



POLITECNICO MILANO 1863

MOQA

Design and Implementation of a Mobile Application

Prof. Luciano Baresi

2019/2020

Design and Technology

Mathyas Giudici

Document version: 1.1

May 28, 2020

Deliverable: Design and Technology Documentation
Title: MOQA Design and Technology
Authors: Mathyas Giudici
Version: 1.1
Date: May 28, 2020
Documentation page: <https://github.com/MathyasGiudici/polimi-dima-moqa-documentation>
Code page: <https://github.com/MathyasGiudici/polimi-dima-moqa>
Copyright: Copyright © 2020, Mathyas Giudici – All rights reserved

Contents

1	Introduction	4
1.1	Purpose	4
1.2	Scope	4
1.3	Stakeholders	5
1.4	Time Constraints	5
1.5	Risk Analysis	6
1.6	Overview	6
2	General Overview	8
2.1	Concept	8
2.2	Core features	9
2.3	Functional Requirements	11
2.4	Non-Functional Requirements	14
3	Architectural Design	15
3.1	Overview	15
3.2	Component View	17
3.3	Deployment View	21
3.4	Runtime View	22
3.5	Component Interfaces	24
3.6	Architecture styles and patterns	27
3.7	Other Design Decisions	28
4	User Interface Design	30
4.1	Overview	30
4.2	Mock-up	30
4.3	Screens	32
5	Frameworks, External services and Libraries	43
5.1	Vuex	44
5.2	Maps, Charts and DateTimePicker	45

5.3	Application Server	46
5.4	ARPA Data	48
6	Test Cases	49
7	Effort and Cost Estimation	54
7.1	Effort	54
7.2	Cost Estimation	55
8	Future Works	57
A	Appendix	58
A.1	Software and Tools	58
A.2	Changelog	58

Chapter 1

Introduction

1.1 Purpose

In this section, I want to introduce briefly my application.

The application is developed for the course of *Design and Implementation of Mobile Applications* of professor Luciano Baresi at Politecnico di Milano.

The goal of the course is to design and implement a mobile application on a platform of our choice. This document illustrates the decisions I made in order to accomplish this goal.

This Software Design Document is a document that provides documentation that will be used as overall guidance to the architecture of the software project. Moreover, I will provide documentation of the software design of the project, including use case models, class and sequence diagrams.

The purpose of this document is to provide a full description of the design of *MOQA*, a cross-platform application, providing insights into the structure and design of each component.

1.2 Scope

The application comes by a need of professor Fulvio Re Cecconi (ABC Department of Polimi).

He made an Arduino board that allows measuring some air quality parameters like temperature, relative humidity, light, etc... Measures are GPS located and saved to a microSD card.

The aim is to increase the accuracy of the GPS location using GNSS Real Time Kinematics (RTK); the increased precision is obtained thanks to a GPS correction signal transmitted via the internet. Thus Arduino must be equipped with a connection and use a mobile phone as a modem to measure more air quality parameters and automatically produce dynamic maps.

During the initial meetings, we also decided to include a dynamic graph and compare Arduino data with the data registered by ARPA.

1.3 Stakeholders

The main stakeholders of my project are Professor Re Cecconi, students and people that have to take measures with the Arduino board.

I have to consider, as a stakeholder, professor Luciano Baresi that holds the *Design and Implementation of Mobile Applications* course and Giovanni Quattrocchi, teaching assistant of the course.

1.4 Time Constraints

I had no precise and punctuated deadlines, but to deliver the project among the different call dates for the course.

Professor Re Cecconi asks for an application as soon as possible; however, without fixing a precise deadlines.

Initial meeting was done at the end of November 2019 and a fist mock-up was provided in the mid of December 2019. During Christmas vacations and in the free-time in January and February I spent time learning the different frameworks used.

I decided to plan the start of development at the beginning of the second semester. Developing, testing and creating documentation is planned to last less than 6 months. Even the server-side part will build and maintained during the same time.

I started development in February 2020 and complete it in June 2020.

1.5 Risk Analysis

In the first phase of analysis, I try to exploit the possible risks that can compromise the project development.

Risk Analysis was made in collaboration with the other two students that decided to develop the same project.

We had to be very careful during the first phases of implementation, especially for the Arduino code, to manage all possible issues, leading to possible code rewriting and the inevitable loss of time.

1.6 Overview

This document is structured as follows:

Section 1: Introduction. A general introduction of the Design and Technology Document. It aims giving general but exhaustive information about what this document is going explain.

Section 2: General Overview. A general overview of the project. In this section, the reader could find the core features of the application and the requirements of the system.

Section 3: Architectural Design. This section contains an overview of the high-level components of the system and then a more detailed description of three architecture views: Component view, Deployment View and Runtime View. Finally, it shows the Component Interfaces and the chosen architecture styles and patterns.

Section 4: User Interface Design. This section contains the screenshots of the application with some comments to give to the reader a general overview of the user interfaces.

Section 5: Frameworks, External services and Libraries. This section aims to explain the main frameworks, external services and libraries used pointing out their advantages.

Section 6: Test Case. This section identifies test cases performed reporting their results.

Section 7: Effort and Cost Estimation. A summary of worked time and cost estimation of the work.

Section 8: Future Works. This section contains some new features that, in future, could be add to *MOQA* or the *Application Server*.

At the end, there is an **Appendix** where software and tools used are reported.

Chapter 2

General Overview

2.1 Concept

MOQA is a cross-platform application that allows to retrieve air parameters registered by an *Arduino* board, send them to a remote server and visualize through a map and a chart.

The application requires users to be authenticated via his/her email and password (a new user could be registered filling up a form in the application). This strong requirement because to retrieve data and to send data to the remote server you need an authentication.

Without these data the application could not provide any type of feature it has designed to; so it has no sense maintain a part without authentication.



Figure 2.1: *MOQA* logo

The main components of the project are:

- A client-side part, that is *MOQA*, which queries the database or retrieves data from *Arduino*, visualizes data on a map and a chart, send new *Arduino* data to the server;
- A server-side part, composed by a relational database that contains air quality data and users data; moreover it contains the application logic that delivers the RestfulAPI for the authentication and the data management.

2.2 Core features

in this section, a list of the core functionalities of the application is exploited. The features are divided by screen where they are present, allowing the reader to easily understand them and where they could be found in the application.

2.2.1 Authentication Screens

- Allows the user to authenticate using his/her email and password;
- Allows the user to register his/her-self in the application.

2.2.2 Arduino Screen

- Allows the user to have a quick view of the last data retrieved by *Arduino*;
- Allows the user to send the data that is retrieved by the *Arduino* board to the remote server;
- Allows the user to fetch new data from *Arduino* (it is available an auto-fetch mode where the phone every second pulls new data);
- Allows the user to decide if he/she wants to visualize the data get by the board or from the remote server.

2.2.3 Map Screen

- Allows the user to see the data on a Map. For each data a circle in its location is created, the radius of the circle depends on the value of the measure considered;

- Allows the user to go to the related *Filter Screen* to change the data visualized.

2.2.4 Chart Screen

- Allows the user to see the data on a Graph ordered by date and time;
- Allows the user to see the data dispersion (tabulated as 1st, 2nd and 3rd quartile);
- Allows the user to go to the related *Filter Screen* to change the data visualized.

2.2.5 Filter Screen

- Allows the user to select the measure (among the sampled one) he/she wants to visualize;
- Allows the user to pick a start and end date to define a time range of data to visualize;
- Allows the user to select if he/she wants to visualize the available ARPA data;
- Allows the user to select the ARPA station from which data are taken.

2.2.6 Settings Screen

- Allows the user to manage his/her account changing personal information or the password;
- Allows the user to logout from the application;
- Allows the user to edit the IP address and port where the *Arduino* board is available;
- Allows the user to edit the IP address and port where the remote server, with the application logic, is available;
- Allows the user to manage the API links to the *Regione Lombardia* OpenData service;
- Allows the user to reset the parameters to the default defined by the application.

2.3 Functional Requirements

In this section, I want to exploit all the functional requirements of my application.

2.3.1 General Requirements

- The mobile application has to be used by many people from Polimi since it is an international university the app language is English;
- The mobile application has to start with a splash activity, while the application state is restored;
- The mobile application need user authentication, so it has also to provide a registration form;
- The mobile has to be designed to establish a connection with an *Arduino* board to retrieve weather and air data;
- The *Arduino* board must be developed to create a simple HTTP server to provide data coming from its sensors;
- The *Arduino* board must be developed to create an HTTP connection to retrieve the RTK GPS correction signal from the *SPIN3 GNSS NTRIPCaster*;

2.3.2 Authentication Requirements

- *Login screen* should be accessible only to users not currently logged in;
- Authentication screens should allow users to register or login using email and password;
- Authentication screens should redirect the user to the *Arduino screen* if the authentication is successful.

2.3.3 Arduino Requirements

- *Arduino Screen* should display the data collected from the *Arduino* board;
- *Arduino Screen* should provide a way to fetch manually the data from the board or provide a tool to an auto-fetching mode;

- *Arduino Screen* should provide a way to send the data retrieved by the *Arduino* board to the remote server (if a connection is available) ;
- *Arduino Screen* should allow the user to decide which type of data (live or remote stored) he/she wants to visualize in the *Map Screen* and *Chart Screen*.

2.3.4 Map Requirements

- *Map Screen* should display the data from ARPA dataset and *Arduino* board (live data or remote one);
- *Map Screen* should clear distinguish ARPA data (red circles) from *Arduino* data (blue circles);
- *Map Screen* should provide a way to allow the user to filter the data he/she wants to visualize (it has to allow the user to go to the relative *Filter Screen*);
- *Map Screen* should display to the user if the page is in a loading mode to refresh the data;
- *Map Screen* should provide a tool to manually refresh the page.

2.3.5 Chart Requirements

- *Chart Screen* should display the data from ARPA dataset and *Arduino* board (live data or remote one);
- *Chart Screen* should display the data in a linear graph, ordered by timestamp;
- *Chart Screen* should clear distinguish ARPA data (red line in the graph) from *Arduino* data (blue line in the graph);
- *Chart Screen* should display the data dispersion (tabulated as 1st, 2nd and 3rd quartile);
- *Chart Screen* should provide a way to allow the user to filter the data he/she wants to visualize (it has to allow the user to go to the relative *Filter Screen*);
- *Chart Screen* should display to the user if the page is in a loading mode to refresh the data;
- *Chart Screen* should provide a way to manually refresh the page.

2.3.6 Filter Requirements

- *Filter Screen* should display the current selected filter parameters;
- *Filter Screen* should provide a way to select a measure to visualize between Temperature, Humidity, Pressure, Altitude, TVOCs, eCO₂, PM_{0.5}, PM₁, PM_{2.5}, PM₄ and PM₁₀;
- *Filter Screen* should provide a way to select a starting and end date to give a range of time at the data to visualize;
- *Filter Screen* should provide an easy way to show or not to show ARPA data;
- *Filter Screen* should provide a way to select the ARPA station where ARPA data are downloaded;

2.3.7 Settings Requirements

- *Settings Screen* should display and allow to change the account information:
 - It should display and change email, name, surname or birthday of the user;
 - It should provide a method to change the account password (with double check confirmation);
 - It should allow the user to delete him/her account;
 - It should allow the user to log-out.
- *Settings Screen* should display and allow to change the IP address and port of the *Arduino* board;
- *Settings Screen* should display and allow to change the IP address and port of the server;
- *Settings Screen* should display and allow to change the ARPA data links:
 - It should display and provide a method to change the link to the GeoJSON API of the *Regione Lombardia* weather and air stations;
 - It should provide a way to select the ARPA station that a user wants to see in the *Filter Screen*;
 - It should display and provide a method to change the link to the JSON API of the *Regione Lombardia* weather and air data;

2.4 Non-Functional Requirements

These are the non-functional requirements that specify criteria that can be used to judge the operations of the application.

- **Availability** the services must be always up and running. In case of failure it must be restored as soon as possible;
- **Extensibility** the application was developed trying to keep a simple structure in order to allow further extensions easily;
- **Maintainability** the code must be clear, readable and with explicative comments to allow future maintenance;
- **Nice User Interface** nowadays is very important to have a cleaned and simple design. The application was developed following the Apple Human Interface Guidelines;
- **Portability** cross-platform implementation allows to use the application both in Android and iOS systems;
- **Reliability** all the data must be trusted and they could not be modified by anyone. The remote database must be protected and the connection between client and server must be handled with an HTTPS protocol;
- **Scalability** the system (considering client and server) should always be available to be used. System failures and server-side crashes must be avoided;
- **Usability** the application was developed to make the user interface as simple as possible keeping all the functionalities needed to provide the best user experience;

Chapter 3

Architectural Design

3.1 Overview

The main high-level components of the system are structured in four layers, as shown in Figure 3.1 below.

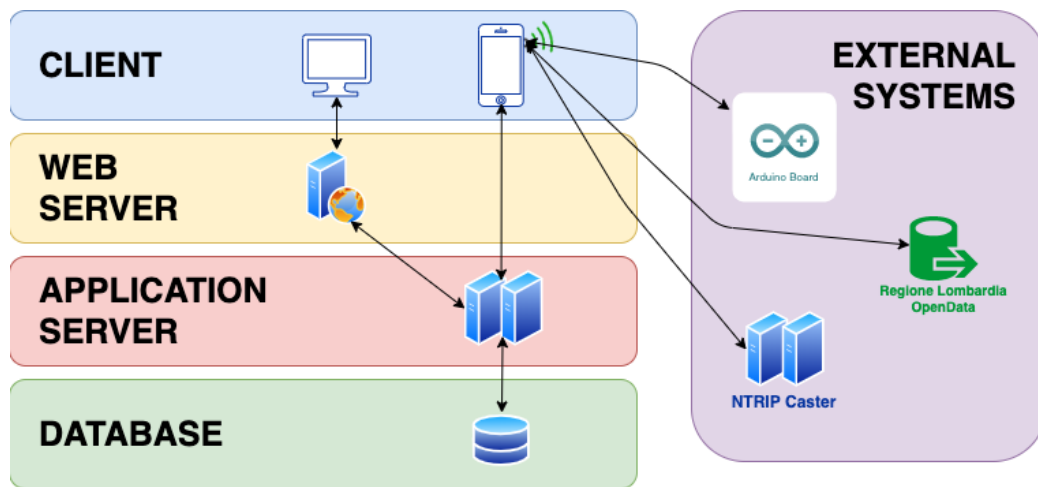


Figure 3.1: *Layered Structure* of the system

The considered high-level components are:

Mobile Application

The *Presentation Layer* dedicated to mobile devices. It communicates with the Application Server to retrieve and send data measured by the *Arduino* board. It contains a wide part of the *Logic Layer* to render data in different ways.

The communication with the *Arduino* board is handled with the phone hotspot service; moreover, *Arduino* board could retrieve the RTK GPS correction with a TCP connection with the NTRIPCaster.

Web Browsers

The *Presentation Layer* dedicated to web browsers; it communicates directly with the Web Server.

Web Server

It communicates with the Application Server and with the Web Browsers. This is the layer that provides web-pages for the Web Browser; currently, it provides only the SwaggerUI documentation of the RestfulAPI.

Application Server

This is the layer in which is contained a part of the *Logic Layer* to manage and store users information and collected data with the board; it communicates directly with the Mobile Application, the Web Server and the Database.

Database

The *Data Layer* of the system; it includes all structures and entities responsible for data storage and management. It communicates with the Application Server.

In the Figure 3.1, *Web Server* and *Application Server* are separated because they represent different function of the application.

As mentioned before, it is not implemented a *Web Application* on the *Web Server*, so, in the current setting they are merged.

If in future a *Web Application* will be implemented it could be useful to separate the two services in order to allow greater scalability.

3.2 Component View

In this section the system will be described in term of its components: their functionalities will be discussed and detailed.

Moreover, interfaces among components and among external systems will be shown.

3.2.1 Database

The application database is managed using a Relational DBMS.

It allows the reading of data, ensuring to the users the possibility to log in, access the application of interest and check the stored data. It is also used for data manipulation (insertion, modification and deletion). The use of a Relational DBMS guarantees the fundamental properties for a database of this type:

- *Atomicity*: no partial executions of operations;
- *Consistency*: the database is always in a consistent state;
- *Isolation*: each transaction is executed in an isolated and independent way;
- *Durability / Persistence*: changes made are not lost.

The database offers to the Application Server an interface that it can use to interact with it. Particular attention must be paid to the encryption of passwords used to the user access to the system. Below is the designed E-R diagram (Figure 3.2).

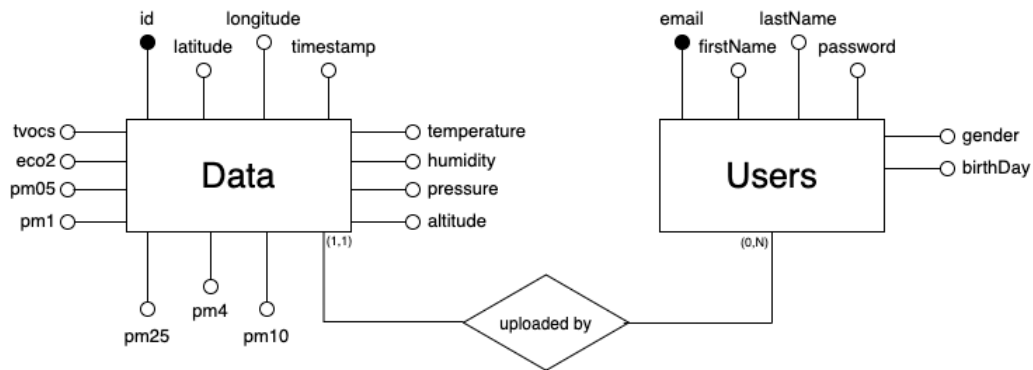


Figure 3.2: *Entity-Relationship* Diagram of the Database

3.2.2 Application Server

The main feature of the *Application Server* is to define rules and work-flows of all the functionalities defined by the RestfulAPI.

The *Application Server* must have interfaces to communicate with the *Mobile Application* and also with the DBMS; the communication must be done through HTTPS protocol.

In the brief introduction below, logic modules and their descriptions are presented, while all the connections among them can be seen in the *Global Component View* in Figure 3.3.

Authentication Service

This module handles the authentication and authorization process. It generates authorization tokens and checks their validity.

Arduino Data Controller

This module receives the requests at the *Arduino Data* endpoint, it checks the authorization using the interface with the *Authentication Service* and pass them to the *Arduino Data Service*.

Arduino Data Service

This module manages the requests for *Arduino Data* interfacing itself with the *Data Service*.

Data Service

This module is the unique interface of the *Application Server* to the database's DBMS.

User Controller

This module receives the requests at the *User* endpoint, it checks the authorization using the interface with the *Authentication Service* and pass them to the *User Service*.

User Service

This module manages the requests for a *User* interfacing itself with the *Data Service*.

3.2.3 Web Server

As already explained before, the *Web Server* is a simple service. It provides a simple web-page to redirect the user to the *SwaggerUI* documentation.

The *Web Server* is hosted in the same space of the *Application Server*; so the connections are done with the HTTPS protocol.

3.2.4 Mobile Application

The *MOQA* mobile application communicates with the *Application Server* through RestfulAPIs that are defined in order to describe the interactions between the two layers and that must be independent from the two implementations.

The application manages the connection with the *Arduino* board to get air and weather measures and send these data to the *Application Server*. Moreover, it manages the connection with the *Regione Lombardia OpenData* service to retrieve ARPA data.

The detailed behaviour of the components in the mobile application is discussed in the Section 3.5.2.

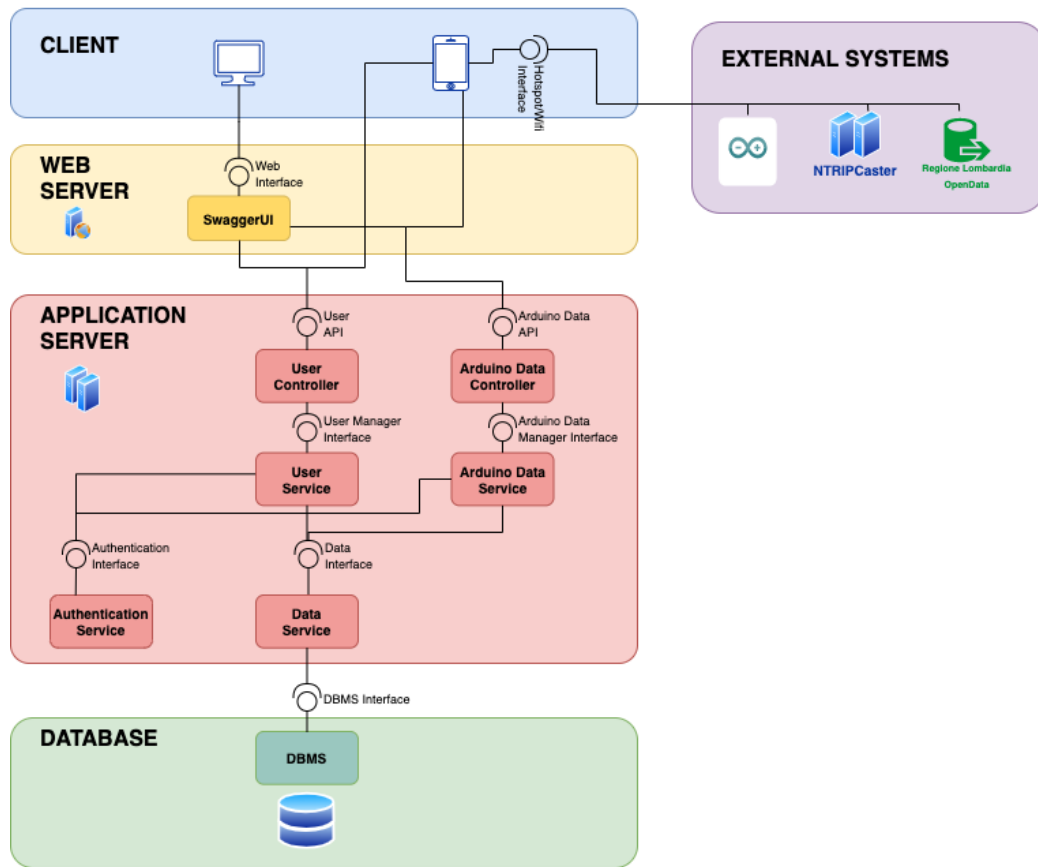


Figure 3.3: *Global Component View*

3.3 Deployment View

Below is the deployment diagram of the system (Figure 3.4).

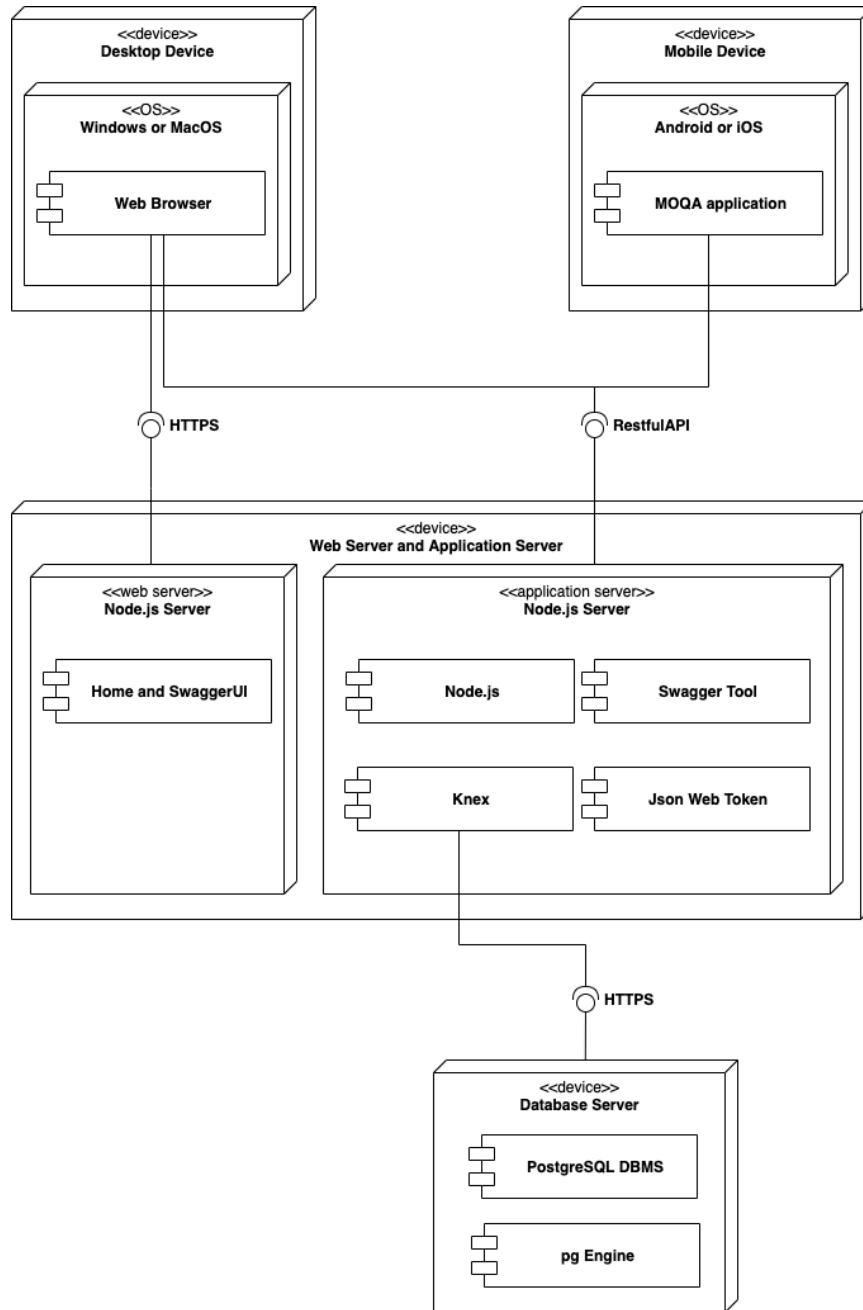


Figure 3.4: *Deployment Diagram* of the system

3.4 Runtime View

The aim of this section is to specify the behaviour of the system. Some relevant cases are selected and exploit using *Sequence Diagrams*.

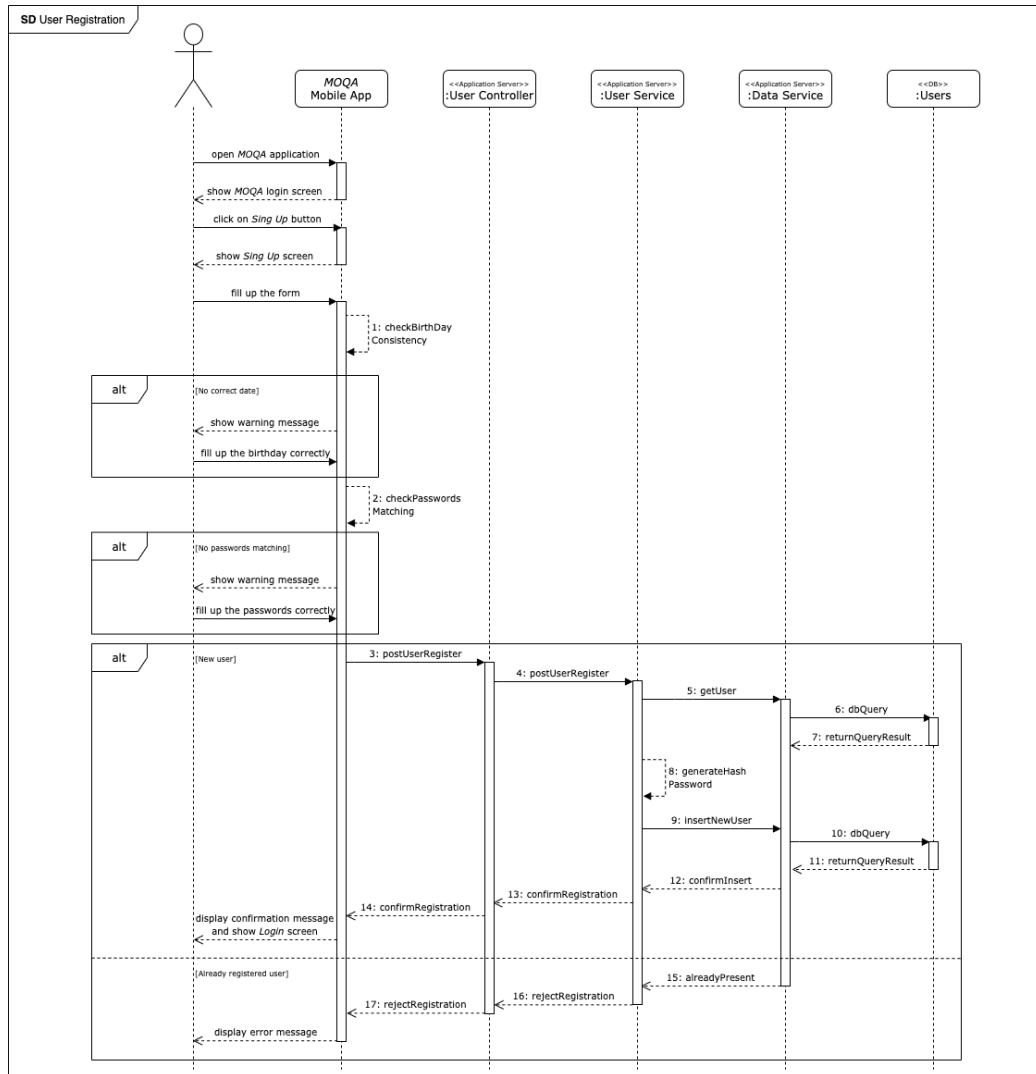


Figure 3.5: *User Registration* sequence diagram: it describes the process through which a user can sign-up itself in the system

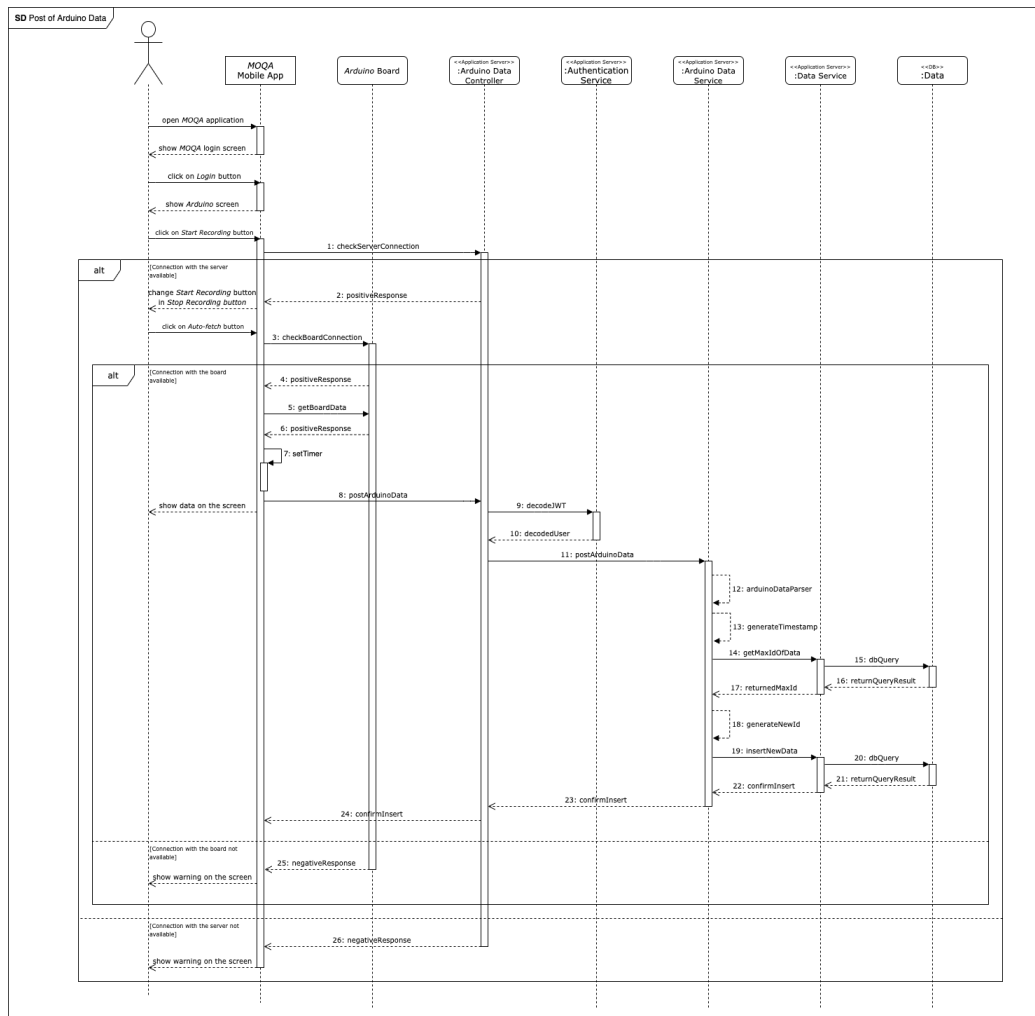


Figure 3.6: *Post of Arduino data* sequence diagram: it describes the process through which a user can store in the *Application Server* data detected with the *Arduino* board

3.5 Component Interfaces

This section contains some detailed information about the interfaces between the components of the system.

The following description focuses the attention over the actions performed by a procedure of a certain component after that it is called by other components.

Huge part of these actions could be seen played in the *Sequence Diagrams* (Section 3.4).

3.5.1 Application Server

Authentication Service

Following procedures are called by other components:

- **generateJWT** This method allows to generate a token for the authentication. The token must be added in the requests as *Header* attribute.
- **decodeJWT** This method allows to get and decode the token, if present, in a request. It returns the object with the information of the.

Arduino Data Controller

Following procedures are called by other components:

- **getData** This method is called when a request is received at the corresponding endpoint. It calls the *Authentication Service* to check the session. If the session is valid, it retrieves the parameters in the request and calls the corresponding method in the *Arduino Data Service*.
- **getDataByDate** This method is called when a request is received at the corresponding endpoint. It calls the *Authentication Service* to check the session. If the session is valid, it retrieves the parameters in the request and calls the corresponding method in the *Arduino Data Service*.
- **postData** This method is called when a request is received at the corresponding endpoint. It calls the *Authentication Service* to check the session. If the session is valid, it retrieves the parameters in the request and calls the corresponding method in the *Arduino Data Service*.

Arduino Data Service

Following procedures are called by other components:

- **getData** This method is called by the *Arduino Data Controller*. It retrieves the data preparing the query to submit to the *Data Service*.
- **getDataByDate** This method is called by the *Arduino Data Controller*. It retrieves the data preparing the query to submit to the *Data Service*.
- **postData** This method is called by the *Arduino Data Controller*. It retrieves the data preparing the query to submit to the *Data Service*.

Data Service

Following procedure is called by other components:

- **databaseInit** This method is called during the initialization of the *Application Server* to create the tables in the database and initialize them with the initials data.

Moreover, it provides an object that has own methods to query the database through the DBMS service.

User Controller

Following procedures are called by other components:

- **deleteUserMe** This method is called when a request is received at the corresponding endpoint. It calls the *Authentication Service* to check the session, if the session is valid it retrieves, from the session, the user email and calls the corresponding method in the *Arduino Data Service*.
- **getUserMe** This method is called when a request is received at the corresponding endpoint. It calls the *Authentication Service* to check the session, if the session is valid it retrieves, from the session, the user email and calls the corresponding method in the *Arduino Data Service*.
- **postUserLogin** This method is called when a request is received at the corresponding endpoint. It retrieves the parameters in the request and calls the corresponding method in the *Arduino Data Service*.
- **postUserLogout** This method is called when a request is received at the corresponding endpoint. It calls the *Authentication Service* to check the session, if the session is valid it logout the user.

- **postUserRegister** This method is called when a request is received at the corresponding endpoint. It retrieves the parameters in the request and calls the corresponding method in the *Arduino Data Service*.
- **putUserMe** This method is called when a request is received at the corresponding endpoint. It calls the *Authentication Service* to check the session. If the session is valid, it retrieves the parameters in the request, checks the consistency* and calls the corresponding method in the *Arduino Data Service*.

*: the consistency is computed checking if the session email is the same in the body in the PUT request.

User Service

Following procedures are called by other components:

- **deleteUserMe** This method is called by the *User Controller*. It retrieves the data preparing the query to submit to the *Data Service*.
- **getUserMe** This method is called by the *User Controller*. It retrieves the data preparing the query to submit to the *Data Service*.
- **postUserLogin** This method is called by the *User Controller*. It retrieves the data preparing the query to submit to the *Data Service*.
- **postUserRegister** This method is called by the *User Controller*. It retrieves the data preparing the query to submit to the *Data Service*.
- **putUserMe** This method is called by the *User Controller*. It retrieves the data preparing the query to submit to the *Data Service*.

3.5.2 Mobile Application

The next diagram shows the flow of operation triggered by an actor that wants to perform a task. The actor could be the user or the system. The *Use Case Diagram*, in Figure 3.7, exploits the main operations that could be performed in the system.

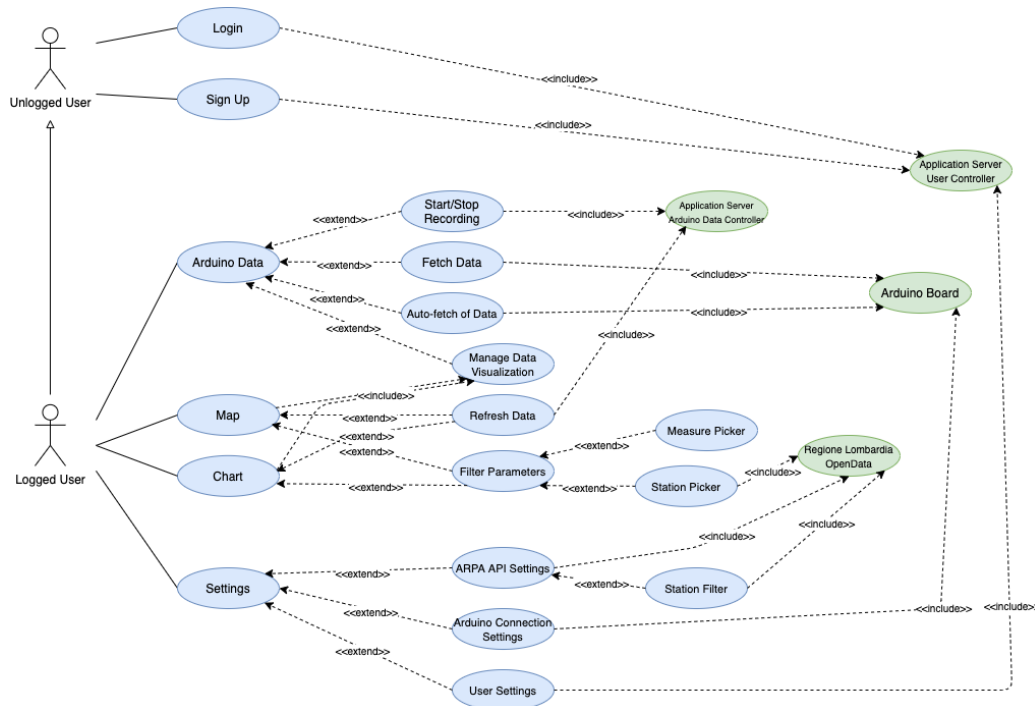


Figure 3.7: *Use Case Diagram* of the system

3.6 Architecture styles and patterns

Following there are the choices made regarding architectural styles and patterns.

3.6.1 Client and Server

A client-server architecture has been chosen for the system implementation: it allows the exchange of information between Internet-connected devices used by users and the centralized server that collects data and offers services (defined with a RestfulAPI).

In particular:

- The *MOQA* mobile application installed on users' phone is connected to the *Application Server* that receives and sends data.
- The *Application Server* is seen as a server by the *MOQA* application installed on smartphones and as a client from the database that receives

and processes its requests (queries).

- The database is seen as a server by the application server that sends the requests and waits for the answers.

3.6.2 Thick Client

The *MOQA* mobile application has to be considered a thick client: in the app, in addition to the graphical interface and the logic related to the acquisition of user data, there is also the logic that allows the processing of Arduino and ARPA data that must be shown in a map and on a chart. Thanks to this, the app has the function to collect data from the *Arduino* board and to allow the user to see the data distribution and make a comparison between Arduino and ARPA data.

3.6.3 Model-View-Control

The MVC model identifies three main components in the management of a client-server application. They are the Model, the View and the Controller. The following diagram shows the management of these three levels according to the use of thick clients (Figure 3.8).



Figure 3.8: Management of the client-server connection according to the MVC model

3.7 Other Design Decisions

3.7.1 User authentication

In order to guarantee the upload of *Arduino* data to trusted users, the access to the system takes place by entering a username and password. The username used for the access coincides with the email address used during registration.

3.7.2 Password storing

The passwords given to the users are saved in the database after being encrypted using a SHA512 algorithm.

This way, the passwords that have to be used to access to *MOQA* and the *Application Server* services are not stored in plain-text in the user's database, but they are the result of a transformation from which, even with very powerful computers, it is very difficult to trace the original ones.

This way if an intruder succeeds in picking up the string containing the password, he won't be able to trace the password used to access the service.

3.7.3 Security in the transfer of information

The transfer of data between the clients and the server is done using the most modern data encryption protocol, HTTPS.

Chapter 4

User Interface Design

4.1 Overview

In this section are present some screenshots by *MOQA* mobile application. During the design and development, the attention was focused on the mobile application, even if the application is developed to adapt itself to a larger screen. Layout, dimensions and colours are picked by the Apple Human Interface Guidelines.

4.2 Mock-up

At the beginning of the project, it was provided to the stakeholders a *mock-up* (Figure 4.1) of the application to have an approval that the draft was meeting the requirements and to show an overview of the flow of the application.

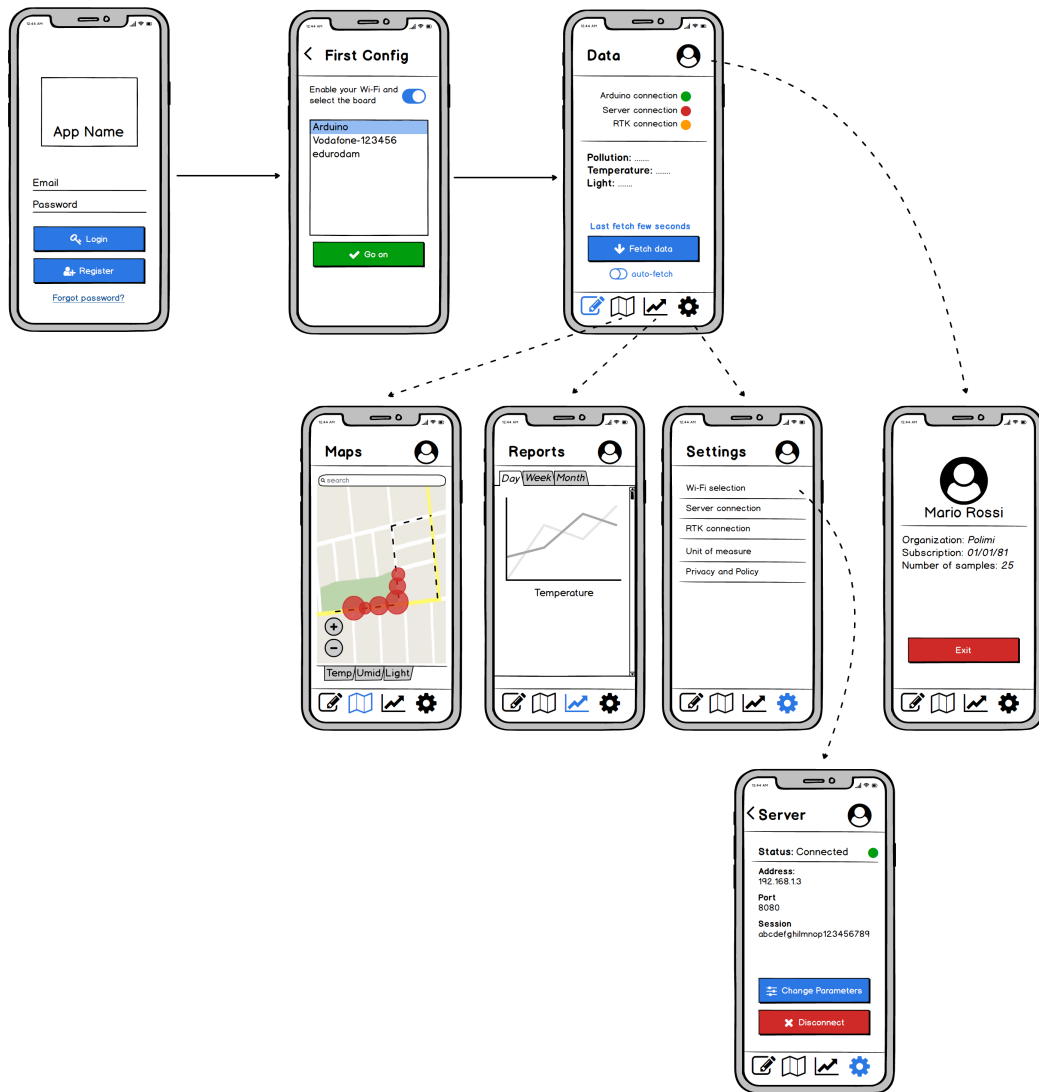


Figure 4.1: *MOQA* initial mock-up

4.3 Screens



Figure 4.2: **Splash Screen**

The **Splash Screen** welcomes the user when the application is starting; meanwhile it restores the state of the application or loads the default one.

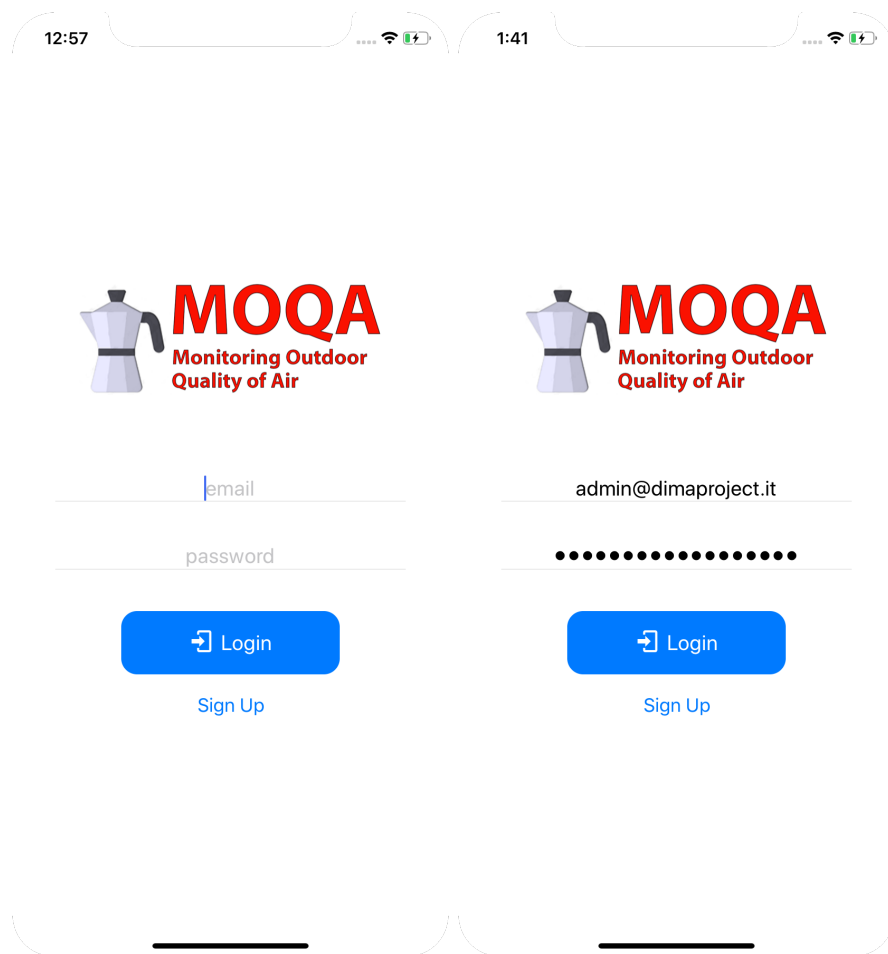
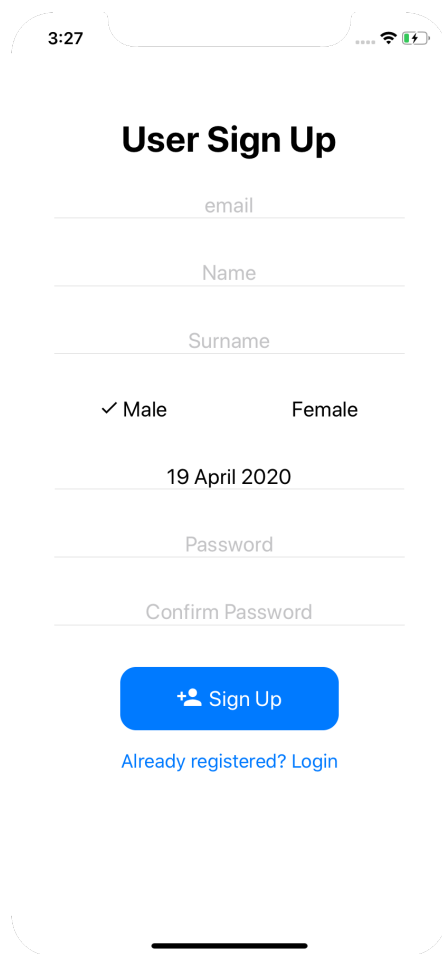


Figure 4.3: Login Screen

The **Login Screen** is the first screen that the user sees after the application is loaded. The user could enter its credential or, if they are in memory, the application fills the fields.



A mobile application mockup for a 'User Sign Up' screen. The screen is white with a light gray header and footer. The header shows the time '3:27' and status icons for signal, Wi-Fi, and battery. The title 'User Sign Up' is centered in bold black text. Below the title are five input fields with light gray placeholder text: 'email', 'Name', 'Surname', '19 April 2020' (for a date), and 'Password'. Below the date field is a row with two radio buttons; the first is selected and labeled 'Male', and the second is labeled 'Female'. Below the password field is a 'Confirm Password' field. A blue rounded rectangular button with a white user icon and the text 'Sign Up' is centered below the form. Below the button is a blue link that says 'Already registered? Login'. The footer is a solid dark gray bar.

3:27

User Sign Up

email

Name


Surname

☒ Male ☐ Female

19 April 2020

Password

Confirm Password

 Sign Up

[Already registered? Login](#)

Figure 4.4: **Sign Up Screen**

The **Sign Up Screen** allows the user to register in the application.

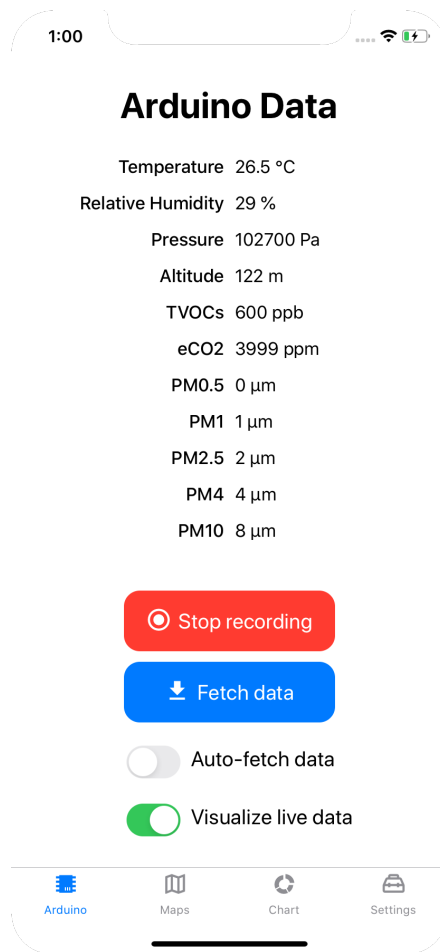


Figure 4.5: **Arduino Screen**

The **Arduino Screen** displays the data coming from the *Arduino* board, if available.

It allows the user to start/stop the recording of data (it means that the data gathered by the board are sent to the server).

It allows the user to *Fetch data* (it is a manual request of data to the board) or enable the *Auto-fetch data* that starts a routine that every second refreshes the data from the board.

Moreover in this screen, the user could decide if it wants to visualize the live data from the board or the data stored in the server in the **Map Screen** and **Chart Screen**.

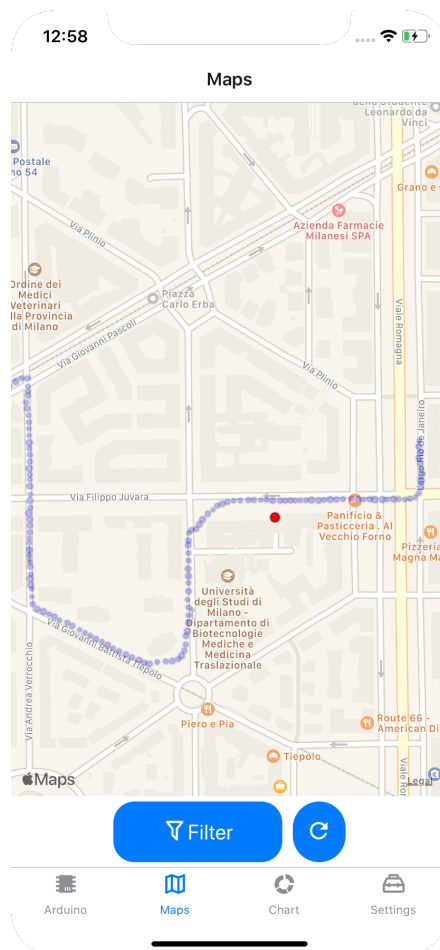


Figure 4.6: Map Screen

The **Map Screen** displays the data coming from the *Arduino* board (blue circles) and *ARPA* dataset (red circles) on a map. The radius of a circle depends on the value of the data it is associated.

The user could filter the data clicking on the *Filter* button and could refresh the map with the blue button on the right.

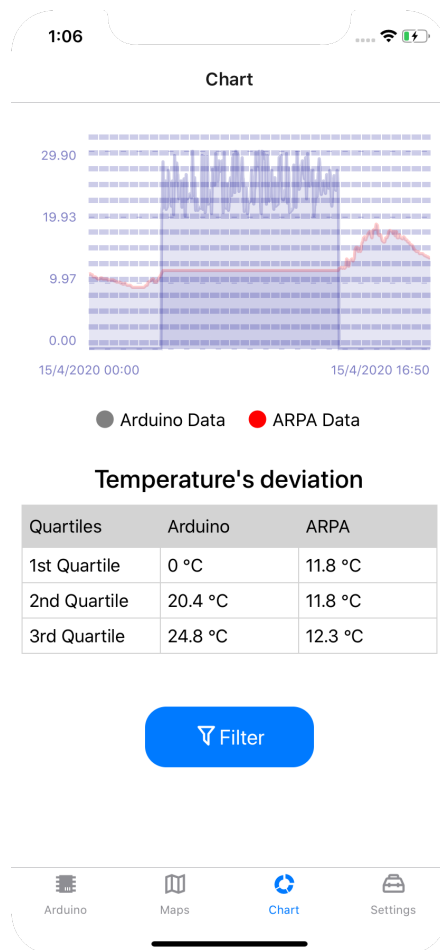


Figure 4.7: **Chart Screen**

The **Chart Screen** displays the data coming from the *Arduino* board and *ARPA* dataset on a graph. Moreover, it provides a table with a computation of the quartile deviation.

The user could filter the data clicking on the *Filter* button and could refresh the page scrolling-up.

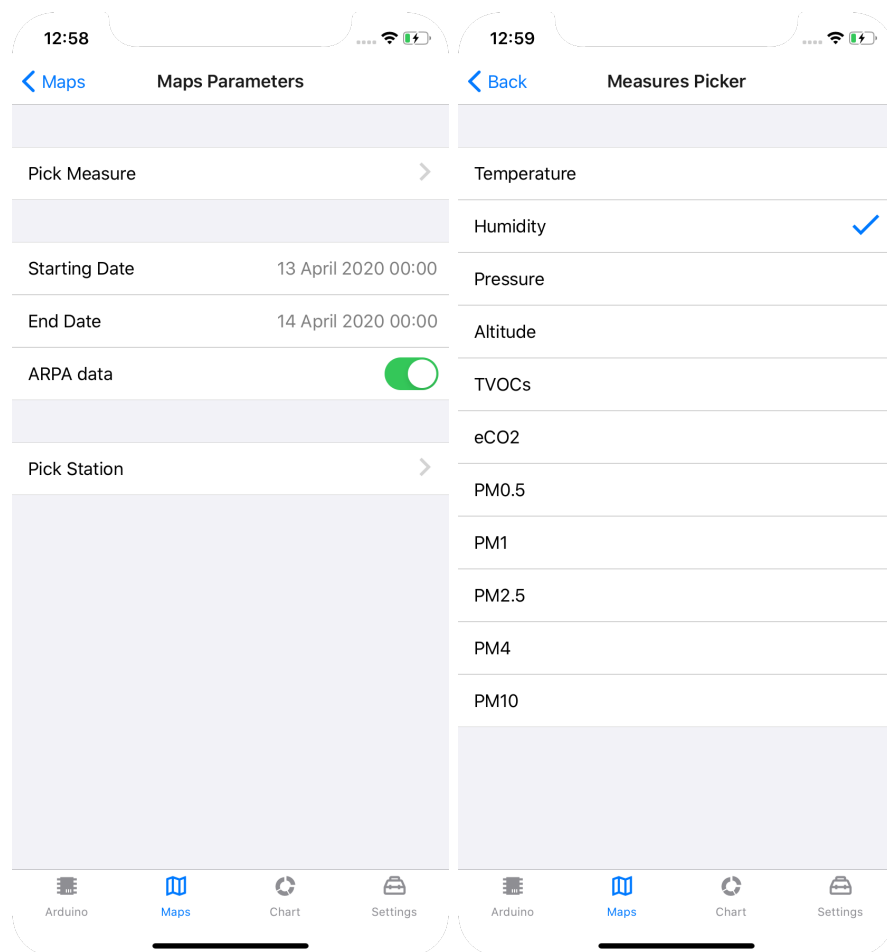


Figure 4.8: **Filter Screens**

The **Filter Screens** allow the user to set the parameters to limit the data visualized in the **Map Screen** and **Chart Screen**.

The user could pick the measure it wants to visualize, select the time range and decide to visualize the ARPA data defining the station.

In the application the **Filter Screen** is used by the **Map Screen** and the **Chart Screen**

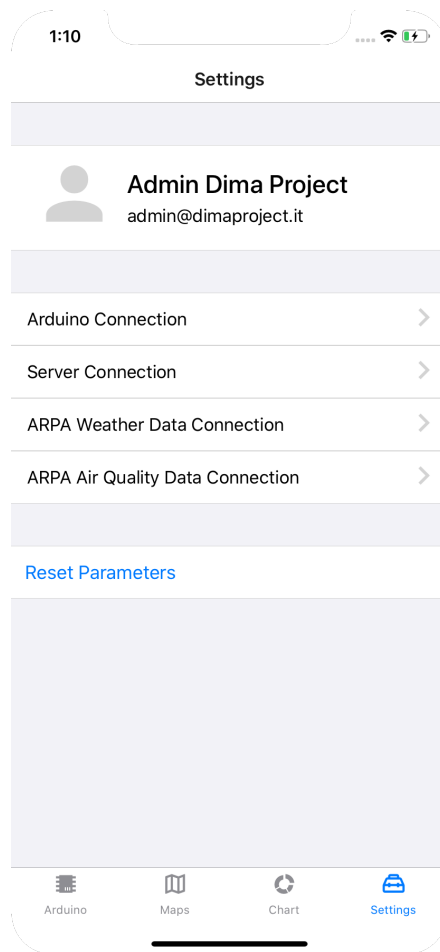


Figure 4.9: **Settings Screen**

The **Settings Screen** allows the user to visualize and to change different parameters of the application.
The user could also decide to reset the parameters to an initial and default state.

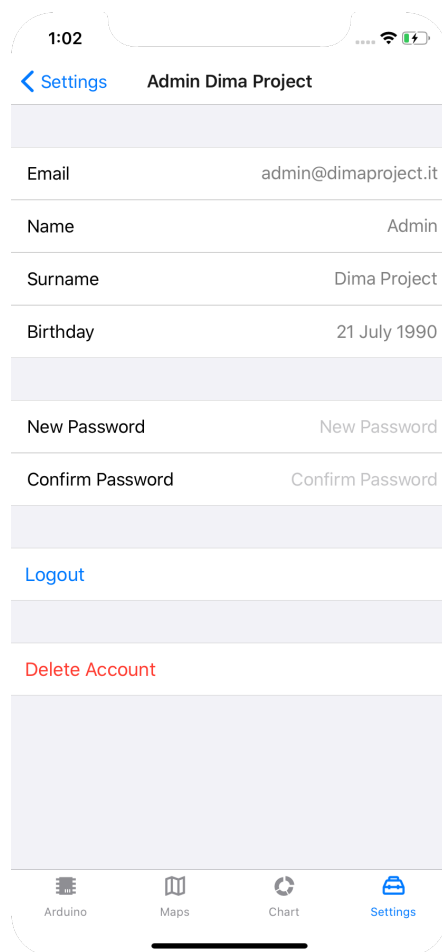


Figure 4.10: **User Settings Screen**

The **User Settings Screen** allows the user to visualize and to change the information about its account. Moreover it allows the user to delete its account or logout from the application.

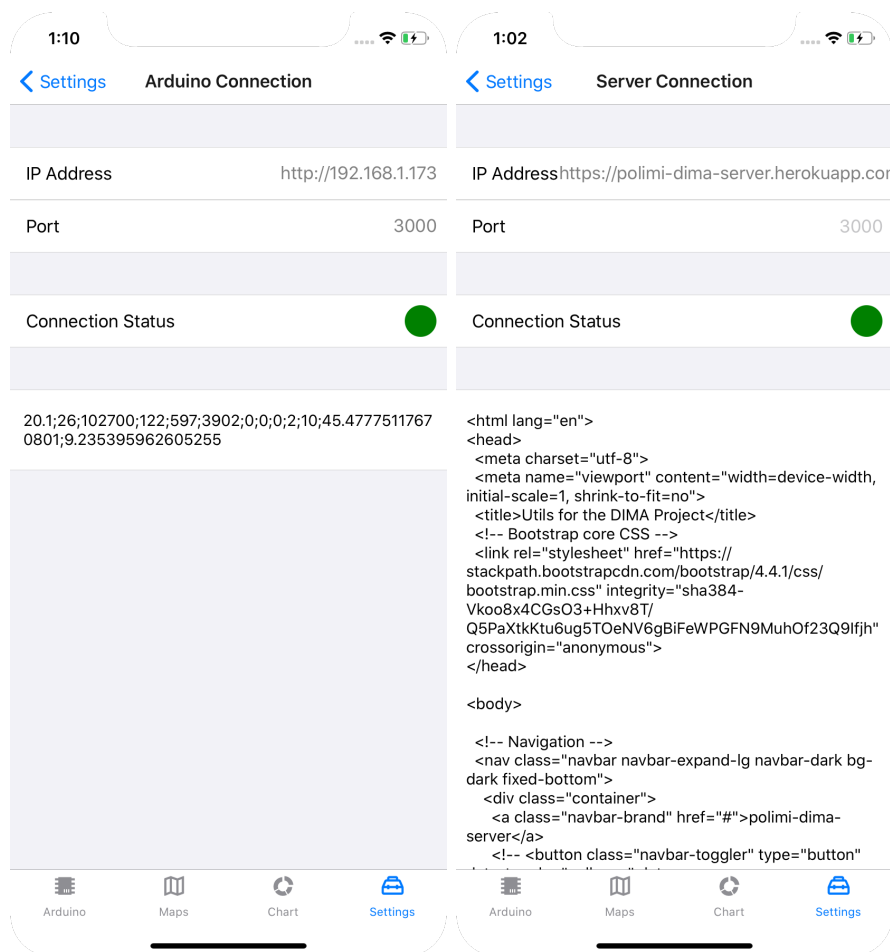


Figure 4.11: Connection Screens

The **Arduino Connection Screen** and **Server Connection Screen** allow the user to change the IP address and the port of the *Arduino* board and the *Application Server*.

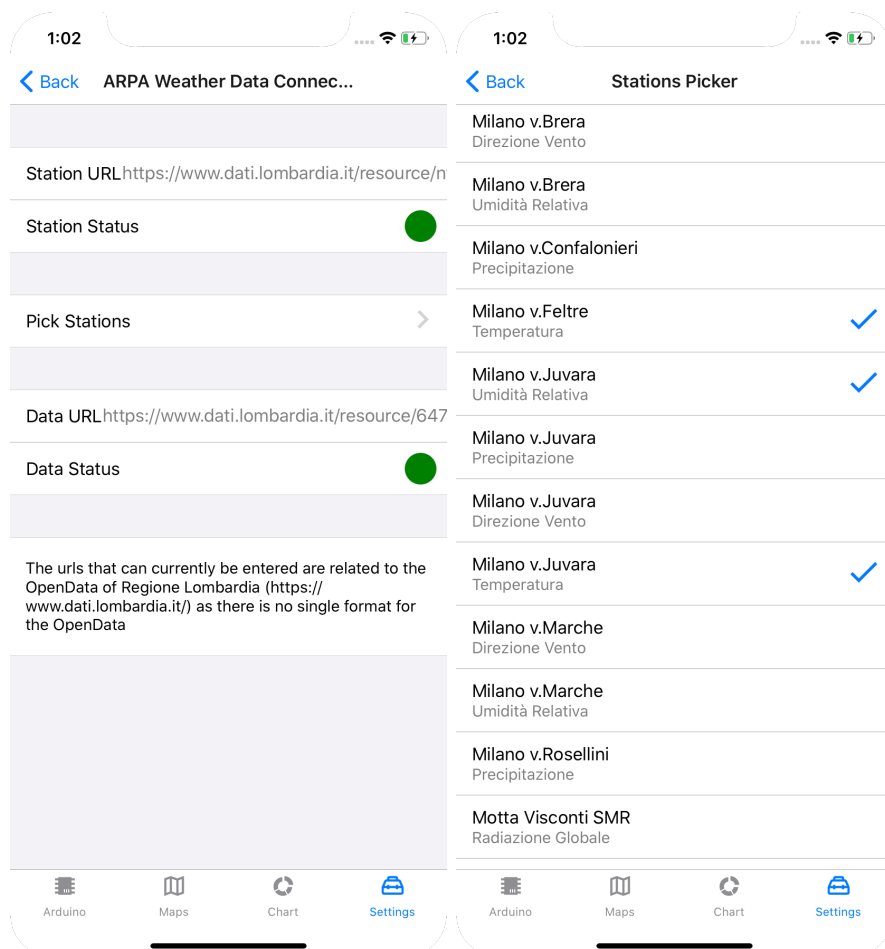


Figure 4.12: ARPA Data Connection Screen

The **ARPA Data Connection Screen** allows the user to visualize and change the links of the stations and the data from *Regione Lombardia* OpenData portal.

Moreover with the **Stations Picker Screen** allows the user to select the stations it wants to visualize in the **Pick Station Screen** of **Filter Screen**.

In the application there are two different *screens*: one for the weather links and an other for the air quality links; however they are managed by the same *Vue Native* component

Chapter 5

Frameworks, External services and Libraries

To develop my application I used *Vue Native* [<https://vue-native.io/>]; it is a Javascript framework to build cross-platform application that inherits the main features from *Vue.js* and *React Native*.



Figure 5.1: *Vue Native* logo

As in *React Native*, *Vue Native* applications could be built using *Expo* or *React Native CLI*.

Expo is designed to allow developers to quickly set up and develop React Native apps, without having to configure Xcode or Android Studio. Since it is most suitable for who come from a Web background, I decided to use it.

I also used several external services and libraries. Some external libraries are necessary for the correct behavior of the application and to have not to waste time in implementing code in order to achieve the objective that these external services already provide.

Now, I will present the main external services and libraries that I have used and integrated in *MOQA*.

5.1 Vuex

Vuex is a state management pattern library for *Vue.js* [<https://vuex.vuejs.org/>] applications. It serves as a centralized store for all the components in an application, with rules ensuring that the state can only be mutated in a predictable fashion. It is based on the state store manager library *Redux* of *React Native*.

I have made a custom change to manage the persistence on the phone, using *AsyncStorage* from *React Native* instead of the *localStorage* used in the web.

```
1 import {AsyncStorage} from 'react-native';
2 import store from './index';
3
4 const storageKey = '@MySuperStore';
5
6 // Function to store the state in the application local
  storage in the phone
7 export async function storeData(obj){
8   try {
9     var copiedObj = JSON.parse(JSON.stringify(obj));
10    delete copiedObj.blob;
11    // Saving
12    await AsyncStorage.setItem(storageKey, JSON.stringify(
      copiedObj));
13    // DEBUG PURPOSE
14    await AsyncStorage.getItem(storageKey).then((result) => {
      console.log("CURRENT AsyncStorage:", result);});
15   } catch (error) {
16     // Error saving data
17   }
18 }
19
20 // Function to get the state saved in the application
  local storage in the phone
21 export async function retrieveData(){
22   try {
```

```

23     await AsyncStorage.getItem(storageKey)
24     .then((result) => {
25         var obj = JSON.parse(result);
26         store.commit('REPLACE',obj);
27     });
28   } catch (error) {
29     // Error retrieving data
30   }
31 }
32
33 // Function to delete the stored state
34 export async function deleteDate(){
35   try {
36     await AsyncStorage.removeItem(storageKey)
37     .then((result) => {
38       console.log(result);
39     });
40   } catch (error) {
41     // Error retrieving data
42   }
43 }

```

5.2 Maps, Charts and DateTimePicker

Vue Native has not its own component to build a map or a chart. So, as explained in the official documentation, I used the community components of *React Native*.

It was easy to import the components and the integration does not require any type of effort. Making an example, in the *<script>* section of the code I had to insert:

```

1  // Import of MapView from react-native
2  // react-native-maps is the library maintained
3  // by the react-native community
4  import MapView, {Circle} from "react-native-maps";
5
6  // Export of the component in Vue Native
7  export default {
8    components: {
9      MapView, Circle, Icon
10    },
11
12  }

```

Using the above code, in the `<template>` section, we can use `<map-view>` as component of *Vue Native*.

5.3 Application Server

According to the other two people involved in the project, we decided to develop an online RestfulAPI to manage the storage and the retrieve of the data generated by the Arduino board.

This RestfulAPI was designed using Swagger [<https://swagger.io/>], developed using *Node.js* [<https://nodejs.org/>] framework and delivered with the web-platform *Heroku* [<https://www.heroku.com/>].



Figure 5.2: *Heroku* logo

The web server is available at <https://polimi-dima-server.herokuapp.com/>.

All the backend is developed using Javascript code and the default *npm* as package manager.

The starting point with function declarations and interfaces was generated through the *Swagger* file using the *Swagger Editor* [<https://editor.swagger.io/>] service.

Relevant package added are:

- *node-postgres* (aka *pg*) is a collection of Node.js modules for interfacing with your PostgreSQL database;
- *knex.js* is a SQL query builder for Postgres designed to be flexible, portable, and easy to use;

- *serve-static* creates a new middleware function to serve files from within a given root directory. The file to serve will be determined by combining req.url with the provided root directory;
- *JSON Web Tokens* (aka jwt) is standard method for representing claims securely between two parties. The package allows you to decode, verify and generate JWT .

5.4 ARPA Data

ARPA data are not available from the official web-site of the institute, but they are provided by the OpenData service offered by *Regione Lombardia* [<https://www.dati.lombardia.it/>].

Regione Lombardia offers a web tool to visualize, aggregate and export data; unlimited downloads of datasets in different formats (e.g. csv, json, ecc...). Moreover, it offers a web API to access a specific dataset.

Data about the monitoring stations are returned as a *GeoJSON* file. The file contains an array of sensors, each one associated to a station, its position, its typology, unit of measure and altitude.

Sampled data are returned as objects in an array. For each datum is present the id number of the sensor, the value and the timestamp.

Chapter 6

Test Cases

This section describes the result of the main test done on *MOQA* mobile application.

Test Case	<i>Sign Up</i>
Goal	Register a new user.
Input	In the Login screen the user clicks on the Sign Up button, fills-up the form and clicks on the Sign Up button.
Expected outcome	The new user is registered in the application.
Actual outcome	CORRECT: After the click on the Sign Up button in the Login screen the application shows the form to register a new user; after the click on the Sign Up button the application goes back in the Login screen that contains the created credentials. If the user is already registered or there are problems in the connection an alert is thrown.

Test Case	<i>Login</i>
Goal	Login in the application.
Input	In the Login screen the user inserts email and password, finally, clicks on the Login button.
Expected outcome	The user logs-in the application.
Actual outcome	CORRECT: After filling up the fields with email and password; after the click on the Login button the application goes to the Arduino screen. If the credentials are wrong or there are problems in the connection an alert is thrown.

Test Case	<i>Data Sampling</i>
Goal	Sample and send to the server Arduino data.
Input	The user logs-in in the application, clicks on Start Recording, then clicks on Fetch data or Auto-fetch data.
Expected outcome	Data are sampled and sent to the server.
Actual outcome	CORRECT: After the login in the application, a click on Start recording and Auto-fetch data: the application starts to automatically fetch data and send the to the server. If there are are problems in the connection with the server or with the board an alert is thrown.

Test Case	<i>Map Tracking</i>
Goal	Visualize live data on the map.
Input	The user logs-in in the application, enables on Auto-fetch data, enables Visualization of live data, goes to Map screen.
Expected outcome	At every new data from the board the Map screen refreshes.
Actual outcome	CORRECT: After the login in the application, enabling the Auto-fetch data and Visualize live data toggles: the application starts to automatically fetch data; going on the Map screen, every time a new data arrives the map is updated with a new circle.

Test Case	<i>Chart Tracking</i>
Goal	Visualize live data on the chart.
Input	The user logs-in in the application, enables on Auto-fetch data, enables Visualization of live data, goes to Chart screen.
Expected outcome	At every new data from the board the Chart screen refreshes.
Actual outcome	CORRECT: After the login in the application, enabling the Auto-fetch data and Visualize live data toggles: the application starts to automatically fetch data; going on the Map screen, every time a new data arrives the chart screen is updated.

Test Case	<i>Change User Information</i>
Goal	Change some user information.
Input	The user logs-in in the application, goes in the Settings screens, clicks on its name. In the new screen, the user changes the parameter it wants and then clicks on Make changes button to confirm.
Expected outcome	New data are correctly updated in the database and a message is notified to the user.
Actual outcome	CORRECT: After the login in the application and reached the User Settings screen the user fills some fields with new information. Clicked on Make changes, the application notify the user that data are updated.

Test Case	<i>Change User Password</i>
Goal	Change the account password.
Input	The user logs-in in the application, goes in the Settings screens, clicks on its name. In the new screen, the user inserts twice a new password and then clicks on Make changes button to confirm.
Expected outcome	New data are correctly updated in the database and a message is notified to the user.
Actual outcome	CORRECT: After the login in the application and reached the User Settings screen the user inserts the new password and the confirmation password (same). Clicked on Make changes, the application notify the user that data are updated.

Test Case	<i>Logout</i>
Goal	Logout from the application.
Input	The user logs-in in the application, goes in the Settings screens, clicks on its name. In the new screen, the user clicks on logout button.
Expected outcome	The application goes back to the login screen.
Actual outcome	CORRECT: After the login in the application, reached the User Settings screen and clicked on the logout button the application shows the Login screen.

Test Case	<i>Change Arduino Connection</i>
Goal	Change Arduino Connection.
Input	The user logs-in in the application, goes in the Settings screens, clicks on Arduino Connection. In the new screen, the user modifies IP Address and port.
Expected outcome	The application changes the parameters and update the status.
Actual outcome	CORRECT: After the login in the application, reached the Arduino Connection screen and modified the IP Address and port the user clicks on Make changes. The connection status color from red becomes green.

Test Case	<i>Pick Up a new Station</i>
Goal	Add the Abbiategrosso stations to the stations to considered for the weather data.
Input	The user logs-in in the application, goes in the Settings screens, clicks on ARPA Weather Data Connection. In the new screen, the user clicks on Pick Stations button and then selects the Abbiategrosso stations.
Expected outcome	The application puts a V near the new picked stations.
Actual outcome	CORRECT: After the login in the application, reached the ARPA Weather Data Connection screen and clicked the Pick Stations button the users picks the Abbiategrosso stations. The picked stations have a V; moreover going in the filter station picker the new stations are shown.

Chapter 7

Effort and Cost Estimation

7.1 Effort

To estimate the size of a project, *Lines of Code* (LOC) is usually used. It is a metric commonly understood, it permits specific comparison and it is easily measurable.

In this section is presented the report of LOC generated by GitHub.

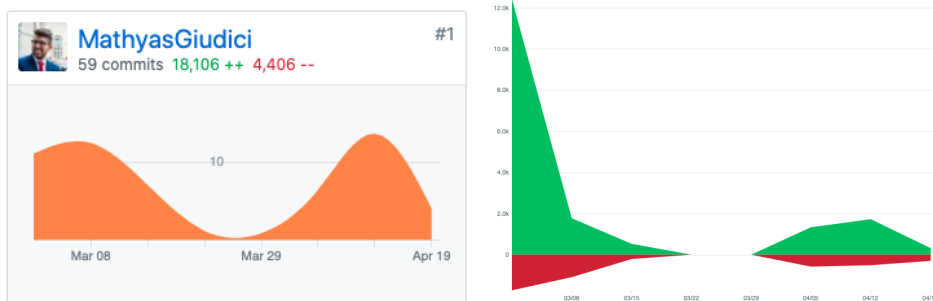


Figure 7.1: Screenshots from *MOQA* repository on *GitHub*

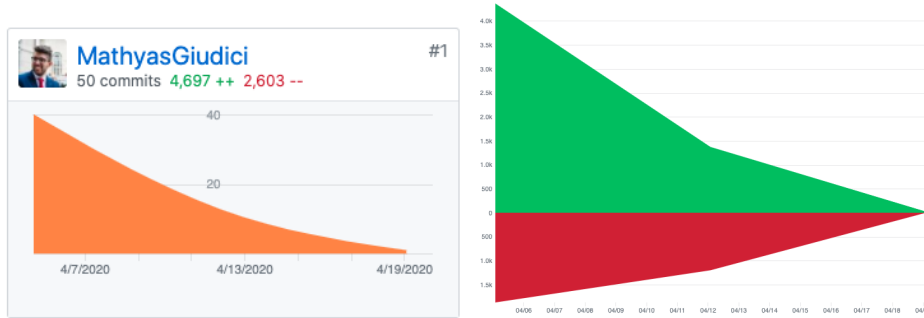


Figure 7.2: Screenshots from *Application Server* repository on *GitHub*

7.2 Cost Estimation

In this section is described a cost estimation based on the *COCOMO* method. The *Constructive Cost Model* (COCOMO) is an algorithmic software cost estimation model developed by Barry W. Boehm.

The model uses a basic regression formula with parameters that are derived from historical project data and current project characteristics.

COCOMO applies to three classes of software projects:

- **Organic projects** *small* teams with *good* experience working with *less than rigid* requirements;
- **Semi-detached projects** *medium* teams with mixed experience working with a mix of rigid and less than rigid requirements;
- **Embedded projects** projects are developed within a set of *tight* constraints. It is also combination of organic and semi-detached projects.

The basic *COCOMO* equations are:

- **Effort Applied** $a(KLOC)^b$
- **Development Time** $c(\text{Effort Applied})^d$

where, *KLOC* is the estimated number of delivered lines (expressed in thousands) of code for project.

The coefficients a, b, c and d are:

Software Project	a	b	c	d
Organic	2.40	1.05	2.50	0.38
Semi-detached	3.00	1.12	2.50	0.35
Embedded	3.60	1.20	2.50	0.32

For the project, I decided to select to consider the project as *Semi-detached*. This consideration comes from a consideration about the team (very small I'm alone), the development that is done with *Vue Native* (first time with *Vue Native* but I already know *Vue.js*) and the fact that I never program an *Arduino* board.

From the previous section, the reader could see that the KLOC value could be fixed at 15.

$$EFFORTAPPLIED = a (KLOC)^b = 3.00 (6)^{1.12} = 22.3$$

$$DEVELOPMENTTIME = C (EFFORTAPPLIED)^d = 2.50 (22.3)^{0.35} = 7.4$$

The *COCOMO* method estimates the time of development in more or less 7 months.

Chapter 8

Future Works

In this section I want to analyse the main future works that the project needs.

- Since the server is working only as RestfulAPI, as future work it could be considered to implement a web-interface to provide the features now available only with the mobile application.
- Now in the *Chart Screen* the visualized data are from a single measure compared with the corresponding available ARPA data from a pinned station; It could be design a system to visualized data coming from different stations or correlate different measures available.

Appendix A

Appendix

A.1 Software and Tools

- *Atom* used as IDE for coding;
- *expo.io* used to build, deploy and debug the mobile application;
- *Simulator* used to simulate and test the application on an iPhone 11 and an iPad Pro;
- *GitHub* used to manage the different versions of the code and this document;
- *Swagger* used to design the RestfulAPI of the server;
- *Heroku* used to deploy the online server;
- *photopea.com* use to draw the application logo and the splash image;
- *Balsamiq Mockups 3* used to draw mock-ups;
- L^AT_EX used to build this document;
- *draw.io* used to draw diagrams.

A.2 Changelog

- **1.0** : First release of this document.
- **1.1** : Some minor fix.