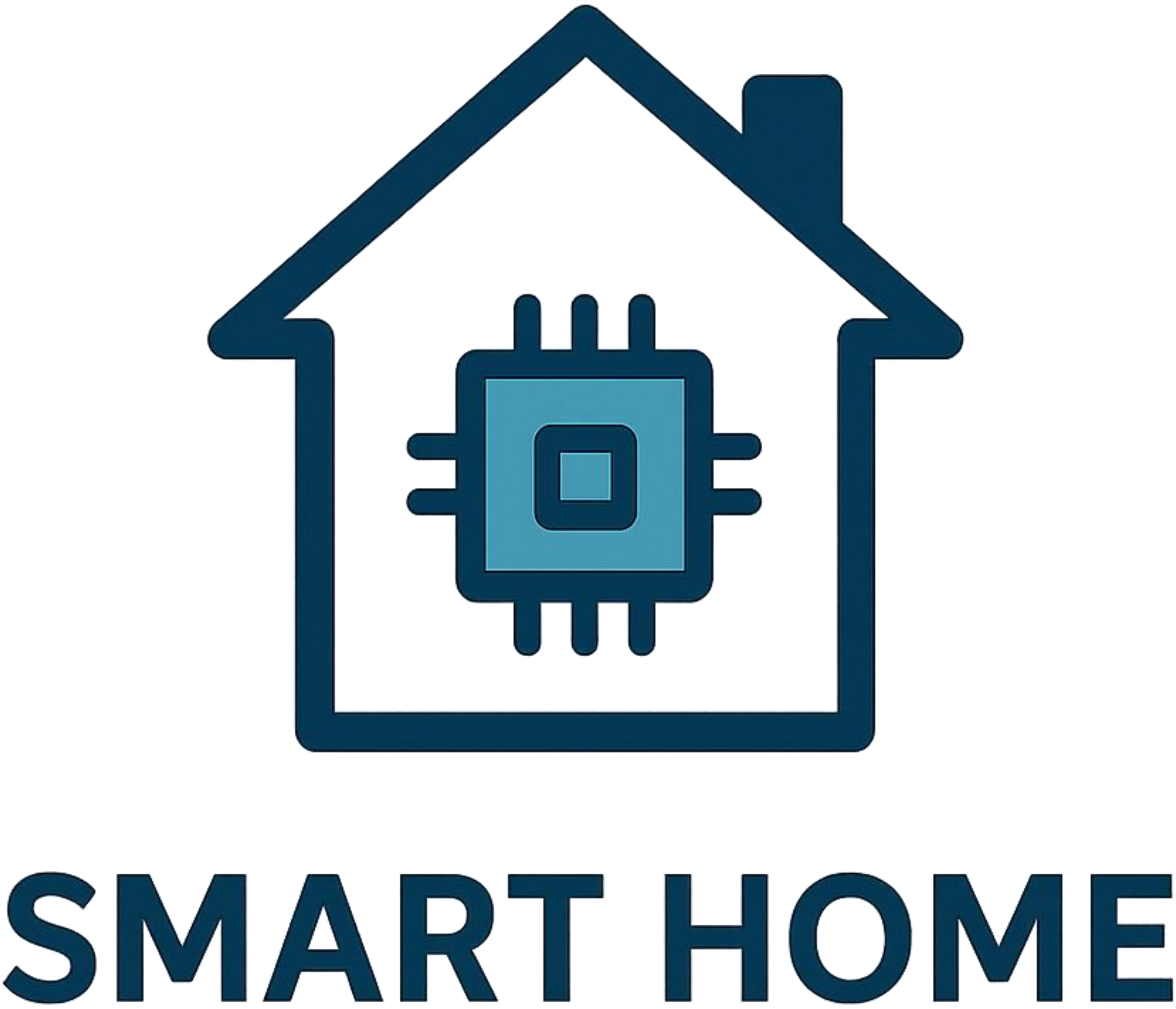
Smart home

Autonomous system project A.Y. 24/25

Dario D'Ercole	288643
Giovanni Spaziani	295397



INDEX

INTRODUCTION

The Smart Home project is a case study that explores the software architecture for adaptable systems in smart home environments, using the MAPE-K paradigm. The goal of the system is to manage various smart devices and sensors, such as the thermostat, roller shutters and dish washer, optimizing energy consumption and improving home comfort.

The MAPE-K architecture is based on five main components: Monitor, Analyze, Plan, Execute and Knowledge, which work together to monitor the state of the environment, analyze the collected data, plan corrective actions and implement strategies to optimize system performance.

During the project, various smart devices will be introduced and managed, including temperature and energy consumption sensors, actuators such as smart roller shutters and dish washer, and a smart refrigerator that helps optimize energy efficiency thanks to new integrated sensors. This project represents a concrete example of how autonomous systems can dynamically adapt to environmental conditions to improve efficiency and user comfort.

SENSORS

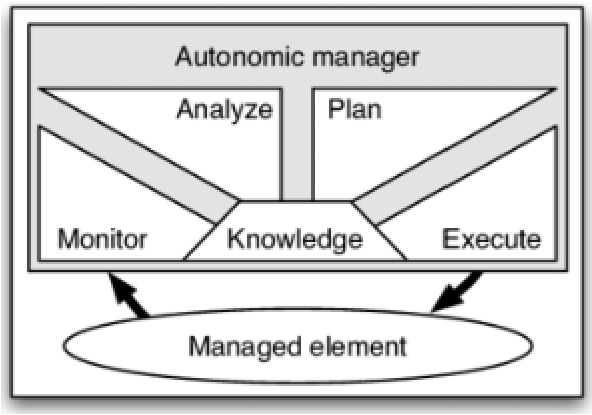
Temperature Sensor (Thermostat)	Detects the internal temperature of the house. Provides continuous data to the monitoring system to adjust heating or cooling optimally, ensuring a comfortable environment and efficient use of energy.
Light Sensor (Photodetector)	Measures the amount of light in your home. The sensor helps you adjust the opening of your blinds and the switching of your lights to make the most of natural light and reduce your energy costs.
Power Consumption Sensor (Energy Monitor)	Tracks the energy consumption of every device connected to the system. This sensor provides real-time data to help the system identify consumption spikes and optimize energy usage.
Smart Refrigerator Sensor	Collects information about the refrigerator's activity, such as internal temperature and the amount of food stored. This data helps optimize the refrigerator's operation to improve energy efficiency.

ACTUATORS

Smart Thermostat	It changes the target temperature for the house based on the data provided by the temperature sensor and the energy saving strategies planned by the system. It can lower or raise the heating in response to environmental changes and clock.
Smart Window	Controls the opening and closing of the roller shutters based on the light detected by the light sensor and the clock. This helps maintain a comfortable temperature inside, reducing the need for artificial heating.
Smart Dish Washer	Manages the wash cycle intelligently. It can start, stop or adjust the cycle based on energy-efficient programs, synchronizing with the electrical grid to reduce costs during periods of low demand.
Smart Refrigerator	It manages the operation of the refrigerator by regulating the internal temperature and optimizing energy efficiency based on sensor data. It can adjust the cooling cycle to keep food at the optimal temperature, reducing energy consumption.
Smart Light	Switches on/off light bulbs, in case of critical energy consumption level.

These sensors and actuators works together to monitor the home environment and adjust conditions in real time, optimizing comfort and energy efficiency.

MAPE-K IMPLEMENTATION



Monitor: collects data from the environment or system. This data can include information about resources, system status, or events that occur.

Analyze: evaluates the data collected by the "Monitor". In this phase, the collected data is processed and analyzed to detect anomalies, issues, or optimization opportunities.

Plan: generates strategies or plans to resolve identified problems or improve system performance.

Execute: deals with implementing the decided plan in the previous phase. Apply planned changes or actions to the system, such as activating resources, reconfiguring components, or changing operational parameters.

Knowledge: is the central repository that stores all the information used by the MAPE cycle. It includes any type of knowledge that helps make decisions. The other components consult and update this knowledge base during their operation.

The "Managed elements" in this case are:

1. Energy (including appliances).
2. Temperature.
3. Light.

The architecture built using Component Diagram is available in the appendix.

ADAPTATION METRICS FOR THE TEMPERATURE MANAGEMENT

If the internal temperature is below 18°C, these actions will be performed in parallel (see sequence diagram in the appendix). For the Thermostat:

1. The Thermostat executor asks the Broker for updates on the execution of the plan to be implemented.
2. The Broker then responds by sending the executor the communication of the plan to follow (in this case, the temperature must be raised).
3. The executor then acts on the Thermostat actuator to execute the plan.

For the Smart Window, instead:

1. The executor of the Smart Window asks the Broker for updates on the execution of the plan to be implemented.

- The Broker then responds by sending the executor the communication of the plan to follow (in this case, adjust the shutters or close them completely).

3. The executor then acts on the SmartWindow actuator to execute the plan.

ADAPTATION GOALS VS METRICS – TEMPERATURE MANAGEMENT

Adaptation goal	**Metrics used**	**Symptoms detected/optimizations**
Maintain optimal temperature	– Internal temperature (from sensors) – Energy consumption of the thermostat	– Temperature too low (<18°C) – Excessive heating consumption
Optimize energy savings	– Overall energy consumption – State of the shutters	– Inefficiencies in consumption related to heating – Shutters not properly closed to maintain heat (then close them to keep the room warm)
Avoid blackout by excessive consumption	– Overall energy consumption	– Energy value exceeds the critical threshold (3 kW, with warning at 2.5 kW)

ADAPTATION METRICS FOR THE CONFLICT MANAGEMENT

If the kW limit is exceeded, these actions will be performed in parallel (see sequence diagram in the appendix). For the Thermostat:

1. The Thermostat executor asks the Broker for updates on the execution of the plan to be implemented.
2. The Broker then responds by sending the executor the communication of the plan to follow (in this case, the temperature set on the thermostat must be lowered).
3. The executor then acts on the thermostat actuator to execute the plan.

For the Dish Washer, instead:

1. The executor of the Dish Washer asks the Broker for updates on the execution of the plan to be implemented.
2. The Broker then responds by sending the executor the communication of the plan to follow (in this case, temporarily suspend the wash or enable eco-mode).
3. The executor then acts the DishWasher actuator to execute the plan.

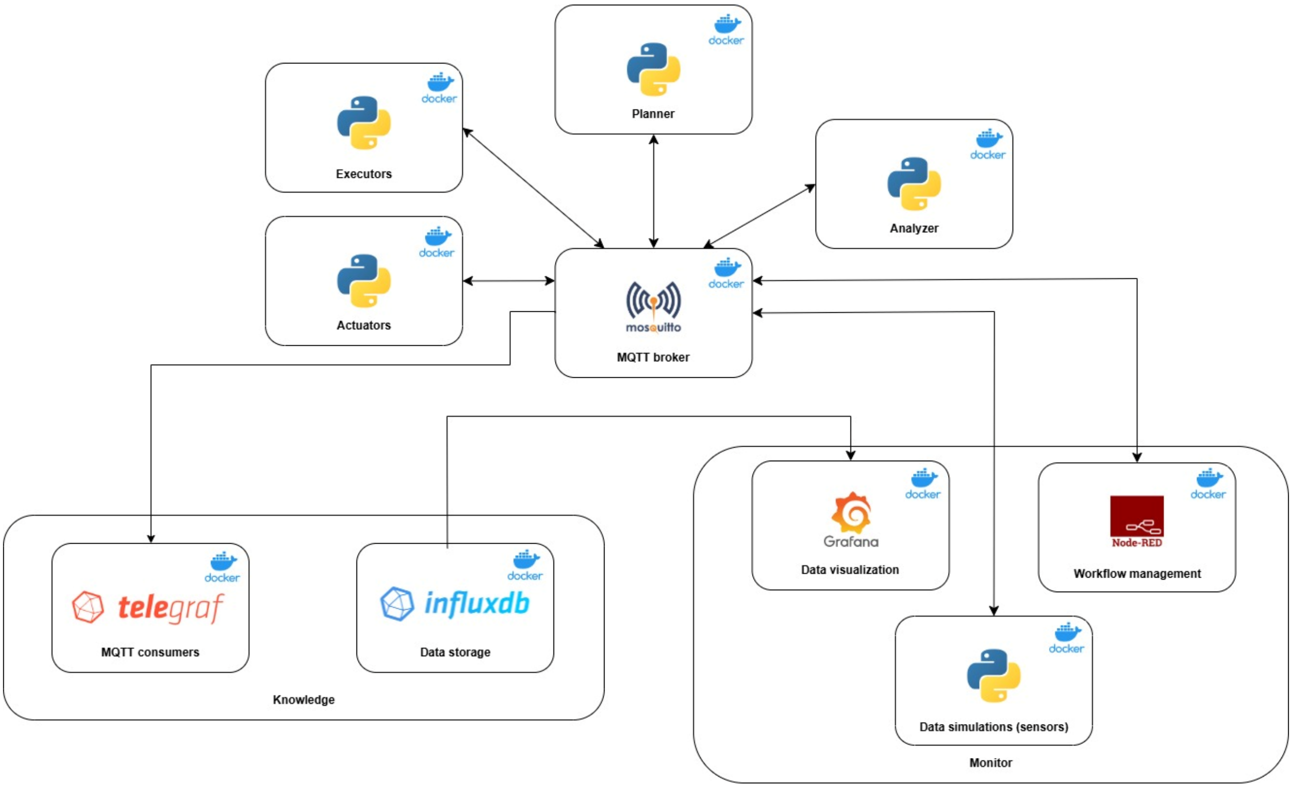
To conclude, in the case of the Smart Refrigerator:

1. The Smart Refrigerator executor asks the Broker for updates on the execution of the plan to be implemented.
2. The Broker then responds by sending the executor the communication of the plan to follow (for example, as said before, reduce the internal temperature of the refrigerator or suggest removing some foods to reduce energy consumption).
3. The executor then acts on the SmartRefrigerator actuator to execute the plan.

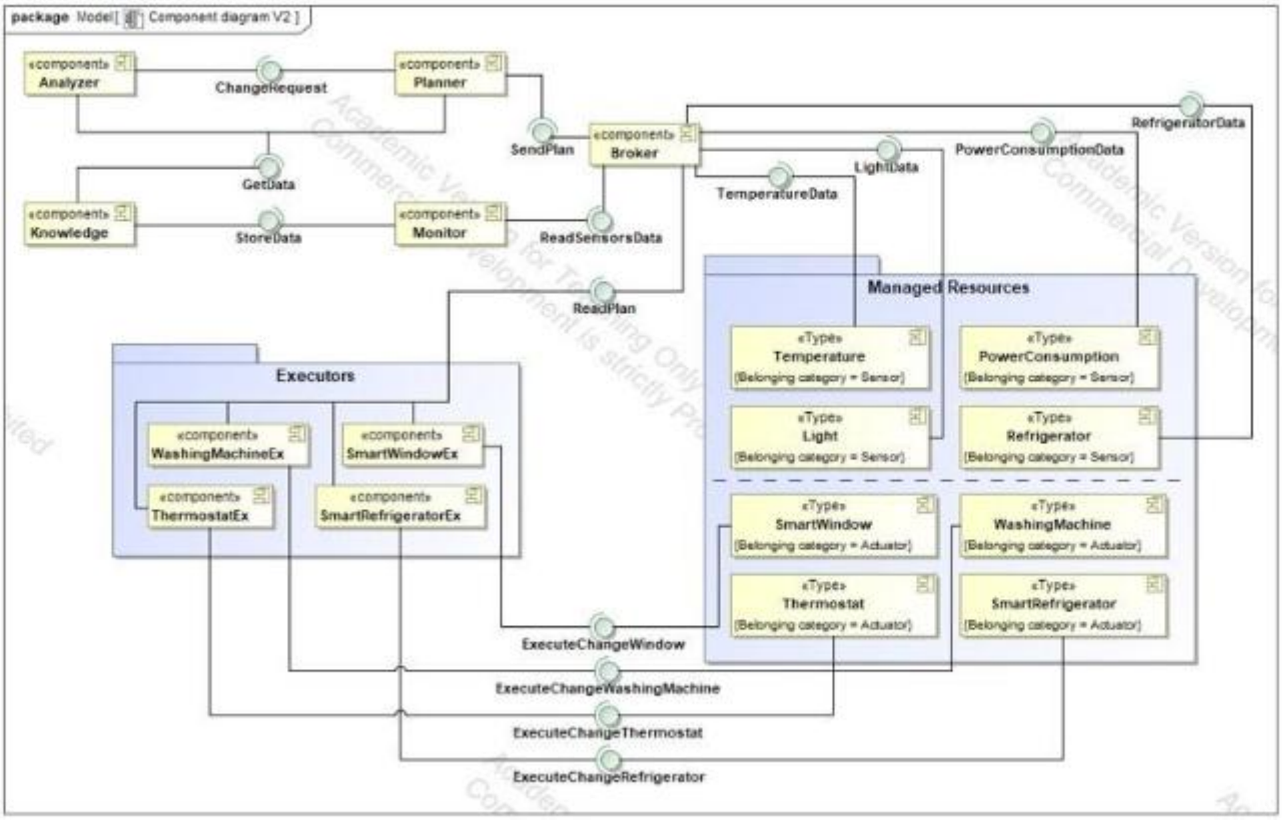
ADAPTATION GOALS VS METRICS – CONFLICT MANAGEMENT

Adaptation goal	Metrics used	Symptoms detected/optimizations
Management of energy conflicts	- Total energy consumption (from sensors) - Specific consumption of individual devices	- Exceeding the kW limit - High consumption by one or more devices
Optimization of the energy load	- Energy consumption of the dish washer - Refrigerator consumption	- Put in eco-mode the devices

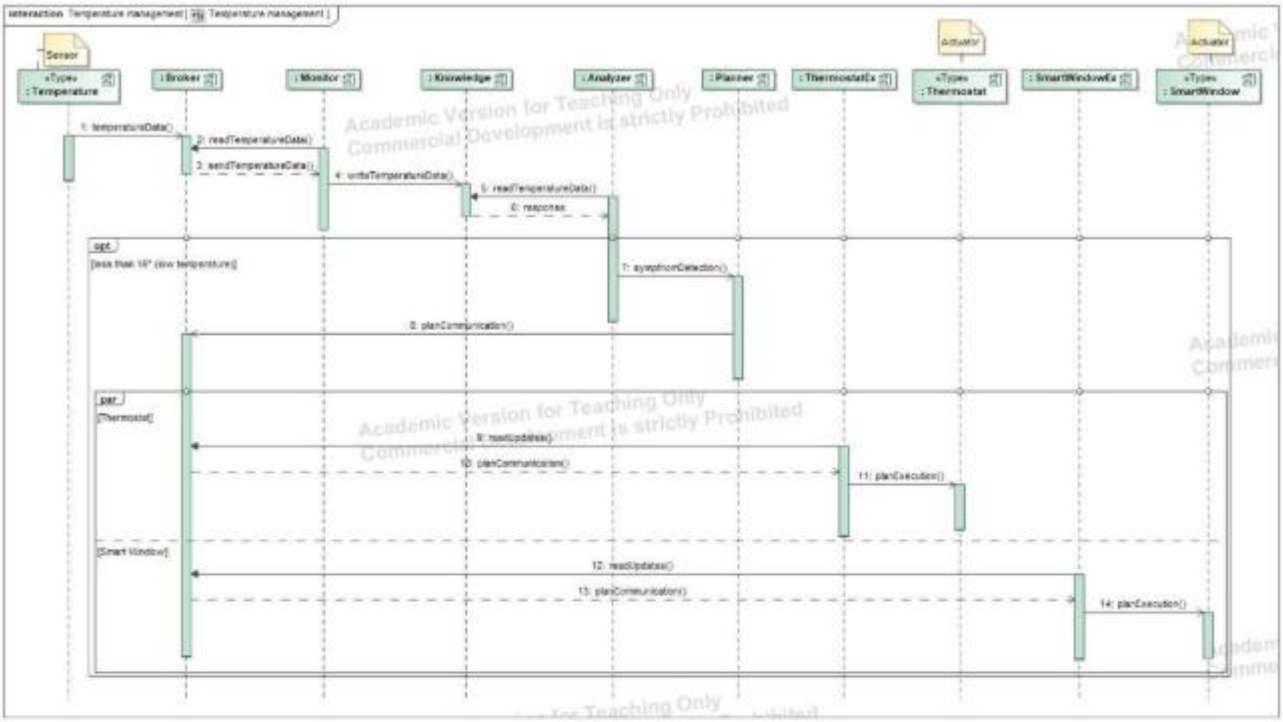
ARCHITECTURE DIAGRAM



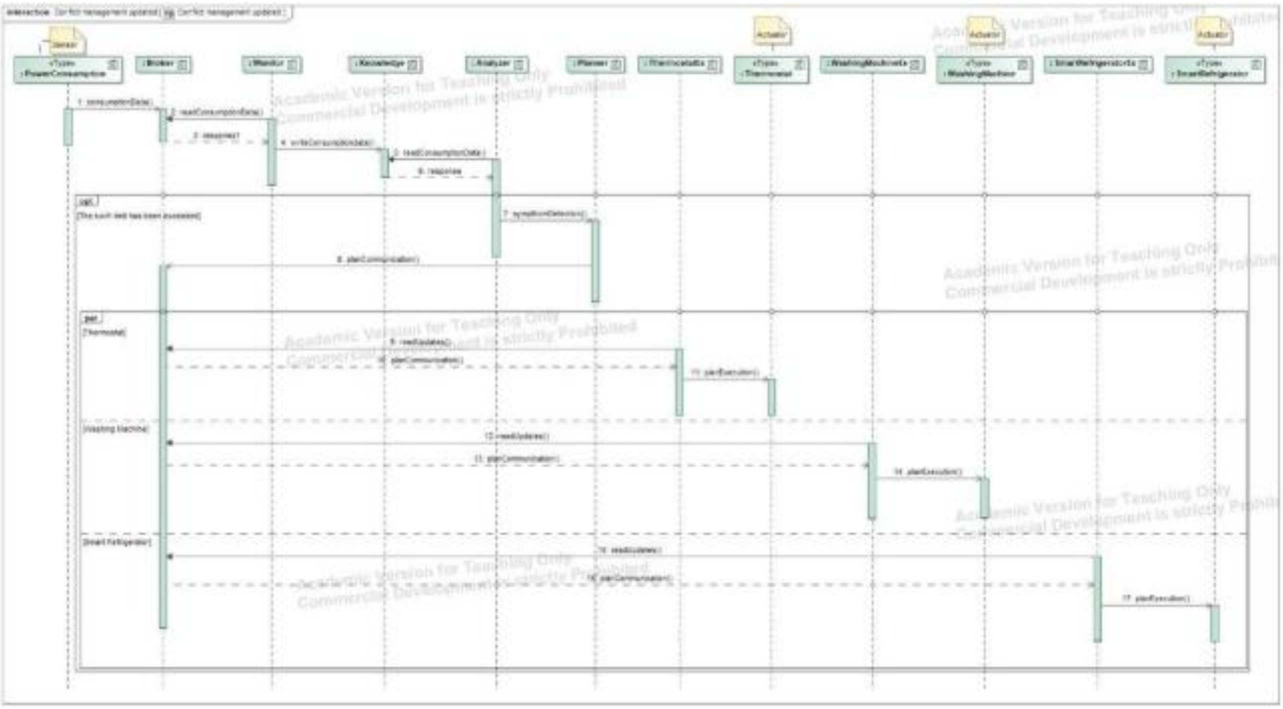
COMPONENT DIAGRAM



SEQUENCE DIAGRAM – TEMPERATURE MANAGEMENT



SEQUENCE DIAGRAM – CONFLICT MANAGEMENT



SIMULATING DATA

For the simulation, we employed a combination of python scripts and Node-RED workflows.

Node-RED publishes on the broker messages containing data regarding the simulated sensoring data and actuators, as well the initial environment state. The sensor and actuator simulation scripts subscribe to these messages in order to setup their local state before starting their loops:

```
def on_message(client, user_data, message):
    global sensors_info, state

    print(f'Received message: ', message.payload.decode('utf-8'))
    payload = extract_values_from_message(message)
    # setup initial sensors if it's initializer
    if JsonProperties.SENSORS_ROOT.value in payload:
        sensors_info = payload[JsonProperties.SENSORS_ROOT.value]
        init()
        if state:
            setup_state()
    if JsonProperties.STATE_ROOT.value in payload:
        state = payload[JsonProperties.STATE_ROOT.value]
        if sensors:
            setup_state()
```

```
def _on_message(self, client, user_data, message):
    print(f'({self.client_id}) Received message: ', message.payload.decode('utf-8'))
    values = extract_values_from_message(message)
    if JsonProperties.STATE_ROOT in values: #state message
        self.state = values[JsonProperties.STATE_ROOT]
    else:
        actuator_id = message.topic.split("/")[-1]
        if actuator_id == self.actuator_id:
            self.set_state(values[JsonProperties.SINGLE_VALUE])
    return values
```

The sensor simulation script works with a 3 parts loop:

1. Simulated data update (for temperature, light and smart fridge sensors.
2. Simulated energy consumption data update.
3. Data publication.

```python
def loop():  Lindbrum +1 *
    global last_publish_time, previous_sensors_update_time, previous_energy_reading_time
    while True:
        current_time = time.time()
        # Update sensors values
        if current_time - previous_sensors_update_time >= sensors_update_interval:
            update_sensors(sensors_update_interval)
            previous_sensors_update_time = current_time
        # Update energy readings
        if current_time - previous_energy_reading_time >= energy_reading_interval:
            calculate_kw()
            previous_energy_reading_time = current_time
        # Publish data
        if current_time - last_publish_time > publish_interval:
            publish_data()
            last_publish_time = current_time
        time.sleep(0.1)
```

When initializing the script, the sensor list is mapped to objects in the state so that when actuators modify the state, sensor values are updated accordingly:

```python
def setup_state():  2 usages  Lindbrum
    '''links sensors to state elements by name'''
    # parse sensors list to check if there are elements that are also in the state
    for sensor_type, sensors_list in sensors.items():
        for sensor_name, sensor_properties in sensors_list.items():
            for state_group, group_list in state.items():
                for state_name, state_properties in group_list.items():
                    if sensor_name == state_name:
                        sensor_properties['linked_state'] = state_properties
```

Data is published using the following topic schemas:

- For simple sensors:

  /SmartHomeD&G/sensor/<sensor_type>[/<room>]/<sensor_id>

  - <room> is omitted for data that is room independent, like the total house consumption.
- For smart appliances (e.g. fridge):
  /SmartHomeD&G/sensor/<sensor_type>/<room>/<appliance_id>/<metric>

DOCKERIZATION

In this part, the docker-compose file will be explained, where all the steps for dockerization of the various components used in our project have been performed. We dockerized:

1. Analyzer.
2. Planner.
3. Executor.
4. Mosquitto.
5. Sensors.
6. Actuators.

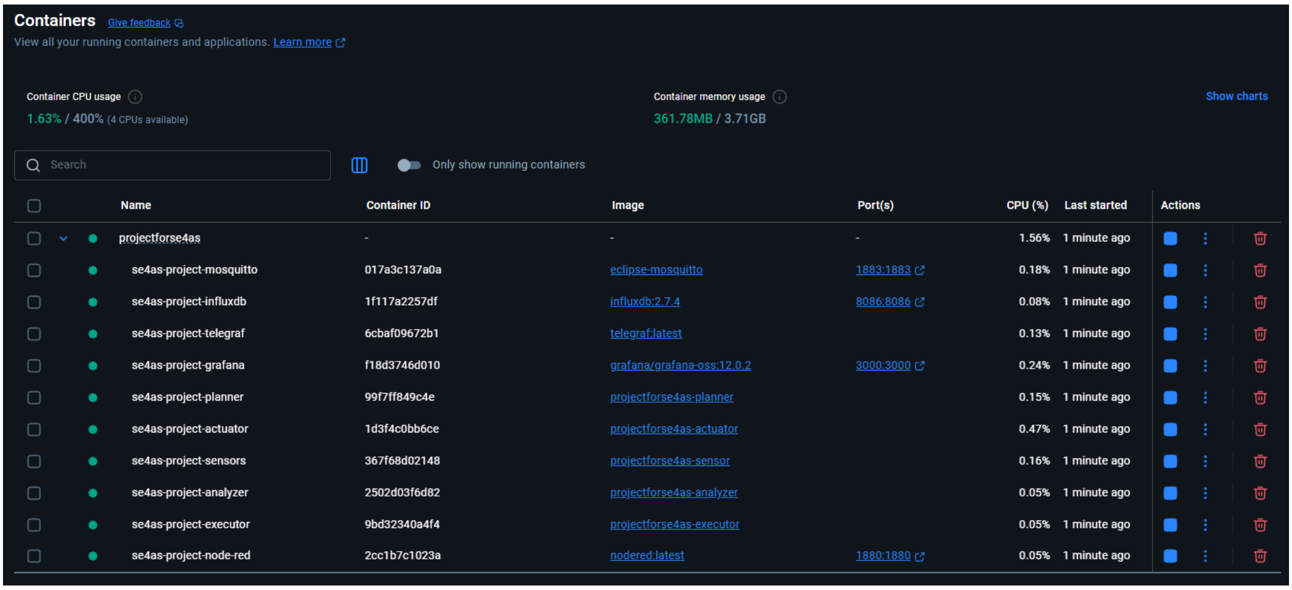
7. Node-RED.
8. Influx-DB.
9. Telegraf.
10. Grafana.

```yaml
node-red:
  container_name: se4as-project-node-red
  build:
    context: ./monitor/nodered/
    dockerfile: Dockerfile
  image: nodered:latest
  env_file:
    - .env
  ports:
    - "1880:1880"
  networks:
    se4as-project-network:
      ipv4_address: 172.30.0.100
  volumes:
    - $PWD/monitor/nodered/data:/data
  depends_on:
    - mosquitto

mosquitto:
  container_name: se4as-project-mosquitto
  image: eclipse-mosquitto
  volumes:
    - $PWD/broker/mosquitto/config:/mosquitto/config
    - $PWD/broker/mosquitto/log:/mosquitto/log
    - $PWD/broker/mosquitto/data/:/mosquitto/data
  restart: always
  networks:
    se4as-project-network:
      ipv4_address: 172.30.0.101
  ports:
    - "1883:1883"

executor:
  container_name: se4as-project-executor
  build:
    context: ./executors/
    dockerfile: Dockerfile
  depends_on:
    - mosquitto
  networks:
    se4as-project-network:
      ipv4_address: 172.30.0.108

planner:
  container_name: se4as-project-planner
  build:
    context: ./planner/
    dockerfile: Dockerfile
  depends_on:
    - mosquitto
  networks:
    se4as-project-network:
      ipv4_address: 172.30.0.109

analyzer:
  container_name: se4as-project-analyzer
  build:
    context: ./analyzer/
    dockerfile: Dockerfile
  depends_on:
    - mosquitto
  networks:
    se4as-project-network:
      ipv4_address: 172.30.0.110

grafana:
  restart: always
  image: grafana/grafana-oss:12.0.2
  container_name: se4as-project-grafana
  volumes:
    - $PWD/monitor/grafana/data:/var/lib/grafana:rw
  depends_on:
    - influxdb
  ports:
    - "3000:3000"
  env_file:
    - .env
  networks:
    se4as-project-network:
      ipv4_address: 172.30.0.105

sensor:
  container_name: se4as-project-sensors
  build:
    context: ./monitor/sensors/
    dockerfile: Dockerfile
  depends_on:
    - mosquitto
  networks:
    se4as-project-network:
      ipv4_address: 172.30.0.106

actuator:
  container_name: se4as-project-actuator
  build:
    context: ./executors/actuators/
    dockerfile: Dockerfile
  depends_on:
    - mosquitto
  networks:
    se4as-project-network:
      ipv4_address: 172.30.0.107

influxdb:
  container_name: se4as-project-influxdb
  image: influxdb:2.7.4
  env_file:
    - .env
  volumes:
  - $PWD/knowledge/influxdb/data:/var/lib/influxdb2:rw
  - $PWD/knowledge/influxdb/config:/etc/influxdb2:rw
  ports:
    - "8086:8086"
  networks:
    se4as-project-network:
      ipv4_address: 172.30.0.103

telegraf:
  restart: always
  container_name: se4as-project-telegraf
  image: telegraf:latest
  env_file:
    - .env
  volumes:
    - $PWD/knowledge/telegraf/mqtt_client.conf:/etc/telegraf/telegraf.conf:ro
  depends_on:
    - influxdb
  networks:
    se4as-project-network:
      ipv4_address: 172.30.0.104
networks:
  se4as-project-network:
    driver: bridge
    ipam:
      config:
        - subnet: 172.30.0.0/16
```

We first set all the container names, then we took the official images from Docker-Hub, which serve as a set of instructions to create a Docker container, as a template. We saved the configurations of the various containers with a folder inside the repository. We set up a network using a driven bridge, we set a 16-bit subnet mask to connect the containers. Subsequently we created a ".env" file to keep all the configurations of the various components under control. They can be seen in the following image.

```
#Influxdb configurations
DOCKER_INFLUXDB_INIT_MODE=setup
DOCKER_INFLUXDB_INIT_USERNAME=adminadmin
DOCKER_INFLUXDB_INIT_PASSWORD=adminadmin
DOCKER_INFLUXDB_INIT_ORG=Se4as-24-25
DOCKER_INFLUXDB_INIT_BUCKET=Se4as-project
DOCKER_INFLUXDB_ADMIN_TOKEN=cbgueyrjgtfcyngnyegti32525dyugw5eajukcnfgybc87cG9yY29kZGlvcG9yY2FjY2lhbWFkb25uYQxdfyugxfyuadsym==

#Telegraf configurations

TELEGRAF_PULL_INTERVAL=10s
TELEGRAF_HOSTNAME=se4as-app-container
TELEGRAF_INFLUX_URL=http://172.30.0.103:8086
INFLUX_BUCKET_TOKEN=0Hy53eOmomDC3jhgdLOoSxu1W7QPWWbvCYzuKSHjrryFENq50WCXuG5dHT9H7NuuG1_GSYTK9j3oOcQMqxcYVQ==
TELEGRAF_MOSQUITTO_SERVER=tcp://172.30.0.101:1883
TELEGRAF_MOSQUITTO_TOPICS='"/SmartHomeD&G/data/#"'
TELEGRAF_MOSQUITTO_TOPICS_INT='"/SmartHomeD&G/data/int/#"'
TELEGRAF_MOSQUITTO_TOPICS_FLOAT='"/SmartHomeD&G/data/float/#"'
TELEGRAF_DATA_FORMAT=value

#Grafana configurations
GRAFANA_READING_BUCKET_TOKEN=-Js8srkHc1vUjCy_DfLisyU-kZlhE5jtOKU-nWO6JpHUU0dJUVIznLNahr6NA27itdUaIAcBWxKWUEEzUzRoMw==
GF_FEATURE_TOGGLES_ENABLE=dashboardNewLayouts,kubernetesDashboards,dashboardScene

#Node-red configurations
TZ=Europe/Rome
```

We used Docker Desktop, a platform that provides an intuitive graphical interface (GUI) to manage containers, applications,and images directly on your computer. It is very intuitive to view all the running containers, the related images, and finally the related ports in localhost.



Containers Give feedback

View all your running containers and applications. Learn more

Container CPU usage 1.63% / 400% (4 CPUs available)

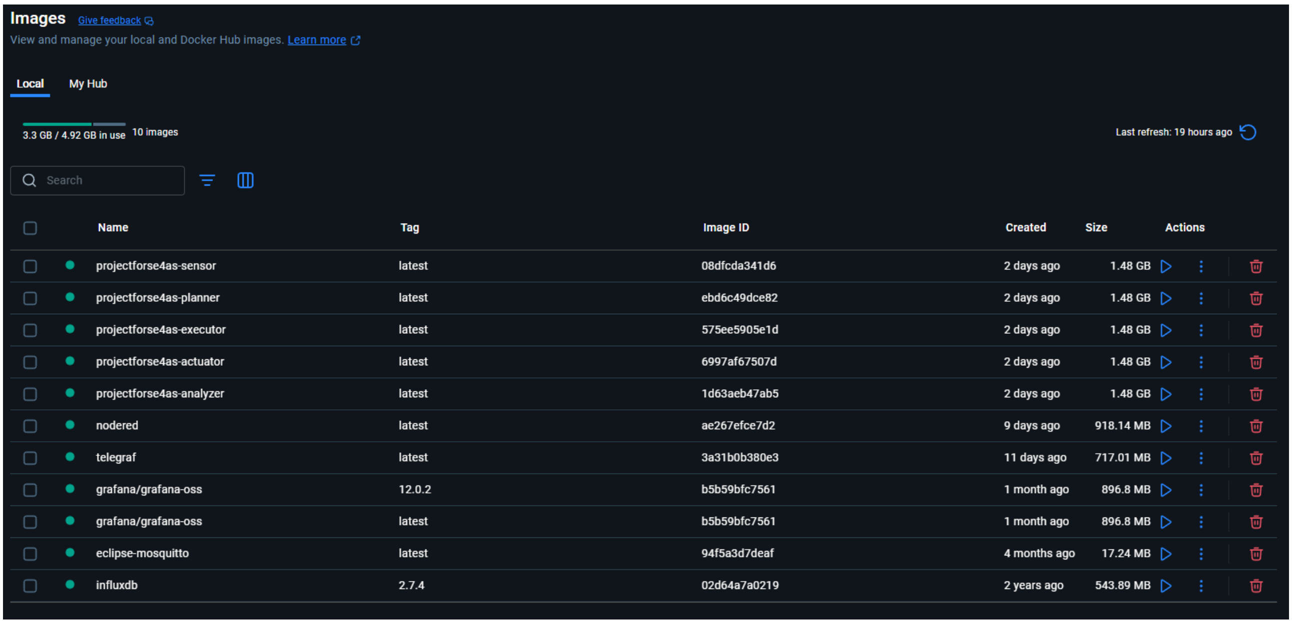
Container memory usage 361.78MB / 3.71GB

Show charts

Search

Only show running containers

Name	Container ID	Image	Port(s)	CPU (%)	Last started	Actions
projectforse4as	-	-	-	1.56%	1 minute ago	
se4as-project-mosquitto	017a3c137a0a	eclipse-mosquitto	1883:1883	0.18%	1 minute ago	
se4as-project-influxdb	1f117a2257df	influxdb:2.7.4	8086:8086	0.08%	1 minute ago	
se4as-project-telegraf	6cbaf09672b1	telegraf:latest		0.13%	1 minute ago	
se4as-project-grafana	f18d3746d010	grafana/grafana-oss:12.0.2	3000:3000	0.24%	1 minute ago	
se4as-project-planner	99f7ff849c4e	projectforse4as-planner		0.15%	1 minute ago	
se4as-project-actuator	1d3f4c0bb6ce	projectforse4as-actuator		0.47%	1 minute ago	
se4as-project-sensors	367f68d02148	projectforse4as-sensor		0.16%	1 minute ago	
se4as-project-analyzer	2502d03f6d82	projectforse4as-analyzer		0.05%	1 minute ago	
se4as-project-executor	9bd32340a4f4	projectforse4as-executor		0.05%	1 minute ago	
se4as-project-node-red	2cc1b7c1023a	nodered:latest	1880:1880	0.05%	1 minute ago	



Images Give feedback

View and manage your local and Docker Hub images. Learn more

Local My Hub

3.3 GB / 4.92 GB in use 10 images

Last refresh: 19 hours ago

Search

Name	Tag	Image ID	Created	Size	Actions
projectforse4as-sensor	latest	08dfcda341d6	2 days ago	1.48 GB	
projectforse4as-planner	latest	ebd6c49dce82	2 days ago	1.48 GB	
projectforse4as-executor	latest	575ee5905e1d	2 days ago	1.48 GB	
projectforse4as-actuator	latest	6997af67507d	2 days ago	1.48 GB	
projectforse4as-analyzer	latest	1d63aeb47ab5	2 days ago	1.48 GB	
nodered	latest	ae267efce7d2	9 days ago	918.14 MB	
telegraf	latest	3a31b0b380e3	11 days ago	717.01 MB	
grafana/grafana-oss	12.0.2	b5b59bfc7561	1 month ago	896.8 MB	
grafana/grafana-oss	latest	b5b59bfc7561	1 month ago	896.8 MB	
eclipse-mosquitto	latest	94f5a3d7deaf	4 months ago	17.24 MB	
influxdb	2.7.4	02d64a7a0219	2 years ago	543.89 MB	

NODE-RED

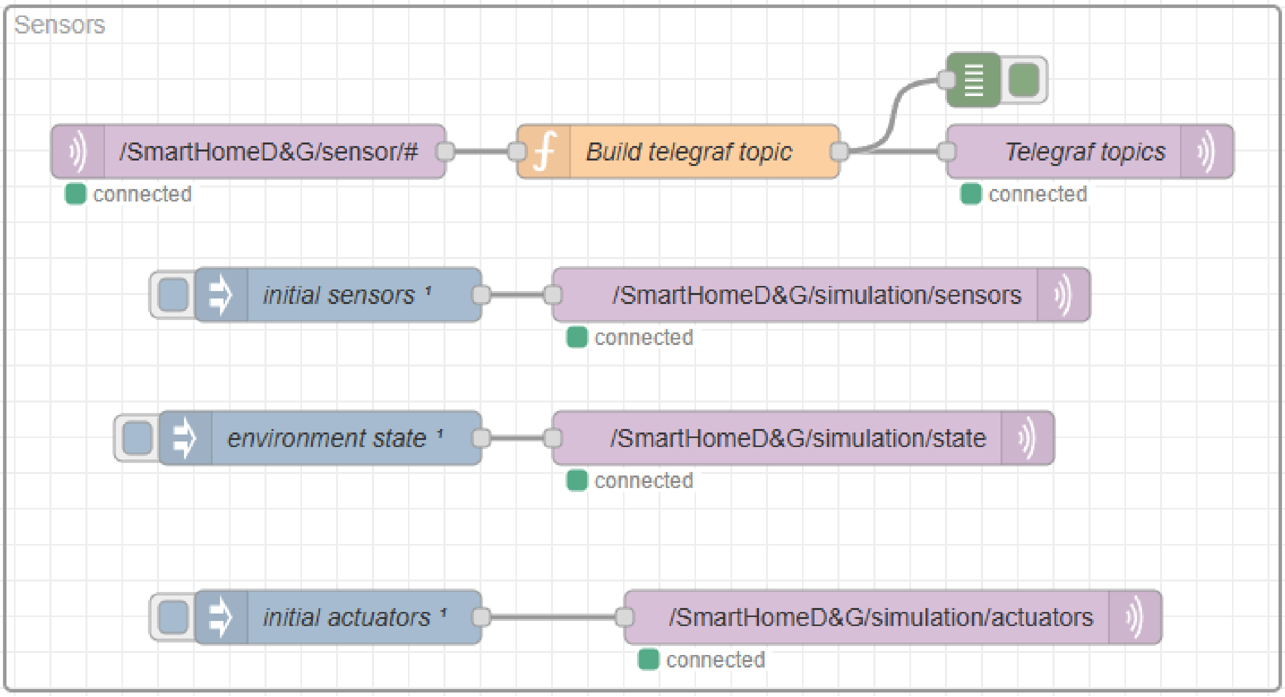
Our Node-RED flowchart can be split in three parts:

1. Sensors – for the processing of simulated sensoring data and setup of simulated states.
2. Analyzer Data and Configurations – for the processing of data fed to the analyzer and the configuration of analyzer and planner.
3. E-mail Report – for automatic notifications if the energy consumed threshold is exceeded.

SENSORS AND SIMULATION FLOWS

This is where the simulated sensors, environmental status, and actuators are managed. The received data is transformed into a format readable by Telegraf for monitoring.

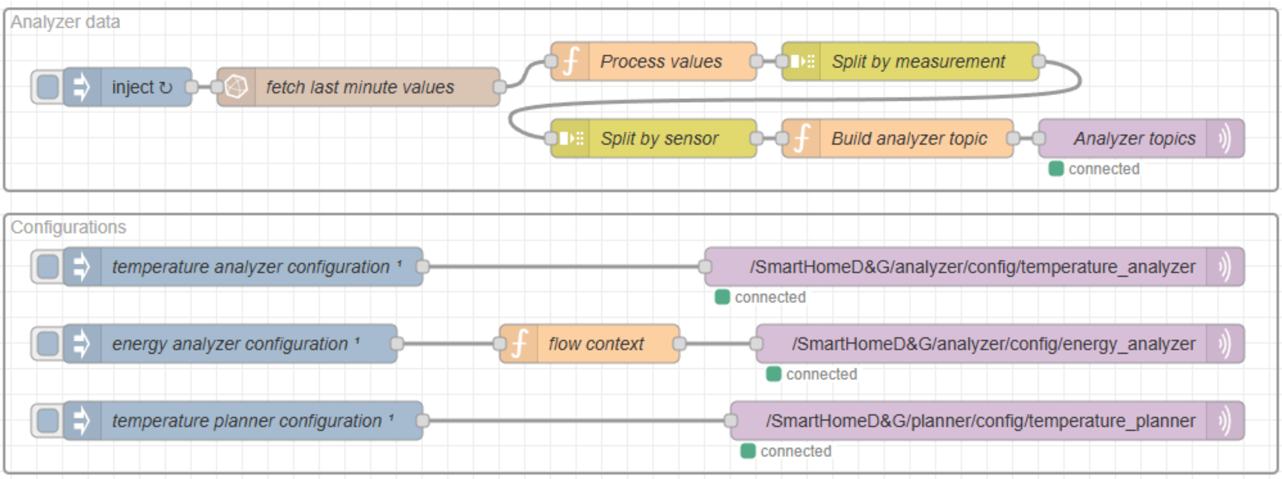
Three start nodes allow you to start the simulation with a predefined state for sensors, actuators and the environment.



ANALYZER DATA AND CONFIGURATIONS FLOWS

This part has two purposes:

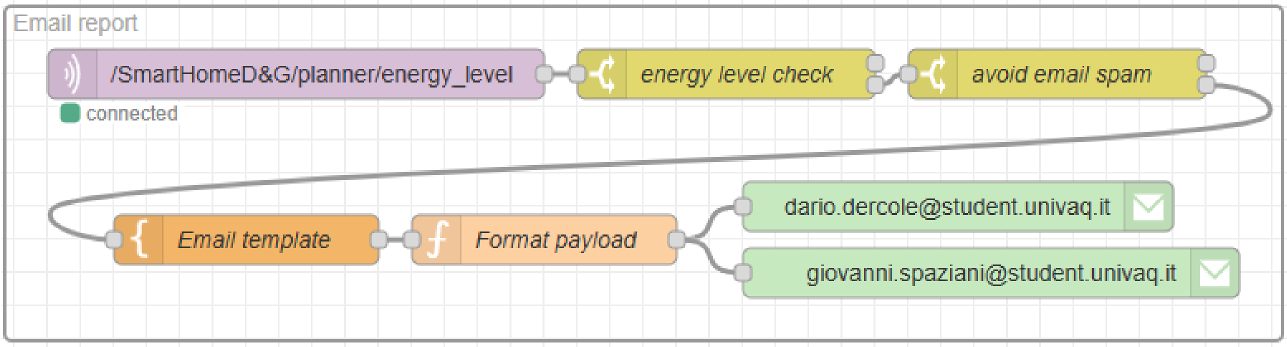
- Analyze the latest data from sensors: they are separated by measurement type and sensor, then published to MQTT.
- Send initial configurations to the analytics and planning modules (temperature and energy) via MQTT messages, so that the system starts up with the correct parameters.



EMAIL REPORT FLOW

This flow is activated when the energy level exceeds a threshold. The data comes from the MQTT topic /SmartHomeD&G/planner/energy_level, is monitored by a function, and if the value is too high, an email is generated.

To prevent spam, an intermediate block limits repeated sending. If everything is valid, the message is created and emailed to two recipients.

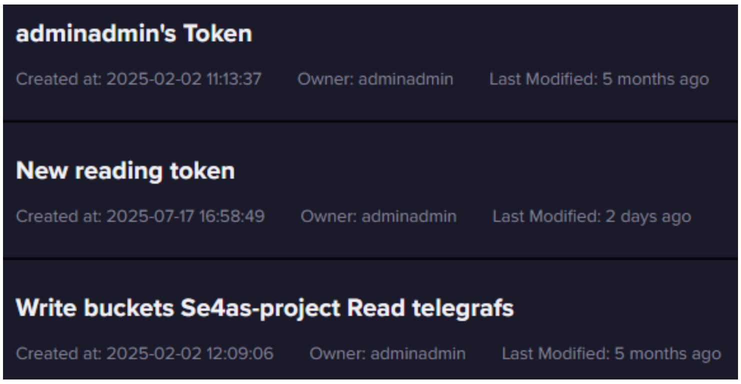


The following image represents a simulation of an email sent by the system.

INFLUX-DB

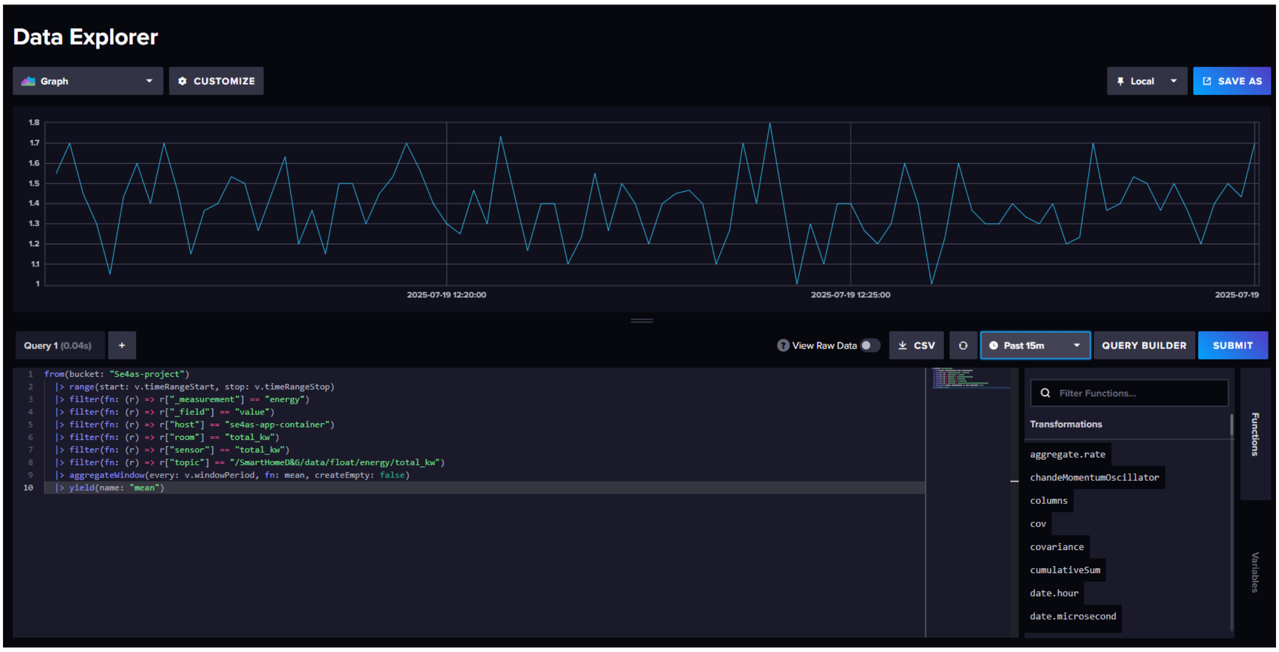
InfluxDB is an open-source time series database developed by the company InfluxData. It is used for storage and retrieval of time series data in fields such as operations monitoring, application metrics, sensor data and real-time analytics. In our project InfluxDB is deployed in a Docker container using the official docker image and environment variables have been set into the .env file to enable automatic configuration.

In addition to the default token (adminadmin), we generated two custom tokens to use in other applications that communicate with InfluxDB (see the following image):



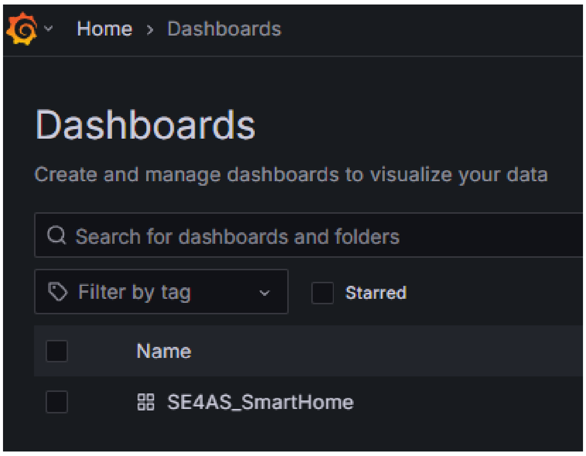
- The second token provides read-only access to the bucket and is provided to Grafana, as part of the InfluxDB data source configuration, and to Node-RED, to process the data for the analyzer.
- The third is a token that we provide to Telegraf, so that it can write into our project bucket when it consumes data from the MQTT broker.

Below is an example of the data that is being written in our bucket.

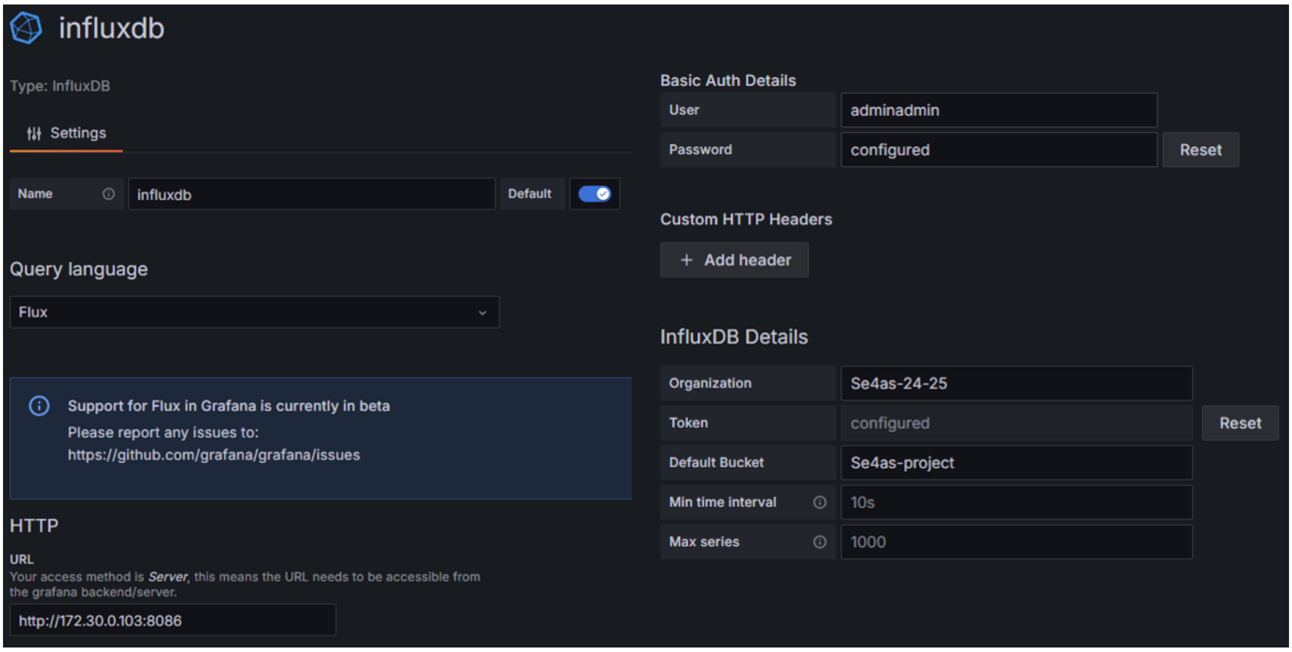


GRAFANA

Grafana is a web application for interactive data visualization and analysis. From the "Dashboards" section we select our personal Dashboard, called " SE4AS_SmartHome".

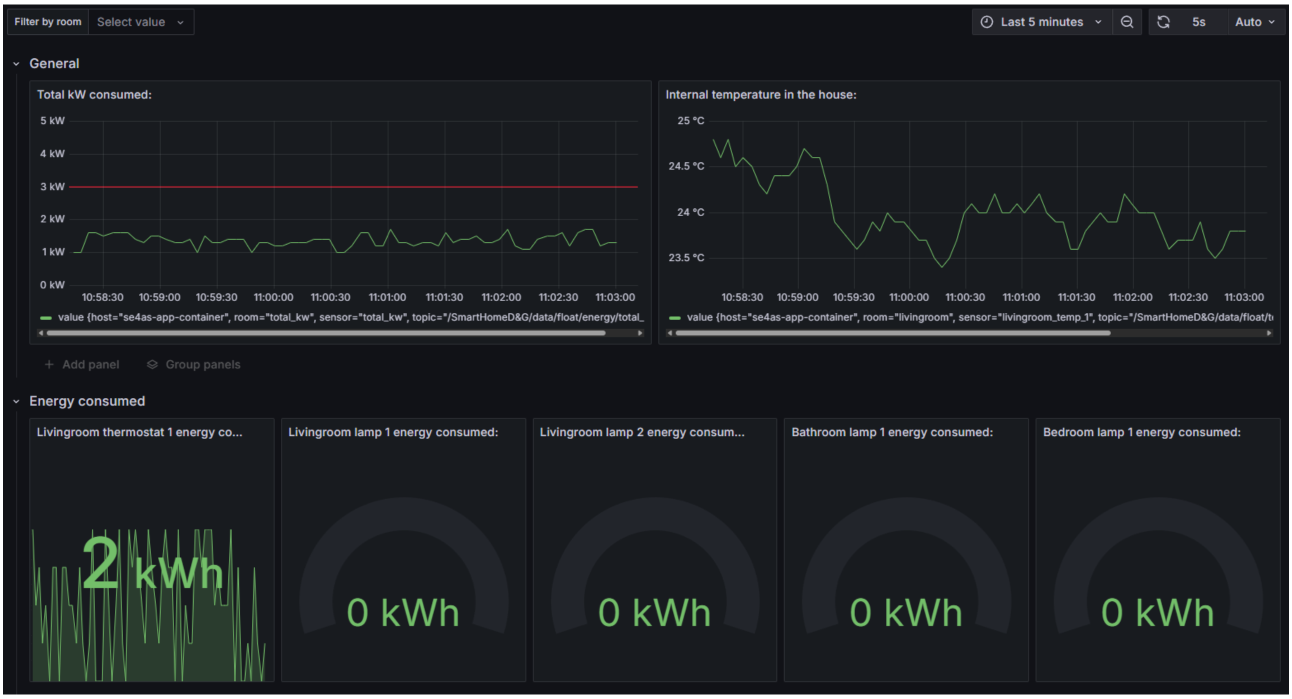


We setup communication to our InfluxDB bucket by using InfluxDB credentials and “Grafana token” API token.

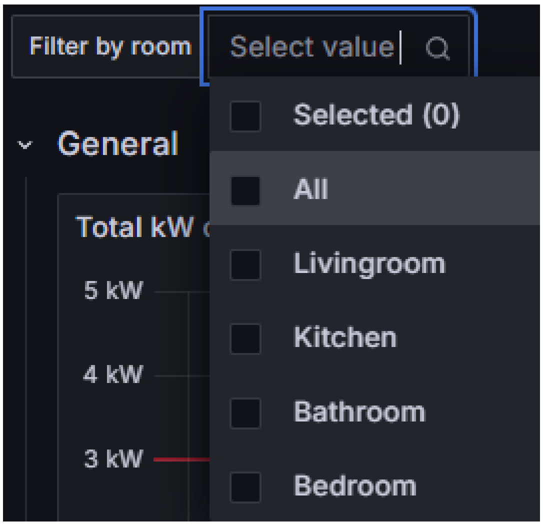


For each dashoard panel we used the quey generated by InfluxDB.

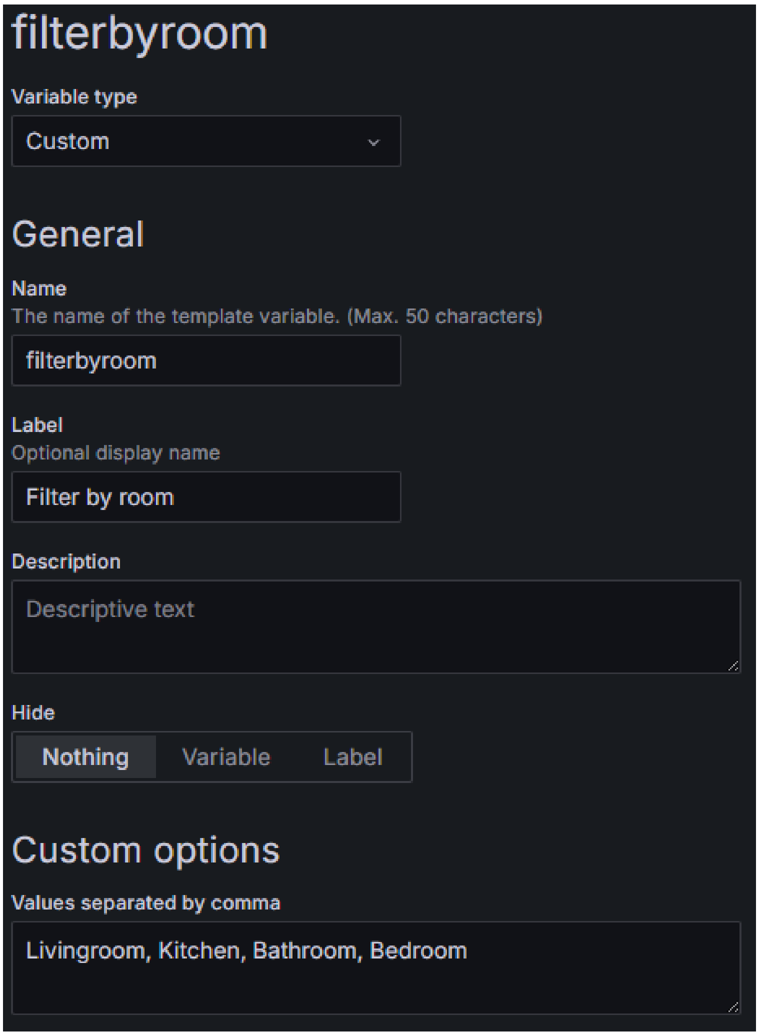
The resulting dashboard can be seen in the following image:



The dashboard allows filtering displayed data by room (eventually all of them). The following image displays the possible filter values, corresponding to the rooms present in our example.

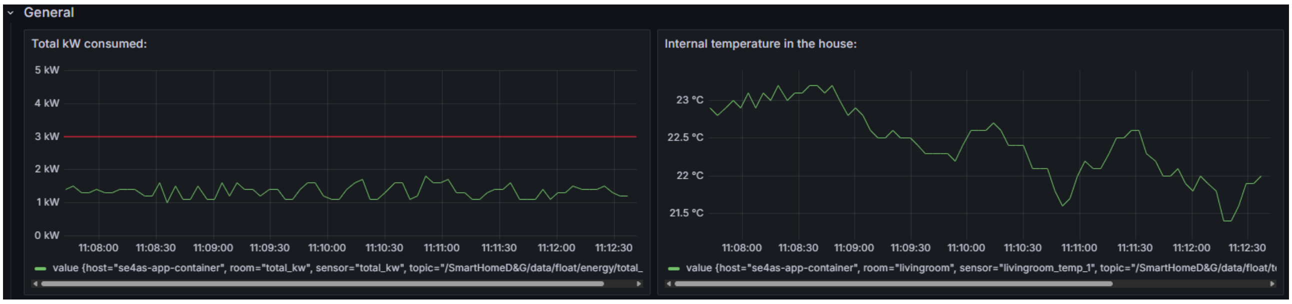


To do this, we created a custom variable from the dashboard settings and set up a value for each room:



Following is an example of the rules defined to hide/show panels:

These rules have been inserted for both individual panels and groups, so based on the selection made, the relevant graphs are shown. In the "General" panel, however, the measurements for the whole house will always be shown, regardless of the selected room, i.e. the Total kW consumed and the internal temperature of the house.



<u>**NOTE**: this feature is only available if you have Grafana version 12.0+, and since it is in BETA, you have to enable it by defining the following environment variable</u>:

```
GF_FEATURE_TOGGLES_ENABLE=dashboardNewLayouts,kubernetesDashboards,dashboardScene
```