LLD

https://www.educative.io/courses/grokking-the-low-level-design-interview-using-ood-principles/getting-ready-the-airline-management-system

Airline Management System
Software used to efficiently manage all actively of airline system
There are several component present in Airline management system 1> Flight Reservation 2> Payment Handling 3> Flight Scheduling 4> Dynamic Pricing 5> Flight Cancellation 6> Staff and crew management
Design Approach Bottom-Up Approach

Requirements

We will focus on the following set of requirements while designing the Airline Management System:

R1: Customers should be able to search for flights for a given date and source/destination airport.

R2: Customers should be able to reserve a ticket for any scheduled flight. Customers can also build a multi-flight itinerary.

R3: Users of the system can check flight schedules, their departure time, available seats, arrival time, and other flight details.

R4: Customers can make reservations for multiple passengers under one itinerary.

R5: Only the admin of the system can add new aircrafts, flights, and flight schedules. Admin can cancel any pre-scheduled flight (all stakeholders will be notified).

R6: Customers can cancel their reservation and itinerary.

R7: The system should be able to handle the assignment of pilots and crew members to flights.

R8: The system should be able to handle payments for reservations.

R9: The system should be able to send notifications to customers whenever a reservation is made/modified or there is an update for their flights or flight cancelled.
R10: Dynamic flight price rate based on number of day left and seat filled
R11: Food Booking in Flight .
R12: Seat booking in Flight based on window seat or Leg Space or Business Seat and cost

	Seat: Type SeatNumber Class

Actors/High Level entity

[Customer] <>--- [Itinerary] <>--- [Reservation] <>--- [Passenger]
| |
| +--> [Payment]
|
+--> [FlightSegment] --> [Flight] --> [Aircraft]
|
+--> [CrewAssignment] --> [Pilot/CrewMember]

[Admin]→ Flight Manager, AircraftManager
[Flight]-> Seat → SeatType
[Flight]→Schedule
[Flight]→Price strategy
[Reservation]→ [Notification]
[FoodBooking]->[MenuItem]

Class Breakdown

Customer and Admin

Class User {
Name
Email
Phone
List<Booking> booking
}
	Flight , FlightSegment and Schedule
	--- 	
	Class Flight{
	Name
	Id
	Aircraft From
	Aircraft To
	List<Schedule> schedule
	List<Seat> seats;
	List<CrewAssigment> crewList;
	PricingStrategy pricingStrategy;
	List<MenuItem> menu;
	
	}
	
	class Flight {
	String id;
	String name;
	Location from;
	Location to;
	Aircraft aircraft;
	List<Schedule> schedule; // recurrence rules
	List<MenuItem> menu;
	}

Class Admin extend User
{
void addFlight(Flight flight)
void cancelFlight(Flight flight)
void addAircraft(Aircraft aircraft)
}
	Class FlightSegment
	{
	segmentId;
	Flight
	LocalDate departure;
	LocalDate arrival;
	List<Seat> availableSeat;
	FlightStatus status
	}
	
	class FlightSegment {
	String id;
	LocalDate flightDate;
	Flight flight; // reference to the recurring flight
	List<Seat> seats; // seat availability
	List<CrewAssignment> crewList; // flight-specific crew
	PricingStrategy pricingStrategy;
	boolean isCancelled;
	FlightStatus status;
	LocalDateTime actualDepartureTime;
	}
	
	class FlightSchedule {
	String scheduleId;
	LocalDateTime departure;
	LocalDateTime arrival;
	boolean isCancelled;
	}

	Class Aircraft
	{
	String aircraftId;
	String model;
	Int totalSeat;
	List<Seat> seatConfiguration;
	}
	
	Class Airport{
	Name
	Id

	City country }

	Enum SeatType
	{
	ECONOMY,
	PREMIUM,
	BUSINESS,
	FIRST_CLASS,
	WINDOW,
	LEG_SPACE
	}
	
	Class Seat{
	seatNumber:int
	seatType:SeatType
	basePrice:double
	isBooked: Boolean
	}

After creating class Flight
I think I have completed core part and this give me a rough idea

class Itinerary{
String itineraryId;
List<FlightSegment> flightsegment;
List<Passanger> passangerList;
Reservation servervation;
}

Class Reservation{
rerservationId
PaymentStatus
List<Notification> notification;
ReservationStatus status;
}

Food Menu

Class FoodMenu{
String itemId;
String name;
Double price;
Boolean veg;
}

Class Passenger{
String name;
String Id;
Seat Seat;
List<MenuItem>
}

What is pending now ?

class CrewMember
{
String id;
String name;
CrewRole role;
}
 	
Enum CrewRole{
PILOT,CO_PILOT, FLIGHT_ATTENDENT
}

```
Class CrewAssignment{
CrewMemberId
FlightSegment segment;
}
```

```
Payment{
String paymentId;
Double amount;
PaymentStatus status;
PaymentMethod method;
LocalDateTime time;
}
```

Dynamic Pricing Strategy

```
Interface PricingStrategy
{
Double calculatePricing(seat seat, LocationDateTime flightDate,int seatLeft)
}
```

```
DynamicPricingStrategy implement PricingStrategy{

public double calculatePrice(Seat seat, LocalDateTime flightDate, int seatsLeft) {
      long daysToDeparture = ChronoUnit.DAYS.between(LocalDateTime.now(), flightDate);
      double multiplier = (daysToDeparture < 5 ? 1.5 : 1.0) + (seatsLeft < 10 ? 0.5 : 0.0);
      return seat.basePrice * multiplier;
   }
}
```

We will use notification service to send the notification to the server.

What is a FlightSegment in Airline LLD?

🔷 Definition:

A `FlightSegment` represents a **leg of a journey** — one continuous flight from a source airport to a destination airport that is part of a larger **itinerary**.

Design Motivation

1> Modularity
2> Flexibile for multi leg itenary
3> Different Airline or aircraft per flight

So Why Track availableSeats in FlightSegment?
Because **seat availability is dynamic and date-specific**, and FlightSegment is **bound to a date** and **itinerary**.

🔄 **Real-world Example**
Flight AI101:

- Aircraft: Boeing 777
- Total seats: 300
- Runs daily: Delhi → London

FlightSegment for 1st August (AI101 on Aug 1):

- Only 30 seats left unbooked

FlightSegment for 2nd August (AI101 on Aug 2):

- 200 seats available

→ You need to track **available seats per segment**, **per date**, **per flight instance**, **not in the Flight class**, because Flight is like a reusable **blueprint**.

- FlightSchedule: Airline’s recurring plan
- FlightSegment: Customer’s specific selected flight instance

---

Yes — you're on the **right track again.** Your current design defines the Flight class as if it represents **both the recurring template and the actual flight instance**, which can lead to coupling between static and dynamic concerns.

Let’s Split Flight into Two Core Concepts

1. Flight or FlightSchedule — Recurring or Planned Flight

```
class Flight {
   String id;
   String name;
   Location from;
   Location to;
   Aircraft aircraft;
   List<Schedule> schedule; // E.g., MON, WED, FRI at 10:30
   List<MenuItem> menu; // Usually same for recurring flights
}
```

This is **static data** — planned, recurring info (unchanging unless the plan changes).

---

2. FlightSegment — Actual Flight on a Specific Date

```
class FlightSegment {
   String id;
   LocalDate flightDate;
   Flight flight; // Reference to the recurring schedule
   List<Seat> seats; // Seat availability for this date
   List<CrewAssignment> crewList; // Crew assigned for this journey
   PricingStrategy pricingStrategy; // Dynamic pricing per segment
   FlightStatus status; // ON_TIME, CANCELLED, DELAYED, etc.
   boolean isCancelled;
   LocalDateTime actualDepartureTime;
}
```

This is **dynamic data** — real-time state of a specific flight on a specific day.

Why This Change Makes Sense

Moved from Flight → FlightSegment	Why
List<CrewAssignment>	Crew changes daily — shifts, rosters are dynamic
PricingStrategy	Pricing varies by date, time, demand
List<Seat>	Seat availability is per flight segment
FlightStatus / isCancelled	Status applies only to specific flight date

---

Final Class Structure

Flight – the Recurring Plan

```
class Flight {
   String id;
   String name;
   Location from;
   Location to;
   Aircraft aircraft;
   List<Schedule> schedule; // recurrence rules
   List<MenuItem> menu;
}
```

FlightSegment – a Specific Instance

```
class FlightSegment {
   String id;
   LocalDate flightDate;
   Flight flight; // reference to the recurring flight
   List<Seat> seats; // seat availability
   List<CrewAssignment> crewList; // flight-specific crew
   PricingStrategy pricingStrategy;
   boolean isCancelled;
   FlightStatus status;
   LocalDateTime actualDepartureTime;
}
```

Sample Use Case

Search Flight:

User wants flights from **DEL to BOM on 2025-08-01**:

1. Query FlightSegment for date = 2025-08-01, origin = DEL, dest = BOM
2. Use data from FlightSegment (seats, pricing, cancellation) for booking UI
3. Link to parent Flight for static info (name, aircraft, menu)

✅ **Summary**

- ✅ **Move dynamic data** like crew, pricing, status, seats to FlightSegment
- ✅ **Keep static, reusable data** in Flight
- 🔄 Think of FlightSegment as a **daily instance** of the Flight
- 📦 Easier to manage changes — e.g., cancel a specific flight, apply surge pricing, assign crew

---

Let me know if you'd like me to:

- Refactor Java classes completely
- Add repository/DB mapping (JPA)
- Build REST API or service methods (searchFlights(date, from, to))

Query Optimization Example

```sql
SELECT fs.*
FROM flight_segment fs
JOIN flight f ON fs.flight_id = f.id
WHERE fs.origin = 'DEL'
  AND fs.destination = 'BOM'
  AND fs.flight_date = '2025-08-01'
  AND fs.is_cancelled = false
  AND fs.available_seats > 0
ORDER BY fs.flight_time ASC
LIMIT 20;
```

> * Index on: `flight_date`, `origin`, `destination`, `is_cancelled`, `available_seats`
> * Partition by `flight_date`

Concurrency Locking

How about when multiple user try to book the same airline at same time where seat are limited to 1 or before booking

Great point. Handling **concurrent seat booking** for the same flight (especially when only **1 seat is available**) is critical to prevent:

- **Double booking**
- **Overbooking**
- **Race conditions**

Let's break down the **problem** and then look at **robust solutions**.

DB Lock	Solution Options (Strong Consistency) 1. Database-Level Locking (Recommended) Use SELECT ... FOR UPDATE Lock the FlightSegment row before checking availability and updating: BEGIN; SELECT available_seats

	FROM flight_segment WHERE id = 12345 FOR UPDATE; -- Check if seat available -- If available_seats > 0 UPDATE flight_segment SET available_seats = available_seats - 1 WHERE id = 12345; COMMIT; 📌 This guarantees only one transaction can modify the row at a time. Pros: Simple and effective Works well with RDBMS like PostgreSQL, MySQL Cons: Locks a row (slight delay for others) Still serial, not massively scalable
2>Optimistic Locking (Version Column)	2. Optimistic Locking (Version Column) Add a version column to FlightSegment, and perform atomic update only if version matches: -- Pseudocode UPDATE flight_segment SET available_seats = available_seats - 1, version = version + 1 WHERE id = 12345 AND available_seats > 0 AND version = CURRENT_VERSION; If rows affected = 0, then someone else booked it already. 🧠 Use @Version annotation in JPA/Hibernate to support this automatically. Pros: Non-blocking Scales better under concurrency Cons: Retry logic required at app layer

4> Distributed Lock (Redis Locking)	Distributed Lock (Redis Locking)
	
	Use a distributed lock (e.g., Redisson, or Redis + Lua):
	
	RLock lock = redissonClient.getLock("flight-segment:12345");
	if (lock.tryLock(3, 10, TimeUnit.SECONDS)) {
	try {
	// Check available_seats and proceed
	} finally {
	lock.unlock();
	}
	}
	
	Lock key: flight-segment:{id}
	
	Pros:
	
	Works across pods/instances

	Prevents concurrent access cleanly Cons: Redis failure can break locking Extra infra dependency

When multile request come at same time

- Use **DB lock** when:
    - All services use **same DB primary node**.
    - You need **strict consistency** and low latency.
    - DB lock the other request waiting for same row lock
- Use **Redis lock** when:
    - Your app is **horizontally scaled**, and DB lock can't coordinate across services.
    - You handle **retries**, **timeouts**, and **crash recovery** gracefully.
    - Does not wait other request and sent null if not able to acquire lock

Final Thought

Use **DB locks** for **simple, local, monolithic systems** or **small services** tightly coupled to the same DB.

Use **Redis or distributed locks** when:

- You have **horizontal scaling,**
- Or **microservices deployed across clusters, region US/UK/INDIA each have its own DB**
- Or want **fine-grained control** on retry/backoff/timeouts.