



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:

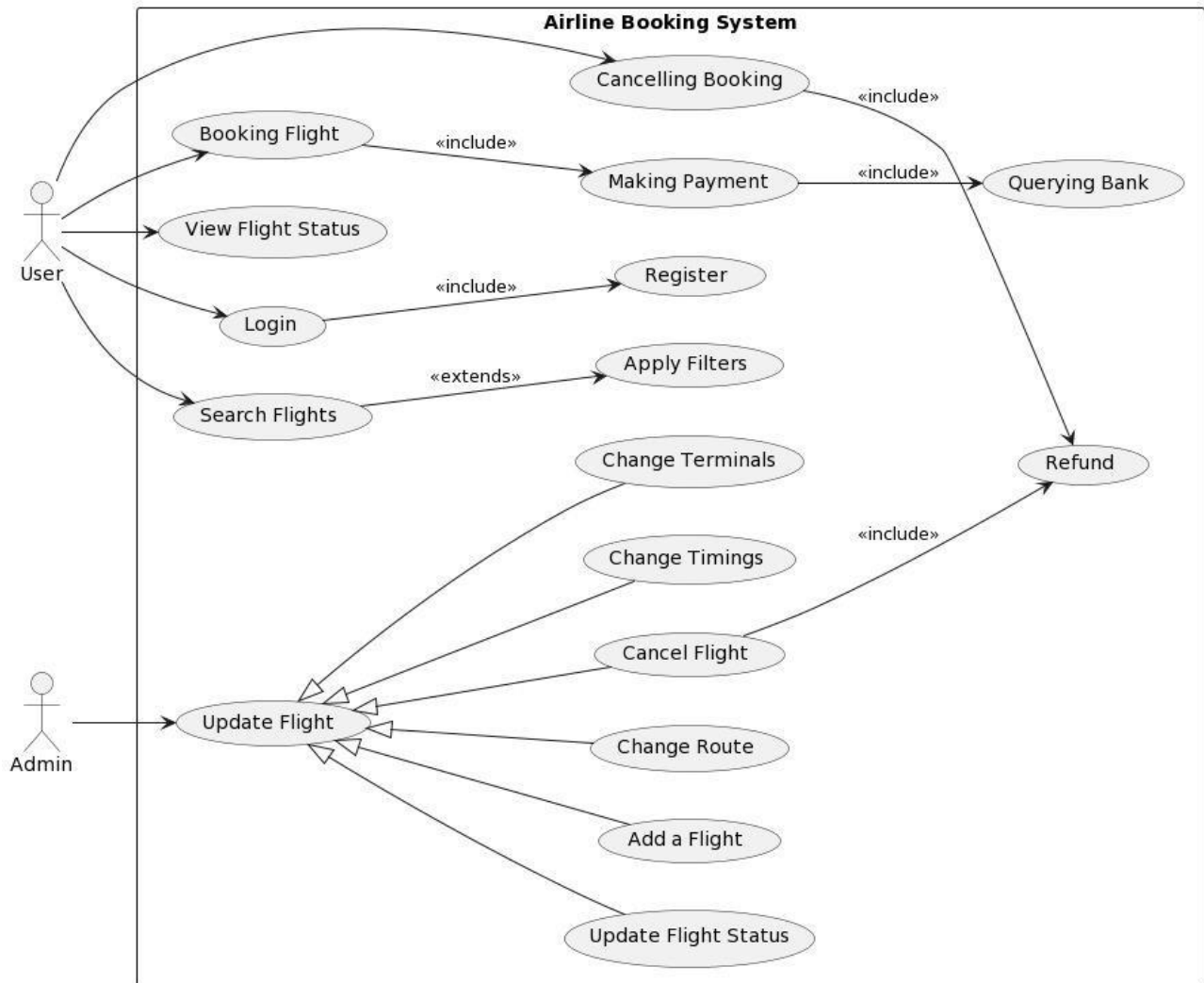
Sno.	Title	PageNo.
1	Abstract	3
2	Use Case Diagram	4
3	Class Diagram	5
4	State Diagram	6
5	Activity Diagram	9
6	MVC Architecture	11
7	Design Patterns	12
8	Design Principles	16
9	Sample Demo Screenshots	19

ABSTRACT & INTRODUCTION:

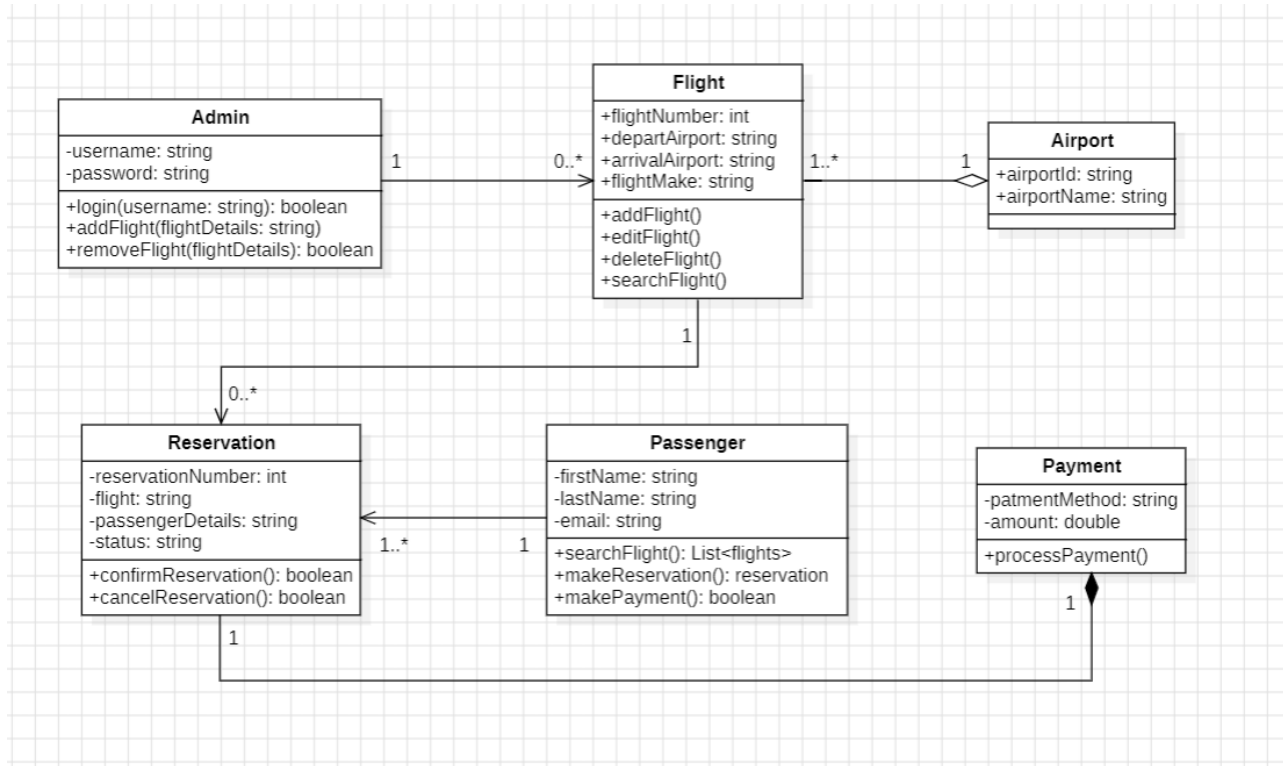
The airline booking system is a web based application involves comprehensive software solution designed to facilitate the reservation and management of flight tickets for passengers. It serves as a central platform where travelers can search, compare, and book flights to their desired destinations. Additionally,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

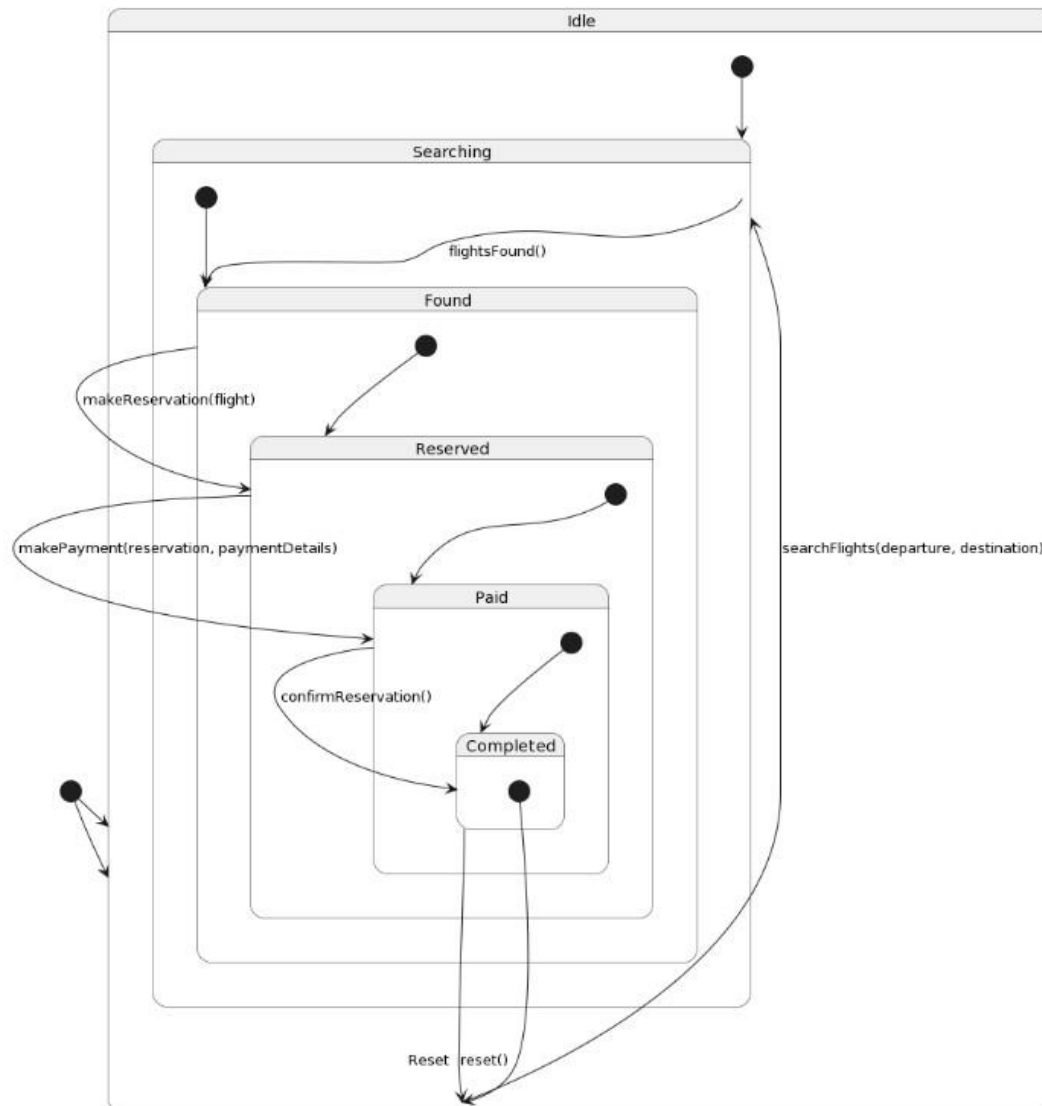


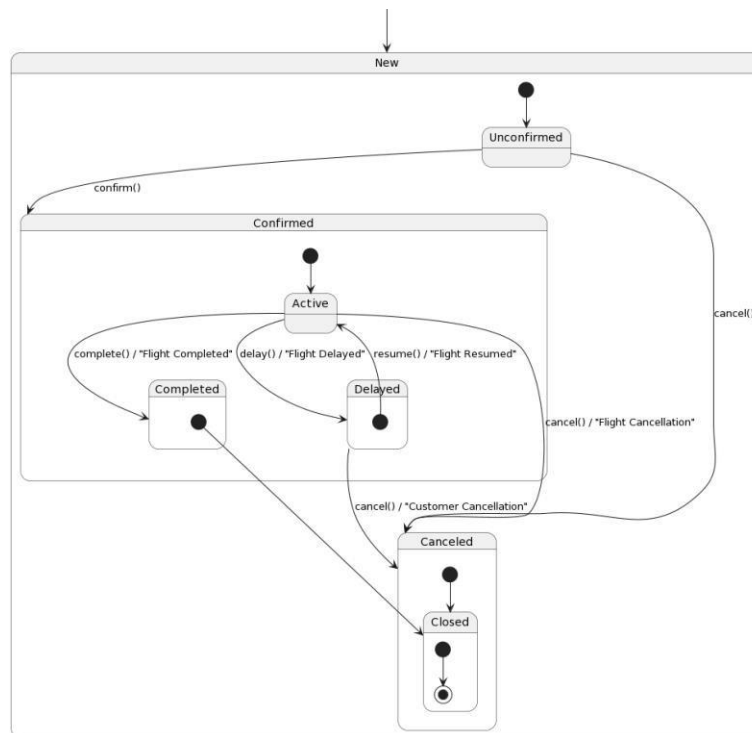
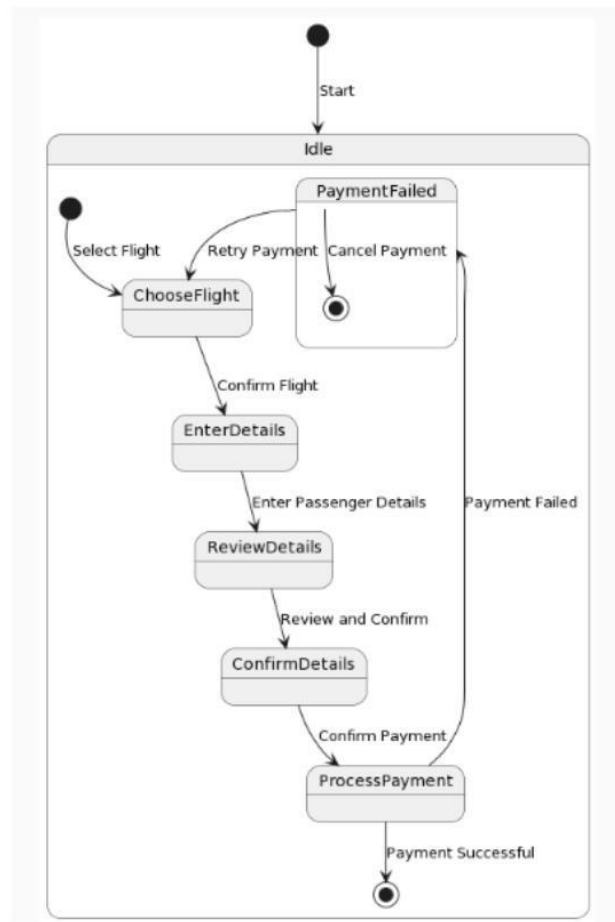
CLASS DIAGRAM:



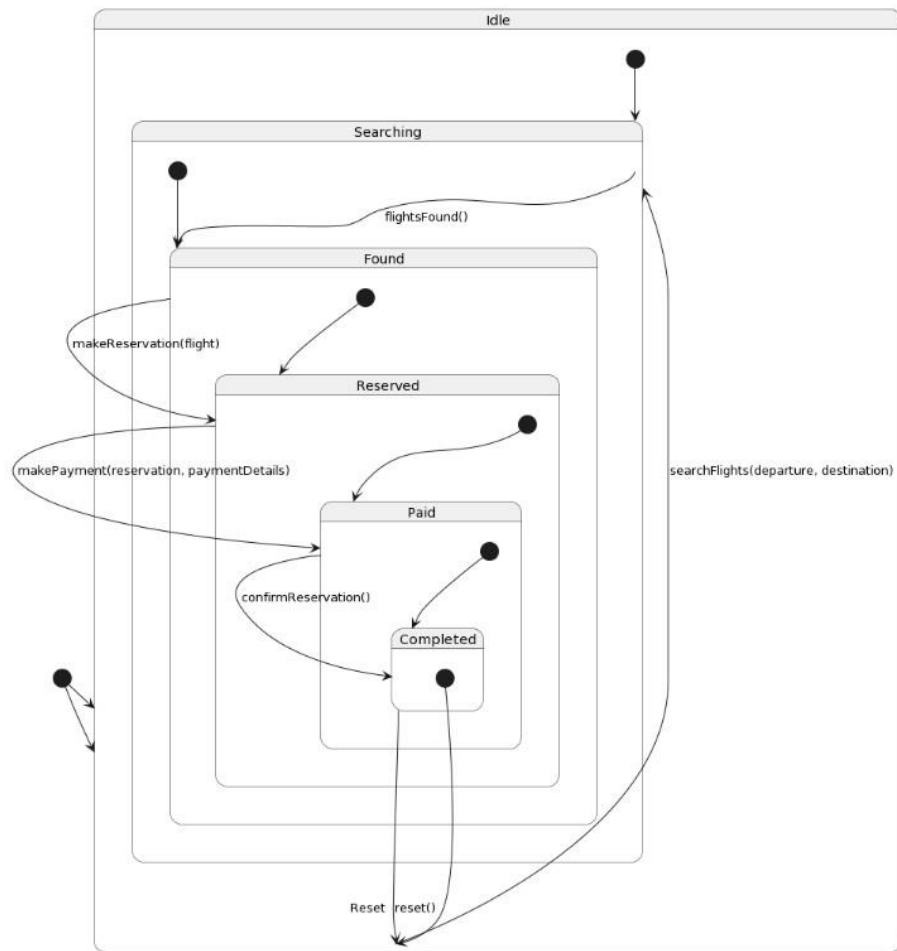
STATE DIAGRAMS:

State Diagram(Passenger Class):



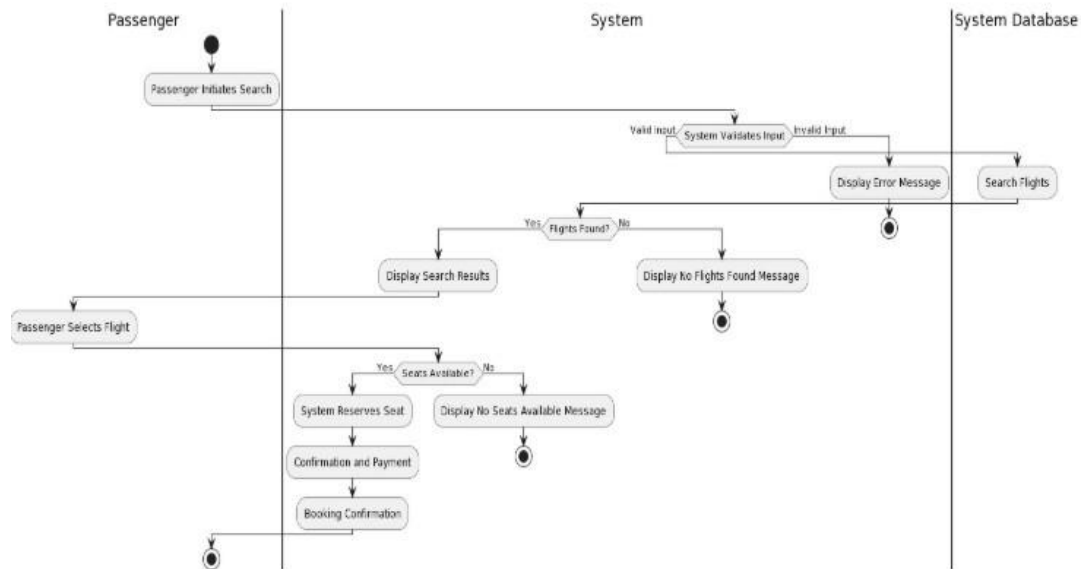


State Diagram(Passenger Class):

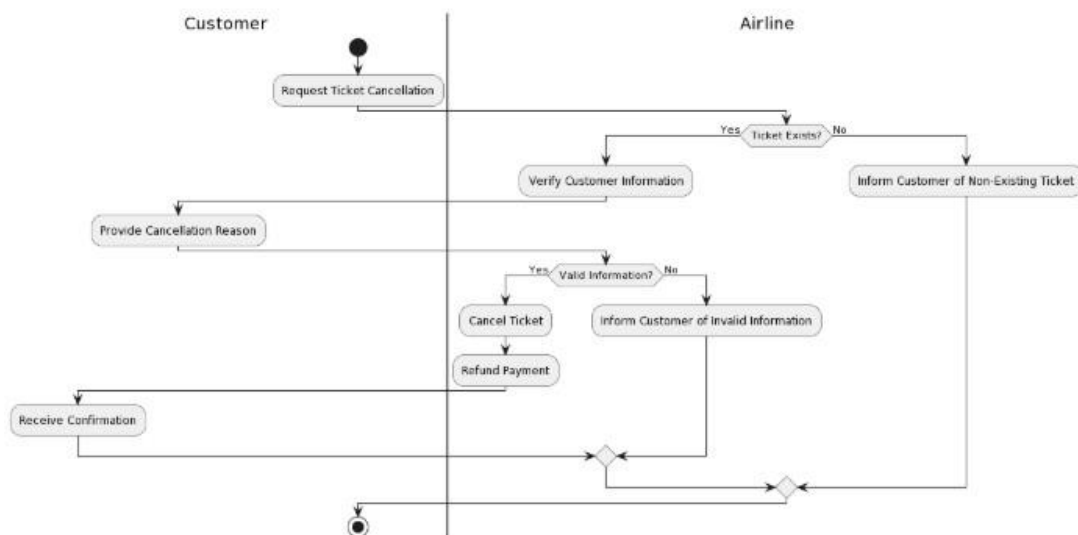


ACTIVITY DIAGRAMS:

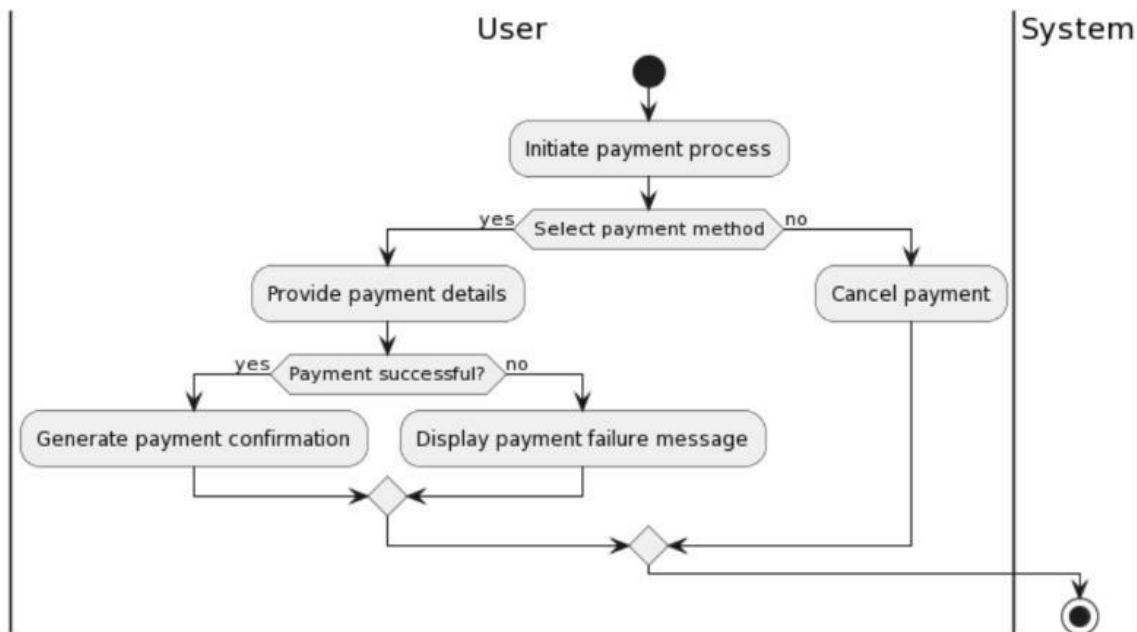
Activity Diagram For Book Flight :



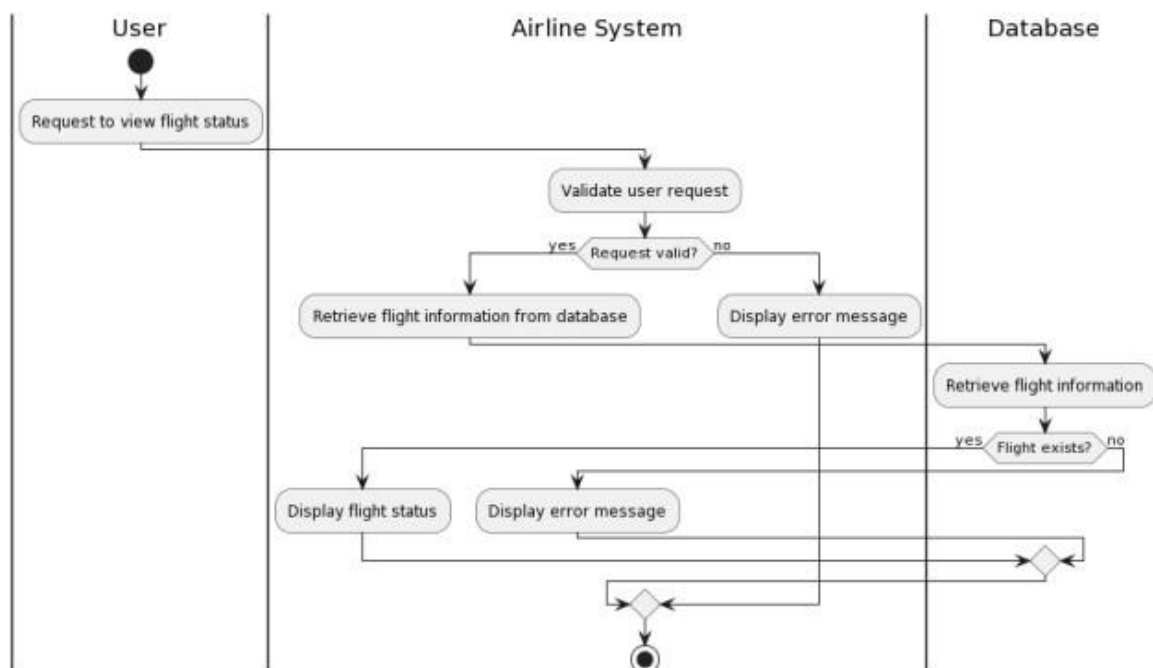
Activity Diagram for Canceling a booked ticket:



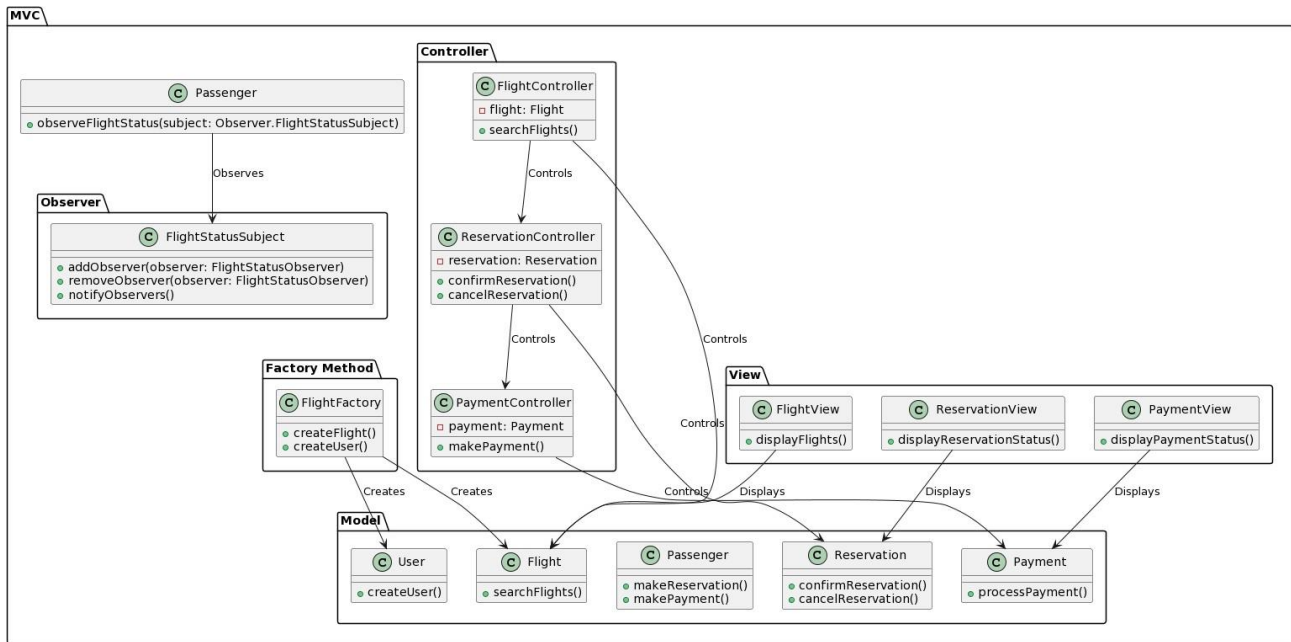
ACTIVITY DIAGRAM FOR THE USE CASE MAKING PAYMENT :



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 > ...
1  package com.cognizant.airline_ticket_reservation_system.controller;
2
3  import com.cognizant.airline_ticket_reservation_system.model.Flight;
4
5  public class FlightFactory {
6      public static Flight createFlight(String airline, String model, String type, Integer economySeats, Integer businessSeats) {
7          Flight flight = new Flight();
8          flight.setAirline(airline);
9          flight.setModel(model);
10         flight.setType(type);
11         flight.setEconomySeats(economySeats);
12         flight.setBusinessSeats(businessSeats);
13         return flight;
14     }
15 }
16
```

```
@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(), flight.getBusinessSeats());

    flightService.addFlight(newFlight);
    modelAndView.setViewName(String.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

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

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

```
@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

```
@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

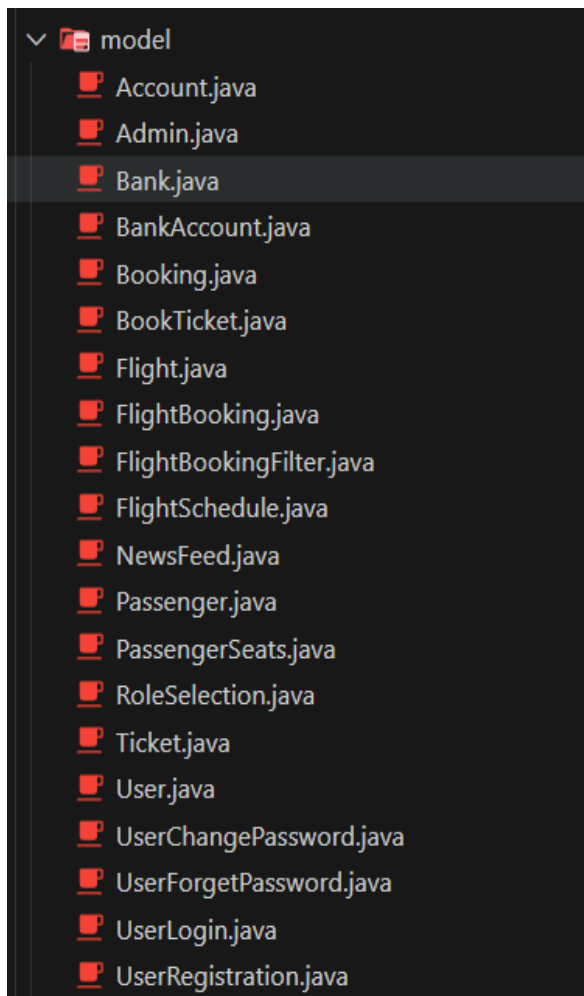
```
BookingDetailsController.java x
src > main > java > com > cognizant > airline_ticket_reservation_system > controller > BookingDetailsController.java > BookingDetailsController > bookingDetails(St
15 public class BookingDetailsController {
16     ...
17 }
18
19 @Autowired
20 public void setUserService(UserService userService) {
21     this.userService = userService;
22 }
23
24 @GetMapping("/admin/admin-home/booking-details")
25 public ModelAndView bookingDetails(
26     @ModelAttribute("flightBookingFilter") FlightBookingFilter flightBookingFilter, //Command Pattern is used here
27     ModelAndView modelAndView
28 ) {
29     List<FlightBooking> flightBookings = flightBookingService.getFlightBookings();
30     flightBookings.forEach(i -> {
31         i.setFlightSchedule(flightScheduleService.getFlightScheduleById(i.getScheduledFlightId()));
32         i.getFlightSchedule().setFlight(flightService.getFlightByNo(i.getFlightSchedule().getFlightNo()));
33     });
34
35     modelAndView.addObject(attributeName:"flightBookings", flightBookings);
36     modelAndView.setViewName(viewName:"admin/admin_home/booking-details");
37
38     return modelAndView;
39 }
40
41 @PostMapping("/admin/admin-home/booking-details")
42 public ModelAndView bookingDetails(
43     @RequestParam(value = "msg", required = false) String message,
44     @ModelAttribute("flightBookingFilter") FlightBookingFilter flightBookingFilter, //Command pattern is used here
45     ModelAndView modelAndView
46 ) {
47     ...
48 }
```

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:

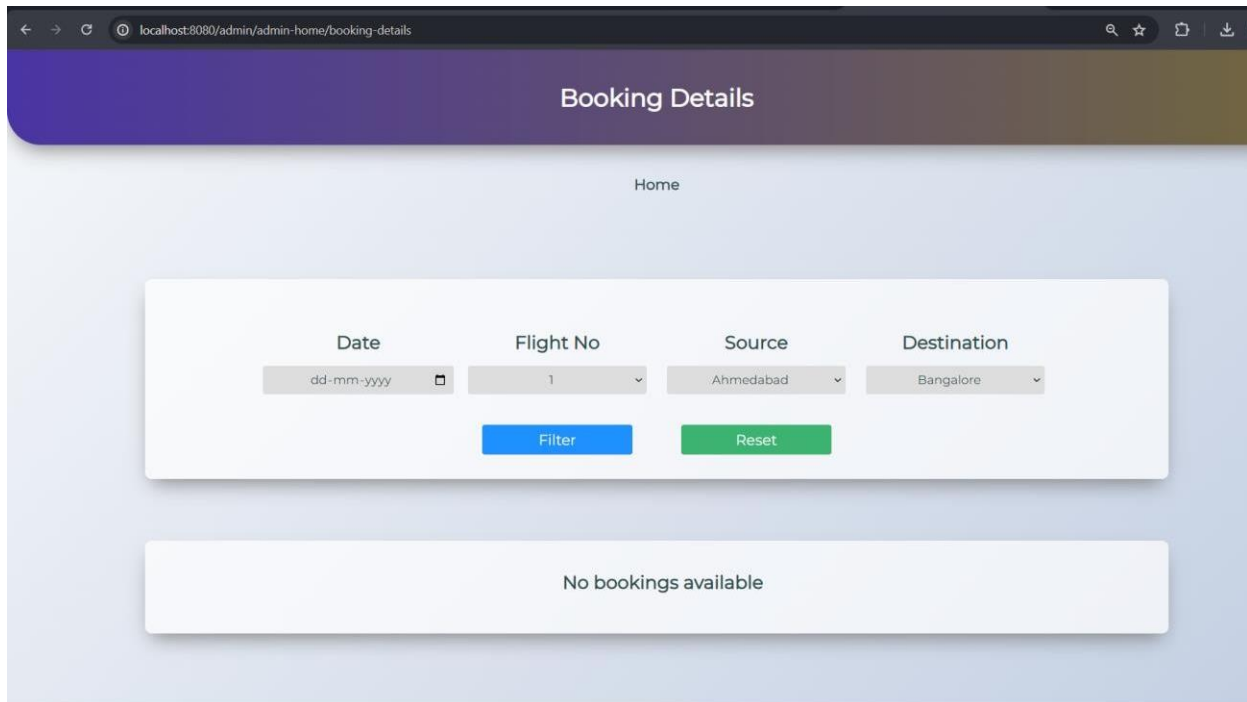
The screenshot shows a web browser at localhost:8080 displaying the 'Airline Ticket Reservation System' login page. The page has a purple header with the title. In the center, there is a white login form with the following fields: 'Select Role' (a dropdown menu with 'Admin' selected), 'Username' (a text input field containing 'oad'), and 'Password' (a text input field with masked characters). Below these fields is a blue 'Login' button. At the bottom of the page, there is a white box with the text 'No news available'.

Manage Flight (Admin):

The screenshot shows the 'Manage Flight' page for an admin user. The page has a purple header with the title 'Manage Flight'. Below the header, there are three navigation links: 'Home', 'Add Flight', and 'Schedule Flight'. The main content area is titled 'All Flights' and contains a table with flight details. The table has columns for 'NO', 'Airline', 'Model', 'Type', 'Economy Seats', 'Business Seats', and two action buttons: 'Update' and 'Delete'.

NO	Airline	Model	Type	Economy Seats	Business Seats		
1	Air India	Airbus A300	Domestic	50	10	Update	Delete
2	IndiGo	Boeing 737	Domestic	100	30	Update	Delete
3	SpiceJet	Airbus A321	Domestic	150	50	Update	Delete
4	Emirates	Boeing 777	International	200	50	Update	Delete
5	British Airways	Boeing 787	International	250	75	Update	Delete

Booking Details (Admin):



The screenshot shows the 'Booking Details' admin page. It features a purple header with the title 'Booking Details'. Below the header, there is a 'Home' breadcrumb. The main content area contains a form with four input fields: 'Date' (with a date picker icon), 'Flight No' (with a dropdown arrow), 'Source' (with a dropdown arrow), and 'Destination' (with a dropdown arrow). The 'Date' field is pre-filled with 'dd-mm-yyyy', 'Flight No' with '1', 'Source' with 'Ahmedabad', and 'Destination' with 'Bangalore'. Below these fields are two buttons: 'Filter' (blue) and 'Reset' (green). Below the form, there is a message box that says 'No bookings available'.

Booking Details

Home

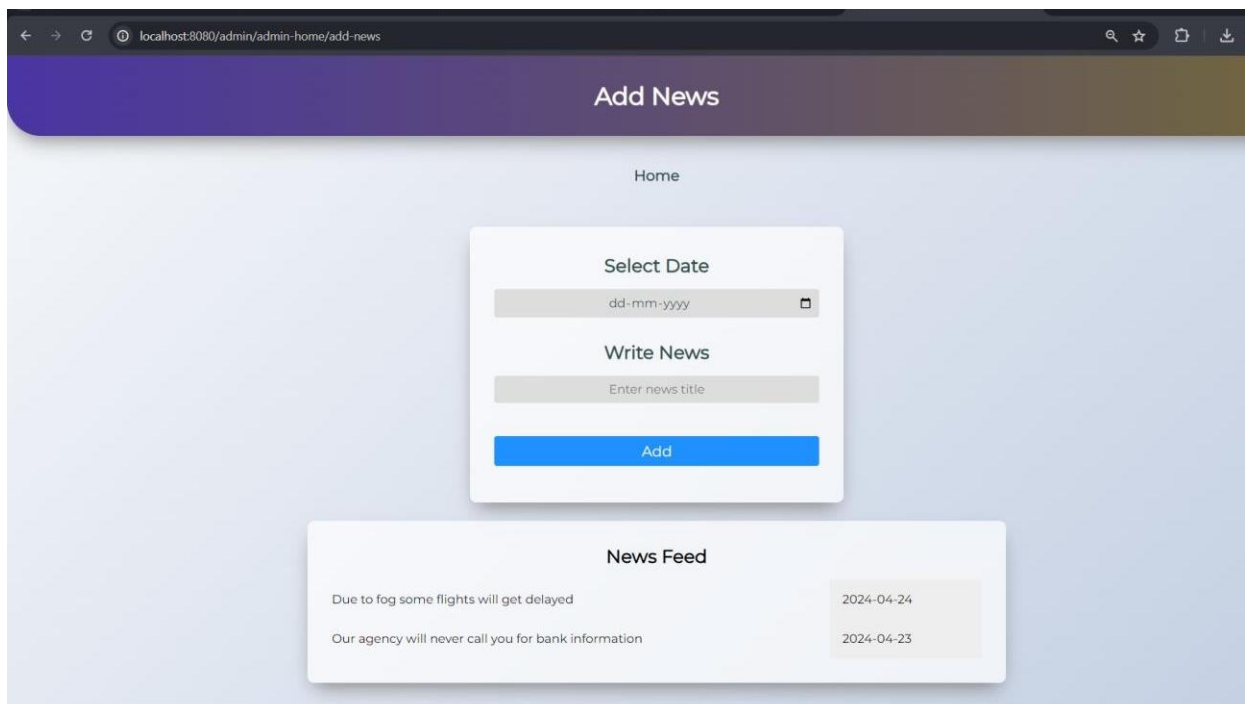
Date Flight No Source Destination

dd-mm-yyyy 1 Ahmedabad Bangalore

Filter Reset

No bookings available

Add News (Admin):



The screenshot shows the 'Add News' admin page. It features a purple header with the title 'Add News'. Below the header, there is a 'Home' breadcrumb. The main content area contains a form with three input fields: 'Select Date' (with a date picker icon), 'Write News' (with a text input field), and 'Add' (blue button). The 'Select Date' field is pre-filled with 'dd-mm-yyyy'. Below the form, there is a 'News Feed' section with two entries: 'Due to fog some flights will get delayed' (dated 2024-04-24) and 'Our agency will never call you for bank information' (dated 2024-04-23).

Add News

Home

Select Date

dd-mm-yyyy

Write News

Enter news title

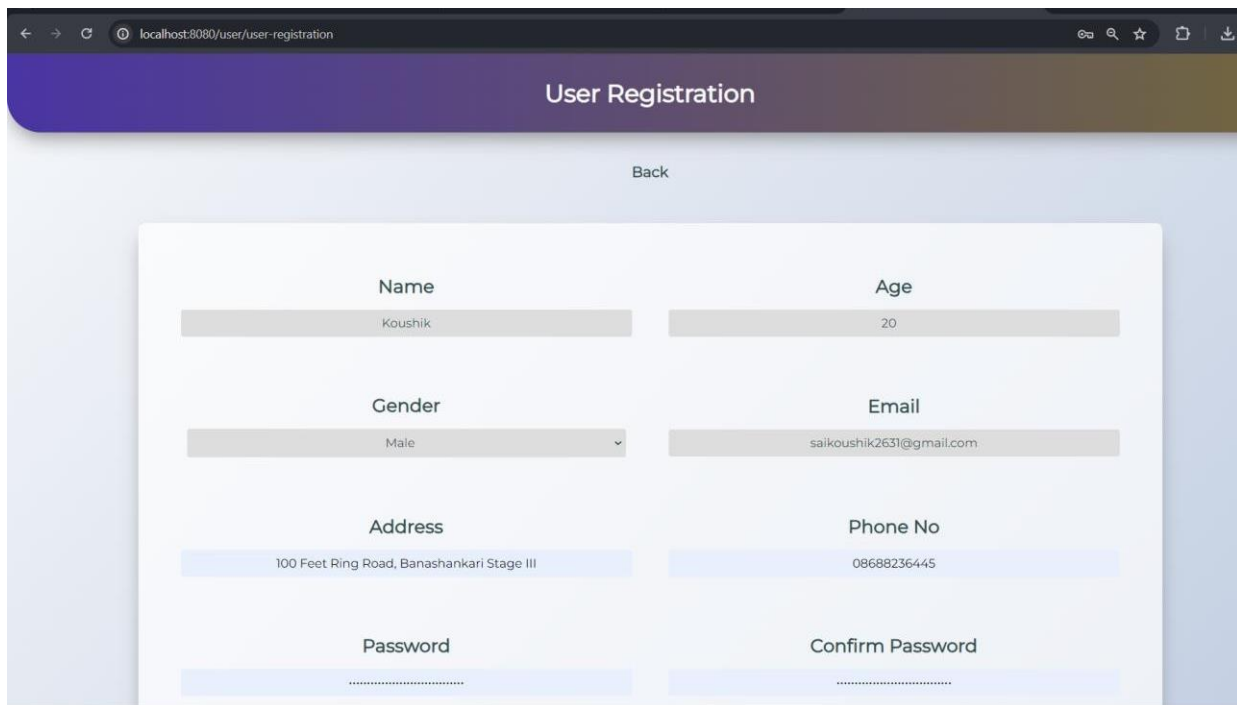
Add

News Feed

Due to fog some flights will get delayed 2024-04-24

Our agency will never call you for bank information 2024-04-23

User Registration:



localhost:8080/user/user-registration

User Registration

[Back](#)

Name
Koushik

Age
20

Gender
Male

Email
saikoushik2631@gmail.com

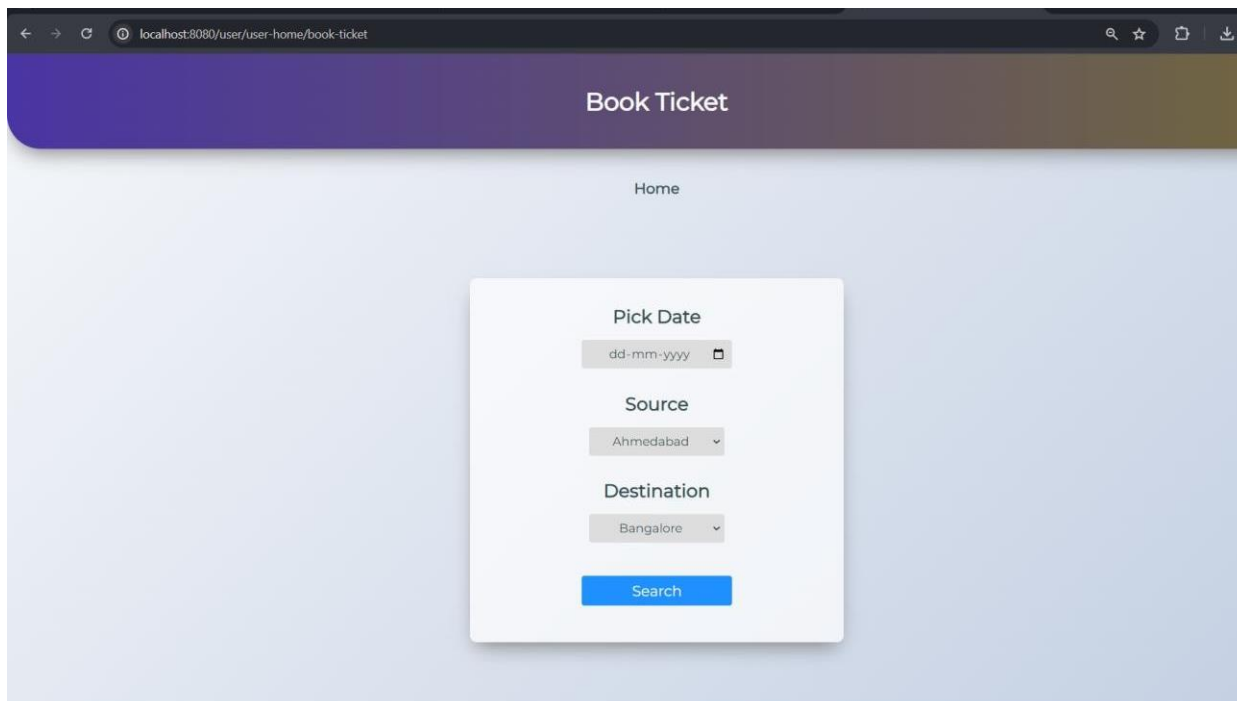
Address
100 Feet Ring Road, Banashankari Stage III

Phone No
08688236445

Password
.....

Confirm Password
.....

Booking Ticket (User):



localhost:8080/user/user-home/book-ticket

Book Ticket

[Home](#)

Pick Date
dd-mm-yyyy

Source
Ahmedabad

Destination
Bangalore

Search

User Profile:

← → ↻ localhost:8080/user/user-home/view-profile 🔍 ☆ 🗑️ ⬇️

User Profile

Home

ID	2
Name	Koushik
Age	20
Gender	Male
Email	saikoushik2631@gmail.com
Address	100 Feet Ring Road, Banashankari Stage III
Phone No	8688236445

Update Details

Change Password