**Flight Booking System – Low Level Design**

Case Study Overview:-

The **Flight Booking System** is a modern, scalable web application built using a **Microservices Architecture**. It allows users to seamlessly search for flights, make bookings, check-in, and perform other related tasks. This system is designed to be efficient, secure, and easily maintainable by breaking down the entire application into smaller, loosely coupled services, each responsible for a specific business function. These services communicate through RESTful APIs and are built using **ASP.NET Core**.

The system also integrates a **Frontend UI** built with **Angular**, which provides a rich, responsive, and user-friendly interface for the end-user. The Angular frontend interacts with the backend microservices through HTTP requests and handles various user actions such as searching for flights, viewing flight details, booking flights, and performing check-ins.

The architecture is designed for high availability, scalability, and fault tolerance, with each microservice maintaining its own independent database. This allows for better data management and ensures that updates in one service do not disrupt the functioning of others.

**Key Components of the System**

1. **Microservices Architecture**:
   * **Search Microservice**: Responsible for fetching available flights based on user input such as departure city, destination city, and travel date.
   * **Booking Microservice**: Handles the booking of flights, including seat allocation, fare calculations, and booking confirmation.
   * **Fare Microservice**: Manages the pricing details of flights, including base price and convenience fees.
   * **Check-In Microservice**: Handles the check-in process, updating booking statuses, and seat allocation for passengers.
2. **Frontend (Angular UI)**: The **Frontend UI** is developed using **Angular**, a popular JavaScript framework known for building dynamic, single-page web applications. The frontend interacts with the backend using **RESTful APIs** to fetch and display data to the user.
   * **Responsive Design**: The UI adapts to various screen sizes, ensuring a smooth experience across devices (desktop, tablet, and mobile).
   * **Dynamic Content**: Angular's two-way data binding and component-based architecture allow real-time updates without the need to reload the entire page.
   * **User Interaction**: Guests and users can search for flights, view available options, make bookings, and check-in directly from the Angular-based user interface.
   * **Security**: The frontend uses **JWT (JSON Web Tokens)** to secure the communication between the client and server for authentication and authorization.
3. **RESTful API Communication**: The system exposes a set of RESTful APIs that the **Angular Frontend** interacts with:
   * **Search API**: Fetches available flights based on user inputs.
   * **Booking API**: Allows the creation of a booking, including fare retrieval and seat allocation.
   * **Check-In API**: Handles the check-in process and updates the booking status.
   * **Fare API**: Provides details about the flight fares, including base price and convenience fees.
4. **Database**: Each microservice interacts with its own independent **SQL Server** database to store data related to flights, bookings, passengers, and fares. This separation ensures that each service is decoupled, making the system easier to maintain and scale.
   * **Flight Database**: Stores details about flights, such as flight number, cities, departure date, and available seats.
   * **Booking Database**: Holds information about bookings, including passenger details, selected flights, and booking status.
   * **Fare Database**: Maintains pricing information for flights, including base prices and additional fees.
   * **City Database**: Stores information about cities, including city codes, names, and airport charges.

**1. Use Case Diagram**

The Use Case Diagram illustrates the interactions between users (guests/admins) and the system. It highlights the main operations performed by each actor and shows the relationships between different use cases.

**Key Use Cases:**

* **Search Flights**: Allows a guest to search for available flights based on the departure and destination cities, and travel date.
* **Book Flight**: Allows a guest to book a flight by providing passenger details.
* **Check-In**: Allows the user to check-in for a flight.

A diagram of a flight system

AI-generated content may be incorrect.

* **Manage Flights (Admin)**: The administrator can add, remove, or update flight details.
* **Manage City Information (Admin)**: The admin can add, remove, or update cities served by the airline.
* **Manage Fare Details (Admin)**: The admin can adjust fare details, including base price and convenience fees.

A diagram of flight booking system

AI-generated content may be incorrect.

**2. Sequence Diagram**

The Sequence Diagram depicts the sequence of interactions between the system components for a specific use case: Searching for Flights.

**Use Case: Searching for Flights**

1. The guest enters the search criteria (departure city, destination city, and travel date).
2. The system sends a request to the Search Microservice to fetch available flights.
3. The Search Microservice queries the database for flights that match the given criteria.
4. The system returns the list of available flights to the guest.
5. The guest selects a flight and proceeds with booking.

**Use Case: Booking a Flight**

1. The guest enters passenger details and selects a flight to book.
2. The system sends a request to the Booking Microservice to create a booking.
3. The Booking Microservice queries the Fare Service to retrieve the applicable fare for the selected flight.
4. The Fare Service returns the fare details to the Booking Microservice.
5. The Booking Microservice updates the available seats for the flight in the Flight Database.
6. The Booking Microservice creates a booking record in the Booking Database.
7. The system returns a booking confirmation to the guest with a reference number.

**Use Case: Check-In**

1. The guest initiates the check-in process by providing the booking reference.
2. The system sends a request to the Check-In Microservice to update the booking status.
3. The Check-In Microservice queries the Booking Database to find the booking by reference.
4. The Check-In Microservice updates the check-in status for the booking in the Booking Database.
5. The Check-In Microservice also updates the booking status to "checked-in."
6. The system returns a check-in confirmation to the guest.

A screenshot of a web page

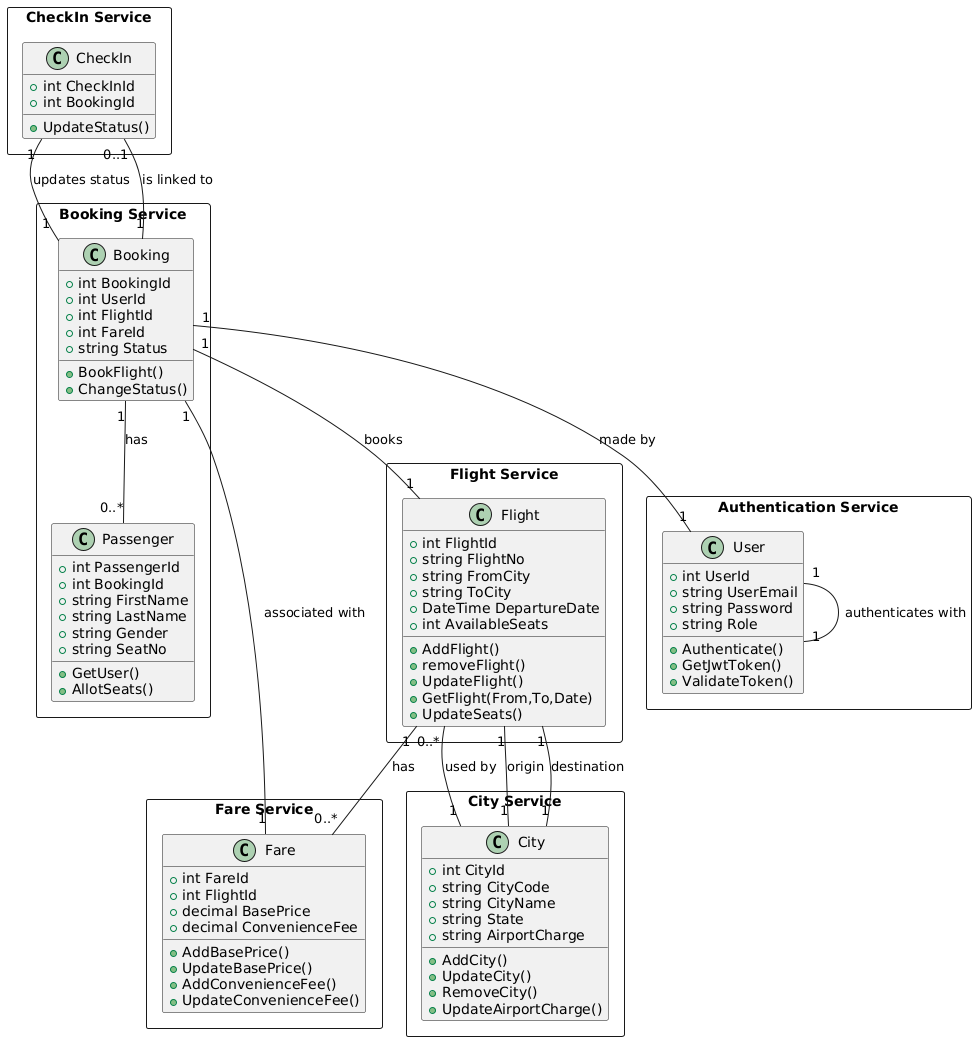
AI-generated content may be incorrect.

**3. Class Diagram**

The Class Diagram provides a detailed view of the system’s classes and their relationships. It includes all the necessary classes, their attributes, methods, and how they interact with each other.

Key Classes:

* Flight: Represents flight information including flight number, departure city, destination, available seats, etc.
* Fare: Contains the pricing information related to flights.
* Booking: Manages the booking details including flight ID, passenger details, and booking status.
* Passenger: Represents a passenger associated with a booking.
* User: Handles user authentication, including guest and admin.
* City: Represents cities where flights are available.
* CheckIn: Manages the check-in process for passengers.



**4. Entity-Relationship Diagram (ERD)**

The ER Diagram represents the logical structure of the database. It shows entities, their attributes, and the relationships between them.

Entities and Attributes:

* Flight: FlightId, FlightNo, FromCity, ToCity, DepartureDate, AvailableSeats
* Fare: FareId, FlightId, BasePrice, ConvenienceFee
* City: CityId, CityCode, CityName, State, AirportCharge
* Booking: BookingId, UserId, FlightId, FareId, Status
* Passenger: PassengerId, BookingId, FirstName, LastName, Gender, SeatNo
* User: UserId, UserEmail, Password, Role
* CheckIn: CheckInId, BookingId, Status

A diagram of a flight

AI-generated content may be incorrect.