Lab 5: ViewModels and System Services

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

In this lab, you will gain experience in using ViewModels in Android applications. Moreover, you will interface with the Sensor service on the smartwatch device to measure the user's heart rate (HR). Finally, you will plot the sensor acquisitions on the tablet using LiveData and Observers.

1 Android Studio Tricks

Here are very useful **Android Studio** tricks you should always use (check *Section 5* of **Lab1** for a 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. **ALWAYS CHECK THE COMPILATION ERRORS!** They are usually quite self-explanatory.
4. **ALWAYS DEBUG AND CHECK THE ERRORS IN LOGCAT!** Read the usually self-explanatory errors and 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 UI Controllers and View Models

The ViewModel class is designed to store and manage the application data in a lifecycle-conscious way. Using ViewModels, it is possible to separate the management of the UI (still performed in Composables and Activities) and the one of the application logic, encapsulated in ViewModels objects.

In this section, we will focus on the mobile module. We will separate the UI controller and ViewModels for the Screen composables that we have already implemented.

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

First, we need to add the proper dependencies. Go to the gradle file of the mobile module, add the following lines, and synch the project:

```
implementation("androidx.compose.runtime:runtime-livedata:1.5.4")
implementation("androidx.lifecycle:lifecycle-runtime-compose:2.6.2")
implementation("androidx.lifecycle:lifecycle-viewmodel-compose:2.6.2")
```

2.1 LoginProfile

Now we can create *ViewModel*s. Right now all of your data and code is in MainActivity or Screen composables. We want to move application logic and data to corresponding ViewModels, and only reference them from composables. Let's start with **LoginProfileScreen**.

Create a new Kotlin class in the **java/com.epfl.esl.sportstracker** directory and call it **LoginProfileViewModel**. In this file, create a class **LoginProfileViewModel** that extends ViewModel.

```
class LoginProfileViewModel : ViewModel() {...}
```

In this class will contain all the methods and variables which are not related directly to the UI. Go to **LoginProfileScreen.kt** and add profileLoginViewModel as the last parameter.

```
fun LoginProfileScreen(
    onEnterButtonClicked: ((LoginInfo) -> Unit),
    modifier: Modifier = Modifier,
    loginProfileViewModel: LoginProfileViewModel = viewModel()
) {
    ...
}
```

Now, you can move the **sendDataToWear()** function from the MainActivity to **LoginProfileViewModel**, since this function is not related to the UI or MainActivity. Add two arguments in the definition of this function:

```
fun sendDataToWear(context: Context?, dataClient: DataClient)
```

The first argument is the application context. The second argument is dataClient, which you will need to pass from the Activity to **LoginProfileScreen**, and from there to **sendDataToWear** inside the viewModel. Consequently, we have to modify the **LoginProfileScreen** again, in particular the function **onEnterButtonClicked** on the **LoginProfileContentDisplaying** composable.

```
fun LoginProfileScreen(
    onEnterButtonClicked: ((LoginInfo) -> Unit),
    dataClient: DataClient,
    modifier: Modifier = Modifier,
    loginProfileViewModel: LoginProfileViewModel = viewModel()
) {
    Surface(
        ...
    ) {
        val context = LocalContext.current
        ...
        LoginProfileContentDisplaying(
            ...
            onEnterButtonClicked = { loginInfo ->
                loginProfileViewModel
                    .sendDataToWear(context.applicationContext, dataClient)
                onEnterButtonClicked(loginInfo)
            }
        )
        ...
}
```

After moving the function to the ViewModel, we move three variables from the composable to the ViewModel: **`imageUri`**, **`password`**, and **`username`**. We need to declare them as private MutableLiveData variables, assign them initial values and create LiveData variables for read-only access. We also need to create functions for updating the values.

```
/** mobile -> LoginProfileViewModel.kt -> LoginProfileViewModel **/
    private var _username = MutableLiveData<String>("")
    private var _password =  MutableLiveData<String>("")
    private var _imageUri = MutableLiveData<Uri?>(null)

    val username: LiveData<String>
        get() = _username
    val password: LiveData<String>
        get() = _password
    val imageUri: LiveData<Uri?>
        get() = _imageUri
```

```
    fun updateUsername(username: String) {
        _username.postValue(username)
    }

    fun updatePassword(password: String) {
        _password.postValue(password)
    }

    fun updateImageUri(imageUri: Uri?) {
        _imageUri.postValue(imageUri)
    }

    fun sendDataToWear(context: Context?, dataClient: DataClient) { ...
```

In order to be able to update the UI and composables accordingly, we need to observe the LiveData in the **LoginProfileScreen** composable. To do so, we have to change the declaration of the variables for the username, password and image. For the username, the code is:

```
loginProfileViewModel.username.observeAsState(initial = "")}
```

We also have to update the update functions for username and password, e.g. for **onUsernameChanged**:

```
onUsernameChanged = { newValue ->
     loginProfileViewModel.updateUsername(newValue) }
```

As for the image, we have to call **loginProfileViewModel.updateImageUri(uri)** in the **onResult()** callback.

In the part of the **sendDataToWear** function in which imageBitmap is defined, you must change first from **this.contentResolver** to **context?.contentResolver**. The declaration of imageBitmap should look as follows, and similarly for the username:

```
var imageBitmap = MediaStore.Images.Media.getBitmap(
    context?.contentResolver,
    _imageUri.value
)
```

We are almost there! In **PutDataRequest**, there are still some errors related to **username**, as this should be changed to **_username.value**. We should also provide a default in case the value is null: **_username.value?: ""**.

Then, **LoginProfileScreen**, in the **onLogOutButtonClicked()** function, use the viewModel's update functions to re-initialise username, password and the image URI.

2.2 NewRecording

Let's go to another Screen: **NewRecordingScreen**. Create a ViewModel for this Screen and call it **NewRecordingViewModel.kt**. Don't forget to extend **ViewModel**.

Move **enum class SPORT {RUNNING, CYCLING, SKIING, CLIMBING}** to the view model file, outside the class. Move also the selectedSport variable, and change it into a private MutableLiveData called **_selectedSport**. Again, create an corresponding LiveData variable with an associated getter, and an update function **updateSelectedSport(sport: SPORT){..}**.

Add the view model as a parameter the **NewRecordingScreen** composable and modify the composable to work with the view model.

```
@Composable
fun NewRecordingScreen(
    username: String,
    imageUri: Uri?,
    onLogoutClicked: () -> Unit,
    modifier: Modifier = Modifier,
    newRecordingViewModel: NewRecordingViewModel = viewModel()
) {
    val selectedSport by newRecordingViewModel.selectedSport
                        .observeAsState(initial = SPORT.NO_SPORT)
    ...
}
```

Then, use **newRecordingViewModel.updateSelectedSport()** in this Screen to update the selected sport.

Launch the app to see the ViewModels that you just created in action. Note that in this step, we are not going to see any difference in the appearance of the app. ViewModels helped us to organize the code, making the app more modular. In the following sections, we will focus on **ExerciseLiveFragment**, highlighting the benefit of employing ViewModels.

3 Reading and displaying heart rate sensor data

Most Android-powered devices have built-in sensors that measure motion, orientation, and various environmental conditions. Usually, Android smartwatches also provide a heart rate sensor.

To track the user sport activity, in our app we want to:

- read the HR sensor
- show the data in a simple layout on the watch
- send the HR to the tablet
- show the HR data in **mobile -> ExerciseLiveFragment**

Let's go in the details of how to implement these steps!

3.1 Permissions

You will need to first add the **WAKE_LOCK** permission to allow the application to keep the watch awake while it's running. This is done by adding the following line in the **AndroidManifest** of the wear module:

```
<!-- wear -> AndroidManifest.xml -->
<uses-permission android:name = "android.permission.WAKE_LOCK"/>
```

Furthermore, in order to allow an application to access data from bio-sensors such as the heart rate one, you need to add to **AndroidManifest** the following line:

```
<!-- wear -> AndroidManifest.xml -->
<uses-permission android:name = "android.permission.BODY_SENSORS"/>
```

Thu user can grant or deny the app request to use the HR data. The following code manages this step:

```
/** wear -> MainActivity.kt -> onCreate(...) **/
if (Build.VERSION.SDK_INT >= Build.VERSION_CODES.M
    && checkSelfPermission("android.permission.BODY_SENSORS")
            == PackageManager.PERMISSION_DENIED) {
    requestPermissions(arrayOf("android.permission.BODY_SENSORS"), 0)
}
```

Feel free to implement a part of the code in case the user denies the HR sensor permissions to the app.

3.2 Preliminary layouts of Wear activity and ExcerciseLiveScreen

For this lab, we add a simple **Text** to the **MainActivity** on the *wear* module. This text view will show the current value of the user's heart rate. For now, on the tablet, we also employ only one **Text** in **ExerciseLiveScreen** that shows the HR value sent from the watch. We will implement a more complete functionality for the Screen of the mobile module later in the lab.

3.3 Read data from the HR sensor

To monitor raw sensor data you need to implement two callback methods that are exposed through the **SensorEventListener** interface: **onAccuracyChanged(...)** and **onSensorChanged()**. The Android system calls these methods whenever the following occurs:

- *A sensor's accuracy changes:* in this case the system invokes the **onAccuracyChanged(...)** method, providing you with a reference to the **Sensor** object that changed and the new accuracy of the sensor. In this lab, we will not use it so it will be left blank.
- *A sensor reports a new value:* in this case the system invokes the **onSensorChanged(...)** method, providing you with a **SensorEvent** object. A **SensorEvent** object contains information about the new sensor data, including: the accuracy of the data, the sensor that generated the data, the timestamp at which the data was generated, and the new data that the sensor recorded.

First, make sure the **MainActivity** implements **SensorEventListener**. To do that, you need to add **SensorEventListener** to the class definition as follows:

```
MainActivity :  ComponentActivity(), SensorEventListener,
                DataClient.OnDataChangedListener{
                    ...
```

After adding the **SensorEventListener** interface, Android Studio suggests to implement the *onSensorChanged()* and *onAccuracyChanged()* callbacks. Do so (otherwise your project won't compile), but leave them blank for now.

Now, in the *onCreate()* function we call the **SensorManager** system service to be able to register the listener (which is in the current activity: **this**).

```
/** wear -> MainActivity.kt -> onCreate(...) **/
mSensorManager = getSystemService(SENSOR_SERVICE) as SensorManager
mHeartRateSensor = mSensorManager.getDefaultSensor(Sensor.TYPE_HEART_RATE)
```

Both **mSensorManager** and **mHeartRateSensor** should be declared as lateinit class attributes (i.e., outside **onCreate()**).

When the listener is registered on the heart rate sensor, it listens to events with a rate suitable to the user interface. We register the listener in the *onResume()*:

```
/** wear -> MainActivity.kt -> onResume(...) **/
mSensorManager.registerListener(this, mHeartRateSensor,
                                SensorManager.SENSOR_DELAY_UI)
```

We also have to unregister the sensor when the app pauses:

```
/** wear -> MainActivity.kt -> onPause(...) **/
mSensorManager.unregisterListener(this)
```

When a new value is retrieved by the registered sensor, the *onSensorChanged(...)* callback is executed. Complete its implementation as follows:

```
/** wear -> MainActivity.kt **/
    private var heartRate by mutableStateOf<Int>(0)
    ...
    override fun onSensorChanged(event: SensorEvent?) {
        heartRate = event?.values?.get(0)?.toInt() ?: 0
    }
```

You will have to add this dependency on wear module build.gradle avoid an error when declaring the *heartRate* variable:

```
implementation("androidx.compose.runtime:runtime-livedata:1.5.4")
```

The variable **heartRate** (which you have to decaler as a state variable of the Activity class as a **mutableIntStateOf(0)**) is updated whenever the sensor changes its value. Note that we take the value from the sensor event with **event?.values?.get(0)**, as some sensors can return multi-dimensional values (e.g. 3-axes accelerometry). For more information about how to get the sensor value for different types of sensors check the *Android Developer* documentation[2].

To show the heart rate in the screen of the watch, add a **Text** to the **Homescreen** composable, for example at the bottom of the ConstraintLayout, and pass to it the heartRate value.

[2] https://developer.android.com/reference/android/hardware/SensorEvent

If you run your application now, the value of your heart rate should be shown in the **TextView** that you use for heart rate measurements and will change whenever the sensor value changes. Figure 1 shows the resulting smartwatch GUI.

Figure 1: HR data on smart watch

3.4 Sending HR from watch to tablet

In the **onSensorChanged(...)** of the **MainActivity** after reading the sensor we must send the data to the mobile app. To this end, create a **sendDataToMobile(...)** and call it inside **onSensorChanged(...)**.

```
/** wear -> MainActivity.kt **/
private fun sendDataToMobile(heartRate: Int) {
    val dataClient: DataClient = Wearable.getDataClient(this)
    val putDataReq: PutDataRequest = PutDataMapRequest.create("/heart_rate")
            .run {
                dataMap.putInt("HEART_RATE", heartRate)
                asPutDataRequest()
            }
    dataClient.putDataItem(putDataReq)
}
```

3.5 Showing HR in ExerciseLiveScreen

In this section, we need to get the heart rate in the **ExerciseLiveScreen** to be able to show it to the user. The implementation is similar to the one in *Lab4* when we sent the profile from tablet to watch, but now the communication is from watch to tablet.

Therefore, you need to implement **DataClient.OnDataChangedListener** in the definition of the MainActivity and add **onDataChanged** function in the activity. Inside this function, you change the heart rate state variable and send it to the **ExerciseLiveScreen** as a parameter based on the event received in the dataEventBuffer.

```
class MainActivity : ComponentActivity(), DataClient.OnDataChangedListener {

    private lateinit var dataClient: DataClient
    private var heartRate by mutableStateOf(0)

    ...

    override fun onDataChanged(dataEvents: DataEventBuffer) {
        dataEvents
            .filter { it.type == DataEvent.TYPE_CHANGED &&
                      it.dataItem.uri.path == "/heart_rate" }
            .forEach { event ->
                heartRate = DataMapItem.fromDataItem(event.dataItem)
                    .dataMap.getInt("HEART_RATE")
            }
    }
}
```

Where **"/heart_rate"** and **"HEART_RATE"** denote the URI path and key, equal to the ones defined in Section 3.4 for the wear module.

The next step is to register the data listener, again in **MainActivity**. Override an **onResume** function and add the following code to it:

```
dataClient.addListener(this)
```

For unregistering the listener do the same way to remove the listener in an override **onP⌋ause** function.

```
dataClient.removeListener(this)
```

Finally, pass the heartrate as a parameter to **ExerciseLiveScreen**, where you can display in a **Text**. Now, if you run the app, you will see the HR value in both the watch and tablet.

3.6 Start and stop sending HR

You might have noticed that the code in Section 3.4 causes the watch to continuously send heart rate data whenever it receives a value in the **onSensorChanged**. Sending data even when it is not processed by the tablet is not efficient from an energy perspective. To address this problem, we establish a procedure in which the tablet sends commands to the watch to start and stop sending heart rate data.

On the mobile side, we need to send commands to the watch. We have already sent profile information from the tablet to watch using the **DataClient** API. Now, we use the **MessageClient** API instead. Add the following code to **MainActivity**:

```
/** mobile -> MainActivity **/
 private fun sendCommandToWear(command: String) {
    Thread(Runnable {
        val connectedNodes: List<String> = Tasks
            .await(
                Wearable
                    .getNodeClient(this).connectedNodes
            )
            .map { it.id }
        connectedNodes.forEach {
            val messageClient: MessageClient = Wearable
                .getMessageClient(this)
            messageClient.sendMessage(it, "/command", command.toByteArray())
        }
    }).start()
}
```

In this function, first, we get all the nodes connected to the tablet, and then, we send the **command** to the tablet. The **command** argument can be **Start** or **Stop**, like:

```
sendCommandToWear("Stop")
```

Therefore, we call this function in **onResume()** and **onPause()** to start, and stop the data communication between the watch and tablet. Note that we put the code that initiates a seach for connected nodes in a **Tasks.await()** clause, so that the GUI does not become unresponsive in the meantime.

Now, in the wear, we need to receive the commands. Go to **MainActivity.kt** in the **wear** module and add the interface for the message listener:

```
/** wear -> MainActivity.kt **/
class MainActivity : ComponentActivity(), DataClient.OnDataChangedListener,
                                    SensorEventListener,
                                    MessageClient.OnMessageReceivedListener {
        ...
    }
```

An error should be raised because we need to override the listener in our code, as well. Thus, add the following function to the class. We will fill it in the next section.

```
/** wear -> MainActivity.kt **/
override fun onMessageReceived(messageEvent: MessageEvent) {

}
```

To activate the message receiver, we need to register the listener. In **onResume()**:

```
/** wear -> MainActivity.kt/onResume()**/
Wearable.getMessageClient(this).addListener(this)
```

And don't forget to remove the listener when the Activity goes in background.

```
/** wear -> MainActivity.kt/onPause() **/
Wearable.getMessageClient(this).removeListener(this)
```

3.7 Timer

The app suffers from another problem: the heart rate data is sent immediately after it is ready. Therefore, we may receive HR data with different frequencies from time to time. We need to use *Timers* to send this HR data more regularly.

First, we define a timer variable. The previously defined **heartRate** activity-level variable is updated whenever the heart rate is received from the sensor, but we only send it to the tablet at every tick of the timer. Add the following variable to the **MainActivity.kt**.

```
/** wear -> MainActivity.kt **/
private var timer = Timer()
```

To launch this timer, we set the timer schedule when we receive the "Start" command from the tablet. Therefore, update the **onMessageReceived** function, to set and cancel the timer based on the command we received from the tablet.

```
/** wear -> MainActivity.kt -> onMessageReceived **/
override fun onMessageReceived(messageEvent: MessageEvent) {
    if(messageEvent.path == "/command") {
        val receivedCommand: String = String(messageEvent.data)
        if (receivedCommand == "Start") {
            timer = Timer()
            timer.schedule(timerTask {
                sendDataToMobile(heartRate)
            }, 0, 500)
        } else if (receivedCommand == "Stop") {
            timer.cancel()
        }
    }
}
```

Remember to remover the call to **sendDataToMobile** from the **onSensorChanged** function.

4 ExerciseLiveViewModel

As mentioned in Section 2, we'd like to separate GUI and application logic concerns, using UI controllers and View Models. In this section, we will implement a ViewModel for ExerciseLive, as this will become more complex in the following

First, add a new file **ExerciseLiveViewModel.kt**. Then, remove the **DataClient.OnDataChangedListener** from the activity definition and add it to **ExerciseLiveViewModel**. Move the **onDataChanged** function from the fragment to the ViewModel.

Note that as we do not have access to the UI from the ViewModel, the ViewModel can't directly display the received HR value. Instead, the heart rate TextView should be updated using *Live Data* and *Observers*. To this end, define a private MutableLiveData variable in the ViewModel for the heart rate. Then, define a corresponding LiveData variable. The code should be as follows:

```
/** mobile -> ExerciseLiveViewModel.kt **/
private val _heartRate = MutableLiveData<Int>(0)
val heartRate: LiveData<Int>
       get() = _heartRate
```

In **onDataChanged**, update the value of **_heartRate** as the end of the function as follows.

```
/** mobile -> ExerciseLiveViewModel.kt -> onDataChanged()**/
_heartRate.value =
       DataMapItem.fromDataItem(event.dataItem).dataMap.getInt("HEART_RATE")
```

Now, go to **MainActivity.kt** and initialize the viewModel.

```
/** mobile -> MainActivity.kt **/
    private val exerciseLiveViewModel: ExerciseLiveViewModel by viewModels()
})
```

We now need to provide the view model as a parameter to the **ExerciseLiveScreen**, where we can observe the heart rate as a state.

```
fun ExerciseLiveScreen(exerciseLiveViewModel: ExerciseLiveViewModel,
            modifier: Modifier = Modifier) {
    val heartRate by exerciseLiveViewModel.heartRate.observeAsState(initial = 0)
    ...
}
```

Note that you need to change the **onResume** and **onPause** functions as well, because there is no **onDataChange** in this activity anymore. Instead, you need to set the *viewModel* when you register or unregister the listener. For instance, to register the listener in **onResume**:

```
dataClient.addListener(exerciseLiveViewModel)
```

Now you can launch the app to see the changes in reading and displaying the heart rate.

4.1 Refactor the lifecycles

Since we moved the **DataClient.OnDataChangedListener** we can refactor the code to be simpler. We move some of the logic from the **MainActivity** to the **ExerciseLiveScreen**, since it only refers to that screen. To do this, we will need to use composable lifecycle callbacks. Let's start by adding the import to the build.gradle file.

```
/** mobile -> build.gradle **/
    implementation("androidx.lifecycle:lifecycle-runtime-compose:2.7.0-rc02")
    implementation("androidx.lifecycle:lifecycle-runtime-ktx:2.7.0-rc02")
```

Now, we can use the **LifecycleResumeEffect** composable to listen for **onResume** and **onPause** events. Remove the **onResume** and **onPause** methods from the **MainActivity** and implement the same logic in the **ExerciseLiveScreen**. Move the **sendCommandToWear** function to the ExerciseLiveScreen.kt file, but outside of the ExerciseLiveScreen composable. Add a Context parameter to the function and change **this** to **context** inside the function.

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

@Composable
fun ExerciseLiveScreen(
    ...
) {
...
}

fun sendCommandToWear(command: String, context: Context) {
...
}
```

Then we can create a **LifecycleResumeEffect** and paste the appropriate method. Likewise, you can instead decide to use **LifecycleEventEffect(Lifecycle.Event.ON_RESUME)** and **LifecycleEventEffect(Lifecycle.Event.ON_PAUSE)**.

```
@Composable
fun ExerciseLiveScreen(
    dataClient: DataClient,
    modifier: Modifier = Modifier,
    exerciseLiveViewModel: ExerciseLiveViewModel = viewModel()
) {
    ...
    val context = LocalContext.current

    LifecycleResumeEffect {
        dataClient.addListener(exerciseLiveViewModel)
        sendCommandToWear("Start", context)

        onPauseOrDispose {
            dataClient.removeListener(exerciseLiveViewModel)
            sendCommandToWear("Stop", context)
        }
```

```
    }
    ...
}
```

We can now remove the ExerciseLiveViewModel instance in MainActivity since we moved all of the logic to the ExerciseLiveScreen and we are instantiating the view model in the constructor. The only thing left is to pass dataClient to the ExerciseLiveScreen instantiation in the NavHost. You can also move the **sendCommandToWear** to the excerciseLive viewModel, since it is not part of the GUI management.

Now launch the application and check if everything is working as expected.

5 Show HR in AndroidPlot

To improve the look-and-feel of our sports tracker app, we will show the HR data in a live plot. For this, we need to add the plot in the layout of **ExerciseLiveScreen** and then draw it every time we receive new HR data.

5.1 Adding YCharts and configure it

We will use the **YCharts** library, an open-source third-library that will plot our data.

```
/** mobile --> build.gradle **/
dependencies {
    implementation("co.yml:ycharts:2.1.0")
    ...
}
```

Since the library uses a higher minSdkVersion, we will have to override it in our AndroidManifest file.

```
/** mobile -> manifests -> AndroidManifest.xml **/
    <uses-sdk android:targetSdkVersion="33" android:minSdkVersion="23"
        tools:overrideLibrary="co.yml.charts.components"/>
```

5.2 Create a list of Points

In order to use the library, we will have to prepare the data. The graph employs a list of **Point**s as input to plot the data. We will add two new variables **_heartRateList**, which

will be used as a helper, and **_heartRateListLiveData**, which will update the composable about the changes. In the **onDataChanged** method we will create a new **Point** object, add it to the **_heartRateList**, and assign the new array to the value of **_heartRateListLiveData**.

```
/** mobile -> ExerciseLiveViewModel **/
private val _heartRateList = ArrayList<Point>()
private val _heartRateListLiveData = MutableLiveData<List<Point>>()

val heartRateList: LiveData<List<Point>>
    get() = _heartRateListLiveData

override fun onDataChanged(dataEvents: DataEventBuffer) {
    dataEvents
        .filter { it.type == DataEvent.TYPE_CHANGED
            && it.dataItem.uri.path == "/heart_rate" }
        .forEach { event ->
            val newValue = DataMapItem.fromDataItem(event.dataItem)
                .dataMap.getInt("HEART_RATE")

            _heartRate.value = newValue

            val x = _heartRateList.size
            val newPoint = Point(x.toFloat(), newValue.toFloat())
            _heartRateList.add(newPoint)

            _heartRateListLiveData.value = _heartRateList
        }
}
```

5.3 Draw HR data on the plot

In the **ExerciseLiveScreen** we have to observe the new list that we created. We have to change the Row into a Column and add the **LineChart** composable.

```
/** mobile -> ExerciseLiveScreen.kt  **/
    val pointsData by exerciseLiveViewModel.heartRateList
                        .observeAsState(initial = listOf())
```

```
Column(
        modifier = modifier.fillMaxSize(),
        verticalArrangement = Arrangement.Center,
        horizontalAlignment = Alignment.CenterHorizontally
    ) {
        Text(text = stringResource(R.string.heart_rate, heartRate))

         LineChart(
            modifier = Modifier
                .fillMaxWidth()
                .height(300.dp),
            lineChartData = lineChartData
        )
    }
```

LineChart is expecting a parameter lineChartData, so let's create a 'LineChardData object.

```
    val lineChartData = LineChartData(
        linePlotData = LinePlotData(
            lines = listOf(
                Line(
                    dataPoints = pointsData,
                    LineStyle(),
                    IntersectionPoint(),
                    SelectionHighlightPoint(),
                    ShadowUnderLine(),
                    SelectionHighlightPopUp()
                )
            ),
        ),
        xAxisData = xAxisData,
        yAxisData = yAxisData,
        gridLines = GridLines(),
        backgroundColor = Color.White
    )
```

We already have the list with points, we need to create xAxisData and yAxisData. We want to x-axis to represent the number of steps and the y-axis to represent the HR values. To want a better visibility and readability of the graph, we will make the ranges of y-axis to

go from the minimum value to the maximum value in the list.

```
    val max = pointsData.maxOfOrNull { it.y }?.toInt() ?: 0
    val min = pointsData.minOfOrNull { it.y }?.toInt() ?: 0
    val steps = pointsData.size - 1

    val xAxisData = AxisData.Builder()
        .axisStepSize(100.dp)
        .backgroundColor(Color.Blue)
        .steps(steps)
        .axisStepSize(20.dp)
        .labelData { i ->
            i.toString()
        }
        .labelAndAxisLinePadding(15.dp)
        .build()

    val yAxisData = AxisData.Builder()
        .steps(max-min)
        .backgroundColor(Color.Red)
        .labelAndAxisLinePadding(20.dp)
        .labelData { i ->
            (i+min).toString()
        }.build()
```

The final modification is that we want to show the graph when the points data list is not empty because if it is empty we have nothing to plot. Below is the final code:

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

    Column(
        modifier = modifier.fillMaxSize(),
        verticalArrangement = Arrangement.Center,
        horizontalAlignment = Alignment.CenterHorizontally
    ) {
        Text(text = stringResource(R.string.heart_rate, heartRate))

        val max = pointsData.maxOfOrNull { it.y }?.toInt() ?: 0
        val min = pointsData.minOfOrNull { it.y }?.toInt() ?: 0
        val steps = pointsData.size - 1
```

```
if (pointsData.isNotEmpty()) {
    val xAxisData = AxisData.Builder()
        .axisStepSize(100.dp)
        .backgroundColor(Color.Blue)
        .steps(steps)
        .axisStepSize(20.dp)
        .labelData { i ->
            i.toString()
        }
        .labelAndAxisLinePadding(15.dp)
        .build()

    val yAxisData = AxisData.Builder()
        .steps(max-min)
        .backgroundColor(Color.Red)
        .labelAndAxisLinePadding(20.dp)
        .labelData { i ->
            (i+min).toString()
        }.build()

    val lineChartData = LineChartData(
        linePlotData = LinePlotData(
            lines = listOf(
                Line(
                    dataPoints = pointsData,
                    LineStyle(),
                    IntersectionPoint(),
                    SelectionHighlightPoint(),
                    ShadowUnderLine(),
                    SelectionHighlightPopUp()
                )
            ),
        ),
        xAxisData = xAxisData,
        yAxisData = yAxisData,
        gridLines = GridLines(),
        backgroundColor = Color.White
    )
```

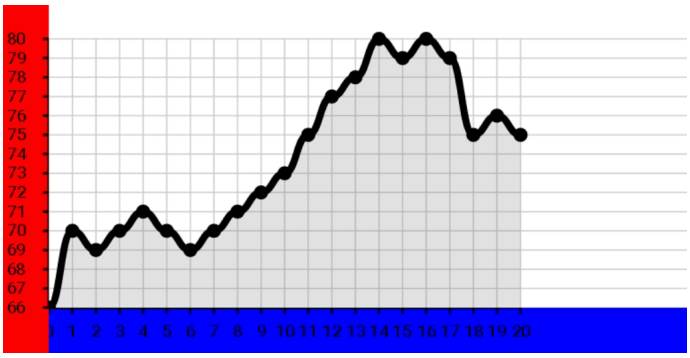

Figure 2: HR data on Android plot

```
            LineChart(
                modifier = Modifier
                    .fillMaxWidth()
                    .height(300.dp),
                lineChartData = lineChartData
            )
        }
    }
```

Run again the app. You should see the HR plot, updated from each value being acquired by the smartwatch and and sent to the tablet. the app you can see that the plot is updated at each value coming from the smartwatch HR sensor, as shown in Figure 2.

There is much more you can do with *YChart*. You can find the library documentation at this LINK[3].

[3] https://github.com/codeandtheory/YCharts