



**Final Report**  
**CSF304(Design Patterns)**  
**Bachelor of Science in Computer Science**  
**AI Development and Data Science Year III, Semester II**

**Taxi Booking System**

Submitted by  
Chimi Rinzin (12210002)  
Deepak Ghalley (12210005)  
Thukten Dema (12210037)  
Dawa (12210046)

*Gyalpozhing College of Information Technology*  
*Guided By: Mrs.Pema Wangmo*

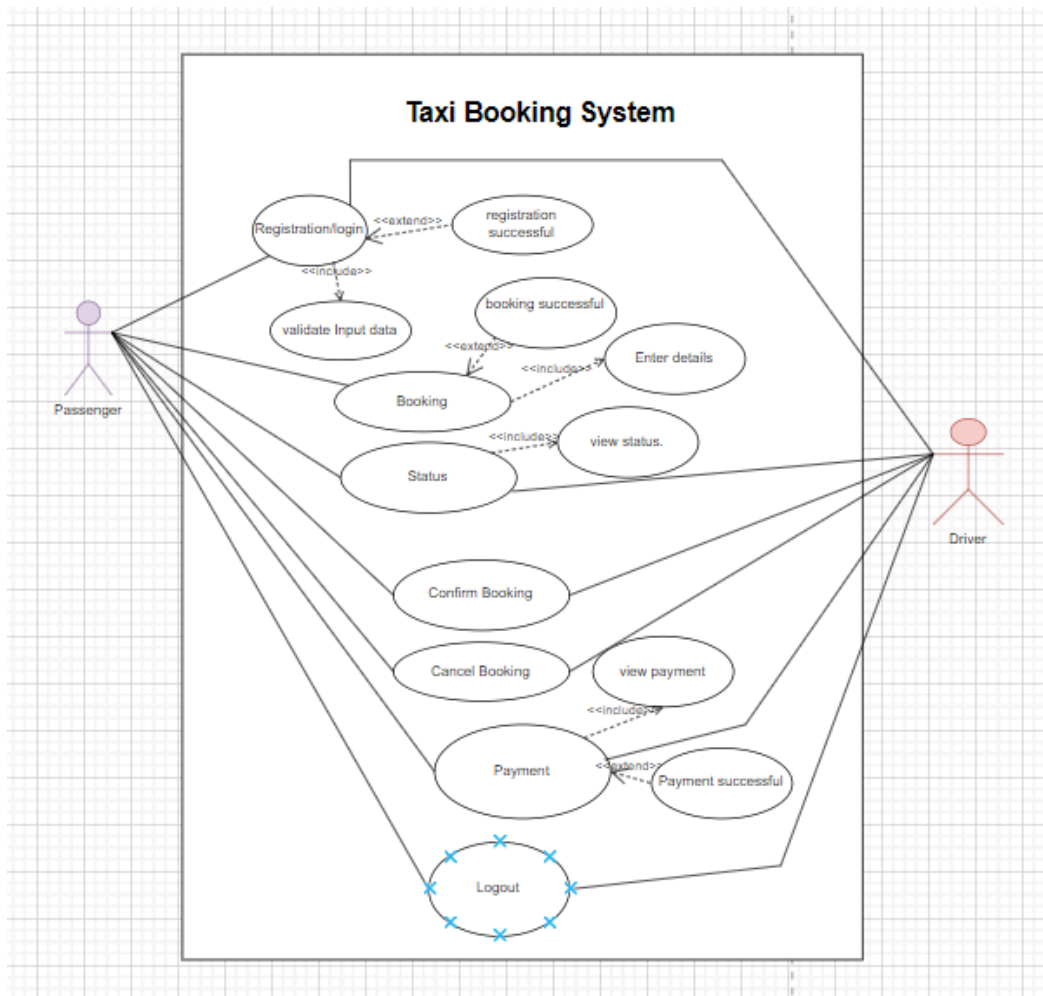
## Table of Contents

<b>Content</b>	<b>Page</b>
<b>Brief Project Description</b>	3
<b>Use Cases</b>	3-4
<b>Source Code</b>	4
<b>Framework</b>	4-10
<b>Class Diagram</b>	11
<b>User Interface</b>	12-26
<b>Justifications for all the design patterns used</b>	27
<b>Challenges</b>	28
<b>Conclusion</b>	28

## 1. Brief project Description

This project aims to develop a comprehensive and user-friendly online taxi booking system using Java, incorporating robust frameworks and design patterns to ensure scalability and reliability. The system will provide registered users with essential features such as booking taxis alongside receiving notifications from the admin about system updates. By integrating a developed framework and using Swing to demonstrate the implementations, the project employs various design patterns to achieve its objectives: the Factory Pattern for user sign-up, the Singleton Pattern for user Database Connection, the State Pattern for managing booking status, the Strategy Pattern for payment, the Command Pattern for handling ride bookings and cancellations, and the Observer Pattern for notifying passengers and drivers. This approach enhances the user experience through secure and efficient booking processes while setting a high standard for future transportation software solutions.

## 2. Use Case



**Users:**

- Passenger
- Driver

**Use Cases of Driver**

- View booked rides by passenger
- Accept or cancel ride

**Use Cases of Passenger**

- Book ride
- View booked ride status
- Make payment

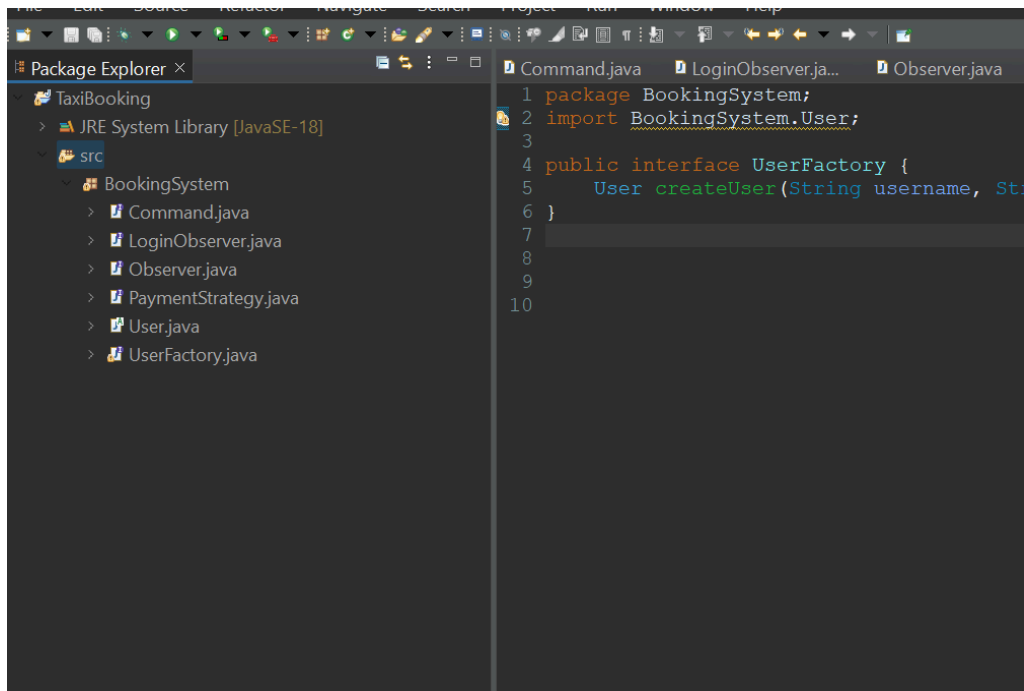
**3. Source Code**

The GitLab link for our project Taxi booking system is:

[https://gitlab.com/12210005/Group7\\_TaxiBookingSystem](https://gitlab.com/12210005/Group7_TaxiBookingSystem)

**4. Framework**

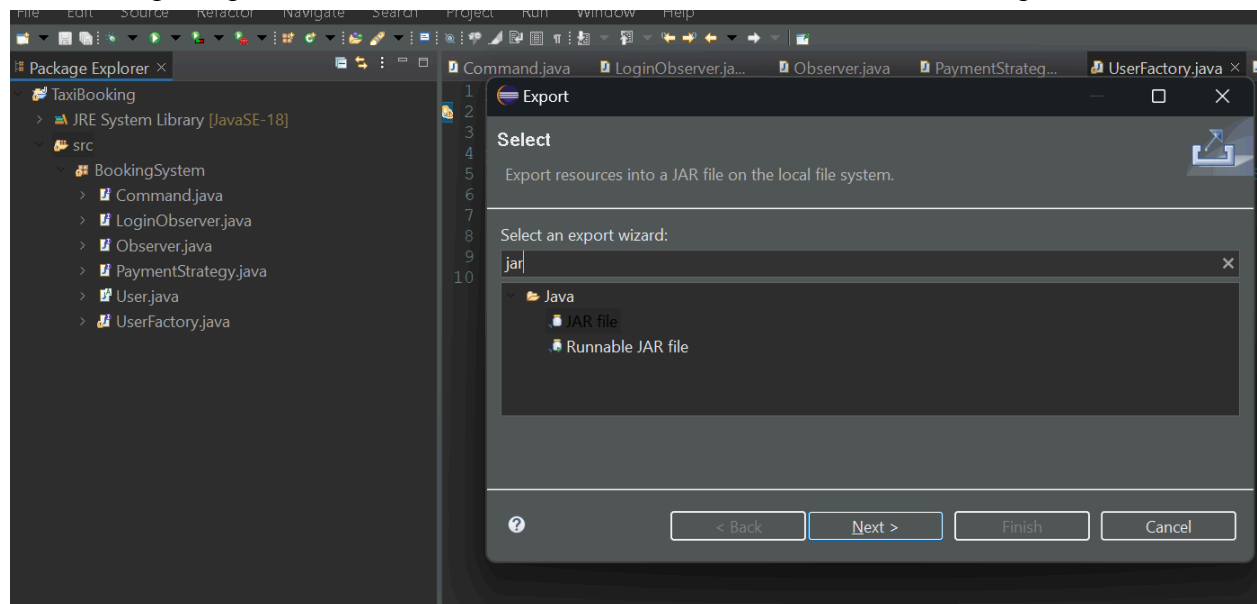
The framework incorporates an immutable state, which is essential for maintaining its extendability, stability, and reliability. This immutability guards against unintended modifications, minimizes errors, and ensures consistent behavior across various applications and developers. Specifically, in the online taxi booking framework, we have defined a pattern interface that remains unchanged. This framework is packaged as a JAR file and imported into our taxi booking application, allowing us to implement it according to our specific requirements.



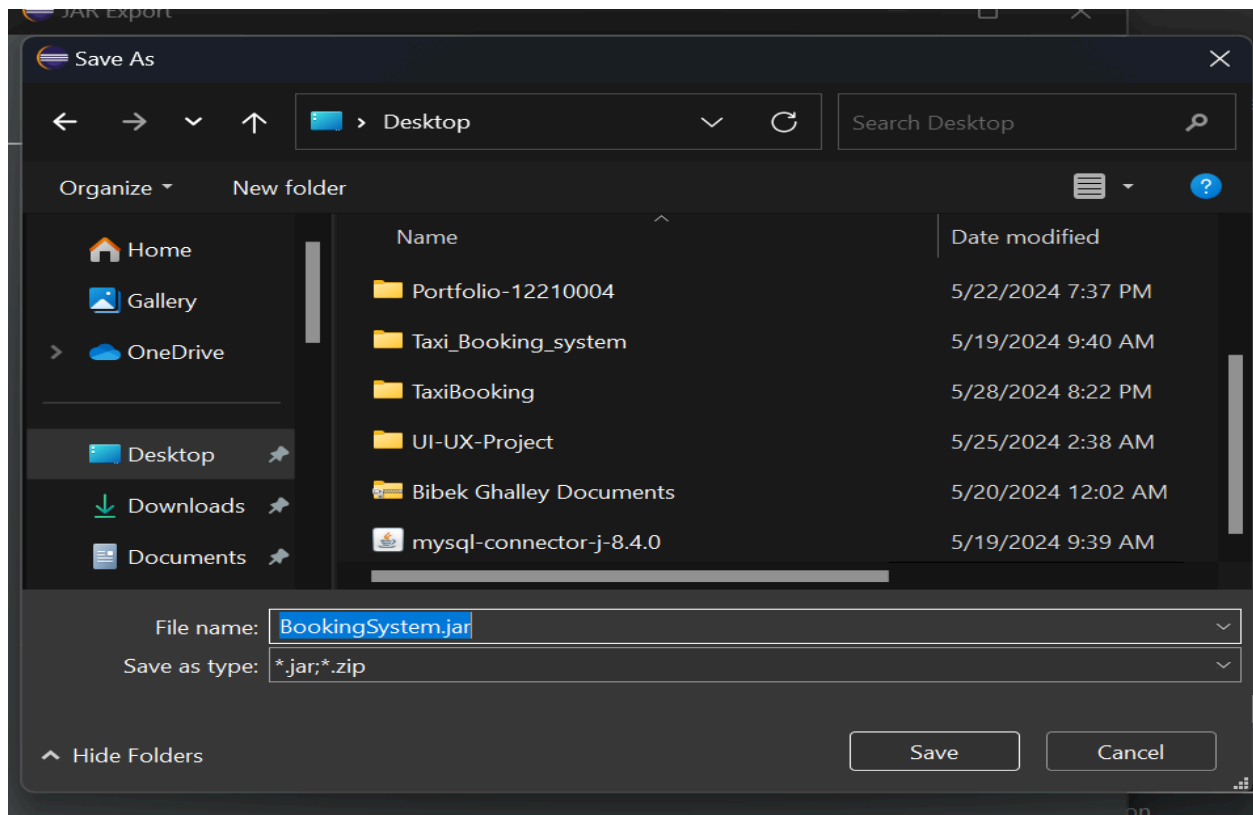
### Framework Conversion to Jar extension

Using Eclipse IDE, we have converted our online taxi booking framework into a JAR file. This approach is highly beneficial as it makes the framework convenient to use, boosts performance, enhances security, and ensures portability. Each directory in the framework contains a class with an immutable state. Packaging and distributing the framework as a JAR file simplifies its integration and use in various applications.

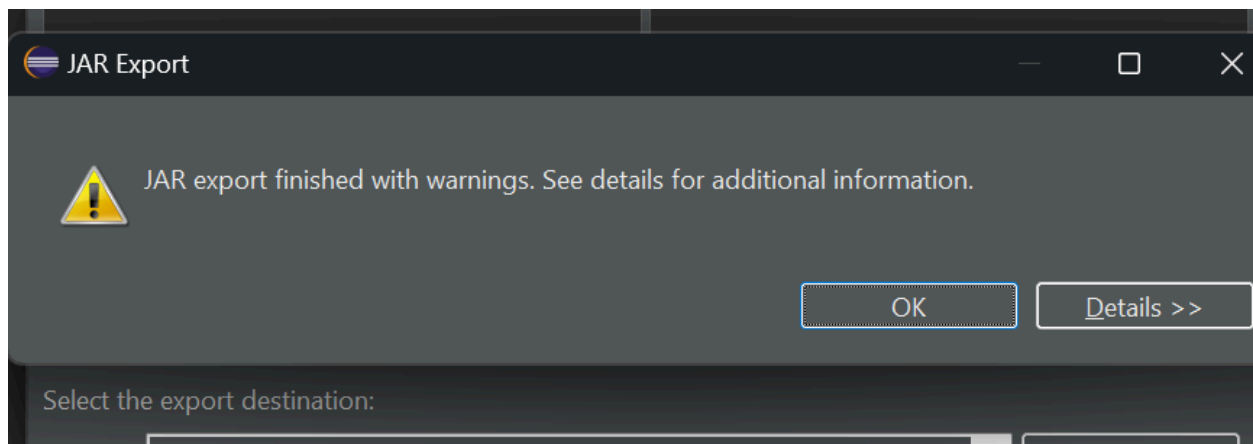
Click on export option and choose 'JAR file' as an extension that should be exported.



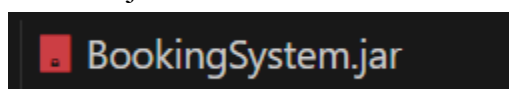
We need to select 'classpath' and 'project' and select the directory where the jar extension file will be saved.



Taxi booking framework is successfully converted to jar extension.



The final jar file will look like the file below:



### Signup for passenger and driver implemented using factory design pattern:

```
package factory;
import BookingSystem.User;
public interface UserFactory {
    User createUser(String username, String phoneNumber, String email,
String hashedPassword);
}
```

This interface defines a contract for creating user objects. Any class that implements this interface must provide an implementation for the createUser method.

### Single pattern for database connection:

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

public Connection getConnection() {
    return connection;
}
```

The Singleton pattern in the DatabaseConnection class is implemented through the getInstance method, which ensures only one instance of the class is created.

### Observer pattern for notification:

#### 1. DriverObserver class

```
package observer;
import BookingSystem.*;

public class DriverObserver implements Observer {
    private String driverEmail;

    public DriverObserver(String driverEmail) {
```

```

        this.driverEmail = driverEmail;
    }

    @Override
    public void update(String message) {
        // Implement logic to notify the driver
        System.out.println("Notification to driver (" + driverEmail
+ "): " + message);
    }
}

```

The DriverObserver class implements the Observer pattern to notify drivers of updates. It is part of the observer package and implements the Observer interface from the BookingSystem package. The class has a private field driverEmail to store the driver's email address, which is initialized via the constructor. The update method, mandated by the Observer interface, prints a notification message to the console that includes the driver's email and the update message. This setup allows instances of DriverObserver to receive and handle notifications about changes or updates, specifically targeting drivers

### State pattern of booking status:

```

package state;

public interface BookingState {

    void handle(BookingContext booking);

    void nextState(BookingContext booking);
}

```

TheBookingState interface, part of the state package, defines methods for handling booking states within a booking context. It includes handle, which specifies actions to be taken based on the current state, and nextState, which transitions the booking to the next state. This interface serves as a blueprint for various booking states, allowing for flexible state management within the application.



### Command pattern for drivers to accept or cancel bookings:

```
package command;

import java.util.ArrayList;

import java.util.List;

import BookingSystem.Command;

public class CommandInvoker {

    private List<Command> commandQueue = new ArrayList<>();

    public void addCommand(Command command) {

        commandQueue.add(command);

    }

    public void executeCommands() {

        for (Command command : commandQueue) {

            command.execute();

        }

        commandQueue.clear();

    }

}
```

The CommandInvoker class, part of the command package, manages a queue of commands to be executed. It adds commands to the queue with addCommand and executes them with executeCommands, leveraging the execute method of each command object. This design enables separation of command creation and execution, enhancing flexibility and extensibility in command-based operations.

### Strategy pattern for making payment:

```
PaymentStrategy paymentStrategy = null;

    if ("Bank of Bhutan".equals(bank)) {

        paymentStrategy = new BankOfBhutanPayment(accountNumber,
bankCode);

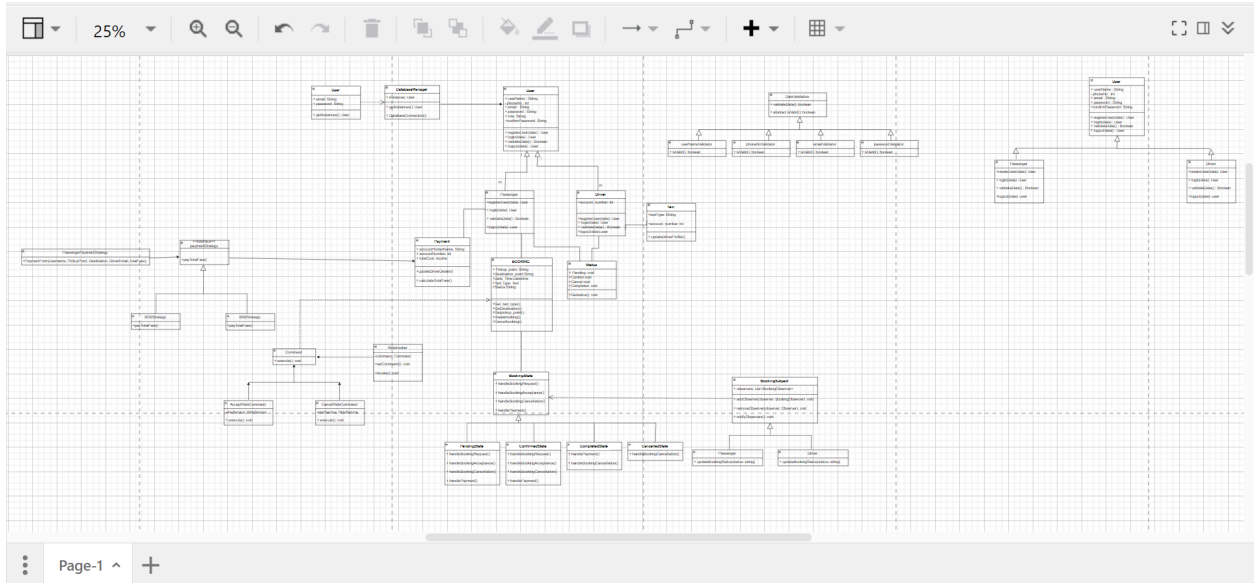
    } else if ("Bhutan National Bank".equals(bank)) {

        paymentStrategy = new
BhutanNationalBankPayment(accountNumber, bankCode);

    }
```

Initializes a PaymentStrategy object based on the provided bank information. If the bank is "Bank of Bhutan", it instantiates aBankOfBhutanpayment strategy with the given account number and bank code. If the bank is "Bhutan National Bank", it creates a BhutanNationalBankPayment strategy with the same parameters.

## 5. Class Diagram



**Link for the class diagram:**

Refer the link below for the class diagram:

[https://drive.google.com/file/d/15buV\\_ptWj8fYPpwhzuIBFB9oYI5UF7\\_u/view?usp=sharing](https://drive.google.com/file/d/15buV_ptWj8fYPpwhzuIBFB9oYI5UF7_u/view?usp=sharing)

## 6. User Interface:


The system's user interface, created with Java Swing, offers an easy-to-use experience. Upon initiation, users are presented with a range of functionalities to explore:

- Sign up if they're new, or log in if they're already registered.
- After logging in, they'll see their personalized homepage with their account details and available options.
- Drivers can update their account details right after logging in.
- Passengers can transfer money using BOB or BNB.
- Both passengers and drivers get notifications about updates from each other.

Our Taxi Booking System user interface starts with a Sign-Up Page where users can select either the passenger or driver role.

The image shows a web browser window with the title "Signup Form". The form contains the following fields and elements:

- Username:** A text input field containing "Phuntsho".
- Phone Number:** A text input field containing "17665152".
- Email:** A text input field containing "phuntsho@gmail.com".
- Password:** A text input field with masked characters ".....".
- Confirm Password:** A text input field with masked characters ".....".
- Role:** A dropdown menu with "Passenger" selected.
- Signup:** A blue button with the text "Signup".
- Login:** A blue link with the text "Already have an account? Login".

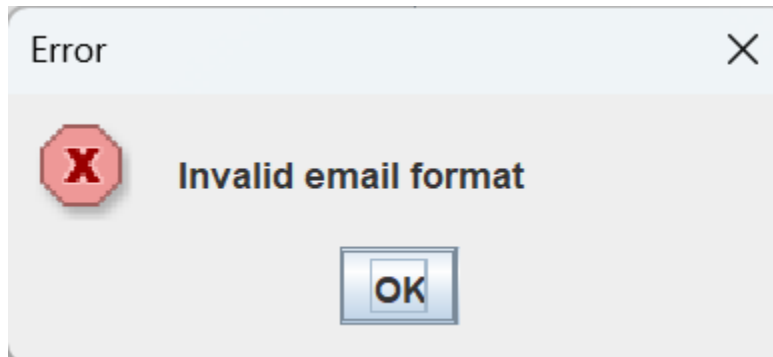
 Signup Form

<b>Username:</b>	<input type="text" value="Tshering"/>
<b>Phone Number:</b>	<input type="text" value="77354332"/>
<b>Email:</b>	<input type="text" value="tshering@gmail.com"/>
<b>Password:</b>	<input type="password" value="....."/>
<b>Confirm Password:</b>	<input type="password" value="....."/>
<b>Role:</b>	<div><div>Driver</div><div></div></div>

Signup

[Already have an account? Login](#)

To ensure that correct user input data is inserted into the database, the system includes form validation for all user inputs and notifies the user when incorrect data is entered. The following is an example of user input validation.



The form validation code is given below:

```
// Validate input
if (username.isEmpty() || phoneNumber.isEmpty() || email.isEmpty() || password.isEmpty() || confirmPassword.isEmpty()) {
    JOptionPane.showMessageDialog(SignupPage.this, message: "Please fill in all fields", title: "Error", JOptionPane.ERROR_MESSAGE);
    return;
}

if (!username.matches(regex: "[a-zA-Z]+")) {
    JOptionPane.showMessageDialog(SignupPage.this, message: "Username must contain only letters", title: "Error", JOptionPane.ERROR_MESSAGE);
    return;
}

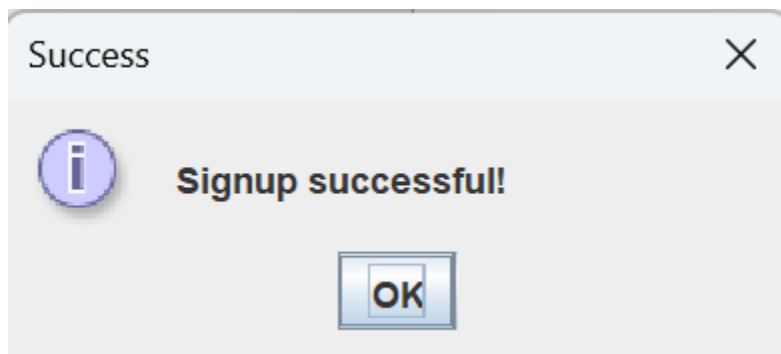
if (!phoneNumber.matches(regex: "\\d{8}")) {
    JOptionPane.showMessageDialog(SignupPage.this, message: "Phone number must contain exactly 8 digits", title: "Error", JOptionPane.ERROR_MESSAGE);
    return;
}

if (!email.matches(regex: "[a-zA-Z0-9._%+-]+@[a-zA-Z0-9.-]+\\.[a-zA-Z]{2,}")) {
    JOptionPane.showMessageDialog(SignupPage.this, message: "Invalid email format", title: "Error", JOptionPane.ERROR_MESSAGE);
    return;
}

if (password.length() < 8) {
    JOptionPane.showMessageDialog(SignupPage.this, message: "Password must be at least 8 characters long", title: "Error", JOptionPane.ERROR_MESSAGE);
    return;
}

if (!password.equals(confirmPassword)) {
    JOptionPane.showMessageDialog(SignupPage.this, message: "Password and confirm password do not match", title: "Error", JOptionPane.ERROR_MESSAGE);
    return;
}
```

After the user enters the correct input data in sign up page, a "Signup Successful" message will be displayed, as shown below:

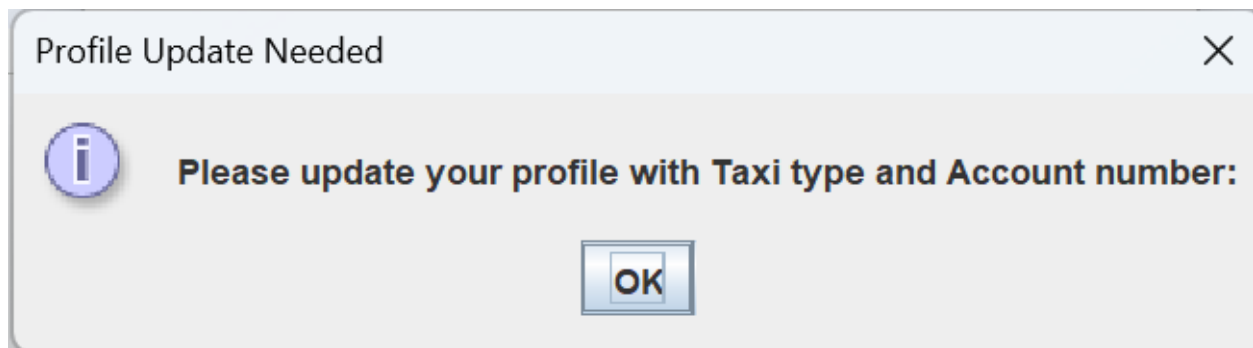


#### **For Driver Login:**

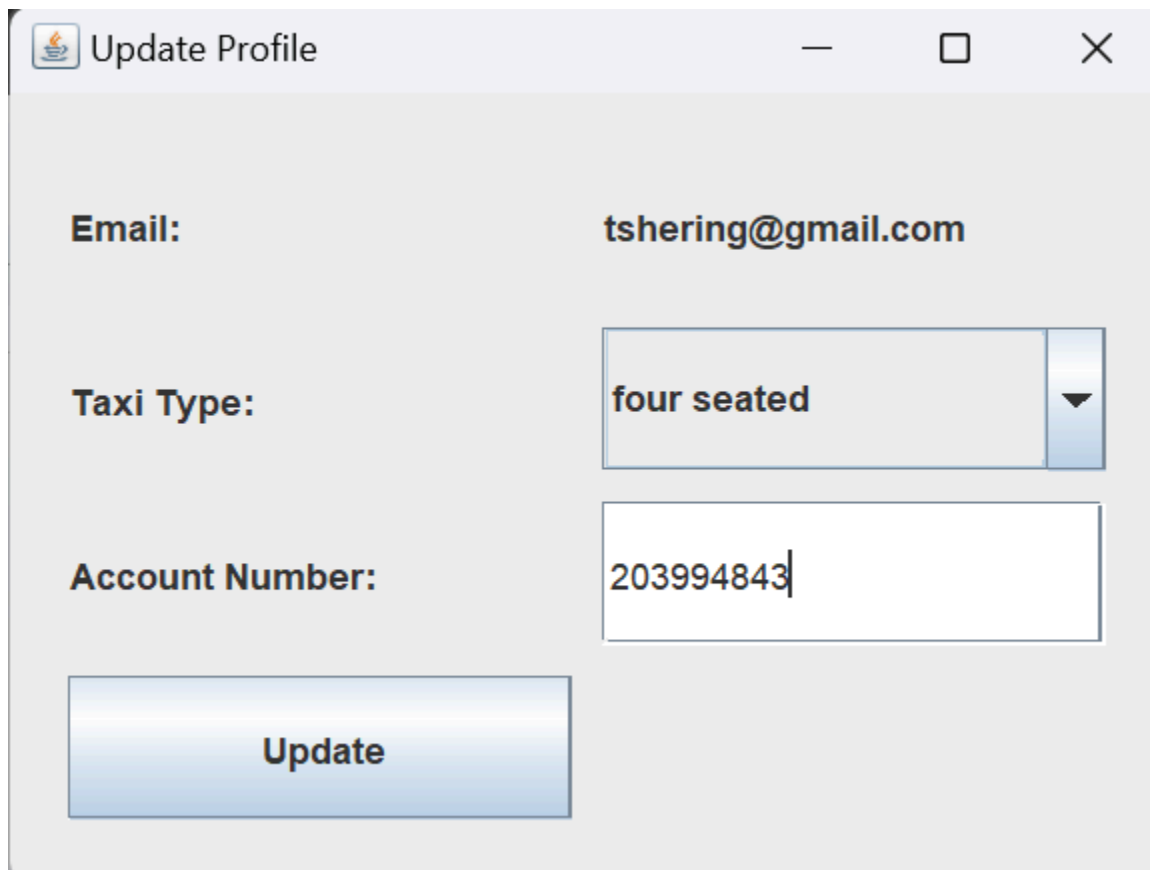
After successfully creating the account, the driver will be directed to the Login Page. Then, they must enter the correct credentials to log in to their account, as shown below:

A screenshot of a web application window titled 'Login Page'. The title bar is light gray and contains a small icon of a coffee cup on the left, the text 'Login Page' in the middle, and standard window controls (minimize, maximize, close) on the right. The main content area is light gray. In the center, the word 'Login' is displayed in a bold, black font. Below this, there are two rows of labels and input fields. The first row has the label 'Email:' followed by an input field containing the text 'tshering@gmail.com'. The second row has the label 'Password:' followed by an input field containing seven dots. Below these input fields are two rectangular buttons with a blue gradient. The left button is labeled 'Login' and the right button is labeled 'Sign Up'.

After the driver clicks the "Login" button, a pop-up message will be displayed, as shown below:

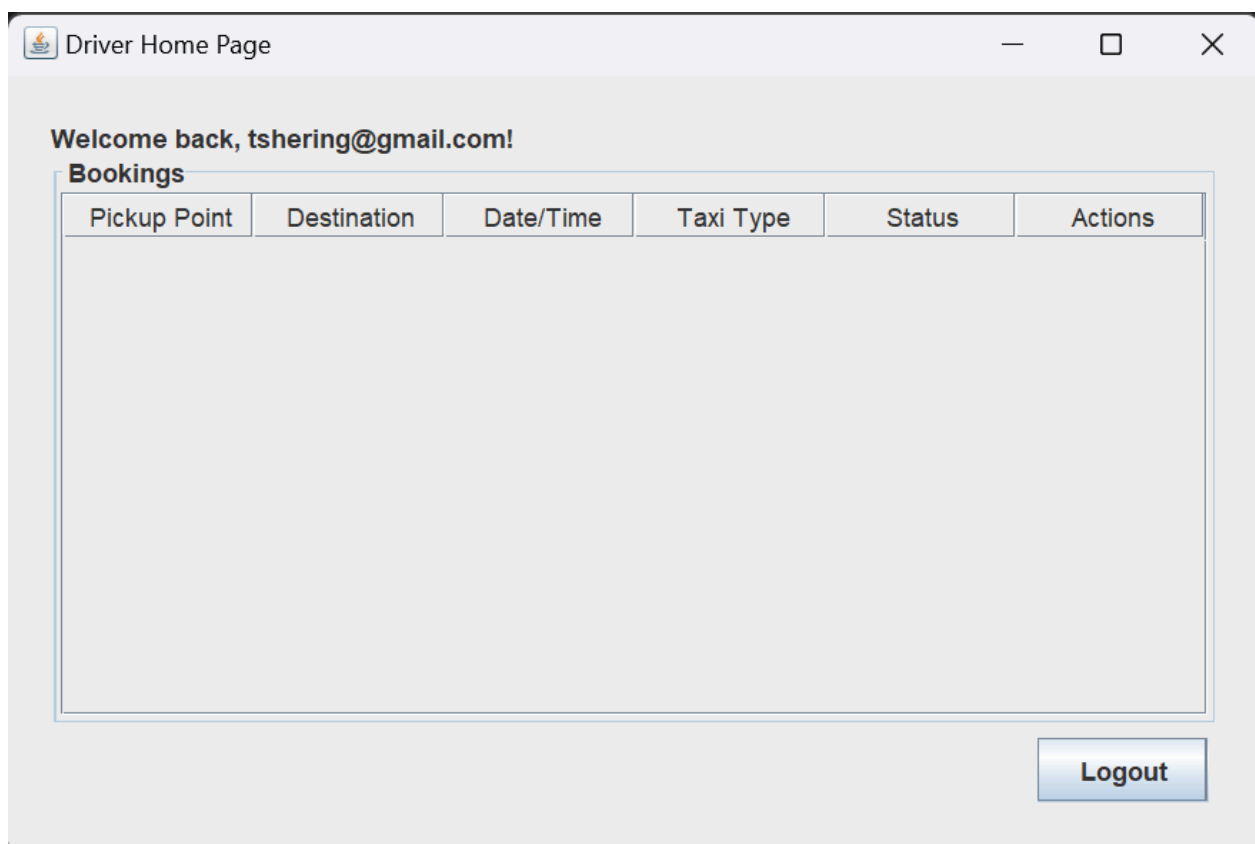
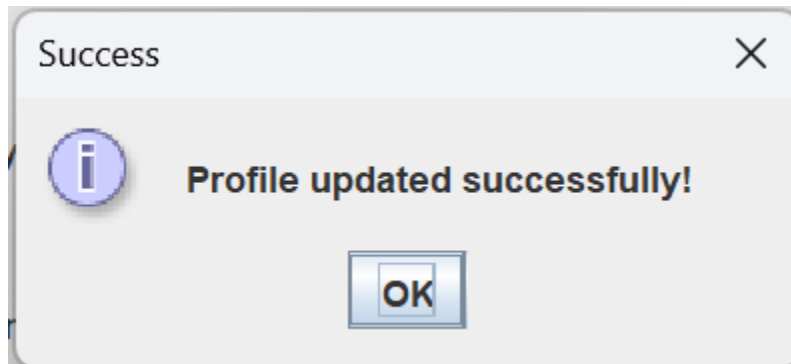


Then, the driver has to click the "Ok" button, and it will redirect to the driver profile update form. There, the driver must fill in the correct credentials such as taxi type and account number, as shown below:

A screenshot of a web form titled "Update Profile" with a small icon of a person and a gear. The form has three input fields: "Email:" with the value "tshering@gmail.com", "Taxi Type:" with a dropdown menu showing "four seated", and "Account Number:" with the value "203994843". At the bottom left of the form is a large blue button labeled "Update".

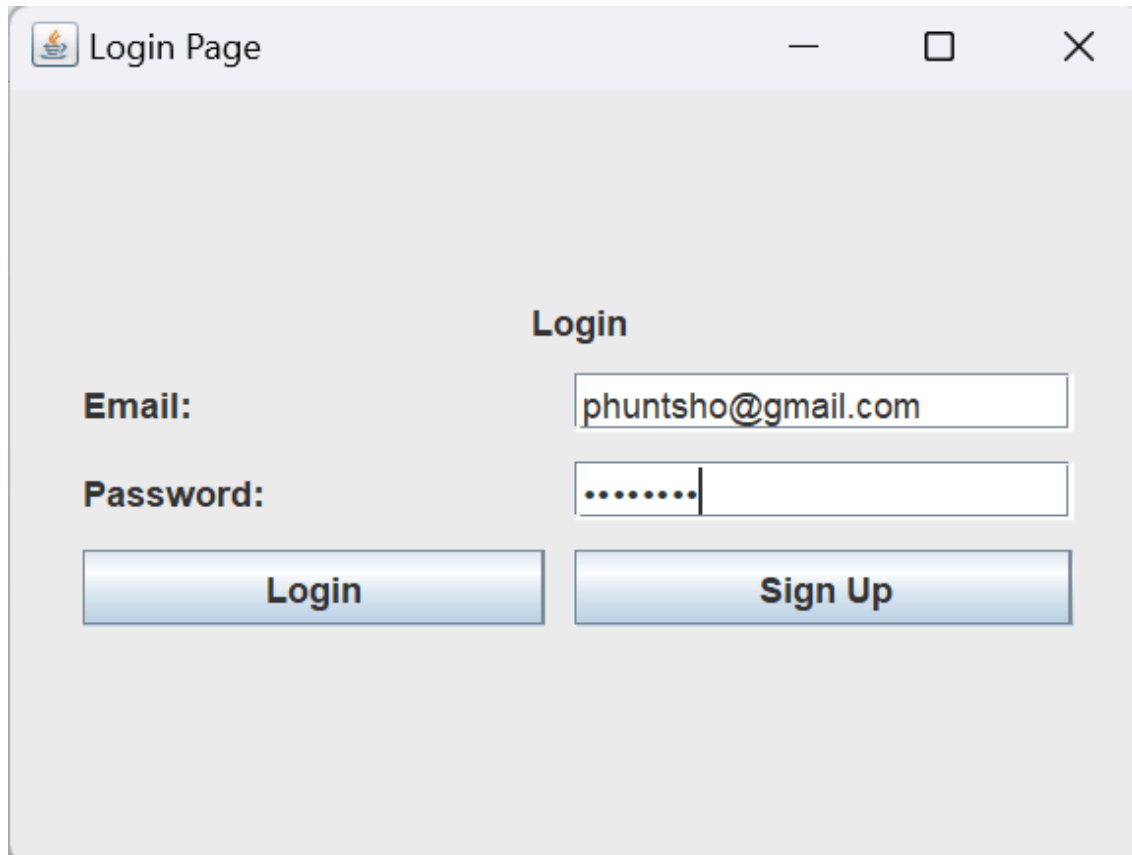


Then the driver should click on the "Update" button, and a pop-up message will be displayed as given below. Afterwards, the driver will be directed to the Driver Home Page.



**For Passenger:**

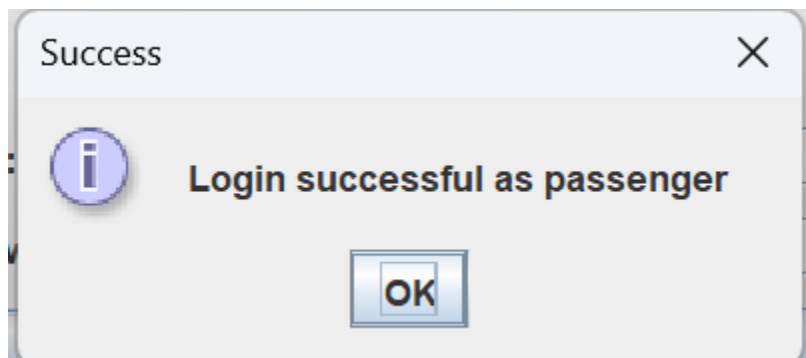
After a successful sign-up, the passenger is redirected to the login page, where they must enter the correct credentials to access their account.



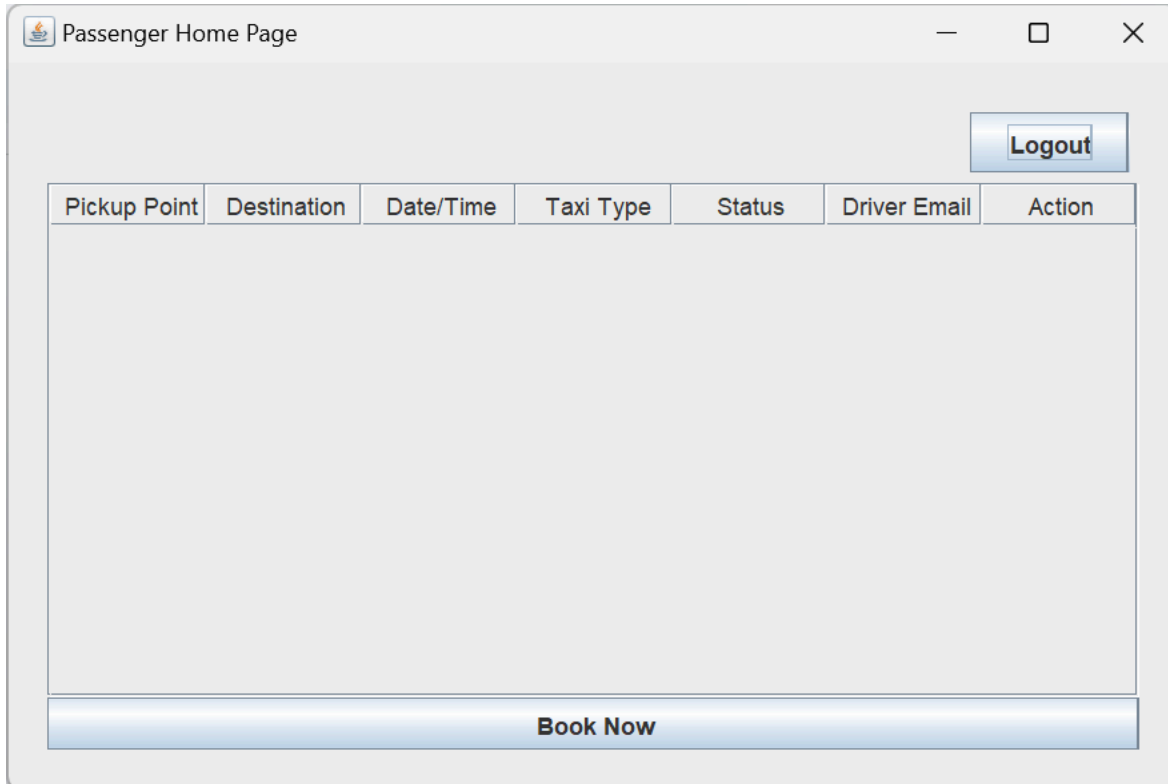
A screenshot of a web application window titled "Login Page". The window has a standard Windows-style title bar with minimize, maximize, and close buttons. The main content area is light gray and contains the following elements:

- A title "Login" centered at the top.
- An "Email:" label followed by a text input field containing "phuntsho@gmail.com".
- A "Password:" label followed by a password input field with masked characters ".....".
- Two buttons at the bottom: "Login" and "Sign Up", both with a blue gradient and a slight shadow.

After correct passenger credentials are entered then the “Login Successful as passenger” message will be displayed as shown below:



After the login successful, it will be redirect to Passenger Home Page as shown below:

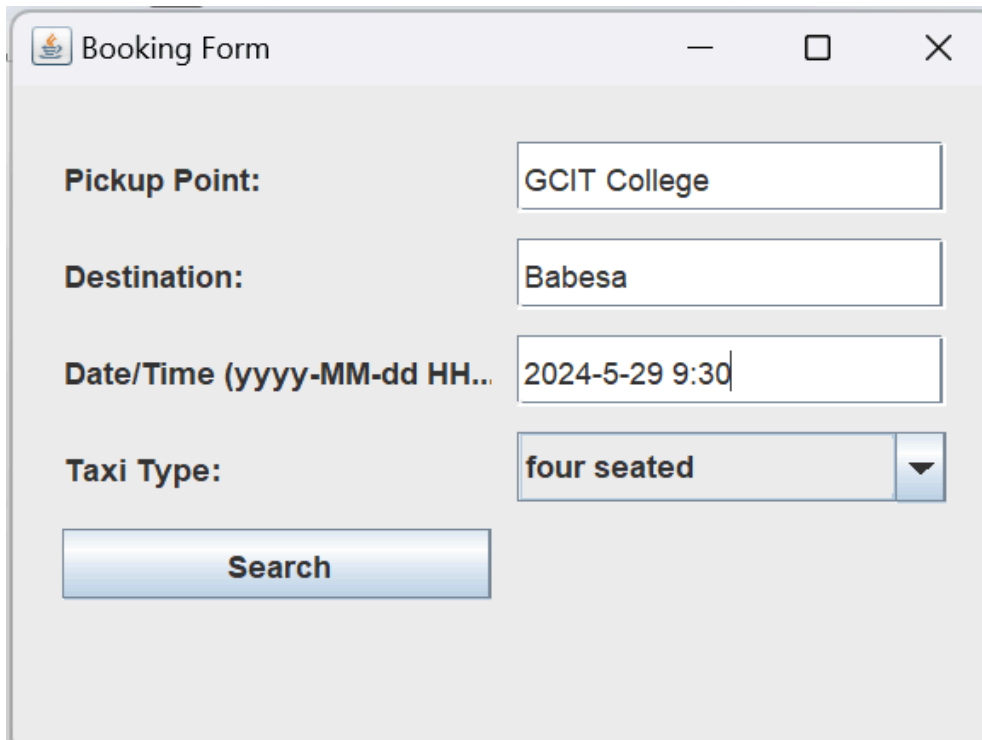


The image shows a web browser window titled "Passenger Home Page". In the top right corner, there is a "Logout" button. Below it is a table with the following headers: "Pickup Point", "Destination", "Date/Time", "Taxi Type", "Status", "Driver Email", and "Action". The table body is empty. At the bottom of the window, there is a "Book Now" button.

Pickup Point	Destination	Date/Time	Taxi Type	Status	Driver Email	Action
--------------	-------------	-----------	-----------	--------	--------------	--------

Book Now

To book a taxi ride, the passenger must click on "Book Now," which redirects to the booking form. There, the user should enter the pick-up point, destination, date/time, and the taxi type.



The image shows a web browser window titled "Booking Form". It contains four input fields with labels: "Pickup Point:" with the value "GCIT College", "Destination:" with the value "Babesa", "Date/Time (yyyy-MM-dd HH...)" with the value "2024-5-29 9:30", and "Taxi Type:" with a dropdown menu showing "four seated". Below these fields is a "Search" button.

Pickup Point: GCIT College

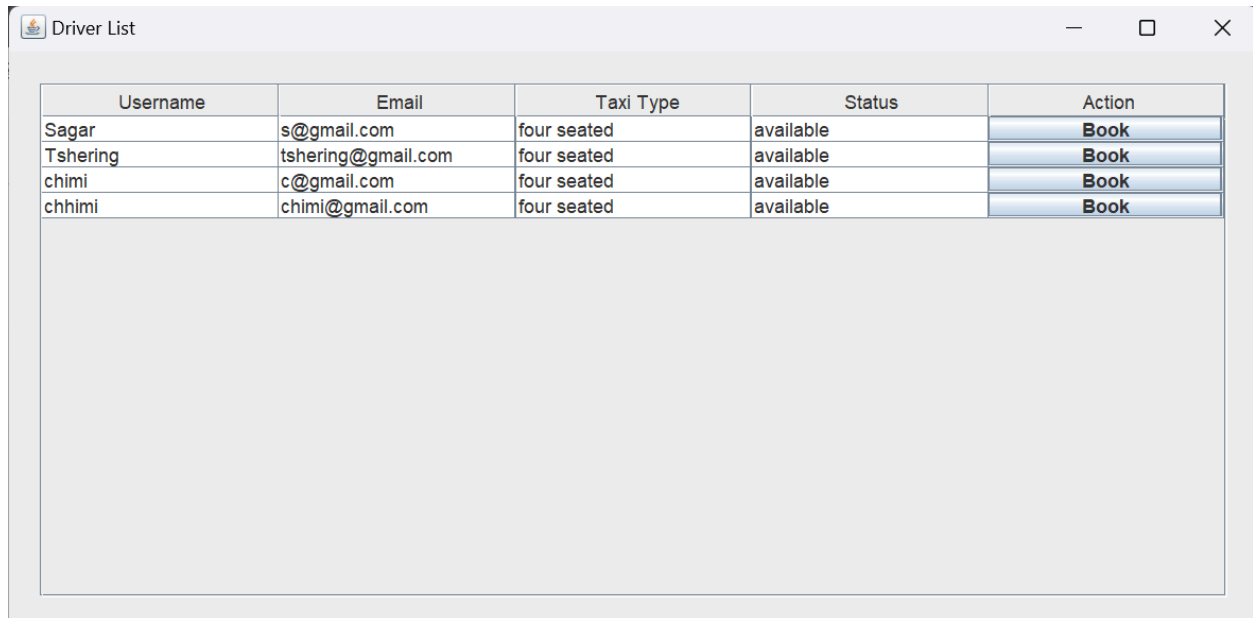
Destination: Babesa

Date/Time (yyyy-MM-dd HH...): 2024-5-29 9:30

Taxi Type: four seated

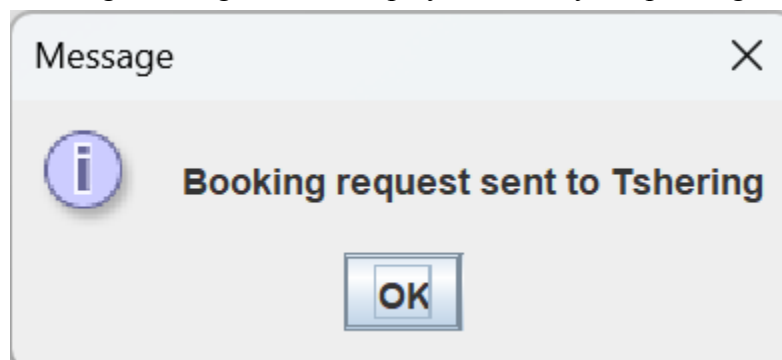
Search

After filling out the booking form, the user must click the "Search" button. Based on the passenger's status and taxi type, a list of matching drivers will be displayed.



Username	Email	Taxi Type	Status	Action
Sagar	s@gmail.com	four seated	available	<b>Book</b>
Tshering	tshering@gmail.com	four seated	available	<b>Book</b>
chimi	c@gmail.com	four seated	available	<b>Book</b>
chhimi	chimi@gmail.com	four seated	available	<b>Book</b>

To book the taxi, the user must click the "Book" button. After that, a "Booking request sent to Tshering" message will be displayed to notify the passenger that the booking has been made.



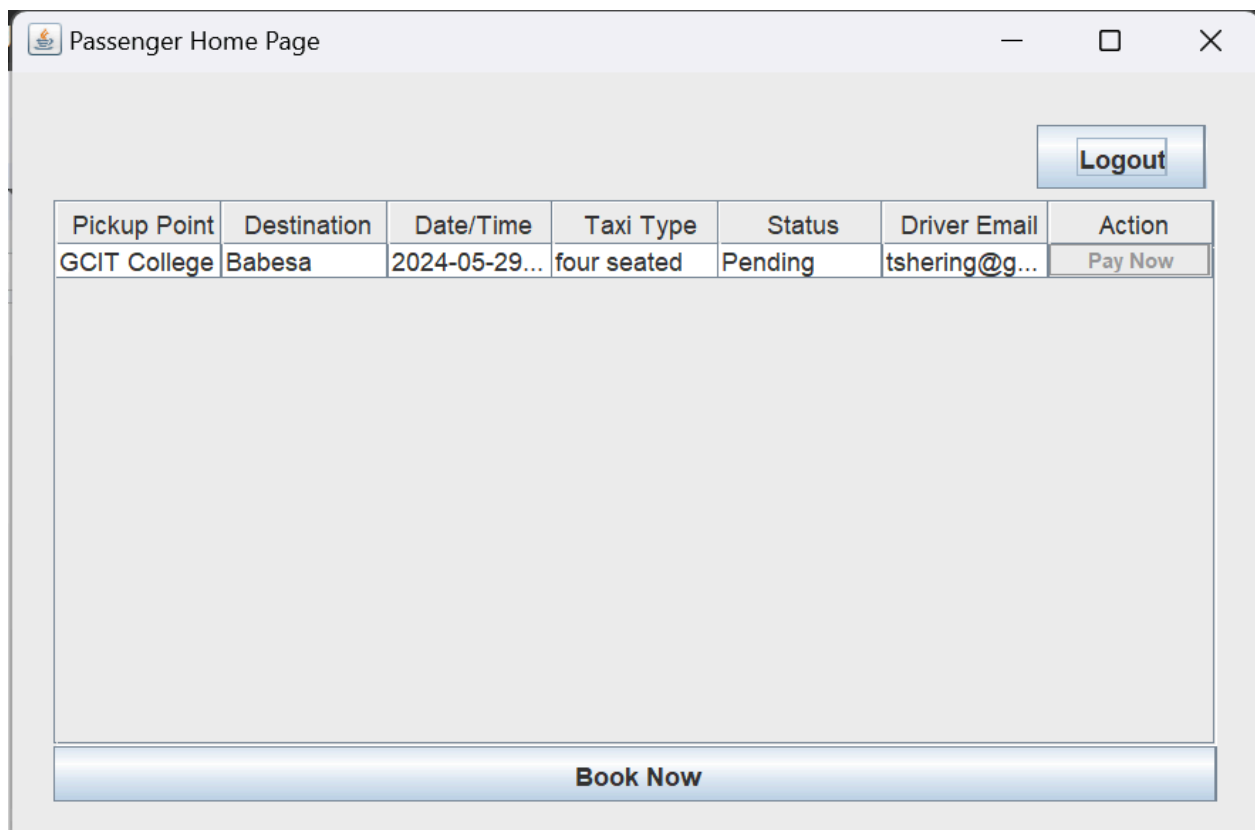
To notify both the passenger and the driver regarding the booking, we have implemented the observer pattern as given below:

```
Notification to passenger (phuntsho@gmail.com): New booking request from phuntsho@gmail.com to Tshering
Notification to driver (tshering@gmail.com): New booking request from phuntsho@gmail.com to Tshering
```

To manage different states for the passenger such as Pending, Accepted, and Completed, we have implemented the state pattern. Since, the passenger has just booked the taxi so the status will be 'Pending'.

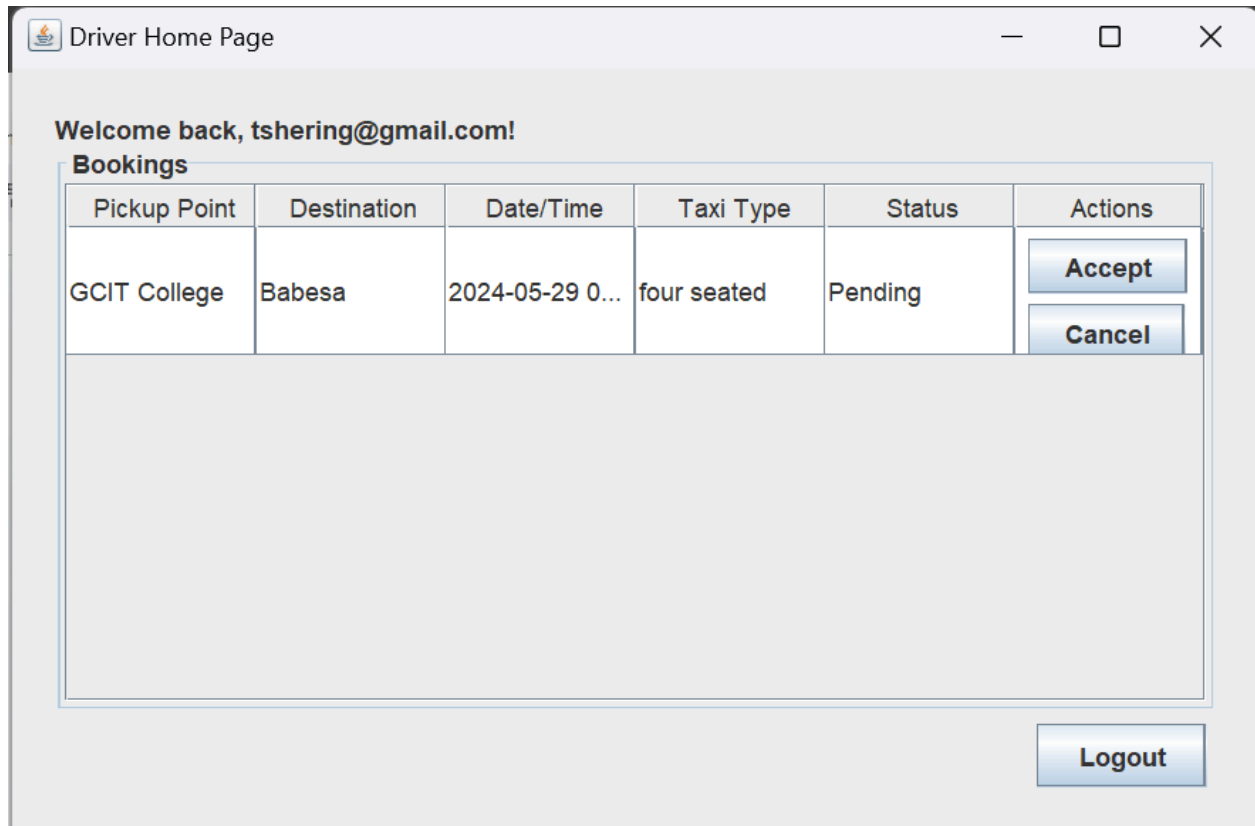
```
Booking is pending. Waiting for driver to accept.
```

Currently, the passenger's status is Pending, and the booking details will be displayed on the Passenger Home Page after booking.

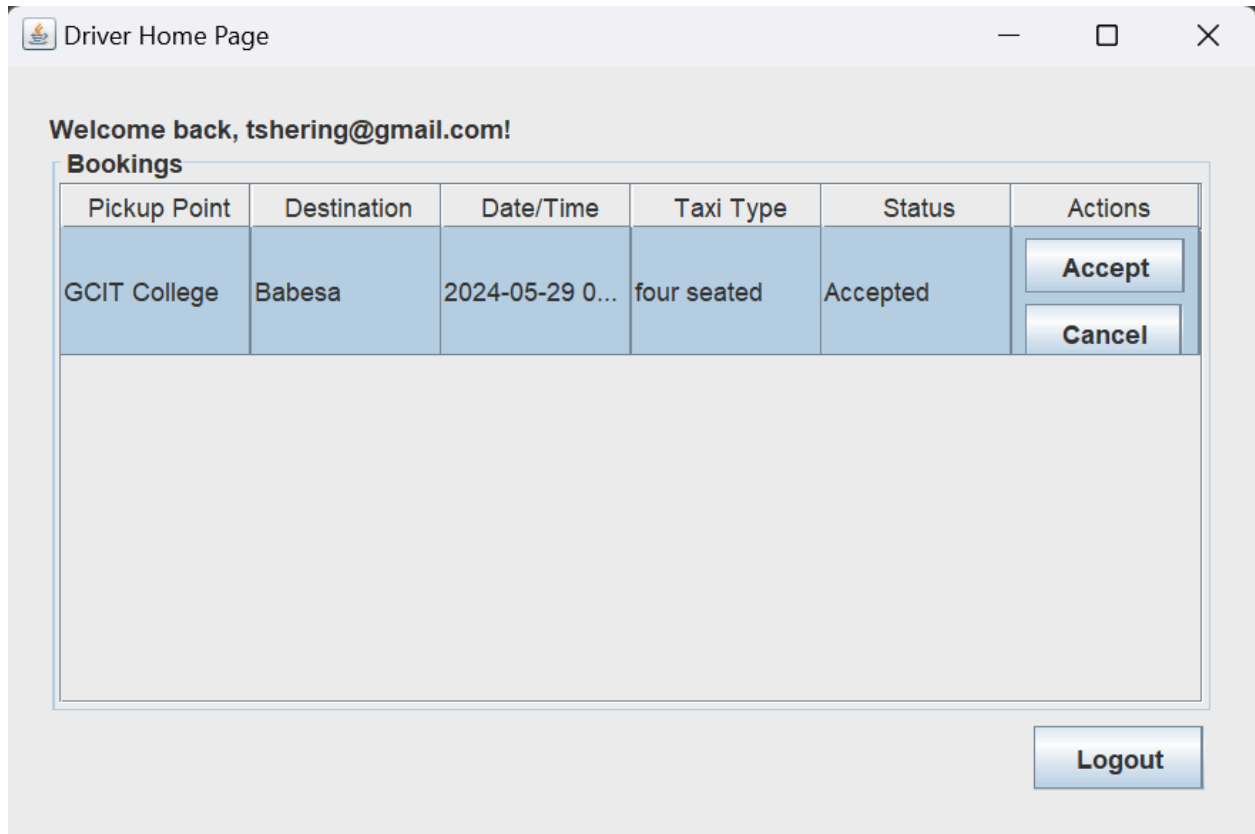


### For Driver:

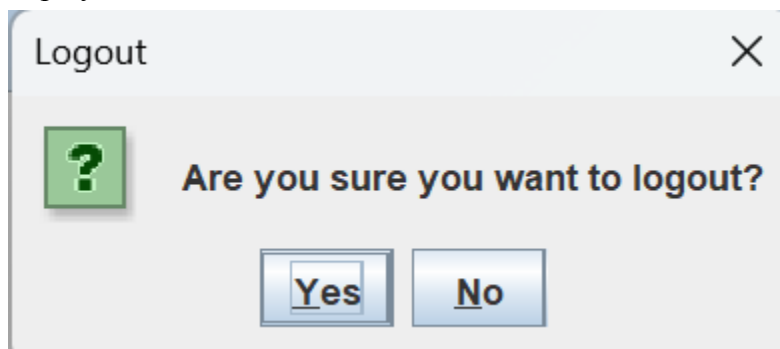
For drivers, after successful login, the booking requests sent by passengers will be displayed on the Driver Home Page, as shown below:



The driver can either accept or cancel the booking request sent by the passenger. If the driver clicks the Accept button, the status will be changed to Accepted. If the driver clicks the Cancel button, the status will be changed to Canceled. For now, we are clicking on the Accept button, as shown below:



After accepting the request, the driver can then log out of their account by clicking on the 'Logout' button. After that, a pop up message "Are you sure you want to logout?" will be displayed.



If the driver clicks the Yes button, they will be redirected to the Login Page. If the driver clicks the No button, they will be redirected back to the Driver Home Page. For now, we are clicking on the Yes button, so it directs to the Login Page.

**Login**

**Email:**

**Password:**

**Login** **Sign Up**

**For passenger:**

To view the booking status, the passenger has to log in to their account again. In the Passenger Home Page, as soon as the booking request has been accepted, the status will be changed to "Accepted," as shown below. After the status changes to Accepted, the "Pay Now" button will be visible for the passenger to proceed with payment.



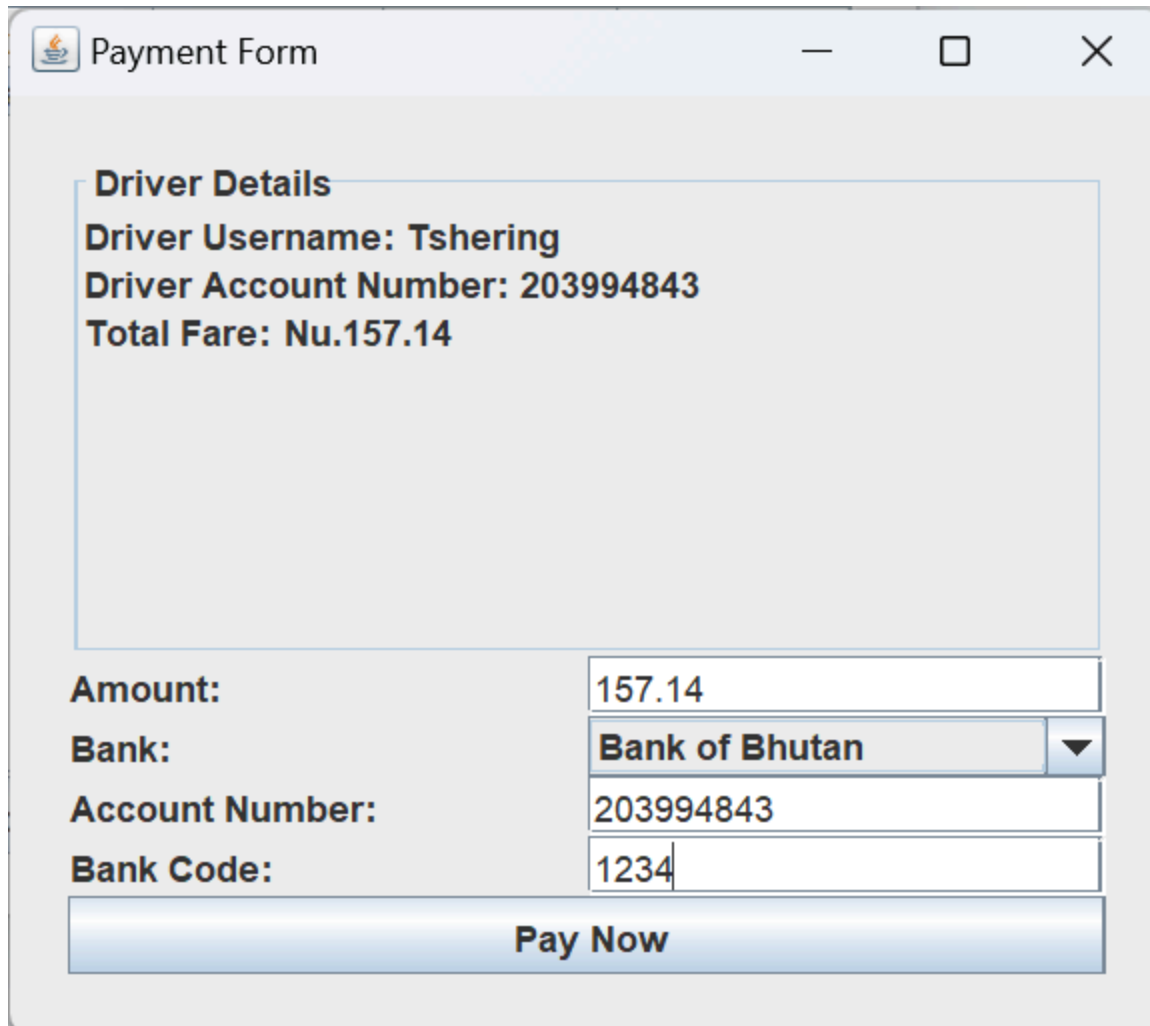
Passenger Home Page

Logout

Pickup Point	Destination	Date/Time	Taxi Type	Status	Driver Email	Action
GCIT College	Babesa	2024-05-29...	four seated	Accepted	tshering@g...	Pay Now

Book Now

After that, the user should click on the "Pay Now" button, and the Payment form will be displayed. In that form, the driver's account details and the Total fare will be displayed, and the user must fill up the form with the Amount, select the Bank type, enter the Account number and Bank Code.



A screenshot of a web application window titled "Payment Form". The window has a standard Windows-style title bar with a minimize button, a maximize button, and a close button. The main content area is divided into two sections. The top section, titled "Driver Details", contains the following information: "Driver Username: Tshering", "Driver Account Number: 203994843", and "Total Fare: Nu.157.14". The bottom section contains four input fields: "Amount:" with the value "157.14", "Bank:" with a dropdown menu showing "Bank of Bhutan", "Account Number:" with the value "203994843", and "Bank Code:" with the value "1234". Below these fields is a large blue button labeled "Pay Now".

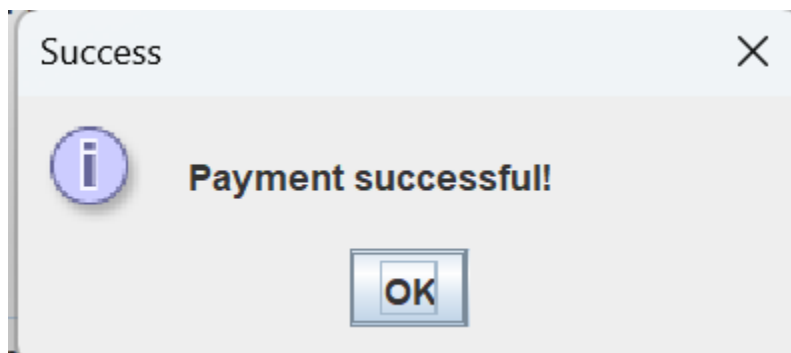
Driver Details	
Driver Username:	Tshering
Driver Account Number:	203994843
Total Fare:	Nu.157.14

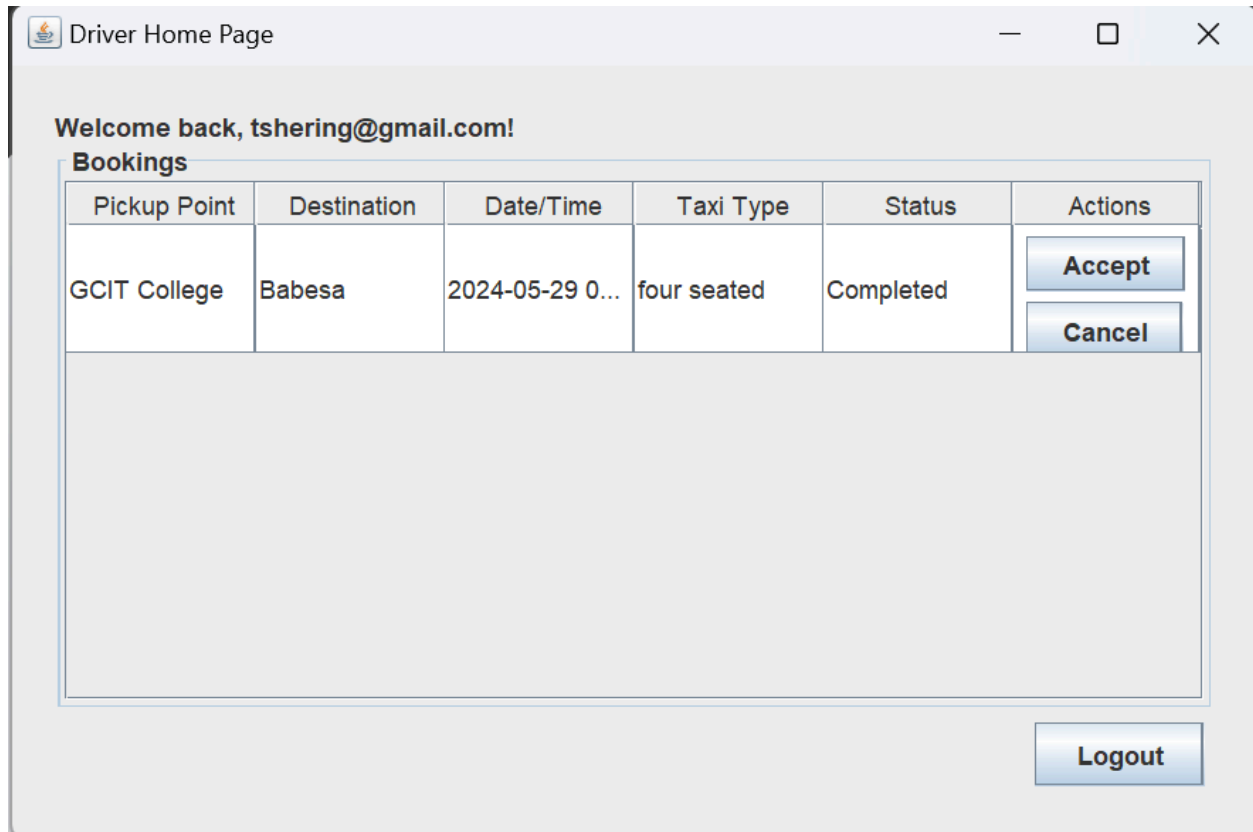
Amount:	157.14
Bank:	Bank of Bhutan
Account Number:	203994843
Bank Code:	1234

**Pay Now**

The passenger must click on the "Pay Now" button to pay for the fare, and a pop-up message will be displayed as shown below:



After that, the passenger will be directed to the Passenger Home Page, and the status will be changed to Completed, indicating that the passenger has completed the payment. Now, the Passenger Home Page is shown below:



## **7. Justifications for all the design patterns used**

1. Factory method: Promotes loose coupling by abstracting object creation based on user type, ensuring scalability without modifying existing code.
2. Singleton pattern: Ensures centralized control over authentication processes, optimizing resource management and system reliability by restricting instantiation to a single instance.
3. Observer pattern: Facilitates real-time updates on booking status changes, enhancing user engagement without the need for constant manual refresh.
4. State pattern: Encapsulates state-specific behavior, promoting modularity and code readability while simplifying maintenance and evolution of the booking process.
5. Command pattern: Decouples request sender from receiver, enabling the addition of new commands and supporting undo/redo functionality without altering client code.
6. Strategy pattern: Allows dynamic selection of algorithms, enhancing flexibility and extensibility by separating algorithm implementations from the context class.

## **10. Challenges**

Developing an online taxi booking system can be quite challenging. One of the main difficulties is finding the right balance between creating components that can be reused and scaled and meeting the specific needs of the application. This often leads to conflicts between the ideal design and what works in practice. Additionally, the initial investment of time and resources can be significant, which might slow down the application's progress. Making sure that the framework and application components work together seamlessly, while keeping the code quality high to avoid issues that could affect functionality and user experience, requires careful planning, continuous communication between teams, and regular testing to ensure everything aligns well with the project's goals.

## **11. Conclusion**

Creating an online taxi booking system using Java and applying various design patterns shows how we can build a strong, adaptable, and user-friendly platform. We start by laying a solid foundation with a framework, which helps us add specific features to the application smoothly. By using these design patterns, we manage our resources efficiently, making it easy to add new features and keep everything running smoothly. This project sets a great example for future transportation software, highlighting the importance of combining smart design principles with practical ways to make things work. Even though there were challenges along the way, successfully finishing this project proves that we can create advanced and reliable online taxi booking systems.