Technical Documentation and System Architecture for Airflow: A Flight Reservation Management Application

Leonardo Acevedo Monroy
Daniel Alonso Chavarro Chipatecua
Andrés Felipe Gómez Durán
Oscar Daniel Aguas Quiroz
Careers of System Engineering and Computer Science
Universidad Nacional de Colombia
Bogota, Colombia

***Abstract*—This paper presents the comprehensive technical documentation of Airflow, a flight reservation management system. The system is designed to efficiently manage flight reservations, providing a robust platform that enables users to search, book, and manage their air travel. The application encompasses comprehensive flight management, from aircraft and seat administration to reservation and flight status control, providinga complete experience for both end users and system administrators. This document details the database design methodology, entity-relationship modeling, system architecture, and implementation patterns used in the development process.**

***Index Terms*—flight reservation system, database design, entity-relationship model, software architecture, data access objects, ontology-based development**

I. Introduction

Modern air travel management requires sophisticated software systems capable of handling complex operations, including flight scheduling, seat management, user authentication, and reservation processing. This paper presents the technical documentation for Airflow, a comprehensive flight reservation management application designed to address these requirements through a well-structured, scalable architecture.

The system integrates multiple components including user management, aircraft inventory, flight scheduling, and reservation processing. The application serves both end users seeking to book flights and administrators requiring comprehensive system management capabilities.

II. Project Repository

The code of the entire project is in this Github repository: https://github.com/Daniel-Chavarro/AirFlow

III. System Architecture

A. General Description

The flight reservation application architecture is based on a layered structure that clearly separates the different responsibilities of the system. The main architectural layers are as follows:

- **Presentation Layer**: User interface and interaction
- **Model Layer**: Data representation in the application
- **Persistence Layer**: Data access and management in the database
- **Service Layer**: Business logic
- **Utility Layer**: Tools for different layers

This layered architecture promotes the separation of concerns, maintainability, and scalability while ensuring clear boundaries between different system components.

IV. Data Persistence - Database Design

A. Development Methodology

The development of the entity-relationship model is based on ontologies as a conceptual foundation. This methodology enables the creation of a more coherent and semantically rich database design, ensuring that the relationships between entities accurately reflect the logic of the business domain.

The development process followed these stages:

1) **Entity Declaration**: Identification and definition of the system's main entities
2) **Attribute Definition**: Specification of each entity's properties with their respective data types
3) **Relationship Establishment**: Development of connections between entities through cardinality analysis
4) **Model Validation**: Verification of the integrity and consistency of the design

B. System Entities

The system comprises the following main entities:

1) Users: Represents both administrators and regular customers of the system. This entity centralizes user management with different privilege levels.

Attributes:

- `id_PK`: Unique user identifier (int, primary key)
- `name`: User's first name (varchar(40), not null)
- `last_name`: User's last name (varchar(40), not null)
- `email`: Email address (varchar(200), not null)
- `password`: Encrypted password (varchar(20), not null)

- isSuperUser: Administrative privileges indicator (boolean, not null)
- created_at: Account creation timestamp (datetime, not null)

2) Airplanes: Manages information about aircraft available in the fleet.

Attributes:

- id_PK: Unique aircraft identifier (int, primary key)
- airline: Airline owner (varchar(20), not null)
- model: Aircraft model (varchar(50), not null)
- code: Aircraft identification code (varchar(10), not null)
- capacity: Total passenger capacity (int, not null)
- year: Manufacturing year (year)

3) Cities: Represents cities that are part of the flight network, used to implement route graph logic.

Attributes:

- id_PK: Unique city identifier (int, primary key)
- name: City name (varchar(100), not null)
- country: Country where the city is located (varchar(100), not null)
- code: Airport code of the city (varchar(10), not null)

4) Flights: Manages information about scheduled flights in the system.

Attributes:

- id_PK: Unique flight identifier (int, primary key)
- airplane_FK: Reference to assigned aircraft (int, foreign key)
- status_FK: Current flight status (int, foreign key)
- origin_city_FK: Origin city (int, foreign key)
- destination_city_FK: Destination city (int, foreign key)
- code: Flight identification code (varchar(10), not null)
- departure_time: Departure date and time (timestamp, not null)
- scheduled_arrival_time): Scheduled arrival date and time (timestamp NOT NULL)
- arrival_time: Arrival date and time (timestamp, null)
- price_base: Base flight price (decimal(10,2), not null)

5) Reservations: Manages reservations made by users.

Attributes:

- id_PK: Unique reservation identifier (int, primary key)
- user_FK: User who made the reservation (int, foreign key)
- status_FK: Current reservation status (int, foreign key)
- flight_FK: Reserved flight (int, foreign key)
- reserved_at: Reservation timestamp (timestamp, not null)

6) Seats: Manages the available seats on the aircraft.

Attributes:

- id_PK: Unique seat identifier (int, primary key)
- airplane_FK: Aircraft to which it belongs (int, foreign key)
- reservation_FK: Associated reservation, if exists (int, foreign key, nullable)
- seat_number: Seat number (varchar(10), not null)
- seat_class: Seat class (enum: ECONOMY, BUSINESS, FIRST)
- is_window: Indicates if it is a window seat (boolean)

7) Flight_Status: Catalog of possible states for flights.

Attributes:

- id_PK: Unique status identifier (int, primary key)
- name: Status name (varchar(15), not null)
- description: Detailed status description (varchar(100))

8) Reservations_Status: Catalog of possible states for reservations.

Attributes:

- id_PK: Unique status identifier (int, primary key)
- name: Status name (varchar(15), not null)
- description: Detailed status description (varchar(100))

C. Entity Relationships

The system establishes the following relationships between the identified entities:

1) User-Reservation Relationships: **Cardinality**: 1:M (One to Many)

- A user can make zero or multiple flight reservations
- Each reservation is assigned to only one user
- This relationship ensures reservation traceability

2) Aircraft-Reservation Relationships: **Cardinality**: 1:M (One to Many)

- An aircraft can be assigned to multiple reservations
- Each reservation can have only one aircraft assigned
- A reservation may initially have no aircraft assigned

3) Aircraft-Seat Relationships: **Cardinality**: 1:M (One to Many)

- An aircraft is composed of multiple seats
- A seat belongs exclusively to a specific aircraft
- This relationship defines the physical configuration of each aircraft

4) Aircraft-Flight Relationships: **Cardinality**: 1:M (One to Many)

- An aircraft may or may not be assigned to an active flight
- A flight must have an aircraft assigned
- This relationship manages flight operations

5) Reservation-Seat Relationships: **Cardinality**: 1:M (One to Many)

- A reservation can include one or multiple seats
- A seat may or may not be assigned to a reservation
- This relationship enables group reservations

6) Reservation-Flight Relationships: **Cardinality**: M:1 (Many to One)

- A reservation is assigned to a specific flight
- A flight can have multiple registered reservations
- This relationship links reservations with scheduled flights

7) Status Relationships: **Cardinality**: 1:M (One to Many)

- A reservation has a unique status at any given time
- A status can be assigned to multiple reservations
- A flight has a specific status
- A flight status can apply to multiple flights

8) City-Flight Relationships: **Cardinality**: 1:M (One to Many)

- A flight must depart from an origin city
- A flight must land at a destination city
- A city can be the origin of multiple flights
- A city can be the destination of multiple flights
- This structure enables implementation of routing algorithms and graphs

D. Design Considerations

1) Referential Integrity: The design implements referential integrity constraints through foreign keys, ensuring data consistency and preventing references to non-existent entities.

2) Normalization: The model is in third normal form (3NF), eliminating redundancies and transitive dependencies, which optimizes storage and reduces update anomalies.

3) Scalability: The structure allows horizontal growth through implementation of appropriate indexes and partitioning of large tables such as `reservations` and `flights`.

4) Flexibility: The use of status tables (`flight_status` and `reservations_status`) provides flexibility to add new states without modifying the structure of main tables.

V. Data Access Object (DAO) Pattern

The system implements the DAO design pattern to abstract and encapsulate database access, clearly separating business logic from persistence logic. The implementation follows these principles:

A. DAOMethods Interface

A generic interface `DAOMethods<T>` is defined that establishes standard methods for CRUD operations:

- `ArrayList<T> getAll()`: Retrieves all objects of type T
- `T getById(int id)`: Retrieves a specific object by its identifier
- `void create(T object)`: Creates a new record in the database
- `void update(int id, T toUpdate)`: Updates an existing record
- `void delete(int id)`: Deletes a specific record

B. Specific Implementations

Each domain entity has its own DAO implementation that extends the generic interface:

- `FlightDAO`: Manages CRUD operations for flights
- `ReservationDAO`: Manages CRUD operations for reservations
- `UsersDAO`: Manages CRUD operations for users
- `AirplaneDAO`: Manages CRUD operations for aircraft
- `CityDAO`: Manages CRUD operations for cities
- `SeatDAO`: Manages CRUD operations for seats

This structure ensures uniformity in data access and facilitates code maintainability by centralizing persistence logic.

C. SQL Query Methods

DAOs implement various SQL queries to satisfy system requirements. Below are some relevant examples:

1) FlightDAO Queries: The FlightDAO class implements complex queries including flight searches with status information and multi-criteria filtering based on origin, destination, and date parameters.

2) ReservationDAO Queries: The ReservationDAO handles user-specific reservation retrieval and seat availability verification through join operations between multiple tables.

D. Database Connection Management

Database connection management is centralized in the `ConnectionDB` class, following the Singleton pattern to optimize resources. The connection configuration includes Unicode support, SSL configuration, and comprehensive exception handling.

1) Transaction Handling: The system implements transaction control for critical operations such as flight reservations, ensuring data integrity even in complex operations affecting multiple tables through commit/rollback mechanisms.

VI. User Interface Design

A. Design Philosophy

The graphical user interface has been designed following principles of minimalism and simplicity, with the aim of providing an intuitive and modern user experience. The key elements that have guided the design are:

- **Simplicity**: Elimination of unnecessary elements to focus on essential functionality
- **Consistency**: Coherent use of visual elements and interaction patterns throughout the application
- **Accessibility**: Clear and intuitive interface that facilitates navigation for all users
- **Modern aesthetics**: Implementation of a contemporary design using the FlatLaf library

B. Technologies Used

- **Java Swing**: Base framework for building graphical components
- **FlatLaf**: External library that provides a modern and flat look to Swing components
- **Responsive Design**: Implementation of layouts that adapt to different screen sizes

C. Interface Structure

The user interface is organized following a hierarchical component architecture:

1) **MainFrame**: Main window that contains all other components
2) **Horizontal Menu**: Top bar with logo, title and navigation options

3) **Content Panel**: Central area with card system to display different sections
4) **Specific Panels**: Components dedicated to specific functionalities

D. Main Components

1) Flight Search Panel: Interface that allows users to search for available flights according to criteria such as origin, destination, and dates. Includes:

- Origin and destination city selectors
- Travel date selector
- Additional filters
- Search button

2) Flight Details Panel: Displays detailed information about a selected flight, including:

- Airline information and flight number
- Departure and arrival times
- Airport details
- Seat selection

3) Seat Selection Panel (BookSeatsPanel): Interactive interface that allows users to visualize and select specific seats on the chosen aircraft.

Visualization features:

- Graphical representation of the aircraft
- Visual differentiation by service class
- Real-time availability indicators
- Legend of symbols and colors

Selection functionalities:

- Multiple seat selection
- Availability validation
- Automatic price calculation
- Seat recommendations

4) Confirmation Panel: Allows the user to review and confirm their booking details before finalizing:

- Flight information summary
- Passenger details
- Payment information
- Confirmation or cancellation buttons

5) Dialog Components: The application implements specific dialog components to handle user authentication:

- **Login Dialog**: Provides a secure interface for users to authenticate and access their accounts
- **Register Dialog**: Allows new users to create accounts by entering their personal information

These dialog components follow the same minimalist design principles as the main interface, maintaining visual consistency throughout the application while serving their specialized functions.

E. Visual Design Elements

- **Color Palette**: Predominance of light tones with subtle accents to highlight important elements
- **Typography**: Use of sans-serif fonts to improve readability
- **Iconography**: Implementation of minimalist icons to represent common actions
- **Spacing**: Design with sufficient white space to reduce visual overload

F. User Interaction and Flow

The navigation flow in the application follows a linear and predictable pattern.

1) Search for available flights
2) Selection and display of flight details
3) Selection of seats and additional options
4) Review and confirmation of the booking
5) Display of confirmation and booking details

This sequential design guides the user through the booking process intuitively, reducing the learning curve and minimizing possible errors during the process.

G. Interface Screenshots

Below are screenshots of the application's main interfaces demonstrating the implemented design principles and user flows.

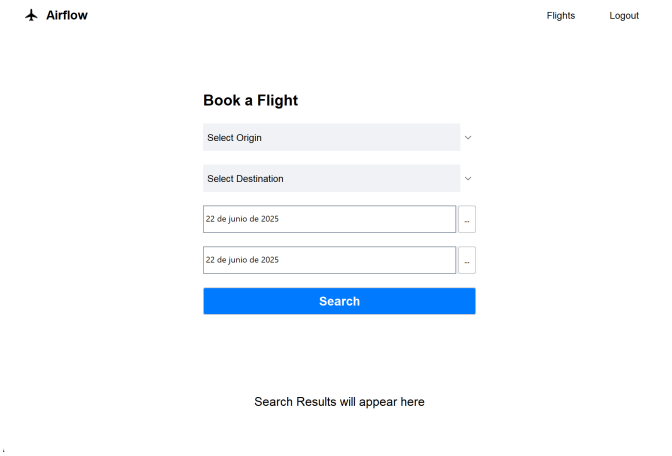

Fig. 1. Main application interface showing the flight search panel

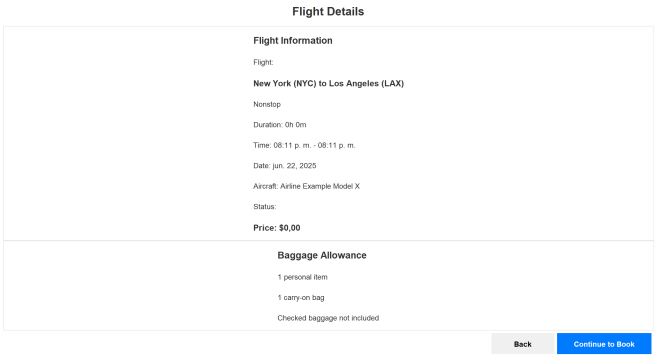

Fig. 2. Flight details panel showing selected flight information

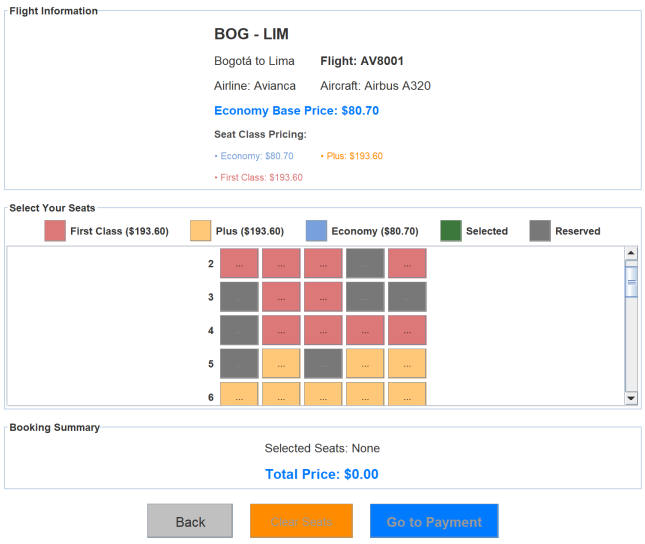

Fig. 3. Book Seats panel showing available seats

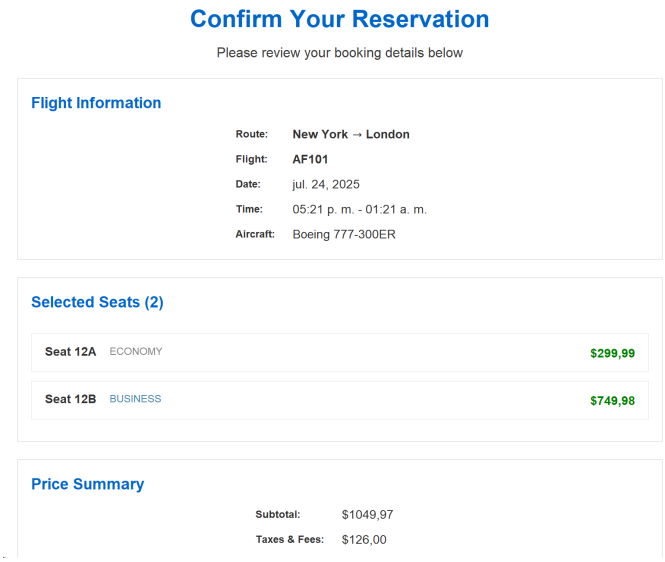

Fig. 4. Confirmation panel showing details of reservation

These screenshots demonstrate how the design principles of simplicity, consistency, and modern aesthetics have been applied throughout the application, resulting in a clean and intuitive user interface.

VII. GUI Implementation with Bridge and Command Patterns

To enhance the modularity and maintainability of the user interface, the Bridge and Command design patterns were implemented. This section details how these patterns were integrated into the GUI architecture.

A. Bridge Pattern for Decoupling

The Bridge pattern was used to decouple the user interface (the "View") from its controlling logic (the "Controller"). This separation allows the UI and the business logic to evolve independently.

- **Abstraction (`View` Interface)**: An interface named `View` was created to define all the operations that the user interface must support. This includes methods for displaying panels, retrieving data from forms, and setting data for display. This interface acts as the "bridge" between the UI implementation and the controller.
- **Concrete Implementor (`MainFrame`)**: The `MainFrame` class implements the `View` interface, providing the concrete implementation of the UI. It manages a `CardLayout` to switch between different panels like `SearchFlightPanel`, `DetailsFlightPanel`, etc.

This structure allows the controller to interact with the UI through the `View` interface, without being coupled to the specific implementation details of `MainFrame` or jego sub-components.

B. Command Pattern for UI Actions

The Command pattern was used to encapsulate all the information needed to perform an action or trigger an event. In the context of the GUI, this means that each user action (e.g., clicking a button) is treated as a command.

- **Commands**: A set of string constants (e.g., `SEARCH_FLIGHT_CMD`, `BOOK_SEAT_CMD`) are defined in the `View` interface. These constants represent the different commands that can be triggered by the user.
- **Invokers**: UI components like `JButton` act as invokers. Each interactive component is assigned a specific action command using the `setActionCommand()` method.
- **Receiver and Controller**: A single `ActionListener` in the (yet to be implemented) controller will act as the receiver for all these commands. It will inspect the action command of the event and delegate the request to the appropriate handler method, effectively executing the command.

By using the Command pattern, the UI components are completely decoupled from the logic that is executed. The buttons don't know what happens when they are clicked; they only know to send a specific command to the listener.

C. Dynamic Flight Display

The flight search results are now displayed dynamically as a series of cards.

- **`FlightCardPanel`**: A new component, `FlightCardPanel`, was created to represent a single flight in the search results. This panel displays key information like the route, time, duration, and price, along with a "View Details" button.
- **Dynamic Population**: The `SearchFlightPanel` now contains a method, `displayFlights()`, which

takes a list of flights and dynamically creates a `FlightCardPanel` for each one. These cards are added to a results panel, which is displayed to the user.

This approach makes the search results more interactive and visually appealing, and it further demonstrates the flexibility of the implemented architecture. The main frame delegates the responsibility of displaying the flights to the search panel, which in turn uses the specialized card panel for the rendering of each flight.

VIII. Service Layer

The service layer contains the business logic of the application. It acts as an intermediary between the presentation layer and the persistence layer. Each service class encapsulates a specific set of functionalities.

- **Controller**: The main controller of the application, implemented as a Singleton. It initializes the GUI and handles user interactions by listening to events from the view.
- **AirplaneService**: Manages business logic related to airplanes, such as retrieving airplane information.
- **CityService**: Manages business logic related to cities, such as retrieving city information.
- **FlightService**: Manages business logic related to flights, such as searching for flights based on different criteria.
- **ReservationService**: Manages business logic related to reservations, such as creating and managing user reservations.
- **SeatService**: Manages business logic related to seats, such as retrieving seat information for a specific flight.
- **SuggestionService**: Provides suggestions for flights based on user preferences or other criteria.

IX. Utility Layer

The utility layer provides a set of reusable tools and functionalities that are used across different layers of the application.

- **ConnectionDB**: A utility class for establishing and managing the connection to the MySQL database.
- **PasswordUtils**: Provides methods for hashing and verifying passwords using the jBCrypt library.
- **DataGenerator**: A utility for populating the database with sample data for testing and development purposes.
- **Runner**: The main entry point of the application, responsible for initializing and starting the system.

X. Project Dependencies

The project is built and managed using Apache Maven. The core dependencies are defined in the `pom.xml` file and include:

- **MySQL Connector/J**: Provides JDBC connectivity for the MySQL database.
- **FlatLaf**: A modern look and feel library for Java Swing applications.
- **jBCrypt**: A library for hashing passwords.
- **LGoodDatePicker**: A date picker component for Swing.
- **JUnit 5**: The testing framework for unit tests.
- **Mockito**: A mocking framework for creating test doubles.

XI. Docker Configuration

The application uses Docker to containerize the MySQL database, ensuring a consistent development and deployment environment. The configuration is defined in the `docker-compose.yml` file:

- **Image**: Uses the official `mysql:8.0.33` image.
- **Container Name**: The container is named `mysql-airflow`.
- **Environment Variables**: Sets the root password, database name, and character set.
- **Ports**: Maps the container's port 3306 to the host's port 3306.
- **Volumes**: Mounts the `db/Airflow.sql` script to initialize the database schema on startup.
- **Healthcheck**: Includes a healthcheck to ensure the database is running before accepting connections.

XII. Testing

The project includes a suite of unit tests to ensure the correctness of the application's components. The tests are written using JUnit 5 and Mockito. The following components are tested:

- **DAO Layer**: Tests for `AirplaneDAO`, `CityDAO`, `FlightDAO`, `ReservationDAO`, `SeatDAO`, and `UsersDAO`.
- **Service Layer**: Tests for `AirplaneService`, `CityService`, `FlightService`, `ReservationService` and `SuggestionService`.
- **Utils**: Tests for the `ConnectionDB` utility.

Appendix

The following is the complete SQL script used to create the database schema.

```
CREATE DATABASE IF NOT EXISTS airflow CHARACTE

USE airflow;

CREATE TABLE IF NOT EXISTS `users` (
  `id_PK` int PRIMARY KEY NOT NULL AUTO_INCREM
  `name` varchar(40) NOT NULL,
  `last_name` varchar(40) NOT NULL,
  `email` varchar(200) NOT NULL UNIQUE,
  `password` varchar(73) NOT NULL,
  `isSuperUser` boolean NOT NULL,
  `created_at` timestamp NOT NULL
);

CREATE TABLE IF NOT EXISTS `airplanes` (
  `id_PK` int PRIMARY KEY NOT NULL AUTO_INCREM
  `airline` varchar(20) NOT NULL,
  `model` varchar(50) NOT NULL,
  `code` varchar(10) NOT NULL,
  `capacity` int NOT NULL,
  `year` year
);
```

```
                                                    FOREIGN KEY (`reservation_FK`) REFERENCES `r
CREATE TABLE IF NOT EXISTS `cities` (       );
  `id_PK` int PRIMARY KEY NOT NULL AUTO_INCREMENT,
  `name` varchar(100) NOT NULL,
  `country` varchar(100) NOT NULL,
  `code` varchar(10) NOT NULL
);

CREATE TABLE IF NOT EXISTS `flight_status` (
  `id_PK` int PRIMARY KEY NOT NULL,
  `name` varchar(15) NOT NULL,
  `description` varchar(100)
);

CREATE TABLE IF NOT EXISTS `reservations_status` (
  `id_PK` int PRIMARY KEY NOT NULL,
  `name` varchar(15) NOT NULL,
  `description` varchar(100)
);

CREATE TABLE IF NOT EXISTS `flights` (
  `id_PK` int PRIMARY KEY NOT NULL AUTO_INCREMENT,
  `airplane_FK` int NOT NULL,
  `status_FK` int NOT NULL,
  `origin_city_FK` int NOT NULL,
  `destination_city_FK` int NOT NULL,
  `code` VARCHAR(10) NOT NULL,
  `departure_time` timestamp NOT NULL,
  `scheduled_arrival_time` timestamp NOT NULL,
  `arrival_time` timestamp NULL,
  `price_base` DECIMAL(10,2) NOT NULL,
  FOREIGN KEY (`airplane_FK`) REFERENCES `airplanes` (`id_PK`),
  FOREIGN KEY (`status_FK`) REFERENCES `flight_status` (`id_PK`),
  FOREIGN KEY (`origin_city_FK`) REFERENCES `cities` (`id_PK`),
  FOREIGN KEY (`destination_city_FK`) REFERENCES `cities` (`id_PK`)
);

CREATE TABLE IF NOT EXISTS `reservations` (
  `id_PK` int PRIMARY KEY NOT NULL AUTO_INCREMENT,
  `user_FK` int NOT NULL,
  `status_FK` int NOT NULL,
  `flight_FK` int NOT NULL,
  `reserved_at` timestamp NOT NULL,
  FOREIGN KEY (`user_FK`) REFERENCES `users` (`id_PK`),
  FOREIGN KEY (`status_FK`) REFERENCES `reservations_status` (`id_PK`),
  FOREIGN KEY (`flight_FK`) REFERENCES `flights` (`id_PK`)
);

CREATE TABLE IF NOT EXISTS `seats` (
  `id_PK` int PRIMARY KEY NOT NULL AUTO_INCREMENT,
  `airplane_FK` int NOT NULL,
  `reservation_FK` int,
  `seat_number` VARCHAR(10) NOT NULL,
  `seat_class` ENUM('ECONOMY','BUSINESS','FIRST') NOT NULL,
  `is_window` BOOLEAN,
  FOREIGN KEY (`airplane_FK`) REFERENCES `airplanes` (`id_PK`),
```

REFERENCES

[1] Date, C.J., "An Introduction to Database Systems," 8th ed. Boston: Addison-Wesley, 2004.

[2] Fowler, M., "Patterns of Enterprise Application Architecture," Boston: Addison-Wesley, 2002.

[3] Silberschatz, A., Galvin, P.B., and Gagne, G., "Database System Concepts," 7th ed. New York: McGraw-Hill, 2019.