Introduction to Demo

What is our SDK?

Installation

To install our SDK to your application, you have to declare the implementation in your dependencies in the build.gradle file.
Attention: there are **multiple** build.gradle files in your application project, add the code to the one in the following path: project -> app -> build.gradle.

```
dependencies }

    implementation(“com.mybraintech.android:sdk:3.0.0.rc3”)

}
```

Meanwhile, since the SDK is not available for public. You have to 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
```

Finally, you may also have to 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').new
DataInputStream())
    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
```

When you have done the previous steps, now you can import SDK to your activities or other classes.

Permission requirement

Considering the security issue, Android doesn't allow to access some of services without permissions, such as internet, internal storage... Thus you will have to declare permissions in your AndroidManifest.xml.

After Android 6, some permissions have to be authorized by users. They cannot be granted if you only declare them inside AndroidManifest.xml. More details can be found in the Android official website. **add link here**

Our demo requires 4 permissions that has to be granted by users, add these 4 permissions to your AndroidManifest.xml and ask users when your applications is running.

```
ACCESS_COARSE_LOCATION

ACCESS_FINE_LOCATION

WRITE_EXTERNAL_STORAGE

READ_EXTERNAL_STORAGE
```

Basic functions and explication

With the help of functions are provided by class MbtClient, you can connect your device, record data…
Before declaring MbtClient, you have to import com.mybraintech.sdk.MbtClient to your code. To to declare and initialize your MbtClient. The code is written by Kotlin but you can also try on in Java.

To initialize the MbtClient, you may have to call function:

Function
MbtClientManager.getMbtClient(Context, Device type)

Where Context is the context of your current activity and Device type has been defined in a Enum class: EnumMBTDevice(). The followings are the devices defined, choose the right one to initialize your MbtClient.

Device type	Description
Q_PLUS	You are going to connect a QPLUS HEADSET
MELOMIND	You are going to connect a MELOMIND HEADSET

Here is an example that illustrates how we did in our demo, you have to declare a variable in type MbtClient in the activity that all the class can view it. And then initialize it when you have decided the device type. We create this demo for QPLUS, thus we just initialize MbtClient inside onCreate() function and set device = EnumMBTDevice.Q_PLUS.

```
private lateinit var mbtClient: MbtClient

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

The following table shows all functions provided by MbtClient, we will give you a short introduction and a simple example to use these functions. You can also modify the demo to see how do they work exactly.

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

When the bluetooth is enable on the device, it's now possible to find other bluetooth devices nearby from your device. To do so with our SDK and your application, simply 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

Class	Description
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.

Function	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.

Class	Description
ScanResultListener{}	Three members required to implement ScanResultListener: 1. **fun** onMbtDevices(List<MbtDevice>): return a list of MbtDevice. 2. **fun** onOtherDevices(List<BluetoothDevice>): return a list of other bluetooth devices. 3. **fun** onScanError(Throwable): return a error when it occurs. All these three member should be implemented inside your listener or to your activity. Then override them by adding functions desired. Finally, these functions will receive the variables when event arrives.

We show you a short version of code in order to make you understand how do these two functions work and how to add your code. Feel free to modify and test on the demo.
(**Remember**: Don't forget to initialize an object MbtDevice after scanning and selecting a device. It will be used when you want to connect to your device.)

```
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… }

})
```

To stop scanning, it's easier: just call stopScan and nothing to worry about, application will stop scanning Bluetooth devices.

```
// stop scan when you have chosen the right MbtDevice
mbtClient.stopScan()
```

Connect and Disconnect

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

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

Class	Description	
MbtDevice	To let MbtClient know which device you want to connect.	
ConnectionListener	7 members have to be implemented to your listener or activity and override.	
Members of Listener	**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.

Class	Description	
	onConnectionError(Throwable)	When an error occurs, you can fetch the error massage here to find out the reason and how to solve it.

In our demo, we implement ConnectionListener to our Activity, thus we add these 7 members to your activity and call connect() in this way:

```
// 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. 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 to get connecting device type

If later you want to record EEG and save as a file, now you have to declare and initialize an object DeviceInformation. It will be used for recording.

Class	Description	
BleConnectionStatus()	This class provides you two public variables: isConnectionEstablished and mbtDevice.	
	isConnectionEstablished: Boolean	To check if connection is established.
	mbtDevice: MbtDevice	To get the bluetooth device being connected.

Class	Description	
DeviceInformationListener	onDeviceInformation(DeviceInformation)	Fetch the information of connecting device
	onDeviceInformationError(Throwable)	Get error when we can't get device information
DeviceInformation()	Variables	Description
	firmwareVersion: String	
	hardwareVersion: String	
	productName: String	
	uniqueDeviceIdentifier: String	

Here is how we get DeviceInformation and initialize our object in demo.

```
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 QPLUS

When everything is ready, we can now start to receive data from MbtDevice through your application. 2 main functions will be called in this step: startEEG() and stopEEG().

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

Class	Elements	Description
EEGParams(sampleRate, isTriggerStatusEnabled, isQualityCheckerEnabled)	sampleRate: Int	set the sample rate in Hz
	isTriggerStatusEnabled: Boolean	set if trigger is enabled or not, see more details about Trigger in this link.
	isQualityCheckerEnabled: Boolean	set if quality checker is enabled, see more details about quality checker in this link.
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>	wwwww
	timestamp: long	a long shows time stamp during the EEG receiving
	qualities: ArrayList<Float>	Return a channel number size of list. – Each number inside list represent the quality of a channel. – Information about quality: 0: bad 0.25: 1: good
	features: float [][]	Coming soon

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) {}
    },)
```

Strat to save data from QPLUS

The framework of startEEGRecording() is similar to startEEG(). But remember to call startEEG() first and then call startEEGRecording() to record desired data.
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	Elements	Description
RecordingOption(outputFile, context, deviceInformation, recordId)	outputFile: java.io.file	
	context: com.mybraintech.sdk.core.model.KwakContext	A class, KwakContext(), from our 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. This function will receive this file if no error occurs. 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 save EEG data 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) { }
    })
```