# Android app

## Introduction

In order to communicate with the glasses, we decided to make an android app (most of our group members have Android phones). Because of the simplicity of the language, and the widespread support for it, we decided to code our Android app in Kotlin. We also used Android Developer Studio as our IDE due to its simplicity and advanced developer features. Our app would manage the Bluetooth connection and communication, as well as notifications and maps handling. In this section of the report, we will explore all the android-related tasks and features that we worked on.

Android apps are divided into Activities, which represent a single “thing” a user can do. In our app, we will have 2 principal activities:

* MainActivity: the purpose of this activity is to be able to navigate around the app and support the Google Maps integration
* BluetoothActivity: the purpose of this activity is to be able to handle all possible communication between the Smart Glasses and our phone. This activity is also responsible for handling notifications.

Each activity needs a callback function called onCreate. This function serves as a pseudo-main function for Activity, and is responsible for initializing them. All the initialization code must be in this method, or else it risks not being executed upon runtime initialization of the Activity.

## Permission Handling

Before we start communicating via Bluetooth or reading the phone’s notifications, we need to make sure that the phone and its user have granted us the necessary permissions (otherwise we would not have access to those respective features/data. The required permissions for our app were (given by their permission string code:

* “android.permission.INTERNET”: this permission allows us to use the internet capabilities of the phone.
* “android.permission.BLUETOOTH”: this permission allows us to look at the basic Bluetooth telemetry of the phone (e.g. connected devices).
* “android.permission.BLUETOOTH\_ADMIN”: this permission allows us to manage the existing Bluetooth connections of the phone.
* “android.permission.BLUETOOTH\_SCAN”: this permission allows us to search for external devices that are advertising (i.e. “announcing” that they are ready to connect).
* “android.permission.BLUETOOTH\_CONNECT”: this permission allows us to use the phone’s Bluetooth to connect to external devices.
* “android.permission.ACCESS\_COARSE\_LOCATION”: this permission allows us to use the phone’s GPS (or GLONASS) capabilities to have a rough idea of the phone’s location.
* “android.permission.ACCESS\_FINE\_LOCATION”: this permission allows us to use the phone’s GPS (or GLONASS) capabilities to have the phone’s precise location.

These permissions go into the Android project’s AndroidManifest.xml file, with a tag that looks like the following:

<uses-permission android:name=[PERMISSION\_STRING\_CODE]>

Where [PERMISSION\_STRING\_CODE] is, as the name suggests, the Permission String code of the permission that we wish the device to grant our app.

This isn’t enough however, as we still need to check upon launch that the user has indeed granted all the necessary permissions to our app, as the app’s functionality depends on it. For this, in our BluetoothActivity, we use a callback function called onRequestPermissionResult. This function is called whenever the app requests permissions from the phone (usually upon the app’s boot) and handles whatever action needs to be taken after the user of the phone allows/denies it any permissions. In our case, we log (i.e. display on “LogCat” the android terminal, if the permissions were correctly granted or denied). We also added buttons to our BluetoothActivity, which, upon click, make the app check whether permissions were granted using a method called checkPermission, which either prints on the LogCat that the relevant permission was granted, or prompts the user to give the permission using ActivityCompat.requestPermissions method (which uses the standard permissions granting UI that you see when an app asks for permissions). In order to know how buttons work in Android and Kotlin, we recommend you look at this tutorial: <https://developer.android.com/guide/topics/ui/controls/button>.

## Searching and connecting

Now that the relevant permissions for the app have been acquired, we can start with the Bluetooth segment! The first thing we need to do is define a BluetoothAdapter object, which we will use to manage our Bluetooth connection(s). We initialize it as:

private val bluetoothAdapter: BluetoothAdapter by lazy{

val bluetoothManager getSystemService(Context.BLUETOOTH\_SERVICE) as BluetoothManager

bluetoothManager.adapter

}

We also need an object to scan devices, we initialize it as:

private val bleScanner by lazy {

bluetoothAdapter.bluetoothLeScanner

}

We also define the scan settings of the code (i.e. a property which decides how our phone is going to scan for external devices, as well as which types of devices it will scan for). Please refer to this source for more details: <https://developer.android.com/reference/android/bluetooth/le/ScanSettings>.

Another aspect we need to handle is our treatment of individual scan results. For storing these results, we use a list of ScanResult (documentation: <https://developer.android.com/reference/android/bluetooth/le/ScanResult>), while the displaying of the results is done using Android’s Recyclerview (documentation: <https://developer.android.com/jetpack/androidx/releases/recyclerview>). The actual managing of the scan results being tapped is done by a ScanResultAdapter, whose aim is to connect to whichever BLE device we tapped on from the results. For this, we take the ScanResult that was tapped and apply the connectGatt method on it, which sets up the Bluetooth connection between the tapped device and our phone. A good tutorial to understand this part can be found here: <https://developer.android.com/guide/topics/connectivity/bluetooth/connect-gatt-server>.

If you read the tutorial, you will realize that we need to define an object of type BluetoothGattCallback. The function of this object will be to provide callback methods for all Bluetooth-related things post-connection. This is why we encourage you to read the documentation for this class: <https://developer.android.com/reference/android/bluetooth/BluetoothGattCallback>. We will also need an object of type BluetoothGatt, which is acts as the instance of our BLE connection with the smart glasses. In effect, our BluetoothGattCallback object is just an object which is used to implement BluetoothGatt callbacks. Hence you should also look at its documentation: <https://developer.android.com/reference/android/bluetooth/BluetoothGatt>. Once you instantiate your BluetoothGatt object upon connection and override the needed BluetoothGattCallback methods with the actions we want the connection to perform, the Bluetooth setup and connection stage is complete! You are now free to set up conventions to send and receive data from your Smart Glasses (or any other BLE device) as you please!

## Reacting to data sent by the Smart Glasses (and sending data)

In order to be able to receive data from the Smart Glasses, we have to “subscribe” to notifications from the Smart Glass. How do we do this? Our Smart Glass runs multiple services (of type BluetoothGattService), which contains fields of bytes which are of type BluetoothGattCharacteristic. The function of these fields (which we will now call “characteristics” as that is their formal name) work in a way such that if their value is changed, they notify all of their subscribers of this change, and send them the new value. These fields also have UUIDs which help identify them. This is how we will communicate with our Smart Glasses. In our program, we have 1 service and 3 characteristics (Note: these are not their official names, as the only “official” identifier of a service/characteristic is its UUID):

* NOTIFICATION\_SERVICE: Manages the characteristics which are used to communicate between the Smart Glasses and our Android app. Contains 3 characteristics as described below:
  + NOTIFICATION\_BUFFER\_ATTR: Handles the sending of notification data from the phone to the smart glasses. Works in a way such that when the phone receives a notification, it sends it to a buffer (which functions in a first-in-first-out manner). The idea is that when the Smart Glasses are ready to process the next notification, they write 0 to the NOTIFICATION\_BUFFER\_ATTR. When the app is notified that this characteristic has changed, it sends the head of the notification buffer through the NOTIFICATION\_BUFFER\_ATTR to the Smart Glasses (the Smart Glasses are also subscribed to changes in this attribute). If this is the first notification that we are sending for processing, then it is send instantly and doesn’t have to wait for a ready signal from the Smart Glasses.
  + TIME: Is used to send the current date and time to the Smart Glasses upon connection, i.e. when the Bluetooth connection between the two devices is successfully established, the app writes the current date and time to this characteristic, which the Smart Glasses uses to initialize its own time state. This characteristic is not used again after connection.
  + MAPS: Handles the sending of Google Maps directions data from the phone to the smart glasses. When the phone received a notification, it sends it to a ByteArray field, and stores the most recent unique direction given by Google Maps. When the smart glasses are ready to process the next instruction, we take the ByteArray and send it via the MAPS characteristic. The protocol works in a similar way to the NOTIFICATION\_BUFFER\_ATTR protocol, with the only real difference being the Characteristics, and the fact that MAPS uses a most-recent policy for sending maps directions instead of sending them through a buffer all in order like NOTIFICATION\_BUFFER\_ATTR.