**Documentation: Weather App Features Implementation**

**Overview**

In this task, I have implemented two key features in the Weather App using Java (Spring Framework):

1. **Daylight Hours Comparison**: Given two city names, the feature compares the length of daylight hours between them and returns the city with the longest day.
2. **Rain Check**: Given two city names, this feature checks which city is currently experiencing rain.

The features were implemented in the **WeatherController** class as new endpoints.

**Key Decisions Made During Implementation**

**1. Daylight Hours Comparison Feature**

To implement the **Daylight Hours Comparison** feature, I followed these steps:

* **API Integration**: I utilized the **Visual Crossing Weather API** to retrieve sunrise and sunset times for the given cities. These times are returned in the **ISO 8601** format.
* **Sunrise and Sunset Calculation**: The API provides the times in UTC. I converted them to the local time of the respective cities.
* **Daylight Hours Calculation**: The daylight hours are calculated by subtracting the sunrise time from the sunset time, which gives the total daylight duration in minutes.
* **Comparison Logic**: Once the daylight duration for both cities was calculated, I compared the two and returned the city with the longer daylight duration.
* **Error Handling**: In case of invalid city names or API errors, I implemented exception handling to return appropriate HTTP responses (e.g., 400 for invalid input, 500 for server errors).

**2. Rain Check Feature**

For the **Rain Check** feature, I followed these steps:

* **API Integration**: This feature also uses the **Visual Crossing Weather API**, specifically the current weather endpoint, to check if the specified cities are currently experiencing rain.
* **Rain Data Handling**: The API provides weather conditions including the presence of rain. I checked the precipitation field from the API response to determine if it contains values indicating rain.
* **City Comparison**: After determining the rain condition for both cities, I returned the city that is currently experiencing rain. If neither city is raining, the feature returns a message indicating that no city is raining.
* **Error Handling**: Similar to the Daylight Hours feature, I added error handling to address potential API errors, invalid city names, or any other unexpected issues during data retrieval.

**3. Exception Handling**

In both features, I included error handling to ensure a smooth user experience and avoid application crashes. The key exception types handled are:

* **Invalid City Input**: If the API doesn't recognize the city name or returns an error, the system responds with an HTTP 400 Bad Request.
* **API Failures**: In case of network or internal server issues (such as timeouts or unreachable endpoints), I respond with a 500 Internal Server Error and provide a user-friendly error message.

**Code Structure**

**Main Components**

1. **WeatherController**:
   * This is where the endpoints for both features are implemented.
   * The controller handles the HTTP requests, interacts with the external API, and sends the appropriate responses.
2. **WeatherService**:
   * This service class handles the logic for fetching weather data from the Visual Crossing API and performing calculations (daylight duration and rain checks).
   * It separates the business logic from the controller for cleaner code and better maintainability.
3. **WeatherAPI**:
   * This is a utility class or component that wraps the calls to the Visual Crossing API and processes the API responses.
   * It ensures that the response data is properly parsed and error-handled before being passed to the WeatherService.
4. **Application.properties**:
   * Stores the **API key** required to interact with the Visual Crossing Weather API.
   * Ensures the key remains secure and easily configurable without hardcoding it in the codebase.

**API Endpoint Design**

**1. Daylight Hours Comparison Endpoint**

* **Path**: /weather/daylight-comparison
* **Method**: GET
* **Parameters**: city1, city2 (Query parameters for city names)
* **Response**:
  + If both cities are valid, return the city with the longest daylight hours along with the duration.
  + If either city is invalid, return a 400 Bad Request with an appropriate error message.

**Example Request**:

bash

Copy

GET /weather/daylight-comparison?city1=London&city2=Paris

**Example Response**:

json

Copy

{

"city": "Paris",

"daylight\_duration": "10 hours 30 minutes"

}

**2. Rain Check Endpoint**

* **Path**: /weather/rain-check
* **Method**: GET
* **Parameters**: city1, city2 (Query parameters for city names)
* **Response**:
  + Return the city that is currently raining.
  + If neither city is raining, return a message indicating no rain.
  + If there is an issue with the city name or the API response, return a 400 or 500 error.

**Example Request**:

sql

Copy

GET /weather/rain-check?city1=London&city2=Berlin

**Example Response**:

json

Copy

{

"city": "London",

"status": "Currently raining"

}

**Summary of Choices**

* I used the **Visual Crossing Weather API** to retrieve sunrise, sunset, and current weather data for cities.
* I kept the **business logic** (e.g., calculating daylight hours and checking for rain) in a separate service class to keep the controller focused on handling requests.
* I implemented **exception handling** to manage potential errors like invalid city input or API failures gracefully.