UE21CS352B - Object Oriented Analysis & Design using Java

Mini Project Report

Travel Management System

Submitted by:

T P Shriambhikesh	*PES1UG21CS659*
Tanmay Praveen Udupa	*PES1UG21CS662*
Srimitravinda Ponnada	*PES1UG21CS621*
Suhas P Shroff	*PES1UG21CS634*

6th Semester K Section

Prof. Bhargavi Mokashi
Assistant Professor

January - May 2024

DEPARTMENT OF COMPUTER SCIENCE AND ENGINEERING
FACULTY OF ENGINEERING
PES UNIVERSITY
(Established under Karnataka Act No. 16 of 2013)
100ft Ring Road, Bengaluru – 560 085, Karnataka, India

1. PROBLEM STATEMENT

In the rapidly evolving landscape of travel and tourism, there is a growing need for a comprehensive Travel Management System (TMS) that integrates various booking functionalities and administrative tools to streamline the travel planning process. Our project aims to develop a robust and user-friendly TMS that caters to the diverse needs of travelers while offering seamless administrative capabilities for managing bookings and resources.

- Booking Functionalities:

  The TMS will offer a range of booking functionalities, including hotel reservations, car rentals, villa accommodations, and flight bookings. Each module will provide users with an intuitive interface to search, compare, and book services tailored to their preferences and requirements.

- Package Booking:

  In addition to individual bookings, the TMS will facilitate the creation and customization of travel packages. Users will have the option to bundle various services into comprehensive packages, allowing for convenient and cost-effective trip planning.

- User Management:

  The system will support user authentication and management features, enabling travelers to create accounts, manage bookings, and access personalized recommendations based on their travel history and preferences.

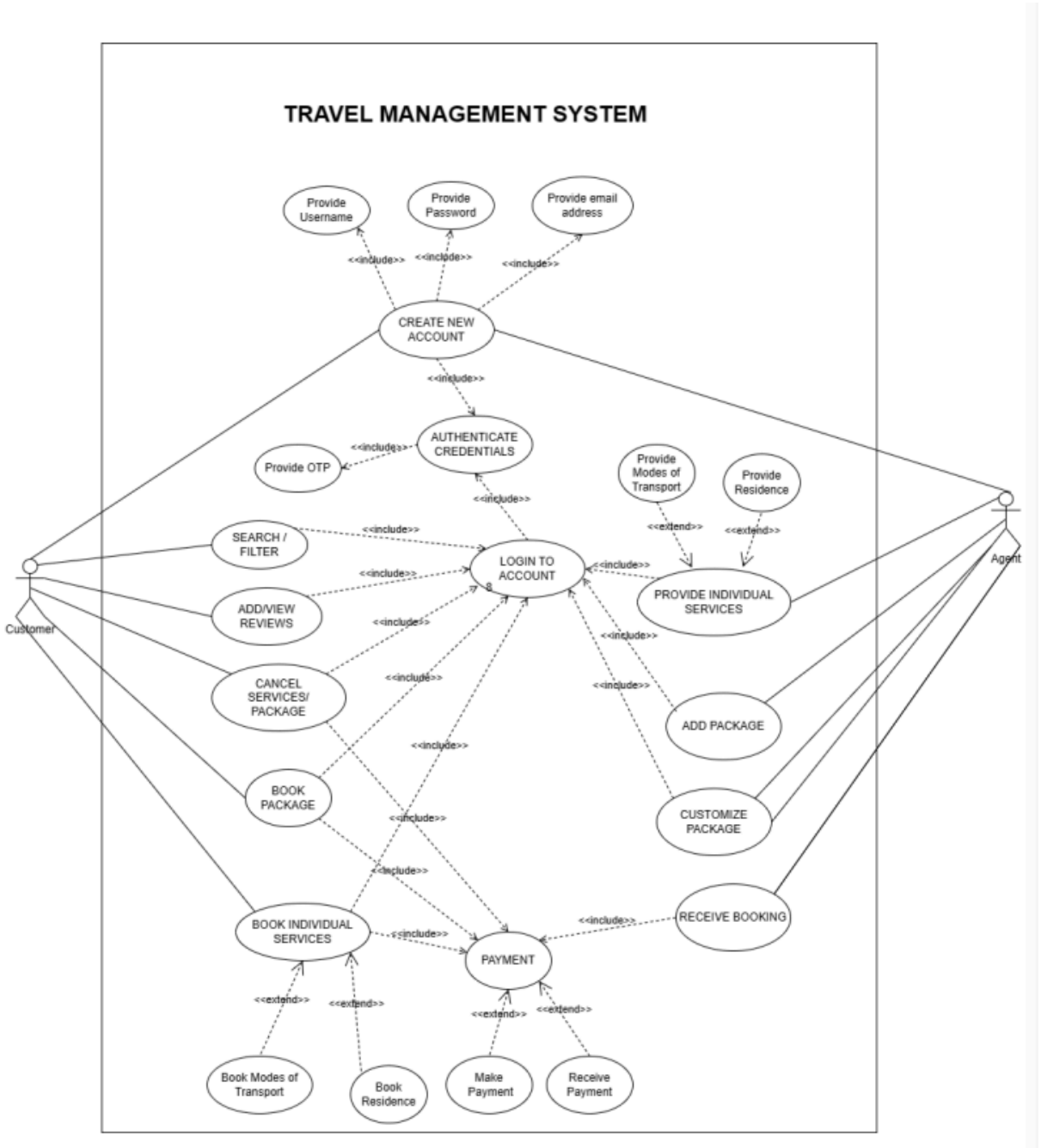
- Admin Dashboard:

  Administrators will have access to a dedicated dashboard equipped with tools for managing bookings, inventory, and user accounts. The admin interface will provide real-time insights into booking trends, revenue generation, and resource utilization, empowering administrators to make informed decisions and optimize operations.

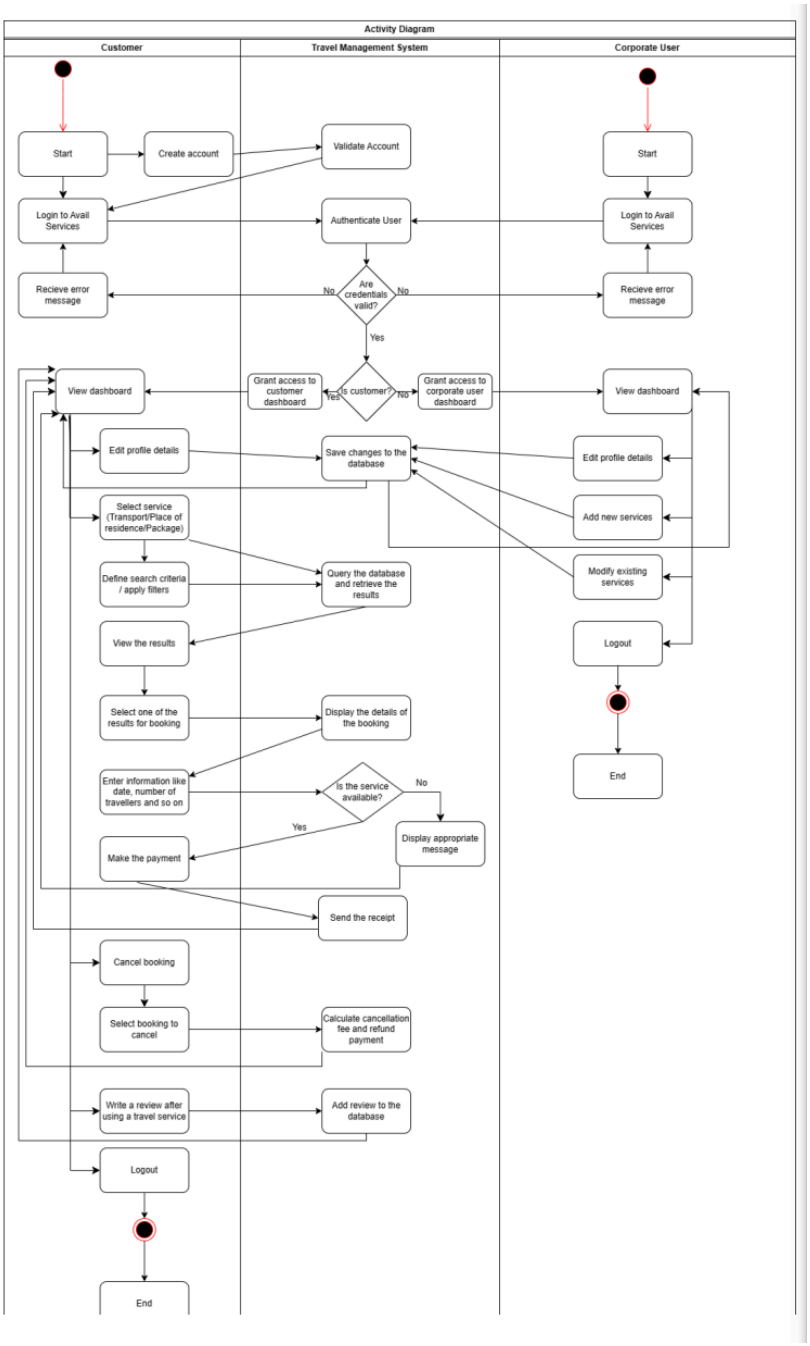
2. UML DIAGRAMS

Link to Diagrams: Travel Management System - Diagrams.pdf

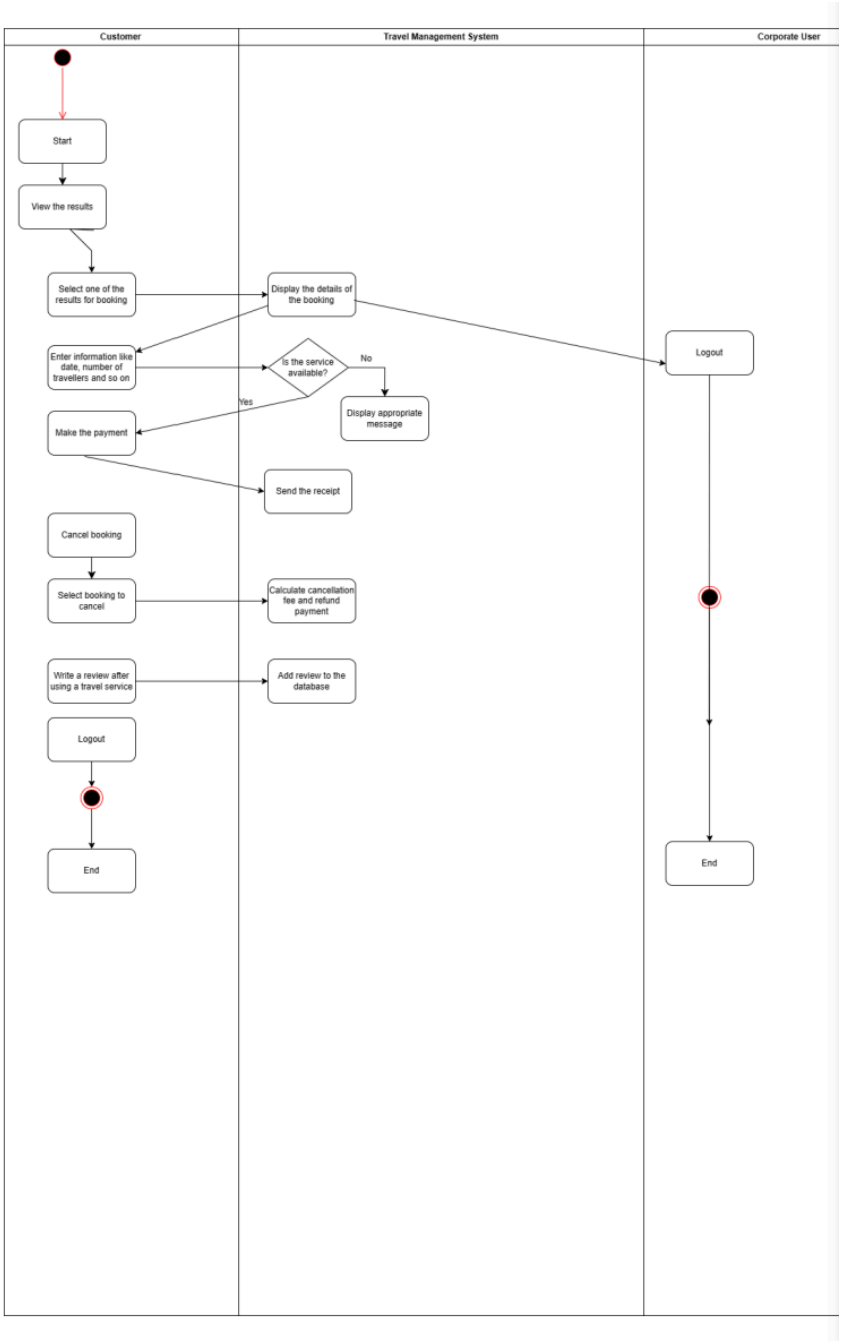
(a) Use Case Diagram



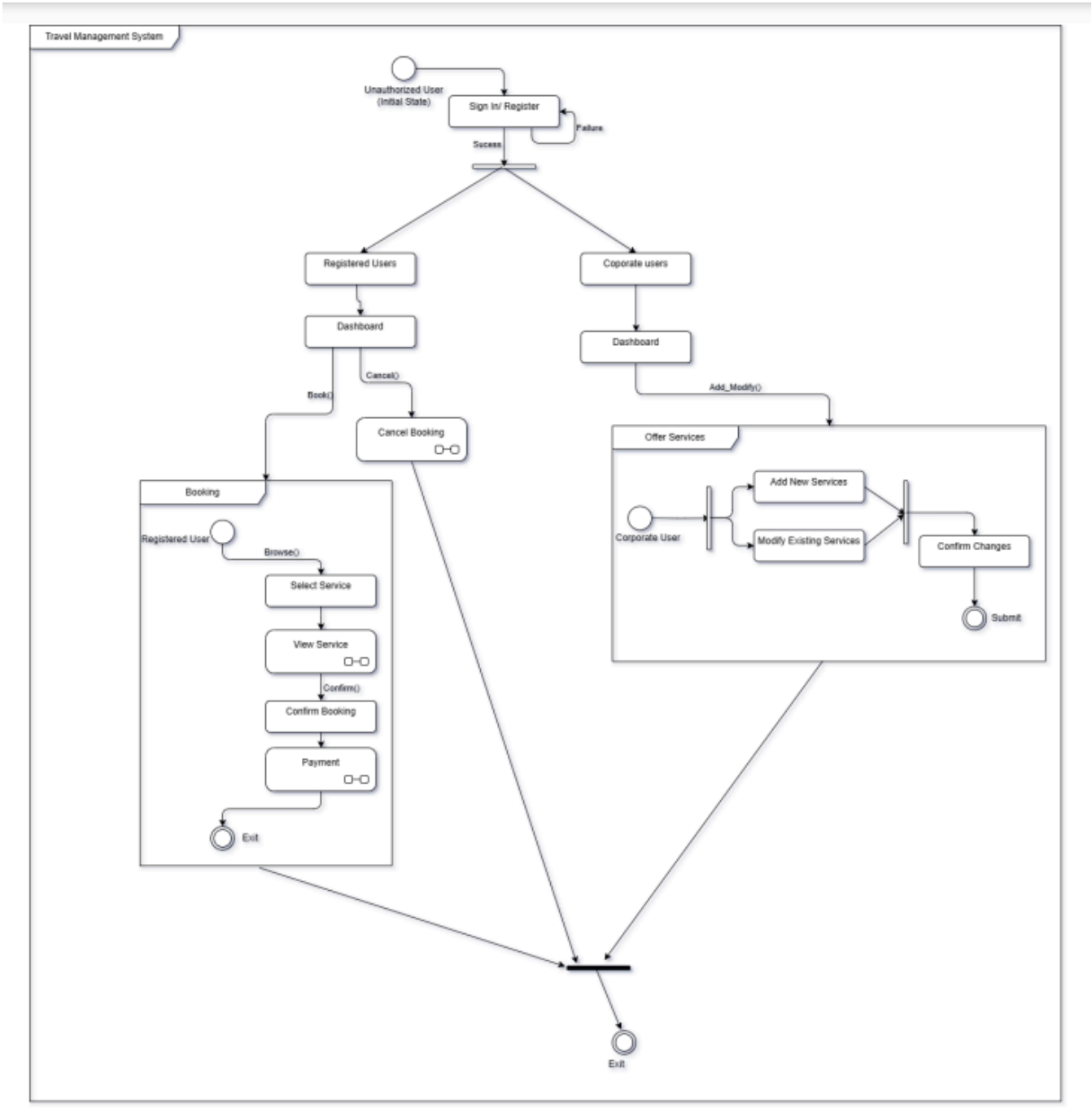
(b) Activity Diagram - 1



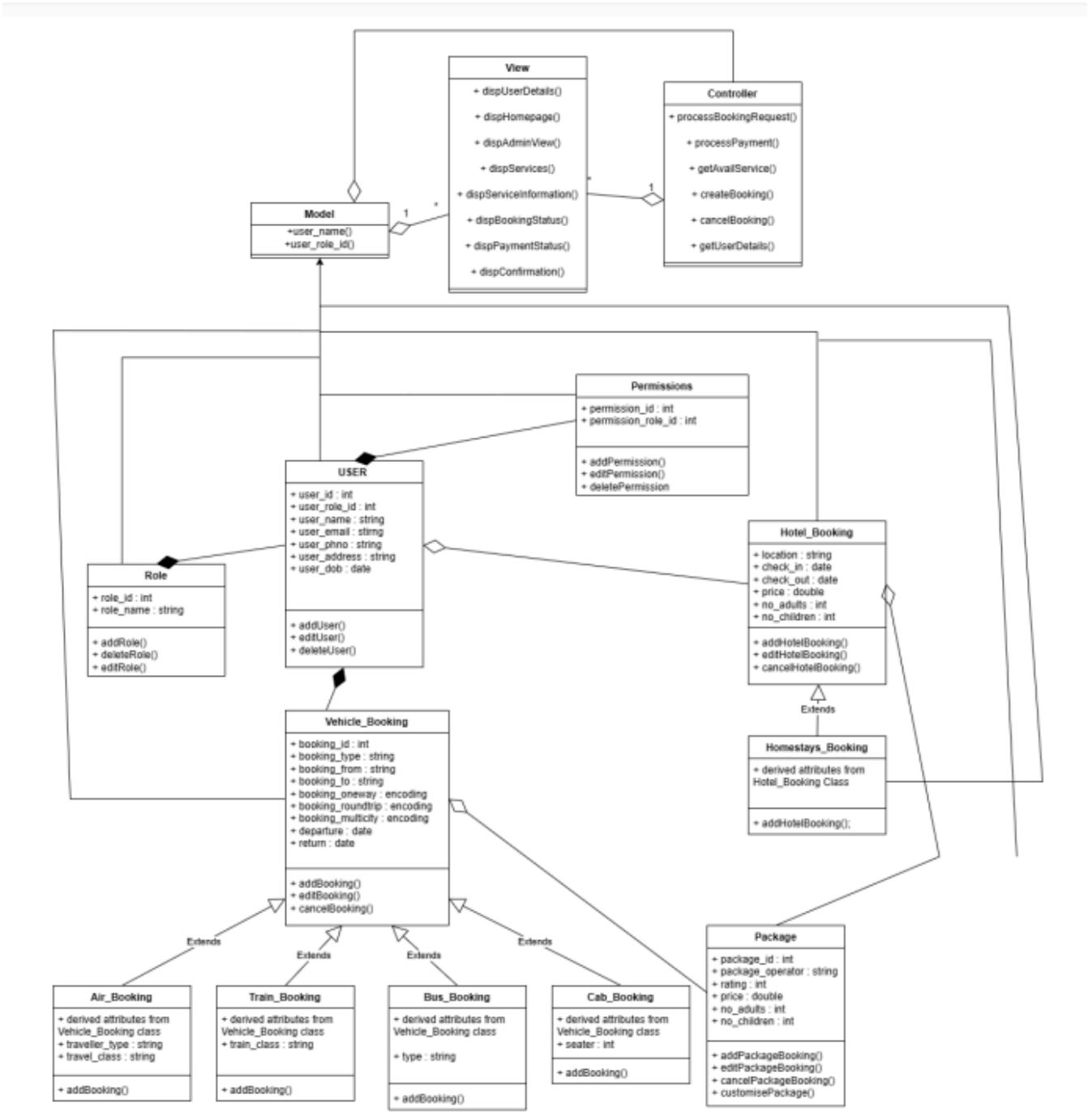
(c) Activity Diagram - 2 (Book Package)



(d) State Diagram



(e) Class Diagram



IMPLEMENTATION

Our project was developed using Vaadin framework

Our project does not violate SOLID principles

MVC architecture pattern is used in our project

- Model:

The Model component represents the data and business logic of the application. In our TMS, the Model includes classes that interact with the database to retrieve and manipulate travel-related data. For example, we have a FlightBooking model, HotelBooking Model classes responsible for managing bookings of flights, hotels respectively. These classes handle database operations such as querying for available flights, fetching hotel details and booking them.

- View

The View component represents the user interface of the application. In our TMS, the View includes the different pages and components that users interact with to search for and book travel services

- Controller

The Controller component acts as an intermediary between the Model and the View. It receives user inputs from the View, interacts with the Model to process requests, and updates the View with the results. For example, we have a Hotel Controller class that handles requests to book hotels. This class invokes methods on the corresponding Model classes to perform booking operations and renders appropriate View templates to display results or error messages.

3. DESIGN PATTERNS

- BUILDER PATTERN

The Builder pattern is utilized in the PackageBuilder class to simplify the creation of Package objects with optional components such as flights, trains, hotels, and villas. By encapsulating the construction logic, it offers a flexible and readable approach, allowing clients to specify only the necessary components for a Package. This promotes code clarity and maintainability, especially in scenarios where the object's structure may vary or evolve over time.

```
package com.example.application.builders;

import com.example.application.models.Package;
import com.example.application.models.Flight;
import com.example.application.models.Train;
import com.example.application.models.Hotel;
import com.example.application.models.Villa;

import java.util.List;

public class PackageBuilder {
   private Package packageInstance;

   public PackageBuilder() {
       packageInstance = null;
   }

   public PackageBuilder startNewPackage(String packageName) {
       packageInstance = new Package(packageName);
       return this;
   }
   public PackageBuilder setPackageName(String packageName) {
       packageInstance.setPackageName(packageName);
       return this;
   }

   public PackageBuilder setTotalPrice(double totalPrice) {
       packageInstance.setTotalPrice(totalPrice);
       return this;
   }

   public PackageBuilder addFlight(Flight flight) {
       packageInstance.addFlight(flight);
       return this;
   }

   public PackageBuilder addTrain(Train train) {
       packageInstance.addTrain(train);
       return this;
   }

   public PackageBuilder addHotel(Hotel hotel) {
       packageInstance.addHotel(hotel);
       return this;
   }

   public PackageBuilder setVilla(Villa villa) {
```

```
        packageInstance.setVilla(villa);
        return this;
    }

    public PackageBuilder addFlights(List<Flight> flights) {
        for (Flight flight : flights) {
            packageInstance.addFlight(flight);
        }
        return this;
    }

    public PackageBuilder addTrains(List<Train> trains) {
        for (Train train : trains) {
            packageInstance.addTrain(train);
        }
        return this;
    }

    public PackageBuilder addHotels(List<Hotel> hotels) {
        for (Hotel hotel : hotels) {
            packageInstance.addHotel(hotel);
        }
        return this;
    }

    public Package build() {
        Package builtPackage = packageInstance;
        packageInstance = null;
        return builtPackage;
    }
}
```

- SINGLETON PATTERN

The Singleton Pattern is employed to ensure that a single instance of the database connection is maintained across the application, preventing unnecessary resource consumption and ensuring consistency in data access. By encapsulating the creation and management of the database connection within a single class, it enhances code efficiency and promotes centralized control over database interactions.

The code for Singleton pattern can be found under the "database" folder and it's used everywhere, for eg, in controllers, managers etc.

```
public TrainController() {
    try {
        // Attempt to connect to MongoDB
        MongoClient connectedClient = MongoClients.create("mongodb://localhost:27017/");

        // Perform pre-flight checks and handle potential issues
        if (!preFlightChecks(connectedClient)) {
            throw new RuntimeException("Failed to connect to MongoDB during pre-flight
checks.");
        }

                            System.out.println("=>    Connection    successful:    "    +
preFlightChecks(connectedClient));
```

```
                                                                trainCollection          =
connectedClient.getDatabase("Travel_Management_System").getCollection("trains");
                                                              trainBookingCollection         =
connectedClient.getDatabase("Travel_Management_System").getCollection("trainBookings");
        // ... rest of the initialization logic using connectedClient
    } catch (MongoException e) {
        // Handle MongoException in case of connection issues
        throw new RuntimeException("Error connecting to MongoDB: " + e.getMessage());
    }
```

- STRATEGY PATTERN

This pattern is used because it easily allows to add new booking types without modifying the existing code. You simply need to create a new concrete BookingStrategy implementation and update the BookingContext to use the new strategy.

To implement this pattern, an interface named BookingStrategy is created that consists of the method book which is a common method for all booking types. Then concrete implementation classes are created for each booking type, such as TrainBookingStrategy, HotelBookingStrategy, etc.

There is a BookingContext class which will take a BookingStrategy object and delegate the booking operations to the appropriate concrete BookingStrategy implementation. In the respective booking views of the Train, Hotel, etc., after the booking details are entered, an instance of the BookingContext is created and is used to set the appropriate BookingStrategy.

The code for this can be found in the strategies folder.

```
package com.example.application.strategies;

import com.mongodb.MongoException;
import com.mongodb.client.MongoClient;
import com.mongodb.client.MongoClients;
import com.mongodb.client.MongoCollection;
import org.bson.Document;

import java.util.Map;

public class FlightBookingStrategy implements BookingStrategy {
    private MongoCollection<Document> flightBookingCollection;
    public FlightBookingStrategy() {
        try {
            // Attempt to connect to MongoDB
                                                        MongoClient     connectedClient     =
MongoClients.create("mongodb://localhost:27017/");

                                                                flightBookingCollection      =
connectedClient.getDatabase("Travel_Management_System").getCollection("flightBookings");
            // ... rest of the initialization logic using connectedClient
        } catch (MongoException e) {
            // Handle MongoException in case of connection issues
            throw new RuntimeException("Error connecting to MongoDB: " + e.getMessage());
```

```
        }


    }
    @Override
    public void book(Map<String, String> bookingDetails) {
        // Implement booking logic for flight
        Document flightBooking = new Document("username", bookingDetails.get("username"))
                .append("flight_number", bookingDetails.get("flight_number"))
                .append("airline", bookingDetails.get("airline"))
                .append("departure_date", bookingDetails.get("departure_date"))
                .append("arrival_date", bookingDetails.get("arrival_date"))
                .append("seats_booked", bookingDetails.get("seats_booked"))
                .append("total_price", bookingDetails.get("total_price"))
                .append("date_booked", bookingDetails.get("date_booked"))
                .append("paid", bookingDetails.get("paid"));

        flightBookingCollection.insertOne(flightBooking);
    }
}
```

- ADAPTER PATTERN

The Adapter pattern is employed to abstract the conversion of MongoDB documents of various types into Java's List format, providing a uniform interface for data retrieval and manipulation. By implementing adapter classes for each MongoDB document type, the pattern facilitates seamless integration with Java code, enhancing flexibility and maintainability.

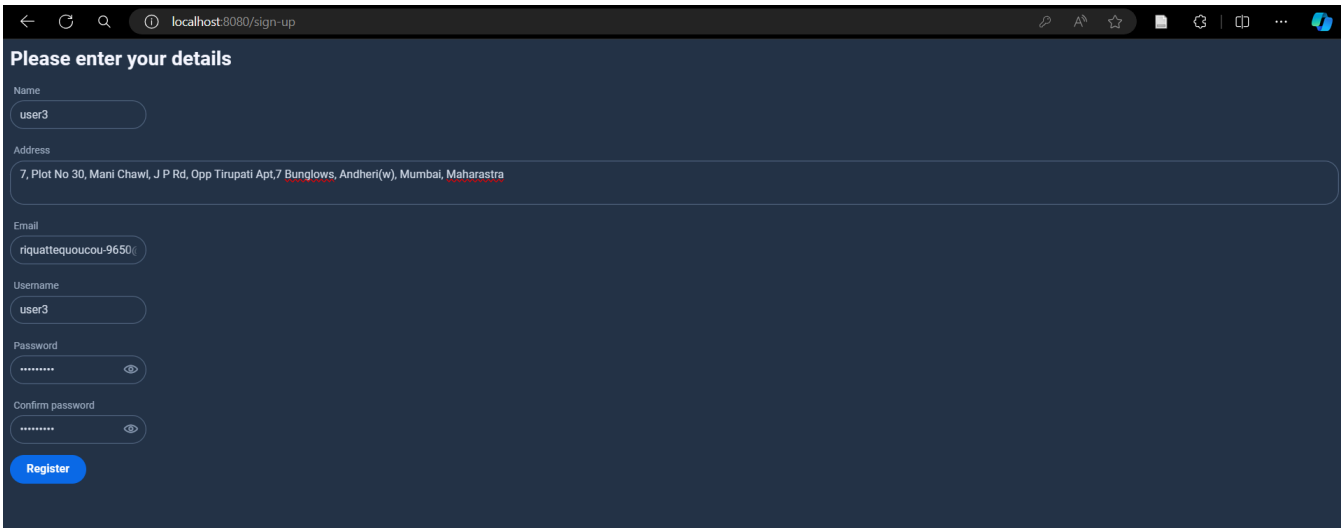
It follows OCP too, since new types of services can be added just by extending the "abstractAdapter" Class. without the need to change any of the existing code

```
package com.example.application.Adapters.Booking;

import com.example.application.Adapters.Booking.abstractBookingAdapter;
import com.example.application.models.FlightBooking;
import org.bson.Document;

public class FlightBookingAdapter extends abstractBookingAdapter {
    private String flightNumber;
    private String flightName;
    private String departureDate;
    private String arrivalDate;
    private String seatsBooked;

    public String getFlightNumber() {
        return flightNumber;
    }

    public void setFlightNumber(String flightNumber) {
        this.flightNumber = flightNumber;
    }

    public String getFlightName() {
        return flightName;
    }
```

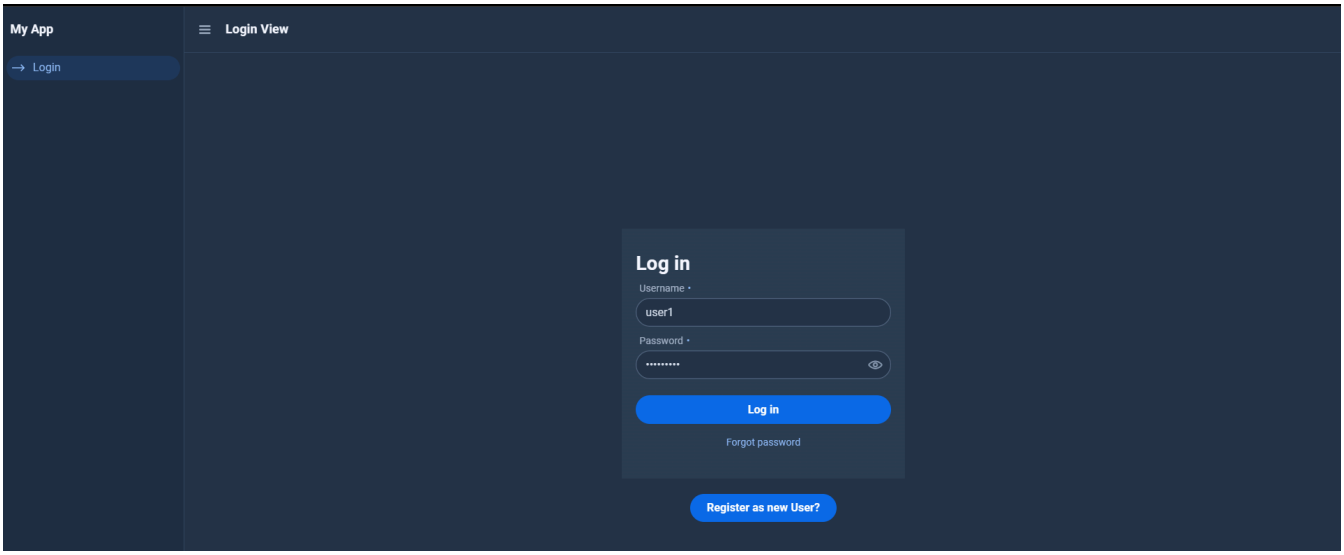
```
    public void setFlightName(String flightName) {
        this.flightName = flightName;
    }

    public String getDepartureDate() {
        return departureDate;
    }

    public void setDepartureDate(String departureDate) {
        this.departureDate = departureDate;
    }

    public String getArrivalDate() {
        return arrivalDate;
    }

    public void setArrivalDate(String arrivalDate) {
        this.arrivalDate = arrivalDate;
    }

    public String getSeatsBooked() {
        return seatsBooked;
    }

    public void setSeatsBooked(String seatsBooked) {
        this.seatsBooked = seatsBooked;
    }

    public FlightBooking toBooking(Document doc) {
        return new FlightBooking(
                doc.getObjectId("_id").toString(),
                doc.getString("username"),
                doc.getString("flight_number"),
                doc.getString("airline"),
                doc.getString("departure_date"),
                doc.getString("arrival_date"),
                doc.getString("seats_booked"),
                doc.getString("total_price"),
                doc.getString("date_booked"),
                doc.getString("paid")
        );
    }
}
```

SCREENSHOTS

Signup

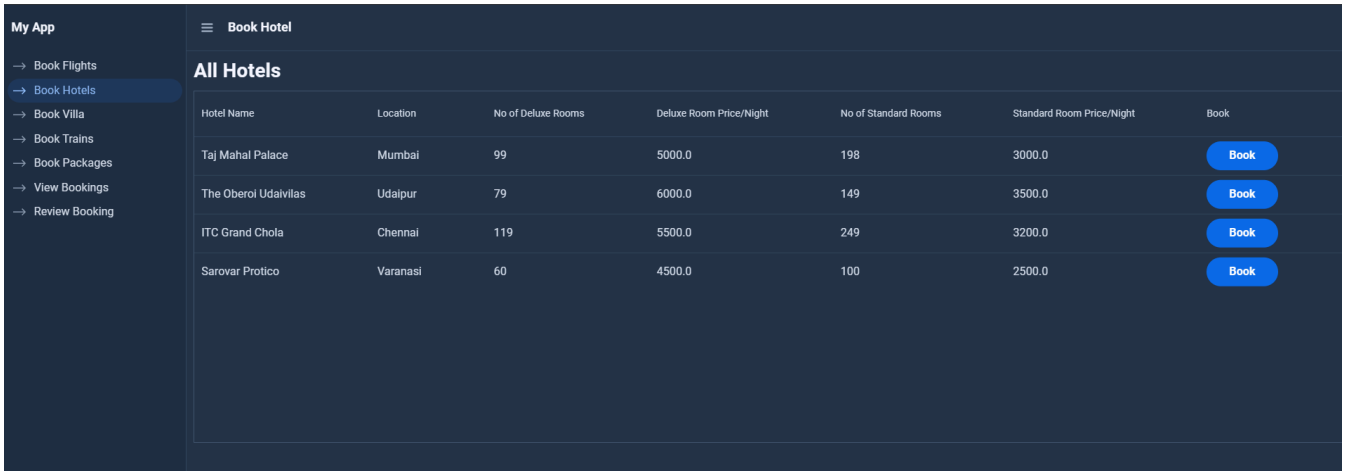


Login

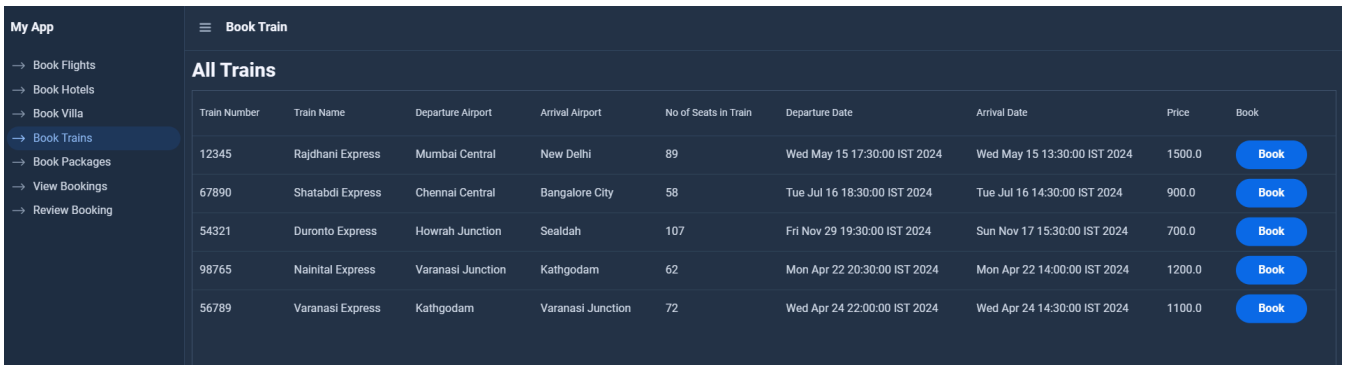


Book Flight

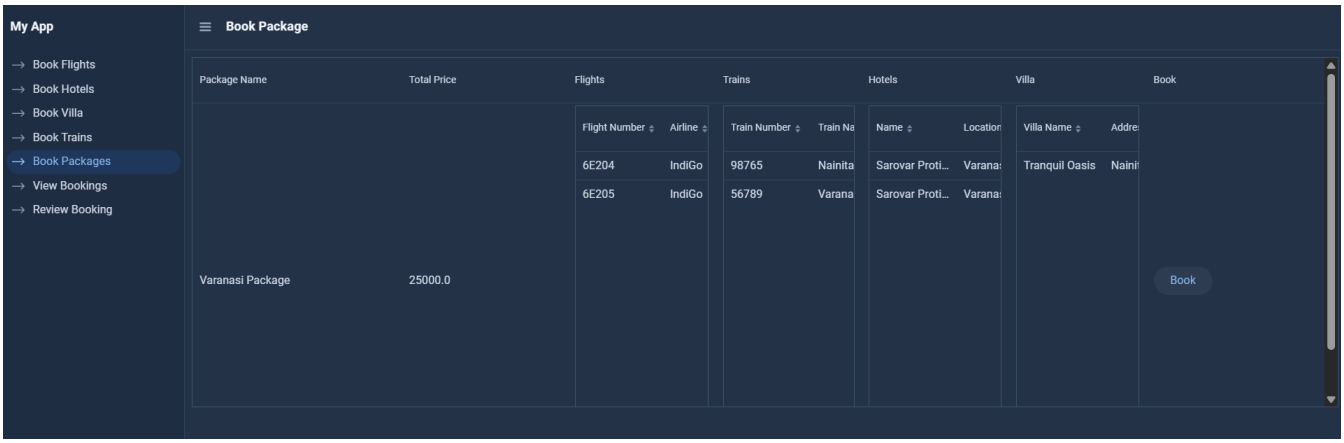
Book hotel



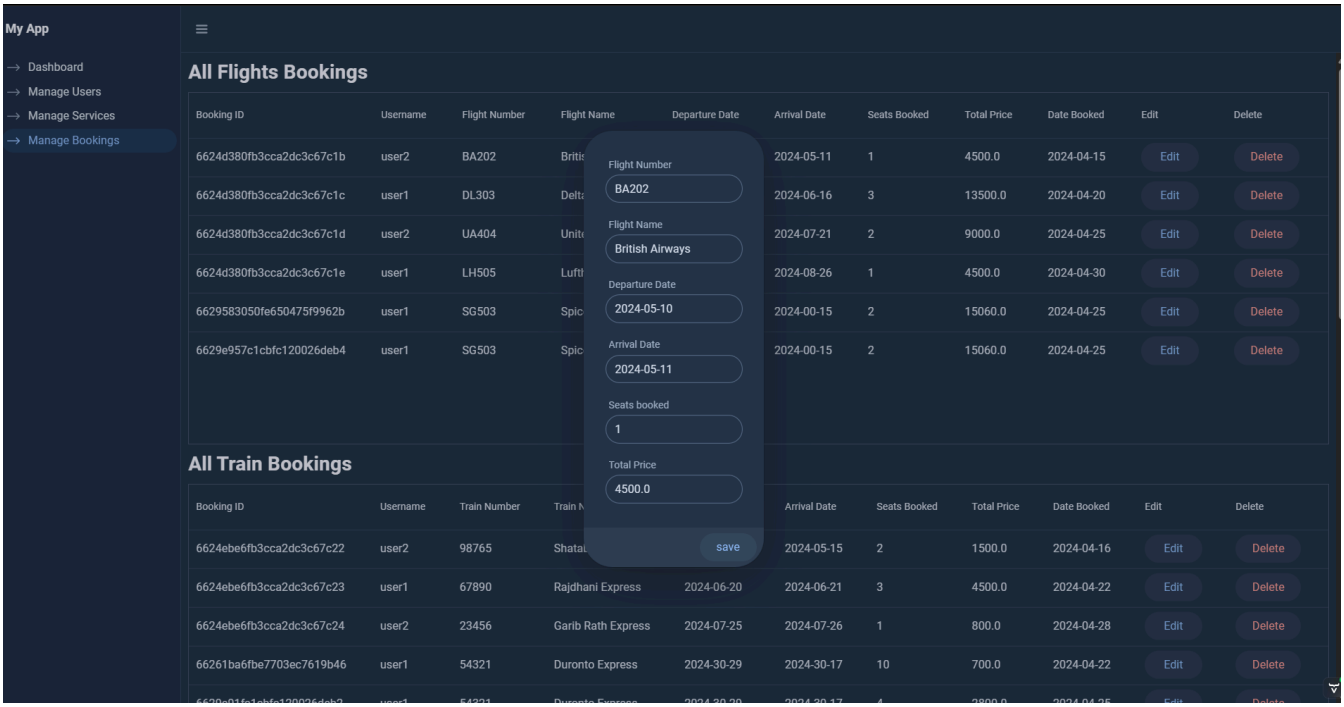
Book train



Book Package

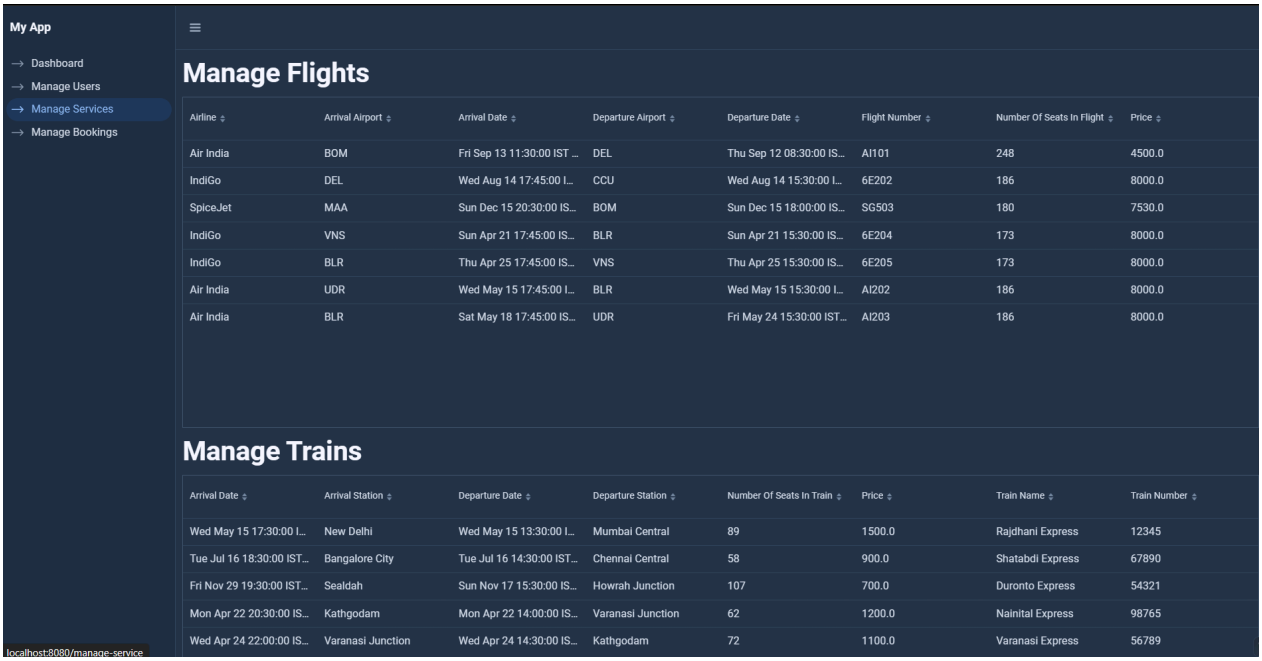


View bookings

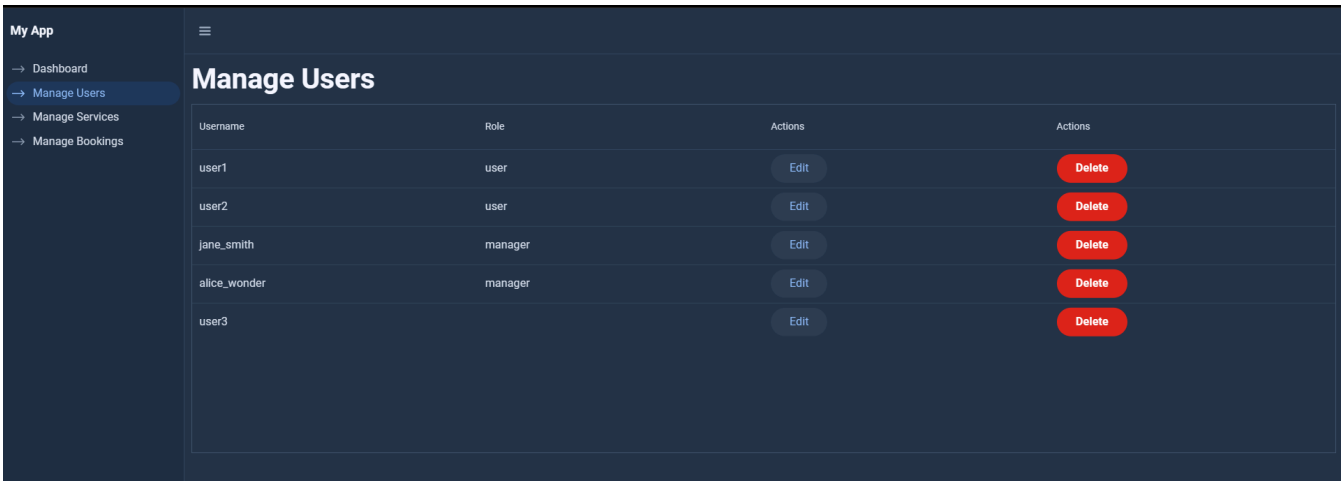


Admin dashboard

Manage services by admin



Manage users by admin



Review your bookings