Weather Station Simulator

Objective

- Develop a networked system simulating multiple weather stations that sends real-time weather data to a centralized server using Python socket programming.
    - Each weather station client (Docker container) may send one or more metrics (e.g., temperature, humidity, wind speed) in its data payload.
    - The server, deployed in a Docker container, collects and stores the data persistently and provides a minimal web interface to display the data.
    - Also inlcude consumer client that retrieves weather data via sockets.

Project Description

You will build a client-server application where:

- **Weather station clients** simulate weather stations, generating and sending variable weather data (including one or more metrics like temperature, humidity, or wind speed) to the server using TCP sockets.
- The **server** uses Python sockets to receive data from multiple clients, stores it persistently (e.g., in a CSV file).
- A minimal web interface to display the data in a simple format.
- A **consumer client** (optional extension) connects to the server via sockets to request and receive weather data, such as the latest data or data for a specific station.
- All elements should be containerized using Docker to simulate a production-like environment.

Learning Outcomes

- **Socket Programming**: Master TCP socket communication, handle multiple clients (weather stations and consumers) concurrently, implement JSON data serialization with variable metrics, and support request-response interactions.
- **Networking**: Understand TCP, IP addressing, ports, and reliable data transfer.
- **Docker**: Deploy a networked application in a container, managing ports and persistent storage.
- **IoT Concepts**: Simulate IoT data flows, focusing on real-time data collection and retrieval with flexible data structures.
- **Web Development (Secondary)**: Create a minimal web interface to visualize collected data.

Core Requirements

1. **Server (Python Sockets)**:
    - Implement a TCP socket server to accept connections from multiple weather station clients.
    - Receive JSON-formatted weather data containing a `station_id`, `timestamp`, and one or more metrics (e.g., `temperature`, `humidity`, `wind_speed`). The server must handle variable metric sets

(e.g., one client may send only temperature, another may send temperature and humidity).
    - Handle multiple clients concurrently using threading or asyncio, ensuring robust connection management.
    - Store received data persistently in a CSV file or lightweight database (e.g., SQLite).
    - Provide a minimal web interface (e.g., using Flask) to display a table of collected weather data, dynamically showing available metrics (e.g., station ID, timestamp, and any of temperature, humidity, wind speed present in the data).
2. **Weather Station Client (Python Sockets)**:
    - Simulate a weather station by generating realistic weather data for one or more metrics (e.g., temperature between -10°C and 40°C, humidity 0–100%, wind speed 0–50 km/h).
    - Send JSON-formatted data to the server periodically (e.g., every 5 seconds) over a TCP socket, including `station_id`, `timestamp`, and a variable set of metrics (e.g., `{"station_id": "station1", "timestamp": "2025-10-08T14:00:00", "temperature": 22.5}` or `{"station_id": "station2", "timestamp": "2025-10-08T14:00:00", "temperature": 18.0, "humidity": 65}`).
    - Allow configuration of `station_id`, data interval, and selected metrics (e.g., which metrics to send) via command-line arguments or a config file.
    - Ensure reliable socket communication with proper connection handling and error reporting.
3. **Docker**:
    - Create a Dockerfile for the server, CLIENT, WEB-UI, including dependencies for socket programming and the minimal web interface.
    - Expose ports for socket communication (e.g., 12345) and the web interface (e.g., 5000).
    - Use a Docker volume to persist stored data (e.g., CSV or SQLite file).
    - Optionally, provide a Docker Compose file to orchestrate the server and test clients.
4. **Features**:
    - Server logs incoming data with timestamps, station IDs, and metrics for debugging and verification.
    - Weather station clients send realistic, periodic weather data with variable metrics over TCP sockets.
    - Server validates incoming data (e.g., reject invalid JSON, ensure `station_id` and `timestamp` are present, validate metric ranges if provided).
    - Web interface displays a simple table of the latest weather data from all stations, dynamically adjusting to show available metrics.
    - Graceful handling of client connect/disconnect and server shutdown.

Deliverables

- **Source Code**:
    - Python code for all componetns.
    - Dockerfile for all components.
    - Optional: Docker Compose file for orchestration.
- **Documentation**:
    - README with clear instructions to build and run the server in Docker, connect weather station clients, access the web interface, and (if implemented) use the consumer client.
    - Brief report (1–2 pages) explaining the socket programming design, handling of variable metrics, challenges (e.g., concurrency, data validation), and how to test the system.

- **Testing**:
  - Demonstrate the system with at least three weather station clients sending data concurrently, each with different combinations of metrics (e.g., one sending temperature, another sending temperature and humidity, another sending all metrics).
  - Verify socket communication reliability (e.g., handling variable metrics, client disconnects, data validation).
  - Show the web interface displaying the collected data in a browser, reflecting variable metrics.
  - Demonstrate the consumer client retrieving data from the server.

Stretch Goals (Optional Extensions)

- Define a simple request-response protocol (e.g., send a JSON request like `{"request": "latest"}`, `{"request": "station", "station_id": "station1"}`, or `{"request": "metric", "metric": "temperature"}` and receive JSON-formatted data in response).
- Add data aggregation on the server (e.g., average temperature per station over time, considering available metrics).
- Simulate network issues (e.g., packet loss, delays) to test socket robustness.
- Enhance the web interface with filters (e.g., show data for a specific station or metric) or basic graphs (e.g., using Chart.js).
- Add a REST API endpoint for programmatic data access.
- Implement basic authentication for client-server communication.

Notes

- Prioritize socket programming: ensure the server handles multiple clients (weather stations and consumers) reliably, validates variable metric data, and manages concurrent connections.
- The web interface should be minimal (e.g., a single HTML table), serving only to verify data collection and dynamically displaying available metrics.
- Test with multiple weather station clients sending different metric combinations and (if implemented) at least one consumer client to demonstrate concurrent socket handling.
- Focus on robust networking, variable metric handling, and error handling before adding optional features.
