

POLITECNICO DI TORINO

Master's Degree in
COMPUTER ENGINEERING



Master's Degree Thesis

Design and Development of an Air Pollution Data Distribution System

Supervisors

Prof. Maurizio REBAUDENGO

Prof. Bartolomeo MONTRUCCHIO

Candidate

Kamil Lukasz SMOLEN

October 2020

Index

List of figures	VI
1 Introduction	1
1.1 General context	1
1.2 Thesis topic	1
1.3 Starting situation	2
1.4 Thesis organization	3
2 Technologies and programming languages	5
2.1 DBMS MySQL	5
2.2 Flask	5
2.3 Python	6
2.4 RESTful WebServices	6
2.4.1 WebServices	6
2.4.2 REST	7
2.5 Android	9
2.5.1 Overview	9
2.5.2 Android software development	10
2.5.3 Android application components	11
2.5.4 MPAndroidChart	13
2.6 Open Street Map	13
3 Platform architecture	15
3.1 Remote server	15
3.1.1 Accumulation of data concerning environmental pollution . .	16
3.1.2 maintenance of auxiliary tables in the DB	16
3.1.3 user management	17
3.1.4 calculation of coefficients for data calibration	21
3.1.5 returning the calibrated data to the client	21
3.2 Android client	23
3.2.1 Main screen	23

3.2.2	Login screen	25
3.2.3	Registration screen	25
3.2.4	Polution screen	26
3.2.5	Last 2 hour history screen	27
3.2.6	General history screen	28
3.2.7	Profile showing screen	29
3.2.8	Profile editing screen	30
3.2.9	Tools screen	31
4	Server implementation	33
4.1	Modularity	33
4.2	Web service startup	34
4.3	Initialization and configuration	35
4.3.1	__init__.py	35
4.3.2	config.py	37
4.4	Utils.py	37
4.5	Templates and static content	39
4.6	Users module	42
4.6.1	Controlllers	42
4.6.2	Models	47
4.6.3	Forms	49
4.6.4	Schemas	49
4.7	Measures module	49
4.7.1	Controlllers	51
4.7.2	Models	62
4.7.3	Schemas	63
4.7.4	Utils	63
4.8	ws_checking_system module	66
4.9	Other project folders	66
5	Mobile application implementation	67
5.1	Android manifest	68
5.2	Activities	69
5.2.1	MainActivity	69
5.2.2	LoginActivity	75
5.2.3	RegisterActivity	77
5.2.4	PollutionActivity	78
5.2.5	HistoryActivity	88
5.2.6	GeneralHistoryActivity	94
5.2.7	ShowProfileActivity	95
5.2.8	EditProfileActivity	96

5.2.9	ToolsActivity	96
5.3	Layouts	97
5.3.1	activity_main.xml	97
5.3.2	activity_pollution.xml	104
5.3.3	activity_history.xml	106
5.4	Values	111
5.5	build.gradle(app)	111
6	Testing and evaluation	113
6.1	Testing context	113
6.2	Testing scripts	113
6.3	Testing machine	119
6.4	Testing procedure	119
6.4.1	First latency/throughput testing	119
6.4.2	First inserting data script execution	120
6.4.3	Second latency/throughput testing	120
6.4.4	Cheking the inserted data	120
6.4.5	Second execution of the inserting data client	120
6.5	Results	121
6.5.1	Latency	121
6.5.2	Throughput	121
6.5.3	Correct insertion of data	122
	Conclusions	125
	Bibliografy	127

List of figures

2.1	Activity lifecycle [18]	12
3.1	Registration email example	18
3.2	User confirmed	18
3.3	Password recovery email example	19
3.4	Reset password form	20
3.5	Result of the reset password procedure	20
3.6	Main Screen	24
3.7	Lateral menu	24
3.8	Login screen	25
3.9	Missing value message	25
3.10	Registration screen	26
3.11	Pollution screen	27
3.12	Pin informations	27
3.13	Current user position	27
3.14	Last 2 hour history screen	28
3.15	General history screen	29
3.16	History charts	29
3.17	Profile showing screen	30
3.18	Editing profile screen	30
3.19	Tools screen	31
4.1	Web service project structure	34
4.2	User record structure	43
4.3	measure_table structure	50
4.4	board_table structure	50
4.5	five_min_avg table structure	51
4.6	hour_avg table structure	51
5.1	Android Manifest and Activities	68
5.2	Res folder organization	68

6.1	Latency results	121
6.2	throughput results	122
6.3	Local DB records	122
6.4	Number of record before script execution	123
6.5	Number of record after script execution	123
6.6	Local DB records after the second execution of the testing script . .	123
6.7	Number of records after the second execution of the script	123

Chapter 1

Introduction

1.1 General context

Living in an era dominated by information, where this is widely available and handy simply by pulling out the smartphone and where staying connected to the network is not difficult at all, it opens up many avenues for innovation of all kinds. Just think of the amount of devices that are becoming increasingly smart, paving the way for services that are becoming essential to our lives, without which every day would be much more difficult. Let's think only of the possibility of checking the departure time of the bus, checking the weather or simply staying up to date with the latest news. The possibilities are nearly endless and our infrastructures, buildings and even cities are becoming "smart". In the latter case, monitoring systems are increasingly used and among these we must not forget environmental monitoring. Living in a period in which to take care of our health and above all of the environment is entering our daily routine, being aware of the environmental situation in our close proximity becomes indispensable. Exactly for this reason, technological innovation should allow us to access to this information with the utmost simplicity.

1.2 Thesis topic

Taking into consideration the need for environmental monitoring and for the people themselves to remain up-to-date on the surrounding situation, the main purpose of the thesis is to allow every inhabitant of the city of Turin to have immediate access to information regarding the quality of the air in their close proximity. The work concerns the design of a system based on client-server architecture [1] capable of distributing air pollution information in a simple and free way. Starting from the in-depth study and the choice of the right technologies to use, the project

concerns the complete development of a backend, that is a web service capable of supporting the collection, distribution and manipulation of measurement data, together with the complete management of the target users of the system. The thesis also includes the creation of a mobile application capable of communicating with the remote server, giving each user the possibility of using all the services offered by the backend for consulting the data regarding air pollution in the simplest and most intuitive way possible.

1.3 Starting situation

The thesis involves the creation of a data distribution system to end users. The work is based on an already existing participatory data collection system. The previous project was created as a thesis and had the purpose of collecting data and their subsequent storage.

This system is based on a network of RaspberryPi-type devices distributed in strategic points of the city of Turin, equipped with sensors for measuring the quantity of particles of the pm10 and pm2.5 type as well as sensors for measuring ambient temperature, pressure and humidity. More specifically, the sensors for collecting data regarding the concentrations of particles pm2.5 and pm10 are in greater number, to be exact they are 4 for each type. This is because the system has been designed to be relatively cheap compared to existing systems and consequently to obtain more accurate measurements and to make the system more robust in case of permanent or temporary failures to one of the sensors, they have been inserted with redundancy. It should also be known that data samples are collected every second. All these data are initially collected and saved locally on an SD card on board of the device. Thanks to a mobile application installed on the smartphones of users who decide to participate in the project and allow it to run in the background on their device, the data is sent to the server using the mobile network of the same device every time a user with this app passes near one of the data collection system, and consequently the data is stored permanently on the remote DB.

One of the key ideas of this previous project was the creation of a system capable of collecting reliable data without investing too much money in sensors. This means that the data collected in order to be used and distributed to end users must first be calibrated. This is possible thanks to the placement of some collecting systems in the Arpa Piemonte [2] weather station in Turin which provides reference data.

The calibration of data together with all the logic of their manipulation and distribution system to end users is instead the subject of the current thesis.

1.4 Thesis organization

The thesis is organized as follows:

- chapter 2 has the purpose of briefly describing the technologies and programming languages used during the development of the system
- chapter 3 conceptually describes the architecture of the entire data distribution system, analyzing each composing it part separately
- chapter 4 aims to explain how the backend was implemented, analyzing its modularity and each of its modules
- chapter 5 is dedicated to explaining the implementation of the mobile application
- chapter 6 describes how the tests were carried out and on which aspect they are mainly oriented, describes the implementation of these tests and finally discusses the results obtained
- finally follows a brief conclusion

Chapter 2

Technologies and programming languages

2.1 DBMS MySQL

MySQL is a complete system for managing databases, in other words it is a DBMS (Database Management System) owned by the Oracle Corporation company. It is mainly based on SQL which is a standardized programming language to allow the management and manipulation of relational model based databases. For this reason MySQL can more specifically be called RDBMS to emphasize the fact that it supports the relational model. The software is currently available both under the GPL (General Public License) and the commercial license[3]. Thanks to being available also in open source, MySQL guarantees excellent performance with a high level of security that can be freely verified by accessing the source code itself.

From the beginning MySQL has been designed to maximize performance, allowing the management of large amounts of data while also offering maximum compatibility and remaining as compliant as possible with ANSI-SQL standards with also offering more functionalities[4]. It is currently available on both Unix and Unix-like systems as well as on Windows which makes it usable on virtually any existing operating system. MySQL is also supported by many programming languages including Java and Python which are the basis of this thesis.

2.2 Flask

Flask is nothing more than a web framework written in Python and distributed under a BSD license, used to support the development of web applications including web services or web API's. More precisely it can be classified as a micro framework

as its core is simple but extensible. Inside there are only the basic features and we do not find any code to implement features for which third-party libraries already exist[5]. This means that there is no abstraction layer for databases or for form validation. The programmer has free choice to use external components of any kind and connect them as if they were native to the framework. Thanks to the simplicity that allows to obtain a modular and flexible code without being burdened by useless functionality, Flask was the best choice for this thesis

2.3 Python

Python is a high-level programming language, this means that it offers a considerable level of abstraction compared to the implementation on the computer. The language is considered as interpreted even if for all intents and purposes the code after the first interpretation is converted into an intermediate language (byte code) which in turn is reused in subsequent interpretations to ensure performance benefits[6].

Released in the early 1990s, Python can be considered a multi-paradigm language as it is suitable for object-oriented programming, procedural programming or functional programming.

The philosophy of the language is based on simplicity, the code written in Python is very readable and thanks to the indentation that involves the use of a large number of whitespaces instead of brackets, the code is clean and understandable compared to other programming languages. Often this aspect makes Python perfect for teaching programming. The language is also often called as "battery included" because it has a feature-rich standard library that allows everyone to immediately start writing code[7].

Python is characterized by the use of non-typed variables but dynamically typed objects which means a strong type checking system. It also includes a dynamic garbage collector.

The language remains among the most used and most popular in the world[8].

2.4 RESTful WebServices

2.4.1 WebServices

A web service is a software system designed to support distributed communications, architected to ensure the interoperability of programs that work on different software and / or hardware architectures using distinct and different computers[9]. Communication occurs through the exchange of SOAP type messages encapsulated within the HTTP protocol, from which the name "web services" comes. Messages are often serialized using the XML standard or the JSON format.

The concept of web service has the characteristic of being abstract, as for the concrete implementation we need agents and services. An agent is a concrete software program, implemented in a programming language and on a specific hardware platform, capable of sending and receiving messages described above. A service, on the other hand, is the concrete implementation of a program capable of satisfying the agent's requests by sending appropriate responses encapsulated in messages, thus carrying out the task of a data server[9].

Often in a web service we can count many agents and a single service, as we tend to have an architecture with centralized data processing that responds to multiple requests from different agents.

A very important feature of web services is that both agents and services can be implemented using programming languages chosen by the developer, there are no rules that bind the web service to the use of a specific programming language. Thanks to the distributed architecture, the two components of the web service do not need to know how the counterpart is implemented, as long as the API (Application Programming Interfaces) or the communication interfaces remain the same.

2.4.2 REST

Representational State Transfer is an architectural style for the design of distributed software services such as web services. This means that REST is not exactly an architecture but not even a standard, it is a set of architectural principles. REST is characterized by the fact that in the communication between the counterparties it is used only on the http protocol without further intermediate levels of encapsulation such as eg. SOAP messages. Furthermore, the concept of session is not envisaged in REST systems, that is why this architecture is called stateless. Any Web API that respects the REST rules can be called a REST API in all the effects[10].

REST can be described by five basic principles:

- Resource identification: A REST-based web service is resource-oriented, where a resource is any element that can be processed. Each resource must be uniquely identified by a URI (Unique Resource Identifier)[11].
- Use of HTTP methods: this means that the methods integrated in the HTTP protocol are used to express actions that we want to perform on the various resources. To request a resource, the get method must be used and to perform CRUD operations, we must take advantage of the PUT, POST and DELETE methods[12].
- Self-describing resources: Each resource returned to the client is conceptually separate from its representation on the server. The server's job is to transform

the requested data into a format compliant with the request sent by the client. The format can be for eg. XML, JSON or CSV.

- Links between resources: the resources must be connected to each other through hypertext links. Furthermore, each resource must contain everything that is necessary for its description and the description of its links, this means that the links must be encapsulated in the resource itself. This mechanism allows navigation between the resources themselves, using only what we have already requested from the server.
- Stateless communication: each request sent by the client to the server must be completely independent of the previously sent requests. This makes the communication stateless but it does not mean that the application must not have any state, simply thanks to the use of hypertext links contained within each object it is possible to have a statefull application with the use of a stateless communication that does not overload the server especially in cases of a large amount of requests[13].

REST APIs can also be divided according to their level of maturity, that is, according to the level of compliance with the principles described above. There are four levels of maturity[14]:

- Level 0: A REST API is said to be level zero if it uses the HTTP protocol as its communication protocol. This is because ideally a distributed application could use a different protocol even if it is not widely adopted.
- Level 1: a REST API is level one when different URIs are used to interact with different resources, ie the principle of identifying resources is applied.
- Level 2: this level of maturity is reached when the methods integrated in the HTTP protocol are exploited to express actions concerning resources. The GET method is used to request the resources, while the following methods are used for CRUD operations: POST, PUT and DELETE. This level of maturity also involves the use of standard HTTP protocol response codes.
- Level 3: a REST API is level three when hyperlinks are used to establish the connection between the various resources of the web application. This level implements the so-called HATEAOS (Hypermedia As Transfer Engine Of Application State) that is, the state of the application is managed through links integrated in the representations of the resources. An API of this level is also called a RESTful API.

2.5 Android

2.5.1 Overview

Android is a mobile operating system developed by Google. It is based on the Linux kernel and it is to all intents and purposes an embedded Linux distribution and not a Unix-like system. Developed to run on embedded systems such as smartphones and tablets but also watches and TVs. The operating system is an open source project distributed under the Apache license. The source code is freely accessible on the Internet and is known as AOSP (Android Open Source Project).

The development of Android has been going on uninterrupted since its first release in 2008. Several versions of the operating system overseen by Google have been developed over the years.

Version(name)	API Level	Release year
1.0(/)	1	2008
2.0(Eclair)	5	2009
3.0(Honeycomb)	11	2011
4.0(Ice Cream Sandwich)	14	2011
5.0(Lollipop)	21	2014
6.0(Marshmallow)	23	2015
7.0(Nougat)	24	2016
8.0(Oreo)	26	2017
9.0(Pie)	28	2018
10.0(Q)	29	2019

Table 2.1: Android available versions

Currently the most updated version of the operating system is version 10 also called "Android Q", released in August 2019.

Despite the availability of the latest version, the distribution on the market of the most popular operating system in the world looks like follows:

Android Platform Version (API Level)	Distribution (as of April 10, 2020)
Android 4.0 “Ice Cream Sandwich” (15)	0.2%
Android 4.1 “Jelly Bean” (16)	0.6%
Android 4.2 “Jelly Bean” (17)	0.8%
Android 4.3 “Jelly Bean” (18)	0.3%
Android 4.4 “KitKat” (19)	4%
Android 5.0 “Lollipop” (21)	1.8%
Android 5.1 “Lollipop” (22)	7.4%
Android 6.0 “Marshmallow” (23)	11.2%
Android 7.0 “Nougat” (24)	7.5%
Android 7.1 “Nougat” (25)	5.4%
Android 8.0 “Oreo” (26)	7.3%
Android 8.1 “Oreo” (27)	14%
Android 9 “Pie” (28)	31.3%
Android 10 (29)	8.2%

Table 2.2: Android distribution[15]

It remains very clear that the latest version is not the most popular on the device market[16]. This means that when we are developing applications for Android devices, we have not to forget about previous versions and pay close attention to backward compatibility.

2.5.2 Android software development

The development of applications for Android devices can take place through the use of different programming languages such as Java, Kotlin or even C ++. The design of each Android application is simplified thanks to the possibility of using an SDK (Software Development Kit), that is a set of tools including a debugger, library, emulator and much more that help the programmer in developing the code. Since 2015, the official development environment, the so-called IDE (Integrated development environment) of Android becomes Android Studio, replacing the previous Eclipse. The development of the SDK and IDE proceeds in parallel with the development of the operating system itself without losing sight of backward

compatibility. Within the SDK we also find ADB (Android Device Bridge) that is a tool that allows you to execute instructions on a connected Android device. This allows debugging directly on the device itself.

2.5.3 Android application components

The components of an Android application can be considered as basic blocks on which to build a program based on this operating system. These components are linked together thanks to a file in XML format, called Android Manifest which contains their complete list together with a description and how they interact[17].

There are four types of Android components:

- **Activities:** this component has the task of presenting the user with the graphical interface, giving the possibility to interact with the application. An Android app can consist of one or more Activities, that is, one or more screens with which the user can interact. When a program is started, the operating system loads the MainActivity that is the main screen which in turn has the ability to call others if the application provides for it.

The activities are managed by the so-called Activity stack, that is, a stack that contains all the activities performed by the application. Each new activity that is executed goes to the top of the stack and becomes the one running while the previous activities remain in the lower places as long as the activity at the top remains active.

Each activity is characterized by a very particular life cycle and can be in 4 different states:

- the activity is at the top of the stack and it is in the running state. This means that we are talking about an activity with which the user is interacting and it is active.
- the activity is visible on the screen but has lost focus, this means that the user no longer has the opportunity to interact with it
- the activity is completely obscured by another and is no longer visible on the screen. This means that it is in the stopped state.
- the activity is ready to be destroyed by the operating system, this means that it will have to be recreated in order to be visible again on the screen.

The following figure shows the complete lifecycle of an activity to better visualize its behavior. In the figure we can see that the colored elements represent the states in which an application can be found while the gray rectangles represent the methods that can be overwritten to implement the actions we want during the transition from one state to another.

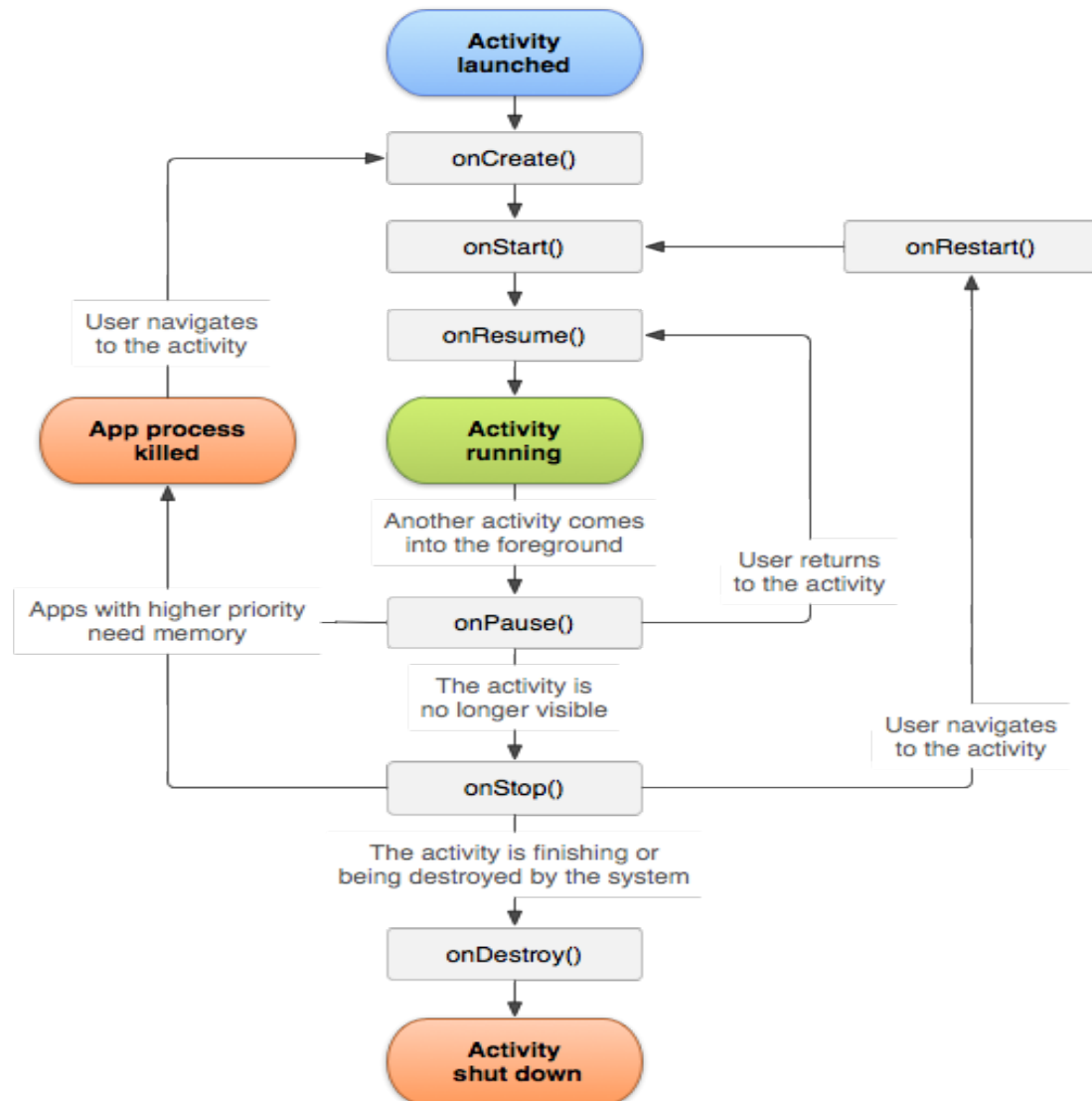


Figure 2.1: Activity lifecycle [18]

- **Services:** they are used by the operating system to perform long-running tasks. These components do not have a graphical interface and always run in the background. They can be used for example for application processing or for playing music in the background while using another application.
- **Broadcast Receivers:** this component is used for communication between the application and the operating system or for inter-application communication. It allows the application to receive broadcast messages, it is the dual component

of the Intent that allow the application to send them.

- Content Providers: this component has the task of dealing with any data problem. It can be used to retrieve data for example from a DB or from another application. This procedure is performed through the use of a Content Resolver.

2.5.4 MPAndroidChart

MPAndroid Chart is an open source library for plotting charts within Android applications. The library was developed by PhijJay and its source code is fully available in its repository on GitHub. It offers the possibility to draw almost any graph without the need to use the Canvas, that is the Android drawing on the screen element. MPAndroid Chart was used in this thesis to plot some LineCharts.

2.6 Open Street Map

Open Street Map is a collaborative project for the creation of geographic maps. The work is aimed at mapping the whole world and making the content freely accessible. For this reason OSM is distributed under an Open Database License which is totally a free license. The use of the maps is also permitted for commercial purposes as long as the source is cited. The project is carried out using free sources such as aerial photographs. Everything is funded by the OpenStreetMap Foundation which is responsible for finding the necessary funds. The work is called collaborative as each user registered to the project can upload new data as GPS tracks of their device[19]. The thesis involves the implementation of these maps as the project is completely free.

Chapter 3

Platform architecture

This chapter describes how the entire project works. It consists of two main elements which are respectively a remote server that performs the task of a web service implemented using Flask micro-framework and a client implemented on the Android platform capable of communicating with the server through the Internet for viewing the data contained on the server.

The project consists of a data distribution system regarding environmental pollution. Specifically, these are particles of type pm2.5 and pm10 collected, as described in chapter 1.3 of this thesis, from an already existing system whose evolution is represented by the project described here. The server has the hit of storing the data collected by the system and saving them in a DB, but also providing the calibrated data return service. The client in turn has the hit of displaying the data present on the server to the end user of the system, that is, every person who has downloaded the Android application and successfully registered in the system.

The project description does not deal with implementation details, which will be described in the following chapters. The work is described from a conceptual point of view by explaining the tasks performed by each of the two main elements of the project, together with the description of how they work and their usage.

3.1 Remote server

The remote server is one of the two main components of the entire project. This key element of the system has been designed to play the role of a web service which means that it must be installed on a computer that performs the work of a server and must be accessible through the Internet, in other words, through a public IP address. The server is very important as it is the core of the project and allows it to work properly. This element, in addition to guaranteeing the basic functions

for the operation of the environmental monitoring system, also has the task of guaranteeing safety and robustness to the system as it must remain online ideally for the entire operating time of the system and any failure inside it can cause the interruption of the service provided.

Among the main tasks of the server we can list:

- accumulation of data concerning environmental pollution
- maintenance of auxiliary tables in the DB
- user management
- calculation of coefficients for data calibration
- returning the calibrated data to the client

3.1.1 Accumulation of data concerning environmental pollution

Since this thesis is based on an already existing data collection system concerning environmental pollution, the server must implement the service that offers the possibility of saving the data collected within a DB. The whole mechanism works by sending an appropriate HTTP request to the server. To carry out this work, the server exposes an endpoint with final address `/measure`, which when it receives a request with the PUT method, analyzes the content of the request body from which it extracts the data to be inserted into the DB. The data must in fact be present within the body of the request in JSON format. Together with the data, the request must contain a particular header called `"x-access-token"` containing the unique token of the currently logged in user necessary for the correct processing of the request itself. Once the server has received the incoming request, and after analyzing its semantic correctness, it proceeds with the insertion of data into the DB. The received data are not analyzed from the point of view of their semantic correctness but are inserted into the DB because the logic of the server provides for the accumulation of raw data. The following sections will describe how this data is manipulated and returned to the client.

3.1.2 maintenance of auxiliary tables in the DB

The operation of the entire system is strictly dependent on the environmental data collected by the monitoring devices, which, due to their design, collect a data sample for every second of their operation. This means that within the table containing the various measurements within the DB, the amount of data is very high. From the point of view of request processing speed as well as from the point of view of

the client and the end user, processing and viewing data with such frequency would be very difficult. Nor should we forget that the communication between the client and the server takes place through the Internet, which means that the amount of data transferred directly affects the response speed of the server perceived by the end user. The following paragraphs also describe how the collected data must be manipulated before being returned to the client for visualization within the Android application. All these reasons explain the need to create two additional tables within the DB containing respectively the average of the measurements made every 5 minutes and the average of the measurements made every hour. In fact, one of the main tasks of the server is to keep the two tables up to date. This task is carried out through the use of two asynchronous tasks which, activated respectively every 5 minutes or every hour, perform the necessary calculations on the data and update the two tables. Thanks to their presence within the DB, the response and processing of requests on the server side can be done faster, thus providing a better service. In this thesis two tables are kept updated but the client only uses the one containing the 5 minutes average data, the second one is left for future uses.

3.1.3 user management

The server internally implements the logic for managing users. This mechanism is strictly necessary to ensure the security of the entire system as it allows interaction with the web service only to registered users. The logic of the server provides that users can cover different roles, more specifically users can be: user, admin, or system_admin. The presence of three different roles within the system has been planned for future use and further extensions of this project. At this time, within the server there are only three functions that can be performed by a user superior to the simple user. The first one concerns the possibility of promoting or degrading the user's role. The second one provides the ability to temporarily disable a user by marking his record within the DB as not active. This feature was also designed for future system extensions and can be performed by a higher user than a simple user. The last one will be described in the following subsections. The only difference between admin and system_admin is that the latter cannot be change his role.

The server offers all the following features necessary for proper user management:

- user registration
- user login and logout
- password recovery
- account editing

User registration

The server offers the possibility for users to create a new account within the service. To do this, the server offers an endpoint with a final address `/register`. The HTTP request for registration of a new user must be sent to this address with the POST method, while the body must contain the user data in JSON format.

Once the server has received the request and, after checking its semantic correctness, proceeds to insert the new user into the DB, marking the corresponding record as "to be confirmed". Each new user is created with role of "user" and active state marked as "true".

At this point the server sends an account confirmation email to the email address specified during the registration phase. Within the message a link with a unique token is inserted that must be clicked by the user to complete his registration phase[fig. 3.1]. Once visited, a web page is displayed with the confirmation of the action in progress[fig. 3.2] and the user is marked as confirmed in the DB. The registration phase is now concluded.

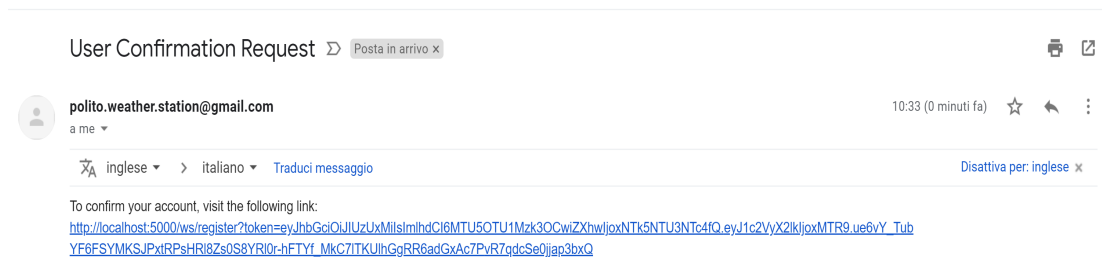


Figure 3.1: Registration email example

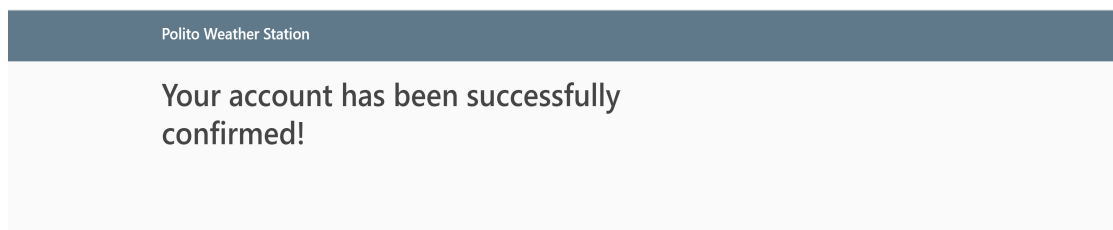


Figure 3.2: User confirmed

User login and logout

In order to manage users correctly, the server provides the login and logout functions within the system. This is because only users who have already logged in have

the full power to interact with the web service. To allow the login procedure, an endpoint with final address `/login` has been created which responds only to requests sent via the `http POST` method. The body must contain the user's authentication credentials, i.e. email and password, encoded in JSON format. Once the login request has been received, the server proceeds with the generation and sending to the client of a unique and valid token until the next logout. This token must be linked to each subsequent request sent to the server by the user in order to confirm his identity. The token is added to future requests by adding a header with the name of: `"x-access-token"`.

The logout is the opposite procedure to login and can in turn be performed by contacting the server through a `POST http` request on an address ending with `/logout`. The corresponding token of the user who wants to exit the service must be attached to this request. Following this request, the server proceeds with the invalidation of the token which will no longer be accepted as valid in future requests.

Password recovery

The server to ensure the possibility of recovering the passwords forgotten by users offers the possibility to change them. This procedure is initiated by sending an `http` request with the `POST` method to the address that ends with `/recover`. This request does not require the use of any token but only the sending within the body of the email address of the user who wants to change their password. When the server receives this request, it proceeds with the generation of a temporary token, valid for 30 minutes from its creation. The token is inserted into a link sent to the user via email[fig. 3.3]. At this point the user, by accessing his mailbox and clicking on the received link, is directed to a web page that allows him to enter a new password[fig. 3.4]. By confirming the form, the user is notified of the result of the current procedure[fig. 3.5]. At this point the user can use the new password to access the service.

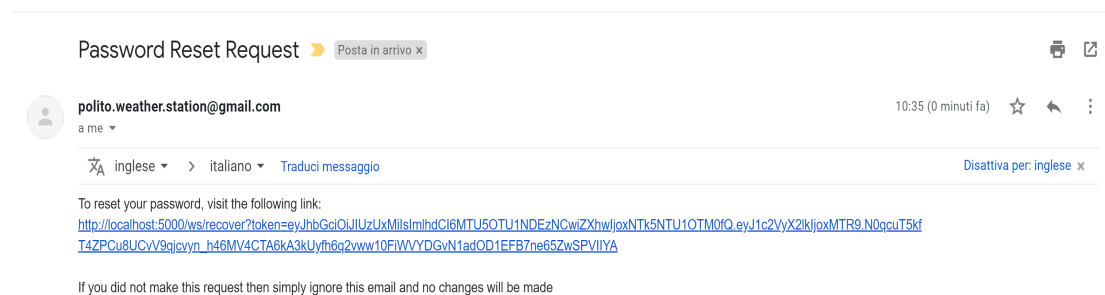


Figure 3.3: Password recovery email example

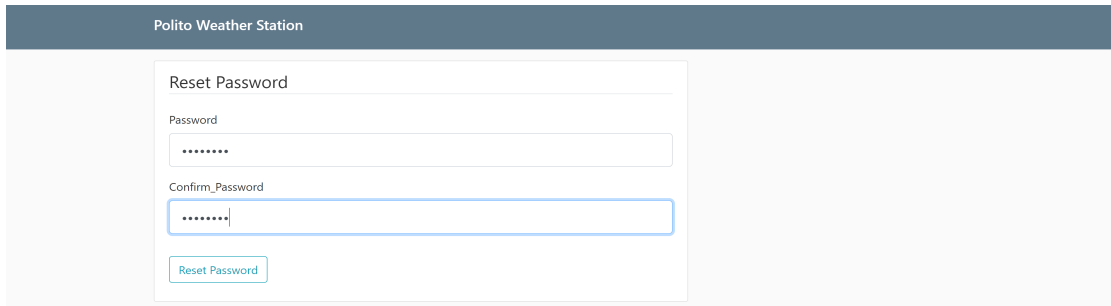
The screenshot shows a web interface for the 'Polito Weather Station'. At the top is a dark blue header with the text 'Polito Weather Station' in white. Below the header is a light gray background containing a white rectangular form. The form is titled 'Reset Password' in a small, dark font. It contains two input fields: the first is labeled 'Password' and the second is labeled 'Confirm_Password'. Both fields contain a series of dots representing masked text. Below the input fields is a small, light blue button with the text 'Reset Password' in a darker blue font.

Figure 3.4: Reset password form

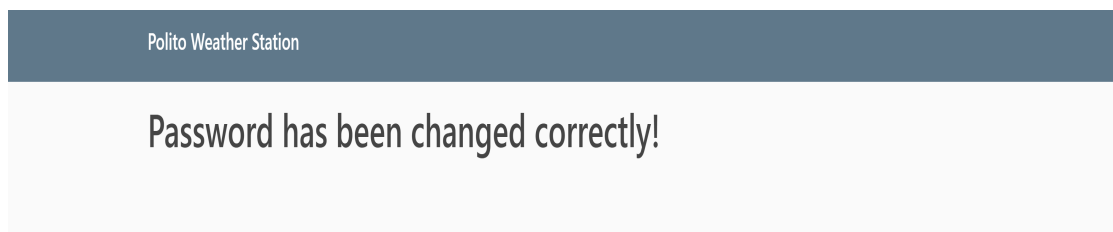


Figure 3.5: Result of the reset password procedure

Account editing

The service offers each user registered within the system the ability to change the data relating to their account at any time, as well as allowing the currently logged in user to view the user's data currently saved on the server. Within the system there are two non-equal procedures for changing data concerning a registered user. The first consists in sending an http request with the POST method to the address that ends with `/users/<user_id>`, where `user_id` is the unique identifier of each user and can be retrieved from the token received during login. The body of this request must contain the data to be modified on the server, encoded in JSON format. The request must also be accompanied by the "x-access-token" header. The second procedure that allows to change the user data is reserved only for users with a role higher than the simple user and allows to change both the role and the active status of a user. The use of this request has been implemented for possible future uses and extensions. The procedure begins by sending an http request with the PUT method to the address that ends with `/user/<user_id>`, specifying the data to be modified in the body (ROLE or active = True / False).

The functionality of viewing the data present regarding a user's account is available by sending an http request with the GET method to the server at the address that ends with `/user/<user_id>`, in response the client will receive the

user data encoded in JSON format. Also the last two methods must include the “x-access-token” header in the request.

3.1.4 calculation of coefficients for data calibration

One of the main tasks of the server is data manipulation. These must be calibrated before they can be sent to the client for viewing. The data calibration procedure is performed at runtime and is described in the next subsection. However, it remains important to know that to perform the calibration, the server must first calculate the regression coefficients. These are calculated for a certain time range and must be recalculated when they expire.

The system at its first start, before the first request received, performs the calculation of these coefficients. The work is done through the use of a secondary thread that does not hinder the operation of the server.

Since these coefficients have a validity period, the server offers the possibility to recalculate them by sending an http request with the POST method to the address that ends with “/measure”. The body must specify both the new effective start date and the new effective end date, encoded in JSON format. This work will only be done if the request is sent by a user who has a higher role than the simple user.

3.1.5 returning the calibrated data to the client

One of the main tasks of the server is to return data regarding environmental pollution ready for visualization on the client. The data is transferred in JSON format after being calibrated. Calibration is the procedure by which a correction is applied to each required record by applying linear regression using the coefficients described in the previous section. Data calibration must be performed because the sensors mounted on the detection devices (boards), which have the task of providing data as reliable as possible without exaggerating the material cost of the device itself, are not as accurate as the sensors used by weather stations in the city of Turin. For this reason, in the calculation of the regression coefficients, data coming from the boards positioned together with the official sensors of the ARPA station in Turin are used. This allows the collected data to be correct and reliable.

Raw data calibration is performed at runtime each time a request arrives, this is because the amount of data is very high within the DB and keeping a table containing the corrected data means doubling the space occupied on the server. The choice of the runtime calibration derives from the fact that each data request is limited in a generally small time span and therefore the data to be corrected are consequently few. The “ws_analysis” library developed at the Politecnico di Torino specifically for this purpose is used to perform the calibration.

There are two ways of requesting data on the server. The first mode can be

called by the client using an http request with the GET method at the address ending with `"/measure /<pm_kind>"`, where `pm_kind` represents the type of polluting particles in the environment. This mode returns the last available record, in chronological order, properly calibrated, for each detection sensor in the DB. In this case, the service works on the table containing the data grouped every 5 minutes in order to make the calculation faster and limit the amount of data transferred. The second method of requesting data takes place following the client's request sent via the http GET method to the address `"/measure/<pm_kind>/<board_id>"`, where `board_id` represents the environmental pollution survey board from which we want to receive the data. Some headers must be added to this request to better specify what data the client wants to receive. Through the use of the headers we can specify the time range for which we want the data, we can specify whether the data must be filtered based on the ID of the detection board or based on the ID of the single sensor (remembering that each board inside contains 4 sensors for detecting pm2.5 particles, 4 sensors for detecting particles pm10, a sensor for detecting environmental temperature, humidity and pressure). Finally, the headers allow the client to specify whether the data to be returned must be those of the last two hours available starting from a certain starting time. This feature has been implemented to quickly visualize the most recent history of the data itself. Also this method of getting data is working on the table containing the five minute average records.

3.2 Android client

The mobile application based on Android technology is the second main element of the entire project. It aims to display data regarding environmental pollution, such as PM 2.5 and PM 10 particles, together with information on environmental temperature and humidity. The application was designed to be a client in all respects, this means that it does not save any type of data internally, but all the information necessary for visualization is requested from the web service described above. The requests are made through a call to the server and follow the rules of the REST API's. The data is downloaded in real time only when strictly necessary so as not to burden the use of the device, thus remaining up-to-date.

The application is structured in different screens:

- Main screen
- Login screen
- User registration screen
- Pollution screen
- Last 2 hour history screen
- General history screen
- Profile showing screen
- Profile editing screen
- Tools screen

3.2.1 Main screen

The main screen represents the first view that the user sees after starting the application. Depending on whether the user has already logged in or not, this screen shows the "Logout" button or the "Login" button [fig. 3.6]. This view mainly allows navigation to other activities. By pressing the "Pollution" button, the user will be shown the screen that allows to obtain and visualize the data regarding the environmental situation, while by pressing the "History" button, the user will be able to consult the history of previous measurements. The "Pollution" and "History" buttons can be clicked only if the user has already logged in to the system, otherwise a temporary message will be displayed indicating to log in first. In the event that the user has not yet been authenticated, by pressing the "Login" button, the view that allows to log in to the system will be shown. Similarly, if the "Logout" button is visible, once pressed, the user logs out from the system.

On this screen, it is possible to access a side pop-up menu by clicking on the icon at the top left corner[fig. 3.7]. The menu in its upper part displays the currently logged in user (if available), while in its second part it allows navigation to two other views. The first, navigable by pressing the "Profile" option, displays the data relating to the current user account, while the second, navigable by pressing the "Tools" button, allows access to the application options screen.

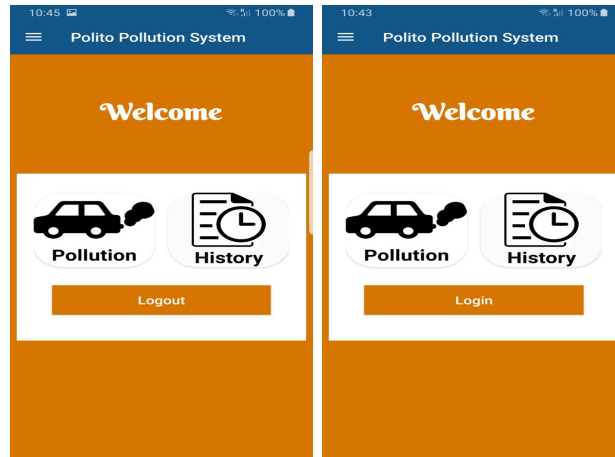


Figure 3.6: Main Screen

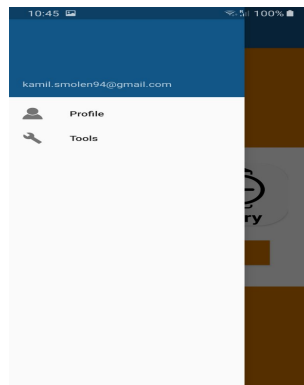


Figure 3.7: Lateral menu

3.2.2 Login screen

This screen allows the user to log in to the application by entering the email and password used during user registration[fig. 3.8]. The current view also allows to switch to the registration screen if the user has not yet performed the registration procedure. To be able to register a new user just press on the word "Not an User?". The current screen also allows the user to recover the login credentials in case of loss of the password. The procedure consists of entering the email in the corresponding box and clicking on the words "Don't remember your password?". In this way, the application will ask the server to change the password by sending an email to the user. In the event that the message for password recovery is pressed but the email box is empty, an information message about it will be displayed near the email field[fig. 3.9].

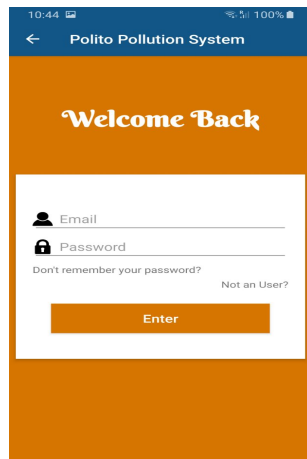


Figure 3.8: Login screen

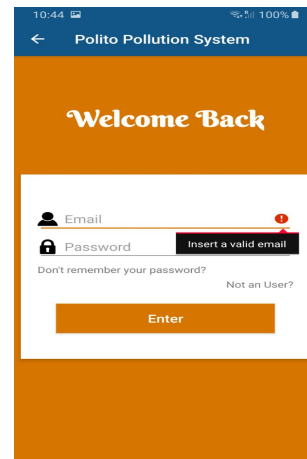


Figure 3.9: Missing value message

3.2.3 Registration screen

This screen has the task of allowing the user to create a new account to register in the system. The procedure consists in filling in all the fields visible on the screen and confirming the operation by clicking on the "Register" button[fig. 3.10]. The application, after verifying the data entered by the user, proceeds with a request to the server for the creation of the new user. In the event that the email is not valid, not all fields have been filled in or the password is not identical to its confirmation, the view shows an appropriate message with the description of the current error near the field concerned. The error message is displayed in the same way as in the "Login" screen[fig. 3.9].

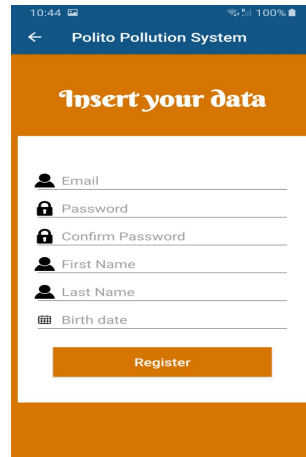
The image shows a mobile application registration screen. At the top, there is a blue header bar with a back arrow, the text "Polito Pollution System", and status icons for time (10:44), signal, and battery (100%). Below the header is an orange banner with the text "Insert your data". Underneath the banner is a white registration form with the following fields: "Email" (with a person icon), "Password" (with a lock icon), "Confirm Password" (with a lock icon), "First Name" (with a person icon), "Last Name" (with a person icon), and "Birth date" (with a calendar icon). At the bottom of the form is an orange "Register" button.

Figure 3.10: Registration screen

3.2.4 Polution screen

This screen has the task of displaying in the most intuitive way possible the data regarding ambient pollution, collected by the environmental monitoring system in the city of Turin.

The view consists of a map, centered according to the specified setting of the "Tools" screen, respectively in the center of the city of Turin or in the current location of the user. For this feature the positioning of the device have to be activated. At the top of the map, two buttons have been placed, one green and one orange. The first represents the type of polluting particles currently visible on the screen while the second allows to change it.

Indicators (pins) are placed on the map, which indicate the position of the environmental pollution detection boards together with their range of action (20 meters)[fig. 3.11]. The color of the circle around the pin is chosen based on the quality of the surrounding air. To determine the color and the message to be displayed in the pin, the AQI (Air Quality Index) compliant scale is used. AQI is an index that indicates the level of air pollution and based on it, the quality of the surrounding air can be established[20]. Pressing on one of the pins displays the latest data available on the server regarding the environmental situation at that point together with the date on which they were collected[fig. 3.12]. It can be seen that the board ids linked to a single pin can be more than one, this means that multiple detection devices have been positioned in the same place. In this case the displayed data represents the average of their measurements.

Through a long press on the pin, the user is directed to an additional screen capable of viewing the most recent history regarding the boards indicated in the pin.

On the map, in the event that the positioning of the device is active, a "man" icon is placed which indicates the current position of the user[fig. 3.13].

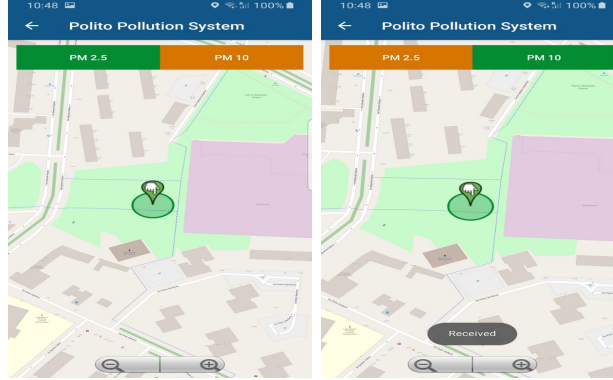


Figure 3.11: Pollution screen

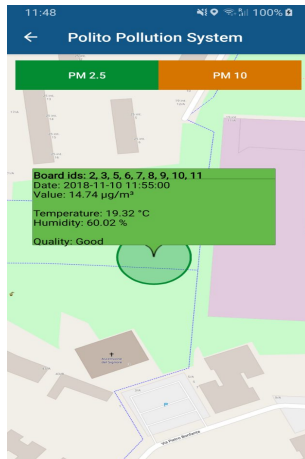


Figure 3.12: Pin informations

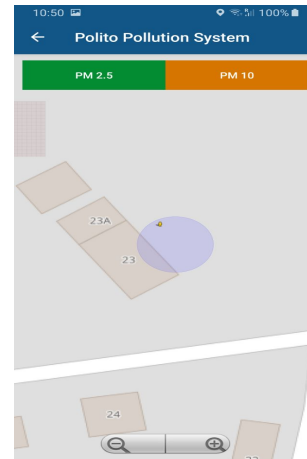


Figure 3.13: Current user position

3.2.5 Last 2 hour history screen

In this screen it is possible to consult the most recent history concerning the measurements performed by the boards contained in the pin pressed to access this screen.

When this view is started, the data regarding the first board in the list are immediately displayed. The story can be consulted through three linear graphs on the screen[fig. 3.14]. Each represents the history of the last two hours of measurements available on the server. The “Sensor chart” shows the quantity of polluting particles currently selected (pm 2.5 or pm 10) measured by the 4 sensors

on the detection board. The other two graphs respectively show the measurements regarding the environmental temperature and humidity. For each graph there are two buttons that respectively allow the download of the image of the graph itself and the download of the data displayed on the corresponding graph.

At the top of the screen, it is the possibility to select the board on which the data is to be displayed.

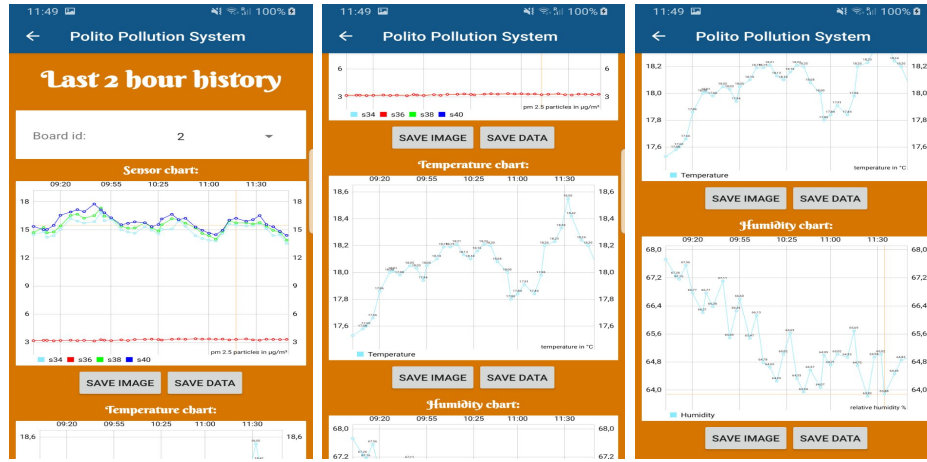


Figure 3.14: Last 2 hour history screen

3.2.6 General history screen

In this screen the user can consult the history of the measurements of each board present in the system. The view initially consists of the form for selecting the time range in which the data must be displayed together with the ability to choose the board id[fig. 3.15]. Once the "Search" button has been pressed, the graphs similar to those described in the previous point are displayed with the same download options[fig. 3.16].

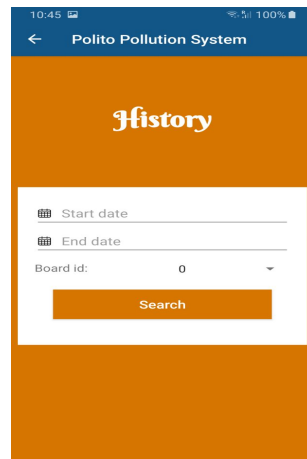


Figure 3.15: General history screen

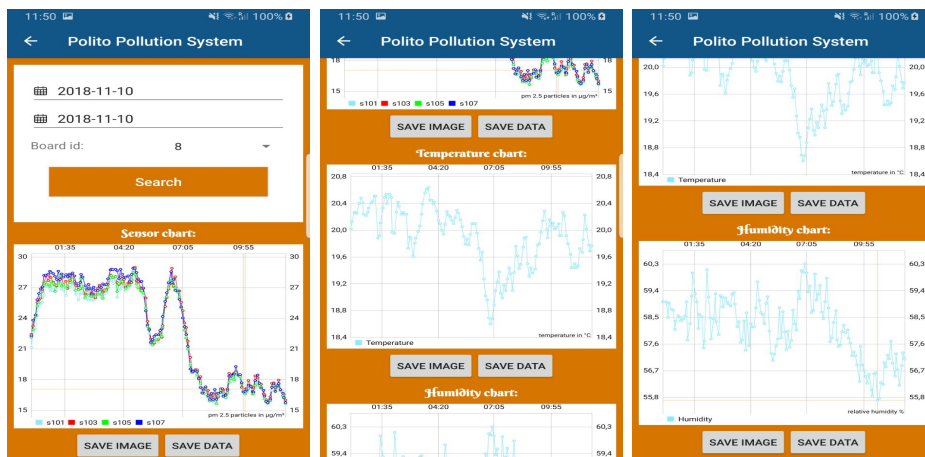


Figure 3.16: History charts

3.2.7 Profile showing screen

This screen was designed to be able to consult the data relating to the profile of the currently logged in user[fig. 3.17]. The view is only able to display the data but, by clicking on the button at the top right of the screen, allows the user to switch to the Activity for editing the data.

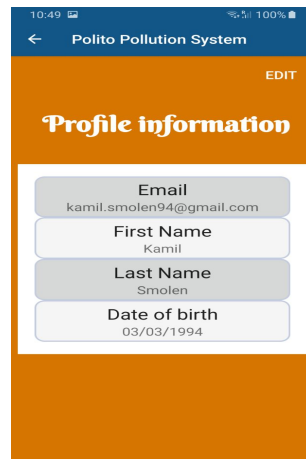


Figure 3.17: Profile showing screen

3.2.8 Profile editing screen

The screen allows the user to change the data relating to the currently logged in account. When the view is loaded, it pre-fills the form with the data currently present on the server in order to make changes faster[fig. 3.18]. To make the changes effective after modifying the form, the user need to press the "save" button located at the top right of the screen. Once pressed, the changes will be saved and the user will be redirected to the screen that displays the user profile.

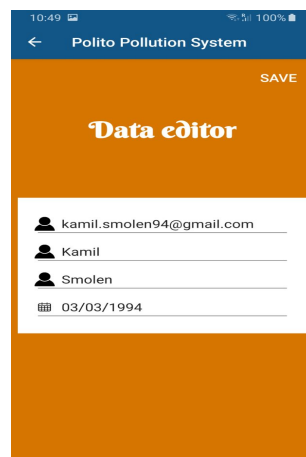


Figure 3.18: Editing profile screen

3.2.9 Tools screen

This Activity was designed to make the application more flexible and customizable. The screen is the only one accessible without logging in first. The view allows the user to change the web service address, which means that by entering an invalid or non-existent value, the application stops working correctly[fig. 3.19]. This option is available to allow the system admin of the current project to freely change the address and port of the server without having to change the source code of the Android application.

The screen also allows the user to choose whether, when loading the "Pollution" Activity, the map should be centered on the user's current position. Otherwise, the zoom is performed on Turin city center[fig. 3.19]. The choice was made possible as the service is available to every person registered who does not necessarily have to be present in Turin to consult the data, thus avoiding being located in an area of the world where the boards have not been positioned.

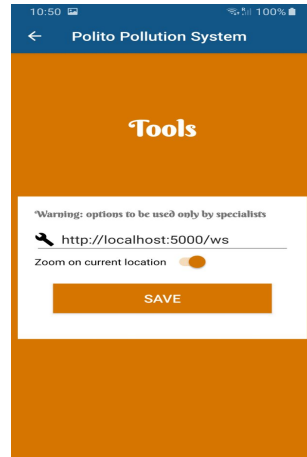


Figure 3.19: Tools screen

Chapter 4

Server implementation

This chapter describes the implementation details of the web service whose operation was presented in the previous chapter. Here all the source files that make up the server are analyzed, paying particular attention to the most important code parts. The program was written using the integrated development environment (IDE) called PyCharm Professional developed by JetBrains[21].

4.1 Modularity

The web service was written with the idea of being a modular project. This means that within it there is a common piece of code that starts the web service itself and manages the configuration of all modules. Each module is a separate part of the program containing code for carrying out specific functions of the web service described here. Thanks to the use of separate modules, the program is not strictly linked to a single implementation, as each module can be easily replaced by changing the configuration in the common code. This programming technique also allows to easily add new features to the web service without going into the existing code, just create a new module and configure it.

The modules have been implemented through the use of Blueprints which are object modules that can contain within themselves a set of operations that can be recorded on a Flask application[22].

The web service project is completely contained within the `politoweatherstation-backend` package which contains a common piece of code necessary for its configuration along with three main modules: `users`, `measures`, `ws_checking_system`[fig. 4.1]. Each part of this program will be described in detail in the following sections.

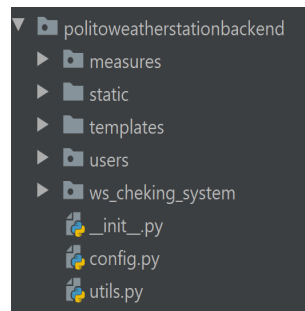


Figure 4.1: Web service project structure

4.2 Web service startup

The web service is started by executing the `run.py` script present within the PyCharm project. This script has the only purpose of starting the system through the initialization of the "app" object through a call to the main project package (`politoweatherstationbackend`) which returns a configured instance of it, that allows the service to function correctly. The code in figure below shows the `run.py` script.

```
1 from politoweatherstationbackend import create_app
2
3 app = create_app()
4
5 if __name__ == '__main__':
6     app.run(debug=False)
```

Listing 4.1: `run.py` script

The web service is also distributed together with a file called: "`requirements.txt`" which has the task of containing a complete list of external modules used within the project and which are necessary and must be installed in the server execution environment. To install these external modules just execute the command: "`pip install -r requirements.txt`" on the console. Every required external module will be described during its usage in the following sections. The image below shows the `requirements.txt` file.

```
1 APScheduler==3.6.3
2 bcrypt==3.1.7
3 Flask==1.1.2
4 Flask-API==2.0
5 Flask-APScheduler==1.11.0
6 Flask-Bcrypt==0.7.1
7 Flask-JWT==0.3.2
8 Flask-Mail==0.9.1
```

```
9 Flask-RESTful==0.3.8
10 Flask-SQLAlchemy==2.4.4
11 Flask-WTF==0.14.3
12 jsonschema==3.2.0
13 mysql-connector==2.2.9
14 pandas==1.0.5
15 PyJWT==1.7.1
16 pygeohash==1.2.0
17 SQLAlchemy==1.3.18
18 WTForms==2.3.1
```

Listing 4.2: requirements.txt file

4.3 Initialization and configuration

The web service is initialized and configured for proper operation through the use of two separate scripts: `__init__.py` and `config.py`. The first gives the task of returning a Flask type object that represents the entire application, while the second has the task of setting a set of configuration parameters.

4.3.1 `__init__.py`

This script consists only of a function called `create_app` which, as described above, gives the task of returning an object of type Flask, which represents the entire application and is executed by the `run.py` script. The source code of the script is as follows.

```
1 db = SQLAlchemy()
2 bcrypt = Bcrypt()
3 mail = Mail()
4
5 def create_app(config_class=Config):
6     app = Flask(__name__)
7     app.config.from_object(Config)
8
9     db.init_app(app)
10    bcrypt.init_app(app)
11    mail.init_app(app)
12
13    from politoweatherstationbackend.users.controllers import users,
14    Register, Login, Recover, Users, Logout
15    from politoweatherstationbackend.measures.controllers import
16    measures, Measure, Live, MeasureI
17    from politoweatherstationbackend.measures.utils import
18    insert_data_five, insert_data_hour
```

```

17 # Start users Blueprint
18 api = Api(users)
19 api.add_resource(Register, '/register')
20 api.add_resource(Login, '/login')
21 api.add_resource(Logout, '/logout')
22 api.add_resource(Recover, '/recover')
23 api.add_resource(Users, '/user/<user_id>')
24 # End users Blueprint
25
26 # Start measures Blueprint
27 api = Api(measures)
28 api.add_resource(Measure, '/measure/<pm>/<bs_id>')
29 api.add_resource(MeasureI, '/measure')
30 api.add_resource(Live, '/measure/<pmKind>')
31 # End measures Blueprint
32
33 app.register_blueprint(users, url_prefix='/ws')
34 app.register_blueprint(measures, url_prefix='/ws')
35
36 # start an async task to add data into five_min_avg table and
into hour_avg table
37 scheduler = BackgroundScheduler()
38 scheduler.add_job(func=insert_data_five, trigger="interval",
seconds=300) # maybe 360
39 scheduler.add_job(func=insert_data_hour, trigger="interval",
minutes=60) # maybe 61
40 scheduler.start()
41
42 Shut down the scheduler when exiting the app
43 atexit.register(lambda: scheduler.shutdown())
44
45 return app

```

Listing 4.3: `__init__.py` script

The script creates an instance of the Flask object, configuring it immediately through the use of the configuration object defined in the `config.py` script.

Following the application object, three external modules are added: SQLAlchemy, Bcrypt and flask_mail which respectively concern the simplified management of communication with the DB, the ability to encrypt data and finally the ability to send emails from the web service.

At this point, we move on to adding the internal modules to the service, that is, registering the Blueprints on the application object. To perform this action, each Blueprint is imported with all its functionalities and the APIs to which the module must respond are defined.

Finally, the script initializes and executes two asynchronous tasks that are necessary to keep the tables containing the averages of the measurements updated,

specifying the interruption of these tasks in the event of the application being closed.

4.3.2 config.py

Only one class called “Config” is defined in this script. The task of this class is to contain all the configuration variables necessary for each internal or external module used in the project. It can be immediately noted that for security reasons most of the set values are defined through the use of environment variables of the underlying operating system. The source code of this script is shown below.

```
1 import os
2
3
4 class Config:
5     SECRET_KEY = os.environ.get('SECRET_KEY')
6     APPLICATION_ROOT = '/ws'
7     DB_USER = os.environ.get('DB_USER')
8     DB_PASS = os.environ.get('DB_PASS')
9     DB_NAME = os.environ.get('DB_NAME')
10    DB_ADDRESS = os.environ.get('DB_ADDRESS')
11    SQLALCHEMY_DATABASE_URI = 'mysql+mysqlconnector://' + DB_USER +
12    ':' + DB_PASS + '@' + DB_ADDRESS + '/' + DB_NAME
13    SQLALCHEMY_TRACK_MODIFICATIONS = False
14    MAIL_SERVER = 'smtp.googlemail.com'
15    MAIL_PORT = 587
16    MAIL_USE_TLS = True
17    MAIL_USERNAME = os.environ.get('MAIL_USERNAME')
18    MAIL_PASSWORD = os.environ.get('MAIL_PASS')
19    BOARDS_JSON = os.environ.get('BOARDS_JSON')
20    ARPA_JSON = os.environ.get('ARPA_JSON')
21    START_FIRST_SENS_CAL = os.environ.get('START_FIRST_SENS_CAL')
22    END_FIRST_SENS_CAL = os.environ.get('END_FIRST_SENS_CAL')
```

Listing 4.4: config.py script

4.4 Uutils.py

This script contains some generic application features that cannot be strictly linked to a single module. The first function has the task of defining a decorator applicable to other functions, capable of verifying whether the request can be served as it is sent by an authorized user, i.e. a user who has successfully logged on to the system. This verification is based on the analysis of the token sent with the request. Its creation will be described in the following sections but, it is important to know that the unique identifier of the user and an access counter parameters are saved

inside it. This token is mixed with a secret key and encrypted. Each user who has logged in to the system receives one that must bind to each of his requests. For this reason, in this function, the token is immediately extracted and decrypted using the secret key of the server to verify its consistency. Once the consistency is confirmed, the token is checked for validity. The logic of the application involves sending an access counter within the token whose specific operation will be described later but, upon receipt, it must be equal to the value saved on the server. In case of any failed check, this function blocks the execution of the requested functionality and provides security to the service. The code is as follows.

```
1 def token_required(f):
2     @wraps(f)
3     def decorated(*args, **kwargs):
4         token = None
5
6         if 'x-access-token' in request.headers:
7             token = request.headers['x-access-token']
8
9         if not token:
10            return make_response({"msg": "Bad request"}, status.
HTTP_400_BAD_REQUEST)
11
12        try:
13            data = jwt.decode(token, current_app.config['SECRET_KEY
14            '])
15            current_user = User.query.filter_by(user_id=data['user_id
16            ']).first()
17            if current_user.counter != data['counter'] or not
18            current_user.confirmed or not current_user.active:
19                return make_response({"msg": "Unauthorized"}, status.
HTTP_401_UNAUTHORIZED)
20        except:
21            return make_response({"msg": "Bad request"}, status.
HTTP_400_BAD_REQUEST)
22
23        return f(current_user, *args, **kwargs)
24
25    return decorated
```

Listing 4.5: token_required funtion

The second function has the task of sending the confirmation email of the newly registered user. The message is sent through the use of the external module "flask_mail" and from the configured address of the "config.py" file. A specific token for this purpose is sent within the message to allow user confirmation. The token is inserted into a link which, once pressed by the user, redirects to a static page on the server that confirms this operation.

```
1 def send_confirm_email(user):
2     token = user.get_confirm_token()
3     msg = Message('User Confirmation Request',
4                   sender='polito.weather.station@gmail.com',
5                   recipients=[user.email])
6     msg.body = f'''To confirm your account, visit the following link:
7 {url_for('users.register', token=token, _external=True)}
8 '''
9     mail.send(msg)
```

Listing 4.6: send_confirm_email function

The last function is very similar to the previous one but has the task of sending the email with the link to allow the user's password to be changed. Also in this case a specific token is inserted within the link to allow this functionality. The code of this function is quite equal to the previous one described.

4.5 Templates and static content

The web service inside contains the source code of some static web pages together with a style file for their visualization. These pages can be visited through the links sent to the user via the emails described above. The features of these pages are used to display the result of operations carried out by the user such as confirmation of your email address or confirmation of the password change. Among the features of these pages there is also that of entering a new password. All pages have been written to have a common layout described in the "layout.html" file and add only the content necessary to provide their functionality. In the following the source code of the common layout is displayed, the source for the password change page and the source of a notification page of the action just carried out.

```
1 <!DOCTYPE html>
2 <html lang="en">
3 <head>
4     <!-- Required meta tags -->
5     <meta charset="utf-8">
6     <meta name="viewport" content="width=device-width, initial-scale
7 =1, shrink-to-fit=no">
8
9     <!-- Bootstrap CSS -->
10    <link rel="stylesheet" href="https://stackpath.bootstrapcdn.com/
bootstrap/4.3.1/css/bootstrap.min.css" integrity="sha384-
ggOyR0iXCbMQv3Xipma34MD+dH/1fQ784/j6cY/iJTQUOhcWr7x9JvoRxT2MZw1T"
crossorigin="anonymous">
    <link rel="stylesheet" type="text/css" href="{{ url_for('static',
filename='main.css') }}">
```

```

11  {% if title %}
12    <title>Polito Weather Station - {{ title }}</title>
13  {% else %}
14    <title>Polito Weather Station</title>
15  {% endif %}
16 </head>
17 <body>
18   <header class="site-header">
19     <nav class="navbar navbar-expand-md navbar-dark bg-steel fixed-
20 top">
21       <div class="container">
22         <h1 class="navbar-brand mr-4">Polito Weather Station</h1>
23         <button class="navbar-toggler" type="button" data-toggle="
collapse" data-target="#navbarToggle" aria-controls="navbarToggle"
24         aria-expanded="false" aria-label="Toggle navigation">
25           <span class="navbar-toggler-icon"></span>
26         </button>
27       </div>
28     </nav>
29   </header>
30   <main role="main" class="container">
31     <div class="row">
32       <div class="col-md-8">
33         {% block content %}{% endblock %}
34       </div>
35     </div>
36   </main>
37   <!-- Optional JavaScript -->
38   <!-- jQuery first, then Popper.js, then Bootstrap JS -->
39   <script src="https://code.jquery.com/jquery-3.3.1.slim.min.js "
integrity="sha384-q8i/X+965
DzO0rT7abK41JStQIAqVgRvZpbzo5smXKp4YfRvH+8abtTE1Pi6jizo "
crossorigin="anonymous"></script>
40   <script src="https://cdnjs.cloudflare.com/ajax/libs/popper.js
/1.14.7/umd/popper.min.js " integrity="sha384-
UO2eT0CpHqdSJK6hJty5KVphtPhzWj9WO1clHTMGa3JDZwrnQq4sF86dIHNDz0W1 "
crossorigin="anonymous"></script>
41   <script src="https://stackpath.bootstrapcdn.com/bootstrap/4.3.1/
js/bootstrap.min.js " integrity="sha384-
JjSmVgyd0p3pXB1rRibZUAYoIIy6OrQ6VrjIEaFf/nJGzIxFDs4x0xIM+B07jRM "
crossorigin="anonymous"></script>
42 </body>
43 </html>

```

Listing 4.7: layout.html file

```

1  {% extends "layout.html" %}

```

```

2 {% block content %}
3   <div class="content-section">
4     <form method="POST" action="">
5       {{ form.hidden_tag() }}
6       <fieldset class="form-group">
7         <legend class="border-bottom mb-4">Reset Password</
8         legend>
9         <div class="form-group">
10           {{ form.password.label(class="form-control-label
11           ") }}
12           {% if form.password.errors %}
13             {{ form.password(class="form-control form-
14             control-lg is-invalid") }}
15             <div class="invalid-feedback">
16               {% for error in form.password.errors %}
17                 <span>{{ error }}</span>
18               {% endfor %}
19             </div>
20           {% else %}
21             {{ form.password(class="form-control form-
22             control-lg") }}
23           {% endif %}
24         </div>
25         <div class="form-group">
26           {{ form.confirm_password.label(class="form-
27           control-label") }}
28           {% if form.confirm_password.errors %}
29             {{ form.confirm_password(class="form-control
30             form-control-lg is-invalid") }}
31             <div class="invalid-feedback">
32               {% for error in form.confirm_password.
33               errors %}
34                 <span>{{ error }}</span>
35               {% endfor %}
36             </div>
37           {% else %}
38             {{ form.confirm_password(class="form-control
39             form-control-lg") }}
40           {% endif %}
41         </div>
42       </fieldset>
43       <div class="form-group">
44         {{ form.submit(class="btn btn-outline-info") }}
45       </div>
46     </form>
47   </div>
48 {% endblock content %}

```

Listing 4.8: recover.html source


```
1 {% extends "layout.html" %}
2 {% block content %}
3     <h1>Your account has been successfully confirmed!</h1>
4 {% endblock content %}
```

Listing 4.9: user_confirmed.html file

4.6 Users module

This module contains all the code concerning user management. Each user is allowed to perform operations such as eg. login/logout or register for a new account. The module composed of 5 source scripts divided according to the functionality of the executed code. The script `__init__.py` is an empty source file as it only serves to make the project recognize that the “users” folder is a separate module. This module contains also:

- `controllers.py`
- `models.py`
- `forms.py`
- `schemas.py`

4.6.1 Controllers

In this file, the module is explicitly referred to be a Blueprint. The functionalities contained in this script concern the definition of the user resources accessible through the web service together with the functions that must be performed once the REST APIs registered to this module are called.

SYSTEM_ADMIN creation

At this point, a function is also defined for creating the user with the role of "SYSTEM-ADMIN" which is performed immediately before the first request received. The “`@users.before_app_first_request`” decorator indicates exactly this behavior of the function.

```
1 @users.before_app_first_request
2 def create_system_admin():
3     # check if the SYSTEM_ADMIN is already in the db
4     user = User.query.filter_by(email='admin').first()
5     if user:
6         return
```

```

7
8 # create the SYSTEM_ADMIN
9 hashed_password = bcrypt.generate_password_hash('admin').decode('
utf-8')
10 user = User(email='admin', password=hashed_password,
11             name='admin', surname='admin', birth='1990-01-01',
12             role='SYSTEM_ADMIN', confirmed=True, active=True,
counter=0,
13             reset_pass=False)
14 db.session.add(user)
15 db.session.commit()

```

Listing 4.10: create_system_admin function

Register resource

The “Register” class is defined in this file and is marked as a resource. This means that it is possible to define functions that can be contacted through the REST APIs. Two functions have been defined within it, one reachable through the POST method, has the task of creating a new user and saving his data in the DB by also sending the email to confirm the account. This procedure is started only if the request body conforms to the predefined JSON schema in the “schemas.py” file. Each user record within the DB has the following structure.

user_id 	bigint(20)
email 	varchar(120)
password	varchar(60)
name	varchar(30)
surname	varchar(30)
birth	date
role	varchar(30)
confirmed	tinyint(1)
active	tinyint(1)
counter	bigint(20)
reset_pass	tinyint(1)

Figure 4.2: User record structure

The confirmed field is used to indicate whether the user has confirmed its email, the active field indicates whether the user is currently active and therefore has access to all the features offered by the service. The counter represents the number of accesses performed by the user. This is for the login phase and is explained later. Finally, the reset_pass field indicates whether the user has started the password change procedure.

Below the source code of the first function is showed. It is possible to note that the password is hashed before being saved on the DB. For this task the “bcrypt” module is used.

```

1  def post(self):
2      data = request.get_json()
3      # check if there is a json data available in the request
4      if not data:
5          return make_response({"msg": "Bad request"}, status.
HTTP_400_BAD_REQUEST)
6      # validation of the json data against the correct json schema
7      try:
8          jsonschema.validate(data, register_schema)
9      except jsonschema.ValidationError:
10         return make_response({"msg": "Bad request"}, status.
HTTP_400_BAD_REQUEST)
11
12     # create a new user to be saved in the bd and send a
confirmation email
13     hashed_password = bcrypt.generate_password_hash(data['
password']).decode('utf-8')
14     user = User(email=data['email'], password=hashed_password,
15                 name=data['name'], surname=data['surname'], birth
=data['birth'],
16                 role='USER', confirmed=False, active=True,
counter=0,
17                 reset_pass=False)
18     db.session.add(user)
19     db.session.commit()
20     send_confirm_email(user)
21     return make_response({"msg": "Created Successfully"}, status.
HTTP_201_CREATED)

```

Listing 4.11: POST function

The second function that can be contacted through the GET method has the task of marking the user as confirmed = true. To do that it has to verify the validity of the token contained in the request.

```

1  def get(self):
2      token = request.args.get('token')
3      # check if the request contains the token
4      if token:
5          user = User.verify_confirm_token(token)
6          if user is None:
7              return make_response({"msg": "Bad request"}, status.
HTTP_400_BAD_REQUEST)
8          # in this case the user is not confirmed yet
9          if user == -1:
10             headers = {'Content-Type': 'text/html'}
11             return make_response(render_template('
user_already_confirmed.html', title='Success'),
12                                 status.HTTP_200_OK,

```

```

13                                     headers)
14         # set the user to confirmed
15         user.confirmed = True
16         db.session.commit()
17         headers = {'Content-Type': 'text/html'}
18         return make_response(render_template('user_confirmed.html
19 ', title='Success'), status.HTTP_200_OK, headers)
20         # if there is not a token in the request return error
21         return make_response({"msg": "Bad request"}, status.
HTTP_400_BAD_REQUEST)

```

Listing 4.12: GET function

Both functions respond with messages with standard http codes such as eg. “200 OK” or “400 Bad request” in case of success or failure.

Login resource

A resource class called “Login” is also defined in the file. Inside, only one function accessible via the POST method has been defined that allows the user to log in to the system.

```

1     class Login(Resource):
2         # http://127.0.0.1:5000/login
3         # used to login into the service
4         def post(self):
5             data = request.get_json()
6             # check if there is a json data with the request
7             if not data:
8                 return make_response({"msg": "Bad request"}, status.
HTTP_400_BAD_REQUEST)
9
10            # validate the json data against the correct json schema
11            try:
12                jsonschema.validate(data, login_schema)
13            except jsonschema.ValidationError:
14                return make_response({"msg": "Bad request"}, status.
HTTP_400_BAD_REQUEST)
15
16            user = User.query.filter_by(email=data['email']).first()
17
18            # check if the user is confirmed and active
19            if not user or not user.confirmed or not user.active:
20                return make_response({"msg": "Bad request"}, status.
HTTP_400_BAD_REQUEST)
21
22            # check is the users password is correct
23            if not bcrypt.check_password_hash(user.password, data['
password']):

```

```

24         return make_response({"msg": "Unauthorized"}, status.
HTTP_401_UNAUTHORIZED)
25
26         # create the jwt token that have to be add to any other
request that requires to be authenticated
27         # this contains a counter that will be different at the next
login just to avoid the re-usage of old tokens
28         token = jwt.encode({'user_id': user.user_id, 'counter': user.
counter}, current_app.config['SECRET_KEY'])
29
30         content = jsonify({'token': token.decode('UTF-8')})
31         return make_response(content, status.HTTP_200_OK)

```

Listing 4.13: Login resource

As you can see from the code, the function, after extracting the email and password from the body of the request, proceeds with the verification of the existence of this user and the verification of his active and confirmed status. If the checks are successfully passed, the token is created which must be attached to the user's next requests to confirm his identity. This token is encrypted through the "jwt" library configured when the application is started. Inside, the information on the user ID and the number of accesses, necessary for its verification, are saved.

The token as it was designed does not have a validity period, to give the user the opportunity to remain logged into the system even after a long period of non-use of the Android client application. This means that to invalidate the token, i.e. log out, it is necessary to change a data present on the server in order to make its verification no longer valid. This data is the "counter" field which is increased at each logout. At this point, if a request arrives with a token whose "counter" field does not match the one saved on the server, the function will not be executed.

Logout resource

This resource is declared in the same way as the resources previously described, inside it contains only one function accessible through the POST method. The "@token_required" decorator indicates that for its execution it is necessary to have a valid token. Among the most important points of the function it is important to note that the user's counter is incremented to invalidate the token and log out.

```

1     counter = current_user.counter
2     counter = counter + 1
3     current_user.counter = counter

```

Listing 4.14: counter incrementation

Recover resource

This resource class contains two functions, one accessible through the POST method to be able to request the password change, and one accessible through the GET method which allows the actual password change. The operation of the two functions is similar to the operation seen during user registration and email confirmation (Register resource section). Also in this case, the first function is intended to send the user the email with a token. The second displays the static page for changing the password and saves the changed data within the DB. The only peculiarity of this function is that the data is taken from an HTML form as the new password is entered within a web page.

```
1 form = RecoverPasswordForm()
2     if form.validate_on_submit():
3         hashed_password = bcrypt.generate_password_hash(form.
password.data).decode('utf-8')
4         user.password = hashed_password
5         user.reset_pass = False
6         db.session.commit()
```

Listing 4.15: getting data from the form

Users resource

This resource was created to allow users to obtain their account data along with the ability to change them. A function accessible with the GET method returns the data of the user specified in the address of the request itself. The method does not need any further explanation as the code has no particularities. There are also two other functions in the resource that are accessible with the POST method and the PUT method respectively. The execution of the first is allowed to every user normally logged in to the system and allows you to change user data such as: email, name, surname and date of birth. The second instead gives the possibility to change the user's role together with the possibility to change the active status. This second method is accessible only to users with a higher role than the normal user as it is a very risky operation from the point of view of the security of the system itself. The code of the functions described here is not shown as it does not show anything that has not already been seen before.

4.6.2 Models

This script has the purpose of defining the class of objects on which the data coming from and to the DB will be mapped. This mechanism is the so-called "object-relational mapping" and is implemented thanks to the use of the external SQLAlchemy module. The so mapped are accessible through simple objects and

editable by changing the properties. The next piece of code how that mapping is done.

```

1 class User(db.Model):
2     __tablename__ = 'user'
3     user_id = db.Column(db.BigInteger, primary_key=True)
4     email = db.Column(db.String(120), nullable=False, unique=True)
5     password = db.Column(db.String(60), nullable=False)
6     name = db.Column(db.String(30), nullable=False)
7     surname = db.Column(db.String(30), nullable=False)
8     birth = db.Column(db.Date, nullable=False)
9     role = db.Column(db.String(30), nullable=False)
10    confirmed = db.Column(db.Boolean, nullable=False)
11    active = db.Column(db.Boolean, nullable=False)
12    counter = db.Column(db.BigInteger, nullable=False)
13    reset_pass = db.Column(db.Boolean, nullable=False)

```

Listing 4.16: user mapping data class

In addition to containing the same properties contained within each user record in the DB, the class also contains two functions for creating the reset and confirmation token. Both code internally only the user id as in this case the counter is not necessary. The first token is also created with a validity period of 30 minutes after which the request to change the password is no longer feasible.

```

1     def get_reset_token(self, expires_sec=1800):
2         s = Serializer(current_app.config['SECRET_KEY'], expires_sec)
3         return s.dumps({'user_id': self.user_id}).decode('utf-8')

```

Listing 4.17: get token function example

Finally, the class contains two other static methods to allow the verification of the two tokens described above.

```

1 @staticmethod
2     def verify_reset_token(token):
3         s = Serializer(current_app.config['SECRET_KEY'])
4         try:
5             user_id = s.loads(token)['user_id']
6             user = User.query.filter_by(user_id=user_id).first()
7             if not user.reset_pass or not user.confirmed or not user.
active:
8                 return -1
9         except:
10             return None
11         return User.query.get(user_id)

```

Listing 4.18: verify token function example

4.6.3 Forms

This script defines the class that models the Form for entering the necessary data when changing the password. The Form is defined thanks to the use of the "FlaskForms" module which also offers the validation functionality of the same thus verifying in a very simple way the correctness of the data entered. The source code of the script described here is visible below.

```
1 class RecoverPasswordForm(FlaskForm):
2     password = PasswordField('Password', validators=[DataRequired()])
3     confirm_password = PasswordField('Confirm_Password',
4                                     validators=[DataRequired(),
5                                     EqualTo('password')])
6     submit = SubmitField('Reset Password')
```

Listing 4.19: Form.py content

4.6.4 Schemas

The script has the sole task of defining some variables containing the JSON schema to which the data coming from the requests within the body must be subject. As can be seen from the code shown in the previous sections, each request before being processed checks the validity of the data received by validating them with the schemes defined here. Below a variable containing a reference schema is showed.

```
1 users_post_schema = {
2     'properties': {
3         'email': {'type': 'string', 'pattern': "^[a-zA-Z0-9_\\-\\.]+@"
4         ([a-zA-Z0-9_\\-\\.]+)\\.([a-zA-Z]{2,5})$"},
5         'name': {'type': 'string'},
6         'surname': {'type': 'string'},
7         'birth': {'type': 'string', 'pattern': "^\\d
8         {4}\\-(0?[1-9]|1[012])\\-(0?[1-9]|12[0-9]|3[01])$"}
9     }
```

Listing 4.20: example of a validation schema

4.7 Measures module

This module contains all the code concerning the management of the measures collected by the boards distributed for the city of Turin. It allows both to upload the data collected within the DB and to query the database to obtain the measurements ready for visualization on the client application. Among the most important tasks

of this module is that of keeping updated the tables containing the averages of the measurements made.

The module interacts with four tables of the database:

- `measure_table` that contains all the measures collected by the system.




1	measureID		bigint(20)
2	sensorID		int(10)
3	timestamp		int(10)
4	data		float
5	geoHash		varchar(12)
6	altitude		float

Figure 4.3: `measure_table` structure

- `board_table` that contains the data about all the detection boards distributed in the city of Turin


1	boardID		int(10)
2	vendor		varchar(60)
3	model		varchar(60)
4	serialNumber		varchar(60)

Figure 4.4: `board_table` structure

- `five_min_avg` that contains the average data, made every five minute from the measurements collected in the `measure_table`

1	id 🔑	bigint(20)
2	date	date
3	hour	int(2)
4	minute	int(2)
5	sensorID	int(10)
6	value	double
7	max	float
8	min	float
9	std	float
10	geoHash	varchar(12)
11	altitude	float

Figure 4.5: five_min_avg table structure

- hour_avg that contains the average data, made every hour from the measurements collected in the measure_table

1	id 🔑	bigint(20)
2	date	date
3	hour	int(2)
4	sensorID 🔑	int(10)
5	data	double

Figure 4.6: hour_avg table structure

The module is divided, in a similar way to those of the user one, in several source files:

- controllers.py
- models.py
- schemas.py
- utils.py

4.7.1 Controllers

This source script contains the definition of the resources contained in this module, together with the functions performed following the calls to the connected REST APIs. In addition, some functions are defined for the calculation of the regression coefficients necessary for the calibration of the data, together with a function that applies the calibration to the required measurements.

caption=startup_sensor_calibration()

The function is executed only once before receiving any request from the web service. This operation is specified by using the "@before_app_first_request" decorator. The task of this function is to start the process of calculating the regression coefficients necessary for data calibration by running the "first_sensor_calibration" function in a separate thread so as not to hinder other requests received from the server during this calculation.

```
1 @measures.before_app_first_request
2 def startup_sensor_calibration():
3     # prepair the data for the calibration
4     start = current_app.config['START_FIRST_SENS_CAL']
5     end = current_app.config['END_FIRST_SENS_CAL']
6     boards = current_app.config['BOARDS_JSON']
7     arpa = current_app.config['ARPA_JSON']
8     user = current_app.config['DB_USER']
9     pwd = current_app.config['DB_PASS']
10    addr = current_app.config['DB_ADDRESS']
11    name = current_app.config['DB_NAME']
12
13    # starts a thread that calibrates the sensors the first time
14    t = Thread(target=first_sensor_calibration, args=(start, end,
15    boards, arpa, user, pwd, addr, name))
16    # you have to set daemon true to not have to wait for the process
17    # to join
18    t.daemon = True
19    t.start()
20
21    print('Calibrating the sensors!')
```

Listing 4.21: startup_sensor_calibration function

first_sensor_calibration()

In this function, first of all it is specified through the use of a boolean global variable that the process of the first calculation of the coefficients is in progress. So if any data requests arrive at the server during this process, they will receive a negative response until it is completed.

```
1     if not calibrating_sensors:
2         calibrating_sensors = True
3     else:
4         return
```

Listing 4.22: setting global variable

At this point a global object "x" is initialized as "SensorAnalysis", a class imported from the "ws_analysis" library developed at the Politecnico di Torino. The object of this type has the task of containing the regression coefficients for each existing data detection sensor. During the initialization of the object, a time range is specified for which the coefficients must be calculated. The "load()" method allows you to load the data necessary for the calculation from the DB.

```

1      x = SensorAnalysis(start, end,
2                          board=boards,
3                          arpa_station=arpa)
4
5      x.db_setup(user, '', addr, name)
6      # connection string to be used is real environment and not
7      localhost!
8      # x.db_setup(user, pwd, addr, name)
9      x.load_data(load_from_db=True)

```

Listing 4.23: x object initialization

At this point, by calling the “calibrate_sensor_lr” function and passing each available sensor as a parameter, the relative coefficients are calculated.

```

1 pmKinds = ['pm25', 'pm10']
2 for pmKind in pmKinds:
3     for board in boardNumbers:
4         for s in board_list[board][pmKind]:
5             x.calibrate_sensor_lr(s, 0, start_dt_calibration,
6                                   end_dt_calibration, use_temp=True, use_rh=True,
7                                   threshold=150.0, report=False)

```

Listing 4.24: regression coefficients calculation

apply_calibration()

The function has the task of calibrating the data received for a sensor by applying the regression coefficients contained within the global object "x". The procedure is performed thanks to the functionality offered by the “ws_analysis” library.

```

1 def apply_calibration(s_name, cal_type, data):
2     r = pd.DataFrame()
3     r['date'] = data['date']
4     r['hour'] = data['hour']
5     r['minute'] = data['minute']
6
7     data.dropna(inplace=True)
8     data.reset_index(drop=True, inplace=True)
9
10    regressor = x.sensors[s_name].s_regressor[cal_type.value]

```

```

11
12     if regressor == 0:
13         print("ERR: Sensor is not calibrated using required type " +
cal_type.name)
14         print("        Returning not calibrated data")
15         return data['sens']
16
17     if data.empty:
18         print("No data available")
19         data['cal'] = data['sens']
20     else:
21
22         if cal_type == CalType.lrth:
23             data['cal'] = regressor.predict(data[['sens', 'temp', 'rh
']]')
24         elif cal_type == CalType.lrt:
25             data['cal'] = regressor.predict(data[['sens', 'temp']])
26         elif cal_type == CalType.lrh:
27             data['cal'] = regressor.predict(data[['sens', 'rh']])
28         elif cal_type == CalType.rf:
29             data['cal'] = pd.Series(
30                 regressor.predict(data[['sens', 'temp', 'rh']]))
31         else:
32             print("ERR: invalid calibration type")
33             print("        Returning not calibrated data")
34             data['cal'] = data['sens']
35
36     return data['cal']

```

c

MeasureI resource

This resource class allows to load measurements into the DB as well as allowing to re-calculate the regression coefficients in a different time range from the one calculated at the service startup.

The function performed after receiving the request with the PUT method allows the client to load a set of measurements within the "measure_table" table. The data before being definitively entered into the DB are checked by the "check_function" offered by the "ws_checking_system" module which has the task of analyzing the correctness of the measurements received. If the answer is "True", the data are loaded as they are in to the target table, if instead a correct data set is obtained as an answer, at this point the result obtained is saved.

```

1 @token_required
2 def put(current_user, self):
3     inserted = 0

```

```

4      data = request.get_json()
5      # check if there is a json data available in the request
6      if not data:
7          return make_response({"msg": "Bad request"}, status.
HTTP_400_BAD_REQUEST)
8      # validation of the json data against the correct json schema
9      try:
10         jsonschema.validate(data, measure_schema)
11     except ValidationError:
12         return make_response({"msg": "Bad request"}, status.
HTTP_400_BAD_REQUEST)
13
14     res = check_function(data['data_block'])
15     if res == True:
16         # add the data to the db
17         for rec_measure in data['data_block']:
18             measure = MeasureDB(sensorID=rec_measure['sensorID'],
19 timestamp=rec_measure['timestamp'],
20 data=rec_measure['data'], geoHash
=rec_measure['geoHash'],
21 altitude=rec_measure['altitude'])
22         db.session.add(measure)
23         inserted += 1
24         db.session.commit()
25     else:
26         # add the data like it was returned from the check
27         function to the db because they are damaged or wrong
28         for rec_measure in res['data_block']:
29             measure = MeasureDB(sensorID=rec_measure['sensorID'],
30 timestamp=rec_measure['timestamp'],
31 data=rec_measure['data'], geoHash
=rec_measure['geoHash'],
32 altitude=rec_measure['altitude'])
33         db.session.add(measure)
34         inserted += 1
35         db.session.commit()
36
37     return make_response({"msg": "OK", "inserted": inserted},
status.HTTP_200_OK)

```

Listing 4.25: put method

The post method, on the other hand, allows you to recalculate the regression coefficients for each sensor. The code inside is identical to that of the "first_sensor_calibration" function with the difference that the time range is received from the body of the POST request. The method is also accessible only to users with the role of admin or system_admin.

Measure resource

This resource class, thanks to the method defined within it, allows to request measurements from the DB.

The get function allows the client to request calibrated data relating to measurements by querying the `five_min_avg` table. Based on how the request is structured, it is possible to get all the data in a selected time range, or data only for a specific board/sensor. In addition, there is also the possibility of obtaining the last two hours of measurement starting from a date and time chosen for a selected board.

The request must be sent to the address that ends with `/measure/<pm_Kind>/<board/sensor_id>`. Thus it is possible to specify immediately what type of partelless and to which board the required data must belong. Through the use of specific headers it is possible to specify the type of filtering of the measures. The "board" header allows the client to specify whether filtering should be performed on the `board_id` entered in the address, the "filter" header specifies the same action on the `sensor_id`. The "last-two" headers together with the "from" headers allow the client to specify that the request concerns the last two hours of measurements starting from the time contained in the "from" attribute.

The get method, after checking the semantic correctness of the request, initializes a data structure capable of containing the measurements requested by the client.

```
1 queried_data = pd.DataFrame()
2
3 # create index for desired date range (hour frequency)
4 dti = pd.date_range(start=start, end=end, freq='5min')
5
6 # create data frame with date and hour and minute
7 queried_data['date'] = pd.Series(dti.date)
8 queried_data['hour'] = pd.Series(dti.hour)
9 queried_data['minute'] = pd.Series(dti.minute)
10 queried_data.date = pd.to_datetime(queried_data.date)
```

Listing 4.26: data structure initialization

At this point, for each existing board, the data regarding measurements, temperature and humidity are requested from the DB.

```
1 # load pmKind
2 for s in x.board_list[i][pm]:
3     # check if filtering on sensor and if it is the selected one
4     if do_any_filter == 'True' and filter_on_board == 'False' and not
       str(s) == bs_id:
5         continue
6
7     one_taken = True
8
9     s_name = 's' + str(s)
```

```

10
11     if last_two == 'True':
12         mysql_stm = "SELECT date, hour, minute, value FROM
WEATHER_STATION.FIVE_MIN_AVG WHERE date = '" + \start + "' AND
hour >= '" + hour + "' AND sensorID = " + \str(s) + ";"
13     else:
14         mysql_stm = "SELECT date, hour, minute, value FROM
WEATHER_STATION.FIVE_MIN_AVG WHERE date >= '" + \start + "' AND
date <= '" + end + "' AND sensorID = " + \str(s) + ";"
15
16         try:
17             print("\t – getting data for sensor " + str(s))
18             cursor.execute(mysql_stm)
19             records = cursor.fetchall()
20         except:
21             print("Unable to get data from DB")
22             return
23
24         hdf = pd.DataFrame()
25         hdf['date'] = pd.Series((pd.to_datetime(r[0]) for r in
records))
26         hdf['hour'] = pd.Series(r[1] for r in records)
27         hdf['minute'] = pd.Series(r[2] for r in records)
28         hdf[s_name] = pd.Series(r[3] for r in records)
29         queried_data = pd.merge(queried_data, hdf, left_on=[
30             'date', 'hour', 'minute'], right_on=['date', 'hour', '
minute'], how="outer")
31
32         # if i have taken some data and i filter on sensors break the
for because i already have the data
33         if do_any_filter == 'True' and filter_on_board == 'False':
34             break

```

Listing 4.27: loading pmKind data from the DB

The loading of temperature and humidity data is made in the same way as seen for the pmKind measures.

Once all the data has been loaded, the calibration process begins. For each measurement requested by the client, the “apply_calibtation” function is performed.

```

1 for i in range(len(x.board_list)):
2     if do_any_filter == 'True' and filter_on_board == 'True' and not
str(x.board_list[i]['board_id']) == bs_id:
3         continue
4
5     for s in x.board_list[i][pm]:
6         if do_any_filter == 'True' and filter_on_board == 'False' and
not str(s) == bs_id:
7             continue
8

```



```

9      s_name = 's' + str(s)
10     sensor = x.sensors[s_name]
11     t_name = 's' + \
12         str(x.board_list[sensor.get_b_id()][ 'temp' ][0])
13     h_name = 's' + str(x.board_list[sensor.get_b_id()][ 'rh' ][0])
14     # a_name = 's' + str(x.arpal_list[0][json_data['pmKind']][0])
15
16     data = pd.DataFrame()
17     data['date'] = queried_data['date']
18     data['hour'] = queried_data['hour']
19     data['minute'] = queried_data['minute']
20     data['sens'] = queried_data[s_name]
21     data['temp'] = queried_data[t_name]
22     data['rh'] = queried_data[h_name]
23     # data['arpa'] = queried_data[a_name]
24
25     if sensor.is_cal():
26         data['cal'] = apply_calibration(s_name, CalType.lrth,
data)
27         s_name_cal = s_name + '_cal'
28         data = data.rename(index=str, columns={'sens': s_name, '
cal': s_name_cal})
29         queried_data = queried_data.merge(data[['date', 'hour', '
minute', s_name_cal]],
30                                           left_on=['date', 'hour
', 'minute'],
31                                           right_on=['date', 'hour
', 'minute'],
32                                           how="left")
33     else: # sensor is not calibrated
34         print('Sensor not calibrated' + str(sensor.get_s_id()))

```

Listing 4.28: data calibration

At this point the data is converted to JSON format and sent as a response to the client.

```

1 queried_data['date'] = queried_data['date'].dt.strftime('%Y-%m-%d')
2 returnData = queried_data.to_json(orient='records')
3 return make_response(returnData, status.HTTP_200_OK)

```

Listing 4.29: returning data

Live resource

This resource class has an internal get method that can return data containing the most recent measurements for each sensor.

Once the semantic correctness of the request has been verified, a for loop cycles through all the existing boards and sensors, querying the DB to obtain the most

recent measurement available for each of them. Immediately after the request for the data, these are calibrated and added to a data structure which at the end of the cycle is converted into JSON format and sent as a response to the client.

```

1 # used to get the last available data for each sensor, it is the live
  display data endpoint
2 @token_required
3 def get(current_user, self, pmKind):
4     global x
5     # check if the sensors calibration is already done almost one
    time
6     if not sens_calibrated:
7         return make_response({"msg": "Conflict: Sensors not
    calibrated yet"}, status.HTTP_409_CONFLICT)
8
9     # check if the pmKind is a reasonable value
10    if not pmKind == "pm10" and not pmKind == "pm25":
11        return make_response({"msg": "Bad request"}, status.
    HTTP_400_BAD_REQUEST)
12
13    # here starts the code for data calibration
14
15    # create a connection cursor for quering the data from the db
16    connection = db.engine.raw_connection()
17    cursor = connection.cursor(buffered=True)
18
19    # object to contain all the data that will be read from the db
    and transformed to JSON
20    result = {'boards': []}
21
22    # loads the data from the db in order to perform calibration on
    them
23    for i in range(len(x.board_list)):
24        board = {'id': i, 'sensors': []}
25        # load pmKind
26        for s in x.board_list[i][pmKind]:
27            sss = {}
28            s_name = 's' + str(s)
29            mysql_stm = "SELECT * FROM WEATHER_STATION.FIVE_MIN_AVG
    WHERE id = (SELECT MAX(id) FROM WEATHER_STATION.FIVE_MIN_AVG WHERE
    sensorID = " + \
30                str(s) + ");"
31            try:
32                print("\t - getting data for sensor " + str(s))
33                cursor.execute(mysql_stm)
34                records = cursor.fetchall()
35            except:
36                print("Unable to get data from DB")
37            return

```

```

38
39         # save the queried data into a dataframe
40         hdf = pd.DataFrame()
41         hdf['sens_id'] = pd.Series(r[4] for r in records)
42         hdf['date'] = pd.Series(r[1] for r in records)
43         hdf['hour'] = pd.Series(r[2] for r in records)
44         hdf['minute'] = pd.Series(r[3] for r in records)
45         hdf['sens'] = pd.Series(r[5] for r in records)
46         hdf['max'] = pd.Series(r[6] for r in records)
47         hdf['min'] = pd.Series(r[7] for r in records)
48         hdf['std'] = pd.Series(r[8] for r in records)
49         hdf['geoHash'] = pd.Series(r[9] for r in records)
50
51         # if there is a result i query the db also for the
52         temperature and humidity, i need them for calibration
53         if not hdf.empty:
54             sensor = x.sensors[s_name]
55
56             # query the sensors temperature
57             mysql_stm = "SELECT * FROM WEATHER_STATION.
58 FIVE_MIN_AVG WHERE date = '" + \
59             str(hdf.iloc[0]['date']) + "' AND hour =
60             '" + str(
61                 hdf.iloc[0]['hour']) + "' AND minute = '" + str(
62                 hdf.iloc[0]['minute']) + "' AND sensorID = " + \
63             str(x.board_list[sensor.get_b_id()]['temp
64             '][0]) + ";"
65             try:
66                 print("\t - getting data for sensor " + str(s))
67                 cursor.execute(mysql_stm)
68                 records = cursor.fetchall()
69             except:
70                 print("Unable to get data from DB")
71                 return
72
73             # add the temperature to the data frame
74             hdf['temp'] = pd.Series(r[5] for r in records)
75
76             # query the sensors humidity
77             mysql_stm = "SELECT * FROM WEATHER_STATION.
78 FIVE_MIN_AVG WHERE date = '" + str(
79                 hdf.iloc[0]['date']) + "' AND hour = '" + \
80                 str(hdf.iloc[0]['hour']) + "' AND minute
81             = '" + str(
82                 hdf.iloc[0]['minute']) + "' AND sensorID = " + \
83                 str(x.board_list[sensor.get_b_id()]['rh
84                 '][0]) + ";"
85             try:

```

```

79         print("\t – getting data for sensor " + str(s))
80         cursor.execute(mysql_stm)
81         records = cursor.fetchall()
82     except:
83         print("Unable to get data from DB")
84         return
85
86     # add the humidity to the data frame
87     hdf['rh'] = pd.Series(r[5] for r in records)
88
89     # if the selected sensor has already the calibration
coefficients    # calibrate i apply the calibration
90     if sensor.is_cal():
91         hdf['cal'] = apply_calibration(s_name, CalType.
lrth , hdf)
92     else: # sensor is not calibrated
93         print('Sensor not calibrated' + str(sensor.
get_s_id()))
94
95     # save the data frame into an object
96     sss['id'] = str(hdf.iloc[0]['sens_id'])
97     sss['date'] = str(hdf.iloc[0]['date'])
98     sss['hour'] = str(hdf.iloc[0]['hour'])
99     sss['minute'] = str(hdf.iloc[0]['minute'])
100    sss['value'] = str(hdf.iloc[0]['sens'])
101    sss['max'] = str(hdf.iloc[0]['max'])
102    sss['min'] = str(hdf.iloc[0]['min'])
103    sss['std'] = str(hdf.iloc[0]['std'])
104
105    # add the sensor object to a board object
106    board['sensors'].append(sss)
107    board['temp'] = str(hdf.iloc[0]['temp'])
108    board['rh'] = str(hdf.iloc[0]['rh'])
109    lat_long = pgh.decode(str(hdf.iloc[0]['geoHash']))
110    board['lat'] = str(lat_long[0])
111    board['long'] = str(lat_long[1])
112    # add the board to the result object
113    result['boards'].append(board)
114
115    # convert the result into json
116    result_JSON = json.dumps(result)
117
118    # response with json body
119    return make_response(result_JSON, status.HTTP_200_OK)

```

Listing 4.30: get method of the Live resource

Board resource

This resource class through the only get method inside it is able to return to the client the list of all the boards available within the DB.

```
1 class Board(Resource):
2     @token_required
3     def get(current_user, self):
4         boards = BoardDB.query.with_entities(BoardDB.boardID).all()
5         result = []
6
7         for board in boards:
8             result.append(board.boardID)
9
10        # convert the result into json
11        result_JSON = json.dumps(result)
12
13        print(result_JSON)
14
15        # response with json body
16        return make_response(result_JSON, status.HTTP_200_OK)
```

Listing 4.31: Board resource class

4.7.2 Models

This source file contains the definition of two classes that represent the "measure_table" and "board_table" tables within the DB. Thanks to their use it is possible to interact with the database by querying and modifying the corresponding tables thanks to the application of the "object-relational mapping" offered by the "SQLAlchemy" library.

```
1 class MeasureDB(db.Model):
2     __tablename__ = 'measure_table'
3     measureID = db.Column(db.BigInteger, primary_key=True)
4     sensorID = db.Column(db.Integer, nullable=False)
5     timestamp = db.Column(db.Integer, nullable=False)
6     data = db.Column(db.Float, nullable=False)
7     geoHash = db.Column(db.Integer, nullable=False)
8     altitude = db.Column(db.Float, nullable=True)
9
10
11 class BoardDB(db.Model):
12     __tablename__ = 'board_table'
13     boardID = db.Column(db.BigInteger, primary_key=True)
14     vendor = db.Column(db.String(60), nullable=False)
15     model = db.Column(db.String(60), nullable=False)
16     serialNumber = db.Column(db.String(60), nullable=False)
```

Listing 4.32: models.py file

4.7.3 Schemas

Within this file it is specified how the data sent to the server must be structured within the body of the http requests directed to this module. Below it can be seen an example of such structure.

```

1 post_measure_schema = {
2     'required': ['start', 'end'],
3     'properties': {
4         'start': {'type': 'string', 'pattern': "^(2[0-9]{3})
      -(0[1-9]|1[012])-(0[1-9]|12)[0-9]|3[01])$"},
5         'end': {'type': 'string', 'pattern': "^(2[0-9]{3})
      -(0[1-9]|1[012])-(0[1-9]|12)[0-9]|3[01])$"}
6     }
7 }
```

Listing 4.33: schema example

4.7.4 Utils

In this source file, the functions for entering data into the tables containing the averages of the measurements have been specified. These functions run every 5 minutes or every hour, respectively, depending on which table they are entering data into. Their use has been declared in the project initialization source file and both are started within separate threads.

In the following, only the “insert_data_five” function is described as the work of the second function is the same by averaging over a wider time range.

The "insert_data_five" function after establishing the connection with the DB starts a for loop on each sensor existing within the system.

In this cycle, the code first establishes in which time range the average of the measurements should be performed. The start date is that of the last record for the sensor analyzed within the five_min_avg table while the end date is the multiple of 5 minutes closest to the date of execution of this function.

```

1 # get the last date and time inserted in the five_min_avg table
2 statement = "SELECT * FROM five_min_avg WHERE sensorID = :y ORDER BY
      id DESC LIMIT 1"
3 query = conn.execute(text(statement), y=s)
4 # save the result into an object which at index 0 contains the date,
      at 1 the hour and at 2 the minutes
5 result = query.first()
```

```

6
7 # creating the start_date value
8 # check if the datetime is valid because if the table is empty there
   will be not be a valid value of start_date
9 # insert a default one
10 if not result:
11     # start_date = pd.to_datetime('2019-06-01 00:00:00').timestamp()
12     # it will take too much time, it is better to run the script
   manually which manually specified periods of time
13     print("There is no data in the five_min_avg table so it is better
   to run the script manually for sensor: " + s)
14     continue
15 else:
16     start = datetime.datetime.strptime(str(result[1]) + ' ' + str(
   result[2]) + ':' + str(result[3]) + ':00',
17                                         "%Y-%m-%d %H:%M:%S")
18     start_date = start + datetime.timedelta(minutes=5)
19     start_date = start_date.timestamp()
20
21 # creating the end_date value
22 # take the current time
23 t = datetime.datetime.now()
24 # check if the time is multiple of 5 min
25 m = t.minute
26 left = 5 - (m % 5)
27 # if is not round the time to the previous entire five min time
28 if left != 5:
29     end = t - datetime.timedelta(minutes=(5 - left))
30 else:
31     end = t
32
33 # create the end datetime for query the database
34 # the time must be like: 14:54:59
35 end_date = (end - datetime.timedelta(minutes=1)).strftime("%Y-%m-%d %
   H:%M:59")
36 end_date = pd.to_datetime(str(end_date)).timestamp()

```

Listing 4.34: time range calculation

At this point the data is requested from the `measure_table` table in the specified time range calculated previously.

```

1 # query that takes all the data from measure_table in the selected
   date and time
2 sqlQuery = '''SELECT *
3     FROM measure_table
4     WHERE timestamp >= %s
5     AND timestamp <= %s
6     AND sensorID = %s
7     , , ,

```

```

8
9 # load the data from the db to a data frame
10 df = pd.read_sql(sqlQuery, con=db_connection, params=(start_date,
    end_date, s))

```

Listing 4.35: querying measure_table

Now it is checked whether the number of measurements obtained equals the number of seconds in that time range to know if there is enough data to perform the average.

```

1 # check how many seconds I have in the interval so I know how many
    values I should have
2 diff = end_date - start_date + 1
3 # just a check to know if I have enough data to make the
    five_min_average, if not skip to the next one
4     if df.size < diff:
5         continue

```

Listing 4.36: checking data

After the check, the data is averaged and the maximum, minimum and standard deviation values are calculated.

```

1 # convert timestamp into datetime
2 df['timestamp'] = pd.to_datetime(df['timestamp'], unit='s')
3 # group the data by sensorID and every 5 minutes
4 grouped = df.groupby([pd.Grouper(key='sensorID'), pd.Grouper(key='
    geoHash'), pd.Grouper(key='altitude'),
5                     pd.Grouper(key='timestamp', freq='300s')])['
    data'].agg({'mean', 'max', 'min', 'std'})

```

Listing 4.37: execute the average value

At this point the data is inserted into the five_min_avg table.

```

1 # split the multi index into separate columns
2 grouped.reset_index(inplace=True)
3 # split the timestamp into separate columns
4 grouped['date'] = grouped['timestamp'].dt.strftime('%Y-%m-%d')
5 grouped['hour'] = grouped['timestamp'].dt.strftime('%H')
6 grouped['minute'] = grouped['timestamp'].dt.strftime('%M')
7 # delete the timestamp column
8 del grouped['timestamp']
9
10 # reorder the columns of the data frame like in the db
11 grouped = grouped[['date', 'hour', 'minute', 'sensorID', 'mean', 'max
    ', 'min', 'std', 'geoHash', 'altitude']]
12 # rename a column like in the db
13 grouped.rename(columns={'mean': 'value'}, inplace=True)
14

```



```
15 # round the result to two decimals
16 grouped['value'] = grouped['value'].round(decimals=2)
17 grouped['std'] = grouped['std'].round(decimals=2)
18
19 # add all the data from the data frame to a five_min_avg table in the
    database
20 grouped.to_sql(name='five_min_avg', con=db_connection, if_exists='
    append', index=False)
```

Listing 4.38: inserting data into five_min_avg table

4.8 ws_checking_system module

This module was inserted within the project to check the measurements received from the boards before inserting them in the DB. The implementation of this module was not developed as it was left to future extensions of the project.

4.9 Other project folders

Within the web service project, two folders with the name of "five_min_avg and hour_avg are included. Each of these folders contains all the material necessary for the creation of the two tables that keep the data concerning the averages within the DB.

The files with the ".sql" extension are used to create the tables inside the database, while the files with the ".py" extension are the scripts that perform the same work as the two asynchronous functions for calculating the average.

The use of these files is necessary for the first filling of the tables containing the averages as the asynchronous functions are not able to do their job when there is no data within the tables. This is because the calculation of the start date of the performing average range is based on the already existing data. For this reason the scripts allow the system administrator to enter the start and end date manually, thus initializing the two tables with the measurements.

Chapter 5

Mobile application implementation

The mobile application was developed within the Android Studio integrated development environment using the Java programming language. The structure of the application is divided into different source files depending on the function performed by the code. Among the various source files it can be recognized a file called “AndroidManifest.xml” whose task is organizational of the project. There is also a set of files whose name ends in "Activity" within the "it.polito.politopollutionsystem" package[fig. 5.1]. These files contain the Java code and perform the functionalities offered by the application. In the project contained in Android Studio it can be also noted a folder with the name “res”[fig. 5.2]. This folder represents all the static content of the application and is further divided into more subfolders:

- drawable: inside we find the images and icons present in the application
- font: represents the special fonts used within the project
- layout: here the source files containing the graphics of each activity in the project are specified
- menu: the folder contains the files that describe the menus in the application
- values: here the files are contained to describe eg. the colors or strings used for the presentation of the application views.

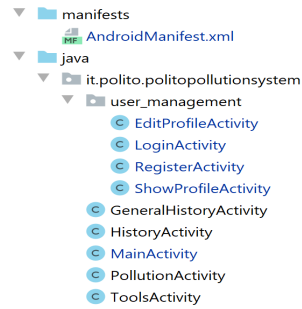


Figure 5.1: Android Manifest and Activities

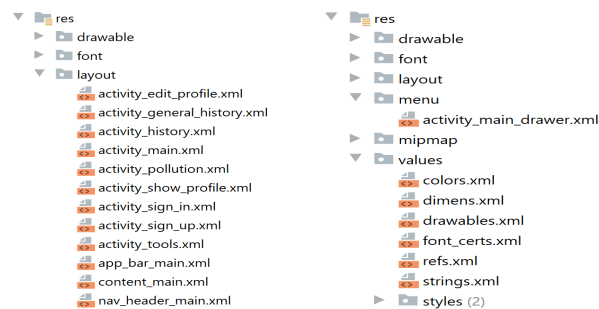


Figure 5.2: Res folder organization

5.1 Android manifest

Within this source file, with an xml extension, the organization of the entire application project is described. All its components and how they interact are listed and also the permissions that the application needs for proper operation.

The permissions are added through the use of the "user-permission" tag. The code below represents the permissions needed by this application along with how they are added to the manifest file.

```

1      <uses-permission android:name="android.permission.INTERNET" />
2      <uses-permission android:name="android.permission.
ACCESS_NETWORK_STATE" />
3      <uses-permission android:name="android.permission.
WRITE_EXTERNAL_STORAGE" />
4      <uses-permission android:name="android.permission.
ACCESS_FINE_LOCATION"/>

```

Listing 5.1: user permissions

The next piece of code represents another significant snippet of the Android-Manifest file.

```

1 <application
2     android:allowBackup="true "
3     android:icon="@drawable/earth200 "
4     android:label="@string/app_name"
5     android:roundIcon="@mipmap/ic_launcher_round "
6     android:supportsRtl="true "
7     android:usesCleartextTraffic="true "
8     android:theme="@style/AppTheme"
9     tools:targetApi="m">
10    <activity
11        android:name=".MainActivity"
12        android:label="@string/app_name"
13        android:screenOrientation="portrait "
14        android:theme="@style/AppTheme.NoActionBar "
15        tools:ignore="LockedOrientationActivity">
16        <intent-filter>
17            <action android:name="android.intent.action.MAIN" />
18
19            <category android:name="android.intent.category.
LAUNCHER" />
20        </intent-filter>
21    </activity>
22
23    <activity
24        android:name=".user_management.LoginActivity "
25        android:screenOrientation="portrait "
26        android:parentActivityName=".MainActivity "
27        tools:ignore="LockedOrientationActivity" />

```

Listing 5.2: application tag

As can be seen from the code, the file also contains an "application" tag which has the task of representing the organization of the entire application. Via the properties of this tag, e.g. the icon, the title or the main theme can be changed. Inside it, all the elements making up the application are inserted, as described in chapter 2. This project contains only elements of the Activity type inserted through the tag with the same name. It can be seen that for each "activity" tag the name and its parent are specified but also the orientation on the screen of that view.

5.2 Activities

5.2.1 MainActivity

This activity is the first visible immediately after starting the application. Its general organization is visible in the code below.

```
1 public class MainActivity extends AppCompatActivity implements
   NavigationView.OnNavigationItemSelectedListener {
2     private RequestQueue queue;
3     private String url;
4     private SharedPreferences sharedPref;
5     private TextView mailHeader;
6     private NavigationView navigationView;
7     private View header;
8
9     @Override
10    protected void onCreate(Bundle savedInstanceState) {...}
11
12    @Override
13    protected void onResume() {...}
14
15    public void logIn(View view) {...}
16
17    public void logout(View view) throws JSONException {...}
18
19    public void generalHistory(View view) {...}
20
21    public void pollution(View view) {...}
22
23    @Override
24    public boolean onNavigationItemSelectedListener(@NonNull MenuItem
   menuItem) {...}
25 }
```

Listing 5.3: MainActivity organization

The MainActivity class to become an actual Activity needs to extend the AppCompatActivity class. Thanks to this procedure it is possible to overwrite e.g. the onCreate() method that allows the application to execute the code while creating the activity itself. This class also implements a specific interface for managing clicks on the side menu items that can be used in this view. This class also defines some private properties that will be explained during their use.

onCreate()

This method has the task of creating the activity. First of all it inserts the layout inside it using the "setContentView()" method.

```
1 setContentView(R.layout.activity_main);
```

Listing 5.4: layout inflation

At this point the method switches to the initialization of the toolbar and the side menu, whose code is omitted as it is not interesting enough for this description.

The initialization of the shared preferences takes place at this instant of the code. This object allows the program to access a memory area reserved for the application in which properties can be saved. This properties contains in fact, the token received during the login phase. In this way it remains easily accessible and is not lost when the application is closed.

```
1      sharedPref = getApplicationContext().getSharedPreferences("
      sharedPrefs", Context.MODE_PRIVATE);
```

Listing 5.5: shared preferences initialization

The last task of this method is to check if the user has already logged in, verifying the presence of the token. If the token is present, the logout button is displayed and the current user is displayed in the side menu. Otherwise the login button remains visible.

```
1      if (sharedPref.getString("token", "").equals("")) {
2          findViewById(R.id.btnLogout).setVisibility(View.GONE);
3          findViewById(R.id.btnLogin).setVisibility(View.VISIBLE);
4          mailHeader=findViewById(R.id.nav_email);
5          mailHeader.setText("");
6      } else {
7          findViewById(R.id.btnLogin).setVisibility(View.GONE);
8          findViewById(R.id.btnLogout).setVisibility(View.VISIBLE);
9          mailHeader=findViewById(R.id.nav_email);
10         mailHeader.setText(sharedPref.getString("email", ""));
11     }
```

Listing 5.6: checking token

onResume()

This method is called each time when the current activity returns to the top of the activity stack. This can happen when for eg. by opening the login activity we decide to close it.

The only task of this method is to check in a manner similar to the onCreate() method if there is a token representing a logged in user.

login()

The method is executed after pressing the Login button. Its only purpose is to open the activity that allows the user to perform the application access procedure.

```
1      startActivity(new Intent(this, LoginActivity.class));
```

Listing 5.7: login method

logout()

This method has the task of sending the request to logout from the web service, i.e. invalidating the user's token.

First of all, since this operation can take a few seconds, a loading indicator is setted as visible.

```
1 findViewById(R.id.loadData).setVisibility(View.VISIBLE);
```

Listing 5.8: show loading status

A JSON object is prepared containing the data to be inserted in the body of the request

```
1 JSONObject jsonBody = new JSONObject();
2 jsonBody.put("email", sharedPref.getString("email", ""));
3 final String requestBody = jsonBody.toString();
```

Listing 5.9: JSON preparing

Two objects are also initialized, one representing the queue of requests to be sent outside the application and the url to which these requests are directed.

```
1 queue = Volley.newRequestQueue(this);
2 url =sharedPref.getString("host", "http://localhost:5000/ws") +"/
logout";
```

Listing 5.10: objects initialization

At this point an object is created that represents the request sent to the web service. The internal method onResponse() specifies the actions to be performed in case the request receives a successful response. In this case, the token and email values within the shared preferences are simply deleted. In the code it can also be seen that there is a similar method in case of a failed request. In addition, three other methods are overridden that allow the application to specify the header, content-type and body of the request.

```
1 StringRequest stringRequest = new StringRequest(Request.Method.
  POST, url,
2         new Response.Listener<String>() {
3             @Override
4             public void onResponse(String response) {
5                 //reset the email ant token in the shared
6                 SharedPreferences.Editor editor = sharedPref.
7                 edit();
8                 editor.putString("token", "");
9                 editor.putString("email", "");
10                editor.apply();
```

```

10         findViewById(R.id.loadData).setVisibility(
View.GONE);
11         Toast.makeText(getApplicationContext(), "
Success", Toast.LENGTH_LONG).show();
12         finish();
13         startActivity(getIntent());
14     }
15     }, new Response.ErrorListener() {
16     @Override
17     public void onErrorResponse(VolleyError error) {
18         findViewById(R.id.loadData).setVisibility(View.GONE);
19         Toast.makeText(getApplicationContext(), "Problems",
Toast.LENGTH_LONG).show();
20     }
21     }) {
22     @Override
23     public Map<String, String> getHeaders() throws
AuthFailureError {
24         //add header params to the request
25         Map<String, String> params = new HashMap<String,
String>();
26         params.put("x-access-token", sharedPreferences.getString("
token", ""));
27         return params;
28     }
29
30     @Override
31     public String getBodyContentType() {
32         return "application/json; charset=utf-8";
33     }
34
35     @Override
36     public byte[] getBody() throws AuthFailureError {
37         //add body to the request
38         try {
39             return requestBody == null ? null : requestBody.
getBytes("utf-8");
40         } catch (UnsupportedEncodingException uee) {
41             VolleyLog.wtf("Unsupported Encoding while trying
to get the bytes of %s using %s", requestBody, "utf-8");
42             return null;
43         }
44     }
45 };

```

Listing 5.11: resquest code

Finally, the code specifies that the request must be sent only once to the server after which the request is added to the queue for sending.


```

1      stringRequest.setRetryPolicy(new DefaultRetryPolicy(
2          0,
3          DefaultRetryPolicy.DEFAULT_MAX_RETRIES,
4          DefaultRetryPolicy.DEFAULT_BACKOFF_MULT));
5      queue.add(stringRequest);

```

Listing 5.12: sending request

pollution()

This method is used to start the activity that presents real-time data on pollution. During this procedure the “pm” parameter is passed to the child view, which indicates the type of pollution particles that must be displayed. Before launching the new activity, it is checked whether the user has already logged in.

```

1      if (sharedPref.getString("token", "").equals("")) { //not logged
2          Toast.makeText(this, "Login first", Toast.LENGTH_LONG).
3          show();
4          } else {
5              Intent intent = new Intent(this, PollutionActivity.class)
6              ;
7              SharedPreferences.Editor editor = sharedPref.edit();
8              editor.putString("pm", "pm25");
9              editor.apply();
10             startActivity(intent);
11         }

```

Listing 5.13: pollution method

generalHistory()

Its task is to launch the activity for viewing the general history of the measurements. The procedure is similar to the previous point, without adding the "pm" parameter.

onNavigationItemSelected()

This method allows the application to manage the click on the options in the side menu. The code inside it, has the only task of starting the activities corresponding to the relative items in the menu.

```

1      public boolean onNavigationItemSelected(@NonNull MenuItem
2      menuItem) {
3          int id = menuItem.getItemId();
4          if (id == R.id.profile){
5              if (sharedPref.getString("token", "").equals("")){

```

```

6         Toast.makeText(this, "Login first", Toast.LENGTH_LONG
7     ).show();
8     } else {
9         startActivity(new Intent(this, ShowProfileActivity.
10    class));
11    }
12    } else if (id == R.id.tools) {
13        startActivity(new Intent(this, ToolsActivity.class));
14    }
15
16    DrawerLayout drawer = findViewById(R.id.drawer_layout);
17    drawer.closeDrawer(GravityCompat.START);
18    return true;
19 }

```

Listing 5.14: menu managing code

5.2.2 LoginActivity

This activity has the task of allowing the user to log in. Its main methods are visible below.

```

1    public class LoginActivity extends AppCompatActivity{
2        private SharedPreferences sharedPref;
3        private RequestQueue queue;
4        private String url;
5        private String email;
6        private String password;
7
8        private Pattern pattern;
9        private Matcher m;
10
11        private EditText edtEmail;
12        private EditText edtPass;
13
14        @Override
15        protected void onCreate(Bundle savedInstanceState) {...}
16
17        public void register(View view) {...}
18
19        public void login(View view) throws JSONException {...}
20
21        public void recover(View view) throws JSONException {...}
22    }

```

Listing 5.15: Login activity organization

onCreate()

The method only performs the function of adding the layout to the activity and initializes the fields for entering data by the user.

```
1      @Override
2      protected void onCreate(Bundle savedInstanceState) {
3          super.onCreate(savedInstanceState);
4          setContentView(R.layout.activity_sign_in);
5
6          sharedPreferences = getApplicationContext().getSharedPreferences("
sharedPrefs", Context.MODE_PRIVATE);
7          edtEmail = (EditText) findViewById(R.id.signInEmail);
8          edtPass = (EditText) findViewById(R.id.signInPwd);
9      }
```

Listing 5.16: onCreate method

register()

Its task is to initiate the user registration activity. The procedure is similar to the start-ups seen previously. It is triggered by clicking on the string "Not an User?".

login()

The method is triggered after clicking on the login button. Before executing the code that sends the request to the web service, the correctness of the data entered by the user is analyzed. In case of problems, the procedure ends immediately.

```
1      Boolean error=Boolean.FALSE;
2
3      email = edtEmail.getText().toString();
4      password = edtPass.getText().toString();
5
6      //checks is some fields are not empty
7      if(email.isEmpty()){
8          edtEmail.setError("Missing value");
9          error = Boolean.TRUE;
10     }
11
12     if(password.isEmpty()){
13         edtPass.setError("Missing value");
14         error = Boolean.TRUE;
15     }
16
17     //check if there is a valid email
18     pattern = Pattern.compile("^[a-zA-Z0-9_\\-\\.]+@([a-zA-Z0-9_\\-\\.]+)\\.([a-zA-Z]{2,5})$");
```

```

19         m = pattern.matcher(edtEmail.getText().toString());
20         if (!m.matches()) {
21             edtEmail.setError("Insert a valid email");
22             error = Boolean.TRUE;
23         }
24
25         if (error) {
26             return;
27         }

```

Listing 5.17: inserted value cheking code

At this point, the procedure for preparing and sending the request to the server is similar to that seen for the logout procedure in the MainActivity. The difference lies in the code within the onResponse() method which, if successful, has the task of saving the token received together with the user's email address in the shared preferences object.

```

1     JSONObject json = new JSONObject(response);
2     SharedPreferences sharedPref = getApplicationContext().
    getSharedPreferences("sharedPrefs", Context.MODE_PRIVATE);
3     SharedPreferences.Editor editor = sharedPref.edit();
4     editor.putString("token", json.getString("token"));
5     editor.putString("email", email);
6     editor.apply();
7
8     findViewById(R.id.loadData).setVisibility(View.GONE);
9
10    Toast.makeText(getApplicationContext(), "Success", Toast.
    LENGTH_LONG).show();
11    finish();

```

Listing 5.18: login response code

recover()

This last method is used to send the password recovery request to the web service. The procedure is similar to the requests seen above. The function takes the value of the email from the edit text present in this Activity, also checking the correctness of the data entered.

5.2.3 RegisterActivity

This activity allows the registration of new users in the system. Inside, only two methods are implemented: onCreate() and register().

onCreate()

In this method, fields are mainly initialized for the user to enter data together with the addition of the layout to the current activity. The code is identical to the code seen in the LoginActivity's onCreate() method.

register()

The code of this function is executed after clicking on the “Register” button. The method initially checks the correctness of the data entered within the various edit text present in the Activity. The procedure for sending the request to the web service is similar to those seen in the previous sections and does not introduce anything new. For this reason, no bits of source code are displayed in its description.

5.2.4 PollutionActivity

This activity has the task of displaying the most updated data relating to environmental pollution to the user. Later you can see its structure.

```
1 public class PollutionActivity extends AppCompatActivity {
2     private final int REQUEST_PERMISSIONS_REQUEST_CODE = 1;
3     private MapView map = null;
4     private FusedLocationProviderClient fusedLocationClient;
5     private OverlayItem i;
6     private Button btnpm25;
7     private Button btnpm10;
8
9     private RequestQueue queue;
10    private String url;
11    private SharedPreferences sharedPref;
12    private String pm;
13    private Boolean myloc;
14
15    @Override
16    protected void onCreate(Bundle savedInstanceState) {...}
17
18    @Override
19    public void onResume() {...}
20
21    @Override
22    public void onPause() {...}
23
24    private void requestPermissionsIfNecessary(String[] permissions)
25    {...}
26
27    private int getAQIFillColor(Double v) {...}
```

```

28     private int getAQIStrokeColor(Double v) {...}
29
30     private String getAQIQuality(Double v) {...}
31
32     public void pm10(View view) {...}
33
34     public void pm25(View view) {...}
35 }

```

Listing 5.19: PollutionActivity organization**onCreate()**

This method does most of the work of this Activity.

First of all, the values necessary for the creation of this activity are read from the shared preferences and saved in variables. The "pm" is used to know what data to request from the server while "myloc" indicates how the initial zoom on the map must be performed.

```

1     sharedPref = getApplicationContext().getSharedPreferences("
    sharedPrefs", Context.MODE_PRIVATE);
2     pm = sharedPref.getString("pm", "");
3     myloc = sharedPref.getBoolean("myloc", false);

```

Listing 5.20: reading configuration

At this point, after entering the layout in the activity, the buttons visible at the top of the Activity are initialized. Based on the value of "pm" they are colored and made clickable or not.

```

1     btnpm25 = findViewById(R.id.btnpm25);
2     btnpm10 = findViewById(R.id.btnpm10);
3     if (pm.equals("pm25")){
4         btnpm25.setBackgroundColor(Color.parseColor("#FF048C32"));
5         btnpm10.setBackgroundColor(ContextCompat.getColor(
    getApplicationContext(),
6             R.color.colorAccent));
7         btnpm25.setClickable(false);
8         btnpm10.setClickable(true);
9     } else{
10        btnpm10.setBackgroundColor(Color.parseColor("#FF048C32"));
11        btnpm25.setBackgroundColor(ContextCompat.getColor(
    getApplicationContext(),
12            R.color.colorAccent));
13        btnpm10.setClickable(false);
14        btnpm25.setClickable(true);
15    }

```

Listing 5.21: buttons initialization

Now is the time to initialize the object that represents the map and ask the user for the necessary permissions if this procedure has not yet been performed.

```
1 //initialize the map
2 map = (MapView) findViewById(R.id.map);
3 map.setTileSource(TileSourceFactory.MAPNIK);
4
5 //permission request
6 requestPermissionsIfNecessary(new String[] {
7     Manifest.permission.ACCESS_FINE_LOCATION,
8     Manifest.permission.WRITE_EXTERNAL_STORAGE
9 });
```

Listing 5.22: map and permissions

At this point it is possible to zoom on the map in the user's current position or in the center of Turin, based on the value in the "myloc" variable.

```
1 final IMapController mapController = map.getController();
2 mapController.setZoom(20);
3
4 if(myloc == true){
5     //use if current loc as starting point
6     fusedLocationClient = LocationServices.
7     getFusedLocationProviderClient(this);
8     fusedLocationClient.getLastLocation()
9     .addOnSuccessListener(this, new OnSuccessListener<
10     Location>() {
11         @Override
12         public void onSuccess(Location location) {
13             // Got last known location. In some rare
14             situations this can be null.
15             if (location != null) {
16                 GeoPoint startPoint = new GeoPoint(
17                 location.getLatitude(), location.getLongitude());
18                 mapController.setCenter(startPoint);
19             }
20         }
21     });
22 }else{
23     GeoPoint startPoint = new GeoPoint(45.0419, 7.6259);
24     mapController.setCenter(startPoint);
25 }
```

Listing 5.23: zoom on map

Now the user's current position on the map is added and the possibilities of zooming and interacting with it is enabled. The location will only be displayed if the GPS on the device is active.

```

1      MyLocationNewOverlay mLocationOverlay = new MyLocationNewOverlay(
      new GpsMyLocationProvider(getApplicationContext()),map);
2          mLocationOverlay.enableMyLocation();
3          map.getOverlays().add(mLocationOverlay);
4
5          map.setBuiltInZoomControls(true);
6          map.setMultiTouchControls(true);

```

Listing 5.24: current user position

In this part of the method, the data request is made to the web service at the endpoint which returns the most updated data. The request is structured and in the same way seen in the previous cases.

In the `onResponse()` method the data regarding the latest most recent measurements are first read from the JSON of the response and are divided according to the geoHash, this is because multiple boards can be placed in one place and during the positioning of the markers on the map the data displayed in its description must represent the average of the values received for all measurements relating to the same location. The following piece of code represents the drawing process of the values received.

```

1      //arrays to track add the data for every board with different
      location (lat_long)
2      ArrayList<Double> ss = new ArrayList<>(); //sensors
3      ArrayList<Double> tt = new ArrayList<>(); //temperature
4      ArrayList<Double> rr = new ArrayList<>(); //humidity
5      ArrayList<String> pos = new ArrayList<>(); //position -> lat long
6      ArrayList<Integer> n = new ArrayList<>(); //number of boards
7      ArrayList<String> bb = new ArrayList<>(); //board ids
8      ArrayList<String> dd = new ArrayList<>(); //date
9      int z = 0; //index for the previous declared arrays
10
11      for(int k = 0; k < json.length(); k++){ //for on every board in
      the response
12          boolean stop=false; //boolean to stop if the date of the new
      sensors is before the current one
13          boolean replace=false; //boolean to replace the data if the
      current date is after the incoming one so the previous data is
      invalid
14          JSONObject board = ((JSONObject)json.get(k)); //read the
      board object from the json
15          JSONArray sensors = board.getJSONArray("sensors"); //read the
      sensors of the current board
16          if(sensors.length() == 0){ //if the board is empty
17              continue;
18          }
19

```



```

20 // check if the board with this location is already in the
arrays so add new one or sum to the previous one
21 if (pos.contains(board.getString("lat")+ ","+board.getString("
long"))){
22     int index = pos.indexOf(board.getString("lat")+"," +board.
getString("long"));
23     //check if the date is after or before or the same of the
previous one
24     for (int u=0; u<4; u++) {
25         if (dd.get(index).equals(((JSONObject)sensors.get(u)).
getString("date")+ " "+
26             ((JSONObject)sensors.get(u)).
getString("hour")+ ":"+
27             ((JSONObject)sensors.get(u)).
getString("minute")+ ":" + "00")){
28             ss.add(index, ss.get(index) + Double.parseDouble
(((JSONObject)sensors.get(u)).getString("value")));
29             }else if (new SimpleDateFormat("yyyy-MM-dd HH:mm:ss").
parse(dd.get(index)).before(new SimpleDateFormat("yyyy-MM-dd
HH:mm:ss").parse(((JSONObject)sensors.get(u)).
getString("date")+ " "+
30             ((JSONObject)sensors.get(u)).
getString("hour")+ ":"+
31             ((JSONObject)sensors.get(u)).
getString("minute")+ ":" + "00"))){
32                 stop = true;
33             }else{ //current is after the new one
34                 ss.add(z, ss.get(index) + Double.parseDouble(((
JSONObject)sensors.get(u)).getString("value")));
35                 replace = true;
36             }
37         }
38         if (stop){
39             continue;
40         }
41         if (replace){
42             n.add(index, 1);
43             bb.add(index, board.getString("id"));
44             tt.add(index, Double.parseDouble(board.getString("
temp")));
45             rr.add(index, Double.parseDouble(board.getString("rh
")));
46         }else{
47             n.add(index, n.get(index)+ 1);
48             bb.add(index, bb.get(index).concat(", " + board.
getString("id")));
49             tt.add(index, tt.get(index) + Double.parseDouble(
board.getString("temp")));

```

```

50         rr.add(index, rr.get(index) + Double.parseDouble(
board.getString("rh")));
51     }
52
53     } else {
54         for (int u=0; u<4; u++) {
55             dd.add(z, ((JSONObject)sensors.get(u)).getString("
date")+ " "+
56                 ((JSONObject)sensors.get(u)).getString("hour
")+ ":"+
57                 ((JSONObject)sensors.get(u)).getString("
minute")+ ":" + "00");
58             ss.add(z, Double.parseDouble(((JSONObject)sensors.get
(u)).getString("value")));
59         }
60         pos.add(z, board.getString("lat")+","+board.getString("
long"));
61         bb.add(board.getString("id"));
62         n.add(z, 1);
63         tt.add(z, Double.parseDouble(board.getString("temp")));
64         rr.add(z, Double.parseDouble(board.getString("rh")));
65
66         z++;
67     }
68
69 }

```

Listing 5.25: sorting data

Each received measurement is analyzed from the point of view of its geoHash, after which the dates in which the data sample was collected are checked, thus keeping only the most up-to-date data and discarding the values with non-uniform dates from the average calculation. This procedure is necessary as the web service returns the most updated value contained within the DB but, with many boards positioned in the same location, the problem could arise that not all data have the same date and time as a board for example could be broken. All the data is sorted into different arrays which contain for each different geoHash found, the sum of the values read up to that moment. Each array contains different data such as eg. the sum of the measurements or the sum of the temperatures.

At this point it is possible to move on to the calculation of the average, iterating the vectors and dividing each value by the number of values read.

```

1 //iterate over the arrays to make the mean of the values summed
in the for before
2 for (int f=0; f<pos.size(); f++){
3     Double s_mean = Math.round((ss.get(f) / (n.get(f) * 4)) *
100.0) / 100.0;

```

```

4      Double t_mean = Math.round((tt.get(f) / n.get(f)) * 100.0) /
100.0;
5      Double r_mean = Math.round((rr.get(f) / n.get(f)) * 100.0) /
100.0;
6      String quality = getAQIQuality(s_mean);
7      if(quality == null){
8          findViewById(R.id.loadData).setVisibility(View.GONE);
9          Toast.makeText(getApplicationContext(), "Quality
Problems", Toast.LENGTH_LONG).show();
10         return;
11     }
12     // add the item to the array
13     i = new OverlayItem("Board ids: " + bb.get(f),
14                         "Date: " + dd.get(f) +
15                         "\nValue: " + s_mean +
16                         "\nTemperature: " + t_mean +
17                         "\nHumidity: " + r_mean +
18                         "\nQuality: " + quality,
19                         new GeoPoint(Double.parseDouble(pos.get(f).split(",")
[0]),
20                                     Double.parseDouble(pos.get(f).split(",") [1])));
21     items.add(i);
22     //adding circles around the points
23     List<GeoPoint> circle = Polygon.pointsAsCircle(new GeoPoint(
Double.parseDouble(pos.get(f).split(",") [0]), Double.parseDouble(
pos.get(f).split(",") [1])), 20);
24     Polygon p = new Polygon(map);
25     p.setPoints(circle);
26     p.setTitle("Range");
27     int strokeColor = getAQIStrokeColor(s_mean);
28     if(strokeColor == -1 || strokeColor == -2){
29         findViewById(R.id.loadData).setVisibility(View.GONE);
30         Toast.makeText(getApplicationContext(), "Strokr Color
Problems", Toast.LENGTH_LONG).show();
31         return;
32     }
33     p.setStrokeColor(strokeColor);
34     int fillColor = getAQIFillColor(s_mean);
35     if(fillColor == -1 || fillColor == -2){
36         findViewById(R.id.loadData).setVisibility(View.GONE);
37         Toast.makeText(getApplicationContext(), "Fill Color
Problems", Toast.LENGTH_LONG).show();
38         return;
39     }
40     p.setFillColor(fillColor); // 50FF0000
41     map.getOverlayManager().add(p);
42 }

```

Listing 5.26: calculating the mean value

During this procedure the “OverlayItems” are created, which will be the points actually visible on the map and they are added to a common array to add them all together at the end of the cycle. The cycle also includes the addition of circles, colored according to the value of environmental pollution, around each marker on the map. The color to be displayed as well as the descriptive text are requested from the functions declared at the bottom of the file.

At this point all the "OverlayItems" are added to the map by also setting the possibility of a long press on the markers which gives the possibility to open the activity capable of presenting the most recent history of the measurements at that point.

```

1      ItemizedOverlayWithFocus<OverlayItem> mOverlay = new
      ItemizedOverlayWithFocus<OverlayItem>(items ,
2          new ItemizedIconOverlay.OnItemGestureListener<OverlayItem
      >() {
3          @Override
4          public boolean onItemSingleTapUp(final int index, final
      OverlayItem item) {
5              //do something
6              return true;
7          }
8          @Override
9          public boolean onItemLongPress(final int index, final
      OverlayItem item) {
10             //on long press opening of the history activity , passing
      the list of boards as extra
11             Intent intent = new Intent(getApplicationContext() ,
      HistoryActivity.class);
12             intent.putExtra("boards", item.getTitle());
13             intent.putExtra("snippet", item.getSnippet());
14             startActivity(intent);
15             return false;
16         }
17     }, getApplicationContext());
18     mOverlay.setFocusItemsOnTap(true);
19     map.getOverlays().add(mOverlay);

```

Listing 5.27: adding markers

onResume()

This method has the only task of restarting the map when the Activity restarts.

```

1      @Override
2      public void onResume() {
3          super.onResume();
4          map.onResume(); //needed for compass, my location overlays ,
      v6.0.0 and up

```

```
5 | }
```

Listing 5.28: onResume()

onPause()

Here the map is paused when the activity pauses.

```
1 | @Override
2 | public void onPause() {
3 |     super.onPause();
4 |     map.onPause(); //needed for compass, my location overlays,
5 |     v6.0.0 and up
6 | }
```

Listing 5.29: onPause()

requestPermissionsIfNecessary()

This method has the task of requesting the necessary permissions from the user to ensure the proper functioning of the activity. It is called during the onCreate() method.

```
1 | private void requestPermissionsIfNecessary (String [] permissions)
2 | {
3 |     ArrayList<String> permissionsToRequest = new ArrayList<>();
4 |     for (String permission : permissions) {
5 |         if (ContextCompat.checkSelfPermission(this, permission)
6 |             != PackageManager.PERMISSION_GRANTED) {
7 |             // Permission is not granted
8 |             permissionsToRequest.add(permission);
9 |         }
10 |     }
11 |     if (permissionsToRequest.size() > 0) {
12 |         ActivityCompat.requestPermissions(
13 |             this,
14 |             permissionsToRequest.toArray(new String [0]),
15 |             REQUEST_PERMISSIONS_REQUEST_CODE);
16 |     }
17 | }
```

Listing 5.30: requestPermissionsIfNecessary()

getAQIFillColor()

The method is invoked during the addition of the circumferences around the marker and returns the color that must be applied to the background of the circle based on the value of environmental pollution.

```

1  private int getAQIFillColor(Double v){
2      if(pm.equals("pm10")){
3          if(v >= 0 && v <= 50){
4              return Color.parseColor("#50048C32");
5          }else if(v >= 51 && v <=100){
6              return Color.parseColor("#5000FF00");
7          }else if(v >= 101 && v <=250){
8              return Color.parseColor("#50FFFF00");
9          }else if(v >= 251 && v <=350){
10             return Color.parseColor("#50FFA500");
11          }else if(v >= 351 && v <=430){
12             return Color.parseColor("#50FF0000");
13          }else if(v >= 431){
14             return Color.parseColor("#50800000");
15          }else{
16             return -2; // it means a value not in the range
17          }
18      }else if(pm.equals("pm25")){
19          if(v >= 0 && v <= 30){
20              return Color.parseColor("#50048C32");
21          }else if(v >= 31 && v <=60){
22              return Color.parseColor("#5000FF00");
23          }else if(v >= 61 && v <=90){
24              return Color.parseColor("#50FFFF00");
25          }else if(v >= 91 && v <=120){
26              return Color.parseColor("#50FFA500");
27          }else if(v >= 121 && v <=250){
28              return Color.parseColor("#50FF0000");
29          }else if(v >= 251){
30              return Color.parseColor("#50800000");
31          }else{
32             return -2; // it means a value not in the range
33          }
34      }else{
35          return -1; //it means a value of pmKind not in pm25 or
36          pm10
37      }
38  }

```

Listing 5.31: getAQIFillColor()

In this activity there are two other similar methods that respectively return the color of the edge of the circle and the descriptive string of the state of the air.

pm10()

This function is called once the button with the same name visible in the high part of the map is clicked. Its task is to recreate the current activity but changing the

"pm" parameter, that is the data that must be shown on the map.

```

1      public void pm10(View view){
2          SharedPreferences.Editor editor = sharedPref.edit();
3          editor.putString("pm", "pm10");
4          editor.apply();
5          this.recreate();
6      }

```

Listing 5.32: pm10()

pm25()

Method with operation similar to that previously described.

5.2.5 HistoryActivity

This activity has the task of displaying the history of the last two hours available of the measurements regarding the boards present under the pin clicked to access it. The history is displayed in the form of three graphs made using the external library "MPAndroidChart". The structure of the source file is visible below.

```

1 public class HistoryActivity extends AppCompatActivity {
2     private RequestQueue queue;
3     private String url;
4     private SharedPreferences sharedPref;
5     private String pm;
6     private Spinner sItems;
7
8     private LineChart chart;
9     private LineChart chartT;
10    private LineChart chartH;
11
12    private String date;
13    private String hour;
14
15    private JSONArray json;
16
17    @Override
18    protected void onCreate(Bundle savedInstanceState) {...}
19
20    public void saveSensImage(View view) {...}
21
22    public void saveSensData(View view) {...}
23
24    public void saveTempImage(View view) {...}
25
26    public void saveTempData(View view) {...}

```

```

27
28     public void saveHumidityImage(View view) {...}
29
30     public void saveHumidityData(View view) {...}
31 }

```

Listing 5.33: HistoryActivity organization**onCreate()**

During the creation of this activity, one of the most important points is the initialization of the graphs for visualization of the past history. Below is the initialization code of one of the three graphics in the activity.

```

1 chart = findViewById(R.id.chart);
2 chart.setBackgroundColor(Color.WHITE);
3
4 XAxis xAxis = chart.getXAxis();
5 xAxis.setValueFormatter(new ValueFormatter() {
6
7     private final SimpleDateFormat mFormat = new SimpleDateFormat("HH:mm
8         ", Locale.ENGLISH);
9     @Override
10    public String getFormattedValue(float value) {
11        long millis = (long) value;
12        Calendar calendar = Calendar.getInstance();
13        calendar.setTimeInMillis(millis);
14        return mFormat.format(calendar.getTime());
15    }
16 });

```

Listing 5.34: chart initialization example

The value on the x axis is transformed into a formatted string to show the hour and minutes related to the measures shown.

At this point, ids are initialized within the spinner for choosing the board whose values should be displayed on the charts.

The spinner also adds the ability to listen when the value inside it is changed. Each time the value changes, a new data request is sent to the web service. The procedure remains the same as those described above.

```

1 ((Spinner) findViewById(R.id.spnBoards)).setOnItemSelectedListener(
2     new AdapterView.OnItemSelectedListener() {
3         @Override
4         public void onItemSelected(AdapterView<?> parentView,
5             View selectedItemView,
6             int position,
7             long id) { //request sending code}

```



```
8 };
```

Listing 5.35: adding listener to the spinner

After receiving the data, these are sorted into different arrays depending on the sensor they come from and added to the “LineDataSets”. Each such object represents a separate data line on the chart. Once this procedure has been carried out, the data can be set to the corresponding graph.

```
1 for(int k = 1; k < json.length(); k++) { //for on line data in the
  response
2   date = format.parse(((JSONObject)json.get(k)).getString("date") +
    " " +
3     ((JSONObject)json.get(k)).getString("hour") + ":"
    +
4     ((JSONObject)json.get(k)).getString("minute")
    + ":00");
5   yValues1.add(new Entry(date.getTime(), Float.parseFloat(((
    JSONObject)json.get(k)).getString(keys.get(2)))));
6   yValues2.add(new Entry(date.getTime(), Float.parseFloat(((
    JSONObject)json.get(k)).getString(keys.get(3)))));
7   yValues3.add(new Entry(date.getTime(), Float.parseFloat(((
    JSONObject)json.get(k)).getString(keys.get(4)))));
8   yValues4.add(new Entry(date.getTime(), Float.parseFloat(((
    JSONObject)json.get(k)).getString(keys.get(5)))));
9   yValuesT.add(new Entry(date.getTime(), Float.parseFloat(((
    JSONObject)json.get(k)).getString(keys.get(0)))));
10  yValuesH.add(new Entry(date.getTime(), Float.parseFloat(((
    JSONObject)json.get(k)).getString(keys.get(1)))));
11 }
12 //add data to sensors chart
13 LineDataSet set1 = new LineDataSet(yValues1, keys.get(2).split("_")
   [0]);
14 set1.setFillAlpha(110);
15 LineDataSet set2 = new LineDataSet(yValues2, keys.get(3).split("_")
   [0]);
16 set2.setFillAlpha(110);
17 LineDataSet set3 = new LineDataSet(yValues3, keys.get(4).split("_")
   [0]);
18 set3.setFillAlpha(110);
19 LineDataSet set4 = new LineDataSet(yValues4, keys.get(5).split("_")
   [0]);
20 set4.setFillAlpha(110);
21 ArrayList<ILineDataSet> dataSets = new ArrayList<>();
22 dataSets.add(set1);
23 dataSets.add(set2);
24 set2.setColor(Color.RED);
25 set2.setCircleColor(Color.RED);
26 dataSets.add(set3);
27 set3.setColor(Color.GREEN);
```

```

28 set3.setCircleColor(Color.GREEN);
29 dataSets.add(set4);
30 set4.setColor(Color.BLUE);
31 set4.setCircleColor(Color.BLUE);
32 LineData data = new LineData(dataSets);
33
34 chart.setData(data);
35 chart.invalidate();

```

Listing 5.36: adding data to the charts

saveSensImage()

This method allows the user to download the image of the chart representing the values read by the various sensors. The code sets the filename as: `date_hour_board_last_2_hour_sens_chart`. In case of a multiple download, the name is followed by a consecutive number starting from 1 closed inside round brackets.

The `saveToGallery` method is used to save the file.

```

1 public void saveSensImage(View view){
2     int counter = 1;
3     String fn;
4     String [] s_date;
5
6     File folder = new File(Environment.
7         getExternalStorageDirectory()
8         + "/DCIM");
9
10    if (!folder.exists())
11        folder.mkdir();
12
13    s_date = date.split("-");
14
15    fn = s_date[0] + "_" + s_date[1] + "_" + s_date[2] + "_" +
16        sItems.getSelectedItem().toString() + "_last_2_hour_sens_chart";
17
18    //check if the file already exists
19    File out = new File(Environment.getExternalStorageDirectory()
20        + "/DCIM/" + s_date[0] + "_" + s_date[1] + "_" +
21        s_date[2] + "_" + sItems.getSelectedItem().toString() + "
22        _last_2_hour_sens_chart.png");
23    while(out.exists()){
24        out = new File(Environment.getExternalStorageDirectory()
25            + "/DCIM/" + s_date[0] + "_" + s_date[1] + "_" +
26            s_date[2] + "_" + sItems.getSelectedItem().toString() + "
27            _last_2_hour_sens_chart(" + counter + ").png");
28    }
29 }

```

```

22         fn = s_date[0] + "_" + s_date[1] + "_" + s_date[2] + "_"
+ sItems.getSelectedItemAt().toString() + "_last_2_hour_sens_chart
+ counter + ")";
23         counter++;
24     }
25
26     chart.saveToGallery(fn);
27     Toast.makeText(getApplicationContext(), "Image correctly
saved", Toast.LENGTH_LONG).show();
28 }

```

Listing 5.37: saving chart image

saveSensData()

This method allows the user to download the data relating to the chart in the form of a file with the extension “.csv”. After specifying the file name in a similar way to the method described above, the data is saved through the use of a separate thread in a file saved in memory. The files thus created are saved in the “Polution_exports” folder, created by this method, in the phone memory.

```

1 new Thread() {
2     public void run() {
3         try {
4             FileWriter fw = new FileWriter(filename);
5             ArrayList<String> keys= new ArrayList<>();
6             Pattern p = Pattern.compile(".*_cal");
7             Matcher m;
8
9             //get keys
10            Iterator<String> iterator = ((JSONObject)json.get
(0)).keys();
11
12            while(iterator.hasNext()) {
13                String currentKey = iterator.next();
14
15                m = p.matcher(currentKey);
16                if (m.matches()) {
17                    keys.add(currentKey);
18                }
19            }
20            //add keys to csv
21            fw.append(" date ");
22            fw.append(' ', ' ');
23
24            fw.append(" hour ");
25            fw.append(' ', ' ');
26
27            fw.append(" minute ");

```

```

27         fw.append(' ');
28
29         fw.append(keys.get(0).split("_")[0]);
30         fw.append(' ');
31
32         fw.append(keys.get(1).split("_")[0]);
33         fw.append(' ');
34
35         fw.append(keys.get(2).split("_")[0]);
36         fw.append(' ');
37
38         fw.append(keys.get(3).split("_")[0]);
39         fw.append(' ');
40         fw.append('\n');
41
42         //add data to csv
43         for(int k = 1; k < json.length(); k++) { //for on
line data in the json that contains the last response
44             fw.append(((JSONObject)json.get(k)).getString
("date"));
45             fw.append(' ');
46             fw.append(((JSONObject)json.get(k)).getString
("hour"));
47             fw.append(' ');
48             fw.append(((JSONObject)json.get(k)).getString
("minute"));
49             fw.append(' ');
50             fw.append(((JSONObject)json.get(k)).getString
(keys.get(0)));
51             fw.append(' ');
52             fw.append(((JSONObject)json.get(k)).getString
(keys.get(1)));
53             fw.append(' ');
54             fw.append(((JSONObject)json.get(k)).getString
(keys.get(2)));
55             fw.append(' ');
56             fw.append(((JSONObject)json.get(k)).getString
(keys.get(3)));
57             fw.append(' ');
58             fw.append('\n');
59         }
60         fw.close();
61     } catch (Exception e) {
62     }
63 }
64 }.start();

```

Listing 5.38: saving chart data

Other methods

The other methods visible in this activity are similar in operation to the last two previously described. The only difference lies in the fact that they concern the download of images and data regarding the other two charts in the activity.

5.2.6 GeneralHistoryActivity

This activity is used to visualize the history of measurements in a time range selected by the user. The source code is totally identical to that of the History activity with the only difference being the addition of two DatePickerDialogs for selecting the two dates.

When the user clicks on one of the two edit text that are used to enter the date, the "start_date" or "end_date" functions are executed. The functions do the same job and have the task of creating and displaying a DatePickerDialog and setting a listener, declared in the onCreate() method, capable of filling the contents of the edit text with the date selected when the dialog is closed.

```
1 public void start_date(View view){
2     Calendar cal = Calendar.getInstance();
3     int year = cal.get(Calendar.YEAR);
4     int month = cal.get(Calendar.MONTH);
5     int day = cal.get(Calendar.DAY_OF_MONTH);
6
7     DatePickerDialog dialog = new DatePickerDialog(
8         GeneralHistoryActivity.this,
9         android.R.style.Theme_Holo_Light_Dialog_MinWidth,
10        mDateListenerStart,
11        year, month, day
12    );
13    dialog.getWindow().setBackgroundDrawable( new ColorDrawable(
14        Color.TRANSPARENT ) );
15    dialog.show();
16 }
```

Listing 5.39: startethod

Later can be seen the code for the initialization of the listener for closing the dialog linked to the start edit text.

```
1 mDateListenerStart = new DatePickerDialog.OnDateSetListener() {
2     @Override
3     public void onDateSet(DatePicker datePicker , int year ,
4         int month , int day) {
5         month = month + 1;
6
7         String strMonth = "";
8         String strDay = "";
```

```

8
9         if (month < 10) {
10             strMonth = "0" + month;
11         } else {
12             strMonth = Integer.toString(month);
13         }
14
15         if (day < 10) {
16             strDay = "0" + day;
17         } else {
18             strDay = Integer.toString(day);
19         }
20
21         edtStart.setText( new StringBuilder().append( year ).
append( "-" )
22                             .append( strMonth ).append( "-" ).append(
strDay ) );
23     }
24 };

```

Listing 5.40: listener for DatePickerDialog

5.2.7 ShowProfileActivity

The only task of this activity is to show the data of the account corresponding to the currently logged in user. The code in the `onCreate()` method sends the request to the web service to retrieve the information it needs to display.

The only peculiarity of this activity lies in the fact that, once the button to navigate to the `EditProfileActivity` is clicked, it is called with the `startActivityForResult()` method. This means that once the child activity is closed, the code will execute the `onActivityResult()` method. The mechanism has been inserted in the code to allow updating the displayed data after any modification (completed successfully), without having to request them again from the server.

```

1     public void editProfile(View view){
2         Intent intent = new Intent(this, EditProfileActivity.class);
3         Bundle b = new Bundle();
4         b.putString("email", email.getText().toString());
5         b.putString("name", name.getText().toString());
6         b.putString("surname", surname.getText().toString());
7         b.putString("birth", birth.getText().toString());
8         intent.putExtras(b);
9         startActivityForResult(intent, 1);
10    }
11
12    @Override

```

```

13 public void onActivityResult(int requestCode, int resultCode,
14 Intent data) {
15     super.onActivityResult(requestCode, resultCode, data);
16     switch(requestCode) {
17         case (1) : {
18             if (resultCode == Activity.RESULT_OK) {
19                 Bundle b = data.getExtras();
20                 int value = -1; // or other values
21                 if(b != null) {
22                     email.setText(b.getString("email"));
23                     name.setText(b.getString("name"));
24                     surname.setText(b.getString("surname"));
25                     birth.setText(b.getString("birth"));
26                 }
27             }
28             break;
29         }
30     }
}

```

Listing 5.41: startActivityForResult procedure

5.2.8 EditProfileActivity

The code contained within this activity is used to send the changes made on user data to the web service. In the onCreate() method, all the objects present within the layout are initialized, while the "save" method sends the request to the server.

The only noteworthy piece of code is represented by how the result is entered to be sent to the parent activity after the data has been changed.

```

1 Intent resultIntent = new Intent();
2 Bundle b = new Bundle();
3 b.putString("email", email.getText().toString());
4 b.putString("name", name.getText().toString());
5 b.putString("surname", surname.getText().toString());
6 b.putString("birth", birth.getText().toString());
7 resultIntent.putExtras(b);
8 setResult(Activity.RESULT_OK, resultIntent);

```

Listing 5.42: setResult procedure

5.2.9 ToolsActivity

This activity has the purpose of saving the base address of the web service within the shared preferences together with the zoom setting on the current position of the user on the map during the execution of the PollutionActivity.

By clicking on the “Save” button the settings are saved.

```

1    public void save(View view){
2        SharedPreferences.Editor editor = sharedPref.edit();
3        editor.putString("host", edtHost.getText().toString());
4        editor.putBoolean("myloc", swtLoc.isChecked());
5        editor.apply();
6        Toast.makeText(getApplicationContext(), "Saved", Toast.
LENGTH_LONG).show();
7    }

```

Listing 5.43: save method

5.3 Layouts

The layouts are files containing xml code for the representation of the graphics visible on the screen. For each activity within the "res/layout" folder there is a corresponding layout file. This happens because each activity has its own graphics to display.

Each element visible on the screen can be added by inserting specific tags specific to each component. The insertable tags can represent both real elements and layouts, that is how they should be arranged on the screen.

Since all the layouts are very similar to each other, three representative files are described below for the description of each element used by this project:

- activity_main.xml
- activity_pollution.xml
- activity_history.xml

5.3.1 activity_main.xml

This layout is linked to the MainActivity, which means that it is the first to be loaded and displayed. Inside, first of all another file is included containing the description of the navigation bar. In the next part, the side menu is added, which in turn is made up of two other xml files.

```

1    <?xml version="1.0" encoding="utf-8"?>
2    <androidx.drawerlayout.widget.DrawerLayout xmlns:android="http://
schemas.android.com/apk/res/android"
3        xmlns:app="http://schemas.android.com/apk/res-auto"
4        xmlns:tools="http://schemas.android.com/tools"
5        android:id="@+id/drawer_layout"
6        android:layout_width="match_parent"

```



```

7     android:layout_height="match_parent"
8     android:fitsSystemWindows="true"
9     tools:openDrawer="start">
10
11     <include
12         layout="@layout/app_bar_main"
13         android:layout_width="match_parent"
14         android:layout_height="match_parent" />
15
16     <com.google.android.material.navigation.NavigationView
17         android:id="@+id/nav_view"
18         android:layout_width="wrap_content"
19         android:layout_height="match_parent"
20         android:layout_gravity="start"
21         android:fitsSystemWindows="true"
22         app:headerLayout="@layout/nav_header_main"
23         app:menu="@menu/activity_main_drawer" />
24
25 </androidx.drawerlayout.widget.DrawerLayout>

```

Listing 5.44: activity_main.xml**app_bar_main.xml**

This file is included for inserting the application toolbar. It also adds the main content displayed by this activity through the inclusion of the content_main.xml file.

```

1 <?xml version="1.0" encoding="utf-8"?>
2 <androidx.coordinatorlayout.widget.CoordinatorLayout xmlns:android="
3     http://schemas.android.com/apk/res/android"
4     xmlns:app="http://schemas.android.com/apk/res-auto"
5     xmlns:tools="http://schemas.android.com/tools"
6     android:layout_width="match_parent"
7     android:layout_height="match_parent"
8     tools:context=".MainActivity">
9
10     <com.google.android.material.appbar.AppBarLayout
11         android:layout_width="match_parent"
12         android:layout_height="wrap_content"
13         android:theme="@style/AppTheme.AppBarOverlay">
14
15         <androidx.appcompat.widget.Toolbar
16             android:id="@+id/toolbar"
17             android:layout_width="match_parent"
18             android:layout_height="?attr/actionBarSize"
19             android:background="?attr/colorPrimary"
20             app:popupTheme="@style/AppTheme.PopupOverlay" />

```

```

20
21     </com.google.android.material.appbar.AppBarLayout>
22
23     <include layout="@layout/content_main" />
24
25 </androidx.coordinatorlayout.widget.CoordinatorLayout>

```

Listing 5.45: app_bar_main.xml**content_main.xml**

Inside the file, first of all, a progress bar is inserted which has the task of showing the loading procedure during requests to the web service. This element is initially set with visibility set to "gone" in order not to take up space on the screen as no loading should be visible when the application is started. The elevation value is used to indicate that the progress bar must appear above every other element on the screen if it is made visible.

Then the string "welcome" is inserted with a relative layout for the positioning of the buttons on this screen. Inside, the elements are still inserted in two linear layouts to ensure a correct arrangement of the components on the screen. As can be seen from the code below, the "Pollution" and "History" buttons are actually pairs of ImageView and TextView. In this configuration, the image is made clickable and the text is used to describe the action performed.

Finally, two Buttons are inserted, one for login and one for logout, visible on the screen depending on the case if the user has already logged into the application.

```

1 <?xml version="1.0" encoding="utf-8"?>
2 <androidx.constraintlayout.widget.ConstraintLayout xmlns:android="
  http://schemas.android.com/apk/res/android "
3   xmlns:app="http://schemas.android.com/apk/res-auto "
4   xmlns:tools="http://schemas.android.com/tools "
5   android:layout_width="match_parent "
6   android:layout_height="match_parent "
7   android:background="@color/colorAccent "
8   app:layout_behavior="@string/appbar_scrolling_view_behavior "
9   tools:context=".MainActivity">
10
11   <ProgressBar
12     android:id="@+id/loadData "
13     style="?android:attr/progressBarStyle "
14     android:layout_width="wrap_content "
15     android:layout_height="wrap_content "
16     android:visibility="gone "
17     android:elevation="7dp "
18     app:layout_constraintBottom_toBottomOf="parent "
19     app:layout_constraintEnd_toEndOf="@+id/relativeLayout "

```

```

20         app:layout_constraintStart_toStartOf="parent "
21         app:layout_constraintTop_toTopOf="parent " />
22
23     <TextView
24         android:id="@+id/textView "
25         android:layout_width="wrap_content "
26         android:layout_height="wrap_content "
27         android:fontFamily="@font/berkshire_swash "
28         android:text="@string/welcome "
29         android:textColor="@android:color/white "
30         android:textSize="36sp "
31
32         app:layout_constraintBottom_toTopOf="@+id/relativeLayout "
33         app:layout_constraintEnd_toEndOf="parent "
34         app:layout_constraintHorizontal_bias="0.496 "
35         app:layout_constraintStart_toStartOf="parent "
36         app:layout_constraintTop_toTopOf="parent "
37         app:layout_constraintVertical_bias="0.479 " />
38
39     <RelativeLayout
40         android:id="@+id/relativeLayout "
41         android:layout_width="fill_parent "
42         android:layout_height="wrap_content "
43         android:layout_marginStart="8dp "
44         android:layout_marginTop="8dp "
45         android:layout_marginEnd="8dp "
46         android:layout_marginBottom="8dp "
47         android:background="#fff "
48         android:orientation="vertical "
49         android:padding="20dp "
50         app:layout_constraintBottom_toBottomOf="parent "
51         app:layout_constraintEnd_toEndOf="parent "
52         app:layout_constraintStart_toStartOf="parent "
53         app:layout_constraintTop_toTopOf="parent ">
54
55     <LinearLayout
56         android:layout_width="match_parent "
57         android:layout_height="wrap_content "
58         android:orientation="vertical">
59
60         <LinearLayout
61             android:id="@+id/mainPanel "
62             android:layout_width="match_parent "
63             android:layout_height="wrap_content "
64             android:clipToPadding="false "
65             android:orientation="horizontal">
66
67             <androidx.cardview.widget.CardView
68                 android:id="@+id/seeRestaurants "

```

```

69         android:layout_width="match_parent"
70         android:layout_height="match_parent"
71         android:layout_gravity="center_vertical"
72         android:layout_marginEnd="8dp"
73         android:layout_marginBottom="8dp"
74         android:layout_weight="1"
75         app:cardCornerRadius="36dp"
76         android:layout_marginRight="8dp"
77         android:layout_marginTop="10dp">
78
79         <LinearLayout
80             android:layout_width="match_parent"
81             android:layout_height="wrap_content"
82             android:layout_gravity="center"
83             android:orientation="vertical">
84
85             <ImageView
86                 android:id="@+id/pollutionImgView"
87                 android:layout_width="match_parent"
88                 android:layout_height="80dp"
89                 android:layout_gravity="center"
90                 android:onClick="pollution"
91                 app:srcCompat="@drawable/pollution_icon"
92                 tools:srcCompat="@drawable/pollution_icon"
93             "/>
94
95             <TextView
96                 android:id="@+id/textView6"
97                 android:layout_width="match_parent"
98                 android:layout_height="wrap_content"
99                 android:text="@string/pollution"
100                 android:textAlignment="center"
101                 android:textAppearance="@style/
TextAppearance.AppCompat.Display1"
102                 android:textColor="#000000"
103                 android:textSize="25sp"
104                 android:textStyle="bold"
105                 android:gravity="center_horizontal" />
106         </LinearLayout>
107     </androidx.cardview.widget.CardView>
108
109     <androidx.cardview.widget.CardView
110         android:id="@+id/seeOrders"
111         android:layout_width="match_parent"
112         android:layout_height="match_parent"
113         android:layout_gravity="center_vertical"
114         android:layout_marginStart="8dp"
115         android:layout_marginBottom="8dp"
116         android:layout_weight="1"

```

```

116         app:cardBackgroundColor="#BFFFFFFF"
117         app:cardCornerRadius="36dp"
118         android:layout_marginLeft="8dp"
119         android:layout_marginTop="10dp">
120
121         <LinearLayout
122             android:layout_width="match_parent"
123             android:layout_height="wrap_content"
124             android:orientation="vertical"
125             android:background="#BFFFFFFF">
126
127             <ImageView
128                 android:id="@+id/infoImgView"
129                 android:layout_width="match_parent"
130                 android:layout_height="88dp"
131                 android:layout_gravity="center"
132                 android:onClick="generalHistory"
133                 app:srcCompat="@drawable/history"
134                 tools:srcCompat="@drawable/history" />
135
136                 <TextView
137                     android:id="@+id/textView5"
138                     android:layout_width="match_parent"
139                     android:layout_height="wrap_content"
140                     android:text="@string/menu_history"
141                     android:textAlignment="center"
142                     android:textAppearance="@style/
TextAppearance.AppCompat.Display1"
143                     android:textColor="#000000"
144                     android:textSize="25sp"
145                     android:textStyle="bold"
146                     android:gravity="center_horizontal" />
147                 </LinearLayout>
148             </androidx.cardview.widget.CardView>
149         </LinearLayout>
150         <Button
151             android:id="@+id/btnLogout"
152             android:layout_width="fill_parent"
153             android:layout_height="wrap_content"
154             android:layout_margin="22dp"
155             android:background="#d67601"
156             android:text="@string/logout"
157             android:textAllCaps="false"
158             android:textColor="#fff"
159             android:textSize="18sp"
160             android:onClick="logout"/>
161
162         <Button
163             android:id="@+id/btnLogIn"

```

```

164         android:layout_width="fill_parent "
165         android:layout_height="wrap_content "
166         android:layout_margin="22dp"
167         android:background="#d67601 "
168         android:text="@string/login "
169         android:textAllCaps="false "
170         android:textColor="#fff "
171         android:textSize="18sp "
172         android:onClick="login"/>
173     </LinearLayout>
174 </RelativeLayout>
175 </androidx.constraintlayout.widget.ConstraintLayout>

```

Listing 5.46: cntent_main.xml**nav_header_main.xml**

This file is included by the side menu entry code and represents its other part. Inside it only the string representing the currently logged in user can be seen.

```

1 <?xml version="1.0" encoding="utf-8"?>
2 <LinearLayout xmlns:android="http://schemas.android.com/apk/res/
  android"
3     xmlns:app="http://schemas.android.com/apk/res-auto "
4     xmlns:tools="http://schemas.android.com/tools "
5     android:layout_width="match_parent "
6     android:layout_height="@dimen/nav_header_height "
7     android:background="@drawable/side_nav_bar "
8     android:gravity="bottom "
9     android:orientation="vertical "
10    android:paddingLeft="@dimen/activity_horizontal_margin "
11    android:paddingTop="@dimen/activity_vertical_margin "
12    android:paddingRight="@dimen/activity_horizontal_margin "
13    android:paddingBottom="@dimen/activity_vertical_margin "
14    android:backgroundTint="@color/colorPrimary "
15    android:theme="@style/ThemeOverlay.AppCompat.Dark">
16
17    <TextView
18        android:id="@+id/nav_email "
19        android:layout_width="wrap_content "
20        android:layout_height="wrap_content "
21        android:text="email@email.com" />
22
23 </LinearLayout>

```

Listing 5.47: nav_header_main.xml

activity_main_drawer.xml

This file contained within the "res/menu" folder specifies the choice options visible in the side menu. The insertion takes place through the use of "item" tags within the "group" tag.

```

1 <?xml version="1.0" encoding="utf-8"?>
2 <menu xmlns:android="http://schemas.android.com/apk/res/android"
3     xmlns:tools="http://schemas.android.com/tools"
4     tools:showIn="navigation_view">
5
6     <group android:checkableBehavior="single">
7         <item
8             android:id="@+id/profile"
9             android:icon="@drawable/user"
10            android:title="@string/menu_profile" />
11        <item
12            android:id="@+id/tools"
13            android:icon="@drawable/ic_menu_manage"
14            android:title="@string/menu_tools" />
15    </group>
16 </menu>

```

Listing 5.48: activity_main_drawer.xml

5.3.2 activity_pollution.xml

The layout mainly consists of the MapView tag which has the task of inserting the map inside it. This occupies the entire visible surface on the screen. In addition, two buttons are added within a LinearLayout with higher elevation to be visible above the map. Finally, there is the progress bar to indicate its loading.

```

1 <?xml version="1.0" encoding="utf-8"?>
2 <androidx.constraintlayout.widget.ConstraintLayout
3     xmlns:android="http://schemas.android.com/apk/res/android"
4     xmlns:app="http://schemas.android.com/apk/res-auto"
5     xmlns:tools="http://schemas.android.com/tools"
6     android:orientation="vertical"
7     android:layout_width="fill_parent"
8     android:layout_height="fill_parent">
9
10    <LinearLayout
11        android:layout_width="match_parent"
12        android:layout_height="wrap_content"
13        app:layout_constraintStart_toStartOf="parent"
14        app:layout_constraintTop_toTopOf="parent"
15        android:layout_marginTop="10dp"
16        android:layout_marginStart="10dp"

```

```

17         android:layout_marginEnd="10dp"
18         android:elevation="7dp">
19
20     <Button
21         android:id="@+id/btnpm25"
22         android:layout_width="wrap_content"
23         android:layout_height="wrap_content"
24         android:layout_weight="1"
25         android:text="@string/pm25"
26         android:textColor="#fff"
27         android:onClick="pm25"/>
28
29     <Button
30         android:id="@+id/btnpm10"
31         android:layout_width="wrap_content"
32         android:layout_height="wrap_content"
33         android:layout_weight="1"
34         android:text="@string/pm10"
35         android:textColor="#fff"
36         android:onClick="pm10"/>
37 </LinearLayout>
38
39 <org.osmdroid.views.MapView
40     android:id="@+id/map"
41     android:layout_width="fill_parent"
42     android:layout_height="fill_parent"
43     app:layout_constraintBottom_toBottomOf="parent"
44     app:layout_constraintEnd_toEndOf="parent"
45     app:layout_constraintStart_toStartOf="parent"
46     app:layout_constraintTop_toTopOf="parent" />
47
48 <ProgressBar
49     android:id="@+id/loadData"
50     style="?android:attr/progressBarStyle"
51     android:layout_width="wrap_content"
52     android:layout_height="wrap_content"
53     android:visibility="gone"
54     android:elevation="7dp"
55     app:layout_constraintBottom_toBottomOf="parent"
56     app:layout_constraintEnd_toEndOf="parent"
57     app:layout_constraintStart_toStartOf="parent"
58     app:layout_constraintTop_toTopOf="parent" />
59
60 </androidx.constraintlayout.widget.ConstraintLayout>

```

Listing 5.49: activity_pollution.xml

5.3.3 activity__history.xml

The content of this layout is inserted inside a ScrollView to allow the user to scroll the content of the view. It consists of a spinner for choosing the board id whose data must be displayed followed by three identical pieces of code for the insertion of a descriptive string, the chart and two buttons to give the possibility to download the content. Charts are added thanks to the LineChart tag contained within the external MPAndroidChart library.

```

1 <?xml version="1.0" encoding="utf-8"?>
2 <androidx.constraintlayout.widget.ConstraintLayout
3     xmlns:android="http://schemas.android.com/apk/res/android"
4     xmlns:app="http://schemas.android.com/apk/res-auto"
5     xmlns:tools="http://schemas.android.com/tools"
6     android:layout_width="match_parent"
7     android:layout_height="match_parent"
8     android:background="@color/colorAccent">
9
10    <ScrollView
11        android:layout_width="match_parent"
12        android:layout_height="match_parent">
13
14        <LinearLayout
15            android:layout_width="match_parent"
16            android:layout_height="wrap_content"
17            android:orientation="vertical"
18            android:textAlignment="center"
19            android:gravity="center_horizontal">
20
21            <TextView
22                android:id="@+id/textView"
23                android:layout_width="wrap_content"
24                android:layout_height="wrap_content"
25                android:fontFamily="@font/berkshire_swash"
26                android:text="@string/last_history"
27                android:textColor="@android:color/white"
28                android:textSize="36sp"
29                android:paddingTop="20dp"
30                android:paddingBottom="20dp"/>
31
32            <RelativeLayout
33                android:id="@+id/relativeLayout"
34                android:layout_width="fill_parent"
35                android:layout_height="wrap_content"
36                android:layout_marginStart="8dp"
37                android:layout_marginTop="8dp"
38                android:layout_marginEnd="8dp"
39                android:layout_marginBottom="8dp"
40                android:background="#fff"

```

```
41         android:orientation="vertical"
42         android:padding="20dp">
43
44     <LinearLayout
45         android:layout_width="fill_parent"
46         android:layout_height="wrap_content"
47         android:orientation="vertical">
48
49
50
51         <LinearLayout
52             android:layout_width="match_parent"
53             android:layout_height="match_parent"
54             android:orientation="horizontal"
55             android:layout_marginTop="10dp">
56
57             <TextView
58                 android:id="@+id/textView2"
59                 android:layout_width="wrap_content"
60                 android:layout_height="match_parent"
61                 android:text="@string/board_id"
62                 android:textSize="16dp"/>
63
64             <Spinner
65                 android:id="@+id/spnBoards"
66                 android:layout_width="match_parent"
67                 android:layout_height="match_parent"
68                 android:paddingLeft="32dp"
69                 android:singleLine="true"
70                 android:textAlignment="center"
71                 android:gravity="center_horizontal"
72                 android:layout_marginBottom="5dp"/>
73
74         </LinearLayout>
75
76
77     </LinearLayout>
78 </RelativeLayout>
79
80 <TextView
81     android:id="@+id/txtCharts"
82     android:layout_width="wrap_content"
83     android:layout_height="match_parent"
84     android:text="@string/sensor_chart"
85     android:fontFamily="@font/berkshire_swash"
86     android:textColor="@android:color/white"
87     android:textSize="16dp"
88     android:visibility="gone"
89     android:layout_marginBottom="8dp"/>
```

```

90
91 <com.github.mikephil.charting.charts.LineChart
92     android:id="@+id/chart"
93     android:layout_width="match_parent"
94     android:layout_height="300dp"
95     android:visibility="gone"
96     android:layout_marginStart="8dp"
97     android:layout_marginEnd="8dp"
98     android:layout_marginBottom="4dp"/>
99
100 <LinearLayout
101     android:id="@+id/btnSens"
102     android:layout_width="wrap_content"
103     android:layout_height="match_parent"
104     android:visibility="gone"
105     android:layout_marginBottom="10dp">
106
107     <Button
108         android:id="@+id/btnSaveSensImage"
109         android:layout_height="wrap_content"
110         android:layout_width="wrap_content"
111         android:text="@string/save_image"
112         android:onClick="saveSensImage"
113     />
114
115     <Button
116         android:id="@+id/btnSaveSensData"
117         android:layout_height="wrap_content"
118         android:layout_width="wrap_content"
119         android:text="@string/save_data"
120         android:onClick="saveSensData"
121     />
122
123 </LinearLayout>
124
125 <TextView
126     android:id="@+id/txtChartT"
127     android:layout_width="wrap_content"
128     android:layout_height="match_parent"
129     android:text="@string/temp_chart"
130     android:fontFamily="@font/berkshire_swash"
131     android:textColor="@android:color/white"
132     android:textSize="16dp"
133     android:visibility="gone"
134     android:layout_marginBottom="8dp"/>
135
136 <com.github.mikephil.charting.charts.LineChart
137     android:id="@+id/chartTemp"
138     android:layout_width="match_parent"

```

```

139         android:layout_height="300dp"
140         android:visibility="gone"
141         android:layout_marginStart="8dp"
142         android:layout_marginEnd="8dp"
143         android:layout_marginBottom="4dp"/>
144
145     <LinearLayout
146         android:id="@+id/btnTemp"
147         android:layout_width="wrap_content"
148         android:layout_height="match_parent"
149         android:visibility="gone"
150         android:layout_marginBottom="10dp">
151
152         <Button
153             android:id="@+id/btnSaveTempImage"
154             android:layout_height="wrap_content"
155             android:layout_width="wrap_content"
156             android:text="@string/save_image"
157             android:onClick="saveTempImage"
158             />
159
160         <Button
161             android:id="@+id/btnSaveTempData"
162             android:layout_height="wrap_content"
163             android:layout_width="wrap_content"
164             android:text="@string/save_data"
165             android:onClick="saveTempData"
166             />
167
168     </LinearLayout>
169
170     <TextView
171         android:id="@+id/txtChartH"
172         android:layout_width="wrap_content"
173         android:layout_height="match_parent"
174         android:text="@string/humidity_chart"
175         android:fontFamily="@font/berkshire_swash"
176         android:textColor="@android:color/white"
177         android:textSize="16dp"
178         android:visibility="gone"
179         android:layout_marginBottom="8dp"/>
180
181     <com.github.mikephil.charting.charts.LineChart
182         android:id="@+id/chartHumidity"
183         android:layout_width="match_parent"
184         android:layout_height="300dp"
185         android:visibility="gone"
186         android:layout_marginStart="8dp"
187         android:layout_marginEnd="8dp"

```

```

188         android:layout_marginBottom="4dp"/>
189
190     <LinearLayout
191         android:id="@+id/btnHumidity"
192         android:layout_width="wrap_content"
193         android:layout_height="match_parent"
194         android:visibility="gone"
195         android:layout_marginBottom="10dp">
196
197         <Button
198             android:id="@+id/btnSaveHumidityImage"
199             android:layout_height="wrap_content"
200             android:layout_width="wrap_content"
201             android:text="@string/save_image"
202             android:onClick="saveHumidityImage"
203             />
204
205         <Button
206             android:id="@+id/btnSaveHumidityData"
207             android:layout_height="wrap_content"
208             android:layout_width="wrap_content"
209             android:text="@string/save_data"
210             android:onClick="saveHumidityData"
211             />
212
213     </LinearLayout>
214
215
216
217     </LinearLayout>
218 </ScrollView>
219
220 <ProgressBar
221     android:id="@+id/loadData"
222     style="?android:attr/progressBarStyle"
223     android:layout_width="wrap_content"
224     android:layout_height="wrap_content"
225     android:visibility="gone"
226     android:elevation="7dp"
227     app:layout_constraintEnd_toEndOf="parent"
228     app:layout_constraintStart_toStartOf="parent"
229     app:layout_constraintTop_toTopOf="parent"
230     android:layout_marginTop="115dp"/>
231
232
233 </androidx.constraintlayout.widget.ConstraintLayout>

```

Listing 5.50: activity_history.xml

5.4 Values

This folder contains xml files that specify some static layout resources. Among the most important files are colors.xml and strings.xml.

Within the first, the colors used by the various file layouts can be defined. The strength of using this approach lies in the fact that by changing a single resource it is possible to change the colors in the entire application.

```

1 <?xml version="1.0" encoding="utf-8"?>
2 <resources>
3     <color name="colorPrimary">#125688</color>
4     <color name="colorPrimaryDark">#125688</color>
5     <color name="textColorPrimary">#FFFFFF</color>
6     <color name="windowBackground">#FFFFFF</color>
7     <color name="colorAccent">#d67601</color>
8     <color name="selected">#8CFF8C00</color>
9 </resources>

```

Listing 5.51: colors.xml

The second file defines all the strings displayed in the various layouts. By default, the application uses the strings.xml file but the corresponding files for each language the application want to support can also be defined. This approach allows the application to show the strings in every possible language by defining them within a single file without having to change all the layouts. Below the definition of some of the strings available for this application can be seen.

```

1 <string name="menu_tools">Tools</string>
2     <string name="menu_history">History</string>
3     <string name="login">Login</string>
4     <string name="logout">Logout</string>
5     <string name="email">Email</string>
6     <string name="pass">Password</string>
7     <string name="notUser">Not an User?</string>
8     <string name="welcome_back">Welcome Back</string>
9     <string name="notImplemented">This feature will be added soon</
string>
10     <string name="edit">EDIT</string>
11     <string name="save">SAVE</string>

```

Listing 5.52: strings.xml

5.5 build.gradle(app)

This configuration file allows the application to link some dependencies of external libraries within the project by inserting them within the "dependencies" tag.

The external libraries used by this application are eg. "MPAndroidChart" or "osmdroid" which respectively allow the creation of charts and management of the map. The code below displays the process of adding external dependencies.

```
1 dependencies {
2     implementation fileTree(dir: 'libs', include: ['*.jar'])
3     implementation 'androidx.appcompat:appcompat:1.0.2'
4     implementation 'androidx.legacy:legacy-support-v4:1.0.0'
5     implementation 'com.google.android.material:material:1.0.0'
6     implementation 'androidx.constraintlayout:constraintlayout:1.1.3'
7     implementation 'androidx.navigation:navigation-fragment:2.0.0'
8     implementation 'androidx.navigation:navigation-ui:2.0.0'
9     implementation 'androidx.lifecycle:lifecycle-extensions:2.0.0'
10    implementation 'com.android.volley:volley:1.1.1'
11    testImplementation 'junit:junit:4.12'
12    androidTestImplementation 'androidx.test.ext:junit:1.1.1'
13    androidTestImplementation 'androidx.test.espresso:espresso-core:3.2.0'
14    implementation 'org.osmdroid:osmdroid-android:6.0.1'
15    implementation 'com.google.android.gms:play-services-location:17.0.0'
16    implementation 'com.github.PhilJay:MPAndroidChart:v3.1.0'
17 }
```

Listing 5.53: dependencies

Chapter 6

Testing and evaluation

This chapter describes the testing phase of the thesis project, explaining how the tests were performed together with an analysis of the obtained results.

6.1 Testing context

During the testing phase of the project, all attention was paid to the web service. Due to the way the work has been structured, the server is the central point of the entire data distribution system and it is its responsibility to perform all the tasks that require long computation time, as well as keeping the data updated within the DB. This means that the stability of the backend is a priority and needs more attention during the testing phase. It should also be noted that the tests for the correct functioning of the server can be performed locally, by writing a testing client capable of sending http requests, as there is only one instance of the server, while to test the mobile application it is first necessary to run the deployment of the web service, subsequently distributing the application to a group of reliable users within the Politecnico di Torino to check its operation.

The tests involving a large number of people is very difficult to organize so all the attention was focused on the backend.

6.2 Testing scripts

To test the functioning of the web service, two scripts were written, located in the "testing_client" folder. The task of the first program (testPUT.py) is to send many simultaneous http requests to the server to check that they are executed correctly.

Since the web service, in addition to being a data distribution system, is mainly a measurement collection system with the task of making them permanent on the server, the requests forwarded by the testing client to the backend are PUT

requests to the address that ends with “/measure”, this means that they have to insert data into the DB and they represents one of the most frequent requests received by the server. By using requests of this type, it is also possible to check if the data is inserted correctly within the database.

The script inside contains a function called "add_data" which has the task of generating random measurements for board 0 and inserting them both within a local DB and sending them with the http PUT request to the web service. The number of data generated can be specified by setting the number of iterations within the for loop.

```
1 def add_data(index):
2     num = random.randint(2, 5)
3     time.sleep(num)
4     print("Thread " + str(index) + " sleeps " + str(num) + " seconds")
5
6     t_conn = sqlite3.connect('TestDB.db', timeout=600)
7     t_c = t_conn.cursor()
8
9     # variables to create a well formatted json file to send
10    data = {}
11    l = []
12
13    # for to generate random data and save then into the local db
14    for x in range(3600):
15        n = random.randint(2, 9)
16        obj = {"sensorID": 12, "timestamp": 2, "data": n, "geoHash":
17"sadsad", "altitude": 1222}
18        t_c.execute('''
19INSERT INTO measure_table (sensorID, timestamp, data, geoHash
20, altitude)
21VALUES (12, 2, ?, "sadsad", 1222)
22'', [n])
23        l.append(obj)
24        obj = {"sensorID": 14, "timestamp": 2, "data": n, "geoHash":
25"sadsad", "altitude": 1222}
26        t_c.execute('''
27INSERT INTO measure_table (sensorID, timestamp, data,
28geoHash, altitude)
29VALUES (14, 2, ?, "sadsad", 1222)
30'', [n])
31        l.append(obj)
32        obj = {"sensorID": 16, "timestamp": 2, "data": n, "geoHash":
33"sadsad", "altitude": 1222}
34        t_c.execute('''
35INSERT INTO measure_table (sensorID, timestamp, data,
36geoHash, altitude)
37VALUES (16, 2, ?, "sadsad", 1222)
```

```

32         ''' , [n])
33     l.append(obj)
34     obj = {"sensorID": 18, "timestamp": 2, "data": n, "geoHash":
"sadsad", "altitude": 1222}
35     t_c.execute('''
36         INSERT INTO measure_table (sensorID , timestamp , data ,
geoHash , altitude)
37         VALUES (18, 2, ?, "sadsad", 1222)
38         ''' , [n])
39     l.append(obj)
40     n = random.randint(2, 9)
41     obj = {"sensorID": 13, "timestamp": 2, "data": n, "geoHash":
"sadsad", "altitude": 1222}
42     t_c.execute('''
43         INSERT INTO measure_table (sensorID , timestamp , data ,
geoHash , altitude)
44         VALUES (13, 2, ?, "sadsad", 1222)
45         ''' , [n])
46     l.append(obj)
47     obj = {"sensorID": 15, "timestamp": 2, "data": n, "geoHash":
"sadsad", "altitude": 1222}
48     t_c.execute('''
49         INSERT INTO measure_table (sensorID , timestamp , data ,
geoHash , altitude)
50         VALUES (15, 2, ?, "sadsad", 1222)
51         ''' , [n])
52     l.append(obj)
53     obj = {"sensorID": 17, "timestamp": 2, "data": n, "geoHash":
"sadsad", "altitude": 1222}
54     t_c.execute('''
55         INSERT INTO measure_table (sensorID , timestamp , data ,
geoHash , altitude)
56         VALUES (17, 2, ?, "sadsad", 1222)
57         ''' , [n])
58     l.append(obj)
59     obj = {"sensorID": 19, "timestamp": 2, "data": n, "geoHash":
"sadsad", "altitude": 1222}
60     t_c.execute('''
61         INSERT INTO measure_table (sensorID , timestamp , data ,
geoHash , altitude)
62         VALUES (19, 2, ?, "sadsad", 1222)
63         ''' , [n])
64     l.append(obj)
65     n = random.randint(19, 25)
66     obj = {"sensorID": 20, "timestamp": 2, "data": n, "geoHash":
"sadsad", "altitude": 1222}
67     t_c.execute('''
68         INSERT INTO measure_table (sensorID , timestamp , data ,
geoHash , altitude)

```

```

69         VALUES (20, 2, ?, "sadsad", 1222)
70         ''', [n])
71     l.append(obj)
72     obj = {"sensorID": 21, "timestamp": 2, "data": 23323, "
73     geoHash": "sadsad", "altitude": 1222}
74     t_c.execute('''
75         INSERT INTO measure_table (sensorID, timestamp, data,
76         geoHash, altitude)
77         VALUES (21, 2, ?, "sadsad", 1222)
78         ''', [n])
79     l.append(obj)
80     t_conn.commit()
81
82     # create the json data
83     data['data_block'] = 1
84     json_data = json.dumps(data)
85
86     response = requests.put(url, data=json_data, headers=headers)
87
88     res = response.json()
89
90     print(res)
91
92     print("Thread " + str(index) + " finished")

```

Listing 6.1: add_data function

The main code of the script has the task of setting the configuration parameters of the connection to the local DB, creating it if necessary, together with the connection parameters to the web service.

```

1 # create the url amd the headers to send data to the server
2 url = 'http://127.0.0.1:5000/ws/measure'
3 headers = {"Content-Type": "application/json",
4           "x-access-token": "eyJ0eXAiOiJKV1QiLCJhbGciOiJIUzI1NiJ9.eyJ1c2VyX2lkIjo1MSwiY291bnRlciI6MX0.gBzzmx3TlC87VKIv1MrfuGEcUw5tBf-FeJGgIlo"}
5
6 # create local DB
7 conn = sqlite3.connect('TestDB.db')
8 c = conn.cursor()
9 # drop the measure_table first
10 c.execute('''
11 DROP TABLE IF EXISTS measure_table
12 ''')
13 conn.commit()
14 # create the local measure_table
15 c.execute('''CREATE TABLE measure_table

```

```

16         ([measureID] INTEGER PRIMARY KEY,[sensorID] INTEGER , [
            timestamp] integer , [data] FLOAT, [geoHash] VARCHAR(12), [altitude
            ] FLOAT )'''
17 conn.commit()

```

Listing 6.2: setting parameters

At this point some parallel program threads are generated that are able to execute the “add_data” function, thus simulating the clients that is sending data to the web service. Thanks to the use of two for loops it is possible to specify how many threads we want to run simultaneously and how many times the procedure must be repeated.

```

1 for z in range(2):
2     threads = []
3
4     # create the threads and start them
5     for i in range(5):
6         t = Thread(target=add_data, args=(i,))
7         threads.append(t)
8         t.start()
9
10    # wait for the threads
11    for x in threads:
12        x.join()

```

Listing 6.3: creating threads

Finally, the testing client queries the local database to obtain the number of data inserted inside it. This value will subsequently allow us to check if the same number of data has been entered into the DB contacted by the backend.

```

1 # display the number of rows in the measure_table
2 c.execute('''
3 SELECT COUNT(*)
4 FROM measure_table
5 WHERE timestamp = 2
6         ''')
7 print(c.fetchall())

```

Listing 6.4: querying local DB

The second script has the task of testing the latency and throughput of the web service, simulating the execution of a variable number of clients through the use of different threads. The numebr of simulated clients as well as the number of requests to make can be setted at the beginning of the script.

```

1 threads = []
2
3 t0 = time.time()

```

```

4
5 # create the threads and start them
6 for i in range(clients):
7     t = Thread(target=send_requests, args=(i,))
8     threads.append(t)
9     t.start()
10
11
12 # wait for the threads
13 for p in threads:
14     p.join()
15
16 t1 = time.time()
17
18 total = t1 - t0
19
20 lat = gen_latency / clients
21 throughput = n_requests / total
22
23
24 print(str(lat))
25 print(str(throughput))

```

Listing 6.5: creation of multiple threads

In the code visible above it is also possible to see that how the throughput and medium latency is calculated, noting that the time interval for sending and receiving all the requests is measured at this point.

It is also visible that each thread executes the “send_requests” function which has the task of generating GET requests and sending them to the web service. The number of requests to be send is splitted between all the generated clients Each request concerns the measurements of the day "2018-11-10", filtering the data so as to obtain only those relating to particles type pm2.5 and board number 8. This type of request was used as it is one of the heavier on the server.

As requests are generated and sent to the web service, the response time for each is measured, establishing is this way the latency value.

```

1 def send_requests(client_index):
2     global gen_latency
3     x = 0
4
5     url = 'http://127.0.0.1:5000/ws/measure/pm25/8'
6     headers = {"Content-Type": "application/json",
7               "x-access-token": "
eyJ0eXAiOiJKV1QiLCJhbGciOiJIUzI1NiJ9.
eyJ1c2VyX2lkIjoxMTQsImNvdW50ZXIiOiR9.
WAYJylypOamJCboap25MPfA29MqZADNTFSgMco7tLE",
8               "start": "2018-11-10",

```

```
9         "end": "2018-11-10",
10        "board": "True",
11        "filter": "True",
12        "last-two": "False",
13        "from": "9"}
14
15    n = n_requests / clients
16
17    for j in range(int(n)):
18        t2 = time.time()
19        response = requests.get(url, headers=headers)
20        res = response.json()
21        t3 = time.time()
22        partial = t3 - t2
23        x = x + partial
24
25
26    latency = x / (n_requests / clients)
27
28    gen_latency = gen_latency + latency
29
30    print("Client " + str(client_index) + ": \n" + "latency: " + str(
    latency))
```

Listing 6.6: send_requests function

6.3 Testing machine

During all the tests, both regarding the web service and the testing client programs has been executed on a personal computer with the Windows 10 operating system with 64-bit architecture installed. The host machine is equipped with an Intel Core i7-7700 HQ processor with 2.8 GHz clock frequency and have 16 GB of RAM memory.

6.4 Testing procedure

Once the web service has been started on the host machine and after waiting for the backend to calculate the regression coefficients for every sensor for the first time, the server testing process has been as described in the following subsections.

6.4.1 First latency/throughput testing

During this phase the testing script (testGET) was performed exactly three times, configuring the number of simulated clients to 1, 2 and 4 respectively and the

number of request to be sent to 12. This test was performed while the web service is free from any other request in order to check its performances, latency and throughput, in the best case.

6.4.2 First inserting data script execution

In this phase the “testPUT.py” script was executed, configured to create 5 separate threads twice, which sent 3600 data per request to the server. This process was designed to keep the web service busy so that to be able to perform the next testing steps.

6.4.3 Second latency/throughput testing

In this phase the “testGET” script was run again three times, configured in the same way as it was in its previous execution. In this case, however, the web service is still busy by the execution of PUT type requests made by the second testing script, thus allowing to obtain data regarding the average latency and throughput when the server remains under pressure.

6.4.4 Cheking the inserted data

In this phase, it is checked whether the data generated at random by the testing client have been correctly inserted into the web service DB. To perform this test, it was checked whether the amount of data entered in the local DB of the testing client, and displayed at the end of the script execution, is equal to the amount of data inserted in the server DB.

6.4.5 Second execution of the inserting data client

In this phase, the “testGET.py” script was run again with different configuration parameters in order to load, through a single execution of 5 separate threads, a big amount of data for each request sent to the server. In fact, each thread had the task of generating 36,000 samples of random data, corresponding to one hour of measurements on a single board. This test was necessary to check if the web service is able handle large data inserting requests.

6.5 Results

6.5.1 Latency

From the results obtained during the tests it can be observed that the latency is strictly dependent on the number of clients running simultaneously. Both in the case of free and busy servers, the increase is considerable[6.1].

This behavior can be generated by the fact that the execution of the web service and the two test scripts takes place on the same physical machine that does not have the same performance as a computer set up to host the backend. For this reason, the results obtained can be considered satisfactory since even in the worst case the latency slightly exceeds one second; which means acceptable waiting times for the client. From the tests it is also possible to notice how the latency undergoes a slight degradation, of about 0.2s, when the server is busy, which means that the drop in performance is acceptable, especially on a personal computer.

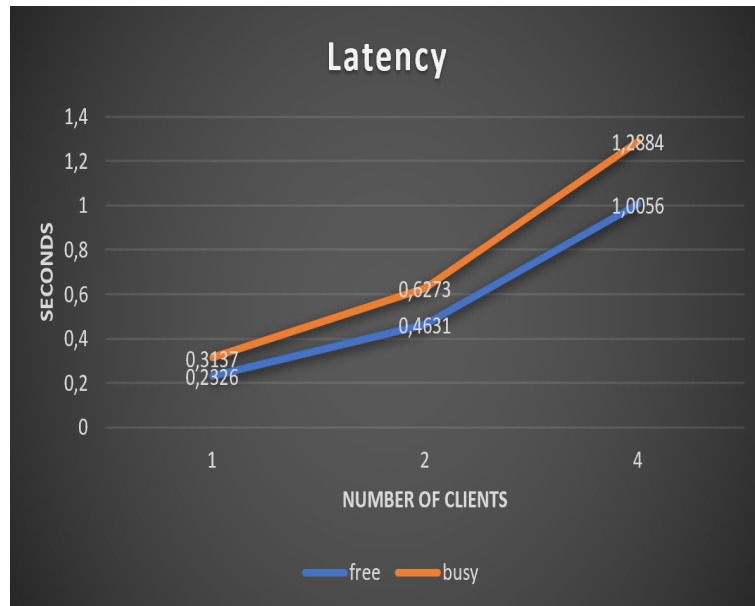


Figure 6.1: Latency results

6.5.2 Throughput

The results obtained in the tests[6.2] show that the throughput is less dependent on the number of clients running at the same time, but more linked to the current situation of the web service. In fact, by analyzing the results obtained, it is possible to note that the performance case due to the increase in the number of clients is always lower than the case due to the occupation of the server. In both

cases, however, the results can be considered satisfactory as the degradation in performance is not so significant, especially considering the fact of running on a personal computer.

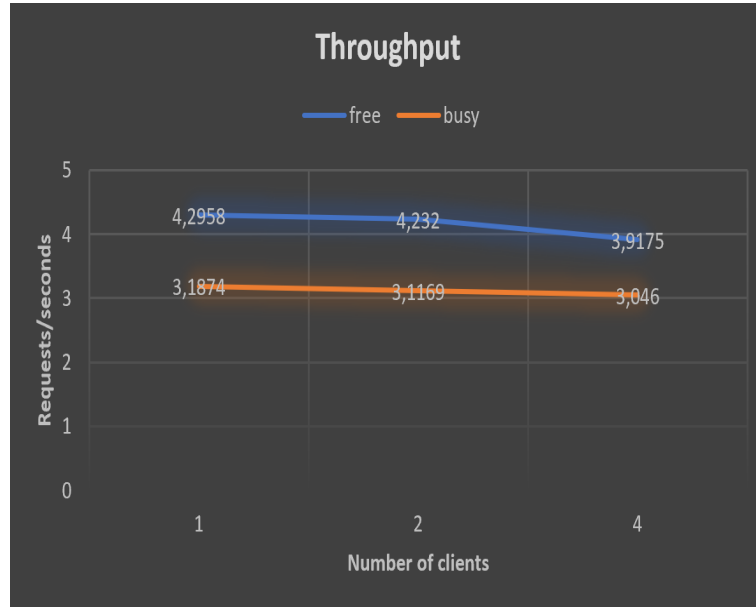


Figure 6.2: throughput results

6.5.3 Correct insertion of data

Analyzing also the aspect of correct data entry within the DB, it can be seen that the first execution of the testing client introduces the creation of 36,000 data records[fig.6.3] within the "TestDB" local to the script. Observing in the meanwhile how the number of records has changed inside the "measure_table" table of the web service data base, before[fig.6.4] and after[fig.6.5] the execution of the script, it can be seen that the number has changed exactly of 36000 units and the data have been entered correctly.

```
[(36000,)]
Process finished with exit code 0
```

Figure 6.3: Local DB records



Figure 6.4: Number of record before script execution



Figure 6.5: Number of record after script execution

From the second execution of the testing client it is possible to observe that it caused the creation of 180000 data records in the local DB [fig. 6.6], and that, after having deleted the random data previously inserted from the web service DB, the number of records is varied by exactly the same amount[fig.6.7]. This test is to be considered satisfactory as the server is able to enter data correctly even in large quantities.

```
[(180000,)]
Process finished with exit code 0
```

Figure 6.6: Local DB records after the second execution of the testing script



Figure 6.7: Number of records after the second execution of the script

Conclusions

The need to monitor environmental pollution is becoming increasingly important, especially in large cities like Turin where the air quality, especially in the winter period, can become very bad. Despite numerous efforts by the authorities, such as car traffic regulations, which can also end up in a total block of traffic in cases of critical pollution, the situation of air quality remains very worrying. For this reason, citizens need to be aware of the quality of the air around them. The monitoring systems of the city of Turin provide very accurate data but, being positioned only in some strategic points of the agglomeration, they are unable to keep the citizen informed about the state of the air in their closest proximity. From this arises the need for a system capable of returning this kind of information to the citizens.

The data distribution system described in this thesis represents an easy and above all immediate way to keep every citizen updated on the state of the surrounding air quality. The use of an Android client for presenting data to the end user allows access to information anywhere in the city. The mobile application also does not appear to be expensive either in terms of battery consumption as it does not have any processes running in the background or from the point of view of memory usage as the data is not saved locally.

Since the central point of the entire system is based on the use of a REST web service, and all data computation is performed remotely, the system is not tied in any way to just using an Android application as a client for data visualization. In fact, the project can be extended at any time by adding other clients based on different operating systems or by creating a website. Moreover, thanks to the modular architecture of the server, it is possible to add new features to the system without affecting the existing code.

In the case of using the system in real situations, it is only necessary to deploy the web service on a suitable server machine and distribute the client to citizens. Once these two operations have been performed, the system is ready for use.

Considering all aspects of the project of this thesis, the data distribution system turns out to be an effective way of informing citizens about air quality, also leaving the doors open to numerous extensions to further improve its operation.

Bibliografy

- [1] *Client Server Architecture*. URL: https://cio-wiki.org/wiki/Client_Server_Architecture.
- [2] *L'Agenzia*. URL: <http://www.arpa.piemonte.it/chi-siamo/lagenzia>.
- [3] *Using MySQL licensing: Open source license vs. commercial license*. URL: <https://searchitchannel.techtarget.com/feature/Using-MySQL-licensing-Open-source-license-vs-commercial-license>.
- [4] S. Suehring. *MySQL Bible*. Wiley, 2002, p. 171.
- [5] *Foreword*. URL: <https://flask.palletsprojects.com/en/1.1.x/foreword/>.
- [6] *How do I create a .pyc file?* URL: <http://effbot.org/pyfaq/how-do-i-create-a-pyc-file.htm>.
- [7] *PEP 206 – Python Advanced Library*. URL: <https://www.python.org/dev/peps/pep-0206/>.
- [8] *THE 10 MOST POPULAR PROGRAMMING LANGUAGES TO LEARN IN 2020*. URL: <https://www.northeastern.edu/graduate/blog/most-popular-programming-languages/>.
- [9] *Web Services Architecture*. URL: <https://www.w3.org/TR/ws-arch/>.
- [10] M. Masse. *REST API Design Rulebook: Designing Consistent RESTful Web Service Interfaces*. O'Reilly Media, Inc, 2011, p. 5.
- [11] *REST, i principi dell'architettura*. URL: <https://www.html.it/pag/19596/i-principi-dellarchitettura-restful/>.
- [12] *Mappare le azioni 'CRUD' sui metodi HTTP*. URL: <https://www.html.it/pag/19597/mappare-le-azioni-crud-sui-metodi-http/>.
- [13] *Stateless, autodefinizione e collegamenti tra risorse*. URL: <https://www.html.it/pag/19598/stateless-autodefinizione-e-collegamenti-tra-risorse/>.

- [14] *4 Maturity Levels of REST API Design*. URL: <https://blog.restcase.com/4-maturity-levels-of-rest-api-design/#:~:text=The%20Richards on%20REST%20Maturity%20Model,3%20in%20this%20maturity%20model.&text=However%2C%20all%20these%20are%20discussing,not%20the%20design%20maturity%20level>.
- [15] *Android Version Distribution statistics will now only be available in Android Studio*. URL: <https://www.xda-developers.com/android-version-distribution-statistics-android-studio/>.
- [16] *Mobile Operating System Market Share Worldwide*. URL: <https://gs.statcounter.com/os-market-share/mobile/worldwide>.
- [17] *Android application components*. URL: <http://www.w3big.com/android/android-application-components.html>.
- [18] *Activity*. URL: <https://developer.android.com/reference/android/app/Activity>.
- [19] *About OpenStreetMap*. URL: https://wiki.openstreetmap.org/wiki/About_OpenStreetMap.
- [20] *Air Quality Index Scale and Color Legend*. URL: <https://aqicn.org/scale/>.
- [21] *The Python IDE for Professional Developers*. URL: <https://www.jetbrains.com/pycharm/>.
- [22] *Modular Applications with Blueprints*. URL: <https://flask.palletsprojects.com/en/1.1.x/blueprints/>.