Politecnico di Torino

Master's Degree in Computer Science Engineering
Academic Year 2020/2021
Graduation Session July 2021

Study and development of a participative air pollution monitoring system

Supervisors
Maurizio Rebaudengo

Filippo Gandino

Candidate
Gabriele Telesca

Contents

Chapter 1

Introduction

1.1 The problem of air pollution

The Earth's atmosphere contains several particles, both solid and liquid, mixed in a gas compound consisting of nitrogen (N_2), oxygen (O_2), carbon dioxide (CO_2) and others. Some of these particles originate naturally, but a significant part of them derives from car emissions, chemicals from factories and combusting fossil fuels.

Indoor and outdoor activities can be strongly affected by the concentration of pollutants in the surrounding area.
Indoor air pollution results from the burning of solid fuels such as crop waste, dung, charcoal and coal for cooking and heating in households. Burning these fuels produces particulate matter – a major health risk, particularly for respiratory diseases [16]. Although nowadays indoor air pollution is still one of the major risk factors for mortality, the number of annual deaths from indoor air pollution has fallen by more than 1 million since 1990 [16], as shown in figure 1.1 on page 4 where the majority of Countries are represented above the grey dotted line (if a Country lies along this line then it has the same number of deaths in both years).
Outdoor air pollution originates mainly from various industrial processes and human activities such as burning fossil fuels in vehicles, smokestacks and electric power plants; though, some pollutants can derive from natural sources including dust storms, volcanoes eruptions and grassland fires. The number of deaths due to outdoor air pollution has increased in most Countries in the world, as shown in figure 1.2 on page 4: the main exceptions are countries in Europe, Australia and New Zealand. In some cases this has occurred because death rates from air pollution have increased, but in many cases, the largest driver of this change has been population growth and ageing populations [17].

Air pollution kills an estimated seven million people worldwide every year. WHO data shows that 9 out of 10 people breathe air that exceeds WHO guideline limits containing high levels of pollutants, with low- and middle-income countries suffering from the highest exposures [14].

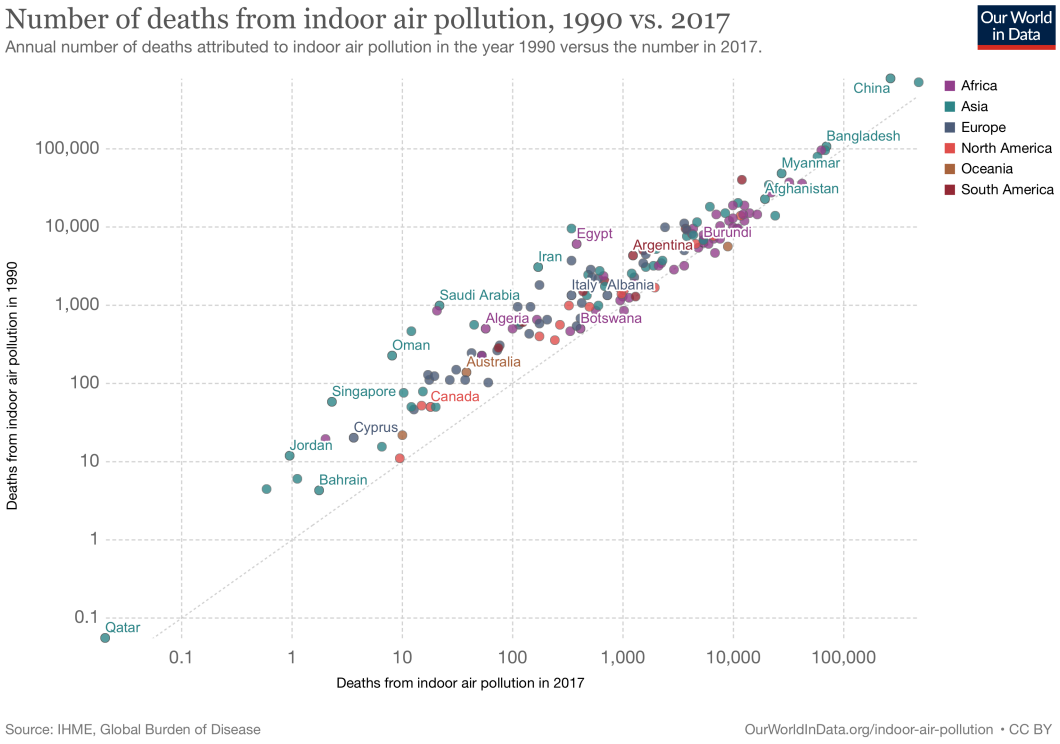

Figure 1.1: Number of deaths from indoor air pollution, 1990 vs. 2017 [18]

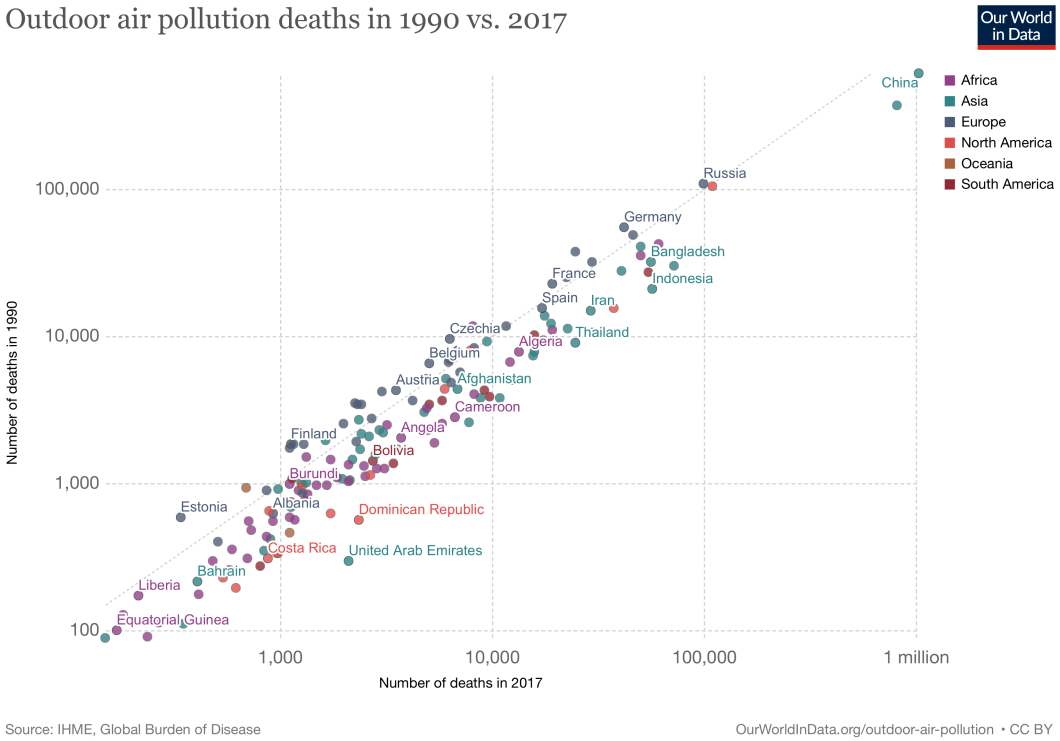

Figure 1.2: Number of deaths from outdoor air pollution, 1990 vs. 2017 [19]

1.1.1 Causes

Alongside ozone pollution, the main contributor to poor health from air pollution is particulate matter. In particular, very small particles of matter – termed '$PM_{2.5}$', which are particles with a size (diameter) of less than 2.5 micrometres (µm). Smaller particles tend to have more adverse health effects because they can enter airways and affect the respiratory system [17].

The combustion of fossil fuels like coal, petroleum and other factory combustibles is a major cause of air pollution. These are generally used in power plants, manufacturing facilities (factories) and waste incinerators, as well as furnaces and other types of fuel-burning heating devices. Providing air conditioning and other services also requires significant amounts of electricity, which in turn leads to more emissions [21].

1.1.2 Effects

The combination of outdoor and indoor exposure to air pollutants is a relevant risk factor for many causes of death, including stroke, lung cancer, heart disease. Considering low-income countries, air pollution tops the list of the most relevant causes of death. In 2017, it was responsible for an estimated 5 million deaths globally: this means that it contributed to 9% deaths [17].

Besides human health's effects, also more general environmental effects must be taken into consideration. If Earth's atmosphere is composed of a certain amount of particles and gases, this balance is essential to the whole ecosystem of the planet. Scientists have pointed out that the environment applies self-regulating mechanisms, but when it comes to the atmosphere then it turns out that human activities are adding pollutants to the air faster than the Earth's natural mechanisms can remove them. The results of this are being felt in terms of acid rain, smog and global warming [21].

1.2 Monitoring air quality

All over the world, most of government bodies have undertaken corrective actions to protect from excessive pollution concentration in the atmosphere, especially in the form of legal directives. For example, in Europe each Member State divides its territory into a number of zones and agglomerations: in these zones and agglomerations, the Member States should undertake assessments of air pollution levels using measurements, modelling and other empirical techniques, and report air quality data to the European Commission accordingly [4]. In case the assessment exceeds the limits or the target values, Member States are required to develop a plan for ensuring compliance within the limits in a reasonable amount of time.

1.2.1 Particulate Matter PM2.5 and PM10

Also referred to as Particle Pollution, the term Particulate Matter describes a mixture composed of solid particles and liquid droplets that can be found in the air. Unlike smoke, dust and soot, these particles are invisible to human eyes. These suspended particles vary

in size, composition and origin. It is convenient to classify particles by their aerodynamic properties because:

- these properties govern the transport and removal of particles from the air;
- they also govern their deposition within the respiratory system;
- they are associated with the chemical composition and sources of particles.

These properties are conveniently summarized by the aerodynamic diameter, that is the size of a unit-density sphere with the same aerodynamic characteristics. Particles are sampled and described on the basis of their aerodynamic diameter, usually called simply the particle size [15]. In particular,

- Particulate matter PM_{10} takes its name from the size of its diameter, which is about 10 micrometers or smaller
- Particulate matter $PM_{2.5}$'s diameter is smaller than PM_{10}'s, with a length of 2.5 micrometers or less.

Particulate Matter is not the only pollutant in the air, but it is the main one at least in Italy. According to Organization for Economic Cooperation and Development (OECD), whose emission data are based upon the best available engineering estimates, the podium of Italian pollutants is held by Particulate Matter $PM_{2.5}$, Particulate Matter PM_{10} and Nitrogen Oxides NO_X respectively. In general, considering the emissions during the period of the last 30 years, each pollutant has a descending trend, except for the two Particulate Matter particles as shown in figure 1.3 on page 7.

1.2.2 Air Quality Index

One of the duties of every government is to provide air quality information to the public, but usually this takes the shape of reports, environment reviews and targeted analyses: it is definitely not so common to have the appropriate background for reading them and extracting the relevant information, which is the reason why usually their audience is very limited.

Thus, a more practical tool has been developed to communicate the health risk of pollutant concentrations using Air Quality Index (AQI). The AQI is a color coded tool for telling the public how clean or polluted the air is. AQI delivers a number between 0 and 500 used by government agencies to communicate to the public how polluted the air is. The greater the AQI, the poorer the air quality. Cities and states use the AQI for reporting and forecasting air quality [10].

Each Country has its own air quality index, corresponding to their quality standards. In Italy every Region decides its own quality index for pollution emissions. Considering the scope of this thesis and the decision to monitor only the concentration of the Particulate Matter pollutants, it is very difficult to adhere to this wide range of standards: thus, the realization of this project will use the Air Quality Index defined by the United States Environmental Protection Agency (US EPA), which devised a piece-wise linear function for individual pollutant concentrations:

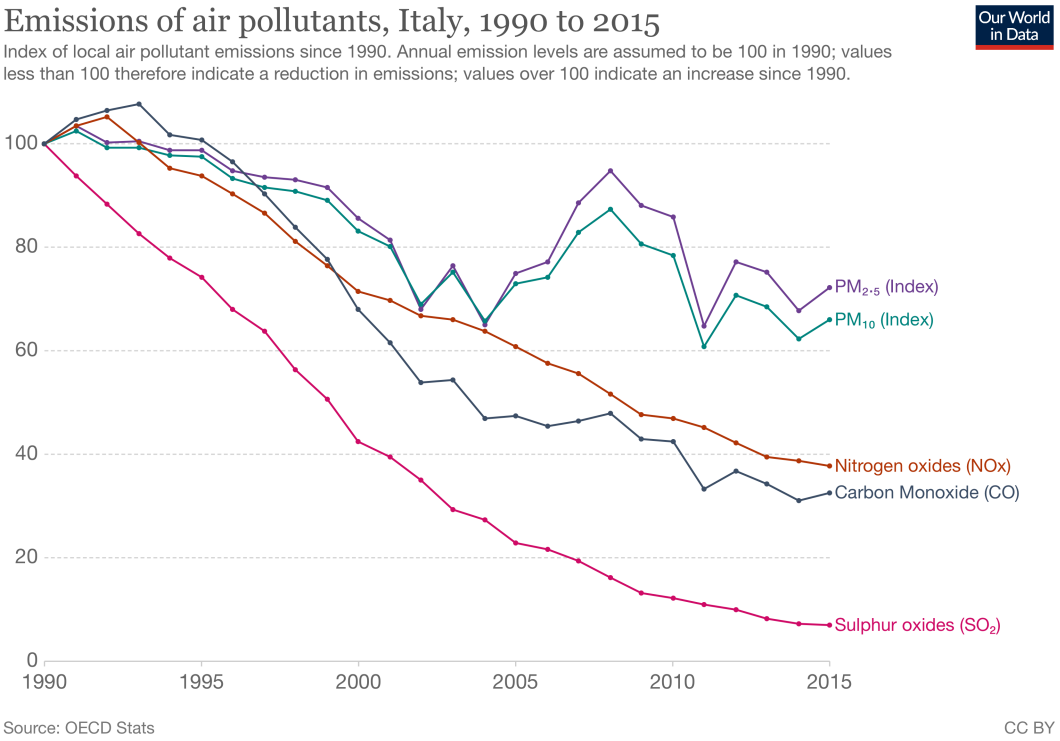

Figure 1.3: Emissions of air pollutants, Italy, 1990-2015 [20]

$$I = \frac{I_{high} - I_{low}}{C_{high} - C_{low}}(C - C_{low}) + I_{low} \tag{1.1}$$

where

- I = the Air Quality Index
- C = the pollutant concentration
- C_{low} = the concentration breakpoint that is $\leq C$
- C_{high} = the concentration breakpoint that is $\geq C$
- I_{low} = the index breakpoint corresponding to C_{low}
- I_{high} = the index breakpoint corresponding to C_{high}

It is perhaps relevant to identify that there is no existing international body to identify and assign these standards to each country and they are formulated by every country's government agencies independently. While the AQI standards vary, the formula to tabulate these indices also varies [10].

1.2.3 Criteria for monitoring

The concentration of particulates in the atmosphere can be a remarkably dynamic measure, and it requires tools to monitor its changes as rapidly as possible. On windy days, it is likely to have strong winds to bring $PM_{2.5}$ AQI values from 200 to 50 in less than one hour: given this situation, it would be annoying to wait for 24 hours before becoming aware that it is safe to have a walk outside. On the other hand, in case of wildfires, air quality can suddenly get worse in less than one hour: making people aware of the situation could be a vital activity in this case.

In order to give higher priority to the most recent measures in the monitoring process when reporting air quality data, the United States Environmental Protection Agency (US EPA) developed a new algorithm called PM NowCast. The original algorithm, known as the Conroy method, was developed in 2003 to make real-time air quality measurements roughly comparable to established regulatory air quality health thresholds (e.g. 24-hour $PM_{2.5}$ standards); however, that method was shown to be slow to respond to rapidly changing air quality conditions [7]. As a consequence, in 2013 EPA developed a new algorithm, known as the Reff method, which has the capability of being more responsive to sudden changes in air quality conditions.

1.2.4 Reff Method

The Reff method is an air quality data analysis algorithm capable of emphasizing recent measurements when values are unstable and reflecting a longer-term average when values are stable. In order to meet these characteristics, on one side Adam Reff proposed to weigh all values evenly when air quality is less variable, but on the other hand he decided to approach an average of the most recent three hours when air quality variation is considerable.

Reff's method relies on the hourly averages from the prior 12 clock hours, unlike Conroy's one which is based on the latest 24 hours. The algorithm assumes that at least two of the most recent three hourly averages are available. The following steps result in the NowCast concentration according to Reff method:

1. Given c_1,c_2, c_3, ..., c_{12} as the hourly averages from the prior 12 clock hours, compute the weight factor w^* as

$$w^* = 1 - \frac{c_{max} - c_{min}}{c_{max}} = \frac{c_{min}}{c_{max}} \tag{1.2}$$

   where

   - c_{max} = the highest value in c_1,c_2, c_3, ..., c_{12}
   - c_{min} = the lowest value in c_1,c_2, c_3, ..., c_{12}

2. Compute the minimum weight factor w as

$$w = \begin{cases} w^* & \text{if } w^* > \frac{1}{2} \\ \frac{1}{2} & \text{if } w^* \leq \frac{1}{2} \end{cases} \tag{1.3}$$

3. Multiply each hourly concentration by the weight factor raised to the power of how many hours ago the concentration was measured (for the current hour, the factor is raised to the zero power)

4. Compute the NowCast by summing these products and dividing by the sum of the weight factors raised to the power of how many hours ago the concentration was measured.

In general, since both $PM_{2.5}$ and PM_{10} use 12 hours of data, it is possible to represent the NowCast equation as follows:

$$NowCast = \frac{w^0c_1 + w^1c_2 + w^2c_3 + w^3c_4 + w^4c_5 + w^5c_6 + w^6c_7 + w^7c_8 + w^8c_9 + w^{10}c_{11} + w^{11}c_{12}}{w^0 + w^1 + w^2 + w^3 + w^4 + w^5 + w^6 + w^7 + w^8 + w^9 + w^{10} + w^{11}} \tag{1.4}$$

and in the extreme case of $w = 1$ the equation can be reduced to

$$NowCast = \frac{\sum_{i=1}^{12} c_i}{12} \tag{1.5}$$

which is just a simple 12-hour arithmetic average [7].

This algorithm is still accurate in case some hourly averages values are not available: for example, let us suppose that only the first three and last three hours of a 12-hour period are available, in this case the form of the previous equation would be

$$NowCast = \frac{w^0c_1 + w^1c_2 + w^2c_3 + [\text{ignored missing data}] + w^9c_{10} + w^{10}c_{11} + w^{11}c_{12}}{w^0 + w^1 + w^2 + [\text{ignored missing data}] + w^9 + w^{10} + w^{11}} \tag{1.6}$$

The figure 1.4 on page 10 is showing how Reff's method, unlike Conroy's one, is able to respond to rapid changes in air quality after a fireworks event: indeed, considering just the 3 hours after the explosion of the fireworks, the concentration of $PM_{2.5}$ (in the right-hand side of the figure) computed using Reff's algorithm (solid line) is much more similar to the actual values (dashed line) than the one computed using Conroy's algorithm (in the left-hand side).

High wind events can cause a spike to reveal in any moment of the day: this is what is represented in the figure 1.5 on page 10, where a steep increase in the concentration of $PM_{2.5}$ is followed by a rapid decrease. As shown in the figure, Reff's method is able to follow this trend when computing the NowCast concentration and thus provides a more reliable measure of current air pollution levels; on the contrary, Conroy's method reacts very slowly to the change because its result is evenly influenced by all the previous values. Using Reff's method strongly improves the credibility of the Air Quality Index because it is much closer to what people see and experiment outdoor.

1.3 Participative system for monitoring air quality

The aim of this project is to monitor air quality conditions by collecting data in the most pervasive way throughout a general city having any size. In Italy, most of the information

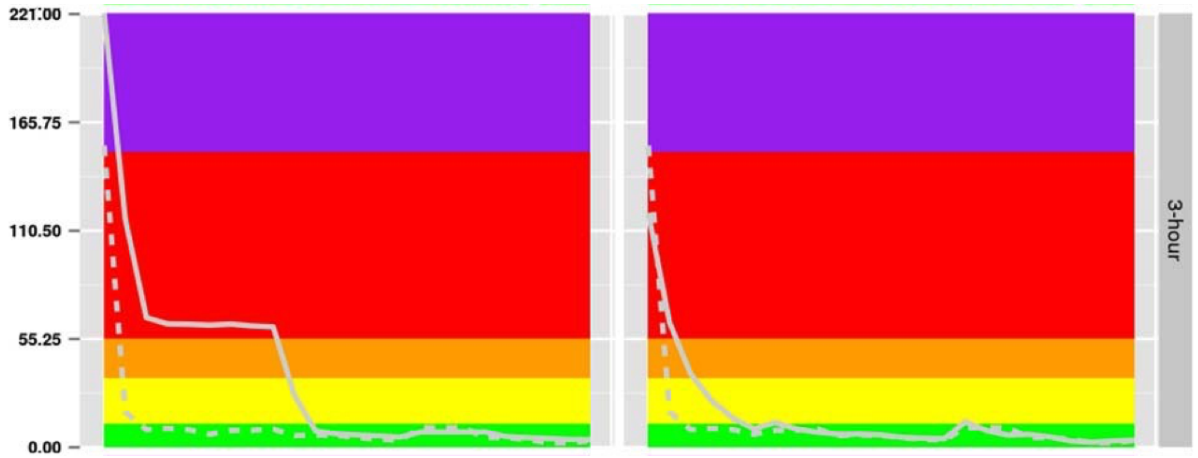

Figure 1.4: Conroy vs. Reff methods comparison after a fireworks event [1]

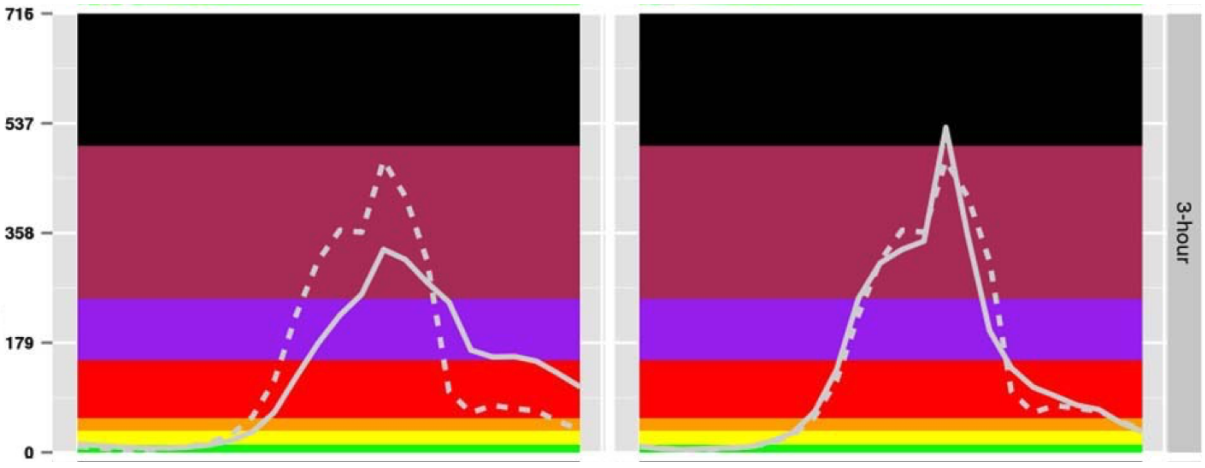

Figure 1.5: Conroy vs. Reff methods comparison during a high wind event [1]

about air pollution is collected by Regional entities thanks to sophisticated technological equipment, which is usually located outside of the chief town because of its dimension. A different approach to this challenge is to monitor the air pollution directly inside the city, where people are effectively spending their time outdoor and citizens breathe while walking, running or waiting for the bus. This aspect enriches the value of the data being collected and can be further used to take the best corrective actions: in fact, let's consider a densely populated district where every day lots of cars and trucks pass through, if the system reveals that the air quality is getting worse, the local government might decide to take corrective actions just for this district in order to preserve the health of citizens living around.

A remarkable concept of this project is the portability and the scalability of the sensors: indeed, the sensors need to be very tiny, and they shouldn't be connected to the mobile network due to the considerable total cost of the service for all the sensors.

The main disadvantages of this approach are the following:

- since each sensor needs to be very small, the precision of the measures being sampled decreases with respect to the ones sampled by the Regional entities
- given that it is preferred to have an intermediary dynamic entity to carry out the transmission of the measures towards a server, there is no certainty that this entity will frequently come across the sensors

The main advantages of this approach are the following:

- the proximity of the sensors to the environment where citizens live around increases the value of the data being collected
- local governments can take targeted corrective actions
- the cost of the implementation is reduced because no mobile network subscription is needed

The advantages of this innovative possibility of monitoring air quality are worth the realization of this project, and there might be different solutions to reduce the downside of the disadvantages.

The system will be composed of three main elements:

- monitoring devices capable of sampling, storing and transmitting data
- Android and iOS application capable of acting as a gateway between the monitoring devices and a remote server (detecting monitoring devices, receiving data and sending it to a remote server) and capable of properly showing air quality levels around the user
- remote server capable of gathering data sent by the mobile application, storing it inside a database and providing web API to be accessed when requesting data

1.3.1 State of the art

At the beginning of this thesis the research group from Polytechnic of Turin had already taken some architectural and technological decisions.

The monitoring device used for sampling and storing the measures consisted of a set of sensors for measuring the quantity of PM_{10} and $PM_{2.5}$ particles together with relative humidity and temperature, a breadboard and a Raspberry Pi 3 running Arch Linux.

A basic version of an Android mobile application had been developed by a previous thesis student: this application was able to trigger the Bluetooth connection to the Raspberry thanks to Beacon technology, and acted like a stateless client requesting data to the server and showing its response without any business logic.

The remote server was developed by the research group using the Flask framework with Python programming language. This remote server was able to receive requests on well-defined web APIs for managing user authentication and sending sets of measures according to specific parameters in the request.

1.3.2 Realization

The contribute of this thesis to the project is the complete redesign of the mobile application and of the interaction with the monitoring device. In particular, given the participative nature of the system, iOS users couldn't be excluded from the inclusive goal of the project; thus, this thesis will describe the design and the development of a new mobile application for both Android and iOS operating systems, created by exploiting the innovative and emerging Flutter framework. In addition, since the research group decided to substitute the Raspberry Pi with a microcontroller-equipped expansion board which can be programmed using the MicroPython programming language, a new way of interaction based on Bluetooth Low Energy technology has been designed.

1.4 Thesis organization

The starting point of the thesis is to explain the global picture of the whole system: a general overview of each element of the system will act as a background for the topic of intercommunication between elements. After that, an extensive space will be dedicated to the Flutter framework and to cross-platform development in general. The architecture, the design and the business logic of the mobile application will be issued hereafter. Next topic of the thesis is a brief description of the microcontroller board and the development of the Bluetooth Low Energy connection on the microcontroller. The conclusion of the thesis highlights the results obtained and introduces possible future works on the project.

Chapter 2

Architecture of the system

2.1 Architecture

This thesis is part of a project that was born with the aim of providing a complementary way of monitoring air pollution within the city of Turin with respect to the current one. Traditionally in Italy official data about pollution levels are provided by local public entities thanks to professional tools strategically located alongside the main cities: in particular, these monitoring stations must be placed according to population's density, urban structure, orographical and meteorological characteristics and emission loads, as specified by Legislative Decree n.155/2010. Moreover, due to their capability of measuring pollution levels with such a high degree of precision, these tools are very expensive and extremely large: thus, it is not possible to spread them around the city for both economical and logistic reasons.

One of the worst effects caused by air pollution are health risks, which is the main problem that this project wants to address: it is evident that the architecture of the actual network of monitoring stations is ineffective for this purpose, because most of the citizens usually live and experience air pollution inside the city, where no traditional monitoring station can be located; in addition, these monitoring tools provide their data hourly, which means that as a matter of fact in case of rapidly changing conditions citizens would be notified about their exposure to pollution potentially after one hour.

A possible way to address this issue is based on the spread of many tiny monitoring devices around the city. Let's imagine that a citizen wants to do a short running session outdoor and that she wants to know whether it is a good moment for doing open-air sport activities: in case of a strong wind event, which is often responsible for decreasing air pollution levels, the citizen would be aware that air quality has recently improved. Though, reducing the size of the monitoring devices requires a trade-off: unlike the official traditional sensors, these devices are much less precise, but they are portable and capable of sampling every second. Moreover, thanks to the ease in changing their location, it is possible to realize both a static or a dynamic structure: indeed, each sensor may be located in a fixed place or it might be positioned on a moving environment such as the roof of a public bus. The problem of the precision of measurements will be handled in the section about data management.

Given that a huge number of monitoring devices could be spread around the city, as

a consequence the volume of data produced by these sensors would become considerable. In order to collect and process this information, all measurements should be sent to a remote server which should be responsible of storing data and elaborating meaningful information. This issue introduces the participative nature of this architecture: since connecting every monitoring device to the Internet would be extremely expensive, the citizen would act as a gateway between the monitoring device and the remote server throughout a mobile application.

2.1.1 Distributed system

The association between Internet of Things (IoT) and the world of distributed systems is usually devised as a concept in which the Internet expands into the real world, including ordinary things. The backbone of this concept is a geographically dispersed system that combines ubiquitous, mobile, and cloud computing technologies. This issue is the result of combining IoT and distributed computing in a synergistic way.

Distributed computing entails the employment of a collection of connected computers to achieve a single computational task. A large number of distributed computing technologies, combined with hardware virtualization, have resulted in Cloud computing; in addition, the execution of complex applications on Cloud facilities with a particular focus on energy efficiency is a popular field of study.

IoT system designs connect networked devices to apps and services, enabling for large-scale data contributions regarding real-world events; things exploit their processing, sensing and communication capabilities in order to interact with the physical and virtual worlds via a variety of communication methods. At its core, the Internet of Things means just an environment that gathers information from multiple devices (computers, vehicles, smartphones, traffic lights, and almost anything with a sensor) and applications (anything from a social media app like Twitter to an e-commerce platform, from a manufacturing system to a traffic control system) [22].

When trying to scale IoT services, a set of remarkable issues must be taken into consideration; in general, most of these problems derive from the current conditions of Internet connection, which is intermittently affected by connectivity issues occasionally. The aspects that shape the architecture of an IoT system are the following ones:

- *Scalability* refers to the number of sensors and actuators linked to the system, the networks that connect them, the volume of data associated with the system and its speed of movement, as well as the necessary computing power

- The analysis of massive amounts of data is required by many advanced IoT devices; for instance, *identifying patterns* from previous data might be very useful to guide future activities. Large quantities of processing may be required to extract valuable information from complicated data; nevertheless, an IoT system's capability to mine existing data for new insights is a feature likely to be part of the system

- *Cloud computing* platforms are commonly used in IoT systems. Cloud computing systems allow huge quantities of resources to be used, both in terms of data storage and the ability to bring flexible and scalable processing resources to data analysis

- IoT systems frequently operate in *real time*: data about ongoing events is constantly streamed in, and rapid reactions to that stream of events may be required; this may entail stream processing, which involves responding on event data as it arrives, comparing it to prior events as well as static data in order to react appropriately
- IoT systems may cover whole buildings, entire cities, and even the entire world: data can be kept at the network's edge or centralized, but it can also be *widely distributed*; processing can also be distributed: some processing is done centrally (in cloud services) or at the network's edge, in IoT gateways or even within sensors
- The *security* and credibility of distributed IoT systems is a difficult challenge with solutions that must expand and change along with the systems: data protection is essential, and there are serious privacy issues about data that relates to people.

Today, common IoT system architectures are based on hierarchical organizations of system components into layers [27], as illustrated in figure 2.1 on page 16. Applications at the top layer rely on cloud computing infrastructures that provide virtually unlimited data storage and data analysis capabilities with global connectivity [27]. In the middle, a network layer contains middleware and network infrastructure components to handle bi-directional communications, i.e. data traffic to the cloud and application-specific system control to the lower layers [27]. At this layer, the components typically have computational power that can be harnessed for example for edge computing; hence the layer is also called edge layer [27]. The lowest layer is a device layer, where the data producing things, i.e. low-power resource-constrained embedded devices and smartphones, operate [27]. Though, with large-scale data transfer to the infrastructure side, operational latencies and bandwidth needs for the core network rise [27]. Data upload is resource-intensive for battery-operated and mobile devices, let alone when connections are sluggish and inconsistent. Devices at the device layer join and exit the systems at whim, making IoT systems fundamentally dynamic [27].

Edge computing proposes a partial solution to these challenges [27]. Application-specific cloud resources are leveraged to the resource-rich infrastructure devices at the logical edges of the IoT networks [27]. Existing edge computing solutions rely on virtual machines (VM) to bring tailored application resources to the edge through code mobility. VM-based applications are then executed in close proximity to end users and devices, generally improving quality of service and providing high bandwidth and low latency [27]. The figure 2.1 on page 16 shows this VM-based edge architecture. This vertical approach including the edge layer is beneficial for separation of concerns between layers and for abstracting heterogeneous communication technologies and device capabilities [27]. Any resource-rich device between the cloud and data sources can be considered as an edge device: typically, edge devices include access points, routers, gateways, cellular base stations and VM-based data centers [27].

Many edge computing solutions have been developed throughout time, but this section focuses only on one of them, called *Fog Computing*, because it is the one that most matches the architecture of the thesis project. Fog computing provides a virtualized location-aware distributed edge computing platform with content storage and networking services in between end devices and cloud [27]. A fog platform consists of dense deployment of fog nodes, i.e. edge devices with a LAN connection, in a single-hop distance from

the end devices [27]. Challenges in *fog computing* include dynamic network access and fog node utilization due to end device mobility [27]. Stationary end devices offload their data processing to the nearest fog node, but for mobile devices the VMs are required to migrate between fog nodes [27]. Other challenges include application-aware resource discovery, provisioning and planning VM migration while also trying to predict user movement and minimize latencies [27]. Maintaining quality-of-service depends on the reliability of network connections and capacity with different workloads [27]. Lastly, security and privacy are general concerns in the distributed remote execution of applications [27].

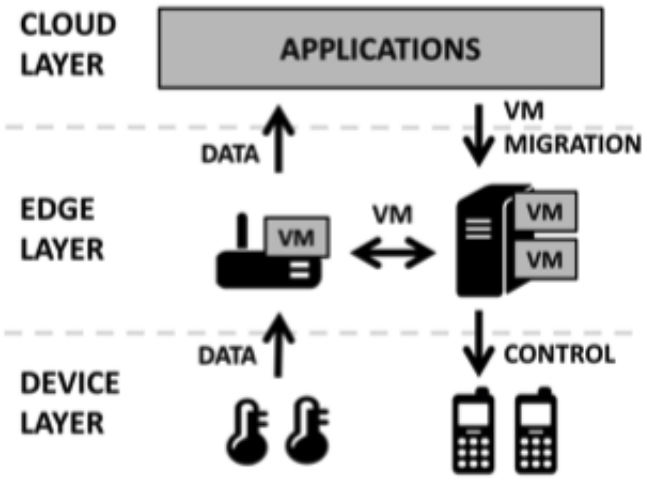

Figure 2.1: VM-based edge computing architecture [27]

The integration of mobile agents as a Multi-Agent System (MAS) into IoT edge computing platforms enables reactive and adaptive edge application execution: application-specific tasks autonomously move with regard to the local resource availability in the layered architecture horizontally and vertically [27]. With mobile agents, it is possible to inject application-specific tasks, such as data processing at the data source, into the system at runtime [27]. As software agents, mobile agents' operations are autonomous and asynchronous and they have the capabilities to observe their environment, react to changes dynamically and adapt their behavior to the changes [27]. Once injected into the system, there is no need for continuous connection between an owner and the agent [27]. With these capabilities, mobility increases robustness and fault tolerance in distributed application execution [27]. As a matter of fact, mobile agents implement a *fog computing* platform by bringing agent-based awareness, interactions, adaptivity and proactivity [27]. The figure 2.2 on page 17 shows this mobile-agent-based edge architecture.

This thesis project's architecture falls into the category of MAS mobile-agent-based edge architecture. Before analyzing the similarities, let's introduce the main differences. In this case, the Cloud computing platform is substituted by a centralized remote server running a web application based on a DBMS to collect air pollution data; in addition, the edge layer doesn't use fog nodes in the form of VMs and thus doesn't need to deal with the problem of migration, instead it consists of smartphones acting as a gateway between the end devices (the sensors) and the remote server. Of course, the role of mobile-agents is played by the users' smartphones running the mobile application, which interacts with

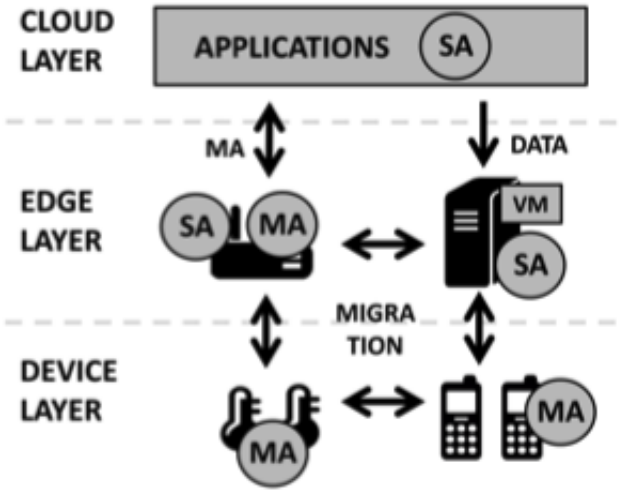

Figure 2.2: Mobile-agent-based edge computing architecture [27]

the monitoring devices to gather and send their measurements as soon as possible to the upper layer, which is the remote server. In reality, given the potentially wide geographic area to be covered (many cities in Italy but even foreign Countries), it is possible to consider the presence of several servers acting as proxy data-centers close to the edge devices (e.g. the smartphones running the mobile application) and offload future complex tasks - like machine learning pattern discovering - to the upper Cloud layer. Indeed, this schema would avoid problems related to centralized architectures such as being a single point of contact - which causes access time issues - and being a single point of failure. Moreover, in order to guarantee that an appropriate summary of the latest measurements is always transmitted to the centralized remote server, each sensor is capable of directly and periodically communicating with the centralized server, even though the amount of data that can be sent is very small due to the underlying technology.

2.1.2 Data communication

The technology used for putting in direct communication each monitoring device with the remote server is called LoRa (which stands for Long Range): as WiFi technology is capable of supporting high throughput but can't perform long-range communication and doesn't take care that much about power consumption, LoRa is the solution for effectively expanding the transmission range while maintaining acceptable throughput, latency and battery consumption. Indeed, LoRa's communication range can reach 600 m with a periodic sending interval of 5 minutes; according to some reasonable electronic configuration, the maximum payload is 51 bytes, but after analyzing the scenario the research group decided to reduce the message size to 20 bytes in order to reach an interval time of less than 3 minutes with the same communication range [8].

The technology used for the communication between the monitoring devices and the users' smartphones is Bluetooth Low Energy (BLE). As explained in the section 2.3.3, BLE is a technology that successfully operates with lower power consumption with respect to classic Bluetooth and it is one of the best technology choices for Internet of Things

scenarios ranging from fitness to health care and entertainment. This technology reduces costs of maintenance, ensures longer battery life and is easy to implement.

2.2 Server

The server, which has been developed by the research group, is a web server running a web application based on the REST paradigm, which is explained in details in section 2.2.1. The main aspect of any web server is the protocol used to exchange information: nowadays the most popular one is the HTTP protocol.

The HTTP protocol is one of the protocols belonging to the application layer in the OSI model, and it is mainly used for creating distributed information systems; in particular, this protocol is able to transport any kind of data from a client to a server and vice-versa. It's quite a sophisticated protocol because it allows to describe the content being requested or sent, to perform conditional requests, to compress information, to use caching policies and to use authentication mechanisms. The typical scenario involving the HTTP protocol requires two actors, a client and a server, interacting with each other; since this protocol doesn't preserve a state, every request coming from the client is independent from the previous ones: despite this is a good news for the transportation layer and in particular for the TCP protocol whose header usually wraps the HTTP packet, it might be an obstacle from the perspective of a distributed application. Nevertheless next sections will explain how to overcome this issue. HTTP is independent from the content, hence request and response can encapsulate any kind of data - text, audio, video, etc.

The touch point between client and server is a Uniform Resource Locator (URL), that is a string following the pattern $schema : //hostname[: port]/path$; in particular, *schema* indicates the protocol used within the interaction, *hostname* and *port* is the address where the server is listening for new connections, and *path* is the relative position of a certain resource inside the server (it doesn't need to be a real path). This mechanism is fundamental because it allows to configure several paths, each responsible for managing a specific information.

Understanding HTTP headers and their meaning is the crucial aspect when creating a web application. Indeed, by using headers the HTTP protocol offers a negotiation mechanism: any HTTP request contains an Action to be performed, the URL indicating a resource as object of the action to be carried out, and the protocol version number used by the client; as a matter of fact, also other items are reported in the HTTP header, but for the time being let's consider only these ones. The list of the possible Actions is the following:

- GET, for requesting the sending of a specific resource indicated by the URL; in case of conditional request, the resource gets transferred provided that a certain characteristic of the header (e.g. if-modified-since, if-match) has a meaningful value
- HEAD, similar to GET, but it requests just the sending of the headers that correspond to the resource referenced by the URL; it's typically used for checking purposes
- POST, for sending data, encapsulated in the body of the request, to the resource pointed by the URL; the result of the operation may be sent as response

- PUT, similar to POST, but it is generally used to update contents that already exist: moreover, unlike POST action, PUT is idempotent, which means that side effects of this action remain the same even if it is carried out several times
- PATCH, a reduced version of PUT used to update a part of already existing contents
- DELETE, for destroying the resource indicated by the URL; also, this action is idempotent too
- TRACE, for debugging the connection with counterpart; it sends back a copy of the sent message
- OPTIONS, for requesting the list of actions that can be carried out on a given URL together with the corresponding constraints
- CONNECT, for creating a TLS/SSL connection.

It is evident that these actions offer an entry point for the typical operations of a database system because they allow the client to create CRUD (Create, Retrieve, Update, Delete) requests.

For what concerns the response, it always contains state information consisting of the protocol version used by the server and most importantly a status code composed of 3 digits together with a textual description. The first digit of this code represents the category of the response; in particular,

- 1xx indicates that the message has been sent while the actual request is being processed
- 2xx indicates that the request has been accepted and understood by the server
- 3xx indicates that the client must undertake further actions to get the result (e.g. 301 Moved Permanently, 302 Moved Temporarily)
- 4xx indicates that the request has been rejected due to a client-side unrecoverable error (e.g. 404 Not Found, 403 Forbidden, 401 Unauthorized)
- 5xx indicates that the server is not able to process the request due to internal issues (e.g. 500 Generic Server Error).

The development of the server is based on a micro-framework, called Flask, written in Python. Usually Flask is not considered as a full Web development framework because it leverages a simple core even though it may be extended with third-party libraries: indeed, unlike more popular frameworks, by default Flask doesn't provide support for database abstraction layers, nor for form validation and many other features. Though, Flask supports basic web functionalities, such as:

- static routing, which allows Flask to trigger a particular function when a certain URL is queried

- dynamic routing, similar to the static one but it is possible to indicate a set of URLs by using a simple notation in order to trigger a function: this means that the content of the URL can be parametrized and its parameters can be used as argument of the function being triggered
- HTTP methods support, in order to trigger the function according to the HTTP method inside the request
- redirection, which routes the user's request towards another endpoint
- error handling, for aborting a request with an error status code
- APIs with JSON, which is fundamental for implementing a RESTful web application, leveraging Python's *dict* object type for creating a dictionary and function support for JSON serialization.

Moreover, the server is able to organize the web application into smaller and re-usable applications thanks to Flask Blueprints: indeed, when an application gets larger, it is strategic to factor it into a set of sub-applications which still share the same application configuration and are able to change an application-level object provided that they have been properly registered; though, Blueprints cannot be unregistered once the application is running, so in order to delete them the whole application must be destroyed.

One of the main extensions used by the server is the Flask-RESTful library: this extension is very useful for quickly developing REST APIs, and in particular combines the advantages of the REST paradigm with the ease of writing code in Python. Also, this library allows Flask to smoothly manage the HTTP headers of both the request and the response.

2.2.1 REST API

There was no standard for how to develop or utilize an API before 2000. Its integration necessitated the use of protocols like SOAP, which were infamously difficult to create, operate, and debug. The actual potential of Web APIs was understood in 2000, when a group of specialists established REST and transformed the API landscape. The stated goal was to establish a standard that would allow two servers to interact and share data from any location; as a result, they created REST, a resource-oriented architecture which consists of a set of concepts, characteristics, and restrictions.

The REST architecture looks at considerable issues such as guaranteeing interoperability between platforms and consuming contents capable to be indexed, linked between each other, alternatively represented, scaled according to volumes and workloads, and easy to develop and to maintain. Therefore, all distributed systems exploiting REST architecture should support heterogeneity, that is the ability of connecting clients and servers of any dimension even if they run different operating systems or code written in different programming languages. Moreover, scalability is a cornerstone of REST architectures: indeed, any service level must be guaranteed to be working regardless of the information volume and the requests rate over time; further, the system should be facilitated in the development of new functionalities without conflicting the existing ones (concept of

evolvability); then, it should be reliable, meaning that despite local malfunction it must not stop working but it should just reduce service levels; finally, the distributed system must offer good performance, otherwise the services being delivered become useless. For sure REST - as well as other architectures - is not able to completely guarantee all these properties, thus a compromise must be found.

Before analyzing how REST works, let's introduce the architecture that was leading online data communication before 2000, that is SOAP. The basic idea behind SOAP is that both request and response are carried inside a XML document: indeed, this document had to report a description of the content following a precise schema, which is in contrast to the the concept of evolvability that was previously mentioned; in addition, since there was no prior mechanism to distinguish read requests from write requests because every message was encapsulated inside a POST request, all caching benefits were ignored; finally, XML encoding was so verbose that even for few information it required several KBytes.

REST architecture has several cornerstones:

- stateless client-server architecture for guaranteeing simple and stable implementations, so that server doesn't have to store sessions; nevertheless, when there is the need of preserving a state in the communication, it will be client's responsibility
- a server hosts one or more services
- each service handles a set of resources, each of which has its own globally unique Uniform Resource Identifier (URI) - often referenced to as a URL -, potentially several representations such as plain text, XML and JSON and offers support for CRUD operations
- the resources associated to a specific URI can reference single items, a collection of items or results of computations; for example,
  - http://my.api.com/app/students lists all the students in the database
  - http://my.api.com/app/students/s14 describes the student having ID equal to 14
  - http://my.api.com/app/students/s14/courses lists all courses attended by the student having ID equal to 14
  - http://my.api.com/app/students/s14/courses/c29 describes the course having ID 29 attended by the student having ID equal to 14

One of the most important HTTP headers is called *Accept*: this header is used by clients to negotiate the encoding of the response, and it is represented as a couple of strings separated by ':' where the first one is the MIME type and the second one is a score (between 0 and 1) attributed to the choice of that MIME type (e.g. "text/plain:0.9" indicates that a plain text response will be well received). Alternatively, if the server offers this support, the client can use the convention of ending the URL with the encoding type (e.g. http://my.api.com/app/students.json).

In 2008 Leonard Richardson, one of the most influential figures in the field of distributed web applications, proposed his so-called REST Maturity Model: this interesting

theory classifies Web APIs according to the observance of the rules explained in the 4 levels of his Model.

The model's lowest level (Level 0) defines a Web API with a single URI (usually POST via HTTP) that accepts all of the service's actions: clearly, this type of resource can't be well-defined. These Web APIs commonly use Remote Procedure Calls (RPC) Plain-Old XML objects as well as numerous SOAP services. For example, the URI "/enrollStudentService" requested via HTTP POST may retrieve all students enrolled to a given course or it may enroll a student to a course or it may exclude a student from a given course.

Instead of providing a single universal endpoint, in Richardson's Level 1 resources are introduced, allowing requests to distinct URIs (still all generally POST) for separate operations: although the API resources are still generic, the scope of each one may be determined. The Level 1 architecture is not RESTful, but it does organize the API in a RESTful manner.

For what concerns Level 2, HTTP verbs are first used by the system: this allows the resource to be further specialized, and therefore the functionality of each particular activity with the service to be narrowed; at this level, the basic separation consists of dividing a given resource into two requests: one for simply receiving data (GET), and the other one for changing data (POST), but it is also possible to fine-tune the granularity. For example, the URI "/courses/c33/enrollOne" requested via HTTP POST enrolls a given student to the course having ID equal to 33, while the URI "/courses/c33/students" requested via HTTP GET sends back all students enrolled to the course having ID equal to 33.

The hypermedia representation is introduced in the last level (Level 3): these hypermedia links are components included in resource's response messages that allow to build a relationship between different data items returned from and passed to APIs, often known as HATEOAS (Hypermedia As The Engine of Application State). For example, the URI "/courses/c33/students" requested via HTTP GET sends back the same content as the one in Level 2, but for each student it may provide several hyperlinks containing the URI relative to that student and the URI for excluding the student from the course; it is remarkable that this Level offers the possibility to dynamically discover new services: hence, server-side code will be able to evolve while keeping self-documented and the new era of micro-services will flawlessly start.

2.2.2 Data management

The monitoring devices are set to sample once every second, and thus a huge amount of data is produced even in a limited amount of time such as one day; besides, in order to detect particular dynamic phenomena and to prevent malfunctions from compromising system's operability, every monitoring devices is equipped with 4 different PM sensors, leading to a considerable increase in the volume of information. Moreover, the research group has also collected public weather data to study the relationship between the air quality as assessed by the sensor stations and other weather information. Managing such a huge amount of data wouldn't be possible without a database management system, so the research group decided to make use of MySQL in order to store information coming from different sources according to a well-defined relational schema.

From the software point of view, the database grows very fast and it requires a powerful computer to be managed efficiently. In order to have a fast response to SQL queries, the research group used a Dell computer with 20 cores, 384 GB of internal memory, and some TB of hard-disk. Smaller computers would have potentially implied very slow query evaluation [3].

2.2.2.1 Data calibration

For what concerns data calibration, accurate and precise calibration models are particularly critical to the success of dense sensor networks deployed in urban areas of developed countries. In these situations, pollutant concentrations are often at the low end of the spectrum of global pollutant concentrations, and poor signal-to-noise ratio and cross-sensitivity may hamper the ability of the network to deliver reliable results [3]. In order to overcome this issue, some calibration techniques come to the rescue: in particular, the research group has studied several historically popular procedures and has finally decided to implement both the *Multivariate Linear Regression Model* (MLR Model) and the *Random Forest* machine learning algorithm.

In Multivariate Linear Regression Models, regression analysis is used to predict the value of one or more responses from a set of predictors. Let (x_1, x_2, ..., x_n) be a set of predictors (dependent variables) believed to be related to a response (independent) variable Y. The linear regression model for the j-th sample unit has the form

$$Y_j = \beta_0 + \beta_1 \cdot x_{j1} + \beta_2 \cdot x_{j2} + ... + \beta_r \cdot x_{jr} + \epsilon_j, \tag{2.1}$$

where ϵ_j is a random error, the β_i are unknown (and fixed) regression coefficients and β_0 is the intercept [3]. Given n independent observations, it is possible to write one model for each sample unit or rather to organize everything into vectors and matrices as:

$$Y = X \cdot \beta + \epsilon. \tag{2.2}$$

A Random Forest Model is a machine learning algorithm for solving regression or classification problems. It works by constructing an ensemble of decision trees using a training data set. The mean value from that ensemble of decision trees is then used to predict the value for new input data [3].

For calibrating purposes, all sensors were positioned near the stationary ARPA station in Turin for a limited amount of time: this station uses β-radiation technology to produce high accuracy measurements; in addition, ARPA offers hourly average data that has been utilized as a baseline for all sensor board data. The research group has observed that, even though Random Forest Model has the capability to create a non-linear regression model and has been shown to be superior to MLR in handling several particles' measurements, MLR outperforms Random Forest when it comes to analyzing the results.

2.3 Sensors board

In the current section each monitoring device is referenced to as sensors board. Each sensors board is a platform composed of several sensors, some of which are replicas of the same sensor for redundancy purposes; in particular, it contains:

- 4 Honeywell ® HPMA115S0-XXX PM sensors
- 1 DHT22 temperature sensor
- 1 DHT22 relative humidity sensor;

in addition, the sensors board is further equipped with an expansion board and a microcontroller (also referenced to as development board) in order to provide storing and communication features.

2.3.1 Expansion board

The expansion board is the Pycom Expansion Board 3 showed in figure 2.3. This expansion board is designed to be compatible with all Pycom modules, including the one used as development board. The board is both USB and LiPo battery powered, features a MicroSD card slot, some LEDs for stating charge status, lots of jumpers to enable or disable features and a handy button (unlike the previous version) to enter "safe mode" easily. Pycom has developed a very useful plugin, called Pymakr Plugin, for popular code editors like Visual Studio Code and Atom, to let developers write, download and upload code in the simplest possible way; also, an interactive shell is accessible through telnet or via serial port for fast debugging. This expansion board is considerably light and tiny: it weighs only 18g and its size is just 65 x 50 x 8 mm.

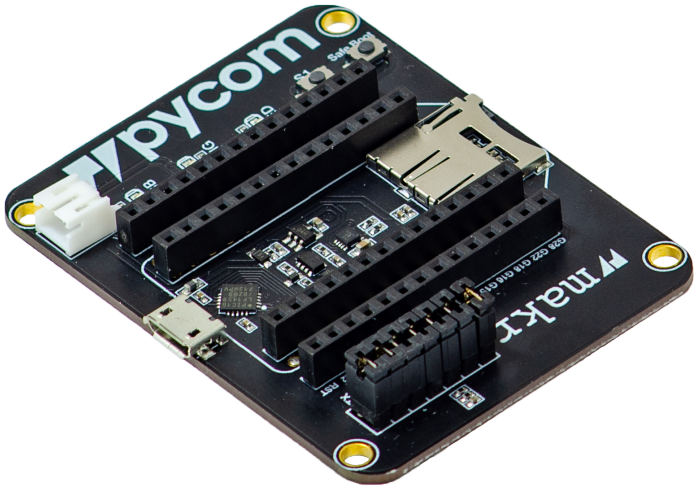

Figure 2.3: Pycom Expansion Board 3 [23]

2.3.2 Development board: FiPy

The development board is the Pycom FiPy Development Board. This board can be programmed using the MicroPyhton programming language, which is a subset of classic Python programming language. The board includes many IoT-ready components which make it one of the most interesting IoT platforms actually on the market: indeed, the

FiPy supports five different networks (WiFi, Bluetooth Low Energy, cellular LTE, LoRa and Sigfox), has a powerful ESP32 microcontroller, fits in a standard breadboard and even more so in one of Pycom's expansion boards, and features ultra-low power usage.

For what concerns computational capabilities, the FiPy has a Xtensa ® dual-core 32-bit LX6 microprocessor, hardware floating point acceleration, python multi-threading and an extra co-processor responsible for monitoring GPIOs and controlling most of the internal peripherals during sleep mode with low energy consumption. Memory support includes a 520 KB + 4 MB RAM space and an external 8 MB flash memory. Also security support is provided, because the FiPy supports both hash and encryption algorithms such as SHA, MD5, DES and AES.

Also this device is remarkably light and small: it weighs 7g and its size is only 55 x 20 x 3.5 mm.

Figure 2.4: Pycom FiPy MicroPyhton-programmable Development Board [24]

2.3.3 Bluetooth networking

Ericsson Mobile Communications set out in 1994 to discover an alternative to using wires for communication between mobile phones and other devices, and the Bluetooth technology was born. The Bluetooth Special Interest Group (SIG) was founded in 1998 by Ericsson, IBM, Nokia and Toshiba, and the first version was released in 1999. The very first public version, referenced to as the version 1.2 standard, had a throughput of 1 Mbps; the second one was capable of sending data with a data rate speed of 3 Mbps; the

third version reached the data rate of 24 Mbps. The fourth version is the one introducing Bluetooth Low Energy (BLE), but before analyzing BLE technology let's have a look at how Bluetooth technology works.

Bluetooth technology is a short-range wireless communications technology that allows a person to listen to music using a wireless headset, utilize a wireless keyboard, and synchronize information from a mobile phone to a computer all while utilizing the same core system. The Bluetooth RF transceiver (or physical layer) operates in the unlicensed ISM band - a portion of the radio spectrum that has been reserved for Industrial, Scientific and Medical (ISM) intents at international level - centered at 2.4 GHz (the same range of frequencies used by microwaves and WiFi). The core system employs a frequency-hopping transceiver to combat interference and fading [2].

An RF architecture known as a *star topology* is used to manage Bluetooth devices. A piconet is a set of devices that have been synchronized in this way; it can have one master and up to seven active slaves, with extra slaves that are not actively participating in the network - a device can also be a master or a slave in one or more piconets. The physical radio channel in a piconet is shared by a set of devices synced to both a common clock and a frequency-hopping pattern, with the master device supplying synchronization references: indeed, this pattern is algorithmically determined by the master device and it basically is a pseudo-random ordering of the 79 frequencies in the ISM band; in particular, the hopping pattern may be adapted to exclude a portion of the frequencies that are used by interfering devices. When static (non-hopping) ISM systems, such as WiFi networks, are in close proximity to a piconet, the adaptive hopping approach improves Bluetooth technology's coexistence.

Bluetooth technology uses the principles of device "inquiry" and "inquiry scan": scanning devices listen in on known frequencies for devices that are actively inquiring. When an inquiry is received, the scanning device sends a response with the information needed for the inquiring device to determine and display the nature of the device that has recognized its signal [2]. In general, the maximum connection range of most Bluetooth devices is around 9 meters, and that distance is decreased when barriers (such as a wall) are present.

The difference between Bluetooth and Bluetooth Low Energy revolves around the concept of power consumption. Bluetooth was created with continuous, streaming data uses in mind: indeed, it allows to send and receive a large amount of data over a short distance. Applications running Bluetooth LE can run on a tiny battery for even four years; despite this is inconvenient when conversing over the phone because of the reduced data rate (1 Mbps), it is necessary for apps that only need to transmit modest quantities of data on a regular basis. BLE works in the same 2.4 GHz ISM band as Bluetooth, but unlike traditional Bluetooth BLE is always in sleep mode unless a connection is established; in addition, unlike Bluetooth, which takes around 100 milliseconds to connect, the real connection times are only a few milliseconds with Bluetooth LE. For what concerns the power consumption, classic Bluetooth has at least twice the power consumption as the one of Bluetooth LE: hence, this is the reason why BLE applications can run for longer periods of time.

2.4 Mobile application

2.4.1 Bluetooth connection

In Bluetooth LE terms, the electronic device initiating request commands and accepting the corresponding response is called client (or central) device, while the device that receives requests and sends back the responses is called server (or peripheral) device; client devices are usually those ones that have greater processing capabilities (like smartphones, computers, etc.) and are responsible for controlling server devices, while server devices usually act as sensors collecting data and transmitting it to client devices without processing it.

In order to connect two devices via BLE, first they need to pair with each other. The pairing process consists of two or three phases (the last one is optional and is used for bonding, a process that enables devices to remember and trust each other when reconnecting later on), during which devices exchange data for authentication purposes, as shown in figure 2.5 on page 28. Devices communicate basic information about their capabilities during the first phase of pairing in order to figure out how to continue with their connection; in particular, devices identify themselves and describe what they are (a smartwatch, a mouse, a headset, etc.) and what they can do; of course, this communication is not encrypted because it doesn't need to. The second pairing phase focuses on key generation and exchange; at this point, two types of BLE connections can happen and BLE module programmers should be able to choose which one to adopt:

- Legacy connections, available for BLE versions 4.0, 4.1 and 4.2, involve the following procedure: a Temporary Key (TK) is sent between devices and is used to produce a Short Term Key (STK), which is subsequently used to authorize the connection; evidently, BLE Legacy connections are unsafe by default, but with the right pairing procedure (e.g. using ECDH cryptography and a pair of public and private keys) they can be secured

- Secure connections, available on BLE version 4.2 but incompatible with previous BLE versions, use the Elliptic-Curve Diffie–Hellman (ECDH) method to generate keys and add a more complicated key authentication procedure; thus, this mechanism protects the device against passive eavesdropping by default.

Finally, devices produce sets of keys to authenticate each other's identities for future connections during the optional third step of pairing; for example, they may be a pair of Connection Signature Resolving Keys (CSRK) for data signing or Identity Resolving Keys (IRK) for generating secret MAC addresses and lookups.

Passive eavesdropping and man-in-the-middle are two types of cyberattacks often linked with Bluetooth Low Energy devices. Passive eavesdropping is an attack that allows an alien device to tap into data transmitted between devices on a BLE network. BLE modules that implement BLE Secure connections are protected from passive eavesdropping by default. A man-in-the-middle attack involves an alien device that pretends to be both central and peripheral at the same time and tricks other devices on the network into connecting to it. While BLE Secure connections offer protection from passive eavesdropping, man-in-the-middle attacks can be averted only with an appropriate pairing method [11].

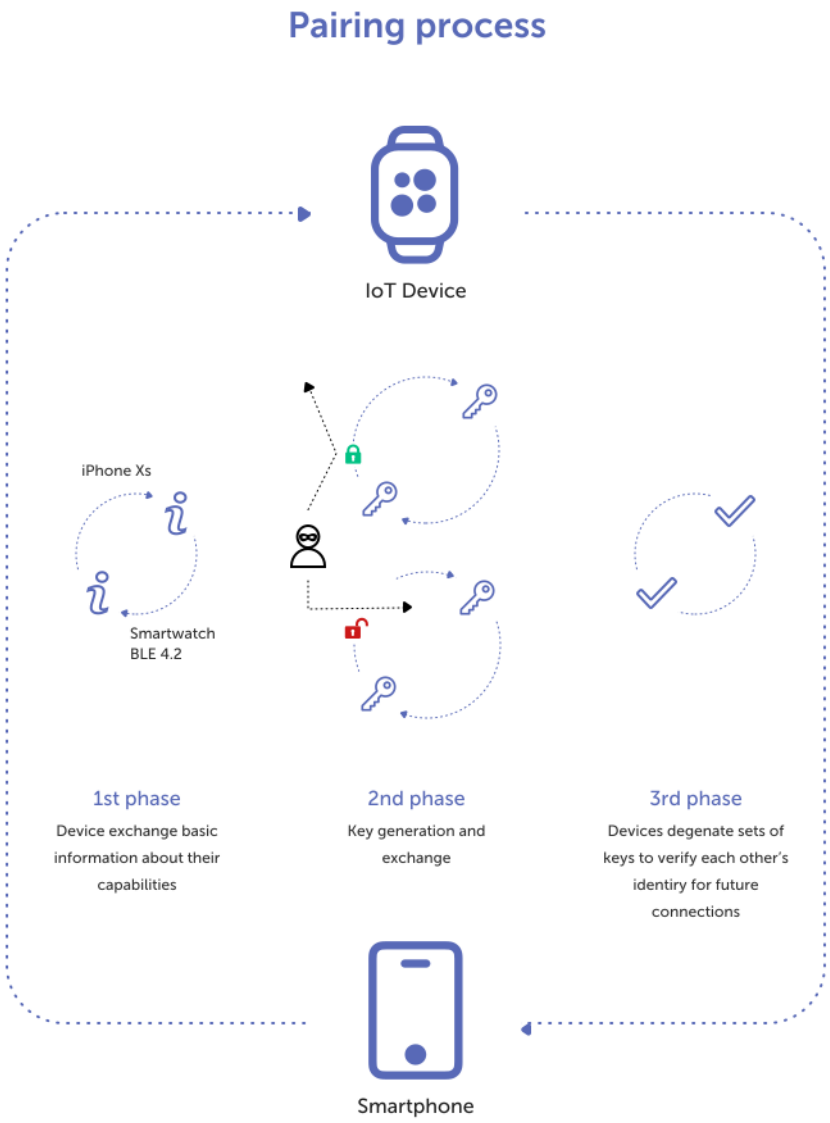

Figure 2.5: BLE pairing process [11]

Bluetooth 5, SIG's most recent release in 2016, gives BLE users the choice of doubling the speed (2 Mbit/s burst) at the price of range, or providing up to four times the range at the penalty of data rate; in addition, when compared to Bluetooth 4, the Bluetooth 5 has been designed to consume less power on the smartphone. Finally, while Bluetooth 4.x was capable of providing a size of only 31 bytes for advertising packets, Bluetooth 5 increased its length to 255 bytes.

Chapter 3

Framework

3.1 Cross-platform development

The main goal of this section is to provide a broad panorama of mobile cross-platform development. Before exploring this topic, it's fundamental to provide some definitions in order to understand the following concepts.

The term $platform$ can be misunderstood when an adequate context is not provided: this term, indeed, can indicate the operating system running on a device or the CPU and hence its Instruction Set Architecture (ISA) or the combination of the two of them. A software application might be developed to leverage the features of a particular platform, even though this means losing the possibility of running the same software on different platforms without any further effort. A common platform for developers is the Java Virtual Machine (JVM): as the name suggests, it is a virtual machine capable of running on several operating systems and hardware architectures, making its success as one of the most favourite platform for which software is developed.

If a software is able to run on multiple computer architectures or operating systems, then it is classified as a cross-platform software. According to this definition, probably the most popular cross-platform software everyone is used to is the web application: indeed, it can be accessed from any web browser regardless of the device's platform. Though, nowadays the most relevant web applications (e.g. Google Mail, Google Maps, etc.) usually depend on features that are available only on the most recent updates of web browsers because they need to support JavaScript, AJAX and all the other components that are commonly used in the so-called Rich Web Applications.

Traditionally software applications are delivered as binary files (also known as executable files). Each binary file supports the hardware architecture and the operating system it was built for. Since this would result in a huge effort in order to deliver the software to the largest set of users, usually the single code base is trans-compiled into different binary files for several target platforms. This is the case of Firefox browser, whose installer can be downloaded in the shape of several executable distributions even if they derive from the same source code.

The use of different tool-sets to perform different builds may not be sufficient to achieve a variety of working executables for different platforms: in this case, the software engineer must *port* it, that is to amend the code to be suitable to a new computer architecture or

operating system [28].

Alternatively, it is possible to run a software compiled for a specific ISA and a specific operating system on computers with a different platform thanks to cross-platform virtualization: the whole procedure relies on both operating system call mapping and dynamic binary translation. This technique introduces a considerable processing overhead resulting in performance decrease with respect to native software; though, this is still used for legacy software running on new machines. The most recent and popular use of cross-platform virtualization happened with the ongoing Apple's Mac transition from Intel's x86-64 CPU to Apple-designed processor which uses ARM64 architecture: indeed, the "Rosetta 2" software, bundled with Mac OS Big Sur release, allows many applications compiled for processors based on x86-64 architecture to be translated for the new hardware architecture.

The interest in cross-platform compatibility and in advanced functionalities has increased throughout the history of software development, so this led to the establishment of several design strategies:

- having multiple code bases is the best strategy in terms of performances and functionalities, but this requires much more effort in development and maintenance
- having a single code base requires the code to be compiled for different platforms: a popular technique, called conditional compilation, allows the developer to avoid duplicating the code for two different platforms and to select code snippets to be compiled (or interpreted) according to the current platform
- third party libraries exploit the concept of API by providing a single interface to access complex functionalities of two different platforms
- the technique of graceful degradation consists in providing the same or similar functionality to all users and platforms, while diminishing that functionality to a least common denominator for more limited client browsers [28]
- Responsive Web Design (RWD) is a Web design approach aimed at crafting the visual layout of sites to provide an optimal viewing experience — easy reading and navigation with a minimum of resizing, panning, and scrolling — across a wide range of devices, from mobile phones to desktop computer monitors: little or no platform-specific code is used with this technique [28]

Given that the most effective approach is having multiple source codes for the different platforms, this is extremely expensive in terms of development cost and development time; moreover, also maintenance costs are considerable because having two (or more) different development teams usually results in two different sets of problems to be fixed. The opposite technique is based on relying on an abstraction layer capable of hiding the differences between platforms: for example, all software applications running on the JVM are unaware of the platform they are running on. Some software houses have adopted an hybrid approach, trying to benefit from the advantages of both solutions: this is the case of Firefox web browser, which uses abstraction to build some of the lower-level components, separate source sub-trees for implementing platform-specific features (like the GUI), and the implementation of more than one scripting language to help facilitate ease of portability [28].

3.2 Cross-platform mobile applications

Most platform frameworks were born basically with the promise of writing the code once and then running it anywhere: this slogan resembles the one devised by Sun Microsystems in 1995, and despite the lapse of time the idea behind cross-platform mobile applications is still the same.

3.2.1 Overview

Nowadays the market of mobile applications is split into two distinct sections: 75% of mobile phones run Android operating system and 25% run iOS operating system, with slight variations depending on the Country. Different policies govern the application store in the two operating systems, starting with the requirement of physically own an Apple device for developing in the iOS market.

In practice, most of the frameworks mentioned later are mainly focused on the creation of the User Interface, and little support is provided for implementing the business logic because this resides in the server, which supports the mobile application in retrieving data.

When a cross-platform mobile application is requested, the most popular reason stands in the market: the reduction of development cost and development time is very attractive for a general stakeholder because the overall sum to be paid will decrease too; moreover, since it's common to find people who know one single framework, it becomes convenient to reuse that knowledge somewhere else.

In the early time of mobile computing, the availability of standard components (meaning that they are the same regardless of the operating system) provided by the WebView software component and the WebKit library, which were the foundations of Safari and Google Chrome browsers, was the core of the development of web-based applications. Though, the web approach was limited by those sandbox constraints that typically characterize web browsers, such as the impossibility to access local file system as well as peripherals (e.g. cameras), limited performances, etc.; thus, frameworks based on WebView ended up relying on external components called plugins whose aim was acting like a bridge between the operating system and the application itself. Plugins introduced a new possibility to interact with the operating system through a browser, but this also meant that a new way of overstepping security barriers was available. For these reasons, this approach got abandoned very soon.

3.2.2 Advantages and disadvantages

Just like every other strategy or decision, also the way of mobile cross-platform development introduces risks and benefits. It is important to analyze them before starting the development of any mobile application.

When a developer devises the mobile application it is quite common to underestimate the misalignment between the distinct user experiences respectively of an Android user and of an iOS user: for example, an Android user expects to find the navigation menu at the bottom of the page and a floating action button as the main button in the screen, while iOS has different user experience guidelines. Moreover, using a framework often

requires the developer to test the result of the user interface on a set of devices as large as possible, because not all platforms are the same (e.g. Android versions have different capabilities) and not all smartphones have the same shape of the screen.

Being on top of the different platforms, cross-platform frameworks are usually able to provide only what is available on both target platforms (in this case Android and iOS): thus, if a certain feature is available only on Android, typically it won't be part of what is offered by the framework itself.

Cross-platform frameworks might require lots of study and practice, and the slope of the learning curve could get quite high when the developer hasn't mastered the programming language used by the framework yet. Also, a cross-platform framework introduces a dependency on the framework provider, that is a risk factor because the developer doesn't have control on future developments of the framework (e.g. compatibility with existing libraries, etc.).

Nevertheless, some advantages can make the difference during the development and along the maintenance activities.

First of all, code re-usability is a remarkable advantage: in fact, developing two separate code bases requires twice the effort for reaching almost the same result on target platforms; in addition, whenever a bug comes up or a new functionality must be developed, the team will accomplish the task in just one go rather than repeating the same tasks twice. Moreover, recently companies are looking very kindly on the cross-platform world mainly for two reasons. Firstly, the process of cross-platform development doesn't need to gather together multiple teams having different skills and to allocate huge financial resources for covering the development of two (or more) applications for different operating systems; in point of fact, given that most cross-platform frameworks rely on popular and well-known programming languages such as JavaScript and C#, developers can leverage on past experiences with those programming languages even if they don't have experience with the native target platform. Secondly, using frameworks speeds up the development process, and this is fundamental for time-to-market reasons and for reacting as soon as possible to customer feedback and market changes.

The need for custom design and the losing interest in native-looking User Experience are some of the cornerstones of a current of thought that values a consistent brand experience across the different touch points. Cross-platform development makes this task very fast and straightforward. Besides, since developing a cross-platform application is much faster, the development team has a prototyping advantage and can leverage on a rapid feedback from testers or end-users.

Finally, cross-platform development helps reaching the greatest exposure to the whole audience, because just one code base is enough to be deployed over different platforms (e.g. for the mobile market Android runs on 74% of smartphones, while iOS on 25%).

3.2.3 Cross-platform frameworks

Modern technologies for mobile cross-platform development can be divided into three categories: WebView-based frameworks, native-widget-based frameworks and custom-widget-based frameworks. WebView-based frameworks belong to the first generation of cross-platform technologies. As mentioned previously, these frameworks are not used anymore. The second generation recognized the need of having a single code base to be

compiled for different architectures. The third generation aims at totally replacing the User Experience provided by the operating system.

In order to understand the differences between these categories, it is fundamental to analyze the architectures of the mobile applications built using their frameworks. The figure 3.1 on page 34 highlights the most relevant differences: indeed, the figure shows the possible interactions between what is provided by the platform (painted in blue) and the code written using the framework (painted in green).

Native code and cross-compiled code (that is the platform-specific code obtained through the process of compilation) can directly invoke platform services (e.g. for using the camera, playing some music, getting the location, etc.).

Code based on WebView (typically written in JavaScript), whose case is reported in the top-right corner of the figure, relies on one single visualization element, called WebView and based on the WebKit library, which is in charge of the layout of the different UI elements on the screen and of catching all possible interaction events coming from the user. In order to escape from the WebView engine, the framework provides a bridge consisting in a code block controlled by the developer by means of plugins: this bridge between the framework and the operating system is able to receive requests from the web browser view and propagate them to the local platform. Besides security issues, the problem of these plugins is that they need to be created specifically for each platform. For example, in order to navigate the file system of the device, both an Android plugin and iOS plugin will be needed. Since WebView is part of both Android and iOS operating systems, it automatically takes care of transforming declarative information received via HTML and CSS code into a visualization reported on a Canvas appearing on the screen.

The case reported in the bottom-left corner is based on non-native code used for describing the mobile application: this architecture requires a much more complex code block, called Bridge and supported by the platform, which is responsible for transforming the description of the desired screen into a set of widgets that the platform itself provides. Thus, when for instance it is required to instantiate a new button on the screen, a real Android or iOS button is instantiated (depending on the underlying platform) along with their well-defined properties: the developer is able to leverage on the real OEM (that is native) widgets that the platform is providing and is aware that these widgets will display themselves and react to events, even though these events must be sent back via the Bridge to non-native code in order to properly react to them. The same happens for services. Thus, it is evident that in this case the Bridge has a central and indispensable role, but the advantage of such a constraint is that the mobile application looks just as if it had been developed natively.

Custom-widget-based frameworks use an architecture like the one represented in the bottom-right corner of the figure. These frameworks are made for taking care of rendering widgets on the screen: thus, instead of relying on existing widgets, these frameworks exploit the capability of accessing the OpenGL computer graphics rendering programming interface for directly controlling the GPU of the smartphone. This makes the duty of designing custom widgets much easier, and the framework itself will provide each widget with the capability of drawing itself inside the screen. Graphic engines on both Android and iOS operating systems are so powerful that they can ensure an acceptable refresh rate for every widget. For accessing the services offered by the underlying platform, these frameworks need a dedicated bridge.

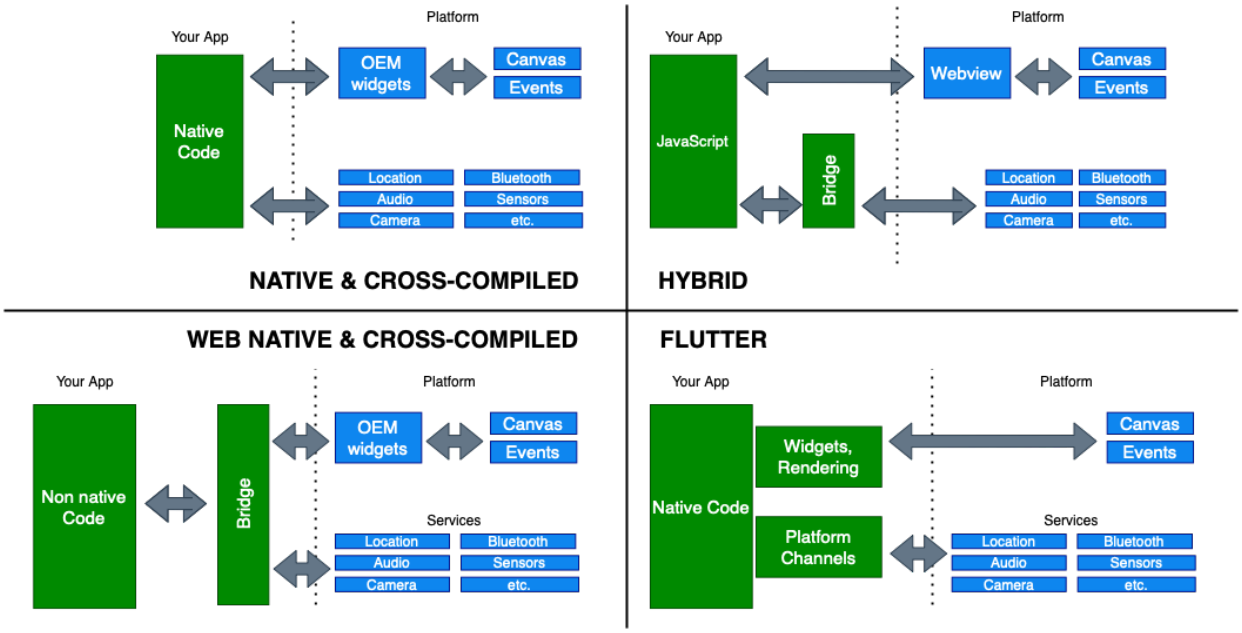

Figure 3.1: Comparison between cross-platform framework architectures [12]

3.2.4 Possible choices

This section summarizes the most popular frameworks from each category introduced in the previous part of this chapter. Finally, this section explains the reasons for which Flutter frameworks has been chosen for the development of the mobile cross-platform application.

3.2.4.1 Apache Cordova and Ionic

The most known framework based on WebView is called Apache Cordova, which originally was a proprietary project but later it was open-sourced. This framework provides the developer with a standard skeleton for each supported platform (e.g. standard project containing one Activity for Android) where a set of predefined plugins are already loaded. A new developer environment, called Ionic, was born following the idea that Apache Cordova was giving the developer too many responsibilities: indeed, it didn't provide any guideline on how to create widgets using ready-made components. Thus, the use of the popular Angular framework smoothly replaced the usage of basic HTML, CSS and standard JavaScript and gave the developer the possibility of structuring the mobile application: the idea behind Ionic (and partially also behind Angular) is that each component has its own User Interface template and follows its own application logic with the capability of accessing system services. Ionic was devised not only for the mobile world but also for the desktop by leveraging the Electron framework. Still the mechanism for accessing the services offered by the platform is provided by Apache Cordova, which is still used behind the scenes. Developing an application based on Ionic framework means using mainly SASS (an extension of CSS which can be compiled) and TypeScript: all these files need to be transpiled, which means that they are transformed into a set of

JavaScript and CSS files together with HTML code and possibly some assets, and then packaged by Apache Cordova into a mobile application; alternatively, it may also become a Progressive Web Application, which doesn't need to pass all verification tests on the app markets.

The main disadvantage of Ionic and of all frameworks based on WebView is the misalignment between the need to emulate the aspect and the behaviour of native components and its slow performances when subject to interaction. This is the foremost reason why these frameworks were abandoned in favor of frameworks providing the capability of using native widgets and all their benefits.

3.2.4.2 Xamarin and React Native

The interest in mobile cross-platform development moved towards the need of assigning the responsibility of creating and managing the User Interface to the underlying platform and creating a separate application layer for handling events and for driving the business logic. Moreover, because the application layer is completely separated by the User Interface, in principle it can be developed using any programming language: thus, people who already master some technology can be potentially dedicated to the application layer of a mobile cross-platform application. Though, sometimes it is required to write native code in order to support specific functionalities and this introduces the need of a Bridge between native and cross-platform part of the code: this mechanism might limit the performances due to the bridging of information.

Xamarin is one of the most popular examples in this context. The idea behind Xamarin is having a set of thread, that belong to a certain process, working in parallel with another set of threads of the same process in order to ensure a communication mechanism. In Xamarin the application logic is written in C#, and the code developed gets compiled into the MSIL (Microsoft Intermediate Language) and executed by the Mono Execution Engine. Sometimes it is not possible to exploit the communication mechanism previously explained, so for the Android operating system Xamarin uses special components, called MCW and ACW, for bridging information; instead, for iOS the mechanism is similar and relies on a binding between the Mono Runtime and the traditional native Objective-C Runtime.

The most popular example of native-widget-based framework, though, is React Native, a framework developed and maintained by Facebook. React Native exploits the same concepts behind React framework: indeed, the developer needs to describe what should be seen on the screen in a declarative fashion so that the framework will take care of matching the description provided with what is already on the screen. Then, the framework is also providing tools for declaring components, specifying dependencies on states and performing some side effects. React Native relies on an internal broker, called JavaScript bridge, for managing data interchange between the main thread of the native application and the JavaScript thread (based on an event loop) executing developer's code.

3.2.4.3 Flutter and Unity

The need of higher performances and custom design implementations on the market pushed the development of different solutions for mobile cross-platform development.

Both Android and iOS operating systems offer a common API for accessing their graphic engines capabilities powered by a Graphics Processing Unit (GPU); in particular, every GPU inside a smartphone is driven by a set of softwares that exploit the OpenGL ES (Embedded Systems) standard, a reduced version of the desktop one. Thanks to this capability, these frameworks are able to create visualizations regardless of the ability of drawing provided by libraries such as Skia and Core Graphics respectively for Android and iOS: the developer has full control on the User Experience and on the User Interface of all components on the screen.

These frameworks are very popular in the gaming environment and made the fortune of Unity, Unreal and other gaming platforms. Though, the most curious and intriguing use of the capability of accessing GPU functionalities is the Flutter framework. Based on the Dart programming language and developed by Google, this framework gives the possibility of defining high-performance graphic components from scratch and drawing them directly on the canvas; in addition, Flutter cross-platform applications are compiled into native code Ahead-Of-Time (AOT), that is the act of compiling Dart source code into Kotlin and Swift code to be executed directly on runtime; finally, Flutter provides an optimized approach for communicating with the native platform, consisting in a set of pipes called channels. More details about this are coming in the next sections.

As accurately explained in the following chapters, Flutter overcomes some of the previously mentioned disadvantages. In particular, one of the worst disadvantages of cross-platform development is the practice of renouncing a certain feature when this is not available on all platforms targeted by the framework: since Flutter uses two separate method channels for communicating with Android and iOS platforms, the developer will be able to leverage the specific functionality on those platforms capable of providing it. Moreover, Flutter's learning curve becomes steep only for those who have no experience with Object-Oriented Programming, even if Dart is still not a popular programming language. Flutter's community is continuously increasing and its popularity is becoming exponential. All these reasons stand behind the decision of using this framework for developing the mobile application for this project.

3.3 Dart programming language

The officially supported language for building cross-platform application using Flutter framework is Dart, a programming language developed by Google.

Dart is completely open-source and its syntax and semantics have been specified by an Ecma standard (the Ecma-408 standard). In few words, Dart is a client-oriented programming language that can be used to create quick apps on any platform; its objective is to provide the most productive programming language for cross-platform development, as well as a versatile runtime platform for app frameworks: indeed, as further explained in the following sections, Dart gives priority to both development and production environments across many compilation targets (mobile, web and desktop).

3.3.1 History

Dart was first revealed in 2011 during a conference held in Denmark, even though version 1.0 was made available only at the end of 2013. At the beginning Dart was not appreciated that much because many web developers criticized its attempt to include a Virtual Machine (VM) inside Chrome browser; nevertheless, Dart project gave up the idea quite soon in 2015 and focused more on compiling Dart to JavaScript for web development purposes. Then, in 2018 Google released Dart 2.0: this version included several language changes that made Dart even more strongly-typed.

Recently Dart has been updated to version 2.12, which is a turning point for what concerns the integration of external packages: indeed, this version introduces a null-safety feature, which automatically ensures that any variable cannot contain the null value unless the developer explicitly tells the compiler it can, and thus turns all runtime null-related segmentation faults into errors during analysis at edit-time. Though, this feature requires external library maintainers to migrate to these changes under penalty of losing the possibility to leverage library's enhancements: due to these issues, this thesis work was built under the previous version (Dart 2.10 version); nevertheless, it's very likely that the majority of maintainers will adapt their libraries within a few weeks or months.

3.3.2 Overview

In order to understand the key concepts around Dart, the best approach is to make a short digression about programming language typing, compilation and virtual machines.

Traditionally, computer languages can be classified into two categories: static languages such as C, C++ and Java, and dynamic languages such as JavaScript and Python. Static languages, besides having the benefit of revealing bugs related to type at compile time, making the code readable and maintainable, are usually compiled into native machine code programs for the target machine, which can be directly executed by the hardware at runtime; instead, dynamic languages require an interpreter for their execution and no target-machine language is produced.

With the Java programming language the notion of a Virtual Machine (VM), which is essentially an enhanced interpreter that emulates a hardware machine in software, gained popularity. A virtual machine makes porting a language to new hardware systems much easier: indeed, a VM's input language is frequently an intermediate language in this scenario; following the well-known example of Java, it is compiled into *bytecode* and then run on a virtual machine (the Java Virtual Machine, or JVM).

Also compilers can be divided into two main types: Just-In-Time (JIT) compilers and Ahead-Of-Time compilers. JIT compilers are actually running during the execution of the software program because they compile the code *on the fly*; instead, AOT compilers are those compilers that run only during the creation of the software program before runtime takes place. Because machine languages usually require to know the type of data, only static languages are suitable to AOT compilation into native machine code; on the contrary, in dynamic languages the type is not established ahead of time and thus most dynamic languages are interpreted or JIT-compiled. The development phase doesn't take much advantage of AOT compilation, because it increases the time period between a change written in the code and code execution for noticing the result of the change;

though, AOT-compiled programs result in better predictability and start their execution faster, unlike JIT-compiled ones which need the JIT compiler to do an analysis before code can be executed. On the contrary, JIT compilation ensures much faster development cycles because the developer doesn't need to wait for the compiler to create an executable program to run.

What does this have to do with Dart? Dart team members had such an experience in compilers and virtual machines for both types of languages that they designed Dart to make it as flexible as possible for compilation and execution. A few languages other than Dart are capable - just like Dart - to be compiled both Ahead Of Time and Just In Time: this results in remarkable advantages, particularly for Flutter. For example, in Dart JIT compilation is used during the development phase, when no developer would like to wait for a long compilation process to notice a slight difference introduced in the User Interface: to do so, Dart employs a standalone VM which leverages Dart as its intermediate language; later on, at the end of the development when the mobile app has to be released, it gets compiled AOT. As a result, Dart can provide the best of both worlds: incredibly quick development cycles and fast execution and startup times. Moreover, since Dart can be compiled into JavaScript too, the code written for the mobile application can mostly be re-used for the web application.

For what concerns the programming language itself, Dart is endowed with a huge set of core libraries, such as the ones providing built-in types, basic and advanced collections (including Hash Maps, Queues and Linked Lists), encoders and decoders (e.g. for JSON and UTF-8), mathematical functions, file system support, HTTP support, I/O support, asynchronous programming, concurrent programming, cryptographic hashing and many more functionalities. Dart is quite similar to Java because of its strongly-typed nature; though, Dart is also able to infer types, so the developer can choose between using the Object class or the *dynamic* base type: while the former actually assigns a type to the object that it is referencing, the latter defers type checking until runtime; also, Dart doesn't use the classic Java visibility keywords such as *public* and *private*, but leverages the way the variable is written: indeed, if it starts with an underscore (_ symbol) then the variable is private to its library. Finally, as most high-level programming languages, Dart provides interesting and handy syntactic sugar in order to reduce the boilerplate code written by the programmer: some examples will be provided and explained in the following chapters while showing the most relevant code snippets of the mobile application.

3.4 Flutter: Software Development Kit for mobile applications

Flutter is a cross-platform UI toolkit that allows apps to interact directly with the underlying platform services while allowing code re-use across operating systems (e.g. iOS and Android). For what concerns developers, their objective is to be able to create high-performance programs that feel natural across platforms, while embracing differences where they exist and sharing the largest possible code-base.

3.4.1 Framework's architecture

Flutter was devised as a system composed of several layers, each one being dependent on the underlying layer; every layer is made of several independent libraries which have no privileged access to the other libraries of different layers. The figure 3.2 on page 40 shows the layered system that characterizes the Flutter framework.

Flutter apps are packaged in the same way as any other native app, from the operating system's point of view. An entrypoint is provided by a platform-specific *Embedder*, which coordinates with the underlying operating system for accessing services (e.g. rendering surfaces and input) as well as for managing the message event loop. The *Embedder* is written in a platform-specific language: for Android the *Embedder* is written in Java and C++, for iOS and Mac OS it is written in Objective-C / Objective-C++, and finally for Windows and Linux it is written in C++. The Flutter engine, which is primarily developed in C++ and implements the primitives required by all Flutter apps, lies at the heart of Flutter. When a new frame has to be painted, the engine is in charge of rasterizing - which is the act of converting an image stored in a certain format into pixels that can be displayed on a screen - complex scenes. This engine implements Flutter's basic API at a low level, including graphics through Skia (an open source 2D graphics library written in C/C++), text layout, file and network I/O, plugin architecture and a Dart runtime and compile toolchain. Developers usually interact with Flutter using the Flutter framework, which is a modern and reactive framework developed using the Dart programming language: it consists of a number of sub-layers that contain a comprehensive collection of platform, layout, and foundational libraries.

Notice that the Flutter framework itself is rather small; nevertheless, many higher-level features have already been built upon the core Dart and Flutter libraries in the form of *packages*, such as platform plugins for using the camera, in-app payments and animations.

In parallel to this architecture, the Flutter team has devised another architecture for web support: this architecture presents some unique characteristics which are worth to be mentioned. Since the very beginning of Dart, it has always been possible to obtain JavaScript code from Dart's compilation process and to differentiate between development and production environments by using the appropriate toolchain. The Flutter Engine, which is further described in section 3.4.1.2, was devised as an interface with the underlying platform rather than with a web browser; hence, the Flutter team has decided to create a separate engine on top of classic web APIs. The figure 3.3 on page 41 shows the layered system that characterizes the Flutter framework for the web. The most evident discrepancy between this architecture and the previous one is the lack of the Engine layer and in particular of Dart's runtime management system due to the fact that the whole Flutter framework (including the code written by the developer) is compiled to JavaScript. In particular, during the development phase, Flutter web exploits *dartdevc*, a compiler supporting incremental compilation and hence allowing Hot Restart; instead, during the production phase, Dart leverages a JavaScript compiler called $dart2js$, which is optimized for the production environment and is responsible for creating a single package, containing both the whole Flutter framework and the application, taking the form of a minified source file.

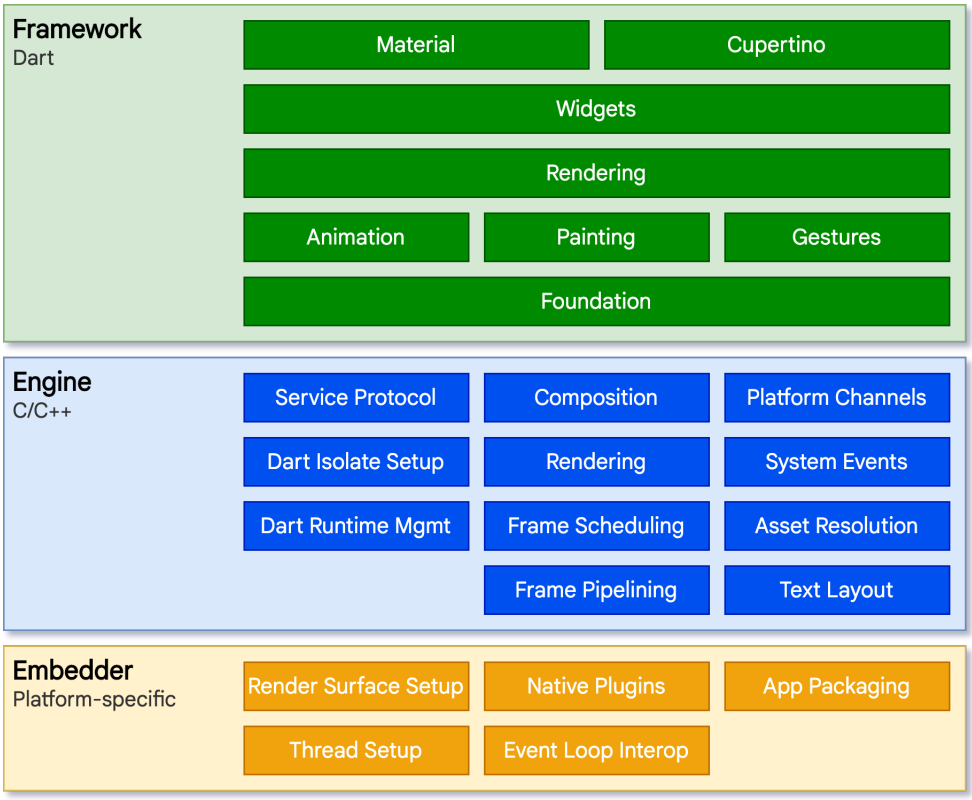

Figure 3.2: Flutter's architectural layers [9]

3.4.1.1 Framework layer

Looking at the most conventional UI frameworks, the initial state of the user interface is defined once and then it gets separately updated at runtime in response to events. One disadvantage of this technique is that, as the program becomes more sophisticated, the developer must be mindful of how state changes propagate throughout the whole user interface. Flutter, like other reactive frameworks, takes a different approach to this issue by separating the user interface from the underlying state.

In Flutter, every widget (taking the form of an immutable class) needs to override the *build*() method in order to declare its own user interface: this method is just a function that, given a state as input, gives the appropriate UI as output; in addition, this function shouldn't have side effects and must return as fast as possible by leaving heavy computational work to be managed asynchronously and by storing its result as part of the state to be used by a *build*() method, which might be potentially called once per frame to be rendered. Widgets, which are the building blocks of any Flutter app, are organized into a hierarchy based on the Composition pattern: indeed, each widget is nested inside its parent and therefore is able to receive the *build context* from its parent. By focusing on describing what a widget is built of rather than the intricacies of changing the user interface from one state to another, the *build*() function simplifies the code. Though, when the user interacts with the app, this must react to events and has to provide the framework with an alternative to the current hierarchy suggesting which widget needs to be replaced: thus, the framework compares previous and future state of the widgets

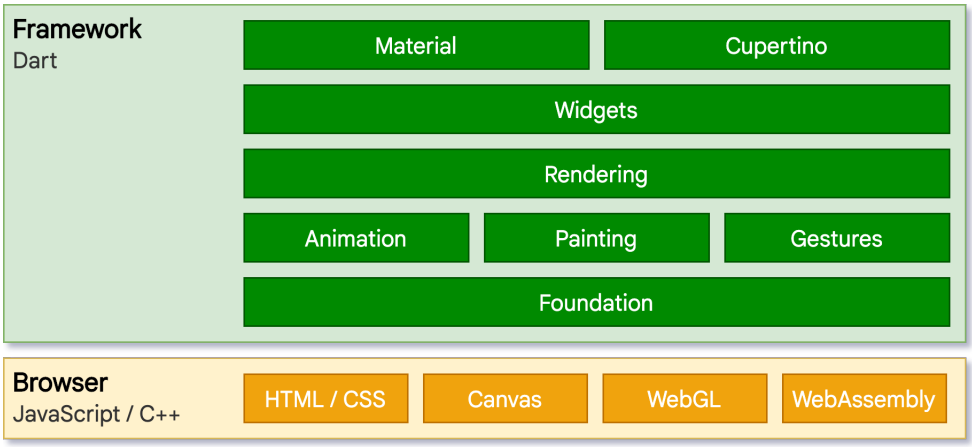

Figure 3.3: Flutter's web architectural layers [9]

and then updates the user interface as efficiently as possible. In order to do so, Flutter introduces two classes of widgets, called *stateful* and *stateless* widgets: while the latter describes widgets having properties that cannot change over time (e.g. a label, an icon), the former characterizes a widget whose attributes need to be changed on the basis of user interaction and other possible events: these widgets have to store a mutable state, and they leverage a separate class that derives from *State* class, which (in place of the original *Widget* class) is responsible for building the widget according to the current state using the overridden *build*() method. To mutate a *State* object it is sufficient to all the *setState*() method, which forces the framework to update the user interface.

The Composition pattern applied to the Flutter framework requires every widget to be potentially composed of many other minor and single-purpose widgets. When dealing with design concepts, for what concerns the sub-layers inside the Framework layer in figure 3.2 on page 40, just a few definitions are provided by the framework, but many small and specific variants are hidden behind each one. For instance, looking at the Animation layer, even though just two concepts - Animation and Tween - characterize this sub-layer, a large plethora of specific variants are further defined such as ColorTween, IntTween, StepTween and many more; moreover, some widgets like *Padding*, *Row* and *Alignment* don't have their own visual representation because their primary purpose is to manipulate the layout of another widget. In practice, in order to maximize the amount of possible combinations, the class hierarchy is intentionally short and wide, focused on tiny and composable widgets, each of them perform one thing well.

In order to convert a hierarchy of widgets into pixels on the device's screen, Flutter follows a series of steps. Traditionally, the Android frameworks needs to call platform-specific Java code for drawing something on the screen; in reality, it is responsibility of the Android native components to draw themselves on a Canvas object so that Android itself can subsequently render them by using the Skia graphic engine. Instead, cross-platform frameworks usually build an abstraction layer over native UI libraries in order to reduce the differences between different platforms. Though, Flutter's approach is totally different: the abstraction layer, which allows most cross-platform frameworks to build a bridge towards system UI widget libraries, is replaced by a mechanism which aims at painting Flutter's own widgets and uses Skia for rendering them after compilation to native code.

In particular, Flutter's render pipeline, which is shown in figure 3.4 on page 42, is made of several simple steps.

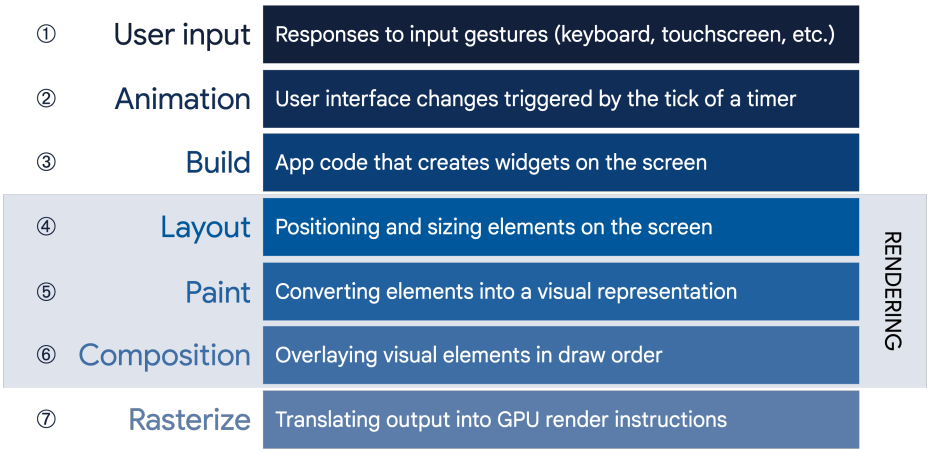

Figure 3.4: Flutter's render pipeline [9]

The third step, called Build, is responsible for introducing small changes in the widget hierarchy so that the widget tree expressed in code can be turned into a more complex *element tree*; furthermore, each of this element is either a *ComponentElement* or a *RenderObjectElement*: while the former is simply a host for other elements, the latter plays an active role in the layout or in the painting. An example of this conversion can be seen in figure 3.5 on page 43. An important aspect of this phase is that the element tree doesn't change from frame to frame, even when the content of a text area is edited: in fact, it is true that any change involving the widget tree causes the creation of a new set of widgets, but the underlying representation, that takes the form of an element tree, doesn't need to be changed; thus, Flutter behaves like the widget hierarchy might be disposed at any moment, yet in reality it caches its underlying representation, which is a key point for Flutter's critical performances.
The fourth phase, called Layout, aims at laying out the hierarchy of widgets by determining the size and the position of every element in the element tree. The base class of each element in the render tree, which is represented in the right-hand side of figure 3.5 on page 43, is called *RenderObject*, but it is just an abstract entity: indeed, this "interface" provides every render object only with the capability of visiting its children and of knowing their constraints. In particular, during this phase, Flutter creates or updates objects inheriting from *RenderObject* for every *RenderObjectElement* in the element tree; for example, from an *Image* widget that has been enriched with a *RawImage* widget, Flutter creates both a *ComponentElement* and a *RenderObjectElement* in the element tree, and then it builds the render tree by considering only *RenderObjectElement*s and by selecting the corresponding *RenderObject*'s sub-class, which is *RenderImage* in this case.

When it is time to perform the layout before rendering on the screen, while traversing the render tree from top to bottom using a depth-first visit, Flutter passes down size restrictions from parent to child; then, the child must adhere to the limitations imposed by its parent while selecting its size: at this point, within the limitations set by the parent,

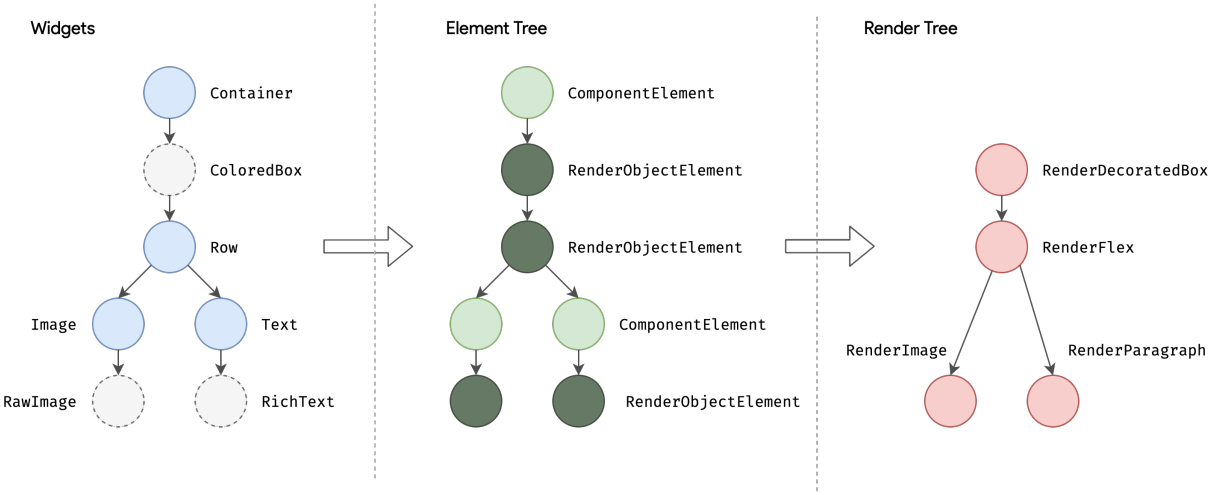

Figure 3.5: Flutter's conversion of widget tree into element tree and then to render tree during the render pipeline [9]

children reply by handing up a size to their parent object. This mechanism is shown in figure 3.6 on page 43. At the end of this procedure, every object in the render tree has negotiated its size, and thus is ready to call the *paint*() method to be painted on the screen. This model, called as box constraint model, is really powerful because it allows the framework to layout objects in $O(n)$ time, where n is the number of objects in the render tree. Notice that it is possible that a parent provides constraints only over width but not over height or vice-versa: this feature is extremely useful when a widget shouldn't be limited on one side, which is the case of some text areas and list views. Finally, the widget can leverage the information provided at this step about the constraints on its size in order to dynamically decide its own layout (e.g. over two columns or over one column). The total output of the render tree is called $RenderView$: this important object is responsible for passing the output of a $SceneBuilder$ object, created every time the platform asks the framework to render a new frame, to the $Window.render()$ method in $dart : ui$ library so that the GPU receives the control of the action and renders everything on the screen.

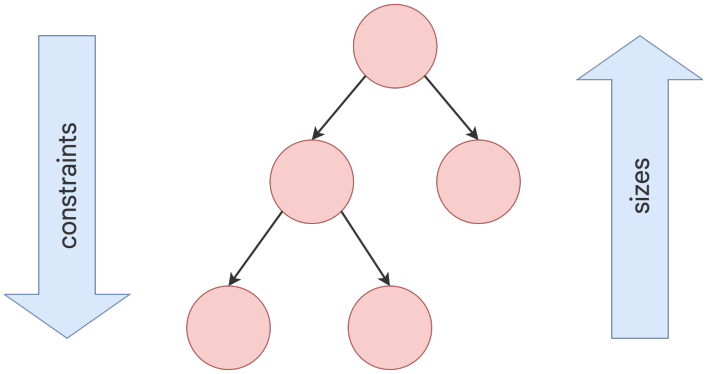

Figure 3.6: Flutter's mechanism of negotiating widget's size according to parent's constraints [9]

3.4.1.2 Engine layer

The Flutter engine is platform-independent, with a solid ABI (Application Binary Interface) that allows platform Embedders to set up and utilize Flutter. In case the developer needs to access code or APIs written in Kotlin or Swift, to contact a native C-based API, to embed native controls in a Flutter app or to embed Flutter in an existing app, Flutter supports a wide range of interoperability techniques.

Flutter mobile and desktop apps are capable of executing custom code by using *platform channels*, which provide a straightforward method to communicate between platform-specific code and Dart code; thus, it is quite easy to send and receive messages between the Flutter application written in Dart and a platform component written in Kotlin or Swift. When using a *platform channel*, messages between the two sides of the communication are serialized from Dart's *Map* type into Kotlin's *HashMap* or Swift's *Dictionary*. The code developed for this thesis work makes use of *platform channels*, so further technical details will be explained in the next chapter.

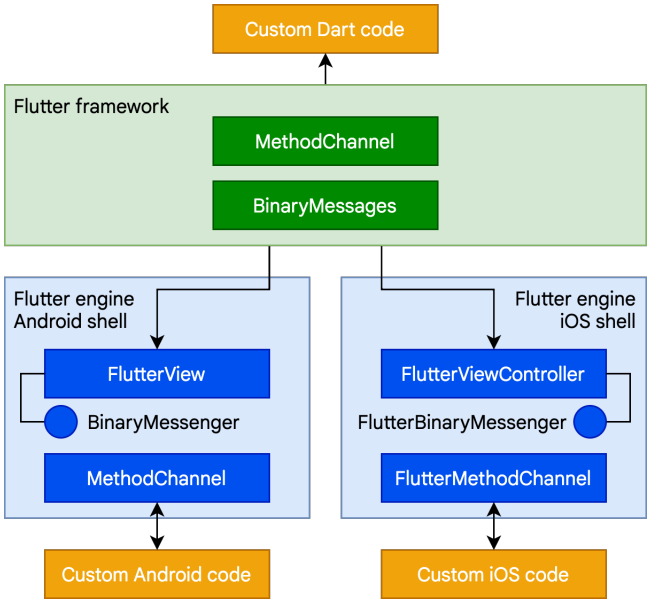

Figure 3.7: Flutter's platform channels' architecture [9]

Every time the app state changes, widgets already depicted are destroyed and rebuilt, even though it has already been discussed that some objects of the element tree are preserved. The same happens when a widget is no longer visible on the screen but later it comes back in sight. Thus, it is quite common that widgets have a short lifespan. Like Java and several other languages, also Dart has its own *garbage collector*, an indispensable element of Dart's runtime in order to automatically allocate and deallocate memory according to objects' reachability.
To minimize the effects of garbage collection on app and UI performance, the garbage collector provides hooks to the Flutter engine that alerts it when the engine detects that the app is idle and there's no user interaction. This gives the garbage collector windows of opportunity to run its collection phases without impacting performance [26].
The activity of Dart's *garbage collector* can be classified in two phases. The first phase

is based on the *weak generational hypothesis* which claims that the majority of Dart objects die young, and thus it focuses on deallocating objects having a short lifespan such as *stateless widgets*; furthermore, this phase is the most common one because of the nature of Flutter framework and because it is way faster that the second phase. The second phase is much heavier that the previous one because it needs both to traverse the entire object graph in order to mark those objects that are still being used and to scan the whole memory for recycling objects that are not marked. This form of garbage collection blocks on the marking phase: no memory mutation can occur and the UI thread is blocked; this phase is more infrequent as short-lived objects are handled by the first phase, but there will be times when the Dart runtime needs to pause in order to run this form of garbage collection [26].

3.4.1.3 Embedder layer

All Flutter content is hosted by platform Embedder, which acts like a native OS application. The *Embedder* provides the entrypoint, initializes the Flutter engine, acquires threads for UI and rastering, and produces a texture that Flutter can write to when the Flutter app is started. Moreover, the *Embedder* manages the lifecycle of the app: indeed, it handles input gestures (e.g. touch events, mouse events), manages threads' lifecycle and dispatches platform messages.

3.4.2 Performance

Flutter makes use of several threads in order to do its job.
The UI thread is the thread that executes Dart code in the Dart virtual machine: indeed, this thread runs both the code written by the developer and the code coming from Flutter framework; as previously mentioned, this thread creates a layer tree (see figure 3.4 on page 42) and sends it to the raster thread so that it can be rendered on the screen: it is fundamental not to block this thread, otherwise performances will get worse because some frames will be skipped, and consequently the app will appear as jerky. Indeed, Flutter attempts to deliver 60 frames per second (fps) UI performance, or 120 fps on devices with 120Hz updates: hence, this means that approximately a new frame must be rendered every 16 ms.
The Raster thread communicates with the GPU after having received the layer tree from the UI thread; Skia, the previously mentioned graphic library, is running on this thread. Although this thread interacts with the GPU, it must not be confused with the thread controlled by the GPU itself: indeed, the Raster thread runs on the CPU of the device.
The Platform thread is the thread executing code coming from plugins and it is the main thread of the platform.
Finally the I/O thread is responsible for executing time-consuming operations (primarily I/O tasks) that would otherwise stymie the UI or Raster threads.

By default, Flutter was built in such a way that most of the times premature optimization is not needed. Broadly speaking, two types of performances are usually taken into consideration for a software application: time performance and space performance.
Performance issues related to time take place when something takes too much time to display or when something forces the device to run at a much higher pace: an example of

this kind of issues is jank, which is the behavior of an animation that should be smooth but it is not, providing an unpleasant effect from user's point of view. When the frame is not ready on time, Flutter has to skip some of the following ones; the deadline of each frame-rendering period is known as *vsync*, so in more technical terms jerk happens when the rendering of a frame lasts more than the *vsync*. Sometimes it might happen that, even though there is no jank, the app is having performance issues with battery or heat: this is due to the battery-hungry nature of the app, which renders a new frame just in time with the next *vsync*; thus, ideally rendering a new frame shouldn't take more than half of the *vsync* time.
Performance issues related to space involve the size of the app in the production environment, memory-hungry behavior or memory leakages.
For both types of issues Flutter provides precise tools to monitor performances.

3.4.3 UI design and development: Hot Reload feature

One of the most interesting and unique features of Flutter is the Hot Reload. By using this characteristic, the developer is capable of experimenting bug fixes, UI enhancements and even slight changes on the screen just after less than one second after editing the code. As explained in section 3.3.2, Hot Reload leverages the JIT compilation made by the Dart Virtual Machine (VM) by injecting source code files that have been edited: indeed, the Flutter framework instantly rebuilds the widget tree once the VM updates classes with new versions of fields and methods, in order to immediately show the consequences of the modifications. Moreover, when the framework rebuilds the widget tree, only the code involved in recreating the widgets is re-executed automatically but the methods main() and initState(), for example, are not called again.
Performing an Hot Reload preserves the state of the app, and thus it will resume from where it was before the hot reload instruction was sent, which is usually the expected behavior; if the app needs a user to log in, for example, the developer may change and refresh a page several levels down in the navigation hierarchy without having to enter login credentials again.

If code changes impact the state of your app (or its dependencies), the data your app has to work with may not be completely consistent with the data it would have if it ran from the beginning. As a result, behavior after a hot reload versus a hot restart may differ. Thus, together with Hot Reload, also the Hot Restart feature might come in help of the developer: in fact, there may be some cases when the developer doesn't want to preserve the state and would prefer to apply the changes with the app starting from scratch. Thus, executing an Hot Restart means that the Flutter app is restarted once the code modifications are loaded into the VM, and the app state is lost.

Of course, a full restart can be performed, but most of the times there is no point in doing so; in addition, a full restart takes longer since the Java / Kotlin / Objective C / Swift code is also recompiled and it also restarts the Dart Development Compiler on the web.

3.4.4 Testing

Testing is a fundamental part of software development because it contributes to guarantee that the app will still work even after adding more features or changing the existing ones (regression testing) and that the functionality developed are performing the right job. In Flutter and, more generally, in Dart there are three types of tests:

- unit tests, which are responsible for testing a single function, a single method or a single class under a wide range of conditions
- widget tests, which ensures that the widget's UI appears and behaves as expected by exploiting a test environment which properly offers the right widget lifecycle context
- integration tests, which test either the whole app or a large part of it with the aim of verifying that all services and widgets behave as expected when working together.

This section focuses on unit testing because this is the only kind of testing developed inside the thesis work. Flutter provides the *test* package for writing unit tests by following a specific syntax. Even if it is not mandatory, usually test files are collected inside the *test* folder at the root of the Flutter application project.

The following code snippet shows how to write a unit test for verifying that the *RegistrationCubit* class is correctly working. Tests are defined using the top-level *test*() function and the corresponding assertions are written using the *expect*() function: both of them are provided by the *test* package. Tests are grouped together using the *group*() function, which receives a description of the tests group; moreover, also the *test*() function receives a description of the test as its first parameter. In order to share code between all the tests of the same group, it is possible to use the *setUp*() and the *tearDown*() functions: in particular, the former is run before every test in the group, while the latter is run after each test in the group (even when the test fails, for cleaning up).

Since these tests are focused on the *RegistrationCubit* class, the *setUp*() function is responsible for instantiating this class; in reality, because this needs to be a unit test and because the *RegistrationCubit* class depends on the *UserRepository* class, in this phase a mock representation of this class is used in its place in order to avoid any influence of that class on the tests: this has been possible by providing a mock class that looks like the *UserRepository* class but actually it is way simpler than that. The *tearDown*() function is responsible for deallocating any resource instantiated by the *RegistrationCubit* object: more about *Cubit* objects will be explained in further details in section 4.3.1.2. The first test is just checking that the initial state of the Cubit is *UnknownRegistrationState* because this is the starting state of every *RegistrationCubit* object; the second test is a *blocTest*, which is a special test designed for *Cubits* and *Blocs*, so it will be further explained later on in section 4.3.1.2: in a few words, this test is divided into three steps, of which the first returns the *Cubit* or *Bloc* object, the second calls the function to be tested, and the third checks the type of *RegistrationState emitted* by the *Cubit* object.

```
import 'package:test/test.dart';
import 'package:bloc_test/bloc_test.dart';
```

```
void main() {
  group('test Registration Cubit', () {

    RegistrationCubit registrationCubit;

    setUp(() {
      initModule(RegistrationModuleHelper());
      registrationCubit = RegistrationCubit(userRepository: Modular.get<MockUserRepository>());
    });

    tearDown(() => registrationCubit.close());

    test('should check that initial state is an UnknownRegistrationState instance', () {
      expect(registrationCubit.state.runtimeType, UnknownRegistrationState);
    });

    blocTest('the cubit should emit a RegisteredState when method registerUser() is called',
      build: () => registrationCubit,
      act: (RegistrationCubit registrationCubit) async {
        await registrationCubit.registerUser(
          name: "randomName",
          surname: "randomSurname",
          birth: "01-01-1970",
          email: "randomEmail",
          password: "randomPassword"
        );
      },
      expect: [isA<RegisteredState>()]
    );

  });
}
```

3.5 Ahead-Of-Time Compiling

This section completes the discussion about how Dart's AOT compilation occurs. Usually the AOT compiler is used in the production phase, when the developer wants to deploy her app on different app stores. The result of AOT compilation is some binary code ready to be executed on a specific runtime environment; in particular, the code resulting from AOT compilation can run inside a Dart runtime such as the one provided in the Engine layer (see section 3.4.1.2). Fast startup and stable runtime performance are assured with AOT-compiled code, with no delay during early runs; instead, JIT-compiled code is slower at first, but once it has run long enough then it can have higher peak performance.

The main disadvantage of AOT compilation is that it depends on users' device architecture: indeed, each different architecture must have different binary code, but this increases the app bundle size. In addition, some operating systems such as Android do not require the binary code to get execution permissions, but others like Apple's iOS is much more strict and makes the process of delivering an app quite cumbersome.

The AOT compilation mechanism requires Dart source code to be translated into assembly files and thereafter the Assembler needs to compile this output into binary code for all the different architectures: notice that the assembly files are still platform-agnostic and that they represent an input to the Assembler, which has to consider all the possible platforms.

Chapter 4

Mobile application

4.1 User Interface

This section is about the User Interface (UI) designed and developed for this thesis work: starting from the design of every page of the mobile application to the code implementing it, every aspect will be illustrated hereafter. Moreover, during the description of the UI sometimes the focus will temporarily move on important characteristics of the User Experience (UX).

As previously explained in the previous chapter, every pixel of the screen showing a Flutter mobile application is powered by custom widget painted by the smartphone's Graphics Processing Unit (GPU), which were designed from scratch. In particular, as shown by the figures in the next section, the overall look and feel of the mobile application was designed in order to communicate a sense of freshness and peacefulness by using colors that are usually associated with the ideas of breathing and pureness.

The starting point for the design of the mobile application's UI was the development of a high-fidelity prototype. Building a high-fidelity prototype is usually one of the last steps in prototyping, because it is not the best tool to solve possible issues with the flow and the main understanding of the UI since it is close to that will be the final result. Instead, this type of prototype is useful when deciding screen layout (e.g. check whether UI elements are overwhelming the screen with information or they are distracting the user), colors and fonts; further, high-fidelity prototypes are also used to test the user reaction on changes into the interface (e.g. whether the user is understanding what the application is telling).

The set of tools For building interactive high-fidelity prototypes is quite large. For this thesis work, the Figma design tool was used.

4.1.1 Graphic tools: Figma

Figma is an online UI tool that allows designers to build prototypes for their projects. As many other softwares working in the cloud, every change in a Figma project is automatically saved and historicized; moreover, Figma allows the user to discuss about the project with other people without having them install anything in their computers: this was fundamental for having feedbacks from the research group. An always growing community lives around Figma: many users create and share their own UI components and ideas so

that people can take their cue from other works; in addition, Figma plugins make some tasks straightforward by providing graphic tools for screen navigation, icon searching and much more.

4.1.2 Data visualization

Considerable amount of data is produced by all the sensors to be spread around the city, and therefore taking care of how this information is presented to the user is a remarkably fundamental point of this mobile application. Data visualization studies the graphical representation of data by exploiting visual components such as charts, graphs and maps; indeed, the main goal of data visualization is having the user understand patterns and trends inside data.

The main visualization instruments exploited in this thesis work are bar charts, line charts and tables. From a technical point of view, these tools were mainly provided by a package called fl_chart shared on the official Dart and Flutter package manager, *pub.dev*; this package is the first of a long list of external UI tools used inside the application. The fl_chart package offers interesting tools for data visualization, is a very popular choice among developers and has the top-of-the-class score for code style, maintainability and platform support.

When designing the Home page of the mobile application, the intent was to communicate the current air quality conditions around the user (see section 4.2.1 for details on how it is computed) and to highlight the trend in the measurements of the latest hours. Thus, the first page that the user will encounter when opening the mobile application needs to summarize this information, and the best way to show the trend of air quality measurements is displaying a line chart: indeed, according to data visualization studies, for highlighting data changes throughout time this is the optimum choice. The figure 4.2 on page 57 is showing part of the Home page: because the two measurements have the same measurement unit, $\mu g/m^3$, it is possible to represent them in the same line chart with a short legend in the top right corner. Moreover, the figure is also highlighting that the user can interact with the chart by touch: in fact, on user touch, a popup label will appear for showing the numerical data of both measurements at that time; finally, the line chart is horizontally scrollable so that the user can see the measurements' trend even after some hours.

In order to get this result, the fl_chart package provides the *LineChart* class together with a set of classes used to set the properties of the chart: in this case, as reported in the code snippet below, the line chart has no left title for the Y axis but it has a bottom title for the X axis, which is represented by labels indicating the hourly passing of time; then, it shows two lines painted with different colors and with with a certain width and curve smoothness on a black background without grids.

```
import 'package:fl_chart/fl_chart.dart';

LineChart(
    LineChartData(
        titlesData: FlTitlesData(
            leftTitles: SideTitles(
              showTitles: false,
            ),
            bottomTitles: SideTitles(
```

```
              showTitles: true,
              interval: 12,
              getTitles: (double x) {
                DateTime measureDate = measures.where((measure) => measure.type == Type.PM25)
                  .toList()[x.toInt()].date;

                return "${measureDate.hour < 10 ? '0${measureDate.hour}' : measureDate.hour}:
                  ${measureDate.minute < 10 ? '0${measureDate.minute}' : measureDate.minute}";
              })),
        gridData: FlGridData(show: false),
        borderData: FlBorderData(show: false),
        lineBarsData: [
          LineChartBarData(
              spots: buildChartSpots(measures, Type.PM25),
              colors: [
                lineChartPM25Color,
              ],
              barWidth: 3.0,
              isCurved: true,
              curveSmoothness: 0.55,
              preventCurveOverShooting: true,
              preventCurveOvershootingThreshold: 2.5,
              dotData: FlDotData(show: false)),
          LineChartBarData(
              spots: buildChartSpots(measures, Type.PM10),
              colors: [
                lineChartPM10Color,
              ],
              barWidth: 3.0,
              isCurved: true,
              curveSmoothness: 0.55,
              preventCurveOverShooting: true,
              preventCurveOvershootingThreshold: 2.5,
              dotData: FlDotData(show: false)),
        ]),
)
```

Another fundamental example of data visualization is the Sensor Details screen, reported in figure 4.3 on page 58: this screen provides detailed information about air quality measurements collected by a specific sensor, which are shown inside a bar chart. The bar chart is one of the most popular visualization tool and it allows the user to compare different data points in a straightforward way; also, the color of each bar delivers information about the severity of the air quality at a certain time. The user can interact with this chart too, in the exact same way as the previous one.

The *fl_chart* package provides the *BarChart* class together with a set of classes used to set the properties of the chart: besides the properties presented in the previous code snippet, for this chart it is possible to set the maximum ordinate of the Y axis and, of course, the color of each bar, which is part of the *buildBarGroups*() function.

```
import 'package:fl_chart/fl_chart.dart';

BarChart(
    BarChartData(
      barGroups: buildBarGroups(measures),
      titlesData: FlTitlesData(
          leftTitles: SideTitles(
            showTitles: false,
          ),
          bottomTitles: SideTitles(
              showTitles: true,
              interval: 12,
              getTitles: (double x) {
                if( x.toInt() % 12 != 0 ) return '';
```

```
                DateTime measureDate = measures.where((measure) =>
                    measure.type == (Modular.get<TabCubit>().currentPage == 0 ?
                        Type.PM25 : Type.PM10)).toList()[x.toInt()].date;

                return "${measureDate.hour < 10 ?
                    '0${measureDate.hour}' : measureDate.hour}:${measureDate.minute < 10 ?
                        '0${measureDate.minute}' : measureDate.minute}";
            })),
      gridData: FlGridData(show: false),
      borderData: FlBorderData(show: false),
      maxY: maxY + maxY / 3,
      barTouchData: BarTouchData(
          touchTooltipData: BarTouchTooltipData(
              tooltipRoundedRadius: 30,
              tooltipBgColor: bottomNavBarColor,
              getTooltipItem: (barChartGroupData,
                      groupIndex,
                      barChartRodData,
                      rodIndex) =>
                  BarTooltipItem('${barChartRodData.y.toStringAsFixed(2)} ug/m³',
                    TextStyle(color: Colors.white))))),
    swapAnimationDuration:
      Duration(milliseconds: 300)
)
```

4.1.3 Animations

Animations are an important part of the User Experience (UX) when building a mobile application because they improve the UI making it more intuitive and polished. Flutter supports developers in building any kind of animation types: indeed, most of the Material widgets already provide standard animation effects in their specifications; nevertheless, Flutter can offer developers full customization.

The Flutter team has realized a very helpful flowchart in order to properly find the best tool for animating a widget. Generally Flutter animations can be classified into two categories: drawing-based animation and code-based animation; while the latter is directly focused on the widget and is still bounded to standard layout primitives (e.g. columns, rows, color), the former looks like it has been drawn and it usually is the core of gaming characters or transformations that otherwise would be difficult to express in code. Drawing-based animations can be obtained by exploiting third-party tools which help building the animation from a graphic point of view and then exporting it into Flutter: these animations are too complex for the purpose of this project, so they won't be addressed. Instead, code-based animations are simpler to realize and don't require any external tool to integrate. These animations are further divided into implicit and explicit animations: implicit animations are based on simply setting a new value for a widget's property and consequently having Flutter taking care of animating it from the current value to the new value; alternatively, explicit animations must be explicitly told when to start, hence they require an animation controller, and they can do basically the same as the implicit ones at the price of managing the lifecycle of the animation controller (inside a $StatefulWidget$). Explicit animations must be used:

- when the animation needs to repeat many times (potentially forever) until a certain condition is true
- when the animation is discontinuous, like a circle repeatedly growing in size from small to large without shrinking back again

- when multiple widgets are animating together in a coordinated way
- when it is important to decide the moment when to start the animation;

for all other cases an implicit animation is enough. In particular, implicit animations are often obtained by using built-in implicit animation widgets, typically named with the *Animated−* prefix followed by the name of the property to animate; also, the *AnimatedContainer* class is quite powerful in realizing these kinds of animation; however, if no built-in implicit animation widget fits the animation needs, the *TweenAnimationBuilder* class provides the right tools for creating a custom implicit animation. Also explicit animations are provided by Flutter as built-in explicit animation widgets, which usually use $-Transition$ suffix preceded by the name of the property to animate; yet, in case these widgets are not suitable for the animation, the Flutter team suggests either to extend the *AnimatedWidget* class when preferring a standalone custom explicit animation or to use the *AnimatedBuilder* class otherwise. Finally, in case of UI performance issues, it is recommended to animate widgets by means of the *CustomPainter* class, which is capable of painting directly to the Canvas.

In order to understand the following code snippets, following the explicit animation paradigm for animating widgets, three class definitions must be introduced.

Every *Animation* object in Flutter doesn't know anything about what the screen is showing: indeed, this abstract class's aim is to take care of the current value of the animation and of its state (e.g. completed, dismissed); over a certain period of time, an Animation object interpolates numbers between two values in a sequential manner. its output may be either linear, a step function, a curve or any other possible mapping. Moreover, the Animation object may run in reverse or even swap directions in the middle. Its *value* member is an important property used for accessing the current value of the object's state.

An *AnimationController* object is a special case of an *Animation* object because it is capable of generating a new value every time the hardware requires a new frame to be displayed. The interesting features of this object are its methods for controlling the animation: indeed, to start an animation the $forward()$ method must be called; of course, the generation of interpolated numbers depends on the screen refresh rate (thus, in general 60 numbers per second); then, every *Animation* object bound to the *AnimationController* will call the attached *Listener* objects in order to react to the change. One of *AnimationController*'s constructor paramaters is the *vsync* argument, which avoids excessive resource use by off-screen animations.

AnimationController is just like an *Animation* but its value ranges from 0.0 to 1.0; thus, in case the animation needs a different range, the *Tween* object helps configuring the animation to interpolate that specific range (its constructor has two parameters, *begin* and *end*). This object doesn't have its own state, but its *evaluate*(*Animation animation*) method is responsible for mapping the value of the *Animation* object to the desired value. Moreover, the *animate*(*Animation animation*) method generates the interpolated desired numbers in that moment.

Despite it is difficult to appreciate an animation from a static screenshot, one of the most effective examples of animations implemented within the mobile application is the palpitating halo that shows the current Air Quality Index (AQI) value in the Home page,

which is represented in figure 4.2 on page 57. This animation was realized with the following code snippets, the first one taken from the *EllipsisWithChartSection* class which in the *build*() method instantiates the *EllipsisHalo* class, reported in the second snippet, by passing *haloTween* and *animation* objects.

```
AnimationController animationController;
Animation curve;
Animation<double> animation;
Tween haloTween;

@override
void initState() {
    super.initState();
    animationController = AnimationController(
        duration: const Duration(milliseconds: 1500), vsync: this);
    animation =
        CurvedAnimation(parent: animationController, curve: Curves.decelerate)
          ..addListener(() {
            AnimationStatus status = animationController.status;
            if (status == AnimationStatus.completed) {
              animationController.reverse();
            } else if (status == AnimationStatus.dismissed) {
              animationController.forward();
            }
            setState(() {});
          });
    haloTween = Tween<double>(begin: 0.49, end: 0.52);
    animationController.forward();
}
```

Notice that the *animation* object becomes strictly bond over the *animationController* object, so that when the animation is started by the $forward()$ method then the interpolated numbers between 0.0 and 1.0 are generated and then mapped into the range indicated by the *haloTween* object; moreover, the code exploits the .. syntax, which allows to call the following method on the object preceding the two dots with no side effects on the return value, to add a listener on the animation status: indeed, this listener will be called every time the value of the animation changes (potentially 60 times per second) in order to force the framework to rebuild the widget by using the *setState*() function.

```
Container(
    width: 204,
    height: 204,
    decoration: BoxDecoration(
        shape: BoxShape.circle,
        gradient: RadialGradient(
          radius: haloTween.animate(animation).value,
          center: Alignment(0, 0),
          colors: [
            paintAQIDarkCircle(...).withOpacity(0.7),
            paintAQIDarkCircle(...).withOpacity(0.6),
            paintAQIDarkCircle(...).withOpacity(0.1),
            paintAQIDarkCircle(...).withOpacity(0),
          ],
          stops: [0.5, 0.8, 0.9, 1],
        ))
)
```

The last code snippet highlights the most important aspects about the UI: the halo behind the circle should create a smooth effect that increases the more it is further from the center of the circle, so a dynamic set of *Color* objects with different opacity is used

to obtain this effect; finally, the animating palpitation effect is realized by dynamically changing the radius of the circular *BoxDecoration* object.

The whole code developed for this mobile application contains many other examples of animating widgets that have been created following this technique; implicit animations have rarely been used because most of the times the context required to animate the widgets after user's interaction or until a certain condition was true. An original and quite time-consuming animation effect is the one that animates the needle of the sensor's card shown in figure 4.4 on page 59.

4.1.4 Authentication, Home, Map, Settings, Profile

This section is meant to show the main concerns about the User Interface (UI) and the User Experience (UX) on the most important pages of the mobile application. This space can be considered also as a collection of screenshots taken from the final version of the mobile application to be referenced throughout this paper. Notice that every information shown in the following figures has been randomly generated for visualization purposes and doesn't reflect the real air pollution levels of any city.

The design of any UI should take into account the kind of users of the system itself. In this case the mobile application doesn't have a clear specific target except for people interested in air pollution conditions; thus, given the broad nature of users, the words, phrases and concepts expressed on the screen should be as general as possible for having every user understand.

The following screenshot displays the Login page of the application. In this page some alternatives are presented to the user for logging in: in particular, the first half of the screen is reserved for traditional login with email and password (and a handful link to recover the password); conversely, the second half features two buttons, one for logging in with Facebook and the other for logging in with Google. Both of these, when pressed, trigger the default browser to open and guide the user into the logging procedure on the selected service: in order to realize this feature, both Facebook APIs and Google APIs have been integrated inside the application. Lastly, the "Sign in" button features a pleasant animation while waiting for the server to respond and when the response arrives.

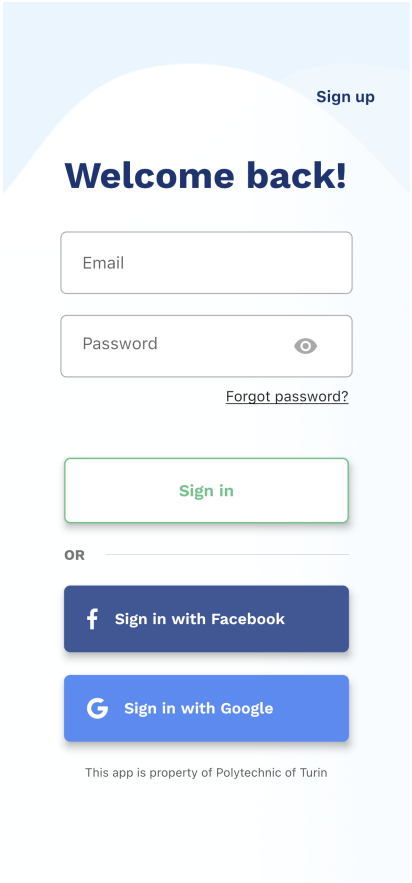

Figure 4.1: Login screen

Next screenshot shows part of the Home page of the mobile application. The first part of the screen shows some *chips* horizontally arranged in such a way which suggests that the user can scroll them: in particular, their design suggests that currently the last chip is in a "selected" status and that this selection can be changed by pressing another *chip*. After that, the following panel is indicating the last update of the nearest sensor for the city which the user is currently located in. Moreover, the next widget summarizes the $PM_{2.5}$ and PM_{10} concentration values according to the Reff algorithm (see section 1.2.4); then, the big circular halo widget indicates the Air Quality Index based on Reff algorithm (for more information about the computation, see section 4.2.1). The next and last widget that can be seen in the figure is the line chart that has been described in section 4.1.2. For the sake of completeness, the widgets that are missing in this figure are dedicated to latest data about temperature and relative humidity together with two big cards that, when the user presses one of them, trigger a dialog raising from the bottom of the screen and containing useful information about air pollution: despite the complexity of the topic, this information has been condensed in a concise space for avoiding overwhelming the main page and for attracting sceptic users' interest.

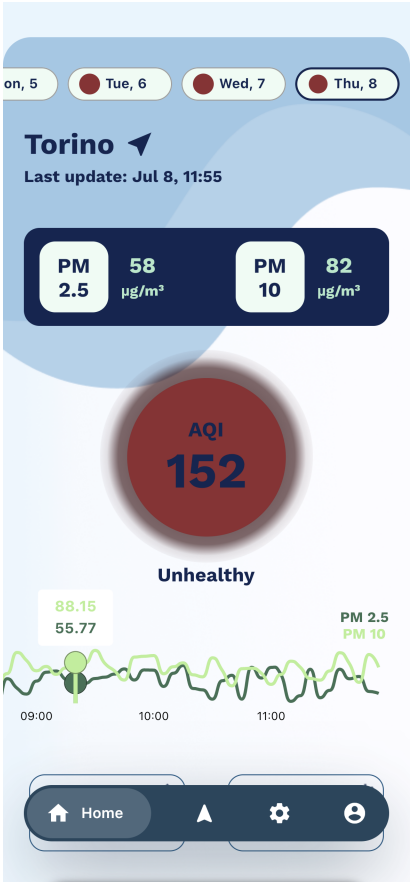

Figure 4.2: Home screen

The following figure shows the bottom part of the page related to the details about the measurements taken from a specific sensor. Notice that every screenshot has been taken using a last-generation iPhone having the notch in the upper part of the screen: this is why it looks like the upper part of the screen is not used, but actually that space is wisely reserved for the notch thanks to the $SafeArea$ widget class. The design of this page deliberately puts a tab bar on top of every other widget in order to make the user aware whether the data represented on the screen refer to $PM_{2.5}$ or PM_{10} measurements; moreover, this design decision always gives the user the possibility to reach the controls for changing the data set even in case the page might be enriched with more information in the future. In the bottom part of the screen, the history panel shows the user the 5 minutes average of the measures collected by that specific sensor: while the bars' height communicates the trend of the measurement, the main information - air quality severity - is delivered by the color of each bar according to the color code of the official US EPA documentation [10]; this is a good example of well-structured information density because two dimensions (height and color) are exploited to deliver information in the same space. Finally, the user can also interact with this bar chart in order to be told the value associated to the selected bar by pressing it: a blue label with the textual information will appear on top of it.

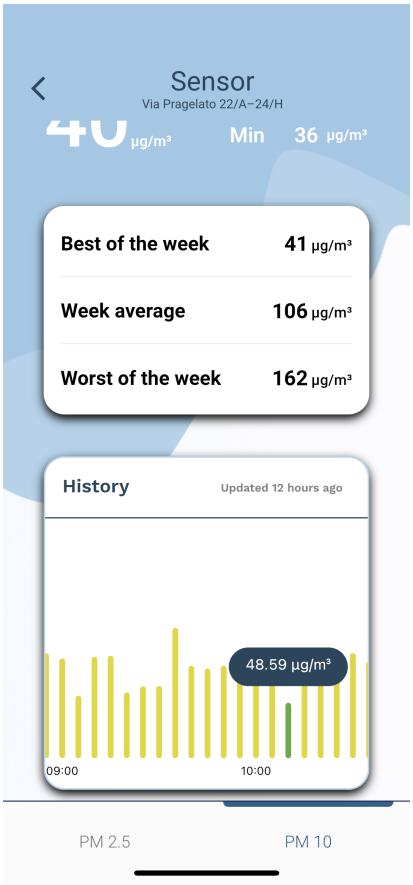

Figure 4.3: Sensor Details screen: PM10 measurements

The next screenshot is the Map page. This page is totally dedicated to show the user where she is with respect to the monitoring devices spread around the city. This screen has been created mainly thanks to the *flutter_map* plugin, which is responsible for providing the most needed map features such as zooming in/out, adding custom markers on specific coordinates, centering the map, creating clusters of markers and so on. The map is a tiled web map, that is a map that is rendered by means of dozens of image or vector data files requested to a tile map service: in this case, the tile layer is provided by MapBox services included in a free account, so currently there is a reasonable limit to the amount of tiles that can be downloaded but it was enough for development purposes. In the upper part of the screen a search bar invites the user to look for a destination to reach on foot: indeed, this output is explicitly shown by the hint label inside the search bar. The bottom part of the screen is a card that raises from the bottom after the user presses on a sensor marker: this card is meant to summarize at a glance the main information about the sensor, which is consequently placed at the center of the map, showing information about the latest air quality data, temperature and relative humidity (RH).

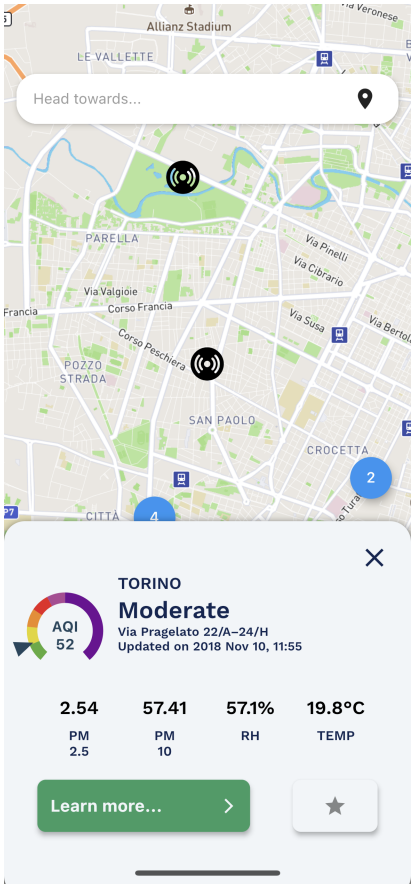

Figure 4.4: Map screen: sensor card

Another screenshot related to Map functionalities is shown below. Thanks to the upper search bar, the user has the opportunity to look for destinations to reach, including Points Of Interest (POI) as displayed in the figure. This feature will be further explained in section 4.2.2 and in section 4.3.1.2 because respectively it offers the possibility to find the best path based on air quality conditions and it is the perfect example to illustrate how *Blocs* work.

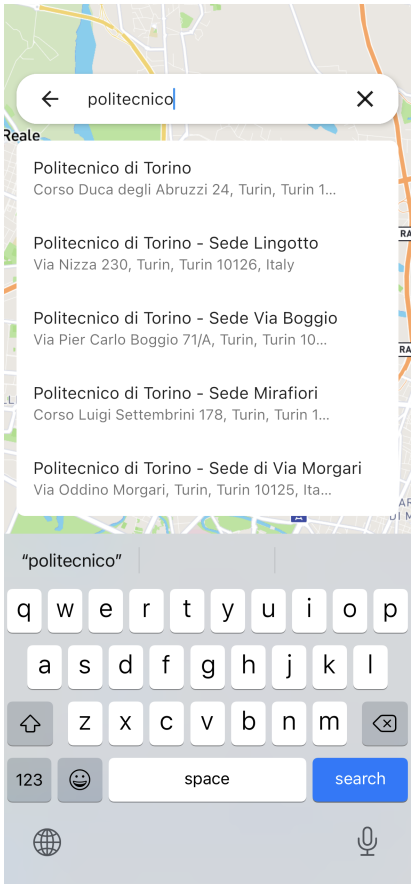

Figure 4.5: Map screen: search bar and results

Let's move on to the next module of the mobile application, which is dedicated to settings. As illustrated in the next figure, this screen is extremely simple but structured at the same time. The upper section is dedicated to user's preferences, where it is possible to set the preferred measurement unit for temperature and the language of the whole application between English and Italian: indeed, every string written on the screen has been translated in these languages thanks to the *easy_localization* package, which allows to store each translation inside a JSON file and hence to easily reference the key (with dot notation in case it is nested) when providing the first argument of a *Text* widget. An example of its usage is reported on the code below: notice how straightforward it is to partition and nest all the keys by dividing the different screens and their different sections; furthermore, it is remarkable to notice that the $showAlertDialog < OkCancelResult > ()$ function, provided by the *adaptive_dialog* package, shows the logout dialog according to the operating system of the smartphone: thus, on an iPhone the dialog (including the buttons to confirm or cancel) will look like traditional iOS dialogs, while on a smartphone running Android the dialog will look like Material Design dialogs. Conversely, the second part of the screen is dedicated to local notifications management: section 4.2.4 will provide precise details about this feature, though for what concerns the UX this feature's goal is to increase the user's engagement with the app itself as notifications encourage users to open the app even after long time since last use. The design is quite standard and features two switches following the Material style.

```
Future<void> _handleLogoutTap(BuildContext context) async {
    await showAlertDialog<OkCancelResult>(
        context: context,
        title: 'profile.logoutMessage.title'.tr(),
        message: 'profile.logoutMessage.subtitle'.tr(),
        actions: [
          AlertDialogAction(
            label: 'profile.logoutMessage.cancel'.tr(),
            key: OkCancelResult.cancel,
            isDefaultAction: true,
          ),
          AlertDialogAction(
            label: 'profile.logoutMessage.ok'.tr(),
            key: OkCancelResult.ok,
            isDefaultAction: false,
            isDestructiveAction: true,
          ),
        ]).then(...);
}
```

```
{
    "profile": {
        "logoutMessage" : {
          "title": "Log out",
          "subtitle": "Are you sure to log out?",
          "cancel": "Cancel",
          "ok": "Log out"
        }
    }
}
```

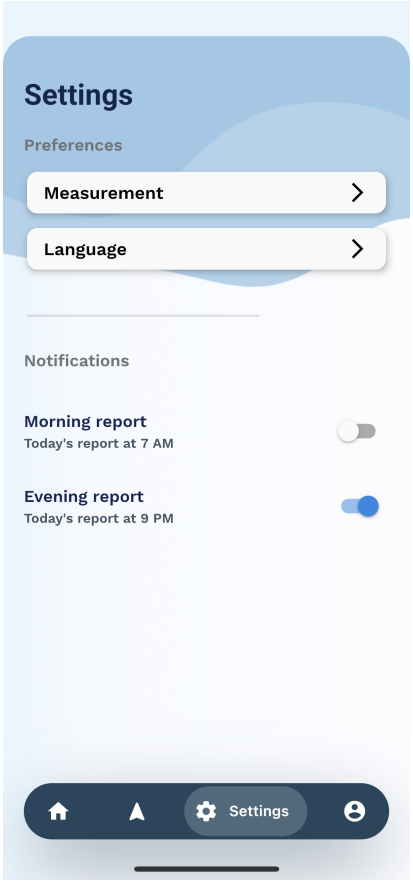

Figure 4.6: Settings screen

Finally, the last screenshot of this series shows the Profile screen. This page is divided into three parts: the upper section dedicated to favorite sensors, the middle one for managing account information and the bottom one for logging out. The section of the favorite sensors currently only acts as a reminder of those sensor whose card in figure 4.4 on page 59 features a gold star button indicating that it has been saved between the favorite ones (in the local storage); thus, these widgets cannot be pressed in order open their card in the Map section, even though this might be a future implementation. This section is horizontally scrollable when more than 2 sensors are present. The middle section summarizes the account information and offers the possibility of modifying profile-related data by pressing the edit button on the right. Lastly, the logout button, painted in red because it suggests a "dangerous" action, when pressed asks confirmation to the user on her will to log out from the account: this is a fundamental cornerstone of User Experience (UX) since the user, who might accidentally push the button, should be abruptly moved to the Login page without expressing her will to do so.

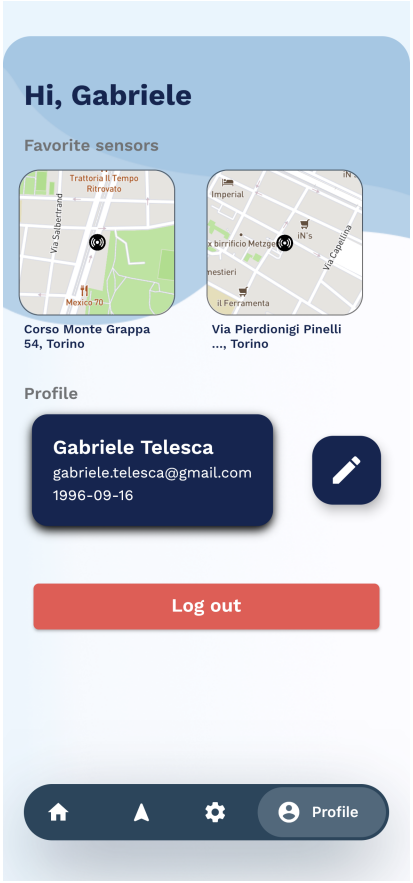

Figure 4.7: Profile screen

In conclusion, a few words to highlight the main aspects and weaknesses of the overall User Interface (UI). Clearly this result is not perfect, as just a few testers had the possibility to provide a feedback about the UI, so for sure issues would naturally emerge; moreover, as explained in the previous paragraph, having the user move between Profile and Map screens in order to retrieve her favorite sensors overloads user's memory, which

is an undesired behaviour; finally, the conflict between the need of speaking the users' language and the necessity of conforming to official labels for communicating the degree of air pollution might be resolved by introducing a documentation in the form of Frequently Asked Questions (FAQ). On the other hand, overall the application is consistent, offers good visibility of system's status so that the user understands when something is loading, prevents errors, keeps the design as minimalist as possible and always offers an "emergency exit" to leave an unwanted action.

4.2 System's business logic

The reason why this mobile application requires a business logic, powered by specific business rules that will be told in this section, is that, for what concerns the exposed REST APIs, the server actually acts as a middleware between the client and the database by offering endpoints that trigger trivial SQL queries on the database itself and replies with the result of the query. This mobile application needs to present data in the most efficient way and thus it needs to summarize the measures: this is a remarkable weakness of the server, which conversely should provide APIs towards a resource that represents an appropriate average of the measures to be computed (even more efficiently) using proper SQL syntax. Though, since the server doesn't provide these services, it must be client's duty to send multiple requests and then elaborate all the data to efficiently show the user aggregate information. An example of this logic is reported in the code below: this code snippet is taken from the *WeekCubit* class, which has the duty to provide weekly averages of the measures collected by a specific sensor having its own *boardID*; since the server doesn't provide an API for averaging the measures taken by a monitoring device over the past seven days, the client is forced to send seven different requests and then to average the values on its own. For what concerns the code syntax, further details about *Cubits*, *Repositories* and *async* methods will be explained in sections 4.3.1.2 and 4.3.1.3.

```
Future<void> retrieveWeekData({@required int boardID}) async {
    ...

    for (int i = 0; i < 7; i++) {
      StreamController<MeasureOnTime> streamController =
          StreamController<MeasureOnTime>();
      final measureOnTimeStream = streamController.stream;

      ...

      await measureRepository.getMeasuresForDay(
          date: DateTime.now().subtract(Duration(days: i))
          streamController: streamController,
          boardID: boardID);
    }
}
```

Moreover, the mobile application applies reasonable business rules for improving the User Experience (UX) and for correctly managing the connection with the monitoring devices. These aspects will be analyzed in the following sections.

4.2.1 Air Quality Index (AQI)

Providing an Air Quality Index (AQI) together with its textual and visual representation in the form of a label and a colour is extremely important because the user is shown a clear and understandable evaluation on the current air quality conditions with a language that is close to her own. The computation of the AQI is carried out by following the Reff algorithm [7] (see section 1.2.4), as demonstrated in the code snippets below.

```
double computeNowCastConcentration(
      {@required Type type, @required DateTime date}) {
    var c = Map<int, double>();
    ...

    var hourlyMeasuresMap = groupBy(
        measures[date].where((measure) => measure.type == type),
        (MeasureOnTime measure) => measure.date.hour);

    for (int hour in hourlyMeasuresMap.keys) {
      c[hour] = hourlyMeasuresMap[hour]
              .map((measure) => measure.value)
              .reduce((a, b) => a + b) / hourlyMeasuresMap[hour].length;
    }

    return applyReffAlgorythm(c: c);
}

double applyReffAlgorythm({@required Map<int, double> c}) {
    double cMin = c.values.reduce(min);
    double cMax = c.values.reduce(max);
    double wStar = cMin / cMax;
    double w = (wStar > 0.5 ? wStar : 0.5);
    double numerator = 0;
    double denominator = 0;

    for (int i = 23, j = 1; i >= 0; i--) {
      if (c[i] != null) {
        numerator += pow(w, j - 1) * c[i];
        denominator += pow(w, j - 1);
        j++;
      }
      if (j > 12) break;
    }

    return numerator / denominator;
}
```

Finally, all the values shown in the Home page (see figure 4.2 on page 57), including the AQI in the halo circular widget, are computed by considering only the monitoring device which is the closest one to the user: of course, this requires the user to enable localization services while using the app (and set "precise position" on iOS devices). This might look like a trivial operation, but actually it leverages the distributed nature of the system to provide localized information rather than a trivial average of all the measurements in the city.

4.2.2 Computation of best path based on air quality

Typically problems related to best paths are solved by means of graphs and graph theory. A graph is an Abstract Data Type (ADT) made up of a finite collection of vertices (also known as nodes), as well as a set of unordered or ordered pairs pairs(respectively

for undirected graphs and directed graphs), known as edges, which might be weighted with an edge value representing a cost, a length or anything else. Graph theory studies graphs as mathematical structures in order to solve problems including the shortest path: this problem is based on finding that path between all paths connecting two nodes that minimizes the sum of the weights on the involved edges. From computer science literature, the most relevant figures that solved this problem are Edsger Dijkstra, Richard Bellman and Lester Ford, from which the two most used algorithm take their name - Dijkstra's and Bellman-Ford algorithms.

The idea behind this feature is building a graph where the nodes are the crossings of the streets and the edges are the roads that connect the crossings. Such a graph has huge dimensions and cannot be stored in a smartphone's memory, so the best solution is having the server build it, update it and provide APIs to obtain the best path (or even the best N paths) given the coordinates of a starting point and of an end point on a map. The research group is currently elaborating this solution by leveraging the *NetworkX* Python package, which was devised especially for creating and manipulating graphs; though, the mobile application UI is already set up for showing different paths on the map at the same time in order to display the best N paths towards the destination point.

So where is the business logic in this topic? In this case the business logic is deliberately hidden from the user and consists in limiting the possible destinations to be queried via the search bar: indeed, the following code snippets taken from the *RoutingAPI* class show that, before querying the MapBox services for a destination name represented by the *query* String, the application computes the edges of a bounded box (see *bbox* query parameter) having the user position as the center of the box and *upRight* and *downLeft* as the upper right edge and the bottom right edge respectively. In practice, the results of the query are limited to only those destinations that are constrained in a box whose sides are 20 km long. The number of km has been arbitrarily chosen and it could be increased or decreased in the future. This constraint makes sense as there would be absolutely no point in showing the best path for reaching a destination that is more than 10 km away: in fact, given the context of this application, the user is supposed to refer to information provided by the mobile application when she moves on foot or by bicycle because in these scenarios she will be experimenting air pollution risks.

```
Future<List<RawDestination>> searchDestination(String query, String locale,
      int maxResults, LatLng currentCoordinates) async {
    LatLng upRight = LatLng(
        _computeLatitude(currentCoordinates.latitude, true),
        _computeLongitude(
            currentCoordinates.latitude, currentCoordinates.longitude, true));
    LatLng downLeft = LatLng(
        _computeLatitude(currentCoordinates.latitude, false),
        _computeLongitude(
            currentCoordinates.latitude, currentCoordinates.longitude, false));

    try {
      Response<Map<String, dynamic>> response =
          await dio.get<Map<String, dynamic>>(
              'https://api.mapbox.com/geocoding/v5/mapbox.places/$query.json',
              queryParameters: {
                ...
                'bbox':
                    '${downLeft.longitude},${downLeft.latitude},
                     ${upRight.longitude},${upRight.latitude}',
                ...
```

```
            },
            options: Options(...));
    }
    ...
}

double _computeLatitude(double latitude, bool up) {
    int earthRadius = 6378137;
    int boundedBoxLimitInMeters = 10000; // 10 km from current position
    double latitudeDifference =
        (180 / pi) * (boundedBoxLimitInMeters / earthRadius);

    return up
        ? (latitude + latitudeDifference)
        : (latitude - latitudeDifference);
}

double _computeLongitude(double latitude, double longitude, bool up) {
    int earthRadius = 6378137;
    int boundedBoxLimitInMeters = 10000; // 10 km from current position
    double latitudeInRadians = pi / 180.0 * latitude;
    double longitudeDifference = (180.0 / pi) *
        (boundedBoxLimitInMeters / earthRadius) /
        cos(latitudeInRadians);

    return up
    ? (longitude + longitudeDifference)
    : (longitude - longitudeDifference);
}
```

4.2.3 Bluetooth Low Energy connection

At the core of the whole project stands the feature that connects each monitoring device with the user's smartphone. As already mentioned in section 2.3.3, the technology choice to support this communication is the Bluetooth Low Energy (BLE); nowadays every smartphone on the market supports Bluetooth 5.0 and the first release of BLE on the market dates back to 9 years ago, so that it is possible to assume that no modern smartphone owner will be excluded from the participative activity of the project.

Despite serving such an important function for the architecture of the system, for the time being the user is completely unaware that its smartphone might be receiving data via BLE or might be sending the measurements just received to the remote server: indeed, as this feature is not something which the user can interact with, she doesn't need to know about its existence. Nonetheless, if the user is not aware of being part of a network and of acting as a gateway for the monitoring devices, she may not be encouraged to run the application when close to a monitoring device: this is an evident weakness of the system, as BLE is a short-range wireless communications technology. A possible countermeasure to this problem, as explained in the last chapter, requires to make the user aware of the situation and to involve her by means of "rewards".

Before entering into details on the pairing and the transfer of information mechanisms by looking at the code, it is fundamental to introduce the concepts of *Service* and *Characteristic* in the context of the Generic Attribute Profile (GATT).

GATT consists of a set of definitions used for describing how two BLE devices transfer profile and user data; in particular, it defines a generic data protocol, called Attribute Protocol (ATT), where *Services*, *Characteristics* and their data are stored and retrieved by means of a lookup-table based on 16 bits long IDs called *handles*. This protocol, which

plays an important role after the connection has been established, is based on exclusive connections: indeed, a *GATT Client* (or *GATT Peripheral*) cannot be connected to more than one *GATT Server* at the same time, but it needs to wait for the current connection to be destroyed; moreover, as soon as the connection has been established, the *GATT Client* ceases to advertise itself and other devices are unable to detect it or connect to it. All standard BLE profiles must comply with GATT-based profiles, which define an association between *GATT Services*' UUIDs - unique identifiers - and the corresponding use case for the sake of interoperability between devices realized by different vendors: for instance, all the devices that implement GATT-based profile know that the *Service* 0x180F is reserved for battery information and that it includes the *Characteristic* 0x2A19 for indicating the battery percentage.

Any *GATT Client* is capable of sending requests to a *GATT Server*, which in turn sends responses. At the very beginning of the connection, the *GATT Client* knows nothing about the attributes of the counter part, so it needs to perform a service discovery; after that, the *GATT Client* can decide to read and write these attributes if it is allowed to do so. The *GATT Server* is in charge of creating, modifying and keeping user data available to the *GATT Client* in the form of *Characteristics*. The role of *GATT Server* can be played by any BLE device on the market, even when it doesn't need to send data to any other device via BLE.

In order to distinguish every *Characteristic*'s or *Service*'s scope, GATT-based profiles make use of Universally Unique Identifiers (UUIDs), a 16 bytes long number which has high probability to be unique on a global scale. For efficiency, and because 16 bytes would take a large chunk of the 27-byte data payload length of the Link Layer, the BLE specification adds two additional UUID formats: 16-bit and 32-bit UUIDs. These shortened formats can be used only with UUIDs that are defined in the Bluetooth specification (i.e., that are listed by the Bluetooth SIG as standard Bluetooth UUIDs) [13]. To reconstruct the full 128-bit UUID from the shortened version, insert the 16- or 32-bit short value as prefix of the Base UUID: -0000-1000-8000-00805F9B34FB. Since the application realized for this thesis work doesn't comply with any SIG use case, it uses a custom 32 bit prefix for both *Services* and *Characteristics* as suggested by the SIG itself.

The fundamental information unit defined by the ATT protocol is the attribute. This little piece of information, which can be easily addressed thanks to its *handle*, may store important user data regarding the *GATT Server*'s partitioning of the various *Characteristics*. Every attribute is format-independent: indeed, there is no recommended structure to use for storing user data; moreover, it is partitioned into the following fields:

- the *handle* field is a 16 bits long section that is used to make the attribute addressable via the *GATT Server*'s lookup-table
- the *type* field is a UUID that defines the kind of data stored in the *value* field
- the *permissions* field specifies which operations can be carried out on the attribute itself and which security requirements must be guaranteed:
  - the *access permissions* sub-field states whether the attribute value can be read and/or written by the *GATT Client*

  - the *encryption* sub-field determines the encryption level necessary to the *GATT Client* for reading the attribute - no encryption, encryption but encryption keys don't need authentication, encryption with authenticated keys
  - the *authorization* sub-field specifies if the *GATT Client* needs to authenticate to access the attribute

- the *value* field stores the user data of the attribute and can be at maximum 512 bytes long.

The attributes in a GATT server are grouped into *Services*, each of which can contain zero or more *Characteristics*. These characteristics, in turn, can include zero or more *Descriptors*. This hierarchy is strictly enforced for any device claiming GATT compatibility (essentially, all BLE devices sold), which means that all attributes in a GATT server are included in one of these three categories, with no exceptions [13]. The figure 4.8 on page 68 shows an example of *GATT* hierarchy based on an heart rate *Service*.

There are two types of *Services*, the primary ones and the secondary ones: while a primary service is a *GATT Service* dedicated to standard functionalities, a secondary service is usually part of a primary service and acts as a modifier; so far, secondary services are rarely used. A *GATT Characteristic* contains user data: indeed, *GATT Clients* can refer to its definition to understand the properties of the *Characteristic* and of course they can actually read and/or write user data according to *Characteristic*'s permissions (e.g. read, write, broadcast). In particular, the $Notify$ property, which will be used in this thesis work on the monitoring devices as described in section 5.1.2, offers the *GATT Client* a way to subscribe to new incoming data provided by the *GATT Server*; moreover, this property is similar to the *Indication* one, which unlike the $Notify$ requires the *GATT Client* to send an acknowledgment message every time a new message is received before sending the next one. It is possible to relate *Indication* to TCP packets and $Notification$ to UDP packets, although they belong to completely different technological stacks. Additionally, *Descriptors* can be added to the *Characteristic* value to expand on the metadata provided in the *Characteristic*'s declaration: thus, the *Characteristic* definition is made up of declaration, value, and any descriptors.

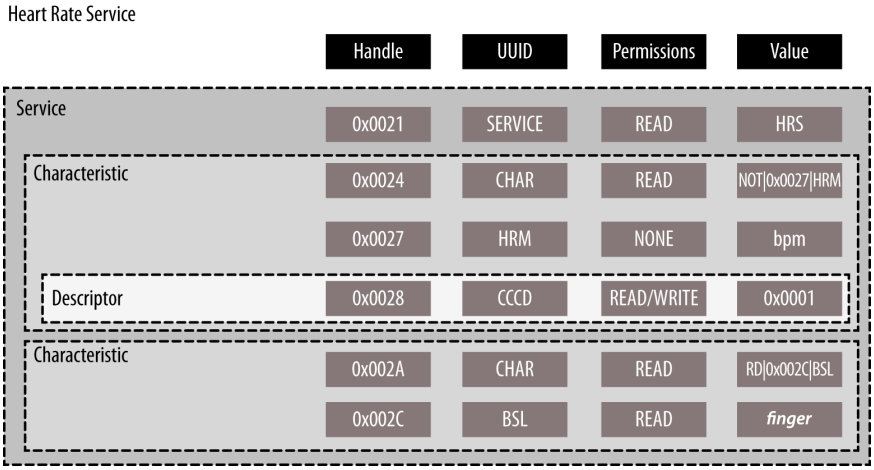

Figure 4.8: GATT hierarchy example for an heart rate service [13]

In the scenario of this thesis work, the role of the *GATT Client* is played by the user's smartphone, which starts the pairing process for connecting to the *GATT Server*, represented by each monitoring device and in particular by the FiPy development board (see section 2.3.2). Let us move on with the code analysis related to the mobile application's BLE connection: all the code snippets are taken from the *BoardConnectionBloc* class, which leverages the *flutter_blue* package to provide BLE functionalities.

The following code snippet creates a periodic activity that will be carried out every *_period* time units (in this case 5 minutes).

```
void _startPeriodicBluetoothScan() {
    ...
    _bluetoothScanTimer = Timer.periodic(_period, (Timer t) {
      _startBluetoothScan();
    });
}
```

The activity to perform every 5 minutes is a Bluetooth scan, which will last a few seconds less than 5 minutes in order to stop the scan before triggering the next (periodic) call to this method. Every time a new device is found during the scan, the name of this device is compared with the default name associated to each FiPy, which is *weather − station*: if the two of them correspond, then it is the moment to connect to it and to subscribe to its characteristic's values because it will progressively contain new measures to be read.

```
Future<void> _startBluetoothScan() async {
    _bluetooothScanSubscription?.cancel();
    _bluetooothScanSubscription = _flutterBlue
        .scan(timeout: _bluetoothScanDuration)
        .listen((ScanResult scanResult) async {
            BluetoothDevice device = scanResult.device;
            if (device.name == _boardBluetoothName) {
                await _connectViaBluetooth(device);
                await _prepareMeasureNotificationViaBLE(device: device);
            }
    });
}
```

Connecting via Bluetooth doesn't mean only to trivially call the *connect*() method, but it also includes listening to the connection state in order to react to device disconnection: indeed, when the *GATT Server* disconnects either because all measures were successfully received or because of some connectivity issues (e.g. user's smartphone too far from the monitoring device), the application will start a new 5 minutes countdown to start the next scan and, in case the smartphone is connected to the Internet, it will send the measures received via BLE to the remote server. The side effect of sending the measures to the remote server is the deletion of a local database's table that temporarily stores the measures received as long as the mobile application is not connected to the Internet.

```
Future<void> _connectViaBluetooth(BluetoothDevice device) async {
    await device.connect(timeout: null, autoConnect: false);
    _bluetoothScanTimer.cancel(); // cancel periodic scanning activity
    device.state.listen((BluetoothDeviceState bluetoothState) async {
    ...
        if (bluetoothState == BluetoothDeviceState.disconnected &&
            bluetoothState != lastBluetoothState) { // all measures were received
                _bluetoothScanTimer.cancel(); // cancel periodic scanning activity
                // start timer before resuming next bluetooth scan period
```

```
                _timeoutBeforeNextBluetoothScan = Timer(const Duration(minutes: 5), () {
                    _startPeriodicBluetoothScan();
                });
                if (lastInternetState is ConnectedToInternetState) {
                    await measureRepository.sendMeasures(measures: measures.toList());
                    await databaseRepository.deleteMeasureTable();
                } else {
                    await _saveMeasuresInsideDatabase(measures.toList());
                }
            }
            lastBluetoothState = bluetoothState;
            ...
        });
}
```

Now it is time to subscribe to value changes for the characteristic that will progressively hold all the measures to be sent. The first step is performing a service discovery; then, the application negotiates a new Maximum Transmission Unit (MTU): in case the application is running on Android, the MTU is increased to the same value as iOS, which is 197 bytes (the FiPy can handle at maximum 200 bytes for each characteristic value); after that, the application looks for the characteristic with a specific UUID, deliberately chosen for sending the measures, inside the service's characteristics. When the UUID corresponds, the application subscribes to its changes: the first change will be the first read, which will be an empty string; the second change will contain the number of measures that the FiPy is about to send; the next changes will actually contain the measures. When all measures have been received, then the application disconnects from the FiPy and thus triggers the sending of the received measures.

```
Future<void> _prepareMeasureNotificationViaBLE(
      {@required BluetoothDevice device}) async {
    List<BluetoothService> services = await device.discoverServices();
    ...

    if (Platform.isAndroid) await device.requestMtu(197); // same as iOS

    BluetoothCharacteristic measureCharacteristic =
        _discoverMeasureCharacteristic(services);
    int read = 0;
    int total = 0;

    await measureCharacteristic.setNotifyValue(true);
    measuresNotificationSubscription =
        measureCharacteristic.value.listen((List<int> measures) async {
      if (read == 1) { // first message is total number of measures to be sent
        read++;
        total = int.parse(String.fromCharCodes(measures));
      } else if (read <= 1) { // skip first value (empty)
        read++;
      } else {
        List<MeasureFromBoard> measuresFromBoard =
            _convertStringToMeasureFromBoard(measures);
        measuresFromBoard.forEach((MeasureFromBoard measureFromBoard) async {
          this.measures.add(measureFromBoard);
          read++;
          if (read >= total) { // all measures received
            measuresNotificationSubscription.cancel(); // cancel subscription
            await device.disconnect();
          }
        });
      }
    });
}
```

In order to decrease the transfer time, not only *Notification* property has been preferred to *Indication* property, but also it was decided to enqueue 7 measures in each single message in order to leverage most of the 197 bytes of the MTU. As a consequence, for each message received, in which every measure is separated from the others by means of a '|' (pipe) character, the application must extract every measure's data and set them as property of as many objects temporarily stored in memory: in fact, after Bluetooth disconnection, all these objects will be either stored in a local database or sent to the remote server.

```
List<MeasureFromBoard> _convertStringToMeasureFromBoard(List<int> measure) {
    List<MeasureFromBoard> measures = [];
    List<String> boardMeasures = String.fromCharCodes(measure).split('|');

    boardMeasures.forEach((String boardMeasure) {
      if (boardMeasure.isNotEmpty) {
        List<String> measureFields = boardMeasure.split(',');
        measures.add(MeasureFromBoard.fromMap(<String, dynamic>{
          columnId: measureId++,
          columnTimestamp: measureFields.elementAt(0),
          columnSensorId: int.parse(measureFields.elementAt(1)),
          columnData: double.parse(measureFields.elementAt(2)),
          columnGeohash: currentGeohash
        }));
      }
    });

    return measures;
}
```

4.2.4 Local notifications

Broadly speaking, notifications can be divided into two main categories, push notifications - also referenced to as remote notifications - and local notifications. While the mobile application may plan local notifications, which can be triggered by time, date or location, a backend (proprietary or via cloud services) is responsible for sending push notifications. This kind of feature requires the user to accept the request of sending notifications from the application. It is important not to exaggerate with the number of scheduled notifications, otherwise the user would get annoyed with this behaviour.

This thesis project makes use of the *flutter_local_notifications* package to implement local notifications on both Android and iOS; the main point behind this feature is to increase user's engagement over time; indeed, it has been proved that push and local notifications, if not misused, can improve app engagement by 88% [25]. Moreover, when the user taps on a notification on her smartphone, she expects the application to open and to perform some actions: in this case, the app will open and will show the Home page (see figure 4.2 on page 57), which summarizes the most recent data based on user's current position.

The first step creates an instance of the plugin class and properly initializes it.

```
Future<void> initialize() async {
    _flutterLocalNotificationsPlugin = FlutterLocalNotificationsPlugin();
    await _configureLocalTimeZone();
    final NotificationAppLaunchDetails notificationAppLaunchDetails =
        await _flutterLocalNotificationsPlugin.getNotificationAppLaunchDetails();
    ...
    const AndroidInitializationSettings initializationSettingsAndroid =
```

```
        AndroidInitializationSettings('app_icon');
    final IOSInitializationSettings initializationSettingsIOS =
        IOSInitializationSettings(
            requestAlertPermission: true,
            requestBadgePermission: true,
            requestSoundPermission: true,
            onDidReceiveLocalNotification:
                (int id, String title, String body, String payload) async {
              Modular.to.pushNamed('/home');
            });
    final InitializationSettings initializationSettings =
        InitializationSettings(
            android: initializationSettingsAndroid,
            iOS: initializationSettingsIOS);
    await _flutterLocalNotificationsPlugin.initialize(initializationSettings,
        onSelectNotification: (String payload) async {
      Modular.to.pushNamed('/home');
    });
}
```

Since the notifications will be triggered based on the local time zone, the application needs to get this information from the underlying operating system. Despite the always expanding set of packages provided by Flutter team and Flutter's community, there is no package for retrieving *TimeZoneNames*, so this is a good chance to explain how to use *MethodChannels*: indeed, they are the only way to pass a message between the client and the host platform in order to call platform-specific APIs, as shown in figure 3.7 on page 44. The following code snippet exploits the *MethodChannel* class to create a channel named *polito_weather_station_app*, and then it invokes a platform-side method called *getTimeZoneName* in order to retrieve a *String*.

```
import 'package:timezone/data/latest.dart' as tz;
import 'package:timezone/timezone.dart' as tz;

class NotificationsHandler {
    ...
    MethodChannel _platform = MethodChannel('polito_weather_station_app');

    ...

    Future<void> _configureLocalTimeZone() async {
        tz.initializeTimeZones();
        final String timeZoneName = await _platform.invokeMethod('getTimeZoneName');
        tz.setLocalLocation(tz.getLocation(timeZoneName));
    }
}
```

The following code is the *MainActivity.kt* Kotlin file located in the main source code folder of the Android application. Part of this file was already written by the framework at the moment of project creation, but now it is time to add the other half of the channel for retrieving the local time zone using code written in Kotlin for calling Android APIs. In particular, the following code states that when a *getTimeZoneName*() method is called using a channel called *polito_weather_station_app* then the TimeZone API is called to return the desired response.

```
class MainActivity: FlutterActivity() {
    override fun configureFlutterEngine(flutterEngine: FlutterEngine) {
        super.configureFlutterEngine(flutterEngine)

        MethodChannel(flutterEngine.dartExecutor.binaryMessenger,
            "polito_weather_station_app").setMethodCallHandler { call, result ->
```

```
            if ("getTimeZoneName" == call.method) {
                result.success(TimeZone.getDefault().id)
            }
        }
    }
}
```

The equivalent code for iOS Swift is reported below.

```
@UIApplicationMain
@objc class AppDelegate: FlutterAppDelegate {
  override func application(
    _ application: UIApplication,
    didFinishLaunchingWithOptions launchOptions: [UIApplication.LaunchOptionsKey: Any]?
  ) -> Bool {

    if #available(iOS 10.0, *) {
        UNUserNotificationCenter.current().delegate = self as UNUserNotificationCenterDelegate
    }

    let controller : FlutterViewController = window?.rootViewController as! FlutterViewController
    let channel = FlutterMethodChannel(name: "polito_weather_station_app",
                                       binaryMessenger: controller.binaryMessenger)
    channel.setMethodCallHandler({
        (call: FlutterMethodCall, result: @escaping FlutterResult) -> Void in
        if ("getTimeZoneName" == call.method) {
            result(TimeZone.current.identifier)
        }
    })

    ...
  }
}
```

Finally, since the user can decide whether to receive a local notification at 7 AM, at 9 PM, both or none at all, when one of the switches in the Settings page (see figure 4.6 on page 61) is activated or deactivated, the following code is triggered so that only the appropriate notifications are scheduled.

```
Future<void> resetNotificationTime() async {
    await _flutterLocalNotificationsPlugin.cancelAll();
    await _computeNotificationTime();

    switch (_notificationTime) {
      case NotificationTime.Both:
        _scheduleDailySevenAMNotification();
        _scheduleDailyNinePMNotification();
        break;
      case NotificationTime.SevenAM:
        _scheduleDailySevenAMNotification();
        break;
      case NotificationTime.NinePM:
        _scheduleDailyNinePMNotification();
        break;
      default:
        break;
    }
}

Future<void> _computeNotificationTime() async {
    _notificationTime = NotificationTime.None;

    String morningReport = await FlutterSecureStorage().read(key: 'Morning report');
    String eveningReport = await FlutterSecureStorage().read(key: 'Evening report');
    if (morningReport == 'true' && eveningReport == 'true') {
      _notificationTime = NotificationTime.Both;
```

```
    } else if (morningReport == 'true') {
      _notificationTime = NotificationTime.SevenAM;
    } else if (eveningReport == 'true') {
      _notificationTime = NotificationTime.NinePM;
    }
  }
```

In particular, the code for scheduling the notification at 9 PM is reported below. Notice that both title and body of the notification are localized to the language selected by the user.

```
Future<void> _scheduleDailyNinePMNotification() async {
    await _flutterLocalNotificationsPlugin.zonedSchedule(
        0,
        'notification.daily9PM.title'.tr(),
        'notification.daily9PM.body'.tr(),
        _nextInstanceOfNinePM(),
        const NotificationDetails(
            android: AndroidNotificationDetails('daily 9 PM',
                'polito_weather_station_app', 'daily 9 PM description',
                importance: Importance.high, priority: Priority.high),
            iOS: IOSNotificationDetails(
              badgeNumber: 1,
            )),
        androidAllowWhileIdle: true,
        ...);
}

tz.TZDateTime _nextInstanceOfNinePM() {
    final tz.TZDateTime now = tz.TZDateTime.now(tz.local);
    tz.TZDateTime scheduledDate =
        tz.TZDateTime(tz.local, now.year, now.month, now.day, 21);
    if (scheduledDate.isBefore(now)) {
      scheduledDate = scheduledDate.add(const Duration(days: 1));
    }
    return scheduledDate;
}
```

4.3 Software Engineering patterns

This section is meant to show the main software engineering patterns exploited during the development of the mobile application. In particular, it focuses on three main topics: application state management, dependency injection and routing.

4.3.1 Business Logic Component: the MVC pattern

Every time the user interacts with the application, it goes into a different state: whether a new page must be displayed or a flag must be toggled, the application logic receives the event, reacts to it and shows the corresponding changes in the User Interface (UI). State management is one of the biggest challenges when learning Flutter. There is no suggested solution for managing the state of the application: indeed, the framework leaves a free choice between different alternatives, provided by several packages; after reading their documentation, though, it is possible to say that there is no clear evidence of a winner, it just depends on the developer's preferences. Actually, Flutter features a quick way of managing state thanks to $StatefulWidgets$, but when building big and production quality applications managing the state in this way becomes cumbersome.

The package chosen for managing the application state is called *flutter_bloc*, because it allows to implement the well-known Model-View-Controller software engineering pattern quite easily: indeed, *Bloc* (which stands for Business Logic Controller) helps with separating presentation from business logic and provides a suitable way for testing the application's logic. *Bloc* was designed with three core values in mind:

- simple: easy to understand, it can be used by developers with varying skill levels
- powerful: help make amazing and complex applications by composing them of smaller components
- testable: easily tests every aspect of an application

Overall, *Bloc* attempts to make state changes predictable by regulating when a state change can occur and enforcing a single way to change state throughout an entire application [5].

Before analyzing some code snippets, it is critical to introduce how *Bloc* works behind the scenes, and to do so the starting point is the concept of *Stream* in the Dart programming language.
A *Stream* is a sequence of asynchronous data. In order to understand this concept it is useful to think of a pipe containing water which is flowing through it: this comparison bases its meaning on the fact that the water is the asynchronous data, while the pipe is the *Stream*; for instance, this data sequence can include user-generated events or information read from a file. Dart's *Streams* are part of the asynchronous programming topic. A typical way of iterating over the events of a stream is the *asynchronous for loop*, which is just like a traditional for loop but it has the *await* keyword prepended. Whenever a function makes use of the *await* keyword, it needs to be marked with the *async* one before the body of the function and to return a Future¡T¿ object (where T can be any type corresponding to the type of the object returned), as shown in many previous examples. The typical way of testing a stream is having a function "return" elements of the stream: in this case the function is marked with the *async* keyword before the body and returns a Stream¡T¿ (where T can be any type corresponding to the "returned" type); finally, instead of using the *return* keyword, this function uses the *yield* keyword in order to insert an element into the stream. When no more elements need to be pushed into the *Stream*, it is said to be "done", and the code that receives the events is notified about the end of the *Stream*; for example, the $await for loop$ stops when the *Stream* is done. Every *Stream* is capable of delivering error events, even more than once if it is the case: when using the $await for loop$ and an error is encountered as element of the stream, the error is *thrown* by the loop statement, so a $try - catch$ construct is required.
In order to handle stream events, the *Stream* class provides interesting helper methods for performing common operations such as filtering, skipping and grouping elements: this aspect is very similar to Java Streams. Moreover, each *Stream* can belong to one of two types:

- single subscription streams: the most common ones, they handle events by delivering them in the correct order and they can't be listened to more than once, otherwise only the events pushed hereafter will be received (which could make no sense)

- broadcast streams: usually used for tap events and similar ones, these streams send events from the beginning even when they are listened to after the first event has been pushed and they can be listened to more than once.

4.3.1.1 Presentation Layer: UI view

This journey towards the state management topic starts from the View, which is the User Interface (UI) built by using Flutter widgets. In Flutter everything is a widget, and so are all *Bloc* concepts. As already mentioned, Flutter widgets composing the UI are nested in a tree-shaped structure: this concept is fundamental to understand the new classes that are about to be introduced.

Let's suppose that a *Bloc* or *Cubit* object has been created and that it needs to be linked to some of the widgets in the UI in order to react to user events (e.g. tap, double tap, long press), how to do this? There is no need to attach it to each of those widgets, because the *BlocProvider* widget class will come in handy to solve this problem. Basically, the solution provided by this class is based on referencing the closest ancestor widget that corresponds to the one that is needed in that part of the tree. In particular, the *BlocProvider* class is responsible for creating and providing a *Bloc* (or a *Cubit*) to all sub-trees in the widget tree: this mechanism is also called *dependency injection*, because only one instance will be provided to the sub-trees below. *BlocProvider* creates the unique instance of the *Bloc* (or *Cubit*) using a function that accepts a *BuildContext* as parameter: this class represents the context where a specific widget is built, it gives information about its position in the widget tree and it is extremely important because, when the *Bloc* (or *Cubit*) must be retrieved, the research will be based on the *BuildContext*. Thus, provided that it is a widget in the sub-tree to need the reference of the *Bloc* (or *Cubit*) created by the above *BlocProvider*, it's enough to write $BlocProvider.of < BlocName > (context)$ or $context.bloc < BlocName > ()$ to retrieve the required object. Moreover, by default the *Bloc* (or *Cubit*) object will be lazily created: this means that the function creating it will be executed when the object itself will be needed for the first time; though, the *lazy* parameter of *BlocProvider* constructor can be set to false to avoid this behaviour. Finally, an important concept to keep in mind when operating with *BuildContext* objects: when a new page is pushed into the screen, actually it does not replace the widget tree of the previous tree with the one of the new page, instead the new widget tree is attached to the widget that triggered the navigation change; this might suggest that a *Bloc* or *Cubit* object is still available in the new widget tree, but it is not because a new *BuildContext* object is governing the new route and therefore the new widget tree. In order to avoid this issue, it is possible to use the $BlocProvider.value()$ static function so that the *Bloc* (or *Cubit*) will be available in the new portion of the tree; as a consequence, when the new page will be closed due to pressing the back button, still the object will be ready to be used in the old page.

This mechanism might get cumbersome, especially at the first experiments: hence, section 4.3.2 will explain how to overcome these issues by introducing a new mechanism for managing dependency injection.

If *BlocProvider* introduces a way to create a *Bloc* (or a *Cubit*) and to "propagate" it in the sub-trees, *BlocBuilder* is the widget responsible for rebuilding the UI on the basis of *Bloc*'s (or *Cubit*'s) state changes: in fact, every time a new state is emitted, the widget

sub-tree contained in this widget is built again in order to reflect the new state in the UI. Though, since rebuilding a large sub-tree may require too much time to compute, a good practice is to wrap that exact part of the UI with the *BlocBuilder* widget. As shown in the following code snippet, taken from the *CustomChip* class, the *BlocBuilder* widget takes some parameters:

- the *builder* function, which takes as parameter the *BuildContext* and the current *BlocState* in order to build the current UI based on this state: this function could be called multiple times - even when there is no state change - due to how Flutter Engine works, hence it must be a pure function, that is a function whose return values depends only on its input
- the *cubit* instance, which is a reference to the cubit whose state will cause the sub-tree to rebuild (see section 4.3.2 to understand how it has been retrieved in the following code snippet)
- optionally, the *buildWhen* function, which decides when the *BlocBuilder* should rebuild the UI based on previous state and current state of the *cubit* instance;

thus, the following code snippet is deciding how to build the circular part of the chip of the Home page (see figure 4.2 on page 57) based on the fact that the data to decide its color is already available or not: indeed, if it is not available, a shimmering animation will signal that the application is loading.

```
BlocBuilder(
    cubit: Modular.get<LatestMeasureCubit>(),
    builder: (BuildContext context, LatestMeasureState state) {
        if (state is DaysMeasureOngoing) {
          return Shimmer.fromColors(
              baseColor: homePMboxColor,
              highlightColor: goodAirQualityColor,
              enabled: true,
              child: Container(
                  width: 30.0,
                  height: 30.0,
                  decoration: new BoxDecoration(
                    shape: BoxShape.circle,
                    color: Colors.white
                  )));
        } else if (state is DaysMeasureAvailable) {
          return Container(
              width: 30.0,
              height: 30.0,
              decoration: new BoxDecoration(
                color: paintAQICircle(
                    AQI: Modular.get<LatestMeasureCubit>()
                        .computeGlobalAQI(
                            date: DateTime.now().subtract(
                                Duration(days: 5 - index)))),
                shape: BoxShape.circle,
              ));
        }

        ...
})
```

The last widget that is worth analyzing is the *BlocListener*. This widget is nearly the same as the *BlocBuilder*, but, unlike its *builder* function which may be called many times,

the corresponding function (called *listener*) is guaranteed to be called only when the state changes. This is very useful whenever the sub-tree is considerably resource-demanding to build or when the reaction to state change is just a side effect and is not UI-related (e.g. navigating to a new screen). The code snippet below uses *BlocListener* to perform a side effect on the map: indeed, after building the map on the UI, the *BlocListener* will add all the boards on the map only when all the coordinates have been retrieved from the server. For fine-grained controls, the *BlocListener* constructor can accept also a *listenWhen* function to decide whether to rebuild the UI on the basis of previous state and current state.

```
BlocListener(
    cubit: Modular.get<MapCubit>(),
    listener: (BuildContext context, MapState state) {
        if (state is CoordinatesAvailable) {
            mapUtils?.addBoardsToMap(); // adds elements inside markers
        }
    },
    child: Builder(
        builder: (BuildContext context) {
          if (markers.length == 0) {
            return ShimmerMap(); // mock of a real map
          }
          return Stack(
            children: [
                MapBoxMap(
                  ...
                  markers: markers,
                  ...
            ],
          );
        })
)
```

What if a sub-tree needs to rebuild according to the latest state and at the same time its needs also to perform a side effect? The *flutter_bloc* package has realized a class, called *BlocConsumer*, that mixed a *BlocBuilder* with a *BlocListener*. Thus, this widget has both a *listener* parameter and a *builder* parameter. The following code snippet exploits the benefits of this class when a destination is being typed by the user on the search bar of the Map page (see figure 4.5 on page 60): indeed, as soon as a suggested destination arrives from the *Bloc* (or *Cubit*), the *destinationSuggestions* list (which is a state variable, that is an attribute of a *StatefulWidget*) is emptied and then filled with the new suggestions so that the builder function can update the UI and show the new elements of the list.

```
class _ReactiveSearchBarState extends State<ReactiveSearchBar> {
  ...

  @override
  Widget build(BuildContext context) {
    return SafeArea(
      child: BlocConsumer(
        cubit: Modular.get<RoutingBloc>(),
        listener: (BuildContext context, RoutingState routingState) {
          if (routingState is SuggestedDestinationState) {
            destinationSuggestions.clear();
            routingState.destinations.forEach((Destination destination) {
              destinationSuggestions.add(ListTile(
                title: Text(destination.placeName),
                subtitle: Text(destination.placeDetails),
                onTap: () => destinationTapHandler(destination),
              ));
```

```
            });
          }
        },
        builder: (BuildContext context, RoutingState routingState) =>
            SearchBar(
                currentPosition: widget.currentPosition,
                floatingSearchBarController: floatingSearchBarController,
                destinationSuggestions: destinationSuggestions,
                destinationMarker: widget.destinationMarker,
                destinationPlaceName: destinationPlaceName,
                statefulMapController: widget.statefulMapController)
      ));
  }
}
```

4.3.1.2 Business Logic Layer: Cubit and Bloc

Given this background about *Streams*, let's introduce the *Cubit* and *Bloc* classes. In Flutter, the best way to keep code organized, clean and maintainable is to use *components*. A *component* mediates between the User Interface (UI) - or Presentation Layer -, and the logic behind: in this case we call this *component* as a Business Logic Controller or *Bloc*. Broadly speaking, *Bloc* is a design pattern developed by Google in order to help separate the Business Logic Layer from the Presentation Layer, allowing developers to easily reuse code; then, the third-party package previously mentioned, called $flutter_bloc$, was published to provide a way to apply this design pattern with Flutter.

First of all, the only difference between a *Cubit* and a *Bloc* is that *Cubit* is a minimal version of a *Bloc*, which indeed extends *Cubit*. A *Cubit* is a special type of *Stream* component: in fact, it is based on both functions to be called from the UI and states to be emitted on a *stream*, as shown in figure 4.9 on page 80. Let's have a looks at the following code snippet, reporting the code of the *LoginCubit* and the *LoginState* classes, in order to understand how it works. This cubit is responsible for receiving user-generated events by means of function calls and for updating the UI by emitting states based on the business logic working behind the scenes. In particular, the $loginUser()$ function is executed when the user expresses the will to sign in by interacting with the proper widget; this function performs the tasks necessary to the login of the user, and on the basis of the output it emits the corresponding state: indeed, if for any reason the login doesn't succeed and produces an *Exception*, the *LoginErrorState* will be emitted and consequently the UI will be updated for guaranteeing the best User Experience (UX). As soon as the *LoginCubit* is instantiated, it must know what is its own current state, so it must be specified in the constructor by calling the $super()$ constructor and passing the starting state as parameter. If the login procedure performed on the server succeeds, then a *LoggedInState* state will be emitted into a stream and the UI will properly rebuild. For the time being, it is not important to understand what is the *userRepository* object as it will be addressed in the next section.

```
part 'login_state.dart';

class LoginCubit extends Cubit<LoginState> {
  LoginCubit({@required this.userRepository}) : super(UnknownLoginState());

  UserRepositoryInterface userRepository;

  Future <void> loginUser({@required String email, @required String password}) async {
    try {
```

```
      Message msg = await userRepository.loginUser(email: email, password: password);

      emit( LoggedInState(token: msg.message) );
    }
    on Exception {
      emit(LoginErrorState());
    }
  }

  ...
}

part of 'login_cubit.dart';

@immutable
abstract class LoginState {}

class UnknownLoginState implements LoginState {}

class LoggedInState implements LoginState {
  final String token;

  LoggedInState({@required this.token});
}

class LoginErrorState implements LoginState {}
```

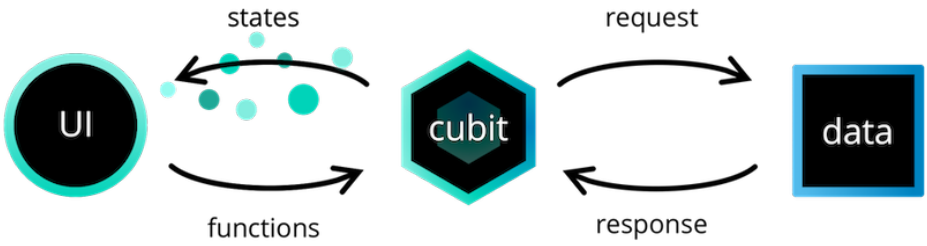

Figure 4.9: Bloc architecture: cubit

Cubits are rather simple controllers which usually turn out to be enough for most of the cases; though, in some scenarios *Cubits* become cumbersome. It has already been said that *Bloc* is a super-set of *Cubit*, so now it is time to see what more it is capable of providing. *Cubit* is a component based on functions that are not part of a stream and states that, instead, are part of a stream; conversely, *Bloc* not only can emit states that belong to a stream, but it also receives a stream of events, as shown in figure 4.10 on page 82. Thus, depending on the kind of interaction of the user, an event will be inserted in a stream which the *Bloc* object is already listening to, so that it will be able to receive the event and convert it into a state to be pushed into a stream for having the UI rebuild.
The perfect example to show how *Blocs* work is the following code, taken from the *RoutingBloc* class. One of the duties of this class is reacting to user's input on the search bar in the upper part of the screen of the Map page (see figure 4.5 on page 60) by querying the MapBox API to retrieve places and streets including the *query* string in their name. Why is this a good example of the scenario where *Bloc* is necessary? Because the user will generate a stream of events, each one after the user edits the input text on the search bar (with some debounce time); conversely, the registration of a user usually starts after clicking a button, which is an action that is usually performed only once. Notice that inside a *Bloc* the main function is the *mapEventToState*() function: this

function receives an element of the stream of events and finally emits a state according to some computation in between; for what concerns this *Bloc*, the state will contain the list of possible destinations received from the MapBox API.

```
class RoutingBloc extends Bloc<RoutingEvent, RoutingState> {
  RoutingBloc({@required this.routingRepository}) : super(RoutingInitial());

  final RoutingRepositoryInterface routingRepository;

  @override
  Stream<RoutingState> mapEventToState(RoutingEvent event) async* {
    switch (event.runtimeType) {
      case TypingDestinationEvent:
        TypingDestinationEvent typingDestinationEvent =
            event as TypingDestinationEvent;
        List<Destination> destinations =
            await routingRepository.searchDestination(
                query: typingDestinationEvent.query,
                currentCoordinates: typingDestinationEvent.currentCoordinates);
        yield SuggestedDestinationState(destinations: destinations);
        break;
      ...
      default:
        break;
    }
  }
}
```

The following classes represent respectively the possible events that the *RoutingBloc* can accept and the possible states that it can have.

```
part of 'routing_bloc.dart';

@immutable
abstract class RoutingEvent {}

class TypingDestinationEvent extends RoutingEvent {
  final String query;
  final LatLng currentCoordinates;

  TypingDestinationEvent({@required this.query, @required this.currentCoordinates});
}

class DestinationChosenEvent extends RoutingEvent {
  final LatLng from;
  final LatLng to;

  DestinationChosenEvent({@required this.from, @required this.to});
}
```

```
part of 'routing_bloc.dart';

@immutable
abstract class RoutingState {}

class RoutingInitial extends RoutingState {}

class SuggestedDestinationState extends RoutingState {
  final List<Destination> destinations;

  SuggestedDestinationState({@required this.destinations});
}
```

Basically a *Cubit* or a *Bloc* should be created for every functionality provided by the application: indeed, the previously shown components are related to login and routing on

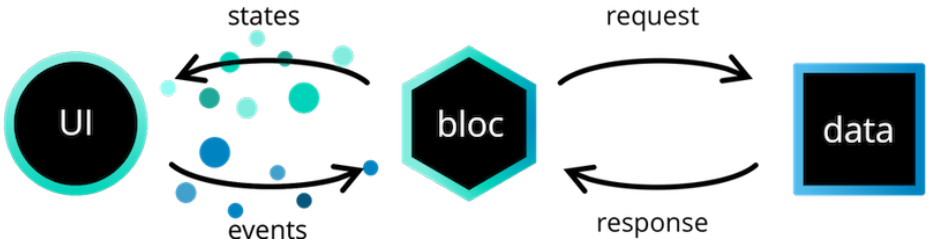

Figure 4.10: Bloc architecture: bloc

a map features respectively. Moreover, this inclination to creating one component for each functionality led the folders management inside the project folder to follow a very precise structure: in practice, the main folder (called *lib*) contains a sub-folder called *features*, which is made of as many folders as the functionalities provided by the app. Usually every feature is also provided by a page, but sometimes it's also a set of pages (e.g. the registration procedure requires more pages to collect user's data). Each of these folders is further divided into two separate folders, called *ui* and *bloc*: as the names suggest, the first one contains the code that represents the UI, while the second one contains the definitions of all the *Blocs* and *Cubits* used inside that UI.

Cubits and *Blocs* don't need to be independent one from the other, because in some cases there is the need to update the UI managed by a *Bloc* (or a *Cubit*) on the basis of the state of another *Bloc*: this is what happens in the following code snippet, where the sending of the measures retrieved from a monitoring device is skipped when there is no Internet connection. In particular, the *BoardConnectionBloc* always knows whether the smartphone is connected to the Internet because it has subscribed to state changes of the *InternetCubit*.

```
final InternetCubit internetCubit;
InternetState lastInternetState;

BoardConnectionBloc(
      {@required this.databaseRepository, @required this.internetCubit,
      @required this.measureRepository}) : super(BoardConnectionInitial()) {
    ...
    internetStreamSubscription = internetCubit.listen((InternetState internetState) async {
        lastInternetState = internetState;
        if (internetState is ConnectedToInternetState) {
            ...
        }
    });
}

Future<void> _connectViaBluetooth(BluetoothDevice device) async {
    await device.connect(timeout: null, autoConnect: false);
    ...
    device.state.listen((BluetoothDeviceState bluetoothState) async {
        if (bluetoothState == BluetoothDeviceState.disconnected &&
            bluetoothState != lastBluetoothState) {
                ...
                if (lastInternetState is ConnectedToInternetState) {
                    ... // send measures to the server
                } else {
                    ... // store measures in local database
                }
                ...
            }
            ...
```

```
    });
  });
}
```

Finally, testing a *Cubit* or a *Bloc* is really straightforward. Indeed, the $bloc_test$ and the $flutter\_test$ packages are very useful for this purpose because they provide a way to test the expect output state of a *Bloc*'s (or *Cubit*'s) method in a very structured fashion. The following code snippet shows some tests run on the $HistoryCubit$ class: notice that not only it is possible to check that the output state is correct via the *expect* list, but the package also allows to verify that side effects on $HistoryCubit$'s attributes have occurred via the $verify$ function.

```
void main() {
  group('test History Cubit', () {
    HistoryCubit historyCubit;

    setUp(() {
      initModule(MapModuleHelper());
      historyCubit = HistoryCubit(measureRepository: Modular.get<MockMeasureRepository>());
    });

    tearDown(() => historyCubit.close());

    ...

    blocTest(
        'the cubit should emit a Last24HoursAvailable state and should save today\'s PM data,
        when retrieveHistoryData() method is called',
        build: () => historyCubit,
        act: (HistoryCubit historyCubit) async {
          await historyCubit.retrieveHistoryData(boardID: 0);
        },
        expect: [isA<Last24HoursAvailable>()],
        verify: (HistoryCubit historyCubit) {
          expect(historyCubit.measures.isNotEmpty, true);
          expect(historyCubit.measures.length, 2);
        });

    blocTest(
        'the cubit should emit a Last24HoursAvailable state, when reloadHistory() method is called',
        build: () => historyCubit,
        act: (HistoryCubit historyCubit) async {
          historyCubit.reloadHistory();
        },
        expect: [isA<Last24HoursAvailable>()]);

    ...
}
```

Further, since the *expect* function is supposed to compare the two objects passed as parameters, when comparing two user-defined objects the developer is required to override the == operator: to do so, the *equatable* package features the possibility to specify which attributes should be compared in order to state whether two user-defined objects are equal. This is the case of the following test, which compares two $User$ objects based on a set of their attributes.

```
blocTest(
    'the cubit should emit a UserAvailable state containing user\'s data,
    when retrieveCurrentUser() method is called',
    build: () => profileCubit,
    act: (ProfileCubit profileCubit) async {
```

```
        await profileCubit.retrieveCurrentUser(),
        User(name: 'Gabriele', surname: 'Telesca',
            email: 'gabriele.telesca@gmail.com',
            birth: '1996-09-16'));
    },
    verify: (ProfileCubit profileCubit) {
      expect(profileCubit.currentUser, User(name: 'Gabriele', surname: 'Telesca',
        email: 'gabriele.telesca@gmail.com', birth: '1996-09-16'));
});

class User extends Equatable {
  String name;
  String surname;
  String email;
  DateTime birth;

  ...

  @override
  List<Object> get props => [name, surname, email, birth];
}
```

4.3.1.3 Model Layer: Repository, Data Provider and Model

This section is about how *Blocs* and *Cubits* process an event. Most applications nowadays are fetching data from the Internet in order to show them on the UI: thus, the need of retrieving data from a remote server introduces the topic of the Data Layer. As indicated by figure 4.9 on page 80, the *Cubit* (or *Bloc*) requests some data to the Data Layer and eventually it will receive a response to be parsed in order to properly update the UI. By adding the Data Layer to the global picture, it is now evident that the Bloc design patter is an application of the more general Model-View-Controller (MVC) software engineering pattern: indeed, the role of the Model is played by the Data Layer, the role of the View by the Presentation Layer, and finally the role of the Controller by the Business Logic Layer implemented by *Blocs* and *Cubits*.

The Data Layer is the furthest layer from the User Interface (UI). Its main duty is to retrieve and manipulate data coming from one or more asynchronous sources of information. For this reason, this layer is further split into three sub-layers:

- the Model sub-layer contains *Models*, which act as a blueprint to the data that this application is going to work with: in practice, each *Model* is a plain Dart class, whose attributes should list the attributes that are going to be used either in the UI or by the business logic; the main concept is that these attributes do not need to be the same as the ones indicated in the response fetched from the Internet, instead they should be completely independent from the source

- the Data Provider sub-layer is responsible for providing raw data to the Repository sub-layer and it actually acts like an API because the methods of a *Data Provider* object act as a communication link towards the data sources on the Internet; every *Data Provider*'s method is strictly dependent on the structure of the data being fetched, and therefore the returned object will likely be different from the *Model*

- the Repository sub-layer is that part of the Data Layer where *Model* objects are instantiated because they need to be accessed by the *Cubit* (or the *Bloc*); the classes used in this sub-layer depend on instances of *Data Providers*, which are called to

retrieve data from the Internet, but then the raw data is converted into a *Model* object that will be available to the *Cubit* (or *Bloc*) of the upper layer: this is the perfect moment for manipulating and fine-tuning data before passing it to the Business Logic Layer.

Being the Business Logic Layer the mediator between the colorful and flawless side of the UI and the unstable and error-prone side of the Internet (e.g. connection might stop working, the API to be queried might be slow), this is the last endpoint where errors can be caught, so it is very likely that *Cubits* and *Blocs* calls to *Repositories*' methods are wrapped inside try-catch clauses.

Moreover, the project folder structure of this thesis work also resembles these sub-layers: indeed, besides the *features* folder inside the *lib* folder that was previously mentioned, a new *data* folder is added; further, this folder contains three sub-folders named *models*, *providers* and *repositories*.

Finally, one of the advantages of the Bloc architecture is the ease in testing *Blocs* and *Cubits* methods. Given that the kind of test is a Unit Test and that the *Bloc* (or *Cubit*) is dependent on the lower layer (in particular to *Repositories*) as explained in the next section, it is fundamental to avoid having the test output influenced by *Repositories*. Thus, for this reason, each *Repository* class in reality implements a *RepositoryInterface* so that for testing purposes its methods, implemented by a mock *Repository* object, return trivial results without sending requests to the other sub-layers. For instance, the following code snippet reports the code of two methods taken from the *MockMeasureRepository* class inside the *test* folder of the project.

```
class MockMeasureRepository implements MeasureRepositoryInterface {
  @override
  Future<Board> getClosestBoard({LocationData position}) async {
    return Board.fromRawBoard(RawBoard.fromJSON({'id': 1, 'sensors': [],
        'temp': '20.0', 'rh': '60.1', 'lat': '45.1', 'long': '7.6'}));
  }

  @override
  Future<void> getMeasuresForDay({DateTime date, StreamController<dynamic> streamController, int boardID}) async {
    streamController.sink.add(MeasureOnTime(
        date: DateTime.now() ,
        value: 25.3,
        type: Type.PM25));

    streamController.sink.add(MeasureOnTime(
        date: DateTime.now() ,
        value: 65.1,
        type: Type.PM10));

    streamController.close();
  }

  ...
}
```

4.3.2 Dependency Injection

All object-oriented applications are based on a set of classes which interact with each other according to some logic: the way an object is linked with other ones builds a set of dependencies; for example, in the last section the dependency between Data Layer's

sub-domains' object and the dependency between *Repositories* and *Blocs* (or *Cubits*) has been highlighted with some code snippets. When the code base becomes larger, it might be stressful and cumbersome to deal with all the dependencies; indeed, this was the case of *BlocProvider* objects which follow a strategy that suits mostly cases where few page navigation actions can happen (see section 4.3.1.1): thus, it would be great to retrieve the correct *Bloc* (or *Cubit*) directly in the UI code without worrying about widget tree mechanisms. So the point of dependency injection is to reduce the static coupling between objects in object-oriented languages, without compromising code quality and maintainability (otherwise it would be enough to write code in just one file). Moreover, since dependencies cannot disappear and they are a solid part of the application structure, the idea behind this software engineering pattern is to move the scope of the dependency from the object to its type: the best way to reach this goal is changing the attribute reference type from the actual class to the interface type that the class implements.
The following code snippet shows an application of these concepts to the dependency between the $LatestMeasureCubit$ class and the $MeasureRepository$ class: in fact, when the *Cubit* needs to be constructed, a third-party entity is responsible for passing a $MeasureRepository$ object that implements the $MeasureRepositoryInterface$. Notice that the $LatestMeasureCubit$ doesn't need to know how a $MeasureRepository$ object is constructed because someone will provide an already built copy of it; further, this is the mechanism that stands behind *Bloc*'s (or *Cubit*'s) unit tests and that allows to substitute the real *Repository* object with a mock one for testing purposes.
Dependency injection derives from a wider technique, called Inversion of Control (IoC): this technique, indeed, consists in substituting the older principle stating that each object directly chooses objects on which it depends with the principle based on the fact that someone else will be providing the objects on which its depends.

```
class LatestMeasureCubit extends Cubit<LatestMeasureState> {
  LatestMeasureCubit({@required this.measureRepository})
      : super(DaysMeasureOngoing());

  MeasureRepositoryInterface measureRepository;
  ...
}
```

How does this work? How are objects injected where needed? The *flutter_modular* package does all the magic work. The application's functionalities will be represented by decoupled and independent modules in a *Modular* application: each module has its own dependencies, routes, widgets and business logic, and it is housed in its own directory; as a result, it becomes straightforward to quickly remove a module from the project and use it elsewhere.

Every mobile application using this package must have a $MainModule$, that is the module that makes the application start. The following code reports the *main*() function (that is also the entrypoint) of the application and part of the *AppModule* class where a list of dependencies to be injected is provided, together with the root widget associated to the main module (usually it is the $MaterialApp$ widget). In this code snippet, the *Bind* object is responsible for configuring the object injection inside the module: thus, the *binds* getter is returning the list of objects that can be accessed at any time provided that this module is not out of scope; indeed, a module becomes out of scope when none of its routes (see next section) is the current route represented on the screen.

```
class AppModule extends MainModule {
    @override
    List<Bind> get binds => [
        Bind((i) => UserRepository()),
        Bind((i) => MeasureRepository()),
        Bind((i) => DatabaseRepository(), lazy: false),
        Bind((i) => AuthenticationBloc(userRepository: i.get<UserRepository>()),
            lazy: false),
        Bind((i) => InternetCubit(connectivity: Connectivity()), lazy: false),
        Bind((i) => BoardConnectionBloc(
                databaseRepository: i.get<DatabaseRepository>(),
                internetCubit: i.get<InternetCubit>(),
                measureRepository: i.get<MeasureRepository>()),
            lazy: false),
        Bind((i) => NotificationsHandler(), lazy: false),
        Bind((i) => RoutingRepository()),
    ];
    ...

  @override
  Widget get bootstrap => PolitoWeatherStationApp();
}

Future<void> main() async {
  ...
  runApp(EasyLocalization(
      ... ,
      child: ModularApp(module: AppModule())));
}
```

In order to retrieve one of the objects that can be inject inside module's code, it's enough to use the static method $Modular.get < ClassName > ()$: in this way, as the code below shows, it is possible to use the reference to that single object belonging to the $ClassName$ class. Actually, there is no mechanism to solve ambiguities, so it is recommended to instantiate a class only once inside the *binds* list getter. Though, despite the natural behaviour of this software engineering pattern requires to use singletones (classes to be instantiated only once), it is possible to create a new instance of the $ClassName$ class every time the $Modular.get < ClassName > ()$ method is invoked by passing the $false$ value to the $Bind$ constructor's *singleton* parameter. Moreover, if the developer wants to instantiate the class $ClassName$ as soon as possible and not after the first invocation of the $Modular.get < ClassName > ()$ method, it is possible to set the *lazy* parameter of the $Bind$ constructor to $false$. Otherwise, the default behaviour will instantiate the class after the first request and it will always retrieve this same object every time when required to.
The following code snippet shows a use case of the $Modular.get < ClassName > ()$ method: here, the dependency injection is exploited to avoid those problems, mentioned in section 4.3.1.1, related to the $BlocProvider$ widget.

```
BlocBuilder(
    cubit: Modular.get<LatestMeasureCubit>(),
    builder: (BuildContext context, LatestMeasureState state) {
        if (state is DaysMeasureOngoing) {
            ...
        } else if (state is DaysMeasureAvailable) {
            ...
        }
    }
)
```

Another interesting implementation of $Modular$'s dependency injection mechanism is achieved by the $ModularState$. This abstract class can be used in a stateful widget when

there is the need to retrieve a controller only in the current Page - that is a stateful widget covering the whole navigation screen. Indeed, considering the example below taken from the *_BodyState* class of the Home page, by using this trick the variable *controller* is automatically instantiated as an injection of the *ChipsController* class, even though this class was not listed in the *binds* list of the module. In particular, *ModularState* will destroy the *controller* variable as soon as the page is destroyed, that is when a new route is requested; this mechanism is very interesting because it binds view and controller as a single component.

```
class _BodyState extends ModularState<Body, ChipsController> ... {
    ...

    Widget build(BuildContext context) {
        ...

        Column(children: [
            ChipsList(controller: controller),
            ...
        ])
        ...
    }
}
```

Finally, this examples introduces the last realization of dependency injection, based on *ChangeNotifier* and *Consumer* classes. As shown in the following code snippet, the *ChipsController* class extends *ChangeNotifier*, which means that, by calling the *notifyListeners*() method this class can trigger the rebuild of a part of the UI according to the changes occurred before the call. Which part of the UI? The sub-tree that has a *Consumer* widget as the root node of the widget sub-tree, as displayed in the code snippet. This mechanism has been strategically used in this scenario, because the *ChipsController* class - that internally knows which of the chips in the upper part of the Home page (see figure 4.2 on page 57) has been selected - can trigger not only the border color of the chip itself but also the UI of the entire page since most of the widgets depend on which day is currently selected.

```
class ChipsController extends ChangeNotifier {
  int selected = 5;
  Map<String, Color> chipsData = {};

  ...

  int setSelected(int index) {
    this.selected = index;
    notifyListeners();
  }
}

class CustomChip extends StatelessWidget {
  final int index;
  ...

  CustomChip({Key key, this.label, this.index, this.color, this.selected})
      : super(key: key);

  Widget build(BuildContext context) {
    return Consumer<ChipsController>(
      builder: (context, chipsController) => AnimatedContainer(
          margin: const EdgeInsets.only(left: 9.0),
          width: 100,
```

```
          height: 30,
          duration: const Duration(milliseconds: 300),
          decoration: BoxDecoration(
            ...
            border: Border.all(
              color: index == chipsController.selected
                  ? darkTextColor
                  : Colors.grey,
              width: index == chipsController.selected ? 2 : 1,
            ),
          ),
          child: InkWell(
            onTap: () async => {chipsController.setSelected(index)},
            child: ...
          )),
    );
  }
}
```

4.3.3 Page routing

The routing of the mobile application is a mechanism that governs which page is shown on the screen according to the current route. In this project, the routing feature is provided by the *flutter_modular* package, the same introduced in the previous section: indeed, every module of the mobile application associates a specific widget tree to each route mentioned by the *routers* getter inside the module definition. For example, as shown in the following code snippet, the *HomeModule* specifies only one route, called */home*; typically the name of the route resembles a Uniform Resource Identifier (URI) (see section 2.2.1), because this convention allows to exploit dynamic parameters in the route as explained later. Notice that any module different from the main module must extend the *ChildModule* abstract class and must be created either by another child module or by the main module itself.

```
class AppModule extends MainModule {
  @override
  List<Bind> get binds => [
        ...
      ];

  @override
  List<ModularRouter> get routers => [
        ModularRouter('/', child: (context, args) => WelcomeScreen()),
        ModularRouter('/login', module: LoginModule()),
        ...
      ];

  @override
  Widget get bootstrap => PolitoWeatherStationApp();
}

class HomeModule extends ChildModule {
  @override
  List<Bind> get binds => [
        ...
      ];

  @override
  List<ModularRouter> get routers => [
        ModularRouter('/home', child: (context, args) {
          return HomeScreen();
        })
```

```
    ];
}
```

In order to push a route into the application and consequently show the relative widget tree, it is possible to use the *Navigator.pushNamed*() static method or the *Modular.to.pushNamed*() method by passing the name of the route. If the name follows the URI convention and if there is the need to push a route whose name has the same prefix of the current route, in the place of the method previously mentioned it is possible to call the *Modular.link.pushNamed*() method by passing only the final part of the route name that characterizes it. An example for both methods is provided in the code snippets below.

```
Column(
    children: [
        UpperRightButton(
            flex: 3,
            text: 'signUp.label',
            onTap: () => Modular.to.pushNamed('/register'))
    ]
);

BoardDialogButton(
    ...
    onTap: () => Modular.link.pushNamed('/board/${boardID}',
        arguments: {'street': street})
)
```

The last code snippet is also an example of a dynamic route: indeed, these kinds of routes are capable of encapsulating a dynamic parameter inside the route name and of attaching an argument to the route call. In particular, the previous information about the *boardID* and the *street* (both String objects) will be passed to the route definition, which will eventually build the widget tree on the bases of parameters and/or arguments passed in this way. The following code snippets show some details about the route referenced in the previous example; moreover, the following code snippet also shows the transition features provided by the *Modular* package.

```
class MapModule extends ChildModule {
  @override
    List<Bind> get binds => [
        ...
    ];

  @override
  List<ModularRouter> get routers => [
        ModularRouter('/map', child: (context, args) => MapScreen()),
        ModularRouter('/board/:boardID',
            child: (context, args) => BoardDetailScreen(
                boardID: num.parse(args.params['boardID']),
                street: args.data['street']),
        transition: TransitionType.rightToLeft,
        duration: const Duration(milliseconds: 400))
      ];

  ...
}
```

Finally, the package also features route guards. These act as middleware objects that control access to a certain route when coming from another route. In this mobile

application, route guards are used to check whether the user accessing a certain module has the rights to do so as the result of the authentication procedure. The following code snippet reports the routes that need authentication for being accessed.

```
class AppModule extends MainModule {
  @override
  List<Bind> get binds => [
        ...
      ];

  @override
  List<ModularRouter> get routers => [
        ...
        ModularRouter('/home',
            module: HomeModule(),
            guards: [AuthGuard()],
            transition: TransitionType.noTransition),
        ModularRouter('/map',
            module: MapModule(),
            guards: [AuthGuard()],
            transition: TransitionType.noTransition),
        ModularRouter('/profile',
            module: ProfileModule(),
            guards: [AuthGuard()],
            transition: TransitionType.noTransition),
        ModularRouter('/settings',
            module: SettingsModule(),
            guards: [AuthGuard()],
            transition: TransitionType.noTransition),
        ...),
      ];

  ...
}
```

The *AuthGuard* object instantiated for the previous routes works regardless of the route name as stated by the *canActivate*() method, and it performs the checks defined in the list of *GuardExecutor* objects provided by the *executors* getter. In particular, the *LoginExecutor* checks if the user has logged in by leveraging the *userHasLoggedIn* attribute of the *AuthenticationBloc*, which is one of the first objects to be instantiated when the mobile application starts.

```
class AuthGuard implements RouteGuard {
    @override
    bool canActivate(String url) {
        return true;
    }

    @override
    List<GuardExecutor> get executors => [
        LoginExecutor(userHasLoggedIn: Modular.get<AuthenticationBloc>().userHasLoggedIn)
    ];

}

class LoginExecutor extends GuardExecutor {
  LoginExecutor({@required this.userHasLoggedIn});

  bool userHasLoggedIn;

  @override
  void onGuarded(String path, {bool isActive}) {
    if( userHasLoggedIn ) {
      return;
```

```
    }
    Modular.to.pushNamed('/');
  }

}
```

Chapter 5

Expansion board

5.1 Characteristics

As mentioned in section 1.3.2, before switching to this expansion board, the monitoring devices were featuring a Raspberry Pi as central computation unit. During the Raspberry Pi phase of this thesis work, some time was spent on understanding how the operating system - Arch Linux - was supervising both pairing and communication processes, and some difficulties quickly arose in the development of a working solution. Indeed, given that the point of the overall procedure is to ensure a smooth sending procedure after Bluetooth pairing, under the supervision of an operating system it is quite difficult both to realize a working solution and to debug errors. When the research group decided to utilize a microcontroller instead of the Raspberry Pi, despite accepting some reasonable compromises, the development of the Bluetooth Low Energy (BLE) pairing and communication module has quickly sped up with the Pycom extension board.

What makes this board easy to program is the MicroPyhton programming language. MicroPython is a lean and efficient implementation of the Python 3 programming language that includes a small subset of the Python standard library and is optimised to run on microcontrollers and in constrained environments [6]. MicroPython is packed full of advanced features such as an interactive prompt, arbitrary precision integers, closures, list comprehension, generators, exception handling and more. Yet it is compact enough to fit and run within just 256k of code space and 16k of RAM [6].

The IDE used to develop code for the expansion board was Visual Studio Code: this choice was due to the need of installing the Pymakr plugin developed by Pycom in order to upload code onto the microcontroller; moreover, the installation of the plugin also creates an environment were the Read Evaluate Print Loop (REPL) terminal can be used to dynamically interact with the board for debugging purposes. Uploading code is really simple because the plugin provides an *upload* button in the bottom part of the IDE. The typical project structure of a MicroPyhton project features a folder, called *lib*, where additional libraries will be added by the developer, and two files called *main.py* and *boot.py*. The former one - reported in the code snippet below - is a Python script that is executed right after the code in the *boot.py* file and it contains the code that the microcontroller should run; conversely, the latter one contains a Python script to be executed as soon as the module turns on. Best practises suggest to partition code into

libraries, typically one per functionality: thus, in this case, since the thesis work focuses only on BLE features for what concerns the expansion board, the contribute given to this project has been included inside a *ble.py* file created inside the *lib* folder. Moreover, the documentation itself recommends not to create new sub-folders inside the *lib* folder.

```
# main.py

import ble as ble

bt = ble.BLE()
bt.start_advertisement()
```

Finally, here are some general details about how the rest of the work is performed on the board. Measures are taken once per second by every sensor: thus, since there are 4 sensors for $PM_{2.5}$ particles, 4 sensors for PM_{10} particles, 1 sensor for temperature and 1 sensor for relative humidity, every second 10 measures are taken by the board. The whole measurement process takes less than half a second to complete, so for the remaining time until next measurement the microcontroller is in idle state. The research group is taking into consideration the idea of considering the geographic location retrieved by an external GPS sensor as a new measure, causing an increment to the number of data produced per second.

5.2 Bluetooth Low Energy

The fundamentals of Bluetooth's GATT and ATT protocol have been explained in section 4.2.3, so this section introduces some small details to the global picture.

The expansion board will play the role of the *GATT Server* in the whole procedure of data communication: indeed, this device will receive connection requests coming from users' smartphones acting as *GATT Clients*.
Considering GATT attributes, in theory they are always located on the server and accessed (and potentially modified) by the client. The specification defines attributes only conceptually, and it does not force the ATT and GATT implementations to use a particular internal storage format or mechanism. Because attributes contain both static definitions of invariable nature and also actual user (often sensor) data that is bound to change rapidly with time, attributes are usually stored in a mixture of nonvolatile memory and RAM [13].

For this thesis work many attempts were performed in order to find out the best strategy to send the highest number of measures in the lowest possible time, and eventually it turned out that the best strategy requires to make some small compromises in terms of data storage. First of all, even though it is possible to store measures in a SD card in order to read them as soon as someone connects to the board via BLE, this is not recommended due to the fact that it takes too much time to read from file and to consequently parse the data. Connection time periods are a priority for this project because the BLE connections will persist as long as the smartphone doesn't get locked or the user walks away from the monitoring device: thus, it is vital to complete the sending process as soon as possible, and the target time was set to no more than 2 minutes. Therefore, the best solution is based on having the RAM memory as main storage of the measures collected during the day: every microcontroller will be programmed to store only data collected in the

current day, and this doesn't totally compromise the objective of the project because the microcontroller will leverage its Lo-Ra antenna to hourly communicate with a remote server even in case no smartphones running the application are passing by the monitoring device.

One of the disadvantages of the lack of an underlying operating system is the fact that the microcontroller allocates RAM memory space only at boot time on the basis of a static code analysis; although this might cause allocation problems, it is justifiable on a resource-constrained device whose execution is not supervised by any operating system. The only solution to this problem was to create a big data structure to hold all the possible measures being collected throughout the day: after some trial-and-error attempts, the best data structure successfully allocated was capable of containing 40000 measures (about 1 MB of data) - see code snippet below. Considering that 40000 measures must represent one-day measurements, this means that 1666 measures will represent one hour of measurements to be stored in RAM, and consequently 27 measures stored in RAM will summarize one minute of measurements: therefore, given that 10 measures are taken in one go, the sampling time becomes 38 seconds. Though, this doesn't really mean that actually one sample must be performed every 38 seconds, but it just means that for every 38 samples taken in 38 seconds, only one value must be stored in RAM memory. So, on the basis on the discussion about the Reff method (see section 1.2.4) the natural choice would be a weighed average of these 38 measures, but unfortunately this would lead to some issues with the calibration process, hence the decision to compute a simple average. Notice that there is actually no point in detecting a short spike in air pollution measurements, because after a few seconds the concentration of pollution will dramatically decrease and return to the previous value; instead, the point of this project is to detect prolonged pollution activities that have long-term consequences on the air quality.

```
# ble.py

...

class BLE:

    def __init__(self):
        ...
        self.data = [bytearray(25)] * 40000
        ...
```

5.2.1 Advertising the presence of the board

In order to trigger the connection process initiated by the *GATT Client*, the board must advertise its presence. The framework used by the development board offers some methods to perform the majority of BLE operations, including the advertisement. In particular, as shown in the following code snippet, not only the presence of the board but also the UUID of a Service is advertised: this UUID refers to the Service which contains only the Characteristic that will progressively hold all the measures to be sent.

```
# ble.py

from network import Bluetooth
```

```
class BLE:
    def __init__(self):
        ...
        self.bluetooth = Bluetooth(...)
        ...

    def start_advertisement(self):
        self.bluetooth.set_advertisement(
            name='weather-station', service_uuid=0x12340)
        self.bluetooth.advertise(True)

    ...
```

5.2.2 GATT Characteristic

Before analyzing how data is transmitted to the user's smartphone, let's see how Bluetooth Low Energy (BLE) connections are handled by the framework. The constructor of the *Bluetooth* class gives the chance to customize the behaviour of the system when new connections are required by passing appropriate arguments as parameters. In particular:

- the *mtu* parameter represents the Maximum Transmission Unit, that is the maximum length of any ATT packet, and it must be a value between 23 and 200
- the *secure_connections* parameter protects from MITM attacks; as this is not required by the current scenario, it is set to False in order to avoid any complication in the connection mechanism.
- the *pin* parameter consists of 6 digits required to connect to the *GATT Server*, and it is not indicated in the code snippet below because its default value is None: indeed, it doesn't make sense to extend the connection time with a PIN when the only operation allowed to the *GATT Client* is reading data from a Characteristic.

Then, a simple callback function has been defined to show on the REPL terminal when a device has connected to or disconnected from the microcontroller: this callback is the *conn_cb* function reported in the code snippet below.

```
# ble.py

from network import Bluetooth

class BLE:
    def __init__(self):
        ...
        self.bluetooth = Bluetooth(mtu=200, secure_connections=False)
        self.bluetooth.callback(trigger=Bluetooth.CLIENT_CONNECTED |
                                Bluetooth.CLIENT_DISCONNECTED, handler=self.conn_cb)
        ...
        self.create_measure_characteristic()
        ...

    def conn_cb(self, bluetooth):
        # returns the flags and clears the internal registry
        events = self.bluetooth.events()
        if events & Bluetooth.CLIENT_CONNECTED:
            print("BLE client connected")
        elif events & Bluetooth.CLIENT_DISCONNECTED:
            print("BLE client disconnected")

    ...
```

Now let's see how to create a Service and its Characteristics. The *Bluetooth* class comes with a straightforward and declarative way of creating Services: indeed, by calling the *service*() method it is possible to set Service's UUID, typology (primary or secondary), number of characteristics (see *nbr_chars* parameter) to be contained and starting time. The following code snippet shows how to do this in practice. Moreover, given the *GattSService* object obtained from the previous call, in order to instantiate a characteristic related to this service, it is necessary to call its *characteristic*() method, as shown below, and to provide the appropriate parameters such as UUID and property. In particular, since the *GATT Server* will send several measures during the connection period, the property of this characteristic is set to $Notify$ for two main reasons: first of all, as soon as its value changes, the *GATT Client* will be informed (provided that it has subscribed to the Characteristic) so that it can perform a new read operation; secondly, in order to speed up the transmission procedure, no acknowledgment message will be required from the *GATT Client* before changing the value of the Characteristic. Finally, the code presented below registers a callback function for each subscribe event for triggering the transmission of measurements data and for debugging purposes.

```
class BLE:
    def __init__(self):
        ...
        self.bluetooth = Bluetooth(mtu=200, secure_connections=False)
        self.bluetooth.callback(trigger=Bluetooth.CLIENT_CONNECTED |
                                Bluetooth.CLIENT_DISCONNECTED, handler=self.conn_cb)
        self.service = None
        self.data = [bytearray(25)] * 2000
        self.measure_characteristic = None
        self.create_measure_characteristic()
        ...

    def create_measure_characteristic(self):
        measure_service = self.bluetooth.service(
            uuid=0x12340, isprimary=True, nbr_chars=1, start=True)
        self.measure_characteristic = measure_service.characteristic(
            uuid=0x54326, properties=Bluetooth.PROP_NOTIFY)
        # characteristic.value(None)
        self.measure_characteristic.callback(
            trigger=Bluetooth.CHAR_SUBSCRIBE_EVENT, handler=self.char_measure_cb_handler)

    def char_measure_cb_handler(self, chrstc, data):
        events, value = data
        if events & Bluetooth.CHAR_SUBSCRIBE_EVENT:
            print("Received notify request on measure characteristic")
            # send data via BLE
            self.send_data_via_ble()

    ...
```

Changing the value of the Characteristic and sending the corresponding notification is done just by calling the *value*() method on the *GattSCharacteristic* object previously obtained and stored in the $self.measure\_characteristic$ variable. Let's imagine that all the measures are already stored in the RAM memory inside the $self.data$ data structure instantiated in the constructor of *BLE* class: the first value to be notified will be the number of measures ready to be sent, followed by all the measures contained in $self.data$. In reality, as explained in section 4.2.3 about *GATT Client*'s BLE implementation, inside a single value of the Characteristic not just one but seven measures will be notified. In this way, the MTU size is optimized since almost all bytes are used: it would have been a pity

not to exploit all of them since the amount of data being sent would have been the same for each notification. For what concerns the data format, as stated by the ATT protocol itself, there is no recommended one, so it has been decided to use the *bytearray* data structure (with UTF-8 encoding) for each measure. Notice that between one notification and the next one a certain amount of time must pass, otherwise some notifications will get lost and the sending of all the measures will fail. In particular, using a trial-and-error approach, this limit has been found out to be 0.01 seconds. Finally, in case the sending process abruptly stops (e.g. because the user is too far), the *GATT Server* will always know which was the last element of the list of measures to be sent: thus, confident that the *GATT Client* has already stored the measures that were successfully sent, the *GATT Server* can decide whether to delete them from the data structure or not.

```
class BLE:
    ...

    def send_data_via_ble(self):
        buffer = bytearray() # contains a set of measures to be sent
        counter = 0 # counts the numbers of measures inside the buffer
        block_number = 0 # counts the number of buffers sent to the Gatt Client

        sleep(5)

        try:
            # send the number of measures ready to be sent
            self.measure_characteristic.value(
                bytearray(str(len(self.data)), 'utf-8'))

            # send each measure separately
            for measure in self.data:
                counter = counter + 1
                if counter <= 7:
                    buffer += measure
                    if counter != 7:
                        separator = bytearray('|', 'utf-8')
                        buffer += separator
                if counter == 7:
                    self.measure_characteristic.value(buffer)
                    sleep(0.01)
                    buffer = bytearray()
                    counter = 0
                    block_number = block_number + 1
            # if last bunch of measures is not a multiple of 7
            # then send them anyway
            if len(buffer) > 0:
                self.measure_characteristic.value(buffer)

        except Exception as e:
            print("Error: {}".format(e))
            # choose whether to delete part of the self.data array's items
            print("Transfer was interrupted, but {} blocks were successfully sent,
                meaning {} measures".format(block_number, block_number * 7))
```

5.2.3 Performances

The code reported in the previous section would not be the same without so many changes in the strategy of the communication with the mobile application: since the user is not aware of her participative role in the system's architecture, it is vital to guarantee that measures will be transferred in the least possible time regardless of their quantity. This section tells some of the most interesting attempts and their performances.

The first attempt was so focused on the need of sending measures in the fastest possible way to the point that it was based on two separate solutions based on the operating system of the user's device. Since Android allows a mobile application to frequently look for WiFi networks with a specific name, the solution for Android devices would require to turn the WiFi Access Point (AP) on and would leverage the fact that both the Client and the Server are in the same LAN to transfer the measures via CoAP, a protocol designed for constrained devices which tries to implement a simplified version of HTTP. Conversely, iOS devices would use Bluetooth Low Energy (BLE) to receive measures because the operating system is very strict on the number of WiFi scans. Though, at that point, if the Android device needs to be in the same WiFi network with the microcontroller, it would be better to use just sockets because data wouldn't need further encapsulation: the natural consequence of this approach ended up using secure sockets via SSL. In practice, the idea of this attempt was to use the fastest possible way compatibly with the smartphone's operating system constraints. The resulting benchmark, based on sending 1000 measures (about 40 KB), is reported in table 5.1: at first impact, someone might say that the solution implemented by Android is too slow, but the two solutions performed almost the same time when comparing just the transfer time. Indeed, the problem with SSL is that it takes too much time to join the board's WiFi network, to allocate a socket, to perform the handshake protocol and to cipher data (although ciphering is definitely other than a bottleneck). Assuming that the 24 seconds indicated in the iOS column are the time required to read 40 KB from disk (at that time measures were still saved on disk), it is possible to say that the board needs to wait 23 more seconds to send the first measure.

Event	Android (seconds from start)	iOS (seconds from start)
Device connects to board	0	0
Device receives first measure	47	24
Device receives last measure	51	29

Table 5.1: Benchmark for sending 1000 measures: Android (via SSL) vs. iOS (via BLE)

The results shown in the first attempt encouraged the utilization of BLE also on Android devices for the transfer of measures. Moreover, many doubts arose about the fact that, by joining the board's WiFi AP for several seconds, in that time period all the interactions between the mobile application and the server won't work: this might compromise the user experience both on the app and for what concerns all other running services using Internet. Therefore, the second attempt is characterized by the alignment of the two solutions towards the BLE technology. Moreover, as explained in section 5.2.2, the alignment involved also the MTU size, which has been set to 197 bytes for both Android and iOS, and the concatenation of seven measures into one single BLE packet for optimizing the MTU size. The resulting benchmark, based on sending 1000 measures (about 40 KB), is reported in table 5.2: it is evident that results improved from previous benchmark, but in the overall process still 24 seconds are wasted in disk read. This introduces the third and last attempt.

The third attempt is based on removing disk read from the whole process. As discussed in section 5.2.2, this results in having all measures stored in RAM so that they are ready to be sent as soon as the Client connects. The benchmark for sending 1000 measures

Event	Android/iOS (seconds from start)
Device connects to board	0
Device receives first measure	24
Device receives last measure	26

Table 5.2: Benchmark for sending 1000 measures via BLE: Android vs. iOS

revealed to be extremely fast: in order to appreciate the effectiveness of this solution and to discover how many measures can be sent within the arbitrary threshold of 2 minutes (the longest acceptable engagement time with the app), the following benchmark test is based on sending 40000 measures (almost 1 MB). From the results shown in table 5.3 two lessons can be learnt: in first place, 40000 is the reference number of measures that can be sent in less than 2 minutes; secondly, BLE throughput is variable and it depends on various factors (e.g. presence of many WiFi networks nearby) - because the expected transfer time should have been 80 seconds, that is 40 times the time required for sending 1000 measures.

Event	Android/iOS (seconds from start)
Device connects to board	0
Device receives first measure	5
Device receives last measure	110

Table 5.3: Benchmark for sending 40000 measures via BLE: Android vs. iOS, measures stored in RAM memory

In conclusion, the last solution explained in this section is the best candidate to transfer measures. Nevertheless, further checks should be performed to evaluate what happens when other modules - besides the *BLE* one - are working at the same time. Finally, a remarkable point for future works is that ensuring to have measures stored in RAM memory at the time of a BLE connection doesn't necessarily exclude that measures can be stored on disk too: indeed, when no device is connected to the board and the board itself is in idle time, the microcontroller might leverage this moment to copy on disk all the measures kept in RAM memory up to that time.

Chapter 6

Results

The goal of this project is to build and deliver into citizens' hands a tool for improving their health conditions and therefore their lives: hence, the participative nature of the overall system. In order to involve the largest possible number of people in this project, it was fundamental to realize a mobile application for both Android and iOS smartphones, and to make the app as engaging as possible by taking care of the User Interface and the User Experience. Moreover, optimized connection and transfer time between application and monitoring devices was a cornerstone of this thesis work and of the project itself. Software development teachings point out that every line of code has responsibility on the real world: raising awareness of air pollution dangers and encouraging citizens to avoid polluting the environment whenever possible are the best outcomes that any contributor to this project would wish to happen. For what concerns the deployment of the mobile application, even though behind the scenes, part of this thesis work dealt with both app installation on testers' devices and standard procedures for signing and uploading the first version of the app to Apple's App Store with the name *Shair* : *Share the Air*.

As pointed out in several sections of this thesis work, despite the great results shown in the previous chapters, still there are features to improve and possibly to add. The first problem is related to user's awareness of being part of a participative system: since the user doesn't have any clue that her smartphone is receiving data from monitoring devices when it's near enough, she won't be encouraged to walk by them and hence this might compromise the goal of the project. A common solution to user engagement's low levels is gamification: this technique, inspired by videogames, motivates users to interact with a software application more frequently by following game principles (e.g. challenges, rankings, rewards) in a context which typically videogames don't belong to such as air pollution monitoring. This could be a strategic and up to date solution to this problem, and therefore it should be taken into consideration for future works.

Finally, another area where no effort was made yet is data analysis. Looking at possible future works in this project, it would be a great challenge to include some activities to analyze data in order to extract models and to discover patterns: this might be integrated in the mobile application too since the results of forthcoming air pollution conditions would help people plan their activities outdoor.

Bibliography

[1] U.S. Environmental Protection Agency. https://web.archive.org/web/20180113035458/https://www3.epa.gov/airnow/ani/pm25_aqi_reporting_nowcast_overview.pdf.

[2] Scientific American. https://www.scientificamerican.com/article/experts-how-does-bluetooth-work/.

[3] Bartolomeo Montrucchio, Edoardo Giusto, Mohammad Ghazi Vakili, Stefano Quer, Renato Ferrero, Claudio Fornaro. A Densely-Deployed, High Sampling Rate, Open-Source Air Pollution Monitoring WSN. *Journal of LaTeX class files*, 14(8):1–13, 2015.

[4] European Commission. https://ec.europa.eu/environment/air/quality/.

[5] Bloc Community. https://bloclibrary.dev/#/whybloc.

[6] MicroPyhton Community. https://micropython.org.

[7] CRAN. https://cran.r-project.org/web/packages/PWFSLSmoke/vignettes/NowCast.html.

[8] Amir Nagah Elghonaimy. LoRaWAN for Air Quality Monitoring System. pages 3–5, 2021.

[9] Flutter. https://flutter.dev/docs/resources/architectural-overview.

[10] IoT2US. http://iot.eecs.qmul.ac.uk/wp-content/uploads/sites/46/2018/06/2018-Aamer-Very-Low-Cost-Open-Wireless-Internet-of-Things-IoT-Air-Quality-.pdf.

[11] Daryna Kacherovska. https://dzone.com/articles/how-secure-is-the-ble-communication-standard.

[12] MWAY. https://mway.io/en/code/flutter/.

[13] O'Reilly. https://www.oreilly.com/library/view/getting-started-with/9781491900550/ch04.html.

[14] World Health Organization. https://www.who.int/health-topics/air-pollution.

[15] World Health Organization. https://www.euro.who.int/__data/assets/pdf_file/0005/112199/E79097.pdf.

[16] OurWorldInData. https://ourworldindata.org/indoor-air-pollution.

[17] OurWorldInData. https://ourworldindata.org/outdoor-air-pollution.

[18] OurWorldInData. https://ourworldindata.org/grapher/indoor-pollution-deaths-1990-2017.

[19] OurWorldInData. https://ourworldindata.org/grapher/outdoor-pollution-deaths-1990-2017.

[20] OurWorldInData. https://ourworldindata.org/grapher/emissions-of-air-pollutants-oecd?country=~ITA.

[21] Phys.org. https://phys.org/news/2016-04-air-pollution.html.

[22] prashun javeri. https://medium.com/@prashunjaveri/rethinking-iot-architecture-the-need-for-distributed-systems-architecture-for-t

[23] Pycom. https://docs.pycom.io/datasheets/expansionboards/expansion3/.

[24] Pycom. https://pycom.io/product/fipy/.

[25] SAVVY. https://savvyapps.com/blog/push-notification-increase-engagement.

[26] Matt Sullivan. https://medium.com/flutter/flutter-dont-fear-the-garbage-collector-d69b3ff1ca30.

[27] Jukka Riekki Teemu Leppänen. Energy Efficient Opportunistic Edge Computing for the Internet of Things. *Web Intelligence and Agent Systems*, pages 1–6, 2018.

[28] Wikipedia. https://en.wikipedia.org/wiki/Cross-platform_software.