UE21CS352B - Object Oriented Analysis & Design using Java

Mini Project Report

Airline Booking System

Submitted by:

Namratha C	**PES1UG21CS359**
Navaneetha N	**PES1UG21CS365**
Nandini B S	**PES1UG21CS744**
M Pavan Prathap	**PES1UG21CS318**

6th Semester F Section

Prof. Bhargavi Mokashi

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

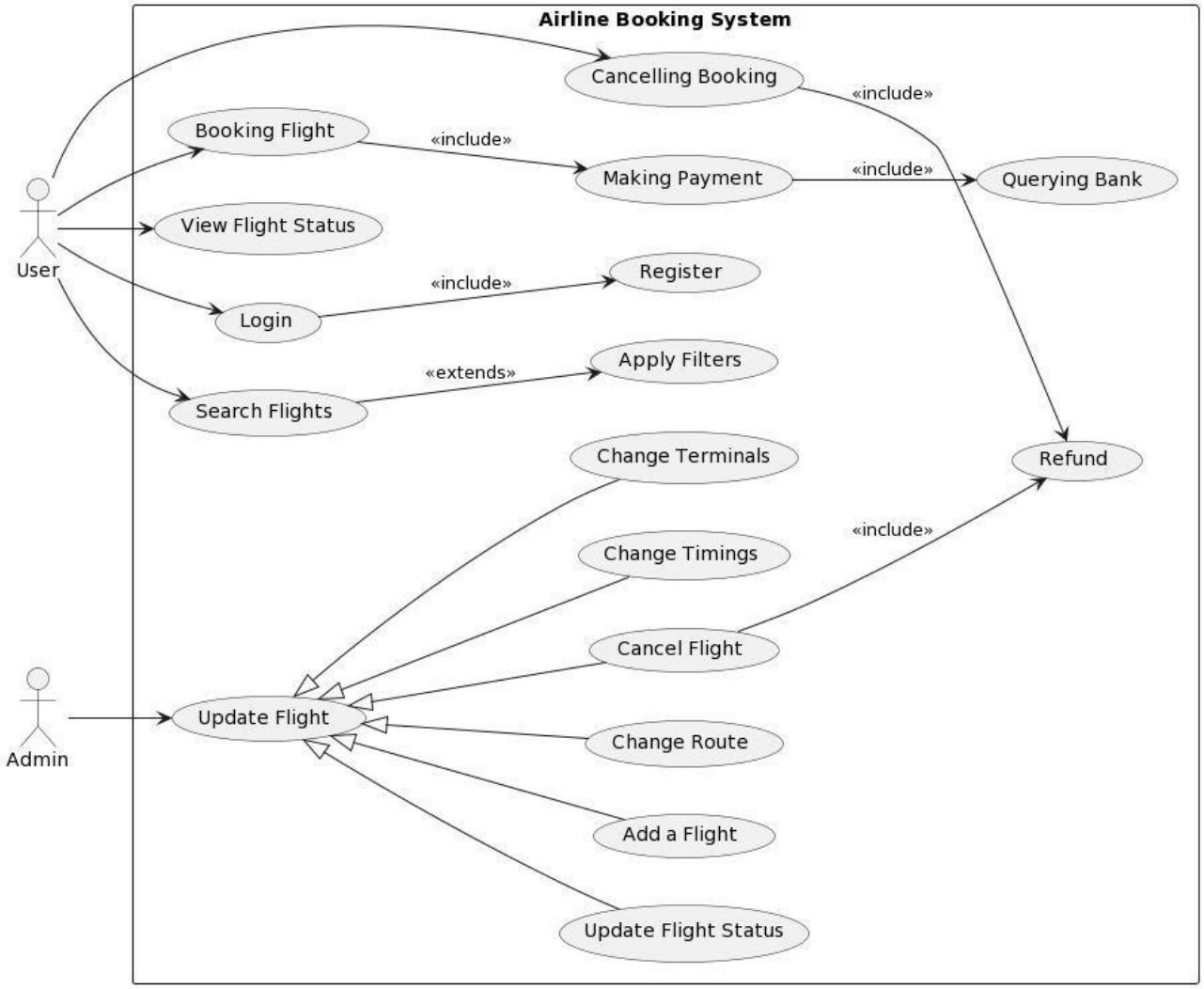
TABLE OF CONTENTS:

ABSTRACT & INTRODUCTION:

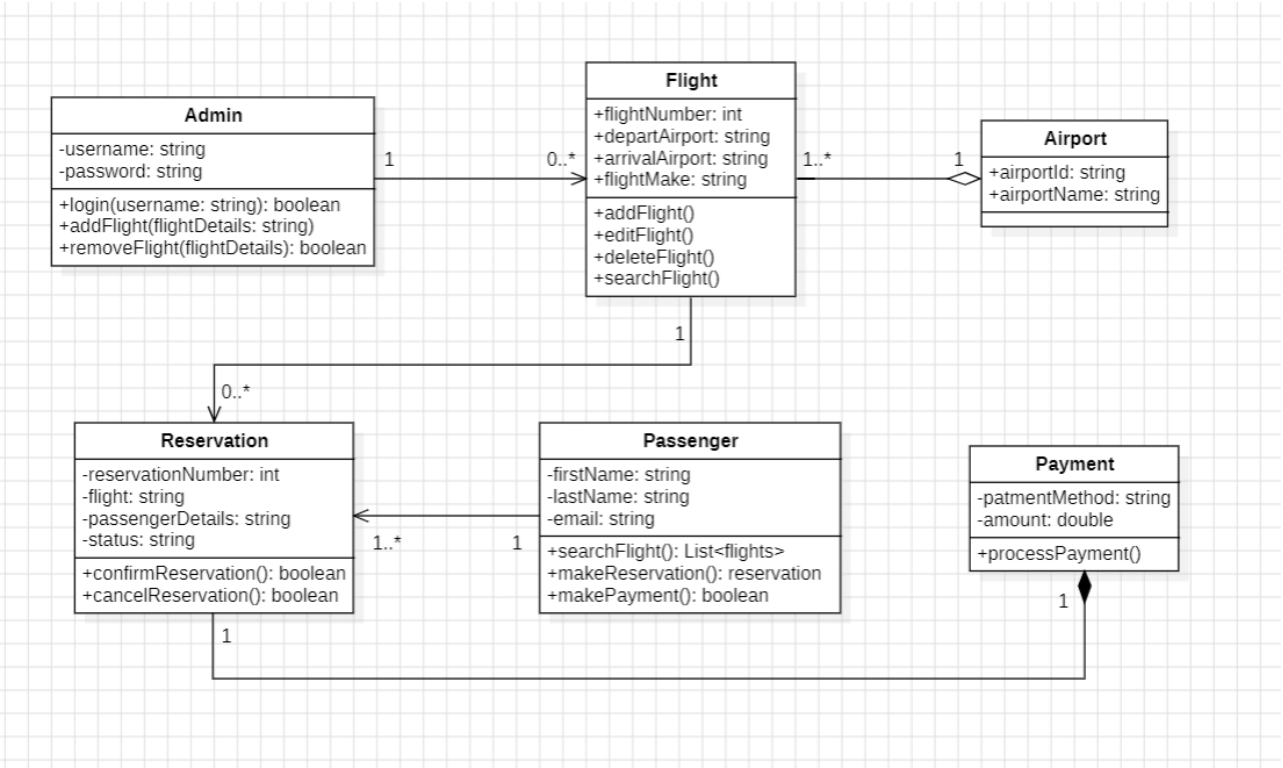
The airline booking system is a web based application involves comprehensive software solution designed to facilitate the reservation and management of flight tickets for passengers. It serves as a central platform where travelers can search, compare, and book flights to their desired destinations. Additionally, the system streamlines the administrative tasks for airlines, including seat allocation, scheduling, and ticketing.

With features such as online check-in, payment processing, and itinerary management, the airline booking system enhances the overall travel experience for both customers and airlines alike, making it an indispensable tool in the modern aviation industry.

USE CASE DIAGRAM:

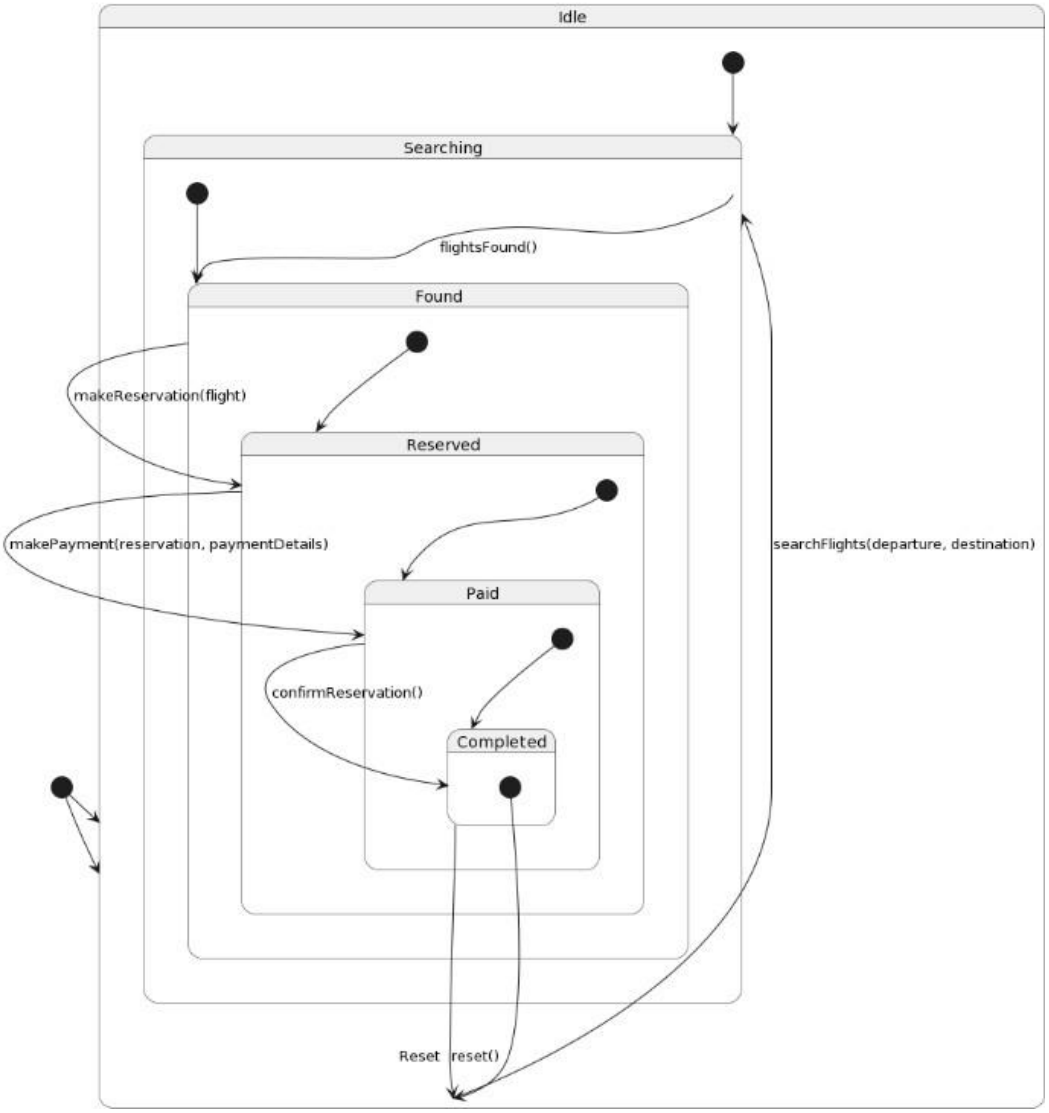


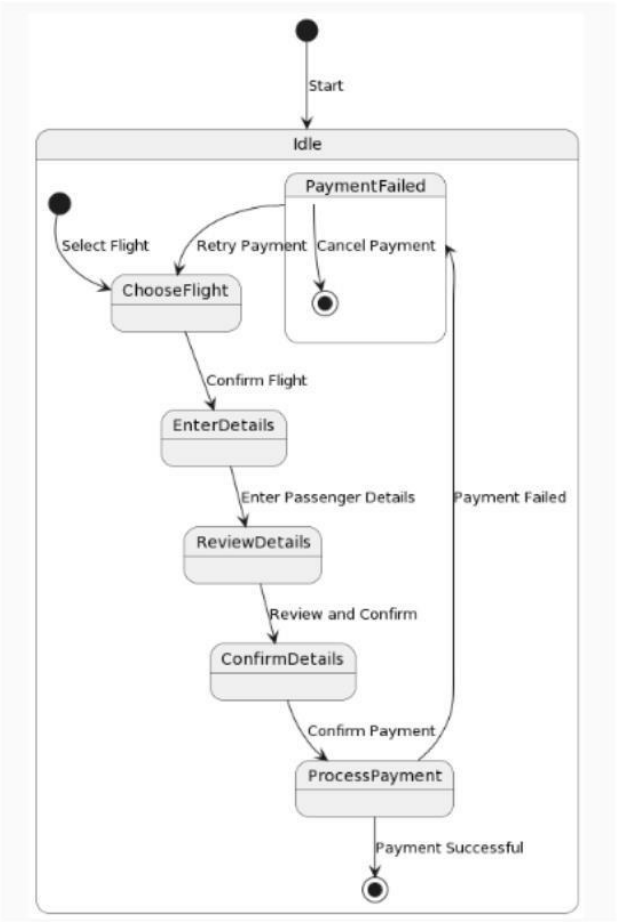
CLASS DIAGRAM:

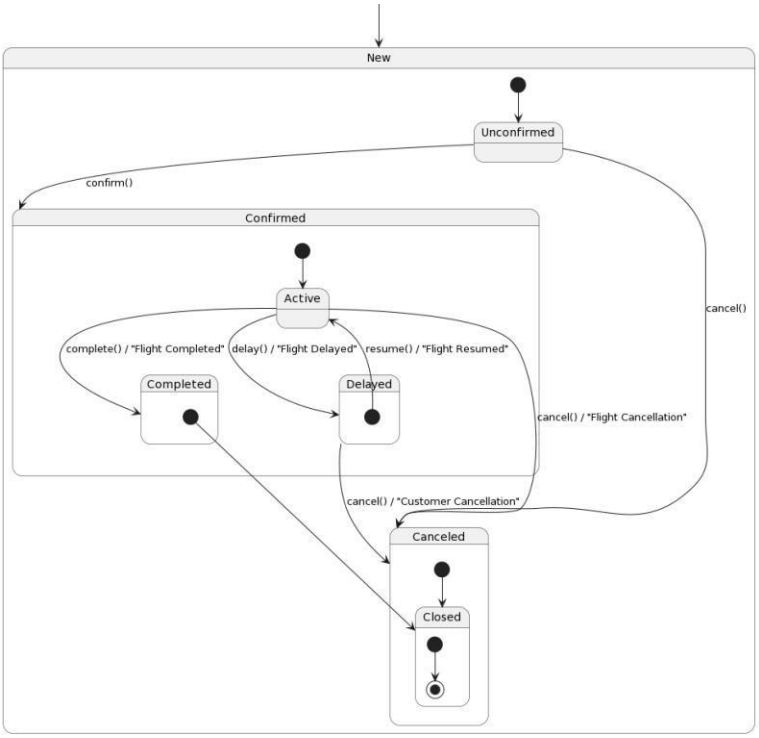


STATE DIAGRAMS:

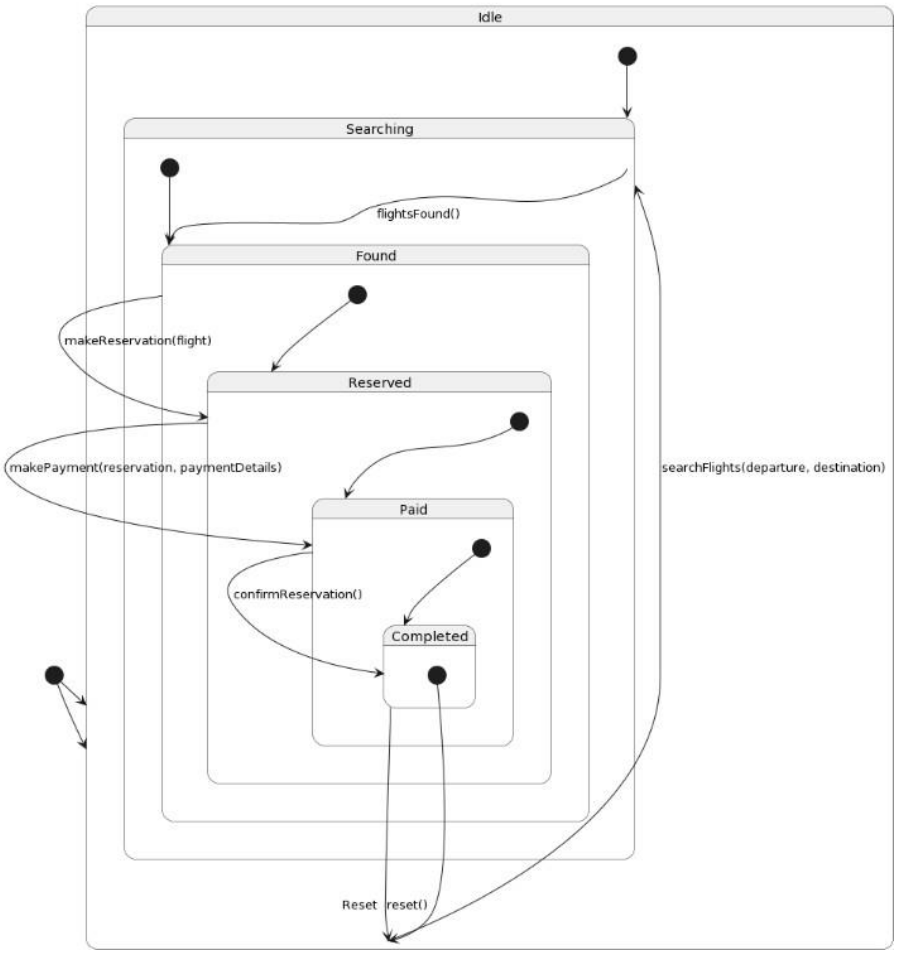
State Diagram(Passenger Class):




Start
Idle
PaymentFailed
Select Flight
Retry Payment
Cancel Payment
ChooseFlight
Confirm Flight
EnterDetails
Enter Passenger Details
Payment Failed
ReviewDetails
Review and Confirm
ConfirmDetails
Confirm Payment
ProcessPayment
Payment Successful

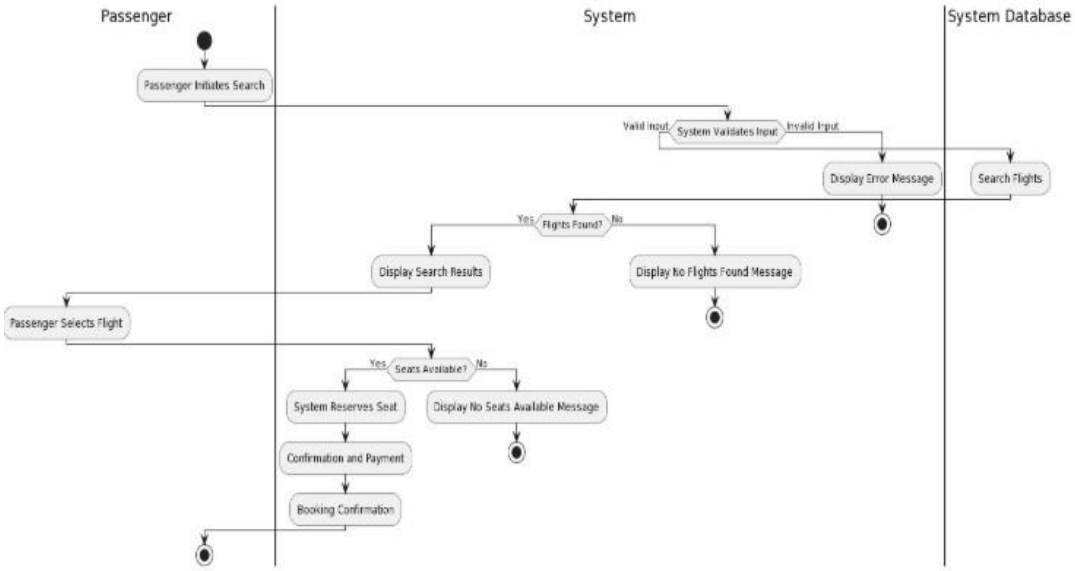

New
Unconfirmed
confirm()
Confirmed
Active
cancel()
complete() / "Flight Completed"
delay() / "Flight Delayed"
resume() / "Flight Resumed"
Completed
Delayed
cancel() / "Flight Cancellation"
cancel() / "Customer Cancellation"
Canceled
Closed

State Diagram(Passenger Class):

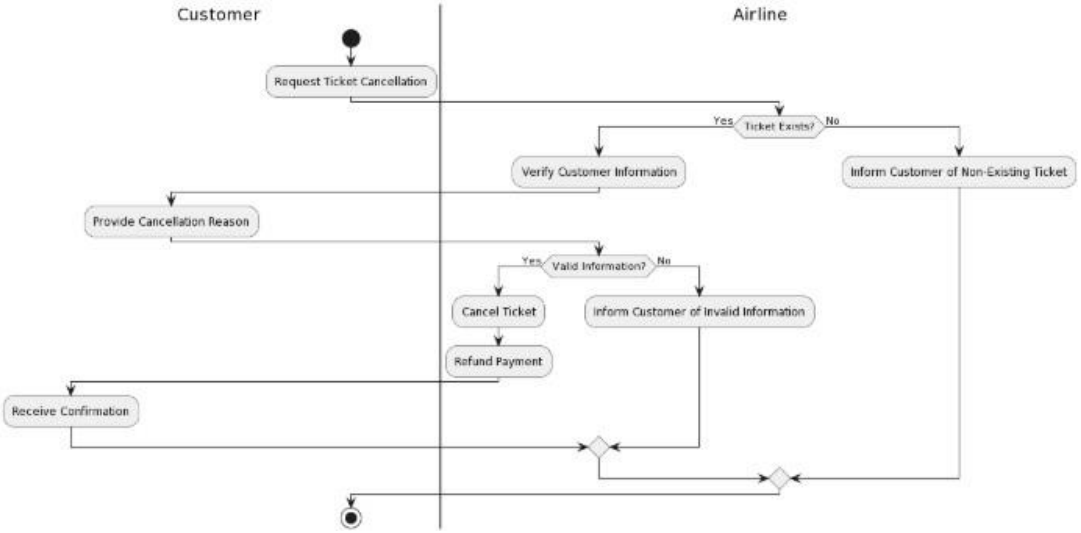


ACTIVITY DIAGRAMS:

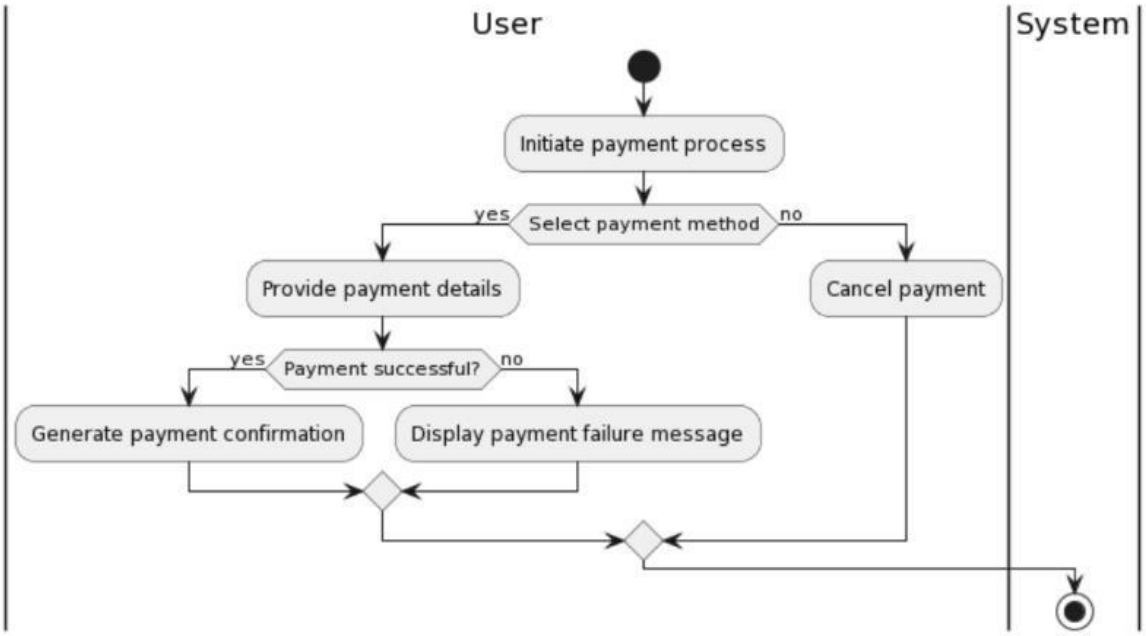
Activity Diagram For Book Flight :



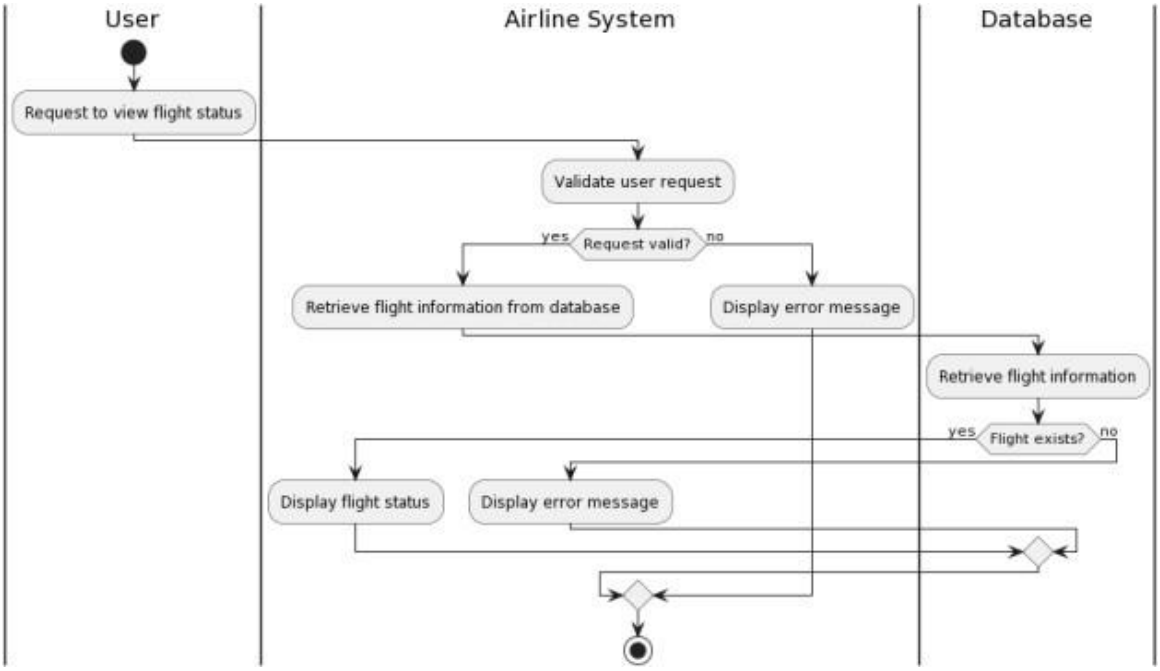
Activity Diagram for Canceling a booked ticket:



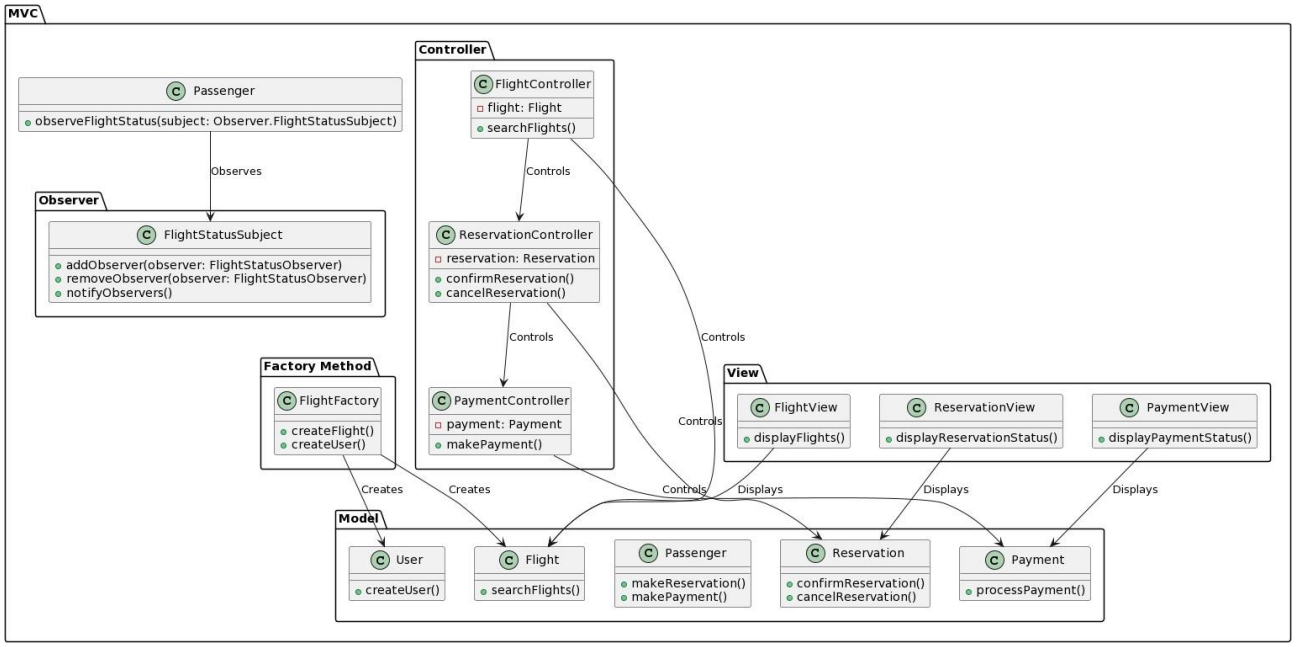
ACTIVITY DIAGRAM FOR THE USE CASE MAKING PAYMENT :



ACTIVITY DIAGRAM FOR FLIGHT STATUS



EXTENDED CLASS DIAGRAM WITH MVC ARCHITECTURE AND PATTERNS:



DESIGN PATTERNS:

1. Factory Pattern

```
data.sql | FlightController.java | build_model.py 2 | FlightFactory.java | Flight.java | NewsFeedController.java | NewsFeedService.java
src > main > java > com > cognizant > airline_ticket_reservation_system > controller > FlightFactory.java > ...
package com.cognizant.airline_ticket_reservation_system.controller;

import com.cognizant.airline_ticket_reservation_system.model.Flight;

public class FlightFactory {
    public static Flight createFlight(String airline, String model, String type, Integer economySeats, Integer businessSeats) {
        Flight flight = new Flight();
        flight.setAirline(airline);
        flight.setModel(model);
        flight.setType(type);
        flight.setEconomySeats(economySeats);
        flight.setBusinessSeats(businessSeats);
        return flight;
    }
}
```

```
    @PostMapping("/admin/admin-home/manage-flight/add-flight")
public ModelAndView addFlight(
        @Valid @ModelAttribute("flight") Flight flight,
        BindingResult bindingResult,
        ModelAndView modelAndView,
        @Value("${flight.additionSuccessful}") String message
        ) {
    if (bindingResult.hasErrors()) {
        modelAndView.setViewName("admin/admin_home/manage_flight/add-flight");
        return modelAndView;
    }


    Flight newFlight = FlightFactory.createFlight(flight.getAirline(), flight.getModel(), flight.getType(), flight.getEconomySeats()


    flightService.addFlight(newFlight);
    modelAndView.setViewName(String.format(format:"redirect:/admin/admin-home/manage-flight?msg=%s", message));

    return modelAndView;
}
```

The Factory pattern is employed to create flights in the airline booking system, offering a centralized method for constructing diverse flight instances with varying parameters. It promotes code flexibility, simplifies flight object creation, and enhances maintainability by encapsulating the instantiation process within a dedicated factory class.

2. Singleton Pattern

```java
@Autowired
public void setFlightScheduleService(FlightScheduleService flightScheduleService) {
    this.flightScheduleService = flightScheduleService;
}

public static synchronized FlightScheduleController getInstance() {
    if (instance == null) {
        instance = new FlightScheduleController();
    }
    return instance;
}
```

```java
@Autowired
public void setPassengerService(PassengerService passengerService) {
    this.passengerService = passengerService;
}

public static synchronized PaymentController getInstance() {
    if (instance == null) {
        instance = new PaymentController();
    }
    return instance;
}
```

Implementing the Singleton pattern in the Payment Controller and Flight Schedule Controller ensures that only one instance of each controller exists throughout the application, guaranteeing centralized access. This approach simplifies resource management, enhances consistency, and promotes scalability by preventing multiple instances from conflicting.

3. Observer Pattern

```java
@PostMapping("/admin/admin-home/add-news")
public ModelAndView addNews(
        @Valid @ModelAttribute("newsFeed") NewsFeed newsFeed,
        BindingResult bindingResult,
        ModelAndView modelAndView
) {
    if (bindingResult.hasErrors()) {
        modelAndView.setViewName("admin/admin_home/add-news");
        return modelAndView;
    }

    // Save news in the database
    newsFeedService.saveNewsFeed(newsFeed);
    modelAndView.setViewName("redirect:/admin/admin-home/add-news");

    // Publish the news (dummy method)
    publishNews(newsFeed);

    return modelAndView;
}

public void publishNews(NewsFeed newsFeed) {
    System.out.println("News published: " + newsFeed);
}

private void subscribeToNews() {
    System.out.println(x:"Subscribed to news feed");
}

@ModelAttribute("newsFeeds")
public List<NewsFeed> newsFeeds() {
    // Subscribe to news feed
    subscribeToNews();

    return newsFeedService.getNewsFeeds()
            .stream()
            .sorted((n1, n2) -> -n1.getDate().compareTo(n2.getDate()))
            .collect(Collectors.toList());
}
```

The Observer pattern is utilized to broadcast news from admin to users in the airline booking system, establishing a one-to-many relationship. Admins publish updates, such as flight changes or promotions, while users subscribe to receive notifications, ensuring loose coupling and scalability.

4. Command Pattern

BookingDetailsController.java

src > main > java > com > cognizant > airline_ticket_reservation_system > controller > BookingDetailsController.java > BookingDetailsController > bookingDetails(St

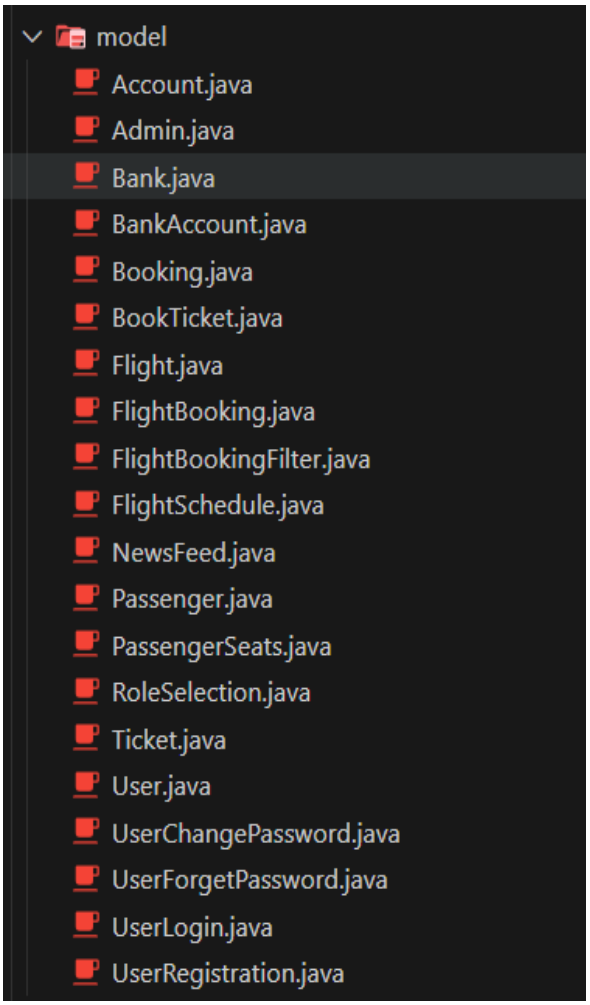
```java
public class BookingDetailsController {
    }

    @Autowired
    public void setUserService(UserService userService) {
        this.userService = userService;
    }

    @GetMapping("/admin/admin-home/booking-details")
    public ModelAndView bookingDetails(
            @ModelAttribute("flightBookingFilter") FlightBookingFilter flightBookingFilter,   //Command Pattern is used here
            ModelAndView modelAndView
    ) {
        List<FlightBooking> flightBookings = flightBookingService.getFlightBookings();
        flightBookings.forEach(i -> {
            i.setFlightSchedule(flightScheduleService.getFlightScheduleById(i.getScheduledFlightId()));
            i.getFlightSchedule().setFlight(flightService.getFlightByNo(i.getFlightSchedule().getFlightNo()));
        });

        modelAndView.addObject(attributeName:"flightBookings", flightBookings);
        modelAndView.setViewName(viewName:"admin/admin_home/booking-details");

        return modelAndView;
    }

    @PostMapping("/admin/admin-home/booking-details")
    public ModelAndView bookingDetails(
            @RequestParam(value = "msg", required = false) String message,
            @ModelAttribute("flightBookingFilter") FlightBookingFilter flightBookingFilter,   //Command pattern is used here
            ModelAndView modelAndView
    ) {
```

The Command pattern is utilized in fetching booking details to encapsulate request parameters and actions, enabling easy execution, logging, and undo functionalities. It decouples the requester from the receiver, enhancing modularity and extensibility while facilitating complex operations on booking data with minimal code changes.

DESIGN PRINCIPLES:

1. Single Responsibility Principle:

In our project, the Single Responsibility Principle is adhered to by designing each class to handle a single part of the functionality.



2. Open/Closed Principle:

If new admin methods are needed, instead of modifying these existing methods, we would extend the service.

```
@GetMapping("/admin/admin-home/user-details")
public ModelAndView userDetails(ModelAndView modelAndView) {
    List<User> userList = userService.getUsers();
    modelAndView.addObject("userList", userList);
    modelAndView.setViewName("admin/admin_home/user-details");

    return modelAndView;
}

@ModelAttribute("newsFeeds")
public List<NewsFeed> newsFeeds() {
    return newsFeedService.getNewsFeeds()
            .stream()
            .filter(newsFeed -> newsFeed.getDate().compareTo(LocalDate.now()) >= 0)
            .collect(Collectors.toList());
}
```

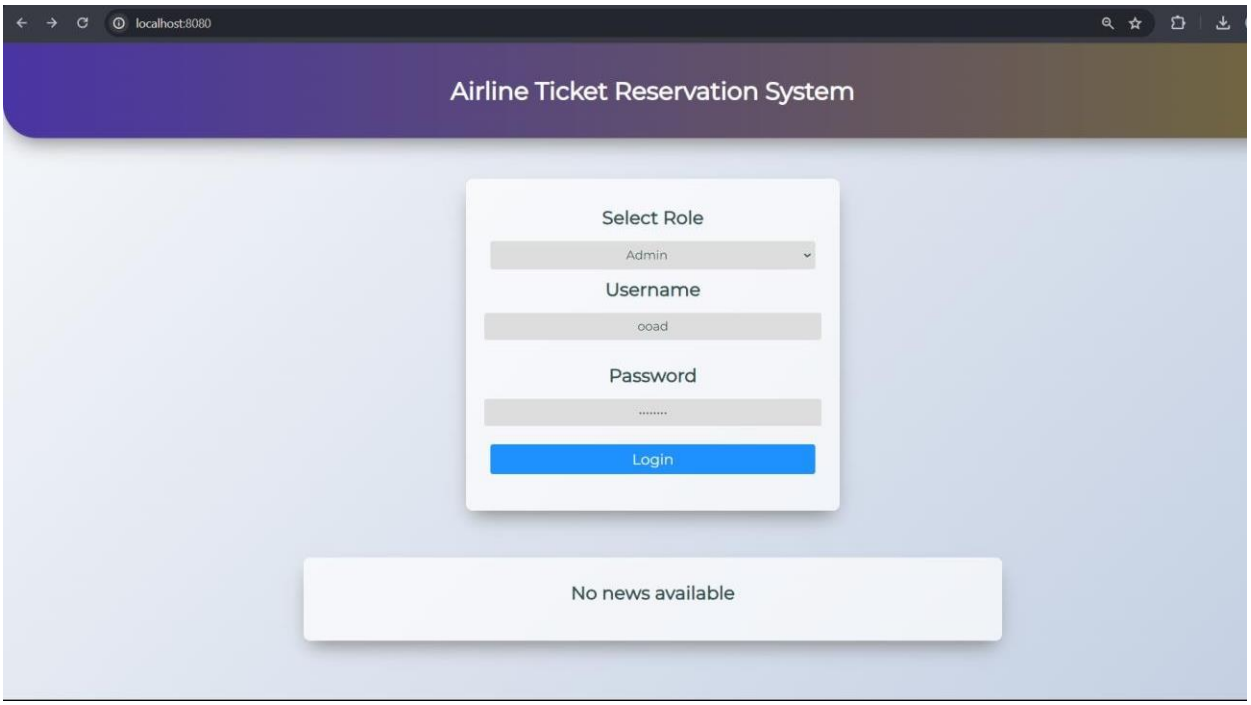
3. Liskov Substitution Principle:

In this code snippet, the calculate_cost method in DomesticFlight calculates the cost based on a fixed rate per kilometer, while the calculate_cost method in InternationalFlight adds an additional fee to a base cost for international flights. The calculate_total_cost function demonstrates how we can use these classes interchangeably, adhering to the Liskov Substitution Principle.

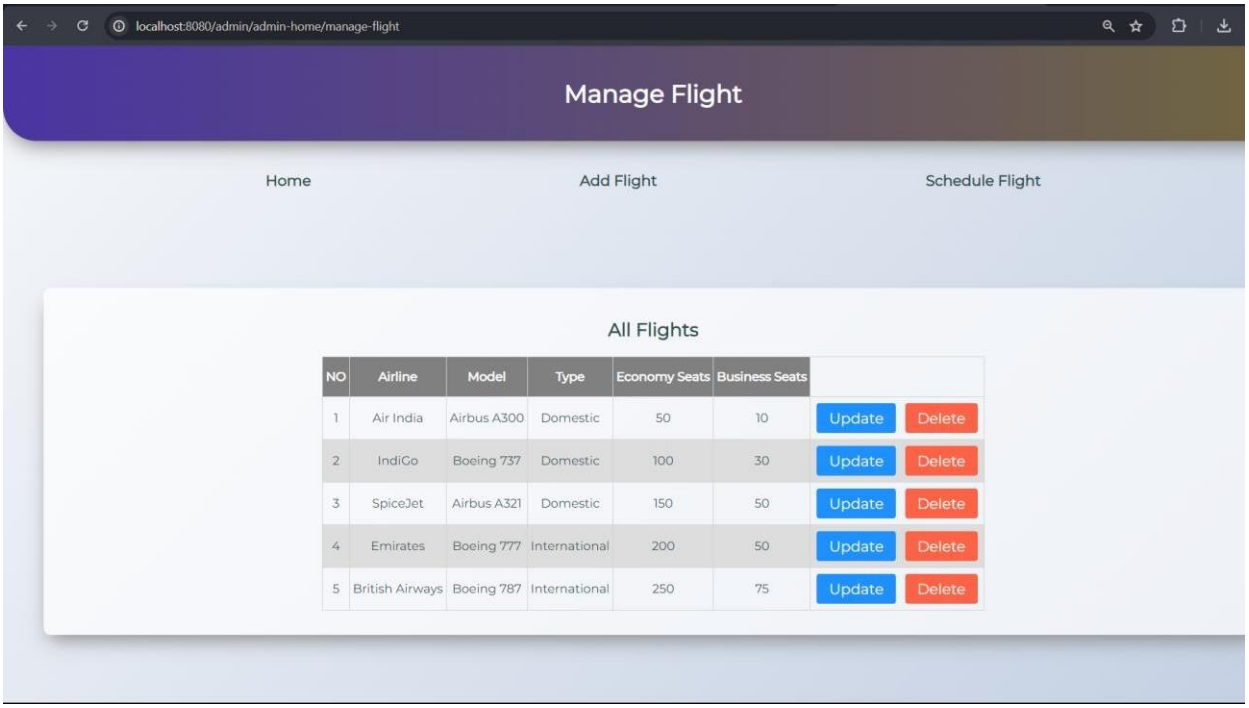
```
class Flight:
    def calculate_cost(self, distance):
        pass

class DomesticFlight(Flight):
    def calculate_cost(self, distance):
        cost_per_km = 0.1
        return distance * cost_per_km

class InternationalFlight(Flight):
    def calculate_cost(self, distance):
        base_cost = 200
        additional_fee_per_km = 0.15
        return base_cost + (distance * additional_fee_per_km)
```

SAMPLE DEMO SCREENSHOTS:

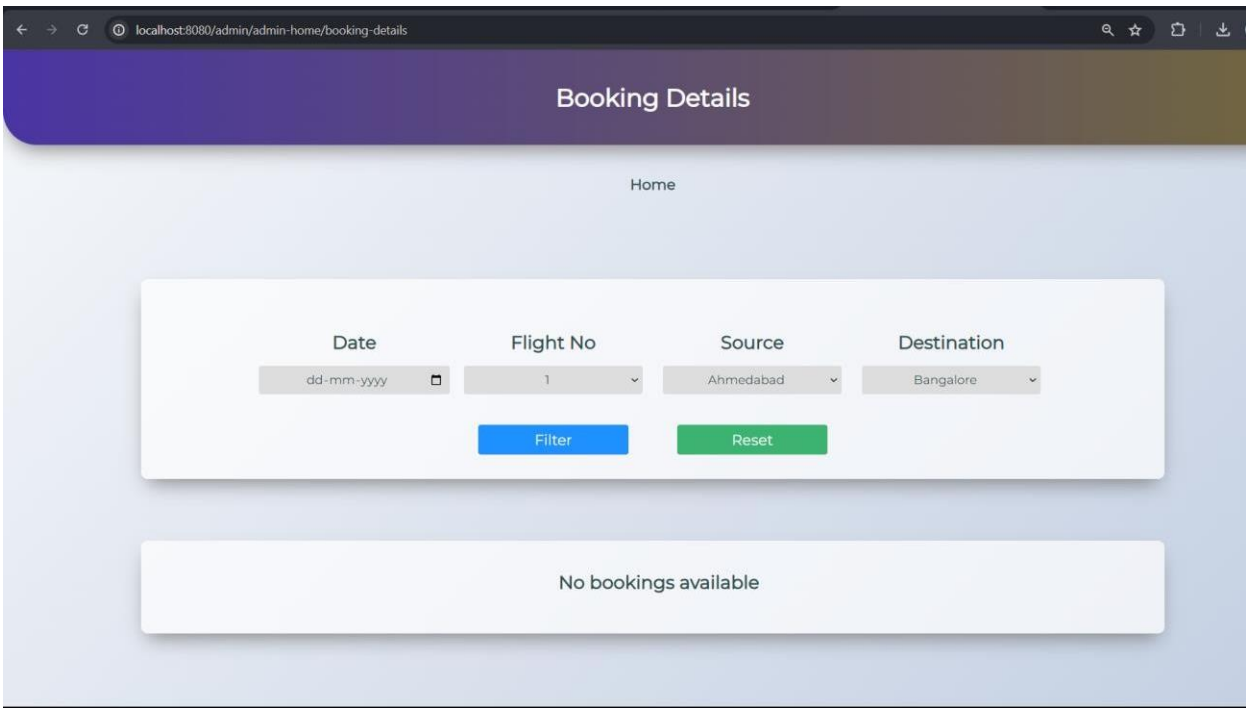
Login Page:



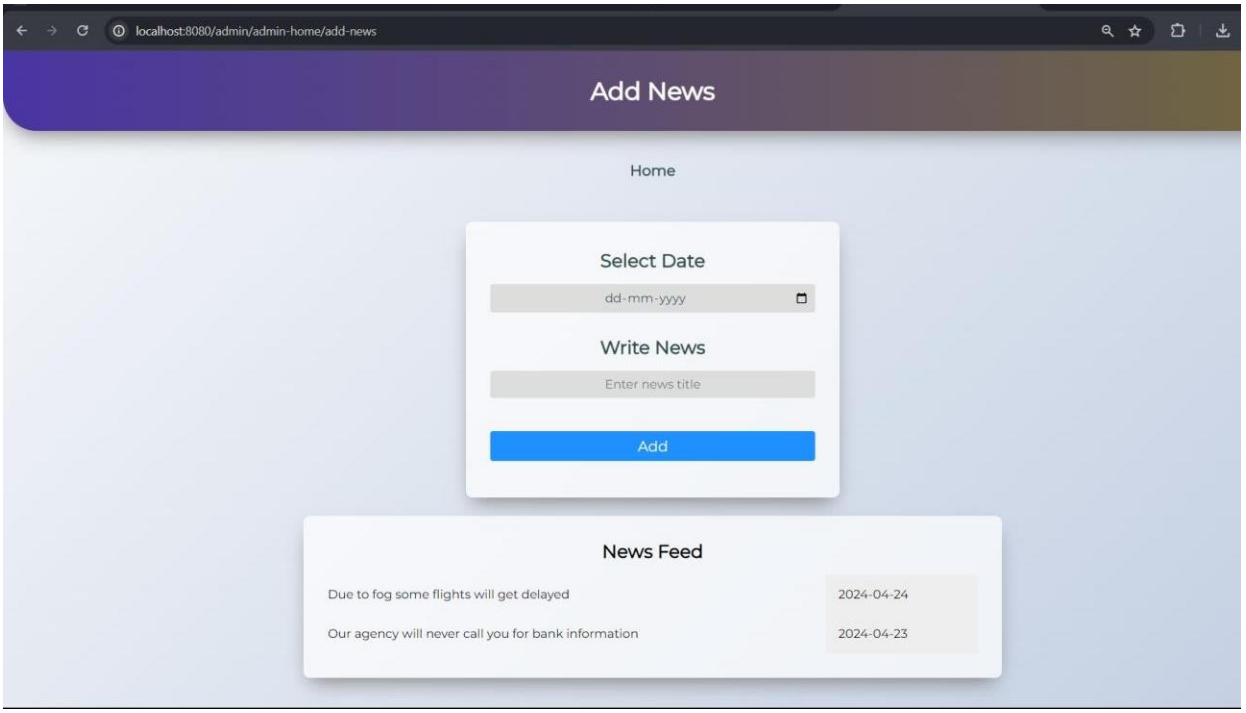
Manage Flight (Admin):



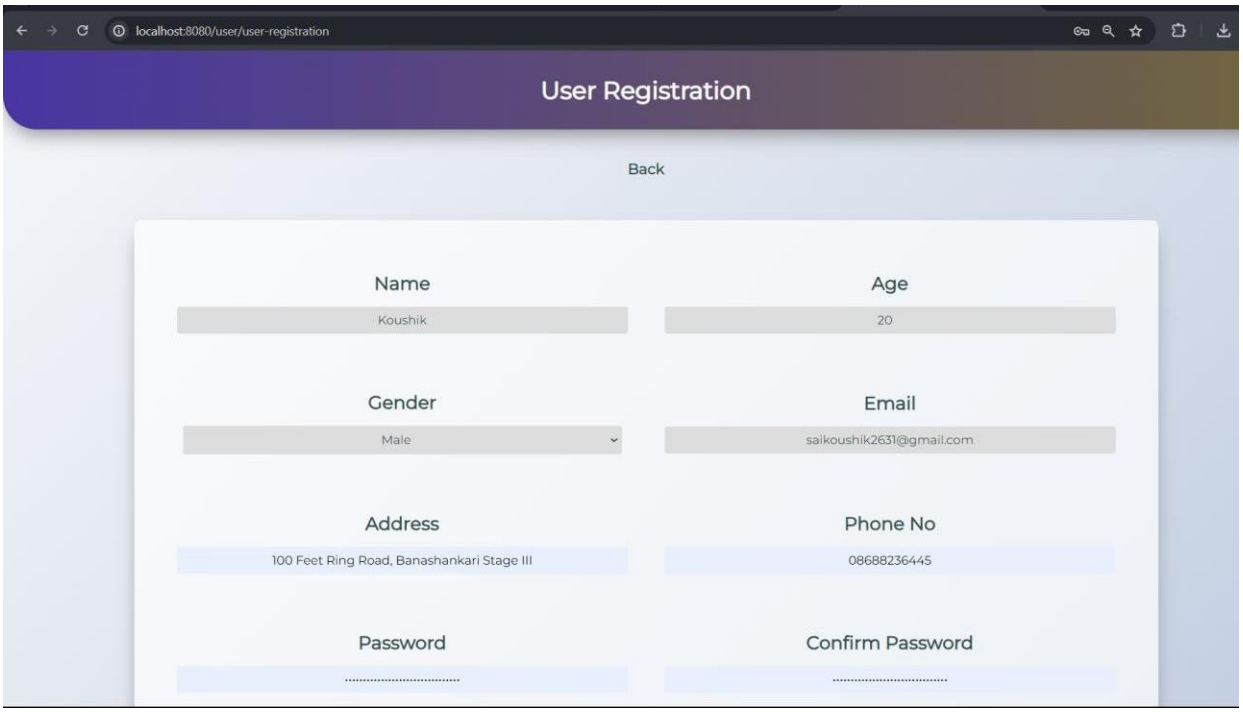
Booking Details (Admin):



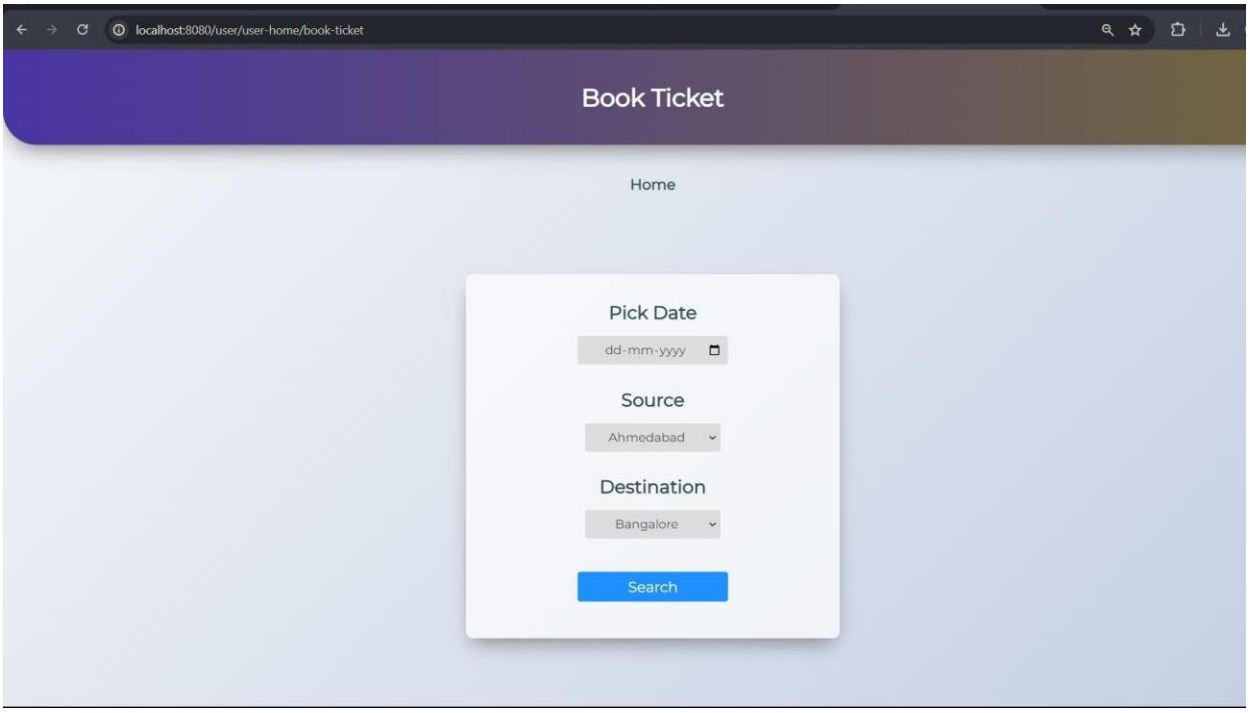
Add News (Admin):



User Registration:



Booking Ticket (User):



User Profile: