Low-Level Design Document for Smart Parking Lot System

Objective

Design a low-level architecture for a backend system of a smart parking lot that handles vehicle entry and exit management, parking space allocation, and fee calculation.

Problem Statement

Design a system to manage a multi-floor parking lot in an urban area with numerous parking spots, automatically assigning parking spots based on vehicle size and availability, tracking the time each vehicle spends in the parking lot, and calculating parking fees upon exit.

Functional Requirements

1. **Parking Spot Allocation**: Automatically assign an available parking spot to a vehicle when it enters, based on the vehicle's size (e.g., motorcycle, car, bus).
2. **Check-In and Check-Out**: Record the entry and exit times of vehicles.
3. **Parking Fee Calculation**: Calculate fees based on the duration of stay and vehicle type.
4. **Real-Time Availability Update**: Update the availability of parking spots in real-time as vehicles enter and leave.

Design Aspects to Consider

1. **Data Model**: Design a database schema to manage parking spots, vehicles, and parking transactions.

2. **Algorithm for Spot Allocation**: Develop an algorithm to efficiently assign parking spots to incoming vehicles.
3. **Fee Calculation Logic**: Implement logic to calculate fees based on parking duration and vehicle type.
4. **Concurrency Handling**: Ensure the system can handle multiple vehicles entering or exiting simultaneously.

Data Model

Entities

1. **ParkingSpot**
   - `id` (UUID): Unique identifier for the parking spot.
   - `floorNumber` (integer): The floor number where the parking spot is located.
   - `spotNumber` (integer): The number of the parking spot.
   - `size` (enum: MOTORCYCLE, CAR, BUS): The size of the parking spot.
   - `isOccupied` (boolean): Indicates whether the spot is currently occupied.
2. **Vehicle**
   - `licensePlate` (string): Unique identifier for the vehicle.
   - `size` (enum: MOTORCYCLE, CAR, BUS): The size of the vehicle.
   - `entryTime` (datetime): The time when the vehicle entered the parking lot.
   - `exitTime` (datetime): The time when the vehicle exited the parking lot (nullable).
3. **ParkingTransaction**
   - `id` (UUID): Unique identifier for the transaction.
   - `licensePlate` (string): Reference to the vehicle's license plate.
   - `parkingSpotId` (UUID): Reference to the parking spot.
   - `entryTime` (datetime): The time when the vehicle entered the parking lot.
   - `exitTime` (datetime): The time when the vehicle exited the parking lot.

- `fee` (decimal): The calculated fee for the parking duration.

Database Schema

```
// models/ParkingSpot.js
const mongoose = require('mongoose');

const ParkingSpotSchema = new mongoose.Schema({
    floorNumber: { type: Number, required: true },
    spotNumber: { type: Number, required: true },
    size: { type: String, enum: ['MOTORCYCLE', 'CAR', 'BUS'],
    isOccupied: { type: Boolean, default: false }
});

module.exports = mongoose.model('ParkingSpot', ParkingSpotSc

// models/ParkingTransaction.js
const mongoose = require('mongoose');

const ParkingTransactionSchema = new mongoose.Schema({
    licensePlate: { type: String, required: true },
    parkingSpotId: { type: mongoose.Schema.Types.ObjectId, re
    entryTime: { type: Date, required: true },
    exitTime: { type: Date },
    fee: { type: Number }
});

module.exports = mongoose.model('ParkingTransaction', Parking
```

Algorithm for Spot Allocation

Steps

1. **Receive Vehicle Entry Request**: Capture vehicle details including size.

2. **Find Available Spot**: Query database for an available spot matching the vehicle size.
3. **Assign Spot**: Mark the spot as occupied and save the vehicle entry time.
4. **Update Database**: Insert a new entry in the `ParkingTransaction` table with entry time.

Fee Calculation Logic

Steps

1. **Receive Vehicle Exit Request**: Capture vehicle details and exit time.
2. **Fetch Transaction**: Retrieve the corresponding parking transaction.
3. **Calculate Duration**: Calculate the duration of the stay.
4. **Calculate Fee**: Apply fee rules based on vehicle type and duration.
5. **Update Transaction**: Update the transaction with exit time and fee.
6. **Free Spot**: Mark the parking spot as available.

Fee Calculation Rules

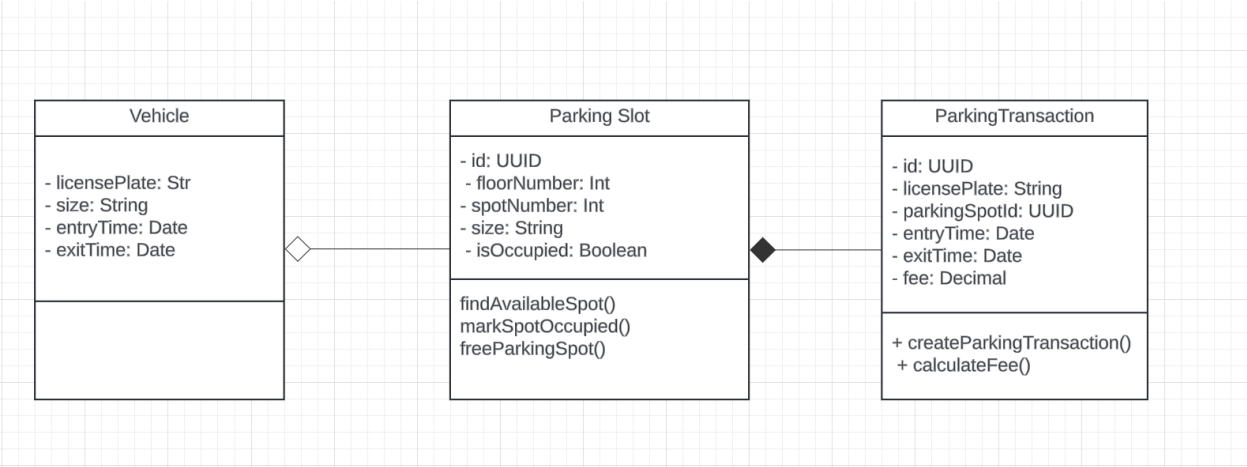
- Motorcycles: $1 per hour
- Cars: $2 per hour
- Buses: $5 per hour

Parking Lot APIS

Concurrency Handling

1. **Database Transactions**: Use database transactions to ensure atomicity of operations.
2. **Locking**: Implement row-level locking when updating parking spot availability to prevent race conditions.
3. **Concurrency Control**: Ensure consistent state using optimistic or pessimistic concurrency control mechanisms.

Class Diagram With Relationship:



Relationships

- **ParkingSpot** has a one-to-many relationship with **ParkingTransaction**:
  - A `ParkingSpot` can be associated with many `ParkingTransactions` over time.
  - A `ParkingTransaction` references a single `ParkingSpot`.
- **Vehicle** does not directly interact with **ParkingSpot** or **ParkingTransaction** in the database schema but is used in the business logic.

Conclusion

This low-level design document outlines the data model, algorithms, and logic required to build a smart parking lot system. The system efficiently manages parking spot allocation, vehicle check-in and check-out, fee calculation, and real-time updates while ensuring concurrency handling.