Smart Airport Ride Pooling Backend - LLD & HLD Design

1. High Level Architecture (HLD)

The system follows a scalable microservice-inspired backend architecture designed to support high concurrency, real-time booking, and distributed processing.

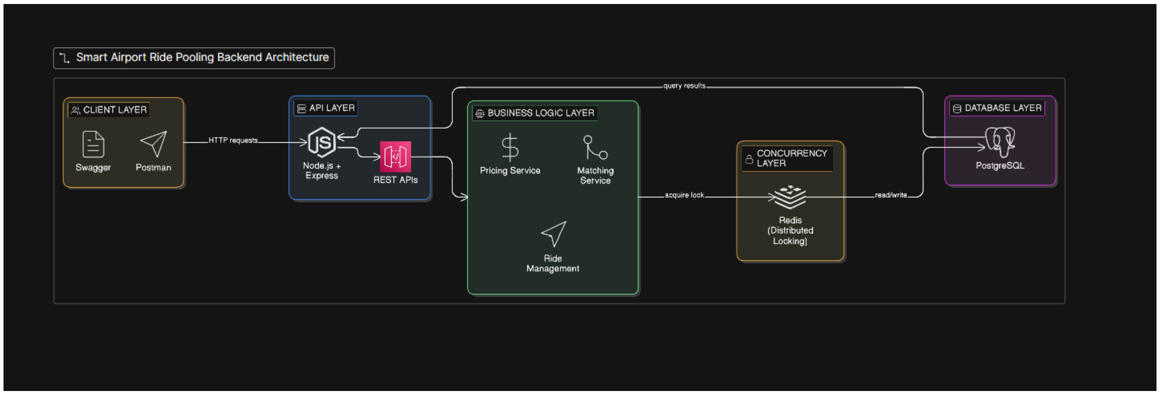
Main Components:
- Client Layer (Postman / Swagger / App)
- API Layer (Node.js + Express)
- Business Logic Layer (Matching, Pricing, Ride Management)
- Redis (Concurrency Locking)
- PostgreSQL (Persistent Storage)

Architecture Flow:

User → API Server → Matching Service → Redis Lock → Database Update → Response

This layered architecture ensures:
- Scalability
- Low latency
- Fault tolerance
- Clear separation of responsibilities

Conceptual Architecture Diagram:



2. Low Level Design (LLD)

The system is divided into modular components for maintainability and clean architecture.

Core Modules:

- Controllers
  - Handle HTTP requests and responses
  - Call services

- Services
  - Matching Service (assigns cab)
  - Pricing Service (calculates price)

- Models
  - Cab Model
  - RideRequest Model

- Config
  - Database connection
  - Redis connection

Class Diagram (Conceptual):

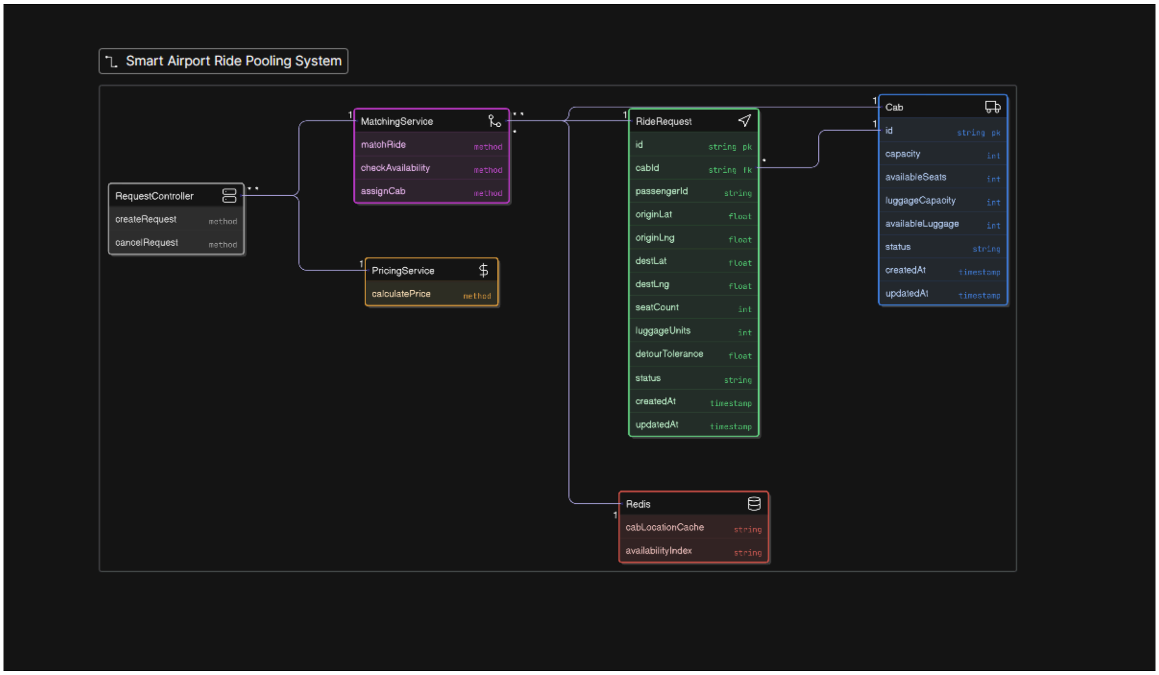
RideRequest
- id
- passengerId
- originLat
- originLng
- destLat
- destLng
- seatCount
- luggageUnits
- detourTolerance
- status
- cabId

Cab
- id
- capacity
- availableSeats
- luggageCapacity
- availableLuggage

- status

Interaction Flow:

Controller → MatchingService → Cab Model → Redis Lock
Controller → PricingService → Return Price
Controller → Save RideRequest



3. Design Patterns Used

1. MVC Pattern:
Separates Controllers, Models, and Business Logic.

2. Service Layer Pattern:
Matching and Pricing logic isolated from controllers.

3. Singleton Pattern:
Database and Redis connections created once and reused.

4. Repository Style (via Sequelize):
Models abstract direct DB queries.

Benefits:

- Clean code structure
- Easy debugging
- High maintainability
- Easy to scale features