UNIVERSITÀ DEGLI STUDI DELL'INSUBRIA

DEPARTMENT OF THEORETICAL AND APPLIED SCIENCES

MASTER'S DEGREE IN
COMPUTER SCIENCE

Smart Home Automation & Monitoring

Sassi Gabriele - Centore Luca

Contents

List of Figures

List of Tables

Chapter 1

Scenario

1.1 Overview

The IoT project focuses on implementing smart home automation and security systems for domiciliary residence. This residence is composed by two different smart home (A-B). These smart home are equipped with a range of sensors and devices designed to enhance energy efficiency, temperature control, and security measures.

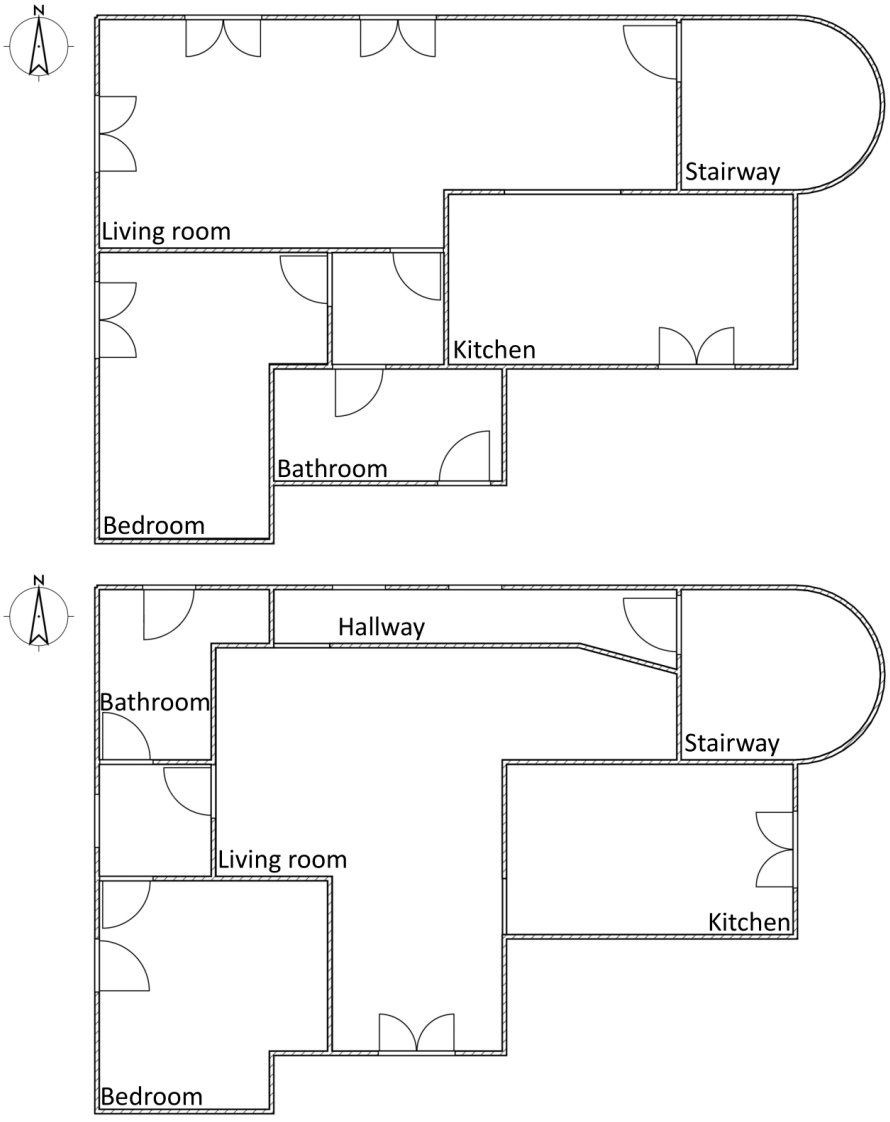


Figure 1.1: Smart home A-B structure

1.2 Goal

The primary objective of the IoT project is to create intelligent living spaces that optimize energy usage, maintain comfortable temperature levels and ensure robust security measures. By integrating advanced technologies, the project aims to provide occupants with a seamless and convenient living experience while promoting sustainability and safety.

1.3 Functionalities

The main functionalities are:

- **Automated Lighting Control**;
- **Automated Heating Control**;
- **Automated HVAC Control**;
- **Alarm and Intrusion Detection System**;
- **Energy Consumption Monitoring**.

1.3.1 Automated Lighting Control

- Each room is equipped with motion sensors to monitor occupancy.
- When someone is present in the room, the lighting systems automatically turns on the lights.
- The lighting system can automatically adjust the brightness of the lights based on user preferences and environmental conditions.

This feature enhance convenience by eliminating the need for manual switching, and also contributes to energy savings by ensuring that lights are only on when needed.

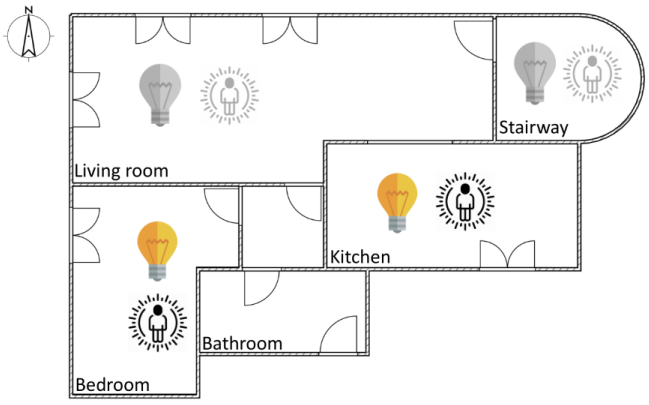


Figure 1.2: Lighting regulation example

1.3.2 Automated Heating Control

- Specific sensors track temperature levels inside the smart home.
- Based on temperature readings, the system autonomously activates the heating when the temperature drops below the set point, which represent the user-defined ideal temperature.

By dynamically responding to temperature changes, the system ensures a comfortable living environment while maximizing energy efficiency.

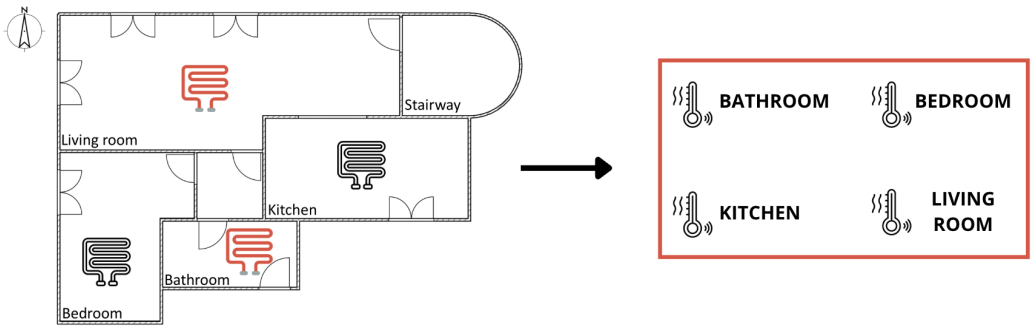


Figure 1.3: Temperature regulation example

1.3.3 Automated HVAC Control

- The HVAC system in each room is programmed to maintain the desired humidity level as set by user.
- The system monitors the current humidity levels and adjust the HVAC operation accordingly, ensuring a comfortable and healthy indoor environment.

This is particularly beneficial in regions with fluctuating humidity levels or during seasons when indoor air quality can be compromised.

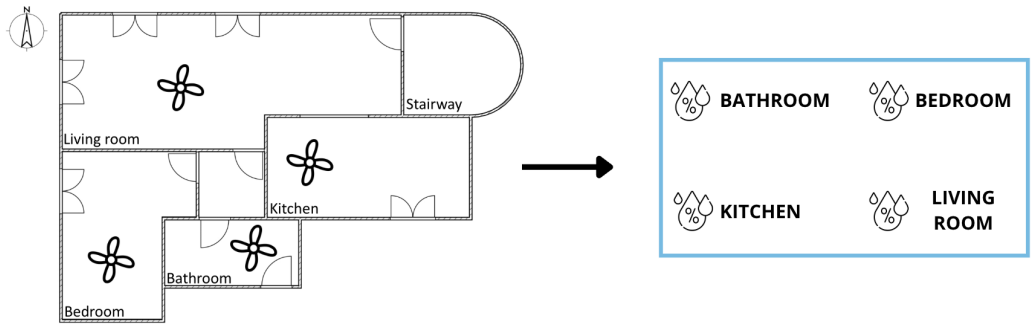


Figure 1.4: Humidity regulation example

1.3.4 Alarm and Intrusion Detection System

- Entrance door sensors are deployed to detect unauthorized access to the premises.
- During specified times when the homes are expected to be unoccupied, the system promptly notifies the homeowner upon detecting any unauthorized entry.

This proactive security measure serves to safeguard the homes against potential intruders and provides peace of mind for the occupants

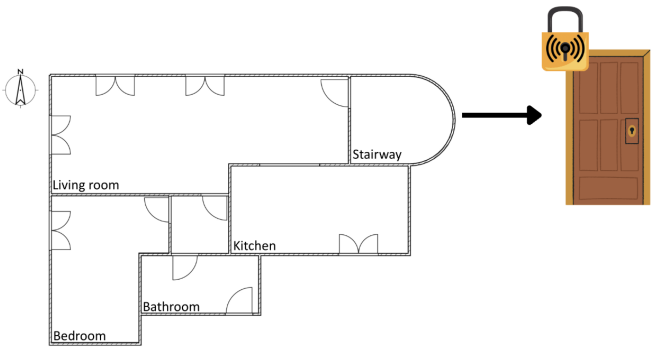


Figure 1.5: Door security example

1.3.5 Energy Consumption Monitoring

- The system offers detailed monitoring of energy consumption for each room and connected device.
- Random realistic consumption data are generated based on accurate metrics for each object involved, allowing users to track and optimize their energy usage.
- The system provides a real-time updates as well as daily and monthly consumption reports, empowering users to make informed decisions about their energy habits and identify areas for potential saving.

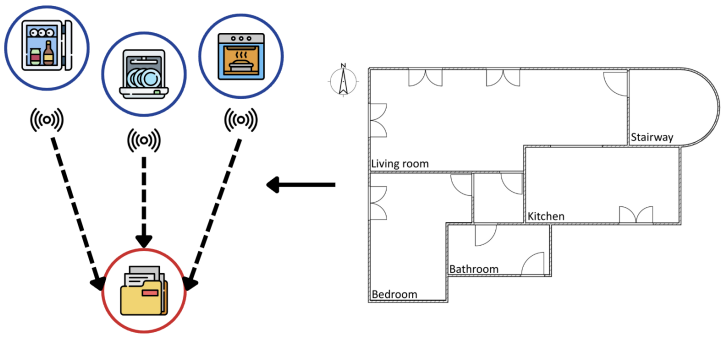


Figure 1.6: Energy consumption example

1.4 Security Policy

Ensuring the security and privacy of data is paramount. To this end, here are the key security features:

- **Data Encryption**: All data pertaining to energy consumption is encrypted using robust encryption protocols before being transmitted over the network. This encryption helps safeguard the confidentiality and integrity of the data, protecting it from unauthorized access or tampering.
- **Role-Based Access Control (RBAC)**: Different roles are assigned to individuals responsible for managing and utilizing resources within the smart home environment. Access permissions are granted based on these roles, ensuring that each user has appropriate access to resources and data.

1.4.1 Access Permissions

- **Administrator Access**: The administrator is granted permissions to view consumption data from both residences for commercial purposes only. This access allows them to analyze energy usage patterns and make informed decisions to optimize resource utilization.
- **User Access**: Users are provided with credentials to access their respective private areas based on their residence. This ensures that information is segregated based on user residences, maintaining a high level of security and privacy. Users can only access data and resources associated with their specific residence, preventing unauthorized access to sensitive information.

1.5 Development environment

The implementation of the smart home automation system involves the integration of several key technologies and protocol.

1.5.1 Key Technologies

- **Node-RED**;
- **Java**;
- **MongoDB**.

Node-RED

Node-RED is a core component responsible for develop the flows that manage various IoT devices and system functionalities. It handles tasks such as generating energy consumption data, managing user logins, controlling temperature and humidity, and providing a user-friendly dashboard for monitoring consumption, which are managed using the MQTT protocol. Node-RED's visual flow-based development environment makes it ideal for defining and implementing the logic needed to manage the complex interactions between devices and the overall system.

Java

Java is used to simulate the physical hardware within the system, including sensors, lights, thermostats, HVAC systems, and other smart devices. These simulated devices communicate with Node-RED via TCP.

MongoDB

MongoDB serves as the database for storing all relevant data generated by the system. Various collection in MongoDB are used to store information such as user credentials, temperature and humidity settings, energy consumption data, and system logs. This NoSQL database is well-suited for handling the large volumes of unstructured data typical in IoT applications, providing flexibility and scalability as the system grows.

1.5.2 Protocols Involved

- **MQTT-(Message Queuing Telemetry Transport)**;
- **TCP-(Transmission Control Protocol)**;

MQTT Protocol

In the smart home automation project, the MQTT protocols plays a crucial role in enabling efficient communication between various devices and systems within the smart home. MQTT is a lightweight protocol based on the publish/subscribe model, where devices (publishers) send message to a central broker, and other devices or applications (subscribers) receive these messages when they subscribe to relevant topics.

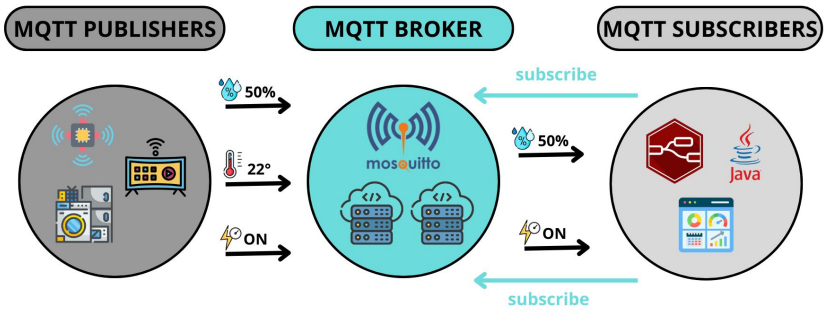


Figure 1.7: MQTT Interaction Schema

The diagram in Fig.1.7 illustrates how devices within the smart home environment communicate with each other through the MQTT protocol, particularly utilizing the *Mosquitto* broker. A simple workflow can be described below:

- **MQTT Publishers**: inside the smart home project different devices acting as MQTT publishers: sensors collect real-time data, such as temperature, humidity, status, etc. These devices publish this information to specific topics hosted on the *Mosquitto* broker.
- **MQTT Broker**: it serves as the intermediary that handles the information flow between publishers and subscribers. It receives the messages form the publishers, store these temporarily in relevant topics and distributes the messages to subscribers that have registered their interested in specific topics.
- **MQTT Subscribers**: which include control system and user interfaces, represented by Node-RED dashboard and Java applications. These system subscribe to topics of interest to receive updates in real-time. For instance, a dashboard application subscribes to a topic that provides real-time temperature updates to the user.

The MQTT topic structure is a key element that organizes the communication between devices in an hierarchical and easily manageable way. Topics are essentially address path that define where data is published and where subscribers can access specific information. Each topic is composed of multi levels, separated by slashes (/), representing different aspects of the smart home environment, such as specific rooms, devices or types of data. In this project, the topics follow a logical hierarchy based on location, devices and data type of the smart home system. Below, the schema in Fig.1.8 shows an example of the general topic hierarchy that applies to all rooms:

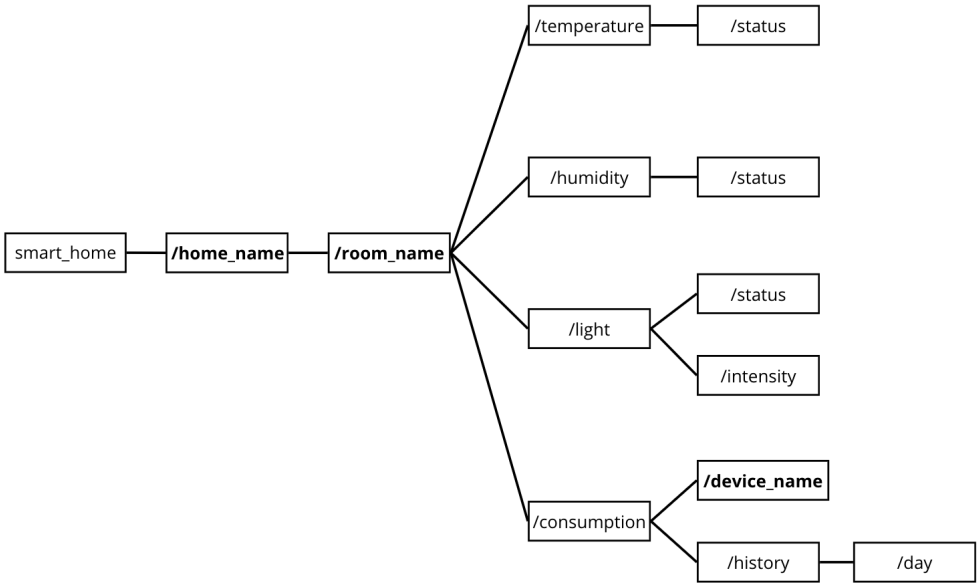


Figure 1.8: MQTT Topic Hierarchy Structure

Let's take for example the Kitchen in smart home A, the topic structure is the following one:

```
'smart_home/A/kitchen/
 /temperature/status
 /humidity/status
 /light/intensity
 /light/status
 /consumption/light
 /consumption/stove
 /consumption/oven
 /consumption/water/history/day
 /consumption/history/day
 /consumption/history/month'
```

This structure is replicated for other rooms, with each rooms having its own set of topics for devices and sensors. The *'consumption'* topic varies based on the devices in each room.

In summary, the MQTT protocol, through the *Mosquitto* broker, ensures seamless and efficient communication between all devices within the smart home, enabling real-time data exchange and enhancing the overall automation and user experience.

1.5.3 UML Diagram

Use Case Diagram

To better understand the functionalities describes at high level in 1.3, the following diagram is presented.

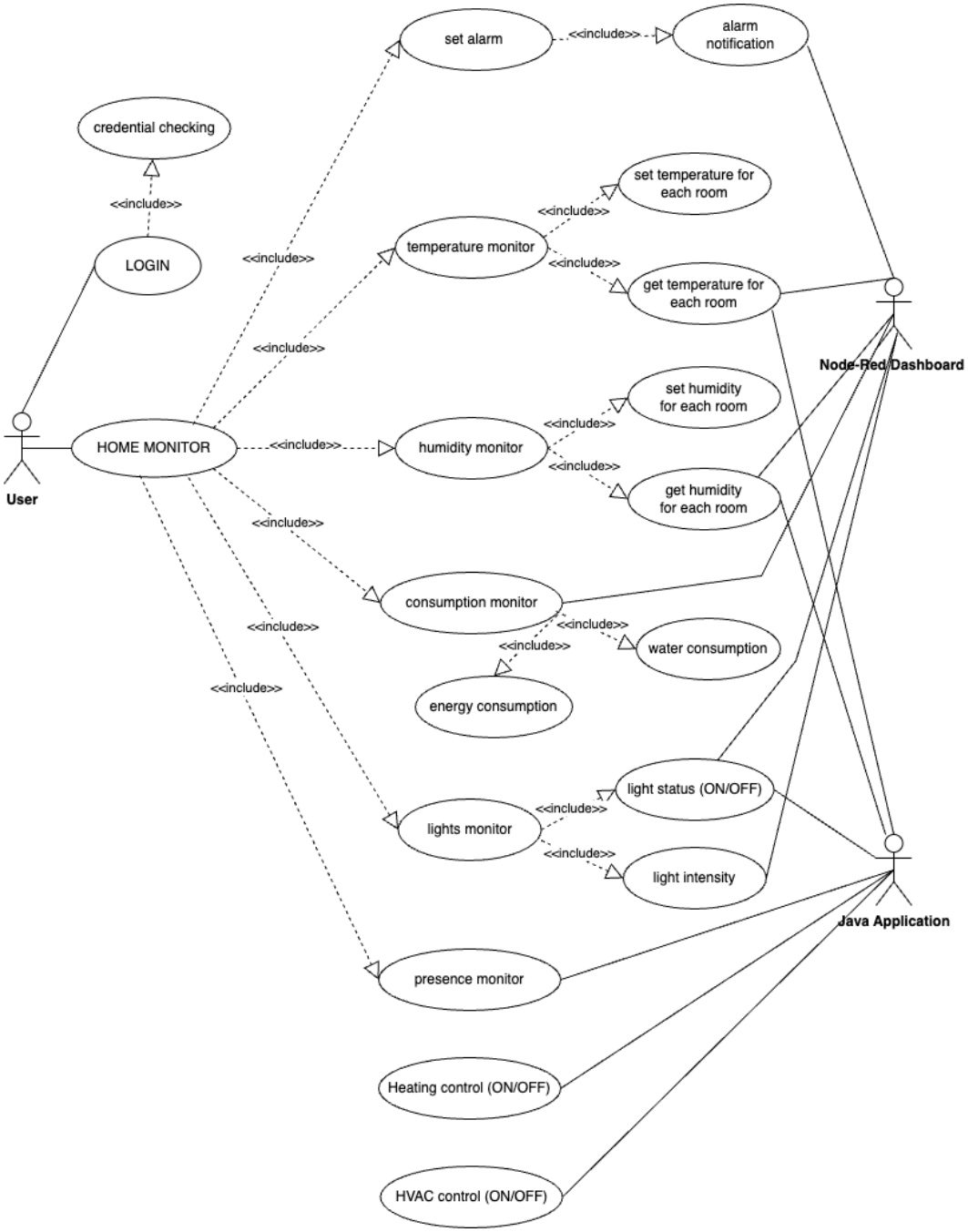


Figure 1.9: Use Case Diagram

This is useful for visualizing the structure and interactions within the system, helping to clarify the relationship between different components and their roles in achieving the overall behavior of the project.

Chapter 2

Smart Home

2.1 Automation

The smart home system provides an intuitive application that allows users to monitor and control various connected devices throughout the home. Thanks to this interface, users can easily view the current status of the smart home, including information about sensors, lighting, temperature, and humidity in each room. The dashboard offers real-time updates on the states of the devices, ensuring that the user can efficiently manages and automate the home environment.

2.1.1 Login

Once the application is launched from the Java platform, the system prompts the user for identification by entering the appropriate credentials, as the *'Login'* form in Fig.2.1 indicates.

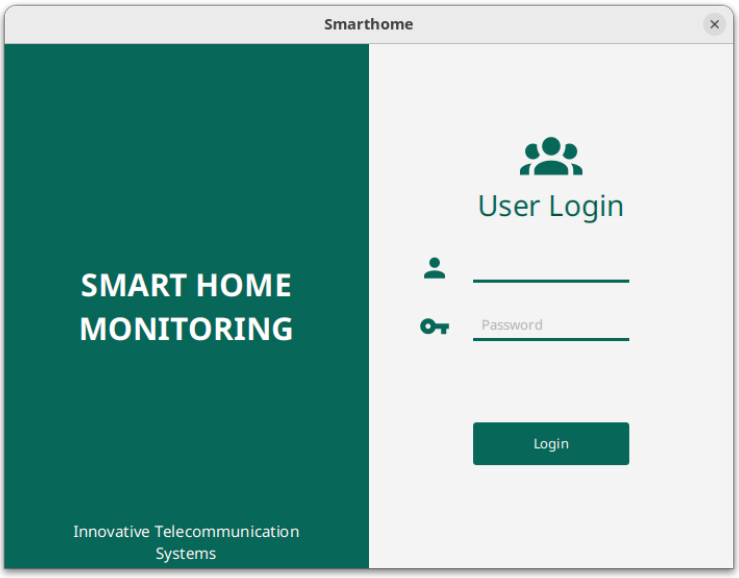


Figure 2.1: Java Login Page

2.1.2 Home A/B Status Dashboard

Depending on the authentication, the main screen is displayed (Home A in Fig.2.2 and Home B in Fig.2.3), showing the floor plan of the smart home with various objects for each room. Users can observe the automated behavior of these smart home devices in real time:

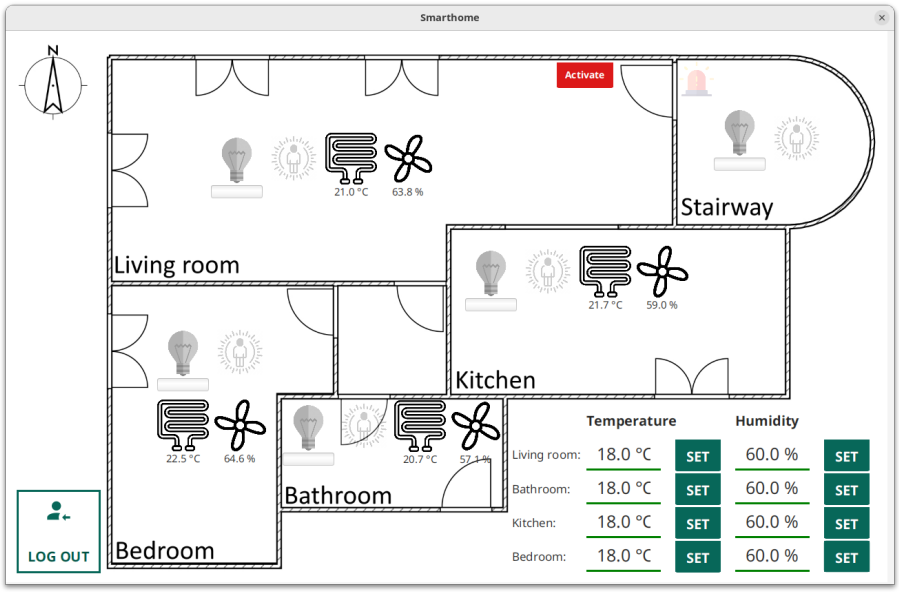


Figure 2.2: Home A Java Dashboard

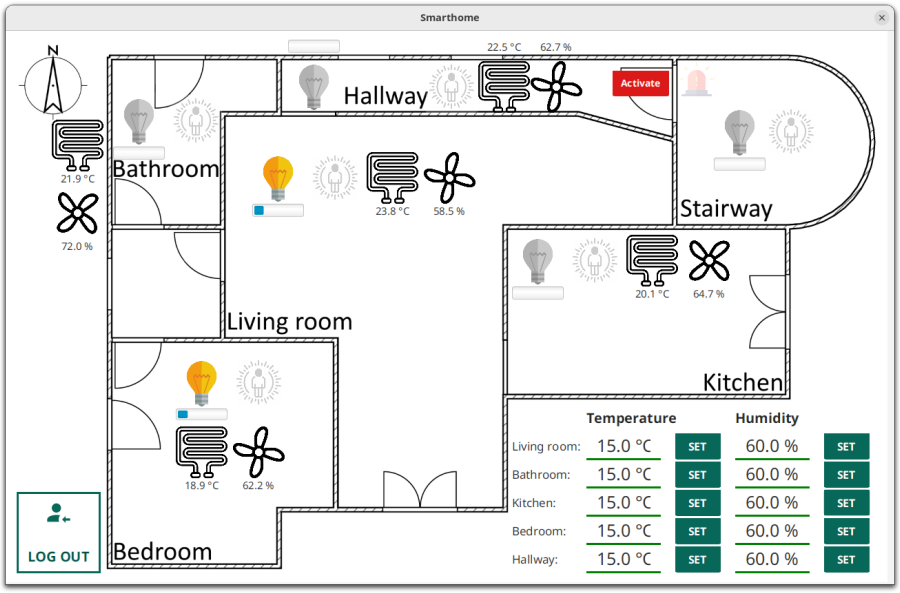


Figure 2.3: Home B Java Dashboard

Presence Sensors

The sensor activates when the system detects someone in the room, and the corresponding icon becomes more visible to indicate its activation.

Description	Icon
Presence sensor off	
Presence sensor on	

Table 2.1: Presence Sensor Possible Status

Lights

When a presence is detected, the lights automatically turn on, and the intensity is adjusted based on the natural daylight available.

Description	Icon
Light off	
Light on	

Table 2.2: Light Possible Status

Temperature and Humidity

Users can set their preferred temperature and humidity levels, by adjusting the values, as shown in Fig.2.4.

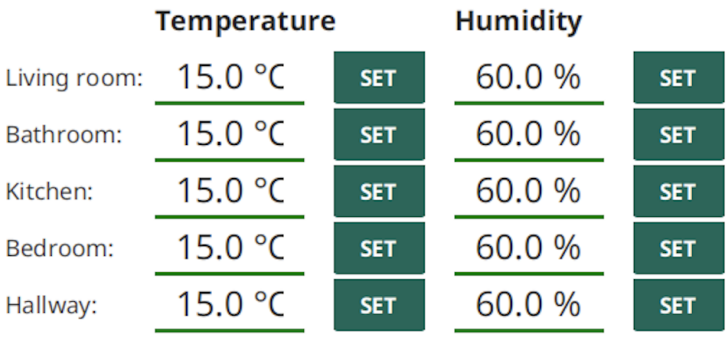


Figure 2.4: Temperature-Humidity Settings

The current temperature and humidity for each room are displayed under the relevant relevant icons. If the detected value fall below the desired settings, the heating or HVAC system will automatically turn on.

When heating is active, the icon updates to reflect this (Tab.2.3), while the HVAC system's icon animates by spinning.

Description	Icon
Heater off	
Heater on	

Table 2.3: Heater Possible Status

Alarm System

The user can activate the alarm system via a dedicated button (Tab.2.4). Once enabled, the system will send email notification to the user in the event of any potential intrusions. In addition, it's possible to display the alarm symbol (Tab 2.5) flashing intermittently, prompting the user to adopt defensive techniques.

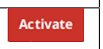
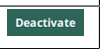
Description	Icon
Alarm off	
Alarm on	

Table 2.4: Alarm Possible Status

Description	Icon
No Intrusion	
Intrusion	

Table 2.5: Alarm Intrusion Status

2.2 Monitoring

The monitoring section of the smart home automation project provides users with comprehensive insights into their home's status and performance. This part outlines they key features and functionalities of the monitoring system.

By visiting http://centore.synology.me:1880/ui, users can access the monitoring dashboard. This link directs them to the Node-RED dashboard, which contains valuable information about their smart home's current state. To ensure security and privacy, the system implements strict authentication policies. Every user must authenticate themselves before gaining access to their personalized area, as the form in Fig.2.5 shows.

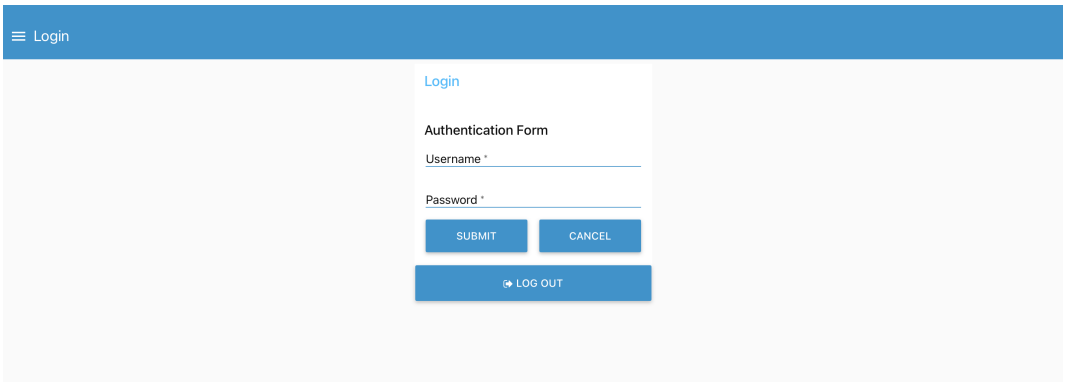


Figure 2.5: Login Dashboard

Upon successful authentication, users are presented with different dashboards based on their assigned roles. This role-based access control ensures that users only see information relevant to their permission and responsibilities.

2.2.1 Home stats

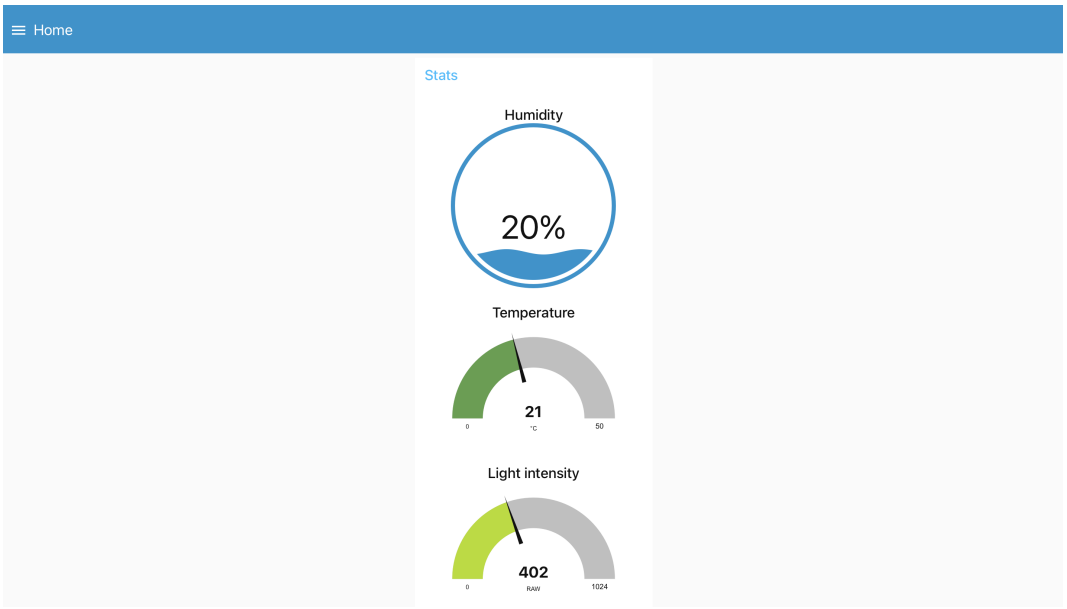


Figure 2.6: Home Stats Dashboard

Only administrator has exclusive access to the *"Home Stats"* dashboard, showed in Fig.2.6. It displays charts containing information extracted from a real house, including temperature, humidity, and light intensity. These data serve as the foundation for simulating the entire monitoring process and automation behavior of smart homes.

2.2.2 Home A/B Dashboard

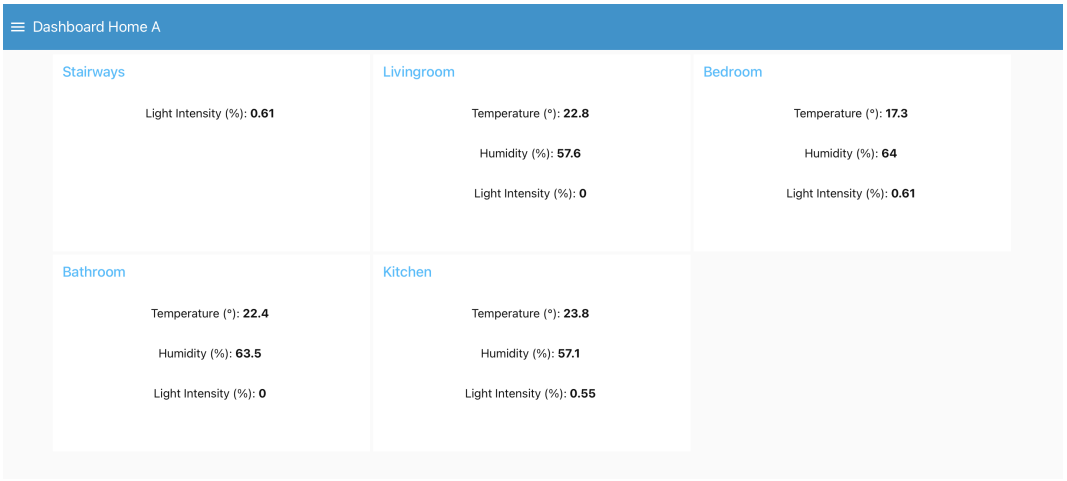


Figure 2.7: Home A Dashboard

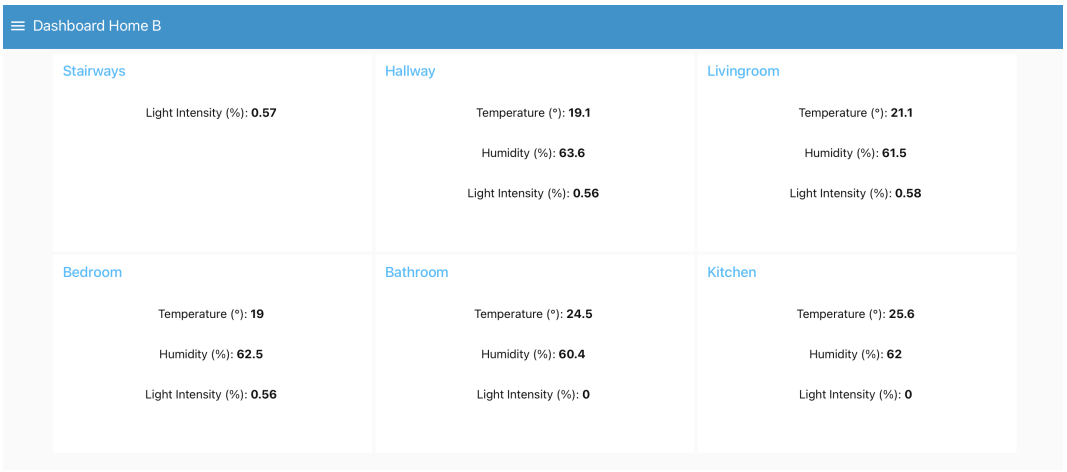


Figure 2.8: Home B Dashboard

These dashboards provides general information about the status of the user's A (Fig.2.7) and B (Fig.2.8) smart home. It displays real-time data on temperature, humidity, and light intensity for different rooms within the house. Users can quickly assess the environmental conditions throughout their home.

2.2.3 Home A/B Consumption

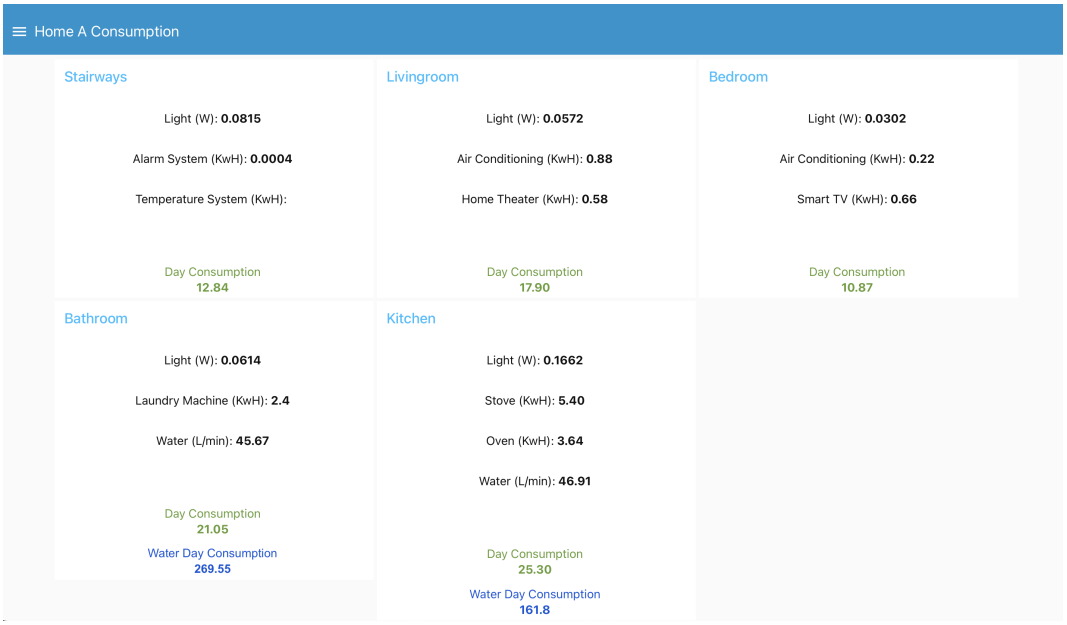


Figure 2.9: Home A Consumption Dashboard

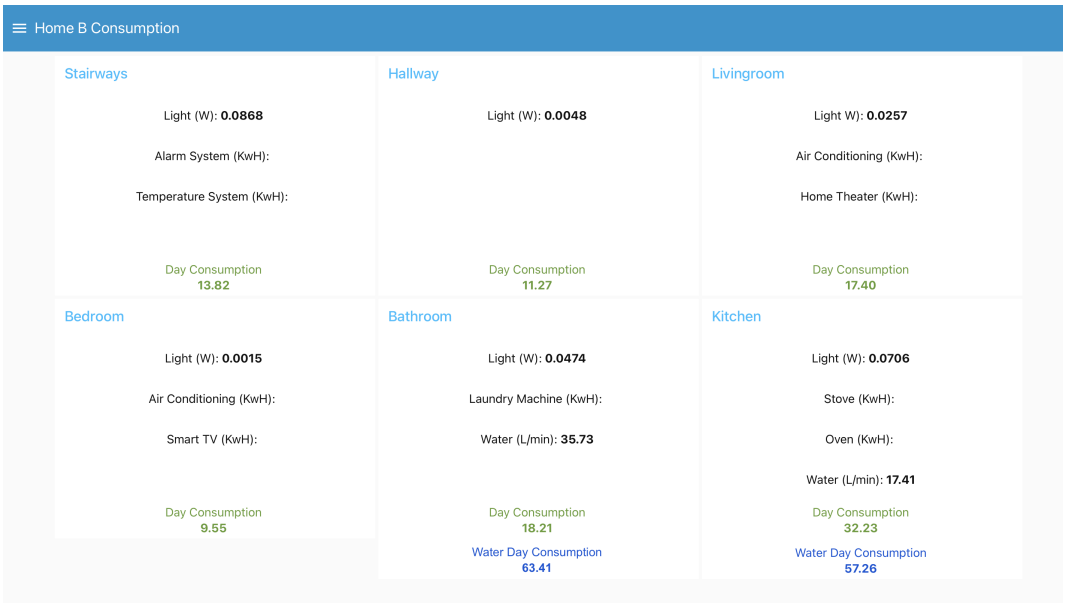


Figure 2.10: Home B Consumption Dashboard

The consumption dashboards in Fig.2.9 and Fig.2.10 offer a summary of both instantaneous and daily energy/water usage, updated in real-time, based on the type of smart home. They provide detailed information on the consumption of installed devices used in each room. This feature allows users to monitor their energy usage patterns and identify potential areas for optimization.

2.2.4 Home A/B History

Figure 2.11: Home A Consumption History Dashboard

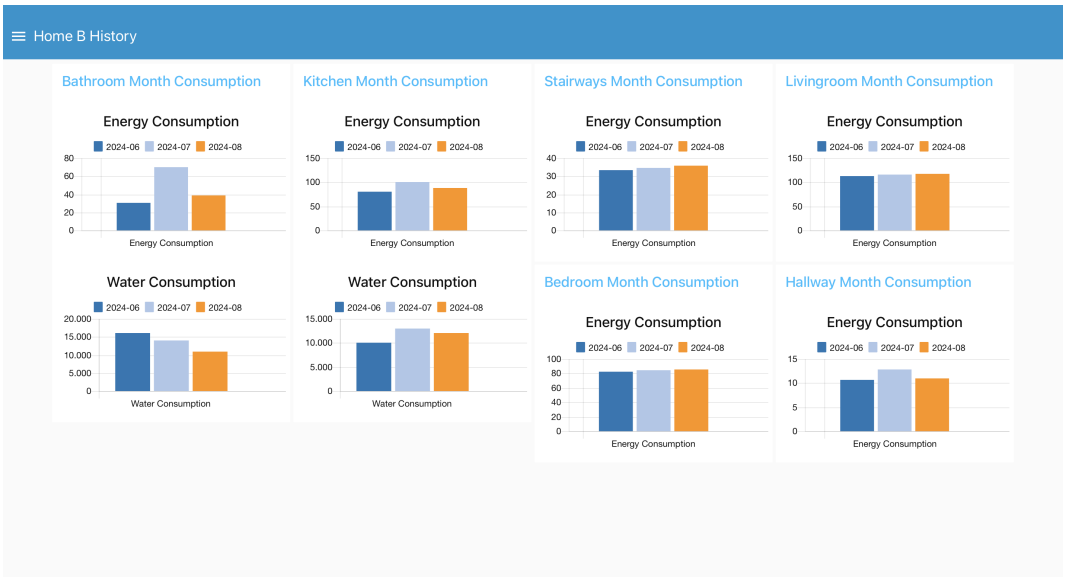


Figure 2.12: Home B Consumption History Dashboard

The history dashboards (Home A in Fig.2.11 and Home B in Fig.2.12) present a graphical summary of consumption trends for each room on a monthly basis. This comprehensive analysis of energy and water usage throughout months enables users to identify long-term patterns and male informed decisions about resource management. By providing these different monitoring functionalities. the smart home automation empowers users with detailed insights into their home's performance, energy consumption, and environmental conditions. This information allows for better decision-making, increased energy efficiency, and a more comfortable living environment.

Chapter 3

MongoDB

For this project, MongoDB was used as the database to store data and perform various operations. The choice as made for several advantages:

- **Flexibility**: document-based structure allows for easy adaptation to changing data requirements.
- **Performance**: it offers high-speed data retrieval and writing operations.
- **JSON-like documents**: the format used making it easier to work with.
- **No SQL injection**: As a NoSQL database, MongoDB is not vulnerable to SQL injection attacks. The solution can be used for future web integration for this IoT application.

The main database is called **"SmartHome"** and contains the following collections.

- **Users**: stores information about the users of the smart home system, used for manages authentication and role.
- **MyHome**: stores data gathered by ESP8266 related to information of a physical smart home. Define the layout and serves as a baseline for further implementation the simulation.
- **HVAC**: store data related to temperature and humidity parameters. It serves as a key-controller to monitors the desired temperature and humidity level.
- **Alarm**: this collection saves records to alarms and notifications. It keeps track of triggered alarms and the responses taken to ensure home security.
- **Setting**: contains configurations settings for the smart home devices, in order to collect the preferences and automation setting that control how system should behave.
- **ConsA-ConsB**: picks real-time consumption data respectively for smart home A/B, in order to track and monitors consumption of resources in specific area within the environment.
- **MonthConsA-MonthConsB**: stores monthly consumption statistics for smart home A/B, providing historical data for energy and water consumption, aiding in analysis and optimization.

Below, the structure and the format of the documents contained in each collection will be illustrated. It should be notice that only the name and type of parameters are shown, without including actual values.

3.1 Collections

3.1.1 Users

```
{
    "_id": {},
    "username": String,
    "password": String
}
```

The credential used by the owners to log in into the automation and monitoring system.

3.1.2 MyHome

```
{
    "_id": {},
    "temperature": {
        "value": Int,
        "unit": String
    },
    "humidity": {
        "value": Int,
        "unit": String
    },
    "light" : {
        "value": Int,
        "unit": String
    }
}
```

Three objects containing information (value and unit) regarding temperature, humidity and light intensity about a real smart home.

3.1.3 HVAC

```
{
    "_id": {},
    "idx": Int,
    "home": String,
    "room": String,
    "type": String,
    "setPoint": Int,
    "current": Int
}
```

- **Idx**: a numerical index used to distinguish the room sensors.
- **Home**: identifies the home.
- **Room**: the name of the specific room within the home where the control is applied.
- **Type**: describe whether the control is for temperature or for humidity management.
- **SetPoint**: the desired value for temperature or humidity that is set by the user.
- **Current**: the current measured value of temperature or humidity.

3.1.4 Alarm

```
{
    "_id": {},
    "idx": Int,
    "home": String,
    "status": String
}
```

- **Idx**: An integer value that represents the ID of the alarm.
- **Home**: the name of the home where this alarm system is installed.
- **Status**: The current state of the alarm system. This field indicate whether the alarm is "armed" or "disable". It is updated based on the user action.

3.1.5 Setting

Temperature System

```
{
    "_id": {},
    "temperature_system": {
        "power": Int
    }
}
```

- **Power**: the baseline power consumption of the temperature system in kW.

Alarm System

```
{
    "_id": {},
    "alarm_system": {
        "power": Int
    }
}
```

- **Power**: the baseline power consumption of the alarm system in kW.

Laundry Machine

```
{
    "_id": {},
    "laundry_machine": {
        "power": Int,
        "cycles": [String],
        "maxTime": Int,
        "minTime": Int,
        "temperatures": [Int]
    }
}
```

- **Power**: the baseline power consumption of the laundry machine in kW.
- **Cycles**: array of the different types of wash cycles the machine can perform.
- **MinTime - MaxTime**: the range of possible duration for a washing cycle, indicating the shortest and longest time a wash might take.
- **Temperatures**: array of the possible water temperatures that can be used during a wash.

Air Conditioning

```
{
    "_id": {},
    "air_conditioning": {
        "power": Double,
        "modes": [String],
        "fanSpeeds": [String],
        "temperatures":
        {
            "minTemp": Int,
            "maxTemp": Int
        }
    }
}
```

- **Power**: the baseline power consumption of the air conditioning in kW.
- **Modes**: the list of possible operating modes of the air conditioning.
- **FanSpeeds**: array composed by the level at which the air conditioner's fan operates.
- **Temperatures**: the minimum and maximum temperatures that can be set on the air conditioning.

Home Theater

```
{
    "_id": {},
    "home_theater":{
        "maxPower": Int,
        "minPower": Double
    }

}
```

- **MinPower**: minimum consumption in kW, when a limited number of devices are turned on.
- **MaxPower**: maximum consumption in kW, fueled by a greater number of devices used at the same time.

Smart TV

```
{
    "_id": {},
    "smart_tv":{
        "power": Double
    }
}
```

- **Power**: the baseline power consumption of the smart tv in kW.

Stove

```
{
    "_id": {},
    "stove": {
        "maxPower": Int,
        "minPower": Double,
        "cooking_areas": [Int],
    }
}
```

- **MinPower**: this represents the minimum power required by the stove for basic operation.
- **MaxPower**: this is the maximum power output of the stove, typically used for high-intensity cooking task.
- **Cooking_areas**: These are the individual cooking zones on the induction stove, which can be used independently or simultaneously.

Oven

```
{
    "_id": {},
    "oven": {
        "power": Int,
        "modes": [String],
        "temperatures": [Int]
    }
}
```

- **Power**: base power consumption, represents the minimum energy usage when the oven is turned on, regardless of other settings.
- **Modes**: the oven can operate in different modes. Each mode affects the energy consumption differently.
- **Temperatures**: array containing the list of temperatures for cooking. They also play a critical role in energy consumption.

Water

```
{
    "_id": {},
    "water": {
        "maxConsumption": Int,
        "maxDuration": Int,
        "minConsumption": Int
    }
}
```

- **MinConsumption**: it represents the lower bound of water usage, simulating low-flow fixtures such as water-efficient faucets or low-flow shower heads.
- **MaxConsumption**: this parameters sets the upper limit of water usage, representing high-flow scenarios like power showers of filling a bathtub.
- **MaxDuration**: it defines the longest continuous period of water usage, preventing unrealistically long periods of constant water flow.

3.1.6 Cons A - ConsB

```
{
    "_id": {},
    ["room": String],
    "item": String,
    "value": Int/Double,
    "addedAt": Date
}
```

- **Room**: the room where the device is located. This field is optional (indicated by [ ]), meaning not all documents include it. It's useful when the same device is applied to multiple room.
- **Item**: identifies the specific device for which the consumption is being recorded.
- **Value**: The amount of consumption of the device, which could represent energy or another unit of resource.
- **AddedAt**: it marks the specific time when the consumption was recorded.

3.1.7 MonthCons A - MonthConsB

```
{
    "_id": {},
    "Month": String,
    "Room": String,
    "Consumption" : {
        "EnergyConsumption" : Double
        ["WaterConsumption" : Double]
    }
}
```

- **Month**: indicates the month during which the consumption data was recorded.
- **Room**: denotes the specific room in the home where the consumption data was measured.
- **Consumption**: an object containing:
  - **EnergyConsumption**: it represents the amount of energy consumed in that room during the specified month.
  - **WaterConsumption**: An optional field (denoted by [ ]) that, if present, contains a number representing the amount of water consumed in the room. This filed may not available for every room, as not all rooms may have water consumption data recorded.

Chapter 4

Java

In this project, a java application was created as a supervisor for the operation of the smarthome. it was created using java FX version 22.0.2 in order to make the realisation of the graphical interfaces easier, faster and more customisable thanks to fxml files.

4.1 Frontend

All graphics have been created using graphical XML files, CSS style files and specially selected icons according to the type of sensor.

4.1.1 FXML

FXML files are XML documents used to define the user interface (UI) of JavaFX applications. These files allow application logic to be separated from presentation, making code development and maintenance easier. The structure of an FXML file is similar to a standard XML file.

- FXML enables the implementation of the Model-View-Controller (MVC) pattern, where the user interface is defined in the FXML file and the application logic is handled by Java controller classes.
- Tools such as Scene Builder have been used to visually design the user interface, making the process faster and less error-prone than writing code by hand.
- FXML files can be easily reused and modified without having to change the underlying logic.

4.1.2 CSS

In this project, the use of CSS helped in the creation of the interfaces, as it was sufficient to create templates for the most frequently used elements. This makes it possible to apply and maintain a default style for the element (e.g. a button) without having to declare it for each individual element. Here are some examples of how CSS has been used:

```
.my-button {
    -fx-background-color: #076759;
    -fx-border-color: #076759;
    -fx-border-width: 3;
    -fx-text-fill: white;
}
.my-textfield {
    -fx-background-color: transparent;
    -fx-border-color: #076759;
    -fx-border-width: 0px 0px 3px 0px;
}
.field-error {
    -fx-border-color: red !important;
}
.field-to-be-approved {
    -fx-border-color: purple !important;
}
.field-info {
    -fx-border-color: blue !important;
}
.field-valid {
    -fx-border-color: green !important;
}
```

An important role of the CSS was to make the operation of the application more intuitive with regard to the exchange of data between the client (dashboard) and the server (node-red). Four main colours are used in the application:

- **red**: reporting errors
- **purple**: waiting for server response
- **blue**: generic information
- **green**: reporting valid field

4.1.3 Icons

Two types of icons are used in this project, small icons used to navigate between pages or to show details of text fields or buttons, and large icons used to represent devices in the smarthome. The application icons are created using a JavaFX plugin called iknoli, version 12.3.1, while the device icons have a legend below:

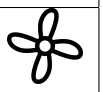
Description	Icon
Main application icon	
Light off	
Light on	
Heater off	
Heater on	
MEV(Mechanical Extract Ventilation)	
Presence sensor	
Alarm	

Table 4.1: Icons used in the application

4.2 Backend

The backend structure is also implemented in Java. It is based on Gradle, a compilation automation tool that provides a domain-specific language (DSL) based on Groovy. It allows the use of Maven, Ivy or other custom repositories for dependency management, including javafx and mqtt.
The following images illustrate the tree structure of the Java *src* directory divided in java and resources.

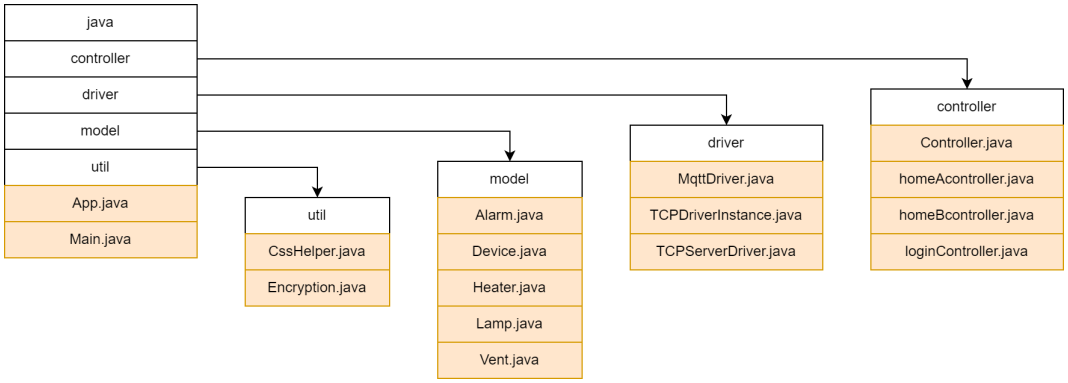


Figure 4.1: Java files directory

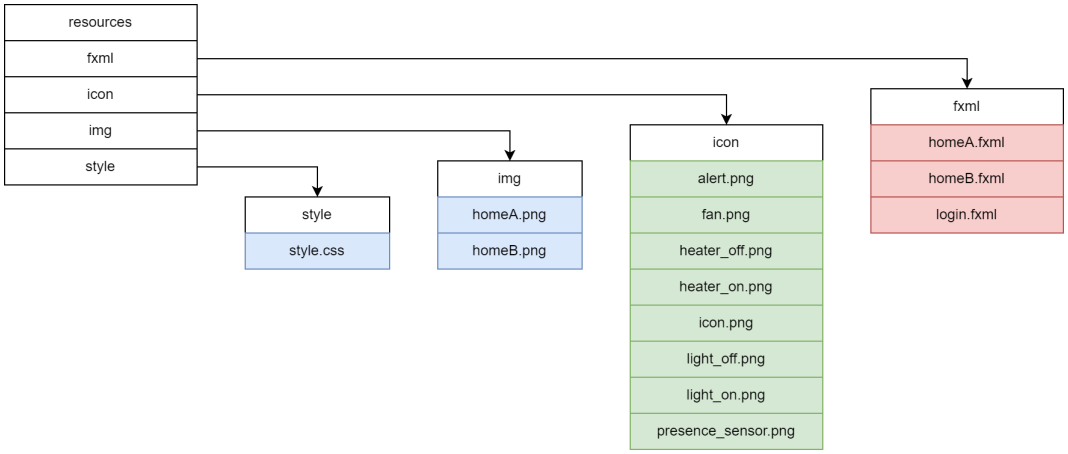


Figure 4.2: Resources files directory

4.2.1 Device

In order to ensure the correct handling of sensors and actuators, special Java objects were created. The *Device* class can be regarded as the starting point for each object, as it serves only to provide a parent class, containing the generic attributes that the sensors have in common, such as:

- **id**: the identifier for the device, which allows it to be readily identified during the course of operations within the smarthome environment.
- **name**: represents the name of the device.
- **home**: indicates the smarthome to which it belongs. The value refers to an enumerator within the Device class and can take values such as: none, A and B.
- **room**: indicates the room to which it belongs. Again, the value refers to an enumerator within the Device class
- **status**: indicates the current activation status of the device.
- **linked_image**: contains the image associated with the device for use in graphics operations. For example, in the case of a light bulb to change the image from on to off.

Once the generic class *Device* had been realised, the device-specific classes *Alarm*, *Heater*, *Vent* and *Lamp* were created. Each class contains the customisation associated with the device type.

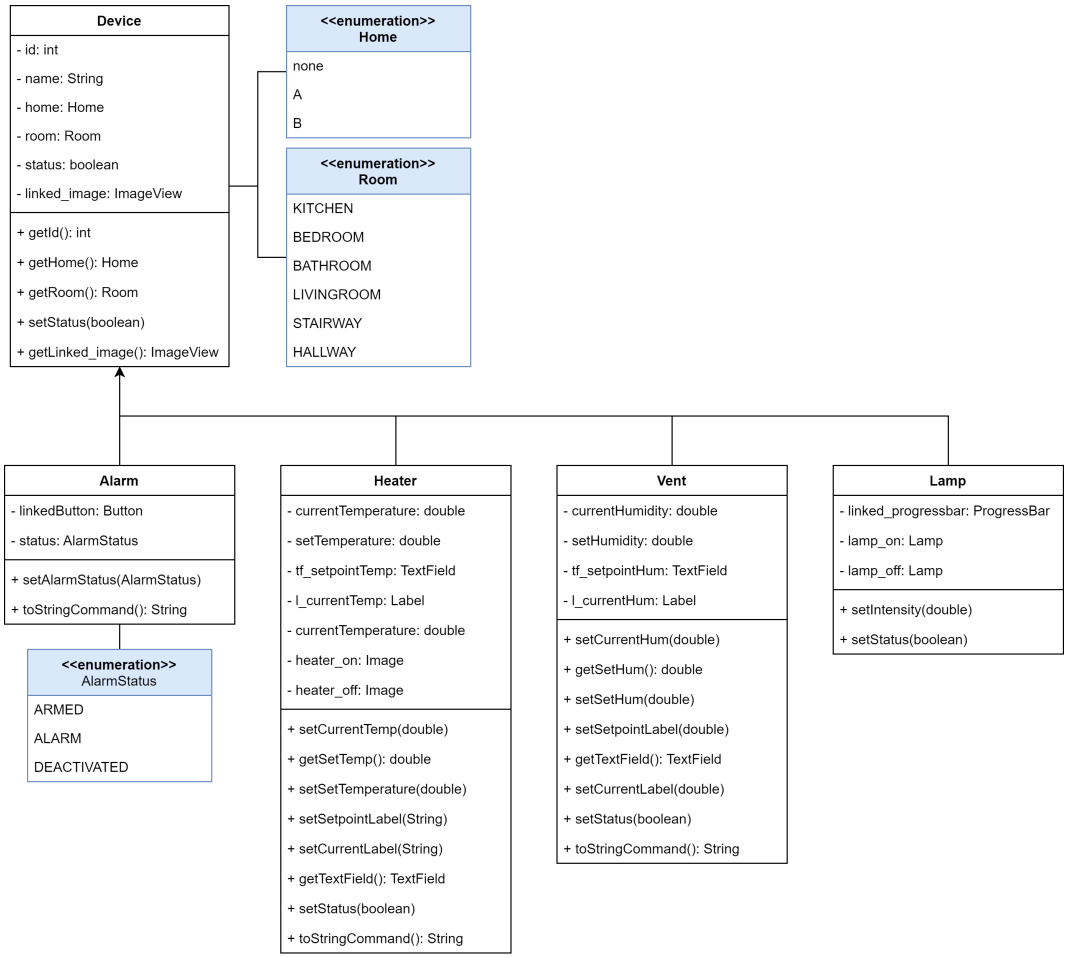


Figure 4.3: Device class diagram

Starting from the *Alarm* class, the linkedButton and status attributes were added. the first one is used to link the dashboard button and manage its behaviour such as colour and text in relation to the alarm status, the second can be seen as a soft override of the status field of the superclass because there is a need to have three different states: ARMED, ALARM and DEACTIVATED. The *Heater* and *Vent* classes are quite similar with regard to the behaviour of the devices. there are fields for the current room temperature and the temperature set by the user and methods for setting the value.
The *Lamp* class has a linked progress bar attribute. This shows the current brightness level on the Java dashboard with a progress bar.

4.2.2 Controller

Whenever an fxml file is created to make it accessible from java code, it is necessary to link it to a special class called 'controller'. The purpose of the controller class is to contain all references to elements in the graphical interface and actions performed on components. Four controllers were created in the project. Similar to the Device class, a superclass and three subclasses were created, one for the login, one for smarthome A and one for B. Although the main Controller class is empty, it is very useful for the creation and linking of the individual controllers for the first start-up, this allows casting and avoiding instance type checking.
The first controller encountered is the login controller. This controller consists of 2 methods, the first server for communicating the username and password to node-red via a tcp message while the second handles the replies received.

```
@FXML
public void login() {
    tcp.setController(this);
    String username = tf_username.getText().strip();
    String password = tf_password.getText().strip();
    if(!username.isEmpty() && !password.isEmpty()) {
        tcp.send(String.join(" ", "LOGIN", username, password));
    }
}

public void loginValidation(boolean status, Device.Home home) {
    Controller c;
    if(status & home.equals(Device.Home.A)) {
        System.out.println("Login to smarthome A");
        l_error.setVisible(false);
        c = App.setRoot("homeA");
    }
    else if(status & home.equals(Device.Home.B)) {
        System.out.println("Login to smarthome B");
        l_error.setVisible(false);
        c = App.setRoot("homeB");
    }
    else {
        l_error.setVisible(true);
    }
}
```

The code for the initialize method of the homeAcontroller class is shown below. This method is part of the library and is called once the GUI has been successfully started.

This method collects all objects, creates the objects 'Lamp', 'Heater', 'Vent' and 'Alarm' and places them inside an array of devices for easy searching. Once the array is completed, a tcp command with the content 'REQ A ALL' is sent to request all data relating to smarthome A.

```
@FXML
private void initialize() {
    /* DEVICES */
    Lamp l_livingroom = new Lamp(200, "", Device.Home.A,
    Device.Room.LIVINGROOM, iv_light_livingroom, pb_light_livingroom);
    Lamp l_bathroom = new Lamp(201, "", Device.Home.A,
    Device.Room.BATHROOM, iv_light_bathroom, pb_light_bathroom);
    Lamp l_kitchen = new Lamp(202, "", Device.Home.A,
    Device.Room.KITCHEN, iv_light_kitchen, pb_light_kitchen);
    Lamp l_stairway = new Lamp(203, "", Device.Home.A,
    Device.Room.STAIRWAY, iv_light_stairway, pb_light_stairway);
    Lamp l_bedroom = new Lamp(204, "", Device.Home.A,
    Device.Room.BEDROOM, iv_light_bedroom, pb_light_bedroom);

    h_livingroom = new Heater(120, "", Device.Home.A,
    Device.Room.LIVINGROOM, iv_heater_livingroom, tf_tempLivingRoom,
    l_tempLivingRoom);
    h_bathroom = new Heater(121, "", Device.Home.A,
    Device.Room.BATHROOM, iv_heater_bathroom, tf_tempBathroom,
    l_tempBathroom);
    h_kitchen = new Heater(122, "", Device.Home.A,
    Device.Room.KITCHEN, iv_heater_kitchen, tf_tempKitchen, l_tempKitchen);
    h_bedroom = new Heater(123, "", Device.Home.A,
    Device.Room.BEDROOM, iv_heater_bedroom, tf_tempBedroom, l_tempBedroom);

    v_livingroom = new Vent(130, "", Device.Home.A,
    Device.Room.LIVINGROOM, iv_fan_livingroom, tf_humLivingRoom,
    l_humLivingRoom);
    v_bathroom = new Vent(131, "", Device.Home.A,
    Device.Room.BATHROOM, iv_fan_bathroom, tf_humBathroom, l_humBathroom);
    v_kitchen = new Vent(132, "", Device.Home.A,
    Device.Room.KITCHEN, iv_fan_kitchen, tf_humKitchen, l_humKitchen);
    v_bedroom = new Vent(133, "", Device.Home.A,
    Device.Room.BEDROOM, iv_fan_bedroom, tf_humBedroom, l_humBedroom);

    alarm = new Alarm(70, "", Device.Home.A,
    Device.Room.LIVINGROOM, iv_alarm_a, b_alarm);
```

```
    devices = new Device[]{l_livingroom, l_bathroom, l_kitchen,
    l_stairway, l_bedroom, h_livingroom, h_bathroom, h_kitchen, h_bedroom,
    v_livingroom, v_bathroom, v_kitchen, v_bedroom};

    tcp.send("REQ A ALL");
}
```

Once all required data has been received, the text strings are analysed and divided according to device type using the following code:

```
public void updateDevice(String[] splitted) {
    int idx = Integer.parseInt(splitted[1]);
    if(idx == 70) {
        switch(Alarm.alarmStatus.valueOf(splitted[4])) {
            case ARMED -> alarm.setAlarmStatus(Alarm.alarmStatus.ARMED);
            case DEACTIVATED -> alarm.setAlarmStatus
            (Alarm.alarmStatus.DEACTIVATED);
        }
    } else {
        boolean status = Boolean.parseBoolean(splitted[4]);
        double currentValue = Double.parseDouble(splitted[5]);
        double setPoint = Double.parseDouble(splitted[6]);
        for (Device d : devices) {
            if (d.getId() == idx) {
                if (d instanceof Heater) {
                    ((Heater) d).setCurrentTemp(currentValue);
                    ((Heater) d).setSetTemperature(setPoint);
                    css.toValid(((Heater) d).getTextField());
                } else if (d instanceof Vent) {
                    ((Vent) d).setCurrentHum(currentValue);
                    ((Vent) d).setSetHum(setPoint);
                    css.toValid(((Vent) d).getTextField());
                }
            d.setStatus(status);
            break;
            }
        }
    }
}
```

Similar methods are used to handle messages received from MQTT.
Another important piece of code concerns the setting of temperatures and humidity values by the user. Initially, a switch is used to intercept the caller of the method and, depending on the device type, set the temperature or humidity value. This is the only way the user can interact with this type of device.

```
@FXML
public void setSetpoint(ActionEvent ae) {
    double value;
    TextField textField = null;
    Device device = null;
    switch(ae.getSource().toString().split("id=")[1].split(", ")[0]) {
        case "b_temp_livingroom":
            textField = tf_tempLivingRoom;
            device = h_livingroom;
            break;
        case "b_temp_bathroom":
            textField = tf_tempBathroom;
            device = h_bathroom;
            break;
        case "b_temp_kitchen":
            textField = tf_tempKitchen;
            device = h_kitchen;
            break;
        case "b_temp_bedroom":
            textField = tf_tempBedroom;
            device = h_bedroom;
            break;
        case "b_hum_livingroom":
            textField = tf_humLivingRoom;
            device = v_livingroom;
            break;
        case "b_hum_bathroom":
            textField = tf_humBathroom;
            device = v_bathroom;
            break;
        case "b_hum_kitchen":
            textField = tf_humKitchen;
            device = v_kitchen;
            break;
        case "b_hum_bedroom":
            textField = tf_humBedroom;
            device = v_bedroom;
            break;
```

```
            default:
                System.err.println("ERROR");
        }

        assert textField != null;
        try {
            value = Double.parseDouble(textField.getText().trim()
            .replace("°", "").replace("C", "").replace("%", ""));
            if (device instanceof Heater) {
                if(value < 15 || value > 50) {
                    css.toError(((Heater) device).getTextField(),
                    "Min: 15 - Max: 50");
                } else {
                    ((Heater) device).setSetTemperature(value);
                    css.toBeApproved(((Heater) device).getTextField(),
                    "Message sent. Waiting for response");
                    tcp.send("SET " + ((Heater) device).toStringCommand());
                }
            } else if (device instanceof Vent) {
                if (value < 35 || value > 65) {
                    css.toError(((Vent) device).getTextField(),
                    "Min: 35 - Max: 65");
                } else {
                    ((Vent) device).setSetHum(value);
                    css.toBeApproved(((Vent) device).getTextField(),
                    "Message sent. Waiting for response");
                    tcp.send("SET " + ((Vent) device).toStringCommand());
                }
            }
        } catch (NumberFormatException nfe) {
            if (device instanceof Heater) {
                css.toError(((Heater) device).getTextField(),
                "Insert number only");
            } else if (device instanceof Vent) {
                css.toError(((Vent) device).getTextField(),
                "Insert number only");
            }
        }
    }
```

4.2.3 Driver

The driver package contains classes for nodered communication methods including TCP and MQTT.

TCP

Two classes were created for TCP communication, TCPServerDriver.java for creating the connection and TCPDriverInstance.java for communication.

```
public TCPServerDriver() {
    TCPDriverInstance tcpInstance = TCPDriverInstance.getInstance();
    try {
        ServerSocket serverSocketR = new ServerSocket(9090);
        System.out.println("Server started on port 9090");
        //noinspection InfiniteLoopStatement
        while(true) {
            Socket clientSocketR = serverSocketR.accept();
            //System.out.println("New client connected: "
            + clientSocketR.getInetAddress().getHostAddress());
            tcpInstance.setClientSocketR(clientSocketR);
            new Thread(tcpInstance).start();
        }
    } catch (IOException e) {
        System.err.println("Error in TCPServerDriver: " + e.getMessage());
    }
}
```

To avoid recreating the class each time by assigning controllers, the TCPDriverInstance class is used as a single instance. If an instance already exists, it is returned, otherwise it is created and sent to the caller. The constructor is private to prevent the creation of the class outside the class itself and bypassing single instance management.

```
public static TCPDriverInstance getInstance() {
    if(instance == null) {
        instance = new TCPDriverInstance();
    }
    return instance;
}

private TCPDriverInstance() {}
```

The process of reading TCP messages is done in a separate thread so as not to block the GUI. Each time a new message is received, it is decrypted, as we will see in the next section on data encryption. If the message is decrypted correctly, it is sent to the *destinationSorting* method, which allows the message flow to be redirected according to the smarthome it belongs to.

```
@Override
public void run() {
    try (BufferedReader in = new BufferedReader(
    new InputStreamReader(socketReceive.getInputStream(),
    StandardCharsets.UTF_8))) {
        String message;
        while ((message = in.readLine()) != null) {
            message = aes.decrypt(message);
            System.out.println("RECEIVED: " + message);
            destinationSorting(message);
        }
    } catch (IOException e) {
        System.err.println("Error in TCPDriverInstance: " +
        e.getMessage());
    }
}
```

The procedure for sending a TCP message is very similar to the receiving one except for the method used for sending and the encryption procedure.

```
public void send(String message) {
    try {
        PrintWriter out = new PrintWriter(new OutputStreamWriter(
        new Socket("localhost", 9091).getOutputStream(),
        StandardCharsets.UTF_8));
        out.write(aes.encrypt(message));
        out.close();
        System.out.println("SENT: " + message);
    } catch (IOException e) {
        e.printStackTrace();
    }
}
```

MQTT v.5

MQTT (Message Queuing Telemetry Transport) is a lightweight and scalable messaging protocol widely used in the Internet of Things (IoT). In this project, it is used to receive real-time messages from the nodered simulator based on subscribed topics. First, a connection is made to the hemostat broker locally with loopback address 127.0.0.1 and port 1883 using a custom dashboard name. Once the connection is successful, the subscription to the topics takes place:

- smart_home/+/+/light/#
- smart_home/+/+/temperature/#
- smart_home/+/+/humidity/#

Given the tree structure of mqtt, the presence of the + and # enables filtering and obtaining only the required data. The first + allows to select both smarthome A and smarthome B while the second + allows to select all rooms.

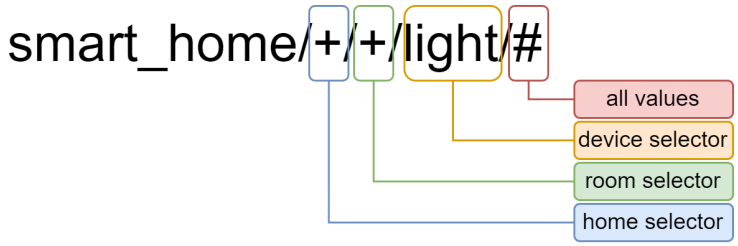


Figure 4.4: MQTT topic description

Once subscribed, the messageArrived method listens for any messages and after decrypting the message is sent to a *destinationSorting* with the same functions as the TCP one but suitable for incoming messages from mqtt in the topic and value structure.

```
private final String broker = "tcp://127.0.0.1:1883";
private final String clientId = "java_dashboard";
private final String topicL = "smart_home/+/+/light/#";
private final String topicH = "smart_home/+/+/temperature/#";
private final String topicV = "smart_home/+/+/humidity/#";

try {
    client = new MqttClient(broker, clientId);
    MqttConnectionOptions options = new MqttConnectionOptions();
    client.connect(options);

    client.setCallback(new MqttCallback() {
        public void messageArrived(String topic, MqttMessage message) {
            destinationSorting(topic,
            aes.decrypt(new String(message.getPayload())));
            System.out.println("MQTT: " + topic + " - "+
            aes.decrypt(new String(message.getPayload())));
        }
    });

    TCPDriverInstance.getInstance().setMqttRef(this);
    client.subscribe(topicL, subQos);
    client.subscribe(topicH, subQos);
    client.subscribe(topicV, subQos);
} catch (MqttException e) {
    throw new RuntimeException(e);
}
```

4.2.4 Utils

The presence of the *utils* package allows all the utility classes to be grouped together, we find the class responsible for the styling of the graphic elements and the class intended to manage the security of the transmitted data.

cssHelper

The cssHelper class is needed for the handling of the appearance of graphic elements especially in the input and feedback phase. Whenever it is necessary to change the style of a component, it is set to default, i.e. without any class, after which the correct class is assigned. This avoids the assignment of several classes of the same style. The presence of tooltips when mousing over the element allows the user to better understand the error problem.

```
public void toError(Control c, String tooltipText) {
    toDefault(c);
    c.getStyleClass().add("field-error");
    if(tooltipText != null) {
        Tooltip t = new Tooltip(tooltipText);
        t.getStyleClass().add("tooltip-error");
        t.setShowDelay(Duration.ZERO);
        c.setTooltip(t);
    }
}

public void toValid(Control c) {
    toDefault(c);
    c.getStyleClass().add("field-valid");
}

public void toBeApproved(Control c, String tooltipText) {
    toDefault(c);
    c.getStyleClass().add("field-to-be-approved");
    if(tooltipText != null) {
        Tooltip t = new Tooltip(tooltipText);
        t.getStyleClass().add("tooltip-to-be-approved");
        t.setShowDelay(Duration.ZERO);
        c.setTooltip(t);
    }
}
```

```
public void toInfo(Control c, String tooltipText) {
    toDefault(c);
    c.getStyleClass().add("field-info");
    if(tooltipText != null) {
        Tooltip t = new Tooltip(tooltipText);
        t.getStyleClass().add("tooltip-info");
        t.setShowDelay(Duration.ZERO);
        c.setTooltip(t);
    }
}

public void toDefault(Control c) {
    c.getStyleClass().remove("field-valid");
    c.getStyleClass().remove("field-be-approved");
    c.getStyleClass().remove("field-info");
    c.getStyleClass().remove("field-error");
    c.setTooltip(null);
}
```

Encryption

This class is also created as a single instance to avoid the creation of multiple identical classes when using the dashboard. The objective of this class is to provide the methods required to encrypt and decrypt text strings according to chosen algorithms. In our case, AES is used in ECB mode with padding, which is the simplest of the AES-based encrypion modes and allows:

- **Simplicity**: Each block of plaintext is independently encrypted using the same key, which makes the encryption and decryption process straightforward and easy to understand.
- **Parallelisation**: As each block is encrypted independently, ECB mode allows parallel processing, improving encryption speed on compatible hardware.
- **Localised Errors**: In the event of a transmission error, the problem is limited to the affected block without propagating to other blocks, which facilitates error handling.

However, it has several disadvantages:

- **Vulnerability to cryptanalysis**: A serious flaw of ECB mode is that it produces the same cipher text for identical plaintext blocks. This behaviour makes the mode susceptible to cryptographic analysis, as an attacker can recognise patterns and repetitions in the ciphertext, compromising overall security.
- **Lack of integrity**: There is no guarantee that the message remains intact. An attacker could manipulate ciphered blocks, reversing or replacing them, without the victim noticing.

Despite these problems that cannot be overlooked, it is still a good encryption method given the amount of data that could be transmitted in the system and the absence of critical information.

```
private final SecretKey SECRET_KEY =
getSecretKey("InnovativeTelecommunication+2024");

public String encrypt(String plainText) {
    try {
        Cipher cipher = Cipher.getInstance("AES/ECB/PKCS5Padding");
        cipher.init(Cipher.ENCRYPT_MODE, SECRET_KEY);
        return Base64.getEncoder().encodeToString(
        cipher.doFinal(plainText.getBytes()));
    } catch (NoSuchAlgorithmException | NoSuchPaddingException |
    IllegalBlockSizeException | BadPaddingException |
    InvalidKeyException e) {
        e.printStackTrace();
    }
    return null;
}

public String decrypt(String ciphertext) {
    try {
        Cipher decryptCipher = Cipher.getInstance("AES/ECB/PKCS5Padding");
        decryptCipher.init(Cipher.DECRYPT_MODE, SECRET_KEY);
        return new String(decryptCipher.doFinal(Base64.
        getDecoder().decode(ciphertext)));
    } catch (NoSuchAlgorithmException | NoSuchPaddingException |
    IllegalBlockSizeException |
    BadPaddingException | InvalidKeyException e) {
        e.printStackTrace();
    }
    return null;
}
```

Chapter 5

Node-Red Flows

This chapter provides an in-depth look at the Node-RED flows that act as the command center for the smart home automation system. Through the meticulous design of these flows, the system automates essential tasks: from maintaining optimal environmental conditions, handling user authentication, to generating, tracking energy usage and displaying data through an interactive dashboard.

5.1 ESP8266

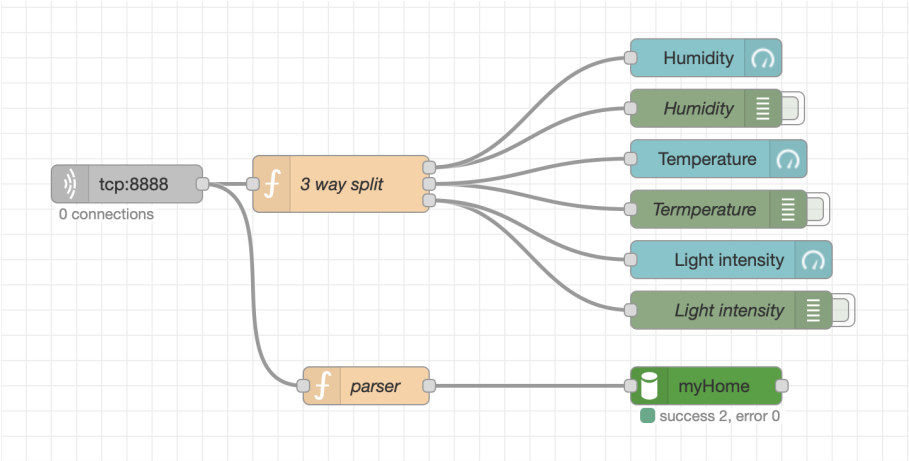


Figure 5.1: ESP8266 Flow

The flow in Fig. 5.1 is designed to capture and process real-world value from a physical home environment using a ESP8266 development board equipped with various sensors. This board includes DHT11 sensors for temperatures and humidity detection, as well as photoresistors for measuring light intensity. The purpose of this flow is to gather accurate baseline data, which will later be used to randomize and generates realistic workflows for the smart home automation system. The board continuously transmit data over TCP. Upon receiving the values are formatted in order to be routed to gauge nodes that visualize the information. These provides a live monitoring interface, allowing users to observe the current environmental conditions as detected by the development board. In parallel, the raw data are processed and structured in order to be stored in the *'myHome'* database collection.

This step is crucial, as the collected data will serve as a reference point for generating realistic,randomized values that simulate various scenario within the smart home system.

5.2 TCP Java

In the smart home system, Node-RED and Java communicate over a TCP connection, both listening on a specific port to exchange messages that execute various operations. Node-RED receive incoming request from Java using the flow in Fig.5.2.

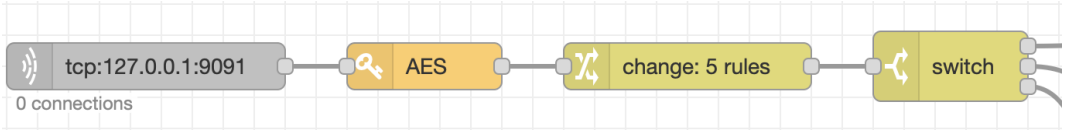


Figure 5.2: Java/Node-RED TCP Communication Flow

Upon receiving a request, Node-RED analyzes the message and process it. Once the operation is executed, Node-RED formulates an appropriate response based on the outcome. This response is then sent back to the Java application, as the flow in Fig.5.3 demonstrates, allowing to handle the result of the operation.

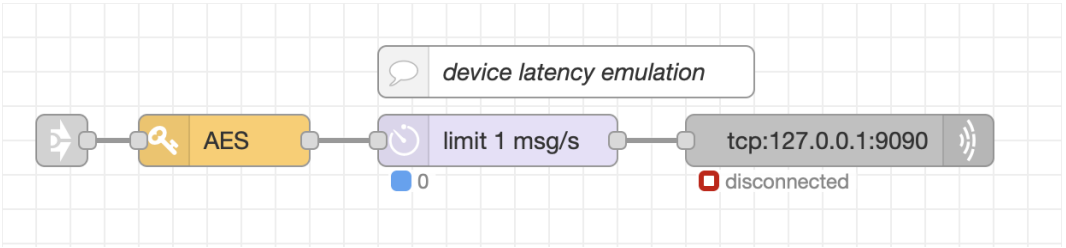


Figure 5.3: Node-RED/Java TCP Communication Flow

Node-RED also simulates network latency by introducing a *'delay'* node, which restricts the output to 1 message per second, providing a more realistic communication environment. It's also crucial to apply an encryption algorithm, using the *'AES'* node, ensuring that all exchange messages are private and the communication remains secure.

5.2.1 Operations

Initially, as the Fig.5.4 describes, Node-RED utilizes a *'switch'* node to determine which operation it needs to perform based on the incoming messages. It checks the content of the *'msg.payload'* using regular expression to match specific command.

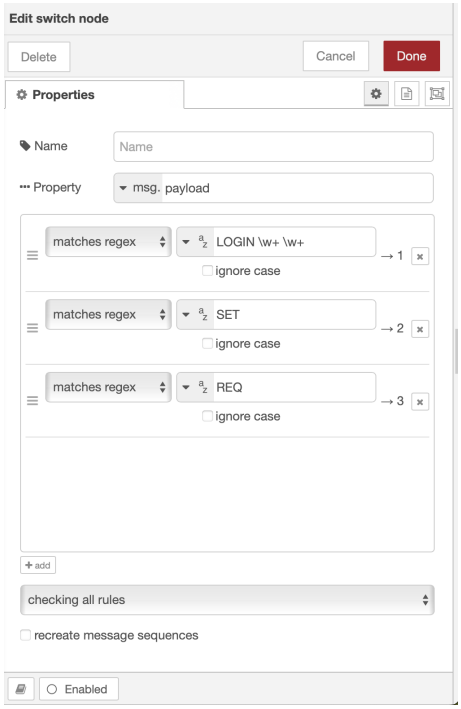


Figure 5.4: Incoming TCP Message Structure Checking

This ensures real-time communication and management of key functionalities such as:

- **Login Operation**: checked using regex pattern **LOGIN \w+ \w+**.
- **Data Initialization Operation**: checked using regex pattern **REQ**.
- **Device Control Operation**: checked using regex pattern **SET**.

By using this switch mechanism, Node-RED can efficiently route the incoming messages to the appropriate flow based on the specific operation required.

Login Operation

Figure 5.5: Login Operation Flow

The login process is initiated when a user attempts to log into the Java application. Here's how the interaction shown in *Fig.5.5* works.

1. The Java application listens for a user login attempt and, upon receiving the login credentials, forwards a TCP request to Node-RED.
2. Node-RED receives this request and is responsible for validating the user credentials, it checks if the provided username and password are correct, by querying the *'Users'* collection in MongoDB.

Once the validation process is completed, if the credential are correct, Node-RED sends response back to Java using the following message format:

```
msg.payload = "LOGIN OK " + msg.payload[0].home
```

This message confirms a successful login, with *'mag.payload[0].home'* referring to the user's associated smart home.

If the credential are incorrect, Node-RED sends a message indicating the failure:

```
msg.payload = "LOGIN KO"
```

This signals to the Java application that the login attempt has failed, allowing Java to handle the appropriate UI response.

Data Initialization Operation

When the Java application is started, it sends a **REQ** command to Node-RED to retrieve the latest system values and initialize the user interface with the most up-to-date information. This operation ensures that the application displays the correct statuses for the smart home devices when it launches.

After receiving the command, Node-RED formats the incoming payload to facilitate processing. The *'splitter'* node divides the payload into different components and organized as follow:

```
let splitted = msg.payload.split(" ");

msg.object = {
    "command": splitted[0],
    "home": splitted[1],
    "value": splitted[2]
};

return msg;
```

Where:

- *msg.object.command*: stores the operation (REQ).
- *msg.object.home*: stores the home identifier.
- *msg.object.value*: carries any additional information or parameters required for the request.

Based on the home specified in *'msg.object.home'*, Node-RED performs a series of *'find'* operations on the MongoDB database to retrieve the necessary initialization data, including:

- The current alarm status;
- The set point values for temperature and humidity;
- The current status and measurements of temperature and humidity from sensors across the home.

These values are required to update the user interface and device states within the application.

After retrieving the required data, Node-RED sends the results back to the Java application in two distinct formats, depending on the device type. For the alarm system, the status is sent back using the following format:

```
SET {{alarm.idx}} {{alarm.home}} undefined {{alarm.status}}
undefined undefined
```

For temperature and humidity, the current readings, are sent using this format:

```
SET {{payload.idx}} {{payload.home}} {{payload.room}}
{{payload.status}} {{payload.current}} {{payload.setpoint}}
```

Device Control Operation

When Node-RED receives a **SET** command from the Java application, it processes the command in the *'splitter'* function node by analyzing the message payload and formatting it according to the logic required for device control, using the following code:

```
let splitted = msg.payload.split(" ");

msg.object = {
    "command": splitted[0],
    "id": splitted[1],
    "home": splitted[2],
    "room": splitted[3],
    "value": splitted[4]
};

return msg;
```

Where:

- *msg.object.command*: stores the operation (SET).
- *msg.object.id*: contains the device ID
- *msg.object.home*: stores the home identifier.
- *msg.object.room*: stores the room identifier.
- *msg.object.value*: contains the specific value or action for the device.

Node-RED uses a *'switch'* node (Fig.5.6) to identify which device is being controlled, based on the *'id'* field. Depending on the device type, different actions are executed:

- **Alarm System Control**;
- **Temperature and Humidity Control**

Figure 5.6: TCP Message Device Type Checking

If the device is an alarm system, and *'msg.object.value'* is equal to "TOGGLE", it means that the user has requested to either activate or deactivate the alarm. Node-RED checks the current alarm state from the MongoDB database and perform the following actions: if the alarm is off, it will arm the system, otherwise it will deactivate. Node-RED updates the alarm status in the MongoDB database to reflect the new state. After completing this operation, it sends a confirmation message back to Java in the following format:

```
SET {{object.id}} {{object.home}} {{object.room}}
{{object.value}} undefined undefined
```

For temperature and humidity settings, the user sets a target value (*'setPoint'*) for a specifc room, from the Java application. Node-RED processes this by:

1. Updating the MongoDB collection with the new setPoint for temperature or humidity.
2. Comparing the current temperature/humidity value (measured by the sensor data) with the setPoint: if the current value is lower than the setPoint, node red may trigger the heating system or HVAC system to turn on in order to adjust the environment.

Once the actions are complete, Node-RED sends the following message to Java, detailing the status of the device.

```
SET {{payload.idx}} {{payload.home}} {{payload.room}}
{{payload.status}} {{payload.current}} {{payload.setpoint}}
```

5.3 Sensor Selector

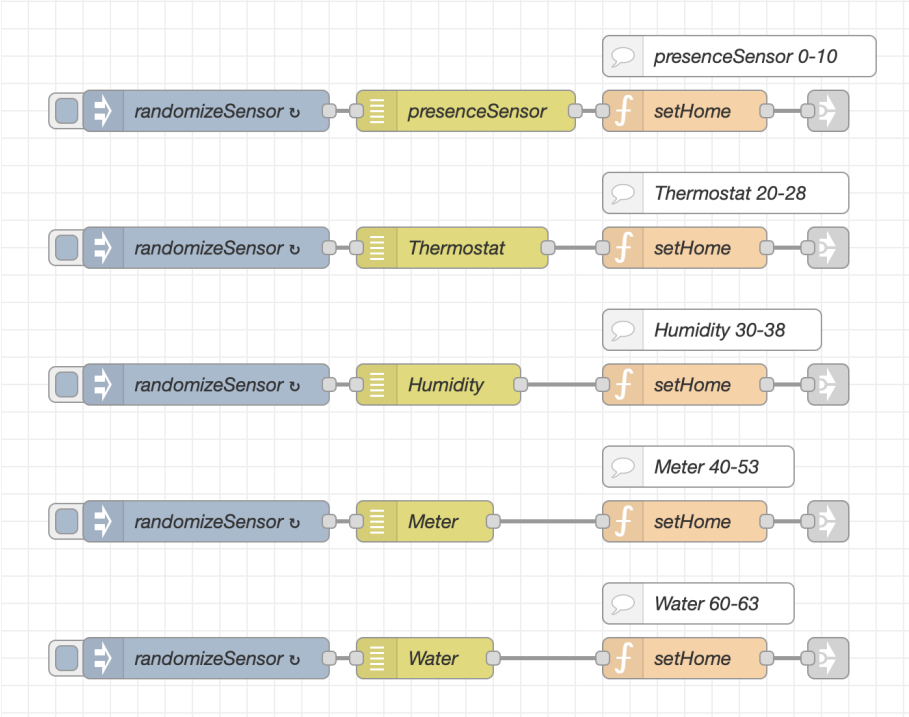


Figure 5.7: Sensor Selector Flow

The sensor selector flow represented in Fig.5.7, is developed to simulate the random selection of different smart home objects and record specific details about them. The inject node triggers automatically at predefined intervals, as shown in the Tab. 5.1. When activated, it initiates the flow by sending a signal to the next node in the sequence.

Item	Activation Time
Presence Sensor/Light	10 seconds
Thermostat	10 seconds
Humidity	10 seconds
Meter Simulator	15 minutes
Water Simulator	30 minutes

Table 5.1: Item activation time

Each Flow represents a different type of smart home sensor or device, such as presence sensors, thermostats, humidity sensors, meters consumption simulator and water simulator. The *'randomizeSensor'* randomly select a specific object within the smart home environment, identified by a unique ID [1]. After a sensor or device is randomly selected, the *'setHome'* function node processes the information, saving relevant data about the selected object, including:

- **Smarthome**: A,B.
- **Room**: bathroom, livingroom, bedroom, kitchen, hallway, stairways.
- **Item**: air conditioning, home theater, smartTV, oven, stove, etc.

Each flow ends with an output node that directs the process toward the continuation of the behavior, ensuring that the system can proceed with further actions based on the data collected in the previous steps.

[1]The entire list of devices is available here: https://dropover.cloud/1a0683.

5.4 Sensors Light Simulator

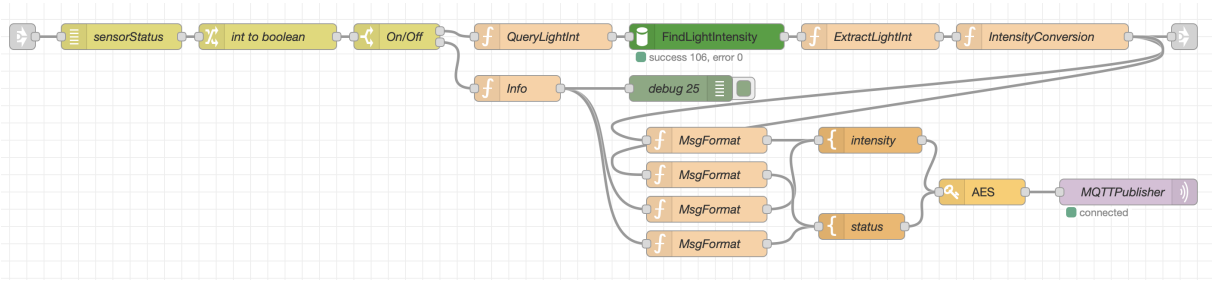


Figure 5.8: Sensor Light Simulator Flow

The Fig.5.8 shows how to control the switching on and off of lights based on the presence of a person in a specific room. The flow starts by selecting a sensor from the previous action, If the sensor is active, it indicates that someone has entered the room. Automatically, the light is switched on. Then, the process works as follows:

- **Intensity Determination**: The intensity is adjusted to optimize energy consumption. To determine the light intensity, the system retrieves the latest reading from a development board stored in the MongoDB database. Basically, if it's daytime, the intensity is reduced; otherwise, it's increased. This value (from 0 to 1024) is then converted to estimate a value between 0 and 1, in order to use to regulate the java progress bar that shows the light intensity for the specified room.
- **Consumption Calculation**: Once the light is switched on, it begins to consume energy. The link out node directs the flow toward the process responsible for calculating the energy consumption.
- **Publishing Information**: Information such as light intensity and status is published to a specific topic using the MQTT protocol. Before publishing, all data is encrypted using the AES algorithm and a private key to ensure security.

```
//MQTT TOPICS

//Status
`smart_home/${home}/${room}/light/status`
//Intensity
`smart_home/${home}/${room}/light/intensity`
```

It's important to observe that the topic is dynamically constructed based on the variable *'home'* and *'room'* derived from the *'setHome'* function of the *'sensorSelector'* flow.

5.5 Thermostat Simulator

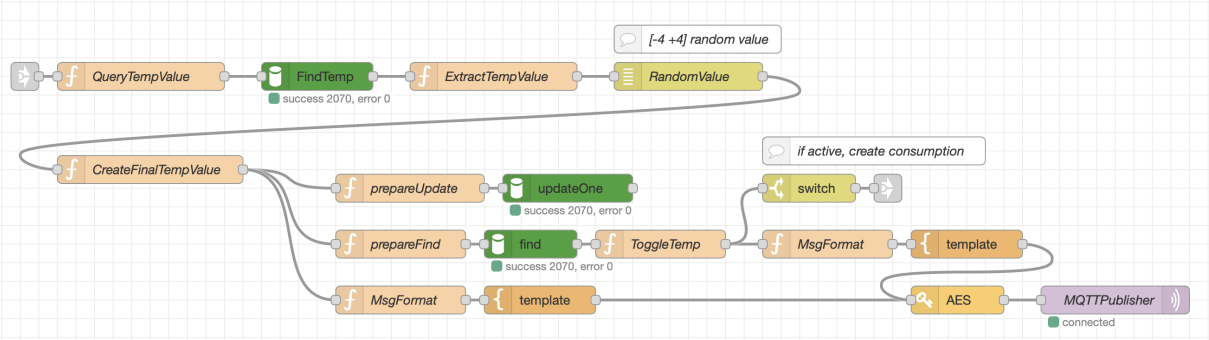


Figure 5.9: Thermostat Simulator Flow

This flow (Fig. 5.9) monitors the temperature in each room of the smart home. After selecting the temperature sensor from the flow *'sensorSelector'*, a random value is generated to represent the actual temperature inside the corresponding room. To ensure realistic values according to the season, the last saved reading from the DHT11 sensor is retrieved from MongoDB. A random temperature is then generated by adding a value between -4 and +4 to this reading, allowing for varying the temperature each time. The currently generated temperature is saved back into MongoDB. Then different scenario are possible:

- **Heating Activation**: The flow retrieves, from the same database collection, the ideal temperatures values set by user via the Java interface. If the temperature obtained from the sensor is lower that the set threshold, it indicates that the heating systems needs to be turned on in that room.
- **Energy Consumption Calculation**: Once the heating system is activated, it begins to consume energy. The link out node directs the flows towards the process responsible for calculating energy consumption.
- **Publishing Information**: By maintaining the same behavior, information such as the temperature value and the activation status of heating system is encrypted and then published to a specific MQTT topic. It is derived dynamically based on home and room information.

```
//MQTT TOPICS

//Status
`smart_home/${home}/${room}/temperature/status`
//Value
`smart_home/${home}/${room}/temperature`
```

5.6 Humidity Simulator

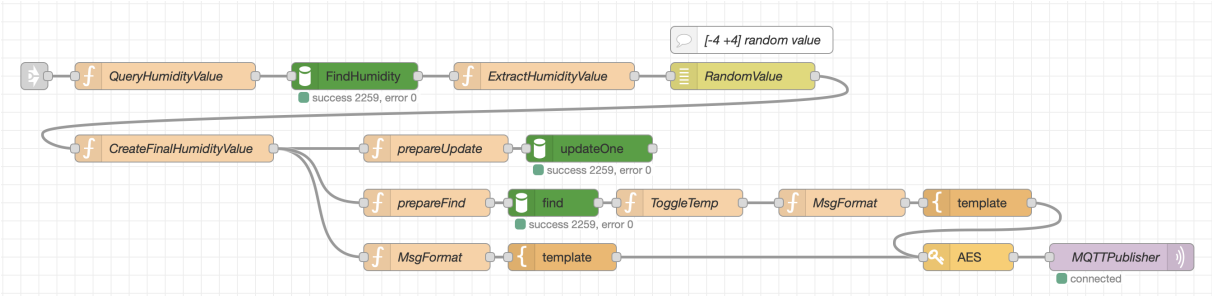


Figure 5.10: Humidity Simulator Flow

The logic behaviour described in 5.10, is similar to the *'Thermostat Simulation'* flow . However it differs by information and data used. This flow creates a loop system for humidity control, integrating sensor data, database storage, decision making based on setpoint, and secure communication with hardware device via MQTT. The flow starts by querying the real humidity value from MongoDB, where measurements from DHT11 sensor are stored. To introduce variability while maintaining realism, a random node between -4 and +4 is generated. This random value is then added to the actual humidity reading, creating a slightly altered but still realistic humidity value. This current humidity value is saved back to MongoDB. Simultaneously, this value is published to a specific MQTT topic for real-time monitoring. Then, the main functionalities are:

- **Threshold Checking**: The flow checks if the current humidity values is below a user-defined threshold. This setpoint is stored in the same MongoDB collection as the humidity data.
- **HVAC Control**: If the humidity falls below the setpoint, the system determines that the HVAC needs to be activated to regulate humidity. The activation status is published via MQTT, allowing the hardware device controlling the HVAC to read and respond to this command.

As before described, all data transmitted through MQTT is encrypted, as indicate by the *'AES'* node before the *'MQTTPublisher'* operations on the following dynamics topics:

```
//MQTT TOPICS

//Status
`smart_home/${home}/${room}/humidity/status`
//Value
`smart_home/${home}/${room}/humidity`
```

5.7 Alarm Simulator

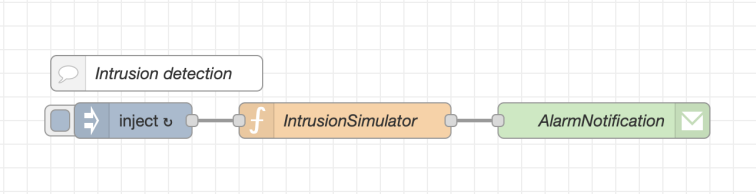


Figure 5.11: Alarm Simulator Flow

The flow shown in Fig.5.11 implements a system that continuously monitors the alarm status of both houses (A,B) and simulates potential intrusions, sending email notifications when an intrusion is detected. The intrusion simulation logic, written in the *'IntrusionSimulation'* function, performs the following steps:

```
const statusA = global.get('Alarm_status_A');
const statusB = global.get('Alarm_status_B');

//Intrusion probability (1%)
const intrusionProbability = 0.01;

//Intrusion Simulator
const isIntrusion = Math.random() < intrusionProbability;

if(isIntrusion) {
    const now = new Date();
    const formattedDate = now.toLocaleString('it-IT',
    { timeZone: 'Europe/Rome' });

    //Intrusion details
    const sensorTriggered = ['hallway', 'kitchen', 'livingroom',
    'bedroom', 'bathroom'][Math.floor(Math.random() * 5)];

    msg.payload = ...;

    if (statusA == 'ARMED') {
        msg.topic = 'ALARM HOME A: Intrusion Detected!';
        return msg;
    } else if (statusB === 'ARMED') {
        msg.topic = 'ALARM HOME B: Intrusion Detected!';
        return msg;
    }

}
return null;
```

1. Retrieves the current alarm status for both houses, from global variables: these variables changes the status based on the manual activation of the alarm in the Java interface.
2. Sets a low probability for an intrusion to occur.
3. Generates a random number to determine if an intrusion happens.
4. If an intrusion is simulated:
   (a) Captures some information like date, time and randomly selects a sensor location.
   (b) Creates a **payload** and **topic** with intrusion details for the email notifications.

This approach allow for realistic simulation of a home security system, providing a way to test and demonstrate the functionalities of a modern alarm system, from detect potential issues to inform the homeowner (Example in Fig. 5.12).

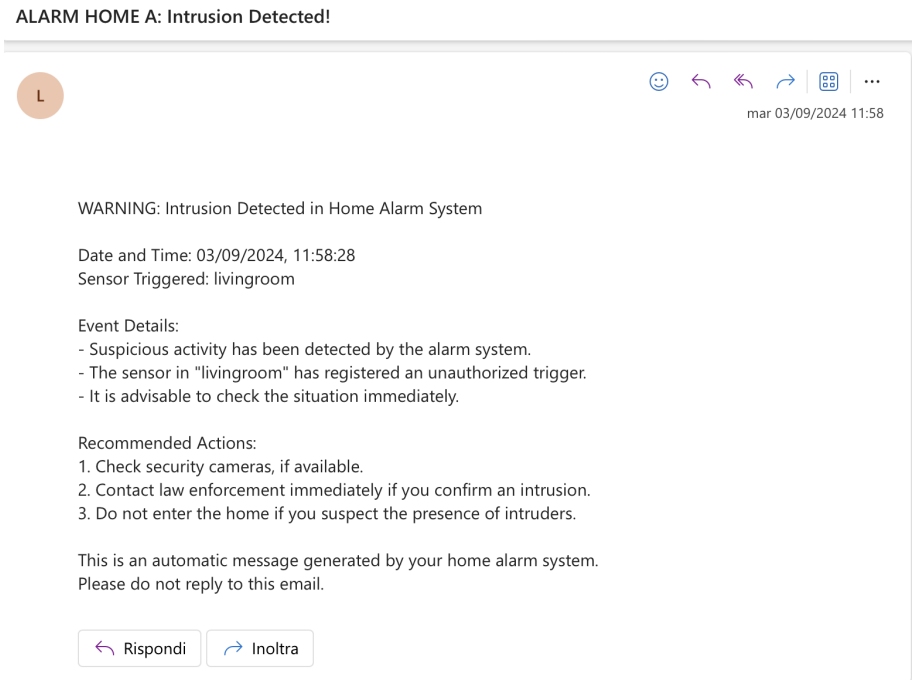


Figure 5.12: Example of alarm email notification

5.8 Meter Simulator

The Meter Simulator is a crucial component of a smart home system. This section outlines the methodology used for emulating the energy consumption patterns of various household devices. The meter simulator is divided into two distinct parts:

- **Java Hardware Consumption**: This relates to the random consumption generated by Node-RED when the hardware, represented in Java, is activated.
- **Node-Red Consumption**: This involves the simulation of consumption, which are not represented in Java. The activation and selection of these devices are automated and randomized by Node-RED, generating consumption based on real metrics and usage combination that vary for each smart home.

For a clear and concise understanding, please refer to Tab. 5.2.

Device	Room	Consumption Categories
Light	All Rooms	Java Hardware Consumption
Temperature System	Stairways	Java Hardware Consumption
Alarm System	Stairways	Java Hardware Consumption
Laundry Machine	Bathroom	Node-Red Consumption
Air Conditioning	Livingroom,Bedroom	Node-Red Consumption
Home Theater	Livingroom	Node-Red Consumption
Smart TV	Livingroom	Node-Red Consumption
Stove	Kitchen	Node-Red Consumption
Oven	Kitchen	Node-Red Consumption
Water	Kitchen,Bathroom	Node-Red Consumption

Table 5.2: Device Consumption Categories

Before starting to introduce how data are generated, it's essential to know that **all information is calculated based on guiding metrics according to the type of device**, stored in MongoDB in the *'setting'* collection. For detailed explanation please see Sec.3.1.5. In addition, all data is encrypted using the AES algorithm with a private key before being published to specifics topics via the MQTT protocol. This allow the simulated data to be communicated to other parts of the smart home system or to external monitoring and control applications that subscribe to these topics. Finally, the generated consumption is also saved in two distinct collections (*'ConsA'* - *'ConsB'*) depending on the smart home, identified by *'msg.home'* based on the device ID, and transmitted between the various nodes. To observe the format of the saved documents, please refer to Sec.3.1.6.

5.8.1 Java Hardware Consumption

In this mode, energy consumption is triggered by specific hardware events within the smart home environment. In summary, the specific device begins to consume energy under the following conditions:

- **Presence Detection-Light**: When a sensor detects presence in a room, the light turn on, generating instant energy consumption.
- **Heating Control-Temperature System**: If the detected room temperature falls below the threshold set by the user, the heating system is activated, leading to immediate energy consumption.
- **Alarm Activation-Alarm System**: When the user enables the alarm system, the activation state itself incurs energy consumption.

Lights

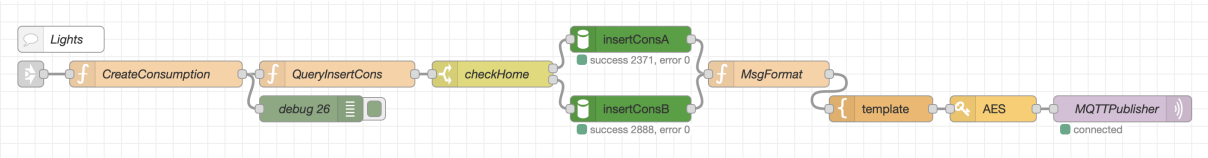


Figure 5.13: Light Consumption Simulator Flow

This flow (Fig.5.13) simulates light consumption in a smart home system. Unlike other devices, the light consumption is not based on preset values retrieved from a database, but is dynamically generated in the *'createConsumption'* function:

```
//base power (watt) for each room
let powerBase;
switch (msg.room) {
    case 'livingroom':
        powerBase = 60;
        break;
    case 'kitchen':
        powerBase = 100;
        break;
    case 'bedroom':
        powerBase = 40;
        break;
    case 'bathroom':
        powerBase = 75;
        break;
    default:
        powerBase = 50;
        break;}
```

```
//retrieve the intensity
let intensity = msg.intensity;

let actualPower = powerBase * intensity; //watt

//random time generator
let minTimeMs = 1 * 60 * 1000;
let maxTimeMs = 3 * 60 * 60 * 1000;
let randomTimeMs = Math.random() * (maxTimeMs - minTimeMs) + minTimeMs;

//hours conversion
let durationHours = randomTimeMs / (1000 * 60 * 60);
durationHours = parseFloat(durationHours.toFixed(2));

//kwh consumption
let energyConsumed = (actualPower * durationHours) / 1000;
energyConsumed = parseFloat(energyConsumed.toFixed(4))

msg.value = energyConsumed;

msg.payload = `The ${msg.room}'s light (idx = ${msg.idx}),
ran for ${durationHours} hours, with a consumption of ${msg.value} kWh.`;

return msg;
```

Several factors are considered:

- **Light Wattage**: The power consumption of light varies depending on the room. For example, a living room might have higher wattage compared to a bedroom.
- **Light Intensity**: The flow takes into account the brightness level of the lights, which can affect energy consumption.
- **Random On-time**: To simulate realistic usage patterns, the system generates random duration for how long the lights remain on.

The *'msg.value'* now represents the light energy consumption, which is published by the MQTT node to an MQTT broker, on the following topic:

```
//MQTT TOPIC

//Value
`smart_home/${home}/${room}/consumption/light`
```

Temperature System

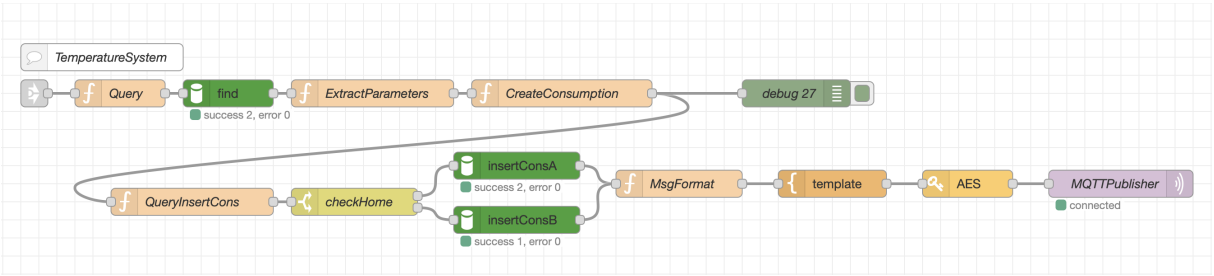


Figure 5.14: Temperature System Consumption Simulator Flow

The flow in Fig.5.14 simulates and manages the energy consumption of a temperature control system in a smart home environment. The flow begins by extracting basic metrics form a database, primarily the base power consumption of the temperature system. Then, the process is developed in the *'CreateConsumption'* function:

```
//base power (kW)
let basePower = msg.power;

//room which the heating is being actived
let room = msg.room;

//consumption Multipliers based on the room size
let sizeMultiplier;
switch (room) {
    case 'kitchen':
        sizeMultiplier = 1.2; // +20%
        break;
    case 'hallway':
        sizeMultiplier = 0.8; // -20%
        break;
    case 'stairways':
        sizeMultiplier = 0.9;
        break;
    case 'bedroom':
        sizeMultiplier = 1.1;
        break;
    case 'livingroom':
        sizeMultiplier = 1.5;
        break;
    case 'bathroom':
        sizeMultiplier = 1.3;
        break;
    default:
```

```
            sizeMultiplier = 1.0;
            break;
    }

    let actualPower = basePower * sizeMultiplier;

    //random time generator
    let minTimeMs = 15 * 60 * 1000;  //15 minutes
    let maxTimeMs = 4 * 60 * 60 * 1000; //4 hours
    let randomTimeMs = Math.random() *
    (maxTimeMs - minTimeMs) + minTimeMs;

    //hours conversion
    let durationHours = randomTimeMs / (1000 * 60 * 60);
    durationHours = parseFloat(durationHours.toFixed(2));

    //consumption (kWh)
    let energyConsumed = actualPower * durationHours;
    energyConsumed = parseFloat(energyConsumed.toFixed(2));

    msg.value = energyConsumed;

    msg.payload = `The temperature system (home = ${msg.home})
    activated to heat ${room} for ${durationHours} hours,
    consumed ${msg.value} kWh.`;

    return msg;
```

Here's a short explanation of the factors used:

- **Base Power**: the fundamental power requirement of the system.
- **Area-based Consumption Increase**: Larger rooms require more energy to heat, so the consumption is adjusted based on room size.
- **Random Usage Time**: To simulate realistic usage patterns, the system applies a random duration of operation.

The *'msg.value'* now represents the temperature system energy consumption, which is published on a specific topic with the IoT application protocol:

```
    //MQTT TOPIC

    //Value
    `smart_home/${home}/stairways/consumption/temperature_system`
```

Alarm System

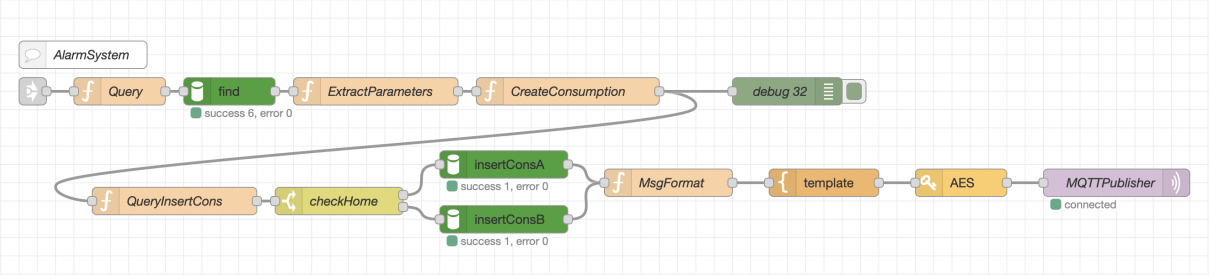


Figure 5.15: Alarm System Consumption Simulator Flow

Before generating the alarm system consumption, the flow in Fig.5.15 starts by extracting the setting parameter for the alarm, the base power. The *createConsumption* node generates the simulated energy consumption of the alarm. The code starts by:

```
//check the home
let home = msg.object.home;

//retrieve info from the global variable
let globalVarName = (home === 'A') ? "alarmDataA" : "alarmDataB";
let alarmData = global.get(globalVarName);
```

This part shows that, after checking which house is using the alarm, it retrieves the alarm data specific to that house from a global variable, which includes the actual time the alarm has been active, which is calculated using the following process in the *'setTime'* function in *'TCPJava'* flow:

```
//get the current status of the alarm
let alarmState = msg.object.value;
let home = msg.object.home;

//current timestamp
let currentTimestamp = Date.now();

//global variable to used based on the smart home
let globalVarName = (home === 'A') ? "alarmDataA" : "alarmDataB";
let alarmData = global.get(globalVarName) || {};

//"armed"= save the timestamp
if (alarmState === "ARMED"){
    alarmData.startTime = currentTimestamp;
    node.warn("Alarm armed at "
    + new Date(currentTimestamp).toLocaleString()
    + " for home " + home);
    msg.goCons = false;}
```

```
//"deactivated" = calculates the time differences
if (alarmState === "DEACTIVATED" && alarmData.startTime){
    let durationMs = currentTimestamp - alarmData.startTime;

    //conversion
    let durationSeconds = Math.floor(durationMs / 1000);
    let durationMinutes = Math.floor(durationSeconds / 60);
    let durationHours = Math.floor(durationMinutes / 60);
    durationMinutes = durationMinutes % 60;
    durationSeconds = durationSeconds % 60;

    //save duration time in the global log
    alarmData.duration = {
    hours: durationHours,
    minutes: durationMinutes,
    seconds: durationSeconds
    };

    node.warn("Alarm disarmed after " + durationHours
    + " hours, " + durationMinutes + " minutes, and "
    + durationSeconds + " seconds for
    home " + home);

    //reset start time
    alarmData.startTime = null;
    msg.goCons = true; //go consumption
}

//store the updated global variable
global.set(globalVarName, alarmData);

return msg;
```

The system continuously monitors the alarm state for each smart home:

- **Activation Time Tracking**: When the alarm is **"armed"** the current timestamp in recorded as the start time. This information is stored in a global variable specific to each home, thanks to *'msg.object.home'*.
- **Deactivation and Duration calculation**: When alarm is disarmed (**"deactivated"**) the system calculates the duration by subtracting the start time from the current time. After converted, the duration is stored in the global variable for the respective home (Fig.5.16).

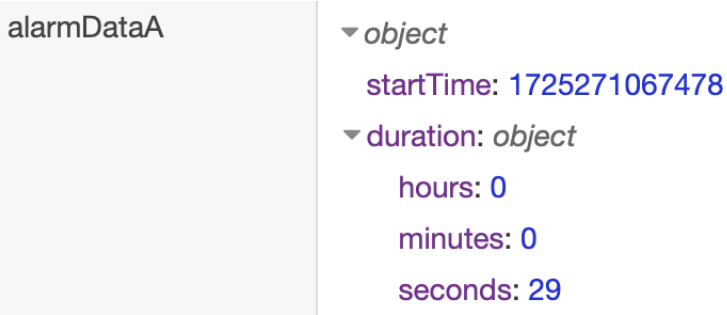


Figure 5.16: Alarm A Global Variable Representation

- **Consumption Calculation Trigger**: Upon deactivation, a flag (*'msg.goCons'*) is set to true, triggering the energy consumption calculation in the *'MeterSimulator'* flow.

This detailed tracking of alarm states and duration ensures that the energy consumption calculation in the *'CreateConsumption'* node is based on the actual precise activation time of the alarm system.

Returning to the *'CreateConsumption'* function, it calculates the alarm activation duration in hours based on this active time and calculates energy consumption by multiplying base power by the actual activation duration:

```
//alarm activation time conversion in hours
let durationHours = alarmData.duration.hours
+ (alarmData.duration.minutes / 60)
+ (alarmData.duration.seconds / 3600);

//alarm base power (kW)
let basePower = msg.power || 0.05;

//consumption in kWh
let energyConsumed = basePower * durationHours;
energyConsumed = parseFloat(energyConsumed.toFixed(4));

msg.value = energyConsumed;

msg.payload = `The alarm system for home ${home} was active
for ${durationHours.toFixed(2)} hours, consuming ${msg.value} kWh.`

return msg;
```

In this way, the *'msg.value'* now represents the alarm system energy consumption, which is published on a specific topic via MQTT:

```
//MQTT TOPIC
//Value
`smart_home/${home}/stairways/consumption/alarm_system`;
```

5.8.2 Node-RED Consumption

This category includes the generation of energy consumption for virtual devices randomly selected using their unique ID in the *'sensorSelector'* flow. Once a device is chosen and identified, it is either turned on or off. **This flow specifically simulates the potential activation of a physical device by the user within the smart home environment**. Since these hardware are not represented in the Java application, as mentioned earlier, the simulation provides a virtual representation of their behavior. If the device is turned on, it starts generating the associated power consumption, simulating its operation. The Fig. 5.17 illustrates this initial setup.

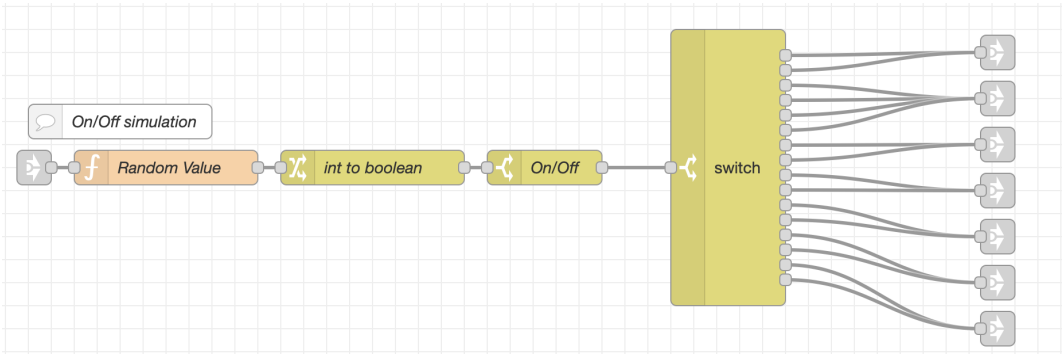


Figure 5.17: On/Off Simulator Flow

Laundry Machine

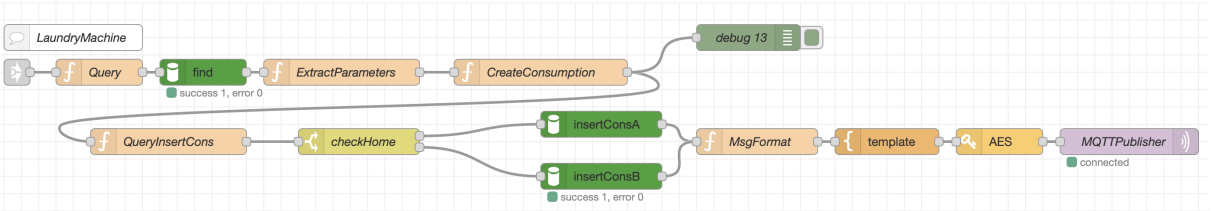


Figure 5.18: Laundry Machine Consumption Simulator Flow

The simulation in 5.18 begins by retrieving essential information from factory settings stored in a database (3.1.5). These settings include:

- **Base Power**;
- **Maximum and Minimum Operation Time**;
- **Available Cycles**;
- **Water Temperature**

Once these parameters are extracted:

```
let document = msg.payload[0];

//values extraction from JSON structure
msg.power = parseFloat(document.laundry_machine.power);
let maxTime = parseInt(document.laundry_machine.maxTime);
let minTime = parseInt(document.laundry_machine.minTime);
let cycles = document.laundry_machine.cycles;
let temperatures = document.laundry_machine.temperatures;
```

The simulation proceeds (*'extractParameters' node*) to mimic a real washing cycle. It randomly selects a wash cycle and a temperature from the available option and simulate a random operation time. This randomness reflects the varying conditions under which the machine might be used in a real-world scenario by the user.

```
//random index based on the cycles array length
let randomIndex = Math.floor(Math.random() * cycles.length);
//random index based on the temperatures array length
let randomIndexT = Math.floor(Math.random() * temperatures.length);

//random select a cycle from the list
msg.cycle = cycles[randomIndex];
//random select a temperature from the list
msg.temperatures = temperatures[randomIndexT];

//random time
msg.time = Math.floor(Math.random() * (maxTime - minTime + 1))
+ minTime;

return msg;
```

Different wash cycles consume different amounts of energy, for example a "cotton" cycle typically requires more power compared to and "eco" cycle. The simulation in *'createConsumption'* function, applies a specific consumption factor for the selected cycle, which adjust the base power accordingly. The temperature also influences the energy, with the assumption that the cost increases proportionally with the temperature. In other words, higher temperatures lead to higher energy consumption.

```
//cyles consumption factor
const cyclesCons = {
    "cotton": 1,
    "synthetics": 0.8,
    "eco": 0.5,
    "quick": 0.6,
    "delicates": 0.4,
    "wool": 0.7,
    "mixed": 0.9,
};

//temperatures consumption factor
let tempFattore = temperatures / 60;
```

Finally, the usage time is converted into hours and with the other key-metrics the total energy consumption in calculated in kwH.

```
let totPower = basePower * cyclesCons[cycle] * tempFattore;
let hourTime = msg.time / 60;

//consumption in kWh
let cons = totPower * hourTime;
msg.value = cons;

return msg;
```

The calculated energy consumption value is then published via MQTT to the following topic:

```
//MQTT TOPIC

//Value
`smart_home/${home}/${room}/consumption/laundry_machine`;
```

Air Conditioning

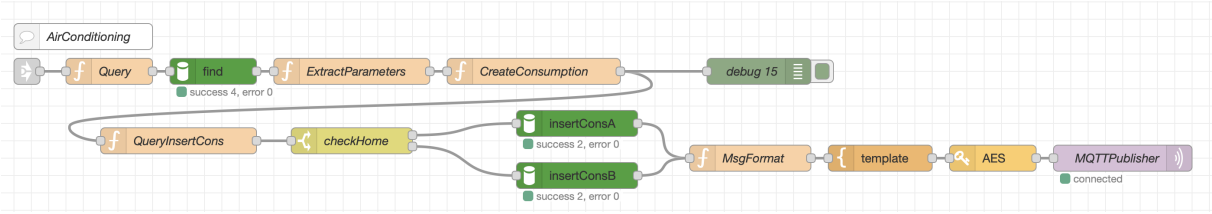


Figure 5.19: Air Conditioning Consumption Simulator Flow

The energy consumption simulation in 5.19, after querying to the database; needs to extract, in the *ExtractParameters* node, essential information (3.1.5), such as:

- **Base Power**;
- **Operating Mode**;
- **Fan Speed Settings**:
- **Temperature Range**

Based on these key parameters a random combination of air conditioner mode, fan speed and a temperature values (within the defined range) is selected for the simulation. This represents a scenario where different operating conditions lead to varying levels of energy consumption. The final formula written in *'CreateConsumption'* node takes into account different consumption factors for modes and fan speeds, adjusting for temperature variations.

```
//factors consumption based on mode
const modesCons = {
    "cooling": 1,
    "heating": 1.2,
    "fan_only": 0.3,
    "dry": 0.8,
};

//factors consumption based on fan speed
const fanSpeedsCons = {
    "low": 0.7,
    "medium": 1.0,
    "high": 1.2,
};

//factors consumption calculation
let tempFactor = (temperature - msg.minTemp) /
(msg.maxTemp - msg.minTemp);
let modeFactor = modesCons[mode] * fanSpeedsCons[fanSpeed];
```

```
//energy consumption in kWh
let totalConsumption = basePower * tempFactor * modeFactor;
msg.value = totalConsumption;

return msg;
```

For example, the combination of the "cooling" mode with the "low fan speed" consumes less energy compared to the "heating" mode with the "high fan speed". By simulating random operational conditions, the system provides a realistic estimation of energy usage, which is essential for optimizing performance and understanding consumption pattern. The calculated results are securely stored and published for further analysis and tracking on the following MQTT topic:

```
//MQTT TOPIC

//Value
`smart_home/${home}/${room}/consumption/air_conditioning`;
```

Home Theater

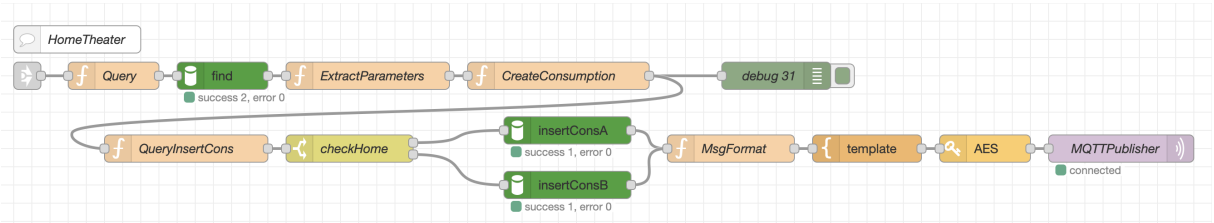


Figure 5.20: Home Theater Consumption Simulator Flow

The energy consumption simulation for the home theater system represented in Fig.5.20, is based on a **power value**, which is determined within a range extracted from the device settings stored in a MongoDB database (3.1.5) and a randomly generated **usage time**.

```
//random power generation
let randomPower = Math.random() * (msg.maxPower - msg.minPower)
+ msg.minPower;

//random time generator for usage
let minTimeMs = 30 * 60 * 1000; // 30 minutes
let maxTimeMs = 3 * 60 * 60 * 1000; // 3 hours
let randomTimeMs = Math.random() * (maxTimeMs - minTimeMs)
+ minTimeMs;
//hours conversion
let durationHours = randomTimeMs / (1000 * 60 * 60);
durationHours = parseFloat(durationHours.toFixed(2));
```

```
//energy consumption (kwH)
let energyConsumed = randomPower * durationHours;
msg.value = energyConsumed;

return msg;
```

In a realistic environment, a home theater system consists of multiple components, each contributing to the total energy consumption. Common elements include television, audio system, media players or consoles. Typically, the total power consumption of a home theater setup can peak between 500 to 1000 watts, depending on the number of devices in use. Once the energy usage is calculated for the simulated session, the consumption value is displayed and transmitted in real-time using the MQTT protocol, ensures that users can observe the power consumption live, thanks to the subscription on the following topics:

```
//MQTT TOPIC

//Value
`smart_home/${home}/${room}/consumption/home_theater`;
```

Smart TV

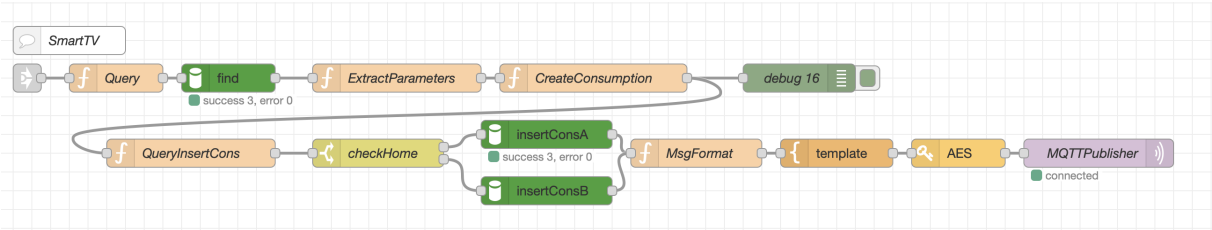


Figure 5.21: Smart TV Consumption Simulator Flow

As shown in the *ExtractParameters* function of the flow in Fig.5.21, the process begins by retrieving the primary information, the **base power**. With the time of use they form the basis for the final calculation of the energy consumed. The unique aspect of the smart TV's energy consumption simulation lies in the randomization of usage time, which is dynamically generated to reflect real-word usage patterns. An advanced algorithm adjust the usage time based on the current day and hour distinguishes between:

- Weekend - Weekday:
- Daytime - Evening/Night

```
//real day and hour
let now = new Date();
let currentHour = now.getHours();
let currentDay = now.getDay(); // 0 = sunday, 6 = saturday

let minUsage, maxUsage;
//randomization of usage time based on time slots and days
if (currentDay === 0 || currentDay === 6) {
    //Weekend
    if (currentHour >= 18 || currentHour < 6) {
        //Evening/night
        minUsage = 3;
        maxUsage = 6;
        }else if (currentHour >= 6 && currentHour < 18) {
            //Day
            minUsage = 2;
            maxUsage = 4;
        }
    } else {
        //Weekday
        if (currentHour >= 18 || currentHour < 6) {
            //Evening/night
            minUsage = 2;
            maxUsage = 4;
        } else if (currentHour >= 6 && currentHour < 18) {
            //Day
            minUsage = 0.5;
            maxUsage = 2;
        }
    }
//usage time generation
var randomTime = Math.random() * (maxUsage - minUsage) + minUsage;
```

This ensures that the usage time reflects typical user behavior. For example, people tend to watch TV more during the evening and on weekends, while daytime usage during morning/afternoon weekday is reduced to account form people's work or school schedules. Then, the consumption model calculates the energy by accounting for this variation:

```
let basePower = msg.power

//energy consumption in kWh
let totalConsumption = basePower * randomTime;

msg.value = totalConsumption;
```

Just like the other devices, the final accurate value are transmitted using the MQTT protocol. The data is sent by subscribing to the specific topic:

```
//MQTT TOPIC

//Value
`smart_home/${home}/${room}/consumption/smartTV`;
```

Stove

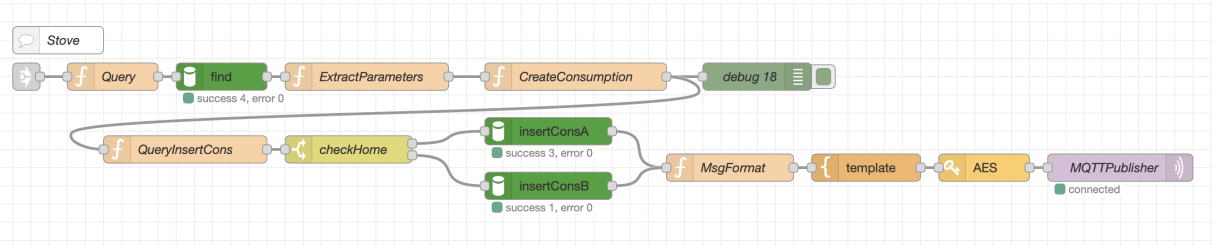


Figure 5.22: Stove Consumption Simulator Flow

In order to calculate the energy consumption by using the flow specified in Fig.5.22 it's essential to gather important key information from the *'setting'* collection (3.1.5), like:

- **Minimum Power**;
- **Maximum Power**;
- **Cooking Zones**

The energy consumption simulation, written in *'createConsumption'* function node, works by randomly determining how many of the four cooking areas (from 1 to 4) will be active during a cooking session.

```
let fornelli = []; //array for save data for each burners

//random generation of the number of burners to turn on
var numAccesi = Math.floor(Math.random()
* cookingAreas.length) + 1;

//randomly select the burners to turn on
var accesi = [];
while (accesi.length < numAccesi) {
    var randomIndex = Math.floor(Math.random()
    * cookingAreas.length);
    if (!accesi.includes(cookingAreas[randomIndex])) {
        accesi.push(cookingAreas[randomIndex]);
    }
}
```

Once the number of active burners is determined, a "while" loop is employed to randomly select the IDs of the specific burners that will be turned on. For each active burner, a random cooking time is generated, depending on two phases:

1. Initial phase (Boiling/High Heat): during the first 20% of the cooking time, the burner operates at high power, simulating tasks such as boiling or bringing food up to temperature. In this phase, the burner consumes power at a rate between 80% and 100% of max power.
2. Simmering Phase (Low Heat): for the remaining 80% of the cooking time, the burner simulates slow cooking or simmering, consuming power between 50% and 100% of min power. This phase represent the majority of the cooking session, where lower energy is required to maintain temperature.

```
let totalEnergyConsumptionKwh = 0;
//consumption simulation for active burner/s
accesi.forEach(function (area) {
    //radom usage time for each burners (0-2 hours)
    var usageTimeHours = Math.random() * 2;

    //splitting the usage time into two phases
    // 20% of time in initial phase
    var initialPhaseTime = usageTimeHours * 0.2;
    // 80% of time slow fire
    var slowCookingPhaseTime = usageTimeHours * 0.8;

    //energy consumption in the initial phase
    //(80%-100% of the max power)
    var initialPhasePower = Math.random() * (maxPowerKw * 0.2)
    + (maxPowerKw * 0.8);
    var initialPhaseConsumption = initialPhasePower
    * initialPhaseTime;

    //energy consumption in the slow fire phase
    //(50%-100% of the min power)
    var slowCookingPower = Math.random() * (minPowerKw * 0.5)
    + (minPowerKw * 0.5);
    var slowCookingConsumption = slowCookingPower
    * slowCookingPhaseTime;

    //sum consumption of two phases
    var energyConsumptionKwh = initialPhaseConsumption
    + slowCookingConsumption;
```

```
        //aggregate the results
        totalEnergyConsumptionKwh += energyConsumptionKwh;
    });

    msg.value = totalEnergyConsumptionKwh.toFixed(2);
```

As the code snippet shows, the energy consumption for each burned is calculated individually, and the total stove consumption is then built by summing up the consumption of all active burners. In this way, user are able to see the instant energy consumed by the device, since it's published on the following MQTT topic:

```
    //MQTT TOPIC

    //Value
    `smart_home/${home}/${room}/consumption/stove`;
```

Oven

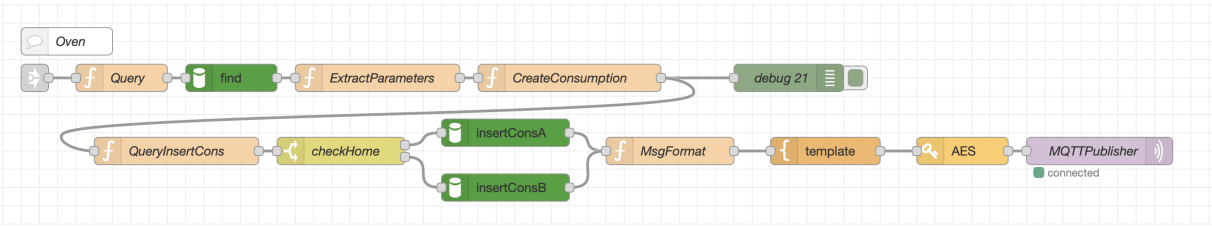


Figure 5.23: Oven Consumption Simulator Flow

As presented in Fig.5.23 the necessary information are fetching from factory settings (3.1.5), including:

- **Base Power**;
- **Oven Mode**;
- **Cooking Temperature**

In the *'extractParameters'* node, the oven mode and cooking temperature are randomly selected from predefined lists. This approach simulates various combinations of oven settings, allowing for the modeling of diverse operating scenarios.

```
//random index based on the modes array length
let randomIndex = Math.floor(Math.random() * modes.length);
//random index based on the temperatures array length
let randomIndexT = Math.floor(Math.random() * temperatures.length);

//random set a mode
msg.mode = modes[randomIndex];
//random set a temperature
msg.temperature = temperatures[randomIndexT];

return msg;
```

The oven's energy consumption in then calculated based on the selected mode, temperature, and the randomly generated usage time. Known consumption factors for each mode and temperature setting are used in this calculation:

```
const minTime = 1;
const maxTime = 120;

//consumption factors based on the mode
const modesCons = {
    "standard": 1,
    "grill": 1.2,
    "ventilated": 1.1,
};

//consumption factors based on the temperature
const temperatureCons = [
    { max: 100, factor: 1.0 },
    { max: 150, factor: 1.1 },
    { max: 200, factor: 1.2 },
    { max: 250, factor: 1.3 },
    { max: 300, factor: 1.4 }
];

//consumption factor based on the selected mode
let modesFactor = modesCons[mode];

//consumption factor based on the selected temperature
let tempFactor;
for (const interval of temperatureCons) {
    if (temperature <= interval.max) {
        tempFactor = interval.factor;
        break;
    }}
```

```
//random usage time (from 1 to 120 mins)
let time = Math.floor(Math.random() * (maxTime - minTime + 1))
+ minTime;
time = time/60;
time = parseFloat(time.toFixed(2))
```

For instance, if the oven is set to "standard" mode at 180°, the consumption might be lower than the one set to "grill" mode at 220°. In this way, the precise and accurate total consumption is calculated, stored in *'msg.value'* variable and published via MQTT on a specific topic.

```
//energy consumption in kWh
let totalConsumption = basePower * time * modesFactor * tempFactor;

msg.value = totalConsumption;

//MQTT TOPIC

//Value
`smart_home/${home}/${room}/consumption/oven`;
```

Water

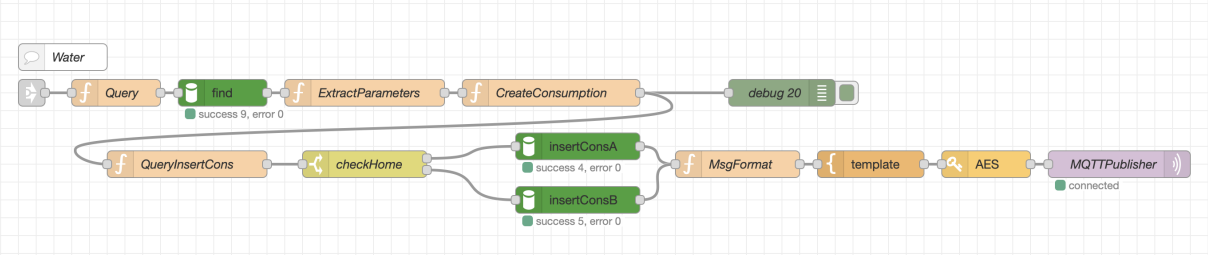


Figure 5.24: Water Consumption Simulator Flow

At the start of the random consumption generation illustrated in Fig.5.24 the flow accesses vital data from the parameters presets archived in the database (3.1.5):

- **Minimum consumption**;
- **Maximum consumption**;
- **Maximum duration**.

These are crucial for creating a different and realistic water consumption profile. They allow to model various household water usage scenarios, from brief hand-washing sessions to longer shower times.

To calculate the consumption, the *'createConsumption'* function employees a random number generator within the bounds set by these parameters: select a random consumption, generates a random duration between 1-*'maxDuration'* and multiplies they to determine the total water used for that event, as the code below resume:

```
let minCons = msg.minCons;
let maxCons = msg.maxCons;
let maxDuration = msg.maxDuration;

//radom consumption between [min-max]
let consLMinutes = Math.random() * (maxCons - minCons)
+ minCons;
consLMinutes = parseFloat(consLMinutes).toFixed(2);
//random time usage until maxDuration
let time = Math.random() * maxDuration;

//water consumption in litres
let consumoTotale = consLMinutes * time;

msg.value = consumoTotale;
```

The final value contained in *msg.value* represents the water consumption in liters for the current session, published for real time monitoring on the MQTT topic.

```
//MQTT TOPIC

//Value
`smart_home/${home}/${room}/consumption/water`;
```

5.8.3 Debug Information

For all the flows related to the consumption simulation described so far, a *'debug'* node is included to verify the information generated in the Node-RED environment. For example, in the case of the stove consumption, described in Sec.5.8.2 (the same logic applies to all other simulations), the *'msg.payload'* contains the recently generated data regarding the stove's energy consumption, as the code below shows:

```
//save data for the current burners
fornelli.push({
    area: area,
    usageTime: usageTimeHours.toFixed(2) + " hours",
    consumption: energyConsumptionKwh.toFixed(2) + " kWh"
    });

var message = `The induction (idx = ${msg.idx}) with `
+ accesi.length + ` stoves turned on (ID: `
+ fornelli.map(f => f.area).join(`, `)
+ `) has consumed ` + msg.value + ` kWh.`;

//details
message += `Usage details: `;
fornelli.forEach(function (f) {
    message += `Cooker ID ` + f.area
    + `: turned on for ` + f.usageTime
    + `, consumption: ` + f.consumption + `. `;
});

msg.payload = message;
```

This data is displayed in the Node-RED console debug, as shown in Fig.5.25, providing a detailed log of the information being generated. This logging is essential for ensuring that the simulated data is accurate and for troubleshooting any issues that may arise during the simulation.

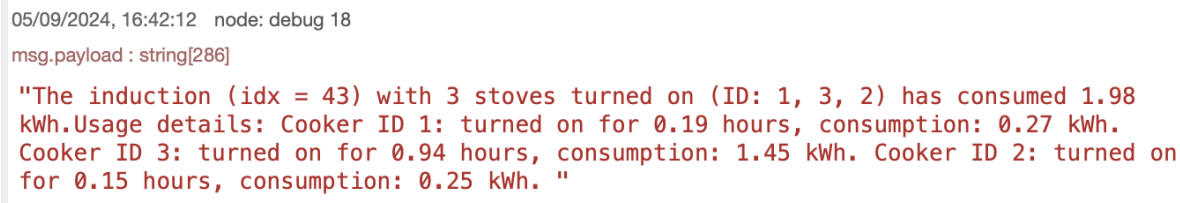


Figure 5.25: Consumption Debug Example

5.9 Meter History

Before introducing this section, it's essential to know that in the system, a dual approach to handling consumption data in employed, combining:

- **Real-time Updates**: instantaneous consumption data is published to an MQTT broker, ensuring that the information is updated in real-time. The MQTT protocol's lightweight nature and publish-subscribe model make it ideal for scenarios where immediate data dissemination is required.
- **Long-term Storage**: while real time is essential for immediate monitoring, it's also fundamental to maintain historical data for in-depth analysis. To this end, the generated instantaneous consumption data is stored in dedicated collections within MongoDB. This storage strategy serves a dual purpose:
  - Statistical Analysis: by preserving historical data, various analysis can be performed, including the calculation of daily and monthly consumption patterns.
  - Long-term Trends: The persistent storage of monthly consumption data is particularly valuable for identifying trends and conducting future analyses.

The combination of real-time MQTT publishing and persistent MongoDB storage provides a robust framework for both immediate monitoring and long-term analysis. The approach ensures that the system can meet the demands of real-time energy management while supporting in-depth, historical data analysis for informed decision-making and future planning.

Daily Consumption Monitoring

For each room in the respective smart home, daily consumption is calculated. The operation is identical for both smart home and their individual rooms. To illustrate how this process works, the example of calculating the daily consumption for the bathroom in Smart Home A is used (Fig.5.26)

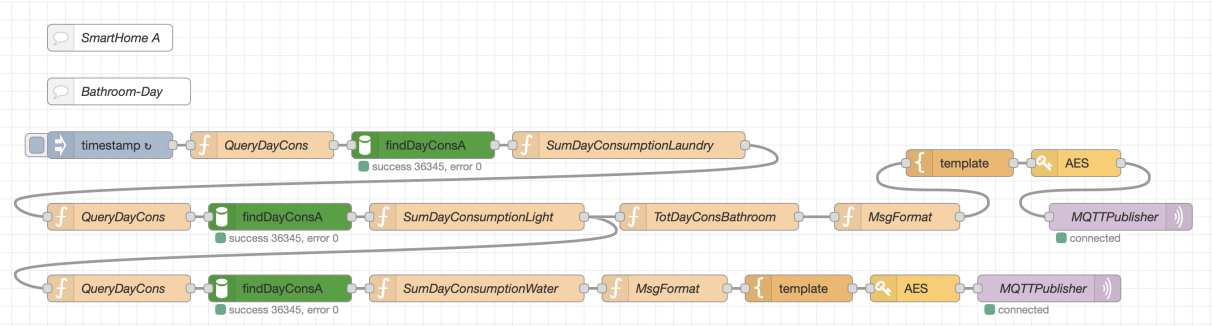


Figure 5.26: Bathroom Daily Consumption Flow

In the bathroom, both water and energy consumption are tracked. To calculate the daily energy consumption, it's essential to know the instantaneous consumption of devices such as the laundry machine and the light. The described flow is triggered every 5 seconds. This frequency ensures that the user receives real-time updates on daily consumption through MQTT, rather than waiting until the end of the day. To obtain daily consumption data, the system works with the respective collections in MongoDB where the instantaneous data is stored. In this case the *'ConsA'* collection. In *'QueryDayCons'* node function, the *'addedAt'* parameter is inspected, extracting today's date (starting at midnight), and tomorrow's date (midnight to limit the range to today).

```
//Today date
let startOfDay = new Date();
startOfDay.setHours(0, 0, 0, 0);
//Tomorrow date (midnight)
let endOfDay = new Date(startOfDay);
endOfDay.setDate(startOfDay.getDate() + 1);

//query
const query = {
    "item": "laundry_machine",
    "addedAt": {
        "$gte": startOfDay, "$lt": endOfDay
    }};


const options = {};
```

```
//payload for mongodb4 node
msg.payload = [query, options];

return msg;
```

After checking which documents fall within this range , the relevant information is extracted for each item and the system proceeds with individual daily consumption calculation, implemented in:

- *'SumDayConsumptionLaundry'*(in the code snippet below):
- *'SumDayConsumptionLight'*;
- *'SumDayConsumptionWater'*

```
function sumConsumption(msg) {
    if (!Array.isArray(msg.payload)) {
    msg.dayConsLaundryA = 0;
    return msg;
    }

    msg.dayConsLaundryA = msg.payload.reduce((total, record) => {
        return total + (record.value || 0);
    }, 0);

    msg.dayConsLaundryA = Math.round(msg.dayConsLaundryA * 100) / 100;

    //record found (to check data)
    msg.recordCountLaundryAD = msg.payload.length;

    return msg;
}

return sumConsumption(msg);
```

Finally the *'TotDayConsBathroom'* node sums up the obtained values to derive the daily consumption at the precise moment in time. This value is encrypted using the AES algorithm and published on a specific MQTT topic:

```
//MQTT TOPIC

//value
msg.topic = `smart_home/A/bathroom/consumption/history/day`;
```

Monthly Consumption Monitoring

Monthly consumption is computed for each room within the respective smart home. The flow for calculation, represented in Fig.5.27, is analogous to the one described for daily consumption (for simplicity and consistency, the monthly calculation of the bathroom consumption in smart home A is used as an example), with a few key differences:

- **Activation Timing**;
- **Monthly Data Extraction**;
- **Persistent Data Storage**

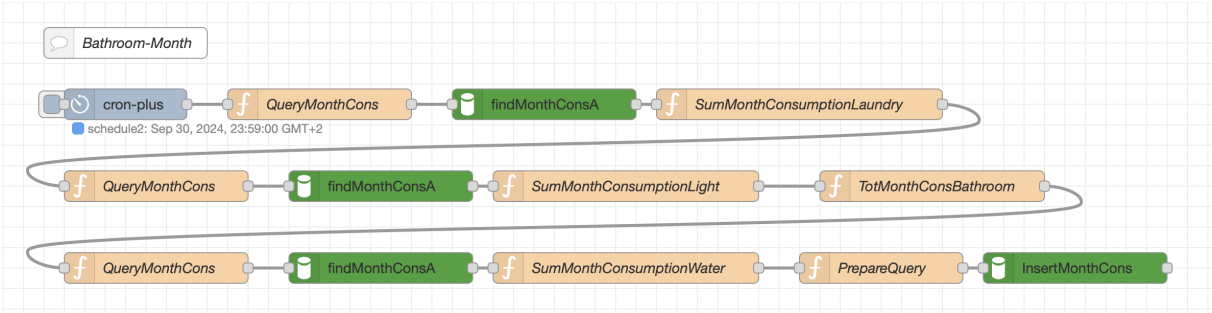


Figure 5.27: Bathroom Monthly Consumption Flow

The flow is activated using the *'cron-plus'* node from the *'node-red-contrib-cron-plus'* palette. This node allows for flexible scheduling of tasks, supporting cron expression that enable users to execute recurring operations at custom intervals.[2]. In this project, the cron expression **'59 23 L * *'** is used to schedule the flow to run at 23.59 on the last day of each month. This ensures that the monthly consumption is calculated at the end of every month.

To extract the relevant documents for the current month, the query written in *'QueryMonthCons'* node, works on the *'addedAt'* parameter. It looks for documents that fall within the range from the beginning of the current month (first day of the month) to the beginning of the next month, which determines the end of the current month.

```
//start of current month
let startOfMonth = new Date();
startOfMonth.setDate(1);
startOfMonth.setHours(0, 0, 0, 0);

//start of the next month
let endOfMonth = new Date(startOfMonth);
```

[2]https://flows.nodered.org

```
endOfMonth.setMonth(startOfMonth.getMonth() + 1);
endOfMonth.setHours(0, 0, 0, 0);

//query
const query = {
    "item": "laundry_machine",
    "addedAt": {
        "$gte": startOfMonth, "$lt": endOfMonth
    }
};

const options = {};

//payload for mongodb4 node
msg.payload = [query, options];

return msg;
```

The process for summing up the consumption for individual items and calculating the final monthly consumption is the same as for the daily consumption.

Unlike the real-time updates for daily consumption, the monthly values are saved exclusively in MongoDB, in the *MonthConsA* or *MonthConsB*, depending on the smart home. These values represent persistent data and are stored in a specific format (for further details please see Sec.3.1.7). This ensures that historical consumption data is retained for future analysis and reporting.

5.10 Login Dashboard

This section outlines the logic that manages the display of the smart home information dashboard in Node-RED based on the registered user. The system enforces specific policies that restrict access to certain data depending on the user type (user A, user B or admin). To access their reserved area, user must log in with their credential.

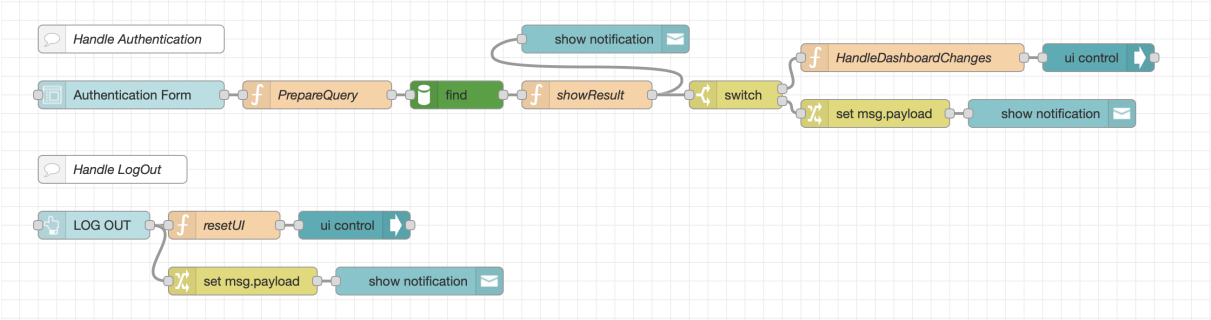


Figure 5.28: Login Handling Flow

Once a user attempts to log in, Node-RED (flow in Fig.5.28) verifies the username and password by querying the database, specifically the *'Users'* collection, which store all user information (3.1.1). If the credentials match an existing user (i.e. the payload returned by the query in not empty), access in granted. Otherwise, access is denied, as is shown in the code below from the *'showResults'* node:

```
    let count = global.get('userCount') || 0;

        //if there is no connected user
        if (count === 0) {
            msg.go = 'go';
            //check payload
            if (msg.payload && msg.payload.length > 0) {
                msg.username = msg.payload[0].username;
                let username = msg.username;
                msg.payload = "Access Permitted for user "
                + username;
            }
        else {
            msg.payload = "Access Denied: check credentials";
        }
    }
    else {msg.go = 'not go';}

return msg;
```

Depending on the connected user, only the dashboards that are authorized for that user, according to the implemented security policies, will be displayed as written in the *'HandleDashboardChanges'* function node. This ensures that each user only sees the data they are permitted to access.

```
let userConnected = msg.username;
let count = global.get('userCount') || 0;

if (userConnected == "a") {
    msg.payload = {
        "tabs": {
            "show": [
                "Dashboard Home A",
                "Home A Consumption",
                "Home A History",
                "Login"
            ]
        }
    };
count++;
}

else if (userConnected == "b") {
    msg.payload = {
        "tabs": {
            "show": [
                "Dashboard Home B",
                "Home B Consumption",
                "Home B History",
                "Login"
            ]
        }
    };
count++;
}
```

```
    else if (userConnected == "admin") {
        msg.payload = {
            "tabs": {
                "show": [
                    "Home",
                    "Dashboard Home A",
                    "Home A Consumption",
                    "Dashboard Home B",
                    "Home B Consumption",
                    "Login"
                ]
            }
        };
    count++;
    }

global.set('userCount', count);
return msg;
```

Additionally, to simulate a kind of concurrency control, a global variable *'userCount'* is used to track any active user connection (represented by a counter). This prevents multiple users from logging in simultaneously. If a user is already logged in, no other user can log in until the current user logs out, ensuring that only one session can be active at a time.

5.11 Dashboard

For the graphical development of the Node-RED interactive interface, the *'node-red-dashboard'* palette was used, with dedicated nodes for visualizing data in specific format[3]. The flows in this section are consistent across all rooms of the smart home, with each room having its own dedicated flow that monitors its specific environment and consumption data.

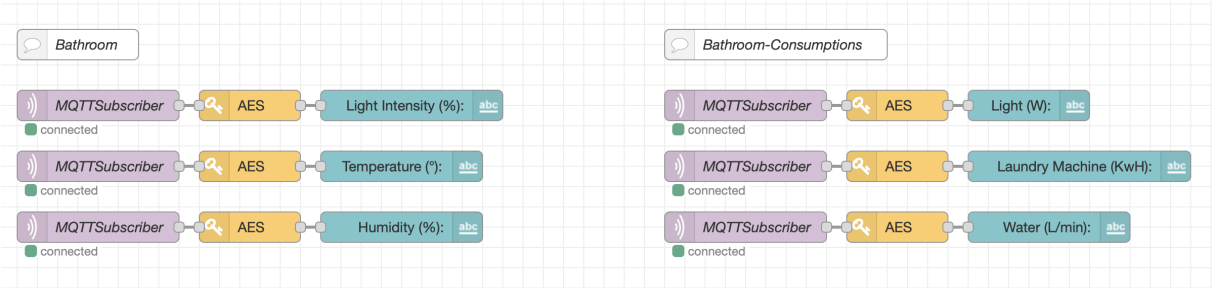


Figure 5.29: Dashboard Monitoring and Consumption Flow

As an example, the provided Fig.5.29 depicts a Node-RED flow which is used to display the bathroom dashboard information for Smart Home A. The left part of the flow is dedicated to graphically represent environmental conditions, such as temperature, humidity, and light intensity. For each metrics, the flow starts with an *MQTTSubscriber* node, which subscribes to the relevant MQTT topics, receiving data from sensors. The receiving data is then passed through an *'AES'* node, which is used to decrypt the information, before being sent to the dashboard, displayable in a label using the *'ui_text'* node.

The right part of the Fig.5.29 collect and show consumption-related data for devices in the bathroom. Like the left section, it starts with *'MQTTSubscriber'* node, which gather data from smart devices regarding consumption. After decrypt with the same algorithm, the information are displayed.

[3]https://flows.nodered.org

5.12 Dashboard History

For the visualization of the consumption history on Node-RED, the same approach is utilized, by employing different format node. In this section the flows are designed to retrieve summary information about:

- **Daily Consumption**;
- **Monthly Consumption**

In order to easy understand the functionalities, let's take the Bathroom scenario in Smart Home A as an example.

Daily Consumption

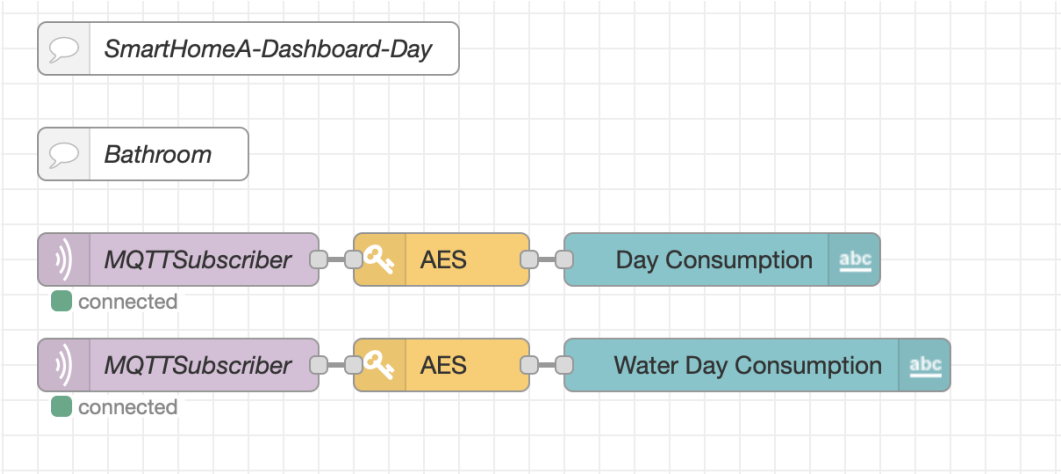


Figure 5.30: Dashboard Daily Consumption Flow

The Node-RED flows in Fig.5.30 are implemented to display different types of consumption during the current day:

- Energy Consumption.
- Water Consumption.

Both flow starts with an *'MQTTSubscriber'* node, which is used to connected and receives data from a specific device. This data then passes through an AES node, which decrypt the information, indicating a focus on data security, which is crucial for smart home applications. The final node is each flow (*'DayConsumption'* and *'WaterDayConsumption'*) are responsible for aggregating and presenting the consumption data in a format suitable for daily monitoring.

Monthly Consumption

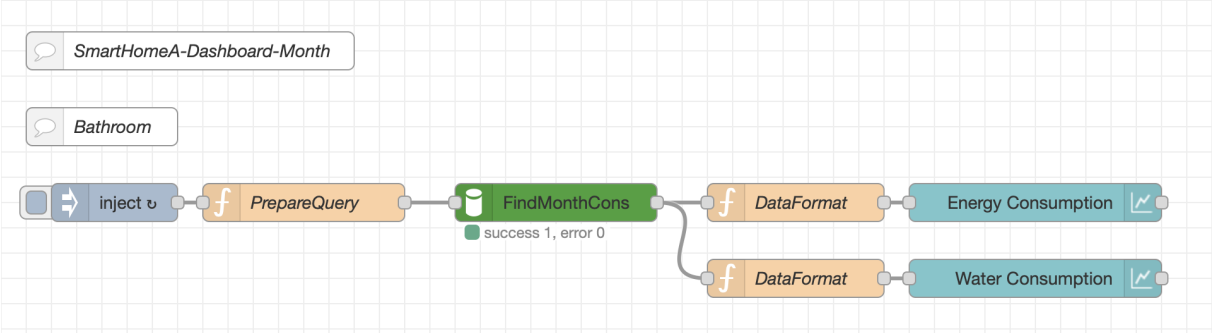


Figure 5.31: Dashboard Monthly Consumption Flow

The flow in Fig.5.31 is developed to retrieve stored consumption (energy and water) data from a database, process, and display it in a clear and user-friendly manner for detailed analysis of resource consumption over a specific period (month).

The *'prepareQuery'* node construct the appropriate query to fetch stored data on the specific collection (3.1.7), depending on the smart home under analysis, where the *'Room'* parameter is based on the room specific name.

```
const query = {
"Room": "Bathroom"
};

// payload for mongodb4 node
msg.payload = [query];

return msg;
```

After extracting the data, the flow splits into two parallel branches. The data passes through a *'DataFormat'* function node, which processes and structure the energy consumption data.

```
//db documents
let data = msg.payload;

//order data by month
data.sort((a, b) => a.Month.localeCompare(b.Month));

//prepare data for chart
let chartData = data.map(item =>
item.Consumption.EnergyConsumption);

//label for month (x-asix)
let labels = data.map(item => item.Month);

//data format for barchart
msg.payload = [{
    series: ["Energy Consumption"],
    data: chartData,
    labels: labels
}];

return msg;
```

The *msg.payload* contains an array of documents. Each document contains information about energy consumption for a specific month. Data are sorted based on the *'Month'* property. Then data are prepared for the bar chart: the codes creates a new array *'chartData'* that contains only the energy consumption value from each document. This array will be used as the data for the Y-axis of the bar chart. It also extracts the *'Month'* value from each document to be used as labels for the X-axis of the bar chart. Finally, the code formats the data into the structure expected by the *'ui_chart'* node. It creates a *'msg.payload'* object that defines the name of the data series, the array of energy consumption data and the array of month names. The function returns the *'msg'* object, which now contains the formatted data ready for use in the bar chart visualization. Similarly, water consumption data goes through its own *'DataFormat'* node for processing.

The final node in each branch, is responsible to represent consumption data with a bar chart using the *'ui_chart'* node.

Chapter 6

Directions For Future Work

One of the future improvements involves enhancing the **concurrent access** capabilities of the Node-RED dashboard. This will allow multiple users to access the system simultaneously without interference. Additionally, a web-based form can be implemented for better data visualization and interaction, making the user experience more fluid and accessible from any device.

Another key development is the integration of **physical sensors** for monitoring power consumption across various devices. Currently, device consumption is simulated in Node-RED, but by installing sensors, it will be possible to obtain real-time data on the actual consumption, leading to more accurate energy tracking and analysis.

Another potential development is the implementation of **real-time cost monitoring** for each room, based on the user's energy contract from the free energy market. This feature would calculate the actual cost of energy consumption, providing users with detailed insights into their energy expenses and promoting more efficient energy usage.

Finally, to improve security, an important enhancement would be the implementation of **password hashing** flow for storing the user credentials. This would provide better protection against unauthorized access, mimicking real-word practices for data security.