UE21CS352B - Object Oriented Analysis & Design using Java

Mini Project Report

"Bus Ticket Management System"

Submitted by:

Srinidhi S A	**PES1UG21CS623**
Sujan D Kripa Das	**PES1UG21CS636**
Suraj P	**PES1UG21CS646**
Syed Tanzeel Shaz	**PES1UG21CS657**

*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

Table of Contents

Synopsis

The Bus Ticket Management System is a Java-based application designed for managing bus ticket bookings. Users can sign up or log in to the application to book tickets. Upon login, users can select their desired source and destination for their journey. The application then calculates the fare price based on the selected route and also takes into account the peak hours and non peak hours and displays a summary of the booking, including the fare price. All payments are processed online. The application stores information such as source, destination, and fare prices in a SQL database for efficient retrieval and management. Additionally, the system includes an admin role responsible for validating or invalidating tickets. This ensures that ticket bookings are properly managed and authenticated by authorized personnel.

Architecture:

The application employs the model - view - controller (MVC) architecture pattern , a widely used design patterns in software engineering . MVC separates an application into three interconnected components , facilitating better organization , maintainability , and scalability.

1 . Model Component:

The Model component manages the application's data, business logic, and rules independently of the user interface. It handles data storage, retrieval, and manipulation, ensuring consistency and integrity.

Characteristics:

● Data Management: Models store and manipulate data structures pertinent to the application's domain.

● Business Logic: Contains algorithms and rules governing data processing and behavior.

This includes entities such as:

- User: Represents the user of the system. It contains attributes such as username, password, email, etc.

- Ticket: Represents a bus ticket. Contains information such as ticket ID, user ID (who booked the ticket), source, destination, fare price, booking status, etc.
- Route: Represents a bus route. Contains information such as source, destination, distance, fare price, etc.

- DatabaseManager: Handles interactions with the SQL database. Includes methods for CRUD operations (Create, Read, Update, Delete) on users, tickets, and routes.

2 . View Component

The View component manages the user interface (UI) and presentation logic of the application.

Characteristics:

● User Interaction: Renders graphical or textual output to users and captures user input.

● Multiple Views: Supports multiple views for a single model, facilitating diverse presentation options.

● Passive Strategy: Implements a passive view approach, separating UI logic from data processing.

It displays data from the Model to users and forwards user commands to the Controller. Views display information retrieved from the Model and provide user-friendly interfaces for performing actions. Our project view includes:

- SignUpView: Interface for user registration.

- LoginView: Interface for user login.

- BookingView: Interface for booking bus tickets. Displays options for selecting source, destination, and payment.

- TicketSummaryView: Displays a summary of the booked ticket, including fare price and payment details.

- AdminPanelView: Interface for admin functions like ticket validation or invalidation.

3 . Controller Component

The Controller component acts as an intermediary between the Model and View, handling user input and application logic. It interprets user actions, manipulates data in the Model,and updates the View accordingly It handles user requests, processes input data, and updates the Model based on user interactions.

Characteristics:

● Event Handling: Processes user events initiated in the View and triggers corresponding actions.

● Coordinating Role: Determines appropriate responses to user interactions, orchestrating interactions between Model and View.

● Input Conversion: Converts user inputs into commands for the Model or View, facilitating data processing.

Controller includes:

- UserController: Handles user-related actions such as registration, login, and authentication. BookingController: Manages the booking process, including calculating fares, creating tickets, and processing payments.
- AdminController: Handles admin actions such as validating or invalidating tickets.
- MainController: Acts as the main controller, routing requests from views to appropriate controllers and coordinating the flow of data between models and views.
- DatabaseController: Interacts with the DatabaseManager to fetch or update data from the database.

Design Principles

1.Single Responsibility Principle:

- In adherence to the Single Responsibility Principle (SRP), every method within the controller fulfills a distinct responsibility.
- Functions such as home(), login(), signup(), bookticket(),viewticket()and paymentticket() exclusively manage individual HTTP requests, seamlessly returning the associated views.
- To maintain a clear distinction of concerns, the controller tactfully delegates intricate business logic to dedicated service classes, upholding a fundamental separation of concerns within the system's architecture.

2.Open Closed Principle:

- In alignment with the Open/Closed Principle (OCP), the controller methods remain open for extension, facilitating the addition of new functionalities without necessitating modifications to their existing implementation.

- Any enhancements or new features can be seamlessly integrated by either introducing new methods or extending the functionality of existing ones, without altering their foundational structure.

- This approach ensures that the controller remains closed for modification, thereby preserving its stability and integrity while accommodating future expansions or enhancements.

3.Dependency Inversion Principle(DIP):

- Following the Dependency Inversion Principle (DIP), the controller relies on abstractions rather than concrete implementations, fostering loose coupling between components and promoting flexibility in the system's architecture.

- By adhering to this principle, the controller depends on interfaces or abstract classes, enabling seamless substitution of dependencies with alternative implementations.

- This decoupling allows for easier maintenance, testing, and scalability, as dependencies can be swapped out without impacting the controller's core functionality. Thus, the Dependency Inversion Principle fosters a modular and extensible design, ensuring that changes to lower-level components do not necessitate modifications to higher-level modules.

Design Pattern

1.Factory Pattern(Creational Design Pattern):

The Factory Pattern is a creational design pattern that provides a way to create objects without specifying their exact class. Instead of directly instantiating objects using a constructor, the Factory Pattern defines a separate class responsible for creating instances of objects based on certain criteria or parameters.

- To facilitate the dynamic instantiation of BusService implementations tailored to specific route types, the project employs the Factory Pattern

- The BusServiceFactory class serves as a factory for generating BusService instances, abstracting the creation logic and promoting code maintainability and extensibility.

- By adhering to the Factory Pattern, the project adheres to the principle of encapsulation, ensuring that the client code is shielded from the complexities of object instantiation.

- Through the use of a factory method, BusServiceFactory.getService(), the application dynamically constructs instances of BusService implementations corresponding to different route types.

- The utilization of the Factory Pattern enhances the project's flexibility, allowing for the seamless addition of new BusService implementations for additional route types in the future.

- By centralizing the instantiation logic within the BusServiceFactory, the project promotes the Dependency Inversion Principle, as client code depends on abstractions rather than concrete implementations.

- The Factory Pattern implementation fosters code reusability and modularity, enabling easy integration with the Spring Framework's dependency injection mechanism, as demonstrated by the @Autowired annotation used for dependency injection.

Code:

```
package com.code.rest.service.factory;

import org.springframework.beans.factory.annotation.Autowired;

import com.code.rest.repository.route1BusRepository;
import com.code.rest.repository.route2BusRepository;
import com.code.rest.service.BusService;
import com.code.rest.service.impl.route1BusServiceImpl;
import com.code.rest.service.impl.route2BusServiceImpl;

public class BusServiceFactory {

    private final route1BusRepository r1Repository;
    private final route2BusRepository r2Repository;

    @Autowired
    public BusServiceFactory(route1BusRepository r1Repository,
route2BusRepository r2Repository) {
        this.r1Repository = r1Repository;
        this.r2Repository = r2Repository;
```

```
    }


    public BusService getService(String routeType) {
        switch (routeType) {
            case "r1":
                return new route1BusServiceImpl(r1Repository);
            case "r2":
                return new route2BusServiceImpl(r2Repository);
            default:
                throw new IllegalArgumentException("Invalid route
type");
        }
    }
}
```

```
@GetMapping("/3")
    public String getBus(Model model) {
        List<Long> busNames = new ArrayList<>();

        // Use the factory method to get BusService instances
        BusServiceFactory serviceFactory = new
BusServiceFactory(r1,r2);
        BusService r1BusService = serviceFactory.getService("r1");
        List<? extends Bus> bus1 = r1BusService.getBus();

        BusService r2BusService = serviceFactory.getService("r2");
        List<? extends Bus> bus2 = r2BusService.getBus();

        for (Bus route : bus1) {
            busNames.add(route.getMid());
        }
        for (Bus route : bus2) {
            busNames.add(route.getMid());
        }

        model.addAttribute("routes", busNames);
        return "routes";
    }
```

2.Facade Pattern(Structural Design Pattern):

The Facade Pattern is a structural design pattern that provides a unified interface to a set of interfaces in a subsystem. It encapsulates a complex system or subsystem behind a simplified interface, making it easier to use and understand.

- The project leverages the Facade Pattern to streamline the interaction between client code and the underlying subsystems, promoting a simplified and cohesive interface.
- By encapsulating the complexities of the StopService within the StopFacadeimpl class, the Facade Pattern provides a unified entry point for accessing stop-related functionalities.
- Through the implementation of the StopFacade interface, the project abstracts away the intricacies of the StopService, shielding client code from the underlying implementation details.
- The StopFacadeimpl class orchestrates the retrieval of stops from multiple sources, aggregating data from the StopService to present a consolidated view of all available stops.
- With the Facade Pattern in place, client code can seamlessly access stop data by invoking the getAllStops() method, which internally delegates the task to the underlying StopService.
- The adoption of the Facade Pattern enhances code maintainability and readability by providing a clear and concise interface for interacting with stop-related functionalities.
- Through the use of dependency injection and autowiring, the StopFacadeimpl class effectively integrates with the Spring Framework, facilitating seamless integration into the project's architecture.
- The Facade Pattern promotes loose coupling between client code and the underlying subsystems, allowing for easier maintenance and extensibility of the system.

Code:

```
package com.code.rest.service.factory;

import java.util.ArrayList;
import java.util.List;

import org.springframework.beans.factory.annotation.Autowired;
```

```java
import org.springframework.stereotype.Service;


import com.code.rest.entity.Stop;

import com.code.rest.service.StopService;

@Service
public class StopFacadeimpl implements StopFacade {

    @Autowired
    private StopService StopService;

    @Override
    public List<String> getAllStops() {
        List<String> allStops = new ArrayList<>();
        allStops.addAll(StopService.getr1());
        allStops.addAll(StopService.getr2());
        return allStops;
    }
}
```

```java
    @GetMapping("/1")
    public String getRoutes(Model model) {
        List<String> routeNames = new ArrayList<>();
        List<Route> routes = rot.getAll();
        for (Route route : routes) {
            routeNames.add(route.getName());
        }
        model.addAttribute("routes", routeNames);
        return "routes";
    }
    @GetMapping("/2")
    public String getStop(Model model) {
        List<String> StopNames=s.getAllStops();
        model.addAttribute("routes", StopNames);
        return "routes";
    }
```

3.Strategy Pattern(Behavioural Design Pattern):

The Strategy Pattern is a behavioral design pattern that enables an algorithm's behavior to be selected at runtime. It allows a class, called the context, to have a strategy interface that defines a family of algorithms, each encapsulated in a separate concrete strategy class. The context can then switch between these strategies dynamically, depending on the requirements or changes in the application.

- The project employs the Strategy Pattern to dynamically select fare calculation strategies based on peak and off-peak hours, enhancing flexibility and adaptability within the fare calculation module.
- With the FareStrategySelector acting as the context, the Strategy Pattern facilitates the seamless integration of multiple fare calculation algorithms, encapsulated within concrete strategy classes.
- By utilizing the Strategy Pattern, the project promotes code maintainability and extensibility, allowing for the addition of new fare calculation strategies without modifying existing code.
- The OffPeakHourFareStrategy and PeakHourFareStrategy classes represent concrete implementations of fare calculation algorithms, encapsulating distinct pricing logic for off-peak and peak hours, respectively.
- Through the FareCalculationStrategy interface, the project defines a common interface for all fare calculation strategies, promoting consistency and clarity in the codebase.
- The Strategy Pattern enhances code readability and separation of concerns by decoupling the fare calculation logic from the FareStrategySelector, enabling each strategy to evolve independently.
- By leveraging dependency injection with Spring's @Component annotation, the project seamlessly integrates strategy selection into the Spring Framework, facilitating inversion of control and promoting modularity.
- The Strategy Pattern enables the project to adapt to changing business requirements or fare calculation policies by allowing strategies to be dynamically switched at runtime, ensuring scalability and future-proofing the system

Code:

```
package com.code.rest.strategy;

import org.springframework.stereotype.Component;
```

```java
import org.springframework.stereotype.Component;

@Component
public class FareStrategySelector {

  public FareCalculationStrategy selectStrategy(boolean isPeakHour)
{
    if (isPeakHour) {
      return new PeakHourFareStrategy();
    } else {
      return new OffPeakHourFareStrategy();
    }
  }
}
```

```java
package com.code.rest.strategy;

public class OffPeakHourFareStrategy implements
FareCalculationStrategy {
  @Override
  public double calculateFare(double distance) {
    return 4.0 * distance;
  }
}
```

```java
package com.code.rest.strategy;

public class PeakHourFareStrategy implements
FareCalculationStrategy {
  @Override
  public double calculateFare(double distance) {
    return 5.0 * distance + 2.0 ;
  }
}
```

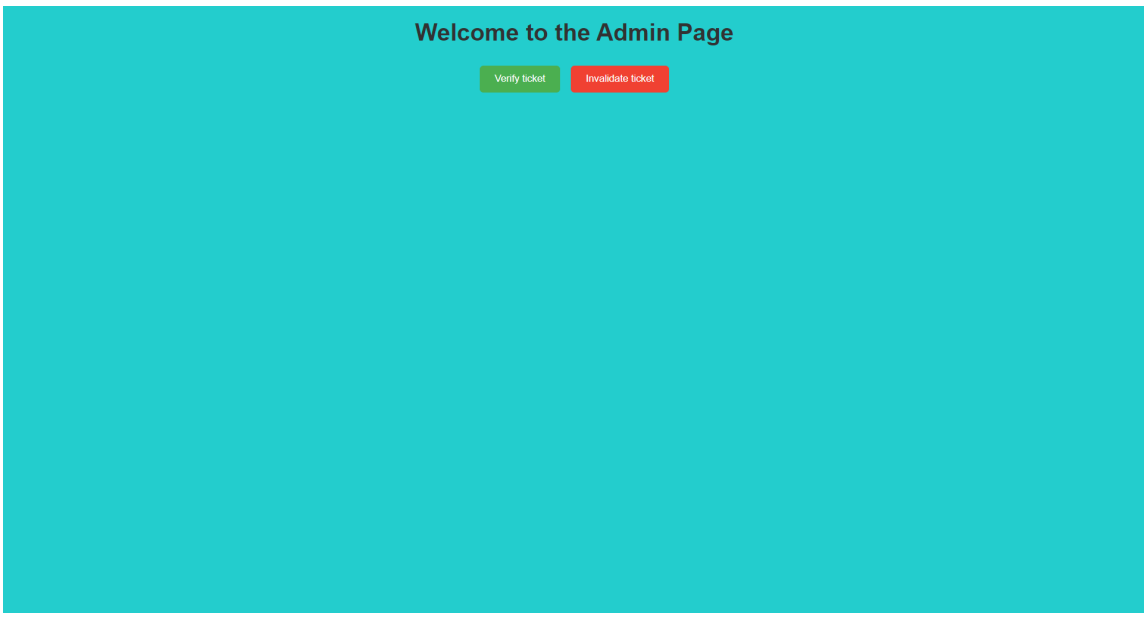
.

Screenshots Of Website

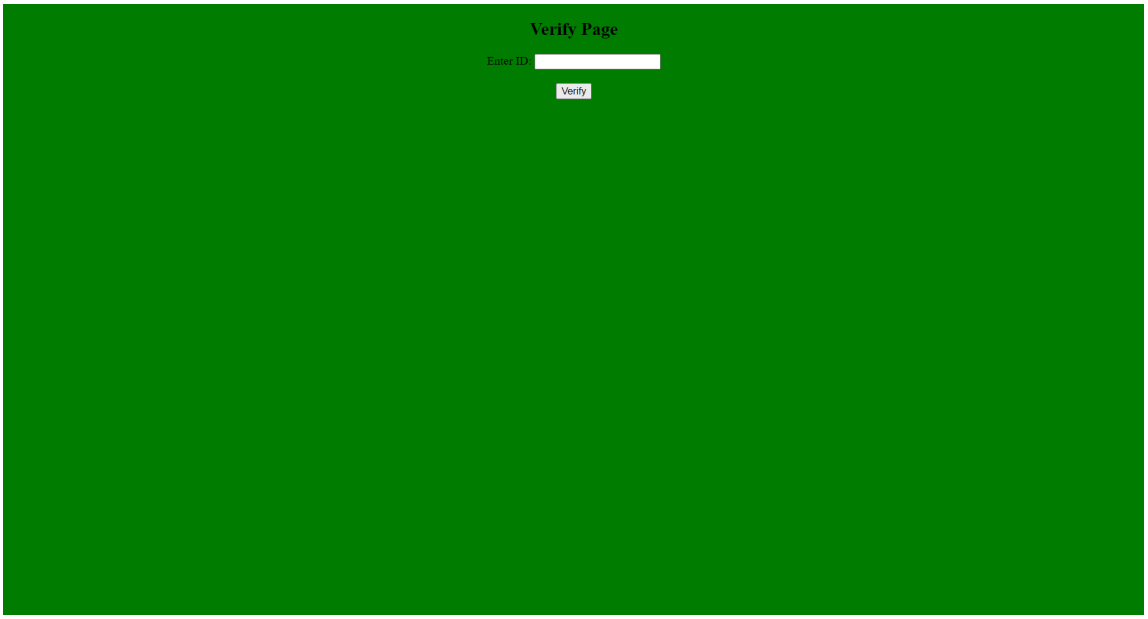
1.Home Page

2.ADMIN Login Page

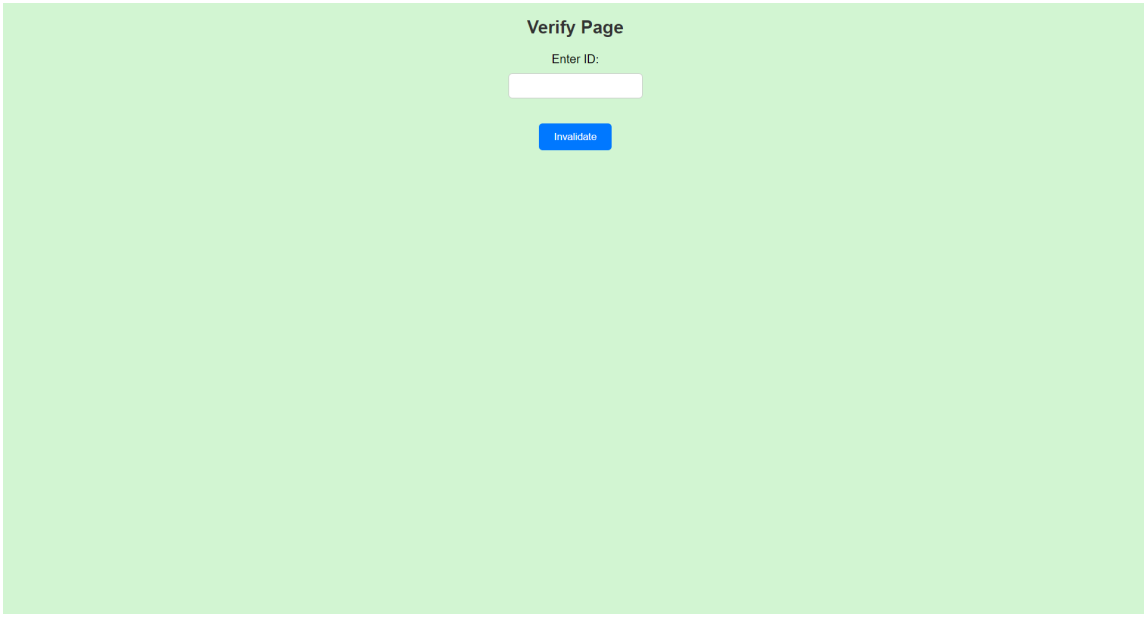
3.ADMIN Page



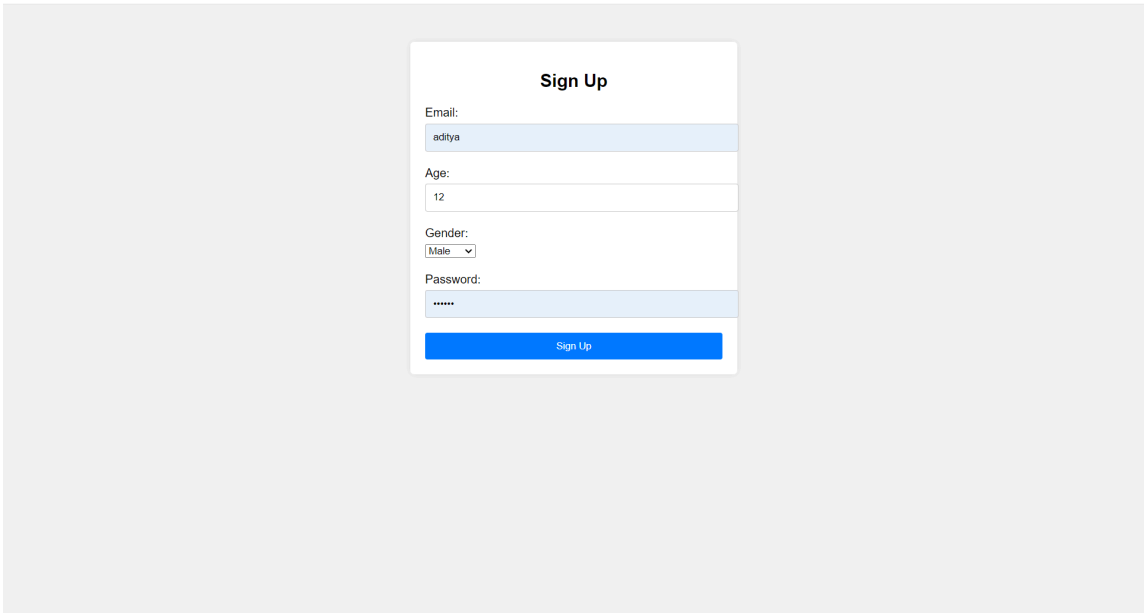
4.ADMIN Verify Page



5.ADMIN Invalidate Page



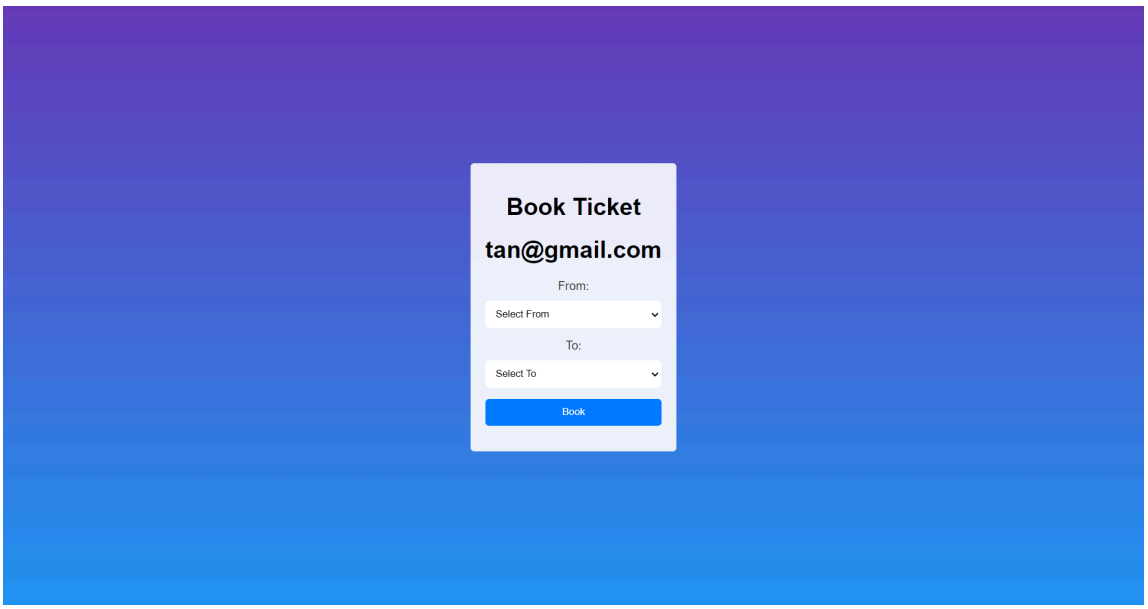
6.User Signup Page



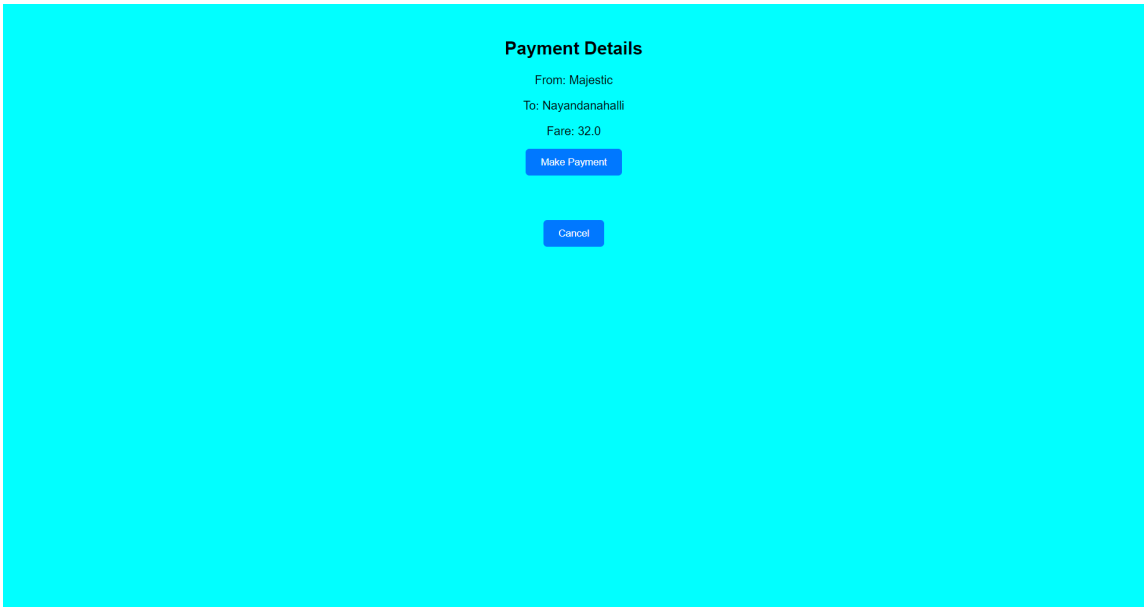
7.User Login Page

8.User Page

9.User Book Ticket Page:



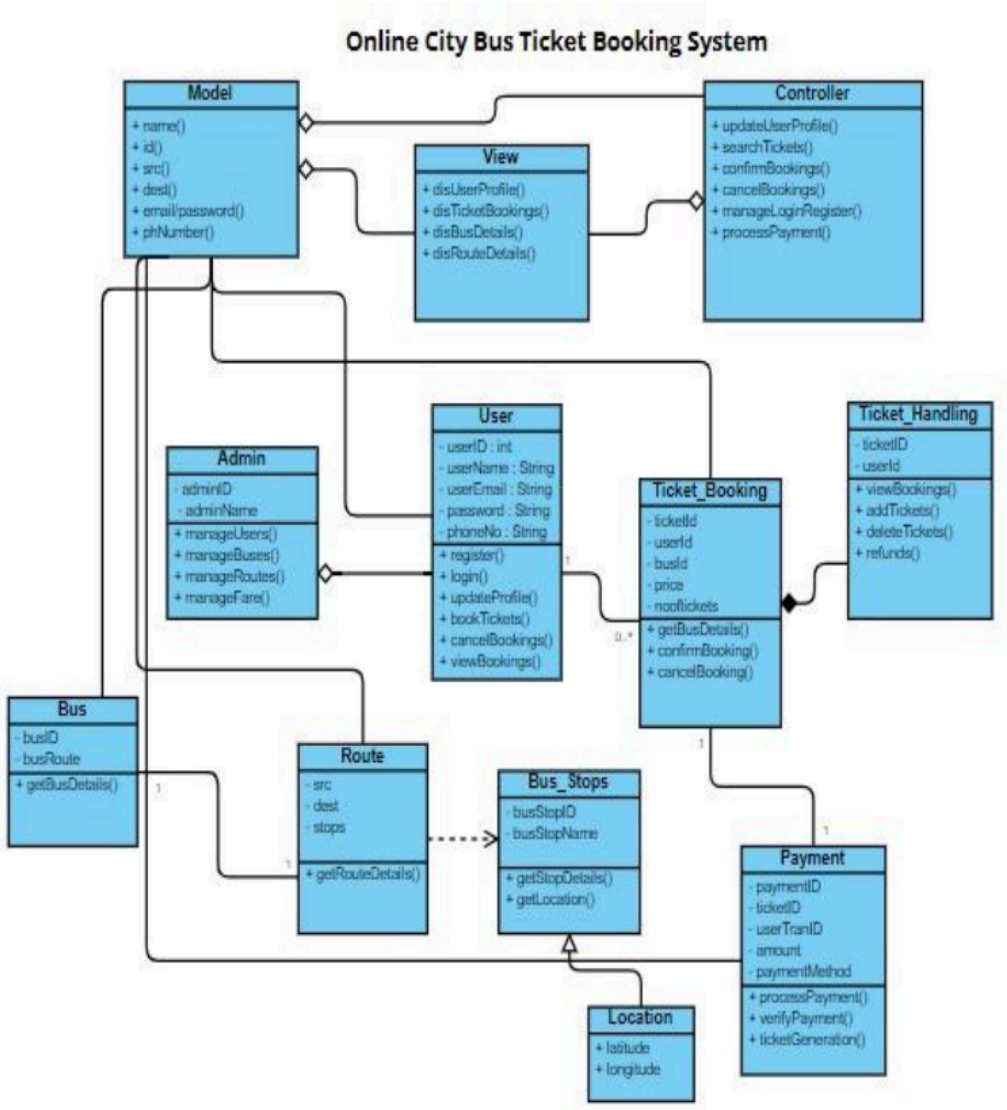
10.User Payment Page



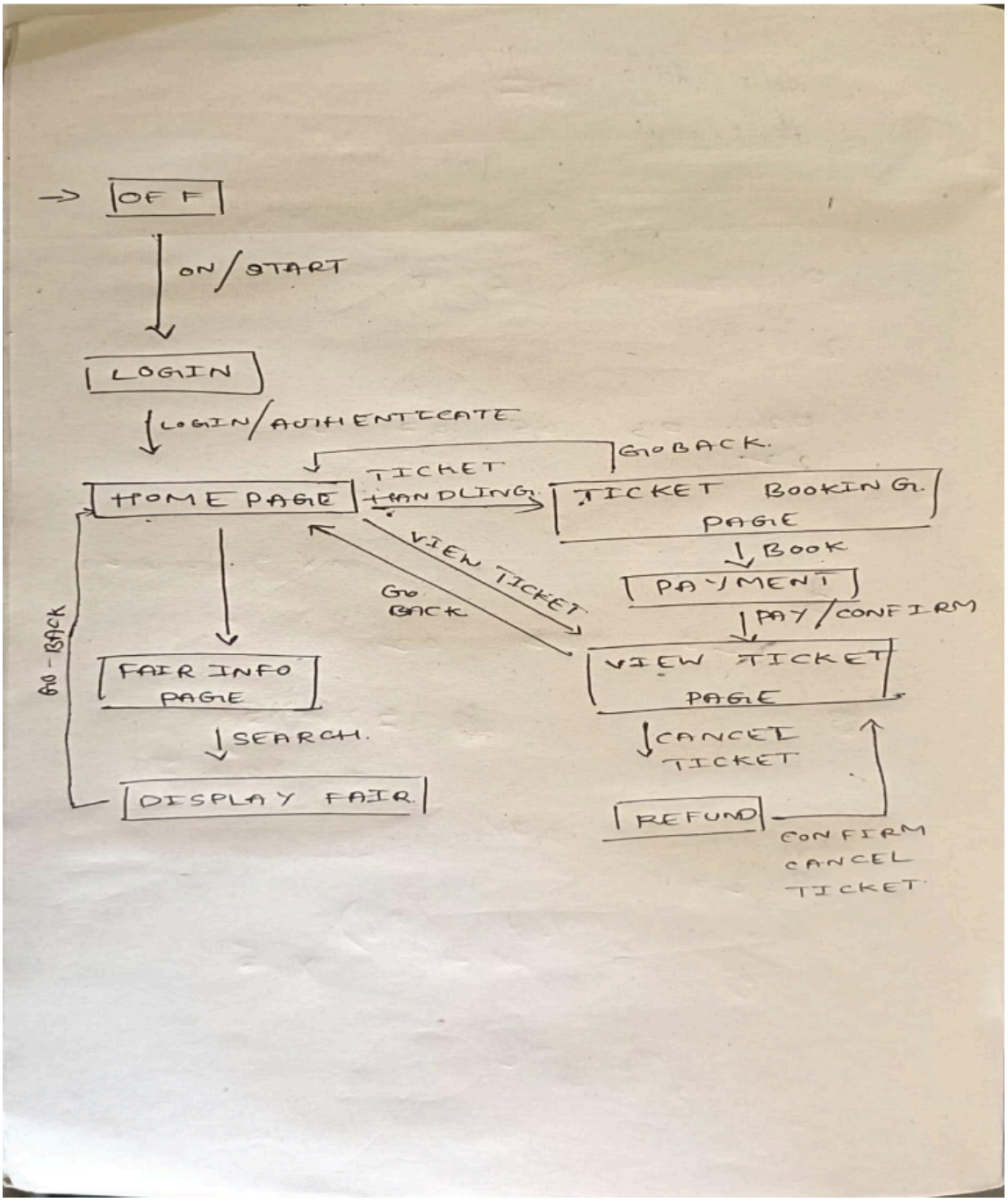
11.User View My Tickets Page:

Models

1.Class Diagram

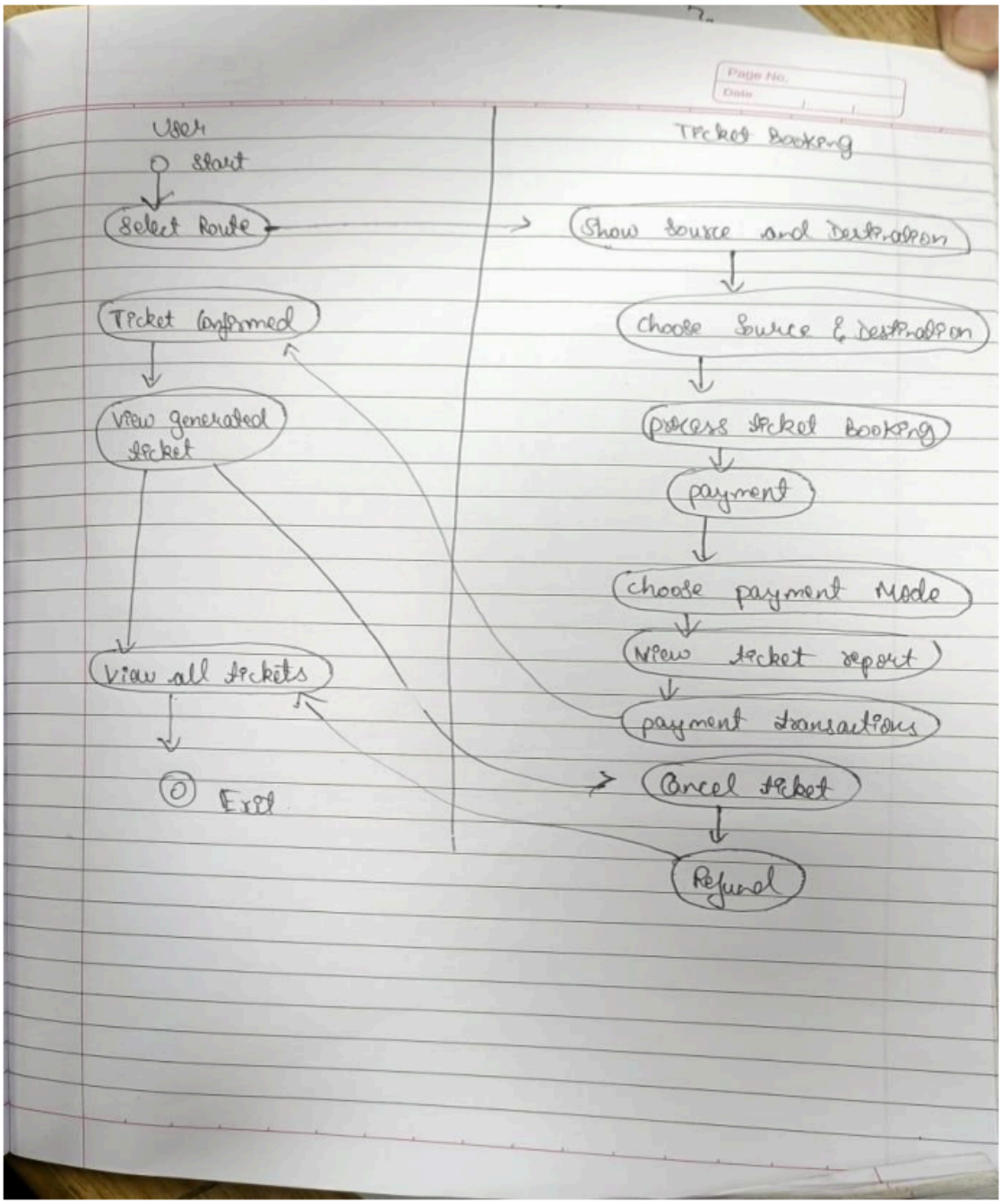


2.State Diagram

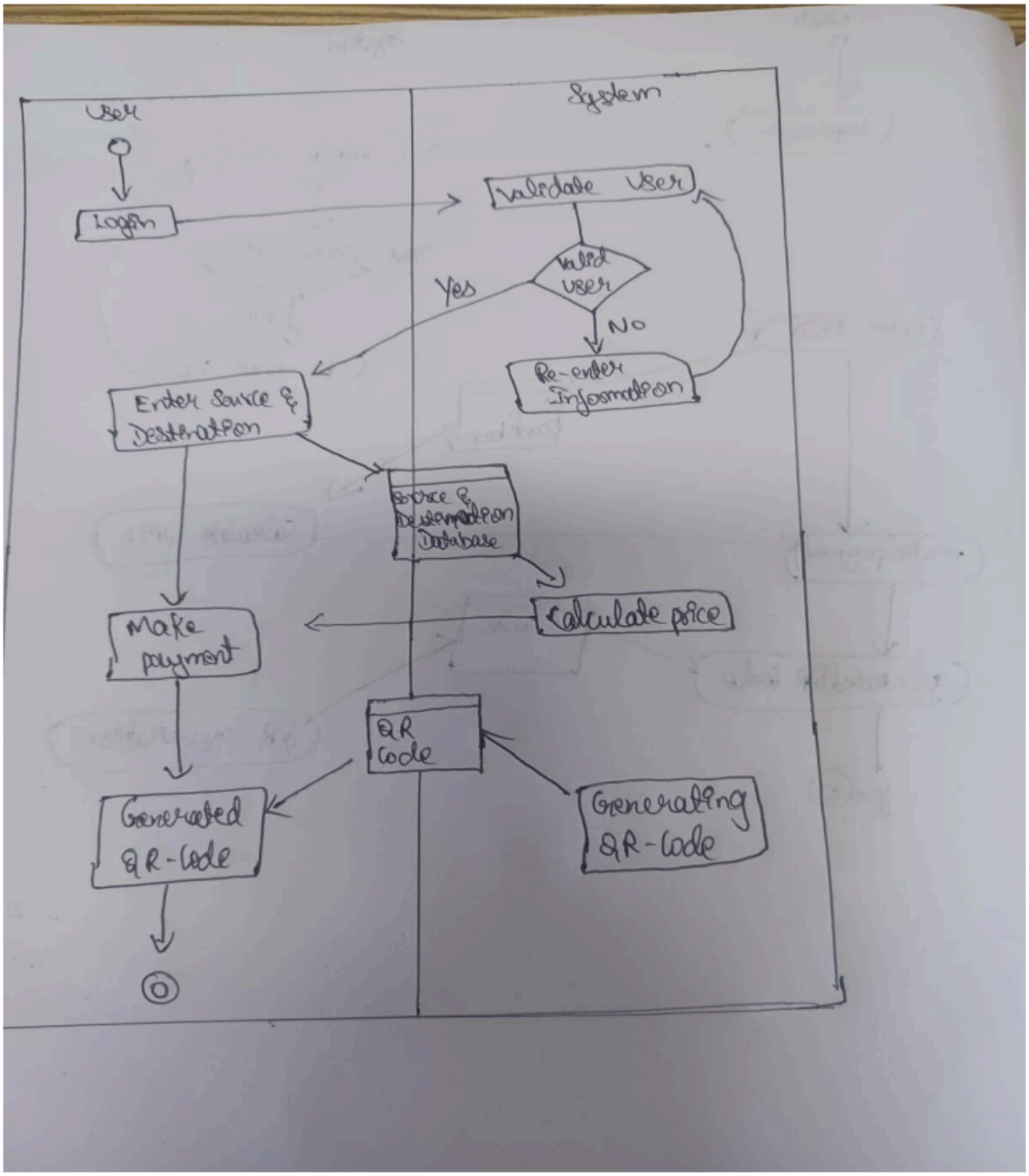


3.Activity Diagram

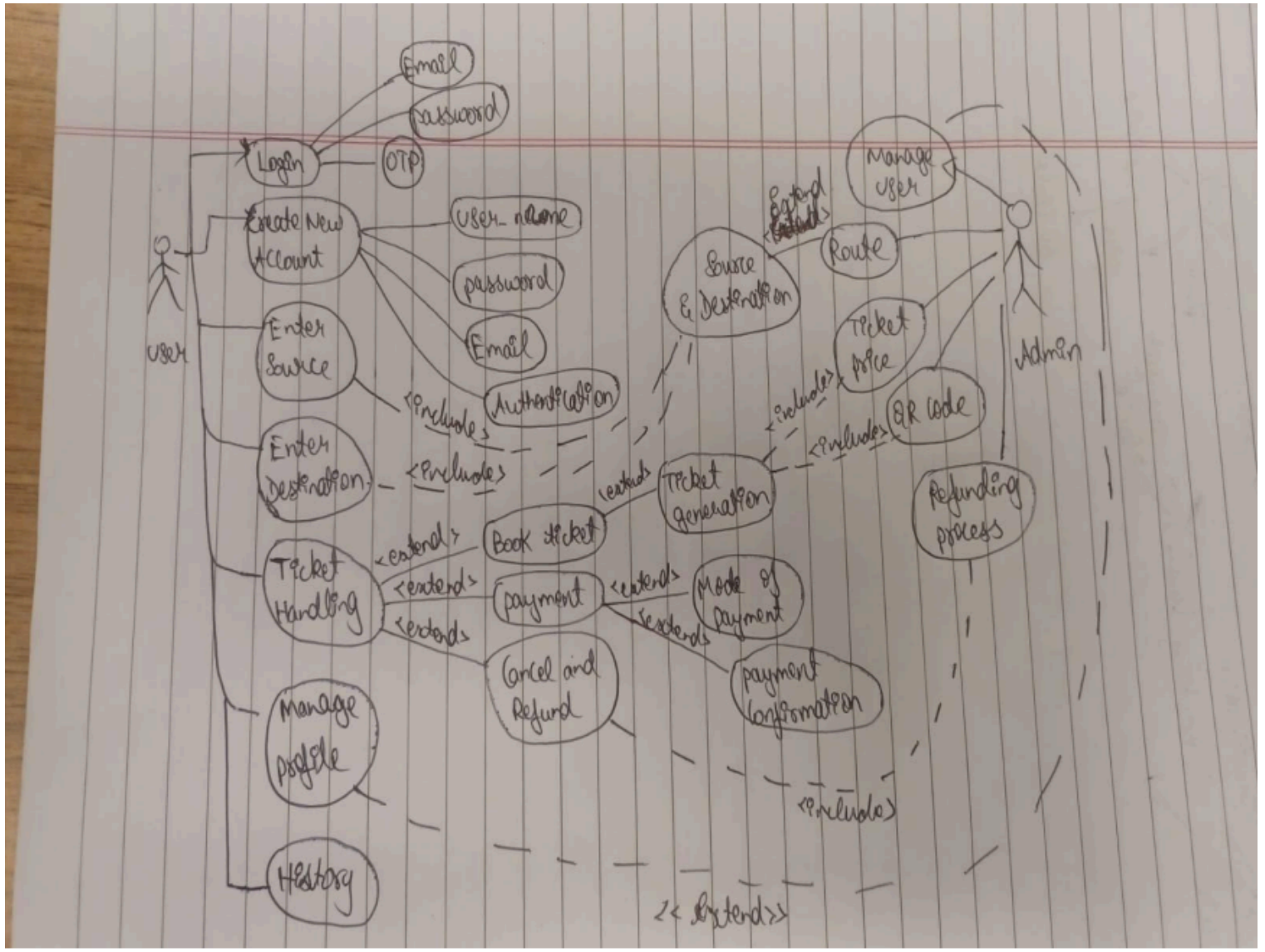
a)Major Case


User
Start
Select Route
Ticket Confirmed
View generated ticket
View all tickets
End
Ticket Booking
Show source and destination
Choose source & destination
process ticket booking
payment
choose payment Mode
View ticket report
payment transactions
Cancel ticket
Refund

b)Minor Case:



4.Use Case Diagram


Email
password
OTP
Login
Create New Account
User_name
password
Email
User
Enter Source
Enter Destination
Authentication
<<include>>
<<include>>
Source & Destination
Manage user
Route
Ticket price
QR code
Admin
<<include>>
<<include>>
Ticket generation
created
Refunding process
Book ticket
<<extend>>
Ticket Handling
<<extend>>
<<extend>>
payment
<<extend>>
<<extend>>
Mode of payment
payment conformation
Cancel and Refund
Manage profile
History
<<include>>
<<extend>>