Software Performance Evaluation Project

Name	ID
Abdelrahman Mohamed	19p5922
Amr Hesham	19p5218
Omar Ahmed Ebrahim	20P5806

Submitted to :

Dr Islam El maddah

Eng Ahmed Abd El-Moneam

Weather Data API Project Report

Introduction

The **Weather Data API** project provides a scalable and efficient solution for managing and analyzing weather-related data. It offers functionality to submit, retrieve, and aggregate weather data. By leveraging modern technologies such as FastAPI, asynchronous programming, and caching, the API is optimized for high performance and scalability.

This report outlines the implementation of the API, the profiling and performance testing conducted to evaluate its efficiency, and the optimizations performed to enhance its functionality and reliability.

Weather Data API Features

1. Submit Weather Data

- **Endpoint**: `/data`
- **Method**: `POST`
- **Description**:
  - This endpoint allows users to submit weather data for specific geo-locations.
  - Users can provide `latitude`, `longitude`, `temperature`, and `humidity` values, which are then stored in the API's in-memory data store.
- **Example Use Case**:
  - Collecting weather data from IoT devices or weather stations and storing it for future analysis.

---

2. Get Weather by City

- **Endpoint**: `/getWeatherByCity`
- **Method**: `GET`
- **Description**:
  - Fetches the weather data closest to the center of a specified city.
  - Integrates with the OpenWeatherMap Geocoding API to determine the latitude and longitude of the city.
- **Example Use Case**:
  - Applications that display weather information for specific cities, such as travel or tourism websites.

---

3. Get Weather by Geo-Location

- **Endpoint**: `/getWeatherByGeo`
- **Method**: `GET`
- **Description**:
    - Retrieves the weather data closest to a given geo-location (latitude and longitude).
    - Uses the Haversine formula to calculate the distance between the geo-coordinates and stored weather data points.
- **Example Use Case**:
    - Systems where geo-coordinates (e.g., GPS data) are available, such as maps or logistics applications.

---

4. Get Aggregated Weather Statistics

- **Endpoint**: `/getAggregatedWeather`
- **Method**: `GET`
- **Description**:
    - Computes statistical insights for weather data within a specified radius of a given geo-location.
    - Outputs include:
        - Average temperature and humidity.
        - Variance of temperature and humidity.
- **Example Use Case**:
    - Applications requiring weather analytics over a region, such as agricultural platforms or environmental monitoring systems.

Profiling

In this project, profiling was implemented using Python's built-in **cProfile** library. A middleware was added to the FastAPI application to automatically profile every incoming request and log the performance metrics to the console. Here's how it works:

1. **Middleware Integration**:
    - A custom middleware was added to wrap each HTTP request and response cycle. This middleware uses `cProfile` to measure the time taken by all function calls during the processing of the request.
2. **Logging Metrics**:
    - The profiling middleware logs the following details for each request:
        - Total time taken for the request.

        - Function call count.
        - Time spent in each function (cumulative and individual).
    - These logs are printed to the console for real-time analysis.
3. **Identifying Bottlenecks**:
    - The profiling results revealed slow operations, such as blocking calls to the OpenWeatherMap API and computationally intensive functions like the Haversine formula and data aggregation calculations.

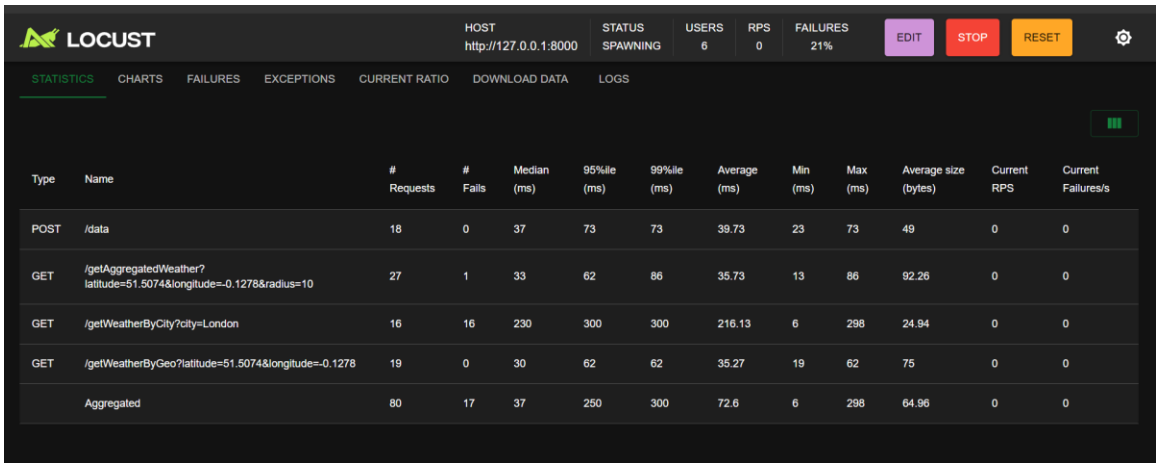
Screen shots :

```
Profiling Results:

         12346 function calls (12186 primitive calls) in 0.092 seconds

   Ordered by: internal time

   ncalls  tottime  percall  cumtime  percall filename:lineno(function)
        6    0.073    0.012    0.073    0.012 {method 'write' of '_io.TextIOWrapper' objects}
       63    0.003    0.000    0.003    0.000 {built-in method nt.stat}
       79    0.001    0.000    0.001    0.000 {built-in method builtins.next}
       66    0.001    0.000    0.001    0.000 {built-in method builtins.compile}
       75    0.001    0.000    0.003    0.000 traceback.py:460(format_frame_summary)
        9    0.000    0.000    0.005    0.001 traceback.py:399(_extract_from_extended_frame_gen)
       78    0.000    0.000    0.000    0.000 weather_data_api.py:38(haversine)
       43    0.000    0.000    0.000    0.000 {method 'fullmatch' of 're.Pattern' objects}
       66    0.000    0.000    0.001    0.000 traceback.py:589(_extract_caret_anchors_from_line_segment)
      450    0.000    0.000    0.001    0.000 traceback.py:318(line)
      689    0.000    0.000    0.000    0.000 {built-in method builtins.isinstance}
      150    0.000    0.000    0.000    0.000 traceback.py:573(_byte_offset_to_character_offset)
      306    0.000    0.000    0.000    0.000 textwrap.py:482(prefixed_lines)
      186    0.000    0.000    0.000    0.000 {method 'join' of 'str' objects}
      635    0.000    0.000    0.000    0.000 {method 'append' of 'list' objects}
  828/820    0.000    0.000    0.000    0.000 {built-in method builtins.len}
      126    0.000    0.000    0.001    0.000 traceback.py:665(emit)
        7    0.000    0.000    0.091    0.013 base_events.py:1909(_run_once)
        3    0.000    0.000    0.000    0.000 {method '_accept' of '_socket.socket' objects}
      525    0.000    0.000    0.000    0.000 {method 'strip' of 'str' objects}
  147/105    0.000    0.000    0.004    0.000 traceback.py:944(format)
       63    0.000    0.000    0.003    0.000 linecache.py:52(checkcache)
    59/57    0.000    0.000    0.091    0.002 {method 'run' of '_contextvars.Context' objects}
      164    0.000    0.000    0.000    0.000 {method 'format' of 'str' objects}
      9/3    0.000    0.000    0.005    0.002 traceback.py:718(__init__)
        8    0.000    0.000    0.000    0.000 {method 'recv' of '_socket.socket' objects}
        9    0.000    0.000    0.003    0.000 traceback.py:525(format)
        6    0.000    0.000    0.036    0.006 h11_impl.py:447(send)
        5    0.000    0.000    0.000    0.000 {function socket.close at 0x0000024666D291C0}
       18    0.000    0.000    0.001    0.000 _connection.py:260(_process_event)
        6    0.000    0.000    0.000    0.000 __init__.py:298(__init__)
        5    0.000    0.000    0.000    0.000 _headers.py:150(normalize_and_validate)
       46    0.000    0.000    0.000    0.000 base_events.py:814(_call_soon)
```

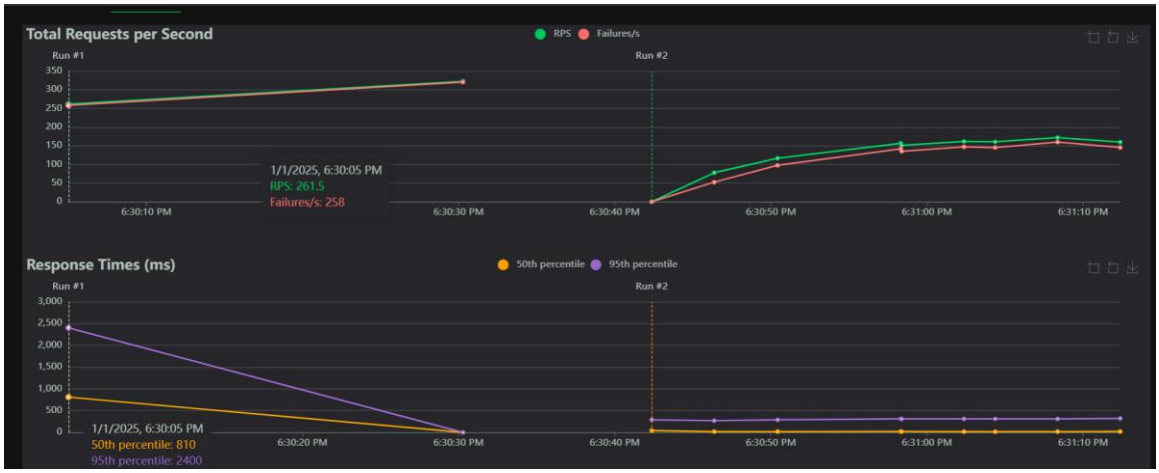
Performance Testing in the Weather Data API

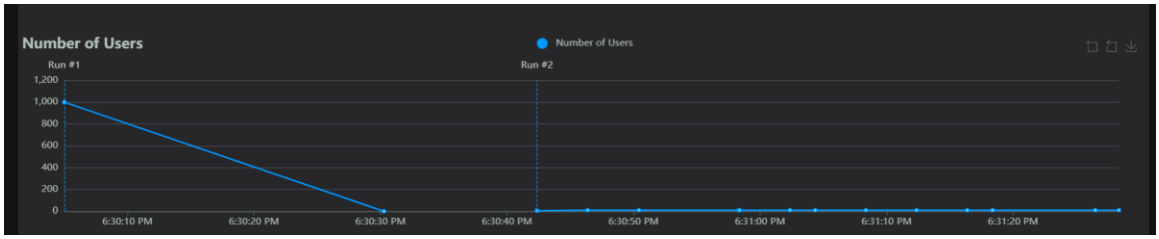
Performance testing is a crucial aspect of ensuring the scalability and reliability of the Weather Data API under varying loads. In this project, performance testing was conducted using **Locust**, a Python-based tool that simulates concurrent user traffic to evaluate the API's responsiveness, throughput, and error rates. A dedicated `locustfile.py` was created to define test tasks for each endpoint (`/data`, `/getWeatherByCity`, `/getWeatherByGeo`, and `/getAggregatedWeather`). These tasks simulate real-world scenarios, such as users submitting weather data, retrieving it by city or coordinates, and calculating aggregated statistics. The Locust web interface allowed the configuration of parameters like the number of simulated users and the spawn rate. Metrics such as requests per second (RPS), response times, and failure rates were monitored during the tests. This approach revealed performance bottlenecks, particularly under high concurrency, and helped validate the impact of optimizations like caching, asynchronous API calls, and multi-worker setups. By incorporating performance testing, the API was fine-tuned to handle real-world traffic efficiently.

Screen Shots :



Type	Name	# Requests	# Fails	Median (ms)	95%ile (ms)	99%ile (ms)	Average (ms)	Min (ms)	Max (ms)	Average size (bytes)	Current RPS	Current Failures/s
POST	/data	18	0	37	73	73	39.73	23	73	49	0	0
GET	/getAggregatedWeather?latitude=51.5074&longitude=-0.1278&radius=10	27	1	33	62	86	35.73	13	86	92.26	0	0
GET	/getWeatherByCity?city=London	16	16	230	300	300	216.13	6	298	24.94	0	0
GET	/getWeatherByGeo?latitude=51.5074&longitude=-0.1278	19	0	30	62	62	35.27	19	62	75	0	0
	Aggregated	80	17	37	250	300	72.6	6	298	64.96	0	0


Total Requests per Second
RPS
Failures/s
Run #1
Run #2
350
300
250
200
150
100
50
0
1/1/2025, 6:30:05 PM
RPS: 261.5
Failures/s: 258
6:30:10 PM
6:30:30 PM
6:30:40 PM
6:30:50 PM
6:31:00 PM
6:31:10 PM
Response Times (ms)
50th percentile
95th percentile
Run #1
Run #2
3,000
2,500
2,000
1,500
1,000
500
0
1/1/2025, 6:30:05 PM
50th percentile: 810
95th percentile: 2400
6:30:20 PM
6:30:30 PM
6:30:40 PM
6:30:50 PM
6:31:00 PM
6:31:10 PM


Number of Users
Number of Users
Run #1
Run #2
1,200
1,000
800
600
400
200
0
6:30:10 PM
6:30:20 PM
6:30:30 PM
6:30:40 PM
6:30:50 PM
6:31:00 PM
6:31:10 PM
6:31:20 PM