Title: EU Flight Project – Internship Assignment

Name: Shubham Shashikant More

Contact: 8010782154

Email: moreshubham1230@gmail.com

Submission Date: 24-03-2025

Additional Comments:

- This project helped me understand real-time data fetching, API integration, and database management using PostgreSQL.
- I implemented data cleaning and validation techniques to ensure the accuracy of stored flight data.
- Error handling mechanisms were incorporated to manage API failures and database connection issues.
- The project provided valuable insights into handling large datasets efficiently and optimizing queries for performance.

1. Database Design

Database Schema

The system uses a relational database (PostgreSQL) to store structured flight data. It includes the following tables:

- Airport: Stores airport details (IATA, ICAO codes, location).
- Airlines: Holds airline names and unique codes.
- FlightStatus: Defines flight statuses such as "On Time," “active”, "Delayed," "scheduled" and "Cancelled"
- Flights: Stores flight numbers, schedules, departure/arrival airports, airline associations, and flight statuses.

Relationships

- Flights reference the Airport table (departure & arrival airports).
- Flights reference the Airlines table (operating airline).
- Flights reference the FlightStatus table (current flight status).

Data Accuracy & Scalability

- **Data Validation:** Ensuring correct formats (IATA codes, timestamps) before insertion.
- **Indexing & Optimization:** Indexing flight_number, departure_time, and status_id for quick retrieval.
- **Handling Duplicates:** Using ON CONFLICT DO UPDATE to prevent redundant data.

2.Data Collection Strategy

Collecting & Storing Airport Data

To provide users with reliable flight tracking, the system must maintain an updated database of airports, including key details such as IATA codes, ICAO codes, city, country, and geographic coordinates. The data collection process involves:

- Using AviationStack API: This API provides real-time information about airports, including their codes, locations, and operational status. The system fetches airport details dynamically when needed.
- Pre-loading Major European Airports: To reduce API dependency and optimize query performance, a predefined dataset of major European airports is stored in the database. This helps in minimizing API calls and ensures a faster response time when processing flight queries.
- Automated Database Updates: Periodic API calls check for new or updated airport data, ensuring that any changes (e.g., new airports, renamed airports, or closed airports) are reflected in the database.

Methods for Real-Time Flight Data

To track flights accurately, the system integrates multiple data sources and follows a structured approach for real-time data retrieval and storage:

- AviationStack API: This API provides real-time flight tracking, including departure and arrival times, airline details, and flight statuses. It is the primary source of flight data.
- Alternative Data Sources: Other APIs like FlightAware API, OpenSky Network, and ADS-B data scraping can supplement real-time data collection. These sources provide additional coverage, such as live air traffic and historical flight data.
- Scheduled API Calls: To maintain up-to-date flight records, the system implements scheduled API requests at fixed intervals (e.g., every 5 minutes). This ensures that the database is continuously updated with the latest flight statuses and reduces the risk of outdated information.

Handling Missing, Delayed, or Inconsistent Data

Ensuring data integrity is essential for providing reliable flight information. The system implements various strategies to handle incomplete, delayed, or inconsistent data:

- Null Checks & Default Values: When certain API responses are missing key details (e.g., a missing arrival time), the system assigns placeholder values or estimates based on historical data. This prevents database inconsistencies and improves query reliability.
- Flight Status Updates: Delayed or rescheduled flights are dynamically updated in the database as new data becomes available. The system constantly compares scheduled and actual departure times to determine delays.
- Error Logging & Debugging: Any API failures, missing data, or inconsistencies are logged for future analysis. This helps in identifying recurring issues and improving data retrieval strategies.

3. Flight Monitoring and Claim Identification

Tracking flight delays and ensuring data accuracy is crucial for an efficient flight monitoring system. This section outlines the proposed monitoring approach, the technical implementation, and strategies for handling large-scale flight data.

Proposed Flight Monitoring System

To ensure efficient tracking and claim verification for delayed flights, the system incorporates a Flask-based API that performs the following functions:

- Live Flight Tracking: The system continuously monitors real-time flight data from the AviationStack API (or alternative sources) and updates the database with the latest flight statuses.
- Delay Detection: The API fetches scheduled departure times and compares them with actual departure times. If the delay exceeds 2 hours, the system flags the flight as delayed.
- Storage & Analysis: Delayed flights are stored separately for further analysis, claim processing, and potential integration with airline compensation policies.

By automating flight monitoring, the system ensures real-time tracking and provides useful insights for both passengers and airline operators.

Technical Approach

The flight monitoring system relies on efficient real-time processing and data updates to keep track of ongoing flights. Key technical components include:

1. **Real-time Monitoring**
   - The system periodically fetches flight data using scheduled API calls (e.g., every 5 minutes).
   - It cross-references flight records in the database to detect changes in flight status, ensuring the latest data is available.
2. **Data Updates**
   - The database updates each flight's departure and arrival times, flight status, and delay duration when new data is retrieved.
   - If a flight’s status changes (e.g., from "On Time" to "Delayed" or "Cancelled"), the system records this change for tracking purposes.

3. **Alerts & Notifications (Future Scope)**
    - The system can be extended to send email/SMS notifications to passengers for significant delays.
    - Airlines and travel agencies can integrate the system to notify customers about potential disruptions in advance.

Efficient Storage & Large-Scale Data Handling

Handling large volumes of flight data efficiently is essential for maintaining system performance. The following optimizations help improve database speed and scalability:

- **Indexing Key Fields**
    - Indexing columns such as flight_number, status_id, and departure_time speeds up searches and retrieval operations.
    - This reduces query time when retrieving delayed flights or specific flight details.
- **Partitioning Data**
    - Flights are stored in partitions based on date or geographical region to enhance query performance.
    - This ensures that recent and frequently accessed flights are retrieved quickly, while older data does not slow down the system.
- **Archiving Old Flights**
    - Completed flights older than a certain period (e.g., 6 months) are moved to an archive table to reduce the size of frequently queried tables.
    - Archiving helps in maintaining system efficiency while preserving historical data for future reference.

4. Future API Development

To ensure long-term efficiency and adaptability, the flight tracking API must be designed with scalability, security, and high availability in mind. This section outlines strategies for expanding API capabilities, ensuring security, and optimizing performance for large-scale deployment.

Scalable API Design

As the project grows, the API should evolve to support new functionalities, handle increased traffic, and optimize data retrieval. Key aspects of a scalable API include:

1. Expanding API Features

- **Historical Flight Data:**
  - Currently, the API fetches real-time flight data, but historical records (past flights, delays, cancellations) can be stored for analysis.
  - This feature can assist airlines in performance evaluation and predictive analytics for delay patterns.
- **Passenger Tracking:**
  - Future versions of the API can allow passengers to track their flight status, receive updates, and manage their bookings via the API.

2. Optimizing Performance for Large Datasets

- **Pagination & Filtering:**
  - As the database grows, retrieving all flights at once can slow down performance.
  - Implementing pagination (returning data in small chunks) and filters (e.g., filtering by date, airline, or status) helps in handling large volumes of data efficiently.

3. Cloud Deployment for Scalability

- **Hosting on Heroku, AWS, or DigitalOcean:**
  - Deploying the API on a cloud platform allows automatic scaling to handle increased traffic.

- **Gunicorn as a WSGI Server:**
  - Instead of using Flask’s built-in development server, Gunicorn (Green Unicorn) can be used as a production-ready web server to handle multiple requests simultaneously.
  - This improves performance and allows efficient API execution under high loads.

Security, Availability, and Reliability

Ensuring data security, system availability, and API reliability is critical for a robust deployment. The following measures help protect user data and enhance API performance.

1. Secure Data Handling

- **Environment Variables (. env Files) for Sensitive Data:**
  - Instead of storing database credentials and API keys in the code, they are kept in .env files.
  - This prevents unauthorized access and accidental exposure of sensitive information.
- **SQL Injection Prevention:**
  - Using parameterized queries (cursor.execute("SELECT * FROM Flights WHERE flight_number = %s", (flight_number,))) instead of direct SQL queries prevents SQL injection attacks, ensuring secure data transactions.

2. Load Balancing & Caching for High Availability

- **Load Balancing:**
  - When deployed on cloud platforms, requests can be distributed across multiple servers to avoid overloading a single instance.
  - This ensures uninterrupted availability even during peak traffic periods.
- **Data Caching:**
  - Frequently accessed queries (e.g., "Get all flights for today") can be cached to reduce database load and speed up responses.
  - Tools like Redis or Memcached can be integrated for caching API responses.

3. Rate Limiting API Calls

- **Preventing Excessive Traffic:**
    - To prevent abuse (e.g., bots making thousands of API calls), Flask-Limiter can be used to restrict requests per user/IP within a specific time frame.
    - Example: Limiting each user to 100 API calls per minute to maintain API availability for all users.