntervalSetting = true;
        newIntervalDaysInt = MIN_UPDATE_INTERVAL;
    }

    // If out of bounds, warn the user
    if (updateIntervalSetting) {
        Toast.makeText(this, String.format(getString(R.string
                .invalid_days_value_out_of_bounds),
                newIntervalDaysInt), Toast.LENGTH_SHORT).show();
    }

}
```

Finally, we know if the user input is an integer or not, if it’s out of bounds or not and whether it requires to be updated or not. The final part is to actually save the updated setting if required, as follows:

```
// Save the modified value (if needed)
if (updateIntervalSetting) {
    sharedPreferences.edit().putString(key, Integer.toString
            (newIntervalDaysInt)).apply();
}
```

6 Alarms

Alarms are a solution provided by Android (as a *system service*) to fire an event *some time in the future*. It can be for starting an **Activity**, a **Service** or sending a **Broadcast**. It is not an alarm in the way it will get the device to ring and vibrate, it's an alarm which wakes-up the app! Keep in mind it's not the right solution for precisely scheduling and ordering short-term tasks. As a rule of thumb, use alarms only if you want to schedule something happening in *some* time (at least a minute) or while your app is *not* running.

6.1 Setting an alarm

Create a new **AlarmUpdater**`.java` file which sets the alarm in the system service (we will leave the **AlarmService** aside for the time being) through a `static` method called **updateNotificationSystem**`(...)`:

```java
class AlarmUpdater {

    static final String TAG = ”AlarmUpdater”;

    static void updateNotificationSystem(Context context) {
        Log.v(TAG, ”Setting up the AlarmManager”);
    }
}
```

The first thing to get is the **AlarmManager** from Android's system services:

```java
// Get the AlarmManager
AlarmManager alarmManager = (AlarmManager) context.getSystemService
        (Context.ALARM_SERVICE);
assert alarmManager != null; // Guarantees that alarmManager isn't null
```

We can now define the **Intent** which will be triggered once the alarm *rings*:

```java
// Create the intent which will be triggered when the alarm expires
// If an alarm exists with the same Intent, it will be canceled
Intent intentForService = new Intent(context, AlarmReceiver.class);
PendingIntent pendingIntent = PendingIntent.getBroadcast(context, 0,
        intentForService, PendingIntent.FLAG_CANCEL_CURRENT);
```

By reading the user preferences, we will define the alarm, or cancel it if required:

```
// Check if we need an alarm at all
SharedPreferences sharedPref = PreferenceManager
        .getDefaultSharedPreferences(context);
boolean reminders = sharedPref.getBoolean(context.getString(R.string
        .key_enable_reminders), true);
if (reminders) {
    // Get the delay between two activities and the last activity time
    long updateIntervalDays = Integer.parseInt(sharedPref.getString
            (context.getString(R.string.key_reminder_days), "3"));
    long lastTimestampMs = PreferenceManager
            .getDefaultSharedPreferences(context).getLong
                    (PrefsActivity.LAST_ACTIVITY_TIMESTAMP, Long
                            .MAX_VALUE);

    // Set the alarm
    long alarmTimestamp = lastTimestampMs + updateIntervalDays *
            AlarmManager.INTERVAL_DAY;
    alarmManager.set(AlarmManager.RTC, alarmTimestamp, pendingIntent);

    Log.v(TAG, "Set alarm in " + (alarmTimestamp - System
            .currentTimeMillis()) / 1000. + " " + "seconds");
} else {
    alarmManager.cancel(pendingIntent);
}
```

Now back in the **PrefsActivity**, after the tests of validity of the preferences (ie. just before exiting the **onSharedPreferenceChanged(...)** function), call the newly created method of the **AlarmUpdater** to ask Android to wake up our app whenever the user has not done any exercise for the number of days defined in the preferences:

```
AlarmUpdater.updateNotificationSystem(this);
```

6.2 One last preference

There is one additional thing connected to alarm we should save in the *Preferences* even though it is not controlled by the user. This is the time of the last exercise, as it's dependent on the next alarm t trigger.

When starting a recording, we will therefore directly save the current timestamp. In the **setOnClickListener()** linked to the **newRecording** button (from the **NewRecordingFr**⌋

agment's **onCreateView(...)**), we can add the following few lines to save the current timestamp, and schedule the next alarm:

```
PreferenceManager.getDefaultSharedPreferences(getContext())
        .edit().putLong(PrefsActivity
        .LAST_ACTIVITY_TIMESTAMP, System.currentTimeMillis())
        .apply();
AlarmUpdater.updateNotificationSystem(getContext());
```

6.3 Receiving the alarm

The only missing part now is the code to run when woken up by Android. Whenever the timer expires, a broadcast will call the **AlarmReceiver**. We therefore need to create it:

```
public class AlarmReceiver extends BroadcastReceiver {

    private final String TAG = this.getClass().getSimpleName();

    @Override
    public void onReceive(Context context, Intent intent) {
        CHANNEL_ID = context.getString(R.string.reminders);
        createNotificationChannel(context);

        Log.v(TAG, ”Woken up by the AlarmManager”);
    }
}
```

The receiver needs to be declared in the **AndroidManifest.xml** in a similar way you declare activities and services (when they have no special properties):

```
<receiver android:name=”.AlarmReceiver” />
```

6.4 Bug: ”My alarm doesn't survive a reboot!”

By design, Android clears all the alarms after a shutdown-reboot cycle. The app therefore needs to register for **BOOT_COMPLETED** events. This way, it is possible to register once again the alarm. If the alarm set is back in time, it will be triggered immediately after being registered.

Therefore, add a new receiver to the **AndroidManifest** accordingly, as well as the corresponding required permission **RECEIVE_BOOT_COMPLETED**:

```
<receiver
    android:name=".BootReceiver"
    android:enabled="true">
    <intent-filter>
        <action android:name="android.intent.action.BOOT_COMPLETED" />
    </intent-filter>
</receiver>
```

Create the **BootReceiver** accordingly (*Android Studio* code analyzer will help you), implement the required interface (*Android Studio* will help you once again), and simply call the method from the **AlarmUpdater** to restore the pending alarm which was removed during the shutdown.

6.5 Debugging alarms

Debugging the alarms can be quite tricky, so here are some little tricks to help you:

- **Use short alarms:** in our case, you can change the **alarmTimestamp** from the **AlarmUpdater.java** to use **lastTimestampMs + updateIntervalDays * 1000 * 60** so it fires in *N* minutes rather than *N* days
- Use the *Android Debug Bridge* to talk directly to the phone, listing the existing alarms. The command is the following: **adb shell dumpsys alarm** and the relevant output looks like the following (where *YYYY-MM-DD hh:mm:ss* is the scheduled date and time and *tttttttttttt* the corresponding timestamp):

```
...
RTC #0: Alarm{99b00ce type 1 when tttttttttttt ch.epfl.esl.sportstracker}
  tag=*alarm*:ch.epfl.esl.sportstracker/.AlarmReceiver
  type=1 whenElapsed=-535ms when=YYYY-MM-DD hh:mm:ss
  window=+44s997ms repeatInterval=0 count=0 flags=0x0
  operation=PendingIntent{e924bef: PendingIntentRecord{14756fc
                        ch.epfl.esl.sportstracker broadcastIntent}}
...
u0a170:ch.epfl.esl.sportstracker +53ms running, 0 wakeups:
+53ms 0 wakes 1 alarms, last -11m15s33ms:
  *alarm*:ch.epfl.esl.sportstracker/.AlarmReceiver
...
```

7 Notifications

Notifications appear in the top bar in Android. They started initially with very simple functionality and then grew over time to provide a very flexible interaction with the user (custom layout, expandable notification, action buttons, channels, etc.). In our case, we will display only a minimal notification, which opens the app when touched while disappearing automatically.

Before using notifications, Android requires since API 26 to use channels, so the user can filter out the unwanted notifications from an app. We therefore need to create a channel accordingly.

```
private String CHANNEL_ID = "";
private int notificationId = 0;

// ... and in the onReceive(...):

    CHANNEL_ID = context.getString(R.string.reminders);
    createNotificationChannel(context);

// ... and finally:

// Taken (almost) directly from:
// https://developer.android.com/training/notify-user/build-notification
private void createNotificationChannel(Context context) {
    // Create the NotificationChannel, but only on API 26+ because
    // the NotificationChannel class is new and not in the support library
    if (Build.VERSION.SDK_INT >= Build.VERSION_CODES.O) {
        CharSequence name = context.getString(R.string.channel_name);
        String description = context.getString(R.string
                .channel_description);
        int importance = NotificationManager.IMPORTANCE_DEFAULT;
        NotificationChannel channel = new NotificationChannel(CHANNEL_ID,
                name, importance);
        channel.setDescription(description);
        // Register the channel with the system; you can't change the
        // importance or other notification behaviors after this
        NotificationManager notificationManager = context
                .getSystemService(NotificationManager.class);
        notificationManager.createNotificationChannel(channel);
    }
}
```

Now the channel is ready, we want to create the notification. In the same way we had a **PendingIntent** which contained the intent to use when the alarm was triggered, we first create a **PendingIntent** which contains the intent to use once the user taps on the notification:

```
Intent intentMainActivity = new Intent(context, LoginActivity.class);
intentMainActivity.setFlags(Intent.FLAG_ACTIVITY_NEW_TASK | Intent
        .FLAG_ACTIVITY_CLEAR_TASK);
PendingIntent pendingIntent = PendingIntent.getActivity(context, 0,
        intentMainActivity, 0);
```

We now have all the building blocks, let's put everything together to have a wonderful notification as in Figure 4

```
NotificationCompat.Builder mBuilder =
        new NotificationCompat.Builder(context, CHANNEL_ID)
                .setContentTitle(context.getString(R.string.lets_exercice))
                .setSmallIcon(R.drawable.ic_logo)
                .setContentText(
                        context.getString(R.string.notification_content))
                .setPriority(NotificationCompat.PRIORITY_DEFAULT)
                .setAutoCancel(true)
                .setContentIntent(pendingIntent);

NotificationManagerCompat notificationManager =
        NotificationManagerCompat.from(context);
notificationManager.notify(notificationId, mBuilder.build());
```

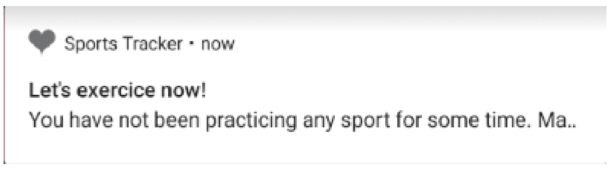

Sports Tracker • now
Let's exercice now!
You have not been practicing any sport for some time. Ma..

Figure 4: Notification displayed in the notification drawer