*COMPUTER SCIENCE MASTER DEGREE*
*Curriculum Security Engineering*
*A.A. 2020/21*

***IoT Security***

# *An IoT security device to protect an ATM*


**TEAM**

Colella Roberto r.colella17@studenti.uniba.it

Forleo Giovanni g.forleo3@studenti.uniba.it

Gasparro Paolo p.gasparro4@studenti.uniba.it

Piccininni Giuseppe g.piccininni5@studenti.uniba.it

Ardavan Shahoveisi a.shahoveisi@uniba.studenti.it

# Index

# Introduction

The project consists to simulate a safe ATM using an Arduino board. To reach this goal has been defined two possible attack scenarios:

- Scenario 1 - ATM Jackpotting
- Scenario 2 - Physical Attack

# Scenario 1 - ATM Jackpotting

In this scenario it is defined the exploitation **of physical and software vulnerabilities in automated banking machines,** that result in the machines dispensing cash.
Examples of this type of attack are:

- Gain **physical internal access** of the **ATM** through the **top-hat** of the **terminal**
- Use an **endoscope** to identify internal portion of the cash machine
- The attackers sync their device, connecting a cable, to the ATM
- Use an input device to access the ATM's computer
- Install the ATM malware
- Collect cash with money mule

For the project, have been implemented these rules to simulate some of these examples

## Rules

- **Rule 1**: brightness & temperature both greater than the highest value of the threshold calculated in the array of values from the sensors through the interquartile range.
- **Rule 2:** microphone & brightness temperature both greater than the highest value of the threshold calculated in the array of values from the sensors through the interquartile range.
- **Rule 3:** motion must be higher than 0, and temperature greater than the highest value of the threshold calculated in the array of values from the sensors through the interquartile range.
- **Rule 4**: motion must be higher than 0, and microphone greater than the highest value of the threshold calculated in the array of values from the sensors through the interquartile range.
- **Rule 5**: the magnetic linear hall value must be between the highest and lowest

value calculated by the interquartile algorithm inside arrays of values from the sensor.

# Scenario 2 – Physical attack

Another possible attack consists of **physical removing** of **ATM** from the site. Physical attacks on ATMs are considered risky, as it not only leads to financial losses but also involves the risk to property and life. The physical attack usually involves solid and gas explosives attacks, along with physical removal of ATMs from the site and later using other techniques to gain access to the cash dispenser and safe.
For the project, have been implemented these rules to simulate some of these examples.

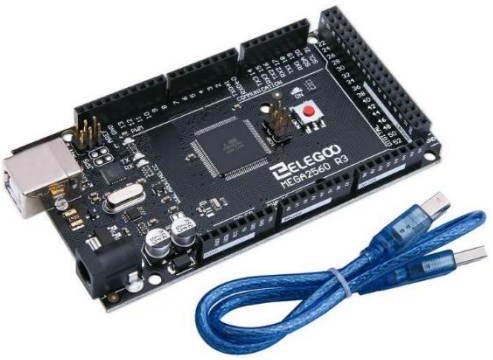
## Rules

- **Rule 6**: find the most frequent value from GPS inside the pre-sized array and compute if the distance from this point is bigger than 200 meters.
- **Rule 7:** accelerometer value from the sensor has to be greater than the highest value of the threshold **or** less than the lowest value of the threshold calculated in the array of values from the sensors through the interquartile range.

# Hardware

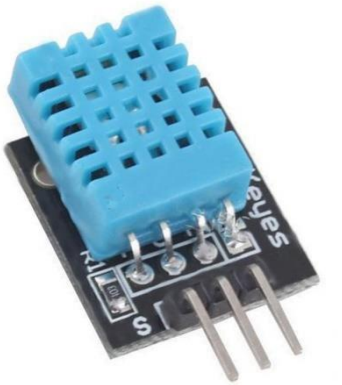
The project uses the following components:

| | |
|---|---|
| Microcontroller: | ATmega2560 |
| Controller chip: | Atmega2560-16au and Atmega16u2 |
| Operating Voltage: | 5V – Input Voltage: 7-9V |
| Digital I/O Pins: | 54 |
| Analog Input Pins: | 16 |
| Flash Memory: | 256 KB of which 8 KB used by bootloader |
| SRAM: | 8 KB |
| EEPROM: | 4 KB |
| Supply: | USB or AC-to-DC |




The hardware is based on **ELEGOO MEGA 2560 R3,** a programmable board which is equivalent to ARDUINO MEGA board. To replicate a real ATM not only with the sensors that we mount on, but even on the aesthetic impact. It has been chosen the ELEGOO MEGA due to the more pins it has, in order to better manage the huge number of sensors and jumper wires.

| | |
|---|---|
| Component name: | Humidity and Temperature Sensor DHT11 |
| Power Supply: | 3.5V to 5V |
| Output: | Serial DATA |
| Detection time: | 2 to 4 s |
| Temperature: | -25 to 68 °C |
| Humidity Range: | 20% to 90% |
| Accuracy: | ±1°C and ±1% |



| | |
|---|---|
| Component name: | Photoresistor KY-018 |
| Operating Voltage: | 3.3V to 5V |
| Output: | Analog DATA |

| | |
|---|---|
| Component name: | PIR Motion Sensor HC-SR501 |
| Input voltage: | 4.8 V – 12 V |
| Delay time: | 0.3s – 200s |
| Output Serial: | Analog DATA when motion is detected |
| Power supply: | 5V-12V |

| | |
|---|---|
| Component name: | Microphone KY-03 |
| Signal: | Analog |
| Power Supply: | +5V |
| Size: | 35mm x 15mm x 14mm |

| | |
|---|---|
| Component name: | Gyroscope GY-521 |
| Chipset: | MPU-6050 |
| Power Supply: | 4.3 to 9 V |
| Gyroscope range: | +/-250, +/-500, +/-1000, +/-2000 °/s |
| Acceleration range: | +/-2g, +/-4g, +/-8g, +/-16g |

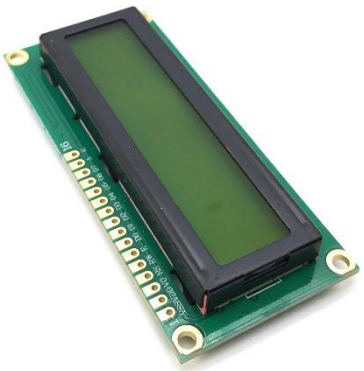

| | |
|---|---|
| Component name: | GSM Module SIM800L |
| Dimensions | 15.8*17.8*2.4mm |
| Control | AT Command |
| Power Supply | 3.4 V – 4.4 V |
| Quad Band | 850/900/1800/1900MHz |

| | |
|---|---|
| Component name: | Magnetic linear hall ky024 |
| Operating Voltage: | 2.7V to 6.5V |
| Sensitivity: | 1.0 mV/G min., 1.4 mV/G typ., 1.75 mV/G max. |
| Board Dimensions: | Board Dimensions: 1.5cm x 3.6cm [0.6in x 1.4in] |

| | |
|---|---|
| Component name: | Gps gy-gps6mv2 neo-6m |
| Input voltage: | 3-5v |
| Data pins voltage: | 3.3v |
| Default baud rate: | 9600 |

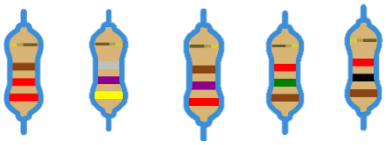
| | |
|---|---|
| Component name: | Relèè ky-019 |
| TTL Control Signal: | 5VDC to 12VDC (some boards may work with 3.3) |
| Maximum AC: | 10A 250VAC |
| Maximum DC: | 10A 30VDC |
| Contact Type: | NC and NO |
| Dimensions: | 27mm x 34mm [1.063in x 1.338in] |

| | |
|---|---|
| Component name: | Display LCD |
| I2C address: | 0x27 Retroillumination |
| Supply voltage: | 5V |
| Dimension: | 82x35x18 mm |
| Package List: | 1 x 1602 I2C Module |
| Interface: | I2C / TWI x1, gadgeteer inteface x2 |

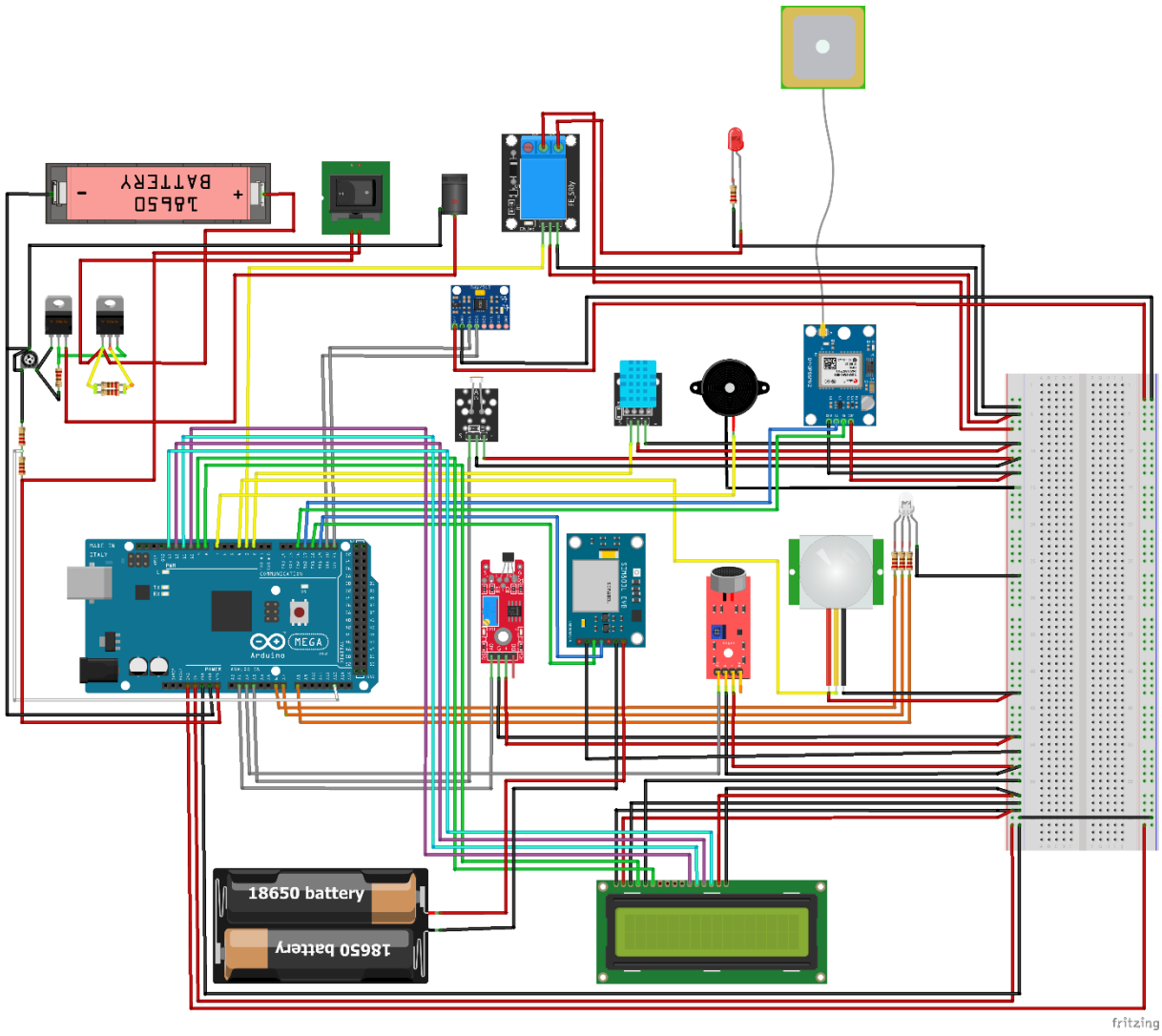


| | |
|---|---|
| Component name: | Passive buzzer |
| Rated Voltage: | 6V DC |
| Operating Voltage: | 4-8V DC |
| Rated current: | <30mA |
| Sound Type: | Continuous Beep |
| Resonant Frequency: | ~2300 Hz |

| | | |
|---|---|---|
| Component name: | Led RED | |
| | Led RGB | |
| | Power Button | |
| | Resistances | 4 (220 ohm)<br>2 (4,7 Ohm)<br>1 (270 Ohm)<br>1 (1,5k Ohm<br>1 (1k Ohm) |
| | Battery-18650 | 2 (4,2 v 2.5 Ampere ) |
| | Trimmer | |
| | Lm317T | |
| | Breadboard | |



# Circuit Scheme




To build the ATM and make it portable it has been decided to buy a white box and put all the hardware inside it.
First of all it has been placed the Elegoo Mega 2560 R3 on one side of the box and the breadboard on the bottom. The sensors have been placed on the other side respect to the Elegoo board following a logic based on the frequency of each sensor: starting from the lowest(DHT11) to the highest (GSM).
It have been measured the components and created the holes which fixed the size of them (like the display, the RGB LED, the serial USB port of Elegoo, the DC jack to charge the battery inside the box and the switch to turn on or off the entire system).
To make the ATM portable and so rechargeable, it has been created a circuit connected with two batteries of type 18650 connected in series (each one of 4,2V 2A for a total of 8,4V). The purpose of the circuit is to regulate the voltage given from outside (in our case we use a 12V DC power supply) and to regulate the amperage (500mA to recharge). This goal has been reached by using 2 IC of type LM317T, a trimmer, two 4,7 Ohm

resistor in parallel, one 270 Ohm resistor, one 1,5k Ohm resistor and one 1k Ohm resistor.
Since the GSM module has frequent peaks of current consumption, it has been powered using a powerbank which is able to provide up to 2A of direct current with a 5V voltage.
At last, the circuit has been connected to the on/off switch to turn on or off the ATM.

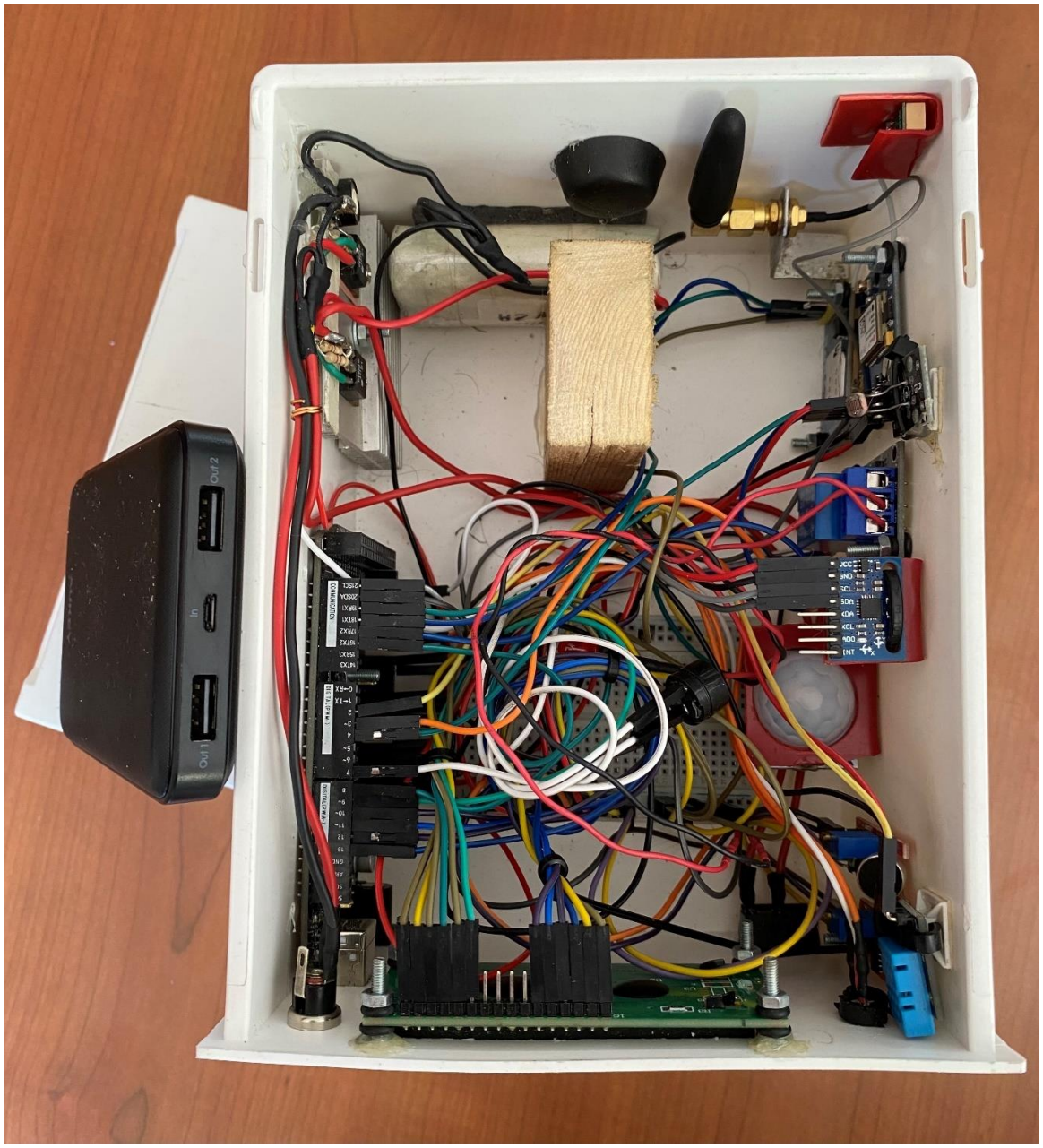
# The atm system

# Outside the ATM


***Normal situation***


***Alarm State***

# Inside the ATM



# RabbitMQ

Talking about the back-end side of the project, has been used **RabbitMQ** broker to store the data received the board in real time. RabbitMQ is a broker that implements the MQTT protocol. It adopts queues to store messages received, in this case, from the board.

| Overview | | | | Messages | | | Message rates | | |
|---|---|---|---|---|---|---|---|---|---|
| **Name** | **Type** | Features | **State** | **Ready** | **Unacked** | **Total** | **incoming** | **deliver / get** | **ack** |
| **mqtt-subscription-atm-group3-00qos0** | classic | AD | idle | 0 | 0 | 0 | | | |
| **mqtt-subscription-node-RED-group3qos0** | classic | AD | running | 0 | 0 | 0 | 0.60/s | 0.60/s | 0.00/s |
| **q_sensors** | classic | D Args | running | 20 | 0 | 20 | 0.60/s | 0.00/s | 0.00/s |

When the whole system is running, these **queues** are **visible**:

## q_sensors

The only queue defined in the RabbitMQ server is **q_sensors.** It takes all the messages coming from the sensors. Inside that queue has been defined all the **routing keys** to correctly manage the different data.

| From | Routing key | Arguments | |
|---|---|---|---|
| (Default exchange binding) | | | |
| amq.topic | atm.brightness.* | | Unbind |
| amq.topic | atm.gps.* | | Unbind |
| amq.topic | atm.gyroscope.* | | Unbind |
| amq.topic | atm.linearhall.* | | Unbind |
| amq.topic | atm.microphone.* | | Unbind |
| amq.topic | atm.motion.* | | Unbind |
| amq.topic | atm.temperature.* | | Unbind |

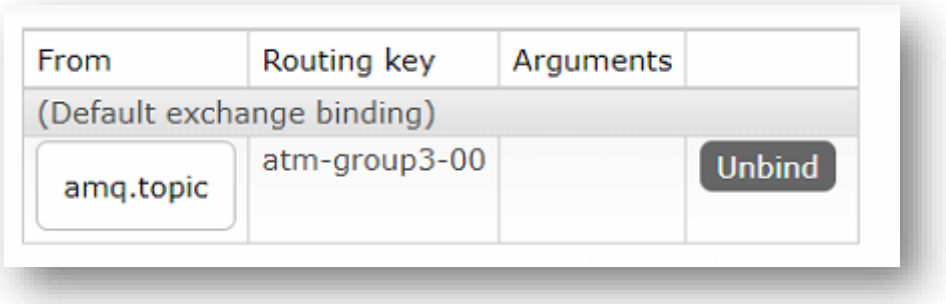
## mqtt-subscription-node-RED-group3qos0

This is the queue that is created when the rules engine (Node-RED) subscribe to the broker specifying the various **routing keys** defined in this way:

| From | Routing key | Arguments | |
|---|---|---|---|
| (Default exchange binding) | | | |
| amq.topic | atm.brightness.value | | Unbind |
| amq.topic | atm.gps.value | | Unbind |
| amq.topic | atm.gyroscope.value | | Unbind |
| amq.topic | atm.linearhall.value | | Unbind |
| amq.topic | atm.microphone.value | | Unbind |
| amq.topic | atm.motion.value | | Unbind |
| amq.topic | atm.temperature.value | | Unbind |

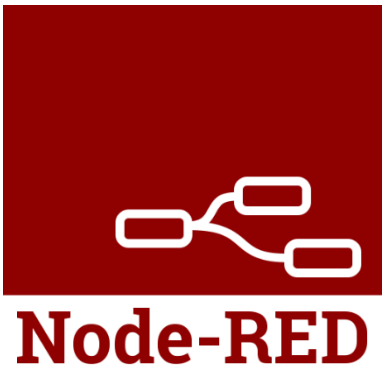
# mqtt-subscription-atm-group3-00qos0

This is the queue that is created when the Arduino board subscribe to the broker specifying the following **routing key**:



| From | Routing key | Arguments | |
|---|---|---|---|
| (Default exchange binding) | | | |
| amq.topic | atm-group3-00 | | Unbind |

# Node-RED



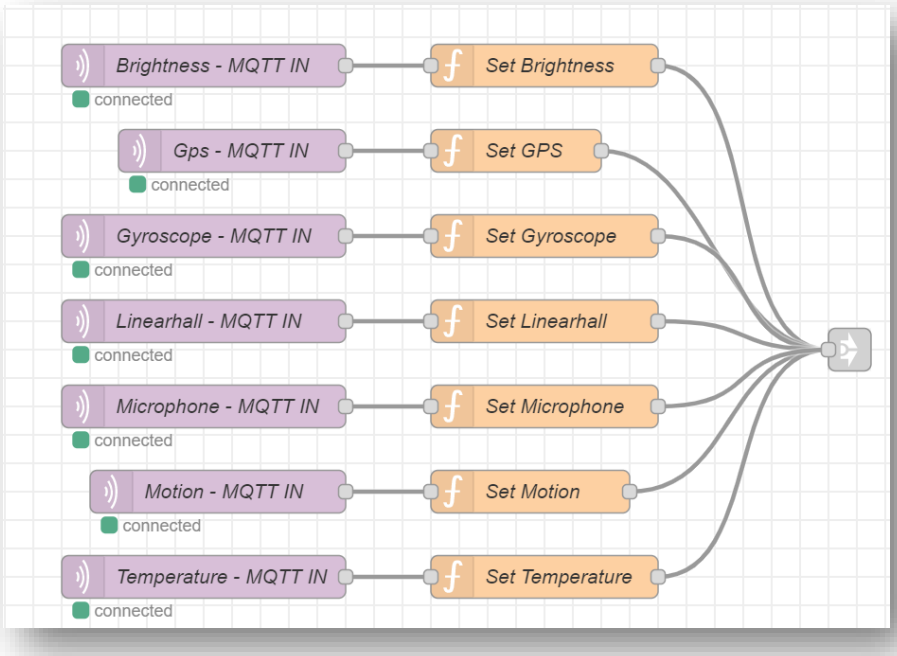

This simple programming tool is used to create a real flow and logic behind the ATM system. It's a browser-based web application in which it's possible to create a flow using nodes, properly linked to the data that the board send to the broker (RabbitMQ). Node-RED allows to easily create real rules in JavaScript for processing the whole data.

After a careful study of the tool and its functionality, the logic to be implemented on the Node-RED system has been designed. The main steps that the team has focus on were:
take the data sent from Arduino to RabbitMQ using the MQTT IN nodes in order to subscribe to a certain topic
find a way to format these data and collect them to allow the rules function to easily manage them
design the smart rules to check if some attacks is running using the previously formatted data
define a methodology to reset the system after an alarm state or dynamically update the rules after a specific time
To reach these goals have been implemented three flows:

- **Sensor flow**
- **Rules flow**
- **Reset flow**

# Sensor flow

The **sensor flow** has been used to extract the data coming from the IoT device and store them in global variables through the function nodes. To receive the data, **MQTT IN** nodes with a "**set function"** for each sensor have been defined. The data received from the ATM are processed by the function to format, parse and store the data for a better elaboration, creating global variables.
Linked to each function node "**Set"** has been added a **LINK OUT** node. This node is used to send the data among the flows, sharing the global variables among them.

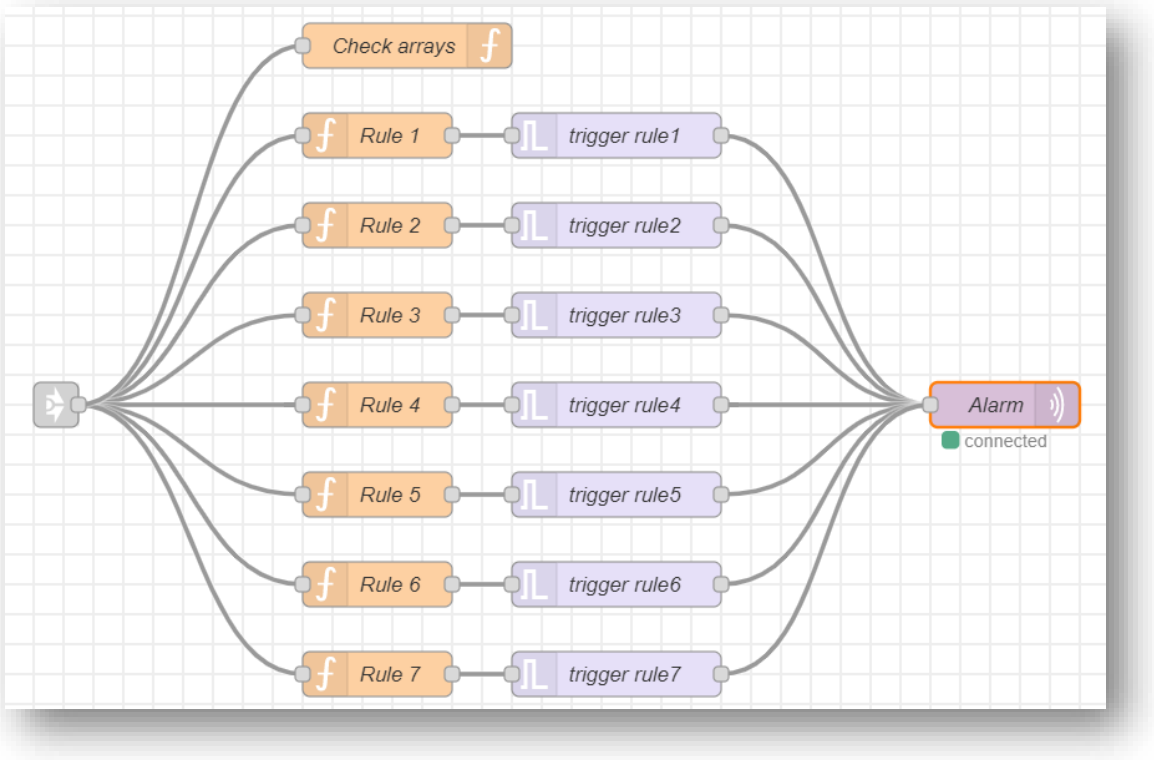



An example of **set** function is shown below:

```
var arraySize = global.get("arraySize") || 20;
var brightnessValue= msg.payload;
var brightnessArray = global.get("brightnessArray") || [];
var msg1={payload:"null"};
brightnessValue= parseInt(brightnessValue.replace(/\D/g,''));

if (brightnessValue >= 0 && brightnessValue < 1025) {
    if (brightnessArray.length < arraySize){
        brightnessArray.push(brightnessValue);
    }
}
global.set("arraySize",arraySize);
global.set("brightnessValue",brightnessValue);
global.set("brightnessArray",brightnessArray);
msg1={payload:brightnessValue};
return msg1;

```

# Rules flow

The most important flow that has been defined is the **rules flow.** In this flow has been created the logic that control the value received from the board, to trigger an eventually alarm, depending on the broken rule.
All the functions nodes implement the logic of the rules, receiving the global variables from the **LINK IN** from Sensor flow.




To obtain a smoother flow and avoid continuously sending unnecessary data to the board, triggers node has been chosen. These nodes are very useful to “filter” the functions’ outputs that arrives from each flow. The only message that can pass through the trigger nodes are the alarm message, and that message can pass only one time. This solution has been designed because the trigger sent the message just in case the data breaks the implemented rules, without continuously sending data to the board.
The **MQTT OUT** node send the message to the IoT system, and this message will activate the alarm state on the board. In the ATM system an alarm will sound, showing the rule label on the display and turning the led into red.
If the system is not reset within three minutes, the atm ink stains the banknotes. Due to the fact that this system doesn’t have real cash, this scenario is simulated with a led inside the atm that turn into red.

# Check Arrays

```
var arraySize = global.get("arraySize") || 20;
var completeCheck= global.get("completeCheck") || false;
var temperatureArray= global.get("temperatureArray");
var brightnessArray= global.get("brightnessArray");
var microphoneArray= global.get("microphoneArray");
var linearHallArray= global.get("linearHallArray");
var gpsArray= global.get("gpsArray");
var zAccellerometerArray= global.get("zAccellerometerArray");
var isAlarmOn = global.get("isAlarmOn") || false;

if (typeof temperatureArray !== 'undefined'
    && typeof microphoneArray !== 'undefined'
    && typeof linearHallArray !== 'undefined'
    && typeof brightnessArray !== 'undefined'
    && typeof gpsArray !== 'undefined'
    && typeof zAccellerometerArray !== 'undefined') {
        if (temperatureArray.length === arraySize
            && microphoneArray.length == arraySize
            && linearHallArray.length == arraySize
            && brightnessArray.length == arraySize
            && gpsArray.length == arraySize
            && zAccellerometerArray.length == arraySize) {
                completeCheck=true;
        }
```

This function checks if the arrays have been properly created in the **Sensor flow**. The global variables **completeCheck** and **isAlarmOn** are used to prevent the rules from overlapping (when triggered) and to store the arrays status.

# Rules

For each rule (except for the 6th) has been used the following function defined in the image below. To compute the Threshold, this function checks if the real time values are in a specific range. Since the data are stored into the arrays, the functions couldn't focus on the single value to compute the threshold, so has been decided to use a method based on the **interquartile range algorithm**. This method, after calculating the median of the values stored in the arrays, calculate the high and low threshold using the third and first quartile. These two values define the range in which the temperature or brightness values have to stay in to not trigger the rule. Due to the poor manufacture of the sensors installed in the system, we have added a 20% to the threshold values, otherwise the rules would be trigger very often.

```
23  function computeThreshold(array,type) {
24      var q2 = getMedian(array);
25      var q1 = getMedian( array.slice(0, q2[0]+1) );
26      var q3 = getMedian( array.slice(q2[0]+1, array.length) );
27      var iqr = q3[1] - q1[1]; //The interquartile range
28      var tlo = q1[1] - 1.5 * iqr ;//low threshold
29      var thi = q3[1] + 1.5 * iqr ;//high threshold
30      if (type == "high") {
31          return (thi + (thi/100*20));
32      }
33      return (tlo - (tlo/100*20));
34  }
35
36  function getMedian (m) {
37      var middle = Math.floor((m.length - 1) / 2); // NB: operator precedence
38      if (m.length % 2) {
39          return [middle, m[middle]];
40      } else {
41          return [middle, (m[middle] + m[middle + 1]) / 2.0];
42      }
43  }
```

# Rule 1

For this rule, the sensors considered were brightness and temperature. As said before, we will operate on prefilled arrays of values. The if statement control if the arrays is filled and if there aren't other rules activated (**isAlarmOn** is false). After that the function checks if the real-time values of brightness and temperature are exceeding the thresholds computed by a specific function that has been implemented.

```
var brightnessValue=global.get("brightnessValue");
var temperatureValue= global.get("temperatureValue");
var brightnessArray= global.get("brightnessArray");
var temperatureArray= global.get("temperatureArray");
var completeCheck= global.get("completeCheck");
var isAlarmOn= global.get("isAlarmOn");
var msg1={payload:"okrule1"};

if (completeCheck
    && !isAlarmOn
    && temperatureValue > computeThreshold(temperatureArray,"high")
    && brightnessValue > computeThreshold(brightnessArray,"high")) {
        msg1={payload:"05rule1"};
        global.set("isAlarmOn", true);
    }

return msg1;
```

# Rule 2

For this rule, the values of the sensors considered were noise and brightness.

```
var brightnessValue= global.get("brightnessValue");
var brightnessArray= global.get("brightnessArray");
var microphoneValue= global.get("microphoneValue");
var microphoneArray= global.get("microphoneArray");
var completeCheck= global.get("completeCheck");
var isAlarmOn= global.get("isAlarmOn");

var msg1;
msg1={payload:"okrule2"};
if( completeCheck && !isAlarmOn
    && microphoneValue > computeThreshold(microphoneArray,"high")
    && brightnessValue > computeThreshold(brightnessArray,"high")) {
    msg1={payload:"05rule2"};
    global.set("isAlarmOn", true);
}
return msg1;
```

## Rule 3

For this rule, the values of the sensors considered were motion and temperature. In this case the statement was a bit different, because the **PIR sensor return 0** if no motion is detected, while **return 1** if motion is detected.

```
var motionValue= global.get("motionValue");
var temperatureValue= global.get("temperatureValue");
var temperatureArray= global.get("temperatureArray");
var completeCheck= global.get("completeCheck") || false;
var isAlarmOn= global.get("isAlarmOn");

var msg1;
msg1={payload:"okrule3"};

if( completeCheck
    && !isAlarmOn
    && temperatureValue > computeThreshold(temperatureArray,"high")
    && motionValue > 0) {
    msg1={payload:"05rule3"};
    global.set("isAlarmOn", true);
}
return msg1;
```

## Rule 4

For this rule, the values of the sensors considered was the motion and the temperature.

```
var motionValue= global.get("motionValue");
var microphoneValue= global.get("microphoneValue");
var microphoneArray= global.get("microphoneArray");
var completeCheck= global.get("completeCheck") || false;
var isAlarmOn= global.get("isAlarmOn");

var msg1;
msg1={payload:"okrule4"};

if( completeCheck
    && !isAlarmOn
    && microphoneValue < computeThreshold(microphoneArray,"low")
    && motionValue > 0) {
    msg1={payload:"05rule4"};
    global.set("isAlarmOn", true);
}
return msg1;

```

## Rule 5

For this rule, the value of the sensor considered was the magnetic field.

```
var linearHallValue=global.get("linearHallValue");
var linearHallArray= global.get("linearHallArray");
var completeCheck= global.get("completeCheck") || false;
var isAlarmOn= global.get("isAlarmOn");

var msg1;
msg1={payload:"okrule5"};

if( completeCheck
    && !isAlarmOn
    && (linearHallValue < computeThreshold(linearHallArray,"low")
    || linearHallValue > computeThreshold(linearHallArray,"high"))) {
    msg1={payload:"05rule5"}
    global.set("isAlarmOn", true);
}
return msg1;

```

## Rule 6

The values of the sensor for this rule are the **longitude** and **latitude**. The algorithm computes the distance between two geolocation points (latitude and longitude) on the Earth. The function takes the value in the array that is more frequently repeated and calculate the distance along these couple of coordinates and the couple received from the board.

```
function computeDistance (lat1, lon1, lat2, lon2) {
    var R = 6371; //radius of the earth in Km
    var x = (lon2-lon1)* Math.cos(0.5*(lat2+lat1));
    var y = lat2 - lat1;
    return (R * Math.sqrt (x*x + y*y));
}

function moreFrequentlyItem(arr) {
  var maxFreq = 1;
  var counter = 0;
  var item = [];
  for (var i=0; i<arr.length; i++) {
    for (var j=i; j<arr[i].length; j++) {
        if (arr[i][0] == arr[j][0] && arr[i][1] == arr[j][1]) {
            counter++;
        }
        if (maxFreq<counter) {
            maxFreq=counter;
            item = arr[i];
        }
    }
    m=0;
  }
  return item;
}
```

To trigger the rule the distance has to be more than 200 meters.

```
var completeCheck= global.get("completeCheck");
var isAlarmOn= global.get("isAlarmOn");
var gpsArray= global.get("gpsArray");
var lat2 = global.get("gpsValueLat");
var lon2 = global.get("gpsValueLon");
var coordinates, distance;
var msg1={payload:"okrule6"};

if (completeCheck && !isAlarmOn) {
    coordinates = moreFrequentlyItem(gpsArray);
    distance = computeDistance(parseFloat(coordinates[0]), parseFloat(coordinates[1]),lat2,lon2);
    if (distance > 0.2) {
        msg1={payload:"05rule6"};
        global.set("isAlarmOn", true);
    }
}

return msg1;
```

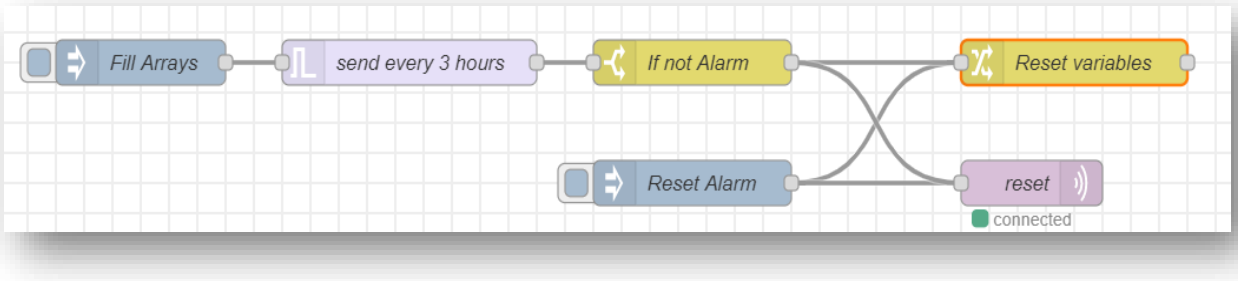
# Rule 7

The values of the sensor for this rule is the z axis of the accelerometer.

```
//Get global variables
var completeCheck = global.get("completeCheck");
var isAlarmOn = global.get("isAlarmOn");
var zAccellerometerArray=global.get("zAccellerometerArray");
var zAccellerometerValue = global.get("zAccellerometerValue");

var msg1={payload:"okrule7"};

if(completeCheck && !isAlarmOn) {

    //extract array to compute the threshold
    var zAccellerometerThi=computeThreshold(zAccellerometerArray,"high");
    var zAccellerometerTlo=computeThreshold(zAccellerometerArray,"low");

    if ( zAccellerometerValue < zAccellerometerTlo ||  zAccellerometerValue > zAccellerometerThi) {
        msg1={payload:"05rule7"}
        global.set("isAlarmOn", true);
    }
}
return msg1;
```

## Reset flow

In this flow the injects buttons update the global variables if a specific rule is triggered and the board goes into an **alarm state.**




In the image there are two inject nodes (**Fill Arrays, Reset Alarm**) that are connected with to node called **Reset variables** and **reset**.
The **Reset variables** node is a change node used to set, change or delete properties of a message inside the flows. It's used to empty the global variables and to store new updated values from the sensors.
The **reset** node is a **MQTT out** node which send the injected message to the ATM's queue in RabbitMQ, to activate the reset procedure on the board (to turn off the alarm, to change the display Status to the normal state and to turn the led into green color).
The **if not Alarm** node is used to check if in that precise moment the system is in alarm state or not.
The **trigger** node sent every three hours the reset message.
The **Fill Arrays** inject node, when pressed, send a message reset in trigger node, the main purpose of this button is to dynamically update the data during the day in order to make the sample of data useful for the thresholds in the rules realistic.
The **Reset Alarm** node is used to inject a string to bring the system immediately out of the alarm state.

# Data Visualization

To show the data from RabbitMQ has been used different tools that take them to create charts useful for monitoring data trends. These tools are:

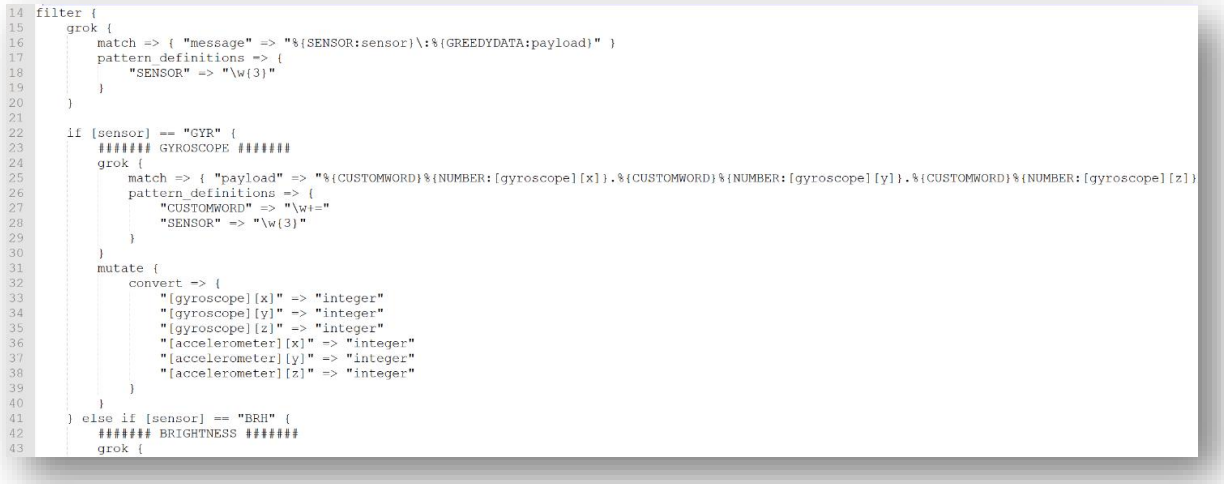
- Logstash
- Elastic Search
- Kibana

# Logstash

**Logstash** is used to process the data gathered from the queues on RabbitMQ. Through a pipeline, Logstash is able to send and transform data to **Elasticsearch**. In the input lines in sensors-pipeline configuration file, has been inserted the RabbitMQ server address, port and the credentials to access to it.

```
input {
  rabbitmq {
    host => "paologas91.chickenkiller.com"
    port => 5672
    user => "atm"
    password => "atm"

    queue => "q_sensors"
    passive => true

    codec => "plain"
  }
}
```

Inside the sensors-pipeline configuration file there are even **grok filters** to **parse unstructured data** in **structured data (JSON format).** The data converted are saved into a txt file, and then redirected to **Elasticsearch**.



```
filter {
    grok {
        match => { "message" => "%{SENSOR:sensor}\:%{GREEDYDATA:payload}" }
        pattern_definitions => {
            "SENSOR" => "\w{3}"
        }
    }

    if [sensor] == "GYR" {
        ####### GYROSCOPE #######
        grok {
            match => { "payload" => "%{CUSTOMWORD}%{NUMBER:[gyroscope][x]}.%{CUSTOMWORD}%{NUMBER:[gyroscope][y]}.%{CUSTOMWORD}%{NUMBER:[gyroscope][z]}
            pattern_definitions => {
                "CUSTOMWORD" => "\w+="
                "SENSOR" => "\w{3}"
            }
        }
        mutate {
            convert => {
                "[gyroscope][x]" => "integer"
                "[gyroscope][y]" => "integer"
                "[gyroscope][z]" => "integer"
                "[accelerometer][x]" => "integer"
                "[accelerometer][y]" => "integer"
                "[accelerometer][z]" => "integer"
            }
        }
    } else if [sensor] == "BRH" {
        ####### BRIGHTNESS #######
        grok {
```

```
output {
    stdout {
        codec => rubydebug
    }
    if [sensor] == "GYR" {
        ####### GYROSCOPE #######
        file {
            path => "./sensors/gyroscope/gyroscope-%{+YYYY-MM-dd}.txt"
        }
        elasticsearch {
            hosts => ["localhost:9200"]
            index => "wws-toaster-gyroscope"
        }
    } else if [sensor] == "BRH" {
        ####### BRIGHTNESS #######
        file {
            path => "./sensors/brightness/brightness-%{+YYYY-MM-dd}.txt"
        }
        elasticsearch {
            hosts => ["localhost:9200"]
            index => "wws-toaster-brightness"
        }
    } else if [sensor] == "GPS" {
        ####### GPS #######
        file {
            path => "./sensors/gps/gps-%{+YYYY-MM-dd}.txt"
        }
        elasticsearch {
            hosts => ["localhost:9200"]
            index => "wws-toaster-gps"
        }

```

# Elasticsearch

**Elasticsearch** is a Lucene-based search server, with Full Text capability, with support for distributed architectures. All the functionalities are natively exposed through the RESTful interface, while the information is managed as JSON documents. Elasticsearch can be used to search any type of document and provides a scalable, near real-time search system with multitenancy support. Is used with Logstash which sends the data to Elasticsearch which stores them in **JSON format**. Elastic can communicate with Kibana to provide a **real-time visualization of data**.

# Kibana

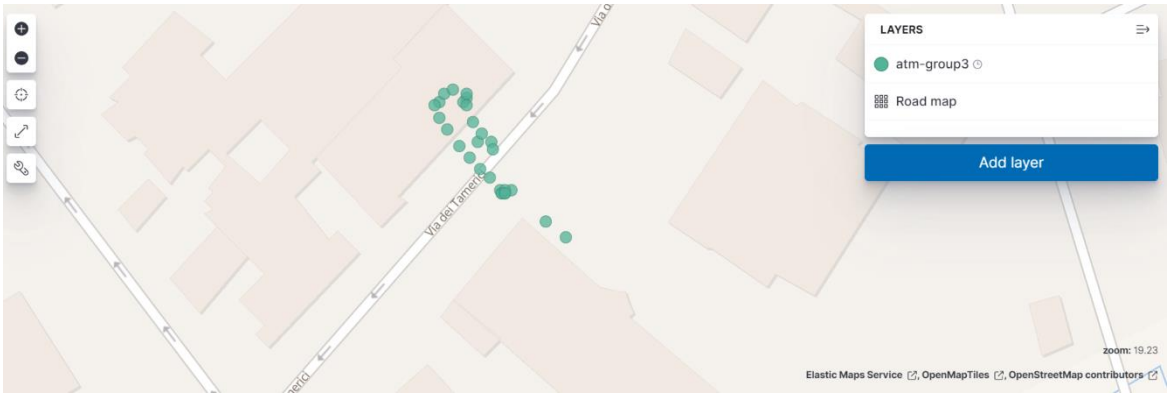
Kibana is the last tool used to provide a final visualization of the data gathered by

Elasticsearch. It provides a data visualization dashboard that allows to monitor the real-time data. Below has been defined some charts for the ATM's project.

# Kibana's Graphs

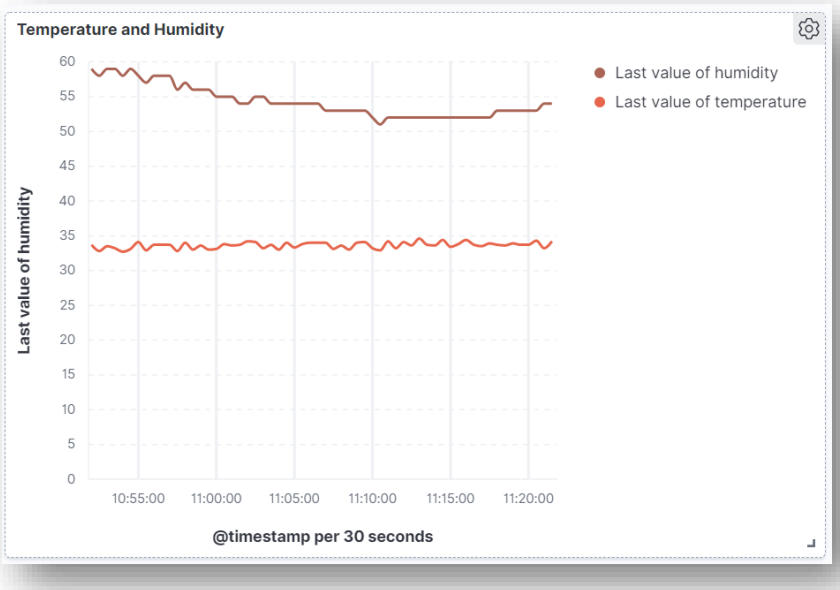
## GPS

The points generated on the map indicate the position in time of the ATM. Due to the sensor and the position of the ATM, a slightly incorrect position with respect to the real one may be reported.




## Brightness

This is the chart generated from Kibana which show the latest values of brightness along the time. On the vertical axis we have the values of brightness gathered while on the horizontal we have the time.

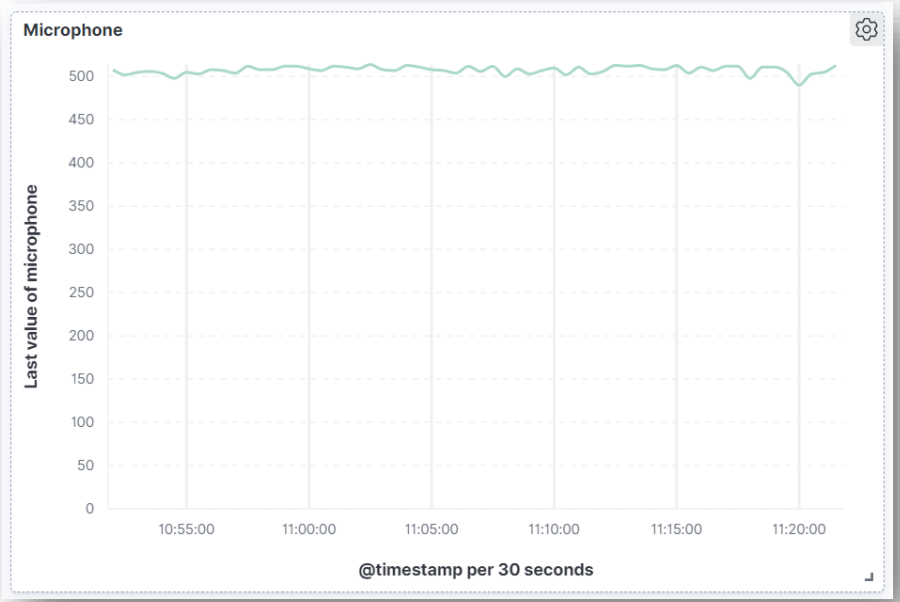



## Temperature Humidity

This is the chart generated from Kibana which show the latest values of temperature and humidity along the time.
On the vertical axis we have the values of temperature and humidity gathered while on the horizontal we have the time.

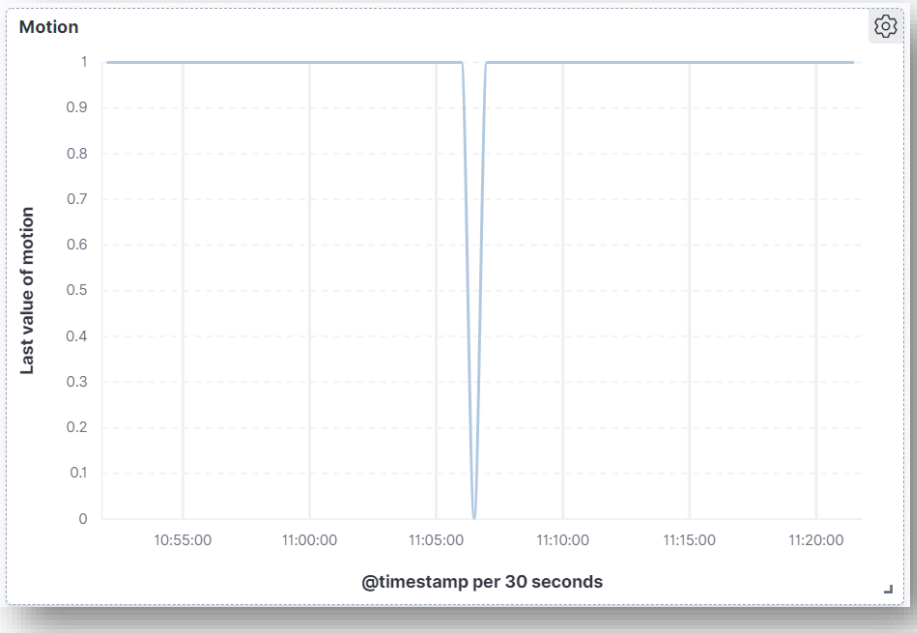
## Microphone

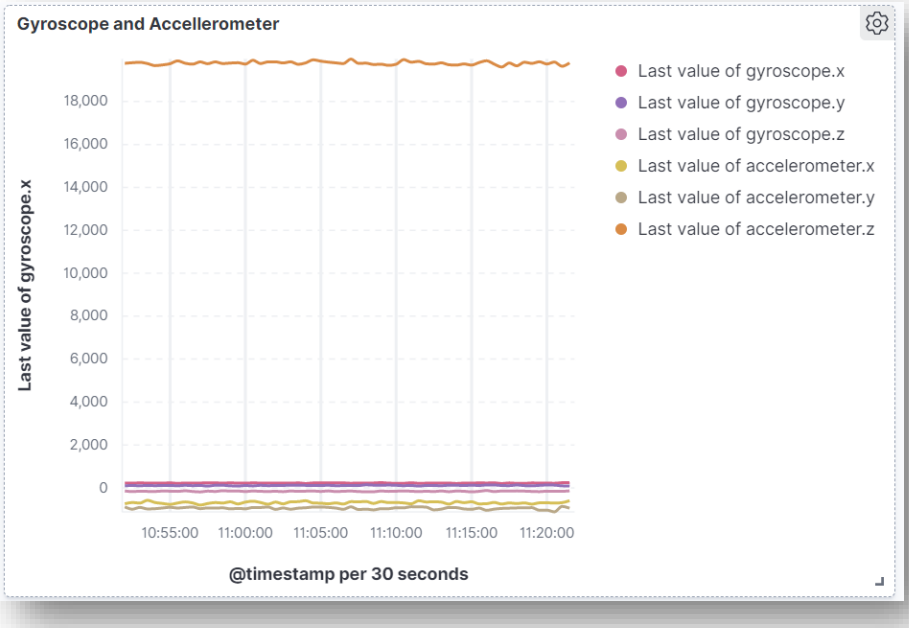
This is the chart generated from Kibana which show the latest values of microphone along the time.
On the vertical axis we have the values of microphone gathered while on the horizontal we have the time.



## Motion Sensor

This is the chart generated from Kibana which show the binary values of the motion gathered along the time. On the vertical axes we have the 0 or 1 value generated by motion, on the horizontal axes we have the time.




## Gyroscope and Accelerometer

This is the chart generated from Kibana which show the latest values of gyroscope (x,y,z axes) and accelerometer (x,z,y axes).

## Magnetic linear hall

This is the chart generated from Kibana which the latest values of magnetic linear hall along the time.

On the vertical axis we have the values of the magnetic linear hall gathered while on the horizontal we have the time.