Weather App – Documentation

Project Overview

The Weather App is a RESTful API built using Spring Boot. This project is integrated with the Visual Crossing API to retrieve weather data which is then processed to provide insight into daylight duration and rain conditions for the specified cities.

Feature Implementation

Feature 1: Compare Daylight Hours Between Two Cities

This feature allows users to compare the daylight duration between two cities by calculating the time difference between sunrise and sunset for each city in seconds.

From the *WeatherService* class, the weather data for each city is fetched using the *WeatherRepository* class. Then after parsing this data into the *CityInfo* class, the *sunrise* and *sunset* fields are extracted. The calculation is then as simple as using this line of code: *Duration.between(LocalTime.parse(sunrise), LocalTime.parse(sunset))*. The duration is seconds for both cities are then compared to return the appropriate response message to the *WeatherController*.

Below are all the relevant code snippets that provide this functionality:

WeatherController.java:

```java
/**
 * Endpoint for comparing the daylight hours of two cities and specifies which city has the longer daylight duration.
 *
 * @param city1 The first city.
 * @param city2 The second city.
 * @return A HTTP response with the body containing a string stating which city has the longer daylight duration.
 */
@GetMapping("/compare-daylight/{city1}/{city2}")
@Operation(summary = "Compare daylight hours", description = "Compares the daylight duration between two cities and states which city has the longer daylight duration.")
public ResponseEntity<String> compareDaylightHours(@PathVariable("city1") String city1, @PathVariable("city2") String city2) {
  String longestDay = weatherService.compareDaylightHours(city1, city2);

  return ResponseEntity.ok(longestDay);
}
```

WeatherService.java

```java
/**
 * Compares the daylight duration between the given cities.
 *
 * @param city1 The first city.
 * @param city1 The second city.
 * @return A message specifiying which city has the longer daylight duration.
 */
public String compareDaylightHours(String city1, String city2) {
  CityInfo ci1 = weatherRepo.getByCity(city1, includeLangId:false);
  CityInfo ci2 = weatherRepo.getByCity(city2, includeLangId:false);

  long city1Daylight = ci1.getDaylightDuration();
  long city2Daylight = ci2.getDaylightDuration();

  if (city1Daylight > city2Daylight) {
    return ci1.getAddress() + " has the longer daylight duration.";
  } else if (city1Daylight < city2Daylight) {
    return ci2.getAddress() + " has the longer daylight duration.";
  } else {
    return "Both cities have the same daylight duration.";
  }
}
```

```java
/**
 * Retrieves the weather forecast for a given city.
 *
 * @param city The name of the city for which to fetch weather data.
 * @return A CityInfo object containing weather details.
 * @throws IllegalArgumentException If the city is not found.
 */
public CityInfo forecastByCity(String city) {
  CityInfo ci = weatherRepo.getByCity(city, includeLangId:false);

  if (ci == null) {
    throw new IllegalArgumentException("City data could not be found for: " + city);
  }

  return ci;
}
```

VisualcrossingRepository.java

```java
/**
 * Retrieves the CityInfo object from Visual Crossing Weather.
 *
 * @param city The name of the city for which to fetch weather data.
 * @param includeLangId specifies whether the weather data should include code names to describe weather conditions.
 * @return A CityInfo object containing weather details.
 * @throws HttpClientErrorException If API request failed (e.g., invalid city, invalid API key).
 * @throws HttpServerErrorException If the server encounters an internal error.
 * @throws Exception If an unexpected error occurs during the API call.
 */
public CityInfo getByCity(String city, boolean includeLangId) {
  String uri = url + "timeline/" + city + "?key=" + key;
  uri = includeLangId ? uri + "&lang=id" : uri;

  try {
    return restTemplate.getForObject(uri, responseType:CityInfo.class);
  } catch(HttpClientErrorException | HttpServerErrorException e) {
    throw e;
  } catch (Exception e) {
    throw e;
  }
}
```

CityInfo.java:

```java
/**
 * Calculates the daylight duration for the city using the sunrise and sunset times.
 *
 * @return The daylight duration in seconds.
 * @throws IllegalStateException If the sunrise or sunset fields are empty.
 * @throws IllegalArgumentException If the time format for sunrise and sunset are invalid.
 */
public long getDaylightDuration() {
  String sunrise = getSunrise();
  String sunset = getSunset();

  if (sunrise == null || sunset == null) {
    throw new IllegalStateException("Sunrise or sunset data is missing for: " + address);
  }

  try {
    return Duration.between(LocalTime.parse(sunrise), LocalTime.parse(sunset)).getSeconds();
  } catch(DateTimeParseException e) {
    throw new IllegalArgumentException("Invalid time format for sunrise or sunet for: " + address);
  }
}
```

```java
/**
 * Accesses the sunrise field.
 *
 * @return The sunrise time.
 */
public String getSunrise() {
  return currentConditions.sunrise;
}
```

```java
/**
 * Accesses the sunset field.
 *
 * @return The sunset time.
 */
public String getSunset() {
  return currentConditions.sunset;
}
```

Feature 2: Rain Check for Two Cities

This feature checks if it is currently raining in the two cities provided.

Similarly, the weather data is fetched and then the *conditions* field is extracted from the *CityInfo* object after being parsed.

A key point to note is the addition of a second parameter to the *getByCity* function. The purpose of this is to indicate whether the *lang=id* should be added to the URI when making the API request to Visual Crossing. This returns the weather data using code names to describe weather conditions. For example, "Heavy Rain" would then be "type_25". This provides a more straightforward approach to checking for the rain conditions and is the recommended approach as stated in Visual Crossing's documentation.

The *conditions* field, which is now provided as a comma separated string of codes, can be split into an array which is iterated over to check whether it contains a code name which indicates rainy weather. This is done using a *Set Collection* of code names which have been

hardcoded. The mapping from code names to human-readable words is provided in the following spreadsheet: Visual Crossing Weather Translations - Google Sheets.

Based on this check, which returns a *Boolean*, the appropriate response message is sent to the *WeatherController* class.

Below are all the relevant code snippets that provide this functionality:

WeatherController.java:

```java
/**
 * Endpoint for checking the weather conditions of two cities and specifies whether they're experiencing raining or not.
 *
 * @param city1 The first city.
 * @param city2 The second city.
 * @return A HTTP response with the body containing a string stating where it is currently raining.
 */
@GetMapping("/rain-check/{city1}/{city2}")
@Operation(summary = "Check rain conditions", description = "Checks the weather conditions of two cities and states where it is raining.")
public ResponseEntity<String> rainCheck(@PathVariable("city1") String city1, @PathVariable("city2") String city2) {
  String rainConditions = weatherService.rainCheck(city1, city2);

  return ResponseEntity.ok(rainConditions);
}
```

WeatherService.java:

```java
/**
 * Checks the weather conditions in the given cities.
 *
 * @param city1 The first city.
 * @param city1 The second city.
 * @return A message specifiying where it is currently raining.
 */
public String rainCheck(String city1, String city2) {
  CityInfo ci1 = weatherRepo.getByCity(city1, includeLangId:true);
  CityInfo ci2 = weatherRepo.getByCity(city2, includeLangId:true);

  boolean city1HasRain = ci1.isRaining();
  boolean city2HasRain = ci2.isRaining();

  if (city1HasRain && city2HasRain) {
    return ci1.getAddress() + " and " + ci2.getAddress() + " are experiencing rain.";
  } else if (city1HasRain) {
    return ci1.getAddress() + " is experiencing rain";
  } else if (city2HasRain) {
    return ci2.getAddress() + " is experiencing rain.";
  } else {
    return "It is not raining in either city.";
  }
}
```

```java
/**
 * Retrieves the weather forecast for a given city.
 *
 * @param city The name of the city for which to fetch weather data.
 * @return A CityInfo object containing weather details.
 * @throws IllegalArgumentException If the city is not found.
 */
public CityInfo forecastByCity(String city) {
  CityInfo ci = weatherRepo.getByCity(city, includeLangId:false);

  if (ci == null) {
    throw new IllegalArgumentException("City data could not be found for: " + city);
  }

  return ci;
}
```

VisualcrossingRepository.java:

```java
/**
 * Retrieves the CityInfo object from Visual Crossing Weather.
 *
 * @param city The name of the city for which to fetch weather data.
 * @param includeLangId specifies whether the weather data should include code names to describe weather conditions.
 * @return A CityInfo object containing weather details.
 * @throws HttpClientErrorException If API request failed (e.g., invalid city, invalid API key).
 * @throws HttpServerErrorException If the server encounters an internal error.
 * @throws Exception If an unexpected error occurs during the API call.
 */
public CityInfo getByCity(String city, boolean includeLangId) {
  String uri = url + "timeline/" + city + "?key=" + key;
  uri = includeLangId ? uri + "&lang=id" : uri;

  try {
    return restTemplate.getForObject(uri, responseType:CityInfo.class);
  } catch(HttpClientErrorException | HttpServerErrorException e) {
    throw e;
  } catch (Exception e) {
    throw e;
  }
}
```

CityInfo.java:

```java
/**
 * Checks whether it is currently raining in the city based on the description in the conditions field.
 *
 * @return A boolean stating whether the city has rain or not.
 */
public boolean isRaining() {
  return Arrays.stream(getConditions().split(regex:","))
    .map(String::trim)
    .anyMatch(RAIN_CODES::contains);
}
```

```java
/**
 * Accesses the conditions field.
 *
 * @return The current weather conditions.
 * @throws IllegalStateException If the conditions field are empty.
 */
public String getConditions() {
  if (currentConditions == null || currentConditions.conditions == null) {
    throw new IllegalStateException("Conditions data is missing for: " + address);
  }

  return currentConditions.conditions;
}
```

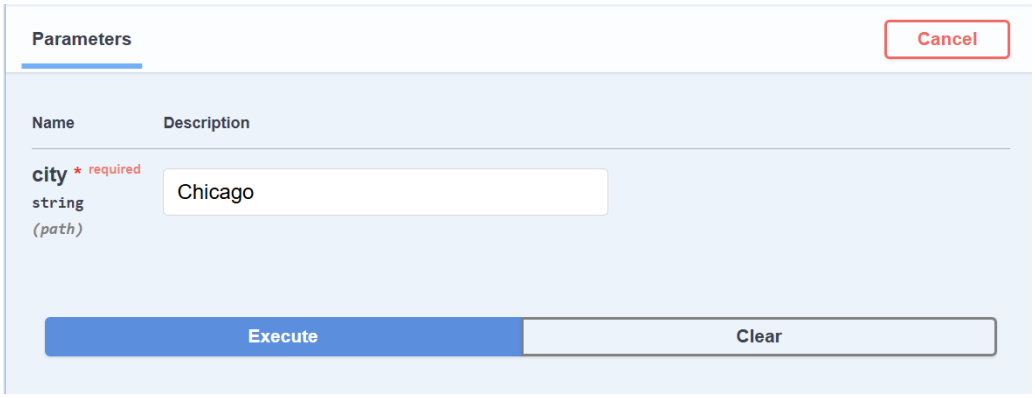
Testing

API Tests

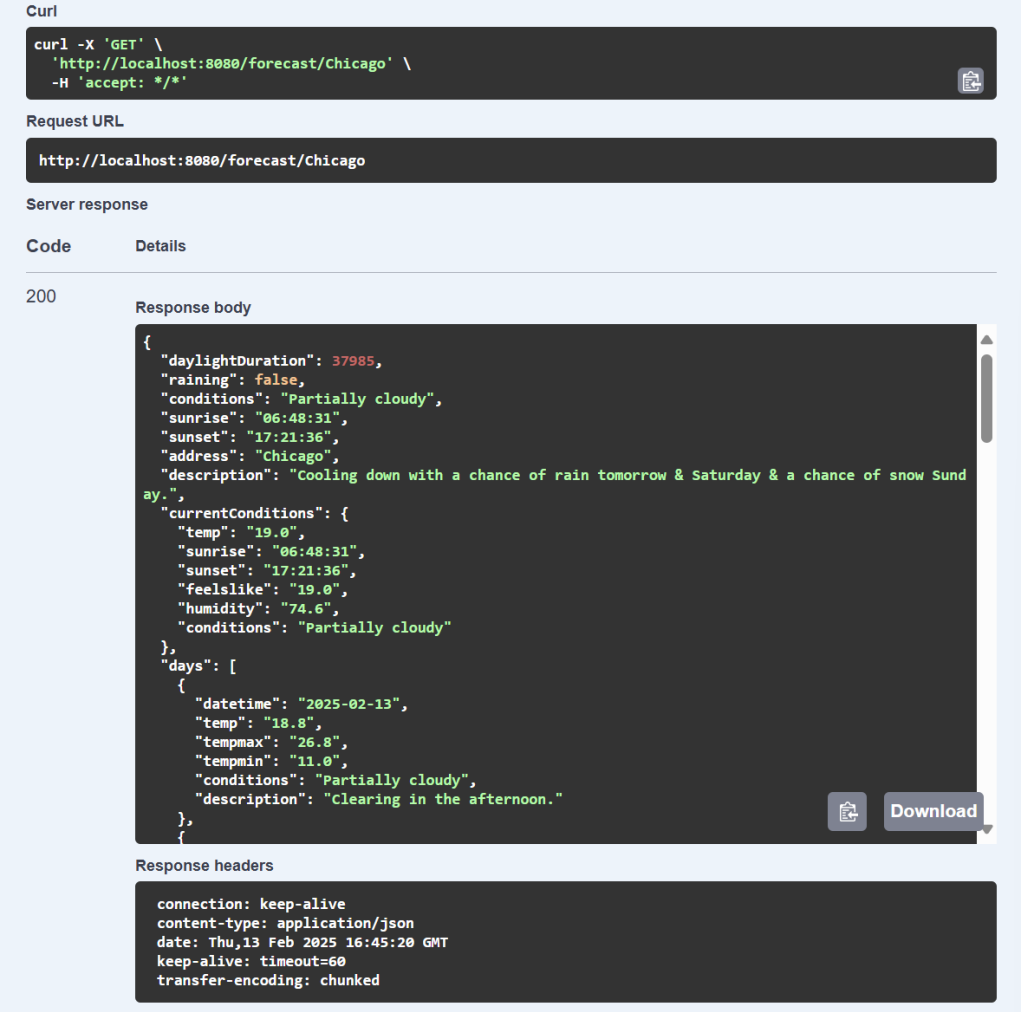
I conducted API tests using Swagger's UI to interact with the endpoints by sending requests and checking the generated responses. This can be accessed by first running the application and then using the link: *http://localhost:8080/swagger-ui.html*. Below are the results for each of the endpoints.

- **GET /forecast/{city}:** Fetches the weather data for the specified city.

  Request parameter:

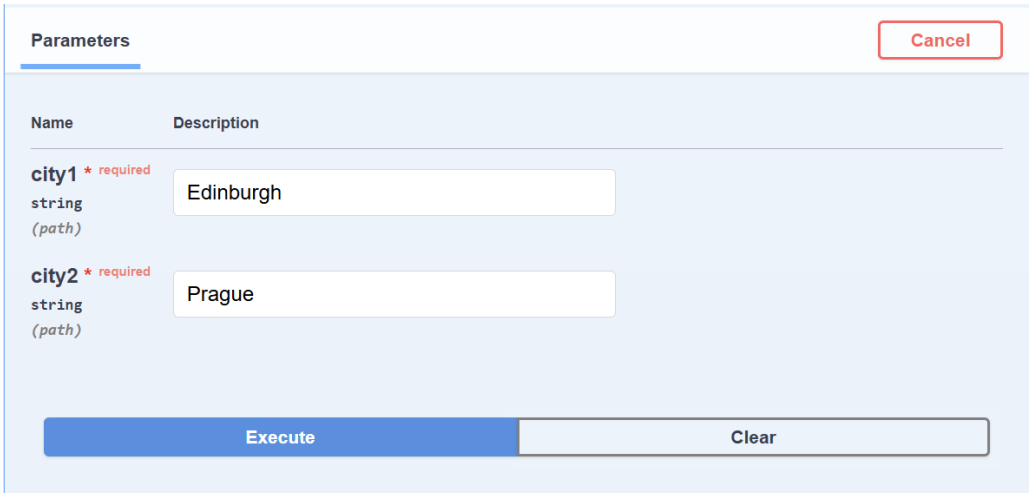


Response message:

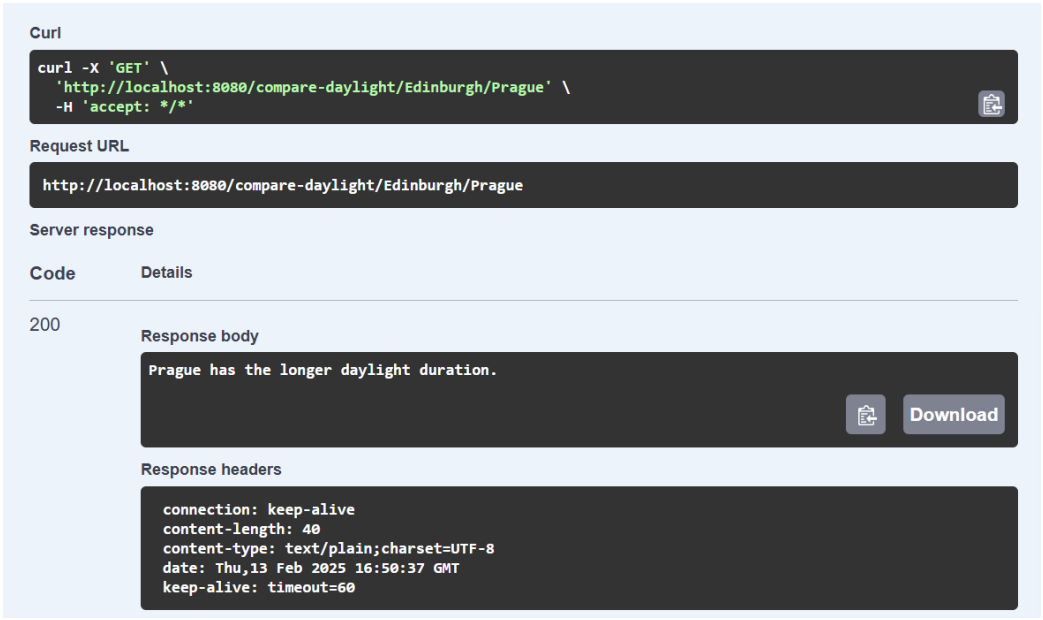


- **GET /compare-daylight/{city1}/{city2}:** Compares the daylight duration between two cities and states which city has the longer daylight duration.

Request parameters:

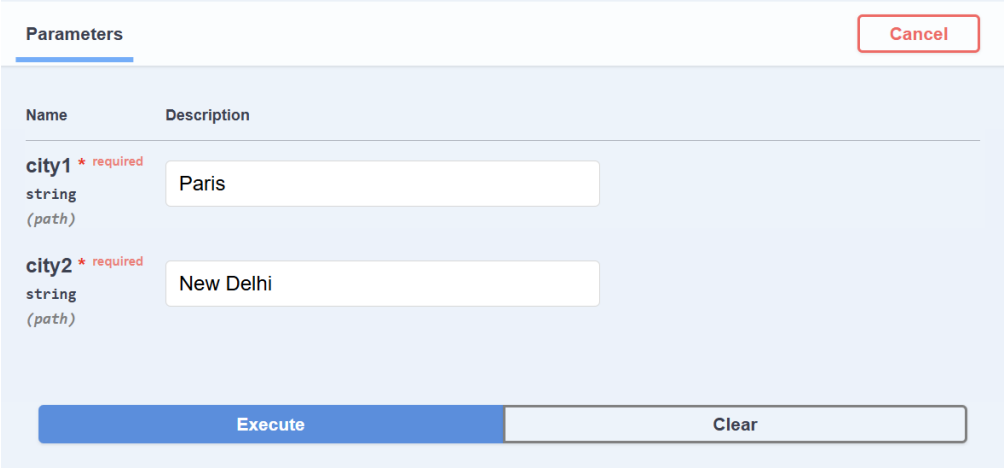


Response message:

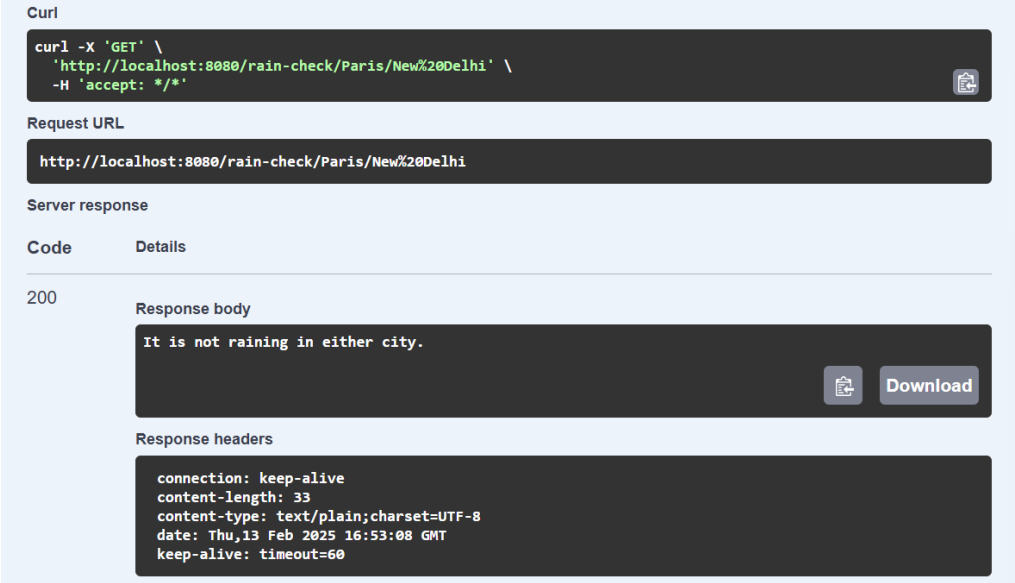


- **GET /rain-check /{city1}/{city2}:** Checks the weather conditions of two cities and states where it is raining.

Request parameters:



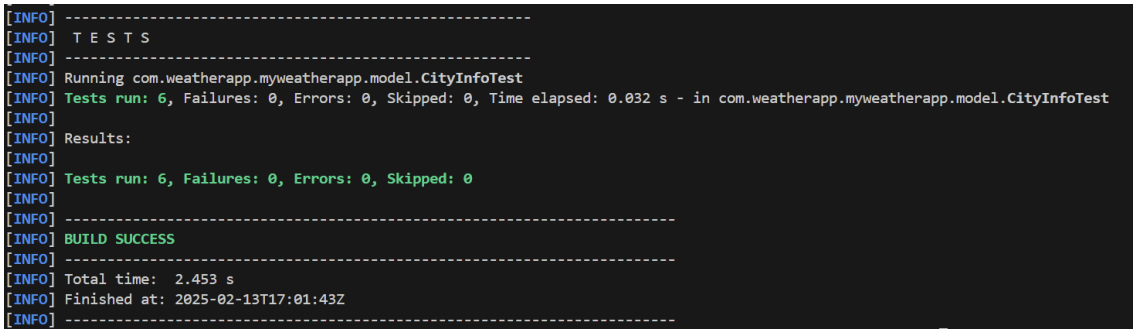
Response message:



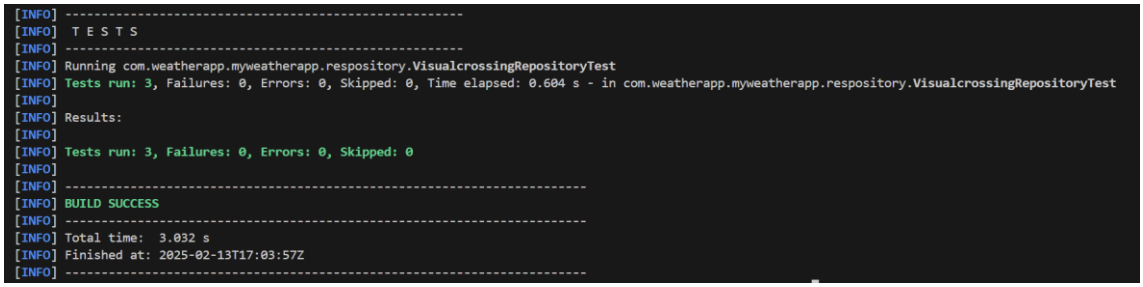
Unit Tests

I thoroughly tested the implementation of each component in the application using Junit5 and Mockito. For each implemented class in the main directory, I provided a corresponding test class which are all stored in their respective subdirectories under the test directory. Blow are the results for each test class:

- **CityInfoTest.java:**



- **VisualcorssingRepositoryTest.java:**



- **WeatherControllerTest.java:**

- **WeatherControllerTest.java:**

Further Improvements

The implementation could be improved by formatting the response message to include details about the weather data. Or optionally, a frontend UI could be provided to provide the user with a better visual of the weather data.

Technologies Used

List of all the tools and frameworks used for this project:

- **Java 17** – Programming language.
- **Spring Boot 3.0.6** – Framework for building REST APIs
- **Swagger** – Generates API documentation.
- **Junit5 and Mockito** – Unit testing.
- **Maven** – build automation and dependency management.

How to Run the Project

1. Clone the repository (BVSanthosh/myweatherapp-tech-test):

   git clone https://github.com/BVSanthosh/myweatherapp-tech-test.git

   cd myweatherapp-tech-test

2. Configure API Key:

   Edit *src/main/resources/application.properties* to include a valid API Key:

weather.visualcrossing.url=https://weather.visualcrossing.com/api/
weather.visualcrossing.key=<your_api_key_here>

3. Run the Application:

   mvn spring-boot:run

4. Run unit tests:

   mvn test

5. Access the Swagger API Docs:

   Open http://localhost:8080/swagger-ui.html