Lab 6: Sensors and GPS

In case you found something to improve, please tell us!
`https://forms.gle/Nf27cXFf7AwaBL55A`

In this lab you will learn how to use build-in heart rate (HR) sensor as well as the current user's location on android wear. Finally, you will add the user's current position along with the heart rate to the Firebase, and retrieve them in order to show them in the history of exercises.

1 Android Studio Tricks

Here are very useful **Android Studio** tricks you should always use (check *Section 5* of **Lab1b** for more detailed explanation on how to use **Android Studio** tools):

1. Use **Alt+Enter** (**Option+Enter** for Mac users) when you have an error in your code: put the cursor on the error and click **Alt+Enter**. You can also use it to update the **`gradle`** dependencies to the latest version.
2. Use **Ctrl+Space** to check the documentation of a **`View`**, method or attribute: put the cursor on the object and do **Ctrl+Space**. You can also use it to complete the typing of these objects. Otherwise **Android Studio** always gives a list of suggestions where you can choose the object you need.
3. **ALWAYS CHECK THE COMPILATION ERRORS!** They are usually quite self-explanatory.
4. **ALWAYS DEBUG AND CHECK THE ERRORS IN LOGCAT!** Read the usually self-explanatory errors and click on the underlined blue line to go in the position of the code where the error is.

For more **useful keyboard shortcuts**, please check this LINK[1]

2 Introduction to sensors

Most Android-powered devices have built-in sensors that measure motion, orientation, and various environmental conditions. These sensors are capable of providing raw data with high precision and accuracy, and are useful if you want to monitor three-dimensional device movement or positioning, or you want to monitor changes in the ambient environment near a device.

Some android wear watches also provide a heart rate sensor. In this lab, we will make use of it. Beginning in Android 6.0 (API level 23), users grant permissions to apps while the

[1] <https://developer.android.com/studio/intro/keyboard-shortcuts>

app is running, not when they install the app. This approach streamlines the app install process, since the user does not need to grant permissions when they install or update the app. It also gives the user more control over the app's functionality. For example, a user could choose to give a camera app access to the camera but not to the device location. The user can revoke the permissions at any time, by going to the app's Settings screen.

You will need to first add the **WAKE_LOCK** permission to allow the application to keep the watch awake while it's performing some processing. This is done by adding the following line in the **AndroidManifest** of the wear module:

```
<!-- wear -> AndroidManifest.xml -->
<uses-permission android:name = "android.permission.WAKE_LOCK"/>
```

Furthermore, in order to allow an application to access data from sensors that the user uses to measure what is happening inside their body, such as heart rate, you need to add to your **AndroidManifest** the following line:

```
<!-- wear -> AndroidManifest.xml -->
<uses-permission android:name = "android.permission.BODY_SENSORS"/>
```

We will handle this permission in a new activity we create in the wear module, called **RecordingActivity**. The following code asks for permissions for using the sensors of the wear:

```
/** wear -> RecordingActivity.java -> RecordingActivity -> onCreate(...) **/
if (Build.VERSION.SDK_INT >= Build.VERSION_CODES.M
    && checkSelfPermission("android.permission.BODY_SENSORS")
            == PackageManager.PERMISSION_DENIED) {
    requestPermissions(new String[]{"android.permission.BODY_SENSORS"}, 0);
}
```

Feel free to implement a part of the code in case of denying permissions to the app.

Remember to add the new activity in the **AndroidManifest.xml**.

3 Reading and displaying HR sensor data

In this lab, we want to acquire actual data for the recording session, meaning that we need to:

- from **mobile -> NewRecordingFragment.java**, start a new activity (**mobile -> ExerciseLiveActivity.java**) which shows live data

- start the wear app from **mobile -> ExerciseLiveActivity.java**
- read the HR sensor
- show the data in a simple layout on the watch
- send the HR to the tablet
- show the HR data in **mobile -> ExerciseLiveActivity.java**
- save the HR in Firebase

3.1 Simple layouts

For this lab, we create the layout of the **RecordingActivity** on the wear app. We will put two **TextView**s for now as a basic template layout. The first one will show the current value of the user's heart rate, whereas the second one will show the user's current position. We can implement the ambient mode as we did in the **MainActivity** in *Lab 2*.

Next, we create the layout of the new activity of the mobile module **ExerciseLiveActivity**. For now we put only one **TextView** that will show the HR value sent from the watch. You can also show the information about the new exercise in other **TextView**s similar to what we did in **MyHistoryFragment**.

3.2 Starting wear recording from tablet

In *Lab 5*, we implemented a button in the **NewRecordingFragment** to save the recording information on Firebase. We need to start the wear app and the new activity **ExerciseLiveActivity** after we save the recording info in Firebase.

In the button callback, we implement the following code to start the recording.

```
/** mobile -> NewRecordingFragment.java -> NewRecordingFragment
    -> onCreateView(...) -> newRecording.setOnClickListener(...)
    -> onComplete(...) **/
// Start watch activity to get HR data if the switch is on
if (switchWatch.isChecked()) {
    startRecordingOnWear();
}
```

We can implement the private method **startRecordingOnWear()**.

```
/** mobile -> NewRecordingFragment.java -> NewRecordingFragment **/
private void startRecordingOnWear() {
    Log.d(TAG, "Entered smartwatch hr reading");
```

```
    Intent intentStartRec = new Intent(getActivity(),
            WearService.class);
    intentStartRec.setAction(WearService.ACTION_SEND
            .STARTACTIVITY.name());
    intentStartRec.putExtra(WearService
            .ACTIVITY_TO_START, BuildConfig
            .W_recordingactivity);
    getActivity().startService(intentStartRec);
}
```

Notice that we are using a key called **recordingactivity**, so remember to add it in the Gradle script of the project.

The starting of a wear activity is already handled in the **WearService** of the mobile module. The only thing we need to add is the new key we use, which refers to **RecordingActivity**. In the **WearService** of the wear module, we add a **case** in the **switch(data)** in the **onMessageReceived(...)** method.

```
/** wear -> WearService.java -> WearService -> onMessageReceived(...) **/
switch (path) {
    case BuildConfig.W_path_start_activity:
        Intent startIntent = null;
        switch (data) {
            ...
            case BuildConfig.W_recordingactivity:
            Log.d(TAG, "Start recording message received");
            startIntent = new Intent(this, RecordingActivity.class);
            break;
        }
...
}
```

Now in **NewRecordingFragment**, in the on click listener callback where we saved the recording on Firebase, we start the new activity which will show the live updates of the exercise we are performing.

```
/** mobile -> NewRecordingFragment.java -> NewRecordingFragment
    -> onCreateView(...) -> onComplete(...) **/
Intent intentStartLive = new Intent(getActivity(), ExerciseLiveActivity.class);
intentStartLive.putExtra(USER_ID, userID);
```

```
intentStartLive.putExtra(RECORDING_ID, recordingKeySaved);
startActivity(intentStartLive);
```

Note that we are sending an **Intent** with two extras: the **userID** to be able to get the data from the logged in user and the **recordingKeySaved**, which is the unique key of the new recording just started. We will save the HR data coming from the watch in this branch of the *Firebase Realtime Database*.

3.3 Access HR sensor

To monitor raw sensor data you need to implement two callback methods that are exposed through the **SensorEventListener** interface: **onAccuracyChanged(...)** and **onSensorChanged()**. The Android system calls these methods whenever the following occurs:

- *A sensor's accuracy changes:* in this case the system invokes the **onAccuracyChanged(...)** method, providing you with a reference to the **Sensor** object that changed and the new accuracy of the sensor. In this lab, we will not use it so it will be left blank.
- *A sensor reports a new value:* in this case the system invokes the **onSensorChanged(...)** method, providing you with a **SensorEvent** object. A **SensorEvent** object contains information about the new sensor data, including: the accuracy of the data, the sensor that generated the data, the timestamp at which the data was generated, and the new data that the sensor recorded.

The following code shows how to use the **onSensorChanged(...)** method to monitor data from the heart rate sensor. Namely, make sure the **RecordingActivity** extends **WearableActivity** and implements **SensorEventListener**. **SensorEventListener** is registered in the **RecordingActivity** (**onCreate(...)** method).

```
/** wear -> RecordingActivity.java -> RecordingActivity -> onCreate(...) **/
SensorManager sensorManager =
        (SensorManager) getSystemService(MainActivity.SENSOR_SERVICE);
Sensor hr_sensor = sensorManager.getDefaultSensor(Sensor.TYPE_HEART_RATE);
sensorManager.registerListener(this, hr_sensor, SensorManager.SENSOR_DELAY_UI);
```

First, we call the **SensorManager** system service to be able to register the listener (which is in the current activity: **this**). When the listener is registered on the heart rate sensor, it listens to events with a rate suitable to the user interface.

```
/** wear -> RecordingActivity.java -> RecordingActivity **/
@Override
```

```
public void onSensorChanged(SensorEvent event) {
    TextView textViewHR = findViewById(R.id.hrSensor);
    if (textViewHR != null)
        textViewHR.setText(String.valueOf(event.values[0]));
}
```

Then we set one of the **TextView** with the heart rate value we listened to. The text is updated whenever the sensor changes its value. Note, that we take the value from the sensor event as **event.values[0]**. For more information about how to get the sensor value for different types of sensors check the *Android Developer* documentation[2].

If you run your application now, the value of your heart rate should be shown in the **TextView** that you use for heart rate measurements and will change whenever the sensor value changes.

3.4 Sending HR from watch to tablet

First we add a key and a path for the HR data in the Gradle script of the project.

```
/** project -> build.gradle -> allprojects{...} -> project.ext{...} **/
constants = [
        ...
        heart_rate_key      : "heart_rate_key",
        heart_rate_path     : "/HEART_RATE_PATH",
        ...
]
```

Then, in the **onSensorChanged(...)** of the **RecordingActivity** after reading the sensor we start an **Intent** by passing the HR data as an extra.

```
/** wear -> RecordingActivity.java --> RecordingActivity
    -> onSensorChanged(...) **/
Intent intent = new Intent(RecordingActivity.this, WearService.class);
intent.setAction(WearService.ACTION_SEND.HEART_RATE.name());
intent.putExtra(WearService.HEART_RATE, heartRate);
startService(intent);
```

Make sure the all the final variables are defined. Then, we have to add a case in the switch of the method **onStartCommand(...)** of the **WearService** in the wear module.

[2] <https://developer.android.com/reference/android/hardware/SensorEvent>

```
/** wear -> WearService.java -> WearService -> onStartCommand(...) **/
case HEART_RATE:
    putDataMapRequest = PutDataMapRequest
            .create(BuildConfig.W_heart_rate_path);
    putDataMapRequest.getDataMap()
            .putInt(BuildConfig.W_heart_rate_key,
                    intent.getIntExtra(HEART_RATE, -1));
    sendPutDataMapRequest(putDataMapRequest);
    break;
```

In the mobile **WearService**, in the **onDataChanged(...)** method, we add a case to handle the HR data.

```
/** mobile -> WearService.java -> WearService -> onDataChanged(...) **/
case BuildConfig.W_heart_rate_path:
    int heartRate = dataMapItem.getDataMap()
            .getInt(BuildConfig.W_heart_rate_key);
    intent = new Intent(ExerciseLiveActivity.ACTION_RECEIVE_HEART_RATE);
    intent.putExtra(ExerciseLiveActivity.HEART_RATE, heartRate);
    LocalBroadcastManager.getInstance(this).sendBroadcast(intent);
    break;
```

3.5 Showing HR in ExerciseLiveActivity

First, we need to get the intent extras in the **ExerciseLiveActivity** in the method **onCreate(...)**, to be able to access the specific branch of the database.

```
/** mobile -> ExerciseLiveActivity.java -> ExerciseLiveActivity
    -> onCreate(...) **/
Intent intentFromRec = getIntent();
String userID = intentFromRec.getStringExtra(MyProfileFragment.USER_ID);
String recID = intentFromRec.getStringExtra(NewRecordingFragment.RECORDING_ID);
```

Then, we get all the information of the new recording from Firebase. Do it as you did it in **MyHistoryFragment** by using the references to the specific user and recording and a method **addListenerForSingleValueEvent(...)**, since we need to get the info once.

In order to get the HR sensor data sent from the watch, in **ExerciseLiveActivity** we implement the **LocalBroadcastManager** with an **IntentFilter** specific to the HR we already defined (**ACTION_RECEIVE_HEART_RATE**, always remember to add these strings). The

BroadcastReceiver for the HR data is registered in the **onResume(...)** and unregistered in the **onPause(...)**.

```
/** mobile -> ExerciseLiveActivity.java -> ExerciceLiveActivity **/
@Override
protected void onResume() {
    super.onResume();
    //Get the HR data back from the watch
    heartRateBroadcastReceiver = new HeartRateBroadcastReceiver();
    LocalBroadcastManager.getInstance(this).registerReceiver
            (heartRateBroadcastReceiver, new IntentFilter
                    (ACTION_RECEIVE_HEART_RATE));
}

/** mobile -> ExerciseLiveActivity.java -> ExerciseLiveActivity **/
@Override
protected void onPause() {
    super.onPause();
    LocalBroadcastManager.getInstance(this).unregisterReceiver
            (heartRateBroadcastReceiver);
}

private int heartRateWatch = 0;

/** mobile -> ExerciseLiveActivity.java -> ExerciseLiveActivity **/
private class HeartRateBroadcastReceiver extends BroadcastReceiver {

    @Override
    public void onReceive(Context context, Intent intent) {
        //Show HR in a TextView
        heartRateWatch = intent.getIntExtra(HEART_RATE, -1);
        TextView hrTextView = findViewById(R.id.exerciseHRwatchLive);
        hrTextView.setText(String.valueOf(heartRateWatch));

    }
}
```

Now, if you run the app, you will see the HR value in both watch and tablet.

4 Show HR in AndroidPlot

In order to improve the look-and-feel of our sport tracker app, we will show the HR data in a live plot. For this we need to add the plot in the layout of **ExerciseLiveActivity** and then draw it every time we receive the HR data. We will also reserve a space for the Google Map to show the live location of our exercise.

4.1 Adding AndroidPlot to layout and configure it

We will use the **AndroidPlot** library, which you can add through the module's *Settings* panel (right click on the module (mobile, in our case), then *Open Module Settings*), in the *Dependencies* tab, click on "+" (*Add Dependency*) to add another library. Search for **com.androidplot:androidplot-core:1.5.7** in the text field provided as shown in Figure 1. Otherwise, it is also possible to edit the Gradle file to manually add the dependency as follows:

```
/** mobile --> build.gradle **/
dependencies {
    implementation 'com.androidplot:androidplot-core:1.5.7'
    [...]
    }
```

We can now add the plot in the layout following this XML code:

```
<!-- mobile -> activity_exercise_live.xml -> YourLayout -->
<com.androidplot.xy.XYPlot
    android:id="@+id/HRplot"
    app:lineLabels="left" />

    <!-- Add another item (such as FrameLayout) to take the
    space which will be used by the map later on -->
```

Notice that we use **com.androidplot.xy.XYPlot** that contains some specific features to the plot like **app:lineLabels**, which shows the graph line labels on the left. Feel free to use the layout you want, here we are constraining it based on the solution of *Lab 5* that relied on a **ConstraintLayout**.

Now we can create a private field of type **XYPlot** in **ExerciseLiveActivity**, which we call **heartRatePlot** and we instantiate it in the **onCreate(...)** as the **View** we added in the xml.

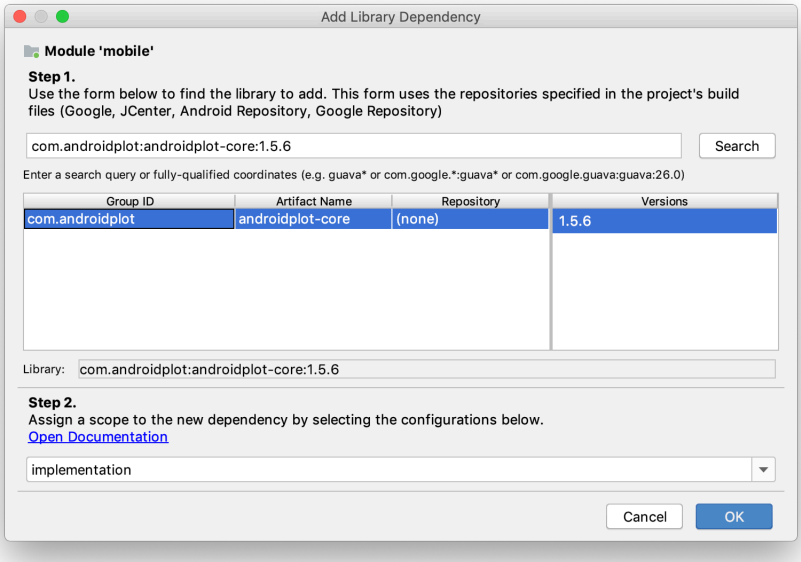

Figure 1: Adding dependency for **AndroidPlot**

```
/** mobile -> ExerciseLiveActivity.java -> ExerciseLiveActivity
-> onCreate(...) **/
heartRatePlot = findViewById(R.id.HRplot);
configurePlot();
```

Then, we configure the plot (colors, styling and axis formatting) in the following private method that we can call after having instantiate the **heartRatePlot**.

```
/** mobile --> ExerciseLiveActivity.java -> ExerciseLiveActivity **/
private void configurePlot() {
    // Get background color from Theme
    TypedValue typedValue = new TypedValue();
    getTheme().resolveAttribute(android.R.attr.windowBackground,
            typedValue, true);
    int backgroundColor = typedValue.data;
    // Set background colors
    heartRatePlot.setPlotMargins(0, 0, 0, 0);
    heartRatePlot.getBorderPaint().setColor(backgroundColor);
    heartRatePlot.getBackgroundPaint().setColor(backgroundColor);
    heartRatePlot.getGraph()
            .getBackgroundPaint().setColor(backgroundColor);
    heartRatePlot.getGraph()
```

```
                .getGridBackgroundPaint().setColor(backgroundColor);
        // Set the grid color
        heartRatePlot.getGraph()
                .getRangeGridLinePaint().setColor(Color.DKGRAY);
        heartRatePlot.getGraph()
                .getDomainGridLinePaint().setColor(Color.DKGRAY);
        // Set the origin axes colors
        heartRatePlot.getGraph()
                .getRangeOriginLinePaint().setColor(Color.DKGRAY);
        heartRatePlot.getGraph()
                .getDomainOriginLinePaint().setColor(Color.DKGRAY);
        // Set the XY axis boundaries and step values
        heartRatePlot.setRangeBoundaries(MIN_HR,
                                         MAX_HR,
                                         BoundaryMode.FIXED);
        heartRatePlot.setDomainBoundaries(0,
                                          NUMBER_OF_POINTS - 1,
                                          BoundaryMode.FIXED);
        heartRatePlot.setRangeStepValue(9); // 9 values 40 60 ... 200
        heartRatePlot.getGraph().getLineLabelStyle(XYGraphWidget.Edge.LEFT)
                .setFormat(new DecimalFormat("#"));
        // This line is to force the Axis to be integer
        heartRatePlot.setRangeLabel(getString(R.string.heart_rate));
}
```

We are using here a **String** value **heart_rate** which does not yet exist, so let's add it in the **res/values/strings.xml** file, giving it the value *Heart Rate (bpm)*. We also are using the integer constants **MIN_HR**, **MAX_HR** and **NUMBER_OF_POINTS** which we need to define in the class. We will give them the respective values 40, 200 and 50. The first constant is the minimal heart rate value to display in the graph, the second is the maximum one, and the third is the number of data points that will be displayed in the graph.

4.2 Create an XY series for the plot

For the sake of having a well structured and clear java code, let's define a class **XYplotSeriesList** where we can define all the necessary objects and functions to plot data in the **heartRatePlot**.

Let's follow these next steps to create the **XYplotSeriesList** class and then use it to initialize the plot:

1. Create a new class named **XYplotSeriesList**.
2. In the class definition, define the following private variables: two **ArrayList** of integer arrays **xList** and **yList**, these are respectively the list of the horizontal and vertical coordinates of the series to-be plotted; an **ArrayList** of list of numbers **xyList** which is a list of the arrays of (x,y) data points of the series. An **ArrayList** of strings **xyTagList** which contains a unique *Tag* for each series to plot. Finally, an **ArrayList** of **LineAndPointFormatter** that we call **xyFormatterList**, which contains the formatters to use to plot each of the series.

```
/** mobile -> XYplotSeriesList.java -> XYplotSeriesList **/
private ArrayList<Integer[]> xList = new ArrayList<>();
private ArrayList<Integer[]> yList = new ArrayList<>();
private ArrayList<List<Number>> xyList = new ArrayList<>();
private ArrayList<String> xyTagList = new ArrayList<>();
private ArrayList<LineAndPointFormatter> xyFormatterList =
        new ArrayList<>();
```

3. Add a method **initializeSeriesAndAddToList()** using the piece of code just below. In this function, we will add an initial **xy** series of length **NUMBER_OF_POINTS** to the plot. We will initialize the heart rate values (y coordinates in **yList**) to a constant value **CONSTANT**, and we will give the **xy** series a unique tag **xyTag**.

```
/** mobile -> XYplotSeriesList.java -> XYplotSeriesList **/
public void initializeSeriesAndAddToList(String xyTag, int CONSTANT,
        int NUMBER_OF_POINTS, LineAndPointFormatter xyFormatter) {
    Integer[] x = new Integer[NUMBER_OF_POINTS];
    Integer[] y = new Integer[NUMBER_OF_POINTS];
    List<Number> xy = new ArrayList<>();
    for (int i = 0; i < y.length; i += 1) {
        x[i] = i;
        y[i] = CONSTANT;
        xy.add(x[i]);
        xy.add(y[i]);
    }
    xList.add(x);
    yList.add(y);
    xyList.add(xy);
    xyTagList.add(xyTag);
    xyFormatterList.add(xyFormatter);
}
```

4. Add a method **getSeriesFromList(...)** by giving a string tag.

```
/** mobile -> XYplotSeriesList.java -> XYplotSeriesList **/
public List<Number> getSeriesFromList(String xyTag) {
    return xyList.get(xyTagList.indexOf(xyTag));
}
```

5. In the **ExerciseLiveActivity** class, define an instance of the **XYplotSeriesList** class called **xyPlotSeriesList**, then initialize it in the **onCreate(...)** method. Finally, use the above function to plot an initial heart rate series. Your code should resemble the one below (note that you need to define a string constant **HR_PLOT_WATCH** with the value *HR from smart watch*, this will be the tag also used to represent the series in the plot legend).

```
/** mobile -> ExerciseLiveActivity.java -> ExerciseLiveActivity
-> onCreate(...) **/
xyPlotSeriesList = new XYplotSeriesList();
LineAndPointFormatter formatterWatch = new LineAndPointFormatter
        (Color.RED, Color.TRANSPARENT, Color.TRANSPARENT, null);
formatterWatch.getLinePaint().setStrokeWidth(8);
xyPlotSeriesList.initializeSeriesAndAddToList(HR_PLOT_WATCH, MIN_HR,
        NUMBER_OF_POINTS, formatterWatch);
XYSeries HRseries = new SimpleXYSeries(
        xyPlotSeriesList.getSeriesFromList(HR_PLOT_WATCH),
        SimpleXYSeries.ArrayFormat.XY_VALS_INTERLEAVED,
        HR_PLOT_WATCH);
heartRatePlot.clear();
heartRatePlot.addSeries(HRseries, formatterWatch);
heartRatePlot.redraw();
```

If you run the app you should have a plot with a red line at the minimum HR value.

4.3 Draw HR data on the plot

Let's now update the plot with the HR data received from the watch. We will follow similar steps to the initialization of the XY series.

1. In the **XYplotSeriesList** class, define a method **getFormatterFromList(...)** the formatter from an existing **XYplotSeriesList** instance.

```
/** mobile -> XYplotSeriesList.java -> XYplotSeriesList **/
public LineAndPointFormatter getFormatterFromList(String xyTag) {
```

```
    return xyFormatterList.get(xyTagList.indexOf(xyTag));
}
```

2. Define a function **updateSeries(...)** that appends a new data point to an existing series with the tag **xyTag** in a **XYplotSeriesList** instance. Since the series has a fixed size, all its previous values need to be shifted first and the new value added to the end of it.

```
/** mobile -> XYplotSeriesList.java -> XYplotSeriesList **/
public void updateSeries(String xyTag, int data) {
    List<Number> xy = xyList.get(xyTagList.indexOf(xyTag));
    Integer[] x = xList.get(xyTagList.indexOf(xyTag));
    Integer[] y = yList.get(xyTagList.indexOf(xyTag));

    xy.clear();
    for (int i = 0; i < y.length - 1; i += 1) {
        y[i] = y[i + 1];
        xy.add(x[i]);
        xy.add(y[i]);
    }
    y[y.length - 1] = data;
    xy.add(x[y.length - 1]);
    xy.add(y[y.length - 1]);

    xyList.set(xyTagList.indexOf(xyTag), xy);
    xList.set(xyTagList.indexOf(xyTag), x);
    yList.set(xyTagList.indexOf(xyTag), y);
}
```

3. Finally, use these three above methods in the **onReceive(...)** method to update the series every time a new HR value data is acquired by the watch. Once the series is updated, it is important to clear the plot before adding it, if not it will be drawn on top of the previous one.

```
/** mobile -> ExerciseLiveActivity.java -> ExerciseLiveActivity
    -> HeartRateBroadcastReceiver -> onReceive(...) **/
xyPlotSeriesList.updateSeries(HR_PLOT_WATCH, heartRateWatch);
XYSeries hrWatchSeries = new SimpleXYSeries(
        xyPlotSeriesList.getSeriesFromList(HR_PLOT_WATCH),
        SimpleXYSeries.ArrayFormat.XY_VALS_INTERLEAVED,
        HR_PLOT_WATCH);
```

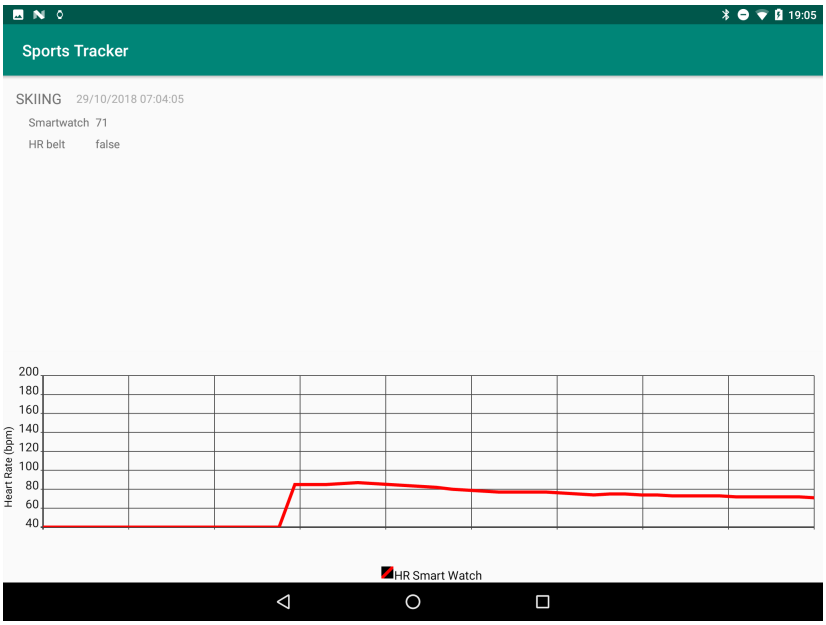

Figure 2: HR data on android plot

```
LineAndPointFormatter formatterPolar =
        xyPlotSeriesList.getFormatterFromList(HR_PLOT_WATCH);
heartRatePlot.clear();
heartRatePlot.addSeries(hrWatchSeries, formatterPolar);
heartRatePlot.redraw();
```

If you run the app you can see that the plot is updated at each value coming from the smart watch HR sensor as shown in Figure 2!

5 Saving HR data on Firebase

In order to save the HR data in Firebase in the recordings branch, we need to save the value in an array and send the data when we stop the recording.

First, we would need a button to stop the recording in the **ExerciseLiveActivity** layout. It can be a normal button, a floating action button or a menu item. We choose a floating action button. Put an **onClick** item and implement the XML callback that we will call **stopRecordingOnWear(...)**.

We need to create an **ArrayList<Integer>** as a private field of the **ExerciseLiveActivity** to add the values of the HR. Then, in the **onReceive(...)** of the **HeartRateBroadcastR eceiver** we add the **heartRateWatch** value to the **hrDataArrayList**.

```
/** mobile -> ExerciseLiveActivity.java -> ExerciseLiveActivity **/
private ArrayList<Integer> hrDataArrayList = new ArrayList<>();

/** mobile -> ExerciseLiveActivity.java -> ExerciseLiveActivity
    -> HeartRateBroadcastReceiver -> onReceive(...) **/
private class HeartRateBroadcastReceiver extends BroadcastReceiver {
    @Override
    public void onReceive(Context context, Intent intent) {
        // Retrieve HR from intent

        //Add HR value to HR ArrayList
        hrDataArrayList.add(heartRateWatch);
    }
}
```

Then, we need to implement the **stopRecordingOnWear(...)** xml callback. Save the HR data in the branch **recordings** in **child(hr_watch)**. Note that you can set the value of the mutable data as the **ArrayList** were we saved the HR data. This will automatically add all the HR values. Show a **Toast** in the **onComplete(...)** to show that the data has been saved successfully and finish the activity (**finish()**).

The last thing to do is to stop the unregister the listener on the watch sensor or, better, stop the **RecordingActivity** in the wear module. You can do this following the steps we did to start the activity.

1. Add a new path to stop the activity called **path_stop_activity**. We will use the key we already created to stop specifically the **RecordingActivity**

```
/** project -> build.gradle -> allprojects{...} -> project.ext{...} **/
constants = [
        ...
        path_stop_activity : "/STOP_ACTIVITY",
        ...
]
```

2. In the **onComplete(...)** of the **ValueEventListener** in the xml callback we previously implemented, before finishing the activity, we will start the mobile **WearService** with an **Intent** action to stop the activity.

```
/** mobile -> ExerciseLiveActivity.java -> ExerciseLiveActivity
    -> stopRecordingOnWear(...) **/
Intent intentStopRec = new Intent(ExerciseLiveActivity.this,
```

```
                        WearService.class);
intentStopRec.setAction(WearService.ACTION_SEND
        .STOPACTIVITY.name());
intentStopRec.putExtra(WearService
        .ACTIVITY_TO_STOP, BuildConfig
        .W_recordingactivity);
startService(intentStopRec);
```

3. Handle the stop activity in the mobile **WearService** in the **onStartCommand(...)** with a case **STOPACTIVITY**. Send a message with the string extra we sent (the activity we want to stop).

```
/** mobile -> WearService.java -> WearService -> onStartCommand(...) **/
case STOPACTIVITY:
    String activityStop = intent.getStringExtra(ACTIVITY_TO_STOP);
    sendMessage(activityStop, BuildConfig.W_path_stop_activity);
    break;
```

4. In the wear module **WearService**, in the **onMessageReceived(...)** handle the case with path **path_stop_activity** and the data with key **recordingactivity** from the **BuildConfig**. Finally, send a broadcast with an intent.

```
/** wear -> WearService.java -> WearService -> onMessageReceived(...) **/
case BuildConfig.W_path_stop_activity:
    switch (data) {
        case BuildConfig.W_recordingactivity:
            Intent intentStop = new Intent();
            intentStop.setAction(RecordingActivity.STOP_ACTIVITY);
            LocalBroadcastManager.getInstance(WearService.this)
                    .sendBroadcast(intentStop);
            break;
    }
    break;
```

5. Last thing to do is to implement the receiver with an **IntentFilter** in the **onCrea**⌋**te(...)** of the **RecordingActivity** in the wear app. In the **onReceive()** we need unregister the **SensorEventListener** and finish the activity.

```
/** wear -> RecordingActivity.java -> RecordingActivity -> onCreate(...) **/
LocalBroadcastManager.getInstance(this).registerReceiver(new BroadcastReceiver() {
    @Override
```

```
    public void onReceive(Context context, Intent intent) {
        sensorManager.unregisterListener(RecordingActivity.this);
        finish();
    }
}, new IntentFilter(STOP_ACTIVITY));
```

If you check your Firebase Console, you will have the HR data saved in a branch of the last recording and the wear app will no longer get HR data from the sensor!

6 Reading GPS location data

Additionally to the heart rate data, we want the application to also record the location during an exercise session. We will hence implement the following functions:

- read the location sensor data from the wear application
- display the live location in the watch
- send the location from the watch to the tablet
- display the live location in **`ExerciseLiveActivity`** of the mobile application using a **`GoogleMap`**

Note: *Before we proceed to implement the above tasks, we need to connect the watch to the Internet to perform online location lookups.*

Because 'epfl' and 'eduroam' are WPA2 Entreprise networks, they are not accessible from the built-in WiFi settings of the smartwatch. However, it is possible to programmatically configure them. We provide you with an app we created that connects to the both networks. If you completed Lab 1, you have already compiled and ran this 'Connect2Wifi' app.

6.1 Access location sensor

To acquire the user location in your Android application, we can utilize either the GPS of your device or the Android's Network Location. The GPS data is the most accurate, however it only works outdoors, quickly consumes battery power, and is slower to return its data. Android Network Location Provider determines user location using cell tower and WiFi signals, thus providing location information both indoors and outdoors. It responds faster, and uses less battery power.

To obtain the user location in your sport tracking application, you can use either the **Network Location Provider**, and test the application in the classroom. Or you can use the **GPS**, and test our application outdoors. However, Google provides another API that op-

timizes the time to get the first result, the accuracy as well as the battery consumption, by performing sensor fusion. It is the **FusedLocationProvider**. Since **Android Wear** has strong constraints for battery life, we will use the **FusedLocationProvider**. We recommend you to try also the Network Provider on the tablet which can be implemented by following similar steps as the HR sensor, since the **LocationManager** is a system service[3]. For more information, check section 6.2.

Let's get the location through **FusedLocationProvider**. The first thing to do is to add a new dependency in the Gradle file to be able to get use this Google Play Services API.

```
/** wear -> build.gradle -> dependencies{...} **/
implementation 'com.google.android.gms:play-services-location:17.1.0'
```

Then, as you have done for the HR sensors, you first need to allow the wear application to access approximate and precise location as well as the Internet (in case you are using the network provider), by adding the corresponding permissions in the **AndroidManifest.xml** of the wear module:

```
<!-- wear -> AndroidManifest.xml -->
<uses-permission android:name="android.permission.ACCESS_COARSE_LOCATION"/>
<uses-permission android:name="android.permission.ACCESS_FINE_LOCATION"/>
```

Then, you need to ask for the permissions to use the location data and internet in the **onCreate(...)** method of the **RecordingActivity** (as you did for the HR sensor).

After requesting permission to the user, we need to call the **FusedLocationProviderClient** and implement the **LocationCallback**. The **LocationCallback** is used to override the method **onLocationResult(...)**, which is triggered everytime a new location is available.

```
/** wear -> RecordingActivity.java -> RecordingActivity -> onCreate(...) **/
fusedLocationClient = new FusedLocationProviderClient(this);
// Location callback
locationCallback = new LocationCallback() {
    @Override
    public void onLocationResult(LocationResult locationResult) {
        if (locationResult == null) {
            return;
        }
        for (Location location : locationResult.getLocations()) {
            // TODO: Change TextView of the location
```

[3] <https://developer.android.com/reference/android/location/LocationManager>

```
            // HINT: to get latitude and longitude use
            //       location.getLatitude()/getLongitude()
        }
    }
};
```

In order to have updates about the location and trigger the **onLocationResult(...)**, we need to *request* the location updates. We will create two methods to start and stop the location updates, implemented as follows.

```
/** wear -> RecordingActivity.java -> RecordingActivity **/
private void startLocationUpdates() {
    LocationRequest locationRequest = new LocationRequest()
            .setInterval(5)
            .setPriority(LocationRequest.PRIORITY_HIGH_ACCURACY);
    fusedLocationClient.requestLocationUpdates(locationRequest,
            locationCallback,
            Looper.getMainLooper());
}

private void stopLocationUpdates() {
    fusedLocationClient.removeLocationUpdates(locationCallback);
}
```

Then, in the **onResume()** and **onPause()** methods we call the new **startLocationUpdates()** and **stopLocationUpdates()** just created to save battery. Stop the location updates when **RecordingActivity** is paused and start it over when it is resumed.

If you start the wear application, you should see the location updated on the corresponding **TextView** (especially if you move!).

6.2 Optional but recommended: Network/GPS provider through LocationManager

You can also use the system service **LocationManager** to get location updates. The permission management is the same for the **FusedLocationProvider** and **LocationManager** system service. What changes is the way the location is retrieved. Let's implement this for the mobile module.

To be able to retrieve the location data, you need your **ExerciseLiveActivity** to implement the **LocationListener**. By doing so, you will need to override several callback

methods (**onProviderEnabled(...)**, **onProviderDisabled(...)**, **onLocationChanged(...)** and **onStatusChanged(...)**). Android Studio will allow you to do it with a simple ALT+RETURN or ALT+ENTER, then **Implement Methods**.

```
/** mobile -> ExerciseLiveActivity.java **/
public class ExerciseLiveActivity extends AppCompatActivity
        implements LocationListener {
...
}
```

Now in the **onCreate(...)** method, you should acquire a reference to the system **LocationManager**, after which you indicate that you want to receive location updates from it by calling **requestLocationUpdates(...)**.

The first parameter in **requestLocationUpdates(...)** is the type of location provider to use. The second parameter is the minimum time interval between notifications and the third one is the minimum change in distance between notifications. Setting both the second and third parameters to zero allows to receive location notifications as frequently as possible. The last parameter is the **LocationListener** (**this**, since we are implementing the listener), which receives callbacks for location updates.

```
/** mobile -> ExerciseLiveActivity.java -> ExerciseLiveActivity
    -> onCreate(...) **/
LocationManager locationManager =
        (LocationManager)this.getSystemService(Context.LOCATION_SERVICE);
if (locationManager != null) {
    locationManager.requestLocationUpdates(
    LocationManager.PROVIDER_NAME, 0, 0, this);
}
```

To get the location data from the LocationListener, you only need to modify the method **onLocationChanged(...)**. You need to get the latitude and longitude of your location and update the corresponding **TextView** in the layout, you can do it as follows:

```
/** mobile -> ExerciseLiveActivity.java -> ExerciseLiveActivity
    -> onLocationChanged(...) **/
longitude = location.getLongitude();
latitude = location.getLatitude();
TextView textViewGPS = findViewById(R.id.Location);
if (textViewGPS != null) textViewGPS.setText("Lat: " +
        latitude + "\nLon: " + longitude);
```

Latitude and longitude can be set as default to the Lausanne coordinates:

```
/** mobile -> ExerciseLiveActivity.java -> ExerciseLiveActivity **/
// Lausanne
private double latitude = 46.5197;
private double longitude = 6.6323;
```

7 Send and display the location data on the tablet

As you did for the HR, try to send the location data (latitude and longitude) to the tablet and display it in a **TextView** in **ExerciseLiveActivity**. Remember to use the **WearService** following the steps explained in *Lab3* or in the *Wear Communication Example*. HINT: follow this flow **RecordingActivity** (wear module) –> **WearService** (wear module) –> **WearService** (mobile module) –> **ExerciseLiveActivity** (mobile module).

8 Optional: display the live location on GoogleMaps

In order to use **GoogleMaps**, we can follow the example of *Android Developers*[4] and then incorporate it in our sport tracker app. To do so we first need to add a **MapsActivity** to the mobile module. Right-click on the mobile module, then click **New** → **Activity** → **Gallery** → **GoogleMapsActivity**.

Now, you have an **Activity** called **MapsActivity**, a layout called **activity_maps.xml** and a **google_maps_api.xml** where you should put a Google API key. This key is necessary to use the Google Maps Android API. The instructions how to create a Google API key is available online[5]. You can add the key to a Google Console project you already have or create a new one. Please note that you have to enable the *Maps SDK for Android* API that you can find in the Google Cloud Platform where you added the API key (you can find it by clicking the button + *Enable API and Services* in the *Dashboard* tab. Once enabled, you have to restrict your API key to the *Maps SDK for Android* API following the instructions on the Google Cloud Platform. Follow these instructions if you are having problems:

1. Open your console[6]
2. Select your project and then Credentials from left
3. Click on your API key
4. Click on *Restrict Key* under API Restrictions

[4] <https://developers.google.com/maps/documentation/android-sdk/start>
[5] <https://developers.google.com/maps/documentation/android-api/signup>
[6] <https://console.developers.google.com/>

5. In the dropdown menu type ”Maps SDK for Android” and add it to your API Restrictions list
6. ”Save”

Since we want to display the map in the **ExerciceLiveActivity** we will implement some of the functionalities of **MapsActivity** there. We start by implementing the **OnMapReadyCallback**, then we create a private **GoogleMaps** instance in the activity. Finally, we need to obtain the reference to the map Fragment in in the layout (which we will add in the next steps):

```
/** mobile -> ExerciseLiveActivity.java -> ExerciseLiveActivity **/
private GoogleMap mMap;

/** mobile -> ExerciseLiveActivity.java -> ExerciseLiveActivity
-> onCreate(...) **/
// Obtain the SupportMapFragment and get notified
// when the map is ready to be used.
SupportMapFragment mapFragment = (SupportMapFragment)
        getSupportFragmentManager()
        .findFragmentById(R.id.GoogleMap);
mapFragment.getMapAsync(this);
```

By implementing the **OnMapReadyCallback**, you will be asked by Android Studio to override some methods (you can do it with the usual code inspector). You also need to add the **Fragment** from **activity_maps.xml** in **activity_exercice_live.xml**, within a layout of your choice depending on the enclosing layout:

```
<!-- mobile -> activity_exercice_live.xml -> YourLayout -->
<fragment xmlns:android="http://schemas.android.com/apk/res/android"
    xmlns:map="http://schemas.android.com/apk/res-auto"
    xmlns:tools="http://schemas.android.com/tools"
    android:id="@+id/GoogleMap"
    android:name="com.google.android.gms.maps.SupportMapFragment"
    android:layout_width="match_parent"
    android:layout_height="match_parent"/>
```

To display the initial position in the map, you will need to override the **onMapReady(...)** method:

```
/** mobile -> ExerciseLiveActivity.java -> ExerciseLiveActivity **/
private Marker mapMarker;
```

```
/** mobile -> ExerciseLiveActivity.java -> ExerciseLiveActivity
    -> onMapReady(GoogleMap googleMap) **/
mMap = googleMap;

// Add a marker in the default location and move the camera
LatLng currentLocation = new LatLng(latitudeWatch, longitudeWatch);

mMap.moveCamera(CameraUpdateFactory.newLatLngZoom(currentLocation, 15));
if(mapMarker!=null){
    mapMarker.remove();
}
mapMarker = mMap.addMarker(new MarkerOptions().position(currentLocation)
    .title("Current Location"));

TextView longitudeTextView = findViewById(R.id.longitudeWatchValue);
longitudeTextView.setText(String.valueOf(longitudeWatch));
TextView latitudeTextView = findViewById(R.id.latitudeWatchValue);
latitudeTextView.setText(String.valueOf(latitudeWatch));
```

You can also update the location in the map whenever a new location is received, in the location **BroadcastReceiver** by adding the following:

```
/** mobile -> ExerciseLiveActivity.java -> ExerciseLiveActivity
    -> LocationBroadcastReceiver (if you implemented) **/
//Update GoogleMaps location
LatLng currentLocation = new LatLng(latitudeWatch, longitudeWatch);
mMap.moveCamera(CameraUpdateFactory.newLatLngZoom(currentLocation,15));
if(mapMarker!=null){
    mapMarker.remove();
}
mapMarker = mMap.addMarker(new MarkerOptions().position(currentLocation)
        .title("Current Location"));
```

The **newLatLngZoom(..)** method allows to set a zoom level (we chose 10 in this example). The following list gives you an idea of what level of detail each level of zoom shows:

- 1: World
- 5: Landmass/continent
- 10: City
- 15: Streets

- 20: Buildings

If you run the app now, the **`ExerciceLiceActivity`** should look like Figure 3!

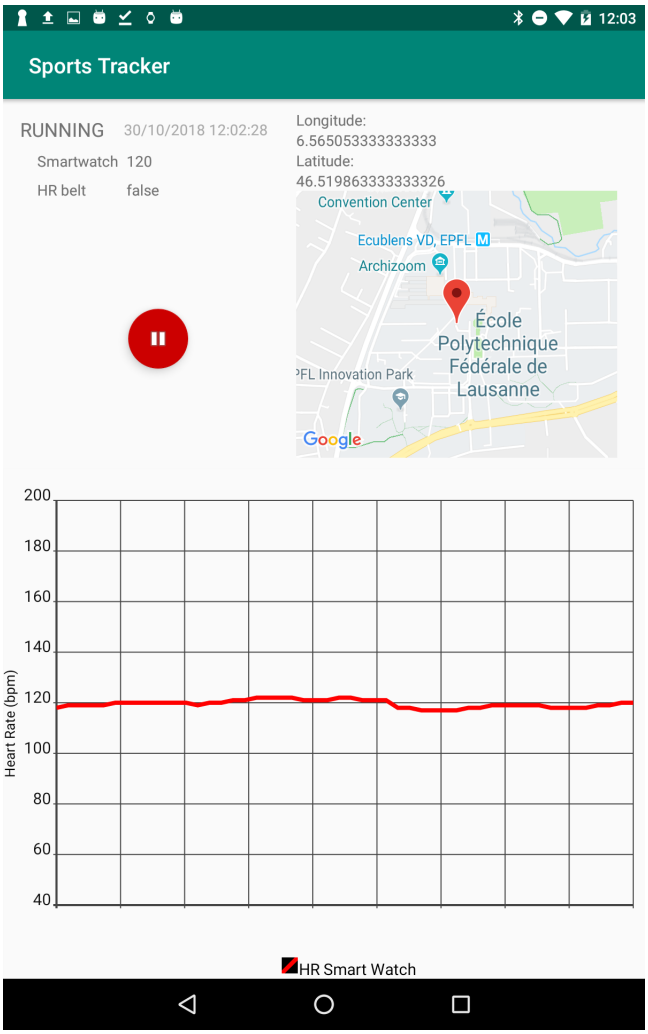

Figure 3: **`ExerciceLiveActivity`** layout

9 Bonus step

Try to save the location data to Firebase (similarly as for HR data). You can then retrieve this data and display it in the **`MyHistoryFragment`** of your **`MainActivity`**.