# WEATHER APP

**PREPARED FOR**

Nuhi Besimi

**PREPARED BY**

Zana Ramuko

Albinot Ismaili

JAN, 03, 2023

**JAVA AND SPRING BOOT**

UNIVERSITETI I EVROPËS JUGLINDORE
УНИВЕРЗИТЕТ НА ЈУГОИСТОЧНА ЕВРОПА
SOUTH EAST EUROPEAN UNIVERSITY

# INTRODUCTION

The project as a basis for the achievements had to enable the connection to the Api and the data that will be returned from the Api to be stored in the database, but during the development we decided not to have a connection to the Api but to be able to use it as a REST API .
The project consists of 4 classes which are: **Forecast**, **location Forecast** , **Location** and **Weather Data**.

To begin with, the project started with the **Location** class, which contains the location data, for example: city, state, longitude, latitude, and also an id which will be connected to the Weather Data class because that is where we will collect the data.

The next class will be **Forecast** which will contain more detailed data and for a longer period, the class will contain: minTemperature, maxTemperature, ForecastTemperature, ForecastPrecipitation, ForecastWindSpeed, ForecastHumidity, we will also have a date column which will be initialised as LocalDate .
The class will also contain an id which will be possible to connect to other classes and will contain the id of the location class as we will first need the location to place the data for those places.

The **LocationForecast** class has only two attributes, which are Location ID and Forecast.

Then we have the **WeatherData** class in which we can find the simple attributes for a weather report in a country, it contains an id, temperature, precipitation, windSpeed, humidity and a LocalDataTime attribute presented as a date.

# Rest endpoints

In the following, we will mention all the rest endpoints for each of the classes of the project

### Location

```
@PostMapping
   public Location createLocation(@RequestBody Location location) {
        locationService.save(location);
        return location;
   }
```

The part to create a Location which requires the data to be filled in that contains for example: country, city, longitude, latitude and id as well

```
@PutMapping("/{id}")
   public Location updateLocation(@PathVariable Long id, @RequestBody
Location location) {
        location.setId(id);
        locationService.update(location);
        return location;
   }
```

Through PutMapping we can update the data of a location which we will find through the id that we set at the beginning when we created it with the Post method

```
@GetMapping
    public List<Location> getAllLocations() {
        return locationService.findAll();
    }
```

This allows us to list all the locations created through the Post method

```
@GetMapping("/{id}")
    public Location getLocationById(@PathVariable Long id) {
        return locationService.findById(id);
    }
```

We can also find locations by their id

```
@DeleteMapping("/{id}")
    public void deleteLocation(@PathVariable Long id) {
        locationService.deleteById(id);
    }
```

If we need to remove any location from the list, we can do it with the delete method

### Forecast

```
@PostMapping
    public Forecast save(@RequestBody Forecast forecast) {
        return forecastService.save(forecast);
    }
```

To add data to a forecast

```
@GetMapping("/{id}")
   public Forecast findById(@PathVariable Long id) {
        return forecastService.findById(id);
   }
```

To find any of the forecasts by id

```
@GetMapping
   public List<Forecast> findAll() {
        return forecastService.findAll();
   }
```

To return all the listed forecasts

```
@PutMapping("/{id}")
   public Forecast update(@RequestBody Forecast forecast) {
        return forecastService.update(forecast);
   }
```

To change the data within a given forecast by id

```
@DeleteMapping("/{id}")
   public void deleteById(@PathVariable Long id) {
        forecastService.deleteById(id);
   }
```

To delete any of the forecasts by id

### WeatherData

All the methods are similar to the classes explained above, even for the Weatherdata class, for that reason we will only show them without explanation.

```
@GetMapping
    public List<WeatherData> getAllWeatherData() {
        return weatherDataService.findAll();
    }

    @PostMapping
    public WeatherData createWeatherData(@RequestBody WeatherData
weatherData) {
        weatherDataService.save(weatherData);
        return weatherData;
    }

    @GetMapping("/{id}")
    public WeatherData getWeatherDataById(@PathVariable Long id) {
        return weatherDataService.findById(id);
    }

    @PutMapping("/{id}")
    public WeatherData updateWeatherData(@PathVariable Long id,
@RequestBody WeatherData weatherData) {
        weatherData.setId(id);
        weatherDataService.save(weatherData);
        return weatherData;
    }

    @DeleteMapping("/{deleteId}")
    public void deleteWeatherData(@PathVariable Long deleteId) {
        weatherDataService.delete(weatherDataService.findById(deleteId));
```

# Spring Initializr

### SPring Data Jpa

Helps us to create dynamic queries based on the requirement at run time. Spring Data Jpa Specifications allows a combination of the attributes or properties of a domain or entity class and creates a query.

### Spring WEB

Helps build  RESTful applications using Spring MVC. Uses Apache Tomcat as the default embedded container.

### MySql Driver

Without the MySQL driver, the Spring Boot application would not be able to access the database and perform any database-related operations. By including the MySQL driver as a dependency in the application, the application can interact with the database and use its data to perform various tasks.