Lab 8: Bluetooth Low Energy

In case you found something to improve, please tell us!
`https://forms.gle/3U6WrZNyNx2nBXQ38`

In this lab we are going to learn how to interact with Bluetooth Low Energy (BLE) peripherals. In particular, we will set up a BLE connection with a HR chest belt to display heart rate measurements.

1 Android Studio Tricks

Here are very useful **Android Studio** tricks you should always use (check *Section 5* of **Lab1** for more detailed explanation on how to use **Android Studio** debug 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. **Check for compilation errors:** They are usually quite self-explanatory.
4. **Check errors in logcat, and use the debugger:** errors are highlighted in logcat, 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

Bluetooth Low Energy (BLE) was included in version 4.0 of the Bluetooth Specifications, adopted in 2010.

It is based on a central-peripheral architecture, meaning that the central device scans, looking for advertisements, and the peripheral one advertises itself and its capabilities. BLE devices communicate using messages in the GATT (Generic Attribute Profile) format. In this lesson, the tablet will be the GATT client. It will scan and discover all services provided by the server as well as the different characteristics of a given service.

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

We will use a cardiac chest band (such as the Geonaoute HR monitor) as the server. We will integrate it into our current Sports Tracker app as an alternative source of data for heart rate acquisition.

3 Key terms and concepts

The following is a summary of key BLE terms and concepts:

- **Generic Attribute Profile (GATT)**: The GATT profile is a general specification for sending and receiving short pieces of data known as ”attributes” over a BLE link. All current BLE application profiles are based on GATT.
- **Profiles**: Bluetooth defines many profiles for BLE devices. A profile is a specification of how a device works in a particular application. Note that a device can implement more than one profile. For example, a device could have a heart rate monitor, but also other sensors.
- **Attribute Protocol (ATT)**: GATT is built on top of the Attribute Protocol (ATT). This is also referred to as GATT/ATT. ATT is optimized to run on BLE devices. To this end, it uses as few bytes as possible. Each attribute is uniquely identified by a Universally Unique Identifier (UUID), which is a standardized 128-bit format for a string ID used to uniquely identify information. The attributes transported by ATT are formatted as characteristics and services.
- **Characteristic**: A characteristic contains a single value and one or several descriptors that describe the characteristic’s value. A characteristic can be thought of as a type, analogous to a class.
- **Descriptor**: Descriptors are defined attributes that describe a characteristic value. For example, a descriptor might specify a human-readable description, an acceptable range for a characteristic’s value, or a unit of measure that is specific to a characteristic’s value.
- **Service** A service is a collection of characteristics. For example, you could have a service called ”Heart Rate Monitor” that includes characteristics such as ”Heart Rate Measurement.”

4 Set up Bluetooth

The first step is adding the Bluetooth permissions to your manifest file to use the Bluetooth APIs. Add the following code to the manifest:

```
/** mobile -> src -> main -> AndroidManifest.xml **/
<uses-permission android:name="android.permission.BLUETOOTH" />
<uses-permission android:name="android.permission.BLUETOOTH_ADMIN" />
<uses-permission android:name="android.permission.ACCESS_FINE_LOCATION" />
```

Once the permissions are in place, we need to have a switch for choosing the heart rate belt in the app. First, in **NewRecordingScreen.kt**, we define a switch button. Users choose a heart rate source using this switch. By default, it should be off, which means the app uses the smartwatch to receive heart rates. When the switch is toggled, the app will use the belt. Add the following code at the bottom of our **NewRecordingScreen** composable:

```
/** mobile -> NewRecordingScreen.kt **/
Row(
    modifier = modifier.padding(top = 32.dp),
    horizontalArrangement = Arrangement.Center,
    verticalAlignment = Alignment.CenterVertically
) {
    Text(text = stringResource(R.string.heart_rate_belt_text))
    Spacer(modifier.padding(end = 8.dp))
    Switch(checked = isChecked, onCheckedChange = { checked ->
        isChecked = checked
        if (checked) {
            device = DEVICE.BELT
        } else {
            device = DEVICE.SMARTWATCH
        }
    })
}
```

For this, we also need a new boolean state variable called **isChecked** to receive updates for the state of the Switch.

```
var isChecked by remember { mutableStateOf(false) }
```

Bluetooth setup is accomplished in two steps using the BluetoothAdapter:

1. Get the **BluetoothAdapter**.

The BluetoothAdapter is required for all Bluetooth activity. The BluetoothAdapter represents the device's own Bluetooth radio. There's one Bluetooth adapter for the entire system, and your app can interact with it using this object. To get the BluetoothAdapter,

call the **getDefaultAdapter()** method. If getDefaultAdapter() returns null, then the device doesn't support Bluetooth.

In mobile's module **NewRecordingViewModel.kt**, define BluetoothAdapter as follows:

```
/** mobile -> NewRecordingViewModel.kt **/
val bluetoothAdapter: BluetoothAdapter? = BluetoothAdapter.getDefaultAdapter()
```

Then, when creating the Screen (before **Surface(..)**), you can check if the device supports Bluetooth:

```
/** mobile -> NewRecordingScreen.kt **/
val context = LocalContext.current
if(newRecordingViewModel.bluetoothAdapter==null){
    Toast.makeText(context,"Bluetooth not supported",
        Toast.LENGTH_SHORT).show()
}
```

2. Enable Bluetooth.

Next, you need to ensure that Bluetooth is enabled. Call **isEnabled()** to check whether Bluetooth is currently enabled. If this method returns false, then Bluetooth is disabled. To request that Bluetooth be enabled, we use an implicit intent. Use **registerForActivityResult** to define **resultLauncher**, and launch it when you request enabling Bluetooth, passing in an **ACTION_REQUEST_ENABLE** intent action. This call issues a request to enable Bluetooth through the system settings (without stopping your app). Add this code, also in the **NewRecordingScreen** composable:

```
/** mobile -> NewRecordingScreen.kt **/
val resultLauncher = rememberLauncherForActivityResult(
    contract = ActivityResultContracts.StartActivityForResult(),
    onResult = { result ->
        if (result.resultCode != Activity.RESULT_OK) {
            Toast.makeText(
                context, "Bluetooth is not enabled!",
                Toast.LENGTH_SHORT
            ).show()
            isChecked = false
            device = DEVICE.SMARTWATCH
        }
    }
)
```

```
/** mobile -> NewRecordingScreen.kt **/
if (isChecked == true) {
    if (newRecordingViewModel.bluetoothAdapter?.isEnabled == false) {
        val enableBtIntent = Intent(BluetoothAdapter.ACTION_REQUEST_ENABLE)
        resultLauncher.launch(enableBtIntent)
    }
}
```

In this way every time we change the isChecked variable, we are going to do this check and launch a dialog requesting user permission to enable Bluetooth if rrequired. If the user grants permission, the system enables Bluetooth, and the focus returns to the app.

If enabling Bluetooth succeeds, your activity receives the **RESULT_OK** result code in the onResult() callback.

If Bluetooth was not enabled (e.g. the user responded "Deny"), then the result code is **RESULT_CANCELED**. The variable in the main activity should be false, a message should be shown to the user (e.g. via a toast), and the switch should go back from the belt to the watch state.

5 Find BLE devices

To establish a connection between the tablet and the HR belt, we first need to scan for available BLE devices using **startScan()**. This method takes a **ScanCallback** as a parameter. You must implement this callback because that is how scan results are returned. Since scanning is battery-intensive, you should observe the following guidelines:

- As soon as you find the desired device, stop scanning.
- Never scan on a loop, and always set a time limit on your scan. A previously available device may have moved out of range, and continuing to scan drains the battery.

In **ExerciseLiveViewModel.kt**, add the following code to scan BLE devices during a limited time defined as **SCAN_PERIOD**. We will call **scanLeDevice()** later on to initiate a BLE link.

```
/** mobile -> ExerciseLiveViewModel.kt **/
private val bluetoothAdapter: BluetoothAdapter? =
            BluetoothAdapter.getDefaultAdapter()
private val bluetoothLeScanner = bluetoothAdapter?.bluetoothLeScanner
private var scanning = false
private val handler = Handler()
```

```
private val SCAN_PERIOD: Long = 5000

fun scanLeDevice() {
    if (!scanning) { // Stops scanning after a pre-defined scan period.
        handler.postDelayed({
            scanning = false
            bluetoothLeScanner?.stopScan(leScanCallback)
        }, SCAN_PERIOD)
        scanning = true
        bluetoothLeScanner?.startScan(leScanCallback)
    } else {
        scanning = false
        bluetoothLeScanner?.stopScan(leScanCallback)
    }
}
```

The following code is a simple implementation of **ScanCallback**, which is the interface used to deliver BLE scan results. In this implementation, Log messages are shown on the consolle when devices are found.

```
/** mobile -> ExerciseLiveViewModel.kt **/
private val leScanCallback: ScanCallback = object : ScanCallback() {
    override fun onScanResult(callbackType: Int, result: ScanResult) {
        super.onScanResult(callbackType, result)
            Log.i(
                ExerciseLiveViewModel::class.simpleName,
                "Name: ${result.device.name}, " +
                        "Address: ${result.device.address}," +
                        " RSSI: ${result.rssi}"
            )
    }
}
```

scanLeDevice should be called only if the user set his/her choosen HR device to DEVICE.BELT. To implement this functionality, we will need to hoist the state of **device** from **NewRecordingScreen** to **MainActivity**, and then provide it to both screens thay require it (i.e., **NewRecordingScreen** and **ExerciseLiveScreen**). We also need to implement a state-hoisting function **onUpdateDevice** for the changing of the value of the device, and restructure all code assigning a calue to the *device* variable accordingly. Also, be sure to delete the declaration of the *device* state variable from **NewRecordingScreen**.

```
/** mobile -> NewRecordingScreen.kt **/
@Composable
fun NewRecordingScreen(
    ...
    device: DEVICE,
    onUpdateDevice: (DEVICE) -> Unit,
    onLogoutClicked: ...,
    ...
) {
    var isChecked by remember { mutableStateOf(device == DEVICE.SMARTWATCH) }
    ...
    Switch(checked = isChecked, onCheckedChange = { checked ->
        isChecked = checked
        if (checked) {
            onUpdateDevice(DEVICE.BELT)
        } else {
            onUpdateDevice(DEVICE.SMARTWATCH)
        }
    })
}

/** mobile -> MainActivity **/
class MainActivity : ComponentActivity() {
    ...
    private var userKey ...
    private var device by mutableStateOf(DEVICE.SMARTWATCH)

    ...

    NewRecordingScreen(
        ...,
        device,
        onUpdateDevice = {
            device = it
        },
        onLogoutClicked = {...}
    )
    ...
    ExerciseLiveScreen(
        device,
```

```
        dataClient
    )

/** mobile -> ExerciseLiveScreen.kt **/
@Composable
fun ExerciseLiveScreen(
    device: DEVICE,
    dataClient: DataClient,
    modifier: Modifier = Modifier,
    exerciseLiveViewModel: ExerciseLiveViewModel = viewModel()
)
```

In **ExerciseLiveScreen.kt**, change the **LifecycleResumeEffect** composable as follows:

```
/** mobile -> ExerciseLiveScreen.kt **/
LifecycleResumeEffect {
    if (device == DEVICE.SMARTWATCH) {
        dataClient.addListener(exerciseLiveViewModel)
        sendCommandToWear("Start", context)
    } else {
        exerciseLiveViewModel.scanLeDevice()
    }
    exerciseLiveViewModel.getLastLocation(context)
    ...
}
```

In this function, callec when the user navigates to the **ExerciseLiveScreen**, we set up the communication with the smartwatch or scan for BLE devices depending on the user choice.

Launch the application now. The scan function callback we defined prints a nearby device's name in the Logcat environment, whenever it finds a new device. Try to find your HR belt in the list based on its MAC address. Remember that the belt must be worn to enable its BLE transceiver. Make sure you find your belt device not the ones of your neighbors! Increasing the value of **SCAN_PERIOD** can help if you do not see your belt. If needed, the app "Heart Rate Monitor" from BM Innovations GmbH can also help to find the device and check if it works.

Once you find your belt's MAC address, save the address in a variable.

```
/** mobile -> ExerciseLiveViewModel.kt **/
```

```
private val myDeviceAddress = "00:00:00:00:00:00"
//TODO: Replace this with your device's address
```

In this implementation, the app will be able only to interact with an HR belt with a given UUID. You may think of a more flexible solution, in which the user provides the belt UUID, or even better chooses among the UUID returned when scanning for devices, as an extension.

6 Connecting to the Bluetooth belt

Once the BLE device is discovered, we can start interacting with it. First, we have to connect to its GATT server using the **connectGatt()** method. This method takes three parameters: a **Context** object, **autoConnect** (a boolean indicating whether to automatically connect to the BLE device as soon as it becomes available), and a reference to a **BluetoothGattCallback**. Use **connectGatt()** in the scan callback after you find your device.

```
/** mobile -> ExerciseLiveViewModel.kt -> leScanCallback
    -> onScanResult() **/
if (result.device.address == myDeviceAddress){
    bluetoothLeScanner?.stopScan(this)
    result.device.connectGatt(context, false, gattCallback)
}
```

We need to provide two parameters for this **connectGatt** function: **context** and **gattCallback**. Since it is impossible to access the application context in a **ViewModel**, we need first to change the view model to an **AndroidViewModel** to pass the context when the view model is created. Change the class definition of **ExerciseLiveViewModel** as follows:

```
/** mobile -> ExerciseLiveViewModel.kt **/
class ExerciseLiveViewModel(application: Application) :
                            AndroidViewModel(application),
                            DataClient.OnDataChangedListener {
```

Then, define a context variable in this AndroidViewModel:

```
/** mobile -> ExerciseLiveViewModel.kt **/
private val context = getApplication<Application>().applicationContext
```

Once the **connectGatt** function tells the application which device to connect to, the app must connect to the GATT server on the BLE device. This connection requires a **BluetoothGattCallback** to receive notifications about the connection state, service discovery,

characteristic reads, and characteristic notifications. Add the following code to **ExerciseLiveViewModel.kt** to have a **BluetoothGattCallback**.

```
/** mobile -> ExerciseLiveViewModel.kt **/
private val gattCallback = object : BluetoothGattCallback() {
    override fun onConnectionStateChange(gatt: BluetoothGatt,
                                         status: Int, newState: Int) {
    }

    override fun onServicesDiscovered(gatt: BluetoothGatt?, status: Int) {
    }

    override fun onCharacteristicChanged(gatt: BluetoothGatt,
            characteristic: BluetoothGattCharacteristic) {
    }
}
```

As you see in the code, there are three different functions in the gattCallback, which we will explain and complete in the following sections.

6.1 Connection State Change

The **onConnectionStateChanged()** function is triggered inside the **gattCallback** when the connection to the device's GATT server changes. Define a variable to save the bluetoothGatt received when a connection is successfully changed to the connected state.

```
/** mobile -> ExerciseLiveViewModel.kt **/
private var bluetoothGatt: BluetoothGatt? = null
```

Then, you can fill in the **onConnectionStateChange** as follows:

```
/** mobile -> ExerciseLiveViewModel.kt -> gattCallback **/

override fun onConnectionStateChange(gatt: BluetoothGatt,
                                     status: Int, newState: Int) {
    val deviceAddress = gatt.device.address

    if (status == BluetoothGatt.GATT_SUCCESS) {
        if (newState == BluetoothProfile.STATE_CONNECTED) {
            Log.w("BluetoothGattCallback",
```

```
                    "Successfully connected to $deviceAddress")
                bluetoothGatt = gatt
                bluetoothGatt?.discoverServices()
            } else if (newState == BluetoothProfile.STATE_DISCONNECTED) {
                Log.w("BluetoothGattCallback",
                        "Successfully disconnected from $deviceAddress")
                gatt.close()
            }
        } else {
            Log.w("BluetoothGattCallback",
                  "Error $status encountered for $deviceAddress! Disconnecting...")
            gatt.close()
        }
}
```

6.2 Discover Services

The next thing to do once you connect to the GATT Server on the BLE device is to perform service discovery. This action provides information about the services available on the remote device as well as the service characteristics and their descriptors. In our example, once the service successfully connects to the device (indicated by the appropriate call to the **onConnectionStateChange()** function of the BluetoothGattCallback), the **discoverServices()** function queries the information from the BLE device. This is done by the code we already implemented as **bluetoothGatt?.discoverServices()**.

First, we need to define some values for the heart rate service. These values are taken from officially adopted BLE services for heart rate.

```
/** mobile -> ExerciseLiveViewModel.kt **/
private val HEART_RATE_SERVICE = "0000180D-0000-1000-8000-00805f9b34fb"
private val HEART_RATE_MEASUREMENT = "00002a37-0000-1000-8000-00805f9b34fb"
private val CLIENT_CHARACTERISTIC_CONFIG =
            "00002902-0000-1000-8000-00805f9b34fb"
```

The app needs to override the **onServicesDiscovered()** function in the BluetoothGattCallback. This function is called when the device reports on its available services.

```
/** mobile -> ExerciseLiveViewModel.kt -> gattCallback **/

override fun onServicesDiscovered(gatt: BluetoothGatt?, status: Int) {
```

```
    if (status == BluetoothGatt.GATT_SUCCESS) {
        val gattService = bluetoothGatt?.
            getService(UUID.fromString(HEART_RATE_SERVICE))
        val gattCharacteristics = gattService?.
            getCharacteristic(UUID.fromString(HEART_RATE_MEASUREMENT))
        setHeartRateCharacteristicNotification(gattCharacteristics!!, true)

    } else {
        Log.w("BluetoothGattCallback", "onServicesDiscovered received: $status")
    }
}
```

In the code above, when the service is successfully discovered, we get the heart rate service using its specific UUID. The UUID class should be imported from the **java.util** package. Then, the measurement characteristic is obtained from the heart rate service. This is the characteristic which has the heart rate values.

Once your app has connected to a GATT server and discovered services, it can read and write attributes, as supported. We will write the **setHeartRateCharacteristicNotification()** function in the following section.

6.3 Heart Rate Characteristics

It's common for BLE apps to ask to be notified when a particular characteristic changes on the device. In our app, we implement a function to call the **setCharacteristicNotification()** method specifically for the heart rate measurement. Add a function in this view model to be able to set the characteristic notification for the heart rate.

```
/** mobile -> ExerciseLiveViewModel.kt **/
private fun setHeartRateCharacteristicNotification(
            characteristic: BluetoothGattCharacteristic,
            enabled: Boolean) {
    bluetoothGatt?.let { gatt ->
        gatt.setCharacteristicNotification(characteristic, enabled)
        val descriptor = characteristic.getDescriptor(
                        UUID.fromString(CLIENT_CHARACTERISTIC_CONFIG))
        descriptor.value = BluetoothGattDescriptor.ENABLE_NOTIFICATION_VALUE
        gatt.writeDescriptor(descriptor)
    } ?: run {
        Log.w("BluetoothCallback", "BluetoothGatt not initialized")
```

```
    }
}
```

6.4 Characteristics Change

Once notifications are enabled for a characteristic, an **onCharacteristicChanged()** callback is triggered if the characteristic changes on the remote device:

```
/** mobile -> ExerciseLiveViewModel.kt -> gattCallback **/
override fun onCharacteristicChanged(gatt: BluetoothGatt,
             characteristic: BluetoothGattCharacteristic) {
    val heartRateReceived = characteristic.value.get(1).toInt()
    Log.i(ExerciseLiveViewModel::class.simpleName,
        "HR : ${characteristic.value.get(1)}")
        _heartRate.postValue(heartRateReceived)
        // Update HR plot series
        val size = _heartRateList.size
        _heartRateList.add(Point(size.toFloat(), heartRateReceived.toFloat()))
        _heartRateListLiveData.postValue(_heartRateList)
}
```

Note that since the BLE functions are not executing in the same thread as the main thread, we need to use **postValue()** function to update the heart rate as the live variable.

6.5 Stop BLE

For the last step, we need to stop BLE when we go out from the **ExerciseLiveScreen** . As you know about the composable lifecycle, inside the **LifecycleResumeEffect** in the **onPauseOrDispose** function is called while leaving from the screen. Thus, we change this function as follows:

```
/** mobile -> ExerciseLiveScreen.kt **/
onPauseOrDispose {
    if (device == DEVICE.SMARTWATCH) {
        dataClient.removeListener(exerciseLiveViewModel)
        sendCommandToWear("Stop", context)
    } else {
        exerciseLiveViewModel.stopBLE()
    }
}
```

Then, we implement the **stopBLE** function in the viewModel to stop the Bluetooth gatt.

```
/** mobile -> ExerciseLiveViewModel.kt **/
fun stopBLE(){
    bluetoothGatt?.let { gatt ->
        gatt.close()
        bluetoothGatt = null
    }
}
```

And now, in the end, we can just add a simple Text to display the selected device. We will place it below the Text with the latest HR value.

```
 Column(
    modifier = modifier
        .weight(1f)
) {
    Text(
        text = stringResource(R.string.heart_rate, heartRate),
        modifier = modifier
            .padding(top = 8.dp, bottom = 8.dp)
    )
    Text(text = "Device ${device.value}")
}
```

7 Finishing the project

We here conclude the guided development of the sport tracker app. You may have noticed that the addition of some further functionalities could greatly add to the app's value. For example, in its current form, the app does not store the acquired HR and location data in the cloud, nor can visualize the acquisitions performed in past exercise activities. We do not devote further labs to the implementations of these features, as their development could be done by using concepts already covered in this series of labs. Next weeks will instead be dedicated to the design of your own app.