1. **Problem Statement: Designing Airline Booking Management System**

We are tasked with building a microservice-based Airline Booking Management System. The system should handle operations like flight search, seat booking, authentication, and reminders. The architecture must be scalable, reliable, and secure, ensuring an efficient booking experience for users.

---

2. **System Requirements and Solutions**

1. **Flight Search Service**
    - *Solution*: A REST API implemented in Node.js using Express.js. The service allows users to search for available flights based on parameters like date, origin, destination, and flight type (economy, business).
2. **Seat Booking Service**
    - *Solution*: A separate microservice responsible for reserving seats. This service interacts with the flight service and ensures that once a seat is booked, it is not available to other users. It uses RabbitMQ for message-based communication with other services to handle booking requests efficiently.
3. **Authentication Service**
    - *Solution*: This microservice handles user authentication and authorization. It uses JWT (JSON Web Token) for session management and securely manages user data and credentials. The service is critical for ensuring only authorized users can access booking or flight information.
4. **Reminder Service**
    - *Solution*: This service is responsible for sending users booking reminders via email or SMS. It uses cron jobs to schedule tasks such as sending reminders 24 hours before the flight departure. RabbitMQ ensures reminders are delivered reliably by queuing messages.
5. **Rate Limiting**
    - *Solution*: To prevent API abuse, rate limiting is applied at the gateway level, ensuring that users are restricted to a certain number of requests within a specified time window. This protects the system from overload and DDoS attacks.

---

3. **Key Classes, Functions, and Relationships**

1. **FlightSearchService**
    - Provides flight search functionality. Takes parameters such as source, destination, and dates to return available flights.
2. **SeatBookingService**
    - Manages seat reservation for flights. This service communicates with the flight service and ensures booking status is updated in real-time.

3. **AuthService**
    - Manages user authentication and authorization. It generates JWTs for session management, allowing secure access to protected routes.
4. **ReminderService**
    - Sends out timely reminders to users about their upcoming flights using RabbitMQ and cron jobs for scheduled delivery.

---

4. **Data Structures Used**

- **MySQL Database (via Sequelize ORM)**:
    - Tables are used to store flight, booking, and user data. Relationships between flights, bookings, and users are managed efficiently using Sequelize.
- **RabbitMQ Queues**:
    - Facilitates communication between services. For example, when a booking is made, a message is sent to the seat booking service, ensuring atomic seat reservations.
- **JWT (JSON Web Token)**:
    - Tokens are used to securely authenticate users and manage sessions across services.

---

5. **High-Level Design**

- **Flight Search**:
    - Exposes an API to search for flights based on the user's input.
- **Seat Booking**:
    - Ensures seats are booked and reserved only if they are available, updating the flight's availability accordingly.
- **Authentication**:
    - Provides secure login, signup, and session management using JWT tokens.
- **Reminder**:
    - Schedules reminders for users using a cron job to notify them about upcoming flights.

---

6. **Clarifications to Ask the Interviewer**

1. **Should we prioritize any specific flight classes (e.g., economy/business)?**
    - *Solution*: If prioritization is required, we can integrate logic to prefer business-class bookings or offer special rates during certain times.

2. **How should we handle failed bookings or payment issues?**
   - *Solution*: A fallback mechanism to retry bookings or notify the user in case of failed payments should be integrated.

---

7. Open-Ended Questions from the Interviewer

1. **How would you handle high traffic during peak seasons?**
   - *Solution*: Introduce horizontal scaling using AWS EKS (Elastic Kubernetes Service), auto-scaling to handle increased traffic. Use distributed databases like Amazon RDS for MySQL for scalability.
2. **How would you ensure fault tolerance across services?**
   - *Solution*: Implement circuit breaker patterns and retry mechanisms for each service. RabbitMQ can also ensure that messages are queued and processed even if a service is temporarily down.

---

8. C++ Code Implementation

```
#include <iostream>
#include <map>
#include <string>
#include <vector>
#include <queue>

using namespace std;

// Auth Service: manages user authentication
class AuthService {
private:
    map<string, string> userCredentials; // Stores user credentials for
authentication
public:
    bool login(string username, string password) {
        return userCredentials[username] == password;
    }

    void registerUser(string username, string password) {
        userCredentials[username] = password;
    }
};

// Flight Search Service: manages available flights
class FlightSearchService {
private:
    map<string, vector<string>> flights; // Maps source to destination and
available flights
public:
    void addFlight(string source, string destination, string flight) {
        flights[source].push_back(flight + " to " + destination);
    }

    vector<string> searchFlights(string source, string destination) {
        return flights[source];
    }
};

// Seat Booking Service: handles seat reservations
class SeatBookingService {
private:
    map<string, int> availableSeats; // Maps flight to the number of available
seats
public:
    void addFlightSeats(string flight, int seats) {
        availableSeats[flight] = seats;
    }

    bool bookSeat(string flight) {
        if (availableSeats[flight] > 0) {
            availableSeats[flight]--;
            return true;
        }
        return false;
    }
};

// Reminder Service: sends flight reminders
class ReminderService {
private:
    queue<string> reminderQueue; // Stores flight reminders
public:
    void addReminder(string flight) {
        reminderQueue.push("Reminder: Flight " + flight + " is soon.");
    }

    void sendReminders() {
        while (!reminderQueue.empty()) {
            cout << reminderQueue.front() << endl;
            reminderQueue.pop();
        }
    }
};

// Main: demonstrate system interaction
int main() {
    // Auth service
    AuthService authService;
    authService.registerUser("user1", "password1");

    // Flight search service
    FlightSearchService flightSearch;
    flightSearch.addFlight("NYC", "LAX", "Flight A");
    flightSearch.addFlight("NYC", "LAX", "Flight B");
    // Seat booking service
    SeatBookingService seatBooking;
    seatBooking.addFlightSeats("Flight A", 10);

    // Reminder service
    ReminderService reminderService;

    // Simulating booking process
    if (authService.login("user1", "password1")) {
        vector<string> availableFlights = flightSearch.searchFlights("NYC",
"LAX");
        if (!availableFlights.empty() &&
seatBooking.bookSeat(availableFlights[0])) {
            cout << "Booking confirmed for: " << availableFlights[0] << endl;
            reminderService.addReminder(availableFlights[0]);
        } else {
            cout << "No seats available!" << endl;
        }
    }

    // Send reminders
    reminderService.sendReminders();

    return 0;
}
```

Explanation of Code

- `AuthService`: Manages user authentication by storing and validating credentials.
- `FlightSearchService`: Stores flight information and allows users to search for flights based on the source and destination.
- `SeatBookingService`: Handles seat reservations for flights.
- `ReminderService`: Queues and sends reminders for upcoming flights.

This microservice-based system allows for future scalability, ensuring it can handle peak traffic and complex booking requirements in an airline management scenario.