Case Study: Flight Reservation System (Monolithic Application)

1. Project Overview

You are tasked with developing a **Flight Reservation System** for a small airline.
The system should allow:

- **Flight Management**
  - Add new flights
  - View all available flights
  - View details of a specific flight
  - Update flight details (origin, destination, time, seats available)
  - Delete a flight
- **Reservation Management**
  - Make a reservation for a specific flight
  - View all reservations
  - View reservations for a specific flight
  - Cancel a reservation (and restore seats to the flight)

This is a **monolithic Spring Boot application** — all functionality will be in a single codebase.

2. Technology Stack

- **Spring Boot** (Web + Data JPA)
- **H2 Database** (in-memory for development)
- **Springdoc OpenAPI / Swagger** (API documentation)
- **Maven** (dependency management)
- **Java 17+**
- **JUnit & Mockito** (optional, for unit testing)

3. Entities

The system will have **two main entities**:

1. Flight

- **id** — Unique identifier (auto-generated)
- **flightNumber** — Unique code for the flight (e.g., AI101)
- **origin** — Departure city/airport
- **destination** — Arrival city/airport
- **departureTime** — Date & time of departure
- **seatsAvailable** — Number of available seats

2. Reservation

- **id** — Unique identifier (auto-generated)
- **passengerName** — Name of the passenger
- **passengerEmail** — Contact email of the passenger
- **seatsBooked** — Number of seats booked
- **reservedAt** — Date & time when reservation was made
- **flight** — Reference to the Flight entity (`Many reservations → One flight`)

4. Relationships

- **One Flight can have many Reservations**
  This means:
  - In the database, `Reservation` will have a `flight_id` foreign key.
  - In JPA, `Reservation` will use `@ManyToOne` to `Flight`.

5. API Requirements

Learners should create **REST APIs** with the following endpoints:

Flight API

- `POST /api/flights` → Add a new flight
- `GET /api/flights` → Get all flights
- `GET /api/flights/{id}` → Get flight by ID
- `PUT /api/flights/{id}` → Update a flight

- `DELETE /api/flights/{id}` → Delete a flight

Reservation API

- `POST /api/reservations` → Make a reservation
  - Reduce the available seats in the flight
  - Reject reservation if seats are not enough
- `GET /api/reservations` → Get all reservations
- `GET /api/reservations/flight/{flightId}` → Get reservations for a specific flight
- `DELETE /api/reservations/{id}` → Cancel a reservation
  - Add back seats to the flight

6. Business Rules

- When making a reservation:
  - Check if the flight exists.
  - Ensure seats requested ≤ seats available.
  - Reduce seat count if successful.
- When canceling a reservation:
  - Add the booked seats back to the flight.
- A flight cannot have a negative number of seats.
- Flight numbers should be unique.

7. Suggested Implementation Steps

1. **Setup Project**
   - Create a Spring Boot project with required dependencies.
   - Configure `application.properties` for H2 database.
2. **Create Entities**
   - Define `Flight` and `Reservation` with appropriate JPA annotations.
   - Set up relationships using `@ManyToOne`.

3. **Create Repositories**
    - Extend `JpaRepository` for both entities.
4. **Write Services**
    - Business logic for managing flights and reservations.
    - Handle seat availability logic in the reservation service.
5. **Write Controllers**
    - Map endpoints to service methods.
    - Use `ResponseEntity` for proper HTTP status codes.
6. **Exception Handling**
    - Create custom exceptions (e.g., FlightNotFoundException, NotEnoughSeatsException).
    - Use `@ControllerAdvice` for global exception handling.
7. **Swagger Integration**
    - Use Springdoc OpenAPI to generate API documentation.
    - Test APIs from Swagger UI.

2. Microservices Overview

We will have **three microservices**, each running independently, with its own database and API.

1. Restaurant Service

Responsibilities:

- Manage restaurant details.
- Manage menu items for each restaurant.

Core Features:

- Add, view, update, delete restaurants.
- Add, view, update, delete menu items for a restaurant.
- List all menu items for a restaurant.

Entity Examples:

- **Restaurant**
  - id (PK)
  - name
  - location
  - contactNumber
- **MenuItem**
  - id (PK)
  - restaurantId (FK to Restaurant)
  - name
  - description
  - price

API Examples:

- `POST /restaurants`
- `GET /restaurants`
- `GET /restaurants/{id}`
- `POST /restaurants/{id}/menu-items`
- `GET /restaurants/{id}/menu-items`

2. Order Service

Responsibilities:

- Handle customer orders.
- Track order status (PLACED, PREPARING, DELIVERED, CANCELED).

Core Features:

- Place an order for one or more menu items (fetch menu from Restaurant Service).
- View all orders for a customer.
- Update order status.

Entity Examples:

- **Order**

    - id (PK)
    - customerName
    - customerAddress
    - totalAmount
    - status
- **OrderItem**
    - id (PK)
    - orderId (FK to Order)
    - menuItemId
    - quantity
    - price

API Examples:

- `POST /orders` (calls Restaurant Service to verify menu item availability & price)
- `GET /orders/{id}`
- `GET /customers/{customerName}/orders`
- `PUT /orders/{id}/status`

3. Delivery Service

Responsibilities:

- Assign delivery agents to orders.
- Track delivery status.

Core Features:

- Assign delivery person when order status becomes "PREPARING".
- Update delivery status.
- Track delivery by order ID.

Entity Examples:

- **Delivery**
    - id (PK)

- orderId
- deliveryPersonName
- deliveryStatus (ASSIGNED, OUT_FOR_DELIVERY, DELIVERED)

API Examples:

- `POST /deliveries` (triggered when Order Service updates order to PREPARING)
- `GET /deliveries/{orderId}`
- `PUT /deliveries/{id}/status`

3. Database Design

Each microservice has its own **independent database**:

- Restaurant DB → Tables: `restaurants`, `menu_items`
- Order DB → Tables: `orders`, `order_items`
- Delivery DB → Tables: `deliveries`