Vehicle Parking App – V2 Project Report

QUICKPARK V2

Author:

Tathagata Banerjee
Roll Number: DS23F3002603
Email: 23f3002603@ds.study.iitm.ac.in

A passionate Data Science student with interest in web applications and system design.

AI/LLM Usage:

Used ChatGPT (GPT-5-mini) to generate boilerplate code, assist with frontend color gradings, implement JWT authentication, and create OpenAPI/Swagger YAML specifications for the APIs. Approximately 20% of the code and documentation leveraged AI assistance**.**

Description:

This project implements a web-based parking management system.
Users can view available parking lots, reserve parking spots, and release them when done.
Admins can create, edit, and delete parking lots, monitor usage, and generate reports.
The system tracks reservation history, calculates parking charges, and caches frequent queries for faster access.
Asynchronous tasks handle report generation and CSV exports without blocking the main application.
Secure access is ensured using JWT authentication and role-based functionality.

Technologies Used:

- **Flask** – Backend API framework for route handling and request processing.
- **Flask-SQLAlchemy** – ORM to handle database operations efficiently.
- **SQLite** – Lightweight database for development and testing.
- **Redis** – Caching frequent queries to reduce DB load.
- **Celery** – Asynchronous tasks for sending monthly reports and CSV exports.
- **Vue.js** – Frontend framework for dynamic UI.
- **Bootstrap** – Styling and responsive design.
- **JWT** – Secure token-based authentication for user and admin.

- **FPDF / CSV** – Report generation in PDF and CSV formats.

Purpose: Each technology was chosen to ensure a lightweight, scalable, and maintainable web application with good performance and security.

DB Schema Design:

- **User** – uid (PK), password, first_name, last_name, age, mob_no, email, is_blocked
- **Admin** – aid (PK), password, first_name, last_name, age, mob_no
- **ParkingLot** – lotid (PK), location, address, pin, price, no_of_slot, description, is_paused
- **ParkingSpot** – spotid (PK), lotid (FK), status
- **ReserveParkingSpot** – id (PK), uid (FK), lot_id (FK), spot_id (FK), price, vehicle_number, reserved_at, released_at, total_cost

Constraints and Notes:

- Foreign Key relations enforce consistency (user cannot reserve invalid spots, spot belongs to a lot).
- vehicle_number validated via regex (XX-00-X-0000).
- Cascading deletes used for spots when lots are deleted.
- Spot status tracks occupancy.

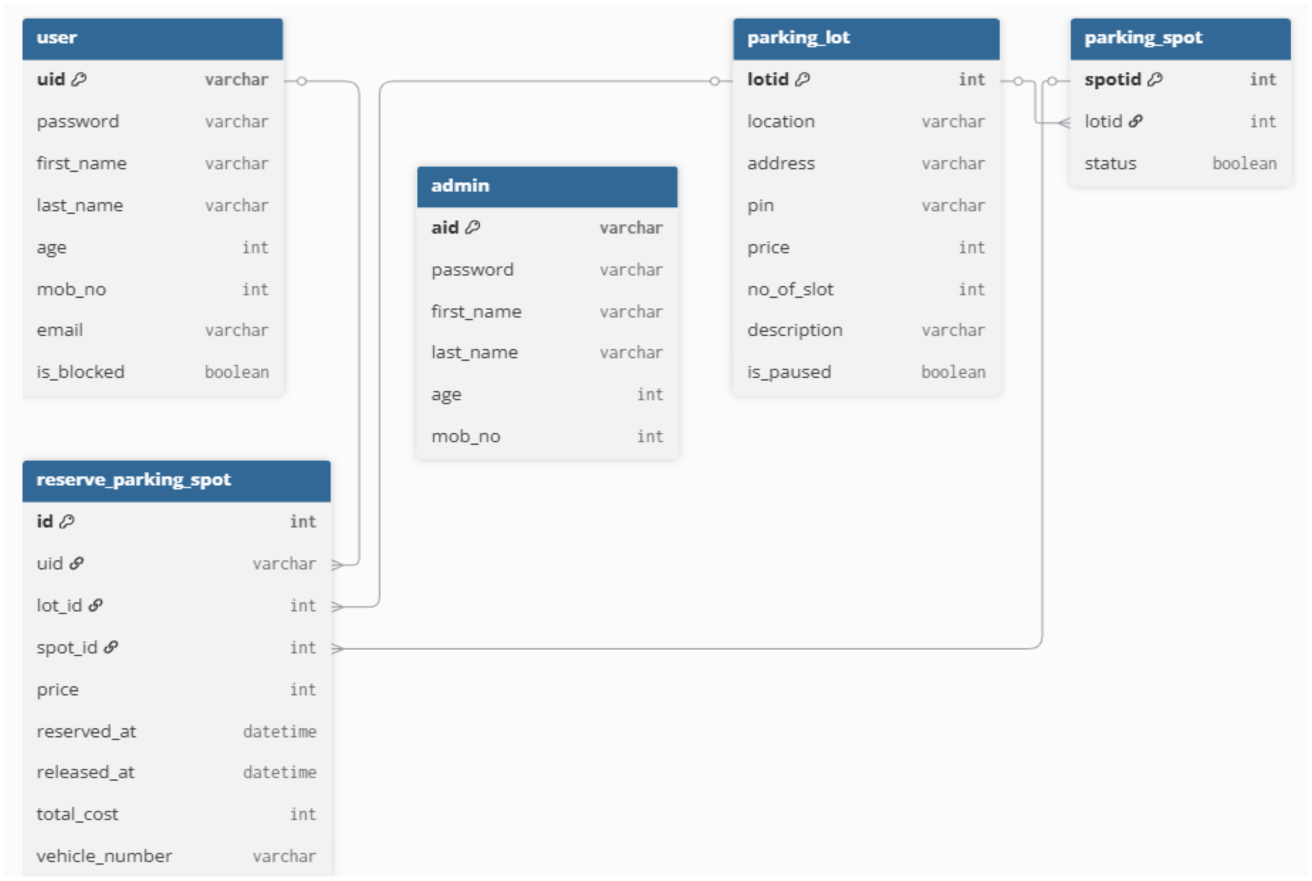
Database Schema Design Reasoning:

1. **Separation of Concerns**
    - Each entity (User, Admin, ParkingLot, ParkingSpot, Reservation) is represented as a separate table to clearly separate responsibilities.
    - This avoids duplication and ensures maintainability.
2. **Normalization**
    - Tables are normalized to reduce redundancy. For example, ParkingSpot references ParkingLot via lotid, instead of storing lot details in every spot.
    - This improves consistency and simplifies updates.
3. **One-to-Many Relationships**
    - A ParkingLot has many ParkingSpots.

- A User can have multiple reservations (ReserveParkingSpot), but each reservation is tied to one spot at a time.
- This aligns with real-world logic of parking management.

4. **Constraints for Data Integrity**
   - Primary keys ensure uniqueness (uid, aid, lotid, spotid).
   - Foreign keys enforce valid references (ReserveParkingSpot.uid → User.uid, spot_id → ParkingSpot.spotid).
   - status in ParkingSpot ensures accurate occupancy tracking.
5. **Support for Features**
   - ReserveParkingSpot table stores timestamps (reserved_at, released_at) for calculating duration and cost.
   - User.is_blocked allows admin to control access.
   - ParkingLot.is_paused allows temporary lot closure without deleting data.

ER- Diagram:



API Design:

The project provides RESTful APIs for both users and admins. User APIs allow signup, login, profile management, parking spot reservation and release, and fetching active/reservation history. Admin APIs manage users, parking lots, and slots, generate reports, and monitor usage statistics. APIs are implemented in Flask using Blueprints for modularity, with JWT-based authentication for security. Caching via Redis improves performance for frequent queries, and asynchronous tasks (Celery) handle report generation, Mailing and CSV exports. The API specifications are documented separately in a YAML file [23F3002603api.yaml].

Architecture and Features:

The project follows a modular Flask architecture. controller_bp contains all API routes, separated into user and admin functionalities. The model folder defines database models using SQLAlchemy ORM. Redis caching is used for frequently accessed data, and Celery handles asynchronous tasks like monthly report generation. Templates and static files are structured under frontend (if used), while configuration and utilities (e.g., caching, JWT helpers) reside in utils.

Implemented Features:

- User features: Signup/login, reserve/release parking spots, view active and historical reservations, monthly report (PDF/CSV).
- Admin features: Login, manage users, block/unblock accounts, create/edit/delete parking lots and slots, monitor bookings and revenue, generate CSV/async reports.
- Other features: Redis caching for faster dashboard/history queries, Celery tasks for background report generation, vehicle format validation via regex, JWT auth for secure access, and PDF/CSV export functionality.

Extra features:

- Chart.js for charts in user and admin dashboard.
- Pause and resume of a parking lot.
- Block and unblock a user
- Search Functionality.
- Some UI improvements.

Video link: Click here

[https://drive.google.com/file/d/17YLbaG9dLp5h5IF38odc391AQ7QORj6g/view?usp=sharing]