Backend Development with Python

Objective:

Design and implement a simple backend system to store and process data using Python, with the ability to handle high-frequency data.

Design a Backend System

The backend system was developed using **FastAPI**, a Python-based framework for building APIs. The system consists of three primary endpoints:

1. **/upload**: Allows uploading a list of location data entries (e.g., timestamp, latitude, longitude). Basic validation ensures data integrity, and exceptions are handled gracefully.
2. **/data**: Retrieves all uploaded location data entries stored in the system.
3. **/summary**: Calculates and returns the average latitude and longitude of all uploaded data. If no data is available, it raises a 404 error.

Simulate Data Generation

A Python script was created to simulate random GPS data generation. The generated data follows this format:

{

"timestamp": "2025-01-23T12:00:00Z",

"latitude": 52.5200,

"longitude": 13.4050

}

This data is used to test the /upload endpoint of the API by sending multiple entries at regular intervals.

Implement the Backend

Functionalities:

1. **Data Storage**:
    - Data is temporarily stored in an in-memory list (data_store).
    - Each data entry adheres to a pydantic model (LocationData) with properties for timestamp, latitude, and longitude.

2. **Endpoints**:
    - **/upload**: Accepts POST requests containing a list of location data entries. Logs successful uploads and validates the input. If data is invalid or missing, appropriate HTTP exceptions are raised.
    - **/data**: Responds to GET requests with all stored data. Raises a 404 error if no data is found.
    - **/summary**: Responds to GET requests by calculating and returning the average latitude and longitude of the stored data. Raises a 404 error if no data is available.
3. **Error Handling**:
    - Implements HTTP exceptions for scenarios such as missing data, invalid inputs, or attempts to access empty data storage.
    - Detailed error messages are provided for better debugging.
4. **Logging**:
    - Configured basic logging to record successful operations and errors.

Test the System

Python Script for Testing:

A Python script was created to test the API's functionalities:

1. **Data Upload**:
    - The script sends POST requests to the /upload endpoint with a list of location data entries.
    - It checks response status codes to confirm successful uploads.
2. **Data Retrieval**:
    - Sends a GET request to the /data endpoint to retrieve all uploaded data.
    - Displays the retrieved data in a user-friendly format.
3. **Summary Retrieval**:
    - Sends a GET request to the /summary endpoint to retrieve the average latitude and longitude.
    - Formats and displays the response for easy interpretation.

Observed Outputs:

Data Upload Output:

bash

CopyEdit

INFO: 127.0.0.1:56475 - "POST /upload HTTP/1.1" 200 OK

INFO:root:Returning 3 data entries.

- Three data entries were successfully uploaded.

2. **Data Retrieval Output:**

bash

CopyEdit

INFO: 127.0.0.1:56476 - "GET /data HTTP/1.1" 200 OK

- The system correctly responded with the uploaded data:

json

CopyEdit


```
{
  "data": [
    {
      "timestamp": "2025-01-23T12:00:00Z",
      "latitude": 52.52,
      "longitude": 13.405
    },
    {
      "timestamp": "2025-01-23T12:05:00Z",
      "latitude": 48.8566,
      "longitude": 2.3522
    },
    {
      "timestamp": "2025-01-23T12:10:00Z",
      "latitude": 34.0522,
      "longitude": -118.2437
```

```
  }
 ]
}
```

3. **Summary Retrieval Output:**

bash

CopyEdit

```
INFO:     127.0.0.1:56477 - "GET /summary HTTP/1.1" 200 OK
```

- The system calculated the average latitude and longitude as:

json

CopyEdit

```
{
 "average_latitude": 45.14293333333333,
 "average_longitude": -34.16216666666667
}
```

Challenges Faced During Task Implementation

1. **Error Handling for Invalid Data**:
    - Ensuring invalid data (e.g., missing fields, incorrect formats) was properly rejected with clear error messages.
2. **In-memory Storage Limitations**:
    - The in-memory data storage faced performance issues as the volume of data increased, potentially leading to memory constraints.
3. **Summary Calculation Logic**:
    - Accurately calculating the average latitude and longitude, especially with irregular or missing data, required careful validation and aggregation.
4. **System Performance Under Load**:
    - Handling high-frequency data uploads and concurrent requests effectively was challenging, requiring performance optimization to maintain smooth operation.