

# **Project activity: AndroidSensor**

**Alma Mater Studiorum - University of Bologna**

# Table of Contents

Introduction	3
<b>1 Initial Setup</b>	<b>3</b>
1.1 Environment setup	3
1.2 Project setup	4
1.3 Errors	9
1.4 Tracing setup	10
1.5 Debugger errors	11
<b>2 Experimenting with Android Sensor Framework</b>	<b>12</b>
2.1 Listing android sensors	12
2.2 Responding to a sensor	16
<b>3 The Android endpoint</b>	<b>20</b>
3.1 Mqtt and mosquitto setup	20
3.2 Testing mosquitto installation	20
3.3 Setting up a mqtt client on android	22
3.4 Creating our mqtt service interface	23
3.5 Testing the MyMqttClient class	25
<b>4 The QActor Endpoint</b>	<b>27</b>
4.1 Environment setup	27
4.2 Project setup	28
4.3 Receiving messages through mqtt in qactor	29
4.4 From Android to QActor	31
<b>5 The Android Application</b>	<b>33</b>
5.1 activity_main.xml	33
5.2 MainActivity.kt	34
5.3 Utils.kt	37
<b>6 Example applications</b>	<b>38</b>
6.1 mqttPlotQakEvents.ipynb	38
6.2 EventLogger	39
<b>References</b>	<b>40</b>

# Introduction

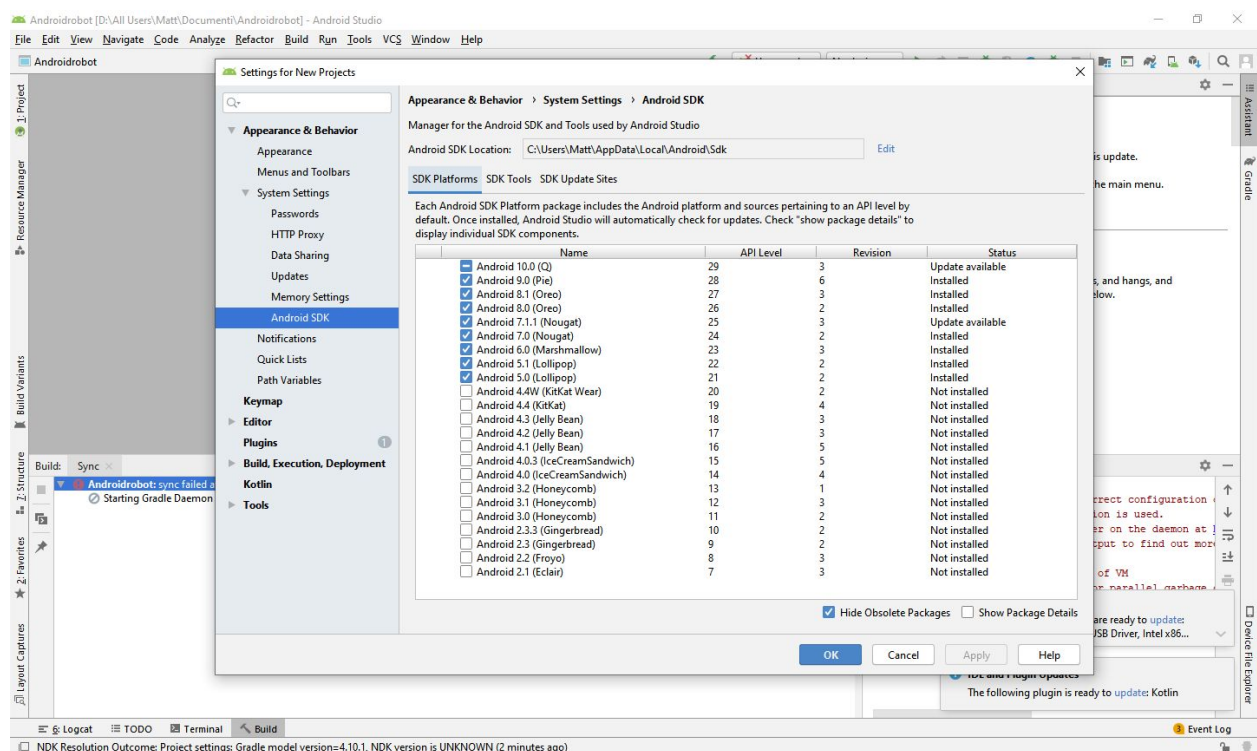
This project documents the creation of the AndroidSensor application. The application can run on any android device with a minimum API level of 24 (Nougat, android 7.0).

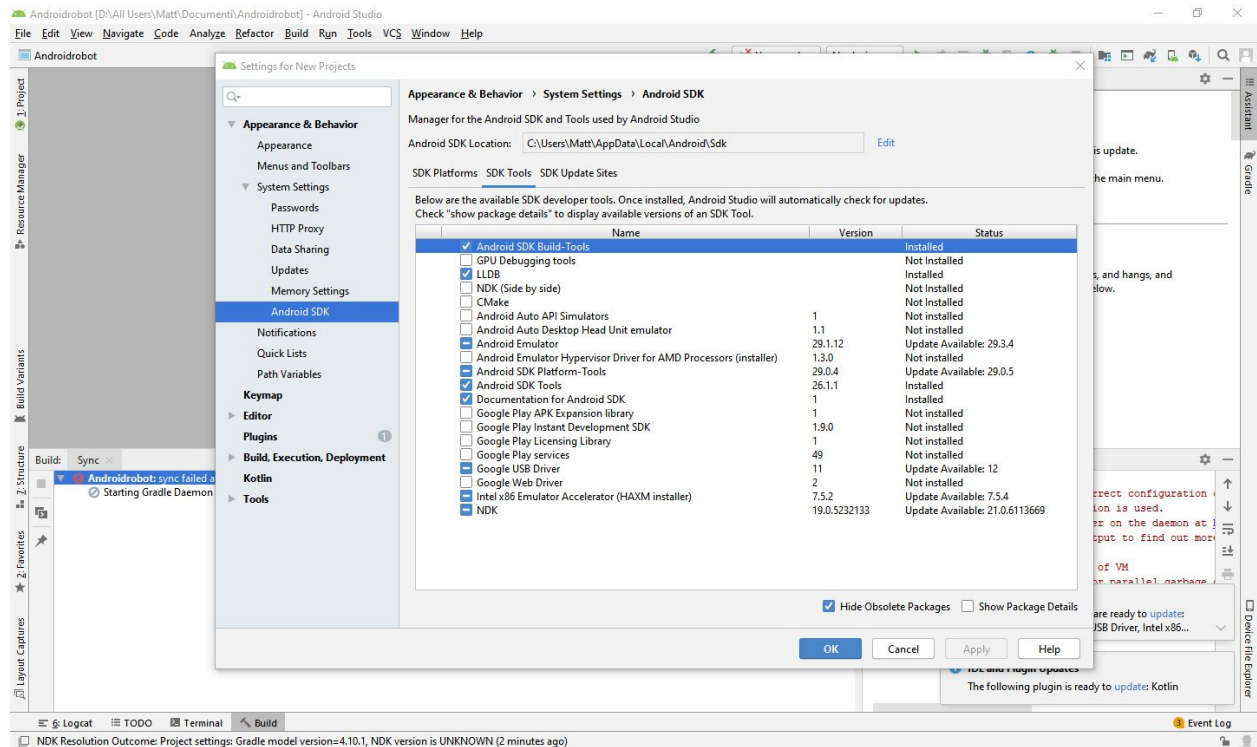
AndroidSensor provides the user with a list of sensors on the device and can connect to a mqtt broker to publish the data the sensors gather. The application allows connection to a single broker, but allows you to enable and disable the different sensors and avoid emitting data you don't intend to use.

## 1 Initial Setup

### 1.1 Environment setup

- Download Android Studio (I used v3.5).
- Under **Tools > SDK Manager > Appearance & Behaviour > System Settings > Android SDK**: make sure to have installed all versions of Android SDK Platforms from 5.0 (lollipop) to 10.0 (Q), as well as the fundamental tools:
  - Android SDK Build-Tools
  - Android SDK Tools
  - Documentation for Android SDK (Optional)
  - Google USB Driver

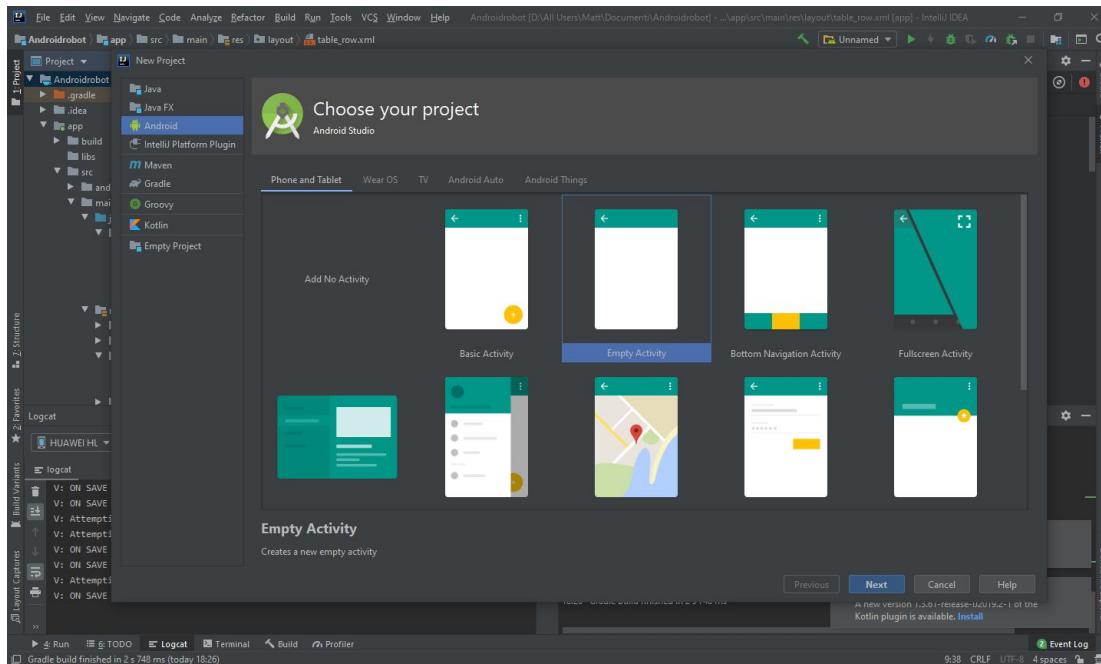




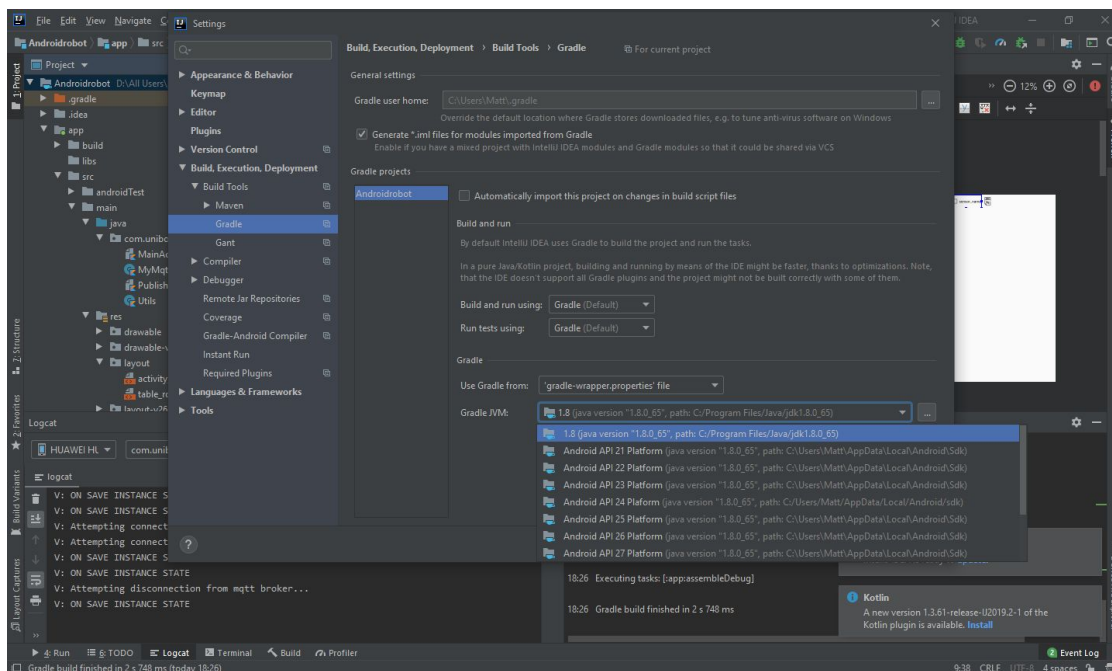
- Install adb (I used version v1.0.41). Usually, adb is included in the Android SDK Tools package.
- Install gradle (I used version v4.8.1).
- Download IntelliJ IDEA (I used v2019.2.2).
- Under **Configure > Plugins > Marketplace** or **Configure > Settings > Plugins > Marketplace** install the **Android Support** and **Kotlin** plugins. If you can't find a plugin in the marketplace, check if it's already installed.

## 1.2 Project setup

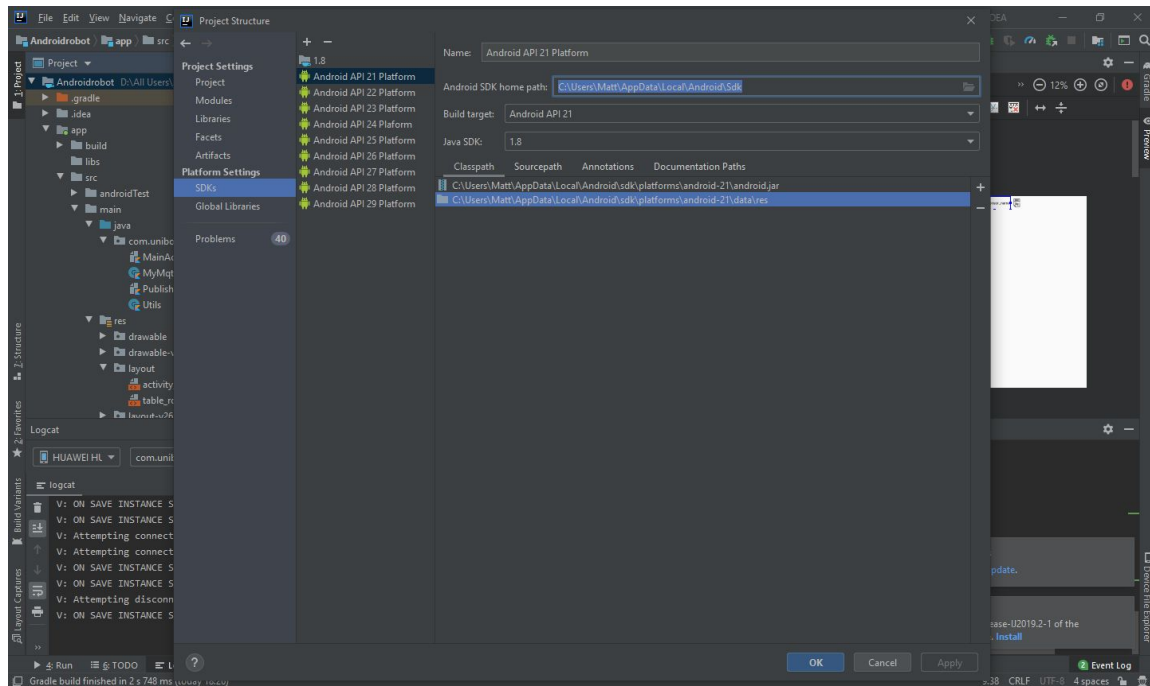
- Open IntelliJ IDEA.
- Create a new Android project. When prompted, download SDKs. Once finished, select **Phone and Tablet > Empty Activity**. Fill in all text fields, then set "Language" to Kotlin and "Minimum API level" to **API 24: Android 7.0 (Nougat)**.



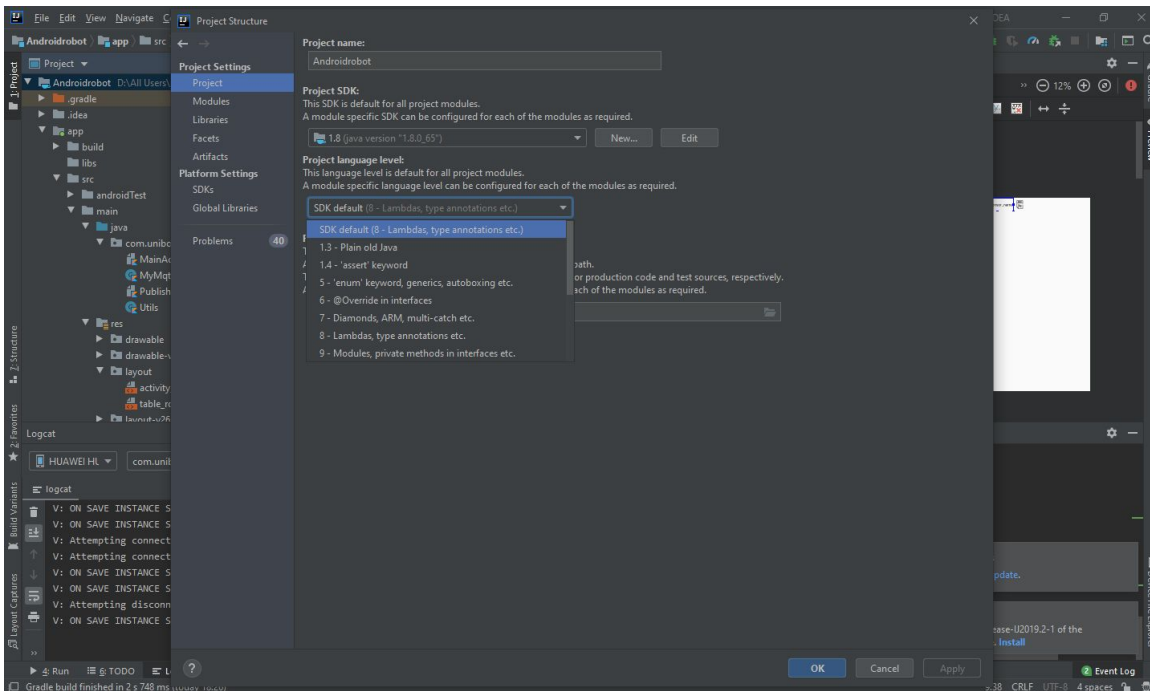
- In the new project, under **File > Settings > Build, Execution, Deployment > Build Tools > Gradle**: check that the Gradle JVM was set to the correct path to the JDK. Since I used Java v1.8.0\_65, I just had to look for the option “1.8 (java version [...])”. Make sure the path does NOT point a JRE. If a prompt pops up in the bottom right corner of the screen, then **Import Changes**.



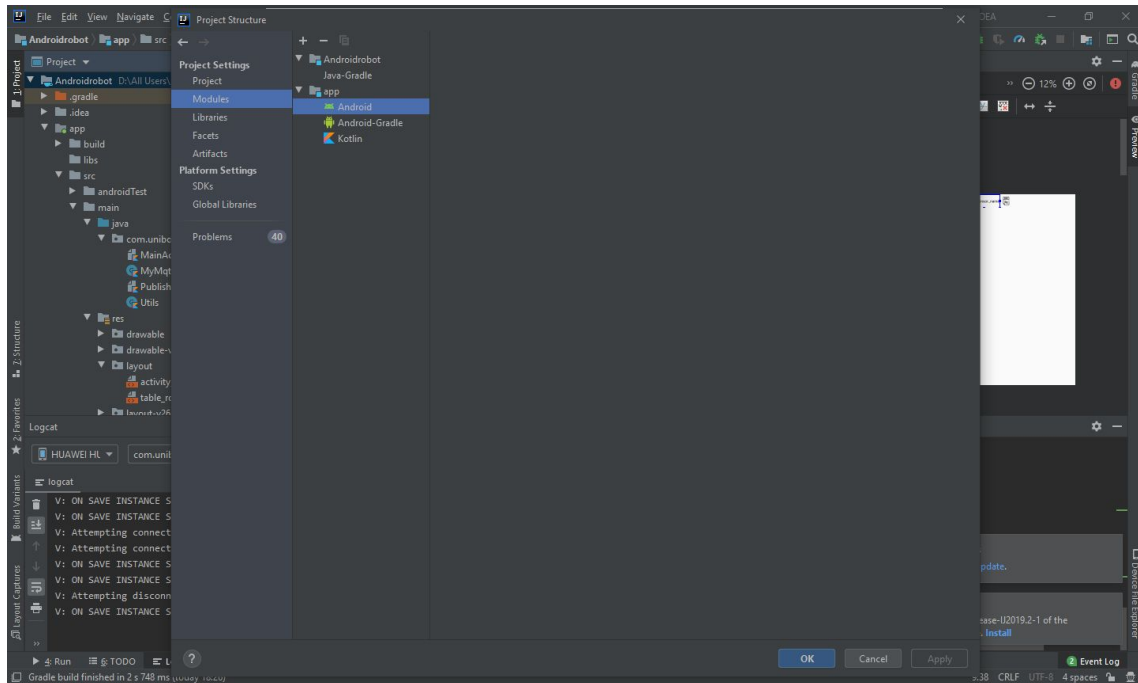
- Under **File > Project Structure > Platform Settings > SDKs**: check that the Android SDKs that were installed automatically pointed to the correct location for the Android SDK.



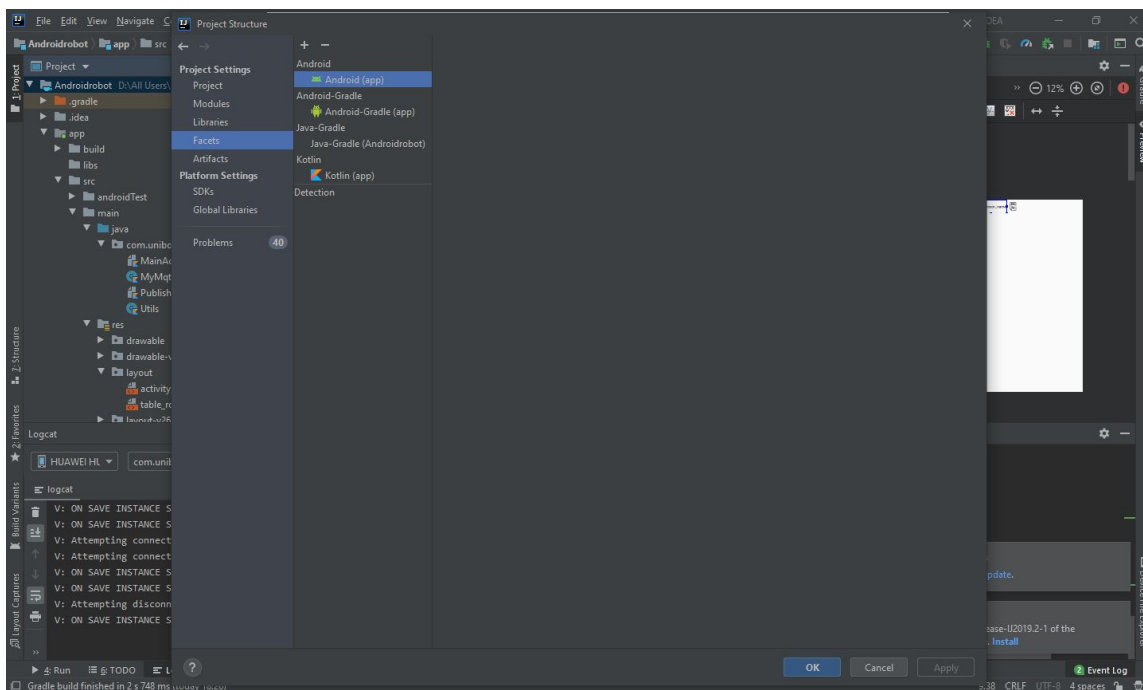
- Under **File > Project Structure > Project Settings > Project**: set Project SDK to the desired version of Android SDK. Set Project language level to SDK default.



- Under **File > Project Structure > Project Settings > Modules**: check that an **Android** module has been automatically created.

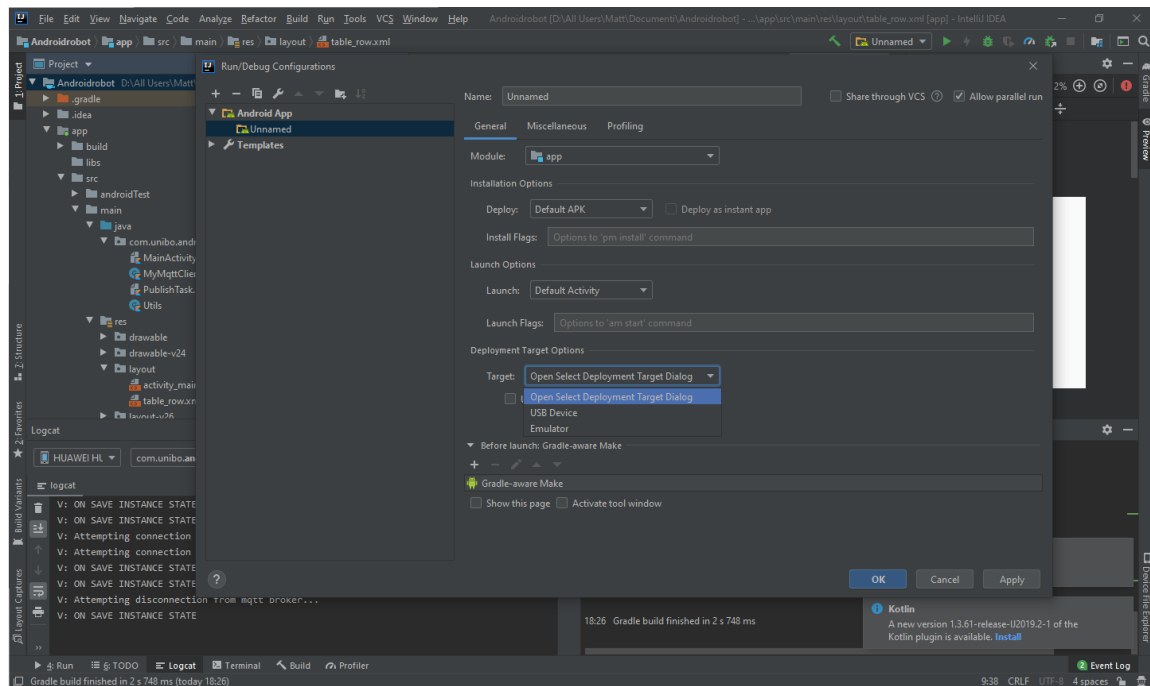


- Under **File > Project Structure > Project Settings > Facets**: check that an **Android** facet has been automatically created as well.



- If you are targeting a device with an API level of 29 or higher, migrate to AndroidX. This can be done with **Refactor > Migrate to AndroidX**.

- In the top-right corner is the debug toolbar. Click on TODO then on **Edit Configurations...** to access the **Run/Debug Configurations** window. Create a new configuration by clicking on the add icon (+), then on **Android App**. Name it, set “Module” to the application (“app”, aka the module gradle creates automatically). Since we want to run the application on an Android phone, set “Deployment Target Options” to Open Select Deployment Target Dialog. This way, when trying to run the application, IntelliJ will provide you with a list of the devices connected to the computer. We could also set it directly to USB Device.



- Finally, click on the debug icon (🐞) and select the device from the list to debug the application.



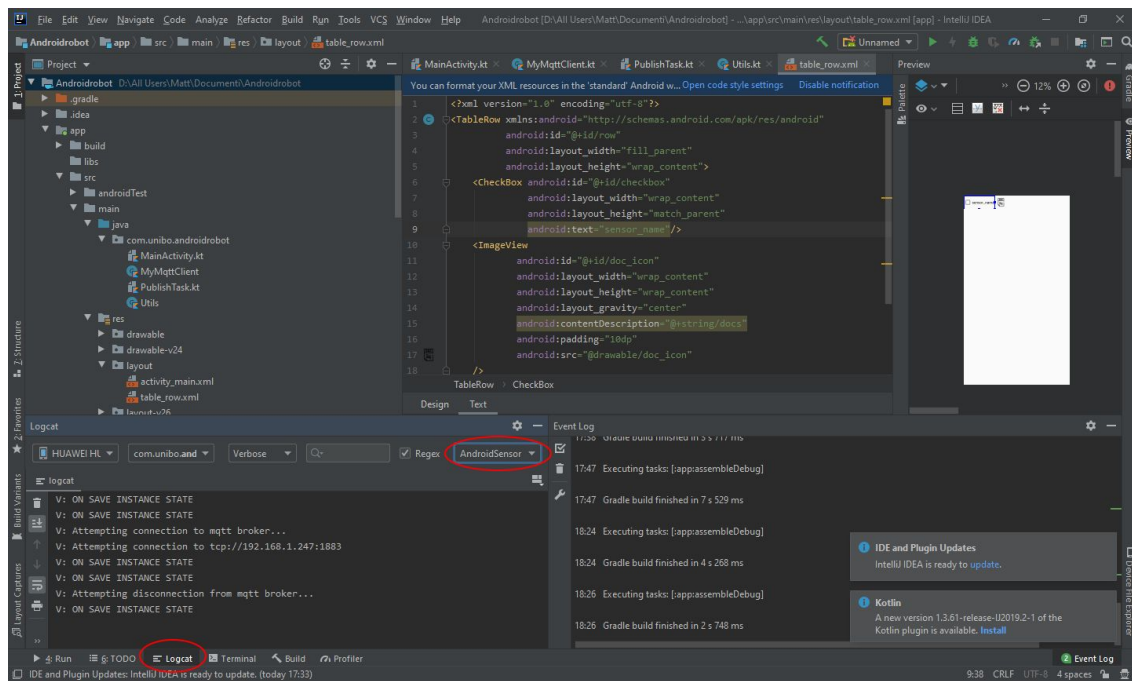
## 1.3 Errors

- If you get the error “Error: Please select Android SDK” in the [Run/Debug Configurations](#) window, then go to **File > Settings > Build, Execution, Deployment > Build Tools > Gradle** and check that the Gradle JVM is set to the version of Java you are using. Make sure the path does NOT point a JRE instead of a JDK. A popup should have appeared on the bottom-right corner prompting you to re import changes to the Gradle project. Click on [Import Changes](#).
- If no devices appear on the [Select Deployment Target](#) window when debugging, unplug and plug your USB Device back in, then pay attention to the screen: a dialog should appear with two options. Grant access. If no dialog appears at all, then you need to activate Developer Mode on your device.
- If you get the error “Error: Gradle project sync failed. Please fix your project and try again.” in the [Run/Debug Configurations](#) window, import changes from Gradle.
- If you get any IDE error, e.g. an Android plugin exception, then you’d better recreate everything from scratch.

## 1.4 Tracing setup

Efficient tracing is crucial to swift and agile application development. For this reason, we will use the app name as a tag that will characterize all logs: `AndroidSensor`.

On the bottom of the IDE there's a series of tabs: `TODO`, `Logcat`, `Terminal` and `Build`. Under the `Logcat` tab, in the upper right-hand corner, create a filter for all logs with tag "`AndroidSensor`".



Let's now create a test project as follows:

```
private const val APP_TAG = "AndroidSensor"

class MainActivity : AppCompatActivity() {

    override fun onCreate(savedInstanceState: Bundle?) {
        super.onCreate(savedInstanceState)
        setContentView(R.layout.activity_main)


        // Testing Log
        Log.e(APP_TAG, "ERROR onStart()")
        Log.w(APP_TAG, "WARN onStart()")
        Log.i(APP_TAG, "INFO onStart()")
        Log.d(APP_TAG, "DEBUG onStart()")
        Log.v(APP_TAG, "VERBOSE onStart()")
    }
}
```

If we were to run this code, we may find that some logs are missing from the logcat window, in particular the DEBUG and VERBOSE level logs. The reason is that some devices are set to cull these messages from applications, as to limit the amount of logging during run time, thus achieving better performances.

The solution is simple: we just need to edit the log settings of your device. The actual process depends on the device. On Huawei P9 Lite, dialing **\*\*\*2846579\*\*\*** will open a menu; simply go to Project Menu > Background Setting > LOG Setting and select AP LOG. This will enable all three options.

Remember to revert this change after development as it can indeed affect performances.

## 1.5 Debugger errors

Try to debug the program using the debug icon (). If the debugger gets stuck, and under the Variables view it says “Collecting data...”, then try to go to File > Settings > Build, Execution, Deployment > Debugger > Data Views > Java and untick the box next to “Enable ‘toString()’ object view”. This has been shown to slow down the debugger to a halt in some situations.

## 2 Experimenting with Android Sensor Framework

### 2.1 Listing android sensors

The first step we will take is to list all of the sensors our device currently offers. From **Android Developers > Docs > Reference > Sensor Event** [2]:

The Android platform supports three broad categories of sensors:

- **Motion sensors**  
These sensors measure acceleration forces and rotational forces along three axes. This category includes accelerometers, gravity sensors, gyroscopes, and rotational vector sensors.
- **Environmental sensors**  
These sensors measure various environmental parameters, such as ambient air temperature and pressure, illumination, and humidity. This category includes barometers, photometers, and thermometers.
- **Position sensors**  
These sensors measure the physical position of a device. This category includes orientation sensors and magnetometers.

Though this classification is complete, the classification of sensors offered by the API is more refined. For instance, motion sensors include accelerometers, gyroscopes, and more. For the full list of sensor types, please refer to the Android Developers Documentation.

Android APIs manage access to resources through what they call “services”. To access sensors, we will need to ask the `SENSOR_SERVICE`. The following code lists all sensors available and prints basic information:

```
private const val TAPP_TAG = "AndroidSensor"

class MainActivity : AppCompatActivity() {

    private lateinit var mSensorManager: SensorManager

    override fun onCreate(savedInstanceState: Bundle?) {
        super.onCreate(savedInstanceState)
        setContentView(R.layout.activity_main)

        // Retrieve sensor manager
        mSensorManager = getSystemService(Context.SENSOR_SERVICE) as
SensorManager
        // Ask the manager for all sensors of the device
        val sensors: List<Sensor> =
mSensorManager.getSensorList(Sensor.TYPE_ALL)
        Log.v(TAPP_TAG, "N of sensors: " + sensors.size)
        sensors.forEach {
            Log.v(TAPP_TAG, "- Sensor data: $it")
        }
    }
}
```

The resulting output (log headers have been removed for simplicity) is the following:

N of sensors: 11

```
- Sensor data: {Sensor name="accelerometer-lis3dh", vendor="STMicroelectronics", version=1,
type=1, maxRange=78.4532, resolution=0.009576807, power=0.23, minDelay=5000}

- Sensor data: {Sensor name="mag-akm09911", vendor="akm", version=1, type=2,
maxRange=2000.0, resolution=0.0625, power=6.8, minDelay=10000}

- Sensor data: {Sensor name="orientation", vendor="huawei", version=1, type=3,
maxRange=360.0, resolution=0.1, power=13.0, minDelay=10000}

- Sensor data: {Sensor name="light-bh1745", vendor="rohmm", version=1, type=5,
maxRange=10000.0, resolution=1.0, power=0.75, minDelay=0}

- Sensor data: {Sensor name="proximity-pa224", vendor="txc", version=1, type=8,
maxRange=5.0, resolution=5.0, power=0.75, minDelay=0}

- Sensor data: {Sensor name="gravity", vendor="huawei", version=1, type=9,
maxRange=9.80665, resolution=0.15328126, power=0.2, minDelay=10000}

- Sensor data: {Sensor name="linear Acceleration", vendor="huawei", version=1, type=10,
maxRange=78.4532, resolution=0.009576807, power=0.2, minDelay=10000}

- Sensor data: {Sensor name="rotation Vector", vendor="huawei", version=1, type=11,
maxRange=1.0, resolution=5.9604645E-8, power=6.1, minDelay=10000}

- Sensor data: {Sensor name="significant Motion", vendor="huawei", version=1, type=17,
maxRange=2.14748365E9, resolution=1.0, power=0.23, minDelay=-1}

- Sensor data: {Sensor name="step counter", vendor="huawei", version=1, type=19,
maxRange=2.14748365E9, resolution=1.0, power=0.23, minDelay=0}

- Sensor data: {Sensor name="geomagnetic Rotation Vector", vendor="huawei", version=1,
type=20, maxRange=1.0, resolution=5.9604645E-8, power=6.1, minDelay=10000}
```

Let's ignore for now all information about vendor, range, resolution, power and delay, and arrange the data we have in a table to more easily parse it:

Name	Type Id	Type
accelerometer-lis3dh	1	TYPE_ACCELEROMETER
mag-akm09911	2	TYPE_MAGNETIC_FIELD
orientation	3	TYPE_ORIENTATION
light-bh1745	5	TYPE_LIGHT
proximity-pa224	8	TYPE_PROXIMITY
gravity	9	TYPE_GRAVITY
linear Acceleration	10	TYPE_LINEAR_ACCELERATION
rotation Vector	11	TYPE_ROTATION_VECTOR
significant Motion	17	TYPE_SIGNIFICANT_MOTION
step counter	19	TYPE_STEP_COUNTER
geomagnetic Rotation Vector	20	TYPE_GEOMAGNETIC_ROTATION_VECTOR

## 2.2 Responding to a sensor

Let's now try to program an app that does something when it receives an input from a sensor. To do so, we need to retrieve the sensor from the sensor service, implement the `SensorEventListener` and finally register to the sensor manager.

We already know how to do the first step: we just need to ask the sensor service for a sensor of the desired type. In this example, we will use the light sensor.

```
mLight = mSensorManager.getDefaultSensor(Sensor.TYPE_LIGHT)
```

Then, we implement the two methods from the `SensorEventListener` interface.

The first method, `onSensorChanged`, is a callback for when the value observed by a sensor changes. There are two different behaviours:

- If the `getMinDelay()` method of a sensor returns a **nonzero value**, then the sensor will take at least that amount of microseconds before generating a new `SensorEvent`. Most sensors are of this kind.
- If the `getMinDelay()` method of a sensor returns **zero**, then the sensor will generate a new `SensorEvent` only when the observed value changes.

The second method, `onAccuracyChanged`, is a callback for when the accuracy of the sensor changes. This method is used when the data needs to meet particular accuracy needs in order to be useful. We won't be using this, but of course we need to implement this method as well.

For the purpose of this test, we will be using a single sensor, light-bh1745. and log the current level of light.

Finally, we need to register the listener to the sensor manager. In order to do this, we could just call the following method in the `onCreate` method of the main activity class:

```
mSensorManager.registerListener(<listener>, <sensor>,<delay ceiling>)
```



Where,

`<listener>`

The listener whose callback are going to be registered

`<sensor>`

The sensor whose events will trigger the callbacks of the listener

`<delay ceiling>`

The maximum delay between two sensor events. The delay that you specify is only a suggested delay. The Android system and other applications can alter this delay. As a best practice, you should specify the largest delay that you can because the system typically uses a smaller delay than the one you specify (that is, you should choose the slowest sampling rate that still meets the needs of your application). Using a larger delay imposes a lower load on the processor and therefore uses less power.

Although this is functional, this is **not** a correct practice. The reason is: Android by default does not disable sensors when the application loses focus. In fact, many applications still need to gather sensor data even while in the background.

Thus, we will instead implement the `onResume` and `onPause` callbacks to enable and disable the light sensor.

Finally, we need to access the data inside the `SensorEvent` object. The `SensorEvent` object always contains information on the sensor's type, the time-stamp, accuracy and the sensor's values, but in order to access the actual values, we need to consult the specific `SensorEvent` we are working with: the values are arranged in an array, and in order to access a specific data (e.g. the position along the x-axis) we need to know the index at which it is stored.

In our case it is simple: the values array for a sensor of type `Sensor.TYPE_LIGHT` consists in just a single value: `values[0]` contains the light intensity in SI lux units.

For the complete list of `SensorEvent` values, please refer to **Android Developers > Docs > Reference > Sensor Event** [2].

In the end, our code will look something like this:

```
private const val APP_TAG = "AndroidSensor"

class MainActivity : AppCompatActivity(), SensorEventListener {

    private lateinit var mSensorManager: SensorManager
    private var mLight: Sensor? = null

    override fun onCreate(savedInstanceState: Bundle?) {
        super.onCreate(savedInstanceState)
        setContentView(R.layout.activity_main)

        // Retrieve light sensor if any (in Huawei P9 Lite, it's
        "light-bh1745")

        mSensorManager = getSystemService(Context.SENSOR_SERVICE) as
        SensorManager
        mLight = mSensorManager.getDefaultSensor(Sensor.TYPE_LIGHT)
        Log.v(APP_TAG, "Light sensor: " + mLight?.name)
    }

    override fun onSensorChanged(event: SensorEvent?) {
        // Log light level
        val lux = event?.values?.get(0)
        Log.v(APP_TAG, "Light level: $lux")
    }

    override fun onAccuracyChanged(sensor: Sensor?, accuracy: Int) {
        Log.v(APP_TAG, "onAccuracyChanged")
    }

    override fun onResume() {
        super.onResume()
        mLight?.also { mSensorManager.registerListener(this, it,
        SensorManager.SENSOR_DELAY_NORMAL)
    }
}

    override fun onPause() {
        super.onPause()
        mSensorManager.unregisterListener(this)
    }
}
```

Let's take a look at the output for a moment:

```
V: Light sensor: light-bh1745
V: onAccuracyChanged
V: Light level: 38.0
V: Light level: 37.0
V: Light level: 37.0
V: Light level: 38.0
V: Light level: 7.0 // ----- At this point, I closed the lid of the case of the phone
V: Light level: 3.0
V: Light level: 3.0
V: Light level: 4.0
V: Light level: 3.0
```

As we can see, the measurements are not perfectly precise. We need to remember the sensors we are working with are actual physical sensors, and as such, the values they provide constantly fluctuate.

If we go back to section 2.1 and take a look at the full logs, we can see that the light sensor **light-bh1745** has a resolution of 1.0, which means that all values should be interpreted with a possible error of 1. Thus a 37.0 should be read as a  $37.0 \pm 1.0$ .

We can also see the `onAccuracyChanged` method is called once before the first `onSensorChanged` call. This is a standard behaviour, as the first measurement made sets the precision for the followings.

## 3 The Android endpoint

### 3.1 Mqtt and mosquitto setup

There are many ways we could send data from our kotlin application to the robot. One possible approach is to use the mqtt publish/subscribe messaging protocol already supported by qactor. We'll also be using mosquitto as a message broker, since it's very easy to pick up and is already part of the course of Software Systems Engineering M. There are of course other methods, such as hiveMQ, for instance.

Make sure you have installed both mqtt and mosquitto on your machine, as they are very useful for testing purposes. Follow the instructions provided in [5] to install both mosquitto and mqtt. It is also advised to configure your environment variables to enable mosquitto to run directly from cmd without the need to first locate and move to its installation folder.

### 3.2 Testing mosquitto installation

To make sure the mosquitto broker works and to get a sense of what we are working with, we will start up a mosquitto broker. Type “**mosquitto -v**” in a command prompt. This will start mosquitto in verbose mode. Verbose mode will allow us to see which port the mosquitto broker is listening on. Typically, it runs on port 1883. You can forego the verbose option once you feel confident with what you are doing, but it is recommended to use it anyways, as the output is very valuable in case something unexpected comes up.

```
D:\mosquitto>mosquitto -v
1574271573: mosquitto version 1.6.7 starting
1574271573: Using default config.
1574271573: Opening ipv6 listen socket on port 1883.
1574271573: Opening ipv4 listen socket on port 1883.
```

It is possible to create brokers specifying a different port with the -p option.

The mosquitto installation comes with two additional commands, mosquitto\_pub and mosquitto\_sub, that come in handy when testing our application. Using these two, we can easily see a message being delivered through the mqtt protocol. First, we run mosquitto\_sub and then using mosquitto\_pub we publish a message. The outcome should look like this:

```
C:\Users\Matt>mosquitto -v
1576514514: mosquitto version 1.6.7 starting
1576514514: Using default config.
1576514514: Opening ipv6 listen socket on port 1883.
1576514514: Opening ipv4 listen socket on port 1883.
1576514529: New connection from ::1 on port 1883.
1576514529: New client connected from ::1 as mosq-1TGv5lkRQ8GdaD3Cud (p2,
c1, k60).
1576514529: No will message specified.
1576514529: Sending CONNACK to mosq-1TGv5lkRQ8GdaD3Cud (0, 0)
1576514529: Received SUBSCRIBE from mosq-1TGv5lkRQ8GdaD3Cud
1576514529:      androidsensor (QoS 0)
1576514529: mosq-1TGv5lkRQ8GdaD3Cud 0 androidsensor
1576514529: Sending SUBACK to mosq-1TGv5lkRQ8GdaD3Cud
1576514557: New connection from ::1 on port 1883.
1576514557: New client connected from ::1 as mosq-CAch8uEmXy5S50rPr3 (p2,
c1, k60).
1576514557: No will message specified.
1576514557: Sending CONNACK to mosq-CAch8uEmXy5S50rPr3 (0, 0)
1576514557: Received PUBLISH from mosq-CAch8uEmXy5S50rPr3 (d0, q0, r0, m0,
'androidsensor', ... (12 bytes))
1576514557: Sending PUBLISH to mosq-1TGv5lkRQ8GdaD3Cud (d0, q0, r0, m0,
'androidsensor', ... (12 bytes))
1576514557: Received DISCONNECT from mosq-CAch8uEmXy5S50rPr3
1576514557: Client mosq-CAch8uEmXy5S50rPr3 disconnected.
```

```
C:\Users\Matt>mosquitto_sub -h localhost -t androidsensor
Hello world!
```

```
C:\Users\Matt>mosquitto_pub -h localhost -t androidsensor -m "Hello world!"
```

As you can see, the -v option allows the mosquitto broker to print a message every time it receives a request for connecting, subscribing, publishing, disconnecting, etc. Client IDs can also be set with the option -i, to avoid unreadable names such as “mosq-CAch8uEmXy5S50rPr3”. See the man page for more information (you can find it in [6]).

### 3.3 Setting up a mqtt client on android

In order to send data to a mosquitto broker, we need a library that allows us to send data through a mqtt client. Fortunately, Eclipse Paho is just that: an open-source implementation of mqtt messaging protocols. To set up Eclipse Paho's library, we'll follow the steps presented in [3, 7]:

1. Open the Gradle project file ([AndroidSensor\build.gradle](#)). In the **allprojects** section, add the maven repository for Eclipse Paho releases:

```
allprojects {
    repositories {
        google()
        jcenter()
        maven {
            url
            "https://repo.eclipse.org/content/repositories/paho-releases/"
        }
    }
}
```

2. In the gradle app file ([AndroidSensor\app\build.gradle](#)), add the dependency for Eclipse Paho Android client and Eclipse Paho Mqtt service:

```
dependencies {
    implementation
    'org.eclipse.paho:org.eclipse.paho.client.mqttv3:1.2.2'
    implementation
    'org.eclipse.paho:org.eclipse.paho.android.service:1.1.1'
}
```

3. Eclipse Paho provides the Mqtt client as a background service, so we need to include the service in the Manifest file ([AndroidSensor\app\src\main\AndroidManifest.xml](#)); for more information on how android services work, please see [4]). To do so, just include the following line inside the <application> tag.

```
<service android:name="org.eclipse.paho.android.service.MqttService"/>
```

4. Finally, the Paho Android Service needs the following permissions to work. Add the following lines inside the <manifest> tag.

```
<uses-permission android:name="android.permission.WAKE_LOCK" />
<uses-permission android:name="android.permission.INTERNET" />
<uses-permission android:name="android.permission.ACCESS_NETWORK_STATE" />
<uses-permission android:name="android.permission.READ_PHONE_STATE" />
```

### 3.4 Creating our mqtt service interface

Eclipse Paho is a very versatile library, and as many others like it, might be a bit impractical to use for simple applications. For this reason, we will be creating a class to act as an interface for the Mqtt Paho client service. This class will do a similar job as the [MqttUtils.kt](#) class provided during the course.

We'll create a kotlin class under [AndroidSensor/app/src/main/java/<your package folder>](#) to use as the interface with the service.

We'll implement the basic functionalities of

1. Connecting to a broker
2. Publishing a message to a topic
3. Disconnecting from a broker

We don't need to also create a method to subscribe or unsubscribe to a topic, as we will not use this application to receive messages from the broker.

The implementation of the MyMqttClient class is as follows:

```
import android.content.Context
import org.eclipse.paho.client.mqttv3.*
import org.eclipse.paho.client.mqttv3.MqttException
import org.eclipse.paho.client.mqttv3.MqttMessage
import java.io.UnsupportedEncodingException
import org.eclipse.paho.client.mqttv3.persist.MemoryPersistence

class MyMqttClient(private val context : Context, private val broker :
String, private val clientId : String) {

    private val client = MqttClient(broker,clientId, MemoryPersistence())

    fun connect() {
        try {
            client.connect()
        } catch (e: MqttException) {
            e.printStackTrace()
        }
    }

    fun publish(topic : String, payload : String) {
        var encodedPayload = ByteArray(0)
        try {
```

```

        encodedPayload = payload.toByteArray(charset("UTF-8"))
        val message = MqttMessage(encodedPayload)
        client.publish(topic, message)
    } catch (e: UnsupportedOperationException) {
        e.printStackTrace()
    } catch (e: MqttException) {
        e.printStackTrace()
    }
}

fun disconnect() {
    try {
        client.disconnect()
    } catch (e: MqttException) {
        e.printStackTrace()
    }
}
}
}

```

A note: the **MqttClient** class used in the code above is a synchronous interface of the otherwise asynchronous service the Paho library provides. For more complex applications, the asynchronous version might be better. In that case, use the class **MqttAndroidClient** instead. The reason we use the synchronous application here is that it's much simpler (we do not need to also set up the callbacks).



### 3.5 Testing the MyMqttClient class

Now that the mosquitto broker is set up on our machine, we'll write a small program to send a message to it. To do so, we need to make sure the android device is connected to the same network as the machine running the broker and then find the local IP of the machine. That can be easily done with the ipconfig command (on the machine the broker is located on).

```
C:\Users\Matt>ipconfig

Configurazione IP di Windows

Scheda Ethernet Connessione alla rete locale (LAN):

    Suffisso DNS specifico per connessione: *****
    Indirizzo IPv6 locale rispetto al collegamento . :
fe80::2589:cfce:80ce:a88b%16
    Indirizzo IPv4. . . . . : 192.168.1.247
    Subnet mask . . . . . : 255.255.255.0
    Gateway predefinito . . . . . : 192.168.1.1
```

The actual code is pretty streamlined thanks to the object we created in 3.2:

```
import android.hardware.SensorManager
import androidx.appcompat.app.AppCompatActivity
import android.os.Bundle
import android.util.Log

const val APP_TAG = "AndroidSensor"
private const val serverIP = "192.168.1.247"
private const val serverPort = "1883"
private const val clientId = "AndroidSensor"

class MainActivity : AppCompatActivity() {

    private lateinit var mSensorManager: SensorManager

    val broker : String = "tcp://" + serverIP + ":" + serverPort
```

```

override fun onCreate(savedInstanceState: Bundle?) {
    super.onCreate(savedInstanceState)
    setContentView(R.layout.activity_main)

    val mqttClient = MyMqttClient(
        this.applicationContext,
        broker,
        clientId
    )

    Log.d(APP_TAG, "Attempting connection...")
    mqttClient.connect()
    mqttClient.publish("androidsensor", "Hello World!")
    Log.d(APP_TAG, "Attempting disconnection...")
    mqttClient.disconnect()
}
}

```

In order:

1. Open a new command prompt and create a mosquitto broker with the command **mosquitto -v**
2. Optional. Open a second command prompt and create a mosquitto client subscribed to the topic you are publishing to. This will help you double check that everything is running smoothly, as you will see the message you published pop up in this prompt after the execution of the application. The command I used is **mosquitto\_sub -h localhost -t androidsensor**.
3. Make sure the android device is connected to the same local network as the machine the broker is running on.
4. Run the test program from IntelliJ IDEA.

If everything works as intended, the command prompt running the mosquitto broker will show an output similar to the one shown in Logcat 4.

## 4 The QActor Endpoint

### 4.1 Environment setup

The following steps are explained more in detail in [10].

- Download Eclipse DSL. Make sure you download a version dating prior to July 2019, as the Kotlin plugin we'll need has issues with the most recent versions of eclipse at the time of writing this report (I used version 2019-06).
- Open eclipse in a new workspace.
- Download the following plugins:
  - a. Xtext (just the Xtext item, leave the others unchecked) [8]
  - b. Kotlin [9]
  - c. Optional: TM Terminal (you can find it in the marketplace). This can come in handy, as it provides a shortcut to open a terminal directly in the project folder.
- Copy in the dropins folder of eclipse (<[eclipse\\_installation\\_dir](#)>\dropins) the following files found in the folder iss2019Lab-master\libs\plugins: [it.unibo.Qactork\\_1.1.x.jar](#), [it.unibo.Qactork.ui\\_1.1.x.jar](#), [it.unibo.Qactork.ide\\_1.1.x.jar](#) (use the latest version at your disposal. I used v1.1.7).

Make sure Gradle is correctly setup and added to the environment variables. Add the gradle\bin folder to the PATH variable if needed.


## 4.2 Project setup

Again, we are going to follow the instructions provided in [10].

- Create a new Kotlin project ([File > New > Other...](#), then [Kotlin > Kotlin Project](#)).
- Right click on the src folder and select [New > Other...](#), then select [General > File](#) and name it. It is important that the name ends with “the .qak” suffix. At this point, Eclipse will ask you if you want to convert the project to an Xtext project. Click Yes.
- Write the basis of your QActor code. This means giving the system a name and defining at least one context. For instance:

```
System hello
```

```
Context ctxHello ip [host="localhost" port=8090]
```

- Open both the [build.gradle](#) and [build\\_Ctx<your context>.gradle](#) (in this example, it would be [build\\_ctxHello.gradle](#)) and look for the “[repositories](#)” section. Change the path specified for the `flatDir > dirs` field so that the path points to the directory containing the unibo support libs that are downloaded with the iss2019Lab-master repository.
- Now open a terminal in the project’s folder. If you decided to also install the TM Terminal plugin, you just need to right click on the project folder, then select [Show in Local Terminal > Terminal](#) or use the shortcut CTRL+ALT+T. Run the command “gradle build eclipse”.
- Finally, if the project icon has an exclamation mark near it () , then right click on the project and select [Build Path > Configure Build Path...](#), then under the Source tab, delete one of the two copies of the src folder. When you are done, click Apply and Close.

## 4.3 Receiving messages through mqtt in qactor

We're finally ready to write the model of our first QActor test application. The first model will contain just one actor, whose only purpose is to receive a single message from a mqtt broker and print it. The code will look something like this:

```
System messagereceiver

mqttBroker"192.168.1.247" : 1883

Event sensor : sensor(X)

Context ctxReceiveMessage ip [host="localhost" port=8090] +mqtt

QActor receiveMessage context ctxReceiveMessage {
  State s0 initial {
    println("Starting up event receiver 1.0")
  }
  Goto waitForEvents

  State waitForEvents {
    println("----- READING -----")
  }
  Transition t0 whenEvent sensor -> printEvent

  State printEvent {
    printCurrentMessage
    onMsg(sensor : sensor(X)) {
      println("Sensor event received! sensor(X) -> X = ")
      println(payloadArg(0))
    }
  }
}
```

There's only one more step to take in order for this to work: we now have a QActor that connects to a mqtt broker and can read a message: as you might have guessed, the fundamental question we need to answer now is: what topic is this actor subscribed to?

We can find it out in a very simple way. First, we run a new instance of mosquitto with the -v option, then we run our QActor model. If the prompt informs you that another instance of mosquitto is already running, you may have set up mosquitto to start on boot. In this case, open a new cmd prompt with administrator privileges and terminate the current instance of mosquitto with the command **net stop mosquitto**. When the actor connects to the mqtt broker, the command console will print a notification. It should look something like this:

```
C:\WINDOWS\system32>mosquitto -v
1578589139: mosquitto version 1.6.7 starting
1578589139: Using default config.
1578589139: Opening ipv6 listen socket on port 1883.
1578589139: Opening ipv4 listen socket on port 1883.
1578589341: New connection from 192.168.1.247 on port 1883.
1578589341: New client connected from 192.168.1.247 as receivemessage (p2,
c1, k480).
1578589341: Will message specified (7 bytes) (r1, q2).
1578589341:      unibo/clienterrors
1578589341: Sending CONNACK to receivemessage (0, 0)
1578589341: Received SUBSCRIBE from receivemessage
1578589341:      unibo/qak/receivemessage (QoS 1)
1578589341: receivemessage 1 unibo/qak/receivemessage
1578589341: Sending SUBACK to receivemessage
1578589341: Received SUBSCRIBE from receivemessage
1578589341:      unibo/qak/events (QoS 1)
1578589341: receivemessage 1 unibo/qak/events
1578589341: Sending SUBACK to receivemessage
```

As we can see from the command console, the actor subscribes to two topics:

- **unibo/qak/receivemessage** is the topic to publish mqtt messages that are intended for this actor. Basically, this is the topic to publish mqtt messages representing qactor events that you want only one actor to respond to. In general, when sending a message to a specific actor, we just need to publish a message to the topic **unibo/qak/<name of the actor>**.
- **unibo/qak/events** is the topic to publish mqtt messages that are of interest to multiple actors. Basically, this is the topic to publish mqtt messages representing qactor events.

## 4.4 From Android to QActor

Finally, we can try to transmit a single message from the Android application to the QActor actor.

First of all, we need to slightly change our Android application:

1. We need to specify the correct topic to publish to. We will publish data as an event, so we are publishing to the **unibo/qak/events** topic.
2. We will also want to modify the message sent, so that we are sending something that follows the convention for messages in QActor. By looking at the examples in `iss2019Lab-master` we can find the general syntax of a message:

```
msg( MSGID, MSGTYPE, SENDER, RECEIVER, CONTENT, SEQNUM )
```

Where

**MSGID** : usually used to give a semantic to the message

**MSGTYPE** : dispatch, request, answer, event

**SENDER** : name of the sender

**RECEIVER** : name of the receiving actor. If the message is an even, RECEIVER is 'none'.

**CONTENT** : the payload

**SEQNUM** : a possibly\* unique sequence number

\*what's important is that if  $SEQNUM1 < SEQNUM2$ , then MSG1 is temporarily antecedent to MSG2.

In this simple test, the message we'll be sending is the following:

```
msg(sensor,event,android,none,sensor(payload),1)
```

Then, we:

1. Run mosquitto in a cmd prompt.
2. Run the **messagereceiver** system we realized in section 4.
3. Run the Android class we used to test mosquitto in section 3.

If everything goes well, we should see an output from the **messagereceiver** system that looks similar to the following:

```
    %% sysUtil | context ctxreceivemessage WORKS WITH MQTT
    %% MqttUtils doing connect for receivemessage to tcp://192.168.1.247:1883
    %% MqttUtils connect DONE receivemessage to tcp://192.168.1.247:1883
    %% MqttUtils receivemessage subscribe to topic=unibo/qak/receivemessage
client=org.eclipse.paho.client.mqttv3.MqttClient@371a67ec
    %% MqttUtils receivemessage subscribe to topic=unibo/qak/events
client=org.eclipse.paho.client.mqttv3.MqttClient@371a67ec
    %% QakContext | localhost CREATED. I will terminate after 600000
msec
Starting up event receiver 1.0
----- READING -----
    %% ActorBasic receivemessage | MQTT messageArrived on
unibo/qak/events: msg(sensor,event,android,none,sensor(payload),1)
receivemessage in printEvent | msg(sensor,event,android,none,sensor(payload),1)
Sensor event received! sensor(X) -> X =
payload
```

If not, then the cmd prompt should give us a good insight into what went wrong: remember you can check every step of the connection in the notifications that mosquitto provides you.



## 5 The Android Application

Let's now start working on the application we described at the start of the report: an android application that sends data from the phone sensors to a mqtt broker.

We will want to write to screen a list of all the sensors present on the device. Since the sensors can vary from device to device, we will want to be as “device agnostic” as possible. For this very reason, we can't make assumptions on the name of the devices: if a programmer wants to use a certain sensor, they will need to run this application and take a look at what sensors are present and their names. Then, they will be able to create a QActor actor to handle that specific event.

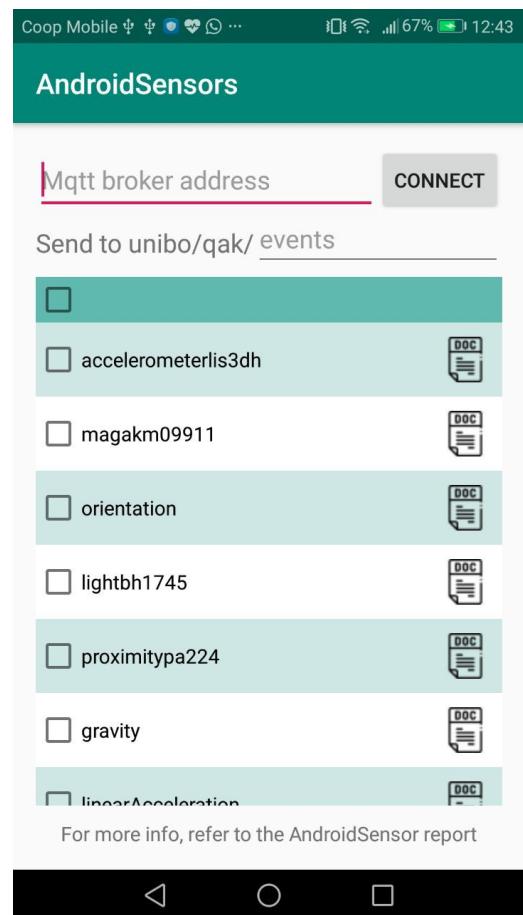
### 5.1 activity\_main.xml

First, let's design the layout of our application. This can be done by editing the **activity\_main.xml** file (since we only have one activity) that can be found in `app > src > main > res > layout`. The layout we want to create is pretty simple: we are going to need:

- A text field to input the address of the mqtt broker
- A text field to input the topic to publish events to
- A button to connect to the mqtt broker
- A list of the sensors in the device

I used a RelativeLayout to position both the text field and the button at the top of the screen, and put the table in the middle so that it stretches the entire screen. Here we have a problem though: the content of the table cannot be known a priori, so we'll need to generate the rows from code.

In a second moment, I also added a header to the table with a checkbox that allows the user to select all sensors at once and a few other small features like a “doc” image that holds a link to the documentation for that type of sensor.



The hierarchy for the activity\_main.xml is roughly the following (for the full code complete with ids, paddings and such, please open the project):

```
<RelativeLayout>                                <!-- anchor info text at bottom and the rest on top -->
  <RelativeLayout>                                <!-- anchor bottom to right and header to bottom -->
    <EditText>                                     <!-- input field for the address. Stretches in width -->
    <Button>                                       <!-- button to start/stop connection -->
    <TextView>                                    <!-- Text to inform the user what the following text field is for -->
    <EditText>                                    <!-- input field for the topic to publish to. Stretches in width -->
    <TableLayout>                                <!-- header for scrollable table. Outside of scrollview so it's fixed -->
      <TableRow>
        <CheckBox/>                               <!-- checkbox to check all sensors with one click -->
      </TableRow>
    </TableLayout>
  </RelativeLayout>
  <ScrollView>                                   <!-- scrollview to scroll table of sensors -->
    <TableLayout>                                <!-- table of sensors. Empty: populate from code -->
    </TableLayout>
  </ScrollView>
  <TextView/>                                     <!-- Info text -->
</RelativeLayout>
```

## 5.2 MainActivity.kt

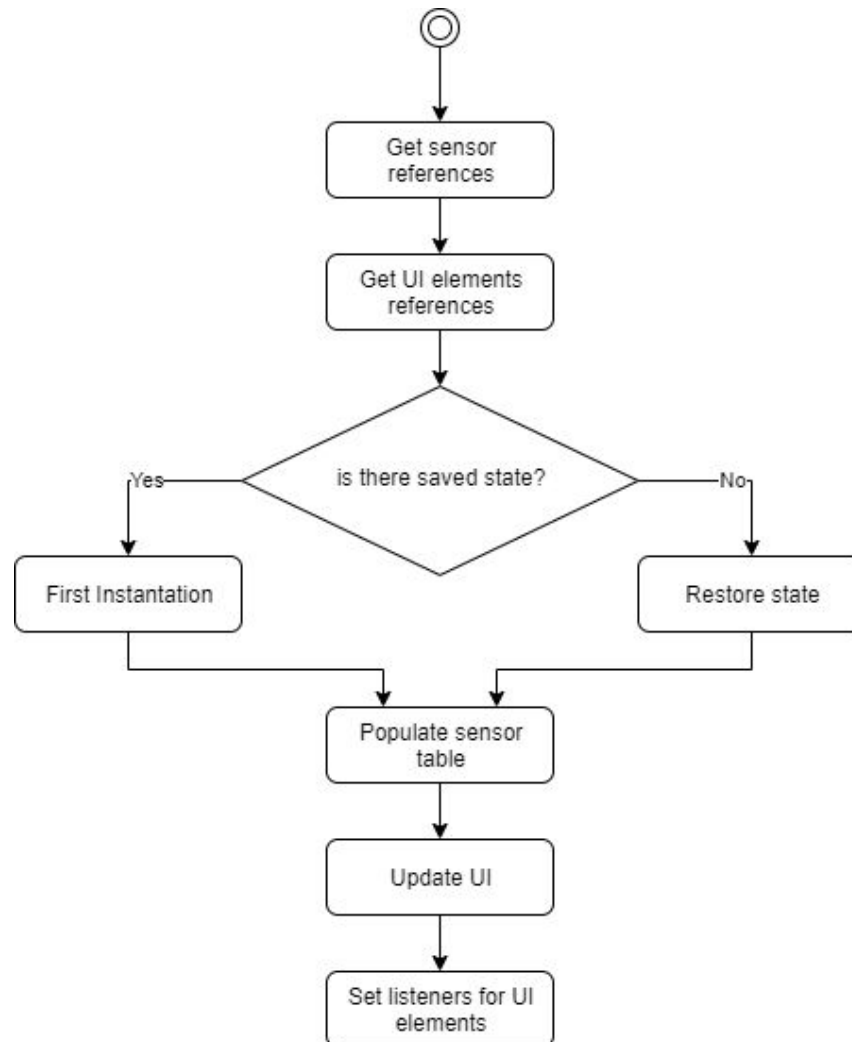
Onto the core of our application. In this section we'll be essentially putting together the various code snippets seen in the previous sections, as they all combine to provide the functionality we desire.

First though, let's talk about the lifecycle of an application in android. Since this is not a toy application anymore, we'll want to control what happens to our application when it's paused, put in the background and so on. The callbacks we'll need to implement are the following:

- **onCreate:** here we'll initialize our application, build the UI and restore state after the application is suspended. We might also be using `onRestoreInstanceState`, but be aware that the `onCreate` callback is performed before the `onRestoreInstanceState` callback. Thus, if you need the state of the application in order to build the UI (as is our case), it gets pretty ugly pretty fast.
- **onSaveInstanceState:** here we'll save the state of the application when it's suspended. This method is called, for instance, when the screen turns off or when the application loses focus.
- **onPause & onResume:** as we've seen in section 2.2, we'll use these methods to temporarily disable the sensors when the application is paused and then enable them again when the application is resumed.

- **onSensorChanged & onAccuracyChanged:** again, as we've seen in section 2.2, we'll use these to respond to changes in the sensors.

Let's take a look at the steps performed in **onCreate**. Again, I won't be reporting here the full code, but rather a schematic representation (for the full code, please open the project). Note: you can search (CTRL+F) the [MainActivity.kt](#) file for the strings used in the flow graph below and it will take you to the appropriate portion of the code.



The basic structure for the `onCreate` method looks relatively simple, as it concerns mainly UI creation and initialization, but a few considerations must be done:

- The following line of code (line 51):

```
window.addFlags(WindowManager.LayoutParams.FLAG_KEEP_SCREEN_ON);
```

is important: this application will probably run for a good amount of time, and if the screen

is turned off during this period, the application will stop transmitting data. Thus, we'll just make it so that the screen doesn't turn off when the application is running in foreground.

- **Populate sensor table:** You can see in the code I did this with a `LayoutInflater`. A `LayoutInflater` is an object that can create an object representation of an xml file with an appropriate schema (the android schema for layouts). This way, I could create each row starting from a template I called `table_row.xml`. I then had to change only a few parameters and append the row in the table.
- **UpdateUI:** This part is necessary, as when the application loads saved state, the UI needs to be consistent with the internal state. For instance, if the screen is rotated, the application does not disconnect from the mqtt as opposed to when the screen is manually turned off. Thus, after the screen rotates, the button should still read "Disconnect" instead of "Connect" (which is the default text for the button). Similarly, we'll want the text input to be disabled and all checkboxes to be checked or unchecked consistently.

Next is the **`onSaveInstanceState`** method. To understand this one, it's important to understand how persistence works in android. As long as we are dealing with simple values, we can store the data we want to retain in a `Bundle` object to be passed around between the `onSaveInstanceState` and the `onCreate` and `onRestoreInstanceState` methods. The `Bundle` object works similarly to a hashmap that maps a string with a value. Unfortunately, this means that in order to store an array of values, we won't be able to store the array itself, but rather we'll need to store a sequence of values. Nothing that a simple loop can't handle.

In this method, we also disconnect from the mqtt client. The reason is that since we will be disabling the sensors when the application is paused, there's no point in keeping the client up and running. Also, with this method we can't persist the client itself, but only the data used to initialize it. Now, since when the screen rotates the application is quickly killed and recreated, you might think the interruption of the connection might pose a problem: in fact, though, the observable behaviour of the application doesn't change much. Sure, when the screen rotates the application will briefly stop emitting events, but will start again as soon as the rotation is complete.

Finally, the method **`onSensorChange`** is the one where we actually send data to the mqtt broker. The code is only a dozen lines long, but makes use of a class called `PublishTask`. The reason is that we need to send data in an asynchronous way as not to block the UI Thread (the main thread in our case) on I/O operations. If we did, we would get a black screen when the frequency the sensor send data at is too high.

Knowing this, the rest of the code is pretty straightforward.

## 5.3 Utils.kt

A helper class called `Utils` is used throughout the application. This class exposes a single method used to remove all white spaces and hyphens (-) from a string.

This class was introduced because in our initial idea, we wanted to use the name of the sensors in the message sent to the broker. This can be done, but poses a problem when the message is parsed by the QActor framework: by default, the payload of the message is processed in Prolog, which will interpret white spaces as separator for tokens and hyphens as the subtraction operation. Thus, to avoid parsing errors, we opted to remove both from the strings we send. Note how the application displays in the table the name of sensor after all white spaces and hyphens have been removed. Ideally, a user only needs to read the name on the screen to know what event to wait for in its QActor models.

## 6 Example applications

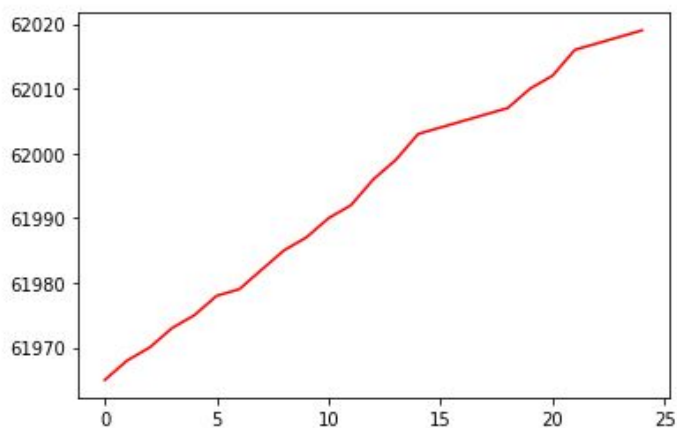
### 6.1 mqttPlotQakEvents.ipynb

To provide the user with an example application that utilizes the sensor data the android application gathers and publishes, and to better grasp the nature of android sensors, we prepared a simple python application that arranges the data gathered in a graph that shows the behaviour of a sensor over time. To do so:

1. Start a mosquitto broker
2. Open the file [mqttPlotQakEvents.ipynb](#) (for instance using Anaconda 3 and opening Jupyter Notebook)
3. Start the AndroidSensor application. Select one of the sensors and connect to the mqtt broker
4. Adjust the configuration parameters of [mqttPlotQakEvents.ipynb](#) (for instance, make sure the name of the sensor you want to plot is active in the AndroidSensor) then run it.

The program will gather data for about 30 seconds and plot a graph.

```
Connected to broker: localhost
Subscribing to unibo/qak/events.
Collecting values; please wait ...
0 > evMsg= msg(androidsensor,event,android,none,androidsensor(stepcounter(61965.0)),4614)
20 > evMsg= msg(androidsensor,event,android,none,androidsensor(stepcounter(62012.0)),4634)
25 > evMsg= msg(androidsensor,event,android,none,androidsensor(stepcounter(62020.0)),4639)
```



Done.

## 6.2 EventLogger

We created a slightly modified version of the QActor model proposed in 4.3 to print all incoming sensor events. The main difference is the presence of two actors. This way, we can test what happens when we ask the android application to send an event to a single actor as opposed to broadcasting it to all actors who are listening.

```
System eventlogger

mqttBroker"192.168.1.247" : 1883

Event androidsensor : androidsensor(X)

Context ctxEventLogger ip [host="localhost" port=8090] +mqtt

QActor eventlogger1 context ctxEventLogger {
  State s0 initial {
    println("Starting up event logger 1")
  }
  Goto waitForEvents

  State waitForEvents {
    println("Logger 1 waiting for events...")
  }
  Transition t0 whenEvent androidsensor -> printEvent

  State printEvent {
    onMsg(androidsensor : androidsensor(X)) {
      println("Logger 1!")
    }
  }
  Transition t0 whenEvent androidsensor -> printEvent
}

QActor eventlogger2 context ctxEventLogger {
  State s0 initial {
    println("Starting up event logger 2")
  }
  Goto waitForEvents

  State waitForEvents {
    println("Logger 2 waiting for events...")
  }
  Transition t0 whenEvent androidsensor -> printEvent

  State printEvent {
    onMsg(androidsensor : androidsensor(X)) {
      println("Logger 2!")
    }
  }
  Transition t0 whenEvent androidsensor -> printEvent
}
```

## References

- [1] Android Developers > Docs > Guides > Sensor Overview @ [https://developer.android.com/guide/topics/sensors/sensors\\_overview](https://developer.android.com/guide/topics/sensors/sensors_overview)
- [2] Android Developers > Docs > Reference > SensorEvent @ <https://developer.android.com/reference/android/hardware/SensorEvent.html>
- [3] Eclipse Paho Mqtt Android Client using Kotlin (2019, August 20). *Medium*. Retrieved from <https://medium.com/@chaitanya.bhojwani1012/eclipse-paho-mqtt-android-client-using-kotlin-56129ff5fbe7>
- [4] Android Developers > Docs > Reference > Service @ <https://developer.android.com/reference/android/app/Service.html>
- [5] Mosquitto Download @ <https://mosquitto.org/download/>
- [6] Mosquitto Documentation @ <https://mosquitto.org/man/>
- [7] Paho Android Service - MQTT Client Library Encyclopedia @ <https://www.hivemq.com/blog/mqtt-client-library-encyclopedia-paho-android-service/>
- [8] Xtext Download Options @ <https://www.eclipse.org/Xtext/download.html>
- [9] Kotlin for Eclipse @ <https://github.com/JetBrains/kotlin-eclipse>
- [10] Labs BO1819 > LAB10 | Using the QActor (meta)model @ <https://github.com/anatali/iss2019Lab>
- [11] Labs BO1819 > LAB11 | The RobotRadar system @ <https://github.com/anatali/iss2019Lab>