# 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)

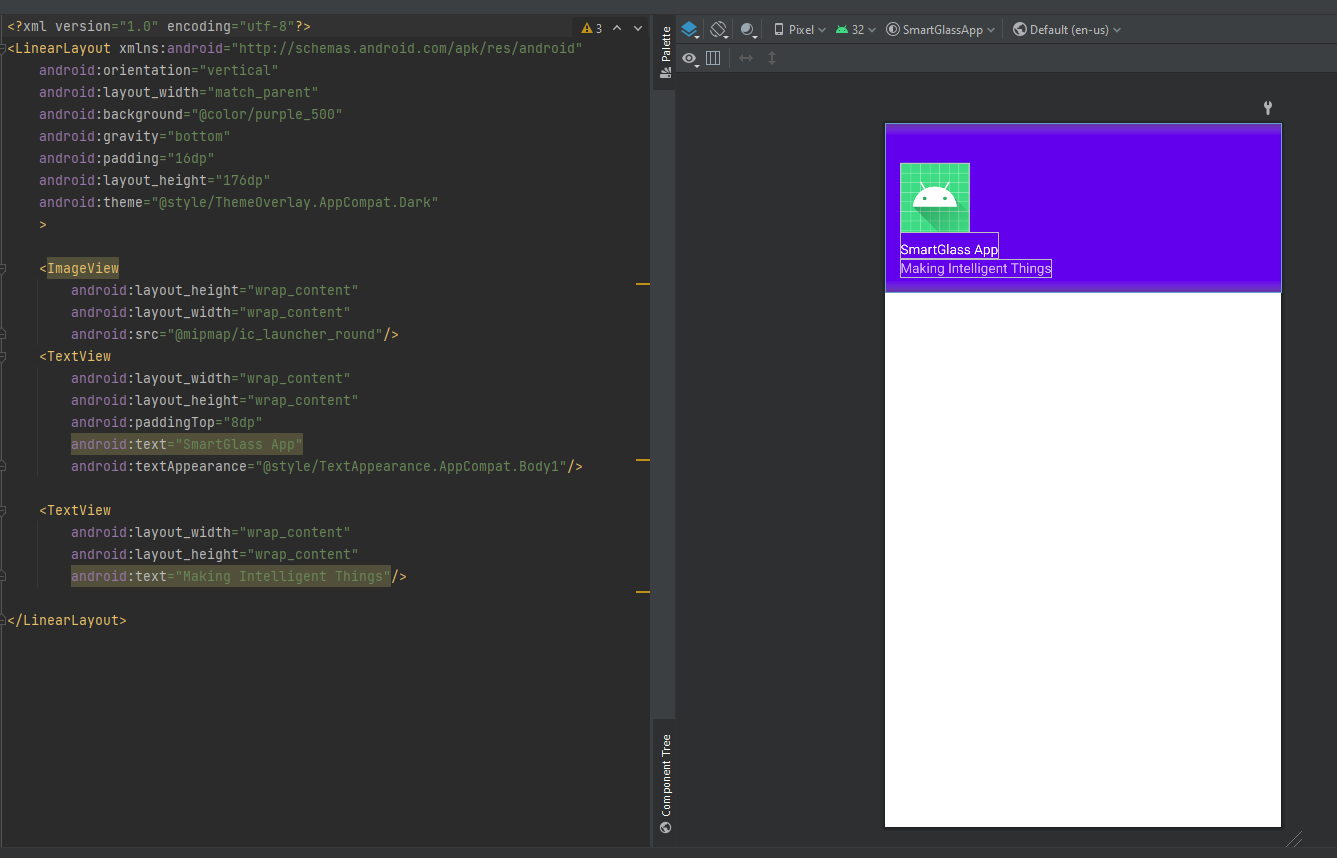
In order to be able to receive data from the Smart Glasses, we have to “subscribe” to notifications from the Smart Glass. How do we do this? Our Smart Glass runs multiple services (of type BluetoothGattService), which contains fields of bytes which are of type BluetoothGattCharacteristic. The function 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.

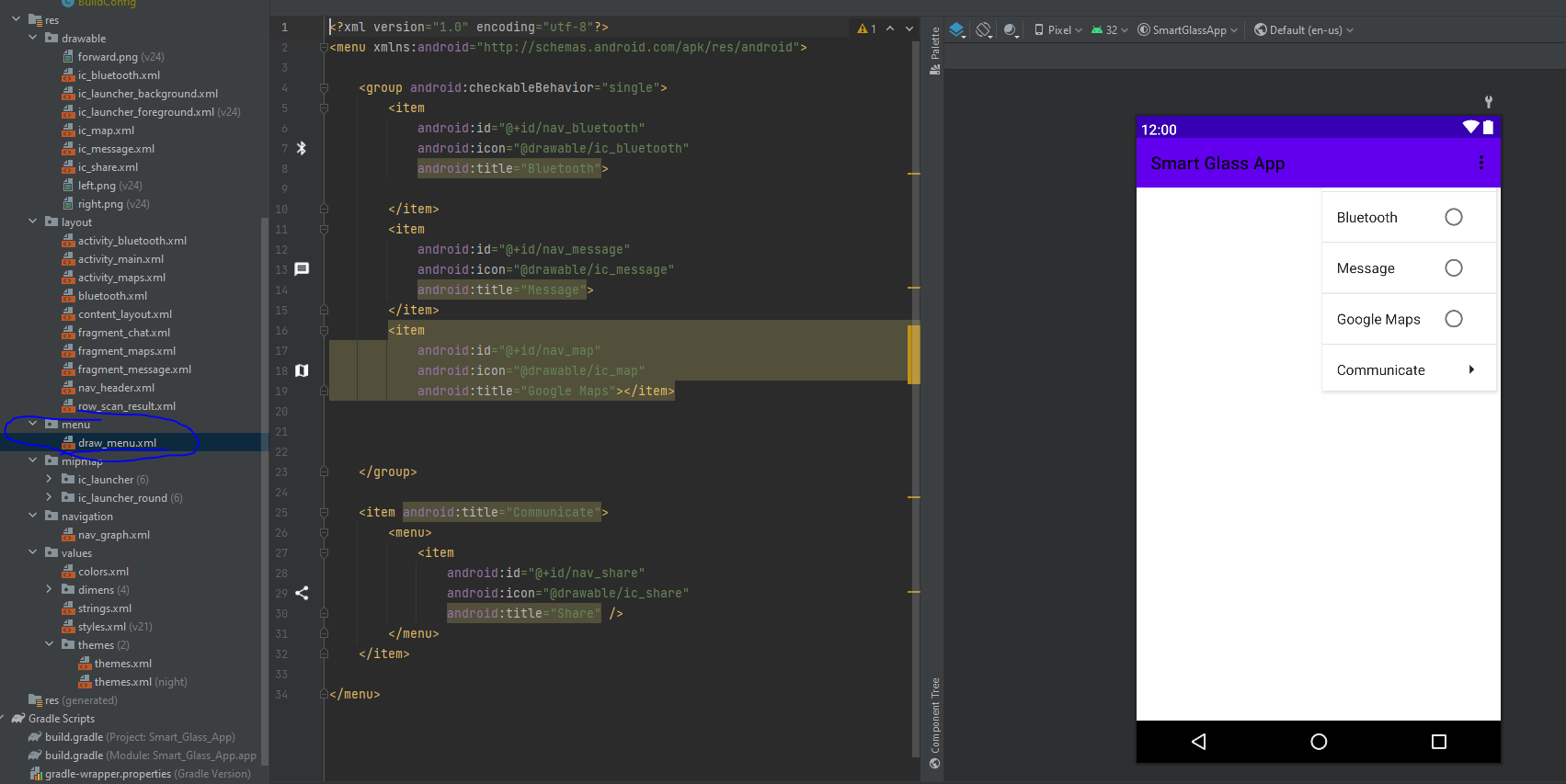
## Implementing a Navigation View that can switch between Activities

One of the important aspects for the overall functionality of the app would be the ability for the app to switch between different pages (fragments and activities) while also being able to run separate services that can be used by the smart glass, such as Bluetooth and google maps. In order to do this, we create a navigation View according to the following 3 part series of tutorials which shows how to implement the navigation view visually, as well as how to make it so that it can contain different activities while still overlaying the navigation menu for navigating between pages.

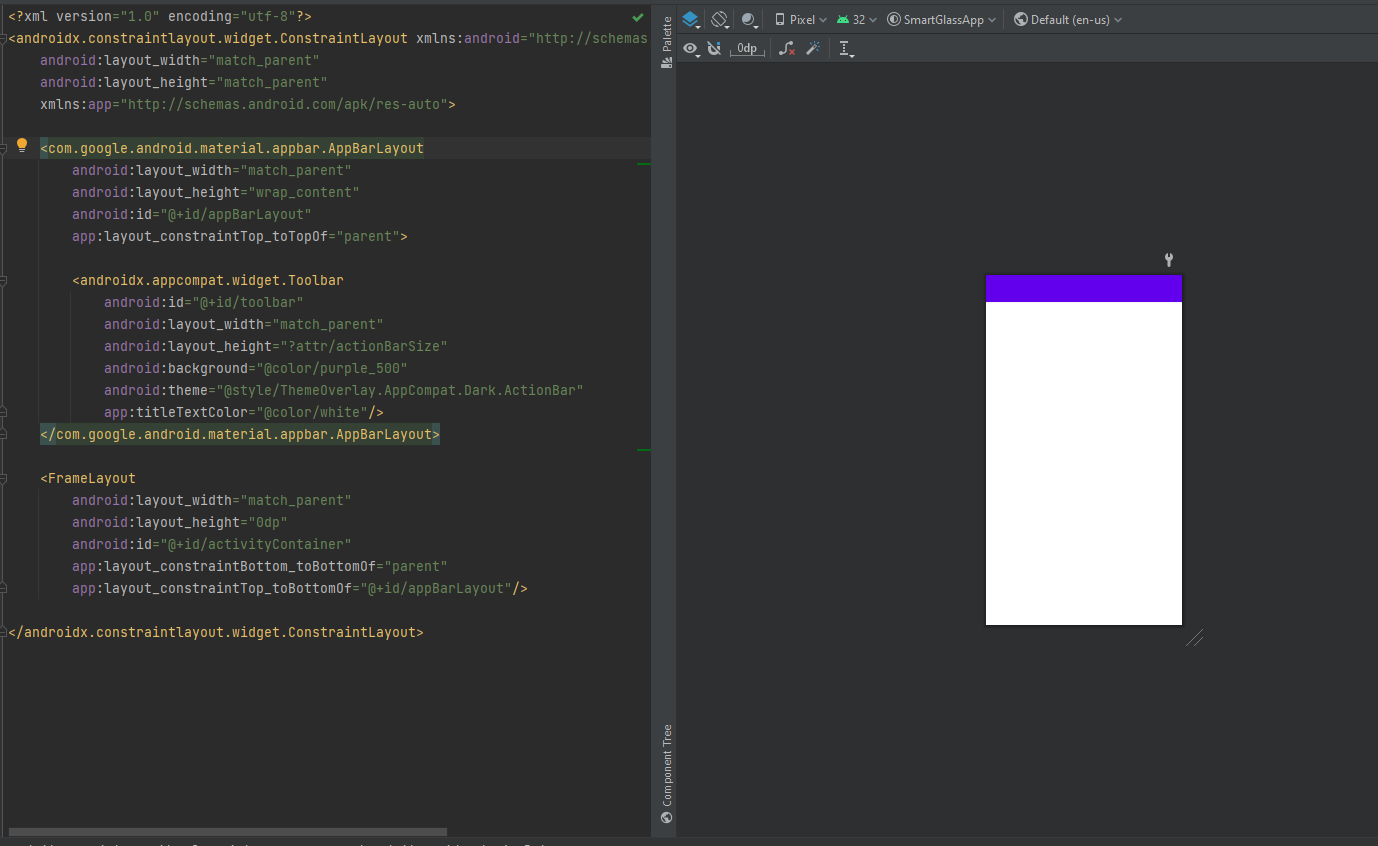
The first part of making this view would be to implement the header of such a navigation view, which we can make into a separate XML file called nav\_header for example. In this file, we can simply make it a linear layout in which simply set a rectangle that sticks to the top with a thickness of 176 dp, in which we then implement an imageView and two textView’s that can simply be used as a descriptor of the app or anything else really. This file, along with its code, will look something along the lines of this.



Next, we want to implement the list of potential items to select from for the menu of the navigation view, which will also be in a separate XML filed called, for example, draw\_menu.xml which will itself be placed in a separate folder from the usual layout folder where all of the other XML files would be often placed. This folder would be often called menu or draw-menu, and is simply a default convention by which android specifies a package where different menu’s can be placed. The code in the file itself is simply using an XML attribute called menu to create the list of items itself. The code

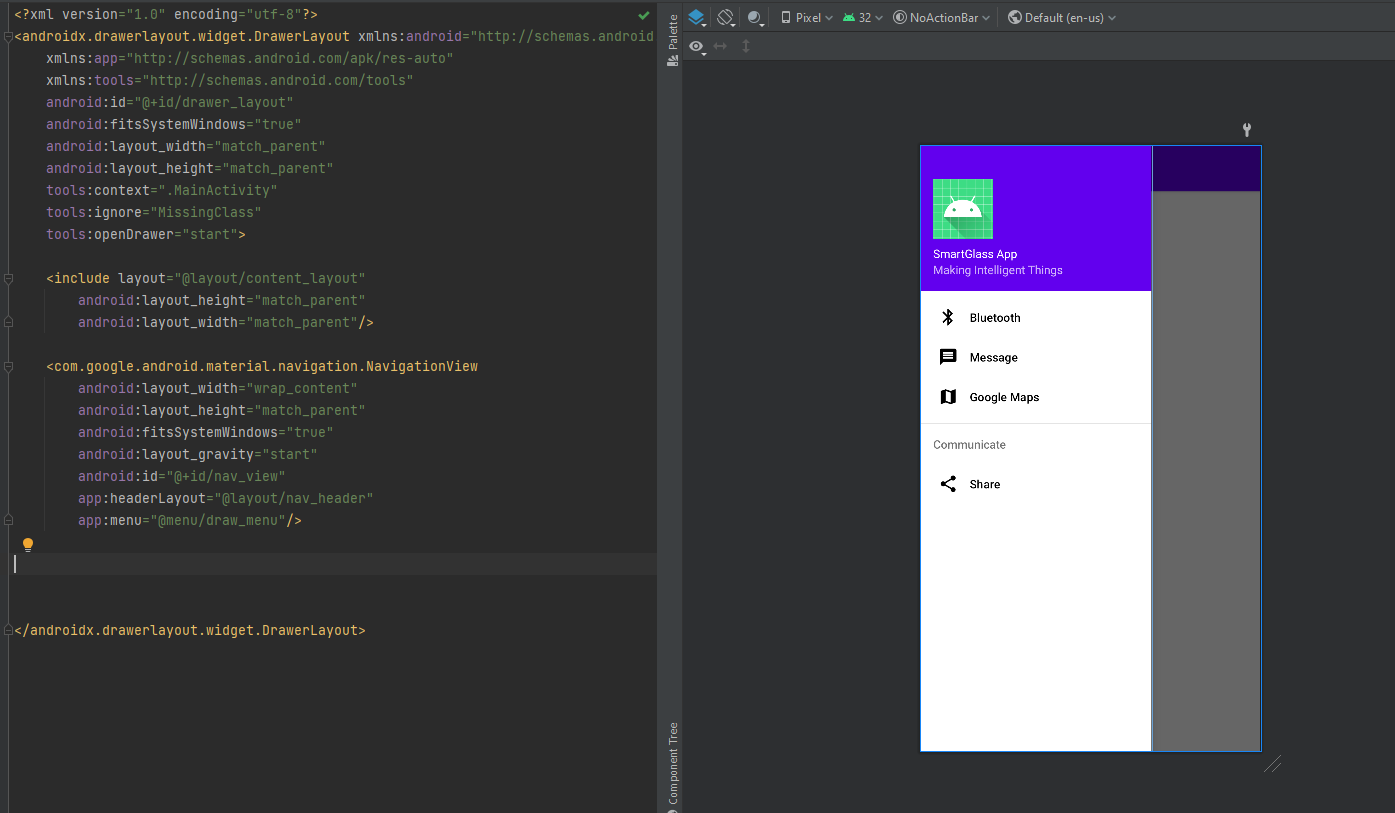


Next, we then want to create a kind of formatting XML that will simply be used to tell us how to organize the navigation view alongside the activity that it will later have to contain. This XML file will be called content layout whose sole purpose is to just allow the containment of another activity within another over-arching navigation view activity. We do this by creating the overall constraint Layout, in which we place two more layouts that will be containing two separate things. The first will be the appBarLayout for containing the toolbar that will let us open the navigation view in order to be able to select from a list of elements. Then, the Frame Layout will be placed below it in order to contain the underlying activity that we want to run upon selecting a certain item from the navigation view. For the time being, this wont really look like anything in the page simulator on the side (as shown below), but it will serve as an important factor for allowing the running of multiple activities and even persistence of their services even when switching to other activities.



Lastly, we then have to combine all 3 of these XML files in the main\_activity.xml file which is usually automatically generated when creating a new blank android studio project. In this file we will combine these 3 things to create the navigation view. We do this by first using a special layout called a drawer layout which itself will contain a content layout and a navigation view. In order to then use the aforementioned files in this file, we specify in the navigation view component that the app:headerLayout is the nav\_header.xml file and the app:menu is draw\_menu.xml. as shown in the image below.

We then simply add another tag within the drawer layout called <include layout> which will then link to our content layout which will then contain the app bar and frame layout for containing the activity. This will then result in the activity main looking something along the lines of the following image.



There are several tutorials that were used to create this navigation view, however one that is similar and can help significantly would be this 3-part series describing how to make something similar, however one of the main differences would be the content layout which does not allow for the effective switching to activities, as without the content layout we would only switch to other fragments which have significantly limited functionality compared to an activity.

<https://www.youtube.com/watch?v=fGcMLu1GJEc&t=13s>

<https://www.youtube.com/watch?v=zYVEMCiDcmY>

Lastly, before moving onto the next part, we need to actually implement something from the more technical kotlin side of things. Thus, in the respective kotlin file of Main\_activity and whichever other activity will be opened up from the navigation view menu, we will have to implement a hierarchy between the main activity (the navigation view) and any other activity contained within it, such that the main Activity is the parent of all of these activities. We will use Bluetooth activity as a general example.

So firstly, in the class declaration we have to specify that said activity effectively inherits from MainActivity, instead of the usual Activity class.

class BluetoothActivity : MainActivity() {

Also, we will then have to make a specific variable called a binding which will be used for the creation of said activity. Effectively, this binding will be in general called, for any specific activity, “Activity[NAME]Binding”, on which we will then call the following functions.

binding = ActivityBluetoothBinding.inflate(*layoutInflater*)  
allocateActivityTitle("Bluetooth")  
setContentView(binding.*root*)

Which, in turn will allow the overlaying of the navigation view on top of whichever activity we want to use and effectively the allowing of persistence in state such that, for example, the Bluetooth service implemented in the Bluetooth activity continues to run even when switching to another activity within the navigation view.

## Implementing Google maps SDK for android as well as Autocomplete API

Next, we need to implement a feature whereby we can open a google maps fragment or activity on our android app which necessitates the usage of Google’s Maps SDK for android which will require the creation of an API key in order to be able to access these services. For this we need to navigate to the google developer console, then create a new project for which we want to use these services and create a new API key for said project. These steps are all described in greater detail with extra guidance in the google maps SDK Kickstarter guide here: <https://developers.google.com/maps/documentation/android-sdk/start>

Once we have all of the necessary google side stuff done, we also need to make sure our android studio project has the necessary dependencies and values that are required for the successful functionality of the app, such as the API Key, whose steps are again shown in the same tutorial where it shows which files to modify and where to put specific values and lines of code such as the API\_key and which library dependencies to allow, as well as which permissions to specify that the app uses. Effectively, the majority of these modifications happen in 3 files, which would be the two build.gradle files, and then the android\_manifest.XML file.

Next, we can create a template maps Activity by right clicking anywhere in the project explorer and then clicking new->Acivity->maps activity, which may not appear all the time in which case you may have to look through the gallery, which is accessed in the same place new->Activity->Gallery.

Once we have the template, we enter it, in our case its called MapsActivity and we will specifically be coding in the kotlin files from now on.

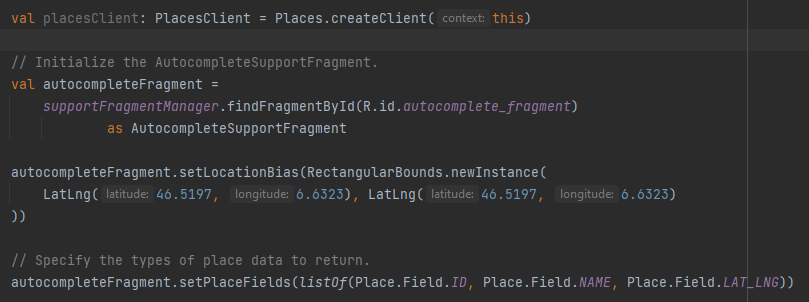
So, we have to implement a few things: firstly, we will need to get a map Fragment in which to place the map and call the onMapReady callback that effectively initializes the map.

val mapFragment = *supportFragmentManager* .findFragmentById(R.id.*map*) as SupportMapFragment  
mapFragment.getMapAsync(this)

Then, we move onto initializing the Places API in order to use autocomplete, which allows us to search a location in the search bar of this activity which will then spew out a list of autocomplete suggestions for the address you type in. Note that the long string In the last argument of the initialize function is the API key necessary for the running of all of the google maps services. (You have to make sure that your google project has billing enabled for this part to work).

if(!Places.isInitialized()){  
 Places.initialize(*applicationContext*, "AIzaSyCgJEFIHYvDnVD\_HIIH1JwXhbtviq3uAGU")  
}

Finally, we can implement all of the features. Firstly, we create a places Client for generating places, and a visual autocomplete fragment which will effectively be the search bar in the following lines of code.



Then, we add a listener to the autocomplete fragment. Specifically, the one called a “onPlaceSelectedListener” which will effectively do something within the code upon having chosen an address from the list of autocomplete suggestions.

In our case, we simply made this open the official google maps app and run dynamic turn-by-turn navigation from there. The reason for this choice that we made, rather than trying to implement a turn-by-turn dynamic navigation directly from our app using google API’s is actually because its simply not allowed. Effectively the google terms of service says that we can use many specific API’s provided by google, but we cannot use dynamic turn-by-turn navigation. To clarify, we can get a list of moves that the user has to do, which is the turn-by-turn navigation from one spot to the other using the Directions API, but we cannot do it dynamically whereby it updates the next instruction based on the users location.

To do this, our work-around would be basically opening google maps from one fragment (this code that we just did) and then reading the notifications that the google maps sends to us while running in the background. Thus, we then started implementing notification Listeners and handlers.

Thus, continuing on from before, once we have selected a place from the autocomplete suggestions (onPlaceSelectedListener) we then execute the code that opens another app and runs a specific service on it, which in this case is running navigation from google maps. This is effectively done by using something called an intent which allows us to perform a wide variety of actions, and which in this case we will use to simply open google maps navigation.

// Set up a PlaceSelectionListener to handle the response.  
autocompleteFragment.setOnPlaceSelectedListener(object : PlaceSelectionListener {  
 override fun onPlaceSelected(place: Place) {  
 // *TODO: Get info about the selected place.* Log.i(TAG, "Place: ${place.*name*}, ${place.*id*}\n")  
 Log.i(TAG, "google.navigation:q=${place.*latLng*.latitude},${place.*latLng*.longitude}&mode=l")  
 val intent: Intent = Intent(Intent.*ACTION\_VIEW*, Uri.parse("google.navigation:q=${place.*latLng*.latitude},${place.*latLng*.longitude}&mode=l"))  
 intent.setPackage("com.google.android.apps.maps")  
 if(intent.resolveActivity(*packageManager*) != null){  
 startActivity(intent)  
 }  
 }  
  
 override fun onError(status: Status) {  
 // *TODO: Handle the error.* Log.i(TAG, "An error occurred: $status")  
 }  
})

As we can see above, its effectively the usage of the value called intent, which specifies that we want to run a google maps related service, and then calling startActivity() on this intent value.

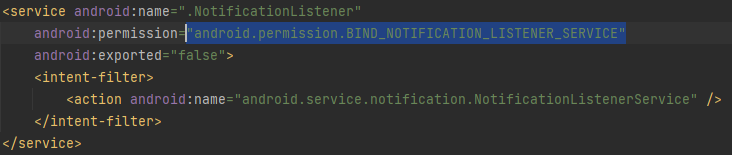
Nextly, once we had all of the underlying kotlin code for the running of the maps Activity, we can go to the XML file that was also generated by creating the Maps Activity, and simply make the modification to it so that it contains a fragment inside of it which represents the map that we put there, and then inside of that fragment itself it contains a cardview at the top which represents the searchbar for the autocomplete suggestions.

<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/map"  
 android:name="com.google.android.gms.maps.SupportMapFragment"  
 android:layout\_width="match\_parent"  
 android:layout\_height="match\_parent"  
 tools:context=".MapsActivity" />  
  
 <androidx.cardview.widget.CardView  
 android:layout\_width="match\_parent"  
 android:layout\_height="wrap\_content"  
 android:layout\_marginRight="6dp"  
 android:layout\_marginLeft="6dp"  
 android:layout\_marginTop="6dp">  
  
 <fragment  
 android:id="@+id/autocomplete\_fragment"  
 android:name="com.google.android.libraries.places.widget.AutocompleteSupportFragment"  
 android:layout\_width="match\_parent"  
 android:layout\_height="wrap\_content" />  
 </androidx.cardview.widget.CardView>

## Implementation of Notification Listener and Handlers

Thus, having finished all of the google maps related shenanigans, we move onto notification Listeners and handlers.

We first have to implement the permissions for notification Listener. Ironically, we cannot actually request the permission necessary to allow the usage of the notification Listener ("android.permission.BIND\_NOTIFICATION\_LISTENER\_SERVICE"). The reason for this is that we believe it is some sort of a system permission which has to specifically be granted by the user. However, even this sometimes may not work as you may need root access on your phone to be able to even give it. Thus, we specify that we “want” to access this permission but do not actually attempt to access it. We do this by going to the manifest file again and adding a “service” called notification listener which has a sub-tag that specifies that one of the permissions it needs is this BIND\_NOTIFICATION\_LISTENER\_SERVICE:



Once we have this implemented, we then have to implement something that kind of requests the permission but not directly. Our solution was simply to switch to the page of the permission required whenever we opened the fragment, which would give the user the option to enable said permission directly from settings. This is done in whichever fragment may require it, which we decided to simply put in both the Bluetooth fragment and maps activity such that it would be impossible to not have been prompted for notification permission before initiating communication with the ESP 32. Effectively, this code would be executed in the onCreate method of whichever activity wanted it. On a side note, this is also done by method intent where we specify an intent to open another service from another app (in this case settings) and then we actually initiate this intent by calling startActivity on it.

val intent = Intent("android.settings.ACTION\_NOTIFICATION\_LISTENER\_SETTINGS")  
startActivity(intent)

Next, we can finally start implementing the actual notification listeners.

We do this by creating a new kotlin class called notification Listener for example, which simply has to extend from the class NotificationListenerService and has to specifically override the function onNotificationPosted, which will allow us to run notification handlers upon receiving any given notification.

private lateinit var lastNotification: StatusBarNotification;  
override fun onListenerConnected() {  
 super.onListenerConnected()  
}  
  
@RequiresApi(Build.VERSION\_CODES.*S*)  
override fun onNotificationPosted(sbn: StatusBarNotification?) {  
 super.onNotificationPosted(sbn)

I don’t believe there is any need to implement onListenerConnected(), but we did so anyways just in case we needed to do something specific upon the creation of the Notification Listener.

To continue, this notification listener does not have to be explicitly created or instantiated anywhere at any time whatsoever as it is done completely automatically by the android app itself and handles all of that area.

Then, for the notification handler, the only thing we have access to Is the status bar notification, referred to as sbn in the code. The sbn allows us to access practically everything that you can physically see on any given notification. The majority of the relevant data that we want to acess in any given notification of sbn will effectively be in sbn.notification.extras. This extra value is a Bundle, which is a data type much like a map, whereby it has a key-value relationship for all of its entries, and all of its values have to be accessed by method of get(”some String”). For this part, all of the names and structure of the data within extras had to simply be discovered by method of running the debugger dozens of times for different types of notifications and then implementing different handlers for different cases.

In general what the handler would do to distinguish which app the notification came from, we would simply do a switch on sbn.packageName, which would quite easily distinguish the app. For example google maps would simply have a packageName of com.google.android.apps.maps, or whatsapp would be com.whatsapp, and so on. This was done by method of using a kotlin when statement.

when(sbn.*packageName*){

Once we had the app, we would simply add a handler inside of each switch statement case that would perform a different set of actions for each different app. Even for the texting services which we added support for, we had to process the notifications quite differently as many of them were structured in a relatively unique way from one another. An example of one of the simpler handlers for whatsapp can be seen below.

"com.whatsapp" -> {  
if(!sbn.*notification*.extras.containsKey("android.textLines")) {  
 app = APP.*WHATSAPP* notif = Notification(  
 app,  
 sbn.*notification*.extras.get("android.title").*toString*(),  
 sbn.*notification*.extras.get("android.text").*toString*()  
 )  
}

So, to elaborate, in the android.title, and android.text variables within sbn.notification.extras, in the case of a whatsapp notification, we have the the person who sent the message is contained in the android.title key-value entry of extras, and the actual tex-message, is in android.text. Then, we create a notification to be sent off to the ESP using this data, which will be explained in some more detail below. Furthermore, in the case of whatsapp, you can see that we also have a strange if statement about the android.textLines. This is effectively to filter out unnecessary whatsapp notifications that say something along the lines “from 2 conversations ago” or some other unnecessary notifications that we didn’t want to display on the glasses. The easiest way to distinguish such whatsapp notifications was to simply check if their bundle contained a key called “android.textLines”.

In general, most of the handlers would be relatively similar in their method of handling the actual notification, just that each one had its own quirks in formatting and whatnot which we had to take into account for each case, so to avoid redundancy we only showed one example.

However, In the case of google maps, we had to also implement a special extra constraint which was primarily the filtering of spam, whereby we would try to reduce the amount of google maps notifications passed onto the ESP. This was necessary due to the relatively large amount of notifications that google maps would keep continually sending to our phone during a journey. Thus, to prevent clogging of the communication channel between the phone and glasses, we simply added a check in the code that would take the previous google maps notification’s data and compare it to the current one and if, for example, the address hadn’t changed or the distance hadn’t changed by more than a couple of meters, we simply wouldn’t notify the ESP until a more significant change had taken place.

Then, once the handling for each different app was done, we would create a new Notification variable, which in our case was a structure we designed for sending notifications to the glasses and packaging data into one structure.

Basically, this notification data structure, located in Notification.kt, would primarily be used to package the data by calling its constructor with the specific set of values that we would want it to contain and then calling packForDelivery() on said notification before sending it off. This method would effectively do bit manipulation on said data in order to format it in such a way that the ESP can recognize it and then manipulate it to access said data. its code wont be posted on this manual here as it is relatively long and its specific format that we decided on will most likely be better explained from the ESP side of things.

Lastly, one of the remaining things to be done for the notification handling was to add a specific couple of Booleans and a queue for managing the sending of notifications in a relatively controlled and manner so as to only send the notifications to the ESP upon receiving communication from the ESP saying that we are allowed to do so. Otherwise, we would simply store notifications to be sent in a queue if we didn’t have the right set of Booleans necessary for sending them off. Simply speaking, the Bluetooth activity would have some parts in its code whereby the Booleans would be set to true upon receiving a communication from the ESP, upon which we would then be able to send the data over the Bluetooth channel.

Thus, concluding the creation of notification listening and handling in said app as well as the overall functionality of the android app itself.