PES University, Bangalore
(Established under Karnataka Act No. 16 of 2013)
Department of Computer Science and Engineering

UE21CS352B: Object-oriented Analysis and Design using Java

Miniproject Report
Project Title: Bus Reservation System

Project Team Members:
1) Rasamsetty Pranavi – PES1UG21CS478
2) Ria R Kulkarni – PES1UG21CS487
3) Riya Jayakumar – PES1UG21CS492
4) Riya Bansal – PES1UG21CS929

TABLE OF CONTENTS

1. **SYNOPSIS:**

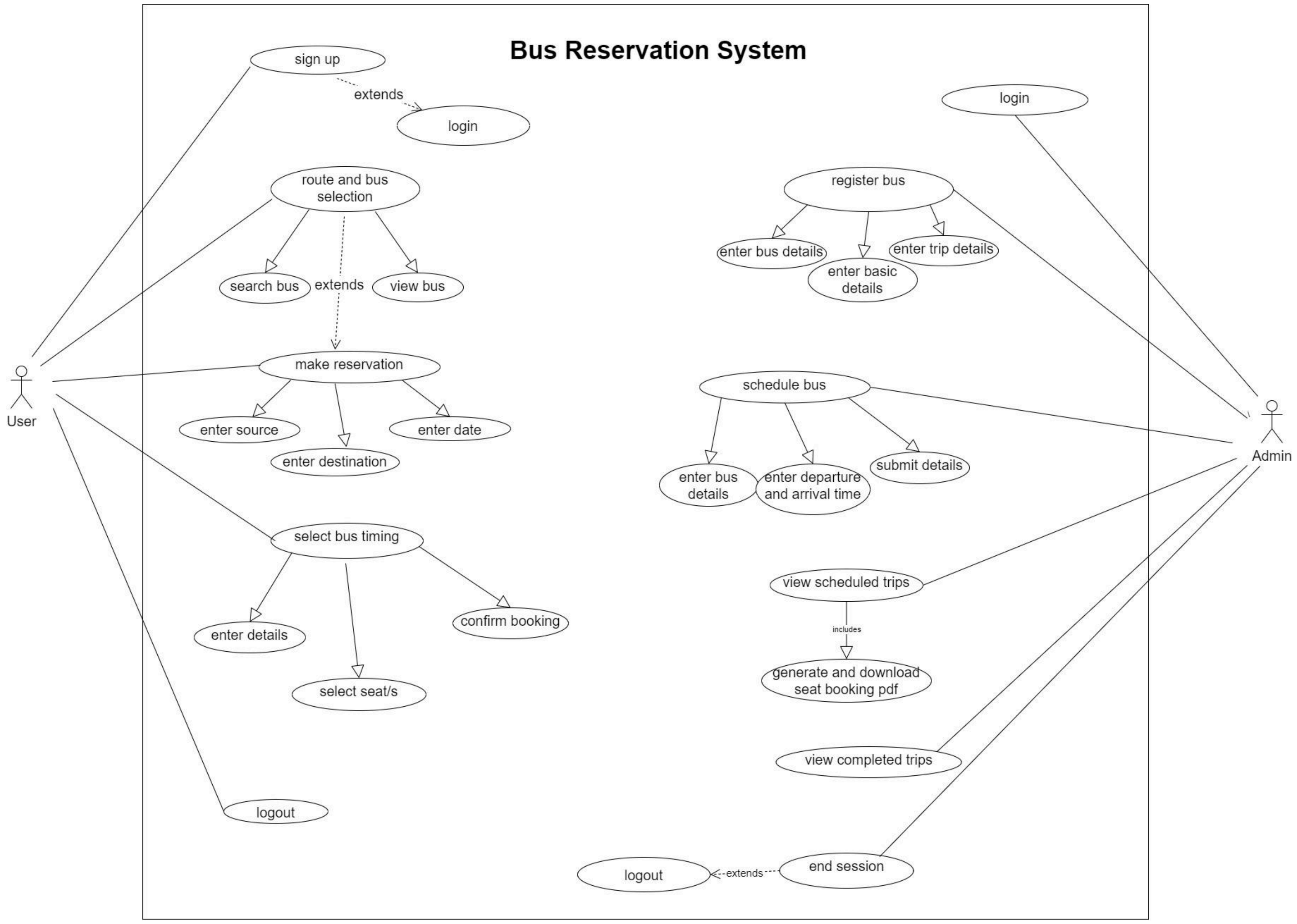
The Bus Reservation System is a web-based application designed to facilitate seamless and efficient bus ticket booking for users. Leveraging Java, the Spring Framework, and associated technologies, the system offers an intuitive interface for both users and administrators. The application ensures secure authentication, enables bus and route management, and provides a user-friendly reservation process.

The Bus Reservation System employs a comprehensive set of classes to facilitate seamless interactions between users and administrators. A user is allowed to register, log in, and manage their profiles. This also enables them to view available buses, make reservations, and submit feedback. A user has an ID, full name, and a list of reservations and feedback. The user's mobile and emails are also stored. One user can book many buses. Administrators get authenticated with name, email and password to gain exclusive privileges for managing routes, buses, and accessing user and reservation details. An admin may handle one or more routes, buses and users.

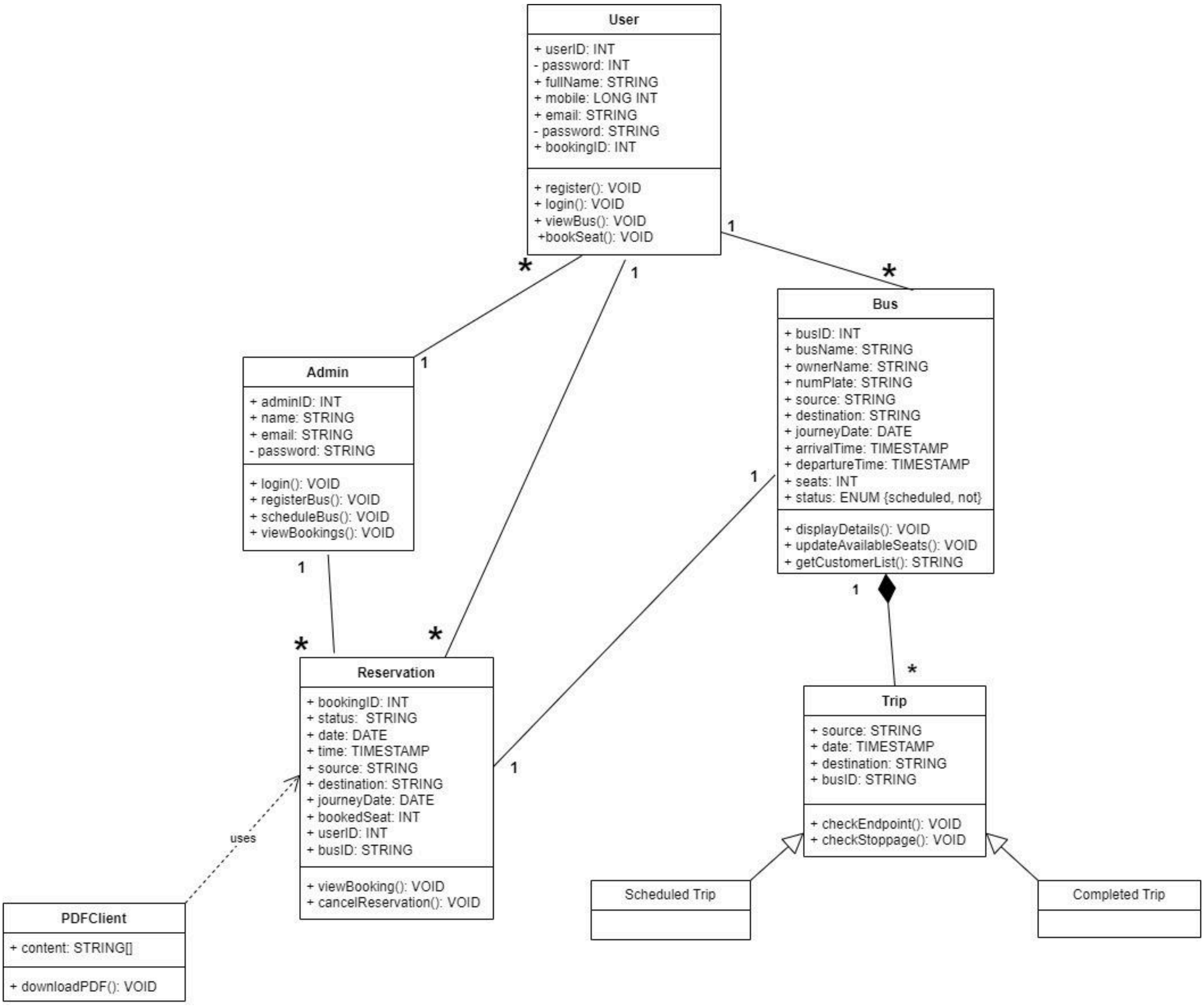
A bus is associated with a route that provides essential details about available routes, destinations, distances and list of buses, aiding users in making informed decisions. A bus also has a name, ID, driver name, type of bus, journey date, arrival and departure time, list of reservation and number of available seats which are updated on booking along with the route and fare. A reservation is used to capture the date of journey, source and destination, booked seat and price of ticket, allowing them to view and cancel reservations. The feedback option records user feedback like driver rating, service rating, overall rating and comments for system improvement. One user can have a lot of feedback but can review each bus only once. The system also verifies user and admin credentials (authentication), ensuring secure access, while also generating session tokens for secure interactions.

Overall, these classes collectively form a robust, secure, and user-friendly bus reservation platform using Java and the Spring Framework.

2. USE CASE DIAGRAM:



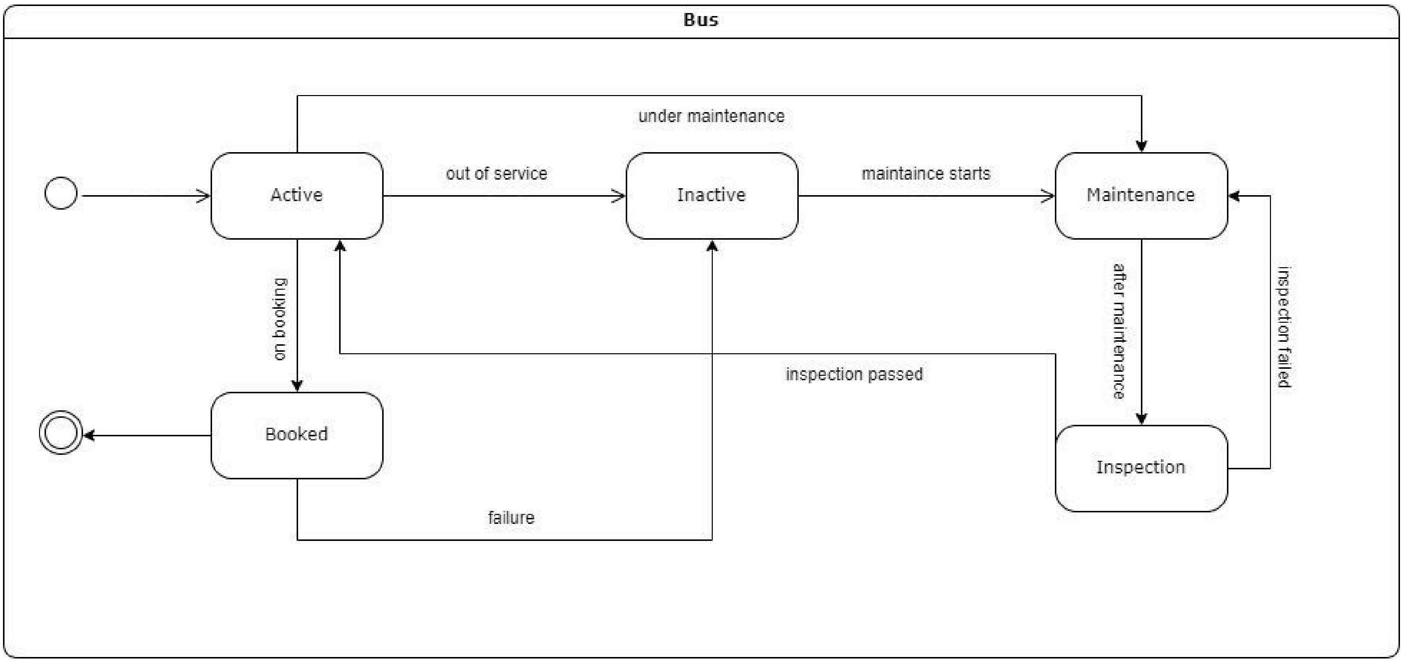
3. CLASS DIAGRAM:



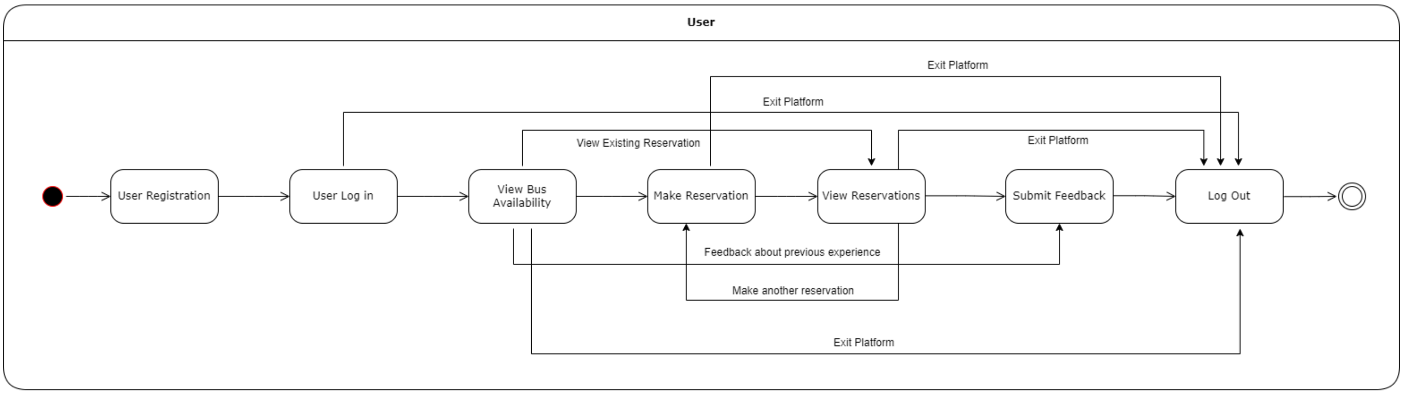
- Open-Closed Principle followed as classes are open for extensions but closed for modifications.
- Single Responsibility Principle followed as each class handles its own data and methods.

4. STATE DIAGRAMS:

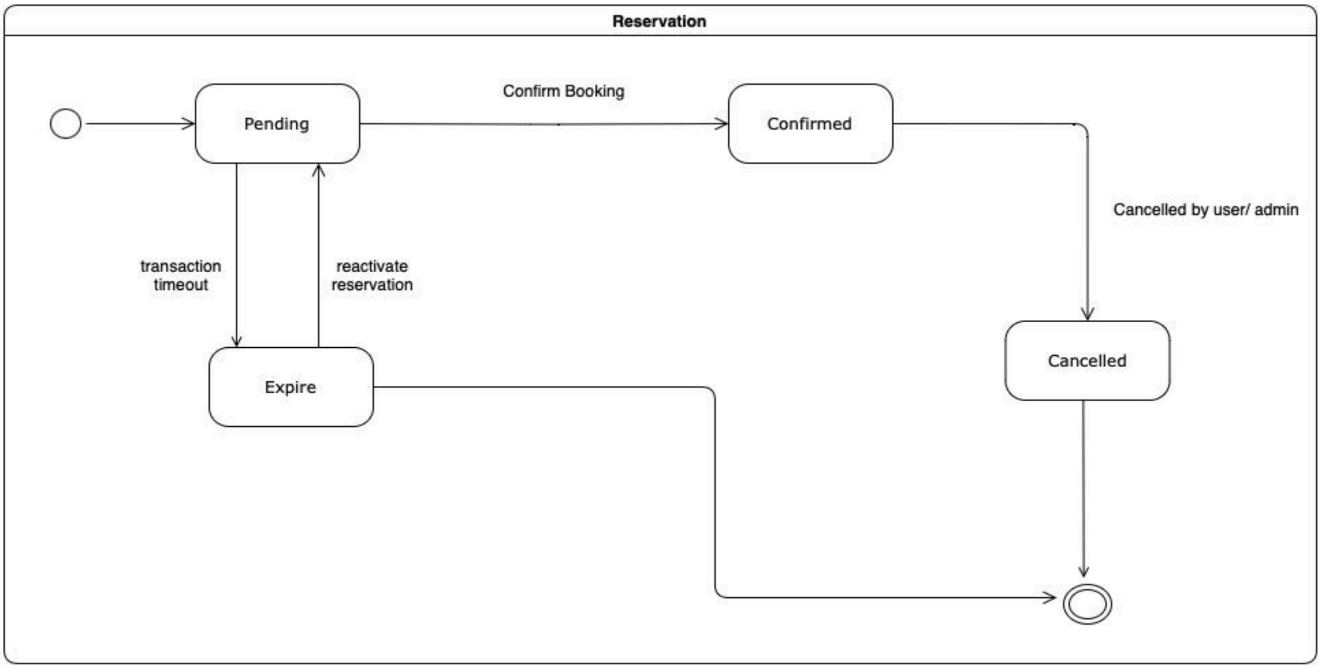
1) Bus:



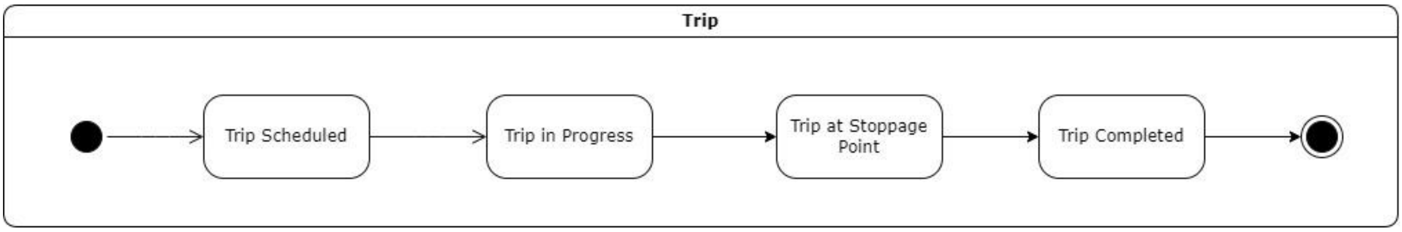
2) User:



3) Reservation:

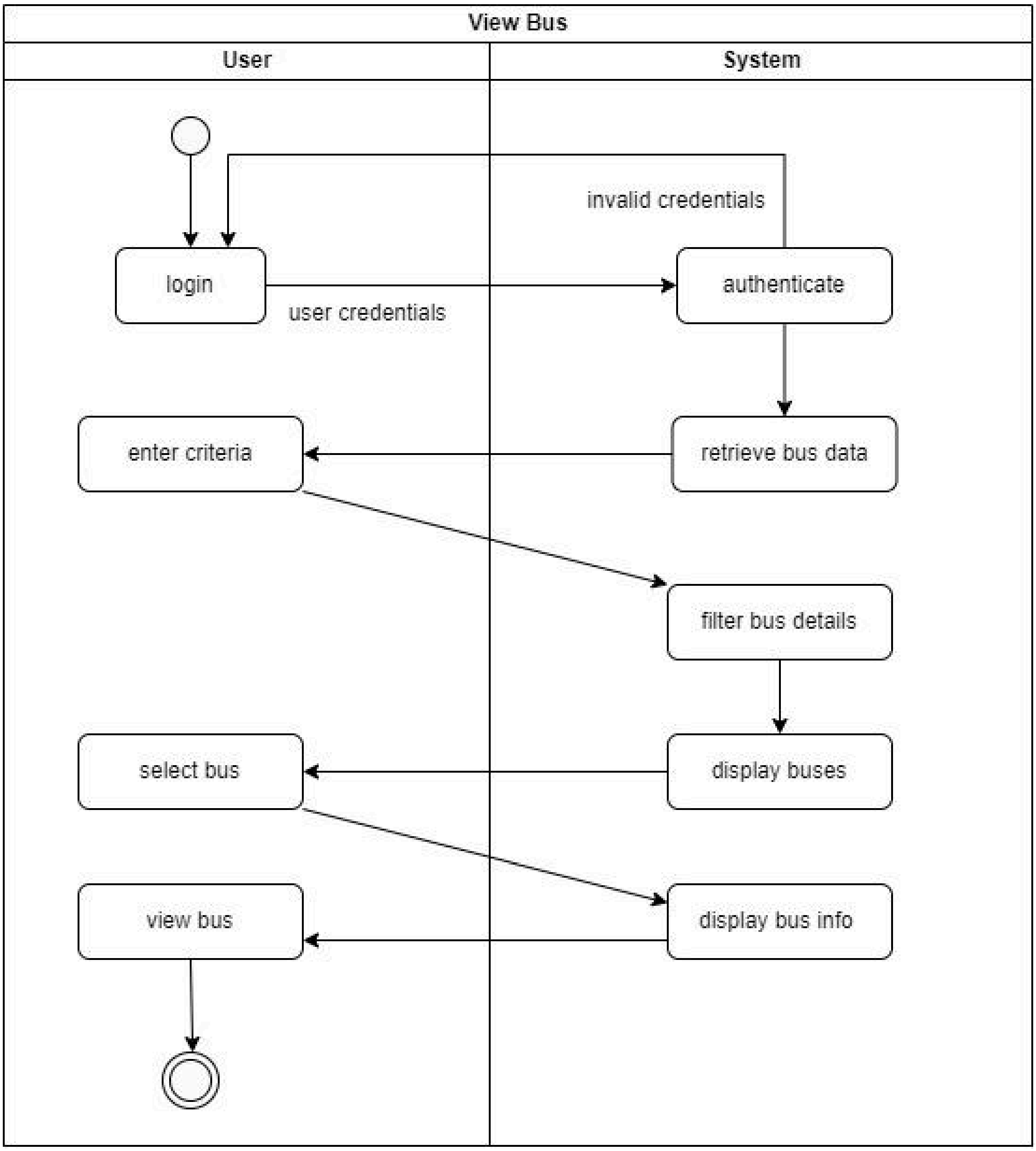


4) Trip:

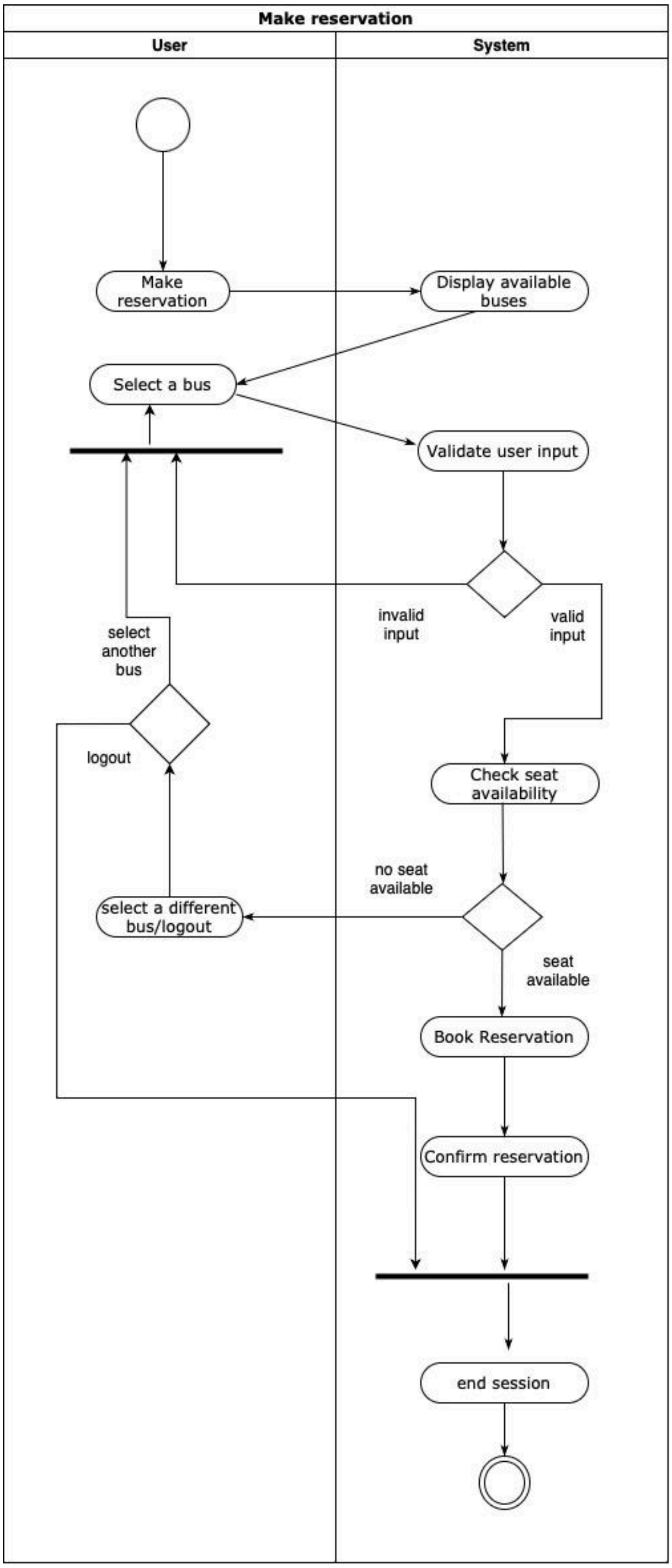


5. ACTIVITY DIAGRAMS:

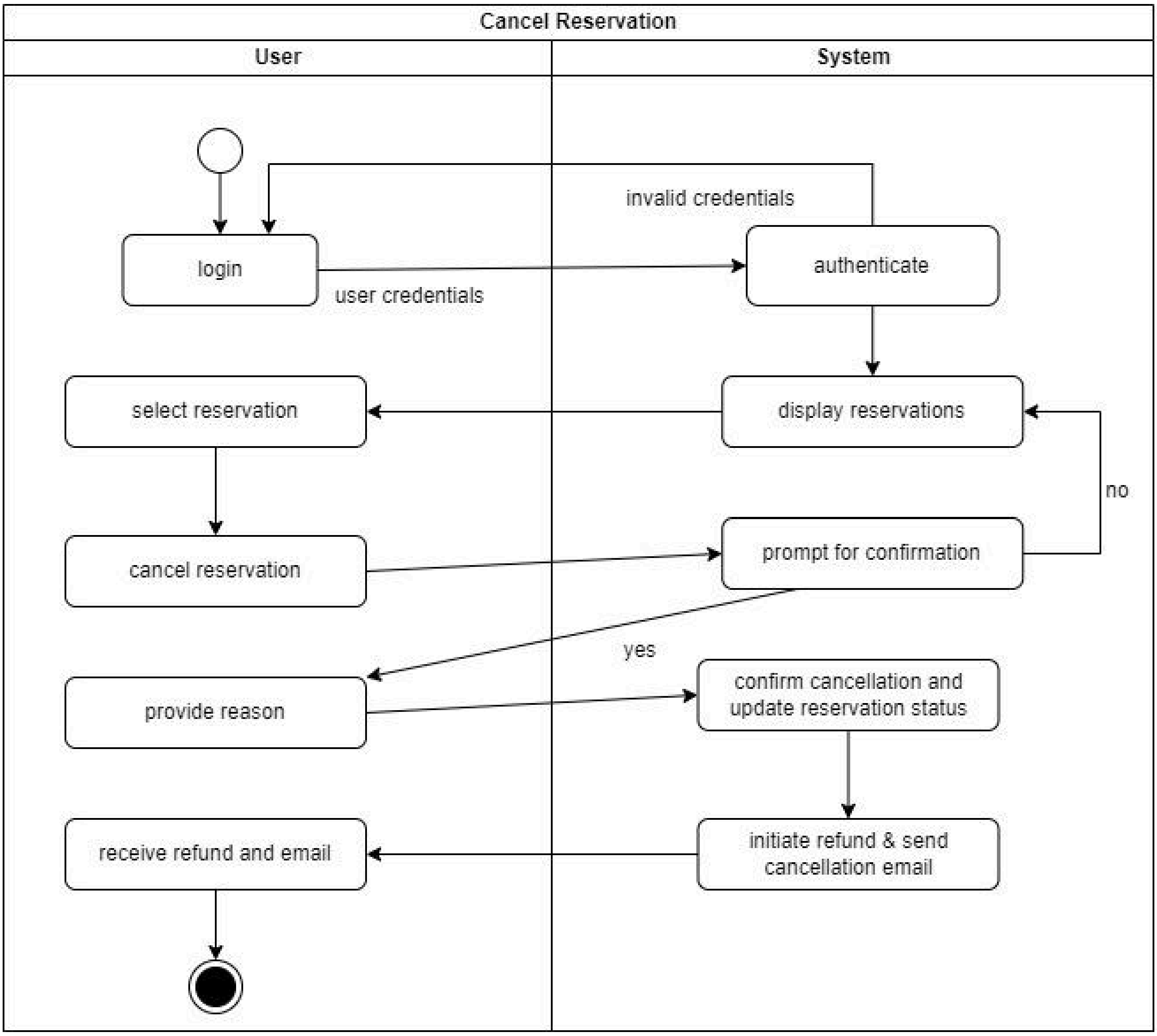
1) View Bus:



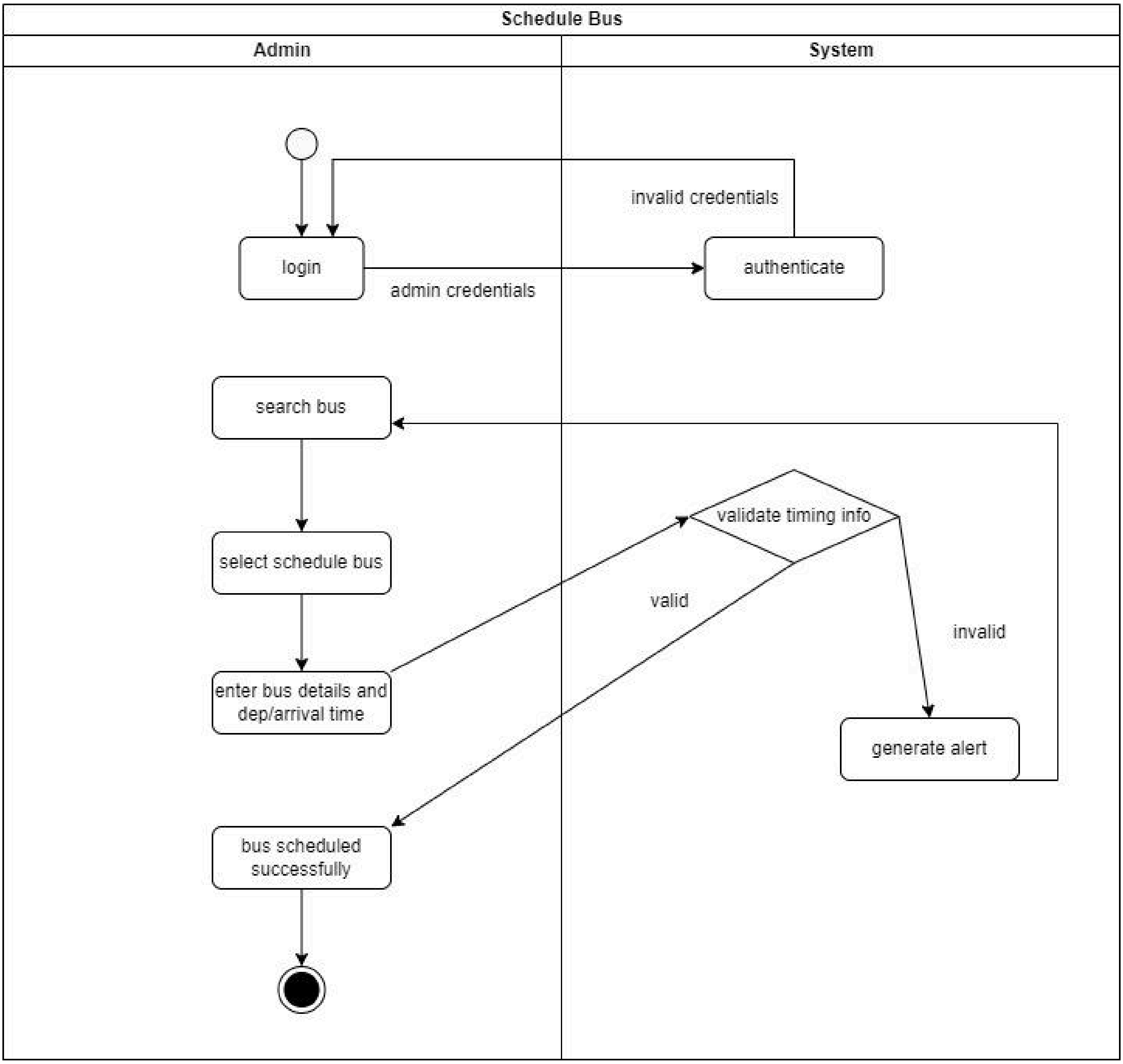
2) Make Reservation:



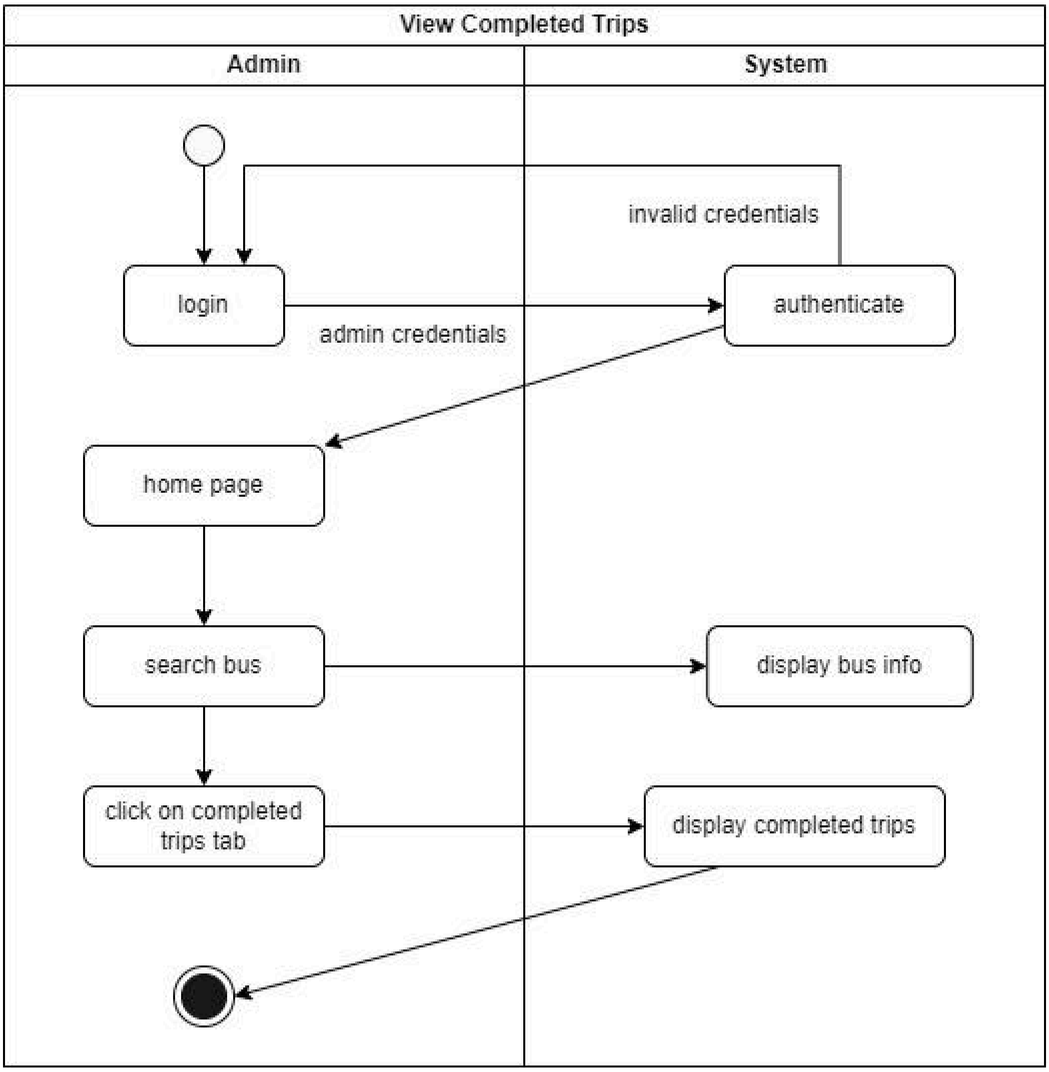
3) Cancel Reservation:



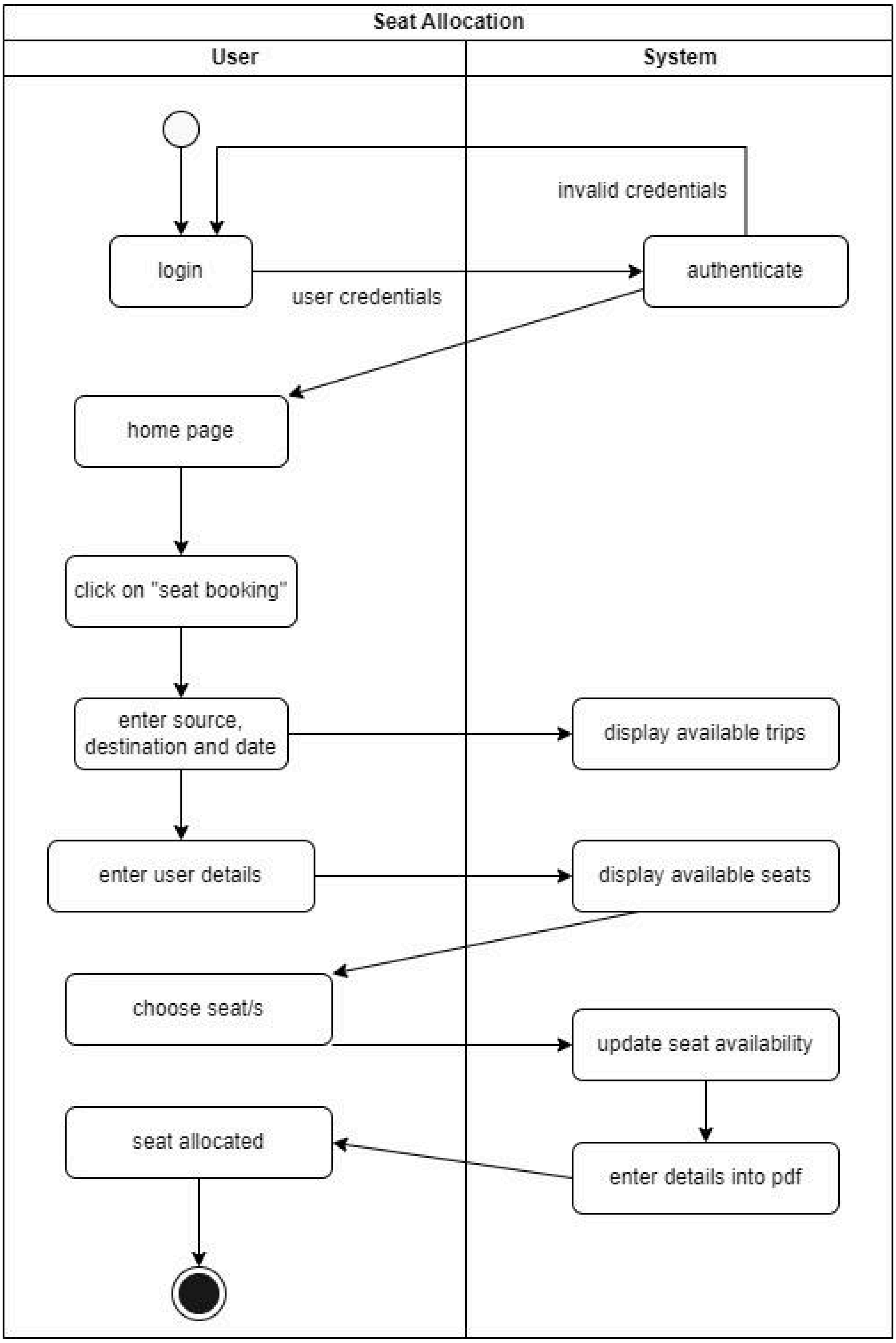
4) Schedule Bus:



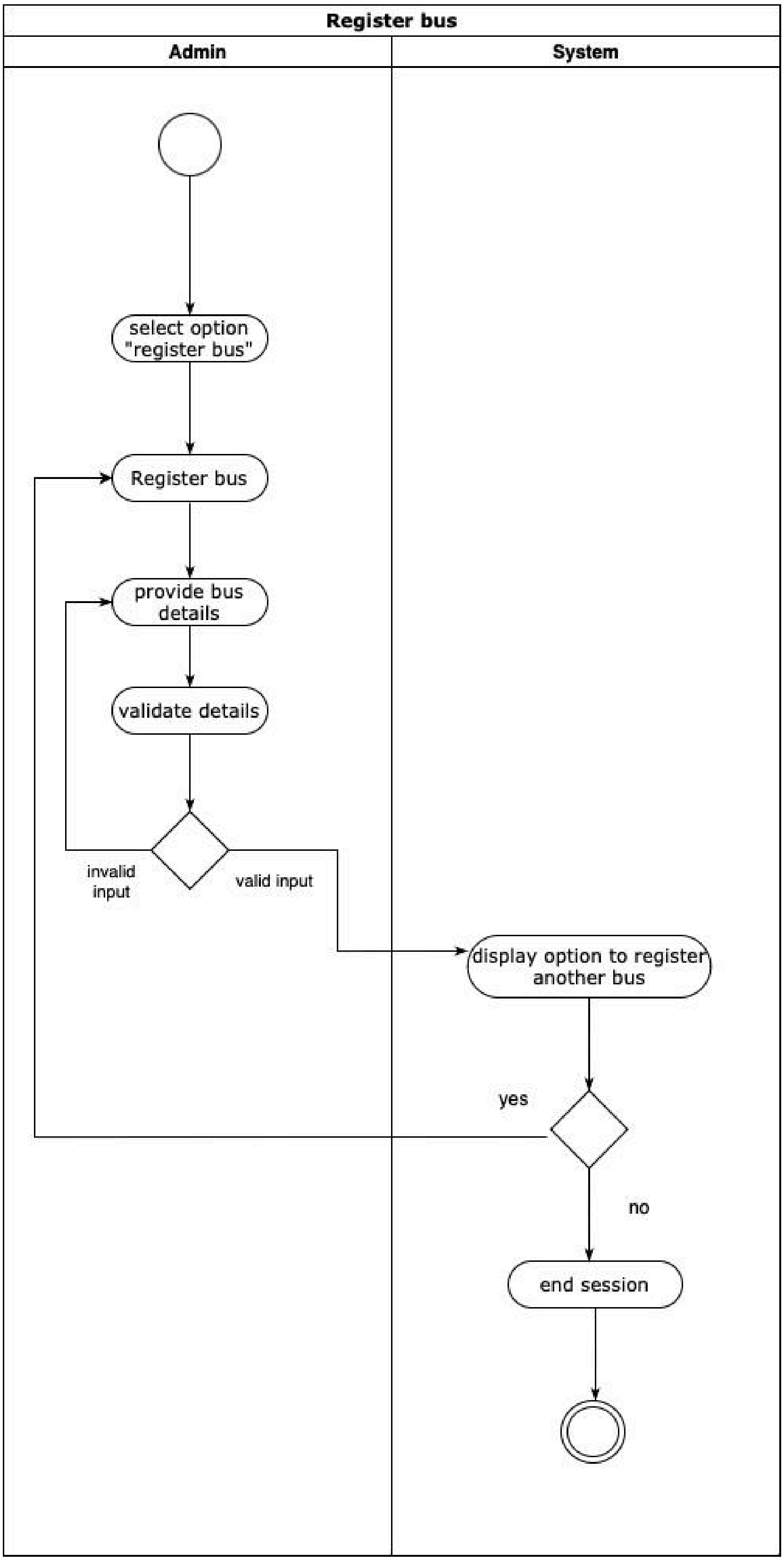
5) View Completed Trips:



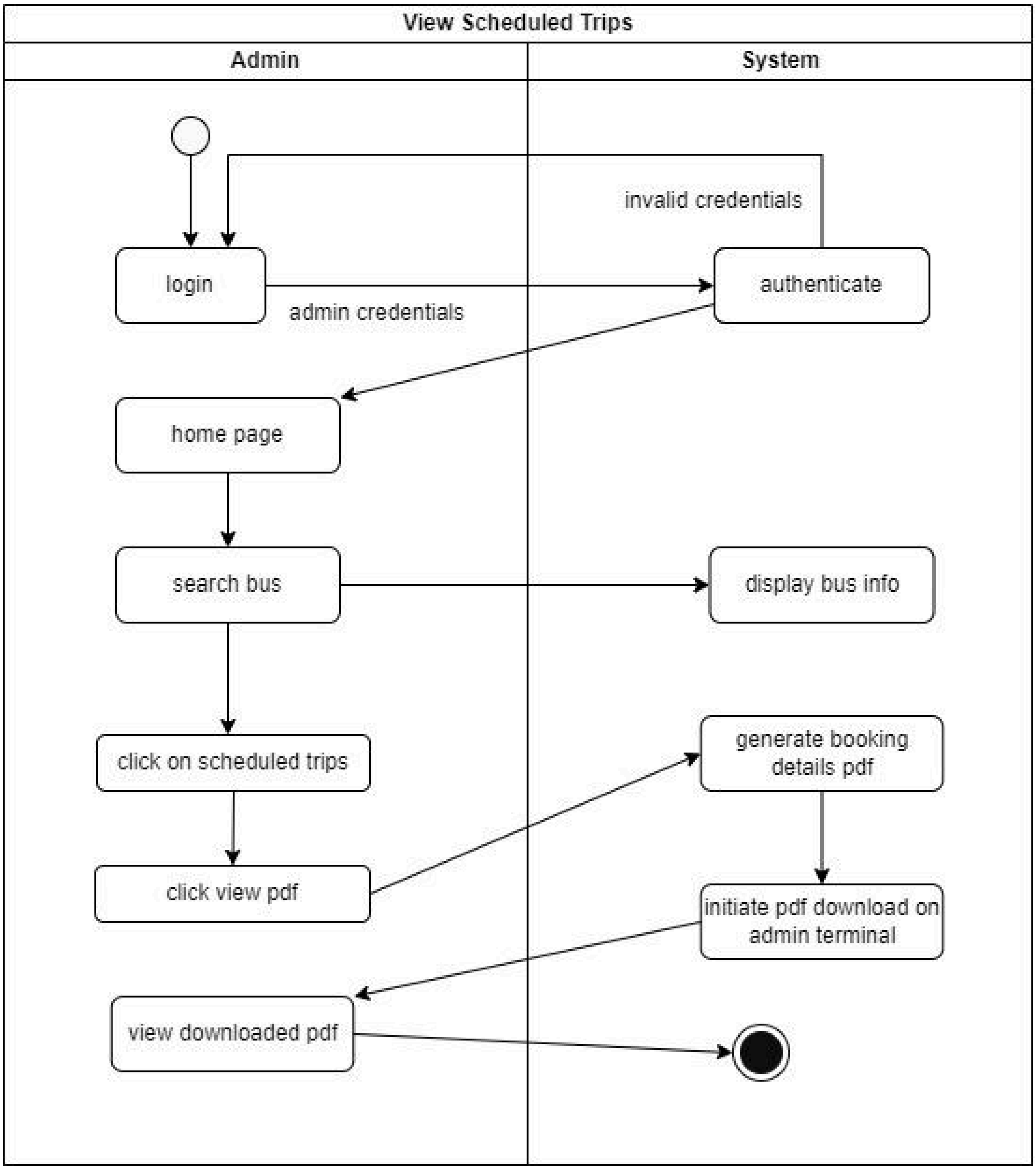
6) Seat Allocation:



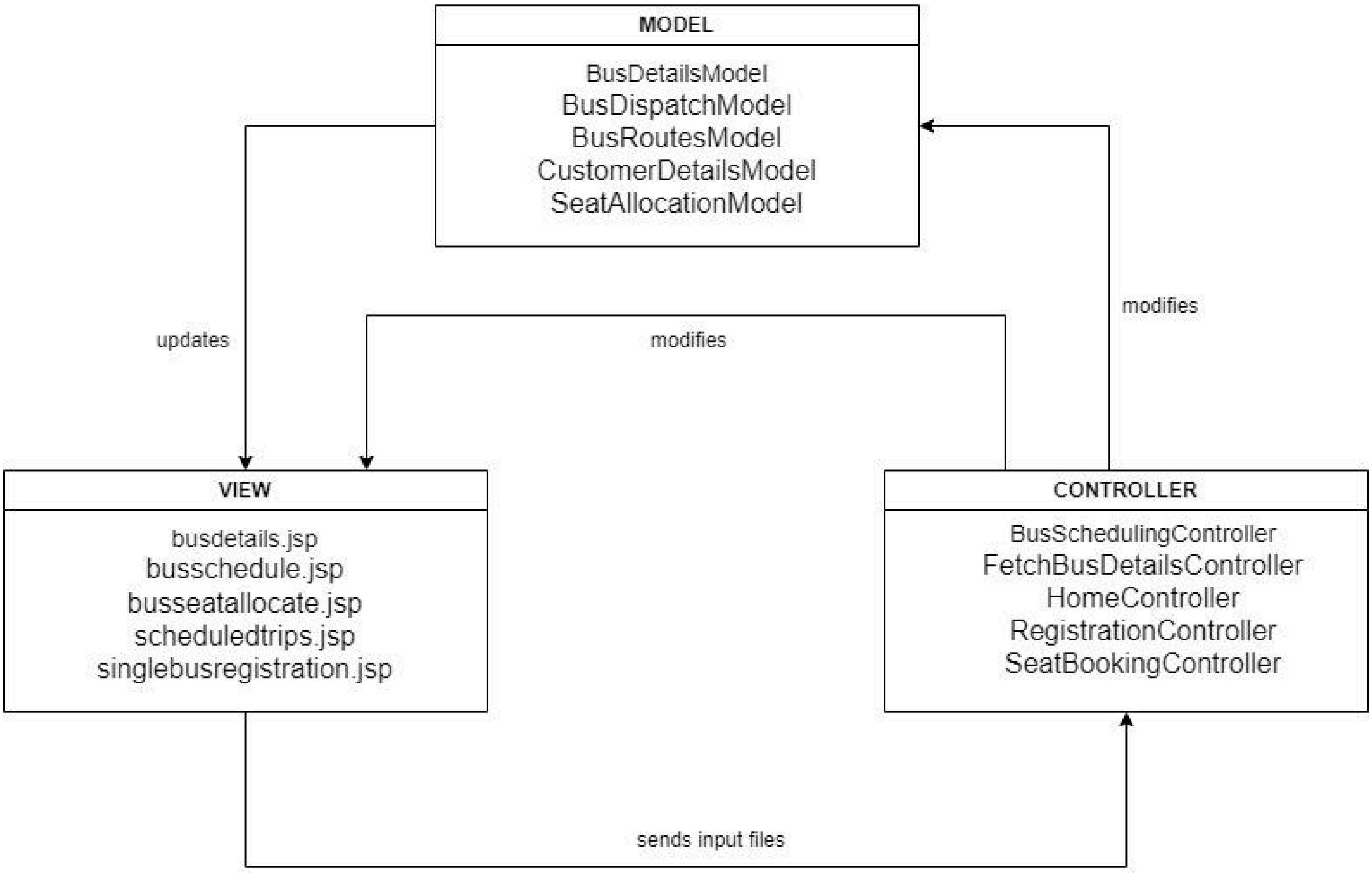
7) Register Bus:



8) View Scheduled Trips:



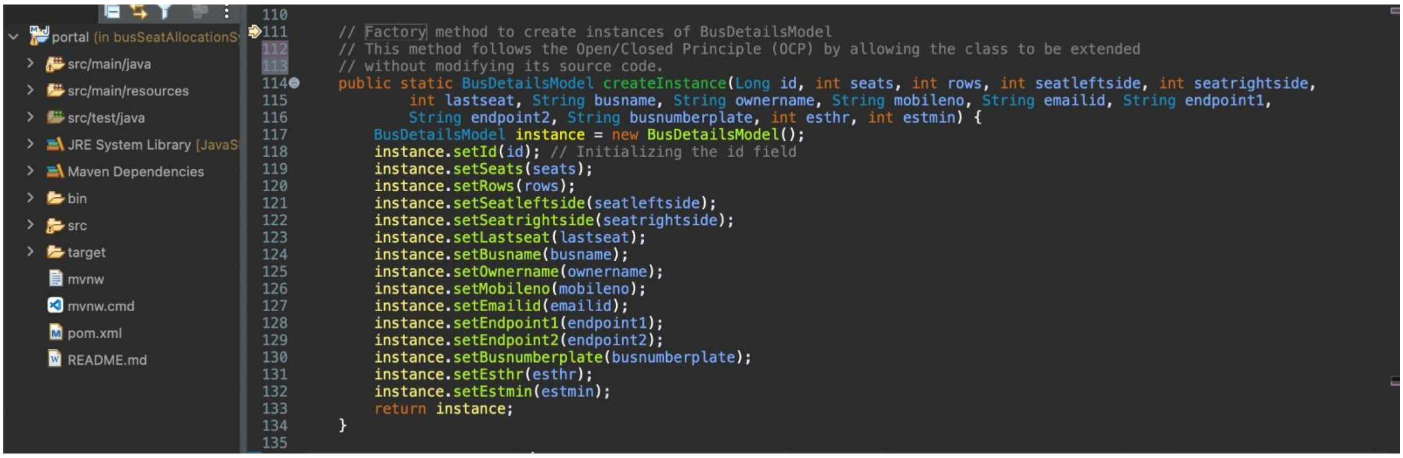
6. MVC ARCHITECTURE:



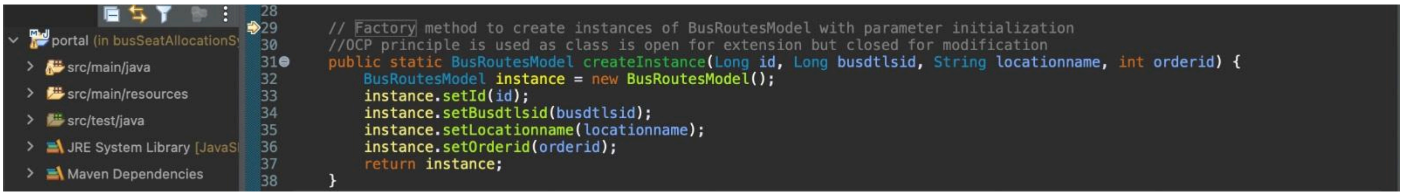
7. DESIGN PATTERNS:

1) Creational Pattern - Factory:

In object oriented programming, the factory method pattern is a creational pattern that uses factory methods to deal with the problem of creating objects without having to specify the exact class of the object that will be created. Here it is used to create instances of BusDetailsModel and BusRoutesModel with parameter initialization.



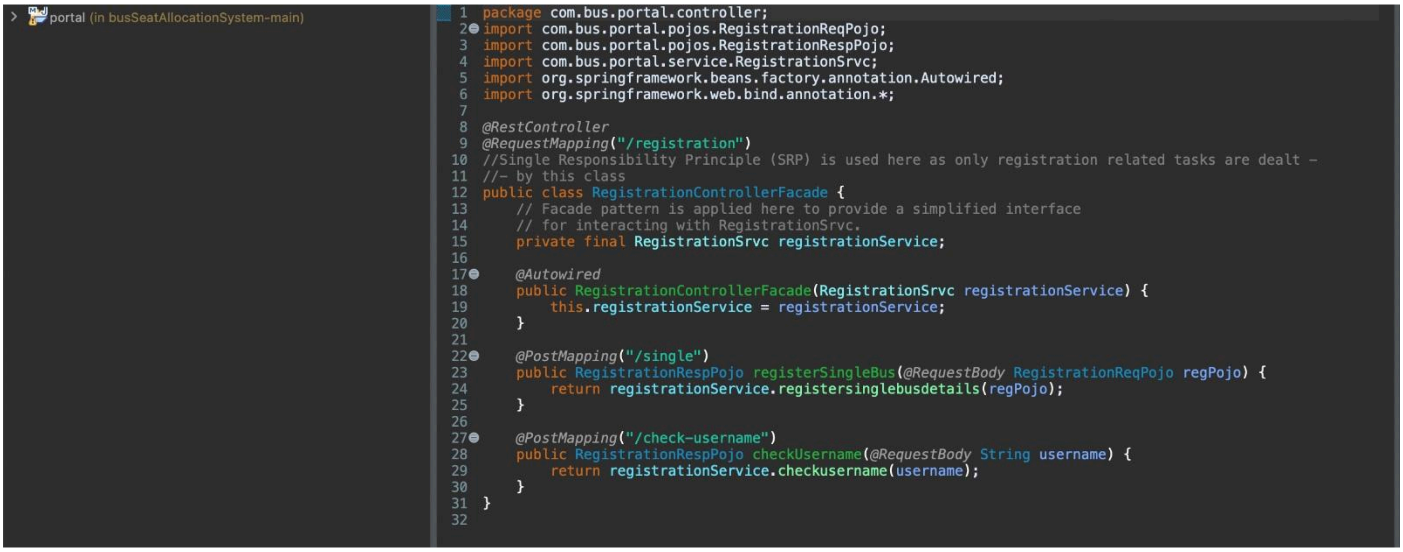
```java

    // Factory method to create instances of BusDetailsModel
    // This method follows the Open/Closed Principle (OCP) by allowing the class to be extended
    // without modifying its source code.
    public static BusDetailsModel createInstance(Long id, int seats, int rows, int seatleftside, int seatrightside,
            int lastseat, String busname, String ownername, String mobileno, String emailid, String endpoint1,
            String endpoint2, String busnumberplate, int esthr, int estmin) {
        BusDetailsModel instance = new BusDetailsModel();
        instance.setId(id); // Initializing the id field
        instance.setSeats(seats);
        instance.setRows(rows);
        instance.setSeatleftside(seatleftside);
        instance.setSeatrightside(seatrightside);
        instance.setLastseat(lastseat);
        instance.setBusname(busname);
        instance.setOwnername(ownername);
        instance.setMobileno(mobileno);
        instance.setEmailid(emailid);
        instance.setEndpoint1(endpoint1);
        instance.setEndpoint2(endpoint2);
        instance.setBusnumberplate(busnumberplate);
        instance.setEsthr(esthr);
        instance.setEstmin(estmin);
        return instance;
    }

```



```java

    // Factory method to create instances of BusRoutesModel with parameter initialization
    //OCP principle is used as class is open for extension but closed for modification
    public static BusRoutesModel createInstance(Long id, Long busdtlsid, String locationname, int orderid) {
        BusRoutesModel instance = new BusRoutesModel();
        instance.setId(id);
        instance.setBusdtlsid(busdtlsid);
        instance.setLocationname(locationname);
        instance.setOrderid(orderid);
        return instance;
    }
```

2) Structural Pattern - Facade:

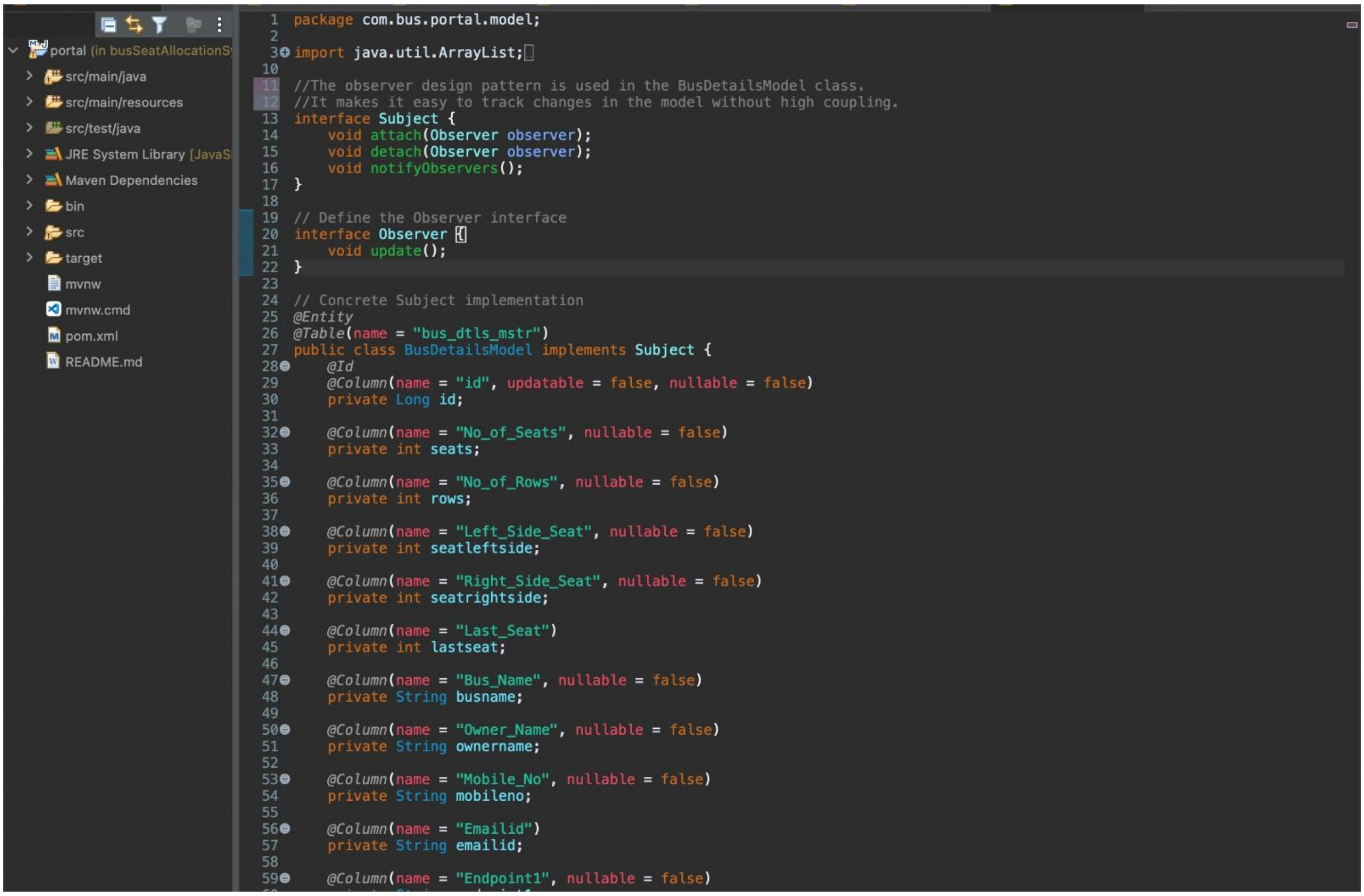
The facade pattern is a software-design pattern commonly used in object-oriented programming. Analogous to a facade in architecture, a facade is an object that serves as a front-facing interface masking more complex underlying or structural code. Here, it is applied to provide a simplified interface for interacting with Java method RegistrationSrvc.



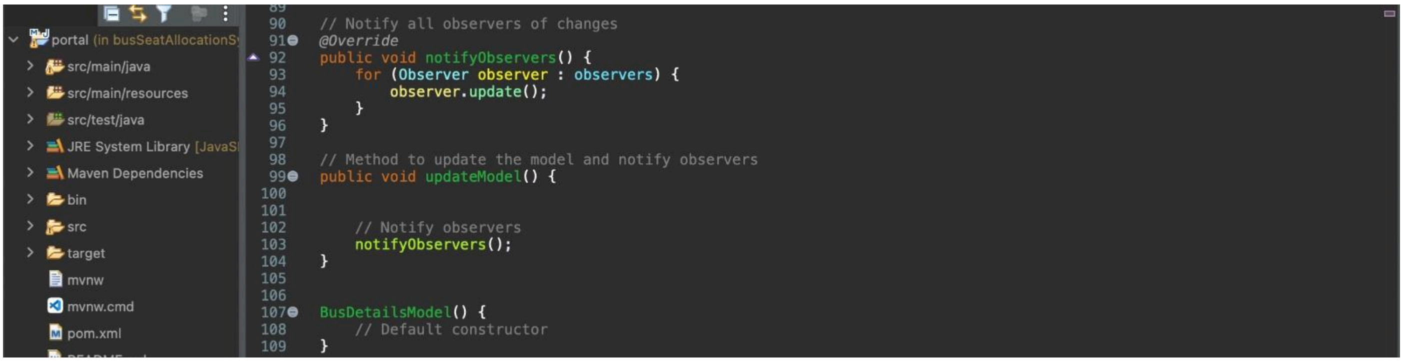
```java
package com.bus.portal.controller;
import com.bus.portal.pojos.RegistrationReqPojo;
import com.bus.portal.pojos.RegistrationRespPojo;
import com.bus.portal.service.RegistrationSrvc;
import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.web.bind.annotation.*;

@RestController
@RequestMapping("/registration")
//Single Responsibility Principle (SRP) is used here as only registration related tasks are dealt -
//- by this class
public class RegistrationControllerFacade {
    // Facade pattern is applied here to provide a simplified interface
    // for interacting with RegistrationSrvc.
    private final RegistrationSrvc registrationService;

    @Autowired
    public RegistrationControllerFacade(RegistrationSrvc registrationService) {
        this.registrationService = registrationService;
    }

    @PostMapping("/single")
    public RegistrationRespPojo registerSingleBus(@RequestBody RegistrationReqPojo regPojo) {
        return registrationService.registersinglebusdetails(regPojo);
    }

    @PostMapping("/check-username")
    public RegistrationRespPojo checkUsername(@RequestBody String username) {
        return registrationService.checkusername(username);
    }
}

```

3) Behavioural Pattern - Observer:

The observer pattern is a software design pattern in which an object, named the subject, maintains a list of its dependents, called observers, and notifies them automatically of any state changes, usually by calling one of their methods. Here, it is used to observe seat bookings and convey booked seats to other methods so as not to cause collision in future bookings.



```java
package com.bus.portal.model;

import java.util.ArrayList;

//The observer design pattern is used in the BusDetailsModel class.
//It makes it easy to track changes in the model without high coupling.
interface Subject {
    void attach(Observer observer);
    void detach(Observer observer);
    void notifyObservers();
}

// Define the Observer interface
interface Observer {
    void update();
}

// Concrete Subject implementation
@Entity
@Table(name = "bus_dtls_mstr")
public class BusDetailsModel implements Subject {
    @Id
    @Column(name = "id", updatable = false, nullable = false)
    private Long id;

    @Column(name = "No_of_Seats", nullable = false)
    private int seats;

    @Column(name = "No_of_Rows", nullable = false)
    private int rows;

    @Column(name = "Left_Side_Seat", nullable = false)
    private int seatleftside;

    @Column(name = "Right_Side_Seat", nullable = false)
    private int seatrightside;

    @Column(name = "Last_Seat")
    private int lastseat;

    @Column(name = "Bus_Name", nullable = false)
    private String busname;

    @Column(name = "Owner_Name", nullable = false)
    private String ownername;

    @Column(name = "Mobile_No", nullable = false)
    private String mobileno;

    @Column(name = "Emailid")
    private String emailid;

    @Column(name = "Endpoint1", nullable = false)
```

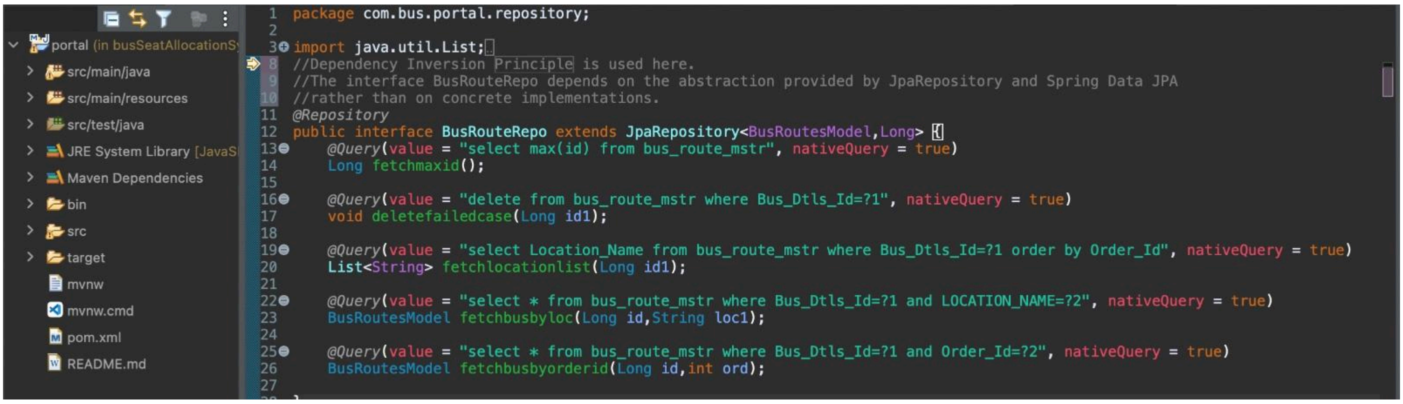


```java
    // Notify all observers of changes
    @Override
    public void notifyObservers() {
        for (Observer observer : observers) {
            observer.update();
        }
    }

    // Method to update the model and notify observers
    public void updateModel() {


        // Notify observers
        notifyObservers();
    }


    BusDetailsModel() {
        // Default constructor
    }
```

8. DESIGN PRINCIPLES:

1) Dependency Inversion Principle:

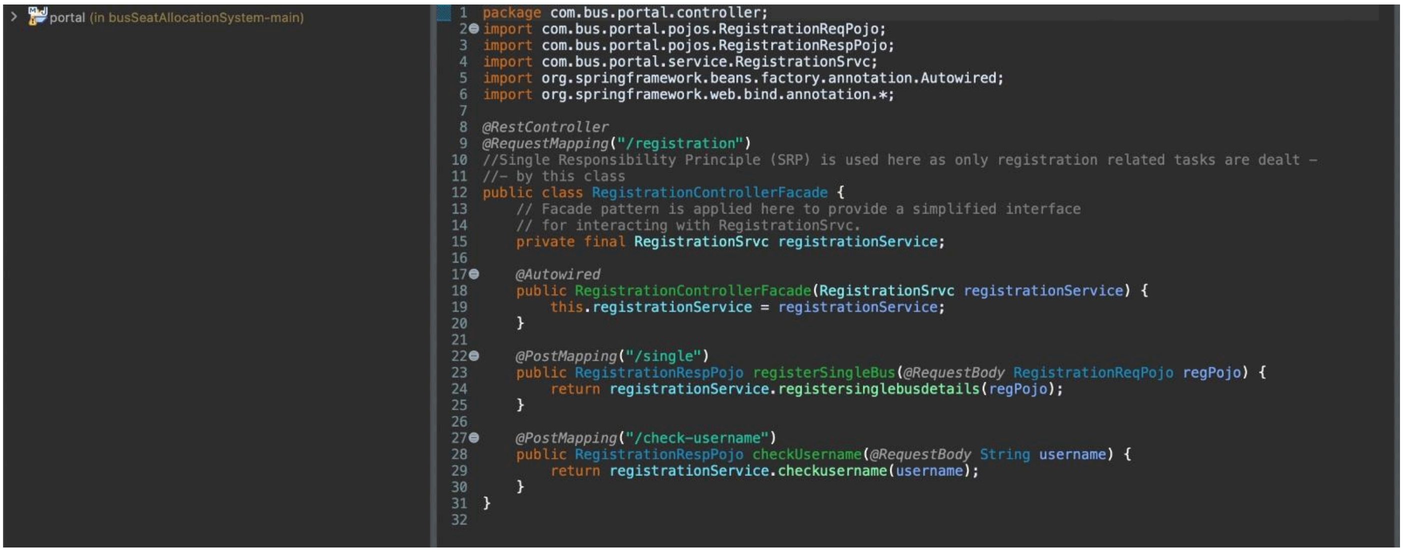
The Dependency Inversion Principle (DIP) states that high level modules should not depend on low level modules; both should depend on abstractions. Abstractions should not depend on details. Here, the interface BusRouteRepo depends on the abstraction provided by JPARepository and SpringDataJPA rather than on one concrete implementation.



```java
package com.bus.portal.repository;

import java.util.List;
//Dependency Inversion Principle is used here.
//The interface BusRouteRepo depends on the abstraction provided by JpaRepository and Spring Data JPA
//rather than on concrete implementations.
@Repository
public interface BusRouteRepo extends JpaRepository<BusRoutesModel,Long> {
    @Query(value = "select max(id) from bus_route_mstr", nativeQuery = true)
    Long fetchmaxid();

    @Query(value = "delete from bus_route_mstr where Bus_Dtls_Id=?1", nativeQuery = true)
    void deletefailedcase(Long id1);

    @Query(value = "select Location_Name from bus_route_mstr where Bus_Dtls_Id=?1 order by Order_Id", nativeQuery = true)
    List<String> fetchlocationlist(Long id1);

    @Query(value = "select * from bus_route_mstr where Bus_Dtls_Id=?1 and LOCATION_NAME=?2", nativeQuery = true)
    BusRoutesModel fetchbusbyloc(Long id,String loc1);

    @Query(value = "select * from bus_route_mstr where Bus_Dtls_Id=?1 and Order_Id=?2", nativeQuery = true)
    BusRoutesModel fetchbusbyorderid(Long id,int ord);

}
```

2) Single Responsibility Principle:

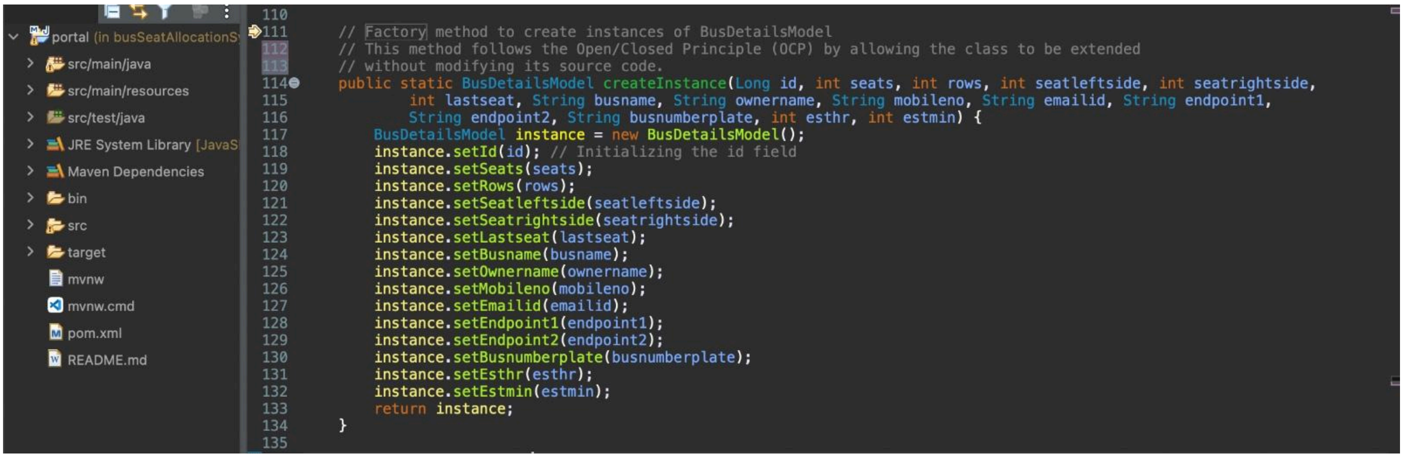
The single responsibility principle is a computer programming principle that states that "A module should be responsible to one, and only one, actor." Here, only registration tasks are dealt with by RegistrationSrvc class.



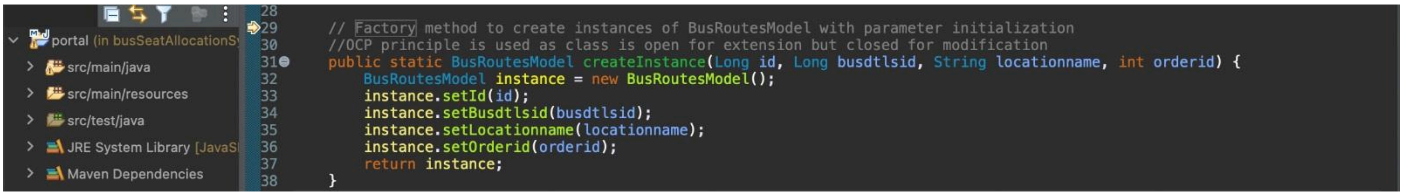
```java
package com.bus.portal.controller;
import com.bus.portal.pojos.RegistrationReqPojo;
import com.bus.portal.pojos.RegistrationRespPojo;
import com.bus.portal.service.RegistrationSrvc;
import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.web.bind.annotation.*;

@RestController
@RequestMapping("/registration")
//Single Responsibility Principle (SRP) is used here as only registration related tasks are dealt -
//- by this class
public class RegistrationControllerFacade {
    // Facade pattern is applied here to provide a simplified interface
    // for interacting with RegistrationSrvc.
    private final RegistrationSrvc registrationService;

    @Autowired
    public RegistrationControllerFacade(RegistrationSrvc registrationService) {
        this.registrationService = registrationService;
    }

    @PostMapping("/single")
    public RegistrationRespPojo registerSingleBus(@RequestBody RegistrationReqPojo regPojo) {
        return registrationService.registersinglebusdetails(regPojo);
    }

    @PostMapping("/check-username")
    public RegistrationRespPojo checkUsername(@RequestBody String username) {
        return registrationService.checkusername(username);
    }
}
```

3) Open-closed Principle:

In object-oriented programming, the open–closed principle states "software entities should be open for extension, but closed for modification"; that is, such an entity can allow its behaviour to be extended without modifying its source code. Here, BusDetailsModel and BusRoutesModel can be extended without modifying original code.

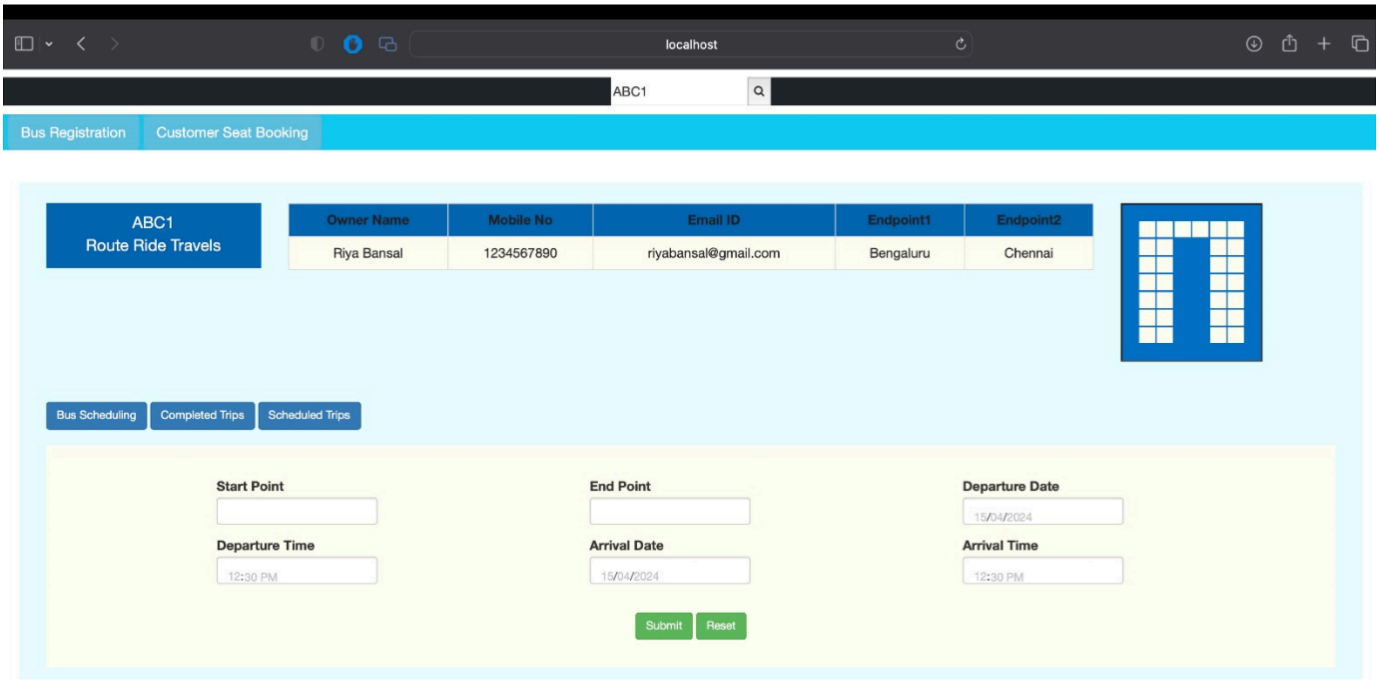


```java

    // Factory method to create instances of BusDetailsModel
    // This method follows the Open/Closed Principle (OCP) by allowing the class to be extended
    // without modifying its source code.
    public static BusDetailsModel createInstance(Long id, int seats, int rows, int seatleftside, int seatrightside,
            int lastseat, String busname, String ownername, String mobileno, String emailid, String endpoint1,
            String endpoint2, String busnumberplate, int esthr, int estmin) {
        BusDetailsModel instance = new BusDetailsModel();
        instance.setId(id); // Initializing the id field
        instance.setSeats(seats);
        instance.setRows(rows);
        instance.setSeatleftside(seatleftside);
        instance.setSeatrightside(seatrightside);
        instance.setLastseat(lastseat);
        instance.setBusname(busname);
        instance.setOwnername(ownername);
        instance.setMobileno(mobileno);
        instance.setEmailid(emailid);
        instance.setEndpoint1(endpoint1);
        instance.setEndpoint2(endpoint2);
        instance.setBusnumberplate(busnumberplate);
        instance.setEsthr(esthr);
        instance.setEstmin(estmin);
        return instance;
    }

```



```java

    // Factory method to create instances of BusRoutesModel with parameter initialization
    //OCP principle is used as class is open for extension but closed for modification
    public static BusRoutesModel createInstance(Long id, Long busdtlsid, String locationname, int orderid) {
        BusRoutesModel instance = new BusRoutesModel();
        instance.setId(id);
        instance.setBusdtlsid(busdtlsid);
        instance.setLocationname(locationname);
        instance.setOrderid(orderid);
        return instance;
    }
```

9. SAMPLE OUTPUT DEMO SCREENSHOTS:



ABC1
Route Ride Travels

Owner Name	Mobile No	Email ID	Endpoint1	Endpoint2
Riya Bansal	1234567890	riyabansal@gmail.com	Bengaluru	Chennai

Start Point

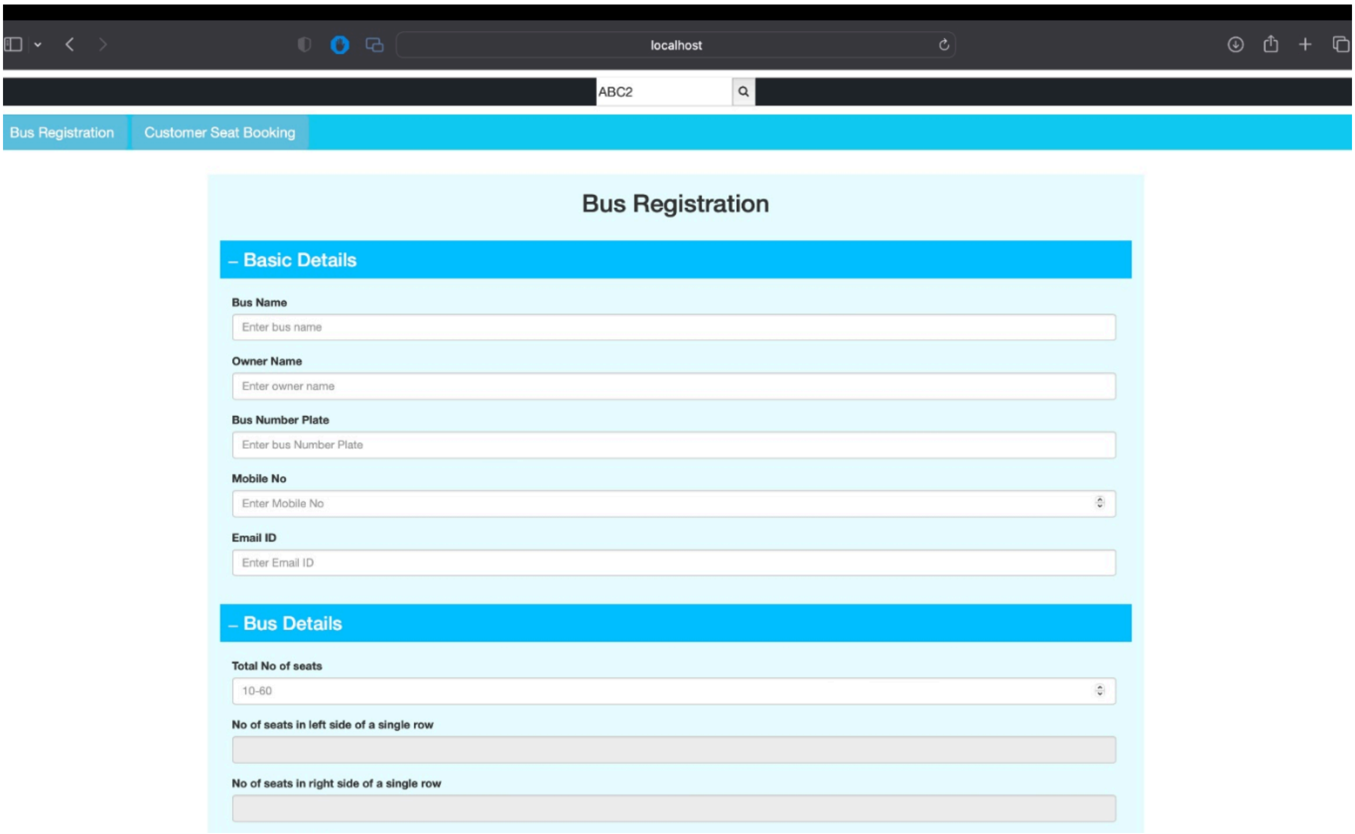
End Point

Departure Date: 15/04/2024

Departure Time: 12:30 PM

Arrival Date: 15/04/2024

Arrival Time: 12:30 PM



Bus Registration

– Basic Details

Bus Name: Enter bus name

Owner Name: Enter owner name

Bus Number Plate: Enter bus Number Plate

Mobile No: Enter Mobile No

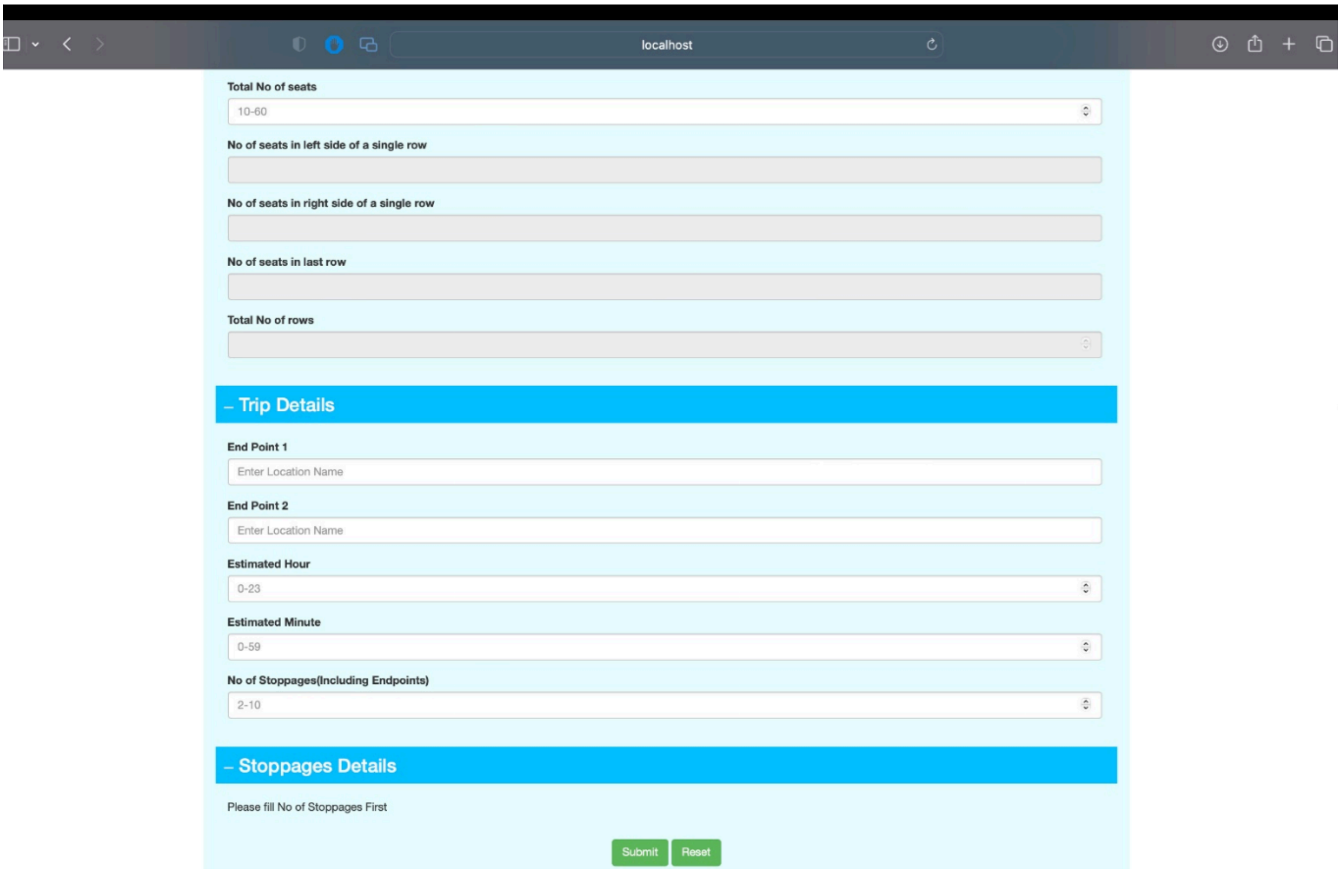
Email ID: Enter Email ID

– Bus Details

Total No of seats: 10-60

No of seats in left side of a single row

No of seats in right side of a single row



Total No of seats

10-60

No of seats in left side of a single row

No of seats in right side of a single row

No of seats in last row

Total No of rows

– Trip Details

End Point 1

Enter Location Name

End Point 2

Enter Location Name

Estimated Hour

0-23

Estimated Minute

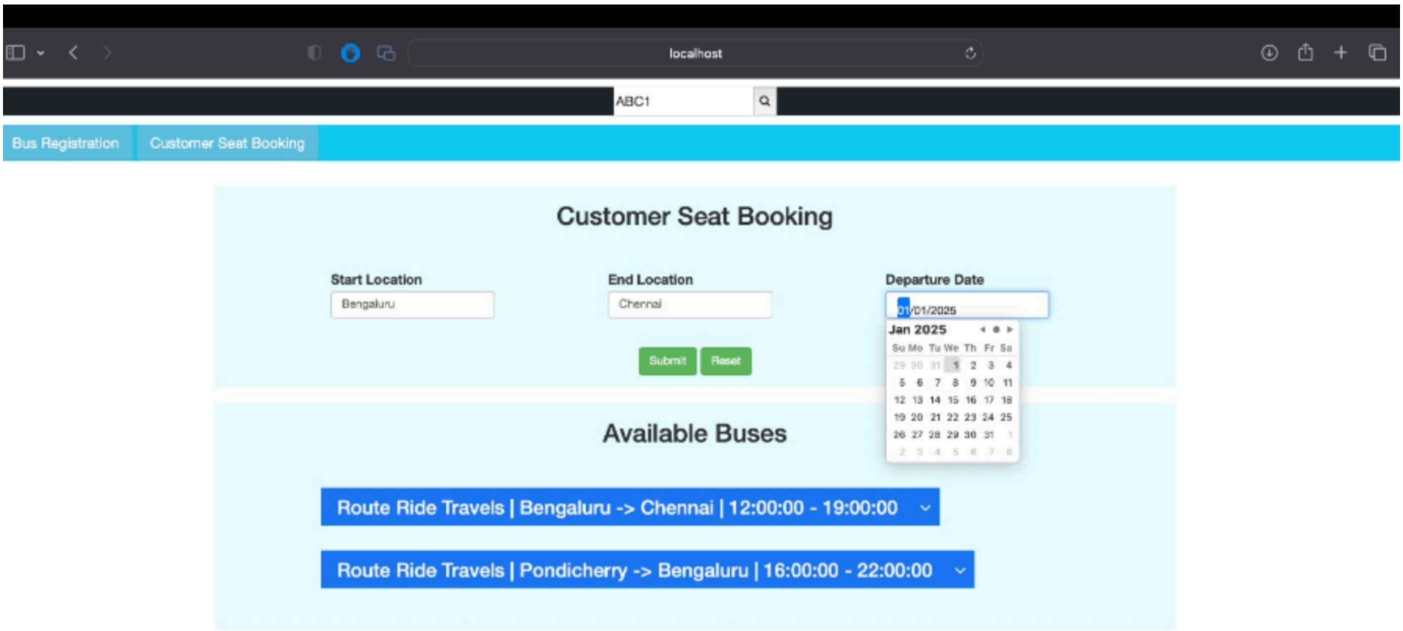
0-59

No of Stoppages(Including Endpoints)

2-10

– Stoppages Details

Please fill No of Stoppages First



Customer Seat Booking

Start Location

Bengaluru

End Location

Chennai

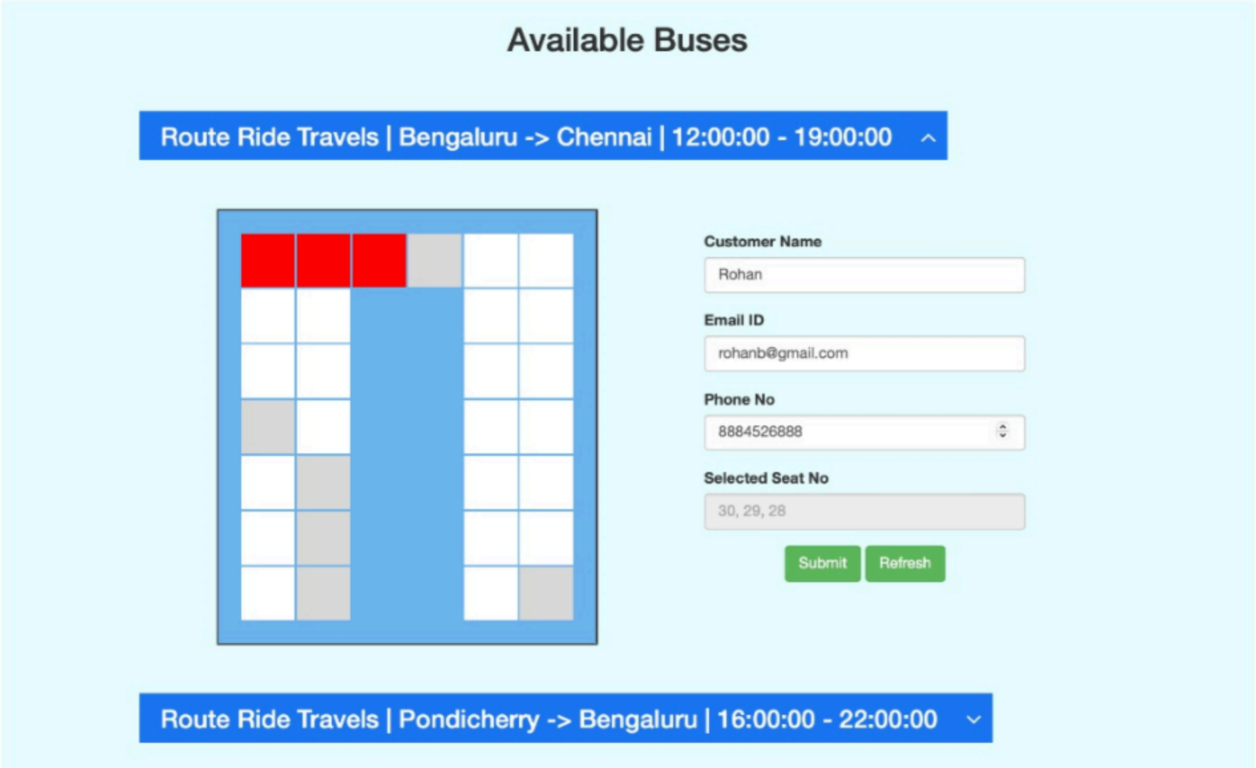
Departure Date

01/01/2025

Available Buses

Route Ride Travels | Bengaluru -> Chennai | 12:00:00 - 19:00:00

Route Ride Travels | Pondicherry -> Bengaluru | 16:00:00 - 22:00:00



Available Buses

Route Ride Travels | Bengaluru -> Chennai | 12:00:00 - 19:00:00

Customer Name: Rohan

Email ID: rohanb@gmail.com

Phone No: 8884526888

Selected Seat No: 30, 29, 28

Route Ride Travels | Pondicherry -> Bengaluru | 16:00:00 - 22:00:00

ABC2

Pondicherry -> Bengaluru

Seat 1:

Seat 2:

Bengaluru - Pune -> Nikhil Pandey (Mobile No : 2534567890)

Seat 3:

Seat 4:

Bengaluru - Pune -> Nikhil Pandey (Mobile No : 2534567890)

Seat 5:

Seat 6:

Seat 7:

Seat 8:

Seat 9:

Bengaluru - Pune -> Meeta Kumar (Mobile No : 2134567890)

Seat 10:

Seat 11:

Seat 12:

Seat 13:

Seat 14:

Bengaluru - Pune -> Meeta Kumar (Mobile No : 2134567890)

Seat 15:

Seat 16:

Bengaluru - Chennai -> Abhinav Rathod (Mobile No : 2234578901)

Chennai - Pune -> Nina B (Mobile No : 2134567267)

****