University of Pisa

Internet of Things

Project

AirOxyCheck

Controlled atmosphere with oxygen reduction for fire prevention

Bruchi Caterina
De Marco Angelo

A.Y. 2022-2023

Contents

Chapter 1

Use Case

Fire Trangle

The fire triangle is a fundamental concept in fire science and safety that helps us understand the three elements necessary for a fire to occur: fuel, heat, and oxygen. These three components form the sides of a triangle, and the presence of all three is required for combustion to take place.

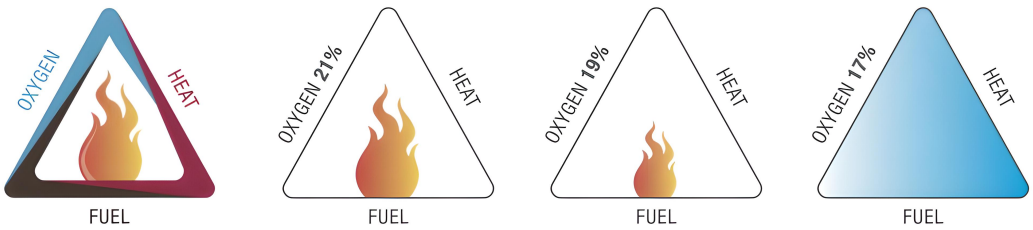


Figure 1.1: Fire triangle

1. Fuel: any material that can burn and sustain a fire, in out case is represented by the materials contained inside the environment we are monitoring. Different fuels have different ignition points and burn at varying rates.

2. Heat: is the energy necessary to raise the temperature of the fuel to its ignition point. Heat sources such as flames, sparks, electrical arcs, or even high temperatures can provide the initial heat required to start a fire. Once a fire is ignited, heat is generated and sustains the combustion process.

3. Oxygen: Oxygen is an oxidizer that supports the chemical reaction of combustion. It is present in the air we breathe and is essential for fires to occur. It combines with the fuel to produce heat, light, and combustion byproducts. Adequate oxygen levels allow fires to burn more intensely.

The fire triangle illustrates that removing any one of these elements can disrupt the combustion process and extinguish the fire. Fire safety measures focus on controlling or eliminating at least one side of the triangle to prevent fires or suppress existing ones.

In our implementation we will be working on the latter 2 sides of the triangle.

Oxygen Reduction System (ORS)

In particular we will realize an IoT Oxygen Reduction Systems(ORS), utilize the principle of reducing or depleting the oxygen content in the protected area's atmosphere, preventing the sustenance of combustion due to insufficient oxidizing agent. Also to sustain the fire prevention we will also monitor temperature as the second side of the triangle interrupting the cycle so that even in the presence of an ignition source, the fuel never reaches the conditions necessary for combustion to occur.

ORS Controlled atmosphere technology for fire protection

Controlled atmosphere technology for fire protection has reached its full maturity. For medium and large-scale storage applications, oxygen reduction systems (ORS) often compete favorably against automatic control and extinguishing systems that use traditional or clean agents.

The ORS system takes advantage of its ability to maintain the protected environment at an oxygen concentration below the "Limit Oxygen Concentration" (LOC) for each type of combustible material present in the area. The as mentioned before LOC is a characteristic value for each material and represents the limit oxygen concentration below which combustion cannot be sustained due to a lack of oxidizing agent, even in the presence of a continuous ignition source.

This limit oxygen concentration typically ranges from 17% to 13% depending on the type of material, in the standard *UNI EN 16750* there are value tables containing the appropriate oxygen concentration to maintain at a given temperature to keep the environment safe and the minimum number of oxygen sensors in each protected area to monitor and control the oxygen concentration.

Therefore, we are developing an integrated system that combines fire prevention and protection into one.

The same standard also provides indication on the suitable safety regulations in each range of oxygen, making it so that the environment protected with such systems are not closed and dangerous for workers. Proper safety measures, including organizational protocols, enable regulated access for workers in low oxygen environments without significant inconvenience.

Volume in m^3		Minimum amount of measuring zones	Minimum amount of oxygen sensors
from	to		
> 0	500	1	3
> 500	4 000	4	4
> 4 000	10 000	6	6
> 10 000	25 000	8	8
> 25 000	50 000	10	10
> 50 000	100 000	12	12
> 100 000	200 000	14	14
> 200 000	300 000	16	16
> 300 000	400 000	18	18
> 400 000		Case by case evaluation	Case by case evaluation

Figure 1.2: Number of sensors requires for m2

Advanatages

Unlike other fire control systems, it doesn't rely on the combustion itself to activate and so the application is particularly aligned with the concepts of *business continuity*, since it takes into account the preservation of the materials and is therefore suitable for protecting high-value assets or assets sensitive to damage from water, foam, smoke, and/or heat.

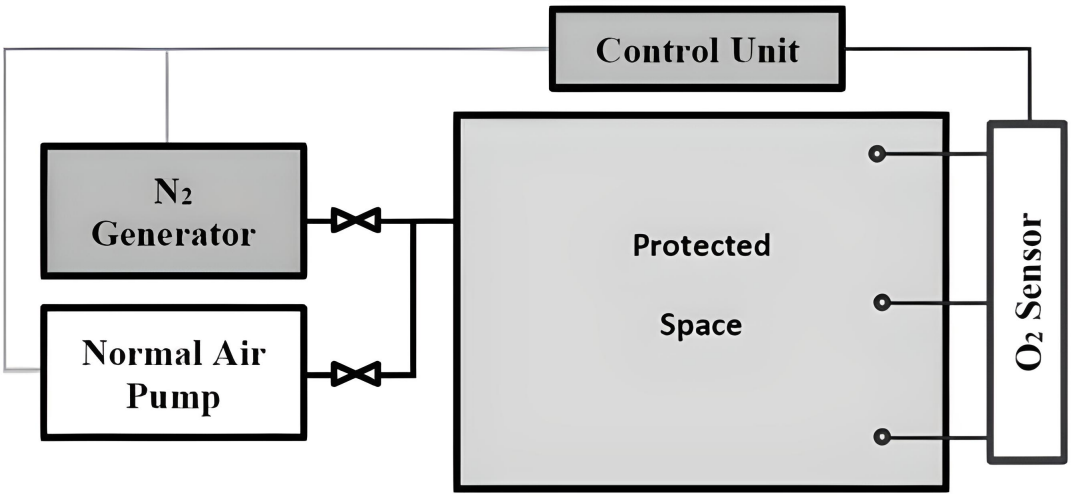


Figure 1.3: ORS Structure

The system is only composed of a generator, a fan and some sensor; usually the nitrogen is produced directly on-site from external air making the system close-loop, so there are the benefits of a "clean" technology that is completely free from issues such as contamination, invasiveness, false interventions, and the need for periodic agent verification or replacement.

Real use cases

The ORS prevention system stands out for its ability to operate in high-risk areas or complex situations where traditional fire protection systems cannot ensure a high level of safety or are not compatible with the required quality standards of the protected environment. It is a highly suitable system for protecting enclosed automated warehouses where continuous personnel presence is not required, like 4.0 industries.

The oxygen reduction system is particularly suitable in cases where technical, architectural, or aesthetic constraints do not allow the installation of conventional fire protection systems. It is also ideal for special environments where the assets to be protected have extremely high economic and/or cultural value, such as historical archives and museum areas, where other fire protection systems could cause significant damage to the valuable assets.

Some examples are:

1. Intensive automated warehouses
2. Cold rooms
3. Data centers

4. Archives and storage facilities
5. Refrigerated warehouses
6. Vaults
7. Pharmaceutical storage facilities

Chapter 2

Implementation

As anticipated in our application we will monitor both the oxygen level and the temperature. In particular we fixed our threshold at average well-known values, but the requirements might change depending on the specific use case:

- Temperature to be maintained: around 20°
  (we will allow values within : 18° - 21°)
- Oxygen to be maintained: around 12.5%
  (We will allow percentage between: 10% - 15%)

2.1 Structure

We will implement 1 RPL border router and for our sensor/actuator network:

- 3 sensors
  - 2 temperture sensors
  - 1 oxygen sensor
- 2 actuators
  - 1 fans
  - 1 nitrogen generator
- 1 border router (directly gathered from *examples/rpl-border-router* provided during lessons.

Then there will be 2 different applications: 1 instance of cloud application and a user control interface. And an instance of database. The overall structure can be found at (2.1).

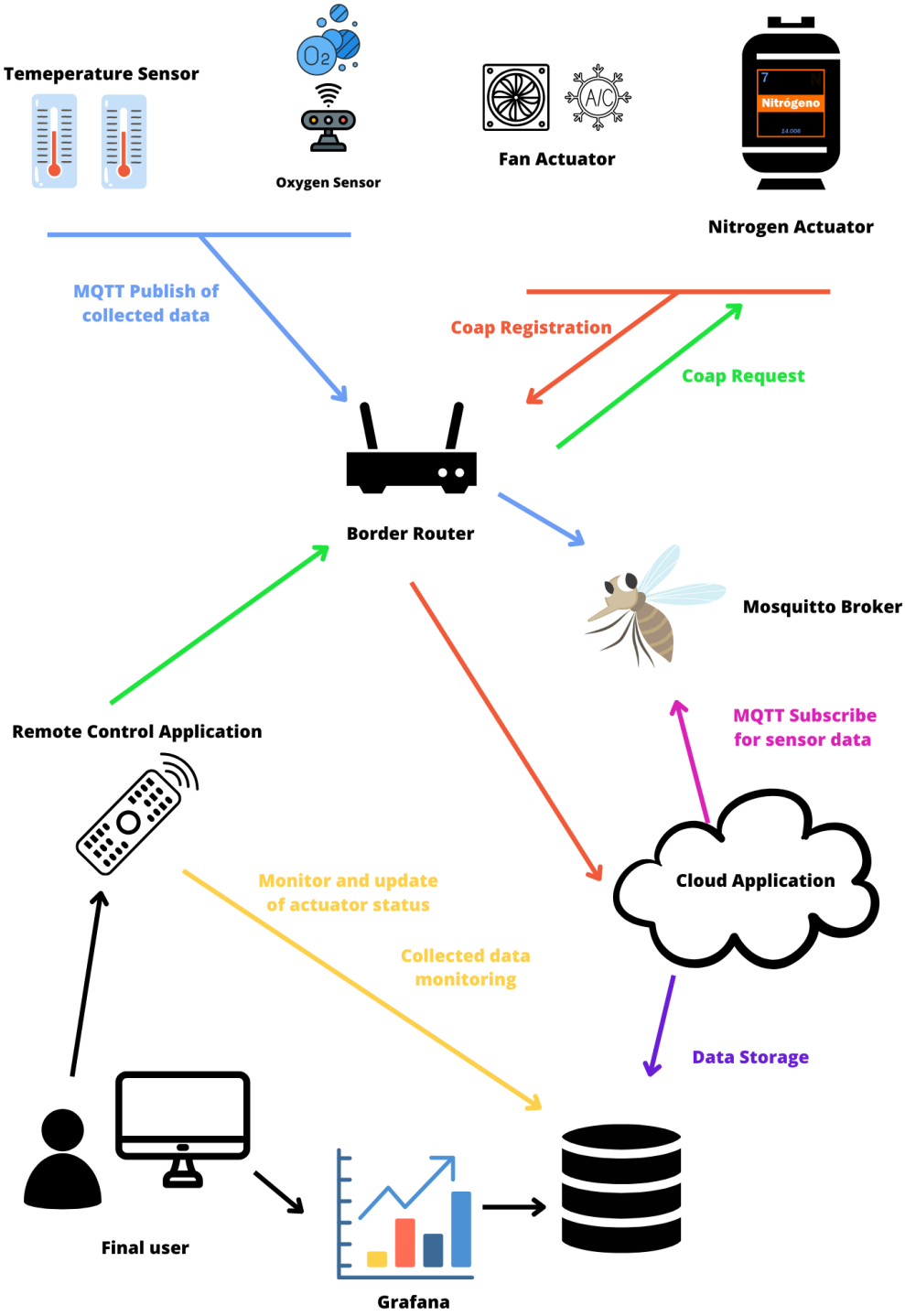


Figure 2.1: Structure Schema

2.2 Workflow

1. The sensors are constantly sending data to the Cloud Application which registers them all on the database, until the data are in the correct ranges no action is performed, when critical values are registered in the last 3 minutes

2. Actuators must be send a registration request to the cloud application before to actually be used

3. the cloud application activate a function to handle the criticality:

    - MAX OXYGEN LIMIT SURPASSED
      The application first checks on its *Actuators* table
        - if the fan for air rechange are active disables them as a initial measure
        - if they aren't then it turns on the nitrogen generator
    - MIN OXYGEN LIMIT SURPASSED
      The application first checks on its *Actuators* table
        - if the nitrogen generator are active disables them as a initial measure
        - if they aren't then it turns on the fan
    - MIN or MAX TERMPERATURE LIMIT SURPASSED
      The application first checks on its *Actuators* table
        - The fan temperature is increased or decreased depending on the need

2.3 Energy Conservation

We tried to save as much as possible the energy of the sensors by:

- Sensing the sensor measurements partially aggregated, we send the average each 5 sampling

- We save on the database a table containing the status of each actuator present in the network, so that when handling critical situation we can have a complete view of the system without having to make a GET on the devices

2.4 Database

We have 2 tables in our database, one that registers the actuators and their status, the status is initialized at registration and updated at every modification done by the cloud application in the environment handling and the ones done by the user itself to modify the parameters or do other action.

ACTUATORS

The columns are:

- **IDActuator**:
  Is an incremental numeric ID that identifies each actuator, for a more human friendly way to identify the devices.
- **IPActuator**:
  Is the actual IP of the device.
- **Type**:
  In our system we have 2 actuators, the fan and the nitrogen generator
- **Status**:
  Since we deal with different type of actuators here is saved a vector with all the involved parameters for the type of the actuator, which will be for the "fan" power and temperature and for the "nitro" on/off.

MEASUREMENTS

The columns are:

- **IDSensor**:
  The sensor which sent the measurement. Is assigned to the device using the library "sys/node-id.h"
- **Sector**:
  The sector in which such sensor is placed
- **Value**:
  The value measured by the sensor
- **Topic**:
  The physical unit registered (ex. oxygen, temperature)
- **Timestamp**:
  The timestamp of the observed measurements

2.5 Human Control

The actions the user can perform are:

- Change the critical measurements thresholds
- View a list of the actuators possibly filtered by sector and/or type
- View the measurements of the last 3 minute for both oxygen and temperature

2.6 Leds and Buttons

This is how we used the buttons and leds in the nodes:

Actuators

- Leds:
  The led will be
  - Green: When the device is registered
  - Red: When the device is unregistered
- Button:
  The button in this case applies a unregistration or registration to COAP server (on the Cloud Application) depending on the state of the device

Sensors

- Leds :
  The led will be
  - Green: When the sensor is correctly connected to MQTT broker
  - Red: When the sensor is disconnected from MQTT broker
- Buttons:
  The button will serve to change the routing of the data sent from regular values to critical ones (this was done for a comfortable demo, in a real environment the use will be simple on/off button). For simulation purposes, the number of times the button has been pressed modulo 3 decides the mode: (0) Correct range, (1) Over Threshold, (2) Under Threshold.

Appendix A

Java Projects

A.1 Remote Control Application

Remote Control Application (its package structure is in (A.1)) aims to reach two main objectives of the project:

- Provide an user interface with some basic operations to test DB iterations
- Interact with the database in order to discover critical sectors from both point of view of oxygen and temperature, and make actuators do their job acccordingly.

In this section, we focus on the second point of this list. We analyse how Controllers detect a critical sector, how DB is accesses and how actuators are called.

A.1.1 Controllers

Controllers are threads that periodically monitor the status of the parameter they are assigned to. In particular, there is a class *Controller.java* that periodically interact with the DB to check if there are some "*critical sectors*" (A.2), which are sectors in which something dropped below the threshold or went above it.

The object Controller is never instantiated as a "stand-alone" object, but It is specialised in one of two sub-classes: *OxygenController.java* and *TemperatureController.java*. In this way we wrapped the thread and monitoring logic inside to the father's code, in order to have less duplicated code, and the specialised control logic inside the sub classes (A.3) (A.4).

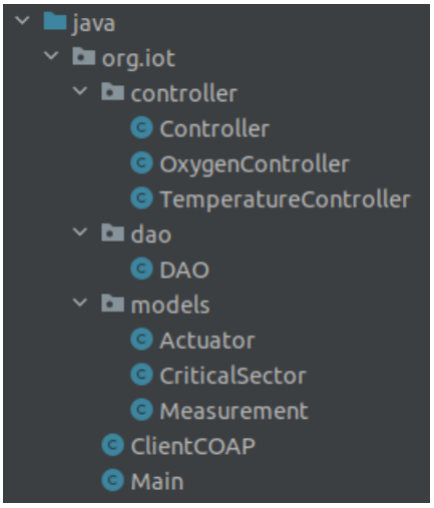


Figure A.1: Remote Control Application package structure

```
while(rs.next()){
    if(rs.getDouble(2) > threshold_M || rs.getDouble(2) < threshold_m)
        // CRITICAL SECTOR : <index_sector, aggregated_value>
        result.add(new CriticalSector(rs.getInt(1), rs.getDouble(2)));
}
```

Figure A.2: How critical sectors are detected by Controller.java

Then, when the control logic evaluates the operation to do, a method from *COAPClient.java* is invoked. In this way, a COAP [POST/PUT/DELETE] request is sent to the actuators (A.5)(IP is stored inside the database) and in case of success the database is correctly updated. In this way, no GET requests are required, because the state of tables is always consistent with the last update requested from the application (power management for actuators).

A.2 Cloud Application

The cloud application handles:

- The registration of the actuators (A.10)
- The reception via MQTT of the data of the sensors and their storage
- The monitoring of the parameters and related operations, so the configuration and setting of the actuators

```java
public void helpSector(CriticalSector cs) {
    int sector = cs.getSector();
    double value = cs.getValue();
    System.out.println("Sector Oxygen : " + sector + " : " + value + "%");


    if(value > MAX_THRESHOLD){
    System.out.println("Trying to reduce oxygen");
        reduceOxygen(sector);
    }

    if(value < MIN_THRESHOLD){
    System.out.println("Trying to increase oxygen");
        increaseOxygen(sector);
    }
}
```

Figure A.3: Example of Control Logic (1)

A.2.1 MQTT Collector

The run function (A.6) of the Collector first establishes the connection with the Broker and then subscribe to both the topic *oxygen* and *temperature*

The MQTT Collector receives the data from the MQTT Broker triggering this function (A.7)

The function parse the message and insert it into a Measurements class with this structure (A.8)

A.2.2 COAP Server and Registration

There is a server exposing and handling the *registration* resource which has the following methods:

GET

Retrieves the whole status of the resource its called on.

POST

Receives from the actuator a String containing the initial status of the device, parses the IP from the exchange and calls the *registerActuator* function (A.9)

```java
private void reduceOxygen(int sector) {
    ArrayList<Actuator> actuators = new DAO().readActuatorST("fan", sector);
    boolean fanOn = false;
    for(int i = 0; i < actuators.size(); i++){
        // Logica di aumento ossigeno
        if(Integer.parseInt(actuators.get(i).getStatus().substring(0,1)) > 0){
            ClientCOAP.changeParam(actuators.get(i), "power", 0);
            fanOn = true;
        }
    }
    if(fanOn)
        return;

    ArrayList<Actuator> actuators_nitro = new DAO().readActuatorST("nitro", sector);
    for(int i = 0; i < actuators_nitro.size(); i++){
        // Logica di aumento ossigeno
        if(actuators_nitro.get(i).getStatus().equals("ON")){
            System.out.println("ALERT! No contain measures available!");
            break;
        }
        ClientCOAP.enable(actuators_nitro.get(i));
    }

}
```

Figure A.4: Example of Control Logic (2)(OxygenController.java)

```
public static void  incrDecrParam(Actuator a, String param, int value){
    String originalStatus[] = a.getStatus().split( regex: ", ");
System.out.println("Value passed : " + value);
System.out.println("Trying to [1 = increment, -1 = decrement] " + param + " in fan actuator");
    CoapClient client = new CoapClient( uri: "coap://[" + a.getIPActuator() + "]:5683/fan");
    CoapResponse response = client.put( payload: param + "=" + value, MediaTypeRegistry.TEXT_PLAIN);

    System.out.println("CODE [PUT REQUEST] : " + response.getCode());
    if(response.getCode() == CoAP.ResponseCode.CHANGED){
        // Cambia il json di a in maniera opportuna
        if(param.equals("temperature")){
            if(value == 1)
            // Cambio il secondo numero della stringa status dentro a
                originalStatus[1] = String.valueOf( i: Integer.parseInt(originalStatus[1]) + 1);
            else
                originalStatus[1] = String.valueOf( i: Integer.parseInt(originalStatus[1]) - 1);
        }
        else{
            if(value == 1) {
                // Cambio il secondo numero della stringa status dentro a
                originalStatus[0] = String.valueOf( i: Integer.parseInt(originalStatus[0]) + 1);
            }
            else
                originalStatus[0] = String.valueOf( i: Integer.parseInt(originalStatus[0]) - 1);
        }

        a.setStatus(originalStatus[0] + ", " + originalStatus[1]);
        new DAO().writeActuator(a);
    }
}
```

Figure A.5: How a COAP request is sent and how response is handled (Increase/Decrease parameter)

```java
public void run(){

    try {
        sleep( millis: 3000);
        MqttClient mqttClient = new MqttClient( serverURI: "tcp://127.0.0.1:1883",  clientId: "CLOUD_APPLICATION");
        mqttClient.setCallback(this);
        mqttClient.connect();
        mqttClient.subscribe( topicFilter: "oxygen");
        mqttClient.subscribe( topicFilter: "temperature");
        System.out.println("MQTT Collector successfully Connected and Subscribed!");
    }
    catch(MqttException me){
        System.out.println("Error " + me.getMessage());
        Main.restartSubscriber();
        System.exit( status: 1);
    }
    catch(Exception e){
        System.out.println("Error " + e.getMessage());
        Main.restartSubscriber();
        System.exit( status: 1);
    }
    while(true)
        ;
}
```

Figure A.6: Run function of the MQTT Broker

```java
public void messageArrived(String topic, MqttMessage message) throws Exception {
    // TODO Auto-generated method stub
    System.out.println("Message arrived!");
    String payload = new String(message.getPayload());
    Gson gson = new Gson();
    Measurement measure = gson.fromJson(payload, Measurement.class);

    LocalDateTime current = readTime();
    current = current.plusSeconds(Long.parseLong(measure.getTimestamp()));
    DateTimeFormatter formatter = DateTimeFormatter.ofPattern("yyyy-MM-dd HH:mm:ss");
    String strDate = formatter.format(current);
    measure.setTimestamp(strDate);


    System.out.println("Trying to register measure :");
    System.out.println(measure);

    save(measure);

}
```

Figure A.7: Handling of an incoming MQTT message

```
public class Measurement {
    private int IDSensor;
    private int sector;

    private double value;

    private String topic;
    private String timestamp;
```

Figure A.8: Measurement Class attributes

Register Actuators

Registers the Actuator data in the Database instantiating a new Actuator object and calling the DAO method.

DELETE

The delete method handles the delete request made by the actuator itself trough the button pression and returns the result. (A.13)

A.2.3 DAO

Contains the connection with the database and the queries needed.

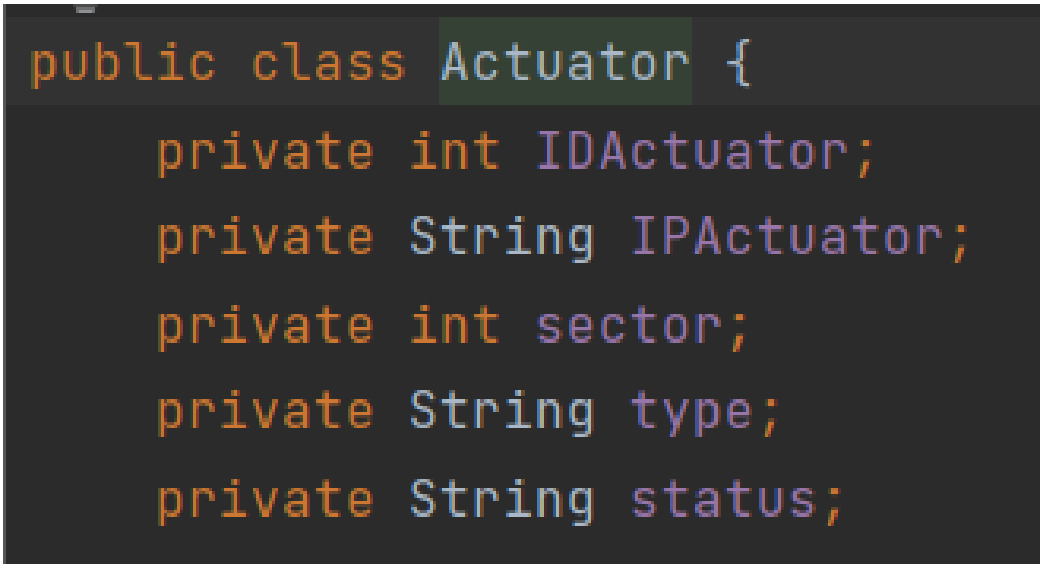
DAO handles the Actuator class (A.11), in particular the field status that is a comma-separated string containing the current value of all the parameters of the actuator.

```
public void handleGET(CoapExchange exchange) {

    Response response = new Response(ResponseCode.CONTENT);

    response.setPayload("OK\n");

    exchange.respond(response);
}

public void handlePOST(CoapExchange exchange) {
    String parameters = exchange.getRequestText();

    System.out.println("Received a POST request\n" + parameters);

    System.out.println("FROM" + exchange.getSourceAddress().toString());
    actuatorIP = exchange.getSourceAddress().toString();
    actuatorIP = actuatorIP.substring( beginIndex: 1);
    Response response = new Response(ResponseCode.CONTENT);
    response.setPayload("OKREG");
    exchange.respond(ResponseCode.CREATED);
System.out.println("Trying to register actuator");
    registerActuator(parameters);
}
```

Figure A.9: GET and POST method of the Actuator Resource

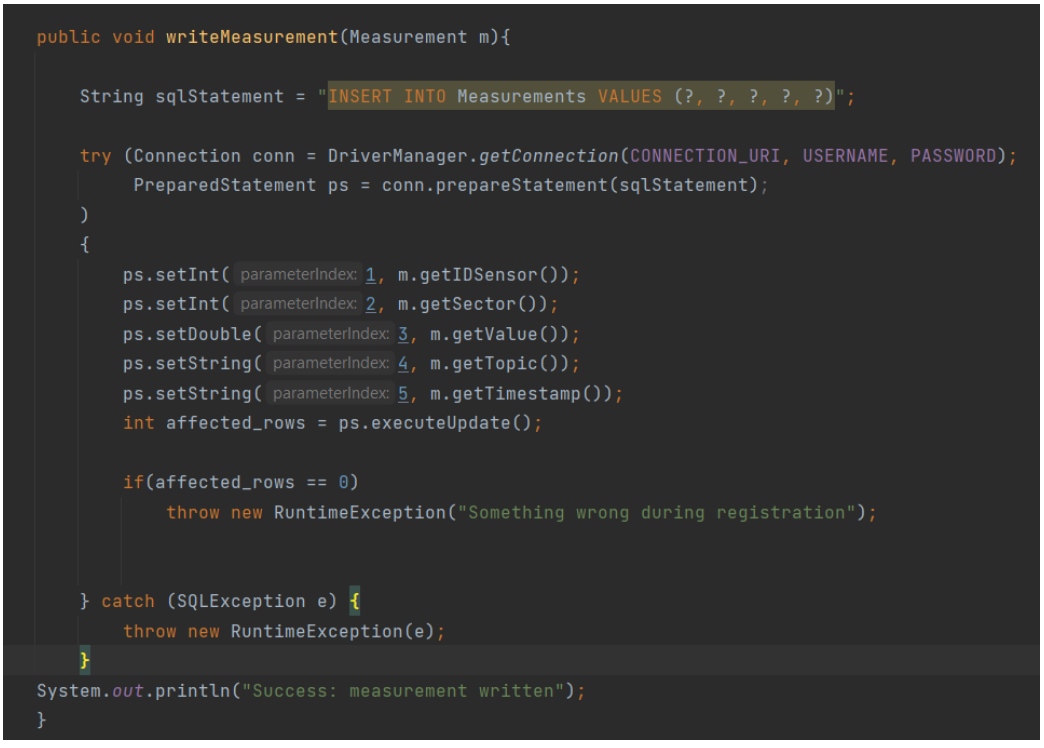
```
public void handleDELETE(CoapExchange exchange) {

    actuatorIP = exchange.getSourceAddress().toString();

    if(new Dao().deleteActuatorIP(actuatorIP.substring( beginIndex: 1))){
        System.out.println("DELETE OK");
    }else{
        System.out.println("DELETE ERROR");
    }
    Response response = new Response(ResponseCode.CONTENT);

    response.setPayload("OKNREG");
    exchange.respond(ResponseCode.DELETED);

}

public void registerActuator(String parameters){
    Gson gson = new Gson();
    Actuator measure = gson.fromJson(parameters, Actuator.class);
    measure.setIDActuator(nextActuatorID++);
    measure.setIPActuator(actuatorIP);
    new Dao().writeActuator(measure);
}
```

Figure A.10: Delete and register Actuator functions



```
public class Actuator {
    private int IDActuator;
    private String IPActuator;
    private int sector;
    private String type;
    private String status;
```

Figure A.11: Actuator Class



```
public void writeMeasurement(Measurement m){

    String sqlStatement = "INSERT INTO Measurements VALUES (?, ?, ?, ?, ?)";

    try (Connection conn = DriverManager.getConnection(CONNECTION_URI, USERNAME, PASSWORD);
         PreparedStatement ps = conn.prepareStatement(sqlStatement);
    )
    {
        ps.setInt( parameterIndex: 1, m.getIDSensor());
        ps.setInt( parameterIndex: 2, m.getSector());
        ps.setDouble( parameterIndex: 3, m.getValue());
        ps.setString( parameterIndex: 4, m.getTopic());
        ps.setString( parameterIndex: 5, m.getTimestamp());
        int affected_rows = ps.executeUpdate();

        if(affected_rows == 0)
            throw new RuntimeException("Something wrong during registration");


    } catch (SQLException e) {
        throw new RuntimeException(e);
    }
System.out.println("Success: measurement written");
}
```

Figure A.12: Write of a Measurement on the database

```java
public boolean deleteActuatorIP(String IP){

    String sqlStatement = "DELETE FROM Actuators WHERE IPActuator = ?";
    boolean result = false;
    try(Connection conn = DriverManager.getConnection(CONNECTION_URI, USERNAME, PASSWORD);
        PreparedStatement ps = conn.prepareStatement(sqlStatement);
    )
    {
        ps.setString( parameterIndex: 1, IP);
        int RowsAffected = ps.executeUpdate();

        if(RowsAffected==0){
            System.out.println("C'è stato un errore!");
            return false;
        }
    } catch (SQLException e) {
        throw new RuntimeException(e);
    }
System.out.println("Correctly DELETED");
    return true;
```

Figure A.13: Delete of the actuators from the database

```java
public void writeActuator(Actuator a) {
    String sqlStatement = "REPLACE INTO Actuators VALUES (?,?,?,?,?)";
    try (Connection conn = DriverManager.getConnection(CONNECTION_URI, USERNAME, PASSWORD);
         PreparedStatement ps = conn.prepareStatement(sqlStatement);
    )
    {
        ps.setInt( parameterIndex: 1, a.getIDActuator());
        ps.setString( parameterIndex: 2, a.getIPActuator());
        ps.setInt( parameterIndex: 3, a.getSector());
        ps.setString( parameterIndex: 4, a.getType());
        ps.setString( parameterIndex: 5, a.getStatus());
        int affected_rows = ps.executeUpdate();
        if(affected_rows == 0)
            throw new RuntimeException("Something wrong during registration");

    } catch (SQLException e) {
        throw new RuntimeException(e);
    }
System.out.println("Success: Actuator registered");
}
```

Figure A.14: Write of an Actuator on the database