UE22CS352B - Object Oriented Analysis & Design

Mini Project Report

Bus Reservation System

Submitted by:

Shilpa M Talawar	**PES1UG22CS559**
Simonna D costa	**PES1UG22CS594**
Sinchana AS	**PES1UG22CS595**
Spandana M Poojary	**PES1UG22CS605**

6th Semester J Section

Dr.Bhargavi Mokashi

January - May 2025

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

Problem Statement:

The existing manual bus reservation process is time-consuming, prone to human errors, and lacks real-time information on seat availability, schedules, and bookings. Passengers often face difficulties in securing seats, tracking bookings, or accessing updated bus schedules. Meanwhile, transport operators struggle with maintaining accurate records, avoiding double bookings, and handling cancellations efficiently.

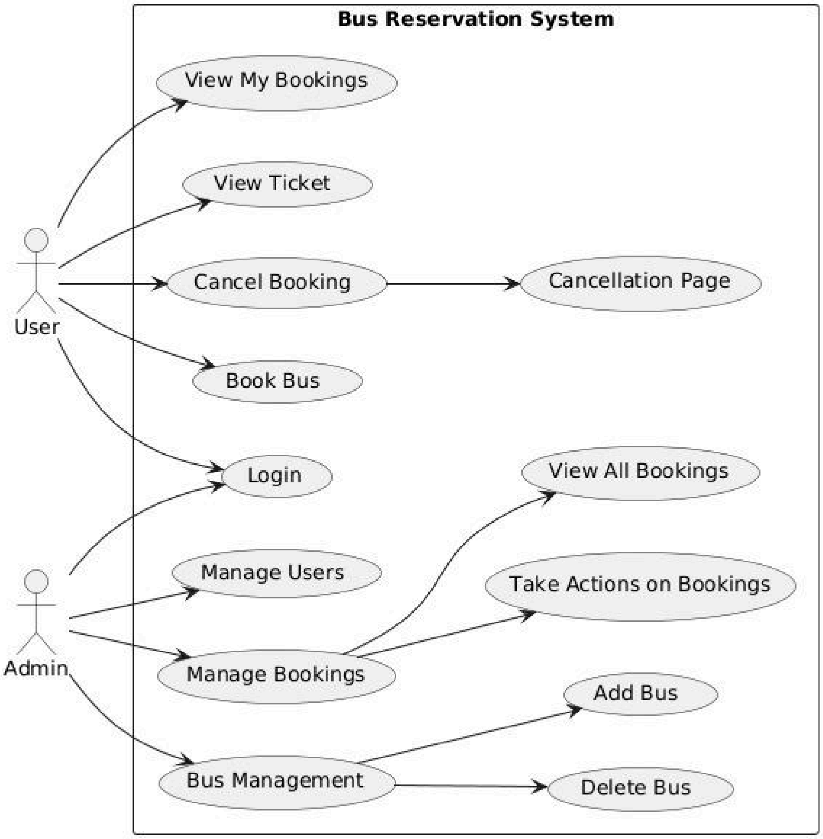
To address these challenges, there is a need for a digital Bus Reservation System that automates ticket booking, provides live seat tracking, and streamlines administrative tasks, ensuring a smoother experience for both passengers and transport providers.

Key Features:

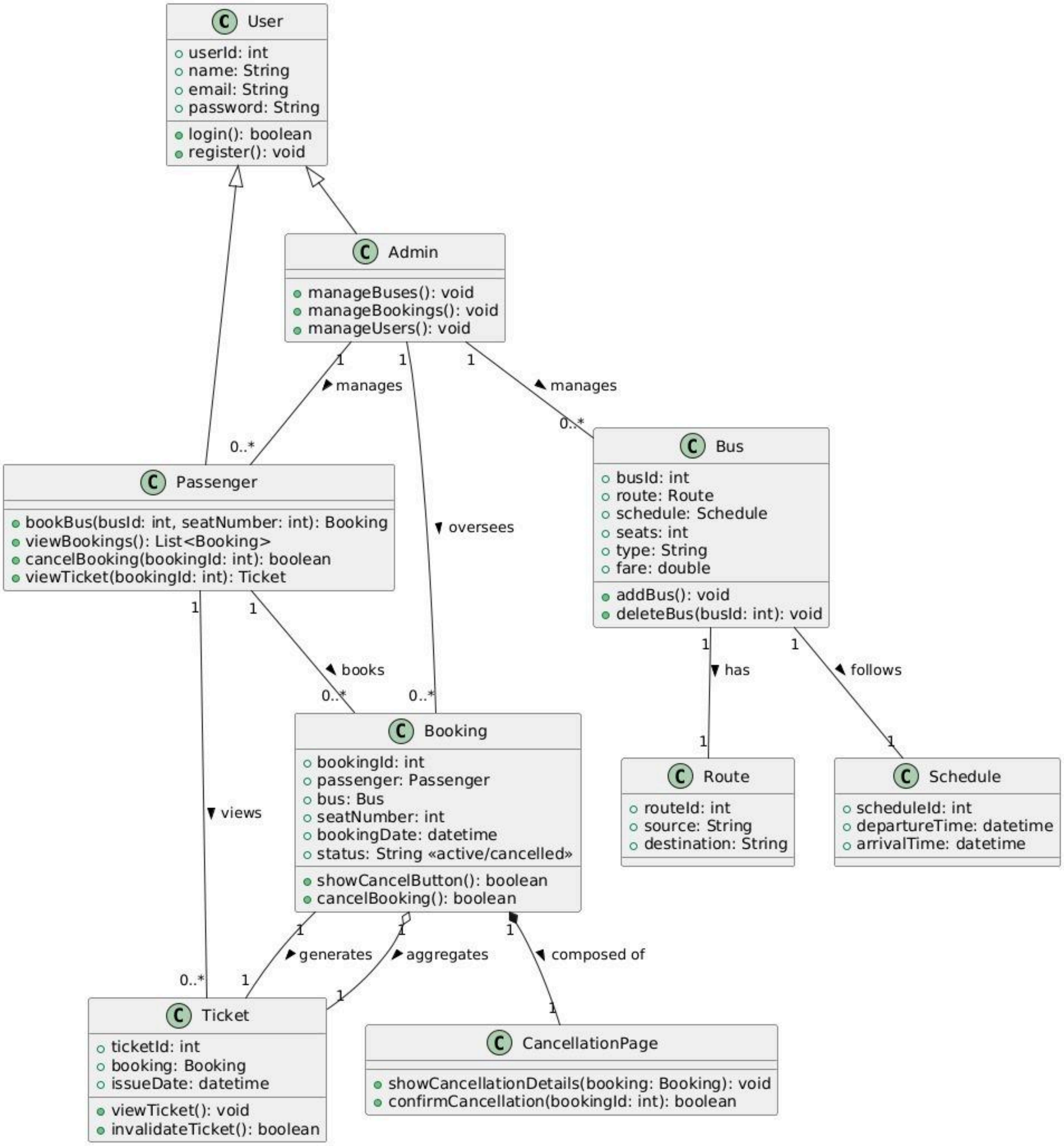
1. **User Registration & Login**
    - Allows passengers to create secure accounts using email and password.
    - Admin and user roles are separated with different access permissions.
2. **Online Ticket Booking**
    - Passengers can select seats and confirm bookings with a unique ticket ID.
    - Instant confirmation message with trip, seat, and fare details.
3. **Ticket Cancellation**
    - Users can cancel bookings before departure and get seat availability updated.
    - Admins can set cancellation policies like deadlines and refund percentages.
4. **Booking History**
    - Displays a list of all previous and upcoming bookings for the user.
    - Useful for rebooking and tracking travel history.
5. **Admin Dashboard**
    - Admins can add/edit/delete buses, routes, and schedules easily.
    - View all bookings, daily reports, and passenger data for management.

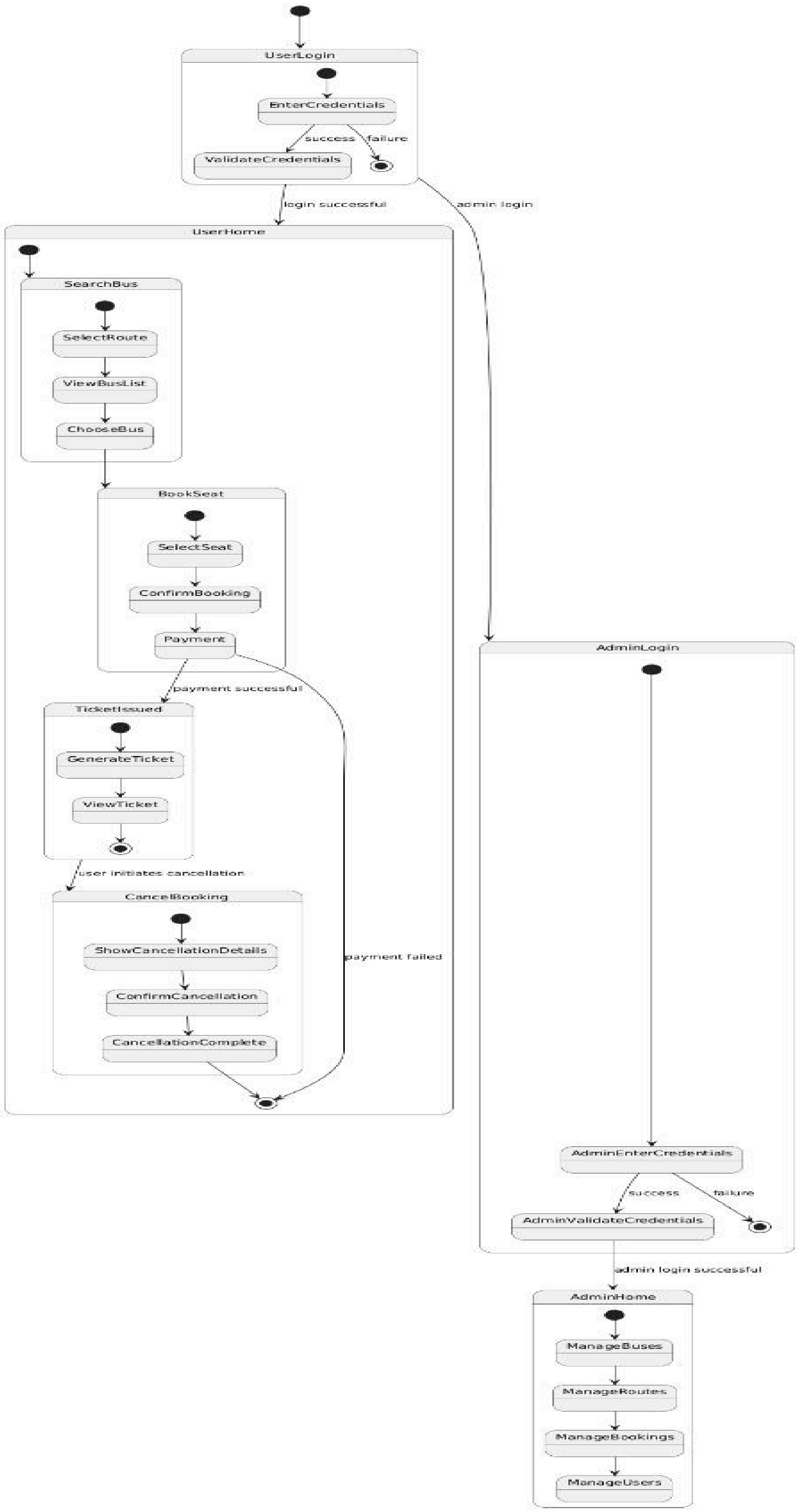
Models:

Use Case Diagram:



Class Diagram:

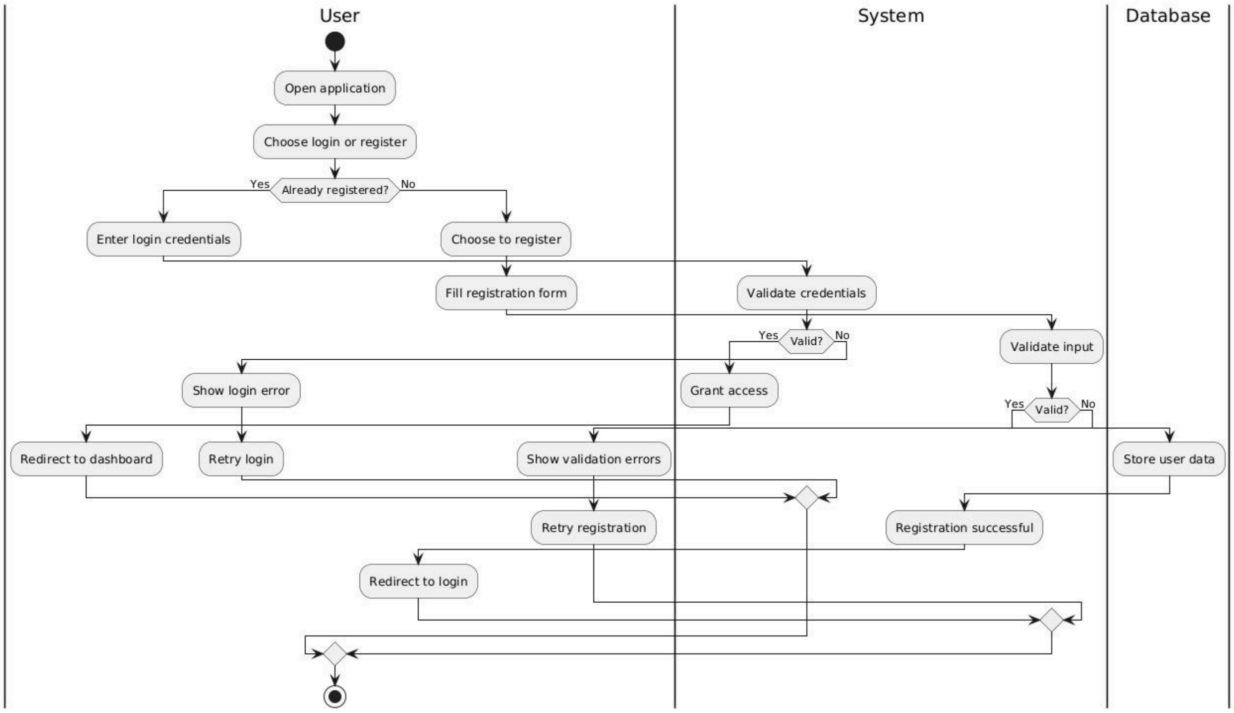




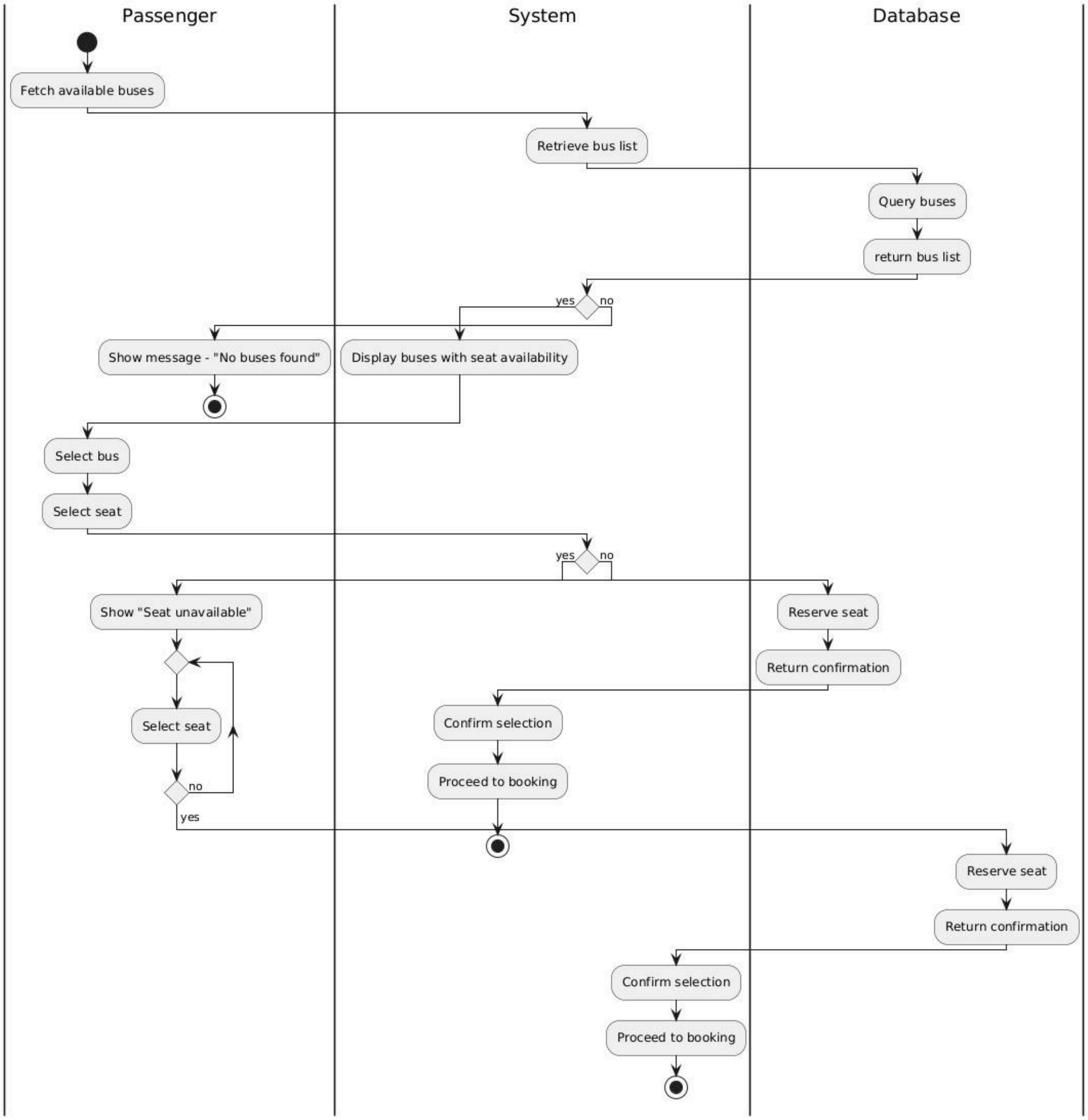
State Diagram:

Activity Diagrams:

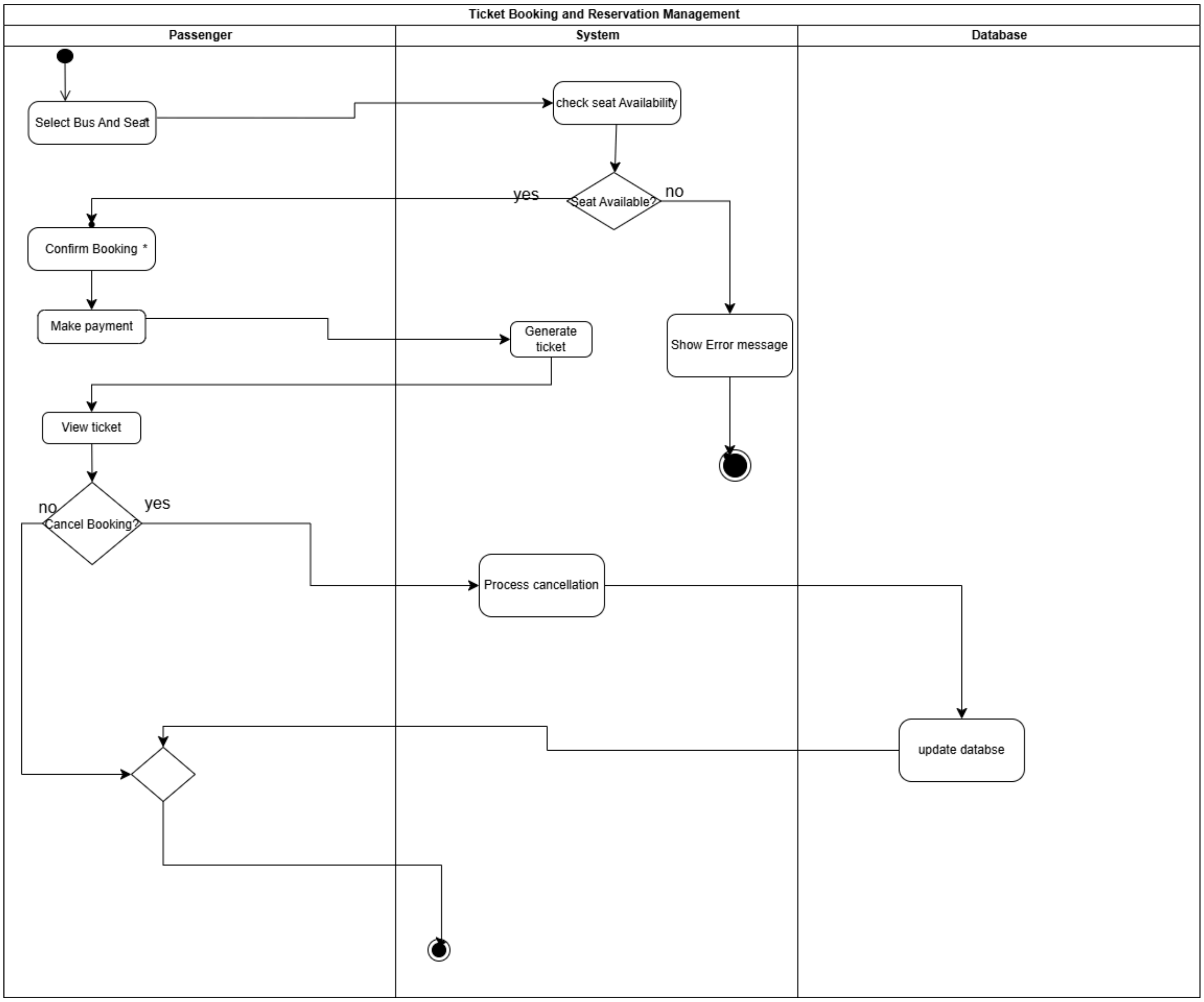
Login usecase:



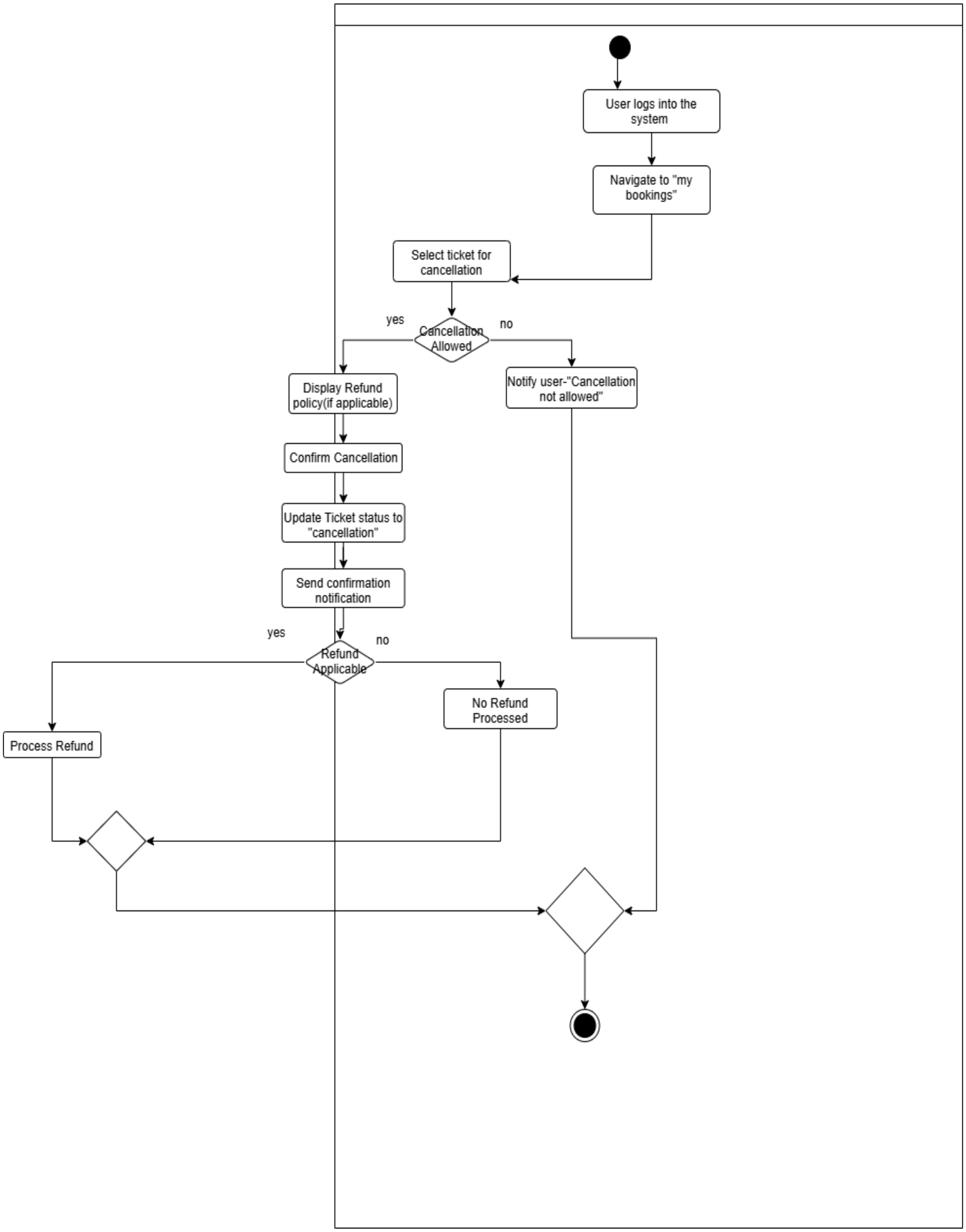
Bus and Seat Selection



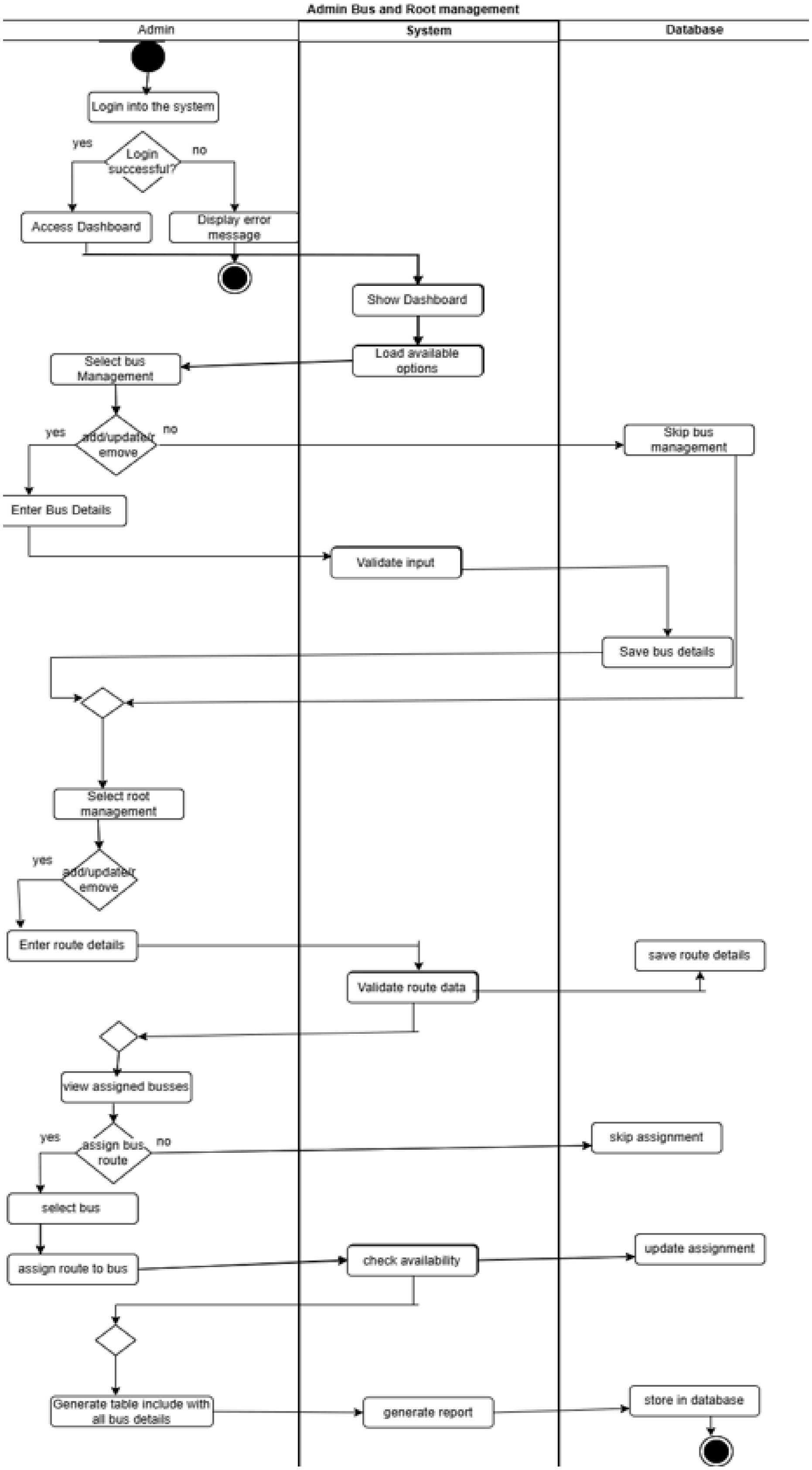
Ticket Booking and Reservation Management



Cancellation


User logs into the system
Navigate to "my bookings"
Select ticket for cancellation
yes
Cancellation Allowed
no
Display Refund policy(if applicable)
Notify user-"Cancellation not allowed"
Confirm Cancellation
Update Ticket status to "cancellation"
Send confirmation notification
yes
Refund Applicable
no
Process Refund
No Refund Processed

Bus and Route Management



Architecture Patterns

Model – View – Controller Pattern (MVC)

Model Layer:

The Model layer forms the data backbone of our Bus Reservation System, encapsulating all business entities and logic. Key components include Bus.java defining vehicle attributes (capacity, type), and Booking.java handling reservations with seat allocation and fare calculations. State patterns like BookingState.java and CancelledState.java dynamically manage booking lifecycles, while User.java and its subclass Passenger.java centralize authentication and profile data. This layer strictly follows domain-driven design principles, with repositories persisting data and services enforcing complex rules like dynamic pricing and cancellation policies.

Controller Layer:

Serving as the system's nervous system, the **Controller** layer processes all user interactions through RESTful endpoints. BookingController.java orchestrates the entire reservation workflow - from seat selection in BookingFormController.java to payment processing. AdminController.java and RouteController.java handle back-office operations like fleet management and route scheduling, while AutoController.java secures authentication flows. The @ControllerAdvice implements uniform exception handling across all endpoints. These controllers maintain thin logic, delegating complex operations to Model services while preparing data for Thymeleaf rendering.

View Layer:

The View layer's Thymeleaf templates create a responsive user experience across booking journeys. Interactive forms like booking-form.html collect passenger details with real-time validation, while bus_book.html renders visual seat maps using CSS grids. Administrative interfaces (admin_dashboard.html) display operational analytics through dynamic charts, and ticket.html generates print-ready boarding passes with QR codes. Error states gracefully degrade through booking-failed.html and error.html, while state transitions (e.g., cancellation flows in cancel_ticket.html) maintain user context. All templates share a consistent design system while adapting presentation logic based on Controller-provided model attributes.

Benefits of MVC in Bus Reservation System:

1.Separation of Concerns

- Clean division between data (Model), UI (View), and logic (Controller)
- Easier maintenance and updates

2.Parallel Development

- Multiple developers can work simultaneously
- Frontend/backend teams work independently

3.Reusability

- Same Model can power web, mobile, and API views
- Controllers handle similar requests uniformly

4.Scalability

- New features added without breaking existing code
- Supports growing business needs

5.Testability

- Each component tested in isolation
- Reduces bugs and improves reliability

Design Principles

GRASP (General Responsibility Assignment Software Patterns) principles:

1. Controller

Where: In the controllers package (e.g., AuthController, BusController).

How: Controllers handle incoming HTTP requests and delegate tasks to the appropriate service layer. They act as intermediaries between the user interface (API endpoints) and the business logic.

2. Creator

Where: In the services or repositories package.

How: Services or repositories are responsible for creating objects (e.g., creating entities like Bus, Route, or Ticket) because they are closely related to the objects they manage.

3. Information Expert

Where: In the models and repositories packages.

How: Models (e.g., Bus, Route, Ticket) encapsulate the data and provide methods to access or manipulate it.

Repositories (e.g., PassengerRepository, RouteRepository) are responsible for retrieving and storing data because they have the necessary knowledge about the database.

4. Low Coupling

Where: Across the entire project structure.

How: Dependency injection (via Spring Boot) ensures that components (e.g., controllers, services, repositories) are loosely coupled.

Each layer (controller, service, repository) has a clear responsibility, reducing dependencies between layers.

5. High Cohesion

Where: In the services and repositories packages.

How: Services (e.g., BusService, UserService) focus on business logic, ensuring that each class has a single, well-defined purpose.

Repositories focus solely on database operations, maintaining high cohesion within their scope.

6. Indirection

Where: In the services package.

How: Services act as intermediaries between controllers and repositories, ensuring that controllers do not directly interact with the database. This abstraction promotes flexibility and maintainability.

7. Polymorphism

Where: If interfaces or abstract classes are used (e.g., for services or repositories).

How: For example, if there are multiple implementations of a service interface (e.g., PaymentService with different payment methods), polymorphism allows the system to decide which implementation to use at runtime.

8. Protected Variations

Where: In configuration files and dependency injection.

How: Spring Boot's use of interfaces and dependency injection protects the system from changes in implementation details. For example, switching from one database to another (e.g., H2 to MySQL) requires minimal changes due to abstraction.

9. Pure Fabrication

Where: In the services and repositories packages.

How: Services and repositories are examples of pure fabrication because they are created to handle specific responsibilities (business logic and data access) that do not naturally belong to any domain object.

10. Information Hiding

Where: In the models and DTOs packages.

How: Models encapsulate their data and expose only necessary information through getters and setters.

DTOs (Data Transfer Objects) are used to hide internal model details when transferring data between layers.

SOLID PRINCIPLES:

1. Single Responsibility Principle (SRP)

The Single Responsibility Principle (SRP) is one of the SOLID principles of object oriented design. SRP states that a class should have only one reason to change, meaning that it should have only one responsibility or job within the system.

Importance of SRP

Maintainability:

- When a class has only one responsibility, it becomes easier to understand and maintain.
- We can change or extend the class without affecting other parts of the system.
- It reduces the complexity of code since there is less interdependency between components.

Testability:

Since a class focuses on a single responsibility, it's easier to write unit tests for it. Testing becomes more predictable and isolated because changes to one aspect of the class don't affect others.

Flexibility:

When the system grows and new features are added, SRP ensures that modifications are localized to one part of the codebase, reducing the risk of introducing bugs in unrelated areas.

Reusability:

By focusing on a single responsibility, classes become more reusable. You can take a class that implements one functionality and use it in different contexts without needing to modify it.

Reduced Coupling:

- When a class has only one responsibility, it typically has fewer dependencies on other parts of the system, which results in lower coupling between components. This makes it easier to modify or replace a class without affecting others.

How SRP is Applied in the Code

BookingController Class:

- The BookingController class has a single responsibility: handling incoming HTTP requests related to bookings and coordinating with the service layer. It doesn't contain business logic; instead, it delegates tasks to the appropriate service classes (e.g., BookingService, BusService, TicketService, UserService).

- The controller is not responsible for booking tickets, fetching bus details, or managing users. It only acts as a bridge between the user interface (view) and the underlying logic.

Service Classes:

- BookingService: This service is responsible for handling the business logic related to bookings. It manages tasks such as booking a ticket and retrieving booking data. It doesn’t concern itself with HTTP requests or user interactions.

- BusService: This class is responsible for managing bus-related operations, such as fetching details of available buses. Its responsibility is limited to operations related to buses.

- TicketService: This service focuses solely on the task of booking tickets. It handles the core functionality of reserving tickets based on the user's input.

- UserService: This service is responsible for managing user data, such as retrieving a user's information based on their ID. It doesn't handle booking logic or bus operations.

TicketForm and Booking Models:

- These model classes represent the data and are designed with a clear, single responsibility: to hold and validate booking or ticket information. They don’t perform any business logic or handle user requests. Instead, they serve as data carriers.

2. Open/Closed Principle (OCP)

Where: In services, repositories, and configuration classes.

How:

Classes are open for extension but closed for modification:

New features can be added by creating new classes or extending existing ones without modifying the core logic.

For example, adding a new payment method can be done by implementing a PaymentService interface without altering existing code.

Spring Boot's use of annotations like @Configuration and @Bean allows for extending functionality without modifying existing code.

3. Liskov Substitution Principle (LSP)

Where: In interfaces and their implementations.

How:

Subclasses or implementations of interfaces can replace their parent classes without breaking the application.

For example, if there is a UserService interface with multiple implementations (e.g., AdminUserService, CustomerUserService), any of these implementations can be used interchangeably.

4. Interface Segregation Principle (ISP)

Where: In service and repository interfaces.

How:

Interfaces are designed to be specific to the needs of their clients:

For example, a BusRepository interface might only define methods related to buses, while a RouteRepository interface defines methods for routes.

This ensures that classes do not implement methods they do not use.

5. Dependency Inversion Principle (DIP)

Where: Across all layers (controllers, services, repositories).

How:

High-level modules (e.g., controllers) do not depend on low-level modules (e.g., repositories); both depend on abstractions (e.g., interfaces).

Dependency injection (via Spring Boot) ensures that controllers depend on service interfaces, and services depend on repository interfaces, promoting loose coupling.

Examples of SOLID Principles in Action:

SRP:

A BusController class handles HTTP requests related to buses, while a BusService class handles the business logic for buses.

OCP:

Adding a new feature, such as a discount system, can be done by creating a new service class without modifying existing services.

LSP:

A PaymentService interface can have multiple implementations (e.g., CreditCardPaymentService, PayPalPaymentService), and any of these can be used without breaking the system.

ISP:

Separate repository interfaces for Bus, Route, and Passenger ensure that each interface is specific to its domain.

DIP:

Controllers depend on service interfaces, and services depend on repository interfaces, ensuring that the implementation details are abstracted.

Design Patterns

1.Facade Pattern (Login Use Case)

The Facade pattern provides a simplified interface to a set of interfaces in a subsystem. It defines a higher-level interface that makes the subsystem easier to user.

Importance:

- **Simplifies Client Interaction:** It hides the complexity of the underlying login subsystem, providing a single, easy-to-use entry point for clients.
- **Reduces Dependencies:** Clients only need to interact with the facade, shielding them from the internal components of the login subsystem and reducing coupling.

Drawbacks:

- **Potential for a God Object:** If not designed carefully, the facade can become responsible for too many aspects of the subsystem.
- **May Not Expose All Functionality:** The simplified interface might not expose all the advanced features of the underlying subsystem.

How it helps in Bus Reservation System:

- It can provide a single loginUser(Username,password) method that internally handles complex tasks like user validation, authentication, session creation, and potentially logging.
- Different parts of the application (e.g., web interface, mobile app) can use the same simple facade for login, without needing to understand the intricacies of the authentication process.

2.Factory Pattern (Bus Booking Use Case)

The Factory pattern provides an interface for creating objects in a superclass but allows subclasses to alter the type of objects that will be created. This decouples the object creation logic from the client code.

Importance:

- **Decoupling:** The client code doesn't need to know the concrete bus types being created, making the system more flexible and less dependent on specific implementations.
- **Abstraction of Object Creation:** It centralizes the object creation logic, making it easier to manage and modify the instantiation process.

Drawbacks:

- **Increased Complexity:** Introducing new concrete product classes often requires adding a new creator subclass, potentially increasing the number of classes.
- **Potential for a Single Large Creator Class:** If not managed carefully, a single factory class can become overly complex.

How it helps in Bus Reservation System:

- It can be used to create different types of buses (e.g., AC, Non-AC, Sleeper) based on user preferences or availability without the booking logic needing to know the specifics of each bus type.

- Adding new bus types in the future becomes easier as you only need to create a new concrete bus class and potentially update the factory.

3.State Pattern (Cancellation Use Case)

The State pattern allows an object to alter its behavior when its internal state changes. This pattern encapsulates state-specific behavior into separate state objects and delegates requests to the current state object.

Importance:

- **Improved Organization:** State-specific logic is localized within state classes, making the code cleaner and easier to understand compared to having numerous conditional statements.
- **Open/Closed Principle:** New states can be added without modifying the context object, adhering to the open/closed principle.

Drawbacks:

- **Increased Number of Classes:** Introducing a new state requires creating a new state class.
- **Potential Complexity:** If the transitions between states are intricate, the number of state classes and their relationships can become complex.

How it helps in Bus Reservation System:

- It can manage the different states of a booking (e.g., booked, cancellation requested, cancelled, refunded), with each state handling cancellation-related actions differently.
- Implementing different cancellation policies (e.g., partial refund before a certain time) becomes easier by defining specific behavior within the corresponding state.

4.Behavioral Pattern

Command Pattern (Admin Use Case):

Behavioral patterns are concerned with algorithms and the assignment of responsibilities between objects. They describe not just the objects or classes but also the patterns of communication between them. For the admin use case, several behavioral patterns could be relevant depending on the specific functionalities.

Importance:

- **Decoupling of Invoker and Receiver:** The object that invokes the operation (e.g., an admin interface button) is decoupled from the object that knows how to perform it (e.g., a database update service).
- **Support for Undo/Redo and Logging:** Commands can be stored, allowing for

the implementation of undo/redo functionality and the logging of admin actions.

Drawbacks:

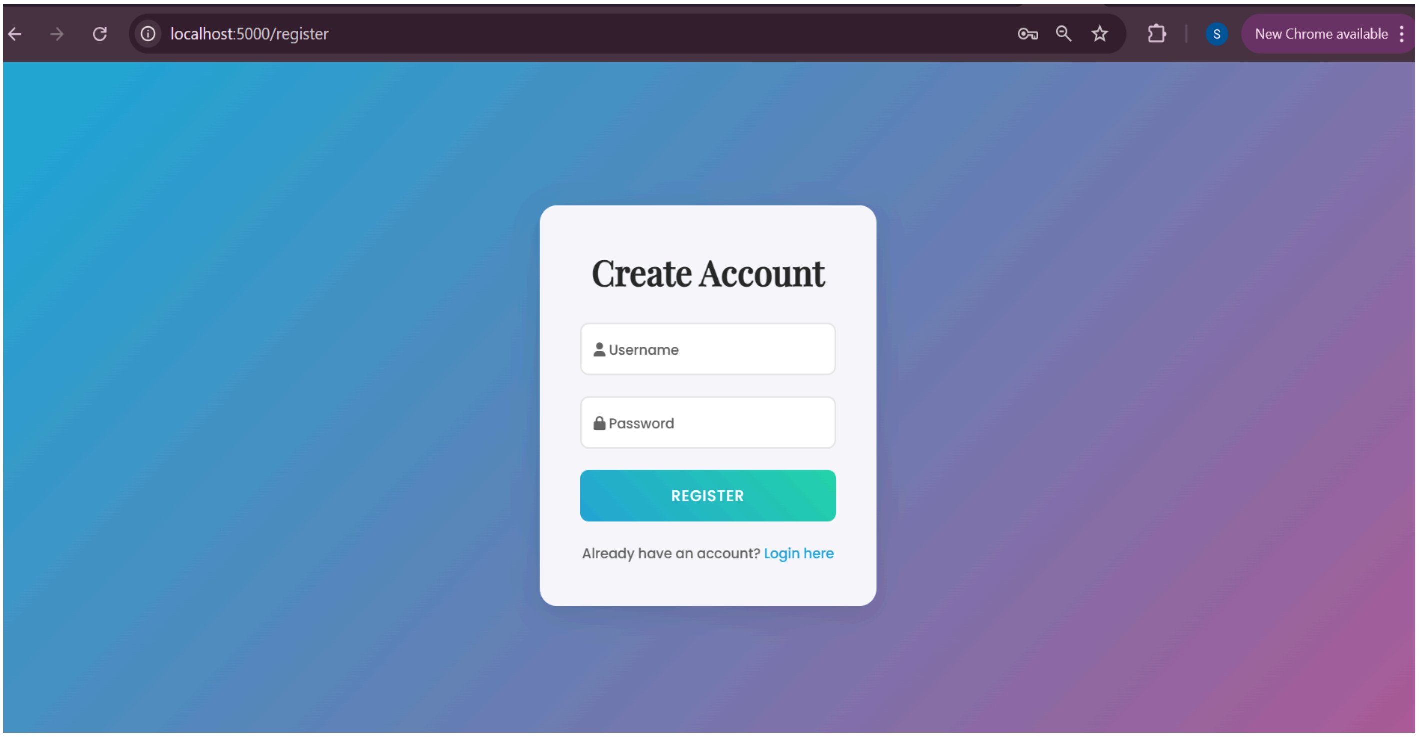
- **Increased Number of Classes:** Each distinct admin action typically requires a separate command class.
- **Potential Complexity for Simple Operations:** For very basic actions, the overhead of creating command objects might seem unnecessary.

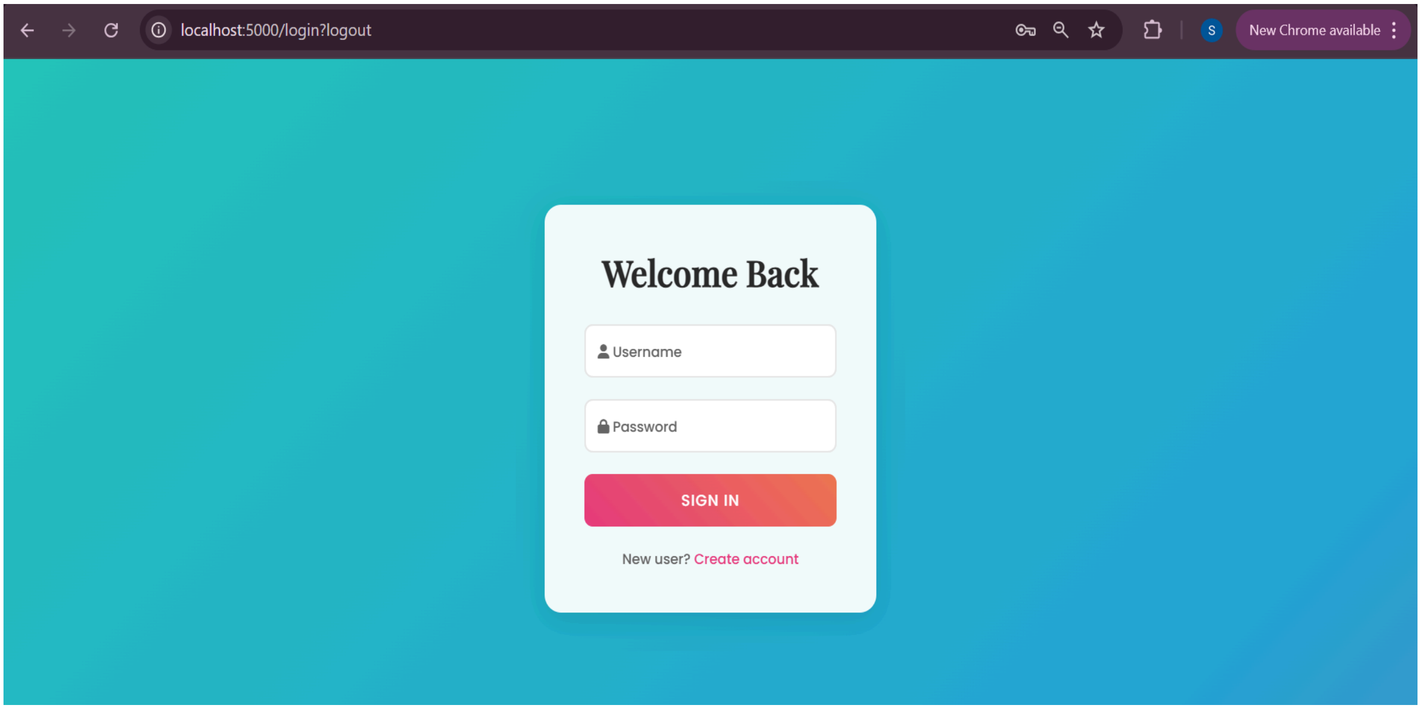
How it helps in Bus Reservation System:

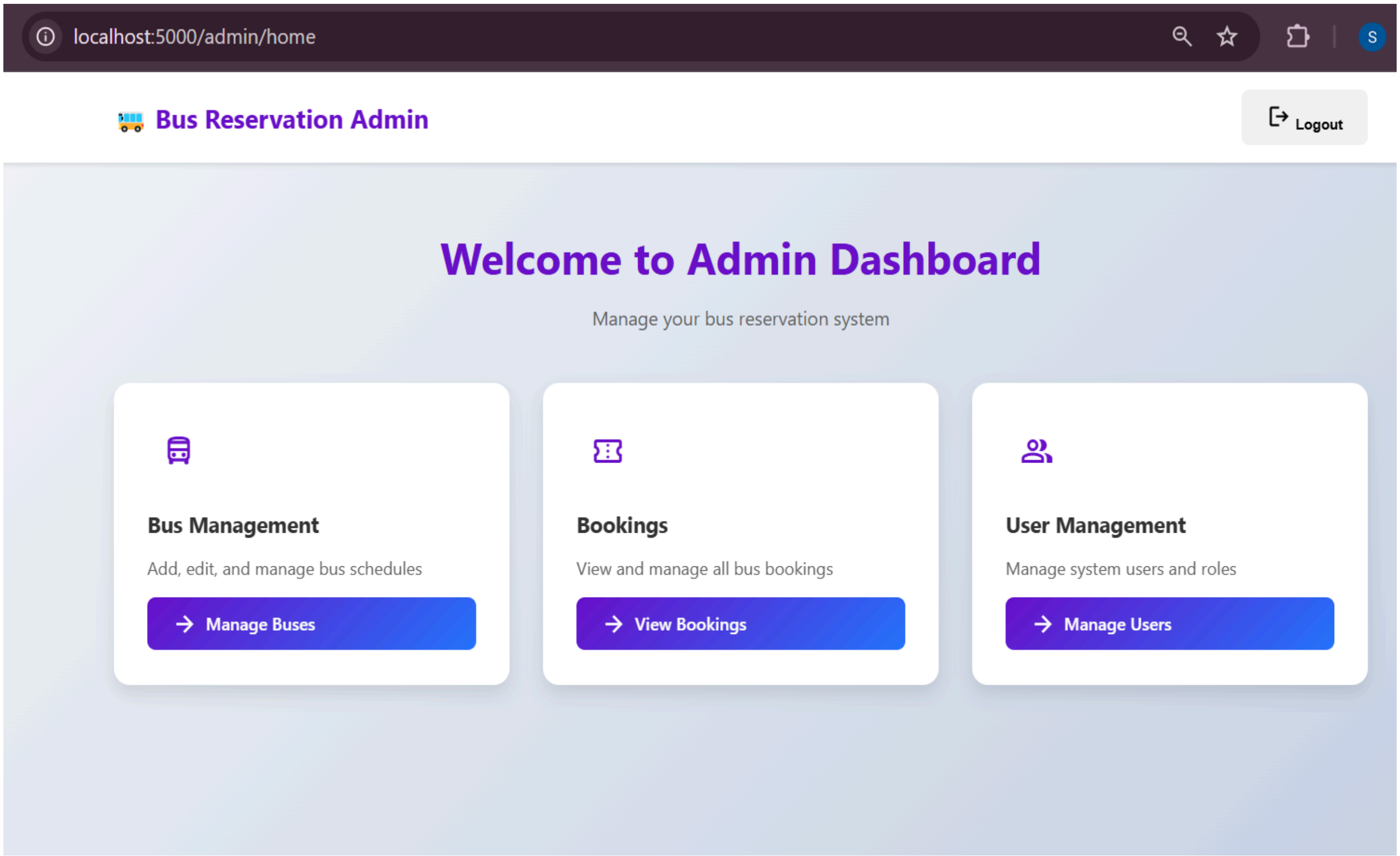
- This approach is used to handle operations like adding, editing, and deleting buses. It enables administrative actions such as introducing new buses, updating routes, and managing user accounts to be structured as command objects. This not only promotes better organization but also allows for the possibility of reversing actions if needed. Additionally, it supports the creation of an audit log to track all administrative changes within the system.

Screenshots

UI:







Welcome to Admin Dashboard

Manage your bus reservation system

Bus Management

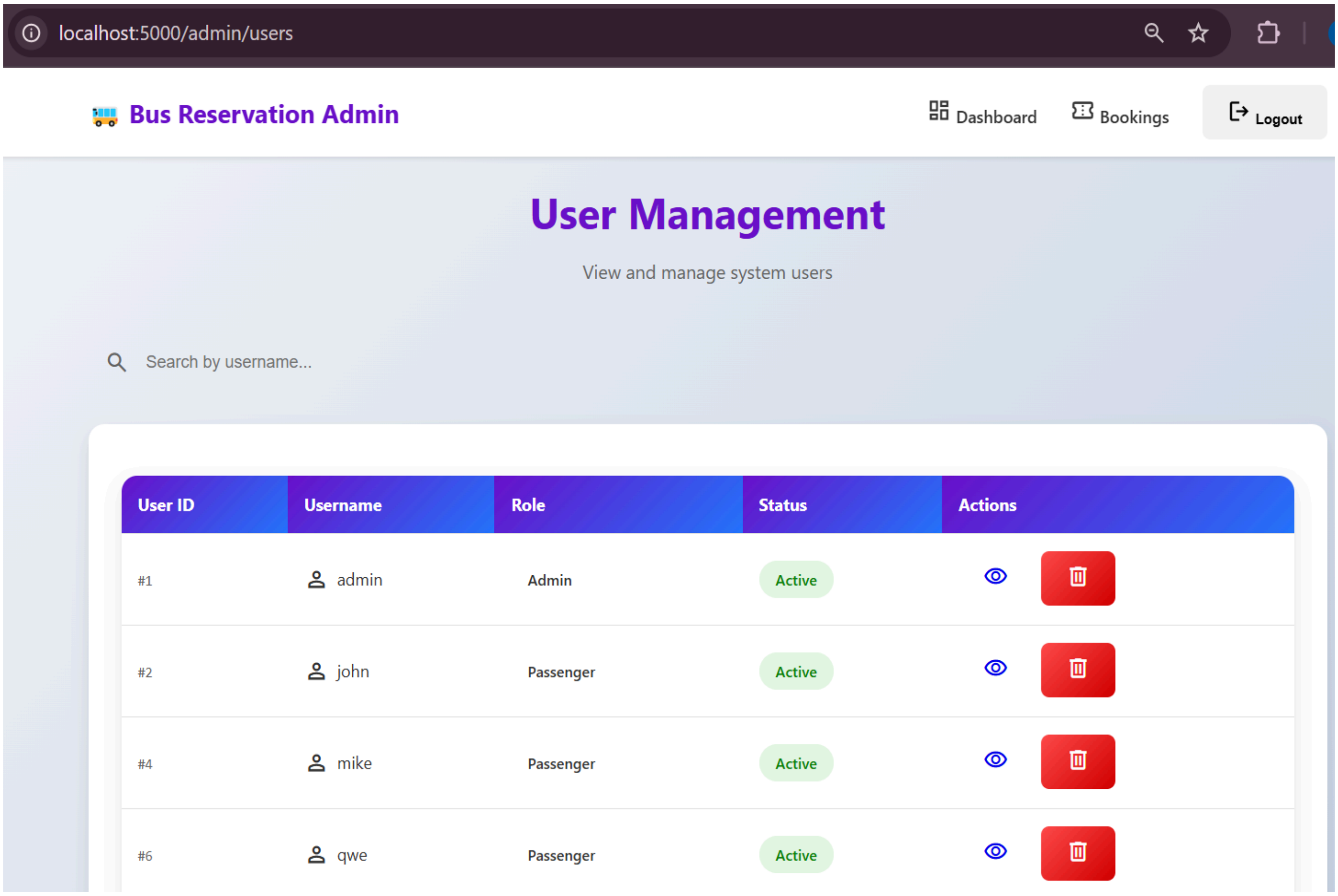
Add, edit, and manage bus schedules

Bookings

View and manage all bus bookings

User Management

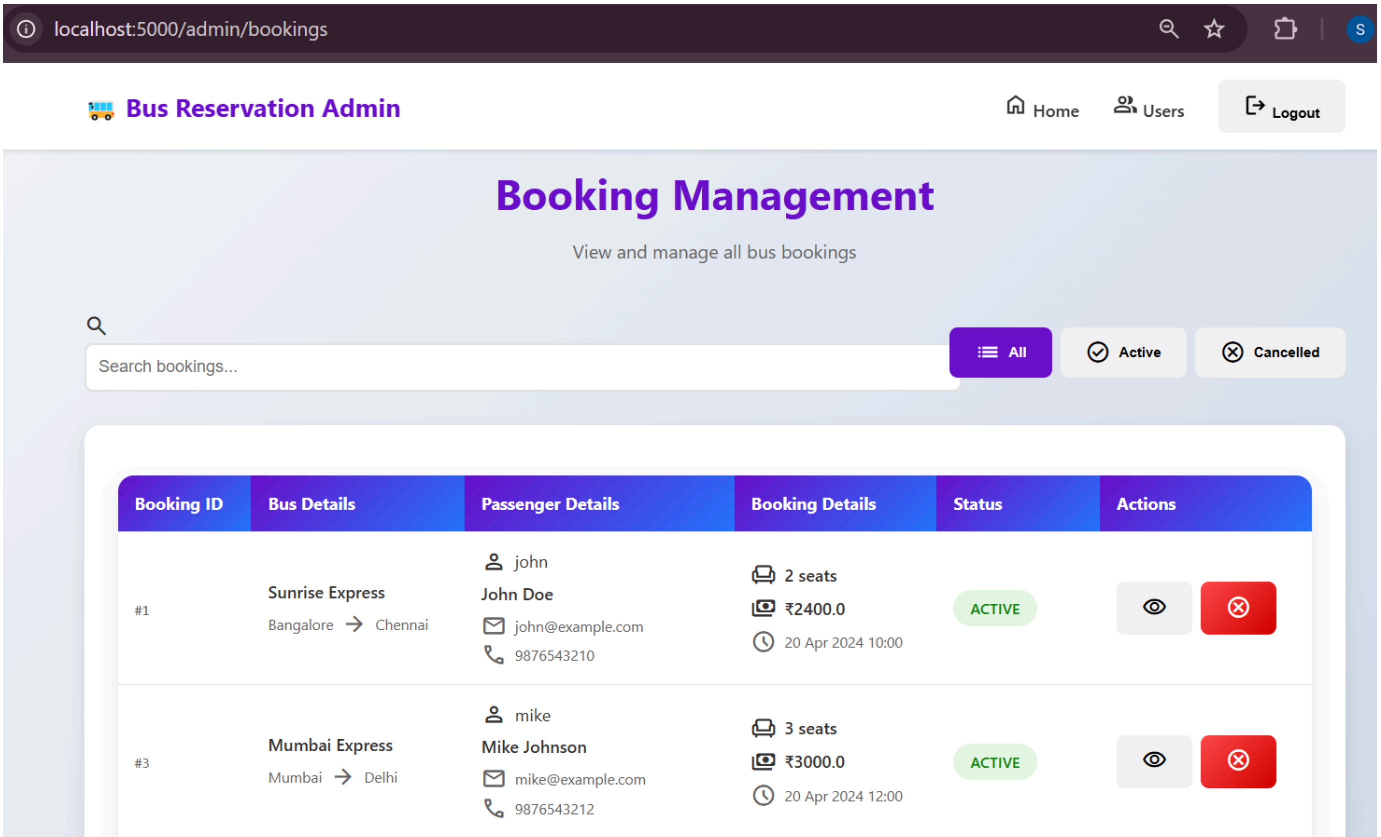
Manage system users and roles



User Management

View and manage system users

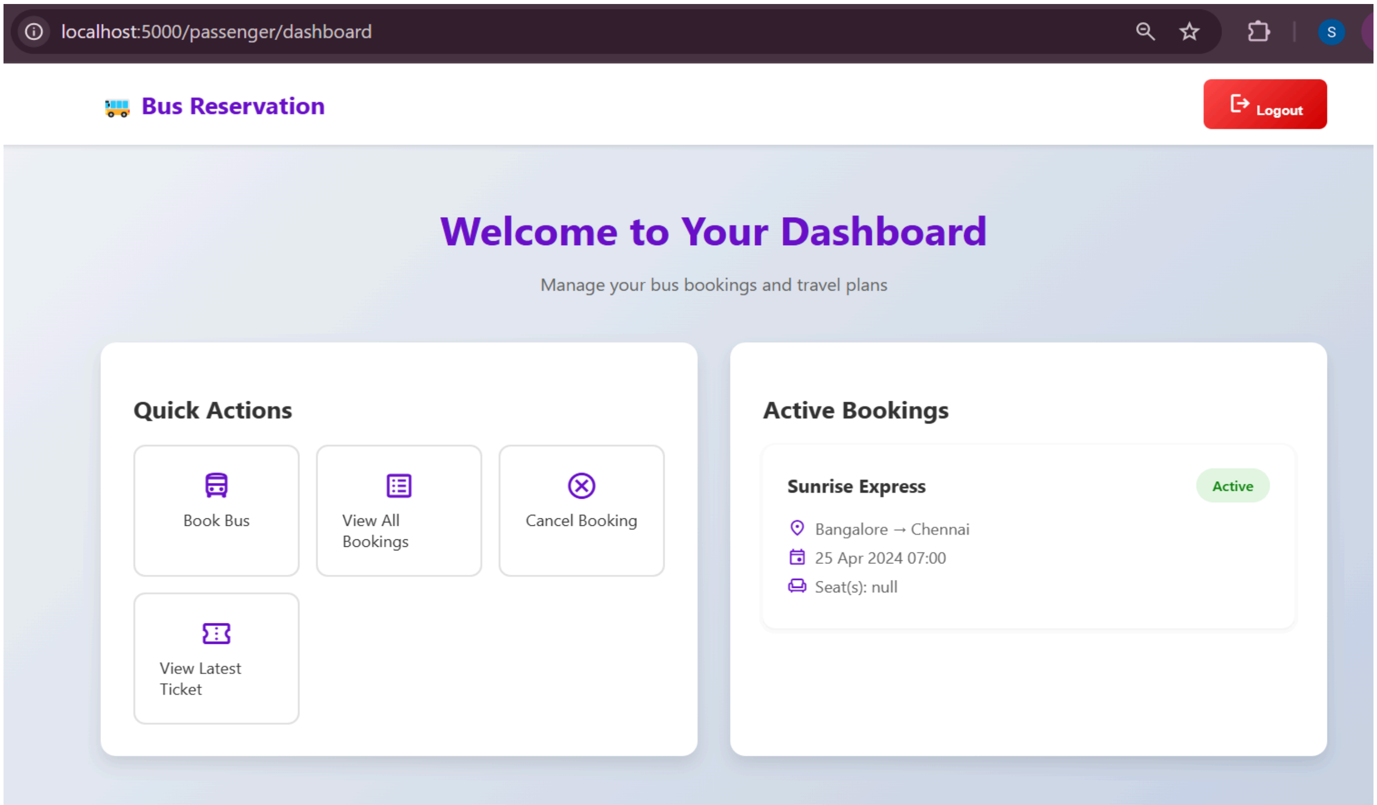
User ID	Username	Role	Status	Actions
#1	admin	Admin	Active	👁 🗑
#2	john	Passenger	Active	👁 🗑
#4	mike	Passenger	Active	👁 🗑
#6	qwe	Passenger	Active	👁 🗑



Booking Management

View and manage all bus bookings

Booking ID	Bus Details	Passenger Details	Booking Details	Status	Actions
#1	Sunrise Express Bangalore → Chennai	john John Doe john@example.com 9876543210	2 seats ₹2400.0 20 Apr 2024 10:00	ACTIVE	
#3	Mumbai Express Mumbai → Delhi	mike Mike Johnson mike@example.com 9876543212	3 seats ₹3000.0 20 Apr 2024 12:00	ACTIVE	



Welcome to Your Dashboard

Manage your bus bookings and travel plans

Quick Actions

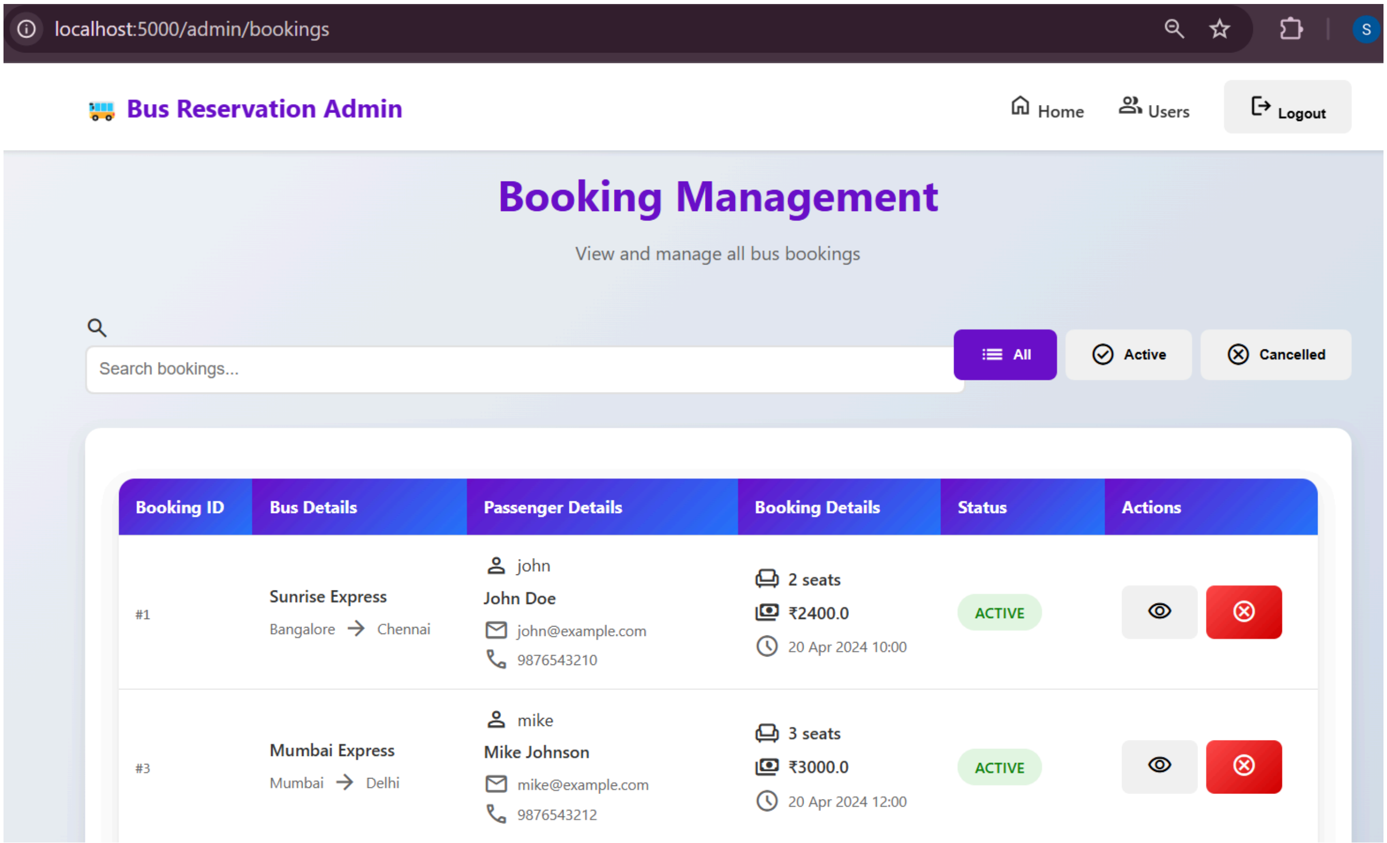
Active Bookings

Sunrise Express

Bangalore → Chennai

25 Apr 2024 07:00

Seat(s): null

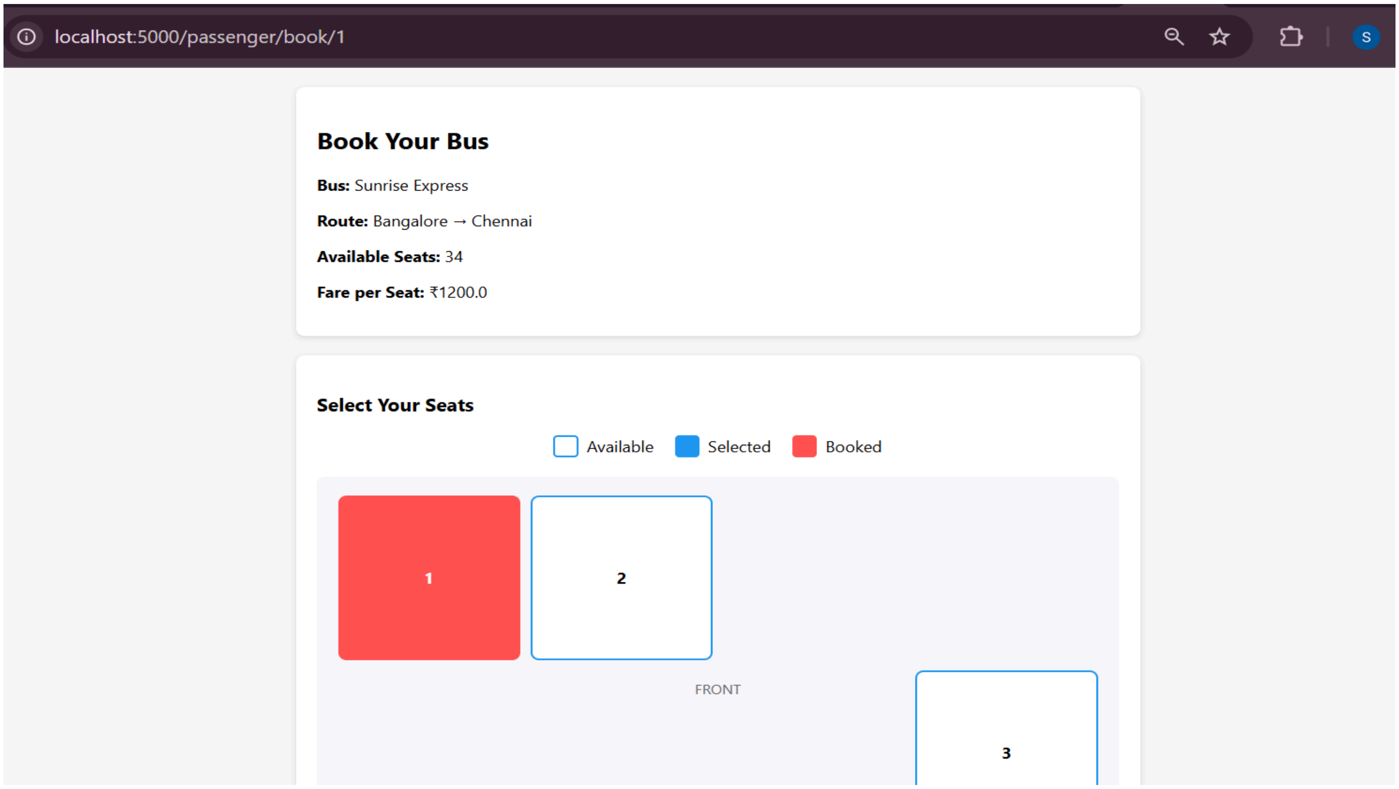


Booking Management

View and manage all bus bookings

Search bookings...

Booking ID	Bus Details	Passenger Details	Booking Details	Status	Actions
#1	Sunrise Express Bangalore → Chennai	john John Doe john@example.com 9876543210	2 seats ₹2400.0 20 Apr 2024 10:00	ACTIVE	
#3	Mumbai Express Mumbai → Delhi	mike Mike Johnson mike@example.com 9876543212	3 seats ₹3000.0 20 Apr 2024 12:00	ACTIVE	



Book Your Bus

Bus: Sunrise Express

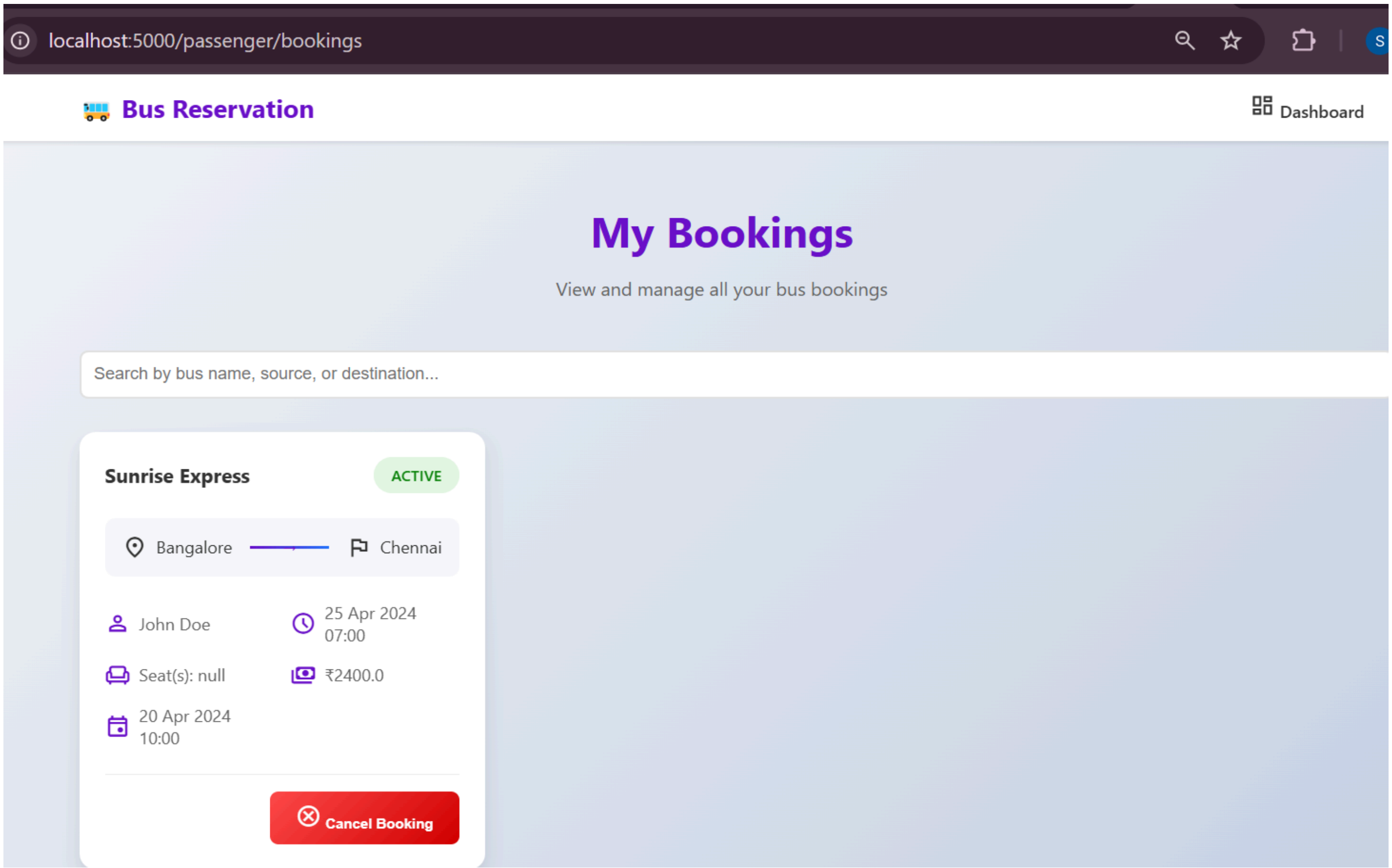
Route: Bangalore → Chennai

Available Seats: 34

Fare per Seat: ₹1200.0

Select Your Seats

☐ Available 🟦 Selected 🟥 Booked



localhost:5000/passenger/bookings
Bus Reservation
Dashboard
Search by bus name, source, or destination...
Cancel Booking

My Bookings

View and manage all your bus bookings

Sunrise Express ACTIVE

Bangalore → Chennai

John Doe

25 Apr 2024 07:00

Seat(s): null

₹2400.0

20 Apr 2024 10:00

Individual contributions of the team members:

Name	**Module worked on**
SIMONNA ANNA DCOSTA	login and registration+UI
SINCHANA AS	Admin functionalities
SHILPA M TALAWAR	Bus booking+Report
SPANDANA M POOJARY	cancellation+Report

Github link:
https://github.com/Simonna123/OOAD.git