UNIVERSITÀ DI PISA

COMPUTER ENGINEERING

Internet of Things

PROJECT DOCUMENTATION

# Design and development of *"SmartSauna"*: an IoT Telemetry and Control System

**Students**

Federica Baldi

Daniele Cioffo

Academic Year 2020/2021

## TABLE OF CONTENTS

# 1 INTRODUCTION

The benefits of the sauna on the human body are well known, but to be able to fully exploit them, continuous work to maintain the environment is necessary. For example, in the first Finnish saunas the temperature was increased by throwing water on hot stones, generating steam.

The goal of this project is to create a smart solution for the management of a sauna, automating all the procedures through IoT technology. Our application must be able to independently manage air humidity, temperature level, and air quality. In these pandemic periods it is also essential to ensure a contingent entry, verifying that the number of people inside the sauna never exceeds a certain threshold.

The application also leaves a high degree of freedom to the owner of the sauna, who can set the thresholds as he wishes and can take advantage of the application to apply chromotherapy, setting the color of the light among those available.

The project code can be visited at the following link: https://github.com/danielecioffo/SmartSauna.

# 2 ARCHITECTURE

The system architecture is as shown in the following image (Figure 1).

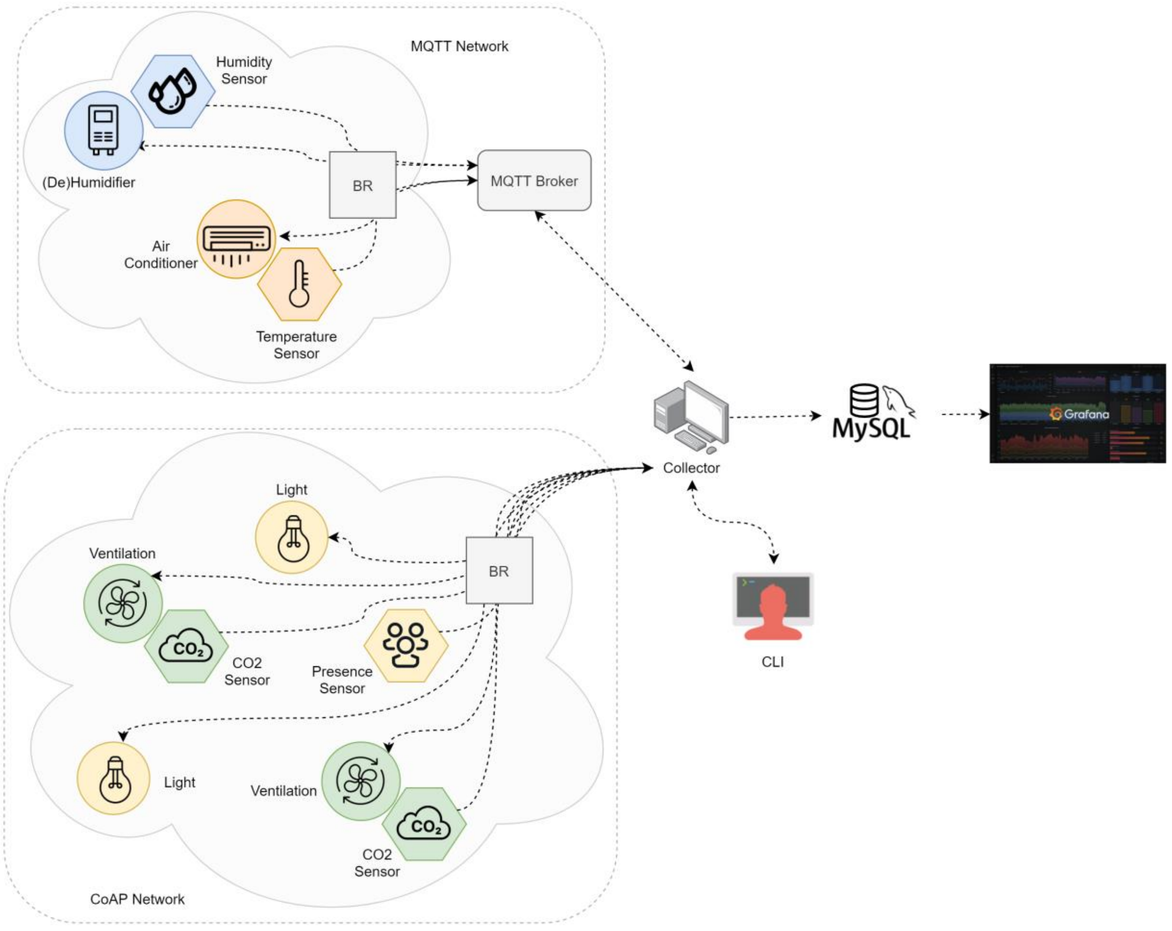


*Figure 1 – System* architecture

As can be seen, the system is composed of two different networks of IoT devices: one network is composed of IoT devices that use MQTT to report data, while the other network uses CoAP as the application protocol.

The MQTT network is deployed using real sensors from the IoT testbed, while the CoAP network is simulated using Cooja. Both networks are connected to a border router that allows them external access. The MQTT broker is deployed on the testbed.

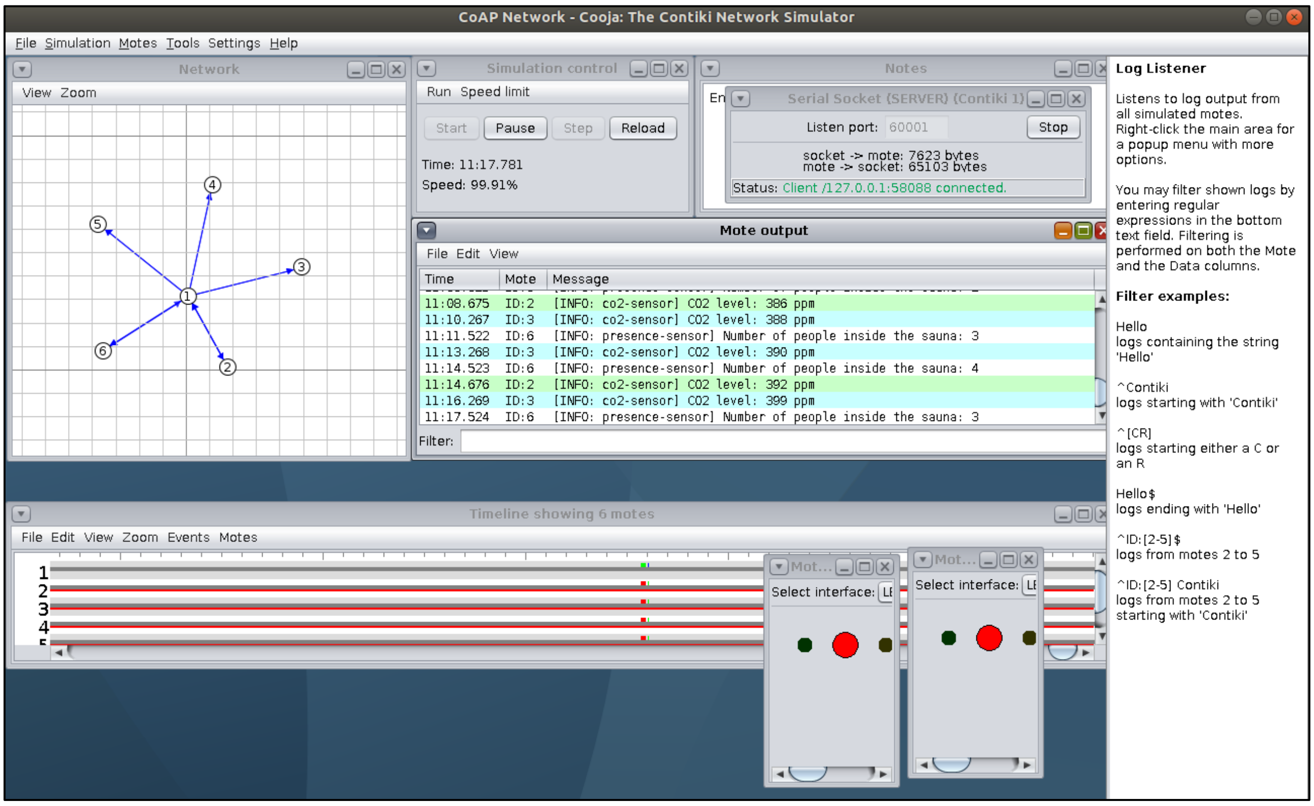
As will be explained in more detail later, there are two sensors (temperature and humidity) and two actuators (an air conditioner and a dehumidifier/humidifier) in the MQTT network. In the CoAP network, on the other hand, we have a presence sensor and two sensors to measure air quality (specifically, they measure CO2 concentration). As actuators, we have two air fans and two light bulbs.

The collector collects data from both MQTT and CoAP sensors and stores them in a MySQL database. The collected data can then be visualized through a web-based interface developed using Grafana.

A simple control logic is executed on the collector in order to enable or disable the actuators and modify the external environment based on the data that has been collected and the desired intervals for temperature, humidity and CO2 concentration. The collector also exposes a simple command line interface through which the user can retrieve the last measured values and change the ranges within which the different physical quantities must lie.

In the next few paragraphs, we will go on to explain the different components of the system in more detail.

## 2.1 CoAP Network


CoAP Network - Cooja: The Contiki Network Simulator
File Simulation Motes Tools Settings Help
Network
View Zoom
Simulation control
Run Speed limit
Start Pause Step Reload
Notes
Serial Socket (SERVER) (Contiki 1)
Stop
Mote output
File Edit View
Timeline showing 6 motes
File Edit View Zoom Events Motes
Select interface:

Time: 11:17.781
Speed: 99.91%

Listen port: 60001
socket -> mote: 7623 bytes
mote -> socket: 65103 bytes
Status: Client /127.0.0.1:58088 connected.

```
Time       Mote  Message
11:08.675  ID:2  [INFO: co2-sensor] CO2 level: 386 ppm
11:10.267  ID:3  [INFO: co2-sensor] CO2 level: 388 ppm
11:11.522  ID:6  [INFO: presence-sensor] Number of people inside the sauna: 3
11:13.268  ID:3  [INFO: co2-sensor] CO2 level: 390 ppm
11:14.523  ID:6  [INFO: presence-sensor] Number of people inside the sauna: 4
11:14.676  ID:2  [INFO: co2-sensor] CO2 level: 392 ppm
11:16.269  ID:3  [INFO: co2-sensor] CO2 level: 399 ppm
11:17.524  ID:6  [INFO: presence-sensor] Number of people inside the sauna: 3
```

Filter:

**Log Listener**

Listens to log output from all simulated motes. Right-click the main area for a popup menu with more options.

You may filter shown logs by entering regular expressions in the bottom text field. Filtering is performed on both the Mote and the Data columns.

**Filter examples:**

Hello
logs containing the string 'Hello'

^Contiki
logs starting with 'Contiki'

^[CR]
logs starting either a C or an R

Hello$
logs ending with 'Hello'

^ID:[2-5]$
logs from motes 2 to 5

^ID:[2-5] Contiki
logs from motes 2 to 5 starting with 'Contiki'

*Figure 2 - Simulation screenshot*

### 2.1.1 Presence Sensor

The presence sensor detects people entering or leaving the sauna and thus reports the number of people currently inside. The data collected is used to limit the number of entrances inside the sauna (and thus comply with COVID-19 regulations) and to save energy through smart lighting - the lights turn on only when someone is there and are left off if the sauna is empty.

The presence sensor exposes an observable resource: *presence_sensor*. This resource accepts GET requests, following which it returns the number of people currently inside the sauna. It also sends a notification to all CoAP observers whenever a change is detected.

The resource also accepts PUT requests whose payload is the new maximum number of people who can be inside the sauna at the same time. If the request is received correctly, the threshold is updated accordingly.

In order to simulate the limitation of entrances - which in reality can be achieved with a monitor at the entrance or with a system (such as a turnstile) that blocks the admissions - we use LEDs (green when it is possible to enter, red when the maximum number of people has been reached).

Also, in order to simulate the real behavior of the sensor, the number of people inside the sauna increases or decreases by 1 pseudo-randomly with some probability, while ensuring that the number of people never exceeds the maximum allowed and is never less than zero.

### 2.1.2 Light

The lights that are installed inside the sauna are smart lights, as they can be turned on or off remotely and can also change their color.

Using the data received from the presence sensor, the collector can assess whether or not it is necessary to keep the light on and then act on the light device accordingly.

In addition, through the command-line interface, users can select the color they prefer for the light and benefit from color therapy. The colors users can choose from are:

- *green*, to drive away anxiety and reach a state of relaxation and good mood;
- *yellow*, to stimulate the intellectual part of the brain and help concentration;
- *red*, to stimulate blood circulation and energize the body.

For this purpose, the light device exposes two resources: the *light_switch* and the *light_color* ones.

The *light_switch* resource accepts PUT requests with a "mode" parameter, whose value can be "ON" or "OFF". If the received request is formatted correctly, the light is turned on or off accordingly.

The *light_color* resource accepts PUT requests with a "color" parameter, whose value can be "GREEN", "YELLOW" or "RED". If the received request is formatted correctly, the color of the light is changed accordingly.

In order to visualize whether the light is on or off and what color it is, we make use of LEDs in our simulation.

### 2.1.3 Air Quality

In order to fully enjoy all the benefits of the sauna, it is essential that the environment is well ventilated. A high concentration of CO2 in the air, in fact, leads to a high concentration of CO2 in the blood of bathers which, in turn, leads to a feeling of suffocation and dizziness, and to a decline of physical and mental abilities.

An automated ventilation system ensures that the CO2 concentration in the air remains within a safe range – even when the number of people increases – without wasting energy (it only activates when necessary).

The air quality system includes a CO2 sensor and a ventilation system. It is then up to the controller to assess whether or not the ventilation system needs to be activated based on the data it receives from the CO2 sensor and to act on the ventilation system accordingly. For this, the air quality system exposes two resources: the *co2_sensor* and the *ventilation_system* ones.

The *co2_sensor* resource is an observable resource that makes measurements of the concentration of CO2 in the air (expressed in parts per million) available to clients. The resource accepts GET requests and it also notifies all CoAP observers whenever it detects a change.

The *ventilation_system* resource accepts PUT requests with a "mode" parameter, whose value can be "ON" or "OFF". If the received request is formatted correctly, the ventilation system is turned on or off accordingly.

In the context of our simulation, in order to visualize when the ventilation system is on or off, we make use of LEDs: if the LED is red, it means that the ventilation system is off, if it is green the ventilation system is on.

Also, in order to simulate the real behavior of the sensor, if the ventilation system is not turned on, the concentration of CO2 in the air can only increase with some probability and at a pseudo-random intensity. If the ventilation system is on, instead, the concentration of CO2 in the air will tend to decrease over time.

## 2.2 MQTT Network

The two main aspects of a sauna are the humidity value and the temperature level. There are different types of saunas (Finnish sauna, sanarium, Turkish bath), and the differences between them concern the temperature and humidity values. One or more sensors are required to measure humidity and temperature, depending on the size of the room.

### 2.2.1 Humidity

To manage the air humidity level, at least one humidity sensor and one actuator (humidifier / dehumidifier) are needed. If the humidifier is turned on, the humidity will tend to rise, while if the dehumidifier is turned on, the humidity will tend to drop. If the humidity is within the range set by the user, then the actuator can be turned off. To simulate the real behavior of the sensor, when the actuator is off there is still a certain probability that the humidity level will change.

The device (*/dev/ttyACM74*) that acts as a sensor and actuator communicates with the MQTT broker, sending the humidity samples and waiting for commands from the collector. The topic in which the humidity samples will be written is called *humidity*, while the one to receive the commands is called *humidifier*.

The commands that can be received are the following:

- **INC**: increases the humidity level by turning on the humidifier.
- **DEC**: decreases the humidity level by switching on the dehumidifier.
- **OFF**: do not change the humidity level, it is in the right range.

#### 2.2.2 Temperature

To manage the temperature level, at least one temperature sensor and one conditioner are needed. Through the air conditioner it is possible to both raise the temperature level and decrease it. To better simulate the real behavior of the sensor, when the air conditioner is off the temperature has a certain probability of being able to change.

The device (*/dev/ttyACM23*) that acts as a sensor and actuator communicates with the MQTT broker, publishing the samples detected on the topic *temperature*, and waiting for the commands on the topic *AC*.

The commands are the same seen for humidity:

- **INC**: increase the temperature level.
- **DEC**: decrease the temperature level.
- **OFF**: do not change the temperature level, it is in the right range.

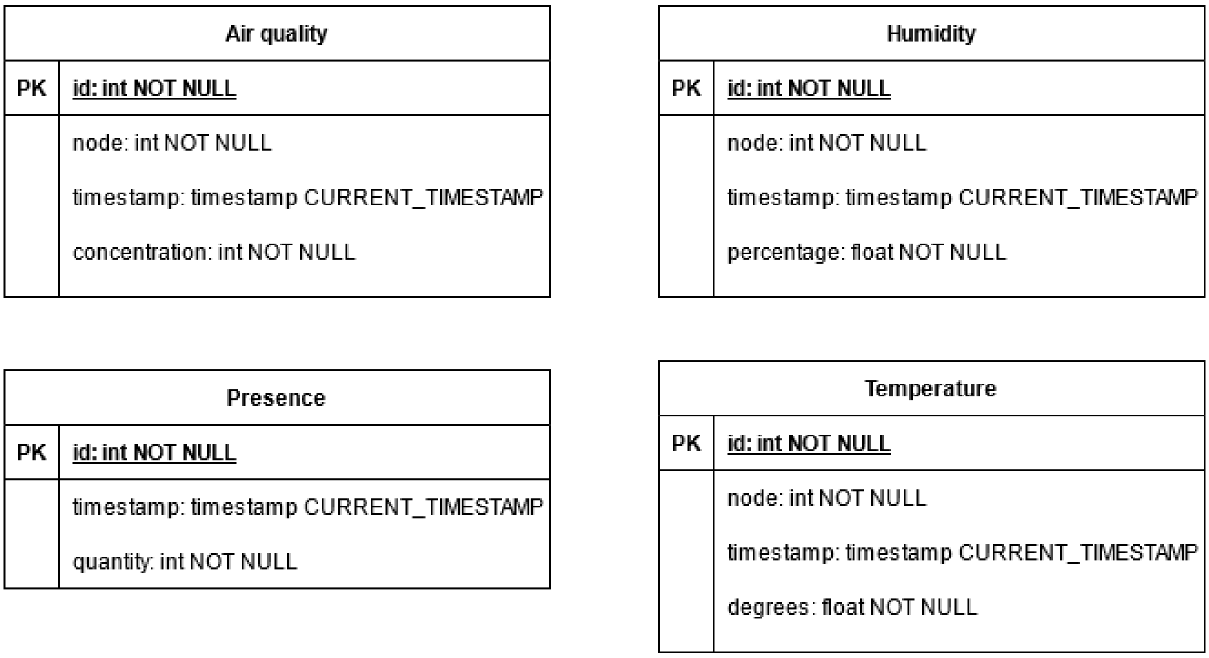
## 2.3 Data Encoding

As data encoding, we decided to use ***JSON***, all the data generated by the sensors are transmitted to the collector as *JSON* objects. This choice is due to the fact that sensors have limited resources, and *XML* has a too complex structure for our needs. *JSON* is more flexible and less verbose, resulting in less overhead. *JSON* is a *text-based* format, it would have been better to use a binary encoding (*CBOR*), but the libraries on Contiki are very recent and still not complete.

## 2.4 Database

It is essential to store the data collected with the sensors, also to be able to analyze them through Grafana. The database (*smart_sauna*) is particularly simple, we have defined a table for each type of measurement (humidity, temperature, etc.), and there are no dependencies between the tables.

For both humidity and temperature and air quality we can have multiple devices, so in the tables it is necessary to enter the identifier of the node from which we have received the sample.



| | Air quality |
|---|---|
| PK | id: int NOT NULL |
| | node: int NOT NULL |
| | timestamp: timestamp CURRENT_TIMESTAMP |
| | concentration: int NOT NULL |

| | Humidity |
|---|---|
| PK | id: int NOT NULL |
| | node: int NOT NULL |
| | timestamp: timestamp CURRENT_TIMESTAMP |
| | percentage: float NOT NULL |

| | Presence |
|---|---|
| PK | id: int NOT NULL |
| | timestamp: timestamp CURRENT_TIMESTAMP |
| | quantity: int NOT NULL |

| | Temperature |
|---|---|
| PK | id: int NOT NULL |
| | node: int NOT NULL |
| | timestamp: timestamp CURRENT_TIMESTAMP |
| | degrees: float NOT NULL |

*Figure 3 - Data Model*

## 2.5 COLLECTOR

The collector is the real heart of our architecture, it takes care of receiving the data, communicating with the database to save the samples received and decide which actions to take. It also implements a CLI to make it easier for the user to manage. We decided to implement the collector in *Java*, taking advantage of the *Paho* and *Californium* libraries.

### 2.5.1 CLI

The functions made available to the user through the CLI are the following:

- **!help <command>**: shows the details of a specific command.
- **!get_humidity**: get the humidity level of the sauna. If there are multiple humidity sensors, the global humidity level is calculated by averaging.
- **!set_humidity <lower bound> <upper bound>**: sets the range within which the humidity must stay.
- !**get_temperature**: get the temperature level of the sauna. If there are multiple temperature sensors, the global temperature level is calculated by averaging.
- **!set_temperature <lower bound> <upper bound>**: sets the range within which the temperature must stay.
- **!get_air_quality**: get the air quality level of the sauna. If there are multiple air quality sensors, the global level is calculated by averaging.
- **!set_air_quality <upper bound>**: sets the limit below which the $C0_2$ concentration must stay.
- !**set_color <color>**: sets the light color (*GREEN*, *YELLOW* or RED). As already mentioned, by changing the color of the lights it is possible to apply chromotherapy inside the sauna.
- **!get_number_of_people**: retrieves the number of people inside the sauna.
- !**set_max_number_of_people <number>**: sets a limit on the number of people allowed inside the sauna.
- **!exit**: terminates the program.

### 2.5.2 Main packages and classes

For a better organization of the code, we have defined several packages, in this section we will define the meaning of these packages and the classes they contain.

#### *2.5.2.1 it.unipi.dii.inginf.iot.smartsauna.app*

This package contains the main class, that starts the application. Classes:

- **SmartSaunaCollector**: This class implements all the functions of the CLI made available to the user.

#### *2.5.2.2 it.unipi.dii.inginf.iot.smartsauna.coap*

This package is used to manage the communication with the various IoT devices within the network whose application protocol is CoAP. Classes:

- **CoapRegistrationServer**: this class extends the CoapServer class, an execution environment for CoAP Resources. In particular, this server has a resource of type *CoapRegistrationResource*. This is also the class that interfaces between the CLI and the classes related to each IoT device in the CoAP network. Specifically, it allows to translate each command received from the CLI into the respective method.
  - ***CoapRegistrationResource*** is an inner class that extends CoapResource. This resource (“registration”) allows the registration of IoT devices to the server via a POST request and the deregistration via a DELETE request.

###### 2.5.2.2.1 it.unipi.dii.inginf.iot.smartsauna.coap.devices
Classes:

- **CoapDevicesHandler**: this class is used to manage the different IoT devices and, in particular, to dispatch the various requests to the right device.

###### *2.5.2.2.1.1 it.unipi.dii.inginf.iot.smartsauna.coap.devices.airquality*
Classes:

- **AirQuality**: this class handles communication with the IoT device(s) that is/are responsible for detecting the CO2 concentration in the air and turning the ventilation system on or off. In particular, this class contains the control logic for the activation of the ventilation system: if, by observing the data detected by the sensor(s), it is noticed that the CO2 level exceeds the desired limit, the ventilation system is activated and remains in operation until the level is sufficiently lowered.

###### *2.5.2.2.1.2 it.unipi.dii.inginf.iot.smartsauna.coap.devices.light*
Classes:

- **Light**: this class handles the communication with the IoT device(s) that take(s) care of turning on/off the light and changing its color.

###### *2.5.2.2.1.3 it.unipi.dii.inginf.iot.smartsauna.coap.devices.presence*
Classes:

- **PresenceSensor**: this class manages the communication with the IoT device that detects the presence or absence of people inside the sauna. It also implements control logic that allows for intelligent light management: the lights inside the sauna are only turned on if someone is actually present.

#### *2.5.2.3 it.unipi.dii.inginf.iot.smartsauna.config*
This package is used to handle the configuration parameters, store in *config.xml*. The schema for the validation is in *config.xsd*. The validation is very important to be sure of the correctness of the *config.xml* file. Classes:

- **ConfigurationParameters**: this class stores all the configuration parameters needed by the application. For example, the IP of the MySQL Database, and the default thresholds. These values do not need to be changed, so only get methods are provided. We used the singleton desing pattern for this class, to read the file only once.

#### *2.5.2.4 it.unipi.dii.inginf.iot.smartsauna.log*
In this package the logging functions have been developed, necessary both in the debugging phase and for the final application. Thanks to the log it will be possible to understand the commands that have been given by the collector, and the reason. Classes:

- **Logger**: This class defines the Logger object, which will be exploited by a good part of the application modules. This class is also Singleton, this is useful for avoiding shared access to the log file.

#### 2.5.2.5 *it.unipi.dii.inginf.iot.smartsauna.model*
This package contains the classes required for the model. These classes are the Java bean for our application. The JSON objects we receive from the nodes will be transformed into these model objects. Classes:

- **AirQualitySample**: this class stores all the information about a sample received by an Air Quality sensor, like the identifier of the node and the value observed.
- **HumiditySample**: this class stores all the information about a sample received by a Humidity sensor, like the identifier of the node and the value observed.
- **PresenceSample**: this class stores all the information about a sample received by a Presence sensor, like the identifier of the node and the value observed.
- **TemperatureSample**: this class stores all the information about a sample received by a Temperature sensor, like the identifier of the node and the value observed.

#### 2.5.2.6 *it.unipi.dii.inginf.iot.smartsauna.mqtt*
This package contains the classes required to implement the MQTT client. Some sensors communicate with the collector through an MQTT broker, so it is necessary to implement an MQTT client that can collect data and give commands to the actuators.

Classes:

- **MQTTHandler**: This class implements the connection with the MQTT broker, receiving the topic *humidity* and *temperature* messages, and sending messages to the topic *humidifier* and *AC*.

##### 2.5.2.6.1 it.unipi.dii.inginf.iot.smartsauna.mqtt.devices
This package contains one package for each type of device that communicates with the MQTT broker. We need some classes for storing the data received and decide what action implement.

###### 2.5.2.6.1.1 *it.unipi.dii.inginf.iot.smartsauna.mqtt.devices.humidity*
In this package there are the classes needed to store the data received from humidity sensors.

Classes:

- **HumidityCollector**: This class keeps the most recent samples of each humidity sensor, in order to compute an average and decide what action to perform on the actuators.

###### 2.5.2.6.1.2 *it.unipi.dii.inginf.iot.smartsauna.mqtt.devices.temperature*
In this package there are the classes needed to store the data received from humidity sensors.

Classes:

- **TemperatureCollector**: This class keeps the most recent samples of each temperature sensor, in order to compute an average and decide what action to perform on the actuators.

#### 2.5.2.7 *it.unipi.dii.inginf.iot.smartsauna.persistence*
This package deals with managing the persitency of data, indeed it contains the class used to interface with the database. Classes:

- **DBDriver**: this class is responsible for the implementation of all the queries for MySQL. We used the Singleton desing pattern, because a single instance of this driver must be shared by all application classes.

## 2.6 Grafana

We have implemented a dashboard on Grafana in order to be able to monitor in real time the data we store in the database, and therefore to be able to view the trend of the monitored parameters. More precisely, we have created a panel for humidity, one for temperature, one for air quality and one for presence. Through this dashboard you can see how the measured values remain within the established ranges.

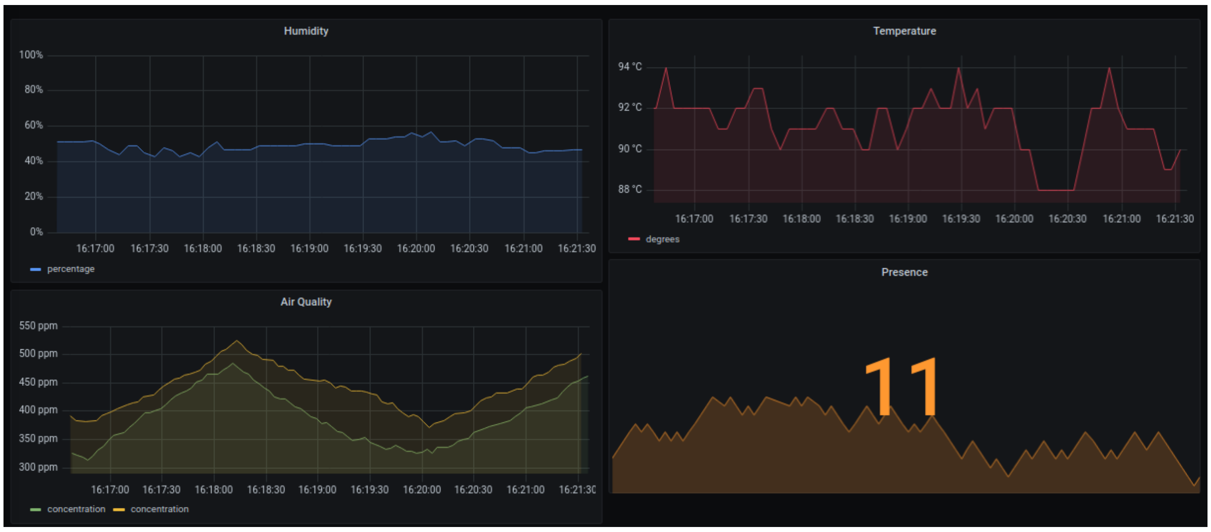


*Figure 4 - Dashboard screenshot*

In this example we have two sensors for monitoring the air quality, so we must take into account the mean value of the two curves, that has to stay below a maximum threshold (see the peak).