# BUS BOOKING SYSTEM

**Sanskar Verma**

**IIT Indore, CSE**

**Move-in-Sync Project**

The Bus Booking System is a web-based application designed to facilitate the management and booking of bus seats. It caters to two distinct user roles: Admin and User, each with specific functionalities tailored to their needs. The system ensures a seamless experience for users to browse, book, and cancel bus seats while providing admins with robust tools to manage bus details efficiently.

## Functionalities

### Admin Capabilities

1. **Manage Bus Details:**
    - **Add, Update, and Delete Bus Information:** Admins can add new buses, update existing bus details, and delete buses from the system.
    - **Bus Details Management:** Admins can manage the following details:
        - Bus Name
        - Total Number of Seats
        - Current Occupancy
        - Available Days of Operation
        - Source City
        - Destination City
        - Distance
        - Estimated time
    - **Bus Route Creation:** Admins can create and manage bus routes, including defining source and destination points.

```python
# Route for managing buses
@app.route('/admin/manage_buses')
def manage_buses():
    if 'admin_id' in session:
        cursor.execute('SELECT * FROM bus')
        buses = cursor.fetchall()
        return render_template('admin_manage_buses.html', buses=buses)
    else:
        return render_template('admin_login.html')
```

### User Capabilities

1. **Browse Available Buses:**
    - Retrieve information on the number of buses available between specific source and destination points.
    - Display the distance and Estimated Time of Arrival (ETA) for each bus.
2. **Check Seat Availability:**
    - View the number of seats available on a specific bus.
    - Display seat availability using color codes.
3. **Seat Booking:**
    - Reserve a seat on a selected bus.
4. **Cancel Seat Booking:**
    - Cancel a previously booked seat on the bus.
    - View all current bookings and select any to cancel.

## Requirements

### System Requirements

- **Backend:** Flask (Python)
- **Frontend:** HTML, CSS, JavaScript
- **Database:** MySQL
- **Other Tools:** MySQL Workbench for database management

### Functional Requirements

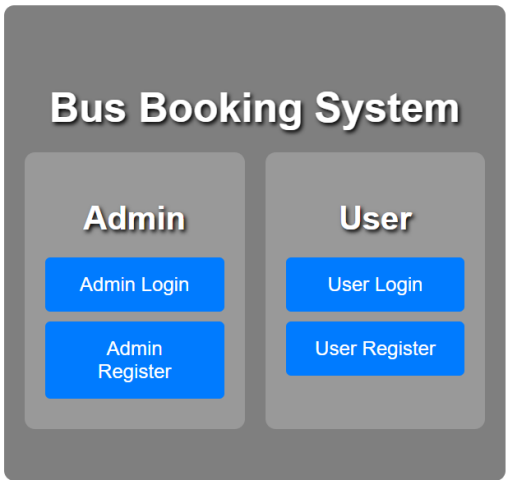
- User authentication and authorization to differentiate between admin and user roles.
- Admin dashboard to manage buses, routes, and seat plans.
- User dashboard to browse buses, check seat availability, book, and cancel seats.
- Concurrent seat booking prevention mechanism.
- Color-coded seat availability indicator.

### Non-Functional Requirements

- User-friendly interface for seamless navigation and interaction.
- Robust backend to handle concurrent user requests.
- Secure data handling and storage.
- Efficient data retrieval and processing to ensure quick response times.

## Unique Aspects of the Project

- **Role-Based Access & Authentication:** Distinct functionalities and interfaces for admin and user roles, ensuring a tailored experience for each type of user. Each has a secure registration and login process and the password is stored in a hashed format for security.




- **Effective Space and Time complexity:** The Bus Booking System has been designed with a focus on efficiency, leveraging optimal algorithms and data structures to ensure minimal time and space complexity.
  Time complexity is managed by utilizing efficient data retrieval and update operations, such as indexing key database fields to facilitate quick lookups and transactions. Primary and foreign keys have been effectively used to speed up transactions and keep them in sync.
  Space complexity is optimized through the careful normalization of database schemas by using normalised tables and avoiding redundancy, reducing redundancy and ensuring efficient storage of data. For instance, the usage of primary and foreign keys ensures that each piece of data is stored only once and referenced as needed, minimizing space requirements.

- **Real-Time Seat Availability:** Implementing a color-coded system to visually represent seat availability, enhancing user experience and decision-making.

  Green (60% Full or Less): If the occupancy percentage is 60% or less, display seats in green. Indicates that there is ample availability, and users can comfortably book their seats.

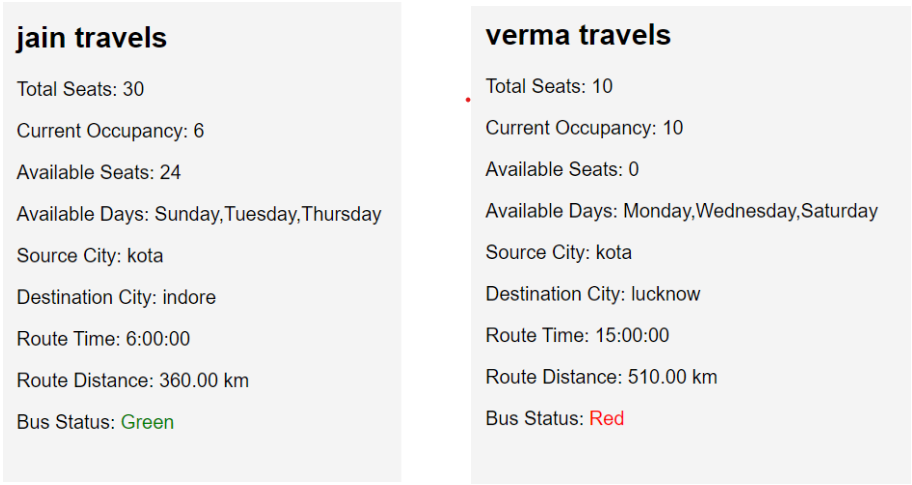
  Yellow (60% - 90% Full): If the occupancy percentage is between 60% and 90%, display seats in yellow. Serves as a cautionary indicator, informing users that seats are filling up but there is still moderate availability.

Red (90% - 100% Full): If the occupancy percentage is between 90% and 100%, display seats in red. Indicates limited availability, and users should act quickly to secure their seats.

```
<!-- Calculate occupancy percentage -->
        {% set occupancy_percentage = (bus[3] / bus[2]) * 100 %}

        <!-- Determine color based on occupancy percentage -->
        {% set bus_color = "green" %}
        {% if occupancy_percentage > 60 and occupancy_percentage < 90 %}
            {% set bus_color = "yellow" %}
        {% elif occupancy_percentage >= 90 %}
            {% set bus_color = "red" %}
        {% endif %}

        <!-- Display bus color -->
        <p>Bus Status: <span class="bus-color {{ bus_color }}">{{ bus_color |
capitalize }}</span></p>
```




- **Searching Buses and Caching:** In the routes table we have routes with all stops. While searching for the buses from point A to point B we use the following approach:
    1. Routes array contains all routes fetched from the database.
    2. Iterate on all the routes and for every route a. Check if both the points A and B are there in the stops. b. Point A is earlier than point B.
    3. If both the conditions are satisfied then add it to selectedRoutes. Till this step will be required only once in a day or two as new routes are not added frequently.
    4. Then iterate over the trips and any trip on that route should be shown on as the search result. The above algorithm takes significant time to execute and also the results for buses from Point A to point B would not change until some buses leave their start point.

So we will use caching to store the results in **Cache.** From then on, if a similar query comes, just remove trips whose startTime is earlier to current time and iterate for only new trips added. You already have the selected routes.

1. Use a key-value store to map unique identifiers to cached data and cache bus details using BusID as the key.
2. Cache frequently queried data such as:
    i. Bus details (BusID, BusName, TotalSeats, CurrentOccupancy, AvailableDays)
    ii. Seat availability for specific buses. Recently retrieved user-specific data.

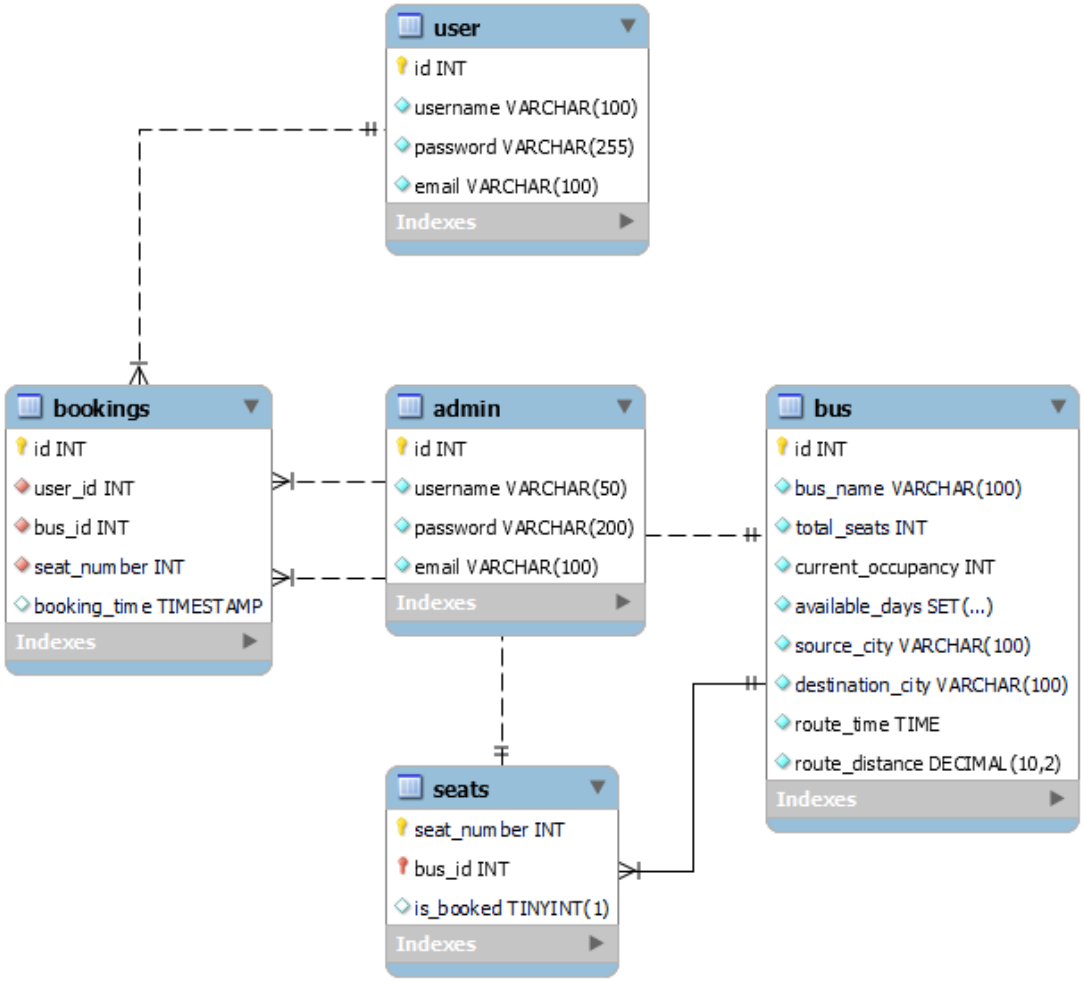
- **Concurrency Control:** Mechanisms to prevent concurrent booking of the same seat, ensuring data consistency and reliability. While the user is booking a particular seat the seat is locked for a period of 5 min to ensure that there is no race condition among users to book the same seat. In this time period no other user can select and book this ticket, while the one booking has to complete the procedure of adding details and payment. If a timeout occurs the transaction is rolled back, changes made reset and user, or any other user is free to book that ticket.

- **Handling System Case Failures:** To ensure the Bus Booking System is resilient to failures, we implement fault-tolerant mechanisms, including data replication and automated failover processes, which guarantee continuity in the event of system disruptions. Backup and recovery strategies are established, with regular automated backups stored in secure, geographically distributed locations to preserve data integrity and facilitate swift restoration. Comprehensive error recovery procedures are developed, incorporating real-time monitoring and alert systems to detect and address issues promptly. Through these measures, the system maintains high availability, data integrity, and seamless user experience even in the face of unexpected challenges.

- **Trade-Offs in the System:** One key trade-off involved choosing between a normalized database schema and denormalization. While normalization reduces data redundancy and optimizes storage, it can lead to complex queries and slower performance. To strike a balance, we opted for a moderately normalized schema that ensures efficient data storage while maintaining reasonable query performance. Another trade-off was between using a monolithic architecture versus a microservices approach. A monolithic architecture simplifies development and deployment but can become challenging to scale and maintain as the system grows. Conversely, a microservices architecture enhances scalability and maintainability by allowing independent deployment of services but adds complexity in terms of communication and data consistency. We opted for a modular monolithic approach, enabling easier management while allowing future transition to microservices if needed.

- **Robustness in Error Handling:** This framework includes comprehensive try-except blocks to catch and handle exceptions gracefully, providing meaningful error messages that facilitate effective debugging and user feedback. Additionally, error logging mechanisms capture detailed information about system anomalies, enabling prompt analysis and resolution. A centralized logging system records all exceptions and errors, capturing critical information such as timestamps, user actions, and system states. This log is invaluable for diagnosing issues, understanding usage patterns, and improving system reliability.

- **OOPs in low-level implementation and Database Details:**
  The database details in the form of tables that we have created for separate entities and they way of connecting them via unique relationships is summarised in the following ER diagram:




We implement an object-oriented approach to achieve the functionality for promoting reusability of code and improve code structure and understanding.

- User Table:
    - Stores user information such as username, password, and email.
    - Manages user bookings with functions for seat reservation and cancellation.
- Admin Table:
    - Holds login details and privileges for system administrators.
    - Includes functions to manage buses, such as adding, updating, and deleting.
- Bus Table:
    - Captures details about each bus, including name, route(source city and destination city), total seats, and schedule.
    - Tracks current occupancy and available days of operation.
- Booking Table:
    - Records seat reservations made by users, linking to specific users and buses.
    - Provides functions to create, retrieve, and cancel bookings.
- Seats Table:
    - Stores information about individual seats on buses, including seat number and location.
    - Manages seat assignment, booking status (booked or available), and visual color-coding based on occupancy levels (green, yellow, red).

➔ One-to-Many Relationships:

- One `User` can have many `Bookings`.
- One `Bus` can have many `Bookings` and many `Seats`.
- One `Route` can be associated with many `Buses`.

Python's OOP capabilities facilitate the implementation of core principles such as encapsulation, inheritance, and polymorphism, which are essential for creating maintainable and extensible codebases. Here's how these principles are applied in our project:

1. Encapsulation:
    - Bus Class: We encapsulate bus-related data such as bus name, total seats, current occupancy, and route within a `Bus` class. This class provides methods to access and modify these attributes, ensuring that the internal state of each bus object is protected and can only be modified through well-defined interfaces.

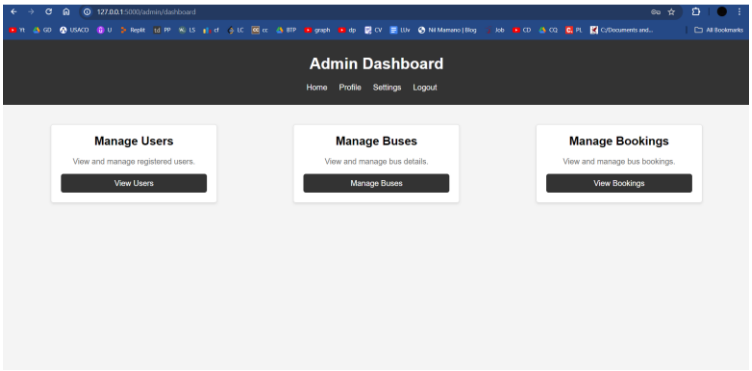
- User Class: Similarly, user-related information such as username, password, and booking history is encapsulated within a `User` class, with methods for authentication and managing bookings. This keeps user data secure and interaction with it controlled.

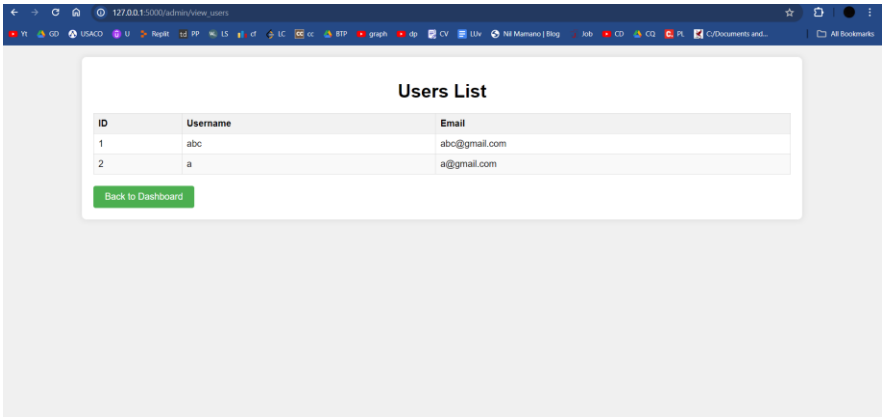
2. Inheritance:
   - Admin and Regular Users: We use inheritance to define specialized user roles. The `Admin` class inherits from the `User` class, adding additional methods for managing buses and routes, such as `add_bus()`, `update_bus()`, and `delete_bus()`. This reduces redundancy by allowing shared functionality to reside in the base `User` class while extending capabilities for the admin role.
   - Bus Types: For different types of buses (e.g., `LuxuryBus`, `EconomyBus`), inheritance allows us to create subclasses that inherit from a base `Bus` class but also have additional attributes or methods specific to their type, such as `extra_legroom()` for luxury buses.

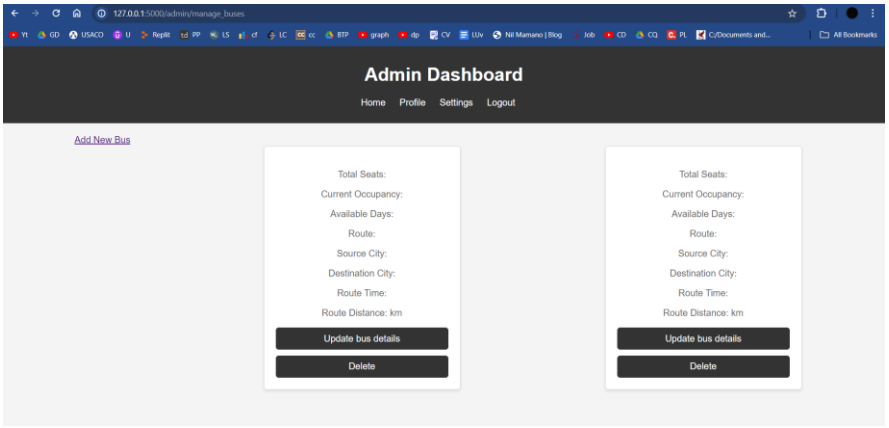
3. Polymorphism:
   - Seat Booking: The system uses polymorphism to handle different types of bus bookings through a common interface. For instance, both `LuxuryBus` and `EconomyBus` objects can use a common `book_seat()` method, but the implementation details may vary based on the bus type. This allows the booking process to be flexible and adaptable to different bus types without changing the core booking logic.
   - User Actions: The `perform_action()` method can be defined in the `User` class and overridden in the `Admin` class to handle different actions based on user roles. This method allows the system to execute different operations depending on whether the user is an admin or a regular user, without changing the method call structure.

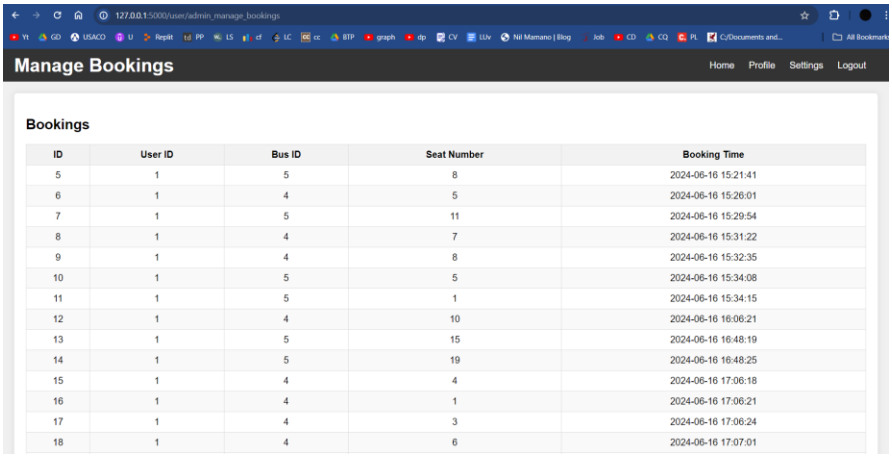
By adhering to these OOP principles, our system benefits from a clean and modular architecture that is easier to understand, maintain, and extend.

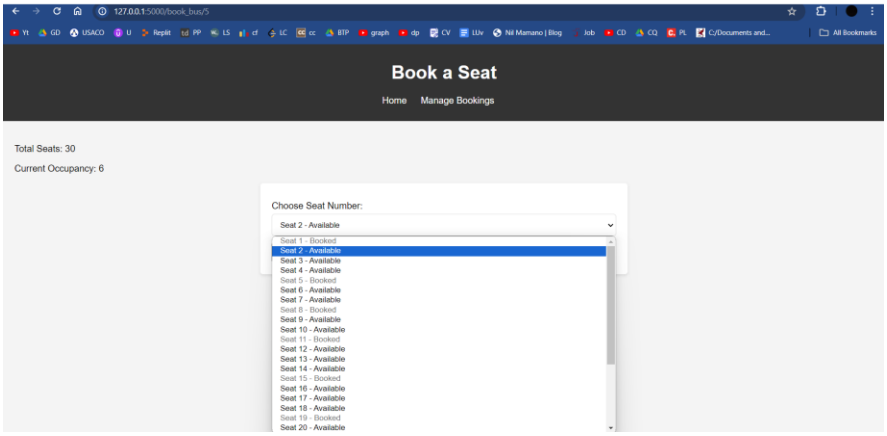
Attached below are few of the screenshots from our bus booking system, highlighting some of the key functionalities and simple user-friendly UI:

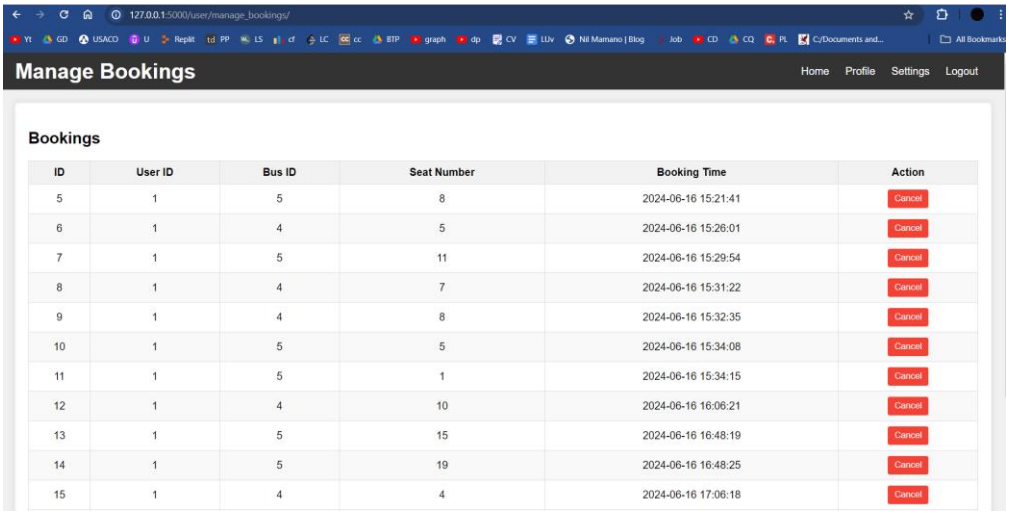

Admin Dashboard
Home Profile Settings Logout
Manage Users
View and manage registered users.
View Users
Manage Buses
View and manage bus details.
Manage Buses
Manage Bookings
View and manage bus bookings.
View Bookings

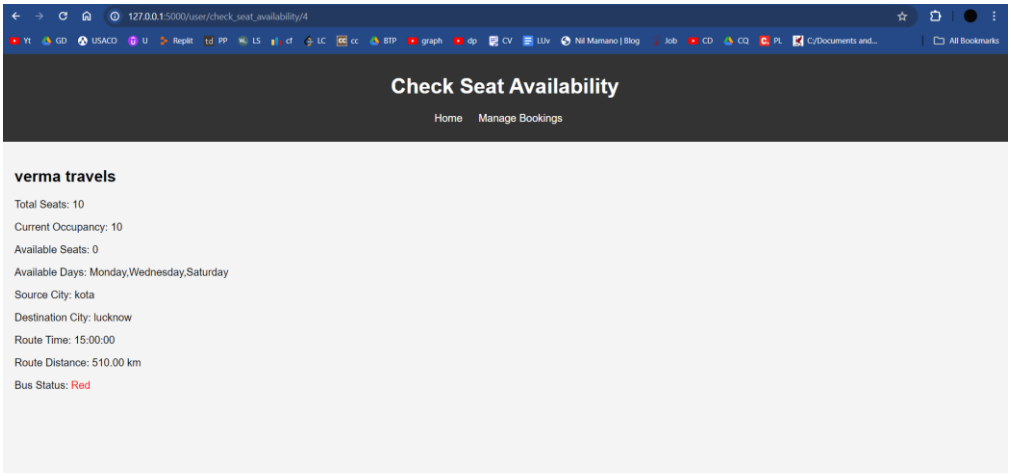

Users List
ID
Username
Email
1
abc
abc@gmail.com
2
a
a@gmail.com
Back to Dashboard


Admin Dashboard
Home Profile Settings Logout
Add New Bus
Total Seats:
Current Occupancy:
Available Days:
Route:
Source City:
Destination City:
Route Time:
Route Distance: km
Update bus details
Delete
Total Seats:
Current Occupancy:
Available Days:
Route:
Source City:
Destination City:
Route Time:
Route Distance: km
Update bus details
Delete


Manage Bookings
Home Profile Settings Logout
Bookings
ID
User ID
Bus ID
Seat Number
Booking Time
5 1 5 8 2024-06-16 15:21:41
6 1 4 5 2024-06-16 15:26:01
7 1 5 11 2024-06-16 15:29:54
8 1 4 7 2024-06-16 15:31:22
9 1 4 8 2024-06-16 15:32:35
10 1 5 5 2024-06-16 15:34:08
11 1 5 1 2024-06-16 15:34:15
12 1 4 10 2024-06-16 16:06:21
13 1 5 15 2024-06-16 16:48:19
14 1 5 19 2024-06-16 16:48:25
15 1 4 4 2024-06-16 17:06:18
16 1 4 1 2024-06-16 17:06:21
17 1 4 3 2024-06-16 17:06:24
18 1 4 6 2024-06-16 17:07:01


127.0.0.1:5000/book_bus/5
Book a Seat
Home
Manage Bookings
Total Seats: 30
Current Occupancy: 6
Choose Seat Number:
Seat 2 - Available
Seat 1 - Booked
Seat 2 - Available
Seat 3 - Available
Seat 4 - Available
Seat 5 - Booked
Seat 6 - Available
Seat 7 - Available
Seat 8 - Booked
Seat 9 - Available
Seat 10 - Available
Seat 11 - Booked
Seat 12 - Available
Seat 13 - Available
Seat 14 - Available
Seat 15 - Booked
Seat 16 - Available
Seat 17 - Available
Seat 18 - Available
Seat 19 - Booked
Seat 20 - Available


127.0.0.1:5000/user/manage_bookings/
Manage Bookings
Home
Profile
Settings
Logout
Bookings
ID
User ID
Bus ID
Seat Number
Booking Time
Action
5 1 5 8 2024-06-16 15:21:41 Cancel
6 1 4 5 2024-06-16 15:26:01 Cancel
7 1 5 11 2024-06-16 15:29:54 Cancel
8 1 4 7 2024-06-16 15:31:22 Cancel
9 1 4 8 2024-06-16 15:32:35 Cancel
10 1 5 5 2024-06-16 15:34:08 Cancel
11 1 5 1 2024-06-16 15:34:15 Cancel
12 1 4 10 2024-06-16 16:06:21 Cancel
13 1 5 15 2024-06-16 16:48:19 Cancel
14 1 5 19 2024-06-16 16:48:25 Cancel
15 1 4 4 2024-06-16 17:06:18 Cancel


127.0.0.1:5000/user/check_seat_availability/4
Check Seat Availability
Home
Manage Bookings
verma travels
Total Seats: 10
Current Occupancy: 10
Available Seats: 0
Available Days: Monday,Wednesday,Saturday
Source City: kota
Destination City: lucknow
Route Time: 15:00:00
Route Distance: 510.00 km
Bus Status: Red