Handbook of Android SDK

- By myBrainTechnologies

Introduction

Who is myBrain Technologies?

myBrain Technologies is a company that develops and markets solutions applied to health and well-being, whose two founders, doctors in neuroscience, come from the Institute of Brain and Spinal Cord at the Pitié-Salpetriere. In close collaboration with the medical and academic world, myBrain is developing new tools for real-time assessment of our mental state by measuring brain activity. The first application of myBrain makes it possible to accompany the user in the management of his stress and his anxiety by a device of “neurofeedback”.

What is our SDK?

The new version of SDK provides powerful tools for Android developers which helps them extract and analyze EEG data from our devices(QPLUS, MELOMIND…).

About this handbook

This handbook aims to be the one-stop-shop for myBrainTech Android SDK documentation, it is divided into 2 main parts:

- Getting start: Presenting the requirements before implementation of SDK.
- Presentation of tools: Designed to give developers a complete introduction to the functions provided by SDK.

Getting start

In this section, the requirements to implement SDK will be presented. Meanwhile, rather than starting from zero, reading and trying a demo provided by us are highly recommended.

Mark: Demo is developed on the platform Android studio, you can use other IDE but using the same may help you to understand easier.

Installation SDK to your App

There are two methods to install an external library to an Android application project:

Add SDK file by hand

1. Create a folder in the route: your_project -> app, inside the folder app.
2. Then add the sdk file into this lib folder.
3. Go to the build.gradle file (your_project -> app -> build.gradle) and add the following code into dependencies to declare the implementation:

   implementation files('libs/full_name_of_your_lib.aar')
4. Sync the project and SDK has been installed to your project now.

Implement directly

1. Declare the implementation in your dependencies in the build.gradle file.

```
dependencies }
    implementation("com.mybraintech.android:sdk:3.0.0.rc3")
}
```

2. Declare your username and password that we provide to you in the local.properties file(project -> local.properties).

```
maven_user= your_username
maven_password= your_password
```

3. Modify the build.gradle(project -> build.gradle) and repositories.gradle(project -> repositories.gradle). That will make your application download and install SDK to your project.

```
buildscript {
    apply from: 'repositories.gradle'
    addRepos(repositories)

    dependencies {
        classpath "com.android.tools.build:gradle:7.0.0"
        classpath "org.jetbrains.kotlin:kotlin-gradle-
plugin:1.5.20"

        // in the individual module build.gradle files
    }
}

allprojects {
    addRepos(repositories)
}

task clean(type: Delete) {
    delete rootProject.buildDir
}
// build.gradle
```

```
def addRepos(RepositoryHandler handler) {
    handler.google()
    handler.mavenCentral()
    handler.jcenter()
    handler.maven { url 'https://oss.sonatype.org/content/
repositories/snapshots' }
    handler.maven { url 'https://maven.fabric.io/public' }
    handler.maven { url "https://jitpack.io" }

    Properties properties = new Properties()

properties.load(project.rootProject.file('local.properties').n
ewDataInputStream())
    def maven_user = properties.getProperty('maven_user')
    def maven_password =
properties.getProperty('maven_password')
    handler.maven {
        url "https://package.mybraintech.com/repository/
maven-2-releases/"
        credentials {
            username maven_user
            password maven_password
        }
    }
    handler.maven {
        url "https://package.mybraintech.com/repository/
maven-2-snapshots/"
        credentials {
            username maven_user
            password maven_password
        }
    }
}

ext.addRepos = this.&addRepos
// repositories.gradle
```

Using SDK in code

Important: some of functions of SDK are based on the Google service, make sure your device have the full Google services in relatively new version.

Mark: In this handbook, example code is written in Kotlin, feel free to try in Java when you understand how to use SDK.

Declare API in your activity

To get start, import com.mybraintech.sdk.MbtClient to the activity and declare an object: MbtClient in your activity, this API provides all the tools that let your application connect to the device, extract data and save data.

```
private lateinit var mbtClient: MbtClient

// initialized mbtClient
mbtClient = MbtClientManager.getMbtClient(applicationContext,
EnumMBTDevice.Q_PLUS)
```

Call function MbtClientManager.getMbtClient() to initialize the MbtClient, this function is from com.mybraintech.sdk.MbtClientManager, don't forget to import it.

Function or class	Variables	Description
MbtClientManager.getMbtClient (Context, Device type)	Context	The context of current activity.
	Device type	- Declare the type of device(QPLUS or MELOMIND). - Creating your own type won't work, all the types have been defined in EnumMBTDevice
EnumMBTDevice()	None	- It is from com.mybraintech.sdk.core.model, just import all element from this package. - Contains only two elements: Q_PLUS and MELOMIND.

Tools provided by MbtClient API

The following table shows all functions provided by MbtClient, the required variables by every functions and the value returned from each function has been displayed.

Function	Return
startScan(ScanResultListener)	void
stopScan()	void
connect(MbtDevice, ConnectionListener)	void
disconnect()	void
getBleConnectionStatus()	BleConnectionStatus
getDeviceInformation(DeviceInformationListener)	void
getDeviceType()	EnumMBTDevice
startEEG(EEGParams, EEGListener)	void
stopEEG()	void
startEEGRecording(RecordingOption, RecordingListener)	void
stopEEGRecording()	void
isEEGEnabled()	Boolean
isRecordingEnable()	Boolean

Scan device and stop scanning

The whole SDK is based on the bluetooth, make sure that device used for debugging application has bluetooth hardware.

Important: After Android 6, some permissions are defined as level dangerous. It means that these permissions can't be granted as default. We will declare two permissions for scanning and connection: ACCESS_COARSE_LOCATION and ACCESS_FINE_LOCATION. They should be granted by users. See demo how to do it.

When the bluetooth is enable, it's now possible to find other bluetooth devices nearby. Make your API call function startScan() to scan the devices nearby. Since there are many possible bluetooth devices, for example, a computer or a earphone. Our SDK divides them into two types: MbtDevice and others. android.bluetooth.BluetoothDevice

Function or class	Description
startScan(ScanResultListener)	- Scan all the devices enabled bluetooth nearby. - Fetch the device names when the listener has received a new device detected event. - Fetch error message when error occurs during scanning.
stopScan()	Stop scanning. Should be called when you finish scanning or want to stop it.

startScan() requires an input variable ScanResultListener. Basically speaking, a listener will receive something when the defined event triggers. For example, onClickListener will detect an event whether a button is pushed.

Mark: If there exists some attract function inside listener class, to implement this listener, you will have to add all theses members and override them.

Class	Description
ScanResultListener{}	Three members required to implement ScanResultListener: - **fun** onMbtDevices(List<MbtDevice>): return a list of MbtDevice. - **fun** onOtherDevices(List<BluetoothDevice>): return a list of other bluetooth devices. - **fun** onScanError(Throwable): return a error when it occurs.
MbtDevice(BluetoothDevice)	- This is a class built in SDK. To construct an object, you have to fill a android.bluetooth.BluetoothDevice to your constructor. - Inside this class, you have only one public variable: **bluetoothDevice** with type android.bluetooth.BluetoothDevice. - Build an object to distinguish other devices and device of MyBrainTech.

This is a short version of code in order to make you understand how startScan() works, a MbtDevice can be selected and defined here, as example shows.

```
mbtClient.startScan(object : ScanResultListener {

    override fun onMbtDevices(mbtDevices: List<MbtDevice>) {

        // call stopScan when a mbtDevice has been found

        mbtClient.stopScan()

        mbtDevice = mbtDevices[0] }


    override fun onOtherDevices(otherDevices: List<BluetoothDevice>) {

        // show device name by otherDevices[0].device.name }


    override fun onScanError(error: Throwable) {

        // error massage… }

})
```

Remember to stop scanning when the task finished, in case there will be some conflicts and waste of resources.

Connect and Disconnect device

Once you have decided which device you want to connect, you can now connect to this MbtDevice. In this step, an object MbtDevice will be used. So don't forget to declare and initialize it before.

Function	Description
connect(MbtDevice, ConnectionListener)	Connect to the device you want. Return the data, variables when events take place.
disconnect()	disconnect the device.

ConnectionListener requires 7 members, here are some descriptions.

Class	Description	
ConnectionListener	**Function**	**Description**
	onBonded(*BluetoothDevice*)	
	onBondingFailed(*BluetoothDevice*)	
	onBondingRequired(*BluetoothDevice*)	
	onServiceDiscovered()	
	onDeviceReady()	This function will be called when the ConnectionListener finds that MbtDevice is ready for collecting data and transferring data.
	onDeviceDisconnected()	When you disconnect the device or lose the connection by accidents, onDeviceDisconnected() will be called.
	onConnectionError(Throwable)	When an error occurs, you can fetch the error massage here to find out the reason and how to solve it.

ConnectionListener can be also implemented to Activity, remember to add functions to your activity in you do so. As following:

```
// make sure that you are not connecting to another device
// and the mbtDevice should not be null. . .
if (!isMbtConnected && mbtDevice != null){
    mbtClient.connect(mbtDevice!!, this)
}
```

We declare the second variable as **this** because our activity has been implemented with ConnectionListener.

Checking status and device information

When you have connected to your QPLUS(MbtDevice), you can call some functions to check connection, get more details about device.

Function	Description
getBleConnectionStatus()	Whatever you have connected to a device to not, you can check the connection status by call mbtClient.getBleConnectionStatus(). Returns a variable: BleConnectionStatus(com.mybraintech.sdk.core.model.BleConnectionStatus)
getDeviceInformation(DeviceInformationListener)	Get an object DeviceInformation() that contains all details about the connecting device.
getDeviceType()	Return a EnumMBTDevice showing device type

Class	Content elements	Description
BleConnectionStatus()	isConnectionEstablished	Return a boolean to show if connection is established
	mbtDevice: MbtDevice	To get the bluetooth device being connected.
DeviceInformationListener	onDeviceInformation(DeviceInformation)	Fetch the information of connecting device
	onDeviceInformationError(Throwable)	Get error when we can't get device information
DeviceInformation()	firmwareVersion: String	Return a string to display information of firmware version.
	hardwareVersion: String	Return a string to display information of hardware version.
	productName: String	Return a string to display the name of product
	uniqueDeviceIdentifier: String	Return a string to display the hardware id

All these information could be useful during the connection or anything unexpected happens like hardware bugs… Problems can be reported with the hardware information and can be solved faster.

Here is how we get DeviceInformation and initialize our object in demo, it will be used as an input variable when you want to save data as a file.

```
mbtClient.getDeviceInformation(object : DeviceInformationListener {
    override fun onDeviceInformation(deviceInformation:
DeviceInformation) {
        this@QplusSimpleActivity.deviceInformation = deviceInformation
        Timber.i(deviceInformation.toString())
    }

    override fun onDeviceInformationError(error: Throwable) {
        Timber.e(error)
    }
})
```

Strat to receive data from device

startEEG() and stopEEG() are two functions provided by API. They are main to receive data and other information from device.

Function	Description
startEEG(EEGParams, EEGListener)	Call this function to receive data from device. Receive an object MbtEEGPacker2 when new data is coming.
stopEEG()	stop receiving EEG from device.

Class	Content elements	Description
EEGParams(sampleRate, isTriggerStatusEnabled, isQualityCheckerEnabled)	sampleRate: Int	set the sample rate in Hz
	isTriggerStatusEnabled: Boolean	set if trigger is enabled or not
	isQualityCheckerEnabled: Boolean	set if quality checker is enabled
EEGListener	onEegPacket(MbtEEGPacket2)	It will receive an object MbtEEGPacket2 when a new dataset from QPLUS(device) is coming.
	onEegError(Throwable)	Return error if occurs.
MbtEEGPacket2()	channelsData: ArrayList<ArrayList<Float>>	This is a [n - by -m] matrix. - n(row number) = the number of channels, for example QPLUS has 4 channels. - m(column number) = sampleRate. For example, under 250Hz sampling frequency, there will be 250 columns. - All data in a row represent all data of a channel in 1s.
	statusData: ArrayList<Float>	The data when trigger is active

Class	Content elements	Description
	timestamp: long	a long shows time stamp during the EEG receiving
	qualities: ArrayList<Float>	- Each number inside list represent the quality of a channel data. - Information about quality: -> 0 : bad -> 0.5: EEG data containing artifacts -> 1 : good
	features: float [][]	Not defined yet, for the future use.

Here is what we did in our demo. The stopEEG() can be called wherever you want, but remember that it can be stopped only if EEG has been started to be received.

```
mbtClient.startEEG(
    EEGParams(
        sampleRate = 250,
        isTriggerStatusEnabled = isStatusEnabled,
        isQualityCheckerEnabled = true),
    object : EEGListener {
        override fun onEegPacket(mbtEEGPacket2: MbtEEGPacket2) {
            // do something here. . . }
        override fun onEegError(error: Throwable) {}
    },)
```

Record and save data from device

The inputs of startEEGRecording() are also two parameters: the first one, RecordingOption(), is to set configuration, and a listener, RecordingListener is to watch the events.

Function	Description
startEEGRecording(Recording Option, RecordingListener)	Set the configuration and start to record data.
stopEEGRecording()	stop recording data and save file.

Class	Content elements	Description
RecordingOption(outputFile, context, deviceInformation, recordId)	outputFile: java.io.file	– A java File, you have to create a File object and define its path before record data. – Important: to write data into external storage, you will have to declare and grantee 2 permissions: WRITE_EXTERNAL_STORAGE and READ_EXTERNAL_STORAGE
	context: com.mybraintech.sdk.core.model.KwakContext	A class, KwakContext(), from SDK. – It has only one variable: ownerId: string – ownerId is the subject name, for the first try, you can just set ownerId = '1'.
	deviceInformation: com.mybraintech.sdk.core.model.DeviceInformation	We have mentioned before, remember to initialize before recording data.
	recordId: string	An unique id for every file. There are many methods to generated unique code, for example: UUID.
RecordingListener	onRecordingSaved(File)	– After calling stopRecordingEEG(), SDK will create a file and save it in the predefined path. – The file received here is an object File, you can call all functions provided by this class(java.io.file).
	onRecordingError(Throwable)	Return error if occurs.

In our demo, we create a provider so that application can provide the file to other applications. But we recommend you to start in a easier way in order to understand the function. The following code a more simple version, it displays the path save the file.
And to find the saved file on your device, you can connect your device to your computer, open your Android Studio(IDE). On Android Studio, there is a tool: Device File Explorer (View -> Tool Windows -> Device File Explorer).

1. Search by the keyword: name_of_your_application(com.example.demoqplus)
2. Find the file in the path.
3. Do what you want.

```
mbtClient.startEEGRecording(
    RecordingOption(
        outputFile,
        KwakContext().apply { ownerId = "1" },
        deviceInformation!!,
        "record-" + UUID.randomUUID().toString()
    ),
    object : RecordingListener {
        override fun onRecordingSaved(outputFile: File) {
            // your data recorded, waiting to be saved…
            val path = outputFile.path }
        override fun onRecordingError(error:Throwable) { }
    })
```

What's more?

For now, you have known all the tools to connect and receive data from myBrainTech devices. You can try to build your own android application and test.
Feel free to tell us if you have some suggestion for our product, SDK and documentation, thanks your and good luck!