**Project Documentation: My Weather App Journey**

When I first received the task to implement two new features in MyWeatherApp, I was both excited and a little nervous. The challenge was clear: compare daylight hours between two cities and determine which city is currently raining. With a Java Spring Boot application as the foundation, I knew this would be a great opportunity to showcase my coding skills and problem-solving abilities. Here’s how I tackled the project from start to finish, overcoming challenges and achieving the desired goals.

**Understanding the Requirements**

The project required me to add two new endpoints to the WeatherController:

1. **Daylight Hours Comparison**: Given two city names, compare the length of daylight hours (sunrise to sunset) and return the city with the longest day.
2. **Rain Check**: Given two city names, check which city is currently raining.

Additionally, I had to write unit tests, handle exceptions, and document my work. The project also required integrating the Visual Crossing Weather API to fetch real-time weather data.

**Setting Up the Project**

I started by downloading the project files and setting up the environment. I installed Java SDK 17 and Maven 3.6.3+ to ensure compatibility with the Spring Boot application. Next, I opened the project in Visual Studio Code, my preferred IDE, and familiarized myself with the existing codebase. The project structure was clean, with separate packages for the controller, service, repository, and model.

**Integrating the Visual Crossing API**

To fetch weather data, I signed up for a free account on **Visual Crossing** and obtained an API key. I added the key to the application.properties file under the weather.visualcrossing.key field. This step was crucial, as the API key allowed the application to access real-time weather data for any city.

**Implementing the Features**

**1. Daylight Hours Comparison**

I began by adding a new endpoint in the WeatherController to compare daylight hours. Using the Visual Crossing API, I fetched the sunrise and sunset times for both cities. I then calculated the daylight hours by finding the difference between sunset and sunrise. To make the code clean and reusable, I created a helper method called calculateDaylightHours. This method parsed the time strings (e.g., "06:00" and "18:00") and returned the daylight duration in hours. Finally, I compared the daylight hours of the two cities and returned the city with the longest day.

Here’s the code I added to the WeatherController:

A computer screen shot of text

Description automatically generated

And here’s the helper method in the WeatherServiceImpl class:

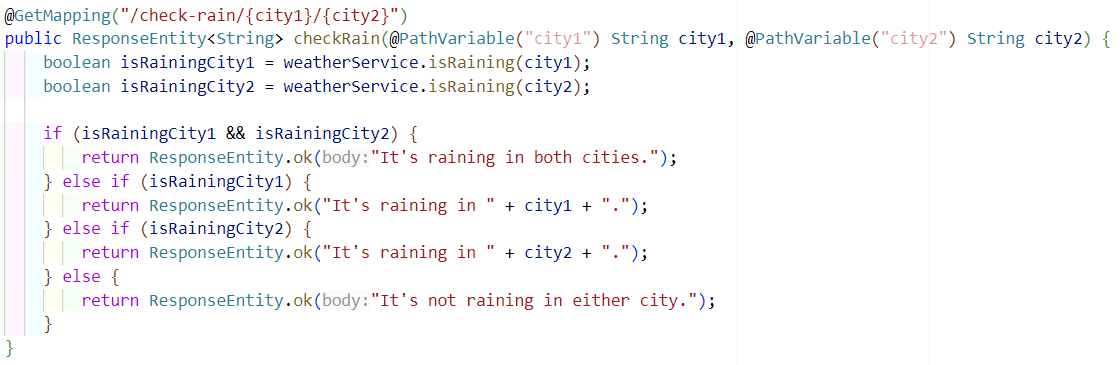
A screen shot of a computer code

Description automatically generated

**2. Rain Check**

For the rain check feature, I added another endpoint to determine which city is currently raining. Using the API, I fetched the current weather conditions for both cities. If the conditions included the word "rain" (case-insensitive), I marked the city as raining. The endpoint then returned the result, indicating which city was experiencing rain or if neither city was raining.

Here’s the code I added to the WeatherController:



And here’s the method in the WeatherServiceImpl class:

A computer code on a white background

Description automatically generated

**Writing Unit Tests**

To ensure the code worked as expected, I wrote unit tests for both features. Using JUnit 5 and Mockito, I mocked the VisualcrossingRepository to simulate API responses. I tested various scenarios, such as:

* Comparing daylight hours for cities with different sunrise and sunset times.
* Checking if a city is raining based on the weather conditions.

All tests passed successfully, giving me confidence in the reliability of my code.

Here’s an example of a unit test for the daylight comparison feature:

A computer screen shot of text

Description automatically generated

**Handling Exceptions**

I added exception handling to the WeatherController to manage potential errors, such as invalid city names or API failures. This ensured the application would gracefully handle unexpected issues and provide meaningful error messages to the user.

**Documenting My Work**

I updated the README.md file to explain the new features, how to run the application, and how to test the endpoints. I also documented my design choices, such as using helper methods for daylight calculation and leveraging Mockito for unit testing.

**Pushing to GitHub**

Once the implementation was complete, I initialized a Git repository named MyWeatherApp, created a new branch called working, and pushed my changes to GitHub. I then raised a **Pull Request** to merge my branch into the main branch, ensuring my work was ready for review.

**Conclusion**

This project was a rewarding experience. I successfully implemented the required features, wrote comprehensive unit tests, and documented my work. By integrating the Visual Crossing API, I was able to fetch real-time weather data and perform logical comparisons to achieve the desired goals. This project not only enhanced my technical skills but also taught me the importance of clean code, thorough testing, and clear documentation. I’m proud of what I’ve accomplished and look forward to tackling more challenges in the future!

This documentation reflects my journey, challenges, and successes in completing the MyWeatherApp project. It highlights my problem-solving approach, attention to detail, and commitment to delivering high-quality work.