Smart Parking Finder

-Management System

(By: Surendra Reddy Gandra – ZPSG-10117)

1.1 Overview

The Smart Parking Finder Backend System is a service-oriented application built using FastAPI and MongoDB, designed to streamline the process of parking management. It provides secure and efficient APIs that allow users to register, authenticate, search for available slots, and manage bookings with ease.

The system eliminates the traditional hassle of manually searching for parking by offering a digital solution where all parking lots, slots, and bookings are stored and managed in a centralized database. With Postman as the testing layer, every API has been validated for correctness, reliability, and scalability.

This backend can be seamlessly integrated with mobile or web applications to create a complete end-to-end smart parking solution, offering faster development cycles and adaptability for future enhancements like payments, notifications, or IoT-based real-time updates.

1.2 Problem Statement

Finding a parking slot in urban areas is often time-consuming, inefficient, and stressful. Drivers typically circle around searching for availability, which increases fuel consumption, traffic congestion, and frustration. Existing systems either lack real-time booking capabilities or do not provide a seamless digital interface for managing parking resources. There is a clear need for a backend system that ensures secure user authentication, reliable slot management, and smooth booking operations.

1.3 Objectives

The key objectives of the Smart Parking Finder Backend System are:

- Provide a **secure user management system** with registration, login, and authentication.
- Enable efficient parking lot and slot management in a centralized database.

- Allow users to book and cancel slots digitally with reliable tracking.
- Ensure APIs are scalable, modular, and ready for integration with mobile/web apps.
- Lay the foundation for future features like payments, real-time IoT integration, and notifications.

1.4 Scope

The scope of the Smart Parking Finder Backend System is limited to providing a secure and efficient backend service for managing users, parking lots, slots, and bookings. It defines the core functionality required for a smart parking solution and is designed to be easily extended in the future.

The system covers:

- 👤 **User Management** – registration, login, and authentication using tokens.
- 🅿️ **Parking Lot Management** – creation and retrieval of parking lot details.
- 🔢 **Slot Management** – defining slots within parking lots and tracking their availability.
- 📌 **Booking Management** – creating and cancelling slot bookings with validation.
- 🗄️ **Data Storage** – using MongoDB for flexible and scalable document-based storage.
- 💉 **API Testing** – validation of all endpoints through Postman to ensure reliability.

2. System Architecture

2.1 High-Level Architecture

The Smart Parking Finder Backend follows a client–server architecture where users interact with the system via RESTful APIs.

- **Client: Any frontend application, mobile app, or Postman tool that consumes the backend APIs.**
- **Backend (Fast API): Handles all requests, executes business logic (authentication, booking, validation), and interacts with the database.**
- **Database (MongoDB): Stores user details, parking lots, slots, and booking records.**

Flow Example:

1. **A user sends a request (e.g., book a slot).**
2. **The Fast API server authenticates the user and validates inputs.**
3. **Business logic is applied (check slot availability, booking rules).**
4. **MongoDB is updated with the booking details.**
5. **The response is returned to the client.**

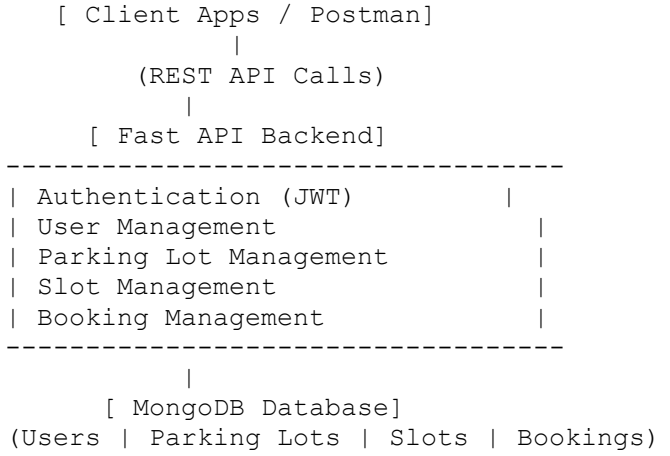
2.2 Components Overview

1. **Authentication & Authorization Module**
    - Handles user registration and login.
    - Generates JWT tokens for secure access.
    - Validates tokens for protected routes.
2. **User Management Module**
    - Manages user accounts and prevents duplicate registrations.
    - Ensures users cannot log in multiple times with the same token.

3. **Parking Lot Module**
    - Stores and retrieves parking lot details.
    - Links lots with available slots.
4. **Slot Management Module**
    - Defines slots for different vehicle types (bike, car, etc.).
    - Tracks real-time slot availability.
5. **Booking Module**
    - Handles creation and cancellation of bookings.
    - Prevents double-booking and enforces business rules.
6. **Database Layer (MongoDB)**
    - Four main collections: **Users, ParkingLots, Slots, Bookings**.
    - Provides persistence and supports scalability.

2.3 Architecture Diagram

Here's the logical flow (can be drawn in your doc as a block diagram):



3. Features

3.1 The **Smart Parking Finder Backend System** provides the following core features:

User Registration & Authentication

- Users can register with unique credentials.
- Duplicate registrations are restricted.
- Secure login system with token-based authentication.
- Prevents multiple logins using the same token at the same time.

Parking Lot & Slot Management (API-based)

- Retrieve available parking lots and slots.
- Slots are categorized based on **vehicle type** (e.g., car, bike).
- Real-time availability updates when a booking is created.

Booking Management

- Users can book available slots for their vehicles.
- Prevents multiple bookings for the same vehicle.
- Ensures a slot cannot be double-booked.
- Booking records include slot, vehicle type, user, and timestamp.

Cancellation & Release of Slots *(if implemented)*

- Users can cancel their booking, freeing up the slot.
- Helps maintain real-time slot availability.

Data Validation & Business Rules

- No multiple registrations for the same user.
- Token validation for secure requests.
- Booking restricted to available slots only.
- User cannot occupy multiple slots with the same vehicle.

API Testing Support

- Postman collection available for easy testing of APIs.
- Each API tested for success, failure, and edge cases

3.2 Technology Stack & State

• Core Language: Python 3.x (with async/await support)

• API Framework: Fast API (ASGI-based)

• Server Engine: Uvicorn

• Database Engine: MongoDB (using Motor async driver)

• Data Handling & Validation: Pydantic

• Logging System: Custom Python logger via get_logger

4. API Design & Functional Modules

The backend system is organized into distinct functional modules, each with dedicated endpoints to handle specific tasks. All APIs follow RESTful principles and use JSON for request and response payloads.

4.1 Authentication & Authorization

- **Purpose:** Secure access to the system and protect sensitive data.
- **Key Features:**
  - User registration (signup)
  - User login with JWT token generation
  - Role-based access control (admin/user)
- **Endpoints Example:**
  - POST /register – register a new user
  - POST /login – authenticate user and return JWT

4.2 User Registration & Login

- **Purpose:** Enable users to create accounts and authenticate.
- **Flow:**
  1. User provides credentials (email/username, password)
  2. Password is hashed and stored securely
  3. JWT token issued on successful login

4.3 Parking Lot Management

- **Purpose:** Manage parking lot information efficiently.
- **Operations:**
  - Create, update, delete parking lots
  - Fetch list of available parking lots
- **Endpoints Example:**
  - POST /parking-lots – add a new parking lot
  - GET /parking-lots – retrieve all parking lots

4.4 Slot Management

- **Purpose:** Maintain parking slot availability and details.
- **Operations:**
  - Add, update, remove slots
  - Check slot availability status
- **Endpoints Example:**
  - POST /slots – add a slot to a parking lot
  - GET /slots – view all slots for a parking lot

4.5 Booking Management

- **Purpose:** Handle all user booking operations.
- **Operations:**
  - Create new bookings
  - Cancel or update existing bookings
  - Check booking status and history
- **Endpoints Example:**
  - POST /bookings – book a parking slot
  - DELETE /bookings/{id} – cancel a booking
  - GET /bookings – retrieve booking details

5.Database Design

The backend system of **Smart Parking Finder** uses **MongoDB** as the primary database. MongoDB is a NoSQL document-oriented database, chosen for its flexibility, scalability, and ease of integration with asynchronous Python frameworks like FastAPI. The asynchronous **Motor driver** is used to interact with MongoDB, allowing non-blocking database operations for high performance under concurrent requests.

Data is organized into collections representing the core entities of the system. Each collection stores related documents with defined fields and supports relationships through references.

5.1 Collections & Schema Design

1. **Users Collection**
    - **Purpose:** Stores all registered users, their credentials, and roles.
    - **Fields:**
        - username (string, unique)
        - email (string, unique)
        - password (hashed string)
        - role (string, e.g., admin or user)
    - **Notes:**
        - Passwords are stored in a hashed format using secure hashing algorithms.
        - Role-based access control is managed via the role field.
2. **ParkingLots Collection**
    - **Purpose:** Maintains information about all parking lots in the system.
    - **Fields:**
        - name (string) – Name of the parking lot
        - location (object or geo-coordinates) – Physical address or GPS coordinates
        - total_slots (integer) – Total number of slots in the lot
        - slots (array) – Embedded documents or references to slots in this lot
    - **Notes:**
        - Each parking lot can contain multiple slots, stored as an array of slot documents or references.
3. **Slots Collection**
    - **Purpose:** Tracks individual parking slots and their status.
    - **Fields:**
        - slot_number (string or integer) – Unique identifier per lot
        - type (string) – e.g., compact, large, EV charging
        - status (string) – available, occupied, or reserved
    - **Notes:**
        - Slot status is updated in real-time when bookings are made or cancelled.

        - Supports future enhancements like slot sensor integration.
4. **Bookings Collection**
    - **Purpose:** Stores all booking transactions made by users.
    - **Fields:**
        - user_id (reference to Users)
        - slot_id (reference to Slots)
        - lot_id (reference to ParkingLots)
        - booking_time (timestamp) – When the booking was made
        - status (string) – active, completed, cancelled
    - **Notes:**
        - Ensures that a slot cannot be double-booked at the same time.
        - Historical booking records are maintained for auditing and reporting.

5.2 Relationships Between Collections

- **ParkingLot → Slots (One-to-Many):** Each parking lot can have multiple parking slots.
- **User → Bookings (One-to-Many):** Each user can have multiple bookings over time.
- **Slot → Booking (One-to-One/Many):** Each slot can have only one active booking at a time, preventing conflicts.

6.Request & Response Examples

User Endpoints

1. Register

```
POST /auth/register
{
  "name": "Surendra Reddy",
  "email": "surendra@example.com",
  "password": "securePassword123"
}
```

Response:

```
{
  "message": "User registered successfully",
  "user_id": "64f1a8b9f2c4e56789abcd12"
}
```

2. **Login**

```
POST /auth/login
```

```
{
  "email": "surendra@example.com",
  "password": "securePassword123"
}
```

Response:

```
{
  "access_token": "eyJhbGciOiJIUzI1NiIsInR5cCI6IkpXVCJ9...",
  "token_type": "bearer"
}
```

3. **Get Profile**

```
GET /users/me
```

Response:

```
{
  "user_id": "64f1a8b9f2c4e56789abcd12",
  "name": "Surendra Reddy",
  "email": "surendra@example.com"
}
```

4. **Update Profile**

```
PUT /users/me/update
```

```
{
  "name": "Surendra Reddy Updated",
  "email": "surendra.new@example.com"
}
```

Response:

```
{
  "message": "Profile updated successfully"
}
```

5. **Delete Profile**

```
DELETE /users/me/delete
```

Response:

```
{
  "message": "User deleted successfully"
}
```

6. Change Password

```
PUT /users/change-password
{
  "old_password": "securePassword123",
  "new_password": "newPassword456"
}
```

Response:

```
{
  "message": "Password changed successfully"
}
```

7. Forgot Password

```
POST /users/forgot-password
{
  "email": "surendra@example.com"
}
```

Response:

```
{
  "message": "Password reset link sent to email"
}
```

8. Reset Password

```
POST /users/reset-password
{
  "token": "resetToken123",
  "new_password": "newPassword456"
}
```

Response:

```
{
  "message": "Password reset successfully"
}
```

9. Logout

```
POST /auth/logout
```

Response:

```
{
  "message": "Logged out successfully"
}
```

7. Parking Endpoints

1. View All Parking Lots

```
GET /parkings
```

Response:

```
[
  {
    "parking_id": "64f1a9f0f2c4e56789abcd34",
    "name": "Vijayawada Central Parking",
    "location": "M.G. Road, Vijayawada",
    "total_slots": 50,
    "available_slots": 30,
    "booked_slots": 20
  }
]
```

2. Get One Parking Lot

```
GET /parkings/64f1a9f0f2c4e56789abcd34
```

Response:

```
{
  "parking_id": "64f1a9f0f2c4e56789abcd34",
  "name": "Vijayawada Central Parking",
  "location": "M.G. Road, Vijayawada",
  "total_slots": 50,
  "available_slots": 30,
  "booked_slots": 20
}
```

3. Show All Slots in Lot

```
GET /parkings/64f1a9f0f2c4e56789abcd34/slots
```

Response:

```
[
  {"slot_id": "S1", "slot_number": "A1", "vehicle_type": "Car",
"status": "Available"},
 {"slot_id": "S2", "slot_number": "A2", "vehicle_type": "Car", "status":
"Booked"}
]
```

4. Show Only Available Slots

```
GET /parkings/64f1a9f0f2c4e56789abcd34/slots/available
```

Response:

```
[
  {"slot_id": "S1", "slot_number": "A1", "vehicle_type": "Car",
"status": "Available"}
]
```

---

8. Booking Endpoints

1. Create Booking

```
POST /book
{
  "user_id": "64f1a8b9f2c4e56789abcd12",
  "parking_id": "64f1a9f0f2c4e56789abcd34",
  "slot_id": "S1",
  "vehicle_type": "Car",

  "start_time": "2025-09-05T10:00:00",
  "end_time": "2025-09-05T12:00:00"
}
```

Response:

```
{
  "message": "Booking confirmed",
  "booking_id": "B1",
  "status": "Booked"
}
```

2. Get All Bookings for Logged-in User

```
GET /bookings
```

Response:

```
[
  {
    "booking_id": "B1",
    "parking_id": "64f1a9f0f2c4e56789abcd34",
    "slot_id": "S1",
    "vehicle_type": "Car",
    "start_time": "2025-09-05T10:00:00",
    "end_time": "2025-09-05T12:00:00",
    "status": "Booked"
  }
]
```

3. Get Booking Details

```
GET /bookings/B1
```

Response:

```
{
  "booking_id": "B1",
  "parking_id": "64f1a9f0f2c4e56789abcd34",
  "slot_id": "S1",
  "vehicle_type": "Car",
  "start_time": "2025-09-05T10:00:00",
  "end_time": "2025-09-05T12:00:00",
  "status": "Booked"
}
```

4. Modify Booking

```
PUT /bookings/B1/update
{
  "start_time": "2025-09-05T11:00:00",
  "end_time": "2025-09-05T13:00:00"
}
```

Response:

```
{
  "message": "Booking updated successfully",
  "booking_id": "B1"
}
```

5. Cancel Booking

```
DELETE /bookings/B1/delete
```

Response:

```
{
  "message": "Booking cancelled successfully"
}
```

9.Error Handling

The backend returns structured error responses for all endpoints using standard HTTP status codes and descriptive messages.

Error Type	Status Code	Example Response
Unauthorized / Invalid Token	401	{ "detail": "Invalid or expired token" }
Resource Not Found	404	{ "detail": "User / Parking lot / Booking not found" }
Bad Request / Validation Error	400	{ "detail": "Required field missing or invalid data" }
Conflict / Already Exists	409	{ "detail": "Email already registered / Slot already booked" }
Internal Server Error	500	{ "detail": "An unexpected error occurred" }
Password Errors	400	{ "detail": "Old password is incorrect / Reset token invalid or expired" }

Notes:

- All error responses follow a JSON format.
- Status codes indicate the type of failure.
- Errors ensure clear communication to the client without exposing sensitive backend details.

10. Conclusion & Future Enhancements

10.1 Summary

The Smart Parking Finder backend system provides a complete solution for managing parking lots, slots, and bookings. Key features include:

- **User Management:** Registration, login, profile management, password handling, and secure authentication using JWT.
- **Parking Lot Management:** Add, view, update, and delete parking lots with real-time slot information.
- **Slot Management:** Track and manage parking slots with availability status.
- **Booking Management:** Create, view, modify, and cancel bookings efficiently.
- **Error Handling:** Standardized error responses ensure reliable communication with clients.
- **Testing & Deployment:** Verified with Postman and deployable locally or on production servers with proper environment configuration.

This backend system is designed to be scalable, maintainable, and easy to integrate with frontend applications or mobile apps.

10.2 Possible Future Enhancements

1. **Payment Integration:**
    - Add payment gateways to allow users to pay for bookings online.
2. **Real-Time Slot Updates:**
    - Implement WebSocket or push notifications to reflect real-time slot availability.
3. **Booking Notifications:**
    - Send email or SMS notifications for booking confirmations, reminders, or cancellations.

4. **Admin Dashboard:**
    - Provide an admin panel for monitoring parking lot utilization, slot statistics, and user activity.
5. **Analytics & Reporting:**
    - Track usage trends, peak times, and revenue reports for better decision-making.
6. **Mobile App Integration:**
    - Develop a mobile application to allow users to check availability and book slots on the go.