*UE21CS352B - Object Oriented Analysis & Design using Java*

# Mini Project Report

## "Metro Management System"

***Submitted by:***


| | |
|---|---|
| **Syeda Saniya** | **PES1UG21CS658** |
| **Stuti Pathak** | **PES1UG21CS629** |
| **Syed Munzer Nouman** | **PES1UG21CS656** |
| **Suraj Kashyap** | **PES1UG21CS645** |

*6th Semester K Section*

**Prof. Bhargavi M Joshi**
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

## Problem Statement:

The Metro Management System seeks to improve the efficiency and effectiveness of a metropolitan rail network by tackling existing challenges and enhancing overall service quality. With a focus on optimizing operations and improving passenger experience, the management system addresses issues such as employee management, train scheduling and accessibility for passengers.

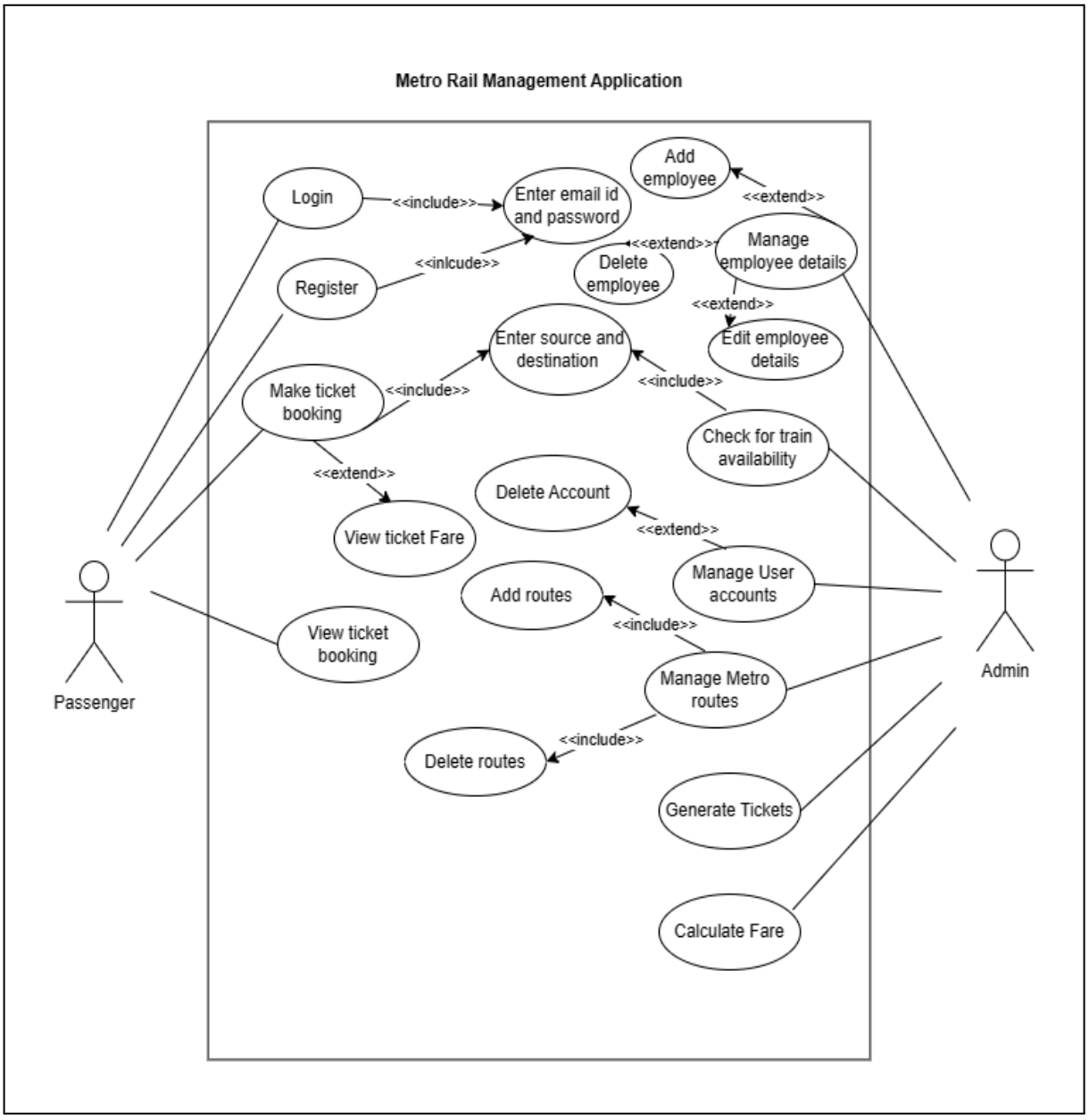
Key stakeholders, including metro authorities, maintenance staff, and passengers, are integral to the success of the Metro Management System. By addressing the diverse needs and concerns of these stakeholders, the system aims to foster collaboration and support for its implementation.

The expected benefits of this Metro Management System implementation include improved reliability of train services as everything is clearly specified in the application, reduced passenger wait times since the passengers are well informed through the application with regard to the time of arrival of the trains, and overall cost savings through more efficient resource allocation.

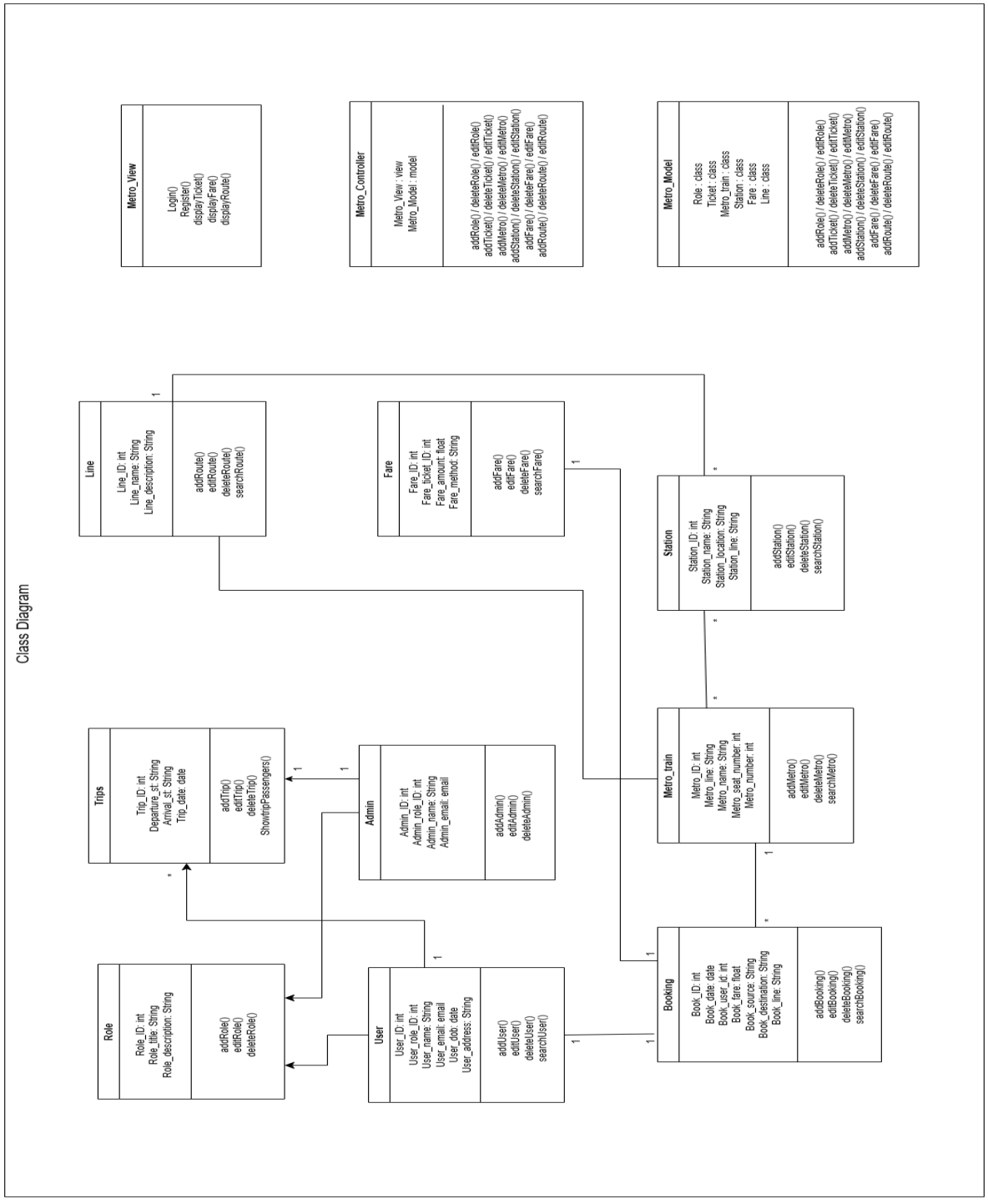
The implementation plan involves the development of a comprehensive technological infrastructure, which includes detailed schedules, passenger information, passenger bookings, train availability, platform management, employee management and many other features that add to the complex and intricate nature of this management system.

# Models

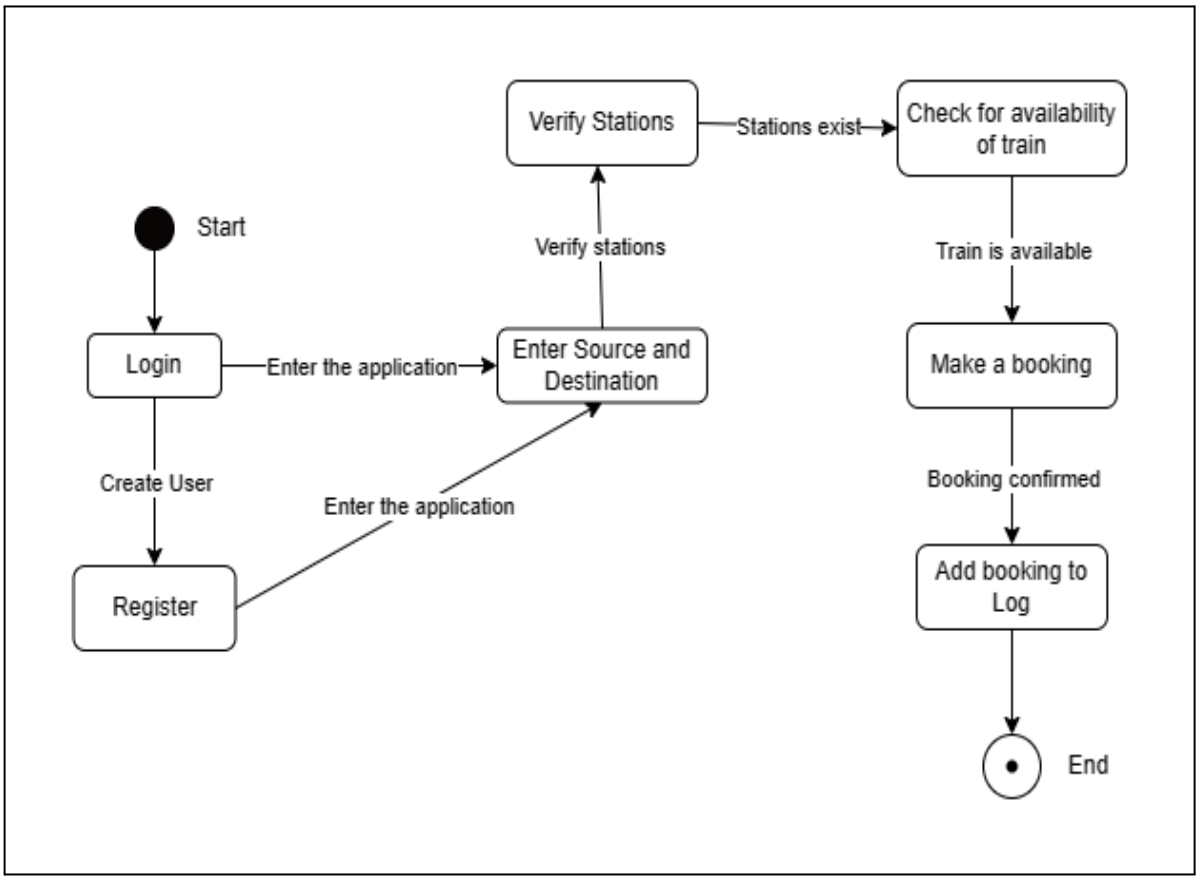
## Use Case Diagram



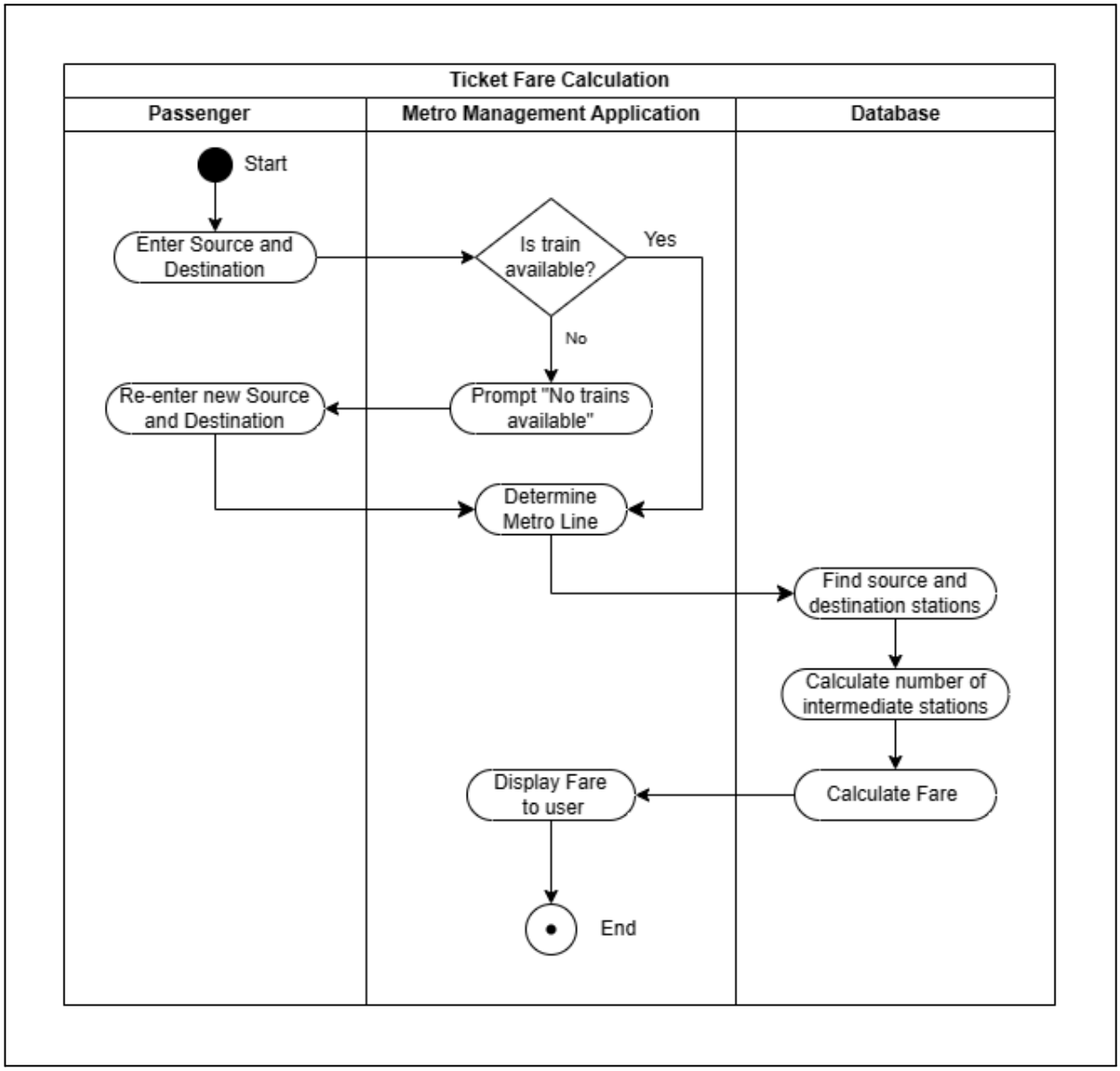
# Class Models Diagram



## State Diagram



## Activity Diagram



## Architecture Patterns

1. *Model-View-Controller (MVC):*
    a. Model: Classes like Employee, Trip, Train, etc., represent the model objects. These classes encapsulate data and define the business logic.
    b. View: Swing components such as JFrame, JPanel, JButton, etc., are used to create the user interface (UI) elements.
    c. Controller: Action listeners and event handlers control the interaction between the model and the view. For example, ActionListener implementations in various classes handle user input and trigger appropriate actions.

2. *Singleton Pattern:* The Database class follows the Singleton pattern to ensure that only one instance of the database connection is created and shared across the application.

3. *Factory Method Pattern:* The Database class has a static factory method getInstance() to create and return the singleton instance of the database connection.

4. *DAO (Data Access Object) Pattern:* Classes like EmployeesDatabase, TripsDatabase, etc., encapsulate the database operations related to their respective entities (employees, trips, trains, etc.). They provide methods to perform CRUD (Create, Read, Update, Delete) operations on the database tables.

5. *Separation of Concerns:* The code separates concerns by dividing functionalities into different classes. For example, UI-related code is separated from database operations, and each entity has its own set of classes responsible for database interaction.

6. *Event-Driven Architecture:* The Swing-based UI is event-driven, where user actions such as button clicks trigger corresponding event handlers to execute specific actions.

7. *Layered Architecture:* The application can be structured into layers such as presentation layer (UI), business logic layer (model), and data access layer (DAOs). This separation allows for easier maintenance and scalability.

8. *Dependency Injection:* Some classes, such as ModifyList, receive dependencies (like the database instance) through their constructors, allowing for easier testing and flexibility.

9. *Observer Pattern:* In BookTrip.java, the DocumentListener implementation serves as an observer for changes in the numOfTickets text field. It updates the total price dynamically based on the number of tickets entered.

10. *Template Method Pattern:* Though not explicitly defined, there's a semblance of the Template Method pattern in the structure of GUI creation across classes. Methods like JLabel(), JComboBox(), and JButton() in the GUI class seem to follow a template for creating Swing components with consistent styles.

*Exception Handling Strategy:* The code utilizes a strategy for exception handling by displaying error messages in dialog boxes (JOptionPane) and handling exceptions at the point where they are most relevant.

*Exception Handling Strategy:* The code utilizes a strategy for exception handling by displaying error messages in dialog boxes (JOptionPane) and handling exceptions at the point where they are most relevant.

## MVC

*Model*

1. Employee: Represents an employee's data, including ID, name, email, telephone, salary, and position.
2. Passenger: Represents passenger data, including ID, name, email, and telephone.
3. Trip: Represents trip data, including ID, destination, departure date, arrival date, and price.

*View*

1. EditEmployee: Allows users to edit employee information. It displays input fields for ID, name, email, telephone, salary, and position.
2. BookTrip: Allows users to book a trip for a passenger. It displays input fields for selecting a passenger, entering the number of tickets, and showing trip details.
3. ShowTripPassengers: Displays the list of passengers for a selected trip.

*Controller*

The controller acts as an intermediary between the model and view. It handles user input, processes requests, and updates the view based on changes in the model. In the provided code, the following components serve as controllers:

1. Action Listeners: These are implemented in various classes to handle user interactions such as button clicks and combo box selections. They trigger actions in response to user input, such as submitting employee edits, booking trips, or refreshing data.
2. Document Listeners: Used in BookTrip.java to dynamically update the total price based on the number of tickets entered.

*MVC Interaction*

1. View-Model Interaction: Views, such as EditEmployee and BookTrip, interact directly with models, retrieving data to display in the user interface and updating model data based on user input.
2. View-Controller Interaction: Views communicate user actions (e.g., button clicks, text input) to controllers through event listeners. Controllers then update the model accordingly.
3. Model-Controller Interaction: Controllers interact with models to perform operations such as fetching data, updating records, or deleting entries from the database. After model updates, controllers may notify views to refresh the display.

*Proper MVC Architecture*

In a proper MVC architecture, the separation of concerns between the model, view, and controller is well-defined and maintained. Models encapsulate data and business logic, views handle user interface presentation, and controllers orchestrate interactions between the model and view without tightly coupling them. The provided code adheres to these principles by clearly separating the responsibilities of each component and promoting modularity, reusability, and maintainability.

# Design Principles

1. *Single Responsibility Principle (SRP):* Each class has a single responsibility. For example, EmployeesDatabase handles operations related to employees, PassengersDatabase handles passenger-related operations, and so on.

2. *Open/Closed Principle (OCP)*: The code is designed to be easily extendable without modifying existing code. For instance, new features can be added by creating new classes rather than modifying existing ones.

3. *Liskov Substitution Principle (LSP):* Subtypes can be substituted for their base types without altering the correctness of the program. For example, subclasses of Employee and Passenger can be used interchangeably with their parent classes.

4. *Dependency Inversion Principle (DIP):* High-level modules do not depend on low-level modules. Instead, both depend on abstractions. For example, the EditEmployee class depends on the Database interface rather than specific implementations.

5. *Interface Segregation Principle (ISP):* Interfaces are tailored to specific client requirements. For example, the Database interface provides only the methods necessary for database interactions.

6. *Don't Repeat Yourself (DRY):* The code avoids duplication by centralizing common functionality. For instance, GUI elements are created using utility methods, reducing redundancy.

7. *Encapsulate What Varies:* Variations in behavior are encapsulated within classes, allowing changes to be localized. For example, the ModifyList class encapsulates different database operations, which can be modified independently.

8. *Composition over Inheritance:* Instead of relying heavily on inheritance, the code emphasizes composition, favoring the use of interfaces and object composition to achieve flexibility and modularity.

By adhering to these design principles, the codebase promotes maintainability, extensibility, and readability, contributing to a robust and scalable software architecture.

## Design Patterns

1. *Singleton Pattern:* Although not explicitly shown in the provided code, it's common in database-related classes like Database, where only one instance of the database connection is needed throughout the application's lifecycle. The Singleton pattern ensures that only one instance of a class is created and provides global access to that instance.

2. *Factory Method Pattern:* In the GUI class, methods like JLabel, JComboBox, JTextField, JButton, etc., serve as factory methods for creating Swing GUI components. These methods encapsulate the process of creating complex objects (Swing components) and provide a simple interface for clients to create instances without exposing the underlying implementation details.

3. *Observer Pattern:* The DocumentListener interface used in BookTrip.java employs the Observer pattern. It allows an object (the listener) to observe changes in another object (the text field) and respond accordingly. The DocumentListener listens for changes in the text field and triggers appropriate actions when text is inserted, removed, or changed.

4. *Model-View-Controller (MVC) Pattern:* The MVC architecture is used throughout the provided code. It separates the application into three interconnected components: Model (data and business logic), View (user interface), and Controller (handles user input and updates the model and view accordingly). Classes like EditEmployee, BookTrip, and ShowTripPassengers represent the View, while models like Employee, Passenger, and Trip represent the Model. The controllers include various action listeners that handle user interactions and update the model and view accordingly.

5. *Data Access Object (DAO) Pattern:* The EmployeesDatabase, PassengersDatabase, and TripsDatabase classes can be considered as DAOs. They encapsulate the logic for interacting with the database and

provide methods for performing CRUD (Create, Read, Update, Delete) operations on database entities (employees, passengers, trips). This pattern helps abstract the database operations from the rest of the application, promoting separation of concerns and maintainability.

6. *Strategy Pattern:* Although not explicitly implemented in the provided code, the Strategy pattern can be applied to the way different database operations (e.g., editing employee, booking trip) are encapsulated in separate methods within the DAO classes. This allows for easy swapping of database implementations or strategies without modifying the client code.

7. *Facade Pattern:* The ModifyList class embodies the Facade pattern by offering a streamlined interface for modifying lists of entities like employees, passengers, and trips. It shields clients from the complexities of database interactions by encapsulating operations with various DAO classes behind a single set of methods. This abstraction of complexity promotes encapsulation and modularity, simplifying maintenance and modification of the system. By providing a unified entry point, ModifyList enhances code readability and understandability, making client code more expressive and self-explanatory. Overall, ModifyList serves as an effective facade, abstracting away implementation details and offering a clear, concise interface for modifying lists within the system.

**Github link to the Codebase (Repository should be public and accessible to all)**

https://github.com/San-1503/Metro-Management-System.git

## Individual contributions of the team member

SYEDA SANIYA (PES1UG21CS658):

- Implemented the MetroManagementSystem class, which was the base class for the application. It included the implementation of the files, Database.java, GUI.java, Main.java, ModifyList.java and ModifyList2.java.
- This class implemented MVC and other design patterns such as facade, etc along with principles like singleton, etc.
- Implemented the Trips class, which handled all the trips and bookings. This involved the implementations for AddTrip.java, BookTrip.java, EditTrip.java, ShowTripPassengers.java and Trip.java.

SYED MUNZER NOUMAN (PES1UG21CS656):

- Implemented the Employee class, which handled all the employee information and all actions in relation to managing and working on the details of employees.
- Involved the implementation of AddEmployee.java, EditEmployee.java and Employee.java.
- It utilized principles such as Singleton, DAO, etc.

STUTI PATHAK (PES1UG21CS629):

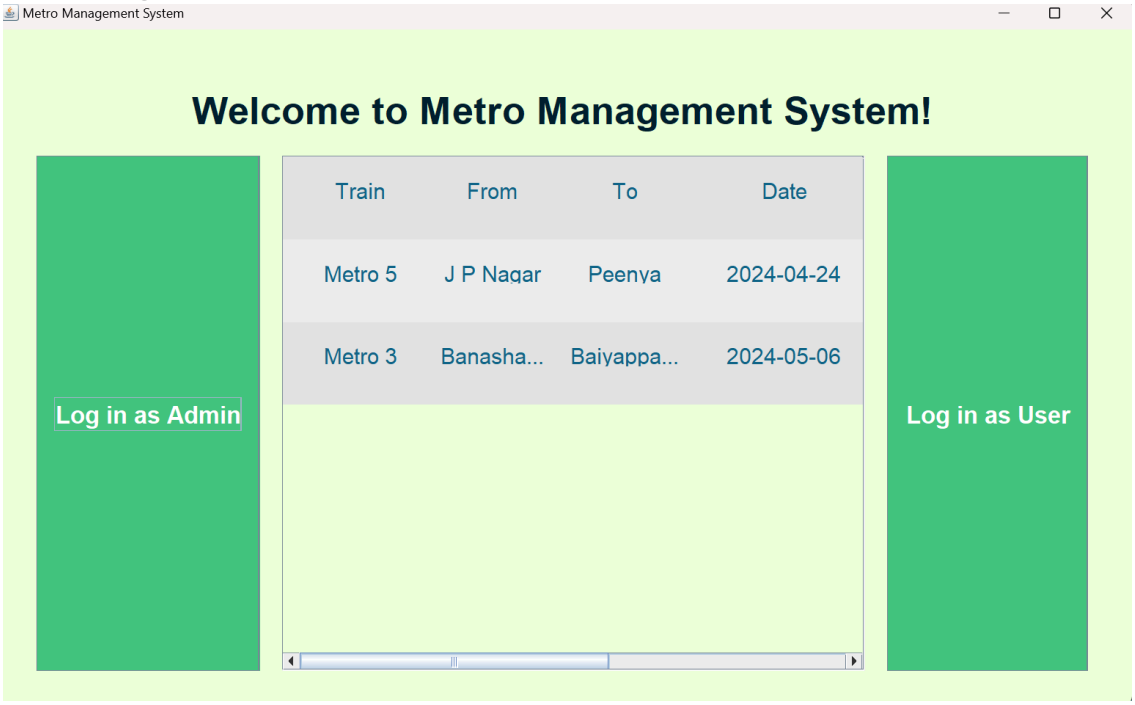
- Implemented the Passengers class, which handled all the passenger information and all actions in relation to managing and working on the details of passengers.
- Involved the implementation of AddPassenger.java, EditPassenger.java and Passenger.java.
- It utilized principles such as Singleton, DAO, etc.

SURAJ KASHYAP (PES1UG21CS645):

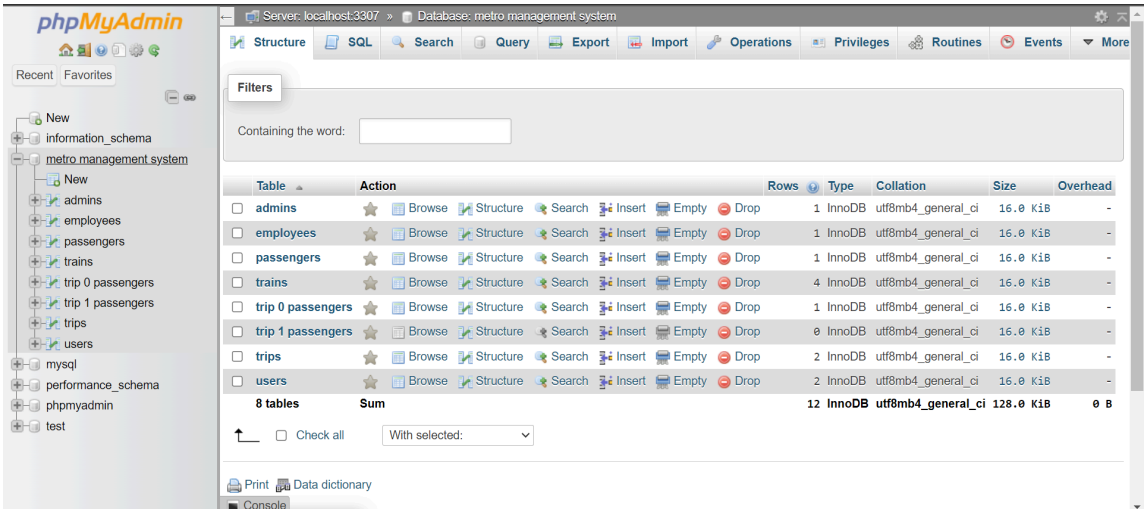
- Implemented the Trains class, which handled all the train information and all actions in relation to managing and working on the details of trains.
- Involved the implementation of AddTrain.java, EditTrain.java and Train.java.
- It utilized principles such as Singleton, DAO, etc.

**Screenshots with input values populated and output shown (Use white background screens).**

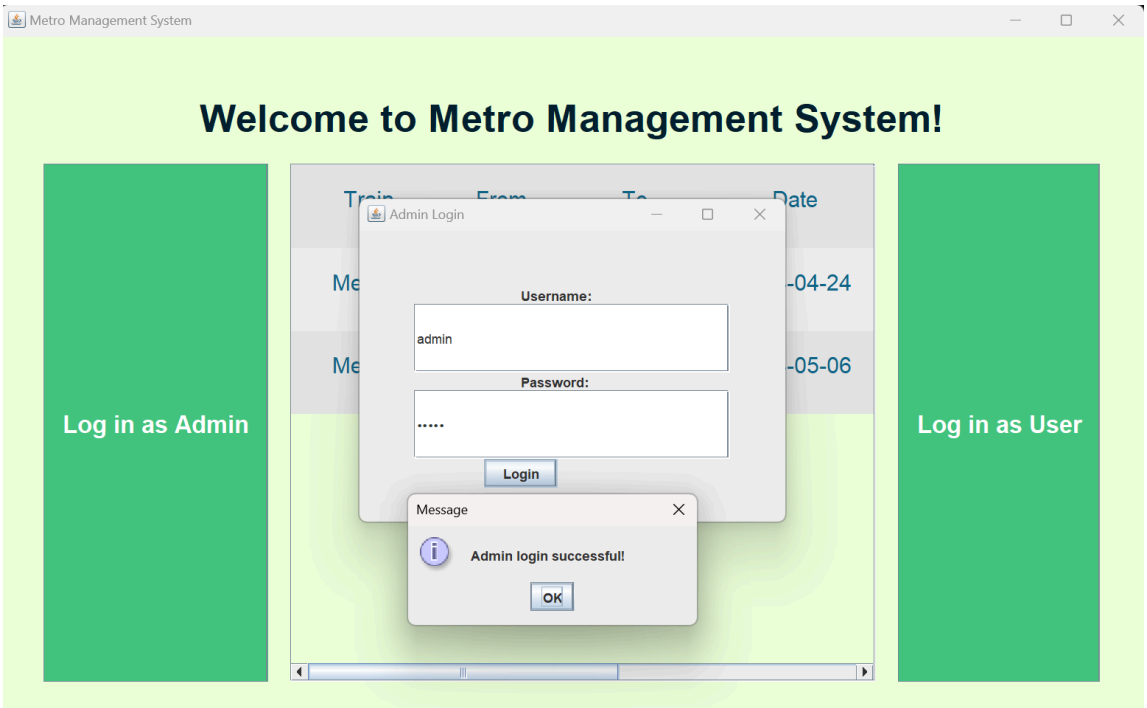
## Main Page:



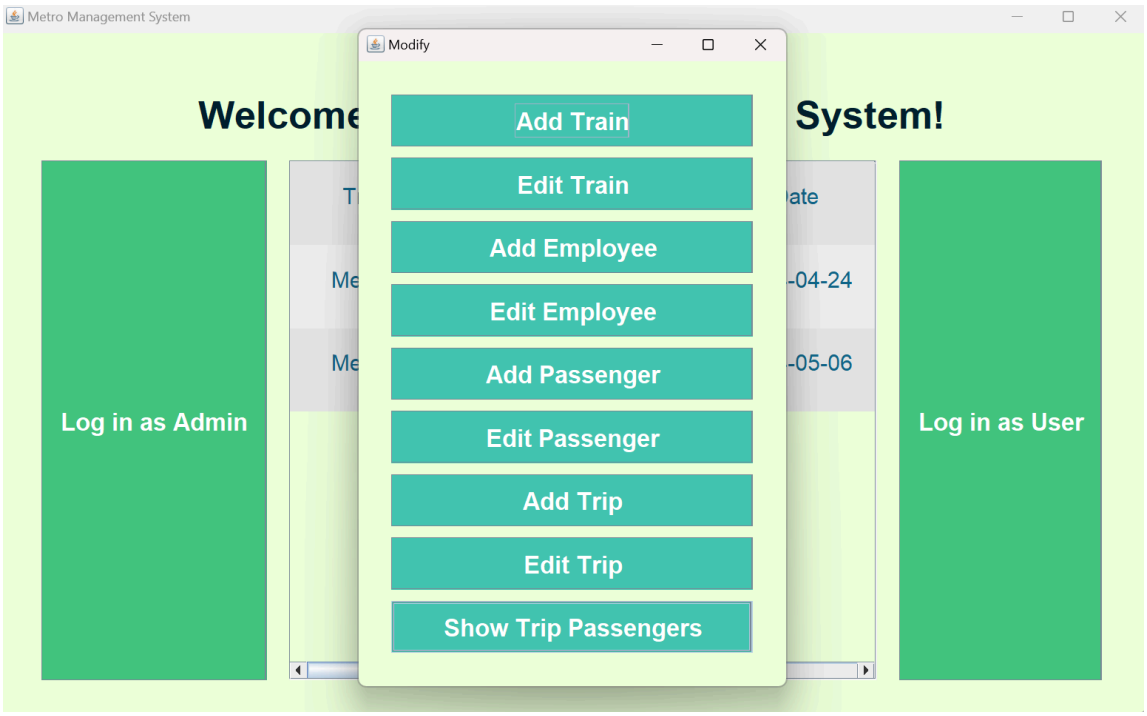
# Database:




# Admin Login:

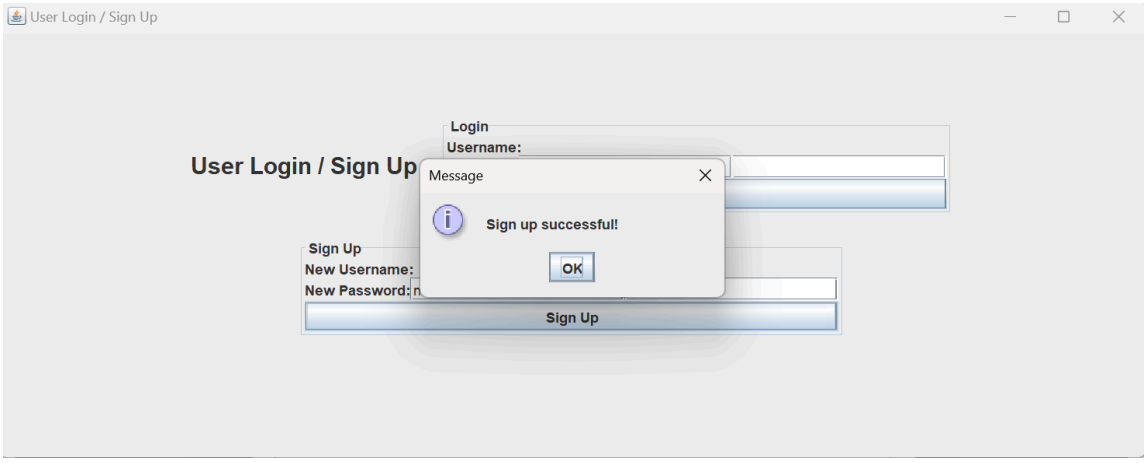


## Admin Options:

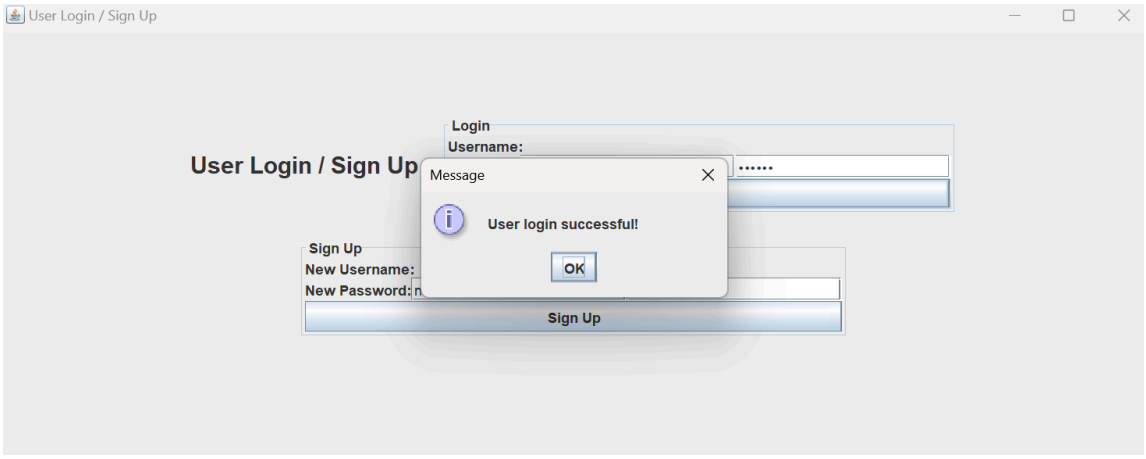



## User Login:

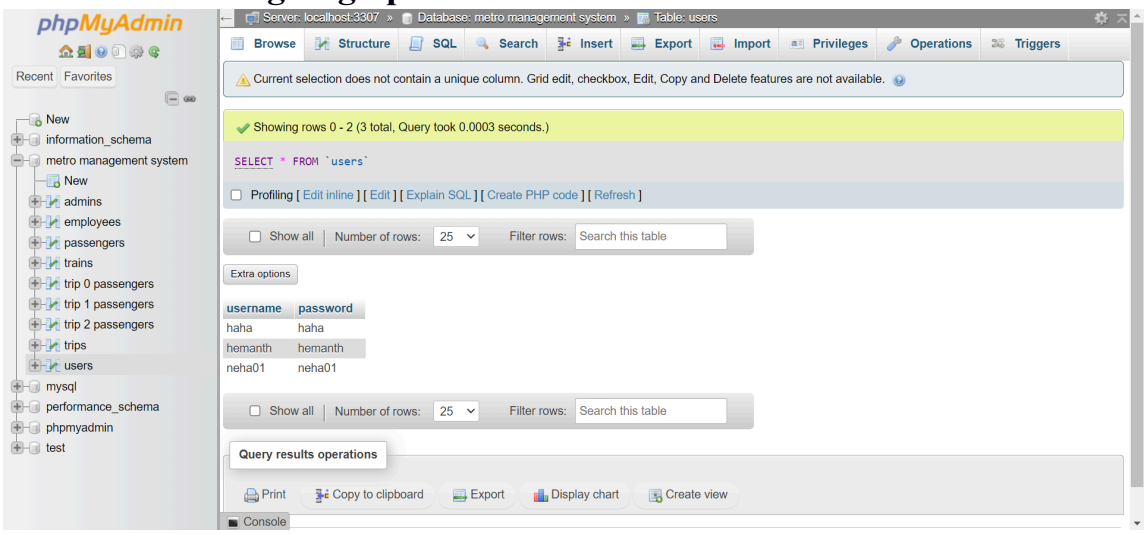
## Sign up:



## Log in:

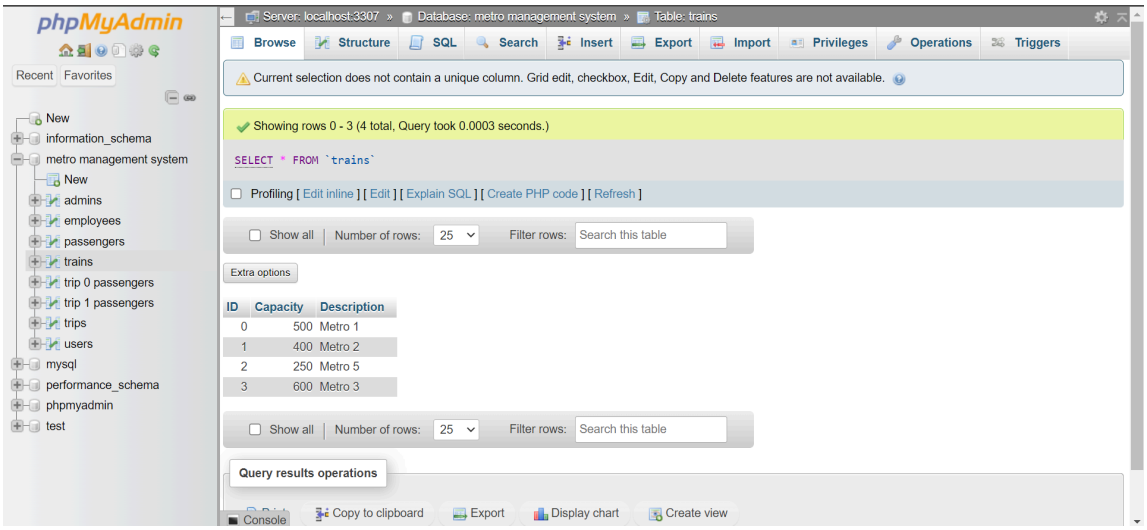



## Database after signing up:

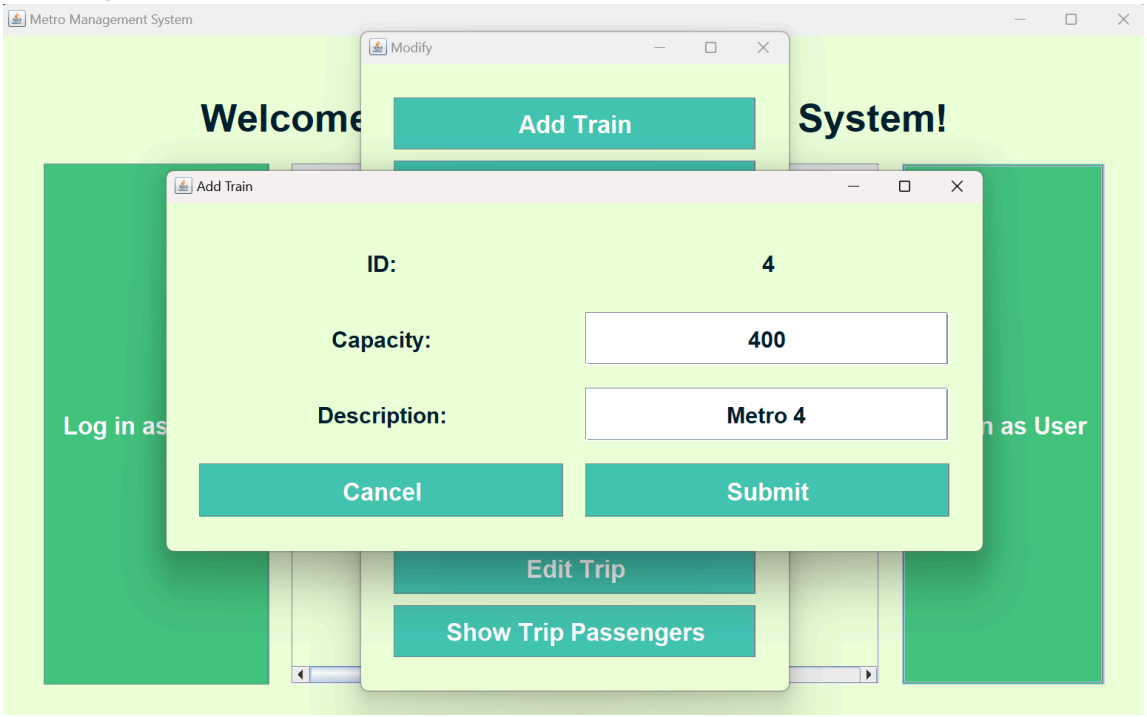



## <u>Add Train:</u>

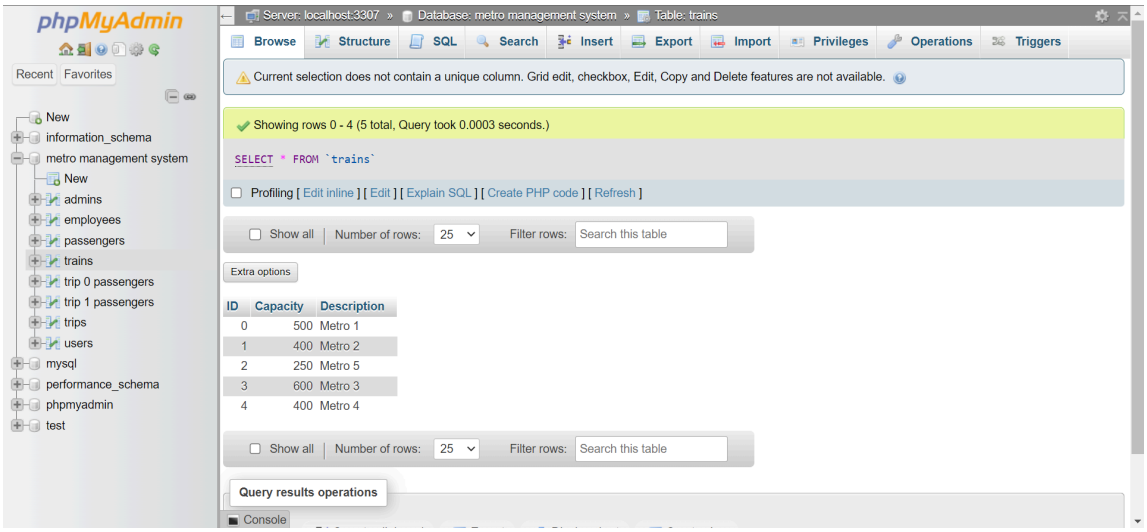
## Database before adding:




# Adding:

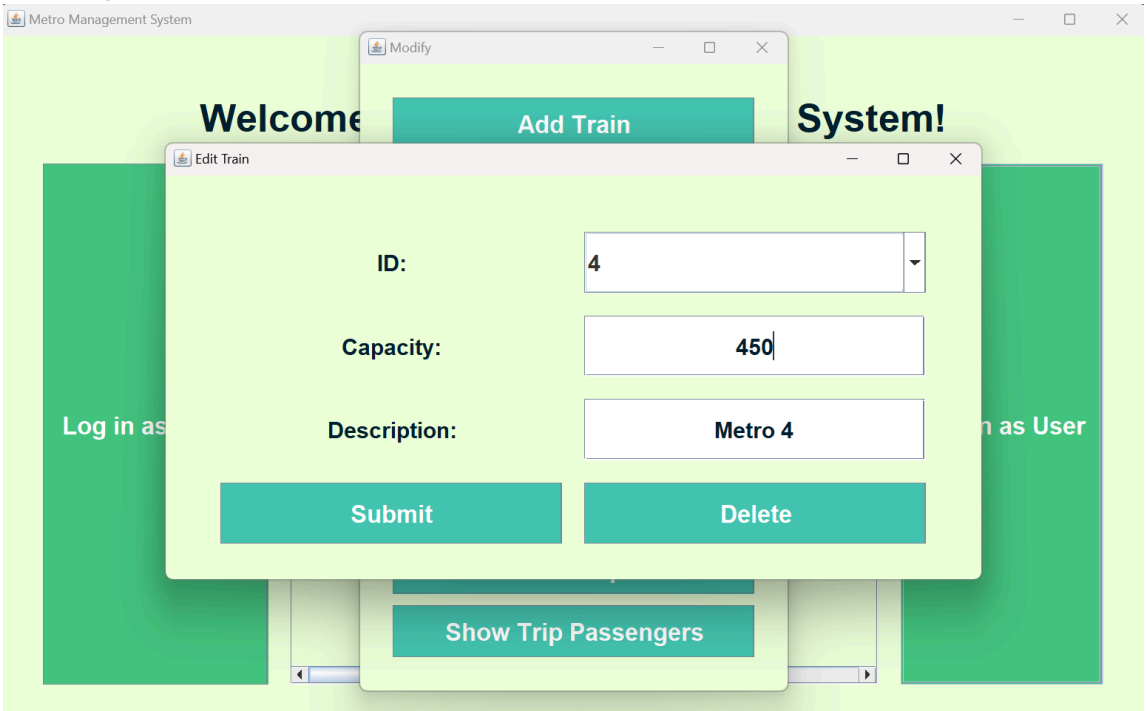



# Database after adding:

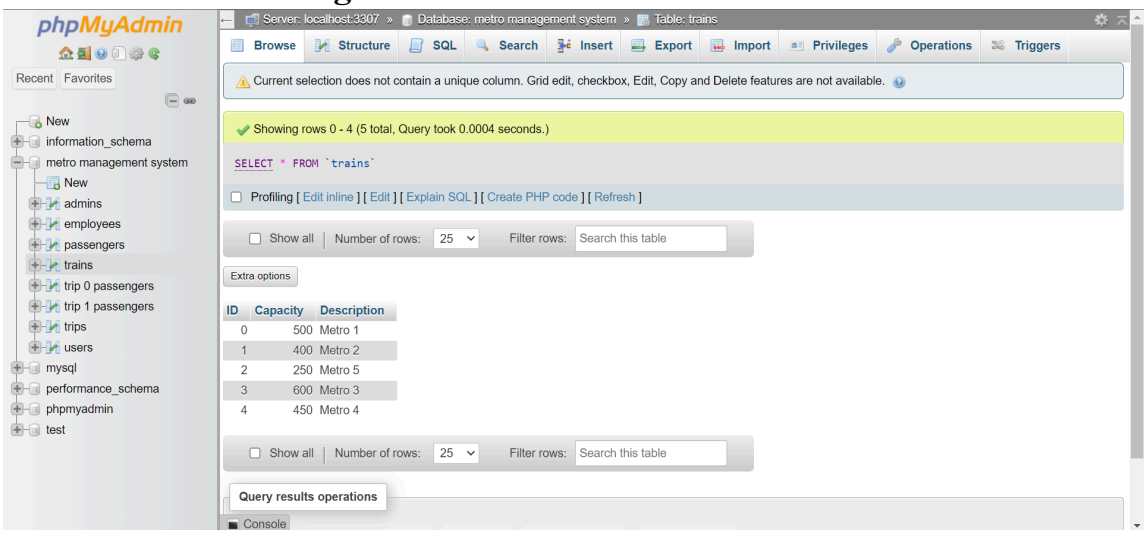



# Edit Train:

# Editing:

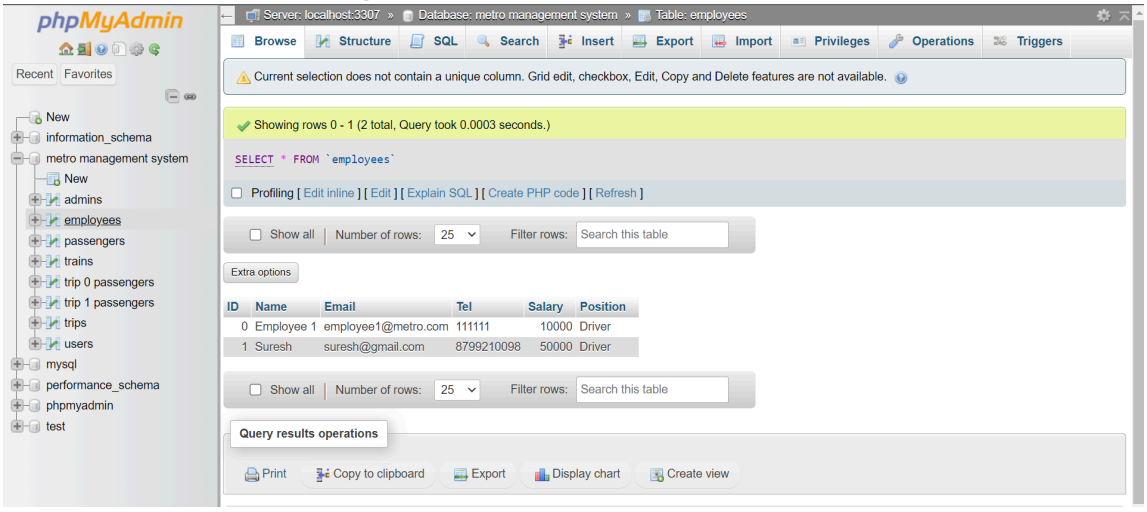


# Database after editing:

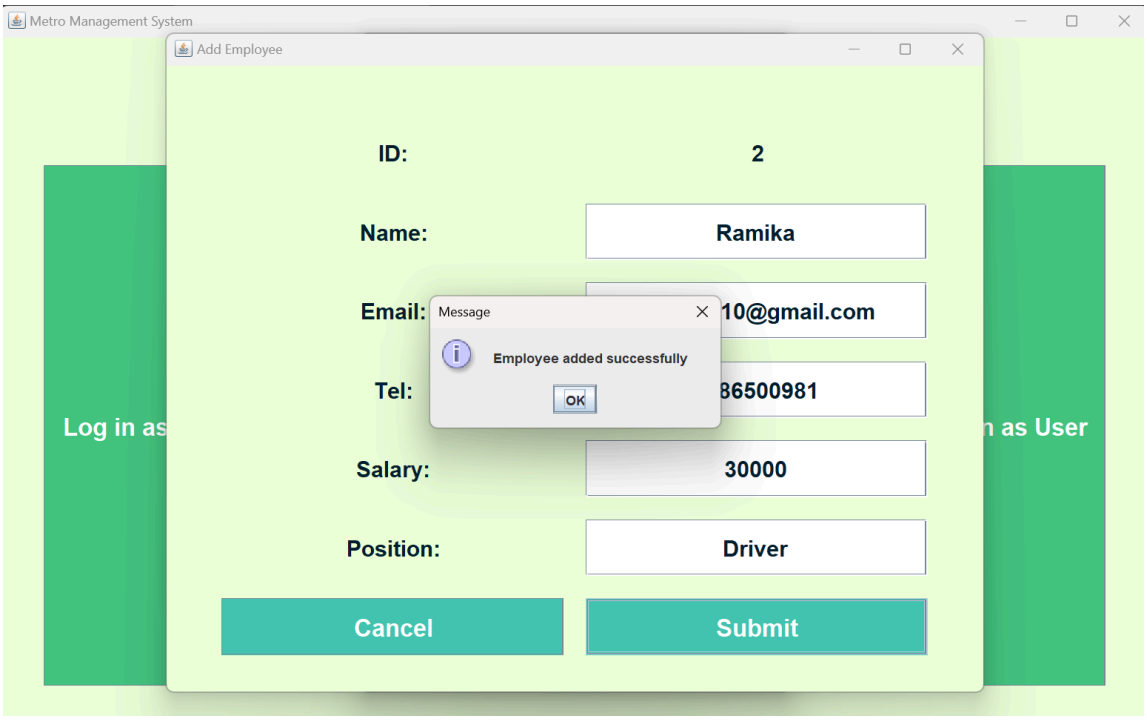



# Add Employee:

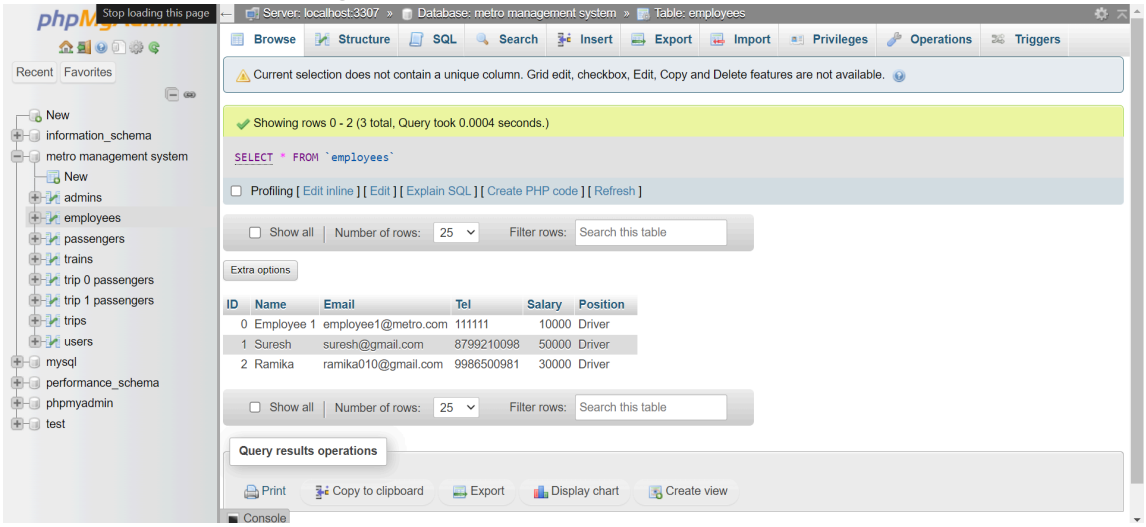
# Database before adding:




# Adding:

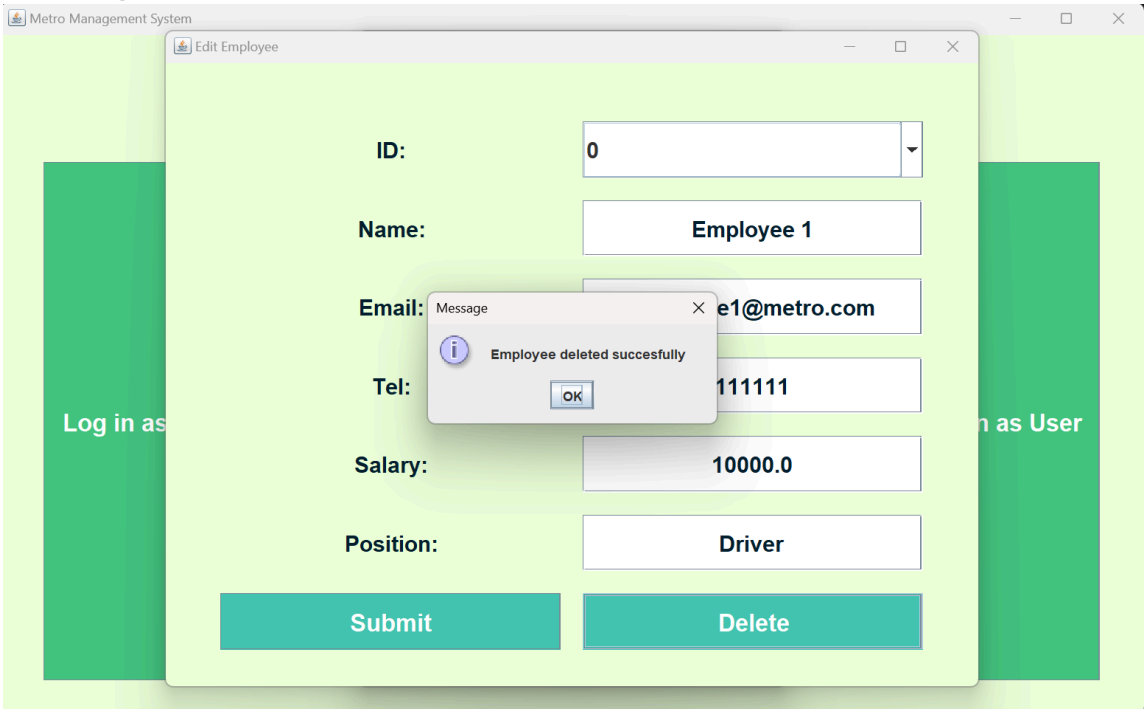



# Database after adding:

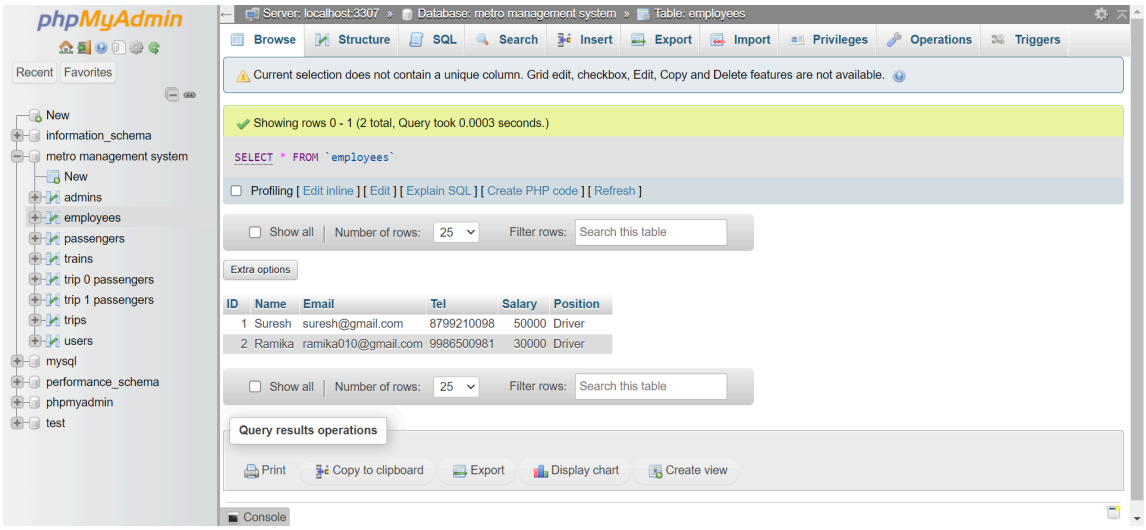



# Edit Employee:

## Deleting:

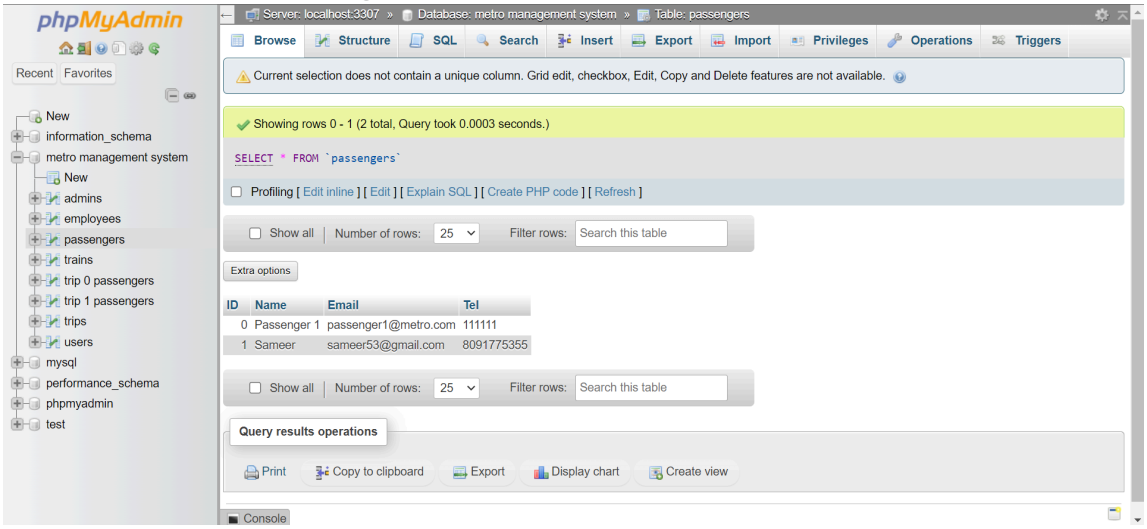



## Database after deletion:

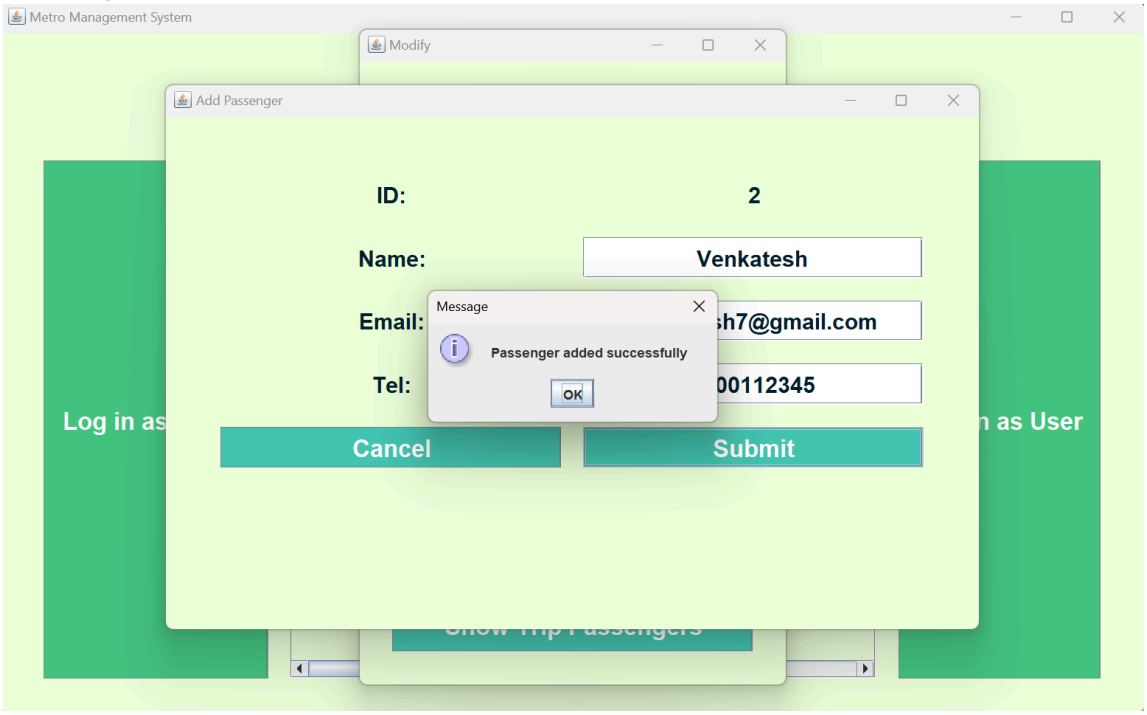



## Add Passenger:

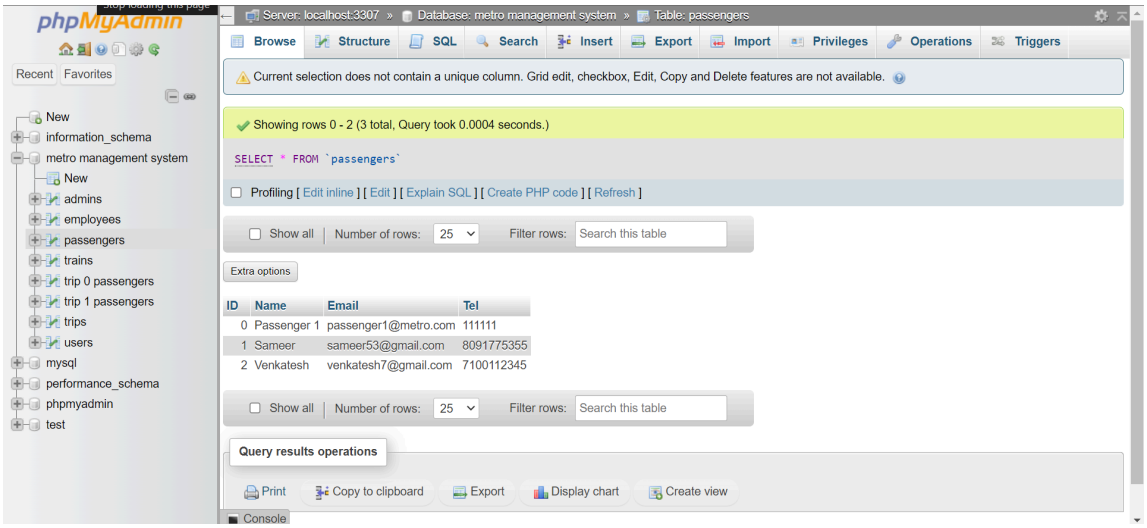
## Database before adding:




## Adding:

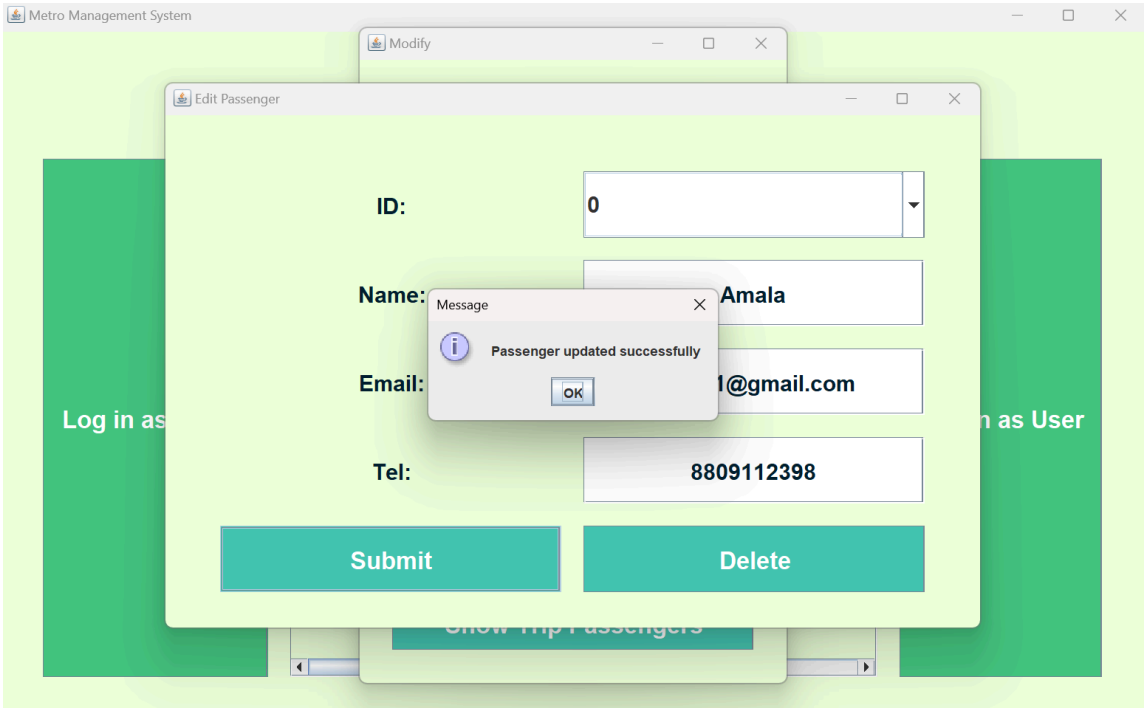



## Database after adding:

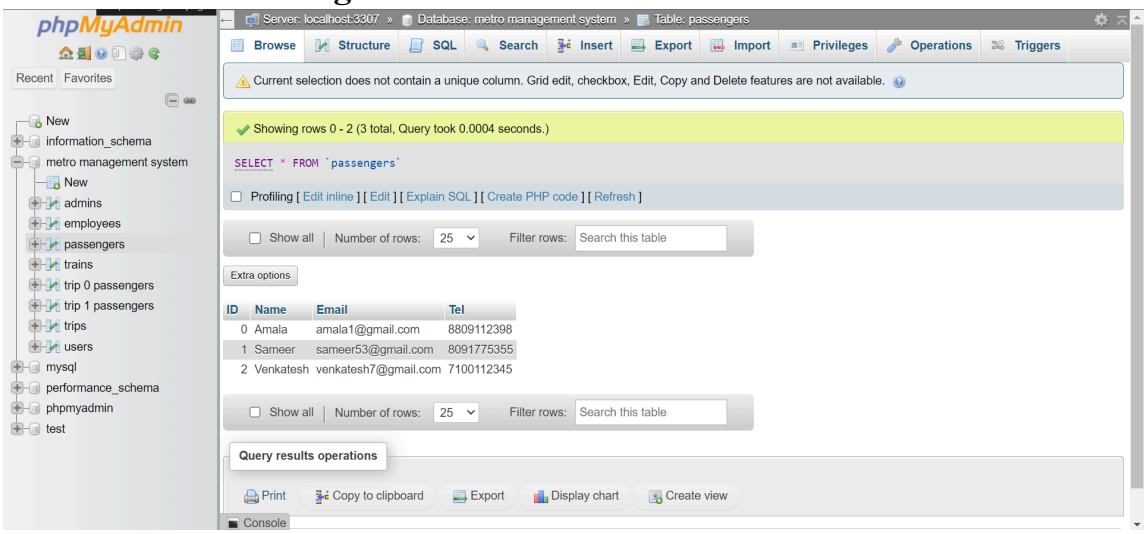



## Edit Passenger:

## Editing:

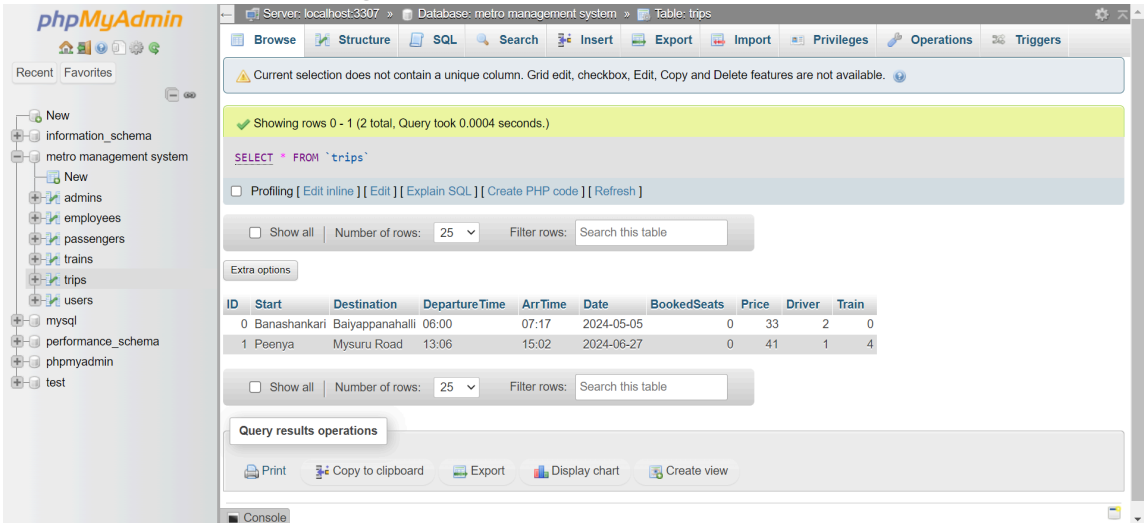


## Database after editing:

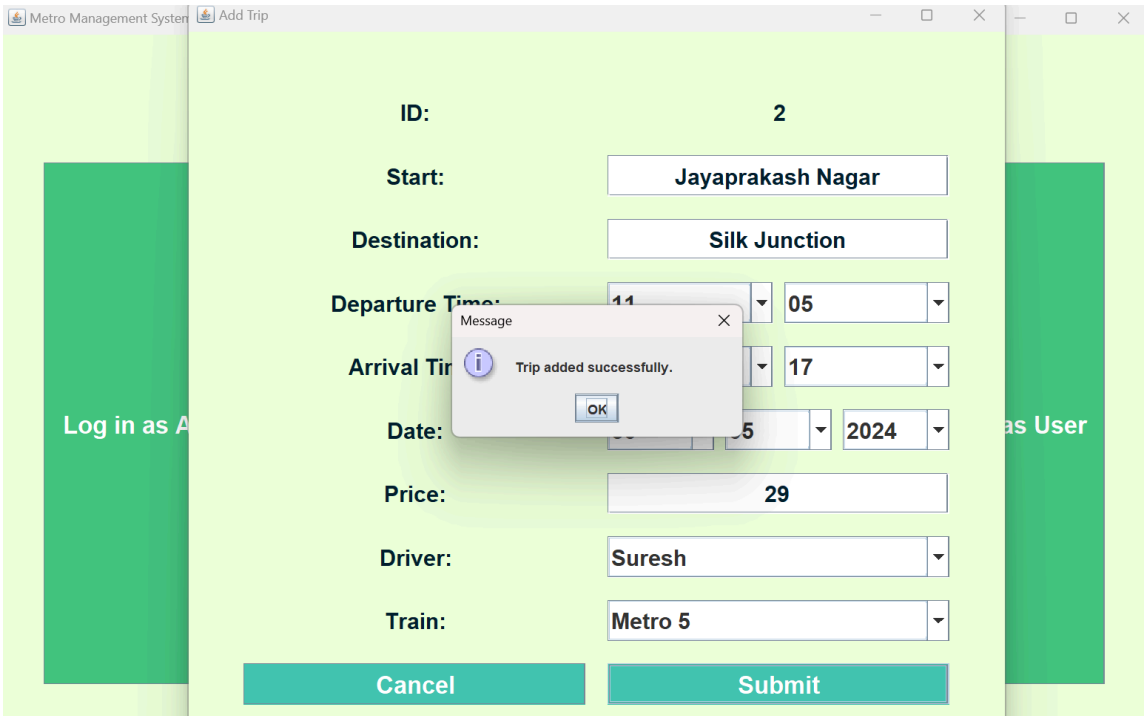



## Add Trip:

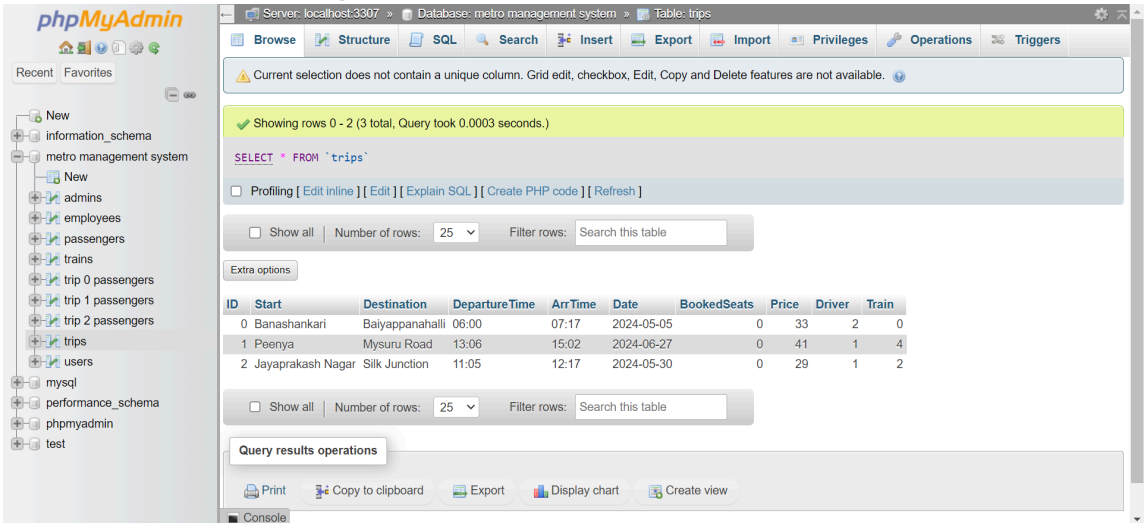
## Database before adding:




## Adding:

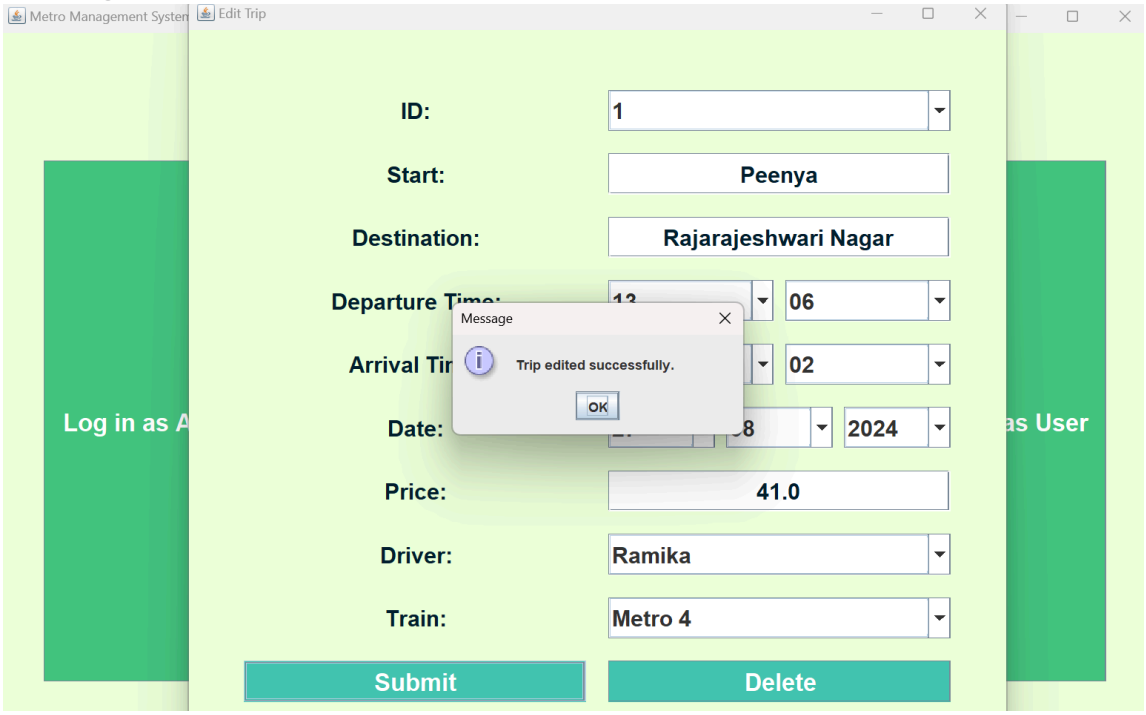



# Database after adding:

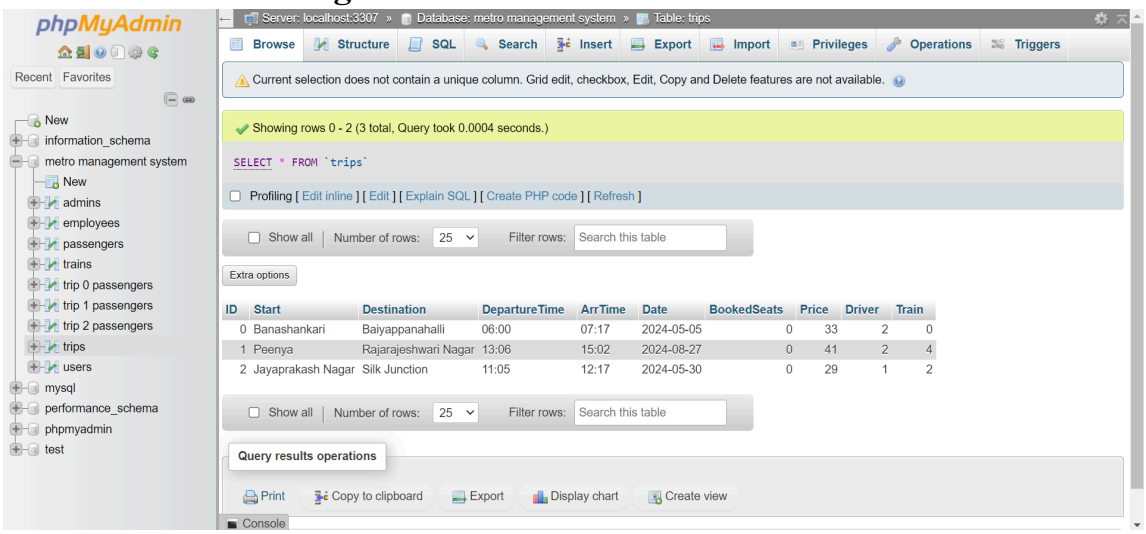



# Edit Trip:

## Editing:

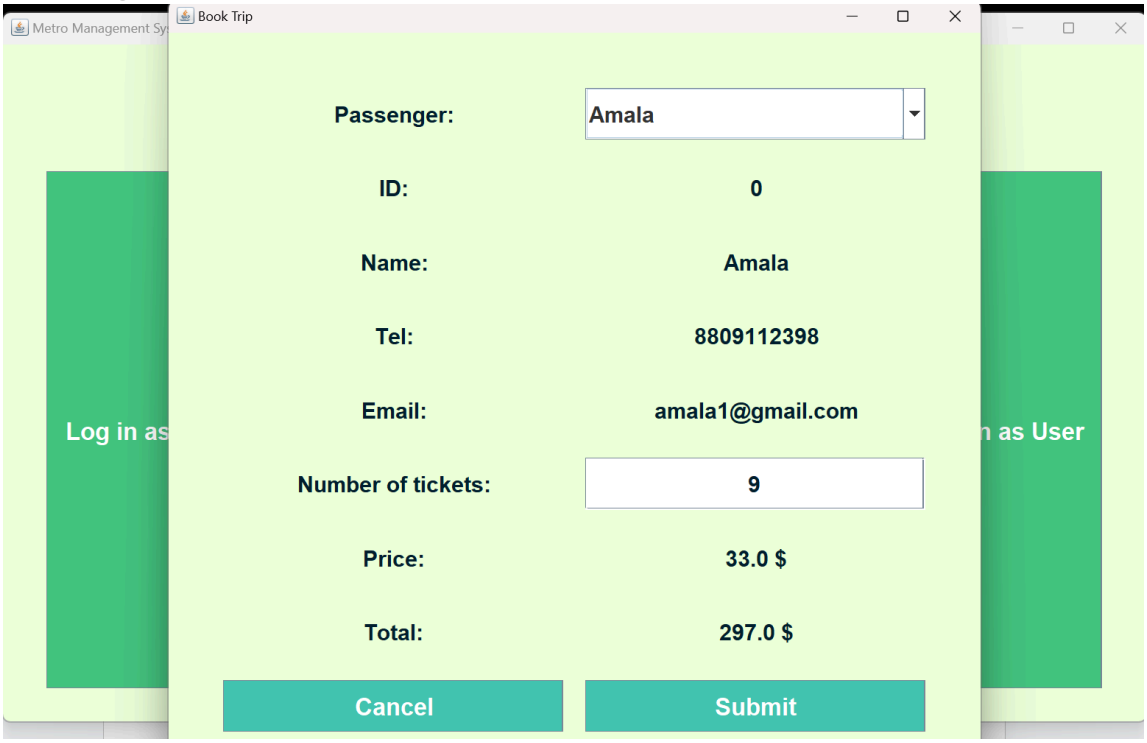



## Database after editing:

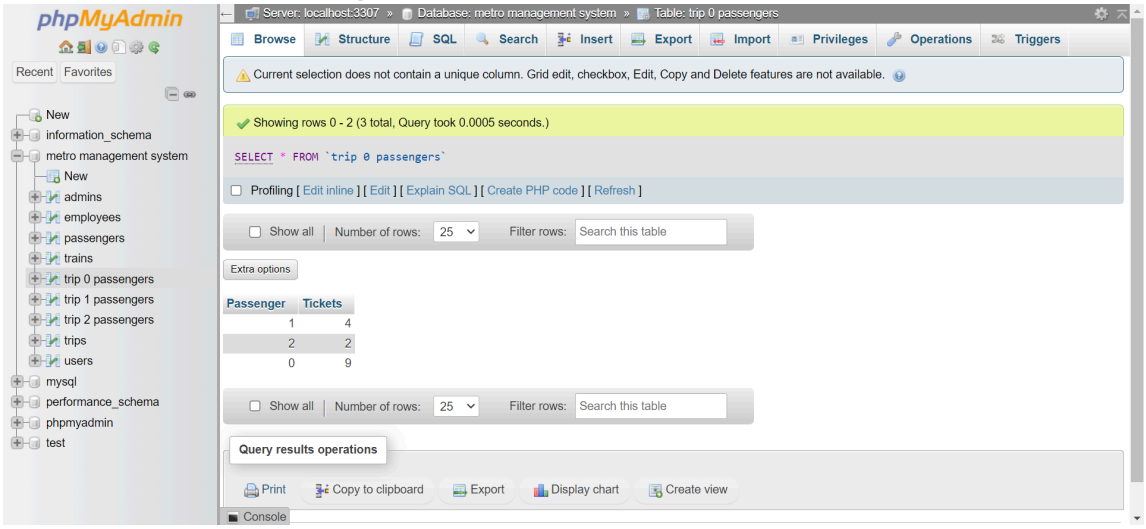



## Book Trip:

# Booking:




# Database after booking:



## Show Trip Passengers: