CASE STUDY ON

TRANSPORT MANAGEMENT SYSTEM

By,

Maheshwari M

Swetha G M

1.ABSTRACT

The Transport Management System (TMS) is a software solution designed to efficiently manage the various components involved in transportation services such as vehicles, drivers, trips, and bookings. This system helps automate the process of vehicle allocation, driver assignment, trip scheduling, and customer bookings while maintaining data integrity and reducing manual errors.The system is developed using Python for backend processing and MySQL as the database to store and manage transport-related records. The project follows a modular structure, adopting Object-Oriented Programming (OOP) principles and the Data Access Object (DAO) design pattern to promote scalability, maintainability, and clear separation of concerns.

The Transport Management System offers functionalities including adding new vehicles, managing driver information, creating and updating trips, and handling customer bookings with real-time availability status. This system helps streamline the operational flow for transport administrators while ensuring accurate record-keeping and smooth service delivery for users.

Overall, the project enhances efficiency, reduces operational overhead, and provides a strong foundation for expanding the system with additional features like route optimization, driver performance monitoring, and real-time GPS tracking in future iterations.

2.OBJECTIVE

The objective of the Transport Management System is to develop a reliable software solution that streamlines and optimizes the management of transportation resources within an organization. The system is designed to simplify core activities such as vehicle registration, driver assignment, trip planning, and booking management.

This project aims to:

- Enhance operational efficiency by automating repetitive and manual transport-related tasks.
- Establish a centralized platform for storing and managing all vehicle and driver records.
- Minimize human errors through systematic data handling and validation techniques.
- Simplify the process of trip scheduling and resource allocation.
- Provide scalable, maintainable, and secure architecture suitable for real-world transport operations.

- Improve data visibility for administrators, enabling informed decisions on fleet and driver management.

3.SCOPE

The Transport Management System (TMS) is designed to address the needs of organizations involved in the coordination and management of transportation services. The system focuses on automating the key operations related to vehicle, driver, trip, and booking management, providing an integrated and user-friendly platform for efficient transport resource planning.

This project covers:

💡 **Vehicle Management:**

Registering new vehicles, maintaining details like model, type, capacity, status, and updating or deleting vehicle records as needed.

💡 **Driver Management:**

Maintaining driver profiles including name, license information, contact details, and availability status.

💡 **Trip and Route Assignment:**

Allocating vehicles and drivers to specific trips based on availability and capacity, while ensuring optimized usage of resources.

💡 **Booking Management:**

Allowing the creation and management of customer bookings, assigning vehicles to bookings, and handling booking modifications or cancellations.

💡 **Database Integration:**

Reliable storage and retrieval of all transport-related data using MySQL, with structured DAO (Data Access Object) patterns for clean database operations.

4.SOFTWARE REQUIREMENTS:

COMPONENT	SPECIFICATION
Programming Language	Python 3.11 or higher
IDE / Code Editor	PyCharm / VS Code
Database	MySQL Server
Python Libraries	mysql-connector-python, unittest, configparser
Operating System	Windows 10 / 11 or Linux

5.SQL TABLE CREATION(Handled by Maheshwari):

```
create database TransportManagement;
use TransportManagement;
CREATE TABLE Vehicles (
    VehicleID INT AUTO_INCREMENT PRIMARY KEY,
    Model VARCHAR(255) NOT NULL,
    Capacity DECIMAL(10, 2) NOT NULL,
    Type VARCHAR(50) CHECK (Type IN ('Truck', 'Van', 'Bus')),
    Status VARCHAR(50) CHECK (Status IN ('Available', 'On Trip', 'Maintenance'))
);
desc vehicles;

CREATE TABLE Routes (
    RouteID INT AUTO_INCREMENT PRIMARY KEY,
    StartDestination VARCHAR(255) NOT NULL,
    EndDestination VARCHAR(255) NOT NULL,
```

```
    Distance DECIMAL(10, 2) NOT NULL CHECK (Distance > 0)
);
desc routes;

CREATE TABLE Trips (
    TripID INT AUTO_INCREMENT PRIMARY KEY,
    VehicleID INT NOT NULL,
    RouteID INT NOT NULL,
    DepartureDate DATETIME NOT NULL,
    ArrivalDate DATETIME NOT NULL,
    Status VARCHAR(50) CHECK (Status IN ('Scheduled', 'In Progress', 'Completed', 'Cancelled')),
    TripType VARCHAR(50) DEFAULT 'Freight' CHECK (TripType IN ('Freight', 'Passenger')),
    MaxPassengers INT CHECK (MaxPassengers >= 0),
    FOREIGN KEY (VehicleID) REFERENCES Vehicles(VehicleID) ON DELETE CASCADE,
    FOREIGN KEY (RouteID) REFERENCES Routes(RouteID) ON DELETE CASCADE
);
desc trips;

CREATE TABLE Passengers (
    PassengerID INT AUTO_INCREMENT PRIMARY KEY,
    FirstName VARCHAR(255) NOT NULL,
    Gender VARCHAR(255) CHECK (Gender IN ('Male', 'Female', 'Other')),
    Age INT CHECK (Age >= 0),
    Email VARCHAR(255) UNIQUE NOT NULL,
    PhoneNumber VARCHAR(50) NOT NULL
);
desc passengers;

CREATE TABLE Bookings (
```

```
    BookingID INT AUTO_INCREMENT PRIMARY KEY,
    TripID INT NOT NULL,
    PassengerID INT NOT NULL,
    BookingDate DATETIME NOT NULL,
    Status VARCHAR(50) CHECK (Status IN ('Confirmed', 'Cancelled', 'Completed')),
    FOREIGN KEY (TripID) REFERENCES Trips(TripID) ON DELETE CASCADE,
    FOREIGN KEY (PassengerID) REFERENCES Passengers(PassengerID) ON DELETE CASCADE
);
desc bookings;

INSERT INTO Vehicles (Model, Capacity, Type, Status) VALUES
('Ford Transit', 15.00, 'Van', 'Available'),
('Mercedes Actros', 30.00, 'Truck', 'On Trip'),
('Volvo Bus', 50.00, 'Bus', 'Maintenance'),
('Scania Truck', 35.00, 'Truck', 'Available'),
('Toyota HiAce', 12.00, 'Van', 'On Trip'),
('MAN Coach', 55.00, 'Bus', 'Available'),
('Iveco Truck', 40.00, 'Truck', 'Maintenance'),
('Isuzu NPR', 25.00, 'Truck', 'Available'),
('Renault Master', 18.00, 'Van', 'On Trip'),
('Setra Bus', 60.00, 'Bus', 'Available');


INSERT INTO Routes (StartDestination, EndDestination, Distance) VALUES
('New York', 'Boston', 350.50),
('Los Angeles', 'San Francisco', 600.75),
('Chicago', 'Houston', 1085.20),
('Miami', 'Orlando', 250.40),
('Dallas', 'Austin', 195.60),
```

('Seattle', 'Portland', 280.30),

('Denver', 'Las Vegas', 1220.50),

('Atlanta', 'Charlotte', 450.80),

('San Diego', 'Phoenix', 590.40),

('Washington D.C.', 'Philadelphia', 230.90);

INSERT INTO Trips (VehicleID, RouteID, DepartureDate, ArrivalDate, Status, TripType, MaxPassengers) VALUES

(1, 1, '2025-04-01 08:00:00', '2025-04-01 14:00:00', 'Scheduled', 'Passenger', 15),

(2, 2, '2025-04-02 10:00:00', '2025-04-02 18:00:00', 'In Progress', 'Freight', 0),

(3, 3, '2025-04-03 07:30:00', '2025-04-03 21:45:00', 'Completed', 'Passenger', 50),

(4, 4, '2025-04-04 09:00:00', '2025-04-04 13:00:00', 'Scheduled', 'Freight', 0),

(5, 5, '2025-04-05 07:15:00', '2025-04-05 09:30:00', 'Scheduled', 'Passenger', 12),

(6, 6, '2025-04-06 08:00:00', '2025-04-06 12:45:00', 'Scheduled', 'Freight', 0),

(7, 7, '2025-04-07 11:30:00', '2025-04-07 20:30:00', 'Scheduled', 'Passenger', 55),

(8, 8, '2025-04-08 09:45:00', '2025-04-08 16:15:00', 'Completed', 'Passenger', 20),

(9, 9, '2025-04-09 10:30:00', '2025-04-09 17:45:00', 'Cancelled', 'Freight', 0),

(10, 10, '2025-04-10 07:00:00', '2025-04-10 10:45:00', 'Scheduled', 'Passenger', 25);

INSERT INTO Passengers (FirstName, Gender, Age, Email, PhoneNumber) VALUES

('John Doe', 'Male', 28, 'john.doe@example.com', '1234567890'),

('Jane Smith', 'Female', 32, 'jane.smith@example.com', '0987654321'),

('Alex Johnson', 'Other', 25, 'alex.johnson@example.com', '1122334455'),

('Michael Brown', 'Male', 45, 'michael.brown@example.com', '2233445566'),

('Emily Davis', 'Female', 29, 'emily.davis@example.com', '3344556677'),

('Sophia Wilson', 'Female', 36, 'sophia.wilson@example.com', '4455667788'),

('David Martinez', 'Male', 50, 'david.martinez@example.com', '5566778899'),

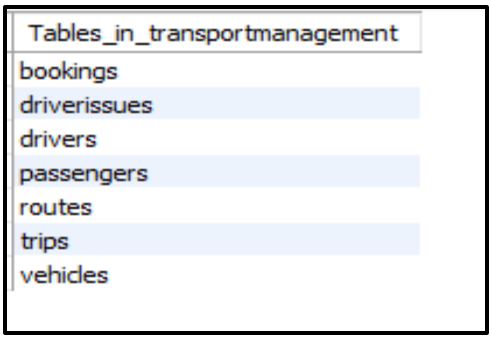
('Olivia Taylor', 'Female', 22, 'olivia.taylor@example.com', '6677889900'),

('William Anderson', 'Male', 40, 'william.anderson@example.com', '7788990011'),

('Emma Thomas', 'Female', 27, 'emma.thomas@example.com', '8899001122');

```
INSERT INTO Bookings (TripID, PassengerID, BookingDate, Status) VALUES
(1, 1, '2025-03-25 09:00:00', 'Confirmed'),
(1, 2, '2025-03-25 09:30:00', 'Confirmed'),
(3, 3, '2025-03-26 10:15:00', 'Completed'),
(4, 4, '2025-03-27 08:30:00', 'Cancelled'),
(5, 5, '2025-03-28 07:00:00', 'Confirmed'),
(6, 6, '2025-03-29 10:20:00', 'Confirmed'),
(7, 7, '2025-03-30 08:45:00', 'Confirmed'),
(8, 8, '2025-03-31 07:10:00', 'Completed'),
(9, 9, '2025-04-01 09:25:00', 'Cancelled'),
(10, 10, '2025-04-02 06:50:00', 'Confirmed');


select* from bookings;
select* from passengers;


CREATE TABLE Drivers (
    DriverID INT AUTO_INCREMENT PRIMARY KEY,
    Name VARCHAR(255) NOT NULL,
    LicenseNumber VARCHAR(100) UNIQUE NOT NULL,
    PhoneNumber VARCHAR(50) NOT NULL,
    Status VARCHAR(50) DEFAULT 'Available' CHECK (Status IN ('Available', 'Assigned', 'Inactive'))
);


desc drivers;
INSERT INTO Drivers (Name, LicenseNumber, PhoneNumber, Status) VALUES
('John Smith', 'DL123456789', '9876543210', 'Available'),
('Jane Doe', 'DL987654321', '8765432109', 'Assigned'),
('Michael Johnson', 'DL456789123', '7654321098', 'Available'),
```

```
('Emily Brown', 'DL789123456', '6543210987', 'Available'),
('Sophia Davis', 'DL321654987', '5432109876', 'Available'),
('David Wilson', 'DL654987321', '5432101234', 'Available'),
('Olivia Taylor', 'DL987321654', '4321098765', 'Available'),
('William Anderson', 'DL321987654', '3210987654', 'Available'),
('Emma Thomas', 'DL654123987', '2109876543', 'Available'),
('James Martinez', 'DL987654123', '1098765432', 'Available');

ALTER TABLE Trips
ADD COLUMN DriverID INT,
ADD FOREIGN KEY (DriverID) REFERENCES Drivers(DriverID) ON DELETE SET NULL;

desc passengers;

CREATE TABLE DriverIssues (
    IssueID INT AUTO_INCREMENT PRIMARY KEY,
    DriverID INT NOT NULL,
    TripID INT NULL,
    Description TEXT NOT NULL,
    ReportedAt DATETIME DEFAULT CURRENT_TIMESTAMP,
    Status ENUM('Pending', 'Resolved') DEFAULT 'Pending',
    FOREIGN KEY (DriverID) REFERENCES Drivers(DriverID),
    FOREIGN KEY (TripID) REFERENCES Trips(TripID)
);
```



Tables_in_transportmanagement
bookings
driverissues
drivers
passengers
routes
trips
vehicles

6.IMPLEMENTATION:

The implementation of the Transport System Management project follows a modular, object-oriented approach to provide a structured and maintainable codebase. The application is divided into multiple functional packages, each responsible for handling specific aspects of the system.

- **Entity Module**: This package defines the core classes representing real-world components such as Driver, Passenger, Trip, Route, Booking, and Vehicle. These classes act as data carriers throughout the system.
- **DAO (Data Access Object) Module**: This layer manages all interactions with the underlying SQL database. Each DAO class (e.g., driver_dao.py, trip_dao.py) is responsible for performing CRUD operations for its corresponding entity, ensuring separation of concerns between data access and business logic.
- **Service Module**: This layer implements the core business logic of the application. It processes the data received from the DAO layer and controls the workflow for each operation. Services are defined for admin, driver, passenger, trip, booking, vehicle, and route management.
- **Admin and Driver Panels**: Specific modules like admin.py and driver_panel.py offer user-specific functionalities. Admins can manage all aspects of the system, whereas drivers can access their schedules and assigned trips.
- **Exception Handling**: Custom exceptions are defined in the exceptions module to manage and respond to application-specific errors gracefully, ensuring robustness and user-friendly feedback.
- **Util Module**: This package provides utility classes for database configuration and connection management. It includes scripts like db_connection.py and db_property_util.py to streamline database operations.
- **Testing Module**: To ensure reliability, the tests package includes unit tests for key service classes. These tests verify that individual components function correctly and help identify potential bugs during development.
- **Main Entry Point**: The application is launched via app.py, which provides a menu-driven interface for smooth interaction with various modules, guiding users through operations such as trip booking, vehicle assignment, and route planning.

This well-structured implementation ensures that each component is easily testable, reusable, and extendable for future enhancements.

Modules: admin/admin.py(Handled by Swetha)

Admin Panel Overview

The **Admin Panel** serves as the core interface for administrative operations in the Transport System Management application. It provides a menu-driven interface to perform **Create, Read, Update, Delete (CRUD)** operations and analytics on various system entities such as trips, bookings, passengers, vehicles, routes, and drivers.

The Admin class acts as the controller, orchestrating interaction between the user (admin) and the business logic defined in the AdminService.

Attributes (Methods) in the Admin Panel

Each method in the Admin class represents a distinct section of the admin panel, providing options for managing specific parts of the transport system:

- **manage_trips()**

Purpose: Manage all trip-related information.

Options Provided:

- Add a new trip
- Update existing trip details
- Delete a trip
- View all scheduled trips
- Return to the main admin menu

- **manage_bookings()**

Purpose: Supervise all booking operations.

Options Provided:

- View all bookings in the system
- Cancel a specific booking
- Delete a booking permanently
- Return to the main admin menu

- **manage_passengers()**

Purpose: Oversee passenger data and profiles.

Options Provided:

- View list of all registered passengers
- Add a new passenger
- Modify existing passenger information
- Delete a passenger record
- Return to the main admin menu

- **manage_vehicles()**

Purpose: Maintain vehicle records for transportation.

Options Provided:

- Add a new vehicle to the fleet
- Update details like type, capacity, or model
- Remove a vehicle from the system
- View all vehicles
- Return to the main admin menu

- **manage_routes()**

Purpose: Handle route information and updates.

Options Provided:

- Add a new route
- Update route details such as start point, end point, or stops
- Delete a route
- View all available routes
- Return to the main admin menu

- **view_reports()**

Purpose: Generate and display system-wide analytics.

Options Provided:

- View booking statistics (e.g., number of bookings per trip)
- Access revenue reports from bookings
- Track passenger activity and frequency
- Check trip utilization rates
- Return to the main admin menu

- **manage_drivers()**

Purpose: Administer driver profiles and allocations.

Options Provided:

- Add a new driver
- Update existing driver details
- Delete a driver from the system
- View all registered drivers
- Allocate a driver to a specific trip
- Deallocate a driver from a trip
- Return to the main admin menu

DAO File: (handled by Swetha)

booking_dao.py

Purpose:

The BookingDAO class in the booking_dao.py file is responsible for interacting with the database to perform all CRUD (Create, Read, Update, Delete) operations related to **Bookings** in the Transport System Management application.

It utilizes a connection from DBConnUtil and performs SQL operations on the Bookings table.

Class: BookingDAO

Attributes & Methods:

1. **__init__(self)**
    - Initializes the DAO by setting up the database connection and cursor.

2. **book_ticket(self, booking: Booking)**
    - Inserts a new booking record into the Bookings table using the provided Booking entity object.
    - Returns the BookingID of the newly inserted booking.
3. **get_all_bookings(self)**
    - Fetches all booking records from the database.
    - Converts each row into a Booking object and returns a list of these objects.
4. **get_booking_by_id(self, booking_id: int)**
    - Retrieves a single booking by its BookingID.
    - Returns a Booking object if found; otherwise, returns None.
5. **cancel_booking(self, booking_id: int)**
    - Updates the Status of the booking with the given BookingID to 'Cancelled'.
6. **delete_booking(self, booking_id: int)**
    - Deletes the booking record with the specified BookingID.
7. **get_latest_booking_by_passenger_id(self, passenger_id: int)**
    - Retrieves the most recent booking (by BookingDate) made by a specific passenger.
    - Returns a Booking object.
8. **get_past_bookings_by_passenger_id(self, passenger_id: int)**
    - Retrieves historical bookings (Confirmed, Completed, or Cancelled) for a specific passenger.
    - Also fetches related trip details via a join on the Trips table.
    - Returns raw tuples with extended details.

Attributes in Booking

- BookingID – Primary key.
- PassengerID – Foreign key referring to the passenger.
- RouteID – Foreign key referring to the route.
- VehicleID – Foreign key referring to the vehicle used.
- BookingDate – Date when the booking was made.
- BookingStatus – Status of the booking (e.g., Confirmed, Cancelled).

driver_dao.py

The driver_dao.py file belongs to the DAO (Data Access Object) layer in your Transport System Management project. It contains all database interaction logic related to **drivers** and their **assigned trips**.

The class DriverDAO encapsulates database operations such as inserting, retrieving, updating, and deleting driver records, along with assigning trips, updating statuses, and logging issues reported by drivers.

Brief Description of Each Attribute / Method in DriverDAO:

1. __init__()

- Initializes the DB connection and cursor object from DBConnUtil.

2. add_driver(driver: Driver)

- Inserts a new driver into the Drivers table.
- Uses driver object to extract Name, LicenseNumber, PhoneNumber, and Status.

3. get_all_drivers()

- Retrieves all driver records from the Drivers table.
- Returns a list of Driver objects.

4. get_driver_by_id(driver_id: int)

- Fetches a single driver based on their DriverID.
- Returns a Driver object or None.

5. update_driver(driver: Driver)

- Updates all attributes of an existing driver in the Drivers table.
- Uses the driver's ID to match the correct record.

6. delete_driver(driver_id: int)

- Deletes the driver record from the database using the given DriverID.

7. get_driver_trips(driver_id: int)

- Returns all trips assigned to a specific driver using the Trips table.

- Returns a list of Trip objects.

8. start_trip(driver_id: int, trip_id: int)

- Sets the Status of a trip to "In Progress" for the specified TripID and DriverID.

9. complete_trip(driver_id: int, trip_id: int)

- Updates the trip's status to "Completed" for the given trip and driver.

10. report_issue(driver_id: int, issue: str, trip_id=None)

- Logs a reported issue in the DriverIssues table.
- Associates the issue with a DriverID and optionally a TripID.

(This is duplicated as log_issue() as well)

11. log_issue(driver_id: int, issue: str, trip_id=None)

- **Same functionality** as report_issue() — likely a redundant method.

12. update_driver_status(driver_id: int, status: str)

- Updates a driver's current Status (e.g., Available, On Duty, Off Duty).

13. update_trip_status(trip_id: int, status: str)

- Updates the trip's status field directly in the Trips table.

14. get_driver_id_by_trip(trip_id: int)

- Retrieves the DriverID assigned to a given trip.

15. deallocate_driver(trip_id: int)

- Removes driver assignment from a trip by setting DriverID to NULL.

Attributes in Driver

- DriverID – Primary key.
- DriverName – Full name of the driver.
- LicenseNumber – Driver's license number.
- PhoneNumber – Contact number of the driver.

PassengerDAO

The PassengerDAO class is a Data Access Object (DAO) responsible for handling all database interactions related to the **Passenger** entity in a transport system management application. This DAO acts as a bridge between the application logic and the database layer, allowing the application to perform CRUD operations (Create, Read, Update, Delete) and other queries related to passengers without writing SQL in the business logic.

This class uses:

- DBConnUtil to establish the database connection.
- Passenger entity class for mapping database records to Python objects.

Attributes/Methods in PassengerDAO

Here's a **brief description of each method** (you called them "attributes") in the file:

1.__init__(self)

- **Purpose**: Initializes the DAO by establishing a database connection and creating a cursor object to execute queries.

2.add_passenger(self, passenger: Passenger)

- **Purpose**: Inserts a new passenger record into the Passengers table.
- **Inputs**: Passenger object containing the passenger's details.
- **Action**: Uses the passenger object's getter methods to extract data and store it in the database.

3.get_all_passengers(self)

- **Purpose**: Retrieves all passengers from the Passengers table.
- **Output**: Returns a list of Passenger objects created from the retrieved records.

4.get_passenger_by_id(self, passenger_id: int)

- **Purpose**: Retrieves a single passenger by their unique ID (PassengerID).
- **Input**: Integer ID of the passenger.
- **Output**: Returns a Passenger object if found, otherwise None.

5.update_passenger(self, passenger: Passenger)

- **Purpose**: Updates the information of a passenger in the database.
- **Inputs**: A Passenger object containing updated data.
- **Action**: Uses SQL UPDATE query to modify the existing record.

6.delete_passenger(self, passenger_id: int)

- **Purpose**: Deletes a passenger record from the database.
- **Input**: Integer ID of the passenger to be deleted.
- **Action**: Executes a SQL DELETE command.

Attributes in Passenger

- PassengerID – Primary key.
- Name – Full name of the passenger.
- Email – Email address of the passenger.
- PhoneNumber – Contact number of the passenger.
- Gender – Gender of the passenger.
- Age – Age of the passenger.

RouteDAO

The RouteDAO class is a **Data Access Object** for managing the Route entity in a Transport System Management application. It provides all the **CRUD operations** to interact with the Routes table in the database. This class encapsulates the database logic, helping to keep business logic separate from data access code.

It uses:

- DBConnUtil to get a connection to the database.
- Route entity class to map database records into Python objects and vice versa.

Attributes/Methods in RouteDAO

Below is a brief description of each **method** in the RouteDAO class:

1.__init__(self)

- **Purpose**: Initializes the DAO.
- **Function**:
  - Establishes a connection to the database using DBConnUtil.
  - Initializes a cursor object to execute SQL queries.

2.add_route(self, route: Route)

- **Purpose**: Adds a new route to the Routes table.
- **Inputs**: A Route object with details like start destination, end destination, and distance.
- **SQL Operation**: INSERT
- **Action**: Extracts values from the Route object using its getter methods and inserts them into the database.

3.get_all_routes(self)

- **Purpose**: Retrieves all route records from the database.
- **Output**: Returns a list of Route objects.
- **SQL Operation**: SELECT * FROM Routes
- **Action**: Fetches all records, maps each row to a Route object, and returns the list.

4.get_route_by_id(self, route_id: int)

- **Purpose**: Retrieves a single route record by its RouteID.
- **Input**: Integer route_id
- **Output**: A Route object if found; otherwise None.

- **SQL Operation**: SELECT ... WHERE
- **Action**: Maps the fetched row to a Route object.

5.update_route(self, route: Route)

- **Purpose**: Updates an existing route's details in the database.
- **Input**: A Route object with updated data.
- **SQL Operation**: UPDATE
- **Action**: Uses the RouteID to find and update the record.

6.delete_route(self, route_id: int)

- **Purpose**: Deletes a route from the database.
- **Input**: Integer route_id
- **SQL Operation**: DELETE
- **Action**: Removes the route record matching the given ID.

Attributes in Route

- RouteID – Primary key.
- StartDestination – Starting point of the route.
- EndDestination – Ending point of the route.
- Distance – Distance covered by the route (e.g., in kilometers).

RouteDAO

The RouteDAO class handles all **database operations related to routes** in your Transport System Management application. It's a part of the **Data Access Object (DAO) layer**, which cleanly separates business logic from database access.

This DAO uses:

- DBConnUtil.get_connection() – to establish a connection with the database.
- The Route entity class – to map and transfer route data between Python objects and database rows.

Methods in RouteDAO

1. **__init__(self)**
    - Initializes the DAO object.
    - Establishes a connection to the database using DBConnUtil.
    - Creates a cursor object for executing SQL queries.
2. **add_route(self, route: Route)**
    - Adds a new route record to the Routes table.
    - Accepts a Route object as input.
    - Uses INSERT SQL to insert values like start destination, end destination, and distance into the table.
3. **get_all_routes(self)**
    - Retrieves all route records from the database.
    - Uses SELECT * FROM Routes.
    - Maps each record to a Route object.
    - Returns a list of all Route objects.
4. **get_route_by_id(self, route_id: int)**
    - Retrieves a single route by its RouteID.
    - Uses SELECT with a WHERE clause to match the ID.
    - Returns the matched Route object, or None if not found.
5. **update_route(self, route: Route)**
    - Updates an existing route's details (like start/end destinations or distance).
    - Accepts a Route object with updated values.
    - Uses UPDATE SQL based on RouteID to apply changes.
6. **delete_route(self, route_id: int)**
    - Deletes a route record from the database.
    - Uses the provided RouteID to find and remove the record.
    - Executes a DELETE SQL statement.

Attributes in Routes Table

- RouteID – Primary Key; uniquely identifies each route.
- StartDestination – The starting location of the route.
- EndDestination – The ending location of the route.
- Distance – The total distance of the route.

TripDAO

This TripDAO class is a **Data Access Object (DAO)** in Python for managing **Trip-related operations** with a database. It encapsulates all the logic for:

- Inserting,
- Updating,
- Fetching,
- Deleting trip records,
- Allocating/deallocating drivers,
- Logging issues related to drivers,
- Managing trip statuses.

It uses a database connection from DBConnUtil, and maps trip records to/from the Trip entity class.

Method-wise Explanation

1. __init__(self)

- Initializes the DAO with a database connection and cursor.

2. add_trip(self, trip: Trip)

- Inserts a new trip into the database using the values from a Trip object.

3. get_available_trips(self)

- Returns all **Scheduled** trips of **Passenger** type, sorted by DepartureDate.

4. get_all_trips(self)

- Returns a list of all trips from the Trips table as Trip objects.

5. get_trip_by_id(self, trip_id: int)

- Returns a specific trip from the Trips table based on TripID.

6. update_trip(self, trip: Trip)

- Updates an existing trip's details in the database using data from a Trip object.

7. delete_trip(self, trip_id: int)

- Deletes a trip from the database based on the given TripID.

8. allocate_driver(self, trip_id: int, driver_id: int)

- Assigns a driver to a trip by setting DriverID for the given TripID.

9. deallocate_driver(self, trip_id: int)

- Removes (nullifies) the driver assignment from a trip.

10. is_driver_available(self, driver_id: int) -> bool

- Checks if a driver is **not assigned** to any **Scheduled** or **In Progress** trips.
- Returns True if the driver is free.

11. get_driver_trips(self, driver_id: int) -> list

- Returns all trips assigned to a particular driver, sorted by departure date.

12. log_issue(self, driver_id: int, description: str, trip_id: int = None) -> bool

- Logs an issue raised by a driver into the DriverIssues table with the current timestamp.

13. get_trips_by_driver(self, driver_id: int)

- Retrieves all trips for a given driver ID and returns them as Trip objects.

14. update_trip_status(self, trip_id: int, status: str) -> bool

- Updates the **status** of a trip (e.g., to 'Completed', 'Cancelled').
- Returns True if update was successful.

VehicleDAO File:

The VehicleDAO (Data Access Object) file is responsible for performing all **database-related operations for vehicles** in the Transport System Management

application. It acts as a bridge between the Vehicle entity (business model) and the Vehicles table in the SQL database.

It contains methods to:

- Insert new vehicles
- Retrieve vehicles (all or by ID)
- Update vehicle details
- Delete a vehicle

Method Descriptions in VehicleDAO

1. **__init__(self)**
    - Initializes a connection to the database and sets up a cursor to execute SQL queries.
2. **add_vehicle(self, vehicle: Vehicle)**
    - Adds a new vehicle to the Vehicles table.
    - Inserts model, capacity, type, and status using data from the Vehicle object.
3. **get_all_vehicles(self)**
    - Retrieves **all vehicles** from the database.
    - Converts each record from the database into a Vehicle object and returns a list of them.
4. **get_vehicle_by_id(self, vehicle_id: int)**
    - Fetches a **specific vehicle** by its ID.
    - Returns a Vehicle object if found; otherwise, returns None.
5. **update_vehicle(self, vehicle: Vehicle)**
    - Updates the details of a vehicle in the database based on its ID.
    - Updates the model, capacity, type, and status with values from the Vehicle object.
6. **delete_vehicle(self, vehicle_id: int)**
    - Deletes a vehicle record from the database using its ID.

DriverPanel class

This class handles the **driver's user interface panel** in your console-based transport system. It provides a menu where drivers can manage their assigned trips, update statuses, report issues, and view their own details.

It connects with:

- DriverService (for driver and issue logic)
- TripService (for trip status updates)
- Custom exceptions: TripNotFoundException, InvalidDriverDataException

Method Descriptions

1.__init__(self, driver_id)

- **Purpose**: Constructor for DriverPanel.
- **Initializes**:
    - driver_id – the ID of the current driver.
    - Instances of DriverService and TripService for service logic.

2.show_menu(self)

- **Purpose**: Displays an interactive menu to the driver.
- **Options**:
    - View trips
    - Start/complete trips
    - Report issues
    - View personal info
    - Logout
- **Control Flow**: Loops until the driver selects logout.

3.view_my_trips(self)

- **Purpose**: Fetches and displays all trips assigned to the current driver.
- **Output**: Displays scheduled trips with In Progress or Upcoming status.

4.start_trip(self)

- **Purpose**: Allows a driver to start a trip.
- **Steps**:
    - Input: Trip ID
    - Calls driver_service.start_trip(...)
    - Updates trip status to "In Progress" using TripService.

5.complete_trip(self)

- **Purpose**: Allows a driver to mark a trip as completed.
- **Steps**:
  - Input: Trip ID
  - Calls driver_service.complete_trip(...)
  - Updates trip status to "Completed" using TripService.

6.report_issue(self)

- **Purpose**: Report an issue related to a trip or general driver problem.
- **Steps**:
  - Optional Trip ID input
  - Text input for issue
  - Calls driver_service.report_issue(...)

7.view_my_details(self)

- **Purpose**: Displays the current driver's personal details.
- **Data Displayed**:
  - Name
  - License number
  - Current status

Entity Files(handled by me):

- **Booking Class Overview:**

The Booking class represents a **trip booking record** made by a passenger. It includes details like which trip is booked, who booked it, when it was booked, and the current status of the booking.

Attributes:

- __booking_id: Unique ID of the booking.
- __trip_id: The trip associated with the booking.
- __passenger_id: The ID of the passenger who made the booking.
- __booking_date: The date the booking was made.
- __status: The status of the booking (e.g., "Confirmed", "Cancelled").

Methods:

- get_booking_id() / set_booking_id(id): Accessor and mutator for booking ID.
- get_trip_id() / set_trip_id(id): Accessor and mutator for trip ID.
- get_passenger_id() / set_passenger_id(id): Accessor and mutator for passenger ID.
- get_booking_date() / set_booking_date(date): Accessor and mutator for booking date.
- get_status() / set_status(status): Accessor and mutator for booking status.
- __str__(): Returns a nicely formatted string with all booking details.

Driver Class Overview:

The Driver class represents a **driver in the transport system**, storing their personal and operational details including their license, contact information, and current status.

Attributes:

- __driver_id: Unique identifier for the driver.
- __name: Full name of the driver.
- __license_number: The driver's license number.
- __phone_number: Contact number of the driver.
- __status: Current status of the driver (e.g., "Available", "On Trip", "Inactive").

Methods:

- get_driver_id() / set_driver_id(id): Accessor and mutator for the driver's ID.
- get_name() / set_name(name): Accessor and mutator for the driver's name.
- get_license_number() / set_license_number(number): Accessor and mutator for the license number.
- get_phone_number() / set_phone_number(number): Accessor and mutator for phone number.
- get_status() / set_status(status): Accessor and mutator for the driver's current status.

◆ Passenger Class Overview:

The Passenger class represents a **person who books and travels** using the transport system. It stores essential passenger details and includes validation for age, gender, and phone number to ensure data integrity.

Attributes:

- __passenger_id: Unique ID to identify the passenger.
- __first_name: Passenger's first name.
- __age: Passenger's age (validated to be between 0 and 120).
- __phone_number: Passenger's contact number (validated to be numeric and 10–15 digits).
- __gender: Gender of the passenger ('Male', 'Female', 'Other' only).
- __email: Email address of the passenger.

Methods:

- Standard **getters and setters** for all attributes.
- Setters include validation for:
  - age: Must be an integer between 0 and 120.
  - gender: Must be one of the allowed values.
  - phone_number: Must be numeric and within valid length.
- __str__(): Returns a formatted string summarizing the passenger's details.

◆ Route Class Overview:

The Route class represents a **travel path** between two destinations in the transport system. It defines the geographical and distance-related information needed to schedule trips.

Attributes:

- __route_id: Unique identifier for the route.
- __start_destination: Starting point of the route.
- __end_destination: Ending point of the route.

- __distance: Total distance covered by the route (e.g., in kilometers).

Methods:

- **Getters and setters** for all fields to retrieve and modify route information as needed.

◆ Trip Class Overview:

The Trip class represents a **scheduled journey** in the transport management system. It encapsulates key details regarding the travel plan, including vehicle, route, schedule, type, and driver.

Attributes:

- __trip_id: Unique identifier for the trip.
- __vehicle_id: ID of the vehicle assigned to the trip.
- __route_id: Route taken by the trip.
- __departure_date: Starting date and time of the trip.
- __arrival_date: Expected end date and time of the trip.
- __status: Current status (e.g., Scheduled, Completed, Cancelled).
- __trip_type: Type of trip (e.g., One-way, Round-trip).
- __max_passengers: Maximum passengers allowed on the trip.
- __driver_id: ID of the assigned driver.

Methods:

- **Getters and setters** for all attributes, enabling full control over trip details.
- __str__() method provides a readable string representation for logging/debugging.

Vehicle Class Overview:

The Vehicle class represents a **transport vehicle** used in trips. It encapsulates all relevant information about the vehicle's identity, specifications, and current state.

Attributes:

- __vehicle_id: Unique identifier for the vehicle.
- __model: Model name or number of the vehicle.
- __capacity: Maximum number of passengers it can accommodate.
- __vehicle_type: Type of vehicle (e.g., Bus, Van, Mini, SUV).
- __status: Current status (e.g., Available, Under Maintenance, In Use).

Methods:

- **Getters and setters** for all attributes to maintain encapsulation and control.
- No __str__() method defined yet — can be added for debugging/logging purposes.

Service File(Handled by Swetha)

AdminService Class - Overview

The AdminService class serves as a centralized interface for administrative tasks, acting as a facade to multiple service layers like TripService, BookingService, PassengerService, VehicleService, RouteService, and DriverService.

Key Responsibilities by Module

1. Trip Management

- add_trip(): Adds a new trip by taking inputs such as vehicle ID, route ID, dates, type, etc.
- update_trip(): Updates details like departure, arrival, status, or max passengers for an existing trip.
- delete_trip(): Deletes a trip based on trip ID.
- view_all_trips(): Displays a list of all available trips.

2. Booking Management

- view_all_bookings(): Lists all bookings stored in the system.
- cancel_booking(): Cancels a booking (admin-initiated), verifying the booking exists and isn't already cancelled.

- delete_booking(): Completely deletes a booking record by ID.

3. Passenger Management

- view_all_passengers(): Displays details of all registered passengers.
- add_passenger(): Initiates the process of collecting and adding a new passenger's details (incomplete in the snippet).

Dependency Management

- Uses **service composition** to access and delegate tasks to:
  - TripService
  - BookingService
  - PassengerService
  - VehicleService
  - RouteService
  - DriverService

BookingService Class – Overview

This service class acts as an intermediary between the controller and DAO layers, handling business logic for booking operations.

Methods & Responsibilities:

- **__init__()**
  - Initializes a BookingDAO instance to interact with the database.
- **book_ticket(booking: Booking)**
  - Validates booking details like trip ID, passenger ID, booking date, and status.
  - Delegates ticket booking to the DAO.
- **get_all_bookings()**
  - Retrieves all booking records from the database.
- **get_booking_by_id(booking_id, passenger_id)**
  - Fetches a booking by ID.
  - Validates passenger ownership.
  - Throws exception if not found or unauthorized access.
- **cancel_booking(booking_id, passenger_id)**
  - Validates booking ownership.
  - Cancels the booking if not already cancelled.

- **delete_booking(booking_id)**
  - Deletes a booking after verifying its existence.
- **get_latest_booking_by_passenger_id(passenger_id)**
  - Fetches the most recent booking for a specific passenger.
- **get_past_bookings_by_passenger_id(passenger_id)**
  - Retrieves all previous bookings made by the passenger.

DriverService Class – Overview

This service layer handles business logic and validation for managing drivers, ensuring only valid data is passed to the DAO for database operations.

Key Responsibilities & Methods:

- **__init__()**
  - Instantiates the DAO class to enable DB operations.

Core CRUD Operations:

- **add_driver(driver)**
  - Adds a new driver after validating name, license number, and status.
- **get_driver_by_id(driver_id)**
  - Retrieves a driver; raises exception if not found.
- **update_driver(driver)**
  - Updates driver details using DAO.
- **delete_driver(driver_id)**
  - Deletes the specified driver from the system.
- **get_all_drivers()**
  - Fetches all driver records.

Driver Functionalities:

- **get_driver_trips(driver_id)**
  - Returns all trips assigned to the given driver.
- **start_trip(driver_id, trip_id)**
  - Updates driver status to Assigned if currently Available.
- **complete_trip(driver_id, trip_id)**
  - Marks driver as Available if they are Assigned.

- **report_issue(driver_id, description, trip_id)**
    - Logs an issue reported by the driver, optionally linked to a trip.

Helper Methods:

- **_validate_driver(driver)**
    - Ensures driver data like name, license number, and status are valid.
- **_validate_driver_status(driver_id, allowed_statuses)**
    - Verifies if the driver is in an acceptable state for an action (like starting or ending a trip).

PassengerService Class – Overview

This service class manages passenger-related operations, ensuring input validation and handling exceptions before interacting with the DAO for database actions.

Key Responsibilities & Methods:

- **__init__()**
    - Initializes the PassengerDAO for database access.

CRUD Operations:

- **add_passenger(passenger)**
    - Validates and adds a new passenger.
    - Checks for valid first name, email, phone number, gender (Male, Female, Other), and non-negative age.
- **get_passenger_by_id(passenger_id)**
    - Fetches a passenger by ID.
    - Raises an exception if the passenger is not found.
- **get_all_passengers()**
    - Retrieves a list of all passengers from the database.
- **update_passenger(passenger)**
    - Validates and updates an existing passenger.
    - Ensures required fields and valid gender/age are provided.
- **delete_passenger(passenger_id)**
    - Deletes a passenger after verifying existence.
    - Raises an exception if the passenger ID is not found.

RouteService Class – Overview

This class handles operations related to **routes** in the transport system. It validates input data and coordinates with the RouteDAO for database interactions while handling possible exceptions.

Key Responsibilities & Methods:

- **__init__()**
  - Initializes the service with a RouteDAO instance for database communication.

CRUD Operations:

- **add_route(route)**
  - Validates and adds a new route.
  - Ensures both start and end destinations are provided and distance is greater than 0.
- **get_all_routes()**
  - Retrieves a list of all available routes from the database.
- **get_route_by_id(route_id)**
  - Fetches a route based on its ID.
  - Raises RouteNotFoundException if no matching route is found.
- **update_route(route)**
  - Validates and updates route details.
  - Ensures required fields and valid distance are present.
- **delete_route(route_id)**
  - Deletes a route after confirming its existence.
  - Raises an exception if the route ID is not found.

TripService Class – Overview

The TripService class manages trip-related operations in the transport system. It validates trip data, interacts with the database via TripDAO, and coordinates driver allocation with DriverService and DriverDAO.

Key Responsibilities & Methods:

- **__init__()**
  - Initializes instances of TripDAO, DriverService, and DriverDAO.

Trip Management:

- **add_trip(trip)**
  - Validates trip data (vehicle ID, route ID, dates, type, status, passengers).
  - Adds a new trip to the database.
- **get_all_trips()**
  - Returns a list of all trips.
- **get_trip_by_id(trip_id)**
  - Retrieves trip details by ID.
  - Raises TripNotFoundException if the trip does not exist.
- **update_trip(trip)**
  - Validates and updates trip data.
  - Automatically updates driver status based on trip status.
- **delete_trip(trip_id)**
  - Deletes a trip after confirming its existence.

Driver Allocation:

- **allocate_driver(trip_id, driver_id)**
  - Assigns a driver to a trip if the driver is available.
  - Raises relevant exceptions if trip or driver doesn't exist or if driver is already assigned.
- **deallocate_driver(trip_id)**
  - Removes driver assignment from a trip.
  - Updates the driver's availability status to "Available".

Trip Status Management:

- **update_trip_status(trip_id, status)**
  - Updates only the status field of a trip record.
- **is_driver_available(driver_id)**
  - Returns True if the specified driver exists and is currently "Available".

VehicleService Class

This service class manages business logic for vehicle-related operations. It interacts with the VehicleDAO to perform database actions and ensures input validation.

Key Responsibilities:

- **Add a vehicle:** Validates model, type, status, and capacity before adding it to the system.
- **Get all vehicles:** Retrieves the list of all vehicles.
- **Get vehicle by ID:** Returns a specific vehicle or raises an exception if not found.
- **Update vehicle:** Validates and updates vehicle details.
- **Delete vehicle:** Deletes a vehicle after ensuring it exists.

Validation Rules:

- vehicle_type: Must be one of ['Truck', 'Van', 'Bus']
- status: Must be one of ['Available', 'On Trip', 'Maintenance']
- capacity: Must be a positive number

7.Exceptions(handled by Maheshwari)

Custom Exception Classes

These exceptions are user-defined and used throughout the transport system to handle specific error scenarios with meaningful messages.

1. TripNotFoundException

Description: This exception is raised when a trip is not found in the system, typically when the system fails to retrieve a trip by its ID or other criteria.

Use Case: Thrown when the user requests a trip that doesn't exist in the database.

2. InvalidTripDataException

Description: Raised when the provided trip data is invalid. This can include incorrect or missing information such as trip date, route, or trip ID.

Use Case: Thrown when trying to create or update a trip with invalid data (e.g., a missing or incorrect trip date).

3. InvalidRouteDataException

Description: This exception is thrown when invalid route data is encountered. It could be invalid route details such as start and end locations or any other route-related information.

Use Case: Thrown when route data is improperly provided for trip creation or update.

4. DriverNotAvailableException

Description: Raised when a driver is not available for a trip or booking.

Use Case: Thrown when attempting to assign a trip to a driver who is unavailable due to other commitments or system constraints.

5. DriverNotFoundException

Description: Raised when the system is unable to find the specified driver. This can happen when the driver ID does not exist in the database.

Use Case: Thrown when trying to retrieve or assign a driver who is not present in the system.

6. PassengerNotFoundException

Description: Thrown when no passenger is found in the system with the given passenger ID.

Use Case: Thrown when attempting to retrieve or modify a passenger that doesn't exist in the database.

7. VehicleNotFoundException

Description: Raised when no vehicle matching the given criteria (e.g., vehicle ID) is found in the system.

Use Case: Thrown when trying to assign a vehicle that doesn't exist or has been removed from the system.

8. RouteNotFoundException

Description: Raised when a route could not be found in the system.

Use Case: Thrown when attempting to retrieve or update a route that does not exist in the database.

9. InvalidPassengerDataException

Description: Raised when invalid passenger data is encountered. This can include missing or incorrect passenger details.

Use Case: Thrown when trying to create or update a passenger with missing or invalid data (e.g., invalid passenger ID or missing name).

10. InvalidDriverDataException

Description: Raised when invalid driver data is encountered. This could be issues such as missing driver ID, incorrect information, or invalid credentials.

Use Case: Thrown when attempting to create or update a driver with invalid or incomplete data.

11. InvalidVehicleDataException

Description: Raised when invalid vehicle data is encountered, such as invalid vehicle ID, missing attributes, or incorrect vehicle details.

Use Case: Thrown when trying to create or update a vehicle with invalid or incomplete data.

12. BookingNotFoundException

Description: Raised when a booking cannot be found. This may happen when a user tries to view, cancel, or retrieve a booking that doesn't exist in the system.

Use Case: Thrown when attempting to retrieve a booking by ID or passenger ID, but the booking does not exist.

13. InvalidBookingDataException

Description: Raised when booking data is invalid. This includes missing, incorrect, or incomplete information in the booking (e.g., trip ID, passenger ID, booking status).

Use Case: Thrown when trying to book a ticket with invalid details such as missing trip ID or invalid booking date.

App.py

The main() function acts as the **entry point** for the entire Transport System Management System. It presents the user with a **role-based selection menu** allowing them to log in as a **Customer**, **Admin**, or **Driver**. Based on their selection, the program navigates to the respective menu for managing or interacting with trips, bookings, vehicles, and users.

“”

```
def main():
    while True:
        print("\n===== Transport Management System =====")
        print("1. Customer Login")
        print("2. Admin Login")
        print("3. Driver Login")
        print("4. Exit")
        choice = input("Enter your choice: ")

        if choice == "1":
            customer_menu()
```

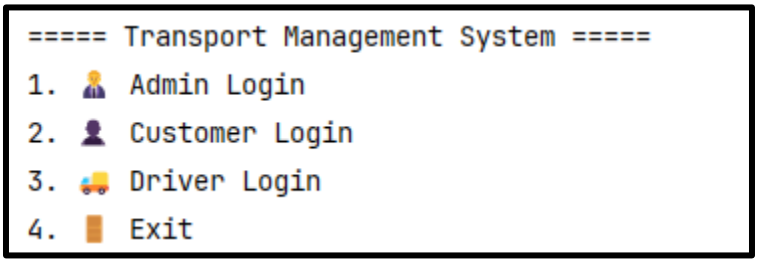
```
    elif choice == "2":
        admin_menu()
    elif choice == "3":
        driver_menu()
    elif choice == "4":
        print("🚪 Exiting system. Goodbye!")
        break
    else:
        print("❌ Invalid choice. Please try again.")
```

“”

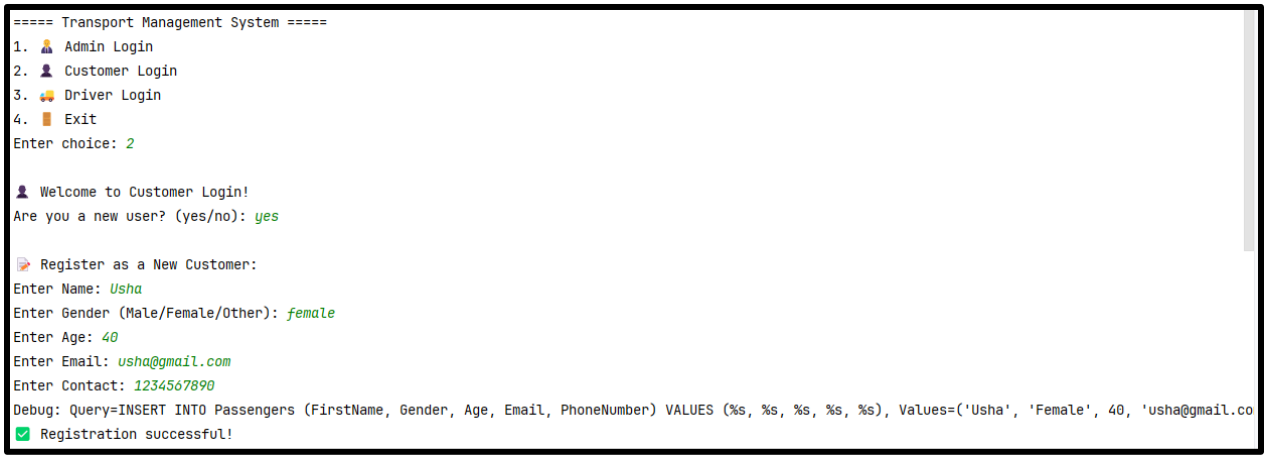
What Happens Here?

- Presents the user with role options.
- Based on the selection, it calls:
  - customer_menu() – for customers/passengers.
  - admin_menu() – for administrators.

  driver_menu() – for drivers.



Customer Registration:



Admin login page:

```
def admin_menu():
    while True:
        print("\n===== Admin Menu =====")
        print("1. Manage Trips")
        print("2. Manage Bookings")
        print("3. Manage Passengers")
        print("4. Manage Vehicles")
        print("5. Manage Routes")
        print("6. Manage Drivers")
        print("7. Reports and Analytics")
        print("8. Logout")
        choice = input("Enter choice: ")


        if choice == "1":
            admin = Admin()  # Create an instance of Admin
            admin.manage_trips()
        elif choice == "2":
            admin = Admin()  # Create an instance of Admin
            admin.manage_bookings()
        elif choice == "3":
```

```
        admin = Admin()  # Create an instance of Admin
        admin.manage_passengers()
    elif choice == "4":
        admin = Admin()  # Create an instance of Admin
        admin.manage_vehicles()
    elif choice == "5":
        admin = Admin()  # Create an instance of Admin
        admin.manage_routes()
    elif choice == "6":
        admin = Admin()  # Create an instance of Admin
        admin.manage_drivers()
    elif choice == "7":
        admin = Admin()  # Create an instance of Admin
        admin.view_reports()
    elif choice == "8":
        print("👋 Logging out. Bye!")
        break
    else:
        print("❌ Invalid choice. Try again.")
```

```
===== Transport Management System =====
1.  Admin Login
2.  Customer Login
3.  Driver Login
4.  Exit
Enter choice: 1

===== Admin Menu =====
1. Manage Trips
2. Manage Bookings
3. Manage Passengers
4. Manage Vehicles
5. Manage Routes
6. Manage Drivers
7. Reports and Analytics
8. Logout
```

Admin Menu page:

```
===== Admin Menu =====
1. Manage Trips
2. Manage Bookings
3. Manage Passengers
4. Manage Vehicles
5. Manage Routes
6. Manage Drivers
7. Reports and Analytics
8. Logout
Enter choice: 1
```

Reports And Analytics:

```
===== Reports and Analytics =====
1. Booking Statistics
2. Revenue Reports
3. Passenger Activity
4. Trip Utilization
5. Back to Admin Menu
Enter choice: 1

📊 Booking Statistics:
Total Bookings: 11
Cancelled Bookings: 2 (18.2%)
Completed Trips: 2 (18.2%)
```

Revenue Reports:

```
===== Reports and Analytics =====
1. Booking Statistics
2. Revenue Reports
3. Passenger Activity
4. Trip Utilization
5. Back to Admin Menu
Enter choice: 2

💰 Revenue Reports:
Total Confirmed Bookings: 7
Estimated Revenue: $350
```

Passenger Activity:

```
===== Reports and Analytics =====
1. Booking Statistics
2. Revenue Reports
3. Passenger Activity
4. Trip Utilization
5. Back to Admin Menu
Enter choice: 3

👥 Passenger Activity:
Top 5 Most Active Passengers:
Emma Thomas (ID: 10): 2 bookings
suresh (ID: 1): 1 bookings
Jane Smith (ID: 2): 1 bookings
Alex Johnson (ID: 3): 1 bookings
Michael Brown (ID: 4): 1 bookings
```

Trip Utlization:

```
===== Reports and Analytics =====
1. Booking Statistics
2. Revenue Reports
3. Passenger Activity
4. Trip Utilization
5. Back to Admin Menu
Enter choice: 4

🚌 Trip Utilization:
Trip ID | Utilization | Status
--------------------------------
1       |   20.0%     | In Progress
3       |    0.0%     | Completed
5       |    8.3%     | Scheduled
7       |    1.8%     | Scheduled
8       |    0.0%     | Completed
10      |    4.0%     | Scheduled
11      |    0.0%     | Scheduled
12      |    0.0%     | Scheduled
13      |    0.0%     | Scheduled
```

Driver Menu:

```
def driver_menu():
    try:
        print("\n🚚 Welcome to Driver Login")
        driver_id = int(input("Enter your Driver ID: "))
        driver_panel = DriverPanel(driver_id)
        driver_panel.show_menu()
    except Exception as e:
        print(f"❌ Error in driver menu: {e}")
```

```
===== Transport Management System =====
1. 👨‍💼 Admin Login
2. 👤 Customer Login
3. 🚚 Driver Login
4. 🚪 Exit
Enter choice: 3

🚚 Welcome to Driver Login
Enter your Driver ID: 1
```

Driver Panel Page:

```
===== Driver Panel (ID: 1) =====
1. View My Scheduled Trips
2. Start Trip
3. Complete Trip
4. Report Issue
5. View My Details
6. Logout
Enter choice: 2
Enter Trip ID to start: 1
✅ Database connection successful!
✅ Trip started successfully
```

Details:

```
===== Driver Panel (ID: 1) =====
1. View My Scheduled Trips
2. Start Trip
3. Complete Trip
4. Report Issue
5. View My Details
6. Logout
Enter choice: 5

👤 Your Details:
Name: John Smith
License: DL123456789
Status: Assigned
```

Exit:

```
===== Transport Management System =====
1. 👨‍💼 Admin Login
2. 👤 Customer Login
3. 🚚 Driver Login
4. 🚪 Exit
Enter choice: 4
👋 Exiting system. Bye!
```

8.Unit Testing:(Handled by Maheshwari)

Purpose:

Unit testing ensures that individual components (classes, methods, services) of the Transport Management System (TMS) behave as expected under various conditions. This increases code reliability, reduces bugs, and confirms smooth database interactions.

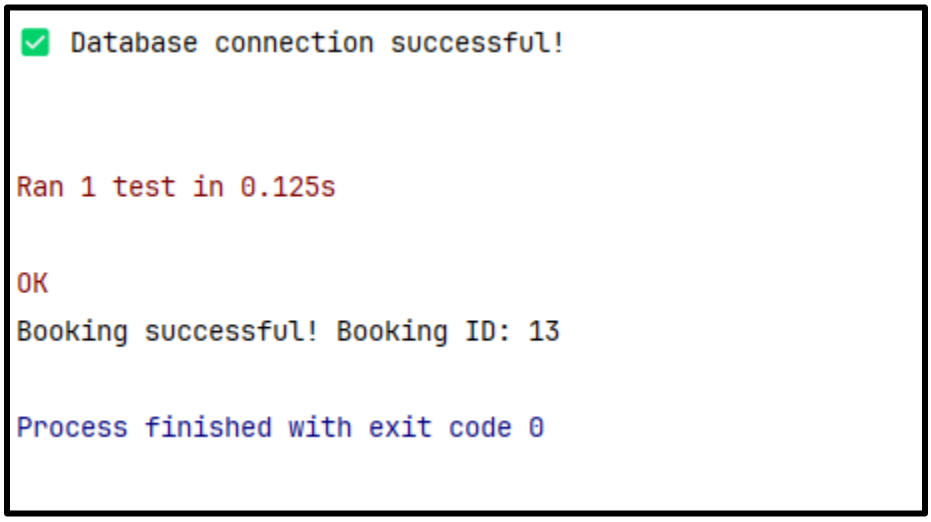
1. Write test case to test booking successfully or not.

import unittest

from entity.booking import Booking

from service.booking_service import BookingService

```
class TestBookingFunctionality(unittest.TestCase):
    def setUp(self):
        self.booking_service = BookingService()


    def test_booking_success(self):
        booking = Booking(None, trip_id=1, passenger_id=10, booking_date="2025-04-14",
status="Confirmed")
        booking_id = self.booking_service.book_ticket(booking)
        self.assertIsNotNone(booking_id)
        print(f"Booking successful! Booking ID: {booking_id}")


if __name__ == '__main__':
    unittest.main()
```



2. Write test case to test driver mapped to vehicle successfully or not.

```
import unittest
from entity.driver import Driver
from entity.vehicle import Vehicle
from service.driver_vehicle_mapper import DriverVehicleMapper


class TestDriverVehicleMapping(unittest.TestCase):
```

```
    def test_driver_mapped_to_vehicle(self):
        driver = Driver(driver_id=1, name="Arun", license_number="DL12345", phone_number="9876543210", status="Active")
        vehicle = Vehicle(vehicle_id=101, model="Toyota", capacity=4, vehicle_type="SUV", status="Available")

        driver_vehicle_mapper = DriverVehicleMapper()
        driver_vehicle_mapper.assign_driver_to_vehicle(driver, vehicle)

        assigned_driver_id = driver_vehicle_mapper.get_driver_for_vehicle(vehicle)

        self.assertEqual(assigned_driver_id, 1)
        print(f"Driver {driver.name} with ID {driver.driver_id} is successfully mapped to vehicle {vehicle.model} with vehicle ID {vehicle.vehicle_id}.")

if __name__ == "__main__":
    unittest.main()
```

```
Driver Arun is assigned to Vehicle Toyota
Driver Arun with ID 1 is successfully mapped to vehicle Toyota with vehicle ID 101.


Ran 1 test in 0.001s

OK
```

3. write test case to test exception is thrown correctly or not when vehicle or booking not found in database.

```
import unittest
from service.booking_service import BookingService
from exceptions.exceptions import BookingNotFoundException, InvalidBookingDataException
from entity.booking import Booking
```

```
class TestBookingService(unittest.TestCase):
    def setUp(self):
        self.booking_service = BookingService()

    def test_booking_not_found_exception(self):
        print("Testing for BookingNotFoundException with invalid booking ID")
        try:
            self.booking_service.get_booking_by_id(-1, 100)  # Pass incorrect booking_id and passenger_id
        except BookingNotFoundException as e:
            print(f"Expected exception caught: {str(e)}")
            self.assertTrue(isinstance(e, BookingNotFoundException))  # Verifies that the exception is of the correct type

    def test_invalid_booking_data_exception(self):
        print("Testing for InvalidBookingDataException with mismatched booking ID and passenger ID")
        try:
            self.booking_service.get_booking_by_id(1, 999)  # Pass correct booking_id but incorrect passenger_id
        except InvalidBookingDataException as e:
            print(f"Expected exception caught: {str(e)}")
            self.assertTrue(isinstance(e, InvalidBookingDataException))  # Verifies that the exception is of the correct type

if __name__ == '__main__':
    unittest.main()
```

```
✅ Database connection successful!
Testing for BookingNotFoundException with invalid booking ID
Expected exception caught: Booking with ID -1 not found.
✅ Database connection successful!
Testing for InvalidBookingDataException with mismatched booking ID and passenger ID


Ran 2 tests in 0.407s

OK
```

4. Write test case to test vehicle, driver successfully or not.

```
import unittest
from dao.vehicle_dao import VehicleDAO
from entity.vehicle import Vehicle

class TestVehicleDriver(unittest.TestCase):

    @classmethod
    def setUpClass(cls):
        cls.vehicle_dao = VehicleDAO()
        print("Database connection established for VehicleDAO.")

    def test_add_vehicle_success(self):
        print("\nStarting test: test_add_vehicle_success")
        vehicle = Vehicle(None, 'Ford Transit', 15.00, 'Van', 'Available')
        try:
            self.vehicle_dao.add_vehicle(vehicle)
            print("Vehicle added to database: Ford Transit")

            # Check if the vehicle was added
```

```
        self.vehicle_dao.cursor.execute("SELECT * FROM Vehicles WHERE Model = 'Ford Transit'")
        result = self.vehicle_dao.cursor.fetchone()

        if result:
            print(f"Record found in DB: ID={result[0]}, Model={result[1]}, Capacity={result[2]}, Type={result[3]}, Status={result[4]}")
        else:
            print("No record found for 'Ford Transit'.")

        self.assertIsNotNone(result)
        self.assertEqual(result[1], 'Ford Transit')
        self.assertEqual(result[2], 15.00)
        self.assertEqual(result[3], 'Van')
        self.assertEqual(result[4], 'Available')

        print("test_add_vehicle_success passed.")

    except Exception as e:
        print(f"Exception during test_add_vehicle_success: {e}")
        self.fail(f"Test failed due to exception: {e}")

@classmethod
def tearDownClass(cls):
    cls.vehicle_dao.conn.close()
    print("Database connection closed for VehicleDAO.")

if __name__ == "__main__":
    unittest.main()
```

```
✅ Database connection successful!
Database connection established for VehicleDAO.
Database connection closed for VehicleDAO.

Starting test: test_add_vehicle_success
Vehicle added to database: Ford Transit
Record found in DB: ID=1, Model=Ford Transit, Capacity=15.00, Type=Van, Status=Available


Ran 1 test in 0.148s

OK
```

9.CONCLUSION

The unit testing of the Transport Management System successfully verified the core functionalities of the system's modules, including booking, driver-vehicle mapping, and vehicle data management. Through these tests, both valid workflows and error-handling mechanisms were confirmed to perform as expected.