**Project Report: Vehicle Parking System**

**Author**

* **Name:** Komal Kumar
* **Roll Number:** 24f1001472
* **Email:** 24f1001472@ds.study.iitm.ac.in
* **About:** I am a passionate developer with a keen interest in building full-stack web applications. I enjoy tackling challenges related to system architecture, database design, and creating seamless user experiences.

**Description**

This project requires building a comprehensive vehicle parking system with a clear separation between a Flask backend API and a Vue.js frontend. The system needs to support two distinct roles (Admin and User) with different functionalities, handle real-time parking spot management, and include asynchronous background jobs for reports and notifications. Approximately 40% of the code for both the backend and frontend was generated using an AI assistant for tasks like initial setup, database modeling, and component generation.

**Technologies Used**

* **Backend**:
  + **Flask**: A lightweight Python web framework used to build the core REST API.
  + **Flask-SQLAlchemy**: An ORM for interacting with the database using Python objects, simplifying database operations.
  + **Flask-JWT-Extended**: Implements JSON Web Token (JWT) authentication for securing API endpoints.
  + **Flask-Bcrypt**: Used for securely hashing user passwords.
  + **Celery**: A distributed task queue for running time-consuming background jobs asynchronously.
  + **Redis**: An in-memory data store used as the message broker for Celery.
* **Frontend**:
  + **Vue.js (v3)**: A progressive JavaScript framework for building a reactive Single Page Application (SPA).
  + **Vue Router**: The official router for Vue.js, used to handle client-side navigation.
  + **Pinia**: The official state management library for Vue, used to manage global application state.
  + **Axios**: A promise-based HTTP client for making API requests from the frontend to the backend.
  + **Bootstrap 5**: The exclusive CSS framework used for styling and creating a responsive user interface.

**DB Schema Design**

The database is designed with four interconnected tables to logically separate concerns and ensure data integrity.

* **User Table (users)**:
  + id (Integer, Primary Key): Unique identifier for each user.
  + username (String, Unique, Not Null): The user's chosen name.
  + email (String, Unique, Not Null): The user's email, used for login.
  + password\_hash (String, Not Null): The securely hashed password.
  + role (String, Not Null, Default: 'user'): Defines user permissions ('user' or 'admin').
* **ParkingLot Table (parking\_lots)**:
  + id (Integer, Primary Key): Unique identifier for each parking lot.
  + name (String, Unique, Not Null): The public name of the lot.
  + address (Text, Not Null): The physical address of the lot.
  + pin\_code (String, Not Null): The postal code.
  + price\_per\_hour (Float, Not Null): The cost to park for one hour.
  + capacity (Integer, Not Null): The total number of spots in the lot.
* **Parking Spot Table (parking spots)**:
  + id (Integer, Primary Key): Unique identifier for each spot.
  + spot number (Integer, Not Null): The number of the spot within a lot.
  + status (String, Not Null, Default: 'Available'): The current status ('Available' or 'Occupied').
  + lot\_id (Integer, Foreign Key to parking\_lots.id): Links the spot to a specific parking lot.
* **Booking Table (bookings)**:
  + id (Integer, Primary Key): Unique identifier for each booking transaction.
  + user\_id (Integer, Foreign Key to users.id): Links the booking to a user.
  + spot\_id (Integer, Foreign Key to parking\_spots.id): Links the booking to a specific spot.
  + park\_in\_time (Date Time, Not Null): Timestamp when the booking started.
  + park\_out\_time (DateTime, Nullable): Timestamp when the booking ended.
  + cost (Float, Nullable): The final calculated cost of the booking.

**Design Rationale:** This schema normalizes data to reduce redundancy. Separating

Bookings from ParkingSpots allows a spot to have a simple current status while the Booking table maintains a complete history of all transactions. Foreign key constraints are used to maintain referential integrity.

**API Design**

The API is designed as a RESTful service, organized by resources and roles using Flask Blueprints.

* **Authentication (/auth)**: This blueprint handles user registration (/register) and login (/login), and is responsible for validating credentials and issuing JWTs.
* **User (/api)**: This blueprint contains all endpoints for standard user actions, such as viewing lots, booking/releasing spots, and viewing booking history. All routes require a valid 'user' role JWT.
* **Admin (/admin)**: This blueprint provides endpoints for administrative control, including full CRUD operations for parking lots and system monitoring. All routes are protected by a custom decorator that requires a valid 'admin' role JWT.

**Architecture and Features**

* **Project Organization**:
  + The project is structured into

backend and frontend directories.

* + The backend follows a modular Flask application factory pattern, with logic separated into

routes/, models/, and tasks/ directories. Configuration is managed in

config.py.

* + The frontend is a standard Vue CLI project with source code organized into

views/, components/, router/, store/, and services/.

* **Features Implemented**:
  + **Role-Based Access Control**: Securely handles 'Admin' and 'User' roles using JWTs and a custom decorator (@admin\_required) to protect administrative endpoints.
  + **Full Parking Lifecycle**: Users can view lots, book spots, view their active bookings, and release spots, with the cost calculated automatically.
  + **Admin Management Dashboard**: A comprehensive dashboard for admins to create, edit, and delete parking lots, as well as monitor all users and the real-time status of every spot.
  + **Asynchronous Background Jobs**:
    - A user can trigger a non-blocking CSV export of their booking history.
    - The system uses Celery Beat for scheduled tasks like daily reminders and monthly reports.
  + **Booking History**: Both users and admins can view historical booking records.

**Video:**

**https://drive.google.com/file/d/1-LRzg2nhqFBk73qizOass4SRWXo-pviQ/view?usp=sharing**