



# WEATHER FORECAST



*Developed By*  
**Duțu Alin Călin**

# CONTENTS

<b>1</b>	<b>Introduction</b>	<b>2</b>
1.1	Context . . . . .	2
1.2	Objectives . . . . .	2
<b>2</b>	<b>Proposed Solution</b>	<b>3</b>
2.1	Solution presentation . . . . .	3
<b>3</b>	<b>Implementation Details</b>	<b>5</b>
3.1	Docker . . . . .	5
3.2	Database . . . . .	5
3.3	PgAdmin . . . . .	5
3.4	Spring app . . . . .	6
3.4.1	REST API . . . . .	6
3.5	Testing . . . . .	10
<b>4</b>	<b>Conclusions</b>	<b>11</b>
	<b>Bibliography</b>	<b>11</b>

# 1 INTRODUCTION

## 1.1 Context

A web application, often called a web app, is a software program that runs on a web server and can be accessed via a web browser. Unlike traditional desktop applications, which are installed on a local computer, web apps are hosted on remote servers, making them accessible from any device with an internet connection. Web apps range from simple websites with basic interactivity to complex platforms that provide robust functionality, such as online banking systems, e-commerce platforms, and social media networks.

Considering the importance of such apps, Spring, which is one of the most popular frameworks in the Java ecosystem for building robust, scalable, and maintainable web applications, has a prominent impact on the actual industry.

## 1.2 Objectives

This project aims to provide an overview of developing a backend application and serves as practice for testing Spring API along with other tools for backend development. The following objectives are taken into consideration:

- Create a database that manages at least 3 entities
- Implement a middle layer that connects the backend app to a database
- Implement multiple REST API endpoints that execute actions and process data using different URLs and inputs

## 2 PROPOSED SOLUTION

### 2.1 Solution presentation

The project aims to implement a web app that registers and stores data about the weather in different cities and countries for a forecast. Knowing the project's purpose and complexity, the following architecture has been considered:

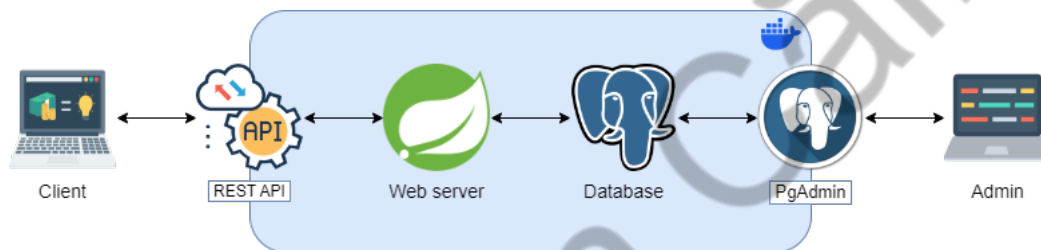


Figure 1: Architecture <sup>[1]</sup>

There are 6 components identified in the architecture:

- Client - The entity that can interact with the app through HTTP requests
- REST API - The middleware component between the client and the web server
- Web server - The component that executes client's requests and returns the results
- Database - The component that stores data about the weather and provides it when requested
- PgAdmin - The middleware tool to administrate the database
- Admin - The authorized device that administers the database

The client is any device that has access to a web browser or that can call the web server using HTTP requests. The client communicates with the web server using the REST API, which can interpret HTTP requests received by the client and call the corresponding functions in the Web server.

The web server is an app developed in Spring that receives the HTTP request from the client, executes a series of commands that can interact with the database, and then provides a response that is sent back to the client in an HTTP reply.

<sup>1</sup><https://www.freepik.com/>. Icons provided by Freepik

The database is an instance of a PostgreSQL[1] that stores the provided data. This database offers an API for data storage and data fetch used by the Spring web server. In addition, for administering the database and its data, PostgreSQL provides the pgAdmin web app to which an administrator can connect.

The web server and database components work as independent Docker services that communicate with each other in an isolated environment through a dedicated network.

Duțu Alin Călin

## **3 IMPLEMENTATION DETAILS**

### **3.1 Docker**

As stated, the Weather Forecast app requires Docker to compile and deploy the components as isolated services that communicate internally within a dedicated network and externally with the client through the REST API component and with the admin using the pgAdmin tool.

The whole configuration is written in the Docker Compose file.

### **3.2 Database**

The database service is used for storing the upcoming data and fetching at request. This service is also administered by the admin using the PgAdmin tool.

In Docker, this service is created with the image of a PostgreSQL database (postgres) and it comes with a persistent volume, the default 5432 port exposed internally in Docker, access to an internal network to communicate with the spring app (spring-postgres), and access to another internal network to communicate with the PgAdmin tool (postgres-pgadmin). Additionally, the service has some environment variables that define the database name and the default credentials for database access.

### **3.3 PgAdmin**

The pgAdmin tool is used to connect and administer the PostgreSQL database through an internal network and can be accessed externally by an administrator.

In Docker, this tool is created using a pgAdmin image and it comes with a persistent volume, the external 5050 port that binds to the internal port 80, and access to the "postgres-pgadmin" internal network to communicate with the database. Additionally, the service has some environment variables that define the default credentials for access to the administration tool.

### 3.4 Spring app

The Spring App, also named as web server, is the main backend component developed to handle operations requested by the client. The entities stored and processed in the app are: Countries, Cities, and Temperatures. The dependencies between these entities, which can be visualized down below, are set programmatically in the app and applied in the database using Hibernate[2].

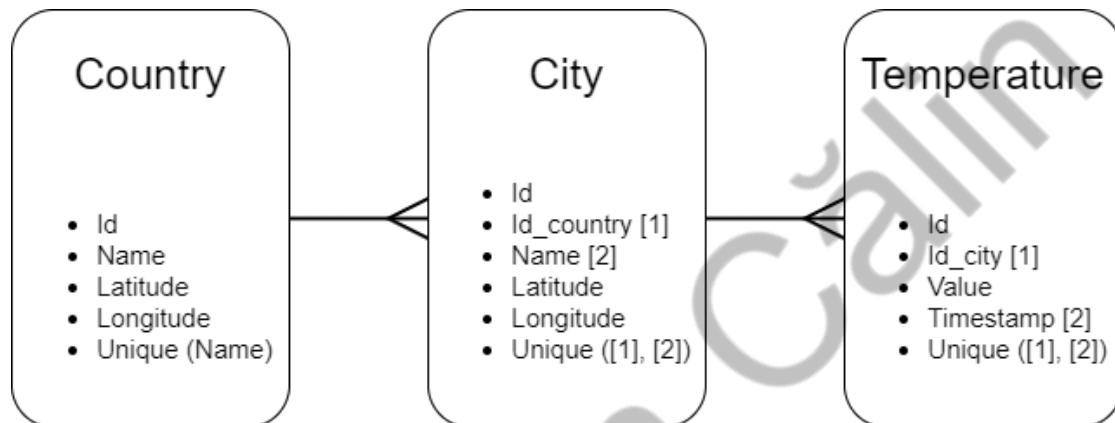


Figure 2: Entity dependency

#### 3.4.1 REST API

The REST API is the middleware component between the client and the web server app. This component captures the HTTP requests sent by the client and lets the web server execute them. The output of the execution will be provided by the web server and sent back to the client as an HTTP reply. The following operations can be processed:

- Add a country
  - URL: "/api/countries"
  - Request type: POST
  - Payload: New country data
  - Results:
    - \* Success - HTTP Code 201
    - \* Data Conflict - HTTP code 409
- Get all countries
  - URL: "/api/countries"

- Request type: GET
- Results:
  - \* Success - HTTP Code 200 + List of countries
- Modify a country
  - URL: "/api/countries/id"
  - Request type: PUT
  - Payload: Country id + modified data
  - Results:
    - \* Success - HTTP Code 200
    - \* Bad request - HTTP Code 400
    - \* Not Found - HTTP Code 404
    - \* Data conflict - HTTP Code 409
- Delete a country
  - URL: "/api/countries/id"
  - Request type: DELETE
  - Payload: Country id
  - Results:
    - \* Success - HTTP Code 200
    - \* Bad request - HTTP Code 400
    - \* Not Found - HTTP Code 404
- Add a city
  - URL: "/api/cities"
  - Request type: POST
  - Payload: New city data
  - Results:
    - \* Success - HTTP Code 201
    - \* Bad request - HTTP Code 400
    - \* Not Found - HTTP Code 404
    - \* Data conflict - HTTP Code 409
- Get all cities
  - URL: "/api/cities"
  - Request type: GET
  - Results:
    - \* Success - HTTP Code 200 + List of cities



- Get all cities from a country
  - URL: `"/api/cities/country/country_id"`
  - Request type: GET
  - Payload: Country id
  - Results:
    - \* Success - HTTP Code 200 + List of cities
- Modify a city
  - URL: `"/api/cities/id"`
  - Request type: PUT
  - Payload: City id + modified data
  - Results:
    - \* Success - HTTP Code 200
    - \* Bad request - HTTP Code 400
    - \* Not Found - HTTP Code 404
    - \* Data conflict - HTTP Code 409
- Delete a city
  - URL: `"/api/cities/id"`
  - Request type: DELETE
  - Payload: City id
  - Results:
    - \* Success - HTTP Code 200
    - \* Bad request - HTTP Code 400
    - \* Not Found - HTTP Code 404
- Add temperature
  - URL: `"/api/temperatures"`
  - Request type: POST
  - Payload: New temperature data
  - Results:
    - \* Success - HTTP Code 201
    - \* Bad request - HTTP Code 400
    - \* Not Found - HTTP Code 404
- Get the temperatures from a location over a certain time interval
  - URL: `"/api/temperatures"`
  - Request type: GET
  - Payload: Latitude, Longitude, Date interval

- Results:
  - \* Success - HTTP Code 200 + List of temperatures
- Get all temperatures for a city over a certain time interval
  - URL: "api/temperatures/cities/city\_id"
  - Request type: GET
  - Payload: City id + Date interval
  - Results:
    - \* Success - HTTP Code 200 + List of temperatures
    - \* Not Found - HTTP Code 404
- Get all temperatures for a country over a certain time interval
  - URL: "api/temperatures/countries/country\_id"
  - Request type: GET
  - Payload: Country id + Date interval
  - Results:
    - \* Success - HTTP Code 200 + List of temperatures
    - \* Not Found - HTTP Code 404
- Modify temperature data
  - URL: "/api/temperatures/id"
  - Request type: PUT
  - Payload: Temperature id + modified data
  - Results:
    - \* Success - HTTP Code 200
    - \* Bad request - HTTP Code 400
    - \* Not Found - HTTP Code 404
    - \* Data conflict - HTTP Code 409
- Delete temperature data
  - URL: "/api/cities/id"
  - Request type: DELETE
  - Payload: Temperature id
  - Results:
    - \* Success - HTTP Code 200
    - \* Bad request - HTTP Code 400
    - \* Not Found - HTTP Code 404

Additionally, there are other libraries used in the project that made developing much easier:

- postgresql - library that provides an adapter for postgresql database
- lombok[3] - for providing constructors using tags only
- mapstruct[4] - for converting instances of objects to entities and back

The app image is created using a Docker file and Maven[5] to compile, build, and pack the app as a Docker image with the default port 5000 exposed. This image is used in Docker Compose to create the container that exposes port 8080 to set the external access to the service.

The app connection with the database is set using the Hibernate[2]. The connection parameters are set in a resource file where the database URL and credentials are found along with other SQL properties.

### 3.5 Testing

For testing the proposed solution, the following methods have been used:

- Postman - Used for testing the REST API using the general cases
- Manual testing - Used for testing corner cases

## 4 CONCLUSIONS

In conclusion, this program aims to provide an overview of backend development experience by developing and deploying a full backend app for weather forecasts. The app serves as practice for discovering multiple backend tools such as Spring, PostgreSQL, and other libraries that helped implement the project at a very fast pace, along with popular deployment tools such as Docker that were used to host the needed services for the app to work.

## Bibliography

- [1] The PostgreSQL Global Development Group. PostgreSQL Database. <https://www.postgresql.org/>. Latest access: 01.09.2024.
- [2] Commonhaus Foundation. Hibernate. Latest access: 01.09.2024.
- [3] Roel Spilker and Reinier Zwitterloot. Project Lombok. <https://projectlombok.org/>. Latest access: 01.09.2024.
- [4] Map Struct - Java bean mapping the easy way! <https://mapstruct.org/>. Latest access: 01.09.2024.
- [5] The Apache Software Foundation. Apache Maven. <https://maven.apache.org/>. Latest access: 01.09.2024.
- [6] SPRC team @ Polytechnic University of Bucharest. REST API & Docker. [https://archive.curs.upb.ro/2022/pluginfile.php/258913/mod\\_folder/content/0/SPRC\\_2022\\_2023\\_Tema\\_2.pdf?forcedownload=1](https://archive.curs.upb.ro/2022/pluginfile.php/258913/mod_folder/content/0/SPRC_2022_2023_Tema_2.pdf?forcedownload=1). Latest access: 01.09.2024.