**Weather and Bike Sharing API Integration Project**

**Overview**

This project is a Spring Boot-based backend application designed to integrate two APIs: a weather forecasting API and a bike-sharing API. The application fetches weather data for specific districts and information about bike availability at various stations. The data from these APIs is exposed through RESTful endpoints for further use.

**Key Functionalities**

1. **Weather Data Integration**
   * **API Used**: [Tourism Weather API](https://tourism.api.opendatahub.com/v1/Weather/District)
   * **Purpose**: Fetches weather data for districts, including forecasts such as maximum temperature, minimum temperature, and weather descriptions.
2. **Bike Data Integration**
   * **API Used**: [Mobility Bike API](https://mobility.api.opendatahub.com/v2/flat/Bicycle)
   * **Purpose**: Fetches bike-sharing station data, including bike availability, bike types, and station details.
3. **Exposed REST Endpoints**
   * /api/weather: Fetches weather data.
   * /api/bikes: Fetches bike-sharing station data.
   * /api/testing/message: A test endpoint returning a sample string.

**Project Structure**

**Packages and Classes**

1. **Package: com.weatherAPI.weatherApplication**
   * **WeatherApplication**: The main class to bootstrap the Spring Boot application.
2. **Package: com.weatherAPI.weatherApplication.controller**
   * **WeatherController**:
     + Exposes the /api/weather endpoint.
     + Delegates calls to the WeatherService to fetch weather data.
   * **BikeController**:
     + Exposes the /api/bikes endpoint.
     + Delegates calls to the BikeService to fetch bike-sharing data.
   * **TestingController**:
     + Exposes the /api/testing/message endpoint.
     + Used for testing the application.
3. **Package: com.weatherAPI.weatherApplication.model**
   * **WeatherData**:
     + Represents the weather data structure.
     + Contains fields like district name, weather forecast, max temperature, min temperature, and weather description.
   * **BikeData**:
     + Represents the bike-sharing data structure.
     + Contains fields like station name, coordinates, bike availability, and bike types.
   * **BikeApiResponse**:
     + Wraps the response from the bike-sharing API.
4. **Package: com.weatherAPI.weatherApplication.service**
   * **WeatherService**:
     + Fetches weather data from the Tourism Weather API.
     + Uses a RestTemplate bean for HTTP requests.
   * **BikeService**:
     + Fetches bike-sharing data from the Mobility Bike API.
     + Uses a RestTemplate bean for HTTP requests.
5. **Package: com.weatherAPI.weatherApplication.config**
   * **AppConfig**:
     + Defines the RestTemplate bean required by the services.

**API Documentation**

**Weather API**

* **Endpoint**: /api/weather
* **Method**: GET
* **Response**:
  + An array of WeatherData objects containing district name, max temperature, min temperature, and weather description.

**Bike API**

* **Endpoint**: /api/bikes
* **Method**: GET
* **Response**:
  + A list of BikeData objects containing station name, bike availability, and types of bikes available.

**Testing Endpoint**

* **Endpoint**: /api/testing/message
* **Method**: GET
* **Response**: “alles gutt” (sample test message).

**Instructions to Run the Application**

1. **Pre-requisites**:
   * Ensure you have Java 17 or higher installed.
   * Install Maven for dependency management and build.
2. **Clone the Repository**:
   * Clone the project from the repository to your local machine.
3. **Build the Project**:
   * Navigate to the project root directory and run the following command:
   * mvn clean install
4. **Run the Application**:
   * Start the application by executing the following command:
   * mvn spring-boot:run
   * Alternatively, you can run the compiled JAR file located in the target directory:
   * java -jar target/weatherApplication-0.0.1-SNAPSHOT.jar
5. **Access the Application**:
   * The application runs on **port 8080** by default.
   * Access the endpoints:
     + Weather API: http://localhost:8080/api/weather
     + Bike API: http://localhost:8080/api/bikes
     + Testing API: http://localhost:8080/api/testing/message
6. **Handle Port Conflicts**:
   * If port 8080 is already in use, modify the application.properties file located in src/main/resources:
   * server.port=<new\_port\_number>
7. **Run Tests**:
   * Execute unit tests using Maven:
   * mvn test

**Testing and Validation**

**Unit Tests**

* **Tools**: JUnit 5 and Mockito.
* **Purpose**: To validate the functionality of the services and controllers.

1. **WeatherServiceTest**
   * Mocks the RestTemplate.
   * Simulates a successful response from the Weather API and verifies the service behavior.
2. **BikeServiceTest**
   * Mocks the RestTemplate.
   * Simulates a successful response from the Bike API and verifies the service behavior.
3. **WeatherControllerTest**
   * Validates the /api/weather endpoint.
   * Mocks the WeatherService to simulate API responses.
4. **BikeControllerTest**
   * Validates the /api/bikes endpoint.
   * Mocks the BikeService to simulate API responses.
5. **TestingControllerTest**
   * Verifies the /api/testing/message endpoint returns the expected message.
6. **WeatherApplicationTests**
   * Verifies the Spring Boot application context loads correctly.

**Test Results**

* Initial test failures were caused by issues with bean configurations and mismatched expectations in mock responses.
* Adjustments to the RestTemplate configuration resolved the issues.
* Final test suite passed successfully, ensuring application functionality.

**Design Patterns Used**

1. **Dependency Injection (DI)**
   * Applied to inject RestTemplate into WeatherService and BikeService.
2. **Singleton Pattern**
   * The RestTemplate bean is defined as a singleton in the AppConfig class.
3. **Separation of Concerns (SoC)**
   * Services focus on data retrieval, while controllers handle HTTP requests and responses.
4. **Mock Objects (Proxy Pattern)**
   * Mocked RestTemplate instances in test cases to simulate API responses.

**Future Enhancements**

1. **Error Handling**:
   * Implement robust error handling for API calls to manage failures and invalid responses.
2. **API Rate Limiting**:
   * Add mechanisms to handle rate limits imposed by the external APIs.
3. **Caching**:
   * Introduce caching for frequently accessed data to improve performance.
4. **Front-End Integration**:
   * Develop a front-end application to visualize weather and bike-sharing data.
5. **Advanced Features**:
   * Combine data from both APIs to provide insights, such as the probability of bike availability based on weather conditions.