RAP airlines
Project Documentation

Roman Volochai, Artur Rubish, Pavlo Protsenko

June 8, 2025

1 Project Overview and Goals

The primary topic of this project is the implementation of a functional, albeit partial, airline management system.

Our initial vision included a wide range of ambitious goals. However, during the development process, we recognized the significant complexity of many of these features. Consequently, we decided to scale back the initial scope and shift our focus away from numerous advanced concepts. Instead, we concentrated on building the core functionalities that are essential for a foundational airline system, ensuring a robust and well-implemented core.

2 Challenges and Solutions

During the development of this project, we encountered several key challenges:

- **Realistic Data Generation:** One of the most significant hurdles was generating realistic flight schedule data. Creating an optimal and conflict-free schedule is highly complex, especially with a limited fleet of aircraft. Our generation process aimed to account for various factors, including city popularity, population, airport capacity, and aircraft range. We successfully generated a coherent set of schedules.

- **Optimal Aircraft Assignment:** Following schedule generation, the next challenge was assigning specific aircraft to individual flights. We developed a custom algorithm to optimize aircraft utilization. A crucial part of this system is a database trigger that ensures each aircraft has a valid and continuous flight path, guaranteeing it has sufficient time to arrive at the next departure airport.

- **Incident Management:** We introduced an incident management system to simulate real-world disruptions. This allows for declaring various types of incidents, including catastrophic events. In such a scenario, the system is designed for an administrator to manually reassign the flights previously handled by the lost aircraft to other available planes.

- **Data History and Logging:** Maintaining a comprehensive history was deemed essential for key operational data. We implemented history tracking for aircraft statuses, flight statuses, and the last known location of each aircraft.

- **Password Security:** To ensure the security of user data, we implemented password hashing. Customer passwords are not stored in plaintext; instead, they are processed through a hashing algorithm before being stored in the database.
- **Maintaining Real-time Data Consistency:** A major difficulty was simulating the dynamic nature of airline operations. In a real-world airline, dispatchers constantly update flight information. Generating this data in a way that remains consistent and logical relative to the current date proved to be a complex task.
- **Efficient Data Loading:** To populate the database, all initial data for each table is loaded efficiently using the PostgreSQL `COPY` command.

3 Database Architecture and Logic

The database is designed using a normalized relational model in PostgreSQL. The structure is intended to be logical, extensible, and to accurately model the core entities and processes of an airline. The architecture can be broken down into several logical modules.

3.1 Geographical and Infrastructure Core

This is the foundational layer of the database, defining where the airline operates.

- **Country, City, Airport:** These tables form a simple hierarchy. Each `Airport` belongs to a `City`, and each `City` belongs to a `Country`. This provides the geographical context for all routes and flights.

3.2 Aircraft Fleet Management

This module describes the physical assets of the airline—the aircraft.

- **Aircraft_Model:** Acts as a template, defining the specifications for a type of aircraft (e.g., Boeing 737), including passenger capacity and range.
- **Aircraft:** Represents a specific, physical aircraft instance in the fleet. Each aircraft is an instance of a specific `Aircraft_Model`.
- **Aircraft_Status_History & Aircraft_Location_History:** These are logging tables. They are crucial for operational awareness and auditing. They track the status of an aircraft over time (e.g., 'InService', 'UnderMaintenance') and its last known airport location. This is vital for scheduling and incident investigation.

3.3 Flight Operations

This is the most complex and central part of the system, modeling the concept of a flight.

- **Route:** A static, abstract concept defining a journey between two airports (`airport_from` and `airport_to`).
- **Flight_Schedule:** Defines a recurring flight pattern on a specific `Route`. For example, "Flight BA202 from London to New York, departing every Monday at 10:00 AM". It has a day of the week and a time.

- **Flight_Instance:** This is a concrete, single occurrence of a flight from a `Flight_Schedule` on a specific date. For example, "Flight BA202 on October 28, 2024". This is the entity that gets an aircraft assigned, has actual departure/arrival times, and can have tickets sold for it.
- **Flight_Status_History:** Similar to the aircraft history, this table tracks the status of each `Flight_Instance` (e.g., 'Scheduled', 'Delayed', 'Landed'), providing a real-time log of flight operations.

3.4 User and Customer Management

This module handles user authentication and customer information.

- **Customer:** Stores Personally Identifiable Information (PII) like name, birth date, and email. This table holds information about the person.
- **App_User:** Manages application access. It contains the `username`, hashed password, and role ('USER' or 'ADMIN'). It is linked to a `Customer` record. This separation of concerns (personal data vs. authentication data) is a good security practice.

3.5 Booking and Ticketing System

This module models the commercial aspect of the airline.

- **Fare & Seat_Layout:** These are configuration tables. `Fare` defines fare classes ('Economy', 'Business') and their price multipliers. `Seat_Layout` defines the seating arrangement for each `Aircraft_Model`.
- **Ticket:** Represents a single seat sold on a specific `Flight_Instance`. Its price is calculated based on the route's base price and the fare class multiplier. Each ticket is unique for a given flight and seat.
- **Booking:** This table connects a `Customer` to a purchased `Ticket`, creating a confirmed reservation. It tracks the status of the booking ('Pending', 'Confirmed', etc.).

3.6 Ancillary Systems: Incidents & Maintenance

These tables handle operational disruptions.

- **Incident:** Records any event that disrupts a `Flight_Instance`, such as technical issues, weather delays, or catastrophic failures.
- **Maintenance:** Logs maintenance activities for a specific `Aircraft`, including start/end dates and descriptions. This is crucial for tracking an aircraft's availability.

4 Installation and Usage

The application is designed to be intuitive.

4.1 User Roles

The system features two user roles: `USER` and `ADMIN`.

- **ADMIN:** Has full privileges to view and edit all data within the airline database directly through the application's interface.
- **USER:** Can register, log in, search for flights, purchase tickets, and manage their bookings. A new user can be registered via the application's registration panel.

4.2 Admin Credentials

Admin registration is not available through the public interface. A pre-configured admin account is provided for testing and management:

```
Login:    romanvolochai
Password: Romanenk14_
```

4.3 GUI Interaction

A key user interface feature to note is that many table cells are interactive. **Double-clicking on a cell** will often open a new menu or editing dialog for that specific record.

4.4 Prerequisites

- Java Development Kit (JDK)
- JavaFX SDK
- Maven
- PostgreSQL Database

4.5 Configuration

1. **Database Connection:** The application connects to a local PostgreSQL database named `airlinedb`. You must configure your database user credentials in the file: `/src/main/java/com/airlines_sql/utils/DatabaseUtil.java`
2. **File Paths for Data Loading:** In the same `DatabaseUtil.java` file, you must update the local file paths to the `.sql` files (`create.sql`, `clear.sql`) used for database setup.
3. **Database Population (Optional):** The `MainApp` class contains a line of code that re-initializes and populates the database every time the application starts. This is useful for development but can be slow. You can comment out this line to prevent the database from being reloaded on every launch.

4.6 Running the Application

Once the prerequisites are installed and the configuration is complete, run the application from the root project folder with the Maven command:

```
mvn javafx:run
```

4.7 Note on Booking Generation

The generation and loading of booking data are handled by a separate Python script: `generate_and_load_bookings.py`. This is because ticket generation is an event-driven process that occurs when a new `flight_instance` is created.