Faculty of Engineering - Alexandria University

Smart City Parking Management System

Assigned For

Database Management Systems

Computer & Systems Engineering 2026

GitHub Repo Link

Student 1:	**Moustafa Esam El-Sayed Amer**	ID:	**21011364**
Student 2:	**Ahmed Mostafa Elmorsy Amer**	ID:	**21010189**
Student 3:	**Ahmed Ayman Ahmed Abdallah**	ID:	**21010048**
Student 4:	**Ebrahim Alaa Eldin Ebrahim**	ID:	**21010017**

Contents

1 Introduction

In the era of rapid urbanization and technological advancement, effective management of parking spaces has become a critical challenge for modern cities. The Smart City Parking Management System is an innovative solution aimed at optimizing parking space utilization, reducing traffic congestion, and enhancing user convenience. This project leverages Internet of Things (IoT) technologies, a robust database management system, and an intuitive software web-interface to provide a seamless parking experience for drivers, parking lot managers, and administrators.

The system is designed to simulate real-world parking scenarios using IoT sensors that monitor the availability and status of parking spots in real-time. Drivers can reserve parking spaces through a user-friendly application, while parking lot managers can oversee reservations and ensure efficient space allocation. Additionally, administrators are provided with tools to manage system-wide operations and generate insightful reports. By integrating cutting-edge technologies and intelligent design, the Smart City Parking Management System aspires to contribute to the vision of a smarter, more sustainable urban environment.

2 Database Design

This script describes a parking management system database schema and includes tables, constraints, and a trigger to manage data integrity and automate certain actions. Below is an analysis of the script and its components.

2.1 Entity-Relationship Diagram

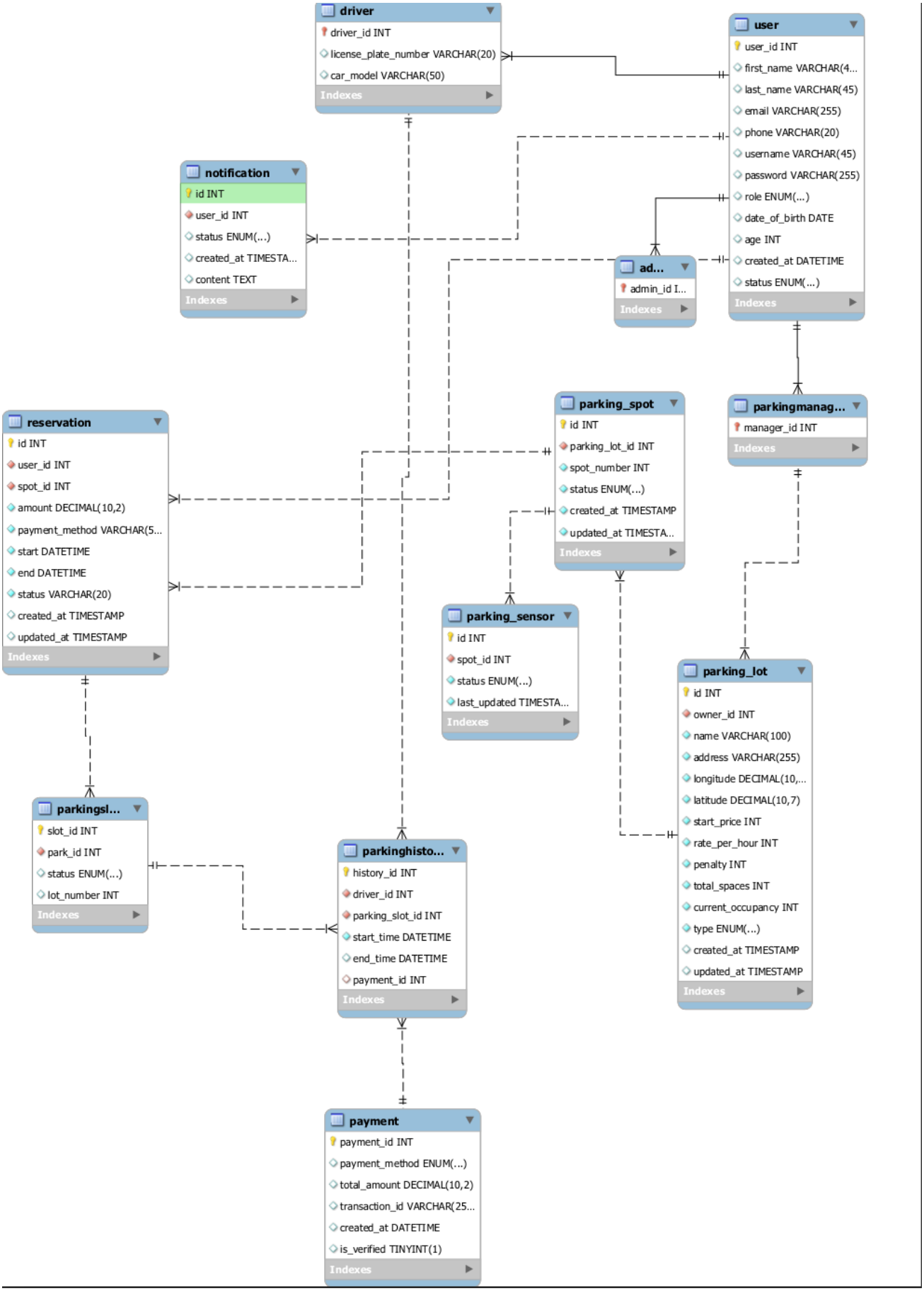

Figure 1: Database Schema

ER Explanation

1. **User Table**
   - **user_id**: Unique identifier for users.
   - **Details**: Personal information like name, contact info, role, and account status.
   - **Security**: Stores hashed passwords.
   - Manages all application users.
2. **Admin Table**
   - **admin_id**: Identifier for administrators.
   - Manages system administrators who oversee the application.
3. **Parking Manageer Table**
   - **manager_id**: Unique identifier for managers.
   - Manages parking managers who control the parking lots.
4. **Driver Table**
   - **driver_id**: Unique identifier for drivers.
   - **license_plate_number**: Driver's vehicle license plate.
   - **car_model**: The make and model of the driver's car.
   - This table handles information about drivers using the service.
5. **Parking Lot Table**
   - **id**: Unique identifier for a parking lot.
   - **owner_id**: Associates the parking lot owner.
   - **Details**: Includes attributes like name, address, geographical coordinates (longitude, latitude), pricing details (start price, rate per hour, penalties), and capacity (total spaces and current occupancy). In addition to, TimeStamps to track creation and updates.
   - Manages individual parking lot details.
6. **Parking Spot Table**
   - **id**: Unique identifier for a parking spot.
   - **parking_lot_id**: Associates the spot with a parking lot.
   - **spot_number**: Numeric identifier within the lot.
   - **status**: Indicates availability, occupancy, or reservations of a parking spot.
   - Tracks spots' status within parking lots.
7. **Reservation Table**
   - **id**: Unique identifier for a reservation.
   - **user_id**: Links to the user making the reservation.
   - **spot_id**: Reserved parking spot.
   - **amount**: Cost of the reservation.
   - **Details**: Includes payment method, start and end times, status, and timestamps.
   - Manages spot-reservations and payment details.
8. **Parking Sensor Table**
   - **id**: Unique identifier for a sensor.
   - **spot_id**: Links to the associated parking spot.
   - Timestamps for the last status update.

- Integrates IoT sensors for real-time parking spots' data update.

9. **Payment Table**
   - **payment_id**: Unique identifier for payments.
   - **Details**: Includes method, total amount, transaction ID, verification status, and timestamp.
   - Handles payment processing.
10. **Notification Table**
    - **id**: Notification identifier.
    - **user_id**: Recipient of the notification.
    - **Details**: Includes content, status, and creation time.
    - Manages system notifications.
11. **Parking History Table**
    - **history_id**: Identifier for historical records.
    - **Details**: Tracks driver activity, including parking spot usage and associated payments.
    - Logs past parking events.

2.2 Generation Scripts

This script describes a parking management system and includes tables, constraints, and a trigger to manage data integrity and automate certain actions. The following is an analysis of the script and its components.

Scripts

The database schema is created as follows:

- Schema name: `parking`
- Character set: UTF-8 (utf8mb4)

Core Tables

1. **User Table**
   - Stores user information including credentials, roles, and status.
   - Uniqueness constraints: `email`, `username`.
   - Enumerated `role` values: `Driver`, `Admin`, `ParkingManager`.
   - Status: `Active`, `Inactive` (default: `Active`).
2. **Driver, ParkingManager, and Admin Tables**
   - These tables store role-specific data.
   - Each references the `User` table using `user_id` with `ON DELETE CASCADE`.
3. **Parking Lot Table**
   - Represents parking lots with details such as location, pricing, and capacity.
   - Foreign Keys: `ParkingManager`.
   - Enumerated Fields: `type` (`ELECTRIC`, `HANDICAP`, `STANDARD`).
   - Includes fields for managing occupancy and parking lot types.
4. **Parking Spot Table**
   - Tracks individual parking spots within parking lots.
   - Ensures unique spot numbers per parking lot using a composite unique constraint on both `parking_lot_id` & `spot_number`.

- Enumerated Fields: `status` (`AVAILABLE`, `RESERVED`, `OCCUPIED`).

5. **Parking Sensor Table**
   - Monitors real-time status of parking spots (`OCCUPIED`, `VACANT`, `FAULTY`).
   - References the `Parking Spot` table.
6. **Reservation Table**
   - Manages reservations, including payment details, time slots, and statuses.
   - Indexes for optimized querying on `user_id`, `spot_id`, `start`, and `end`.
   - References `User` and `Parking Spot` tables with `ON DELETE CASCADE`.

2.3 Triggers

- `after_parking_spot_update` **Trigger**
  - Activates after updates to the `Parking Spot` table.
  - Automates reservation status updates:
    - Marks reservations as `COMPLETED` when a spot becomes `AVAILABLE`.
    - Marks reservations as `ACTIVE` when a spot becomes `OCCUPIED` and the reservation is valid (based on start and end times).

2.4 Constraints for Data Integrity

- **Primary and Foreign Keys**
  - Maintain valid relationships between tables.
  - Cascading deletes to remove dependent data (e.g., `User` deletions remove related reservations).
- **Unique Constraints**
  - Prevent duplicate `email`, `username`, and `parking spot_numbers` within a lot.
- **Enumerations**
  - Restrict values for fields like roles, statuses & types.

3 Performance Optimization

- `reservation` **Table Indexes**
  - **Indexed Columns**
    - `user_id`
    - `spot_id`
    - `start`
    - `end`
  - **Purpose**
    - `user_id` Facilitates efficient lookups for reservations made by a specific user.
    - `spot_id` Accelerates queries for reservations associated with a particular parking spot, crucial for checking availability.
    - `start` Speeds up queries involving the reservation start time, aiding in scheduling and conflict detection.
    - `end` Optimizes queries filtering or sorting by the reservation end time.
  - **Usage Scenarios**
    - Checking for overlapping reservations to prevent double booking.
    - Retrieving active reservations for a user or a parking spot.

    - Generating reports on reservation patterns over time.
  - **Benefits**
    - The indexed `start` and `end` columns are critical for efficiently detecting overlapping reservations. The database can quickly determine whether a new reservation's time range conflicts with existing ones.

4 Iot Simulation

The logic of the script is structured to simulate the dynamics of a parking lot using IoT-based interactions. Below is a detailed breakdown of the main components and processes:

```
class ParkingSimulator:
    def __init__(self, config: SimulationConfig):
        # self.env = simpy.Environment()
        self.env = simpy.rt.RealtimeEnvironment(factor=30)
        self.config = config
        self.api_service = ParkingAPIService(config.api_endpoint)
        self.vehicle_generator = VehicleGenerator()

    def vehicle_arrival(self, lot_config: ParkingLotConfig):
        while True:
            inter_arrival = self.vehicle_generator.get_next_arrival_time(lot_config)
            yield self.env.timeout(inter_arrival)

            vehicle = self.vehicle_generator.generate_vehicle(self.env.now, lot_config)
            self.env.process(self.handle_vehicle(vehicle, lot_config.lot_id))

    def handle_vehicle(self, vehicle: Vehicle, lot_id: int):
        parking_data = self.api_service.find_available_spot(lot_id)

        if parking_data:
            if self.api_service.update_spot_status(parking_data['id'], ParkingSpotStatus.OCCUPIED):
                yield self.env.timeout(vehicle.parking_duration)
                self.api_service.update_spot_status(
                    parking_data['id'],
                    ParkingSpotStatus.VACANT
                )
        else:
            print(f"No parking spot available for vehicle {vehicle.id}")

    def run(self):
        for lot_config in self.config.parking_lots:
            self.env.process(self.vehicle_arrival(lot_config))
        self.env.run()
```

4.1 Environment Setup

- **Simulation Environment:** A `simpy.rt.RealtimeEnvironment` is used to enable real-time simulation. The argument `factor=30` accelerates the simulation time by 30 times relative to real time, allowing scenarios to be tested within a shorter timeframe.

4.2 Vehicle Arrival Process

- The `vehicle_arrival` function continuously generates vehicles based on an inter-arrival time obtained from a `VehicleGenerator`.
- **Inter-Arrival Time:** This is calculated to simulate the randomness of vehicle arrivals at a parking lot, creating a realistic flow of traffic.

- After the inter-arrival time:
  - A new vehicle is generated with attributes like arrival time.
  - The process to handle the vehicle's interaction with the parking lot is initiated.

4.3 Vehicle Handling Logic

- The `handle_vehicle` function manages the interaction between a vehicle and the parking system.
- **Finding a Parking Spot:** The script queries the availability of parking spots in the designated lot using the `ParkingAPIService`.
- **Spot Assignment:** If a spot is available, its status is updated to `OCCUPIED`.
- **Parking Duration:** The vehicle occupies the spot for a predefined period, simulated using `timeout`.
- **Vacating the Spot:** After the parking duration, the spot's status is updated to `VACANT`, making it available for the next vehicle.
- **Unavailability Handling:** If no parking spot is available, the system logs the event (e.g., "No parking spot available for vehicle X").

4.4 Multi-Lot Simulation

- The `run` method initializes and manages the simulation across multiple parking lots.
- For each parking lot in the configuration:
  - A `vehicle_arrival` process is started.
  - These processes run concurrently, simulating simultaneous activity across different parking lots.

4.5 Modular Design

- **Separation of Concerns:** The simulator separates the generation of vehicles, parking lot interactions, and spot availability checks. This modular design facilitates easy modification and extension of the simulation logic.
- **API Integration:** The `ParkingAPIService` acts as an interface to simulate interactions with an external IoT-based parking system.
- **Vehicle Attributes:** Vehicles are dynamically generated with attributes like arrival time and parking duration, ensuring diversity in the simulation.

4.6 Event-Driven Simulation

- The simulation follows an **event-driven approach**, where:
  - Events like vehicle arrivals and parking durations are triggered based on a timeline managed by `simpy`.
  - Processes yield control using `timeout` to wait for the next scheduled event.
- This approach ensures efficient simulation without blocking other activities.

4.7 Scalability

- The logic is scalable and can accommodate:
  - New types of vehicles or parking lots.
  - Advanced policies for parking spot allocation.
  - Integration with other IoT services.

4.8 Summary

The script effectively simulates the flow of vehicles in and out of parking lots, leveraging randomness, modularity, and API interactions to mimic real-world conditions. This makes it a powerful tool for testing and refining parking management systems.

5 Reporting & Analysis

- **Administrator Analytics** This is a Revenue Analysis Report generated through Jasper Analytics. It provides a detailed summary of revenue metrics for a parking lot labeled "Test." The report highlights the following key statistics:
  - Reservations: The total number of reservations made is 184.
  - Total Revenue: The revenue generated from these reservations is $1,840.00.
  - Average Revenue per Reservation: Each reservation contributed an average of $10.00.
  - Peak Daily Revenue: The highest revenue recorded for a single day is $1,840.00.
  - Average Daily Revenue: The average revenue earned daily matches the total, at $1,840.00.

  The bar chart visually represents revenue performance for the "Test" parking lot, showing a single large red bar for total revenue. This report is useful for assessing revenue trends and understanding the performance of parking operations.

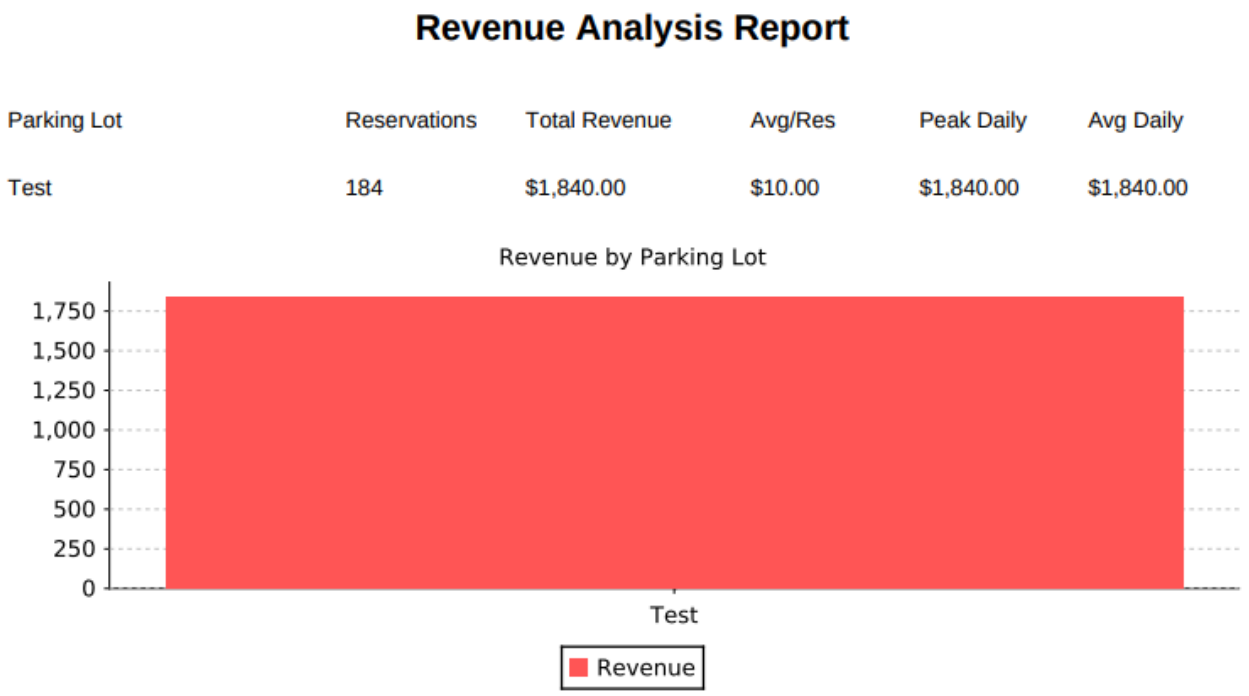

Figure 2: Admin Analytics For Revenue

6 Screenshots & Functionality

- **Driver**

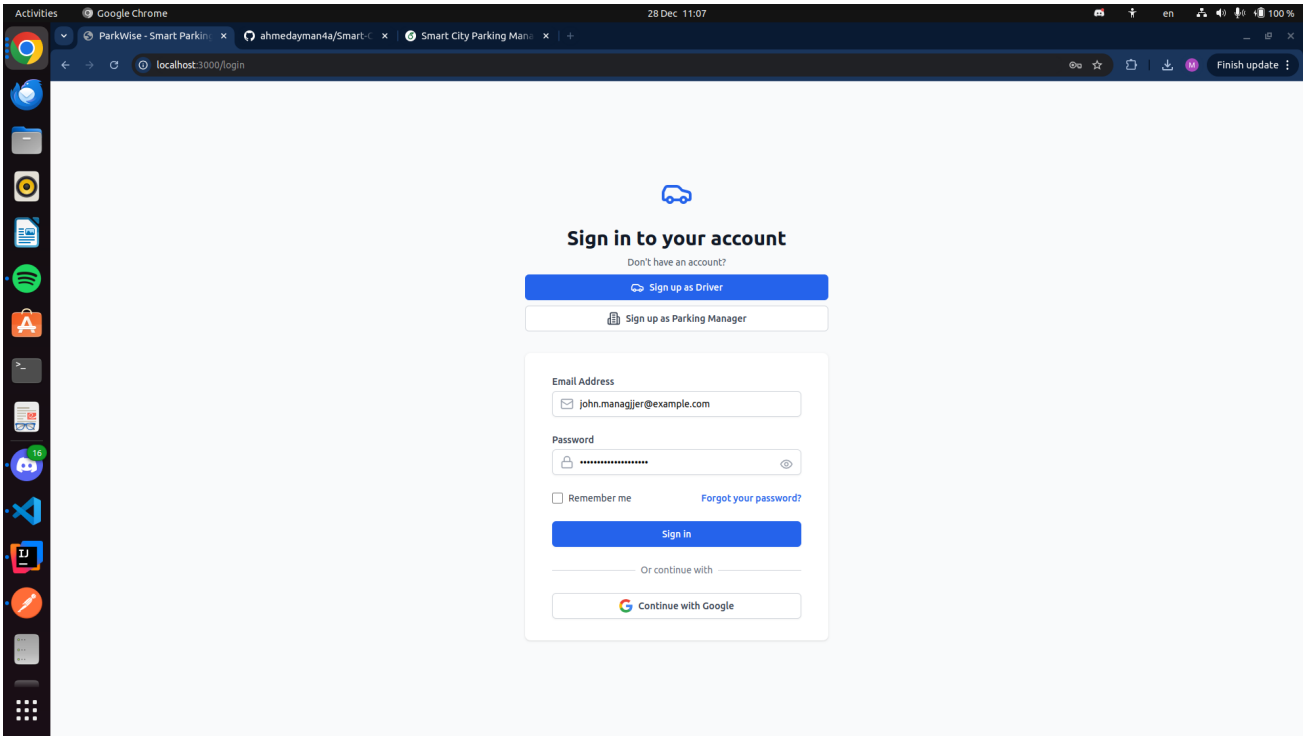

Figure 3: User Login

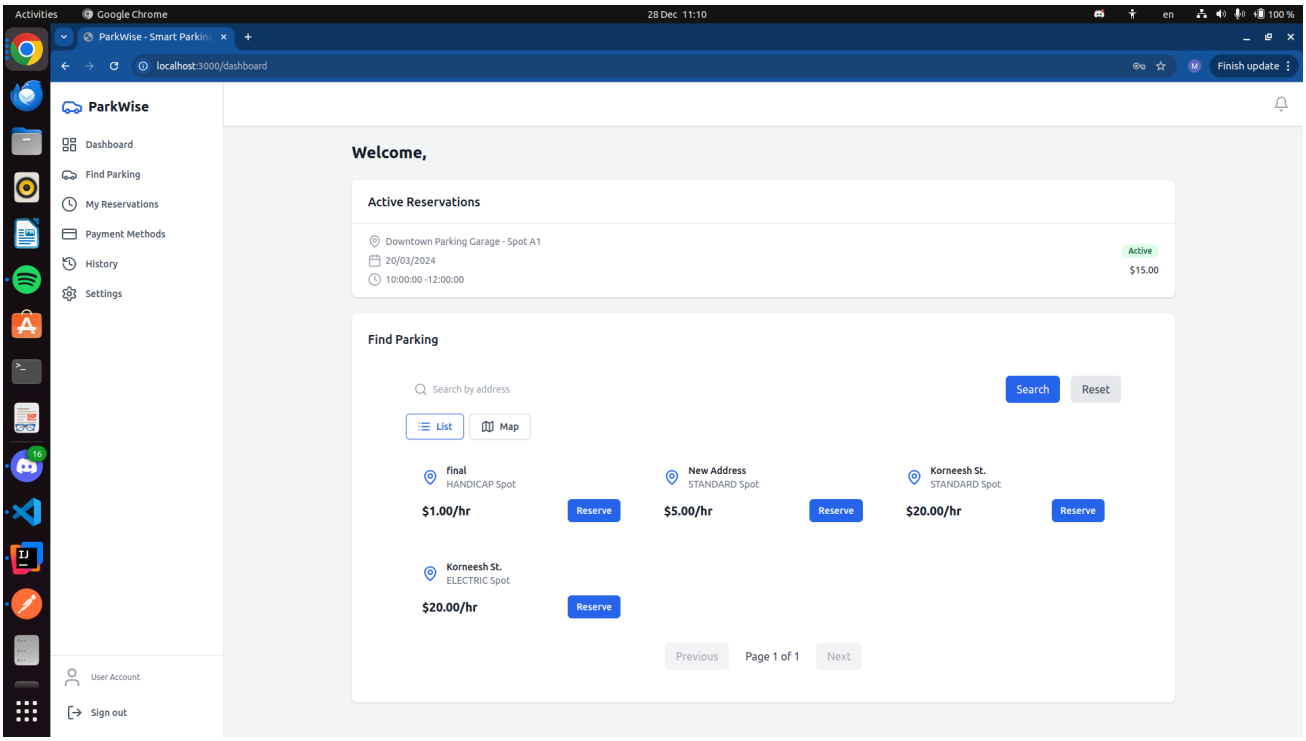

Figure 4: Parking Lots List View

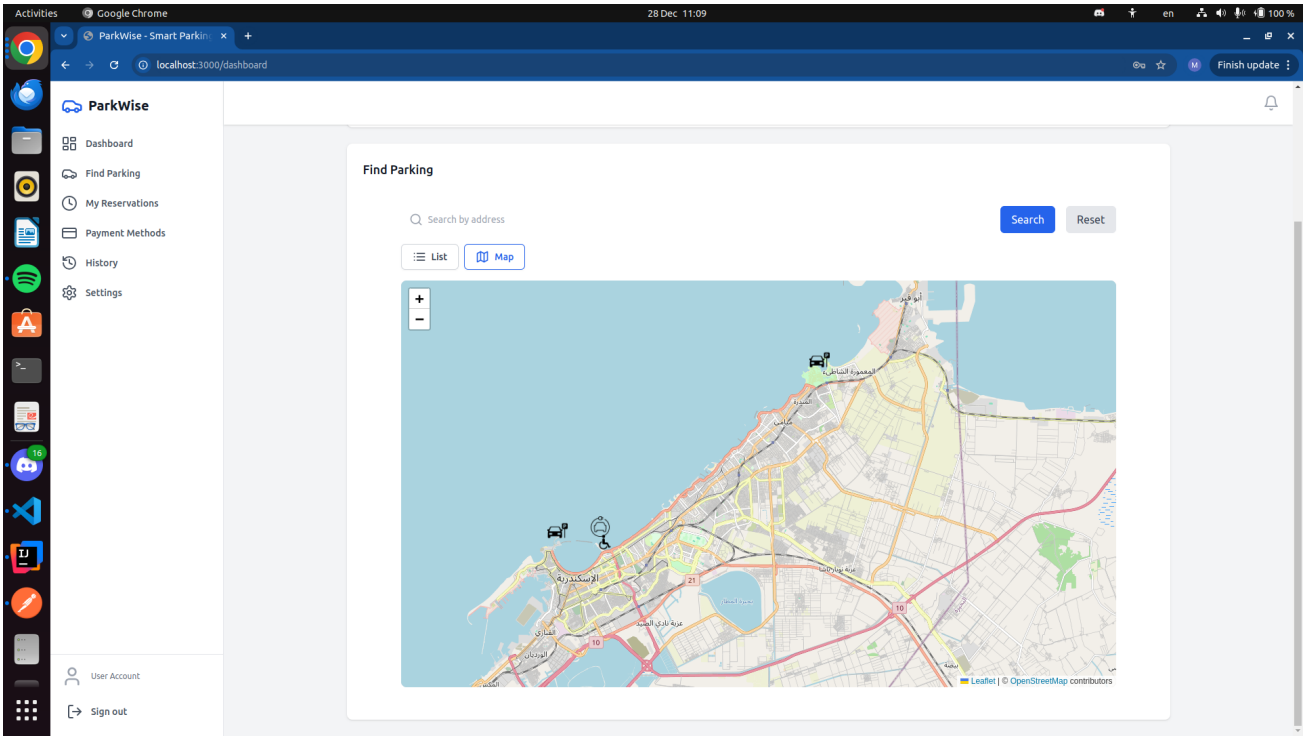

Figure 5: Parking Lots Map View

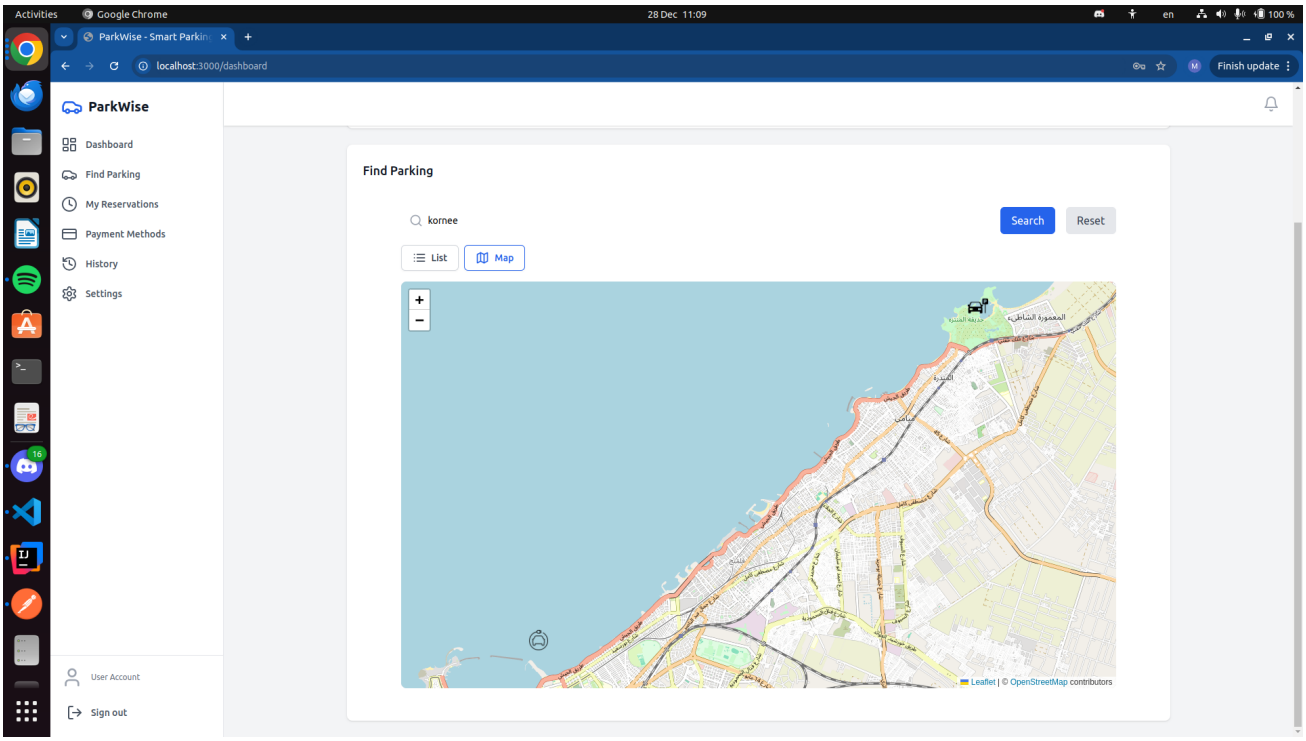

Figure 6: Parking Lots Map Search

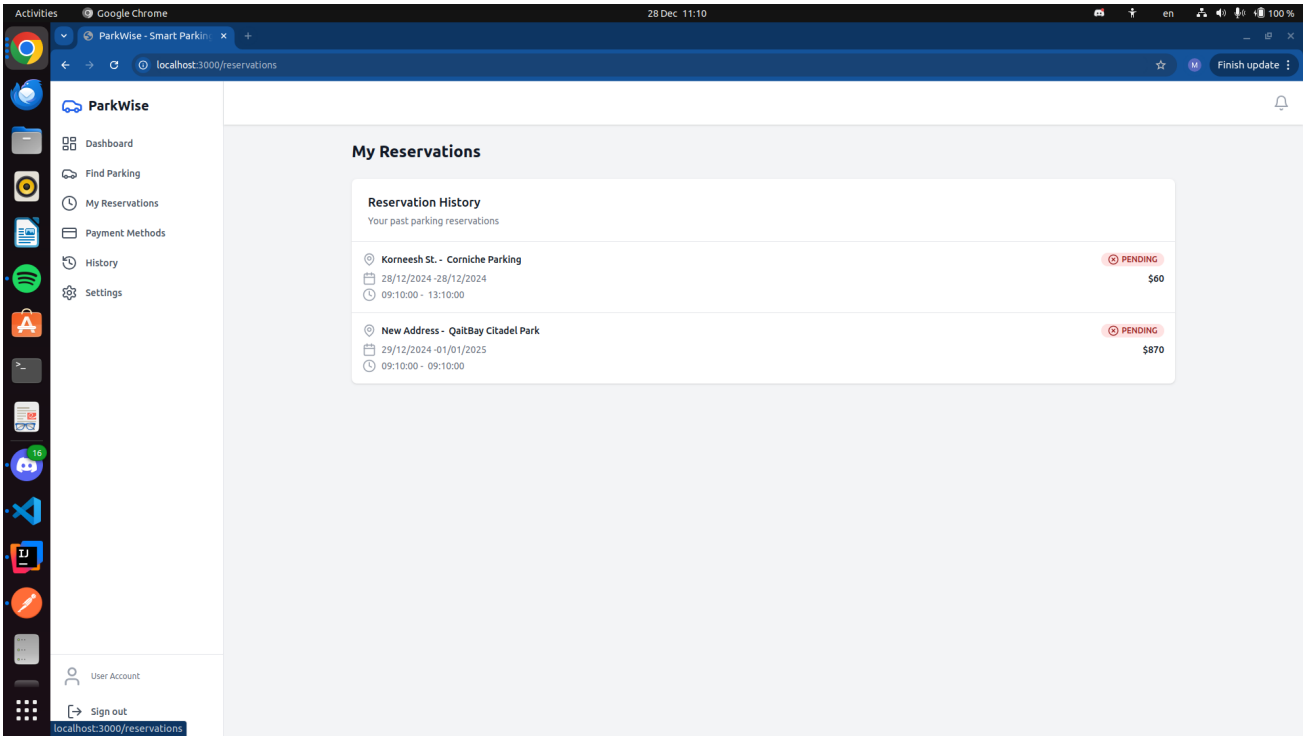

Figure 7: Current Reservations

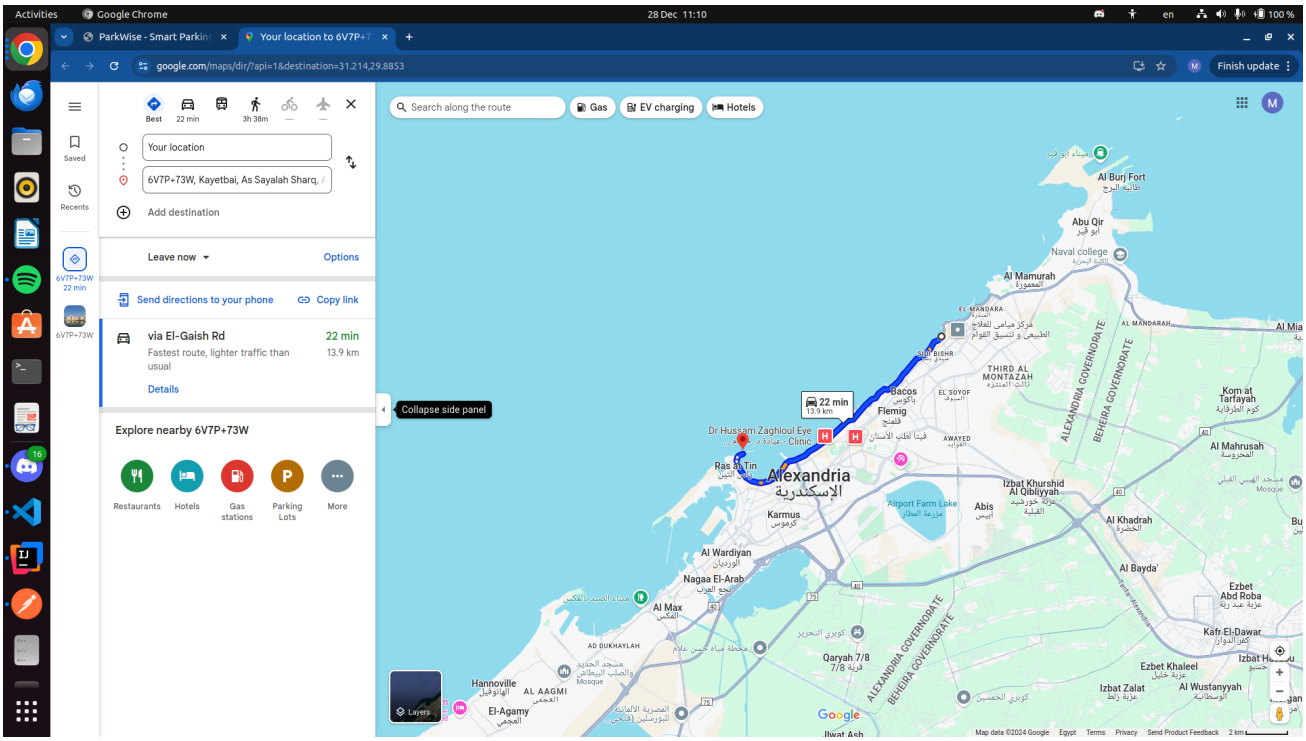

Figure 8: Navigation To Parking Lot

- **Parking Manager**

1 — 2

Parking Facility Name

New Parking Lot

Total Parking Slots

10

Parking Type

Handicap

Address

123, Main St. City

Rate per Hr ($)

$ 15

Penalty Fee ($)

$ 25

Start Price ($)

$ 12.5

Select Location on Map

Register Facility

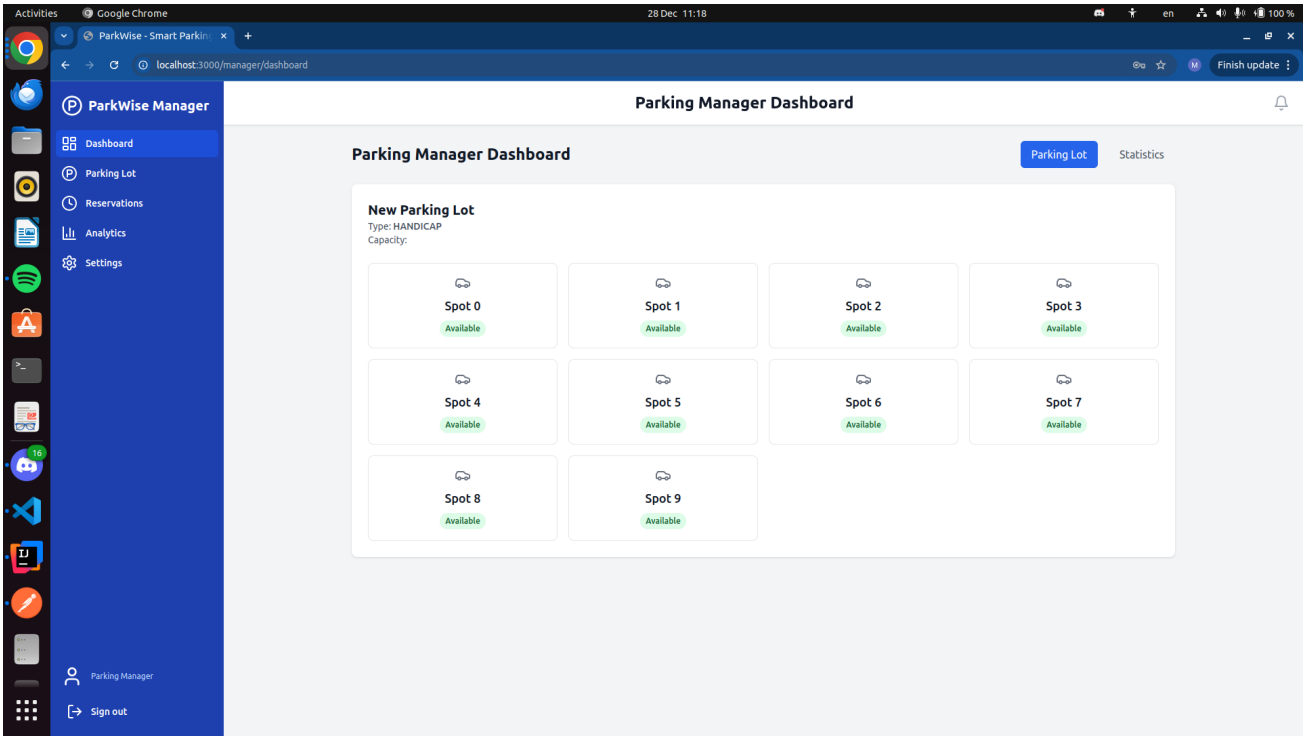

Figure 10: Parking Lot Status

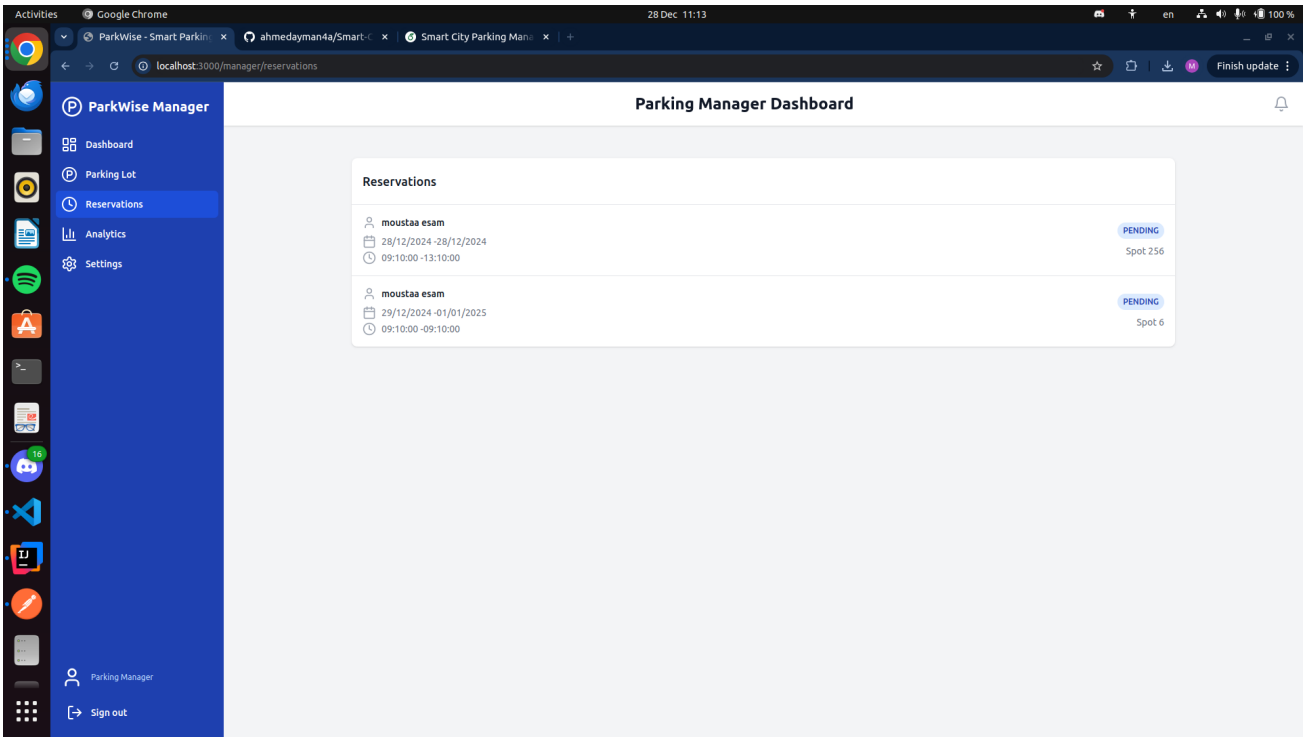

Figure 11: Parking Lot Manager's Reservations View

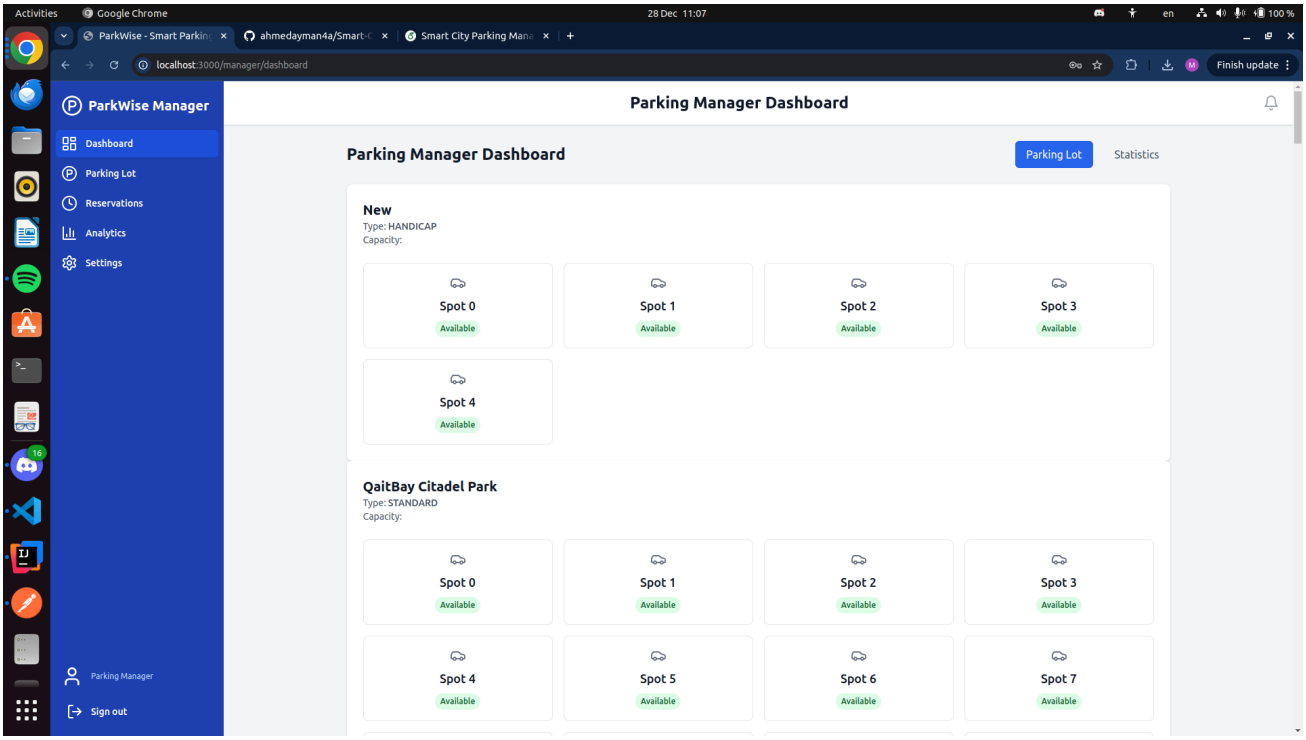

Figure 12: Multiple Lots

- **Admin**

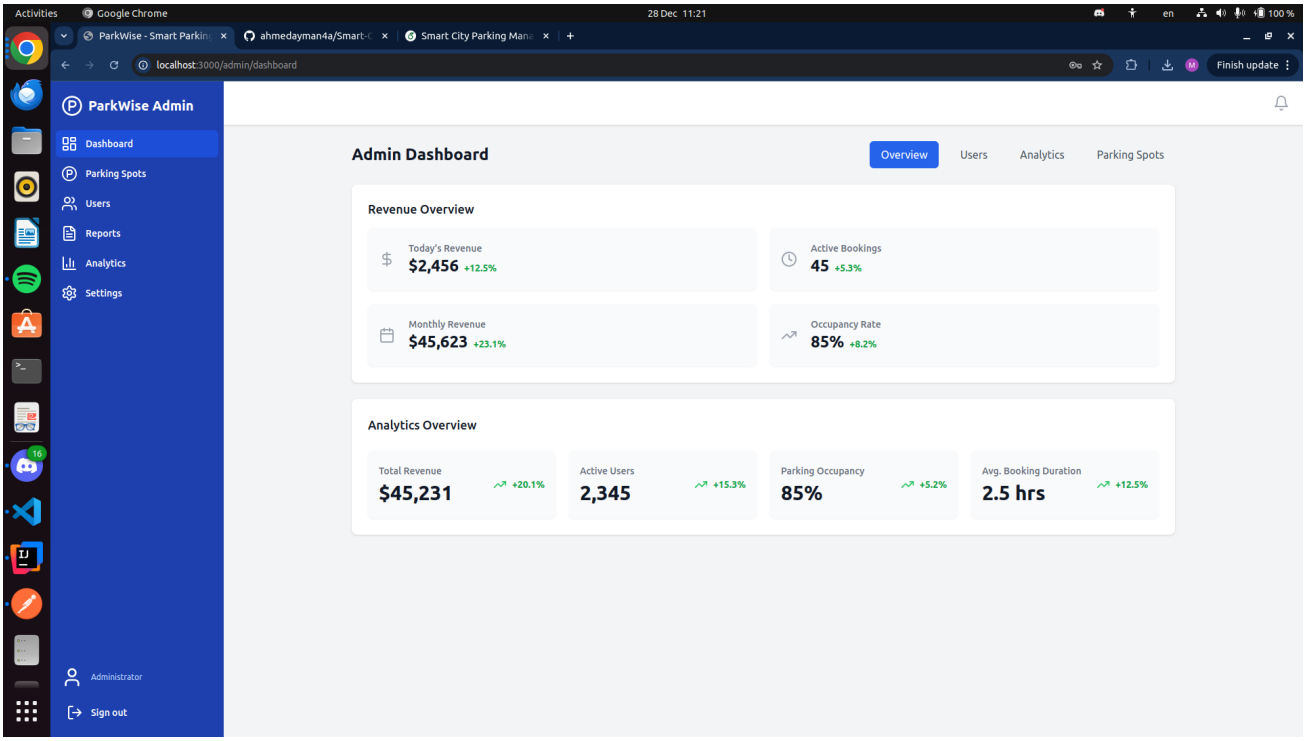

Figure 13: Admin Dashboard

7 Conclusion

The Smart City Parking Management System demonstrates the transformative potential of integrating IoT, database management, and user-centric design to address urban challenges. Through this project, we have developed a scalable and efficient solution that not only improves parking space utilization but also enhances the overall experience for users.

The system's modular architecture ensures adaptability to diverse urban scenarios, while its performance optimization techniques guarantee responsiveness and reliability. By simulating real-world conditions and conducting extensive performance testing, we have validated the system's effectiveness in reducing congestion and facilitating smooth parking operations.

In conclusion, this project serves as a stepping stone towards smarter city infrastructure, showcasing the ability of technology to create sustainable and user-friendly solutions for everyday problems.