POLITECNICO DI TORINO

Master's Degree in Computer Engineering

Master's Degree Thesis

A software architecture for database management in a real-time IoT monitoring service

Supervisors

Prof. Antonio SERVETTI

Dr. Pietro CHIAVASSA

Candidate

Thomas OSORIO

April 2024

Summary

This work is a contribution to the multisensor project. The main objective of this project is to implement a cost-effective solution for monitoring the environmental quality of many rooms of the Politecnico Di Torino. The contribution of this thesis work is to enable the user to visualize real-time measurements and perform statistics (average, standard deviation, minimum, maximum, 10th percentile, and 90th percentile) of time group intervals (3h, 24h, 3d, 1W, and 1M) over some time in the past.

The Multisensor is designed to monitor environmental quality in various indoor locations at Politecnico di Torino. Innovative boards have been developed with sensors to measure metrics such as temperature, humidity, illuminance, air pollution, and sound pressure. To compute the environmental quality, these objective metrics are used to calculate indexes: thermal comfort, visual comfort, acoustic comfort, and indoor air quality. The average of all these indexes corresponds to the indoor environmental quality. Additionally, the project involves creating a web application named 'Promet&O' to visualize the collected data on dashboards and to collect subjective feedback on environmental comfort through questionnaires. This dual approach of objective measurements and subjective assessments aims to provide a comprehensive understanding of indoor environmental quality.

This work primarily focuses on the development and implementation of a scheduling component. Its main functionality is to facilitate the calculation of statistics and comfort indexes. Utilizing the APScheduler library, the scheduler is designed to perform pre-aggregation of data collected by IoT sensors to enhance the efficiency of database operations. It schedules tasks at regular intervals and operates in the background, managing tasks. For this work, it has been decided to store the pre-aggregated statistics into a five-minute and one-hour views table. It reduces considerably the number of points to aggregate when visualizing the time-series dashboard. Moreover, it calculates the different comfort indexes to get values at frequent intervals.

The development and functionality of a RESTful API are implemented to serve as a bridge between the project's database and its web application frontend. This API is crafted to meet the project's requirements for accessing aggregated statistics

about the environment and comfort indices across the time group intervals described above over a defined time in the past. Built on the Flask framework and employing the Dependency Injection technique for modular and maintainable code, the API architecture incorporates additional extensions like Marshmallow for serialization and deserialization of data. It supports diverse endpoints, enabling requests for specific data aggregations, and comfort indices calculations, and facilitating the dynamic visualization of environmental metrics through the web application.

To ensure flexibility, innovation, and configurability all components are deployable using docker. All of these components can be deployable parametrizing the environmental variables. Moreover, a specific MQTT client has been developed to attend different experiments involved with the multisensor project. The MQTT Client facilitates real-time data flow between IoT devices and the server, ensuring efficient communication within the monitoring system.

This work implemented a complete backend management system to improve responsiveness and efficiency for an IoT monitoring web application. The strategy and architecture chosen for each component provide a flexible and scalable software system that can be easily evolved in the future.

Acknowledgements

I would like to thank my supervisors, Dr. Pietro Chiavasssa and Pr. Antonio Servetti, who have supported me throughout this thesis work. Their guidance and experience have been crucial for this production.

I also wish to dedicate this thesis to my parents, who have given me everything needed to grow into the professional I aspire to be and achieve my goals. From my earliest years, they have stimulated my curiosity and inspired me to pursue a career in engineering.

I would also like to express my gratitude to everyone who has contributed during my studies. A special mention goes to my brothers and grandparents, whose encouragement and belief in me have been a constant source of strength and inspiration.

Thomas Osorio, 2024

Table of Contents

List of Tables VIII

List of Figures IX

Acronyms XI

1 Introduction 1
- 1.1 Data management for real-time IoT monitoring service . . . 1
- 1.2 Environmental monitoring IoT system . . . 2
- 1.3 Purpose of the project . . . 2
  - 1.3.1 The multisensor project . . . 2
  - 1.3.2 Starting point . . . 3
  - 1.3.3 Challenges . . . 4
  - 1.3.4 Project goals . . . 5
  - 1.3.5 Methodology and Approach . . . 6
  - 1.3.6 Organization . . . 6

2 The multisensor project 7
- 2.1 Objectives . . . 7
- 2.2 The metrics . . . 7
  - 2.2.1 Sensors measurements . . . 7
  - 2.2.2 Comfort Domains . . . 8
  - 2.2.3 Thermal comfort . . . 8
  - 2.2.4 Acoustic Comfort . . . 10
  - 2.2.5 Visual Comfort . . . 11
  - 2.2.6 Indoor Air Quality - IAQ . . . 11
  - 2.2.7 Indoor Environment Quality - IEQ . . . 12
- 2.3 The web application . . . 12
  - 2.3.1 Visualization . . . 12
  - 2.3.2 Questionnaire . . . 14

3 The scheduler 16
3.1 Theory . . . 16
3.2 Motivation and Requirements . . . 16
3.3 APScheduler library . . . 17
3.4 Architecture and Implementation . . . 18
3.4.1 The scheduler object . . . 19
3.4.2 External Data Aggregation . . . 21
3.4.3 Indoor Aggregations . . . 25

4 The API 35
4.1 Requirements . . . 35
4.2 Architecture . . . 36
4.2.1 Flask API . . . 36
4.2.2 Dependency Injection technique . . . 37
4.2.3 Marshmallow . . . 38
4.2.4 Other extensions . . . 39
4.3 Implementation . . . 40
4.3.1 Application entrypoint . . . 40
4.3.2 Working hours calendar . . . 42
4.3.3 Endpoints structure . . . 43

5 Applications 52
5.1 MQTT Client . . . 52
5.1.1 Definition . . . 52
5.1.2 Paho MQTT Python . . . 52
5.1.3 Client Architecture . . . 53
5.1.4 Subjective Sensors . . . 54
5.2 Docker . . . 56
5.2.1 DockerFiles . . . 56
5.2.2 Environmental Variables . . . 57
5.2.3 Docker compose . . . 60
5.2.4 Configuration by projects . . . 61

6 Conclusion 64

List of Tables

2.1 Maximum, minimum, and default values for different timings. . . . 9
2.2 Report time for comforts 9

3.1 FiveMinAvgMeasure and OneHourAvgMeasure tables 26
3.2 ViewUpdate table 27

4.1 Table selection according to measurement type and avg parameter . 47

5.1 Configuration Parameters 58
5.2 MQTT Configuration Parameters 58
5.3 Scheduler and API Configuration Parameters 59
5.4 Environmental Quality Thresholds 60
5.5 Configuration Parameters for API 60

List of Figures

1.1 Software architecture 3
1.2 Excerpt from Database model 5

2.1 Image of a Board 8
2.2 Statistics of one board over the last 3 hours 13
2.3 Customizable dashboard in grafana 14
2.4 Time series plot 14
2.5 Time series comparison 14
2.6 Global satisfaction 15
2.7 Metric selection 15
2.8 Score performed 15
2.9 Subjective (yellow points) vs Real scores comparison 15

3.1 Scheduler Architecture 19
3.2 Scheduler Overview 21
3.3 External Data Overview 21
3.4 Indoor Data Overview 25
3.5 Example for five-minute aggregations 27
3.6 Indoor Aggregation 34

4.1 API Architecture 39

Acronyms

IoT
: Internet of Things

MQTT
: Message Queuing Telemetry Transport

ORM
: Object-Relational Mapping

API
: Application Programming Interface

EPA
: Environmental Protection Agency

PCB
: Printed Circuit Board

Chapter 1

Introduction

1.1 Data management for real-time IoT monitoring service

Data management is a key aspect of real-time IoT monitoring services. Internet of Things refers to a network of physical devices that are interconnected and capture heterogeneous data from different places. The more locations and measured quantities there are, the more data needs to be collected. Moreover, real-time services transmit data at high frequency. This leads to a vast amount of information that needs to be collected and organized with specific strategies for various applications. The effective handling of this data fosters advancements in areas ranging from environmental monitoring to healthcare systems. This constant flow of high-volume data underscores the importance of developing robust storage and processing systems that can efficiently handle this information.

The two main challenges in real-time monitoring and managing the massive data volumes collected by processing and analysis services involve time efficiency and resource effectiveness. Time efficiency is crucial because it involves accomplishing objectives or tasks swiftly without sacrificing output quality. On the other hand, resource effectiveness focuses on maximizing the use of available computational resources. In the context of IoT (Internet of Things) management, databases require specifically tailored systems and algorithms capable of processing and analyzing large data volumes efficiently, without excessive computational demands or excessive time consumption. Moreover, the diverse nature of IoT data complicates its management. Data types vary widely, from simple temperature readings to complex quality indices, necessitating flexible and adaptive database management systems that can cater to the varied requirements of different IoT applications.

According to Forbes [1], by the end of 2024, more than 207 billion IoT devices will be operational in the field. Transform Insights provides another relevant

statistic: by 2025, 75% of the devices in the world will be connected [2]. This growth shows that there is a real need for investment not only in device technology improvement but also in managing the data.

1.2 Environmental monitoring IoT system

One of the most important applications of IoT services is the analysis of environmental conditions to optimize, understand, or predict phenomena such as air or water quality. Connected devices offer metrics to obtain accurate and up-to-date information to make strategic decisions on specific environments and strategies. The network is composed of sensors that are strategically deployed across various environments, capable of measuring a range of metrics including atmospheric gases, particulate matter, temperature, humidity, light intensity, and sound pressure. The collected data is transmitted in real-time to centralized databases, where it undergoes processing and analysis. This real-time data collection enables immediate responses to environmental changes.

The Conserv platform [3] illustrates the importance of environmental real-time monitoring by IoT devices. Through an innovative approach, the platform offers real-time data collection, analysis, and response that enable better management of the environmental condition in the context of museum conservation. The sensors provide information on temperature, humidity, and light exposure, which are crucial for the preservation of museum objects. The platform provides immediate alerts and actionable insights to improve conservation strategies. The challenges faced and solutions implemented by Conserv in managing and analyzing large-scale sensor data represent a significant trend in real-time monitoring within the IoT field. They use sophisticated database systems to efficiently handle, store, and query large datasets. This platform shows the necessity of developing accurate software solutions to manage real-time sensor databases.

1.3 Purpose of the project

1.3.1 The multisensor project

The thesis is a contribution to the multisensor project. Its objective is to provide a solution for monitoring environmental quality in various indoor locations. The goal is to develop sensors that measure data such as temperature, humidity, illuminance, air pollution, and sound pressure. These sensors, deployed across the offices of Politecnico di Torino, return individual environmental information about each room. All these metrics offer information to evaluate comfort scores about the environment as thermal comfort, acoustic comfort, visual comfort, and indoor air quality comfort.

These scores are useful to understand the quality of the environment. The second objective is to create a web application that provides a dashboard for the users in the office but also enables data analysis for experts. In addition to the dashboard, users also need the ability to complete questionnaires about their satisfaction with the environment. Utilizing these subjective values, the application will enable a comparison of users' satisfaction levels with theoretical comfort indices computed by the software solution.

1.3.2 Starting point

This project is part of a bigger one that intends to develop a solution to collect data from research IoT devices positioned in different places. The thesis work focuses on the web application, specifically on the server-side management of the database. This project continues the work previously carried out [4]. The software architecture

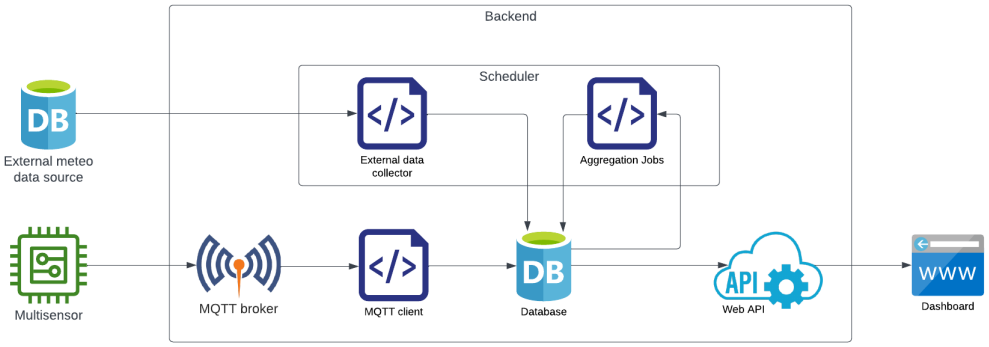

Figure 1.1: Software architecture

was already designed. The IoT hardware board, equipped with a variety of sensors designed to measure different metrics, sends data to the Mosquitto MQTT broker. Mosquitto is an open-source message broker that uses MQTT (Message Queuing Telemetry Transport) as a protocol. Like any broker, it enables data exchange, in this case, between the IoT device and the MQTT client.

The MQTT Client, previously implemented in Python, communicates with the broker to receive messages coming from the sensors and save the contained data inside the database. The client, when started, registers itself to topics with the broker to specify a channel (topic) of interest. The objective is to define the target when a message is received. In simpler terms, the MQTT client updates the database each time a sensor sends a measurement, regardless of the type of measurement.

The previous work also modeled the database structure [4]. It is a MySQL storage hosted on a virtual machine on the university cloud. The database utilizes Python with SQLAlchemy as the Object-Relational Mapping (ORM) tool. SQLAlchemy allows for database manipulation through Python code, making it easier to interact with the database using Python objects. This approach simplifies the integration of the database with other components of the system. This database is designed to support the complexities of a real-time IoT monitoring service. It covers essential characteristics like real-time data handling. The Measure table ensures with the logical sensor table an efficient handling and storage of real-time data. This table receives the data from the MQTT client in the following format: ID of the sensor, value of the measurement, and date of the measurement. Secondly, the model supports detailed sensor management using experiment tracking. The logical and physical sensors represent respectively the hardware sensors and their measured time-series. They belong to a specific board (IoT device). Those boards are completed with the BoardExperiment and Experiment tables. They enable the organization of sensors and boards according to the experiments to which they belong. Moreover, this database facilitates the configuration of boards and sensors using a table named BoardConfig, where users can record changes in the board's configurations. Finally, the database ensures consistent data representation with the UnitOfMeasure table which provides information on the measured quantity of each logical sensor.

1.3.3 Challenges

The challenge of this software architecture lies in the growth of the database, as sensors send measurements at least once per minute. Additionally, each board will have different sensors measuring various metrics and at least one board will be positioned in each location. Thus, there can be tens of thousands of measurements in one day.

Furthermore, with IoT measurement data in time series format, various analyses can be conducted, including real-time reporting and statistical analysis over time. Generally, users aim to identify patterns, unusual data, or trends over time within the data. As the database size increases, optimally loading the data can become challenging when the time interval for analysis is too large.

In the realm of environmental data management, there is a growing interest in calculating specific quality indices or comfort values to evaluate the environment using defined metrics like thermal comfort, visual comfort, acoustic comfort, air quality index, and environmental quality index. To begin with, these calculations may be too complex to be performed on simple sensors and can be hard to configure directly as it is possible to use many parameters to estimate these values. Secondly, the thermal comfort index requires the retrieval of external data each day as

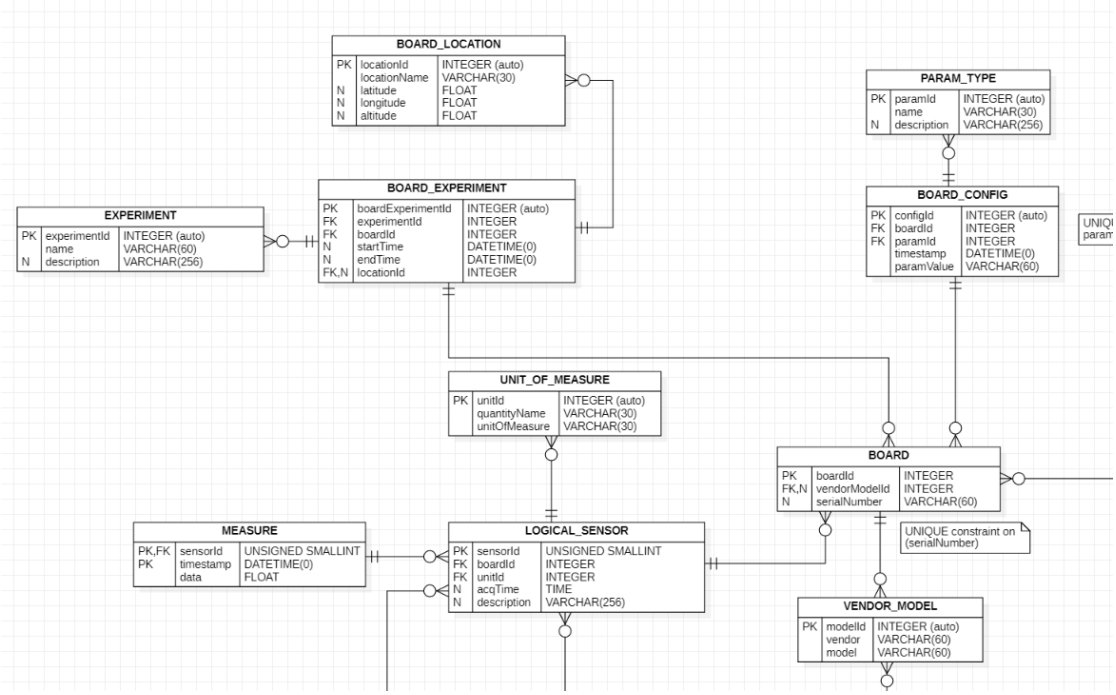

Figure 1.2: Excerpt from Database model

the outdoor temperature. This can be sourced from external services that the sensor can't access. Finally, the solution should be integrated with commercial sensors that can't compute arbitrary indexes. Therefore, there is a need to perform these calculations using a software-based approach on the server side of the web application.

1.3.4 Project goals

Consequently, the primary objective of this project is to optimize the backend server, thereby enhancing the efficiency of database aggregation and enabling rapid analysis. The specifications of the thesis were to enable specific statistics from different time aggregations from now to any point in the past. The required time aggregations included 3 hours, 24 hours, 3 days, 1 week, and 1 month, while the necessary statistics were the average, standard deviation, minimum, maximum, 10th percentile, and 90th percentile. Additionally, these statistics must be visualized using Grafana on time-series dashboards, adhering to its specifications. Grafana is an open-source analytics and interactive visualization web application that provides charts, graphs, and alerts for the web when connected to supported data sources. For instance, it should enable the visualization of metrics from the present to the previous day, grouped in 3-hour intervals. Another aspect involved calculating

comfort values at regular intervals to ensure frequent storage of these measurements in the database. This required developing algorithms that are executed regularly, and performing operations on the data received from the sensors and external data sources.

Therefore, the project necessitated a system for optimizing loading times when performing these statistical analyses and enabling database connectivity via a client-server architecture.

1.3.5 Methodology and Approach

To address the requirements of this project, multiple software solutions were implemented on the server side. These are the main contributions of this project.

1. **The scheduler** performs pre-aggregation to reduce the number of points to analyze for each specific interval. The project required developing a solution to perform regular database operations, focusing on computing key statistics such as average, standard deviation, minimum, and maximum, and assessing environmental quality and comfort through specific metrics as seen above.

2. **The API** enables the aggregation of these statistics over the required time intervals. It should be easily accessible by Grafana but also other solutions in the future. It should also facilitate CSV downloads to easily retrieve reports for specific intervals.

3. **Docker containerization** of software components to be highly configurable.

4. **MQTT clients** capable of transmitting subjective values from questionnaires completed by individuals to the database, as well as retrieving data from the multi-sensor system.

1.3.6 Organization

The thesis is structured into 4 chapters and a conclusion. The first chapter will present the multi-sensor project, its purpose, and its characteristics. The second and third chapters explain the core part of the work which is the development of the scheduler and the API. Finally, the last chapter describes the deployment and applications of the solution.

Chapter 2

The multisensor project

2.1 Objectives

This chapter introduces the multisensor project, elucidating the rationale behind the thesis and addressing the challenges previously outlined. It is important to note that this chapter describes the project in its entirety and goes beyond the contribution of the thesis work. The contributions will be highlighted in detail in the following chapters. The multisensor project aims to create a cost-effective, easily deployable, and manageable solution for monitoring environmental quality in various rooms at Politecnico di Torino. The research group's involvement spans the entire spectrum of the project, from the hardware development of the devices to the visualization of data on dashboards.

2.2 The metrics

2.2.1 Sensors measurements

The electronic group is developing new IoT devices designing PCBs, communication components, and more. The hardware IoT devices were designed as small boards assembled with various sensors. These boards are placed in different rooms of the Politecnico di Torino and measure specific metrics at varying frequencies. Each sensor on the board is responsible for measuring one of these metrics.

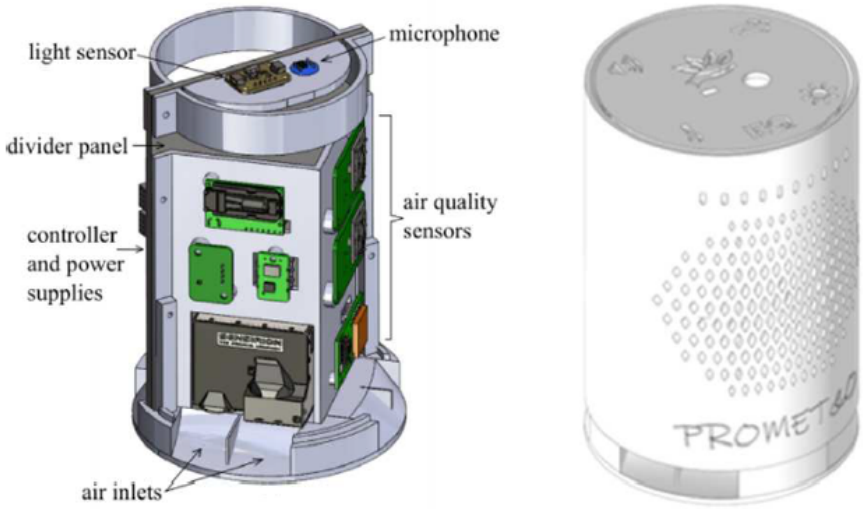

Figure 2.1: Image of a Board

All sensors measure units within specific comfort domains, including thermal, indoor air quality, and visual, and acoustic comfort. Each of them takes measurements at a defined sampling time and transmits statistics on the measured data to the broker at a specific report time. Some comfort values, such as visual comfort, indoor air quality, and acoustic comfort, are calculated directly by the sensors. However, as will be elaborated later, these values could also be computed directly on the server. They are all expressed as ratios, represented in percentages. The real-time determines the frequency with which the compliance values of the various quantities are calculated and with which the data is sent to the server to update the dashboard more quickly. These times are defined for each sensor according to the following table 2.1.

2.2.2 Comfort Domains

In each comfort domain, it is possible to calculate the domain's score by examining all its units. Furthermore, the combination of all these domain scores reveals the indoor environmental quality score. These metrics contribute not only to optimizing the users' environmental experience but also play a crucial role in the energy efficiency and sustainability of the rooms.

2.2.3 Thermal comfort

In the thermal domain, air temperature sensors record measurements in degrees Celsius (°C), a key metric for calculating thermal comfort. These sensors transmit

Table 2.1: Maximum, minimum, and default values for different timings.

Measures	SAMPLING TIME			REPORT TIME		
	min	def	max	min	def	max
Temperature	0.1 s	1 s	30 s	30 s	30 s	15 min
Relative Humidity	0.1 s	1 s	30 s	30 s	30 s	15 min
Illuminance	1 s	1 s	30 s	30 s	30 s	15 min
Sound Pressure	22 kHz/0.5 s			5 s	5 s	15 min
Formaldehyde	0.5 s	3 s	60 s	30 s	1 min	15 min
Particulate	1 s	3 s	60 s	30 s	1 min	15 min
TVOC	3 s	3 s	60 s	30 s	1 min	15 min
NO2	1 s	3 s	60 s	30 s	1 min	15 min
CO	1 s	3 s	60 s	30 s	1 min	15 min
CO2	1 s	3 s	60 s	30 s	1 min	15 min

Table 2.2: Report time for comforts

Measures	REPORT TIME	COMPUTED BY
Thermal comfort	5 min	server
Visual comfort	30s	multisensor
Indoor Air Quality	1 min	multisensor
Thermal comfort	1 s	multisensor
Indoor Environmental Quality	5 min	server

data to the database every 30 seconds or one minute (report time). Subsequently, thermal comfort analysis is conducted every 5 minutes using these values.

The work focuses on a specific thermal comfort metric, known as the Adaptive Thermal Comfort metric. It assesses the relationship between indoor and outdoor temperatures. According to the European Standard EN16798-1 [5], an individual's thermal comfort is influenced by their recent thermal history and current environmental conditions. This suggests that individuals can naturally adapt to a wider range of temperatures based on their prior experiences and expectations. This concept defines an optimal range representing the equilibrium point at which the indoor environment is perceived as comfortable by most occupants. This range is referred to as the operative temperature. The center of this interval is deemed the ideal comfort temperature or the optimal operative temperature.

This approach is mathematically modeled by correlating indoor comfort temperatures with an averaged outdoor temperature: the running mean outdoor temperature, denoted as θ_{rm}. This variable signifies the thermal adaptation of

occupants over time. The outdoor running mean is determined using the formula:

$$\theta_{\text{rm}} = (1-\alpha)(\theta_{\text{ed-1}} + \alpha * \theta_{\text{ed-2}} + \alpha^2 * \theta_{\text{ed-3}})$$

where $\theta_{\text{ed-i}}$ represents the daily mean outdoor air temperature for the i^{th} day preceding the current one, and alpha is a constant between 0 and 1 (in general 0.8), captures this adaptive process. This formula employs an exponentially weighted average that prioritizes recent temperatures but still considers past conditions. The weights assigned decrease exponentially with each preceding day, illustrating the diminishing influence of older temperatures on the present sense of comfort.

Based on this running mean, it is possible to establish the comfort zone. The operative temperatures can be modeled into three categories (I, II, III) where the upper and lower limit are defined as:

1. Category I:
   (a) upper limit: $\theta_0 = 0{,}33 * \theta_{\text{rm}} + 18{,}8 + 2$
   (b) lower limit: $\theta_0 = 0{,}33 * \theta_{\text{rm}} + 18{,}8 - 3$
2. Category II:
   (a) upper limit: $\theta_0 = 0{,}33 * \theta_{\text{rm}} + 18{,}8 + 3$
   (b) lower limit: $\theta_0 = 0{,}33 * \theta_{\text{rm}} + 18{,}8 - 4$
3. Category III:
   (a) upper limit: $\theta_0 = 0{,}33 * \theta_{\text{rm}} + 18{,}8 + 4$
   (b) lower limit: $\theta_0 = 0{,}33 * \theta_{\text{rm}} + 18{,}8 - 5$

It is essential to note that the optimal operative temperature is common and therefore expressed as $\theta_c = 0{,}33 * \theta_{\text{rm}} + 18{,}8$.

Finally, adaptive thermal comfort is defined as the ratio of the number of temperatures (°C) measurements within a specified category range to the total number of temperature measurements within a given time interval. This approach, as outlined in the EN16798 standards [5], offers a dynamic and nuanced method for maintaining comfort. It proves particularly advantageous in climates experiencing significant temperature fluctuations and in enhancing energy efficiency.

2.2.4 Acoustic Comfort

Acoustic comfort pertains to the degree of satisfaction of room occupants with the sound quality of the environment. It is quantified by the noise level, also known as the sound pressure level, which represents the total volume of sound within a

space measured in decibels (dB), and has a report time of 5 seconds. Excessive sound pressure levels can result in discomfort. Therefore, acoustic comfort can be expressed as the ratio of the number of sound pressure levels in dB that fall below a specified threshold to the total number of measurements of sound pressure within a given time interval. Its report time is also 5 seconds.

2.2.5 Visual Comfort

Visual comfort is an aspect of environmental quality that informs how well a space is lit. It is a state in which the lighting in an environment allows for the clear and comfortable visibility of tasks like writing, reading, or typing. This index is determined by the illuminance measured in lux (lux), which measurements are returned every 30 seconds to the database. The visual comfort is also performed every 30 seconds by the sensors and represents the ratio of measurement above a specific threshold.

2.2.6 Indoor Air Quality - IAQ

According to the EPA (United States Environmental Protection Agency), Indoor Air Quality (IAQ) is the metric that reflects the impact of the air quality on the health and comfort of building occupants [6].

To evaluate the IAQ index, it is required to retrieve data from different factors. There are many substances in the air considered as air pollutants and their levels are indicators of IAQ. In this project, six of them are measured and sensors send these measurements each 1 minute:

- Carbon Monoxide (CO, mg/m^3): It can emanate from incomplete combustion in appliances such as stoves and heaters [7].
- Carbon Dioxide (CO2, ppm): It is not toxic but elevated concentration can be dangerous.
- Nitrogen Dioxide (NO2, μ/m^3): It comes from combustion processes like gas stoves and tobacco smoke [8].
- Particulate Matter (PM, μ/m^3): A mixture of solid particles and liquid droplets found in the air like dust, dirt, soot, and smoke. They are categorized by the diameter size in micrometers. In this thesis, only the following quantities are measured:
  1. PM 2.5
  2. PM 10

- Total Volatile Organic Compounds (TVOCs, μ/m^3): Compounds that have a high vapor pressure and low water solubility. This group of pollutants is composed of solvents or more generally human-made chemicals.
- Formaldehyde (CH20, μ/m^3): A gas found in many building materials like resin, wood, or pesticides.

As for acoustic and visual comfort methodology, for each measured quantity the number of measurements below specific thresholds is counted and compared to the total number of measurements to calculate the ratio. The IAQ is the average of these ratios.

2.2.7 Indoor Environment Quality - IEQ

The Indoor Environment Quality (IEQ) index represents a comprehensive measure that combines the four previous and needs to be performed every 5 minutes using the following formula for each interval of time:

$$IEQ = \frac{TC + VC + AC + IAQ}{4}$$

It is a holistic metric that describes the conditions inside the room as a physical characterization of indoor environments in terms of thermal, acoustic, visual comfort, and indoor air quality. As the thermal comfort, it needs to be performed every 5 minutes.

All these measurements and scores offer theoretical insights into environmental quality. However, these are raw values that require processing for analysis.

2.3 The web application

This project is responsible for developing a web application named 'Promet&O'. This application serves as a human interface to display visual information about the rooms and also facilitates the completion of questionnaires to gather subjective information about room occupant satisfaction.

This web application will be made available to all occupants of rooms at Politecnico di Torino (Polito), accessible through a login process. The objective is to gather a sufficient number of responses to enable meaningful analysis of these data.

2.3.1 Visualization

Enabling users to visualize and analyze data is an essential component of the multisensor project. For each board located in different rooms, the goal is to allow

both application users and the occupants of the room to understand their current environmental conditions and to analyze the environment's evolution over time. In the context of IoT, visualizing time series data entails plotting data points along a time scale, which is instrumental in analyzing trends, patterns, and anomalies in the data collected from various IoT devices. The challenge of this task lies in representing the data in diverse ways to provide multiple perspectives on a specific metric.

The multisensor web application represents data for each metric using various time aggregations. The initial aggregation is real-time, illustrating on a plot the measurements transmitted by the sensor. Subsequent aggregations group measurements into specific time intervals: 3 hours, 24 hours, 3 days, 1 week, and 1 month and compute statistical analyses for these periods. It includes average, standard deviation, minimum, maximum, 10th percentile, and 90th percentile.

For each board, the application displays its dedicated dashboard. The initial representation is numerical, reflecting the global state of the room. This is achieved by displaying either the most recent measurements for real-time aggregation or the latest statistics over a selected period. Figure 2.2 illustrates the statistics of one board over the past three hours, showing all the average values for each metric and the statistics for the IEQ score (bottom left). Similarly, when the RT (Real-Time) button is selected, the application displays the latest measurements for all metrics, sourced directly from the sensors.

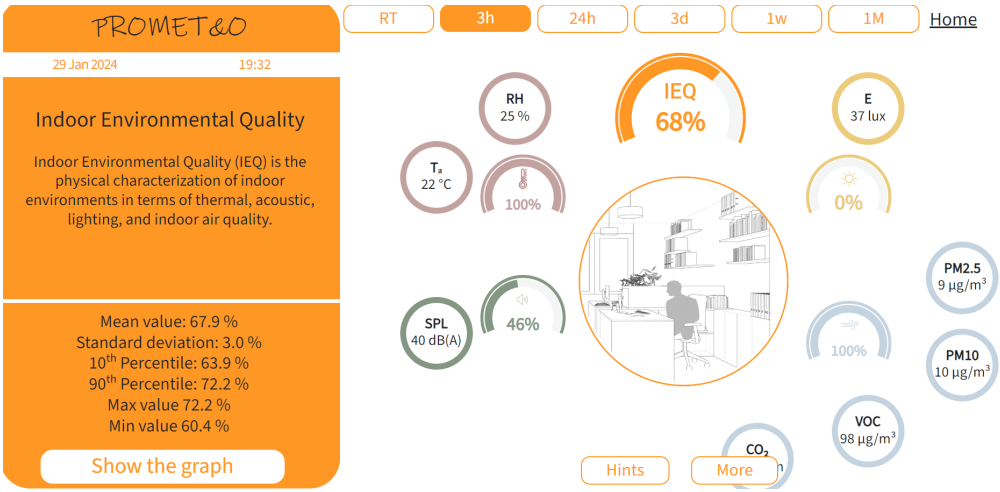

Figure 2.2: Statistics of one board over the last 3 hours

This initial data depiction provides an immediate or recent understanding of the environmental conditions. However, it is useless to understand evolution, trends, or patterns over time for environmental metrics. To address this, the application offers time series plots (By clicking on the bottom left button "Show the graph").

The selected tool for these plots is Grafana, an open-source platform optimized for monitoring and observability of data points. Users can visualize data from various sources on a customizable dashboard (Figure 2.3). Moreover, these dashboards are accessible in the front-end through a specific URL format exclusive to Grafana (Figure 2.4).

Lastly, Promet&O facilitates the comparison of time series plots across various metrics, as illustrated in figure 2.5. This feature significantly enhances analytical capabilities and user comprehension. By selecting specific units, users can correlate trends in indoor environmental analysis and examine the variables that influence it. On the left side of the menu, the user can select the comfort domain score, and on the right side the specific units.

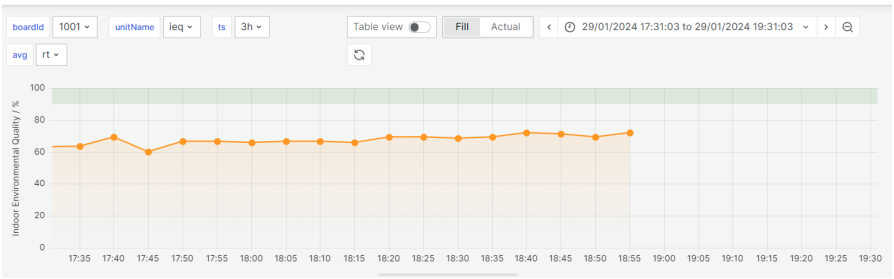

Figure 2.3: Customizable dashboard in grafana

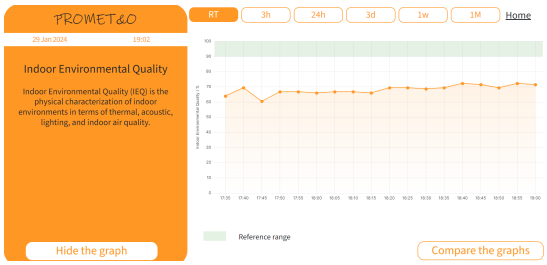

Figure 2.4: Time series plot

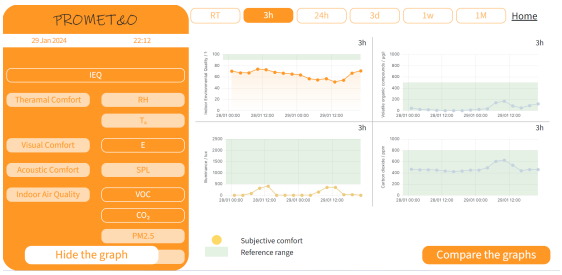

Figure 2.5: Time series comparison

2.3.2 Questionnaire

As previously mentioned, the measurements and aggregations displayed on the dashboard are objective information. The Promet&O application stands out by also incorporating subjective information provided by the occupants of the rooms. The web application aims to gauge the satisfaction of individuals within these rooms, offering insights into the relationship between subjective opinions about environmental quality and the actual measured scores. Users have the opportunity

to rate their global satisfaction across four metrics: thermal comfort, acoustic comfort, visual comfort, and indoor air quality (Figure 2.6). In Figure 2.7, if they answer they are happy about the environment, they are asked to select the most satisfying domains. Otherwise, the are ask to select the less satisfying ones. Thus, they choose which ones they will give a score. Their responses generate a score for each selected metric (Figure 2.8), which is then transmitted via a specific MQTT client for subjective data to the database. This feature enables users to compare the actual environmental scores, as measured by sensors, with the subjective assessments provided by room occupants (Figure 2.9).

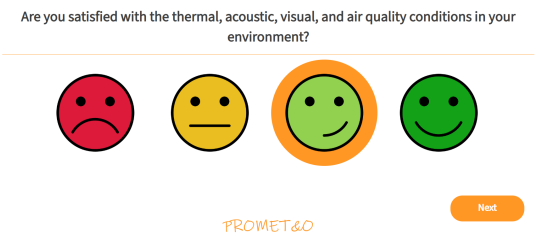

Figure 2.6: Global satisfaction

Figure 2.7: Metric selection

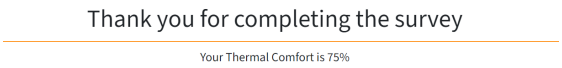

Figure 2.8: Score performed

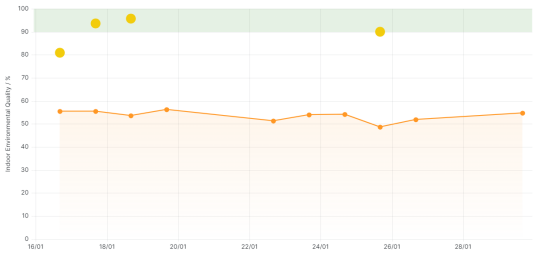

Figure 2.9: Subjective (yellow points) vs Real scores comparison

Chapter 3

The scheduler

This chapter explains the design and implementation of the scheduler. In our context, it refers to the software solution that provides the capability to run regular tasks for performing statistics and comfort index calculations.

3.1 Theory

A scheduler is a program or a component of a system responsible for managing the execution of tasks, functions, or jobs within an application. At the application level, schedulers determine the timing of and the methods for executing tasks. Moreover, it ensures that resources are utilized efficiently and tasks are completed in a reasonable time. These jobs vary from basic tasks like data backup to more complex ones like those in real-time data processing systems.

3.2 Motivation and Requirements

IoT real-time systems involve handling large amounts of data. It requires time to process insights or control information. For this project, users need to be able to perform statistics and retrieve measurements obtained from sensors over different time intervals. Instead of computing this aggregation entirely on-the-fly, a scheduler has been implemented to perform pre-aggregation. It enables faster and more responsive manipulation of the database.

As seen in the previous chapter, to handle data for larger time aggregations (3 hours, 24 hours, 3 days, 1 week, 1 month) and compute comfort values (thermal, acoustic, visual, air quality, environmental), the application needs to process sub-aggregation and store the results. This approach enhances the efficiency and accuracy of data calculations. It involves breaking down the data into smaller groups.

For this project, it has been decided that there will be two tables of pre-aggregated values. The scheduler performs five-minute and one-hour aggregations: it stores the aggregation results in two tables called *FiveMinAvgMeasure* and *HourAvgMeasure*, respectively. This frequent calculation ensures that the data is up-to-date and ready for further aggregation into larger time frames. For instance, if a sensor sends 2 measurements per minute instead of having 2880 values to aggregate over 24 hours, we will have 24 and 288 measurements aggregated in the *HourAvgMeasure* and *FiveMinAvgMeasure*, respectively.

Secondly, the daily thermal comfort calculation also requires the implementation of a scheduler. Each day, the running mean is computed to get the boundaries. This involves analyzing outdoor temperature data provided by external sources and storing it in the database. The scheduler also needs to be able to compute all the comfort indexes, as some experiments use commercial sensors that cannot calculate the indexes by themselves.

3.3 APScheduler library

The Advanced Python Scheduler (APScheduler) is a Python library that enables the scheduling of code execution [9]. It is the core framework of the scheduler object. It can schedule tasks to run later, either once or repeatedly. The library allows easy changes to tasks and has useful features for routine task management. APScheduler is particularly useful in web applications where background tasks need to be performed without disrupting the primary server process. Therefore, it is ideal for implementing the scheduler in a real-time Internet of Things (IoT) monitoring service.

APScheduler offers three primary scheduling systems:

1. Cron-style scheduling is inspired by the traditional Unix cron scheduler. It allows jobs to run at regular intervals as defined by cron expressions, with the option to set start and end times. APScheduler's cron expressions use lists of values in eight fields to describe the scheduling of the job.

2. Interval-based execution allows jobs to be executed at specified intervals, such as every 5 minutes or 3 hours, and supports optional start and end times.

3. One-off delay executes tasks at a specific date and time in the future. This is ideal for scheduling a single occurrence of a task.

These scheduling systems determine when and how often a job should be run. Three types of schedulers define how the scheduler operates within the application.

- The **Blocking Scheduler** is designed for applications where the scheduler is the main component. This category takes control of the program and blocks

any further processes until explicitly shut down. It is suitable for applications where the scheduler executes simple background processes or tasks that do not require concurrent execution with other parts of the application.

- **BackgroundScheduler** runs alongside other tasks. This scheduler is used in scenarios where the scheduler needs to coexist with other running processes, such as user interface components or other services. Its key advantage is its ability to allow scheduled tasks to be handled asynchronously alongside the primary operations of the system.

- **AsyncIOScheduler** is particularly beneficial for asynchronous applications. It provides seamless integration with the asyncio event loop. This scheduler is especially useful in network applications, web servers, and other I/O-bound processes that require efficient handling of asynchronous operations using the asyncio module.

Another main component of APScheduler is the Executor. It is responsible for the actual execution of tasks. Once a scheduler triggers a job, the executor takes over to run the task. In this context, the scheduler can be seen as a planner, deciding when and how a job should be executed. On the other hand, the executor is like a worker and carries out the task.

The third component of this library are the triggers. They are responsible for the scheduling logic and are the conditions that determine when a job should be executed. APscheduler supports various types of triggers, such as cron, interval, and date triggers. It allows a high degree of flexibility in scheduling.

3.4 Architecture and Implementation

In this section, it is explained how the scheduled tasks are implemented. The scheduler design, show in Figure 3.1, is also described

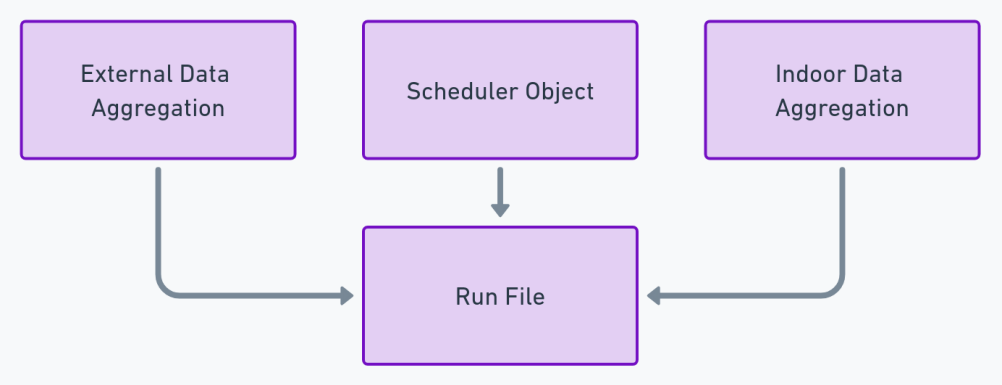

Figure 3.1: Scheduler Architecture

3.4.1 The scheduler object

The scheduler object is a class (Figure 3.2) representing the system's ability to handle concurrent tasks and efficiently perform data aggregation. The class initializes a BackgroundScheduler from the APScheduler library. The constructor for BackgroundScheduler sets another class variable, ThreadPoolExecutor. BackgroundScheduler uses this functionality to execute jobs in separate threads. The number of threads in the pool affects how many jobs can run in parallel. If more jobs are scheduled to run than there are threads available, they'll sit in the queue until a thread is free.

The second parameter of the BackgroundScheduler is a dictionary that sets the default job behaviors. It offers flexibility and configurability to the scheduler object. This is particularly important for ensuring that jobs are run in a predictable and controlled way, keeping them consistent with the specific requirements of the application. The users of this class can set the following parameters:

- **Coalesce**: When set to true, if a job is missed due to the scheduler being paused, or due to server failure, only the most recent instance of the job will run when the scheduler is resumed. This prevents jobs that would have been scheduled during the downtime, from being run.
- **Max instances**: The maximum number of instances of this job that can be run concurrently. The default is 1 which means that by default a new instance of the job will not start if a previous instance is still running. Increasing this number allows the same job to run in parallel. This can be useful in scenarios where the jobs are independent and can safely run simultaneously.
- **Replace existing**: It is a boolean variable that enables the scheduler to replace an existing job when adding a new job with an ID that already exists.

- **Misfire Grace Time**: This is a period (in seconds) that provides a window for the job to run even if it misses its scheduled time. For example, if a job is scheduled to run at a specific time but can't run at that exact moment (due to the scheduler being busy or offline), it will still run as long as the scheduler is started within this grace period.

- **Timezone**: This sets the timezone for the job's schedule. It ensures that the job's run times are interpreted in the correct timezone, which is essential for applications that operate across multiple time zones or need to adhere to a specific time zone regardless of the server's local time.

Job management capabilities form another aspect of the class. Methods for adding, removing, and checking jobs offer comprehensive control over the scheduled tasks.

The **add_job** function in this class serves as a method for scheduling new jobs in the APScheduler's BackgroundScheduler. It requires several parameters: *func* is the callable method to run at the times specified by the trigger. The trigger parameter determines when the job should be executed, such as at a certain time or regular intervals. *Id* is an optional parameter that uniquely identifies the job, useful for referencing the job at a later time. The function also accepts additional **options, which can be any other keyword arguments supported by APScheduler for job creation. Thus, this function adds jobs according to its parameters. On the other hand, the **remove_job** function removes a job from the scheduler using the job's identifier. The parameter *job_id* is the unique identifier of the job that needs to be removed.

print_jobs is a utility method within the class. It prints out a list of all the jobs currently scheduled in the APScheduler. It doesn't have any parameters and is useful for debugging and monitoring.

start and **shutdown** are two methods that control the operational state of the scheduler. The first initiates the scheduler, allowing it to begin executing the scheduled jobs. In contrast, the second stops the scheduler, halting the execution of all jobs.

Finally, there are two job-checking methods, **check_job** and **check_job_by_func**. **check_job** checks for the existence of a job with a specific **job_id** in the scheduler, prints a message, and returns a boolean value indicating whether or not the job does exist. **check_job_by_func** operates similarly but checks for a job with a specific function. It iterates through all jobs in the scheduler, comparing the function associated with each job to the provided *func* parameter.

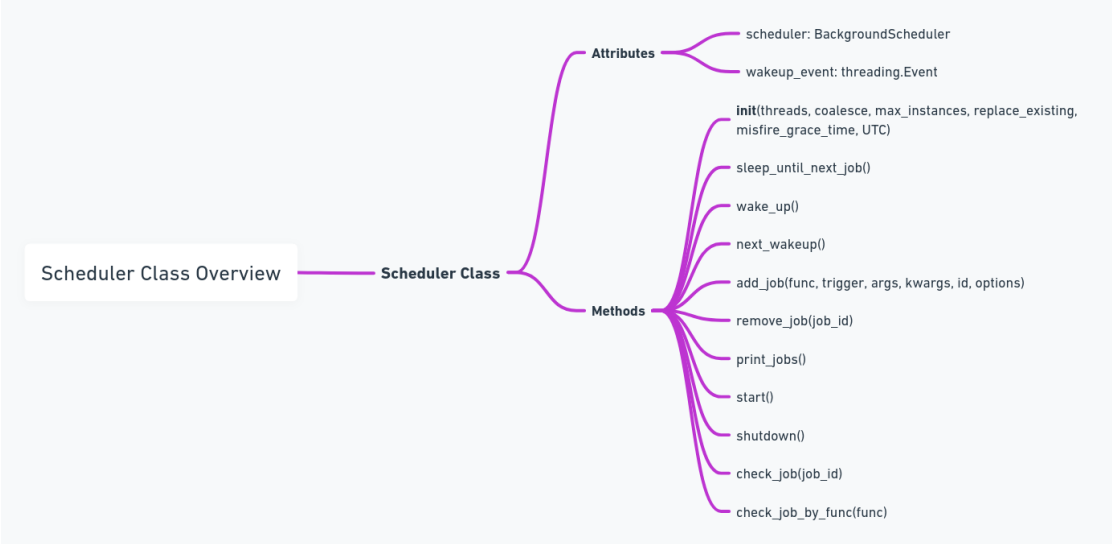

Figure 3.2: Scheduler Overview

3.4.2 External Data Aggregation

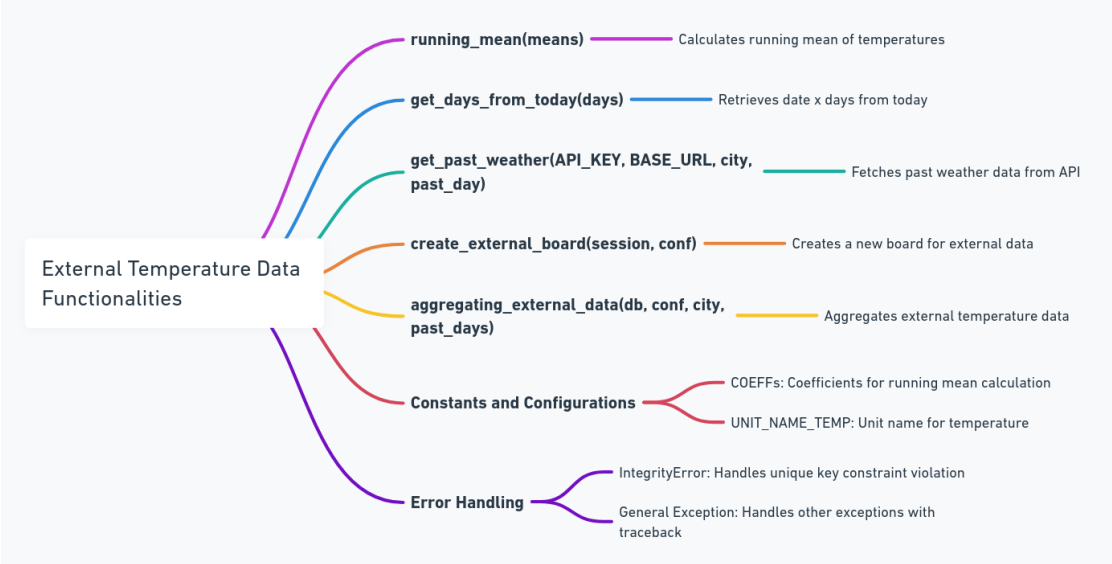

Figure 3.3: External Data Overview

As seen in the precedent chapter, the thermal comfort calculation requires fetching the outdoor temperature data. This functionality is enabled by a group of functions inside a specific file: external_data_aggregation.py. The structure of this file is

represented in Figure 3.3.

The **running_mean** function calculates the running mean of external temperatures over seven days as explained in section 2.2.3. This calculation provides a weighted average of recent temperature data. The function takes a list of mean temperatures for each of the previous seven days as its input. It then applies a set of predefined coefficients to these temperatures, representing their respective weights in the calculation. These coefficients are stored in a global variable COEFFS to be easily reusable in case of modifications. The weighted temperatures are summed and divided by a normalization factor (3.8) to obtain the running mean.

The **get_past_weather** function is designed to retrieve historical external temperature data for a specific city and day from the WeatherAPI [10]. It takes as input the API key, base URL, the city for which the weather data is needed (defaulting to "Torino"), and the number of days in the past to look up (defaulting to 7 days). It constructs a request to the WeatherAPI's history endpoint with these parameters, and upon successful retrieval, it extracts and returns the average temperature of the specified day. The function also includes error handling to log any issues encountered during the API request or data extraction process, such as request exceptions or unexpected response structures. The endpoint reached is called "http://api.weatherapi.com/v1/history.json". It takes as parameters "key", "q" and "qt", which are, respectively, the API key, the city, and the date. The API response is a JSON file that returns a field avgtemp_c for the average temperature of the day in °C.

The main function to aggregate the outdoor temperature is **aggregating_external_data** represented in the following code 3.1. First of all, the function has an optional parameter that represents the when it should perform the aggregation for the running mean. If the parameter is None, it takes the day of today. Then, it checks if the sensor to store the outdoor temperature exists in the database. If not, it creates a new entry board in the board table, which will be a virtual outdoor board with a single sensor. Another verification is to check if the running mean has already been calculated on the chosen day. As it is a daily measurement, it doesn't need to be calculated again e.g. in the case the scheduler is restarted. Otherwise, the algorithm calls the API to get the average temperature of the last 7 days throughout a loop. With these lists of averages, it is possible to perform the running mean of the current day and add it to the Measure table. If the *return_bool* parameter is set to true, it will return the running mean value inserted. The function includes robust error handling, management exceptions related to database integrity, and other unforeseen issues. It also logs these errors for troubleshooting and auditing purposes.

Listing 3.1: External Aggregation

```
def aggregating_external_data(db: DBInterface, conf:dict, city='Torino', past_days: int = 7, date = None, return_bool = False) -> None:
    try:
        with db.Session() as session:

            # region Calculate time of the day
            now_utc = datetime.now(timezone.utc)
            if not date:
                start_of_today = datetime(
                    now_utc.year, now_utc.month, now_utc.day, tzinfo=timezone.utc)
            if date:
                start_of_today = start_of_today = datetime(
                    date.year, date.month, date.day, tzinfo=timezone.utc)
            end_of_today = start_of_today + timedelta(days=1)
            # endregion

            # region Check if the sensorId exists and running mean
            sensor = session.query(LogicalSensor).filter_by(
                sensorId=conf["API_SENSOR_ID"]).first()

            if sensor is None:
                sensor = create_external_board(session, conf)

            running_mean_selection = (
                session.query(Measure.data)
                .filter(Measure.sensorId == sensor.sensorId)
                .filter(Measure.timestamp >= start_of_today)
                .filter(Measure.timestamp < end_of_today)
                .order_by(desc(Measure.timestamp))
```

```
29                  .first()
30              )
31              # endregion
32
33              if running_mean_selection:
34              else:
35                  # region 1. Fetch the WeatherAPI
36                  means = []
37                  for date in range(1, past_days+1):
38                      means.append(get_past_weather(
39                          conf["API_KEY"],conf["BASE_URL"
    ], city=city, past_day=date))
40
41                  rm = running_mean(means)
42                  # endregion
43
44                  # region 2. Add running mean into the
    measure table
45                  # Insert new records into the Measure
    table
46                  new_row = Measure(
47                      sensorId=sensor.sensorId,
48                      timestamp=start_of_today + timedelta
    (minutes=1),
49                      data=rm  # assuming running mean is
    the data we want to store
50                  )
51
52                  session.add(new_row)
53                  session.commit()
54                  session.close()
55                  if return_bool:
56                      return rm
57                  # endregion
58
59      except IntegrityError as e:
60          # Handle unique key constraint violation
61          session.rollback()
62          logging.error(f"IntegrityError: {e}")
63
64      except Exception as e:
```

```
65          tb = traceback.format_exc()  # Gets the
    traceback object
66          session.rollback()
67          logging.error(f"Database transaction error: {e}\
    nTraceback: {tb}")
```

3.4.3 Indoor Aggregations

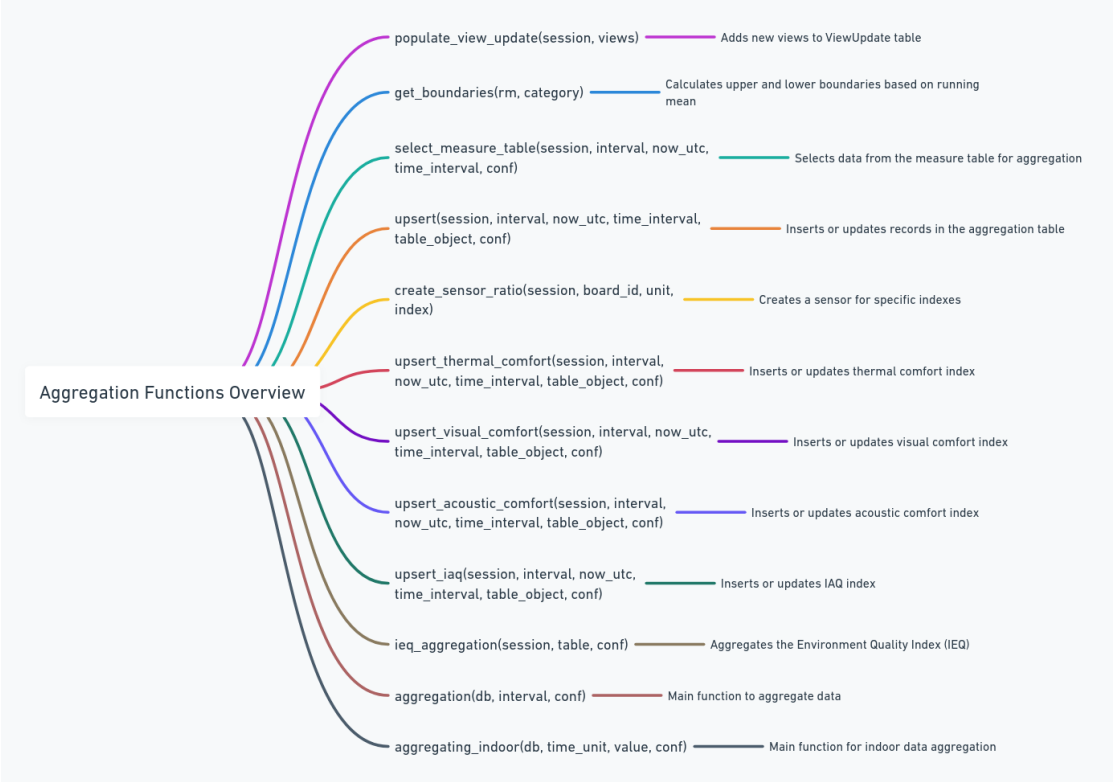

Figure 3.4: Indoor Data Overview

Indoor data aggregation is a key component of the system that employs SQL Alchemy as the main library for database management. It enables the scheduler to manipulate the MySQL database using object-oriented principles. This file is specifically designed for aggregating indoor environmental data over a specified time interval and is part of a larger suite of tools for managing and processing environmental data as shown in Figure 3.4.

The function **aggregating_indoor** serves as the main procedure for aggregating indoor data in a specified time interval. It takes as parameters the database instance,

the time unit in string format, and the absolute value of the time interval. It starts by logging information about the ongoing aggregation process, including the time unit and value of the interval (e.g., hours, minutes) specified by the parameters. In the thesis context, the time unit will be "minutes" or "hour" and, respectively, 5 and 1 as values, because the goal is to group aggregations by five minutes or one-hour intervals. An Interval object is created using these time parameters.

The **Interval** class is a utility class designed to represent a time interval. Its constructor takes two parameters: unit, a string that specifies the unit of time (e.g., 'minutes', 'hours'), and value, an integer representing the magnitude of the interval in the specified unit. The class stores these values in private instance variables _unit and _value. Additionally, the Interval class provides a method to_seconds(), which converts the interval to its equivalent duration in seconds. This conversion is based on the specified unit: if the unit is 'minutes', the value is multiplied by 60 (the number of seconds in a minute); if the unit is 'hours', it is multiplied by 3600 (the number of seconds in an hour). This class is used in the **aggregating_indoor** function to create an interval object that represents the period for which the indoor data aggregation is being performed.

The first step is to check if the views table already exists or has already been updated. In the context of this work, two specific view tables are required: the one_hour_avg_measure, and five_min_avg_measure. These tables are our target tables for the one-hour and five-minute aggregations. Both tables are structured in the same way with three mandatory and three optional attributes (Table 3.1). The *sensorId*, the *timestamp* of the interval, and the average of the measured data in this interval are the non-nullable ones. On the other hand, the table can be filled with the minimum value (*min)*, the maximum value (*max*), and the standard deviation (*std*) of the measurements inside the interval. A nested function, **populate_view_update** adds new view tables to the table ViewUpdate (Table 3.2) only if they are not already present in the current set of views, updating their last update timestamp to the current time.

sensorId	timestamp	avg	max	min	std
1001	2023-10-03 12:10:00	310.5	310.5	310.5	0
1001	2023-10-03 12:15:00	293.8	293.8	293.8	0
1001	2023-10-03 12:20:00	305.1	305.1	305.1	0

Table 3.1: FiveMinAvgMeasure and OneHourAvgMeasure tables

The second step is to define the time window that needs to be aggregated (Figure 3.5). The right boundary is the present time in UTC. This reference time is rounded down to the nearest interval time. For instance, if the performed aggregation is for five-minute aggregations and it is 15:43:00 UTC, the present time

name	lastUpdate
five_min_avg_measure	2024-02-15 00:30:00
hour_avg_measure	2024-02-15 00:00:00

Table 3.2: ViewUpdate table

will be defined as 15:40:00 UTC. As the timestamp always refers to the preceding interval, the 15:40:00 will refer to the aggregated statistics between 15:35:01 and 15:40:00. On the contrary, the left boundary refers to the last update registered in the ViewUpdate table. Therefore according to if it is a one-hour aggregation or a five-minute aggregation, it will select the table and check for the last update using the following SQLAlchemy query 3.2:

Listing 3.2: Query for the ViewUpdate table

```
1 last_update = session.execute(select(ViewUpdate.
    lastUpdate).where((ViewUpdate.name == table_name))).
    first()
```

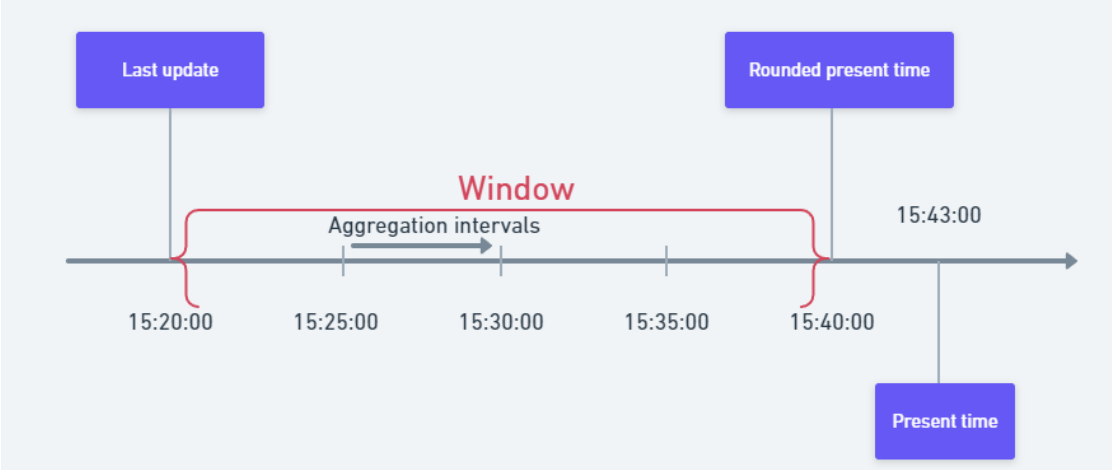

Figure 3.5: Example for five-minute aggregations

Finally, the scheduler starts to aggregate according to interval aggregation and inserts the statistics into the respective tables. Firstly, it performs the calculation for the non-comfort unit. The function performs a selection on the Measure table of the average, the minimum, the maximum, and the standard deviation grouped by intervals and sensorId. To get this interval, the measurement timestamps are normalized to the left boundary interval aggregation time (if a measurement corresponds to 15:43:01, it will be affected to 15:45:00). It executes a function

called **round_interval** 3.3 useful for grouping records by time intervals in SQL queries, when aggregating data over specific periods:

Listing 3.3: Round interval method

```
def round_interval(element, time_s: int):
    return func.from_unixtime(
        func.ceil(func.unix_timestamp(element) / time_s)
    * time_s
    ).label('timestamp_interval')
```

The **round_interval** function in Python takes two parameters: *element*, which is expected to be a timestamp or datetime-like object from a database, and *time_s*, an integer representing a time interval in seconds. This function rounds the element up to the nearest interval defined by *time_s*. It does so by converting the element into a Unix timestamp (the number of seconds since the Unix epoch) using $func.unix_timestamp(element)$, then dividing this timestamp by *time_s* and applying the $func.ceil$ function to round the quotient up to the nearest integer. This integer is then multiplied back by *time_s* to give a Unix timestamp that is rounded up to the nearest interval. Finally, $func.from_unixtime$ is used to convert this rounded Unix timestamp back into a datetime-like object. The .label('timestamp_interval') method is a SQL Alchemy feature that labels the resulting expression in the output of a query, making it easy to reference in the results. The final selection query is defined in code 3.4, with timestamp_interval representing the rounded interval group:

Listing 3.4: Selection method

```
select_measure_query = (select(
        Measure.sensorId,
        timestamp_interval,
        func.min(Measure.data).label('min'),
        func.max(Measure.data).label('max'),
        func.avg(Measure.data).label('avg'),
        func.std(Measure.data).label('std'),
    ).join(LogicalSensor, LogicalSensor.sensorId ==
Measure.sensorId)
        .join(Board, Board.boardId == LogicalSensor.
boardId)
        .join(VendorModel, VendorModel.modelId == Board.
vendorModelId)
        .join(BoardExperiment, Board.boardId ==
BoardExperiment.boardId)
```

```
12          .join(Experiment, BoardExperiment.experimentId
    == Experiment.experimentId)
13          .where(
14          and_(
15              Measure.timestamp >= time_interval,
16              Measure.timestamp <= now_utc
17          )))
```

Measurements aggregation

Then, the function (code 3.5) begins by checking whether the measurements list is empty or None. If either condition is true, it logs a message indicating that no measurements were found between two given times (present time and last update) and then exits the function early with a return statement. Following that, the code iterates over each measurement in the measurements list. For each measurement, it constructs an SQL insert statement into the targeted table view. The values for the insert statement are obtained by unpacking the measurement dictionary. Suppose a record with a matching key already exists in the database. In that case, the SQLAlchemy *on_duplicate_key_update*() method is called on the statement, which creates an update statement using the same measurement data to update the existing record. The method is designed to work with databases that support this kind of operation natively like MySQL's *ON DUPLICATE KEY UPDATE* or PostgreSQL's *ON CONFLICT DO UPDATE*. When performing an insert operation and a row with the unique key already exists, instead of failing with a unique constraint violation, the database updates the existing row with the values you specify. The *session.execute*(*on_duplicate_key_stmt*) line executes the upsert command (insert or update) for each measurement. After processing all measurements, the database session is committed, ensuring that all changes are saved in the pre-aggregation table of the current job (one-hour or five-minute table)

Listing 3.5: Upsertion method

```
1 def upsert(session, interval: Interval, now_utc:
    datetime, time_interval: datetime, table_object, conf
    : dict) -> None:
2     try:
3         # Select the measurements in the measure table
    in interval
4         measurements = select_measure_table(
5             session, interval, now_utc, time_interval,
    conf
```

```
6           )
7           if ((measurements == []) | (measurements is None
    )):
8               logging.info(f"No measurements  from {
    now_utc} to {time_interval}")
9               return
10          timestamps = [data['timestamp'] for data in
    measurements]
11          minimal_timestamp = min(timestamps)
12          maximal_timestamp = max(timestamps)
13          logging.info(
14              f"Minimal time: {minimal_timestamp}, Maximal
     time: {maximal_timestamp}")
15          for measurement in measurements:
16              # unpack each dict row and insert or update
    inside the table
17              stmt = insert(table_object).values(**
    measurement)
18              on_duplicate_key_stmt = stmt.
    on_duplicate_key_update(**measurement)
19
20              session.execute(on_duplicate_key_stmt)
21          session.commit()
22          logging.info(
23              f'Inserted or updated: {len(measurements)}
    records.')
24      except Exception as e:
25          session.rollback()
26          logging.error(f"An error occurred during the
    upsert process: {str(e)}")
27          raise
```

Comfort aggregations

For each board corresponding to a location, it is necessary to compute the comfort levels. The boards for the selected project, as specified in the configuration file, are chosen. The management of projects and experiments will be explained later in this work.

In some of these experiments or projects, the scheduler is required to compute the thermal comfort for each aggregation interval. After upserting and committing the new measurement statistics into five-minute or one-hour aggregation tables,

the algorithm performs the thermal comfort computation. The algorithm theory is presented in section 2.2.3. To obtain the running mean, the function checks if the running mean has already been calculated for each day from the last update day until today. If not, it performs the **aggregating_external_date()** method described above for each day. With the running mean, it is possible to obtain the boundaries of the optimal comfort zone for each day. These boundaries are stored in a dictionary where the key is the day, and the values are the boundaries (max, min). For this work, it was decided that the work would implement category one interval. The following code performs this calculation according to the desired category and returns a tuple where the first value represents the upper limit and the second one is the lower limit. The code 3.6 represents this computation.

Listing 3.6: Comfort boundaries computation method

```
def get_boundaries(rm: int, category: int = 1) -> tuple:
    if category == 1:
        return (rm*0.33 + 18.8 + 2, rm*0.33 + 18.8 - 3)
    elif category == 2:
        return (rm*0.33 + 18.8 + 3, rm*0.33 + 18.8 - 4)
    elif category == 3:
        return (rm*0.33 + 18.8 + '', rm*0.33 + 18.8 - 5)
```

As the boards don't provide real thermal comfort sensors for every project, it can be useful to create virtual sensors to store the thermal comfort data. At each job, the method tries to retrieve the thermal comfort sensor ID, if this does not exist, it creates a new virtual sensor for this index. The last step is to count the number of measurements inside the boundaries and divide it by the total number of temperature measurements inside the aggregation interval (five minutes or one hour). SQLAlchemy offers the possibility to create subqueries before executing it. Therefore it is possible to create 2 subqueries, the first one that counts the measurements within the boundaries and the second one that counts the total number of temperature measurements. Then, the query retrieves the time intervals and calculates the ratio of the number of sensor readings within the boundaries to the total number of readings. This ratio asses the thermal comfort. The calculation uses a case statement, ensuring robustness by handling special cases: it returns zero when the total count is zero to avoid division by zero errors, and also when the count is within boundaries is null, it addresses the data absence issue. For instance, if there are no values inside the boundaries it returns zero instead of a None value. The use of outerjoin in the query ensures that all intervals from the total count are included, even if they don't have a corresponding count within boundaries, thereby ensuring completeness in the data set. The final result of this query is a comprehensive list of time intervals alongside their respective thermal comfort ratios, essential for subsequent data processing and analysis. Finally, the ratios are

upserted in the same way that the other measurements as presented above. The code 3.7 shows the ratio computation for the comfort indexes.

Listing 3.7: Ratios computation method

```
# count in the interval the number of sensors inside the measure
count_within_boundaries = (
    session.query(
        Measure.sensorId,
        timestamp_interval,
        func.count(Measure.data).label('count_within'))
    .join(LogicalSensor, LogicalSensor.sensorId == Measure.sensorId)
    .filter(
        LogicalSensor.boardId == board_id,
        LogicalSensor.unitId == TEMP_UNIT_ID,
        Measure.data.between(boundaries[1], boundaries[0]),
        Measure.timestamp.between(time_interval, now_utc)
    )
    .group_by(Measure.sensorId, timestamp_interval)
    ).subquery()

total_count = (
    session.query(
        Measure.sensorId,
        timestamp_interval,
        func.count(Measure.data).label('total_count'))
    .join(LogicalSensor, LogicalSensor.sensorId == Measure.sensorId)
    .filter(
        LogicalSensor.boardId == board_id,
        LogicalSensor.unitId == TEMP_UNIT_ID,
        Measure.timestamp.between(time_interval, now_utc))
    .group_by(timestamp_interval, Measure.sensorId))
    .subquery()

# Compute the ratio and insert it into the aggregation table
```

```
ratios = session.query(
    total_count.c.timestamp_interval,
    case(
            (total_count.c.total_count == 0, 0),
            (count_within_boundaries.c.count_within ==
None, 0),
             else_=(count_within_boundaries.c.
count_within / total_count.c.total_count))
    .label('ratio'))
    .outerjoin( count_within_boundaries,
                total_count.c.timestamp_interval ==
count_within_boundaries.c.timestamp_interval)
    .all()
```

The methodology is the same for the acoustic and visual indexes. Instead of checking the boundaries, it compares them to the desired thresholds, which are expressed, respectively, as a sound pressure and illuminance value.

The Indoor Air Quality (IAQ) index is different because it is the average of the ratios of measurements complying with the threshold for each air component (TVOC, PM2.5, PM10, CO2, NO2, CO, and CH2O). For each of these elements, the methodology is the same as for the thermal comfort measurement. However, if any measurement is missing during the interval, then the IAQ will not be calculated for that interval.

After upserting and committing each comfort calculation inside the desired pre-aggregation table, the IEQ is calculated for each board. As with the other indexes, it is checked if the four comforts are present within each interval from now until the last update of the pre-aggregation table, as it is the average of the four comforts. As with IAQ, if one comfort is missing, the IEQ is not calculated for that interval.

The Figure 3.6 represents the main steps of the indoor measurements aggregation.

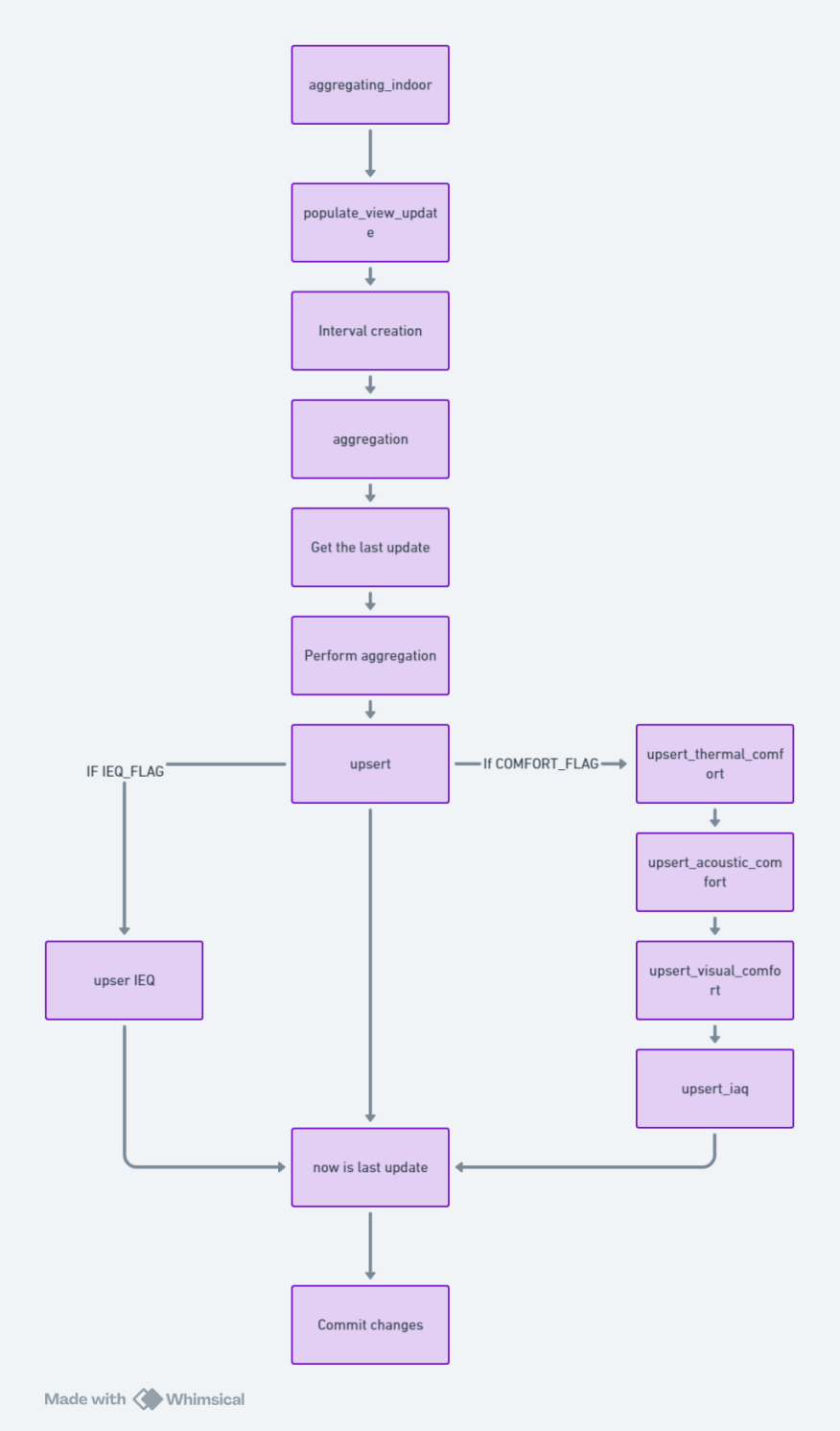

Figure 3.6: Indoor Aggregation

Chapter 4

The API

An API (Application Programming Interface) is a set of rules and protocols that allows different software applications to communicate with one another. APIs work with various models, and for this project, it has been decided to build a REST API (Representational State Transfer API). This type of API enables the client to request data from the server. It defines a set of methods such as GET, PUT, DELETE, and POST that the client can use to access the data. REST API offers many advantages, including easy integration, quick deployment of new services, easy access by many systems, and easy maintenance. This chapter explores the architectural design and application of an API to connect the database with the client side, or frontend, of the software solution.

4.1 Requirements

The API should enable the client to request data from the server to aggregate and visualize various statistics over different time frames, from any specified point in the past until the present instant. It should facilitate the analysis and understanding of trends and patterns in time-series data across several metrics. This API is designed to compute and deliver the average, standard deviation, minimum, maximum, 10th percentile, and 90th percentile of the stored measurements. These calculations are performed over predefined intervals, including real-time, 3 hours, 24 hours, 3 days, 1 week, and 1 month from the current date back to a specified date in the past. These aggregations are performed on the Measure table, filled directly by the sensor, for real-time, and also on the FiveMinAvgMeasure and OneHourAvgMeasure view tables for the other aggregation intervals.

An essential feature of the API is its integration with Grafana, an open-source platform for analytics and monitoring. Grafana is chosen for its performance in creating dynamic, interactive dashboards that visually represent the aggregated

data. The API must ensure compatibility with Grafana's specifications to allow users to visualize metrics over time.

Moreover, the API addresses the need for working days management in intervals of more than 24 hours. It should calculate the statistics over such intervals according to a calendar that should be dynamically generated.

The API should be accessible to an endpoint specifying the ID of the board and the name of the desired measure (temperature, humidity, etc.). The last mandatory parameter should be the aggregation interval desired, *avg* (rt, 3h, 24h, 3 days, 1 week, and 1 month). It also provides one optional parameter, which is the duration of time in the past up to which it is necessary to analyze the points *ts*. Thus, the endpoint should look like:

url/{boardId}/{measureName}?avg={}&ts={}.

To ensure good user experience and consistency, the implementation of both decorators and error-handling mechanisms is necessary for this work. Decorators can be used to validate input parameters before a function executes. For instance, it ensures that parameters like *avg* and *ts* are compatible and within acceptable ranges. This means checking that the *avg* is not greater than the *ts* value, which would be logically inconsistent. Additionally, decorators can confirm that the measureName and boardId exist and are valid. Error handling is important to manage and respond to issues such as nonexistent resources (e.g., measureName or boardId), invalid time ranges, or unsupported aggregation intervals. By capturing these errors and providing meaningful feedback to the client, the API not only becomes more robust but also easier to debug and use.

4.2 Architecture

4.2.1 Flask API

To build this API, it has been decided to use Flask, a WSGI (Web Server Gateway Interface) framework in Python. WSGI offers a simple and universal interface between web servers and web applications or frameworks. According to the Flask documentation, it should be described as a microframework because its core functionalities are simple and focus on the essentials of web development: request handling, routing, and response generation [11].

Flexibility and simplicity are the two main characteristics of this framework. Flask gives developers complete control over the application, from request handling to response formatting. The components and behavior are customizable, allowing Flask to be adapted to projects of varying complexities and configurations.

The core functionalities are:

- **Routing and URL Mapping**: the utilization of decorators for URL routing represents a pivotal aspect of Flask's architecture. It enables the creation of intuitive and highly customizable URL schemes, a requirement for the development of RESTful applications.
- **Template Engine Integration**: Jinja2 template engine significantly enhances its framework by enabling dynamic content generation. It allows developers to separate application logic from the presentation layer for cleaner codebases.
- **Development Server and Interactive Debugger**: Flask includes a built-in development server and an interactive debugger. The interactive debugger, accessible directly within the web browser, offers detailed error logs and robust debugging tools.
- **Session Management and Security**: Flask ensures secure session management through secure cookies. It offers a personalized user experience and also ensures security features like secure cookie handling for building safe web applications.
- **Extensibility through Extensions**: The framework's simplicity is complemented by its extensibility. It presents a wide range of Flask extensions available that allow for easy addition of functionalities like ORM integration, form validation, and authentication.

4.2.2 Dependency Injection technique

For this work, the API was designed using a specific design pattern: the dependency injection technique. It aims to separate the usage of an object from its creation. In the context of API programming, it is the decoupling of the instantiation of a class's dependencies from its behavior. This technique offers modularity and easier maintainability. Moreover, this kind of design pattern achieves Inversion of Control (IoC). IoC gives control of a program's flow to a framework or library instead of having the program control its flow.

To manage the object creation and its dependencies, "containers" are often implemented. They can hold configurations, services, factories, or other containers. Python offers a package called **dependency_injector** that facilitates the implementation of the Dependency Injection design pattern. In the context of a Flask API, a container class (**Container**) is created, extending from **containers.DeclarativeContainer**, a module from **dependency_injector**.

According to this technique, the API architecture should have:

- a main directory: contains the API run and configuration file

- a subfolder called "API": contains 3 subfolders: decorator, routes that instantiate endpoint objects, services that implement endpoint functionalities, and the container file.

.

The class owns a wiring_config attribute that specifies which packages have their dependencies automatically injected. In this work, it wires api.routes and api.services; these two folders are inside a main directory API. The first contains the endpoint instantiation, and the second, the query and functionality of each endpoint. When objects in these packages are instantiated, the dependency injector will provide them with the necessary dependencies. An instance of the Container class is created and attached to the Flask app object. This setup enables dependencies to be injected into routes and services as configured in the container. Utilizing the dependency injection design pattern through the **dependency_injector** package in a Flask API offers significant advantages, including enhanced modularity and maintainability of the codebase. Decoupling the creation of objects from their use simplifies the management of dependencies, promotes cleaner code, and facilitates easier testing and future modifications.

4.2.3 Marshmallow

Marshmallow is a library in Python that provides utilities for converting complex data types. This package allows to define schemas. They are the core of Marshmallow which allows the definition of the structure of the data.

It offers many features for data conversion. Marshmallow automates the conversion process, reducing the code complexity and potential errors. It also owns validation mechanisms that ensure the integrity and reliability of the incoming data into the system. Furthermore, serialization logic enables the system to handle specialized data formats and structures efficiently. The last advantage is that this efficient data handling minimizes delays, facilitating real-time data analysis and response.

The schemas are organized within a folder named **schemas** inside **API** main folder, enhancing modularity. Each schema is represented as a Python class and inherits from the superclass Schema provided by Marshmallow. It is a base schema class that facilitates serialization and deserialization. It includes field declaration according to specific types (String, Integer, DateTime) as serialization, deserialization, or validation methods to handle complex data types like ORM models.

Figure 4.1 shows the structure of the code using the Dependency Injection pattern and Marshmallow.

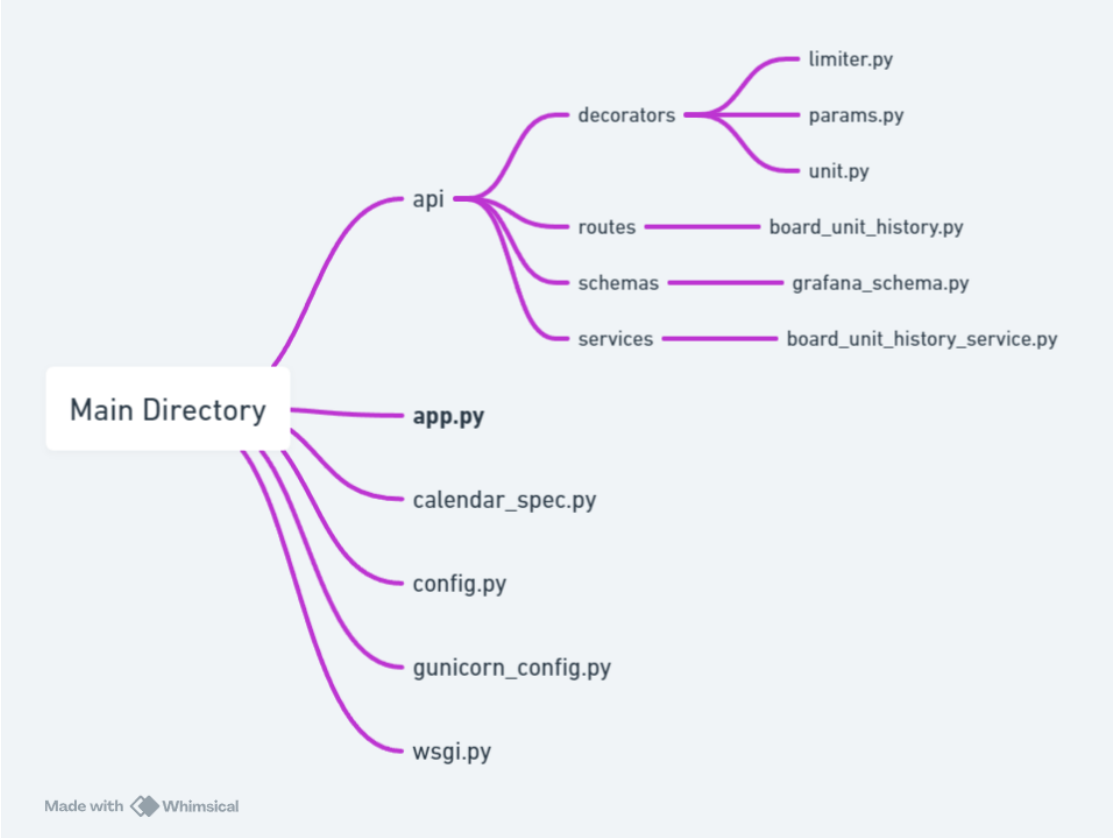

Figure 4.1: API Architecture

4.2.4 Other extensions

The API uses other extensions other than Dependency Injector, Flask, and Marshmallow. These extensions enhance the functionality of the Flask application in various ways and are stored outside the core directory API:

- **Flask-CORS**: CORS, an acronym for Cross-Origin Resource Sharing, serves as a mechanism enabling servers to authorize browsers to execute requests from specific origins. When the frontend provisioning is hosted on a distinct domain, browsers may block requests attempting to fetch data from the backend. Flask-CORS, an extension for Flask, facilitates the incorporation of CORS support into Flask applications. This is crucial for allowing various projects to access the API.
- **Flask-Limiter** adds rate-limiting capabilities to Flask routes, thereby restricting the number of requests a client can submit to the API within a specified period. This functionality is essential for preventing misuse and guaranteeing

that the application remains stable and responsive, even under heavy request volumes.

- **Flask-Smorest** is an extension that facilitates the construction of REST APIs using Flask by simplifying the management of endpoints, as well as input and output data, and automating documentation. This streamlines the development process. A notable feature of Flask-Smorest is its integration with OpenAPI (previously known as Swagger), enhancing API documentation. Additionally, it employs Marshmallow for data serialization and validation, contributing to the development of more efficient, robust, and user-friendly APIs.
- **APISpec** is a Python library designed to generate OpenAPI specifications, enabling users to comprehend the functionalities and services offered by APIs without requiring access to the source code.

4.3 Implementation

4.3.1 Application entrypoint

This API is centralized through the **app.py** module. It initiates the application instance and integrates essential components such as Cross-Origin Resource Sharing (CORS), SQLAlchemy, and API routing.

It configures the application to connect to the database using credentials and parameters extracted from environment variables. Like the scheduler, the API connects to the MySQL database. Instead of creating an entire database, a MySQL connector URL is created to instantiate a SQLAlchemy object. The Flask app object stores this URL in its configuration key "SQLALCHEMY_DATABASE_URI". This is a configuration key used by Flask applications to determine the database connection information. The URI (Uniform Resource Identifier) specifies the database dialect and connection arguments needed by SQLAlchemy to establish a connection with the database.

Secondly, it sets the extensions discussed in the previous section. It defines the APISpec documentation parameters as the title, the version, and the URL. It also includes the Marshmallow object and the limiter in the API object.

Finally, the application can be run with configurable server port settings, allowing for flexibility in deployment environments. The use of command-line arguments and environment variables for configuration is allowed.

To facilitate the production deployment of the Python API Application, Gunicorn has been used. It is a Python Web Server Gateway Interface (WSGI) HTTP server. Gunicorn is used as the webserver to run the Flask application. The file **gunicorn_config.py** stores the configuration variables to run the API.

Listing 4.1: Gunicorn configuration file

```
import os

default_port = "5003"  # Your default port
bind = "0.0.0.0:" + os.environ.get('API_PORT',
    default_port)
workers = 1
threads = 1
timeout = 120
```

The Gunicorn configuration file variables in 4.1 means the following:

- **bind** is the server's socket address, set to "0.0.0.0" to make the app accessible on all network interfaces, with the port taken from the API_PORT environment variable or default if not specified.
- **workers** is the number of worker processes created to handle requests concurrently. This functionality optimizes the app's ability to manage high traffic where the ideal number depends on the application's demands and the server's capabilities.
- **threads** is the number of threads per worker, allowing each worker to handle multiple requests simultaneously. It is especially useful for IO-bound applications.
- **timeout** is the maximum time, in seconds, a worker is allowed to spend on a request before being restarted, preventing indefinitely hung processes and ensuring resources are efficiently used.

To use Gunicorn, the application needs to be executed through the command line 4.2:

Listing 4.2: Gunicorn command line

```
    gunicorn --config gunicorn_config.py wsgi:app
```

It tells Gunicorn to use the specified configuration to run the application object (app) located in the wsgi module. The wsgi.py looks like 4.3:

Listing 4.3: WSGI file

```
    from app import create_app
    app = create_app()
    app.config["DEBUG"] = False
```

4.3.2 Working hours calendar

One of the requirements is to consider only the working day and hours for some aggregation intervals. The calendar to manage the 24h, 3d, 1W, and 1M aggregations intervals has been developed dynamically. An automated calendar generation module is executed each time the API is started in the server.

The main function, **get_working_schedule**, operates the generation loading the parameters from a JSON file as shown in 4.4. In this file, stored in the server, each year is represented as a key in the JSON object. A year is structured using a set of configurations. The year field is an integer specifying the year. The standard_hours field is an array with two integers. The first value represents the starting hour of the working day and the second value the last working hour of the same day. The third parameter, working_days, is an array containing a string that informs which are the working days of the week. The exception field outlines deviations from the standard work schedule, where each key represents a date in the dd/mm format. The associated value is either null, signifying a day off as a holiday. It can also be an array providing altered working hours for this day.

Listing 4.4: JSON file to generate the calendar

```
{
    "2024": {
        "year":2024,
        "standard_hours": [9,19],
        "working_days":["Monday", "Tuesday", "Wednesday", "Thursday", "Friday"],
        "exception": {
            "01/01": null,
            "25/12": null,
            "31/03":null,
            "26/12": [10, 15]}
    }
}

```

The function reads the parameter of the JSON file and iterates over each key, or each year. It converts the names of the working days into integers from 0 to 6 (Monday to Sunday) that are used by Python's **datetime** library. Then, from the first day of the year (01/01) until the last day (31/12), the function checks whether the day is working, an exception (with custom hours or non-working status), or a regular non-working day (weekend or not listed as a working day).

Once the year's schedule has been generated, the function prepares to save the data to an output JSON file. It constructs a file path using the current script's

directory and the provided output file name. Finally, it writes the generated schedule to this file in a formatted JSON structure.

4.3.3 Endpoints structure

Routes

A route defines the paths and methods to be executed within an API. To organize these routes and enable scalability, Flask API can use Blueprints. It helps organize the endpoints and the API's code into subdirectories.

The API code is inside the main directory that provides the app.py file, already discussed above. In the app.py file, endpoints are registered through a specific function **api.register_blueprint(file_path, url_prefix)**. This method uses two parameters: the subdirectory file path and the url prefix.

At the same level as app.py there is the api directory, that contains the subdirectory "routes". In this subdirectory, a file **board_unit_history.py** has been created. It is the file where the Blueprints are declared, as shown by 4.5. A Flask blueprint named board_unit_ts_avg is created to group routes related to board data under the namespace 'Board'. Thus, the first parameter of **api.register_blueprint** (in app.py) is "routes.board_unit_ts_avg" (from the folder api.routes).

Listing 4.5: Blueprint declaration

```
1 board_unit_ts_avg = Blueprint('Board', __name__)
```

Inside **board_unit_history.py**, the endpoint is defined using the "@board_unit_ts_avg.route" decorator as in the following code 4.6. It maps the endpoint URLs that follow the prefix parameter of **api.register_blueprint** to this Python callback function. Each callback function returns a response, which Flask converts into an HTTP message to be sent to the API client.

Listing 4.6: Route decorator

```
1 {
2 @board_unit_ts_avg.route('<int:boardId>/unit/<unit_name>
    ', methods=['GET'])
3 def get_unit_board_avg_ts(boardId, unit_name):
4     ...
```

The main endpoint implemented is the one that gets statistics according to interval groups over a specific time frame for a specific unit within a board. This route listens for GET requests at URLs matching the pattern below. The url_prefix parameter in the **api.register_blueprint** is set to "/api/board/". For the user, the endpoint is accessible through a URL ending with 4.7.

Listing 4.7: Endpoint URL

```
/api/board/<int:boardId>/unit/<unit_name>
```

The advantages of using Blueprints are essential for this project, where different projects may require different endpoints. Blueprints enable a distributed development model by allowing developers to focus on their specific areas of the application without interfering with one another. Moreover, this modular structure supports the principle of separation of concerns, where each module is responsible for a distinct feature or functionality, thereby reducing code complexity and improving maintainability.

Decorators

When a request is made by the user to an endpoint, it is preferable to validate his query before processing the request. In Python and Flask, it is possible to implement decorators. A decorator is a function that wraps another function or method. It enables the preprocessing of the parameters before the wrapped function. In the context of this work, to query the database, the user should provide the BoardId, the measure name, the average group interval, and the time range in the past parameters. To ensure that the user provides well-formatted and existing parameters and does not perform too many requests, decorators have been implemented.

Flask-Limiter offers an object "Limiter" that ensures a maximum number of requests by a client within a given timeframe. The decorator *conditional_limiter* (4.8) is calling this object. By adjusting an environmental variable 'Limiter', it is possible to enable, disable, or change the rate limit for different deployments without changing the code. As shown in the code below, the limiter object sets a limit_value per minute that should raise an error and not serve the request in case it has not been respected. If no limit_value is provided (i.e., the LIMITER variable is unset or None), the function is returned unmodified. This effectively disables rate limiting for that function, allowing unlimited requests.

Listing 4.8: Request limiter

```
def conditional_limiter(f):
    limit_value = config("LIMITER", default=None)
    if limit_value:
        return limiter.limit(f'{limit_value} per minute'
   )(f)
    return f
```

Validating *avg*, which represents the aggregation time interval, is the first parameter checked. The decorator *check_avg_param* looks if the requested parameter

for *avg* is one of rt, 3h, 24h, 3d, 1w, 1M. If it doesn't exist, it returns an error 1007 message with all the possible values.

Secondly, **check_unit_name** is called to look for the measure name requested. It queries the database for the available units in the *UnitOfMeasure* table. If the name written by the user doesn't exist, it returns a 400 message with the existing units. It utilizes the method shown in 4.9

Listing 4.9: Query the units name in the database

```
def get_all_units():
    global _units_cache
    if _units_cache is None:
        _units_cache = [unit[0] for unit in db.session.
    query(
            UnitOfMeasure.measureName).all()]
    return _units_cache
```

Finally, **check_ts_param** decorator ensures the temporal coherence and manageability of data queries. The decorator recognizes specific temporal units for seconds ("s"), minutes ('m'), hours ('h'), days('d'), weeks('w'), months('M'), and years ('y'). The first level of validation involves checking if the requested ts parameter aligns with these units, otherwise, it returns an error message with all available values. Another aspect is the coherence of the request between avg and ts. If a request specifies a ts value that is less than avg (e.g., avg = 24h and ts = 5h), the decorator intercepts this as an incoherent request. In such cases, it returns an error message indicating that ts should be equal to or greater than *avg*, thereby preventing inefficient or illogical data queries. The second one is optional according to the configuration option when deploying the API. If the user sets the environmental variable NB_SAMPLE, it limits the volume of data requested. This configurational aspect is particularly relevant when dealing with requests for extensive periods, such as real-time data over the last year, which could potentially involve gigabytes of information. By enabling this functionality, the system gains the ability to moderate its data throughput, balancing between user needs and operational constraints.

The endpoint header with theses decorators looks like 4.10:

Listing 4.10: Decorator call

```
@board_unit_ts_avg.route('<int:boardId>/unit/<unit_name>
    ', methods=['GET'])
@conditional_limiter
@check_avg_param
@check_unit_name(only_validate=False)
@check_ts_param
```

```
6 def get_unit_board_avg_ts(boardId, unit_name):
```

After being validated by the decorators, the endpoint can execute the callback function **get_unit_board_avg_ts(boardId, unit_name)**. The first step is to retrieve the query parameters that are avg, and ts, by using the Flask method request.args.get(params). Then, the endpoint executes its service functionality stored in a services folder.

Services

The services for the two endpoints are stored in a folder named **services**, as seen before. The **BoardUnitHistory** class implements the core functionalities of each endpoint contained in the routes.

get_avg_ts extracts time-aggregated sensor measurements from the database, applying specific conditions like working hours and experimental parameters. It ingests parameters defining the aggregation interval (avg), the historical timestamp duration (ts), the board ID (boardId) from which to retrieve data, the unit of measurement (unit_name), and optionally the name of an experiment (experiment). It also takes as input the SQLAlchemy session and the unit name desired by the client. These parameters allow endpoints to tailor queries to precise analytical requirements according to the project.

Initially, the method reads the JSON file containing working hours (working_hours_dict) to incorporate working hours constraints into data aggregation. It then determines the start and end times for data retrieval based on avg, ts, and working hours, selecting the appropriate database table and constructing a SQL expression for interval aggregation.

The **setup_time_and_table** function serves as an essential preliminary step. To begin with, it calculates the start time and end time according to the parameters. If the ts parameter is provided, the start time will be the subtraction of this parameter from the present date. Otherwise, it is the date corresponding to the difference of the present date with the avg parameter value. Furthermore, if the avg parameter corresponds to 24h or bigger intervals, it also checks if the end time is correct. If the end time falls on a non-working day, it will take the most recent working day and hour as the end time. To set which table to use between the Measure, FiveMinAvgMeasure, or OneHourAvgMeasure table views developed in the scheduler part, this choice is done according to the experiment or project that is running the service, the value of the avg parameter, and also the measurement name. If the user asks for statistics about a comfort index, it will be the FiveMinAvgMeasure or OneHourAvgMeasure table because it depends on the project and the avg parameter. The Table 4.1 below describes this selection.

The **setup_time_and_table** function also returns how the aggregation interval should be aligned according to the avg value to calculate the statistics. If

Measure vs avg	RT	3h	24h	3d	1w	1M
Measurements	Measure table	5min or RT	5min or RT	5min or RT	1h	1h
Comfort indexes	5min or 1h	5min or 1h	5min or 1h	5min or 1h	1h	1h

Table 4.1: Table selection according to measurement type and avg parameter

avg equals "rt" for real-time, the interval groups will coincide with measurement time (timestamp attribute of the selected table) directly without any processing on the interval. For 24h average group, the statistics should be calculated from 00:00 to 23:59 for each day desired. For instance, if today is 09/03 12h00, and the ts parameter was defined as "3d", then the statistics should be calculated for 09/03 from 00:00 to 12:00, for 08/03 from 00:00 to 23:59, and so on. The other average group intervals are aligned to the present datetime and the interval size. Thus, if today is 09/03 12h00, avg parameter is set to 3h and ts to 9h, it calculates the statistics between 9:00 and 12:00, 6:00 and 9:00, and finally, between 3:00 and 6:00. This interval is computed using the following code (4.11):

Listing 4.11: Interval Category

```
1 def interval_category(category: int, table, avg_duration
    = None, end_time = None):
2     if category == 0:
3         return func.unix_timestamp(func.date(table.
  timestamp))
4     elif category == 1:
5         interval_seconds = avg_duration.total_seconds()
6         time_difference = end_time.timestamp() - func.
  unix_timestamp(table.timestamp)
7         next_interval_offset = time_difference %
  interval_seconds
8         return func.unix_timestamp(table.timestamp) +
  next_interval_offset
```

With the preliminary setup, the script constructs an SQLAlchemy query (measure_query) by joining relevant tables and applying filters based on the input parameters, working hours, and experimental conditions. This process involves the **common_query** (4.12) function to establish the basic query structure. It gets the values inside the selected table and if they belong to an interval group (*interval_expr*) as previously defined.

Listing 4.12: Common query

```
def common_query(session, interval_expr, table,
    start_time, end_time, boardId, unit_name):
     return session.query(
                 interval_expr.label('interval_start'),
                 table.avg if table != Measure else table
    .data
             ).join(LogicalSensor, LogicalSensor.sensorId
     == table.sensorId)\
             .join(UnitOfMeasure, LogicalSensor.unitId ==
     UnitOfMeasure.unitId)\
             .join(BoardExperiment, BoardExperiment.
    boardId == LogicalSensor.boardId)\
             .join(Experiment, BoardExperiment.
    experimentId == Experiment.experimentId)\
             .filter(
                 and_(
                     table.timestamp >= start_time,
                     table.timestamp <= end_time,
                     LogicalSensor.boardId == boardId,
                     UnitOfMeasure.measureName ==
    unit_name
                 )
             )
```

The **apply_additional_filters** function is used to refine the **common_query** with additional constraints. Firstly, this filter applies the working day constraint using a specific SQLAlchemy filter generated by the method shown in 4.13. For all the dates inside the query, it creates an "and" clause to check if the date is inside the working hours. Finally, **the apply_additional_filters** checks the experiment name to filter according to the experiment or project name.

Listing 4.13: Filter method of the working hours

```
def is_inside_working_hours_sql_expression(timestamp,
    working_hours)-> Union[and_, or_]:
     work_intervals = []
     for key in working_hours:
         for date, hours in working_hours[key].items():
             if hours:
                 day_time, month, year = date.split('/')
                 start_time = f"{year}-{month}-{day_time}
     {hours[0]:02}:00:00"
```

```
8                 end_time = f"{year}-{month}-{day_time} {
    hours[1]:02}:00:00"
9                 work_intervals.append((start_time,
    end_time))
10     time_filters = [and_(timestamp >= start, timestamp
    <= end) for start, end in work_intervals]
11     return or_(*time_filters)
```

After executing the query, the method processes the retrieved data through **calculate_stats** (4.14). For this work, it is required to perform a statistical summary for each time interval: percentiles, averages, standard deviations, minimum, and maximum. for each interval. Unfortunately, SQLAclchmey is not able to calculate the percentiles directly using ORM. That is why, instead of performing the averages and standard deviations and then manually the percentiles, it has been decided to use only one query (**common_query** above) and perform all statistics using native Python functionalities. The algorithm starts by creating a dictionary where each key represents a point of the interval (12:00, 9:00, 6:00, 3:00 for example) and the value is the list of data queried previously. Finally, it performs the statistics for each list of each interval key.

Listing 4.14: Statistic computation

```
1 def calculate_stats(percentiles_results,avg):
2     grouped_results = {}
3     for interval_start, avg_value in percentiles_results
    :
4         if interval_start not in grouped_results:
5             grouped_results[interval_start] = []
6         grouped_results[interval_start].append(avg_value
    )
7     final_results = {}
8     for interval_start, values in grouped_results.items
    ():
9         if interval_start not in final_results:
10            avg = sum(values)/len(values)
11            if avg != "rt":
12                n = len(values)
13                percentile_10th = values[int(n * 0.1)]
14                percentile_90th = values[int(n * 0.9)]
15                std = (sum((x - avg) ** 2 for x in
    values) / len(values))**0.5
16                mini = min(values)
17                maxi = max(values)
```

```
18                 final_results[interval_start] = (
    percentile_10th, percentile_90th, avg, std, mini,
    maxi)
19             else:
20                 final_results[interval_start] = (None,
    None, avg, None, None, None)
21     return final_results
```

The function **format_results** formats the date to get a rounded interval to the second, and the comfort indexes to be from 0-100 (%) instead of a value between 0 and 1. Then, it structures a list of dictionaries, each representing an aggregated data point with associated statistics such as average, standard deviation, min/max values, and percentiles. Finally, the **get_avg_ts** returns this formatted result.

This result is stored in the routes endpoints file where the blueprint is registered and stored in a dictionary (4.15).

Listing 4.15: Result dictionary

```
1 response_data = {
2             'target': unit_name,
3             'data': avg_data
4         }
```

Schemas

One of the requirements of the API is that the data should be readable by Grafana to visualize time-series data. One solution is to use the JSON API Plugin [12] provided by Grafana. It integrates JSON data into Grafana. However, the response message should be in a pre-defined format. The document should have a key list "datapoints," and inside this list, an object that represents a point of the graph with the following characteristics: "average," "date" in UNIX timestamp format, "max," "min," "ninth," "tenth," and "unit" (the unit name).

As described in the preceding section, Marshmallow was used to define a specific schema. To ensure differentiation between real-time requests (without statistics) and time interval-based ones (with statistics), two classes were implemented: GrafanaResponseSchema (shown in 4.16) and GrafanaResponseSchemaRT. They utilize Marshmallow's **fields.Method** attribute type to specify that the datapoints attribute will be filled by calling a custom method, get_datapoints. It takes an object ('obj') as a parameter, expected to be a dictionary with at least a "data" key and a "target" key. It first checks if the "data" key is None, a string, or a dictionary, returning an empty list in any of these cases, indicating that there is no data to process. If there is valid data, it iterates over each item in obj["data"] and constructs

a new dictionary for each item. For GrafanaResponseSchema, this dictionary includes several statistical measures (average, std, max, min, 10th_percentile, 90th_percentile) and the timestamp of the data point, converted to milliseconds since the epoch (a common format for time series data in JavaScript and thus Grafana). It also includes the "unit" from obj["target"]. On the other hand, for real-time monitoring, it explicitly sets min, max, tenth, and ninth to None, indicating these statistical measures are not provided in the real-time scenario.

Listing 4.16: Grafana schema transformation

```
class GrafanaResponseSchema(Schema):
    datapoints = fields.Method("get_datapoints")

    def get_datapoints(self, obj):
        if obj["data"] is None or obj["data"] is str or obj["data"] is dict:
            return []
        return [
            {"average":item["average"], "std":item["std"], "max": item["max"],\
             "min":item["min"],"tenth":item["10th_percentile"],\
              "ninth":item["90th_percentile"],   "date": int(item["date"].timestamp() * 1000), "unit":obj["target"]}
            for item in obj["data"]
        ]
```

Thus, following the creation of the response_data dictionary in the routes (code 4.17), the dedicated schema is instantiated. The schema.dump(response_data) serializes the object into a Grafana-readable format. The response is created using Flask's make_response and jsonify to set the HTTP status code to 200 and ensure the response content type is application/json. **make_response** is used to create a response object that can be customized before being returned from a view function. This customization includes setting the status and headers.

Listing 4.17: Response format

```
    schema = GrafanaResponseSchema()
    formatted_response = schema.dump(response_data)
    response = make_response(jsonify(formatted_response), 200)
```

Chapter 5

Applications

This section explores the configurability and management for different projects, or experiments of this work.

5.1 MQTT Client

5.1.1 Definition

In the context of IoT monitoring, saving data coming from sensors is a challenge. The sensors send data to the server, and the application should be able to understand these messages. To achieve that, protocols have been developed; one of them is MQTT (Message Queuing Telemetry Transport). It is a application-level network protocol that uses a publish-subscribe methodology.

MQTT operates on the principle of publishing messages and subscribing to topics. When a device publishes a message to a topic, all clients that are subscribed to that topic receive the message. It ensures efficient communication between devices and a central server.

In the context of this work, the client is a software application that communicates with an MQTT broker. The broker receives all messages from the clients, filters them, and then publishes those messages to the subscribed clients. The broker acts as a central point that manages the distribution of messages to the correct recipients, making the MQTT network scalable and efficient.

5.1.2 Paho MQTT Python

The architecture of the client has already been developed in previous work, but it was necessary to create new instances of the client for the different experiments linked to this work.

A specific Python package, Paho MQTT [13], is used to implement the client. This library provides client classes for communicating with MQTT brokers using the MQTT protocol. It is an open-source client implementation and enables Python applications to connect to an MQTT broker, publish messages, and subscribe to topics to receive published messages. The main functionalities include:

- **Connect to MQTT broker**: Establish a connection over TCP/IP with optional SSL/TLS for security.
- **Publish Messages**: Send messages to the broker on specified topics.
- **Events**: Callbacks for handling specific events such as on_connect, on_message, on_disconnect, etc.
- **Asynchronous Operation**: Support for non-blocking operations through loop_start() and loop_stop() methods, allowing the client to manage its network connection in a separate thread.
- **User Authentication**: Support for username and password authentication with the MQTT broker.

5.1.3 Client Architecture

Client object

In the previous work, the class ClientMQTT was implemented. It inherits from the package base **paho.mqtt.client.Client** with specific configurations such as the client ID, session management preferences, and transport protocol (TCP in our case). A modification has been added to this class because the Paho package has been updated. The parameter callback_api_version has been added to the initialization.

The initialization defines also the security configuration by managing TLS encryption and username/password authentification for secure MQTT communication, ensuring encrypted data transfer and controlled access to the MQTT broker. Finally, it connects to the database.

Updaters

When this object is started by the services, it first defines an Updater. It is an object that inherits from an abstract class DBThread. The DBThread class is a Thread object from the threading package. It provides a way to perform concurrent execution of code.

The Thread class represents an activity that is run in a separate thread of control. This allows a program to carry out multiple tasks simultaneously, improving the efficiency of I/O-bound operations.

Thus, the DBThread class inherits from Thread to manage the database operations in separate threads of control. Each instance of DBThread or its subclasses (Updaters) can perform its database operations concurrently with the main program flow or other threads. It is essential to manage concurrency and maintain the responsiveness of the application. In the context of MQTT messages coming and multiple project management that interact simultaneously with the database, it ensures reliability and efficiency.

Each updater inherits from this class. Each subclass implements specific behaviors required to handle their types of data and operational contexts. They convert MQTT messages into database-friendly formats with **_data_conversion**, determine which MQTT topics to subscribe to with **_get_topics** and execute the insert operations into the database with the**_execute_insert** method. Each updater is dedicated to one broker that corresponds to one type of device network. It allows the software to load data coming from multiple sources in the same database.

Running configurations

When started, the Client object defines the updates according to the configuration of each broker. The function then initiates a connection to the MQTT broker asynchronously through **connect_async**. After establishing the connection, it subscribes to the determined topics and starts a loop **loop_forever** that continuously listens for messages. Moreover, it places the massage into a thread-safe queue (_msg_queue). Thus, the updater threads, effectively decouple message reception from processing.

5.1.4 Subjective Sensors

As explained in the introduction, one of the goals of the project is to allow occupants of rooms, and indoor environments to evaluate the comfort of the room by looking at the thermal, acoustic, air quality, and visual comfort. This work implemented the management of such requirements.

To manage these subjective metrics, the answers of the occupants are seen as sensors sending data to the database. The survey frontend sends the answer to a broker and then a specific Updater is defined.

The updater class is called UpdaterEurixSubjective. This object defines its topics. The topics format defined as **prometeo/{boardId}/questionnaire**. Therefore, the frontend of the questionnaire should send the data to this topic.

The **__get__topics** method dynamically generates the set of MQTT topics that the instance should subscribe to, based on database queries that filter specific conditions. The query is constructed to include boards linked to subjective units. These units are thermal_comfort_sbj, visual_comfort_sbj, acoustic_comfort_sbj, iaq_sbj, ieq_sbj ('_sbj' for subjective measurements). They are created and represented by a virtual LogicalSensorId inside the Board when populating the database with the device's information. It starts by creating a database session to execute a query, which selects distinct board IDs from a set of joined tables (Board, LogicalSensor, UnitOfMeasure, BoardExperiment, Experiment, and VendorModel). It further narrows down the selection based on specific experiments and vendor names if specified in the configuration. This results in a tailored list of board IDs related to the application's current experiment.

For each board ID retrieved, the method generates a topic string following the pattern **prometeo/boardId/questionnaire**. This pattern indicates that the topics are meant for collecting questionnaire responses (or similar subjective data) from various devices or boards participating in the experiment. The generated topics are stored in a dictionary, associating each board ID with its corresponding topic, ready for the MQTT client to subscribe to these topics and start receiving messages.

A user can use the software platform to answer a questionnaire and send his answers to the server. The frontend sends the data to the Mosquitto MQTT Broker in the following format 5.1.

Listing 5.1: Subjective measurements message format

```
prometeo/{boardId}/questionnaire -m {
"thermal_comfort_sbj"=value,
"visual_comfort_sbj"=value,
"acoustic_comfort_sbj"=value,
"iaq_sbj"=value,
"timestamp"=value
}
```

The information received by the broker needs to be transformed before it can be added to the database. The **__data__conversion** function is specifically designed to handle MQTT messages by turning the incoming JSON payload into a format that's ready for database entry. It starts by decoding the JSON payload to pull out the necessary data, excluding the 'timestamp' to change it into a Python datetime object, which ensures the timing information is precisely recorded. Next, the function analyzes the MQTT topic to pinpoint the exact board ID, assuming the topic follows a pattern like **prometeo/boardId/questionnaire**. Once the board ID is determined, it performs a database search to align measure names with sensor IDs for that board, concentrating on measures labeled as subjective ('_sbj'). For

each measure discovered in the JSON payload, if its name is found in the collected measure names dictionary (signifying it's an acknowledged subjective measure for that board), it adds a new Measure object to the measurements collection, filling it with the sensor ID, timestamp, and measure value. This method effectively turns the unprocessed MQTT message into a series of Measure objects that encapsulate both the sensor data and metadata, making them ready for additional analysis or storage, thus aiding in the incorporation of subjective measures from different sources into the system's data management practices.

After converting the data, the **__execute__insert** function becomes essential for saving the organized data in the database. It utilizes the **MeasureService.add__measurements__bulk** function, crafted to add numerous Measure objects to the database in one go, thus boosting efficiency and performance. This function takes a session and the data list, which holds the converted measurements. Before insertion, each Measure object within the data list is converted into a dictionary format through the **object__as__dict** utility function, to ensure it fits the database insertion process. The advantage of bulk insertion comes into play especially with large data sets, as it lessens the need for numerous database transactions, cutting down on overhead and likely hastening the data intake process. This function marks the conclusive phase of incorporating MQTT message data into the database of the system, tracing the data's path from its initial raw form to structured and stored data.

This is a specific Updater developed for this work, but the example also shows how a general updater can be configured. All of the updaters have the same structure but are customized for the broker and the type of data.

5.2 Docker

Docker [14] is a tool developed to improve the efficiency when deploying an application. It relies on the principles of containers, which encapsulate the code with all the dependencies such as libraries or packages. Each component can be run separately and configured via environmental variables. This offers flexibility, portability, and consistency.

5.2.1 DockerFiles

A Dockerfile is the main document when using Docker. It contains all the configuration variables to run the application. It facilitates the creation of application images following specific configuration parameters.

In the context of this work, we will have a dockerfile for the API, the MQTT client, the database, and the Scheduler. The general structure of these Dockerfiles

follows a common pattern: they start by specifying a base image, setting a working directory, copying requirements, projecting files into the container, installing dependencies, and finally, defining the command or entry point that will be executed when the container starts. This standardized approach ensures consistency and reproducibility across different development and deployment environments, facilitating easier management and scaling of the application components.

The Dockerfile for the database sets the foundation with **python:3.10.1-bullseye** as its base image, establishing a Python-centric environment for the database operations. Setting the **WORKDIR to /db-lib** and adjusting the **PYTHONPATH**, ensures that the custom database library can be easily imported from anywhere within the container. Dependencies are installed after copying the **requirements.txt** file, and the database library code is copied into the container, making it ready for use by other components that rely on database interactions.

The other Dockerfiles are similar to this last one. They start with **public/db-lib:latest** as the base image, indicating that it shares commonalities with the database library, such as base dependencies. After that, all of them install the required library from a requirements.txt file into a specific directory, /client for MQTT, /scheduler for the Scheduler and /api-connection for the API.

The MQTT client's Dockerfile involves installing specific requirements, creating a TLS directory for Transport Layer Security, and copying the client scripts into the container. The MQTT client is designed to run as a standalone module, with its command specified to execute the client scripts, facilitating communication over the MQTT protocol. The scheduler's Dockerfile sets up the environment to run the scheduler's main module. This setup demonstrates how the scheduler component is prepared to manage timed tasks or operations within the application ecosystem. Lastly, the Dockerfile for the API prepares the environment for the API connection operations. The use of Gunicorn as the entry point indicates that this container is designed to serve a web application or API, handling HTTP requests and routing them as needed. The inclusion of a **chmod** command to adjust permissions suggests that the container may interact with shared resources or files that require specific access rights.

5.2.2 Environmental Variables

The backend of the web application requires it to be highly configurable to use it for different projects. Each project has its properties and environmental variables that are very useful when deploying the services according to each project's specificities. They are outside the main program and are made up of a name/value pair.

Each service, the MQTT client, the API, and the scheduler have these variables. For all services, there are common environmental variables, mainly the ones linked

to the database management and the experiment/project destination (Table 5.1)

Parameter	Value
LOGGER	None
EXPERIMENT	Eurix
MYSQL_SERVER	mysql
MYSQL_USER	root
MYSQL_PASSWORD	password
MYSQL_DB	PROMETEO
MYSQL_PORT	3306
VENDOR_NAME	iSemar

Table 5.1: Configuration Parameters

The EXPERIMENT is the one to define the project's Board that should be filtered in each service (For example, when running the scheduler). The MY_SQL parameter enables the connection to the database. The VENDOR_NAME was a requirement for some projects to filter by specific board vendors. Finally, the LOGGER is defined to see which level of logs needs to be applied for this service (warning, None, all, debug, etc...)

MQTT Client

For each broker involved in the project needs a new MQTT client container should be created. The variables are here to define to which broker the client will connect to (URL address and port) 5.2, and also the behavior of this client (Delay, number of thread parallelized, or maximum interval for re-attempting connection to the broker (MAX_BRK_DELAY). It also defines which Updater to instantiate with the variable UPDATER_NAME.

Parameter	Value	Description
BROKER_ADDRESS	"test.mosquitto.org"	broker hostname on bridged network
BROKER_PORT	1883	broker port
CLEAN_SESSION	discard	
KEEPALIVE	60	Restart the application if failures
MAX_BRK_DELAY	60	
DB_DELAY	10	
NUM_THREADS	3	Number of threads for the updater
UPDATER_NAME	"updater_eurix"	

Table 5.2: MQTT Configuration Parameters

Scheduler

The scheduler has several configuration parameters (Table 5.3) that are useful for the optimal management of the database. The FIVE_MIN, ONE_HOUR, and DAY parameters define if the aggregations can be executed every five minutes, one hour, or each day. The threads parameters define how many jobs can be parallelized by the scheduler object. Coalesce, max_instances, replace_existing, UTC, and misfire_grace_time are the other Scheduler object parameters described in Chapter 4. The last parameters are about the external API to retrieve outdoor temperature for the thermal_comfort.

Parameter	**Value**
FIVE_MIN	True
ONE_HOUR	True
DAY	True
threads	20
coalesce	False
max_instances	10
replace_existing	True
misfire_grace_time	30
UTC	UTC
VENDOR_NAME	iSemar
API_KEY	**********
BASE_URL	`http://api.weatherapi.com/v1/history.json`
API_SENSOR_ID	1
API_BOARD_ID	1
API_VENDOR	externalAPI
API_MODEL	weatherAPI

Table 5.3: Scheduler and API Configuration Parameters

The other environmental variables of the scheduler are the values of the different thresholds (Table 5.4). COMFORT_FLAG and IEQ_FLAG define if the scheduler should compute the comfort itself.

Parameter	Value
LUX_LIMIT	500
ACOUSTIC_LIMIT	40
TVOC_LIMIT	500
PM2_5_LIMIT	25
PM10_LIMIT	50
CO2_LIMIT	800
NO2_LIMIT	20
CH2O_LIMIT	100
CO_LIMIT	10
COMFORT_FLAG	True
IEQ_FLAG	True

Table 5.4: Environmental Quality Thresholds

API

The following environmental variables define some requirements of the Flask API, which are the PORT and the HOST, which can be chosen by the user when deploying the API (Table 5.5). It also defines a parameter for the limiter, in case the developer wants to define a rate limiter to control how often a client can request the Flask API. If the limiter is not defined, it will consider no limit. Finally, the NB_SAMPLE parameter permits a maximum of data that can be queried by the user. As described in the precedent chapter, it enables control of the volume of data aggregated by the API.

Parameter	Value
LIMITER	200
NB_SAMPLE	1000
API_PORT	5002
API_HOST	0.0.0.0

Table 5.5: Configuration Parameters for API

5.2.3 Docker compose

Docker Compose stands out as a tool designed to streamline the handling of applications composed of multiple Docker containers. It enables the definition and execution of multi-container Docker applications through a YAML file, commonly named docker-compose.yml. This file acts as a blueprint for the application's services, detailing the containers, network configurations, volumes, and other necessary

dependencies in a clear, declarative manner. Docker Compose facilitates the orchestration of container operations, such as startup and shutdown, in a synchronized way, thereby simplifying the deployment of complex applications with just a handful of commands. To deploy the different services, there are two docker-compose file>: **docker-compose.scheduler.yml** and **docker-compose.api.yml**. For each service, they define the image, the volumes, the network, and the environmental variables.

Scheduler services

The first one is used to run the services required by the scheduler:

- MySQL: This service uses the official MySQL Docker image. It connects to the database
- PhPmyAdmin: It provides a web-based interface for managing the MySQL database, accessible on the host via port 8082.
- MQTT clients: They share a similar setup but are configured with different environment variables to connect to different MQTT brokers and perform distinct roles within the system. They are configured to connect to the MySQL service for database operations
- Scheduler: It is configured extensively via environment variables to manage scheduler-related operations.

API services

These are the different components defined by the docker-compose file of the API:

- The file defines a single service named web, which builds its image from the ./api-connection directory. This implies that the Dockerfile within this directory contains all the necessary instructions for creating the image required by the web service.
- The web service maps port 5002 of the container to port 5002 of the host, enabling external access to the API served by the web container. This port configuration is crucial for communication with the web service from outside the Docker network.

5.2.4 Configuration by projects

Eurix

The Eurix project uses similar boards to the ones of Promet&o but they are from commercial vendors. The sensors have different report times than Promet&o (table

2.1). No official table was provided for this work, but it has been noticed that for each sensor data are coming at least every 15 minutes. Moreover, some sensors for the air quality measurement are not provided (CO, NO2, and ch20).

To manage the report time rate, it has been decided to only run the one-hour job for the scheduler. The API queries only on the Measure table for the following *avg* values: rt, 3h, 24h, and 3d and on the OneHourAvgMeasure table for the 1w, and 1M values. Thus, there was no need to use the FiveMinAvgMeasure table for such a project. The comfort flags (COMFORT_FLAG and IEQ_FLAG) have been configured to "True" and are calculated by the scheduler itself. The IAQ is calculated for all values managed by the Eurix sensors.

Regarding the client, the topics follow the format: "aircare/{serialId}/notification" As for the subjective values, the **__get__topics** method retrieves the boards of this experiment (excluding the subjective ones) and registers the topics. The messages have a the following pattern:

```
1    aircare/{serialId}/notification -m {
2    "sender": serialNr
3    "measurements":{
4        "ts": datetime value,
5        "temperature":value,
6        "humidity":value,
7        etc...
8       }
9    }
```

Promet&o

The Promet&o project uses the boards developed by the research group. The sensors provide more measurements than the one in the previous experiment. As shown in Table 2.1, the devices have a greater rate of measurement and more types of measures like CO, NO2, and ch20. Moreover, the sensors calculate the values for the visual, acoustic, and IAQ indexes. Therefore, the COMFORT_FLAG and IEQ_FLAG are set to False for this experiment. The client Updater is also specific for this experiment because the broker can send two kinds of messages. The first type involves statistical data for physical quantities, such as temperature, humidity, and noise level. Each message in this category includes comprehensive statistical information like mean, mode, median, maximum, minimum, standard deviation, and percentiles, uniquely identified by a sensor identifier and a specific physical quantity. These messages are sent under topics that follow the pattern ms<id>/report/<measureName>. Therefore, according to this type of message,

the database needs to be populated differently. For each measure name, there are 8 logical sensors, each one representing a statistic. They are recognizable by the description that follows the format measureName_statistic (e.g.: temperature_media). When the message is sent to the client, the **__data_conversion** function looks for the sensor IDs of the logical sensors with the corresponding measure name and affects the matching statistical value for each of them.

The second type of message pertains to comfort values related to visual, acoustic, and air quality comforts, based on the latest reports. These are transmitted under the topic ms<id>/indexes and include average values and compliance assessments for the mentioned environmental parameters. Regarding thermal comfort, it requires a comparison with the daily running mean and the boundaries calculation. Moreover, the IEQ needs also to be calculated. As the rate for the comfort messages is every 5 minutes, the client calculates directly the thermal comfort and the IEQ index for this interval. All these values are stored in the FiveMinAvgMeasure table.

Chapter 6

Conclusion

This thesis explored the design and implementation of a software architecture to optimize database management in a real-time IoT monitoring service. It contributes to the development of the multisensor project, providing a cost-effective, easily deployable, and manageable solution for monitoring environmental quality in various rooms at Politecnico di Torino. This project presented some challenges that were handled in this work. The continuous growth of the database due to frequent sensor measurements and the diversity of data types necessitate ongoing efforts to improve data management strategies and scalability. The size of a database growing constantly and the diversity of data types necessitates the implementation of an efficient backend server. This should facilitate the data flow from the reception of the sensor's measurements to their accessibility through client services. Thus, The main goal is to offer a responsive and efficient user experience.

First of all, the development of a robust MQTT client that pre-processes, cleans, and standardizes the incoming data from the sensors enhances the capability of analysis and communication between the sensors and the database. The architecture of this component enables the connection of different sources with different data formats, such as messages coming either from custom and commercial devices, or the questionnaire answers filled by the occupants of the rooms.

Furthermore, the implementation of a highly configurable scheduler contributes to the development of a system capable of handling high-frequency data transmission. The utilization of libraries such as APScheduler or SQLAlchemy supports data manipulation through pre-aggregation, ensuring data integrity, qualitative analysis, and efficient reporting. The possibility of computing indexes or statistics at frequent intervals optimizes the computing resources of the server and promotes efficient queries.

The Flask API enables easy access to the database and essential features to rapidly analyze the environment. This system supports dynamic aggregation over time intervals, allowing for detailed analysis and visualization of environmental

metrics. The possibility to integrate the API with Grafana complemented this. Thus, Developing a RESTful API with this framework ensures a responsive and scalable solution. The dependency injection technique promotes the possibility of adding new endpoints or evolving the existing ones. Moreover, Docker, Gunicorn, and wsgi promote the customization of the API for different projects and experiments across different ports. It represents a strategic choice to enable the integration of the software solution with different clients.

In general, this work participates in enhancing the monitoring of the indoor environment by calculating indexes such as thermal comfort, acoustic comfort, visual comfort, indoor air quality index, and indoor environmental index. The comparison of these objective values with the subjective participation of room occupants as the computation of time-series statistics offers an innovative way to analyze the environment.

This work was completed and all required functionalities worked. A real effort has been made regarding the code readability and documentation. The employment of subqueries or common expressions to reduce the frequency of database interactions is a significant advantage in terms of efficiency. Nevertheless, some improvements could be made to increase efficiency and code quality. The aggregations for the IAQ or the IEQ could be better modularized with a common method to encourage code reusability and readability. Moreover, The code for the Promet&o client hasn't been fully tested because the broker wasn't ready at the time of this thesis work. The last feature that comes at the end of the project and hasn't been implemented is the transmission of a message to active leds inside the board in case the IEQ is lower than a certain threshold.

In conclusion, this thesis results in a complete backend that should be extended for the different use cases. Some choices and development of more endpoints such as a CRUD API should be conducted to provide access to more content. Finally, the standardization, development, or discussion of environmental variables to support different projects could be a very interesting point of improvement.

Bibliography

[1] Bernard Marr. «2024 IoT And Smart Device Trends: What You Need To Know For The Future». In: *Forbes* (2024) (cit. on p. 1).

[2] Lionel Sujay Vailshery. «Number of IoT connected devices worldwide 2019-2023, with forecasts to 2030». In: *Transforma Insights* (2023) (cit. on p. 2).

[3] Lacey Butler Nathan McMinn. «How Conserv Safeguards History: Building an Environmental Monitoring and Preventive Conservation IoT Platform». In: *Timescale* (2021) (cit. on p. 2).

[4] Alessandro Ricciuto. *Reliable and efficient solutions for a participative air pollution monitoring system*. 2022 (cit. on pp. 3, 4).

[5] CEN/TC 156. «EN 16798-1:2019 - Energy performance of buildings – Ventilation for buildings – Part 1: Indoor environmental input parameters for design and assessment of energy performance of buildings addressing indoor air quality, thermal environment, lighting and acoustics – Module M1-6». In: (2019) (cit. on pp. 9, 10).

[6] U.S. Environmental Protection Agency. *What are volatile organic compounds (VOCs)?* 2024. URL: `https://www.epa.gov/indoor-air-quality-iaq/what-are-volatile-organic-compounds-vocs` (visited on 03/27/2024) (cit. on p. 11).

[7] U.S. Environmental Protection Agency. *Basic Information about Carbon Monoxide (CO) Outdoor Air Pollution*. 2023. URL: `https://www.epa.gov/co-pollution/basic-information-about-carbon-monoxide-co-outdoor-air-pollution` (visited on 03/27/2024) (cit. on p. 11).

[8] U.S. Environmental Protection Agency. *Basic Information about NO2*. 2023. URL: `https://www.epa.gov/indoor-air-quality-iaq/what-are-volatile-organic-compounds-vocs` (visited on 03/27/2024) (cit. on p. 11).

[9] Alex Grönholm. *Advanced Python Scheduler - APScheduler*. 2023. URL: `https://apscheduler.readthedocs.io` (visited on 03/27/2024) (cit. on p. 17).

[10] WeatherAPI. *Weather API Docs*. URL: `https://www.weatherapi.com/docs/` (visited on 03/27/2024) (cit. on p. 22).

[11] Pallets. *Flask documentation*. URL: https://flask.palletsprojects.com/ (visited on 03/27/2024) (cit. on p. 36).

[12] Grafana Labs. *JSON API for Grafana*. en. URL: https://grafana.github.io/grafana-json-datasource/ (visited on 03/27/2024) (cit. on p. 50).

[13] Roger Lights. *Paho MQTT documentation*. URL: https://eclipse.dev/paho/files/paho.mqtt.python/html/client.html (visited on 03/27/2024) (cit. on p. 53).

[14] Docker Inc. *Docker overview*. URL: https://docs.docker.com/get-started/overview/ (visited on 03/27/2024) (cit. on p. 56).