University of Pisa

School of Engineering

Department of Information Engineering

MSc Artificial Intelligence and Data Engineering

MSc Computer Engineering

Large Scale and Multi-Structured Databases

Weather Management Application

Authors:

Lorenzo Vezzani

Rojan Shrestha

Alessio Simoncini

Academic Year 2024/25

Summary

Chapter 1

Introduction

1.1 Application Goal

This application is designed to offer a comprehensive meteorological service that integrates both real-time **weather forecasting** and access to **historical weather** data for the most relevant and populous Italian cities.
The first functionality of the application is a standard, yet essential, weather forecasting service. This is crucial for all types of users the application may have, as it supports daily activity planning. For this component, we focused particularly on responsiveness and service availability. Alongside the forecasting service, we developed a module for the analysis of historical weather measurements, enabling accurate and detailed examinations of trends, specific events, and anomalies.
Building upon historical weather data, we included a dedicated section for particularly significant meteorological phenomena, referred to as **Extreme Weather Events** (sometimes abbreviated as EWE). These events can be crucial for identifying patterns and trends in climate behavior, providing a high-level overview without the need for in-depth analysis of individual measurements.

1.2 Big Data Analysis

Meteorological data is characterized by both high **volume**—due to the numerous variables involved—and high **velocity**, stemming from the frequency at which the data must be updated. These characteristics make meteorological data an ideal candidate for the application of various Big Data Analysis paradigms, which we employed in both the forecasting and historical measurement components of the system.

1.3 Code repository

This documentation describes the main ideas and strategies used to implement the application, along with an overview of the completed final product and some snapshot of the code. The entire code can be found in the repository:
`https://github.com/AlessioSmn/WheaterAppServer/tree/vmCluster`

Chapter 2

Application Design Overview

2.1 Structure of the application

The application has been developed using **Java** programming language, following the **client-server** paradigm. The server side has been implemented with the **Spring** framework. It provides to the client a **set of APIs**, described in the following chapters, to access weather forecasts and historical data. The client side, even if not developed (as per specifications), is supposed to be composed of a graphical interface with a section dedicated to sign-up/login and another to display the data received from the server via the API.

2.2 Use-Case Diagram

In the use case diagram, three different actors are identified: **Guest**, **Registered User**, and **Admin**.

- **Guest** can exclusively view weather forecasts and historical data.
- **Registered User** can add any city to their list of *followed cities*, a feature particularly useful for enabling a personalized graphical interface and a client-side notification system.
- **Admin** has extended privileges and can trigger the replacement of outdated data with more recent information, as well as update thresholds, which are described in the next sub-section. Moreover, the admin can add cities in the database. This is a key feature, in fact, initially the application only provides the information of the Italian province capitals. This can be not enough to fully describe the weather situation in Italy, since some parts are really far from the province capitals. Some examples can be Elba Island or Abetone, which are both relatively far from their province capital (respectively, Livorno and Pistoia).

2.3 UML

The UML class diagram models the structure of a weather monitoring and forecasting system with three types of users: **Guest**, **Registered**, and **Admin**, all of which inherit from a common `User` class. `Guest` users have no credentials and are limited to viewing forecasts and historical data. `Registered` users, identified by a username, email, and password, can follow multiple cities, enabling features such as notifications and personalized interfaces. `Admins` have similar credentials but are responsible for managing the system, including updating data and threshold parameters. The core entity is `City`, characterized by attributes like name, region, geographic coordinates, elevation, and the number of followers. Each city is associated with:

- Multiple `Predictions`, which contain forecast data such as temperature, rainfall, snowfall, wind speed, and the number of hours in advance.
- Multiple `HourlyMeasurement`, which store historical weather observations with timestamps. Every observation consists in: temperature, rainfall, wind-speed and snowfall.

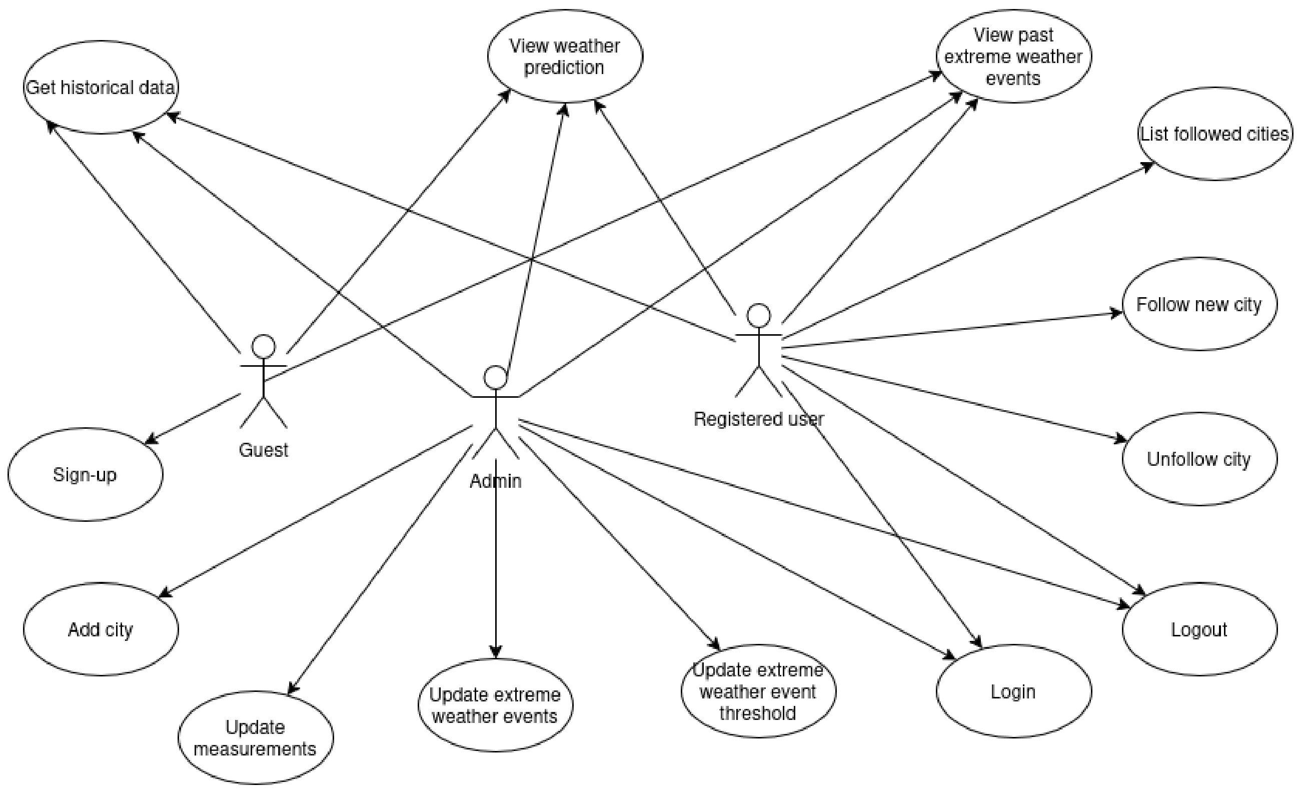


Figure 1: Main actors

- A single set of `ExtremeWeatherThresholds`, specifying maximum and minimum limits for temperature, rainfall, snowfall, and wind speed[1].
- Multiple `ExtremeEvents`, described by category (Rainfall, Snowstorm, ecc.), strength, and a date range.

2.4 Functional Requirements

A list of the main functional requirements of the application:

- Users can register, log-in and log-out.
- The application allows users to view forecasts weather conditions for the next few days for any city (both province capital and not).
- Authenticated users have access to advanced features, such as: saving *favorite cities* for quick access to related forecasts.
- The application stores historical measurements and records of extreme weather events: users can explore these historical data through built-in analytical tools.
- Both historical weather data and forecast weather data is obtained from external APIs and is updated regularly.
- Admin users can update parameters and add elements to the database in order to obtain a more precise service;

[1]These thresholds are used to define whether or not a measurement is an extreme event. For example, a heavy snowfall in Aosta is not considered extreme as if the same amount of snow fell in Syracuse.

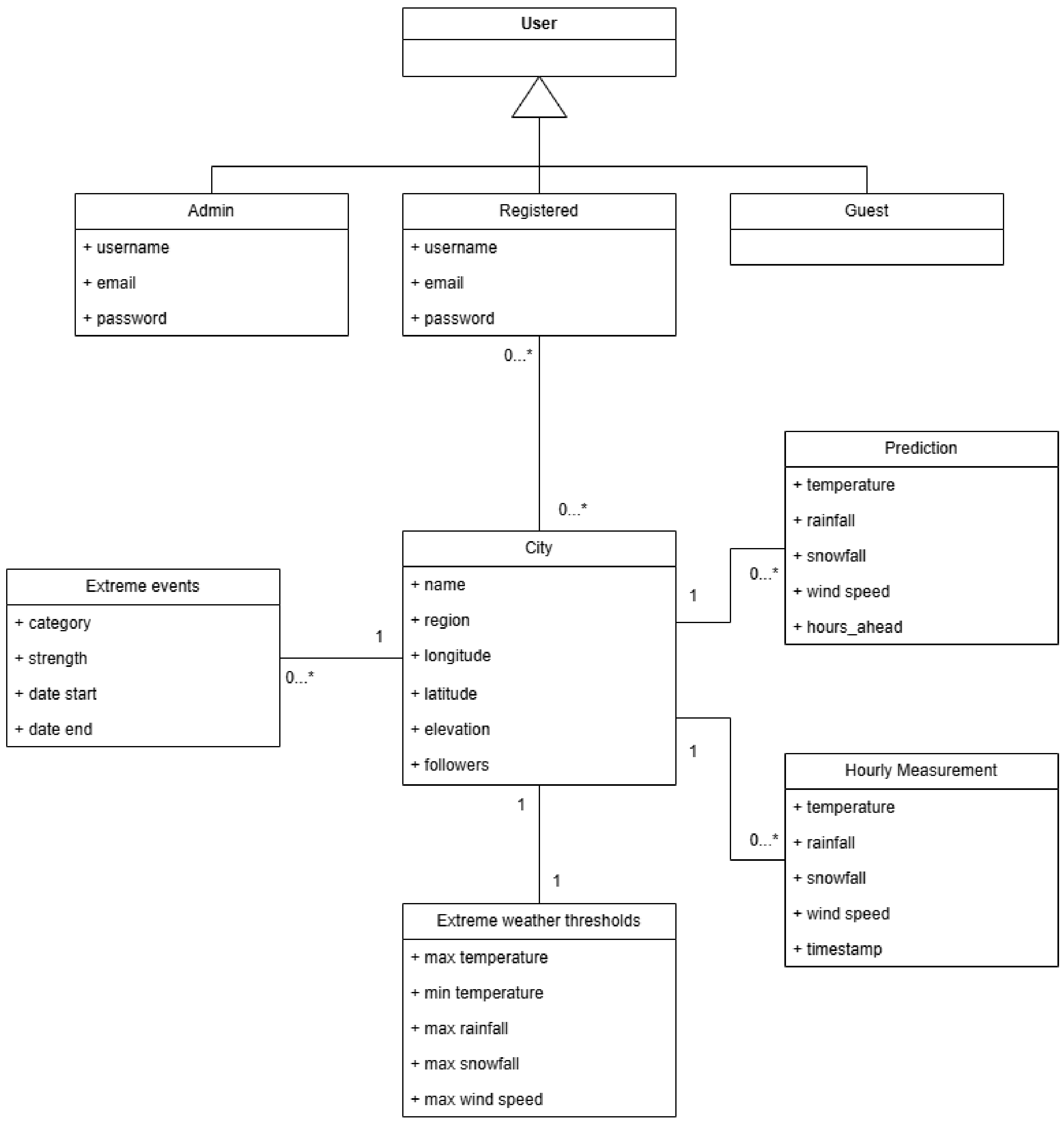

User
Admin
+ username
+ email
+ password
Registered
+ username
+ email
+ password
Guest
0...*
0...*
Prediction
+ temperature
+ rainfall
+ snowfall
+ wind speed
+ hours_ahead
City
+ name
+ region
+ longitude
+ latitude
+ elevation
+ followers
1
0...*
Extreme events
+ category
+ strength
+ date start
+ date end
1
0...*
1
Hourly Measurement
+ temperature
+ rainfall
+ snowfall
+ wind speed
+ timestamp
1
0...*
1
1
Extreme weather thresholds
+ max temperature
+ min temperature
+ max rainfall
+ max snowfall
+ max wind speed

Figure 2: UML

2.5 Non-Functional Requirements

The primary non-functional requirement of the application is **performance**, with a specific focus on minimizing read time as much as possible. This requirement serves as the foundation for many of the implementation and data modeling decisions that will follow. In the NoSQL database context, performance translates not only into fast response times, but also into having a highly available service, capable of serving user requests with minimal latency even under heavy load. Ensuring fast access to data is critical, especially in scenarios where users expect instantaneous feedback or where large volumes of read operations are performed frequently.
Additionally, an authentication mechanism is implemented to ensure secure access to the database. Upon login, the user receives a token, which must be included in the HTTP requests when accessing certain APIs. This token also serves to distinguish between Admin and Standard users.

2.6 API used to retrieving data

All the data stored in the database are retrieving from API-service called **Open Meteo**. The service provides both historical measurements from 1940 and forecast, filterable using data, timestamp, latitude and longitude. Moreover, data can be retrieving both using CSV and JSON. Our application uses JSON, since also MongoDB uses this format. Here's the APIs uri, respectvely historical and forecast:
`https://archive-api.open-meteo.com/v1/archive`
`https://api.open-meteo.com/v1/forecast`

2.7 Databases architectures used

The NoSQL architecture used includes both a **DocumentDB** and a **Key-ValueDB**. The key-value store was chosen because the application requires fast reads and does not involve complex queries. Moreover, since forecast data changes frequently and must be updated or retrieved quickly, it is crucial to rely on a DBMS that supports high-performance access. On the other hand, the DocumentDB is well-suited for analytics purposes. For this reason, real-time forecast data is stored in the Key-Value DB, while historical records and extreme weather events are stored in the DocumentDB. Moreover, the DocumentDB stored the user information, since the this type of information is not too volatile. To identify a session is defined a token: this is stored within key-valueDB.

2.8 Some Mockups

These are very simple mockups that represent a possible GUI for our application. A registered user called *John Baker* is seeing Florence forecast, selecting a new city from the followed ones and searching for historical data.

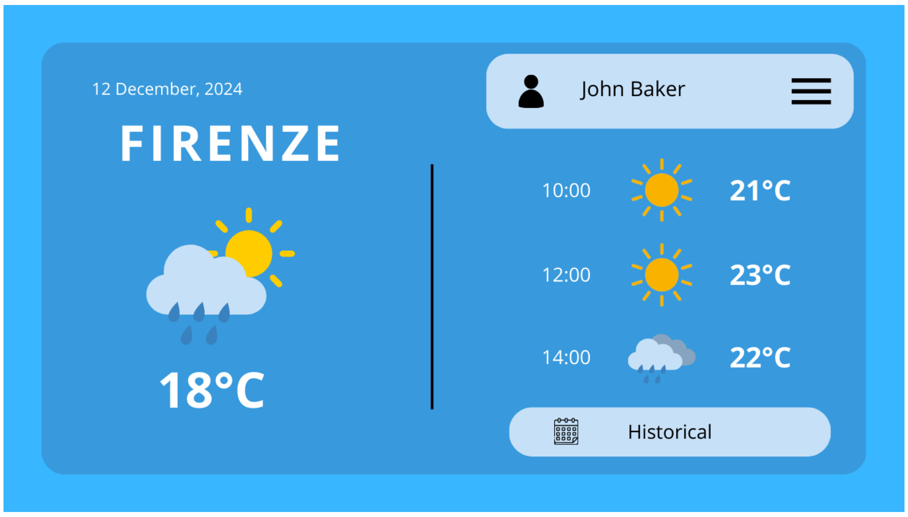


Figure 3: Mockup - page 1

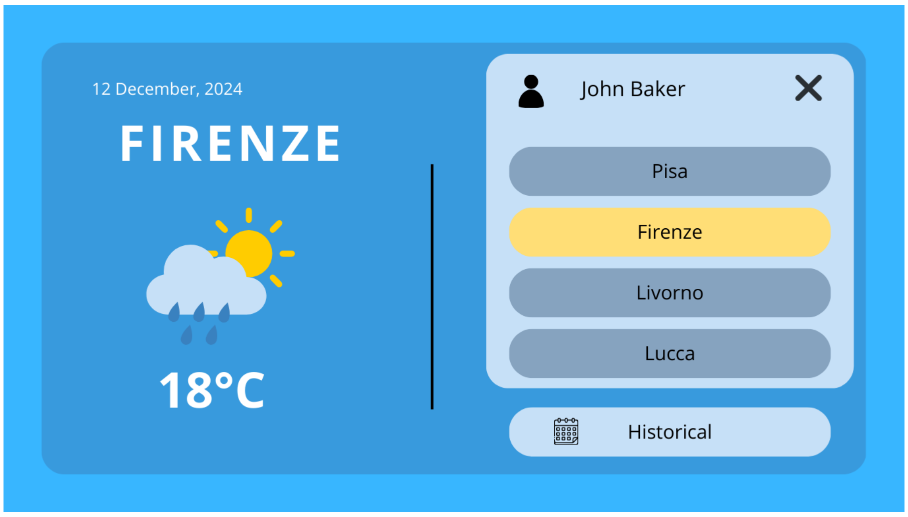
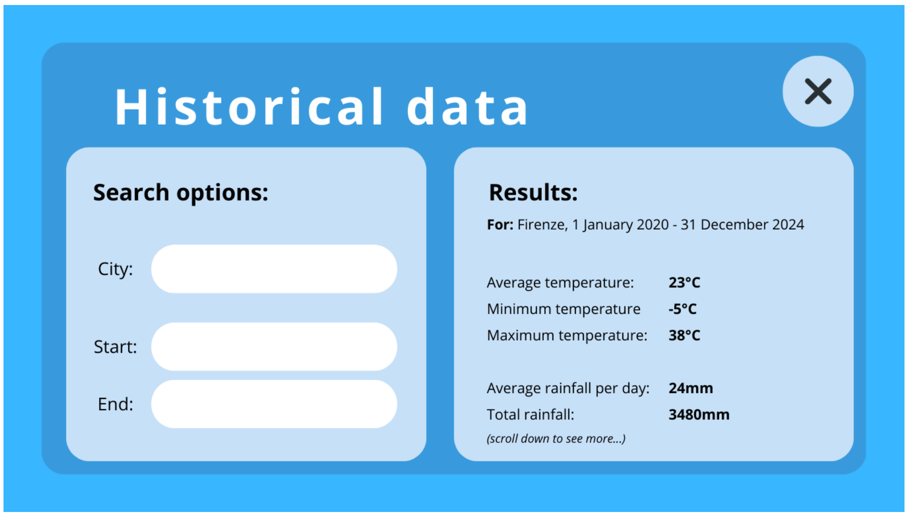


Figure 4: Mockup - page 2



Results:

For: Firenze, 1 January 2020 - 31 December 2024

Average temperature:	**23°C**
Minimum temperature	**-5°C**
Maximum temperature:	**38°C**
Average rainfall per day:	**24mm**
Total rainfall:	**3480mm**

(scroll down to see more...)

Figure 5: Mockup - page 3

Chapter 3

Data Modeling and Indexes

3.1 MongoDB entities

The application uses MongoDB to store cities information, user information, historical weather data and extreme weather event entities. The data is organized across multiple collections, each with a specific purpose and structure.

3.1.1 Users Collection

User information is stored in a collection named `users`. Each document in this collection contains the following fields:

- `username` — defined as **unique**
- `password` — saved **encrypted**
- `email` — defined as **unique**
- `listCityId` — a list representing the cities followed by the user

The data modeling strategy adopted is *data linking*, as only the identifiers of the followed cities are needed—there is no requirement to include additional city information within the user documents.

3.1.2 Cities Collection

The `cities` collection holds information about each city. Every document includes:

- `name`
- `region`
- `coordinates` — both latitude and longitude
- `threshold` — a city-specific array weather threshold
- `followers` — number of users that follow city: it can be useful from a client-side point of view
- `city_id` — a unique identifier in the format: `region-code_city-code_latitude_longitude`
- `eweList` — a list of the extreme weather event (**ewe**) that have affected the city. Every ewe has this structure
  - `category`: Indicates the type of event. The possible categories are defined in the **ExtremeWeatherEventCategory** enum.
  - `strength`: Represents the severity or intensity of the event.
  - `dateStart`: The timestamp indicating when the event was first detected.
  - `dateEnd`: The timestamp indicating when the event concluded.

In this case, *data embedding* was prefer in respect to than data linking. This is a (1:N) entity betweeen city entity and ewe entity and access for the extreme weather event is performed mainly using the `city_id`. Since ewes are rare (of course, they are extreme), the size of the document would not be excessively large even if a dataset from the 80s is used.

3.1.3 Measurements Collection

Historical weather data is stored in a separate collection, where each document represents an hourly measurement. The fields include:

- `temperature` [C°]
- `rainfall` [mm/h]
- `snowfall` [mm/h]
- `windspeed` [km/h]
- `timestamp` — in the form: *YYYY-MM-DDTHH:MM:SS:sss.Z*
- `city_id` — linking the measurement to a specific city

The chosen data modeling strategy is *document linking*, rather than embedding. Embedding hourly measurements within each city document would have led to excessively large documents, approaching the MongoDB document size limit of 16MB. By separating measurements into their own collection and linking them via the `city_id`, the application achieves better scalability and maintainability.

3.2 Mongo Indexes

Within the DocumentDB, some indexes were defined in order to make reads faster. Since our application is more focused on reads, the side effects that the performance indicators of the writings have not been taken into account.
All the indexes has been defined in the Java class `MongoInitiliazer`.

3.2.1 User Indexes

Considering that the most frequent operation is certainly the login, an index has been defined on the **username** field.

3.2.2 City Indexes

Access to the city is largely done using the city_id, so the **default index** defined by MongoDB is sufficient.[2] This is fundamental since extreme weather events are performed using city-id as parameter.

3.2.3 Measurement Indexes

The analytics defined within the application analyze the measurements by city. Therefore, the index has been defined using `cityId` and `time`. This is certainly an expensive index (from a storage point of view), but it allows to have fast read of the `hourly_measurements`, a fundamental feature for the analytics.

[2]MongoDB provides a default index for the _id field.

3.3 Redis entities

Redis is used to store cities and their corresponding forecasts. City keys follow the pattern: `city:{regioncode}-citycode-latitude-longitude`,
e.g., `city:{tus}-pis-43.7085-10.4036`.
These keys are essential for supporting the arbitrary city forecast feature (see next section), which requires retrieving cities by region and computing distances between them. The values retrievable using these keys is useful the information (*region* and *elevation*) used in the arbitrary city forecast functionality. Thus, the value-type used is **hash**.
Forecasts are stored using the format:
`forecast:{regioncode}-citycode-latitude-longitude:date`,
e.g., `forecast:{tus}-pis-43.7085-10.4036:2025-06-08`.
This key structure allows retrieving the entire daily forecast, consistent with the design of the external APIs, which operate on daily or weekly granularity. The value associated with each forecast key includes temperature, rainfall, snowfall, windspeed, and timestamp, matching the schema used for historical measurements. This is simply stored as a **string**.

3.4 Intra-DB consistency

The only common entity between the two databases is `city`. As a result, the city insertion operation is entirely **rolled back** if one or both write operations fail. Additional details are illustrated in the implementation.

Chapter 4

Code Implementation

The main components of the server application, located in the `src/main/java/it.unipi.lsmsd/` folder, are the following:

- `controller`: handles HTTP requests and routes them to services
- `service`: contains business logic
- `DTO`: defines data transfer objects for external communication
- `model`: defines domain entities and data structures
- `repository`: manages data access and persistence
- `utility`: provides helper functions and common utilities

4.1 Controllers

The `controller` classes define the API endpoints of the application. They do not implement logic directly; they delegate all processing to the appropriate service classes. Controllers are responsible for handling HTTP requests, invoking the corresponding service methods, and managing errors by mapping them to appropriate HTTP status codes and response formats.

- `UserController`: Handles all user-related operations, such as registration, login, and logout.
- `CityController`: Handles all admin-reserved operations related to cities.
- `FavoriteCitiesController`: Handles all operations reserved for registered users related to their list of followed cities.
- `EweController` - Extreme Weather Event Controller: Handles all admin-reserved operations related to extreme weather events.
- `HourlyMeasurementController`: Handles all operations related to the retrieval of forecasts.
- `AnalyticsController`: Handles all API calls related to analytics on both historical data and extreme weather events.
- `DataManagerController`: Handles all the admin-reserved API calls to update measurements and forecasts in the database. It is also to initialize the databases: this part is not intended for use during the normal operation of the application.
- `GlobalExceptionHandler`: Handles exceptions caused by invalid `enum` values passed as parameters. It is used by all APIs that take either an `ExtremeWeatherEventCategory` or a `MeasurementField` as input.

4.2 Service

The `service` package contains the classes in which the core logic of the application is implemented. Each controller delegates the execution of operations to a corresponding service class, which encapsulates the main logic and often interacts with the database through repository functions.

UserService Defines the functions `login`, `logout`, `register`, `getUserFromToken`, and `checkRole`. The last two functions are central to the implementation of authorization control for specific functionalities.

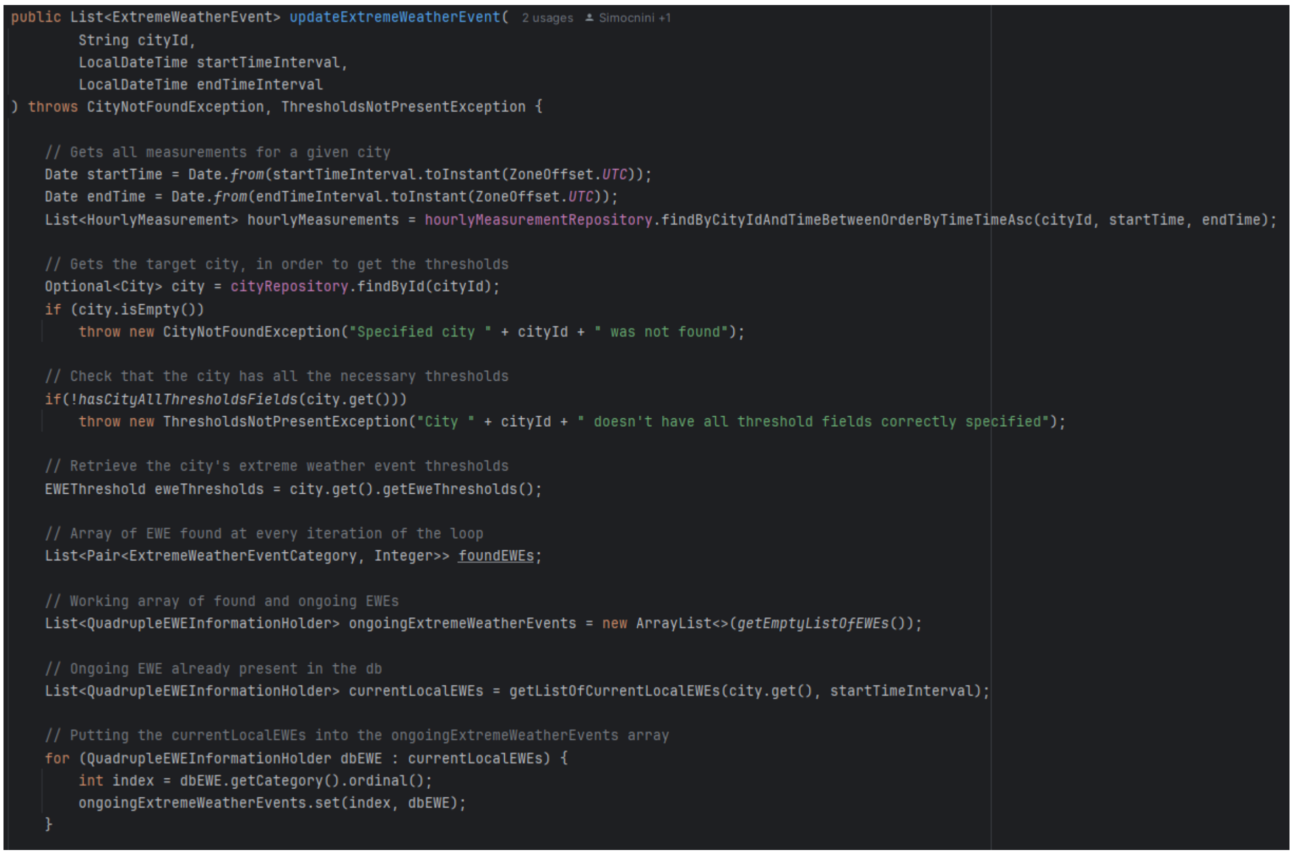
CityService Defines the functions `saveCity` and `updateCityThresholds`, which implement the logic for adding a new city to the database and updating the extreme weather event threshold values for a given city, respectively. Other functions include `setLastEweUpdateById`, `getLastEweUpdateById`, and `setLastMeasurementUpdateById`, which serve as utility methods to support the refreshing of historical measurements and extreme weather events. All these functions interact with the database through repository methods. The function `saveCity` is fundamental in the application enviroment. It adds a city in Mongo and Redis (if only one write fails, everything is rolled back), in order to obtain a better prediction and historical service.

FavoriteService Defines the functions `getFavorites`, `addToFavorites` and `removeFromFavorites`, which manage the list of followed cities for each user. Both functions rely on `getUserFromToken` from `UserService` to identify the requesting user based on the authorization token. The last two functions are annotated with @Transactional to ensure atomicity of two operations: updating listCityId and followers.

ExtremeWeatherEventService Defines the main functions `updateExtremeWeatherEvent` and `cleanExtremeWeatherEventDuplicates`. Additional functions include `updateExtremeWeatherEventAll`, `updateExtremeWeatherEventAutomatic`, `cleanExtremeWeatherEventDuplicatesRange`, and `cleanExtremeWeatherEventDuplicatesAll`, which internally invoke the main methods with different time range scopes and are the ones directly called by the controllers.

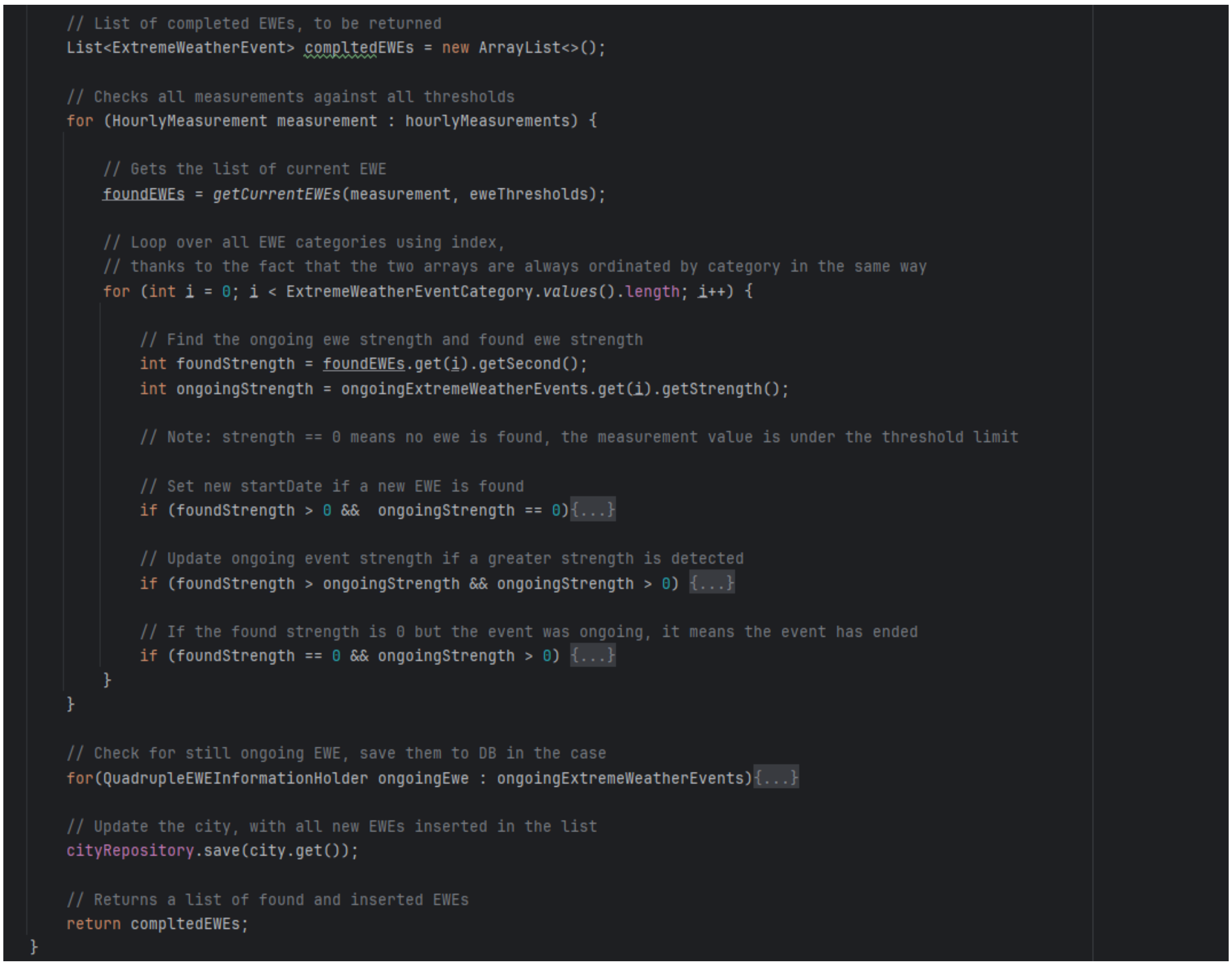
- `updateExtremeWeatherEvent` (Figures 6 and 7) — This function takes as input a target city and a time range. It retrieves all measurements within the specified range and the corresponding extreme weather event thresholds for the given city. It then compares the measurements against the thresholds: when an out-of-threshold value is detected, a new extreme weather event entity is created. The entity may be further updated in terms of strength and duration if subsequent out-of-threshold measurements are found.

- `cleanExtremeWeatherEventDuplicates` — Since the `updateExtremeWeatherEvent` function does not check for pre-existing overlapping extreme weather events, this method is designed to clean up potential duplicates by removing entities that overlap in time.



```java
public List<ExtremeWeatherEvent> updateExtremeWeatherEvent(
        String cityId,
        LocalDateTime startTimeInterval,
        LocalDateTime endTimeInterval
) throws CityNotFoundException, ThresholdsNotPresentException {

    // Gets all measurements for a given city
    Date startTime = Date.from(startTimeInterval.toInstant(ZoneOffset.UTC));
    Date endTime = Date.from(endTimeInterval.toInstant(ZoneOffset.UTC));
    List<HourlyMeasurement> hourlyMeasurements = hourlyMeasurementRepository.findByCityIdAndTimeBetweenOrderByTimeTimeAsc(cityId, startTime, endTime);

    // Gets the target city, in order to get the thresholds
    Optional<City> city = cityRepository.findById(cityId);
    if (city.isEmpty())
        throw new CityNotFoundException("Specified city " + cityId + " was not found");

    // Check that the city has all the necessary thresholds
    if(!hasCityAllThresholdsFields(city.get()))
        throw new ThresholdsNotPresentException("City " + cityId + " doesn't have all threshold fields correctly specified");

    // Retrieve the city's extreme weather event thresholds
    EWEThreshold eweThresholds = city.get().getEweThresholds();

    // Array of EWE found at every iteration of the loop
    List<Pair<ExtremeWeatherEventCategory, Integer>> foundEWEs;

    // Working array of found and ongoing EWEs
    List<QuadrupleEWEInformationHolder> ongoingExtremeWeatherEvents = new ArrayList<>(getEmptyListOfEWEs());

    // Ongoing EWE already present in the db
    List<QuadrupleEWEInformationHolder> currentLocalEWEs = getListOfCurrentLocalEWEs(city.get(), startTimeInterval);

    // Putting the currentLocalEWEs into the ongoingExtremeWeatherEvents array
    for (QuadrupleEWEInformationHolder dbEWE : currentLocalEWEs) {
        int index = dbEWE.getCategory().ordinal();
        ongoingExtremeWeatherEvents.set(index, dbEWE);
    }
```

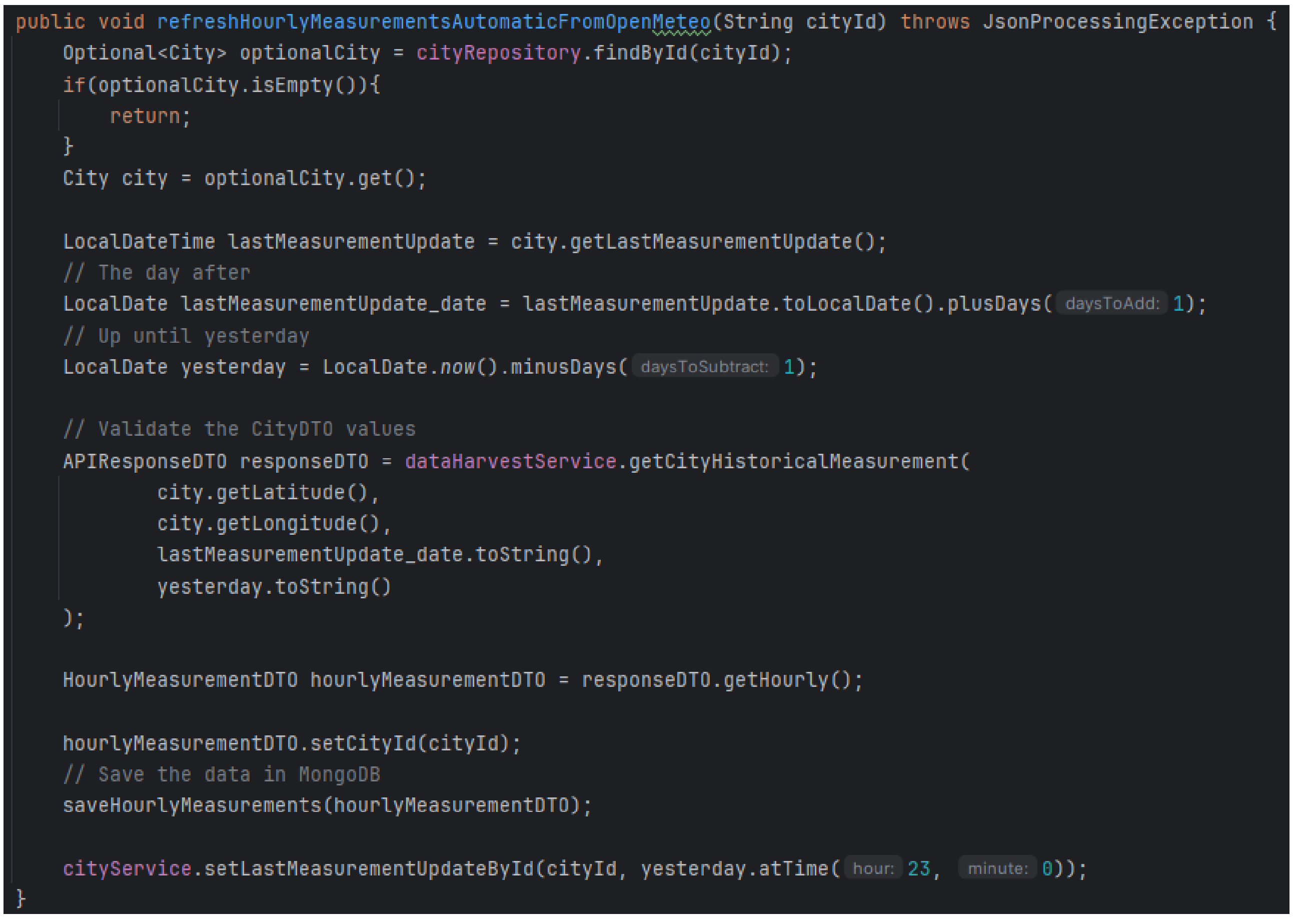
Figure 6: updateExtremeWeatherEvent function - part 1



```
    // List of completed EWEs, to be returned
    List<ExtremeWeatherEvent> compltedEWEs = new ArrayList<>();

    // Checks all measurements against all thresholds
    for (HourlyMeasurement measurement : hourlyMeasurements) {

        // Gets the list of current EWE
        foundEWEs = getCurrentEWEs(measurement, eweThresholds);

        // Loop over all EWE categories using index,
        // thanks to the fact that the two arrays are always ordinated by category in the same way
        for (int i = 0; i < ExtremeWeatherEventCategory.values().length; i++) {

            // Find the ongoing ewe strength and found ewe strength
            int foundStrength = foundEWEs.get(i).getSecond();
            int ongoingStrength = ongoingExtremeWeatherEvents.get(i).getStrength();

            // Note: strength == 0 means no ewe is found, the measurement value is under the threshold limit

            // Set new startDate if a new EWE is found
            if (foundStrength > 0 &&  ongoingStrength == 0){...}

            // Update ongoing event strength if a greater strength is detected
            if (foundStrength > ongoingStrength && ongoingStrength > 0) {...}

            // If the found strength is 0 but the event was ongoing, it means the event has ended
            if (foundStrength == 0 && ongoingStrength > 0) {...}
        }
    }

    // Check for still ongoing EWE, save them to DB in the case
    for(QuadrupleEWEInformationHolder ongoingEwe : ongoingExtremeWeatherEvents){...}

    // Update the city, with all new EWEs inserted in the list
    cityRepository.save(city.get());

    // Returns a list of found and inserted EWEs
    return compltedEWEs;
}
```

Figure 7: updateExtremeWeatherEvent function - part 2

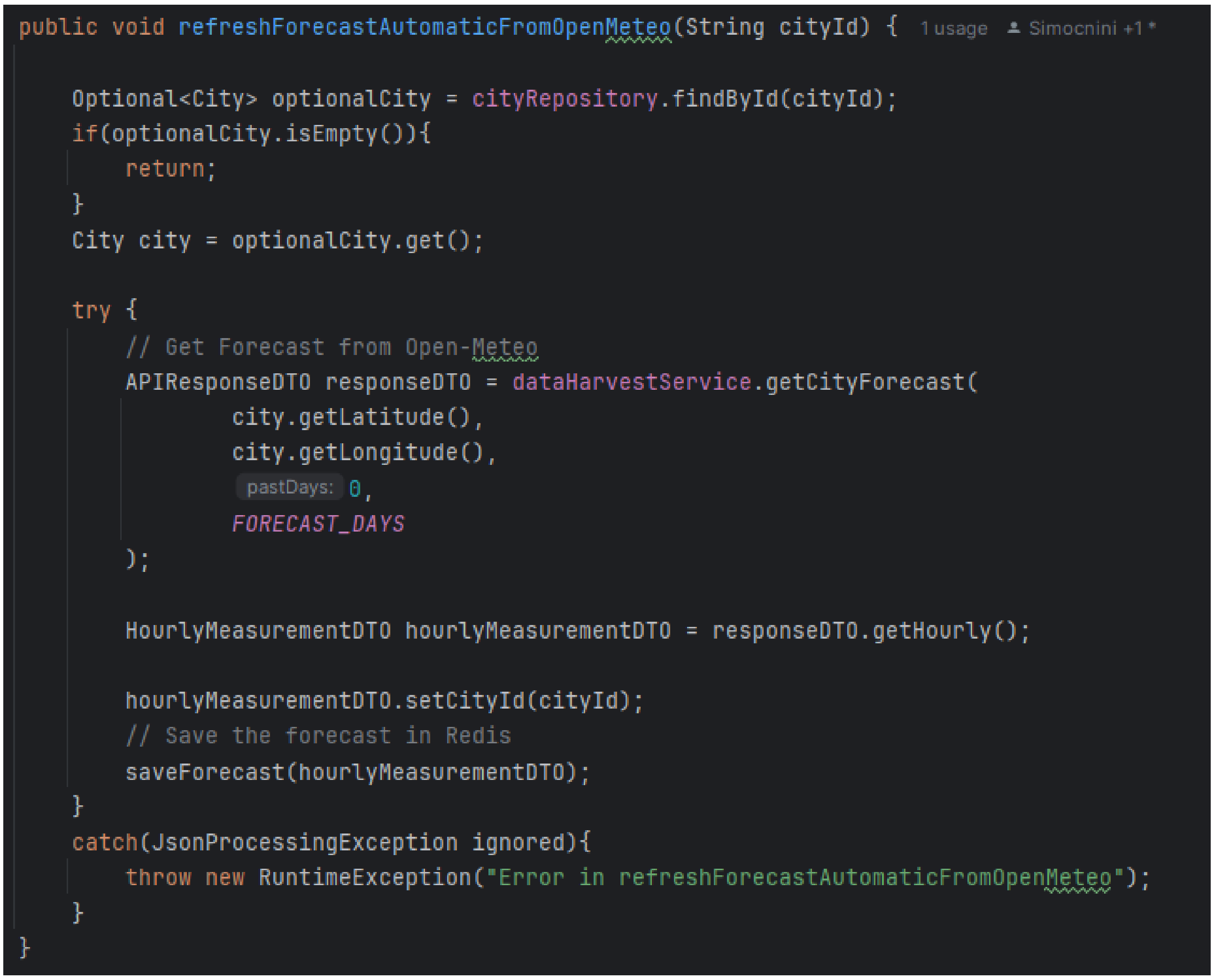
HourlyMeasurementService Defines the functions `saveHourlyMeasurements`, used during initialization to insert all historical measurements from a JSON file into MongoDB, and `refreshHourlyMeasurementsAutomaticFromOpenMeteo` (Figure 8), a function designed to be called periodically to retrieve the latest measurements from OpenMeteo and store them in the database.



```java
public void refreshHourlyMeasurementsAutomaticFromOpenMeteo(String cityId) throws JsonProcessingException {
    Optional<City> optionalCity = cityRepository.findById(cityId);
    if(optionalCity.isEmpty()){
        return;
    }
    City city = optionalCity.get();

    LocalDateTime lastMeasurementUpdate = city.getLastMeasurementUpdate();
    // The day after
    LocalDate lastMeasurementUpdate_date = lastMeasurementUpdate.toLocalDate().plusDays( daysToAdd: 1);
    // Up until yesterday
    LocalDate yesterday = LocalDate.now().minusDays( daysToSubtract: 1);

    // Validate the CityDTO values
    APIResponseDTO responseDTO = dataHarvestService.getCityHistoricalMeasurement(
            city.getLatitude(),
            city.getLongitude(),
            lastMeasurementUpdate_date.toString(),
            yesterday.toString()
    );

    HourlyMeasurementDTO hourlyMeasurementDTO = responseDTO.getHourly();

    hourlyMeasurementDTO.setCityId(cityId);
    // Save the data in MongoDB
    saveHourlyMeasurements(hourlyMeasurementDTO);

    cityService.setLastMeasurementUpdateById(cityId, yesterday.atTime( hour: 23,  minute: 0));
}
```

Figure 8: refreshMeasurementsFromOpenMeteo function

RedisForecastService This class defines all functions related to the management of weather forecasts. The function `refreshForecastAutomaticFromOpenMeteo` (Figure 9) is responsible for retrieving updated forecasts from OpenMeteo and saving them into Redis via the `saveForecast` function. The method `deleteAllForecast` is used to remove existing forecasts before inserting new data into Redis.

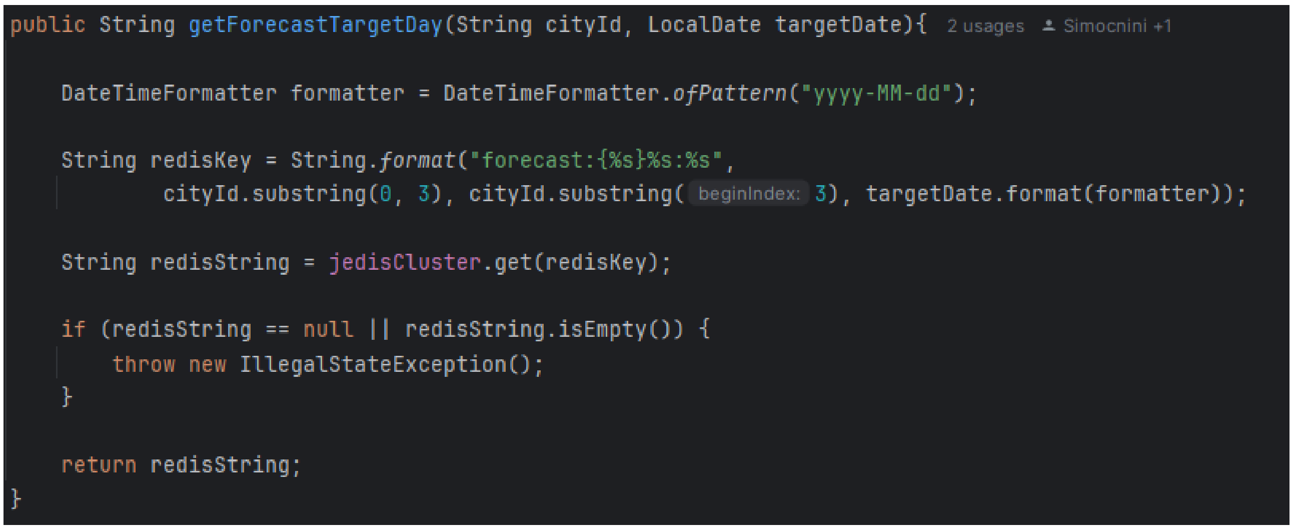


```java
public void refreshForecastAutomaticFromOpenMeteo(String cityId) {  1 usage  Simocnini +1 *

    Optional<City> optionalCity = cityRepository.findById(cityId);
    if(optionalCity.isEmpty()){
        return;
    }
    City city = optionalCity.get();

    try {
        // Get Forecast from Open-Meteo
        APIResponseDTO responseDTO = dataHarvestService.getCityForecast(
                city.getLatitude(),
                city.getLongitude(),
                pastDays: 0,
                FORECAST_DAYS
        );

        HourlyMeasurementDTO hourlyMeasurementDTO = responseDTO.getHourly();

        hourlyMeasurementDTO.setCityId(cityId);
        // Save the forecast in Redis
        saveForecast(hourlyMeasurementDTO);
    }
    catch(JsonProcessingException ignored){
        throw new RuntimeException("Error in refreshForecastAutomaticFromOpenMeteo");
    }
}
```

Figure 9: refreshForecastFromOpenMeteo function

The functions `getForecastTargetDay` (Figure 10), `get7DayForecast`, and `getForecastArbitraryCityTargetDay` implements the logic required to handle requests from controllers and to retrieve or estimate weather forecasts for an arbitrary location specified by the client. The method takes the following input parameters: *region*, *latitude*, *longitude*, and *elevation*. To estimate the forecast at the target location, the system first retrieves the `city_ids` of all cities within the specified region, as stored in the database. Then, it computes a **weighted average** of the weather values (e.g., temperature, rainfall) retrieved from the `Forecast` entity. The weights used in the average are based on the inverse of a parameter called *effective-distance* which is calculate with the difference of the distance between the target location and each city and the *elevation difference*[3]. The distance is calculated using the **Haversine** algorithm, which relies on latitude and longitude values. Elevation data for the cities is retrieved from **Redis** by accessing the `City` entity using the corresponding `city_ids`.

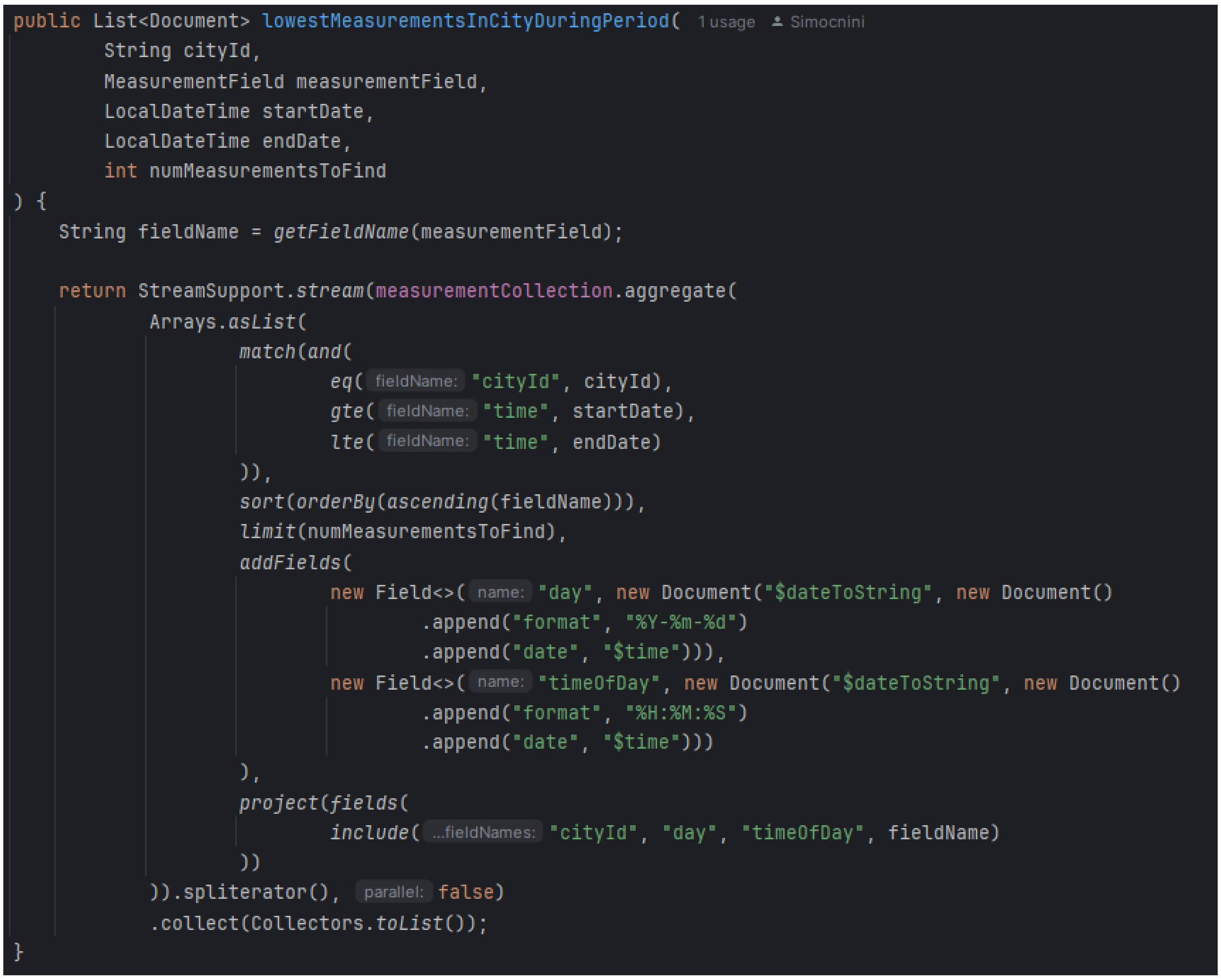
[3]Both the distance and the elevation difference are scaled by two empirical factors, called *alpha* and *beta*, which are determined by comparing the output of the method with actual forecasts obtained from some websites.



```
public String getForecastTargetDay(String cityId, LocalDate targetDate){  2 usages  Simocnini +1

    DateTimeFormatter formatter = DateTimeFormatter.ofPattern("yyyy-MM-dd");

    String redisKey = String.format("forecast:{%s}%s:%s",
            cityId.substring(0, 3), cityId.substring( beginIndex: 3), targetDate.format(formatter));

    String redisString = jedisCluster.get(redisKey);

    if (redisString == null || redisString.isEmpty()) {
        throw new IllegalStateException();
    }

    return redisString;
}
```

Figure 10: getForecastTargetDay function

AnalyticsService Defines all the functions related to analytics on historical measurements. A large number of methods are implemented — one for each corresponding controller endpoint— but they all share a common structure: they receive query parameters as input and perform the computation using a MongoDB aggregation pipeline.
We take this function (Figure 11) as example: it retrieves the lowest measurement values for a specified city and measurement type within a given time range. It returns the `numMeasurementsToFind` lowest entries by applying a MongoDB aggregation pipeline that filters by city and date, sorts by the selected field, and adds to the result the formatted date and time fields.



```java
public List<Document> lowestMeasurementsInCityDuringPeriod(  1 usage  Simocnini
        String cityId,
        MeasurementField measurementField,
        LocalDateTime startDate,
        LocalDateTime endDate,
        int numMeasurementsToFind
) {
    String fieldName = getFieldName(measurementField);

    return StreamSupport.stream(measurementCollection.aggregate(
            Arrays.asList(
                    match(and(
                            eq( fieldName: "cityId", cityId),
                            gte( fieldName: "time", startDate),
                            lte( fieldName: "time", endDate)
                    )),
                    sort(orderBy(ascending(fieldName))),
                    limit(numMeasurementsToFind),
                    addFields(
                            new Field<>( name: "day", new Document("$dateToString", new Document()
                                    .append("format", "%Y-%m-%d")
                                    .append("date", "$time"))),
                            new Field<>( name: "timeOfDay", new Document("$dateToString", new Document()
                                    .append("format", "%H:%M:%S")
                                    .append("date", "$time")))
                    ),
                    project(fields(
                            include( ...fieldNames: "cityId", "day", "timeOfDay", fieldName)
                    ))
            )).spliterator(),  parallel: false)
            .collect(Collectors.toList());
}
```

Figure 11: lowestMeasurementsInCityDuringPeriod function

AutomatingService Defines the functions `updateMeasurements`, `updateExtremeWeatherEvents`, and `updateForecasts`. The purpose of this service is to automate the daily tasks that would otherwise need to be performed manually for each city by an administrator. The main update routine is designed to be executed daily when the system load is estimated to be at its lowest. These functions are invoked by the `/update` endpoints of the `DataManagerController`.

- `updateMeasurements` — Retrieves from OpenMeteo the measurements of the previous day for all cities stored in the database and inserts them into the measurements collection. It calls the function shown in Figure 8 for all the cities in the database.
- `updateExtremeWeatherEvents` — Calculates any new extreme weather events or updates existing ones based on the newly acquired measurements.
- `updateForecasts` — Retrieves from OpenMeteo the forecasts for the upcoming week for all cities stored in the database and inserts them into Redis. It calls the function shown in Figure 9 over all the cities in the database.

4.3 DTO

The DTO (Data Transfer Object) folder includes classes used to encapsulate and transfer data between the different layers of the application. These classes are particularly useful as they allow the inclusion of additional fields or transformations without modifying the structure of the model classes. For this reason, they serve as flexible data carriers tailored to specific use cases such as API responses or service outputs. A detailed discussion of each DTO is omitted, as they concern purely implementation-level aspects.

4.4 Model

In the `model` folder are defined all the classes representing the core entities of the application.

- `User`
- `City`
- `ExtremeWeatherEvent`
- `HourlyMeasurement`

This folder also contains `enum` types, used to standardize and constrain values across the application:

- `Role` — Defines the possible roles of a user: `ADMIN` or `REGISTERED_USER`.
- `ExtremeWeatherEventCategory` — Defines the possible categories of extreme weather events: `RAINSTORM`, `SNOWSTORM`, `HURRICANE`, `HEATWAVE`, `COLDWAVE`.
- `MeasurementField` — Used in analytics APIs to specify a target measurement; possible values are: `RAINFALL`, `SNOWFALL`, `WINDSPEED`, `TEMPERATURE`.

4.5 Repository

The `repository` folder contains all the interfaces responsible for interacting with the MongoDB database. Each repository extends the `MongoRepository` interface and provides access to the corresponding collection. These interfaces define the methods used throughout the various service classes to access data, mainly defining standard operations such as `findById`. The defined interfaces and relative methods are: The following interfaces are defined to handle data persistence:

- **UserRepository** – manages user data access.
  - `Optional<User> findByUsername(String username);`
    Retrieves a user by their username.
- **CityRepository** – manages access to city-related data.
  - `List<City> findAllByName(String name);`
    Returns all cities with the given name.
  - `List<CityBasicProjection> findAllBy();`
    Returns all cities using the `CityBasicProjection` interface.
  - `List<CityBasicProjection> findAllByNameOrderByFollowers(String name);`
    Returns cities with the specified name, ordered by follower count.
  - `Optional<City> findCityWithOngoingEvents(String cityId);`
    Returns a city only if it has at least one ongoing event (i.e., `dateEnd = null`).
  - `Optional<City> findCityWithEventsOfCategory(String cityId, String category);`
    Returns a city only if it contains at least one event of the specified category.

  - `Optional<City> findCityWithEventsStartedBefore(String cityId, LocalDateTime startTime);`
    Returns a city only if it has events that started before the specified time.
  - `Optional<City> findCityWithEventsInRangeAndCategory(String cityId, String category, LocalDateTime start, LocalDateTime end);`
    Returns a city only if it has events in the given time range and category.

- **HourlyMeasurementRepository** – handles hourly measurement data.
  - `List<HourlyMeasurement> findByCityIdAndTimeBetweenOrderByTimeTimeAsc(String cityId, Date startTime, Date endTime);`
    Returns all measurements for a given city within a time range, sorted by time.
  - `Optional<HourlyMeasurement> findFirstByCityIdOrderByTimeAsc(String cityId);`
    Returns the earliest recorded measurement for a given city.
  - `void deleteByCityIdAndTimeBetween(String cityId, Date startTime, Date endTime);`
    Deletes all measurements for a city within the specified time range.

Chapter 5

Clustering and Sharding

To deploy our database into a distributed environment, we have access to 3 UniPi virtual machines: `10.1.1.9`, `10.1.1.84` and `10.1.1.87`.
Since the application is heavily **read-based** the **master-slave** architecture is by far the best option. The "standard" master-slave architecture would not provide **sharding**. However, completely abandoning sharding would be problematic, since the scalability relative to the amount of data would be compromised. To address this, the adopted solution combines both paradigms by assigning a dedicated master and replica to each shard — a pattern that corresponds to the default architecture of a Redis cluster.

5.1 MongoDB clustering

The mongoDB replicas have been configured using the following parameters:

```
rs.initiate({
  _id: "rswa",
  members: [
    { _id: 0, host: "10.1.1.87:27017", priority: 5 },
    { _id: 1, host: "10.1.1.84:27017", priority: 3 },
    { _id: 2, host: "10.1.1.9:27017", priority: 1 }
  ]
});
```

The connection was configured with $w = 1$ to ensure write acknowledgment from at least one node, thus ensuring **availability**, and $readPreference$ = local to allow fast reads from the **nearest** node.

5.2 MongoDB sharding

The parameters shown in the previous subsection do **not** actually involve sharding. A possible design for sharding could be the following:

- `user`: shard by `_id`;
- `city`: shard by `_id`;
- `hourly_measurement`: shard by `cityId`;

This sharding structure allows you to perform analytics without pulling data from multiple nodes, since hourly measurement from the same city are stored in the **same physical machine**.

5.3 Redis Replica

Each node runs a Redis master and a Redis slave. The configuration is really simple:

```
port 6379 # for the slave, the port used is 6380
cluster-enabled yes
cluster-config-file nodes.conf
```

```
cluster-node-timeout 5000
appendonly yes
bind 0.0.0.0
maxmemory 512mb
maxmemory-policy allkeys-lfu
```

5.4 Redis Sharding

By default, Redis uses the **CRC-16** algorithm to calculate which hashslot a given key belongs to. Unless otherwise specified, this algorithm is performed on whole keys. Using {}, you can specify a substring of the key on which to perform CRC-16. Choosing the region-code as a substring results in data sharding that allows for fast operations, since you only need to access a single node to obtain forecast information for the same city/region.
Here some example regarding the city Pisa:

```
region:{tus}
city:{tus}-pis-43.7085-10.4036
forecast:{tus}-pis-43.7085-10.4036:2025-06-08
```

5.5 CAP theorem

The application is designed to be **AP**. While partition tolerance is essential, it is preferable to provide users with an always-available weather service rather than a strictly consistent one. Therefore, MongoDB parameters are configured to favor fast reads and high availability, as described in the previous section. Redis, by design, also operates as an AP system by default.

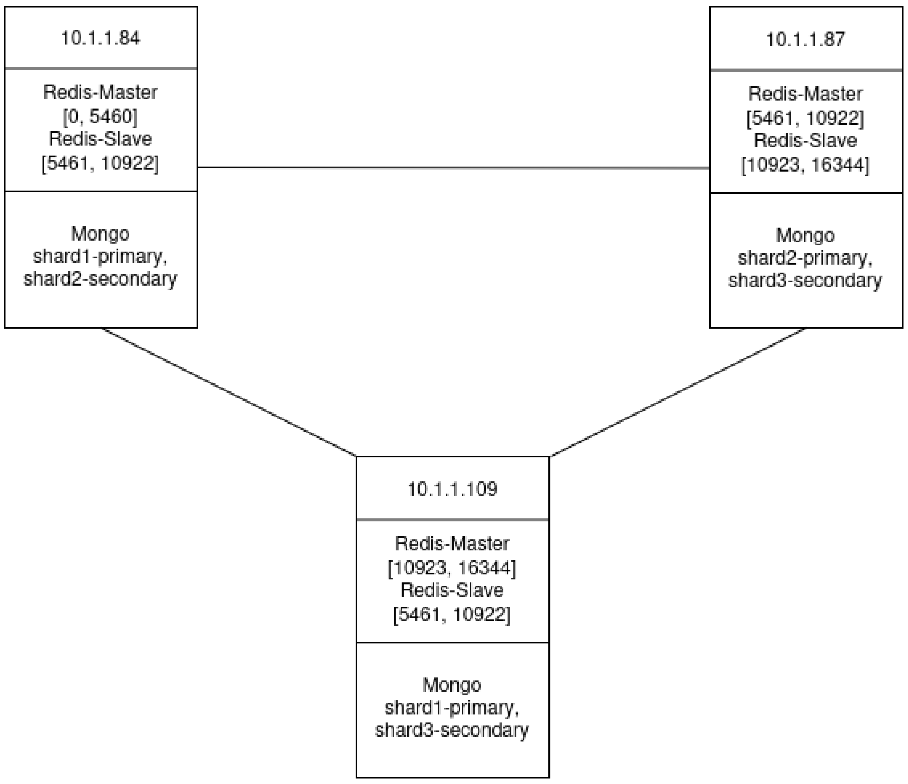

10.1.1.84
Redis-Master [0, 5460] Redis-Slave [5461, 10922]
Mongo shard1-primary, shard2-secondary
10.1.1.87
Redis-Master [5461, 10922] Redis-Slave [10923, 16344]
Mongo shard2-primary, shard3-secondary
10.1.1.109
Redis-Master [10923, 16344] Redis-Slave [5461, 10922]
Mongo shard1-primary, shard3-secondary

Figure 12: Clustering Architecture

Chapter 6

API List

This is the list of APIs in the application. Some APIs are reserved for Admin users, others for Registered users.

6.1 Extreme Weather Event

`DELETE /ewe/duplicates/range` — Deletes duplicate Extreme Weather Events in the given time range, for the given city.

- `cityId`: string
- `startTime`: string
- `endTime`: string

`DELETE /ewe/duplicates/all` — Deletes duplicate Extreme Weather Events for the given city.

- `cityId`: string

6.2 User

`POST /user/register` — Lets a new user register to the application

- `UserDTO`: UserDTO

`POST /user/login` — Lets a registered user or an admin to login into the application

- `UserDTO`: UserDTO

`POST /user/logout` — Lets a registered user or an admin to logout into the application
The UserDTO is structured as:

```
{
    "username": "string",
    "password": "string",
    "email": "string",
    "role": "ADMIN or REGISTERED_USER",
    "adminCode": "string"
}
```

6.3 Forecasts

`GET /forecast/today` — Gets the forecast for the target city for the current day

- `cityId`: string

GET /forecast/today/arbitrary-city — Gets the forecast for any arbitrary city, not saved in the database, for the current day.

- `region`: string
- `latitude`: number
- `longitude`: number
- `elevation`: number

GET /forecast/day — Gets the forecast for the target city for the target day

- `cityId`: string
- `targetDate`: string

GET /forecast/day/arbitrary-city — Gets the forecast for any arbitrary city, not saved in the database, for the target day

- `region`: string
- `latitude`: number
- `longitude`: number
- `elevation`: number
- `targetDate`: string

GET /forecast/week — Gets the forecast for the target city for the next seven days

- `cityId`: string

6.4 City

POST /city/add — Inserts a new city into the application.

- `CityDTO`: CityDTO

Where the cityDTO is structured as:

```
{
    "name": string,
    "latitude": number,
    "longitude": number,
    "elevation": number,
    "eweThresholds": {
        "rainfall": number,
        "snowfall": number,
        "maxTemperature": number,
        "minTemperature": number,
        "windSpeed": number
    },
    "region": string
}
```

`PUT /city/update-thresholds` — Updates the Extreme Weather Event threshold values for the specified city.

- `CityDTO`: CityDTO

`GET /city/by-name` — Retrieves information on all cities with the specified name.

- `cityName`: string

`GET /city/all` — Retrieves information on all cities stored in the database.

`GET /favorites` — Retrieves a list of the user favorite cities in the form: (*name*, *region*).

`PUT /favorites` — Allows a user to add a city to their list of followed cities.

- `targetCityId`: string

`DELETE /favorites` — Allows a user to remove a city from their list of followed cities.

- `targetCityId`: string

6.5 Admin periodic update

`PUT /data-manager/update/forecasts` — Updates the forecasts for the next seven days for all cities.

`PUT /data-manager/update/measurements` — Updates the measurements of all cities with the data from the previous day.

`PUT /data-manager/update/ewes` — Updates the extreme weather events for all cities based on the latest measurements.

6.6 Analytics on measurements per city

All these APIs take as arguments the target city, a measurement type, and a time range within which to search for the measurements.

`GET /analytics/measurement` — Gets a list of measurements of the given measurement type in the time range.

- `cityId`: string
- `measurementField`: string
- `startDate`: string
- `endDate`: string

`GET /analytics/measurement/city/lowest` — Gets the lowest measurements in the time range.

- `cityId`: string
- `measurementField`: string
- `startDate`: string
- `endDate`: string

`GET /analytics/measurement/city/highest` — Gets the highest measurements in the time range.

- `cityId`: string
- `measurementField`: string
- `startDate`: string
- `endDate`: string

`GET /analytics/measurement/city/average` — Gets the average measurement value in the time range.

- `cityId`: string
- `measurementField`: string
- `startDate`: string
- `endDate`: string

`GET /analytics/measurement/city/average-per-month` — Gets the average measurement value per month in the time range.

- `cityId`: string
- `measurementField`: string
- `startDate`: string
- `endDate`: string

6.7 Analytics on measurements per region

All these APIs take as arguments the target region, a measurement type, and a time range within which to search for the measurements.

`GET /analytics/measurement/region/average` — Gets the average values of the given measurement type for each city in the given region.

- `region`: string
- `measurementField`: string
- `startDate`: string
- `endDate`: string

GET /analytics/measurement/region/lowest — Gets the lowest value of the given measurement type recorded for each city in the given region.

- `region`: string
- `measurementField`: string
- `startDate`: string
- `endDate`: string

GET /analytics/measurement/region/highest — Gets the highest value of the given measurement type recorded for each city in the given region.

- `region`: string
- `measurementField`: string
- `startDate`: string
- `endDate`: string

6.8 Analytics on recent measurements

All these APIs take as arguments the number of days prior to the current day and a measurement type.

GET /analytics/measurement/recent/city/total-per-day — Gets the total measurement, calculated as the sum of the hourly measurements, of the target city for each day. [4]

- `cityId`: string
- `measurementField`: string
- `pastDays`: number

GET /analytics/measurement/recent/city/average-per-day — Gets the average measurement of the target city for each day.

- `cityId`: string
- `measurementField`: string
- `pastDays`: number

GET /analytics/measurement/recent/region/total — Gets the total measurement over the past days of each city of the target region.

- `region`: string
- `measurementField`: string
- `pastDays`: number

[4]This API make sense only with a cumulative measurement, such as rainfall and snowfall.

GET /analytics/measurement/recent/region/average — Gets the average measurement over the past days of each city of the target region.

- `region`: string
- `measurementField`: string
- `pastDays`: number

6.9 Analytics on Extreme Weather Events

All these APIs take as argument an extreme weather event category.

GET /analytics/ewe/strength/maximum — Gets the maximum recorded strength of extreme weather events in the given time range for each city in the target city.

- `extremeWeatherEventCategory`: string
- `cityId`: string
- `startDate`: string
- `endDate`: string

GET /analytics/ewe/strength/average — Gets the average strength of extreme weather events in the given time range for each city in the target city.

- `extremeWeatherEventCategory`: string
- `cityId`: string
- `startDate`: string
- `endDate`: string

GET /analytics/ewe/duration/longest — Gets the maximum recorded duration, expressed in hours, of extreme weather events in the given time range for each city in the target city.

- `extremeWeatherEventCategory`: string
- `cityId`: string
- `startDate`: string
- `endDate`: string

GET /analytics/ewe/duration/average — Gets the average duration, expressed in hours, of extreme weather events in the given time range for each city in the target city.

- `extremeWeatherEventCategory`: string
- `cityId`: string
- `startDate`: string
- `endDate`: string

GET /analytics/ewe/count — Gets the number of extreme weather events in the time range for all cities within the specified city.

- `extremeWeatherEventCategory`: string
- `cityId`: string
- `startDate`: string
- `endDate`: string

GET /analytics/ewe/count-per-month — Gets the number of extreme weather events per month of the target city.

- `extremeWeatherEventCategory`: string
- `cityId`: string
- `startDate`: string
- `endDate`: string

GET /analytics/ewe/count-of-at-least-strength — Gets the number of extreme weather events, for all cities within the specified city, whose strength is greater than or equal to the given value.

- `extremeWeatherEventCategory`: string
- `cityId`: string
- `minimumStrength`: number
- `startDate`: string
- `endDate`: string