BECS 12243

QR Bus Ticket Booking System

Final Report

EC/2022/008 - I.D.N.B Amaranath

EC/2022/012 - T.M.D.P.M De Alwis

EC/2022/015 - K.A.S.M Jayawardhana

EC/2022/037 - K.G.H.D Bandara

EC/2022/063 - M.K.S Vinhara

1. Introduction to QR Bus Ticket Booking System

The QR Bus Ticket Booking System is designed to improve the efficiency of public transportation in Sri Lanka by modernizing the ticketing process. Traditionally, bus conductors issue paper tickets, which can lead to delays, inefficiencies, and challenges with cash handling during peak hours. To address these issues, our system allows passengers to pre-book tickets online and use a unique QR code as a digital ticket for boarding.

The system provides the following benefits:

- **Streamlined Boarding Process**: Quick validation through QR code scanning, avoiding manual checks and long queues.
- **Enhanced Convenience**: Instant digital tickets available on mobile apps or websites, eliminating the need for physical tickets.
- **Increased Accuracy and Accountability**: Digital records of user information and ticket details improve accuracy and reduce human error.
- **Eco-Friendly**: Reduces paper usage by eliminating paper tickets.

Key Functionalities:

- Pre-booking of seats via a mobile app or website.
- QR code generation for each ticket.
- Ticket validation using QR code scanners at bus stations.

2. Classes, Attributes, and Methods

The system is designed using Object-Oriented Programming (OOP) principles. Below are descriptions of the key classes, their attributes, and methods, demonstrating the OOP features like **Encapsulation**, **Inheritance**, **Polymorphism**, and **Abstraction**.

UserManager Class

- **Attributes**:

  - ✓ **name** - Stores the user's name.
  - ✓ **email** - Stores the user's email.
  - ✓ **password** - Stores the user's password.
  - ✓ **mobileNo** - Stores the user's mobile number.
  - ✓ **returnedName** - Holds the user's name upon login.
  - ✓ **storage** - Uses StorageManager class to handle data storage.

- **Methods**:

  - ✓ **UserManager (StorageManager storage):** This constructor initializes the UserManager with a StorageManager instance, allowing it to use storage functionality via dependency injection.
  - ✓ **userRegister()**: Registers a new user by collecting their details.
  - ✓ **userLogin()**: Authenticates users based on email and password.
  - ✓ **isValidMobileNumber()**: Validates the entered mobile number.
  - ✓ **getUserName()**: Returns the name of the logged-in user.
  - ✓ **getEmail()**: Returns the email of the user.

- **OOP Concepts**:

  Inheritance:
  The UserManager class extends the abstract class **Manager**, allowing it to inherit and customize behavior.

```
public class UserManager extends Manager implements UserAction
```

Encapsulation:
The UserManager class encapsulates attributes such as name, email, password, returnedName and mobileNo. These attributes are kept private, and public methods like userRegister and userLogin provide controlled access to them.

The storage field is marked as **private final**, meaning it can only be set once during the object's construction and cannot be modified afterward.

```
private String name;
private String email;
private String password;
private String mobileNo;
private static String returnedName;
private final StorageManager storage;
```

Abstraction:
The UserAction interface defines the method signatures userRegister() and userLogin(). The implementation of these methods is abstracted, leaving the specific logic to be implemented in the UserManager class.

```
@Override
public void userRegister() {
```

```
Override
public void userLogin() {
```

Polymorphism:
The userRegister() and userLogin() methods are overridden from the UserAction interface. This allows different classes to implement these methods with their own logic.

```
@Override
public void userRegister() {
```

```
Override
public void userLogin() {
```

LocationManager Class

- **Attributes**:
  - ✓ **StrtingLocation** -Stores the starting location entered by the user.
  - ✓ **EndingLocation** -Stores the ending location entered by the user.
  - ✓ **Duration** -Stores the travel duration fetched from the Google Distance Matrix API.
  - ✓ **Distance** -Stores the travel distance fetched from the Google Distance Matrix API.
  - ✓ **tCost** -Stores the total travel cost calculated based on the distance.
  - ✓ **AVG_COST_PER_KM** - Represents the average cost per kilometer.
  - ✓ **API_URL** - Stores the base URL for the Google Maps Distance Matrix API.
  - ✓ **Dotenv** - Used to load environment variables, particularly the Google Maps API key.
- **Methods**:
  - ✓ **gettingLocations():** Prompts the user to input the starting and ending locations.
  - ✓ **getTravelDistanceTime():** Fetches travel details (distance, duration, and cost) from the Google Distance Matrix API.
  - ✓ **getStartingLocation()**: Returns the startingLocation attribute.
  - ✓ **getEndingLocation()**: Returns the endingLocation attribute.
  - ✓ **getDistance():** Returns the distance attribute.
  - ✓ **getDuration()**: Returns the duration attribute.
  - ✓ **getTotalCost():** Returns the tCost attribute.
- **OOP Concepts**:

✓ **Inheritance**:
The LocationManager class extends the abstract class Manager, allowing it to inherit and customize behavior.

```
public class LocationManager extends Manager {
```

- ✓ **Encapsulation**:
  The LocationManager class contains private variables like startingtLocation, endingLocation, ,duration, distance, tCost. These variables store trip-related information and are encapsulated within the class, with methods provided to calculate and retrieve the fare and trip details.

  These private static final constants represent immutable values (AVG_COST_PER_KM and API_URL) that cannot be changed after initialization, ensuring data consistency. Encapsulation protects these variables by restricting access to within the class, preventing external modification and exposing only controlled behavior.

```
private String startingLocation;
private String endingLocation;
private String duration;
private String distance;
private double tCost;
private static final Dotenv dotenv = Dotenv.load();


private static final double AVG_COST_PER_KM = 3.093;
private static final String API_URL =
"https://maps.googleapis.com/maps/api/distancematrix/json?";
```

TicketGenerator Class

- **Attributes**:
  - ✓ **dotenv -**Loads environment variables from a .env file for secure configuration management.
  - ✓ **QR_API_URL** –Stores the URL for the QR code API.
  - ✓ **QR_SIZE –**Defines the fixed size of the generated QR code.
  - ✓ **locInfo** – Instance of LocationManager for location handling.
  - ✓ **user** –Instance of UserManager for user related actions.
  - ✓ **storage** –Instance of Storagemanager class for handling data storage.

- **Methods**:
  - ✓ **generateQR():** Uses an external API to generate a QR code for the ticket.
  - ✓ **encodeURL():** The purpose of this method is to generate a URL-safe, encoded string containing ticket information for transmission or storage in a URL.
- **OOP Concepts**:

Encapsulation:

```
private static final Dotenv dotenv = Dotenv.load();
private static final String QR_API_URL = dotenv.get("QR_API_URL");
private static final String QR_SIZE = "&size=300x300";
private final LocationManager locInfo;
private final UserManager user;
private final StorageManager storage;
```

This code shows **encapsulation** by keeping sensitive variables (dotenv, QR_API_URL, etc.) private, making them inaccessible outside the class. It also uses **final** to ensure immutability and encapsulates LocationManager, UserManager, and StorageManager as instance variables.

Polymorphism:

Below code demonstrates polymorphism **t**hrough method overloading. This oop concept allows this methods to have the same name but differ in the number or types of their parameters. The first method is used to encode detailed information about a ticket, creating a structured URL-safe string containing travel and fare details.The second method is a simpler utility function for encoding a single String value, likely for individual location encoding.

```
public static String encodeURL(String fName, String tID,
                               String startLocation, String endLocation,
                               String distance, String duration, double totalFare,
String seatsCount) {
    String data = "Ticket ID: " + tID + "\n\n" +
            "Name: " + fName + "\n\n" +
            "Travel Route: " + startLocation + " -> " + endLocation + "\n" +
            "Distance: " + distance + "\n" +
            "Duration: " + duration + "\n" +
            "Seat Count: " + seatsCount + "\n" +
            "Total Fare: " + totalFare;

    return URLEncoder.encode(data, StandardCharsets.UTF_8);
}

public static String encodeURL(String location) {
    try {
        return URLEncoder.encode(location, "UTF-8");
    } catch (UnsupportedEncodingException e) {
        e.printStackTrace();
    }
    return "";
}
```

UIManager Class

- **Attributes:**
  - ✓ No attributes

- **Methods**:
  - ✓ **ShowMainMenu():** Displays the main menu for the user to choose options.
  - ✓ **ShowLoginProcess():** Displays the login process menu.
  - ✓ **ShowSystemArt():** Displays the system's ASCII art.

StorageManager Class

- **Attributes**:

- ✓ **dotenv -** Loads environment variables from a .env file for secure configuration management.
- ✓ **url -**Holds the **connection string** for a MySQL database retrieved from the .env file.
- ✓ **username –**Holds the database username retrieved from the .env file.
- ✓ **password –**Holds the database password retrieved from the .env file.
- ✓ **savedPass –**Temporarily stores the password retrieved from the database.
- ✓ **savedName –**Temporarily stores the name retrieved from the database.
- ✓ **savedTicketID –**Temporarily stores the ticket ID retrieved from the database.
- ✓ **savedUserID -** Temporarily stores the user ID retrieved from the database.

- **Methods**:
  - ✓ **getConnection():** Establishes and returns a connection to the database using the provided URL, username, and password.
  - ✓ **travelDataInsert():** Inserts travel data into the trips table of the database.
  - ✓ **userDataInser():** Inser user data into the Users tables of the database**.**
  - ✓ **ticketSaving():** Inserts ticket data into the Tickets table of the database.
  - ✓ **hashPassword():** Hashes a plaintext password using BCrypt.
  - ✓ **verifyPassword():** Verifies a plaintext password against a hashed password.
  - ✓ **getPassFromTable():** Retrieves the password, name, and user ID for a given email from the Users table.
  - ✓ **getTicketID():** Retrieves the ticket ID for a given user name from the Tickets table.

- ✓ **updateSeatsTable():** Inserts the ticket ID and user ID into the seats table and updates the availability of seats for the specified ticket by decrementing it.
- ✓ **cancelTicket():** Updates the availability of seats for the specified ticket by decrementing it and updates seat availability by incrementing it for the specified ticket.

- **OOP Concepts**:

Encapsulation:

In below code **Encapsulation** ensures sensitive data (like database credentials) is protected.It ensures anyone can not be directly accessed or modified from outside the StorageManager class.

```
private static final Dotenv dotenv = Dotenv.load();
private static final String url = dotenv.get("DB_URL");
private static final String username = dotenv.get("DB_USERNAME");
private static final String password = dotenv.get("DB_PASSWORD");
private static String savedPass;
private static String savedName;
private static String savedTicketID;
private static String savedUserID;
```

Inheritance:

```
public class StorageManager extends Manager
```

The code defines StorageManager as a subclass of Manager, inheriting its properties and methods. This allows StorageManager to reuse or extend the functionality of Manager while focusing on storage-related tasks.

Ticket Booking Class

- **Attributes**

- ✓ **locationManager**: Manages location and travel-related functionalities.
- ✓ **storageManager**: Handles storage and retrieval of data, such as saving tickets and updating seat tables.
- ✓ **generateTicket**: Responsible for generating tickets and QR codes.
- ✓ **userManager**: Manages user-related operations such as retrieving user data.

- **Methods**

✓ **TicketBooking()**: Constructor to initialize locationManager, storageManager, generateTicket, and userManager.

✓ **bookTicket()**: Handles the ticket booking process by collecting travel and user details, calculating costs, and saving ticket data.

✓ **cancelTicket()**: Cancels an existing ticket by taking the ticket ID as input and interacting with the storage system.

- **OOP Concepts**

Encapsulation

The attributes are private and can only be accessed or modified through methods, ensuring controlled access.

```
private final LocationManager locationManager;
private final StorageManager;
private final TicketGenerator generateTicket;
private final UserManager;
```

Inheritance:

The TicketBooking class extends the Manager class, inheriting its properties or methods if any.

```
public class TicketBooking extends Manager {
```

Abstraction:

Complex operations like generateQR (in TicketGenerator) and ticketSaving (in StorageManager) abstract their implementation details, allowing TicketBooking to use them without knowing how they work.

```
storageManager.ticketSaving(TravelInformation);
```

```
generateTicket.generateQR(TravelInformation);
```

MainUI Class

- **Attributes**

✓ **inputNumber**: Stores user input for the login process.

✓ **inputMenuNumber**: Stores user input for the main menu options.

- **Methods**

✓ **main(String[] args)**

The `main` method in the main file acts as the entry point for the application, where different components like `LocationManager`, `StorageManager`, `TicketGenerator`, and `UserManager` are integrated to work together. It orchestrates the execution of the program by calling methods like `bookTicket()` and `cancelTicket()` to perform the intended operations, managing the flow of the application.

- **OOP Concepts**

Encapsulation:

The attributes inputNumber and inputMenuNumber are private, ensuring controlled access to their values.

```
private static int inputNumber = 0;
private static int inputMenuNumber = 0;
```

Abstract Class: Manager

- **Attributes**

✓ **input**:
The line **protected final Scanner input = new Scanner(System.in);** initializes a Scanner object to read input from the console. The protected access modifier allows it to be accessed within the current class and any subclasses, while final ensures that the input object cannot be reassigned after initialization.

- **Methods**

✓ No methods are explicitly defined in the Manager class.

- **OOP Concepts**

Abstraction

The Manager class is declared as abstract. It provides a base structure for subclasses but cannot be instantiated directly.

```
Public abstract class Manager {
    protected final Scanner input = new Scanner(System.in);
}
```

Encapsulation:

The input attribute is marked as protected, restricting its access to the Manager class and its subclasses, ensuring controlled input functionality.

Interface: UserAction

- **Methods**

✓ **userRegister():** Abstract method to define the user registration process.

✓ **userLogin():** Abstract method to define the user login process.

- **Attributes**

✓ No attributes are explicitly defined in the UserAction interface.

- **OOP Concepts**

Abstraction

The UserAction interface defines method signatures (userRegister and userLogin) without providing implementation.

```
public interface UserAction {
    void userRegister();

    void userLogin();
}
```

4. Illustration of the Demonstration of Our Final System

Dear Madam, please take a moment to watch our Demonstration:

Video link

Technologies Used

- Java
- Maven
- Google Cloud Distance Matrix API
- goqr.me API
- bcrypt
- MySQL

Setup Instructions

1. Set up a MySQL database and execute the provided schema and triggers.
2. Configure API keys for:
    - Google Cloud Distance Matrix API.
    - goqr.me API.
3. Configure the **.env** file
    - ***GOOGLE_MAPS_API_KEY***
    - ***DB_URL***
    - ***DB_USERNAME***
    - ***DB_PASSWORD***
    - ***QR_API_URL***
4. Build the project with Maven:

mvn clean install

Database Schema

1. Users Table

```
CREATE TABLE Users (
   user_id VARCHAR(10) PRIMARY KEY,
   name VARCHAR(50) NOT NULL,
   email VARCHAR(75) UNIQUE NOT NULL,
   password VARCHAR(60) NOT NULL,
   phone VARCHAR(10),
   created_at TIMESTAMP DEFAULT CURRENT_TIMESTAMP
);
```

Trigger for User ID Generation:

```
DELIMITER $$
CREATE TRIGGER before_insert_users
BEFORE INSERT ON Users
FOR EACH ROW
BEGIN
   DECLARE next_id INT;
   SELECT COALESCE(MAX(CAST(SUBSTRING(user_id, 2) AS UNSIGNED)), 0) + 1 INTO next_id FROM Users;
   SET NEW.user_id = CONCAT('U', LPAD(next_id, 4, '0'));
END$$;
DELIMITER ;
```

2. Trips Table

```
CREATE TABLE Trips (
   trip_id VARCHAR(10) PRIMARY KEY,
   start_location VARCHAR(100) NOT NULL,
   end_location VARCHAR(100) NOT NULL,
   distance VARCHAR(20) NOT NULL,
   duration VARCHAR(20) NOT NULL,
   fare DECIMAL(10, 2) NOT NULL,
   created_at TIMESTAMP DEFAULT CURRENT_TIMESTAMP
);
```

Trigger for Trip ID Generation:

```
DELIMITER $$
CREATE TRIGGER before_insert_trips
BEFORE INSERT ON Trips
FOR EACH ROW
BEGIN
   DECLARE next_id INT;
   SELECT COALESCE(MAX(CAST(SUBSTRING(trip_id, 2) AS UNSIGNED)), 0) + 1 INTO next_id FROM Trips;
   SET NEW.trip_id = CONCAT('R', LPAD(next_id, 4, '0'));
END$$;
DELIMITER ;
```

3. Tickets Table

```
CREATE TABLE Tickets (
   ticket_id VARCHAR(10) PRIMARY KEY,
   user_name VARCHAR(50) NOT NULL,
   start_location VARCHAR(100) NOT NULL,
   end_location VARCHAR(100) NOT NULL,
   distance VARCHAR(20) NOT NULL,
   duration VARCHAR(20) NOT NULL,
   total_fare DECIMAL(10, 2) NOT NULL,
   created_at TIMESTAMP DEFAULT CURRENT_TIMESTAMP
);
```

Trigger for Ticket ID Generation:

```
DELIMITER $$
CREATE TRIGGER before_insert_tickets
BEFORE INSERT ON Tickets
FOR EACH ROW
BEGIN
   DECLARE next_id INT;
   SELECT COALESCE(MAX(CAST(SUBSTRING(ticket_id, 2) AS UNSIGNED)), 0) + 1 INTO next_id
   FROM Tickets;
   SET NEW.ticket_id = CONCAT('T', LPAD(next_id, 4, '0'));
END$$
DELIMITER ;
```

4. Seats Table

```
CREATE TABLE Seats (
   seat_id VARCHAR(10) NOT NULL PRIMARY KEY,
   ticket_id VARCHAR(10) NOT NULL,
   user_id VARCHAR(10) NOT NULL,
   availability TINYINT(1) NOT NULL DEFAULT 1
);
```

Trigger for Seat ID Generation:

```
DELIMITER $$
CREATE TRIGGER before_insert_seats
  BEFORE INSERT ON Seats
  FOR EACH ROW
BEGIN
  DECLARE next_id INT;
  SELECT COALESCE(MAX(CAST(SUBSTRING(seat_id, 2) AS UNSIGNED)), 0) + 1 INTO next_id
  FROM Seats;
  SET NEW.seat_id = CONCAT('S', LPAD(next_id, 4, '0'));
END;
DELIMITER ;
```

Trigger to limit the Seat Count to 50:

```
DELIMITER $$
CREATE TRIGGER limit_seats_before_insert
  BEFORE INSERT ON Seats
  FOR EACH ROW
BEGIN
  DECLARE row_count INT;

  SELECT COUNT(*) INTO row_count FROM Seats;

  IF row_count >= 50 THEN
    SIGNAL SQLSTATE '45000'
      SET MESSAGE_TEXT = 'Cannot add more rows. Maximum seat limit (50) reached.';
  END IF;
END;
DELIMITER ;
```

The system features a simple and intuitive user interface:

1. **Login Screen**: Users can log in or register using the system's simple interface.
2. **Main Menu**: Once logged in, users can:
    - Calculate distances and ticket costs.
    - Book tickets.
    - Cancel tickets.
    - Quit
3. **Ticket Booking**: After booking a ticket, users receive a QR code.
4. **QR Code Validation**: During boarding, the QR code is scanned for quick validation.

- We used the **try-catch block** in several places to handle errors by catching exceptions that may occur during execution, allowing the program to continue running without crashing and providing a way to respond to specific errors.

5. Modifications for the Initial Plan and Reasons

While the project proposal outlined the basic system requirements, some modifications were made during the development process:

1. **Refinement of Classes**: Certain classes, such as LocationManager, were expanded to include additional functionality, like the ability to validate locations and calculate fares.
2. **Added a few extra Classes**: **Manager, UIManager** and an **interface** called **UserAction** were added.

These changes were made to improve the system's efficiency, readability, usability, and scalability.

6.Output

```
1) Register
2) Login
0) Quit
2
--Login--

Enter Email: malandealwis@gmail.com
Enter Password: 2419624196pasindu
Successfully Logged In!
Hey, Malan
---Select Option---
        1) Calculate Distance & Ticket Cost
        2) Book a Ticket
        3) Cancel a Ticket
        0) Quit
2
Enter Starting Location: university of kelaniya
Enter Ending Location: kiribathgoda

university of kelaniya -> kiribathgoda
Distance: 1.9 km
Duration: 6 mins
Travel Cost: RS.32.88

Enter the number of seats you want to book: 4
How many Full Tickets you want to book? (Default: 1)  : 3
How many Half Tickets you want to book? (Default: 0)  : 1

-----Booking Summary-----
From: university of kelaniya
To: kiribathgoda
Booked Seats: 4
Total Cost: Rs. 115.06845

Do you need to Checkout? (Y/N):
Y
Thank you! Enjoy the journey!
Your Ticket QR code here: http://api.qrserver.com/v1/create-qr-code/?data=Ti
Seats table updated successfully!
Thank You for using our Ticket Booking System!
```

Teamwork

- **T.M.D.P.M. De Alwis**: Implemented the StorageManager class and TicketGenerator class.
- **I.D.N.B.Amaranath**: Implemented the TicketBooking class.
- **K.G.H.D. Bandara**: Implemented the MainUI class, UIManager class, Manager abstract class and UserAction interface.
- **M.K.S. Vinhara**: Implemented the UserManager class.
- **K.A.S.M. Jayawardhana**: Implemented the LocationManager class.