# MAD-II Project Report

## Vehicle Parking App V2


| | |
|---|---|
| **Student Name:** | [Your Name] |
| **Roll Number:** | 24f2000305 |
| **Email:** | [Your Email] |
| **Course:** | Modern Application Development II |
| **Term:** | May 2025 |
| **Submission Date:** | November 24, 2025 |

# 1. Project Overview

I have developed a multi-user vehicle parking management application that efficiently manages 4-wheeler parking slots. The system has two roles: Admin (Superuser) who can create, modify, and delete parking lots, and User who can browse available parking lots, book spots, and release them with automatic cost calculation based on parking duration.

## 1.1 Approach to Solution

• Database Design: Designed database schema with proper relationships and foreign keys

• Backend API Development: Created RESTful APIs using Flask framework

• Frontend Development: Built single-page application using Vue.js

• Background Jobs: Implemented Celery for asynchronous operations

• Caching Strategy: Applied Redis caching for performance optimization

• Testing: Conducted comprehensive manual testing of all features

# 2. Frameworks and Libraries Used

| Technology | Version | Purpose |
|---|---|---|
| Flask | 2.3.3 | Web framework for REST APIs |
| Flask-Login | 0.6.3 | Session-based authentication |
| Flask-Caching | 2.1.0 | Redis-backed caching |
| Celery | 5.3.6 | Background task processing |
| Redis | 5.0.4 | Cache store and message broker |
| Vue.js | 3 (CDN) | Progressive JavaScript framework |
| Bootstrap | 5.3.3 | CSS framework |
| SQLite | Built-in | Database |

# 3. Database Schema (ER Diagram)

The database consists of 5 main tables with proper foreign key relationships:

• **users** - Stores user authentication data and role management (admin/user)

• **parking_lots** - Contains parking lot details including pricing information

• **parking_spots** - Represents individual parking spaces with status (A=Available/O=Occupied)

• **reservations** - Records booking details with timestamps for cost calculation

• **export_jobs** - Tracks asynchronous export jobs for CSV generation

### Key Relationships:

• parking_spots.lot_id → parking_lots.id (Many-to-One)

• reservations.spot_id → parking_spots.id (Many-to-One)

• reservations.user_id → users.id (Many-to-One)

• export_jobs.user_id → users.id (Many-to-One)

## 4. API Resource Endpoints

### Authentication APIs:

• POST /api/auth/register - New user registration

• POST /api/auth/login - User login with session

• POST /api/auth/logout - User logout

• GET /api/auth/profile - Get current user details

### Admin APIs:

• GET /api/admin/lots - List all lots with availability

• POST /api/admin/lots - Create new parking lot

• PATCH /api/admin/lots/<id> - Update lot details

• DELETE /api/admin/lots/<id> - Delete empty lot

• GET /api/admin/users - List registered users

• GET /api/admin/reservations - List all reservations

• GET /api/admin/dashboard - Dashboard statistics with charts

### User APIs:

• GET /api/user/lots - List available lots

• POST /api/user/reservations - Book parking spot

• POST /api/user/reservations/<id>/release - Release spot

• GET /api/user/exports - Request CSV export

# 5. Key Features Implemented

## 5.1 Authentication & Authorization

• Session-based authentication using Flask-Login

• Role-based access control (admin vs user)

• Admin automatically created on database initialization

• Password hashing with Werkzeug

## 5.2 Admin Functionality

• Create/edit/delete parking lots

• Automatic spot generation when creating lots

• View dashboard statistics with visual charts

• View all registered users

• View all active and completed reservations

• Delete validation (prevents deleting occupied lots)

## 5.3 User Functionality

• Browse available lots with real-time availability

• Book parking spot (automatic first-available assignment)

• Release spot with automatic cost calculation

• View reservation history with visual charts

• Export booking history to CSV

• Real-time cost calculation based on parking duration

## 5.4 Background Jobs

• Daily Reminders at 18:00 (Celery beat scheduled task)

• Monthly Reports on 1st of month (HTML report generation)

• CSV Export (User-triggered async Celery task)

• All jobs tracked in database with status

## 5.5 Performance Optimization

• Redis caching on frequently accessed endpoints

- Cache expiry: Admin (300s), User lots (120s)
- Automatic cache busting on data modifications
- Efficient SQL queries with proper indexing

## 6. Project Structure

The project follows a modular architecture with clear separation of concerns:

**Backend Structure:**
• backend/app.py - Flask application factory and Celery configuration
• backend/extensions.py - Flask-Login and Cache instances
• backend/models/ - Database models and business logic
• backend/routes/ - API endpoints (auth, admin, user)
• backend/tasks.py - Celery background tasks

**Frontend Structure:**
• frontend/index.html - Bootstrap shell and entry point
• frontend/src/main.js - Vue application bootstrap
• frontend/src/api.js - API fetch wrappers
• frontend/src/components/ - Vue components (AuthPane, AdminPanel, UserPanel)

## 7. Use of AI/LLM

I used GitHub Copilot in this project for code autocompletion, documentation suggestions, and identifying simple syntax errors. However, all core logic, architecture design, database schema, API structure, and business logic decisions were my own original work. AI assistance was limited to improving coding efficiency and identifying potential bugs.

## 8. Challenges and Solutions

**Session Management with CORS:** Resolved CORS issues by adding credentials: 'include' in all fetch calls and proper Flask session configuration

**Concurrent Spot Allocation:** Fixed race condition using SQL transaction with immediate status update to prevent double booking

**Celery Flask Context:** Created custom ContextTask class to provide Flask application context in background jobs

**Cache Invalidation Strategy:** Implemented centralized cache keys with automatic cache busting on data modifications

## 9. Conclusion

I have successfully developed a fully functional parking management application that meets all core MAD-II requirements including: two-role system with RBAC, session-based authentication using Flask-Login, comprehensive RESTful APIs, admin lot management with CRUD operations, user booking system with automatic spot allocation, cost calculation based on parking duration, Redis caching for performance optimization, Celery background jobs for scheduled tasks (daily reminders and monthly reports), and CSV export functionality. The code is well-structured and maintainable with clear separation of concerns. Visual charts have been implemented using Chart.js for both admin and user dashboards. The admin panel includes a detailed reservation view showing all active and completed bookings.

## 10. Video Presentation

The project demonstration video (approximately 3 minutes) is available here:

**Video Link:** [https://drive.google.com/your-video-link]

**Video Contents:**

• Introduction and project overview (30 seconds)

• Approach to problem statement (30 seconds)

• Key features demonstration (90 seconds)

• Additional features and enhancements (30 seconds)

**Declaration:** I hereby declare that this project is my own work and the code has been written by me with assistance from AI tools as mentioned in Section 7. All references and resources used have been properly cited. The implementation follows best practices and adheres to the project requirements specified in the MAD-II course.

## 11. References


• Flask Documentation - https://flask.palletsprojects.com/

• Vue.js Documentation - https://vuejs.org/

• Celery Documentation - https://docs.celeryproject.org/

• Flask-Login Documentation - https://flask-login.readthedocs.io/

• Bootstrap Documentation - https://getbootstrap.com/

• Chart.js Documentation - https://www.chartjs.org/

• Redis Documentation - https://redis.io/documentation


Signature: _____ Date: _____