Software Engineering for the

Internet of Things

Project:

**Climate MQ**

Student:

**Giuseppe Gasbarro**, matricola: 271395

Table of Contents

[Introduction 3](#_Toc173174221)

[Requirements 3](#_Toc173174222)

[Functional Requirements 3](#_Toc173174223)

[Non-functional Requirements 3](#_Toc173174224)

[Use Case Diagram 4](#_Toc173174225)

[Sequence Diagrams 6](#_Toc173174226)

[Register Station 6](#_Toc173174227)

[Accept Station 7](#_Toc173174228)

[Send Climatic Data 8](#_Toc173174229)

[Design Decisions 9](#_Toc173174230)

[How to send data from stations? 9](#_Toc173174231)

[How to implement the platform? 9](#_Toc173174232)

[How to store data? 11](#_Toc173174233)

[Used technologies 11](#_Toc173174234)

[How to add a custom station 12](#_Toc173174235)

# Introduction

The goal is to develop a GIS platform capable of receiving climate data from stations worldwide and presenting this information to end users. These stations will send various climate metrics such as temperature, wind speed, and humidity from their sensors.

The platform, then, will show the locations of these stations on an interactive map, allowing users to navigate through and analyse the transmitted data.

The key thing is that, as a GIS platform, it observes standards and rules commonly used in these solutions.

The following sections will detail the overall system requirements, both functional and non-functional, and outline the chosen technologies for implementation.

# Requirements

## Functional Requirements

In order to act as a GIS platform oriented to climatic data, the system must:

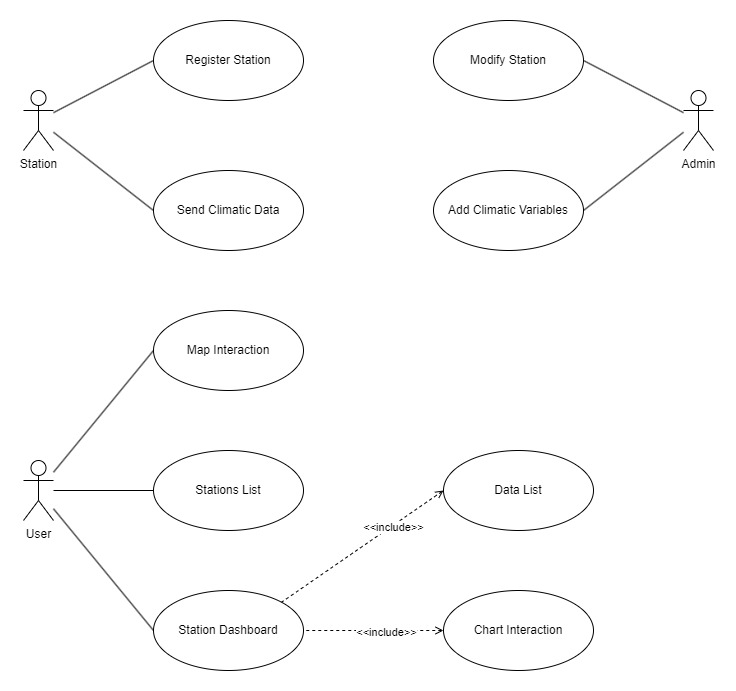
* **Allow new stations to be added:** Facilitate the addition of new climate monitoring stations to the network.
* **Support data acquisition from sensors:** Be prepared to receive and process data from various stations in real-time.
* **Display stations and data on a map:** Provide visual representations of climate stations and their corresponding data on an interactive map.
* **Present climate data effectively:** Offer comprehensive and accessible displays of climate data for analysis and reporting.

## Non-functional Requirements

To ensure the effective monitoring and management of climate data, the system must meet several non-functional requirements:

* **Usability**: The system must feature an intuitive user interface that allows users to easily inspect climate data across all monitoring stations.
* **Scalability**: The system must be designed to accommodate an increasing number of stations and handle an exponential growth in incoming sensor data efficiently.
* **Security**: Only authorized accounts should be able to access the admin panel to add or modify stations and climate variables. Data should only be saved if it is sent from authenticated and approved stations.
* **Availability**: The system must be operational 24/7 to ensure continuous data collection without interruption.

# Use Case Diagram

To achieve functional requirements, a Use Case Diagram was made, defining the functionalities of the platform and the actors involved in its use.

The actor *Station* represents a climatic station and can interact with the system through:

* **Register Station**: a way for the station to register itself in the system.
* **Send Climatic Data**: a method for stations to send their data to the platform.

The actor *Admin* represents the administrator users, who are the only ones able to:

* **Modify Station**: modify the station's details, from the name to the geolocation, and set a state of “accepted” for stations whose data can be saved.
* **Add Climatic Variable**: declare new climatic metrics and their units of measure.

The actor *User* is the general user, without any authentication or specific role, who can access:

* **Map Interaction**: a map displaying all the stations and their geolocated data.
* **Stations List**: a simple list of all the stations accepted by the *Admin*.
* **Station Dashboard**: a detailed view of a single station's climatic data provided in two ways: a filterable list of all the data and an interactive chart showing some statistics (e.g., average value).

# Sequence Diagrams

To better understand how the system works and the data flow, some sequence diagrams were created, selected from the Use Case Diagram functionalities. Only three of them were produced: Register Station, Send Climatic Data, and Accept Station (a sub-functionality of Modify Station).

It was decided to define only three functionalities because they represent the core of the working system and are the most important ones, with a focus on sending data.

## Register Station

In this diagram, there is one actor, the *Station*, that wants to register itself with the system.

To do this, it has to send a message through a *Broker*, acting as a producer, where it has to define all its information (including the name and location).Immagine che contiene testo, diagramma, Piano, Parallelo

Descrizione generata automaticamente

This message will be received by the platform *Consumer* (which previously requested to consume messages from a certain queue of the broker). With the help of *ORM*, the *Consumer* registers the station as a new user and saves all station information in the database. At the end, the *Consumer* sends an acknowledgment message to the *Broker*.

## Accept Station

To accept a station, the *Admin* requests the *Admin Panel* to modify a station.

Through the *Admin Controller*, the selected station is retrieved and shown in the *Admin Panel*. The *Admin* then selects the “accepted” flag and requests to save it.

As before, the *Admin Controller* receives the station to save and asks the *ORM* to do so. Finally, the accepted station is displayed on the *Admin Panel*.Immagine che contiene testo, diagramma, Parallelo, Rettangolo

Descrizione generata automaticamente

## Send Climatic Data

As in the *Register Station* diagram, the *Consumer* is already listening on a *Broker* queue to receive data from the stations. Once the *Station* sends its detected metrics to an exchange topic of the broker, the message arrives at the *Consumer*.

At this point, the *Consumer* needs to verify if the station is 'accepted' and if the credentials are correct. To do this, it asks the *Authenticator* if the user (station) exists and can be authenticated.

If the credentials are correct, the climatic data will be saved; otherwise, nothing will be done. An acknowledgment message will be sent in all cases.Immagine che contiene testo, diagramma, Parallelo, Piano

Descrizione generata automaticamente

# Design Decisions

To meet the system requirements, several key decisions were made, as outlined below.

## How to send data from stations?

The need to send data continuously from station and to receive them in the platform drives the choice to a message-oriented solution.

In this way, RabbitMQ is chosen.

RabbitMQ is an open-source message broker software that facilitates communication between distributed systems. It implements the Advanced Message Queuing Protocol (AMQP) and allows applications to exchange messages in a reliable, efficient, and scalable manner, making it a popular choice for building microservices, integrating systems, and managing background tasks.

Key Features of RabbitMQ

* Reliability: Ensures messages are delivered and processed with high reliability through features like message acknowledgments and delivery guarantees.
* Flexibility: Provides various exchange types (direct, topic, fanout, headers) to route messages flexibly.
* Scalability: Can be scaled horizontally by adding more nodes to the cluster, allowing it to handle increasing loads.
* Fault Tolerance: Offers clustering and high availability features to ensure continuity and resilience in case of failures.
* Management and Monitoring: Provides an easy-to-use management interface and comprehensive monitoring tools to manage and observe the system’s performance.

In the end, RabbitMQ was chosen due to its secure and reliable nature (mainly for ensuring that the data are received and processed by the platform) and its scalability and flexibility to accommodate a growing number of stations.

## How to implement the platform?

For the platform implementation Django web framework is used.

Django is a high-level Python web framework that encourages rapid development and clean, pragmatic design. It takes care of much of the hassle of web development, so you can focus on writing your app without needing to reinvent the wheel. It’s free and open source.

Django was chosen due of its:

* **Security**: Django takes security seriously and helps developers avoid many common security mistakes, such as SQL injection, cross-site scripting, cross-site request forgery and clickjacking. Its user authentication system provides a secure way to manage user accounts and passwords. In our case we used Django authentication to check whether the station can send or not its data on our platform.
* **Scalability**: Django is able to quickly and flexibly scale to meet the heaviest traffic demands.
* **Features**: e.g. Django’s ORM makes easier to communicate with the database, without having to write complex SQL statements.
* **Additional modules and libraries**: in this case the possibility to integrate the platform with GeoDjango.

GeoDjango is a world-class geographic web framework. Its goal is to make it as easy as possible to build GIS web applications and harness the power of spatially enabled data. It is an included contrib module for Django that turns it into a world-class geographic web framework.

Its features include:

* Django model fields for OGC(Open Geospatial Consortium) geometries and raster data.
* Extensions to Django’s ORM for querying and manipulating spatial data.
* Loosely coupled, high-level Python interfaces for GIS geometry and raster operations and data manipulation in different formats.
* Editing geometry fields from the admin.

Since we have geolocated stations, thanks to GeoDjango, we can manage data without losing their geospatial information.

Another feature achieved with GeoDjango is the generated API that return data formatted as GeoJSON.

GeoJSON is a format for encoding a variety of geographic data structures based on JSON. It is widely used for representing geographical features along with their non-spatial attributes and supports several types of geographic data structures.

It is used for:

* Mapping and Visualization: GeoJSON is commonly used in web mapping libraries like Leaflet to visualize geographical data.
* Data Exchange: It provides an easy way to exchange spatial data between different applications and services.
* APIs: Many web services and APIs, such as those provided by GIS platforms and spatial databases, support GeoJSON for spatial data input and output.

Thanks to those features the platform is able to transform stored climatic data in a GeoJSON format and to achieve the functional requirements of showing them on a map.

Moreover, the platform will provide APIs with GeoJSON response.

## How to store data?

Because of the will to implement a GIS platform, we need a database oriented to geospatial data and well supported by GeoDjango (without losing its powerful features).

So that we choose PostGIS, an open-source spatial database extender for the PostgreSQL relational database. It adds support for geographic objects, allowing the database to be used for geographic information systems (GIS) and location-based services. PostGIS follows the Simple Features for SQL specification from the Open Geospatial Consortium (OGC), which defines standard SQL-based methods for storing, retrieving, querying, and processing spatial data.

Thanks to PostGIS we can:

* Add geometry and geography types to PostgreSQL, enabling the storage of our spatial data (complying with OGC standards).
* Perform spatial operations and analyses, such as distance calculations, spatial joins, intersections, and buffering.
* Work seamlessly with PostgreSQL, allowing spatial and non-spatial data to be managed within the same database system.

Moreover, PostGIS utilizes R-Tree over GiST (Generalized Search Tree) indexing to improve the performance of spatial queries.

# Used technologies

At the end, design decisions drove us to the following technologies:

* **Django**: used for the ClimateMQ Frontend and Backend implementation.
* **GeoDjango**: used to extend Django to implement a GIS platform.
* **RabbitMQ**: used as message broker software to send data from stations to ClimateMQ.
* **PostGIS**: used as database to store all data.
* **Docker**: used to containerize all the system components.
* **OpenStreetMap** + **Leaflet**: used for implementing the interactive map.
* **Datatables**: used for listing data and stations in tables.
* **Chart.js**: used to realize the dashboard’s line chart.
* **SpringBoot**: this is not related to the platform, but it was used to simulate stations and sending data to RabbitMQ.

# How to add a custom station

As of now, the SpringBoot application that simulate sensors generate random stations, but what we have to do if we would add a custom station?

Firstly, we have to add a new climatic variable, otherwise no climatic data can be saved.

To add the new variable, you need an Administrator account to access the Admin Panel. At this point, you have to add a new unit of measure through the Units “Add” button, and after that, add a new variable through the Variables “Add” button.

Immagine che contiene testo, schermata, software, Software multimediale

Descrizione generata automaticamente

Now we can start thinking about sending data from stations.

To add a new station, you can register it through RabbitMQ by sending the station’s name, location, username and password. Then, the station needs to be accepted in the Admin Panel to start sending detections. Data are sent by declaring station’s username and password, along with the detected value and its unit of measure (which must match one of the system’s saved Units). If the unit of measure is correct and the station is accepted and authenticated, data are correctly stored.

To simulate stations registration and detection dispatch, a SpringBoot application was made and is already runnable through a Docker container (inside the same docker-compose file as the climateMQ project). As of now, stations are randomly generated, but modifying some code, you can create your own station as follows:

* Open mq-sensors folder and go in “src\main\java”.
* Open JsonMessageGenerator class in “it\univaq\iot\mq\_sensors\amqp\utils” package.
* From line 23 to line 28 you can change the required station information and the value min and max for the random value generation.

Immagine che contiene testo, schermata, Carattere

Descrizione generata automaticamente

* Now go to the application.yml file in the mq-sensors folder and replace “random” with “fixed” in the active profiles.
* Run “docker-compose build spring-app”.
* Run “docker-compose up spring-app” to start the SpringBoot Application.
* Then, the application will send station registration and start to send data every 5 seconds (as defined in the @Scheduler annotation in the FixedScheduler class; the maximum time of execution is set in the application.yml).
* When you see from the terminal log that the registration was sent, go to the Admin Panel and modify the newly added station to set the “accepted” flag to true.

Be careful that the ClimateMQ consumer is active (if not run “docker-compose exec app python manage.py start\_consumer” on a terminal before starting the spring-app).