Final Report on Taxi Booking System

University of Bedfordshire

CIS093-1 - MATHEMATICS AND CONCEPTS FOR COMPUTATIONAL THINKING

Student id: Krijen Shahi (2432413)

Contents

Introduction

Comparing this with some years back ride-sharing apps have revolutionized the archaic process of booking taxi and negotiating over fares. In recent years, the transportation industry has witnessed a significant transformation with the advent of mobile and web-based booking systems (Lee, S., Lee, W., Vogt, C.A. and Zhang, Y, n.d.). This changing trend forced us to create a reliable and easy-to-use taxi booking software. Our project was to develop a solution that provides an easy way for customers to book rides, drivers to manage their trips and the admin panel interfaces everything.

In this project, a desktop-based taxi booking management software is adopted, which supports interactions among three user types: customers, drivers, and administrators. The system uses the concept of object-oriented programming and the best practices of the software engineering fields to maintain, scale and be reliable. (Sommerville, 2011)

This report describes how the system has been designed and built, how the test cases are done, and what I have learned along the way including the hurdles I faced and how I overcome through them.

Objective

The main objective of this project was to design and build a well performing taxi booking system which supports three different types of users i.e customer, admin and driver. I have made sure that the system worked well for all three user types that are customers, drivers, and administrators, each with their own responsibilities.

One of my main objectives was to build a well functional, role-based system (Rayle, L., Dai, D., Chan, N., Cervero, R. and Shaheen, S., n.d.). Using Python and object-oriented programming, I have developed a software where each user type has a different type of roles and features which helped in keeping the system organized and made it easier to understand how different users interact with it.

I have also worked on applying problem-solving and computational thinking throughout the development process which involved breaking the system into smaller parts, manageable parts, like separating user management from ride-booking logic. I reused common patterns, where

possible (e.g., having the same login structure for all users), I also tried to hide some of the complex processing that was needed, like password security, in order to keep main code clean and readable.

Al last, I have been involved to ease the system flow. I have tested the entire system, user registration, secure entry into the system, and booking of rides as well as assigning the driver to the user to make sure that the user has a good user experience. In general, this was to provide a smooth and trusted experience to everyone using the system.

Task Description

The project requirement was to create a taxi booking with different three user types with their respective roles

- Customer Functions: Can Register their details, book a taxi, view, and cancel bookings.
- Driver Functions: Log in to view assigned trips.
- Administrator Functions: Log in, view all bookings, assign drivers to customer bookings, and prevent overlapping bookings.

The system is to be implemented in Python using a text-based interface; hence, the users will interact with the system through the selection of options from a menu. The project also involves the use of data persistence using text files: data for customers, trips, and driver assignments will be stored. Optional functionality includes GUI for enhanced user interaction.

Requirement Analysis

As per our requirement to develop taxi booking system. I have listed down functional and non-functional requirement below.

Functional Requirement

Table 1 Functional Requirement

Req No	Requirement	Priority
FR1	Customer must be able to register by providing their basic details like name, address, phone number and email address	MUST
FR2	Customer must be able to book a taxi by entering their pickup location, drop-off location with date and time.	MUST
FR3	Customer must be able to view their booking and cancel them as well.	MUST
FR4	Driver must be able to login and see the details of their assigned rides.	MUST
FR5	Admin user must be able to login and view all the bookings, assign the free drives to a customer booking.	MUST
FR6	Admin must ensure that bookings assignment is not overlapped to drivers.	MUST
FR7	Admin should be able to view and manage customer and trip data.	SHOULD
FR8	The system should support the creation and editing of customer basic info.	SHOULD
FR9	The system should provide summary of bookings to the customer.	SHOULD
FR10	Driver should be able to confirm or reject the assigned trips.	SHOULD

Non-Functional Requirement

Table 2 Non-Functional Requirement

Req No	Requirement	Priority
NFR1	The system must load user data within 2 seconds of startup.	MUST
NFR2	The system must securely store user data, using encryption for sensitive information such as passwords.	MUST
NFR3	The system must be able to handle at least 1000 active users without performance degradation.	SHOULD
NFR4	The system must provide clear and user-friendly feedback in case of invalid inputs.	MUST
NFR5	The system should have a simple, text-based interface that is easy to navigate for all user types.	MUST

Usability Requirements

Table 3 Usability Requirement

Req No.	Requirement	Priority*
1	The interface should follow a consistent design across all screens.	MUST
2	Booking forms should be logically organized and easy to complete.	MUST
3	The system should provide clear navigation between customer, driver, and admin functions.	MUST
4	Text, labels, and messages should be clear, concise, and free of spelling or grammatical errors	MUST
5	Color schemes should be appropriate for a taxi booking application	SHOULD
6	Important actions (book ride, cancel booking, assign driver) should be clearly distinguishable	MUST
7	The system should provide feedback for user actions (success, error, or warning messages)	SHOULD
8	Data entry fields should include validation where appropriate (login, booking details, registration)	MUST
9	The layout should remain consistent across different screen sizes and resolutions	SHOULD

Use Case Diagrams

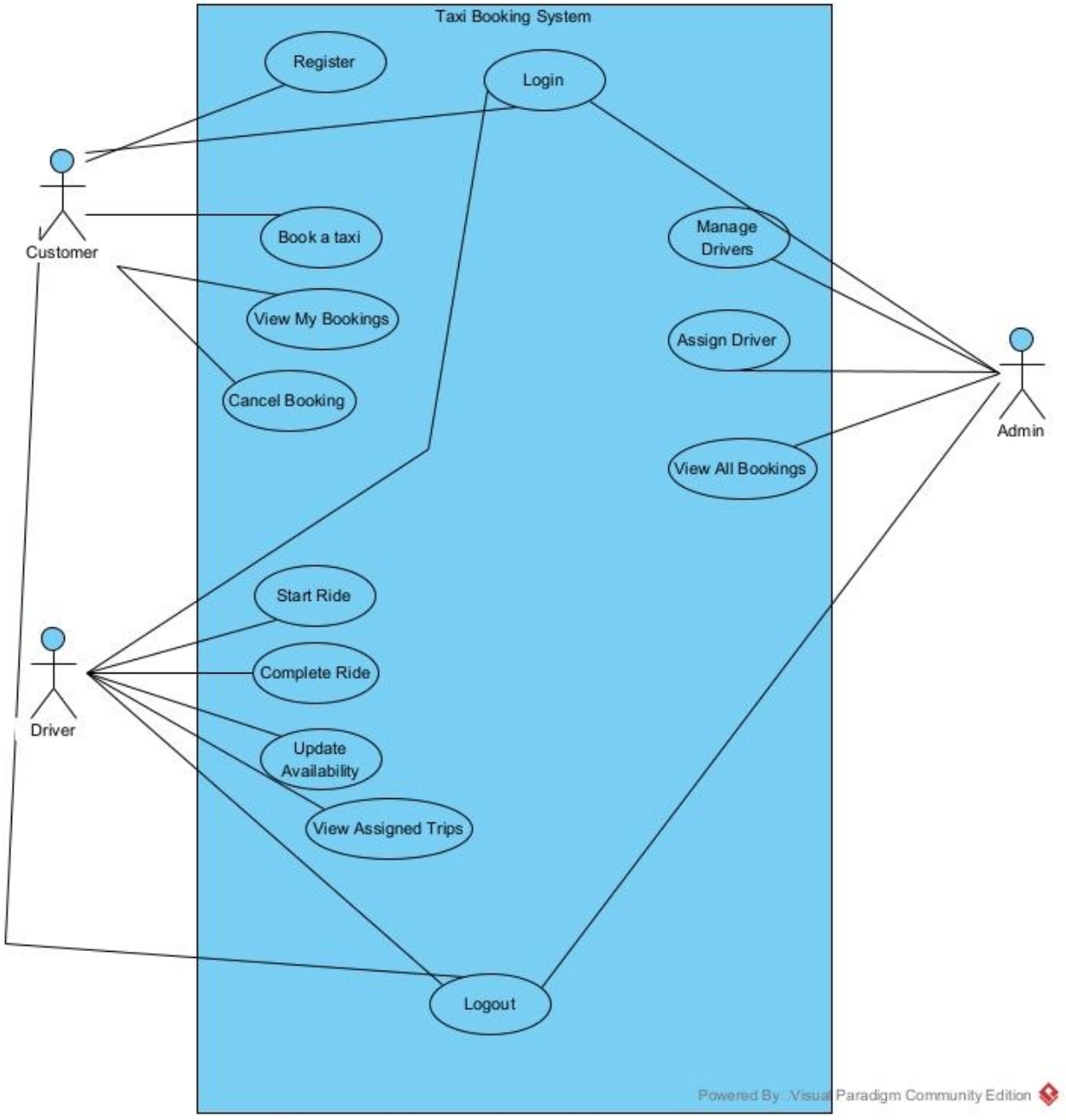

Figure 1 Sea Level

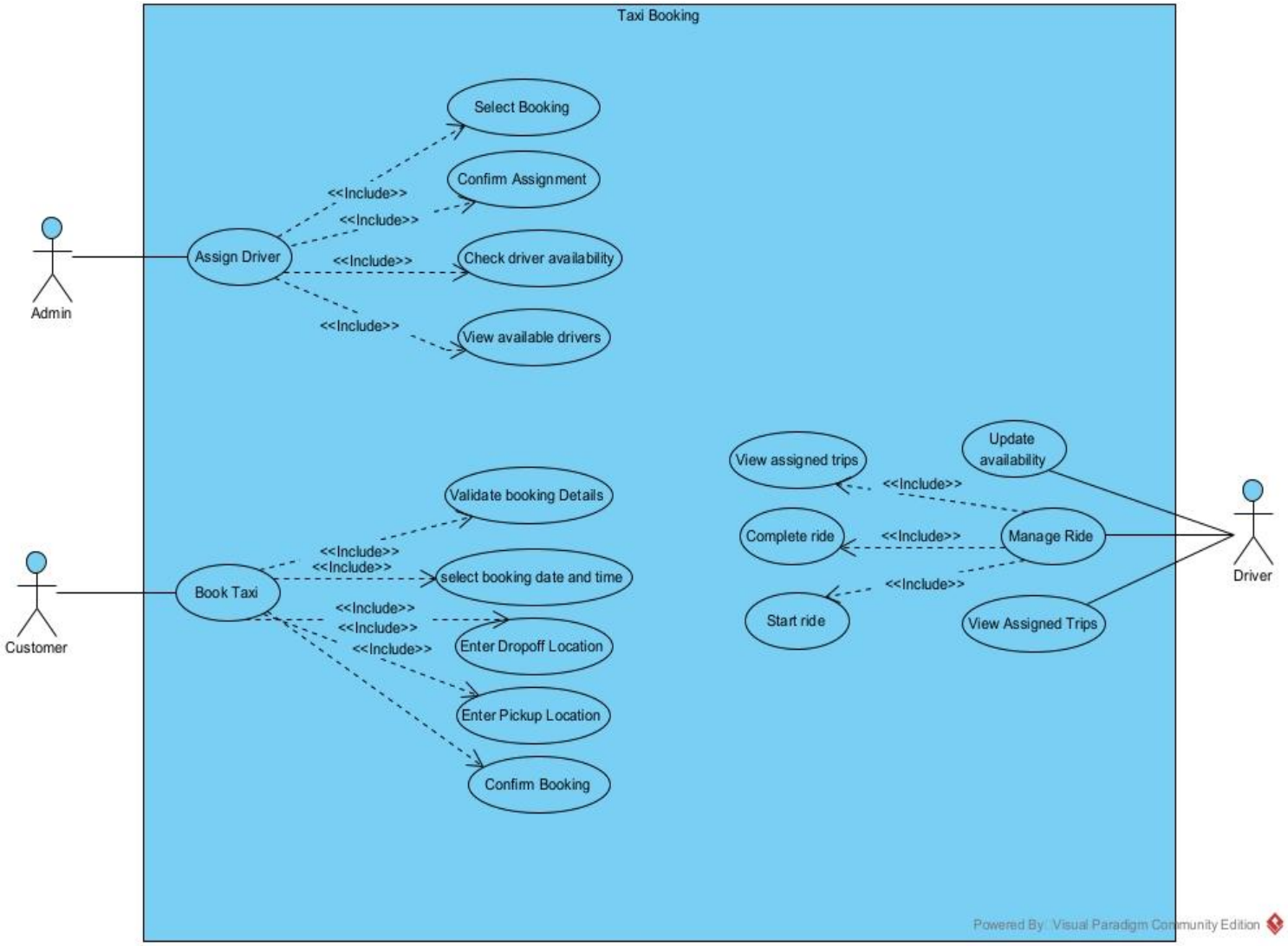

Figure 2 Fish level

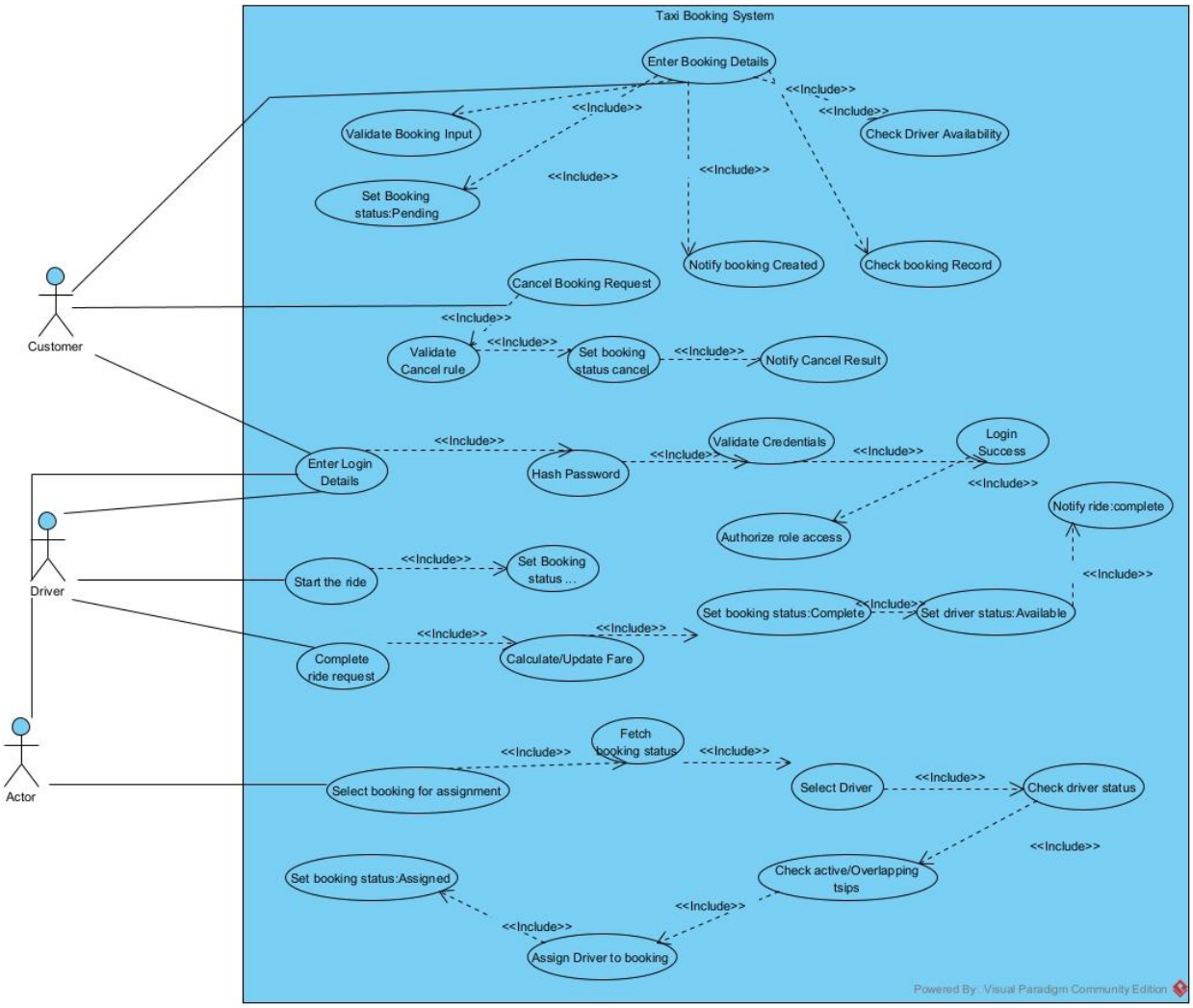

Figure 3 Calm level Use Case Diagram

Activity Diagrams

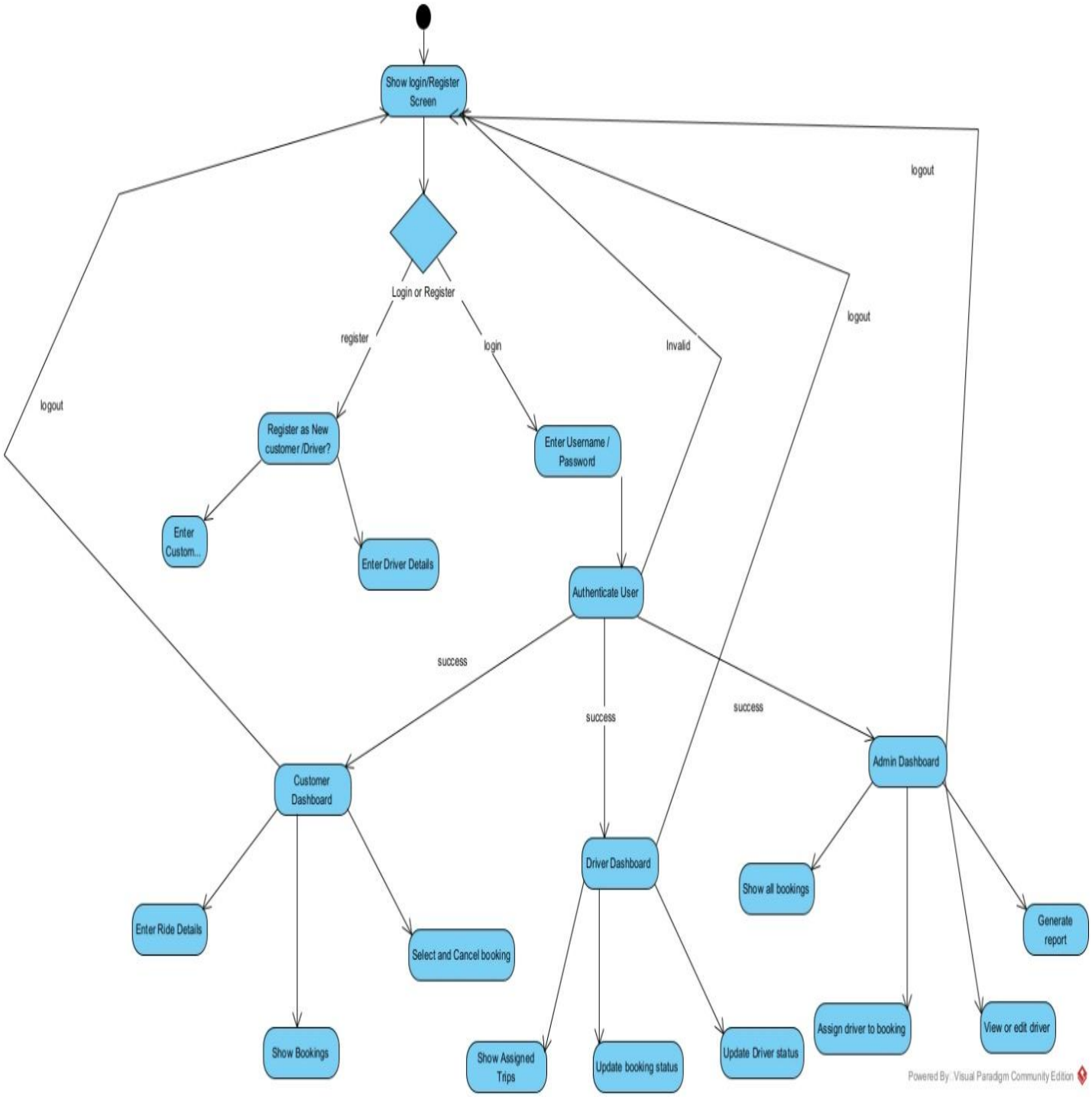

Figure 4 Activity Diagram

Class Diagram

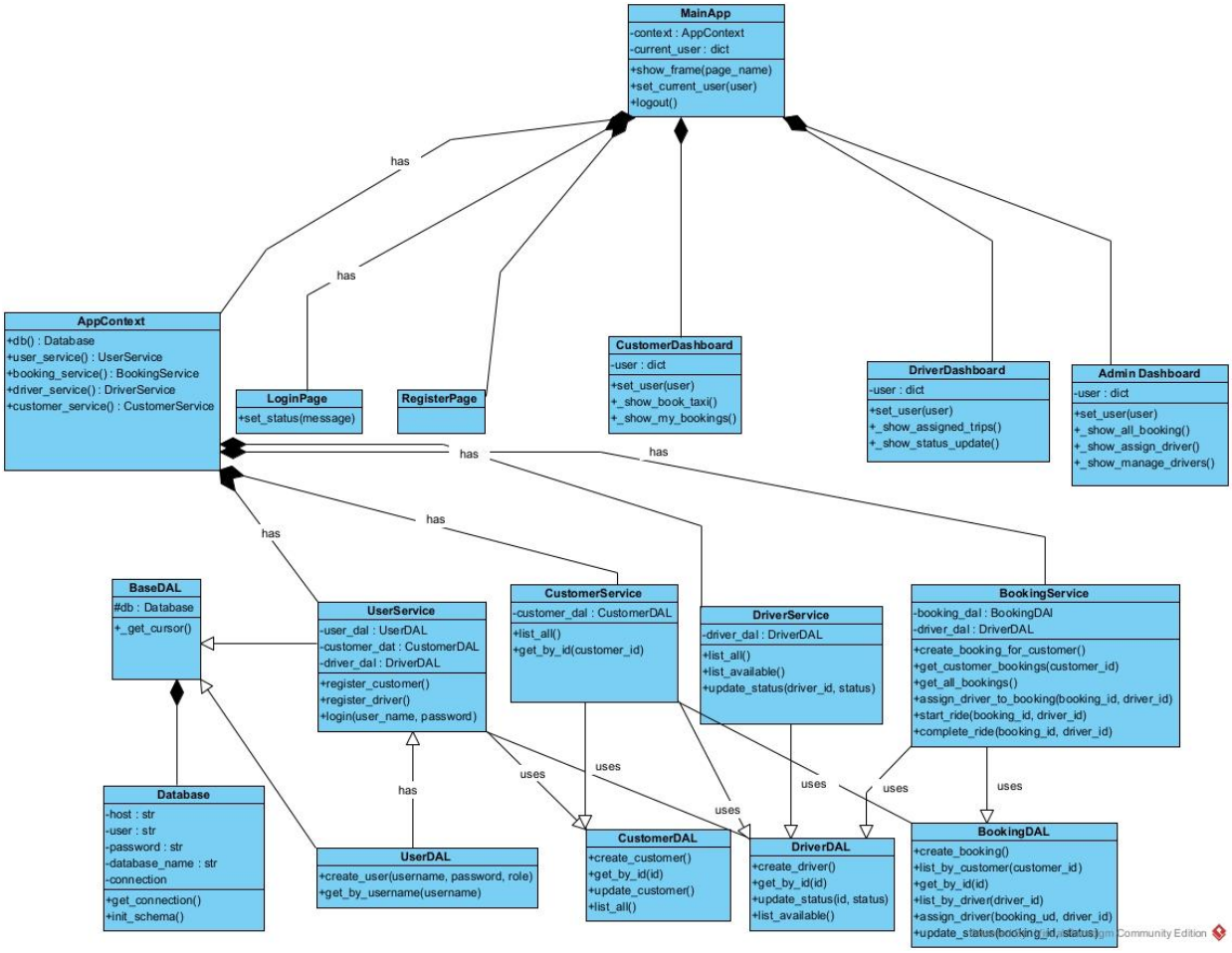

Figure 5 Class Diagram

Database Design ERM/ERD

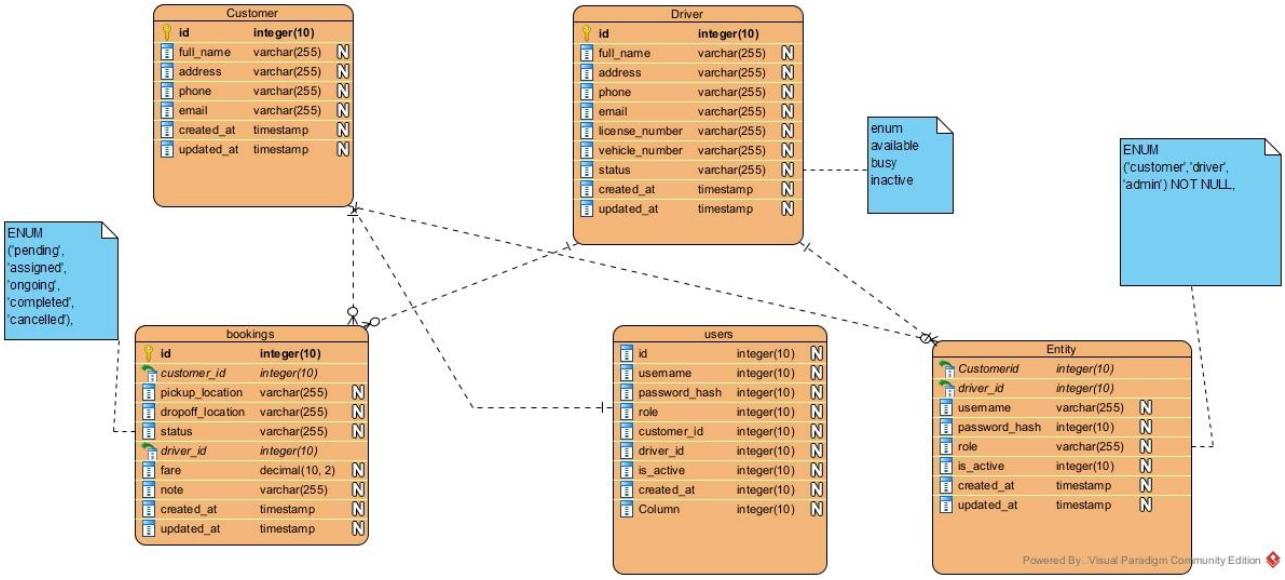

Figure 6 ERM Diagram

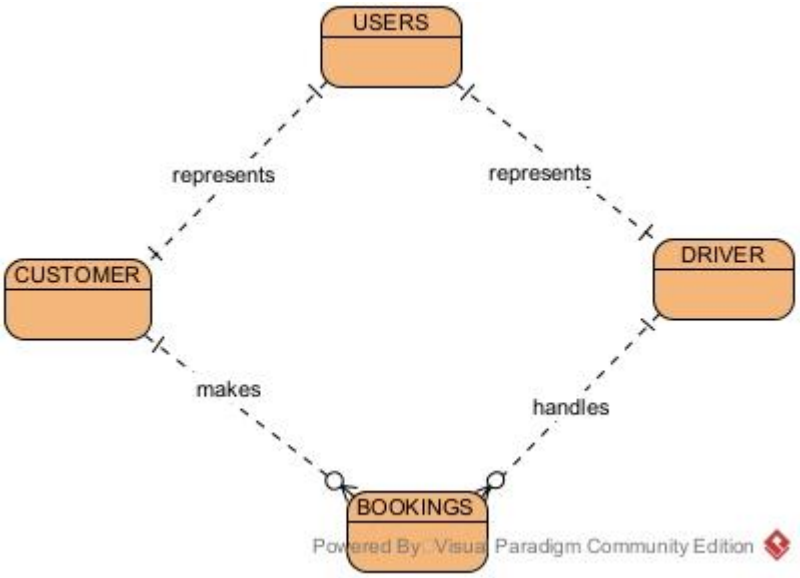

Figure 7 ERD

Physical Database Design

Skeleton Tables

• customers

Field	Type	Null	Key	Default	Extra
id	int	NO	PRI	NULL	auto_increment
full_name	varchar(100)	NO		NULL	
address	varchar(255)	YES		NULL	
phone	varchar(20)	YES		NULL	
email	varchar(100)	YES	UNI	NULL	
created_at	timestamp	YES		CURRENT_TIMESTAMP	DEFAULT_GENERATED
updated_at	timestamp	YES		CURRENT_TIMESTAMP	DEFAULT_GENERATED on update CURRENT_TIMESTAMP

Figure 8 Customer Table

• drivers

Field	Type	Null	Key	Default	Extra
id	int	NO	PRI	NULL	auto_increment
full_name	varchar(100)	NO		NULL	
address	varchar(255)	YES		NULL	
phone	varchar(20)	YES		NULL	
email	varchar(100)	YES	UNI	NULL	
license_number	varchar(50)	YES		NULL	
vehicle_number	varchar(50)	YES		NULL	
status	enum('available','busy','inactive')	YES		available	
created_at	timestamp	YES		CURRENT_TIMESTAMP	DEFAULT_GENERATED
updated_at	timestamp	YES		CURRENT_TIMESTAMP	DEFAULT_GENERATED on update CURRENT_TIMESTAMP

Figure 9 Driver Table

• bookings

Field	Type	Null	Key	Default	Extra
id	int	NO	PRI	NULL	auto_increment
customer_id	int	NO	MUL	NULL	
pickup_location	varchar(255)	NO		NULL	
dropoff_location	varchar(255)	NO		NULL	
pickup_datetime	datetime	NO		NULL	
status	enum('pending','assigned','ongoing','completed',...	YES		pending	
driver_id	int	YES	MUL	NULL	
fare	decimal(10,2)	YES		NULL	
notes	text	YES		NULL	
created_at	timestamp	YES		CURRENT_TIMESTAMP	DEFAULT_GENERATED
updated_at	timestamp	YES		CURRENT_TIMESTAMP	DEFAULT_GENERATED on update CURRENT_TIMESTAMP

Figure 10 Bookings table

• users

Field	Type	Null	Key	Default	Extra
id	int	NO	PRI	NULL	auto_increment
username	varchar(50)	NO	UNI	NULL	
password_hash	varchar(255)	NO		NULL	
role	enum('customer','driver','admin')	NO		NULL	
customer_id	int	YES	MUL	NULL	
driver_id	int	YES	MUL	NULL	
is_active	tinyint(1)	YES		1	
created_at	timestamp	YES		CURRENT_TIMESTAMP	DEFAULT_GENERATED
updated_at	timestamp	YES		CURRENT_TIMESTAMP	DEFAULT_GENERATED on update CURRENT_TIMESTAMP

Figure 11 Users table

Data Dictionary

Table 4 Data Dictionary of Customer Table

Customers							
Description: This table stores customer information used for taxi bookings.							
Field Name	Data Type	Length	Index	Null	Default value	Validation Rule	Description
customer_id	Int		PK	No	Auto_increment		Unique customer identifier
full_name	Varchar	100		Yes			Customer full name
address	Varchar	255		Yes			Customer address
phone	Varchar	20		Yes			Customer contact
Email	Varchar	100	UQ	Yes			Customer email address
created_at	timestamp			Yes	CURRENT_ TIMESTAMP		Record creation date
updated_at	timestamp				CURRENT_ TIMESTAMP		Record creation date

Table 5 Data Dictionary of Drivers table

Drivers							
Description: This table stores driver details and availability status.							
Field Name	Data Type	Length	Index	Null	Default value	Validation Rule	Description
driver_id	Int		PK	No	Auto_increment		Unique driver identifier
full_name	Varchar	100		Yes			Driver full name
address	Varchar	255		Yes			Driver address
phone	Varchar	20		Yes			Driver contact
email	Varchar	100	UQ	Yes			Driver email address
license_number	Varchar	50		Yes			Driver License number
Vehicle_number	Varchar	50		Yes			Vehicle registration number
Status	Enum			Yes	Available	Available, busy, inactive	Driver availability status
Created_at	Timestamp			Yes	CURRENT_TIMESTAMP		Record creation date

Table 6 Data Dictionary of Bookings Table

Bookings							
Description: This table stores taxi booking details made by customers.							
Field Name	Data Type	Length	Index	Null	Default value	Validation Rule	Description
Booking_id	Int		PK	No	Auto_increment		Unique booking identifier
Customer_id	Int		FK	No		Must existing	Customer making the booking
address	Varchar	255		Yes			Driver address
phone	Varchar	20		Yes			Driver contact
email	Varchar	100	UQ	Yes			Driver email address
license_number	Varchar	50		Yes			Driver License number
Vehicle_number	Varchar	50		Yes			Vehicle registration number
Status	Enum			Yes	Available	Available, busy, inactive	Driver availability status
Created_at	Timestamp			Yes	CURRENT_TIMESTAMP		Record creation date
Updated_at	Timestamp			Yes	CURRENT_TIMESTAMP		Record update date

Table 7 Data Dictionary of Users Table

Users							
Description: This table stores system login credentials for customers, drivers, and administrators.							
Field Name	Data Type	Length	Index	Null	Default value	Validation Rule	Description
user_id	Int		PK	No	Auto_ increment		Unique user identifier
username	Varchar	50	UQ	No		Unique	Login Username
password_ hash	Varchar	255		No		Encrypted	User Password
role	Enum	20		No		Customer, driver, admin	User role
customer_id	int		FK	Yes		References Customer	Linked customer id
driver_id	int		FK	Yes		References Driver	Linked driver id
is_active	Boolean			Yes	1		Account Status
created_at	Enum			Yes	CURRENT_ TIMESTAMP		Record creation date

Implementation (OOP in the context of the project)

Taxi booking System has completed used an object-oriented, layered architecture which separates databased access, business logic, and user interfaces. This process helped on code maintainability, reusability, and clarity.

The system is divided into four layers: Data Access Layer (DAL), Business Logic (Service) Layer, User Interface (UI) Layer, and an Application Context.

Data Access Layer is a layer that wraps all the interactions with the database. The MySQL connection and schema initiation is handled by a central Database class. Entity-specific classes including UserDAL, CustomerDAL, DriverDAL and BookingDAL share a common BaseDAL which enables the re-use of logic to handle the cursor. This illustrates encapsulation and inheritance, and abstracting lower-level SQL out of upper ones.

The Service Layer works with business rules and workflows. Classes like UserService and BookingService coordinate multiple DAL operations to perform tasks like user authentication, booking creation, driver assignment, and ride lifecycle management. These services provide high-level abstractions to the UI and encapsulate validation and rule enforcement logic, illustrating abstraction and composition.

The UI Layer is implemented using Tkinter, where each screen (e.g., login, customer dashboard, driver dashboard, admin dashboard) is represented as a class inheriting from tk. Frame. The MainApp class controls screen navigation and session state. This use of inheritance and role-based screen behavior demonstrates polymorphism and encapsulation in the presentation layer.

The AppContext class initializes and shares the database and service objects across the application, simplifying dependency management and ensuring consistent access to core resources.

Overall, the system effectively applies encapsulation, abstraction, inheritance, and polymorphism to achieve a clean separation of concerns. This object-oriented design makes the Taxi Booking System easy to understand, maintain, and extend with additional features.

Computational Thinking

Computational thinking is the mental skill set used to solve problems in a way that can be carried out by a computer or, more broadly, to understand and structure complex processes logically. (Wing, 2006). The project of Taxi Booking System uses the Computational Thinking in the whole project. This system implies all four aspects that must be covered in Computational Thinking.

Application of computational thinking within the Taxi Booking System can enable the system to produce an effective and properly arranged solution. The system demonstrates the main principles of computational thinking which are decomposition, abstraction, pattern recognition, and algorithmic thinking. This is achieved by decomposition whereby the system is broken down into layers where Data Access Layer, Service Layer, and User Interface Layer are the most suitable classes with some of them being user management, bookings, and driver assignment classes.

One abstraction technique, separation of database operations, business logic and user interaction between layers, facilitates the concealment of details of implementation and therefore promotes clarity and maintainability. The repeated CRUD operations, validation rules and customer, driver and administrator role-based behavior can be used to prove pattern recognition. The procedure of user authentication, creation of bookings, allocation of drivers and Ride completion is done step by step, and is deemed to be algorithmic and ensures the correct and reliable work of the system.

UI Design

User interface design plays a critical role in system usability and user satisfaction (Norman, 2013).

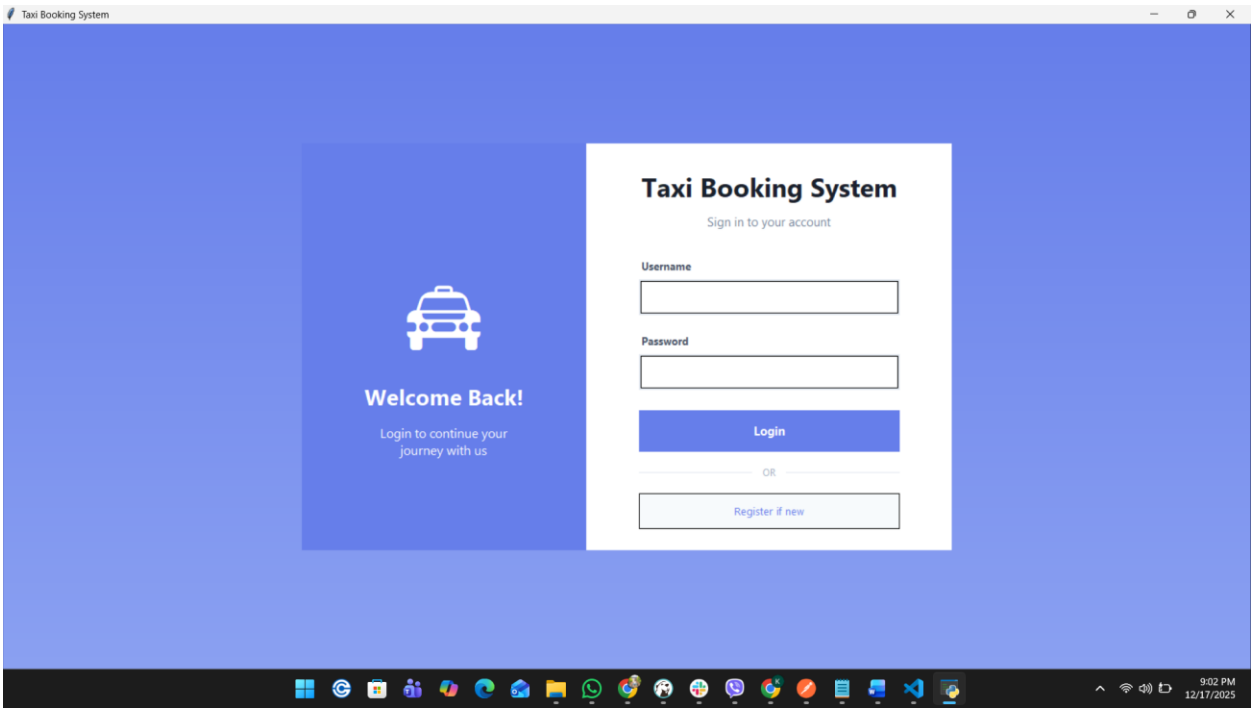

Figure 12 Login UI

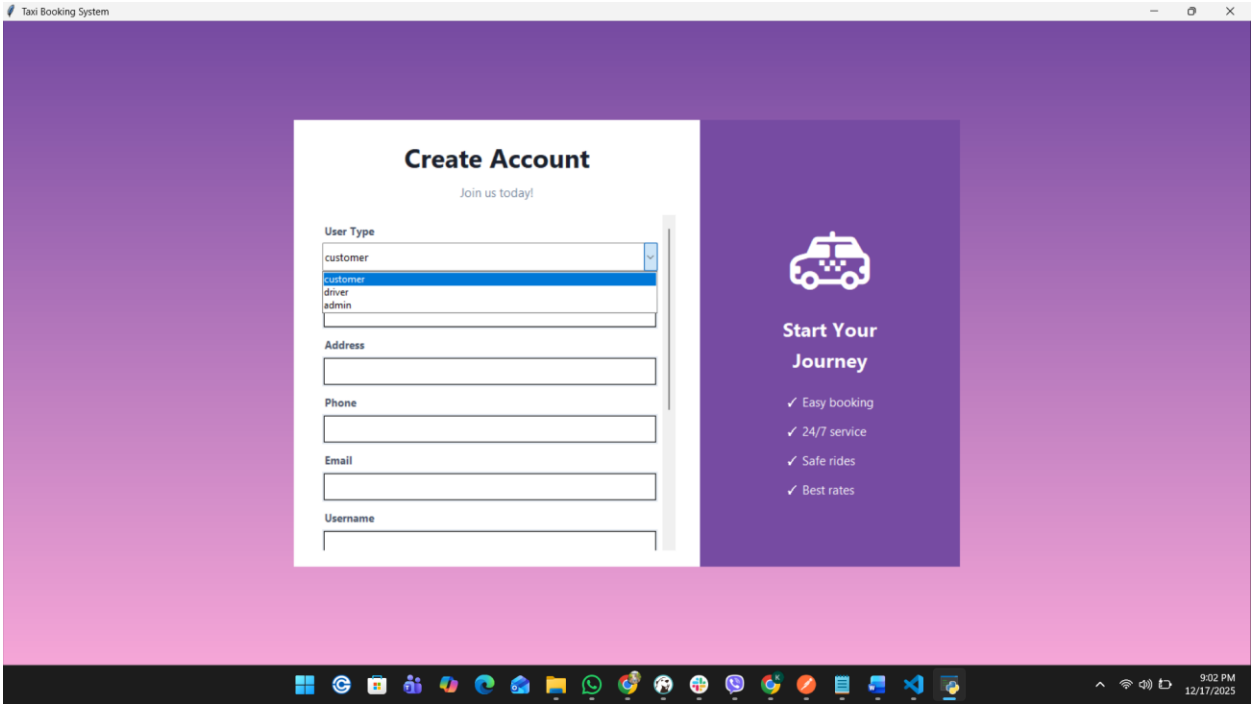

Figure 13 Registration Page UI

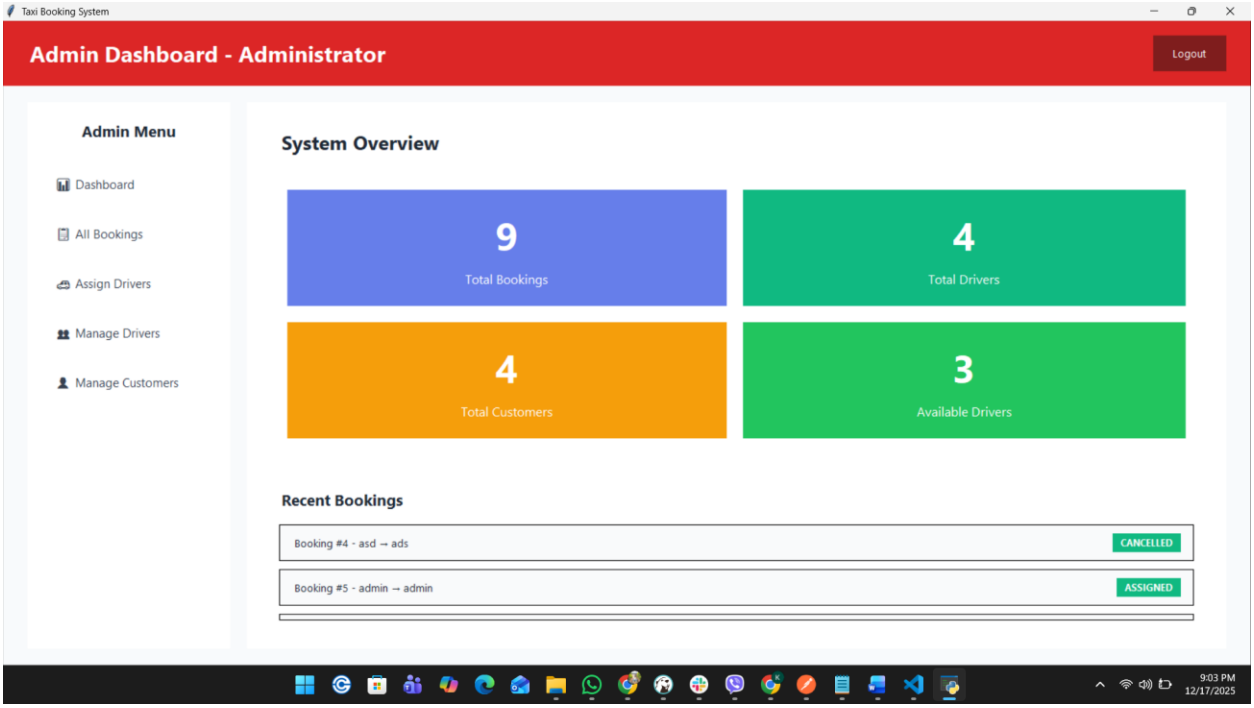

Figure 14 Admin Dashboard UI

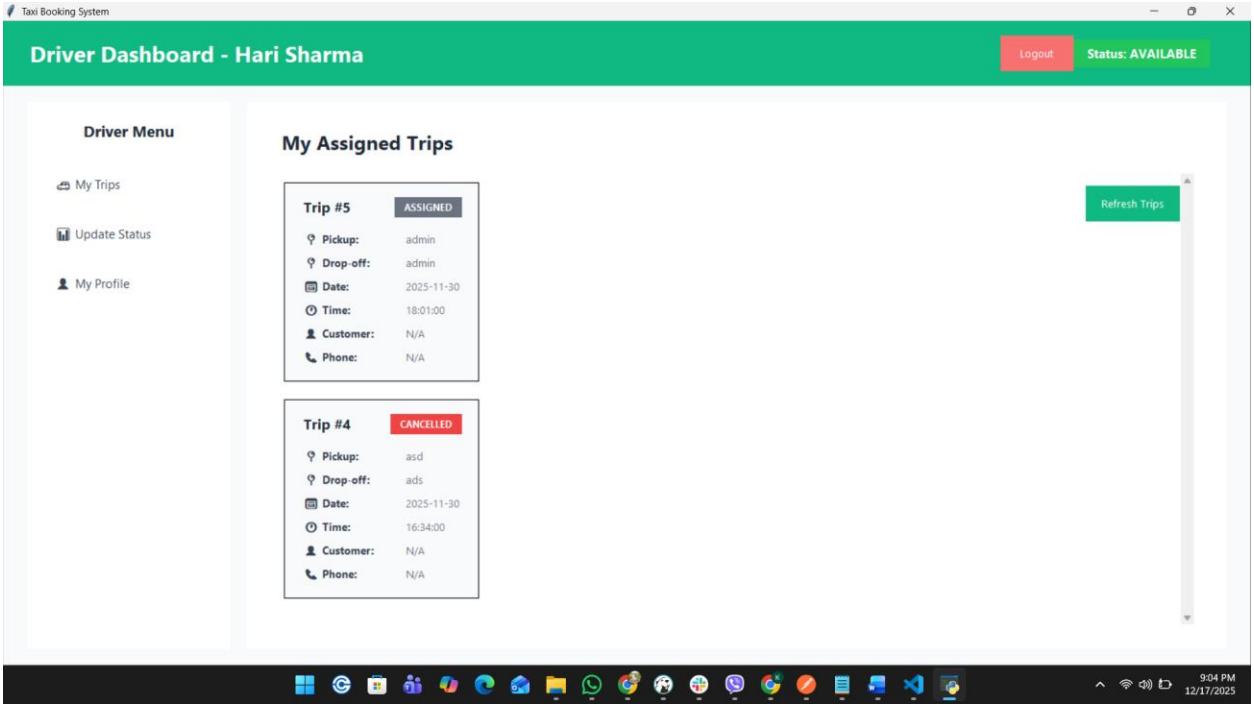

Figure 15 Driver Dashboard UI

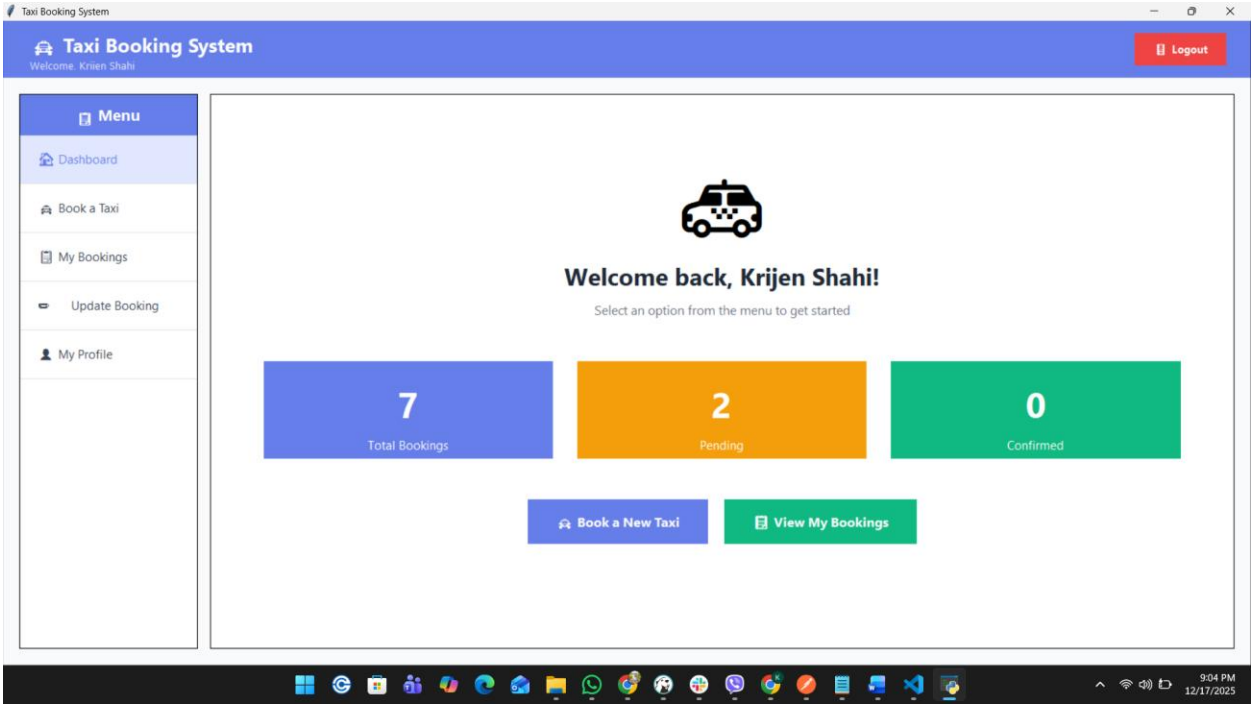

Figure 16 Customer Dashboard UI

Testing

Test cases were designed following best practices outlined by (Myers, G.J., Badgett, T., Thomas, T.M. and Sandler, C., 2004):

Table 8 Test Cases

Test No.	Date	Purpose of Testing	Input Data	Expected Output	Actual Output	Pass/Fail	Screenshot
T-1	Dec -14	Verify admin login functionality	Username: admin Password: admin	Redirect to admin portal	Dashboard loaded successfully	Pass	
T-2	Dec -14	Test invalid login rejection	Username: admin Password: Test	Error message "Invalid username or password"	Error message displayed	Pass	
T-3	Dec -14	Verify customer registration	Full Name: Suman Thapa Address: Nepal Phone: 9860165286 Email: suman1@gmail.com Username: suman1 Password: Suman	Registration successful as a customer	Customer created successfully	Pass	
T-4	Dec -14	Validate already email exists	Full Name: Suman Thapa Address: Nepal Phone: 9860165286 Email: suman@gmail.com Username:	A customer with this email already exists	Validated successfully	Pass	

T-5	Dec -14	Validate user name exists	Full Name: Suman Thapa Address: Nepal Phone: 9860165286 Email: suman12@gmail.com Username: suman Password: Suman	User name is already taken	Validated successfully	Pass	
T-6	Dec -14	Verify driver registration	User Type: Driver Full Name: Jivan Gurung Address: Satdobato Phone: 9860827827 Email: jivangurung@gmail.com License Number: 12399812 Vehicle Number: 89132 Username: jivan	Driver created successfully	Registration successful as driver.	Pass	
T-7	Dec -14	Create taxi booking	Pickup Location: Satdobato Drop-off Location: Thamel Pickup	Booking created successfully	Booking successfully created	Pass	

T-8	Dec -14		Validate booking without entry of mandatory field	Validated successfully	All fields are required.	Pass	
T-9	Dec -14	Assign driver to booking (Admin)	Booking Id: 13 Driver: Jivan Gurung (Vehicle no 89132)	Driver assigned successfully	Assignment successful	Pass	
T-10	Dec -14	Prevent driver double booking	Booking ID: 11 Driver: Jivan Gurung (Vehicle no 89132)	Driver has already active booking	Failed to assign driver: This driver already has an active booking. They must complete the current ride before a new one can be assigned.	Pass	
T-11	Dec -14	Cancel booking by customer	Booking id 8	Booking cancelled successfully	Booking cancelled	Pass	
T-12	Dec -14	Cancel the ride with cancel status	Booking id 6	Validated successfully	Failed to cancel booking: Cannot cancel a booking with status 'Cancelled'	Pass	
T-13	Dec -14	Validate logout functionality	Click Logout	Redirect to Login Page	Login page displayed	Pass	

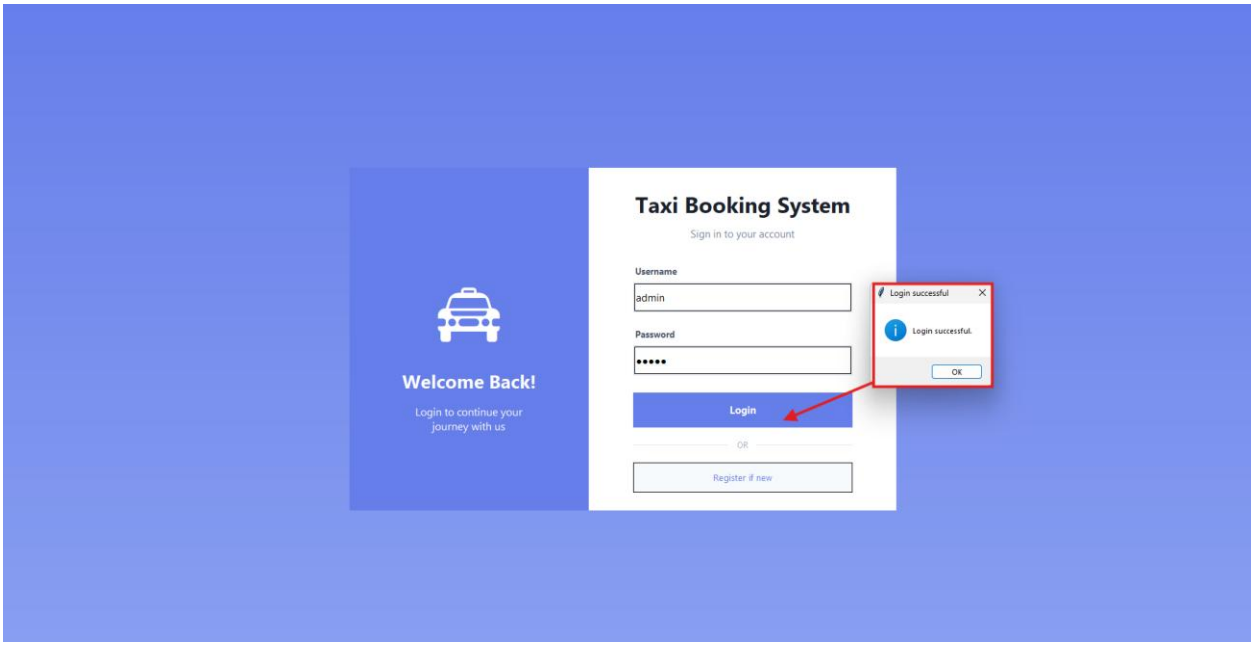

Figure 17 T-1 Verify Admin Login Functionality

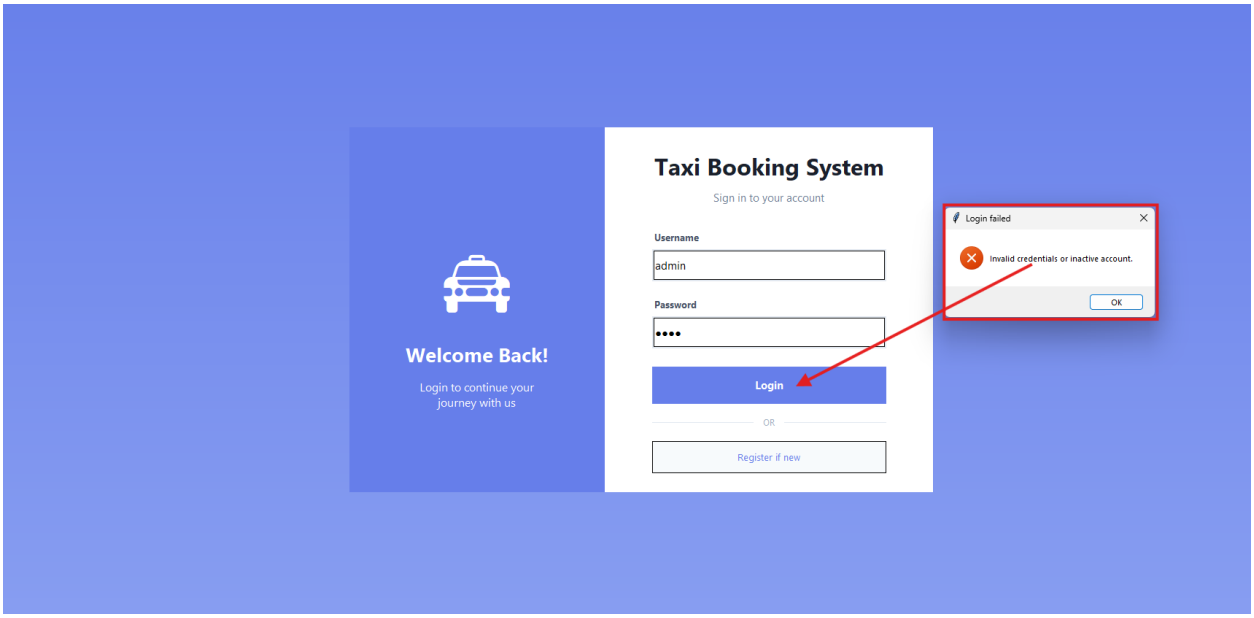

Figure 18 T-2 Test invalid login rejection

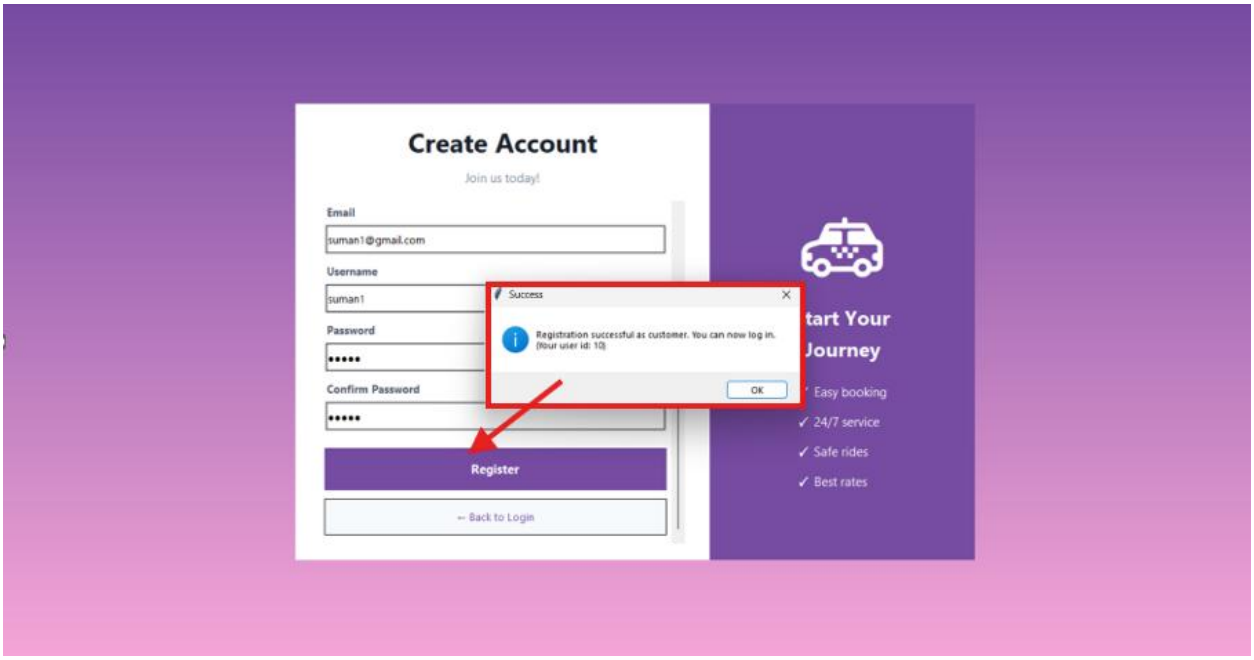

Figure 19 T-3 Verify customer registration

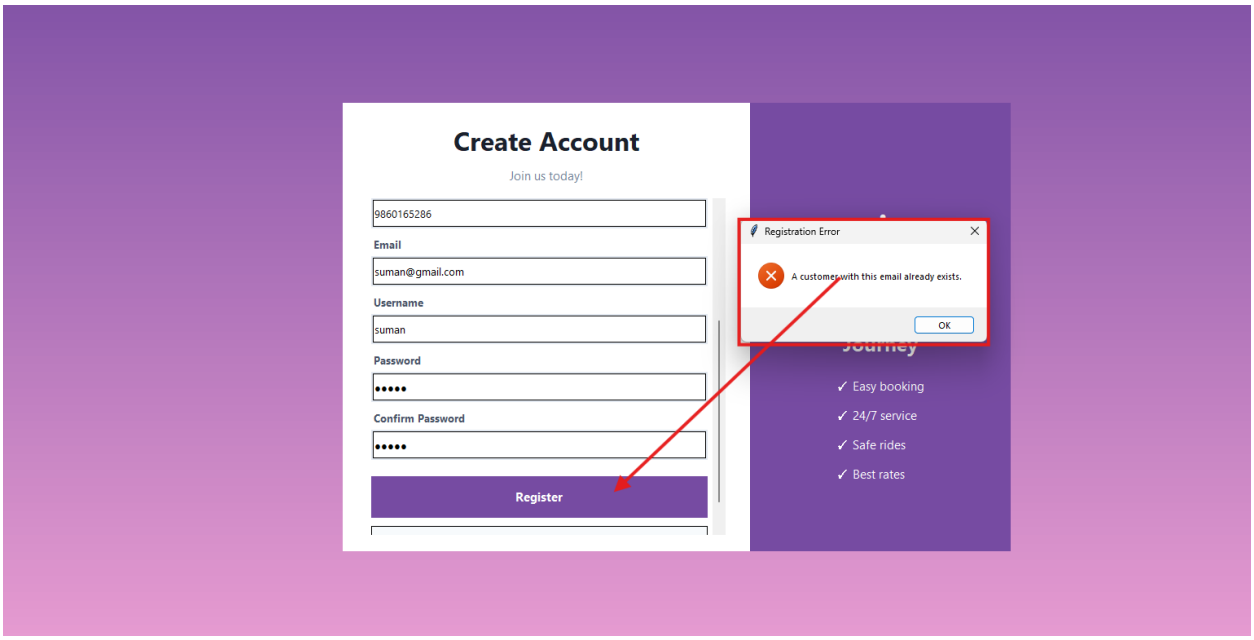

Figure 20 T-4 Validate already email exists

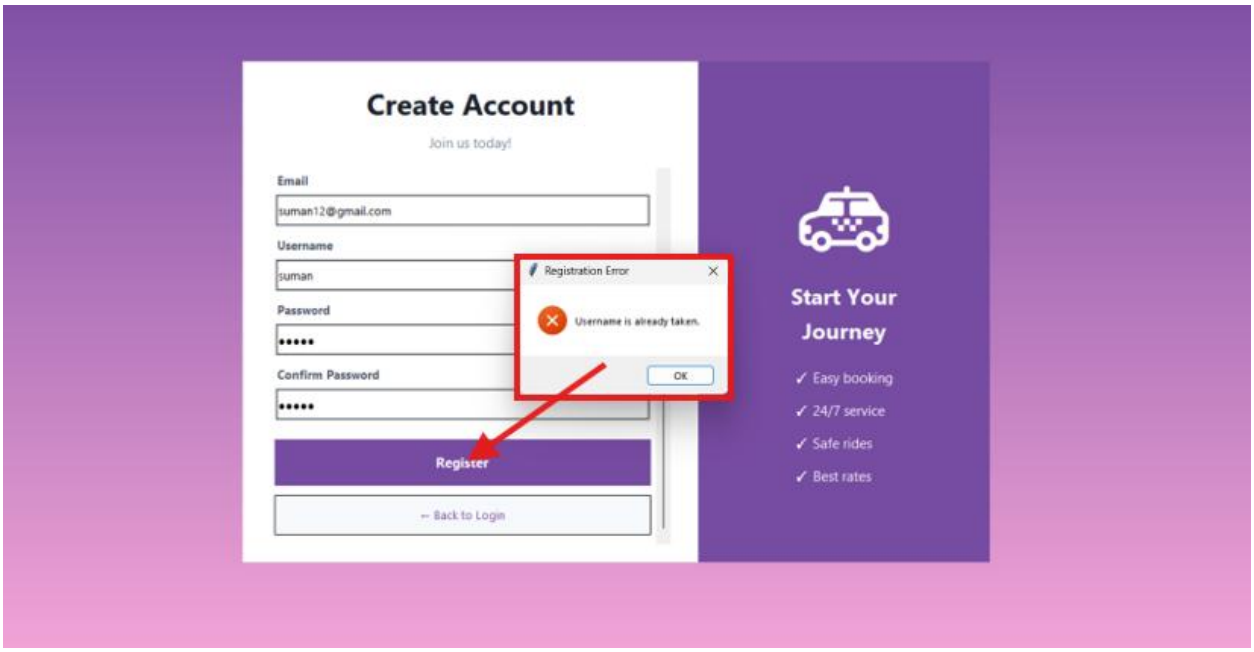

Figure 21 T-5 Validate user name exists

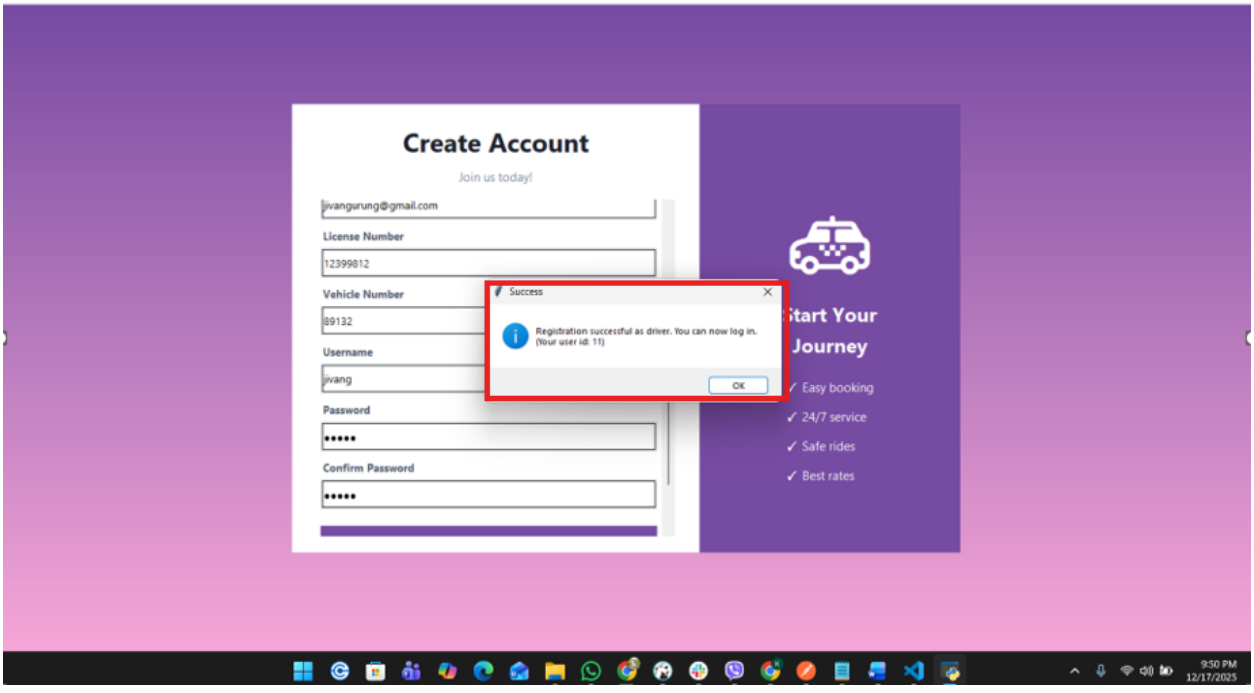

Figure 22 T-6 Verify driver registration

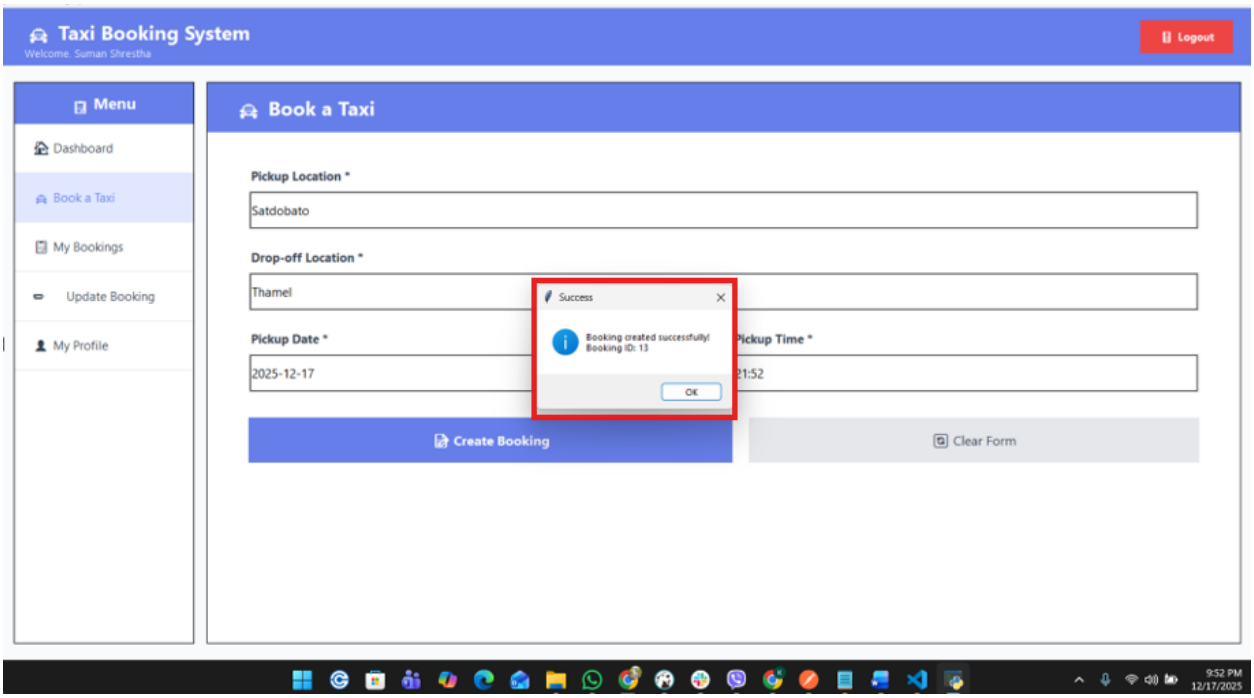

Figure 23 T-7 Create taxi booking

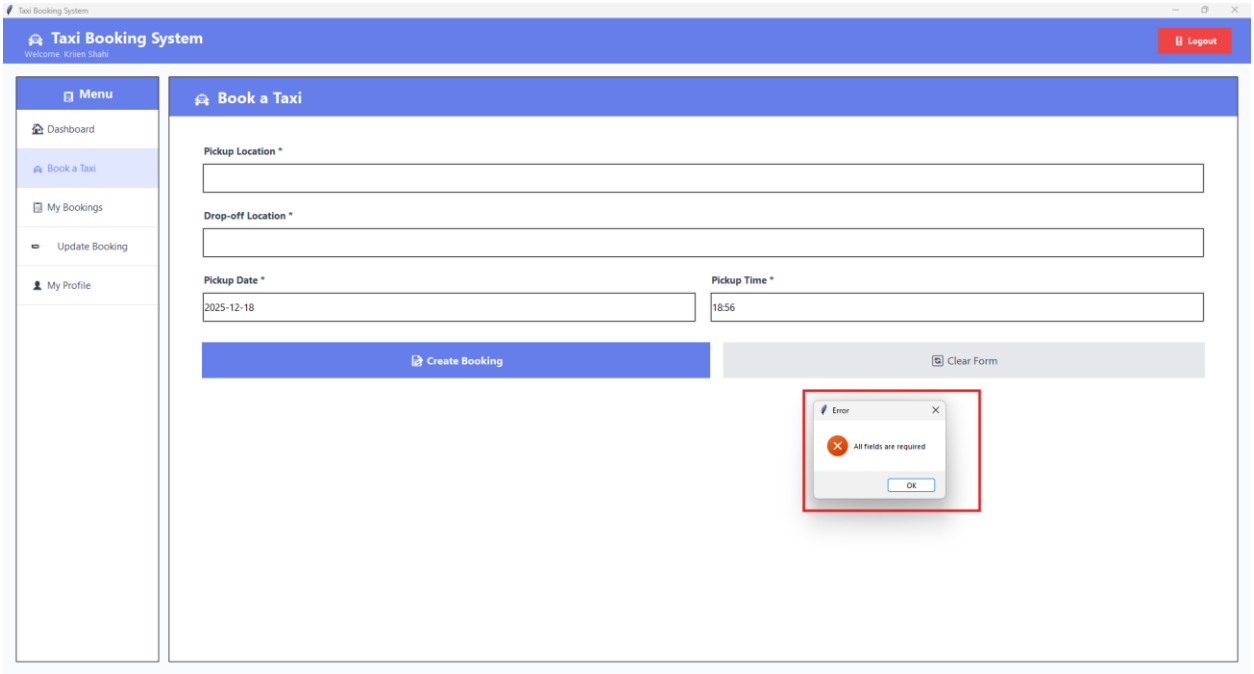

Figure 24 T-8 Validate booking without entry of mandatory field

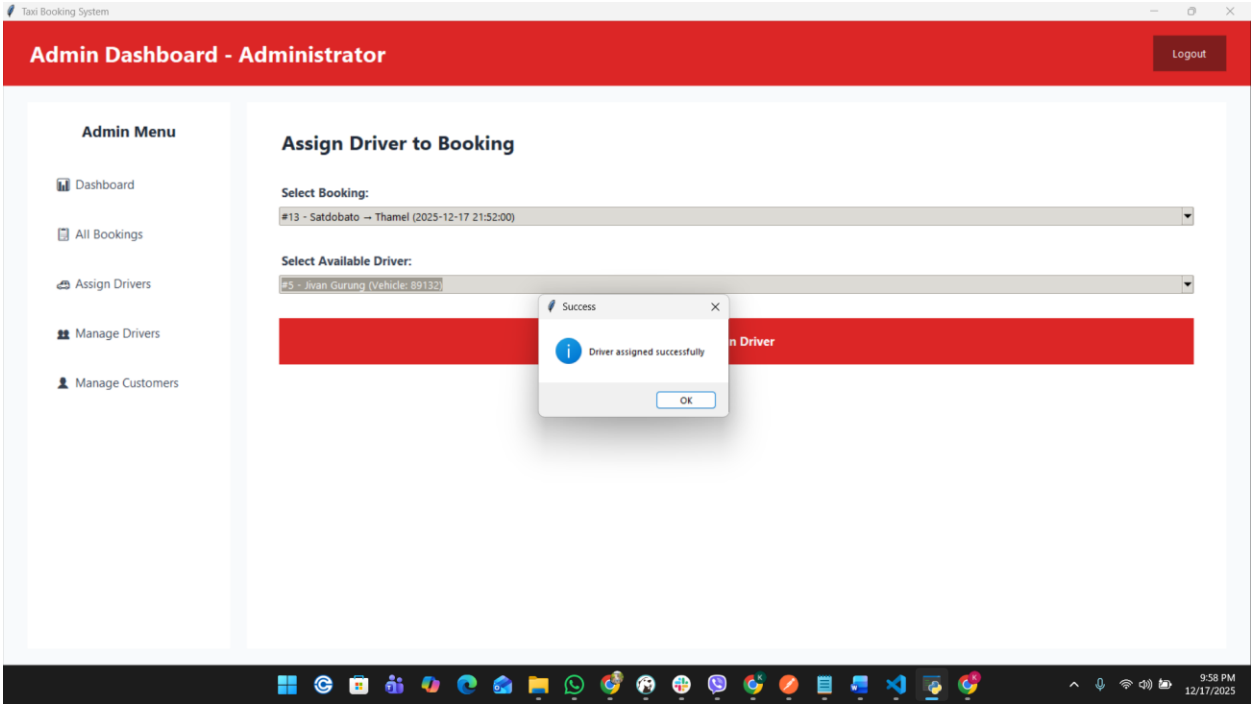

Figure 25 T-9 Assign driver to booking (Admin)

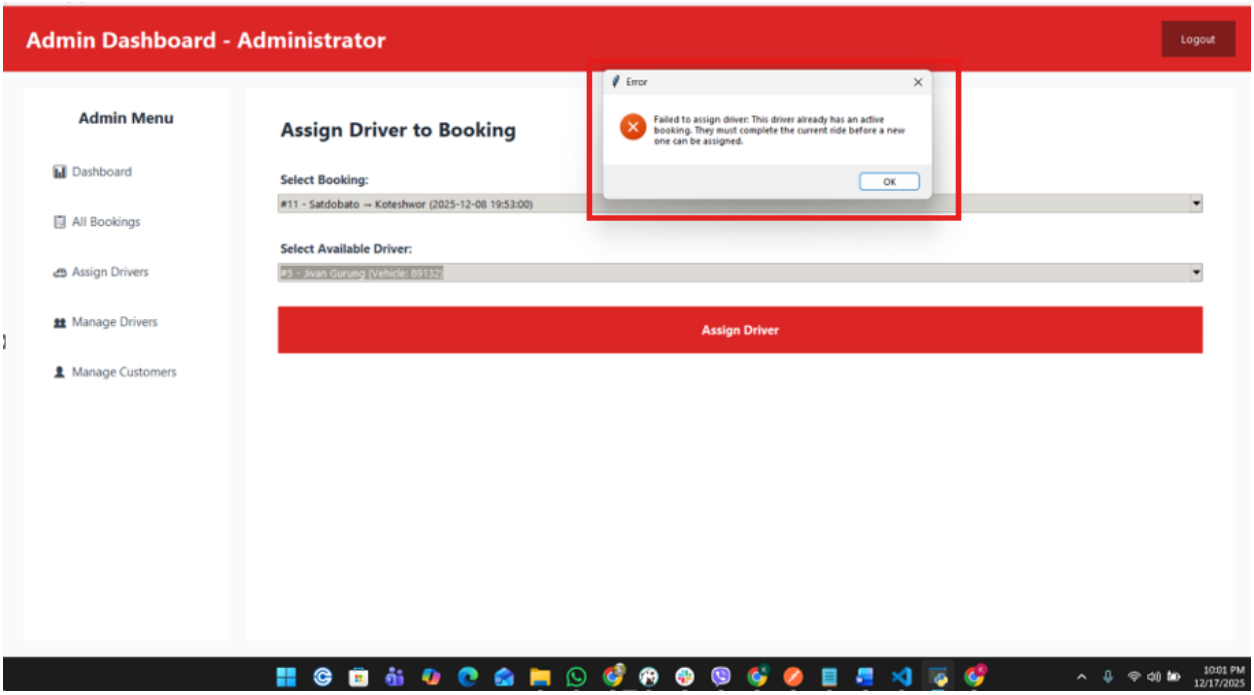

Figure 26 T-10 Prevent driver double booking

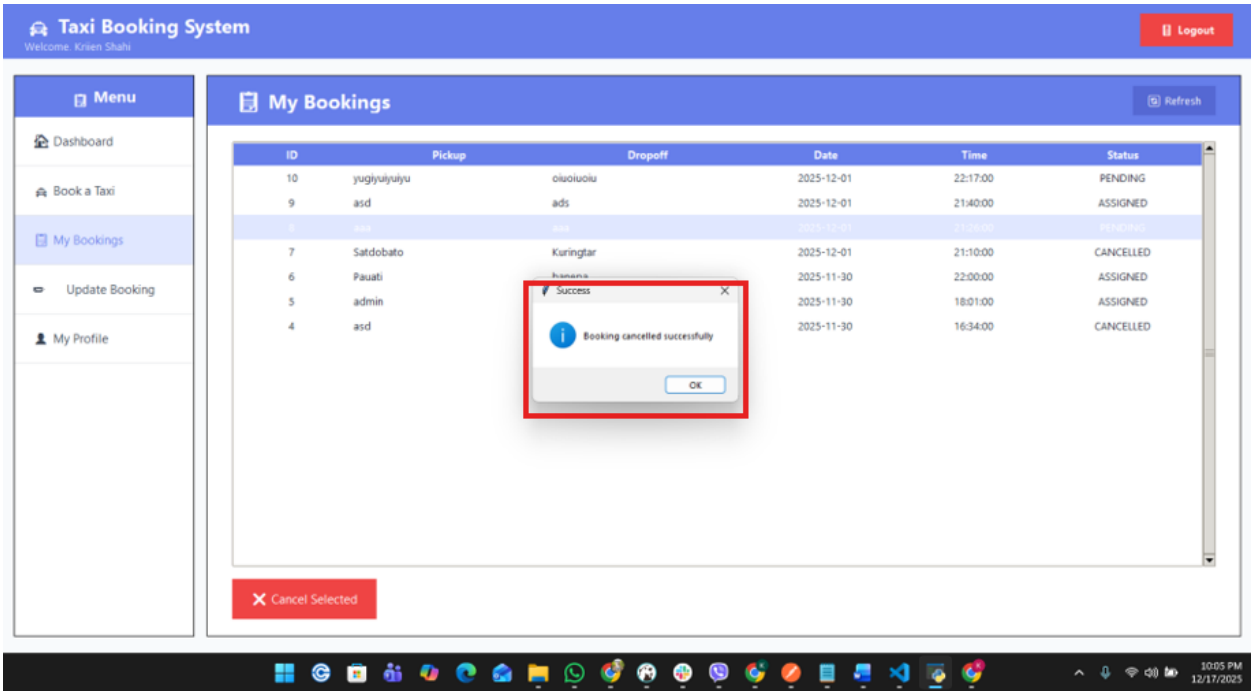

Figure 27 T-11Cancel Booking by Customer

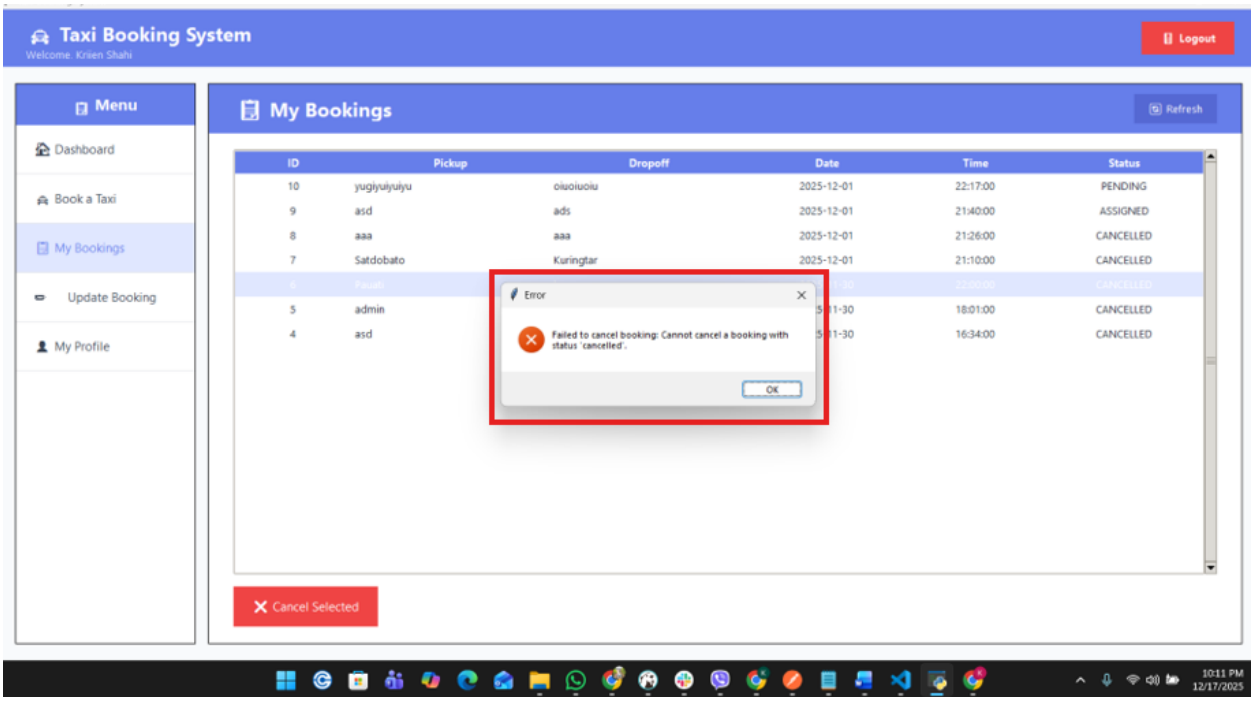

Figure 28 T-12 Cancel the ride with cancel status

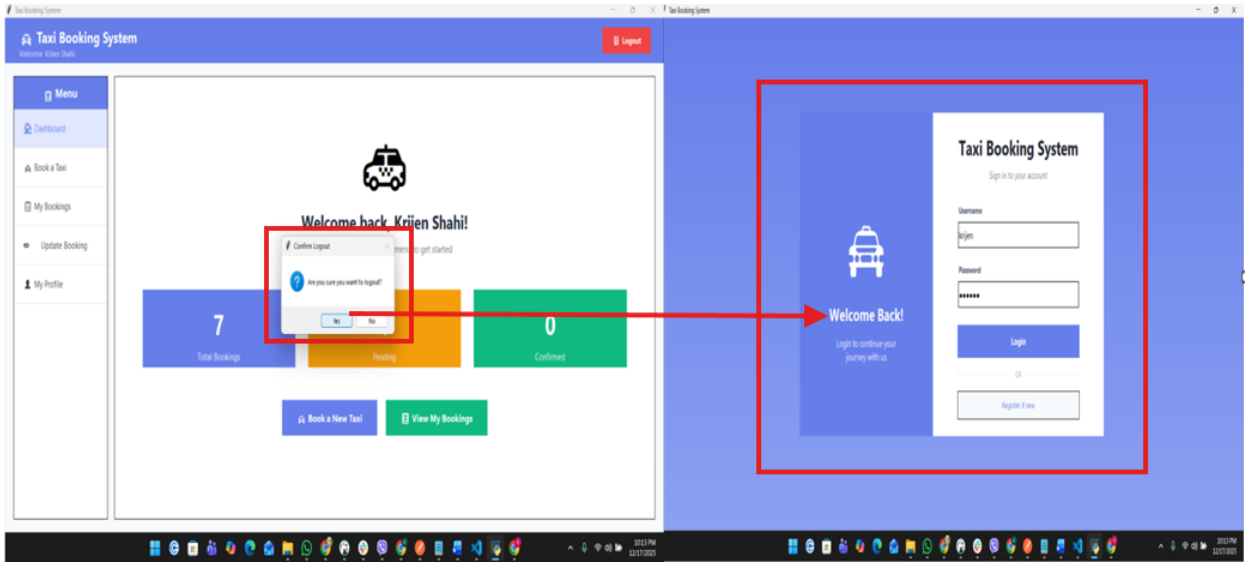

Figure 29 T-13 Validate Logout Functionality

Discussion/Reflection/Critical Analysis

Taxi Booking System

In this assignment, I was tasked to create Taxi Booking. The primary purpose of the system was to handle the daily taxi activities like user registration, booking of taxi by the customer, allocation of a taxi driver to the customer and the process of managing the booking. The approach that was followed in the development of the system was that of an object-oriented approach with a layered structure, which assisted us in isolating the user interface, business logic and database operations.

What Went Well

Usage of the concept of object-oriented programming was one of the most successful elements of the project. The code was simplified by the fact that it separated the system into the users, drivers, bookings, and services classes to be easier to maintain and understand. The layered structure was also beneficial in that it was easier to debug the problems, since I could trace the problems to a particular layer.

The basic functions including the validation of the login, taxi booking, assigning the driver, booking cancellation, and role-based dashboards were successfully implemented and worked as

intended. Role controls and verification allowed customers, drivers and administrators to do what was system configuration allowed them to.

What Went Wrong

The major problem experienced in the project was time management. The preliminary calculation of the time to be used to combine the user interface with the backlog system and database functionalities was underestimated. This has led to a hasty development towards deadlines particularly in the process of testing and polishing the user interface. The other issue was the enforcement and administration of business regulations, including ensuring a driver was not assigned to several active bookings. Validation rules required coordination across multiple layers, which was complex. In some parts of the system, validation logic was repeated between the UI and backend, making the code harder to manage (Nielsen, 1994). With better early planning, these issues could have been reduced.

Future Improvements

A number of enhancements can be made in future. The user interface can be made more responsive and this would make the usability better at various screen sizes. Other functionalities including booking reminders, notifications, payment integration, ride history analytics would make the system more realistic.

Technically, the system architecture might be more refined by ensuring further separation of validation and business logic and the database layer. It would decrease the duplication and simplify the system in terms of extension and maintenance.

Learning Experience and Challenges

This was an important learning experience that made me learn how software systems in the real world are designed and implemented. I enriched my skills in the object-oriented programming, integration of databases, and the layered architecture. Balancing this project with other academic and personal tasks was not easy but it helped me to understand the relevance of planning, collaboration, and time management.

Conclusion

In general, the project of Taxi Booking System proved to be difficult, but also demanding. Although time and technical issues were a problem, the system delivered required information on the assignment and gave critical insights into the real-life software development. The experience and the lessons acquired in the course of this project will be helpful in further studying and work.

Folder Structure

Figure 30 Folder Structure

References

Lee, S., Lee, W., Vogt, C.A. and Zhang, Y, n.d. A comparative analysis of factors influencing millennial travellers' intentions to use ride-hailing. Information Technology & Tourism. pp. 133-157.

Myers, G.J., Badgett, T., Thomas, T.M. and Sandler, C., 2004. The art of software testing. Volume 2.

Nielsen, J., 1994. Usability Engineering. San Francisco: Morgan Kaufmann.

Norman, D., 2013. *The Design of Everyday Things.* New York: s.n.

Rayle, L., Dai, D., Chan, N., Cervero, R. and Shaheen, S., n.d. A survey-based comparison of taxis, transit, and ridesourcing services in San Francisco. *ransport policy,* pp. 168-178.

Sommerville, I., 2011. Software engineering 9th Edition. Volume ISBN-10 137035152.

Wing, J. 2. C. t., 2006. Computational thinking. Communications of the ACM. pp. 33-35.

Appendix

main.py

main.py > ...

```
3
4   from ui.app_context import AppContext
5   from ui.main_app import MainApp
6
7
8   def main():
9       #(Database + Services)
10      context = AppContext()
11
12      # To run the Tkinter main application
13      app = MainApp(context)
14      app.mainloop()
15
16
17  if __name__ == "__main__":
18      main()
19
```

Ui

app_context.py

main.py | app_context.py | base_dal.py

ui > app_context.py > ...

```python
from dataacesslayer.db_connector import Database
from services.user_services import UserService
from services.booking_service import BookingService
from services.driver_service import DriverService
from services.customer_service import CustomerService


class AppContext:
    """
    Holds shared objects for the whole application:
    - Database
    - Services (UserService, BookingService, etc.)
    """

    def __init__(s
        self.db = Database()
        self.db.init_schema()

        self.user_service = UserService(self.db)
        self.booking_service = BookingService(self.db)
        # Services used by admin / driver dashboards
        self.driver_service = DriverService(self.db)
        self.customer_service = CustomerService(self.db)

```

class Database()

Handles MySQL connection and database/schema initialization.

main_app.py

main.py | booking_service.py | login_page.py | main_app.py

ui > main_app.py > MainApp

```python

import tkinter as tk
from typing import Dict

from ui.app_context import AppContext
from ui.login_page import LoginPage
from ui.register_page import RegisterPage
from ui.customer_dashboard import CustomerDashboard
from ui.driver_dashboard import DriverDashboard
from ui.admin_dashboard import AdminDashboard


class MainApp(tk.Tk):

    def __init__(self, context: AppContext):
        super().__init__()
        self.title("Taxi Booking System")
        self.geometry("900x600")

        self.context = context
        self.current_user = None

        container = tk.Frame(self)
        container.pack(side="top", fill="both", expand=True)

        container.grid_rowconfigure(0, weight=1)
        container.grid_columnconfigure(0, weight=1)

        self.frames: Dict[str, tk.Frame] = {}

        for PageClass in (
            LoginPage,
            RegisterPage,
            CustomerDashboard,
            DriverDashboard,
            AdminDashboard,
        ):
            page_name = PageClass.__name__
            frame = PageClass(parent=container, controller=self)
            self.frames[page_name] = frame
            frame.grid(row=0, column=0, sticky="nsew")

        self.show_frame("LoginPage")

    def show_frame(self, page_name: str):
        """
        Bring a frame to the front.
        """
        frame = self.frames.get(page_name)
        if frame is None:
            raise ValueError(f"No frame named '{page_name}'")
        frame.tkraise()

    def set_current_user(self, user: dict):
        """
        Store the logged-in user dict.
        """
        self.current_user = user

    def logout(self):
        """
        Log out current user and go back to LoginPage.
        """
        self.current_user = None
        self.show_frame("LoginPage")

```

login_page.py

main.py | booking_service.py | login_page.py

ui > login_page.py > LoginPage

```python
# ui/login_page.py

import tkinter as tk
from tkinter import messagebox
from PIL import Image, ImageTk


class LoginPage(tk.Frame):

    def __init__(self, parent, controller):
        super().__init__(parent)
        self.controller = controller

        self.configure(bg="#667eea")

        self.bind("<Configure>", self._on_resize)

        self.canvas = tk.Canvas(self, highlightthickness=0)
        self.canvas.place(x=0, y=0, relwidth=1, relheight=1)

        self.main_container = tk.Frame(self.canvas, bg="white")
        self.main_container.place(relx=0.5, rely=0.5, anchor="c", width=800, height=500)

        self.left_frame = tk.Frame(self.main_container, bg="#667eea", width=350)
        self.left_frame.pack(side="left", fill="both", expand=False)
        self.left_frame.pack_propagate(False)

        self.side_image = None
        try:
            img = Image.open("assets/login_image.jpg")
            self._load_side_image()
        except Exception:
            self._create_fallback_content()

        self.right_frame = tk.Frame(self.main_container, bg="white", width=450)
        self.right_frame.pack(side="right", fill="both", expand=False)
        self.right_frame.pack_propagate(False)

        form_container = tk.Frame(self.right_frame, bg="white")
        form_container.place(relx=0.5, rely=0.5, anchor="c")

        title = tk.Label(
            form_container,
            text="Taxi Booking System",
            font=("Segoe UI", 24, "bold"),
            bg="white",
            fg="#1a202c",
        )
        title.pack(pady=(0, 5))

        subtitle = tk.Label(
            form_container,
            text="Sign in to your account",
            font=("Segoe UI", 11),
            bg="white",
            fg="#718096",
        )
        subtitle.pack(pady=(0, 30))

        form_frame = tk.Frame(form_container, bg="white")
        form_frame.pack()

        # Username field
        tk.Label(
            form_frame,
            text="Username",
            font=("Segoe UI", 10, "bold"),
            bg="white",
            fg="#4a5568",
```

main.py | login_page.py ●

ui > login_page.py > LoginPage > handle_login

```
8   class LoginPage(tk.Frame):
16      def __init__(self, parent, controller):
68          bg="white",
69          fg="#718096",
70      )
71      subtitle.pack(pady=(0, 30))
72
73      # ===== FORM =====
74      form_frame = tk.Frame(form_container, bg="white")
75      form_frame.pack()
76
77      # Username field
78      tk.Label(
79          form_frame,
80          text="Username",
81          font=("Segoe UI", 10, "bold"),
82          bg="white",
83          fg="#4a5568",
84          anchor="w"
85      ).pack(fill="x", pady=(0, 5))
86
87      self.username_entry = tk.Entry(
88          form_frame,
89          font=("Segoe UI", 11),
90          relief="solid",
91          bd=1,
92          width=35,
93          highlightthickness=2,
94          highlightcolor="#667eea",
95          highlightbackground="#e2e8f0"
96      )
97      self.username_entry.pack(fill="x", ipady=8, pady=(0, 20))
98
99      # Password field
100     tk.Label(
101         form_frame,
102         text="Password",
103         font=("Segoe UI", 10, "bold"),
104         bg="white",
105         fg="#4a5568",
106         anchor="w"
107     ).pack(fill="x", pady=(0, 5))
108
109     self.password_entry = tk.Entry(
110         form_frame,
111         show="•",
112         font=("Segoe UI", 11),
113         relief="solid",
114         bd=1,
115         width=35,
116         highlightthickness=2,
117         highlightcolor="#667eea",
118         highlightbackground="#e2e8f0"
119     )
120     self.password_entry.pack(fill="x", ipady=8, pady=(0, 25))
121
122     # ===== BUTTONS =====
123     login_btn = tk.Button(
124         form_frame,
125         text="Login",
126         font=("Segoe UI", 11, "bold"),
127         bg="#667eea",
128         fg="white",
129         activebackground="#5568d3",
130         activeforeground="white",
131         relief="flat",
132         cursor="hand2",
133         command=self.handle_login,
134         bd=0,
135         width=35
```

main.py login_page.py ●

ui > login_page.py > LoginPage > __init__

```
class LoginPage(tk.Frame):
    def __init__(self, parent, controller):
        )
        login_btn.pack(fill="x", ipady=10, pady=(0, 15))

        # Divider
        divider_frame = tk.Frame(form_frame, bg="white")
        divider_frame.pack(fill="x", pady=(0, 15))

        tk.Frame(divider_frame, bg="#e2e8f0", height=1).pack(side="left", fill="x", expand=True, padx=(0, 10))
        tk.Label(divider_frame, text="OR", font=("Segoe UI", 9), bg="white", fg="#a0aec0").pack(side="left")
        tk.Frame(divider_frame, bg="#e2e8f0", height=1).pack(side="left", fill="x", expand=True, padx=(10, 0))

        register_btn = tk.Button(
            form_frame,
            text="Register if new",
            font=("Segoe UI", 10),
            bg="#f7fafc",
            fg="#667eea",
            activebackground="#edf2f7",
            activeforeground="#5568d3",
            relief="solid",
            bd=1,
            cursor="hand2",
            width=35,
            command=lambda: self.controller.show_frame("RegisterPage"),
        )
        register_btn.pack(fill="x", ipady=8)

    def _on_resize(self, event=None):
        """Handle window resize to redraw gradient"""
        self.after(10, self._draw_gradient)

    def _draw_gradient(self):
        """Draw a gradient background on canvas"""
        self.canvas.delete("gradient")
        width = self.canvas.winfo_width() or 900
        height = self.canvas.winfo_height() or 600

        step = max(1, height // 100)
        for i in range(0, height, step):
            ratio = i / height
            r = int(102 + (139 - 102) * ratio)
            g = int(126 + (161 - 126) * ratio)
            b = int(234 + (242 - 234) * ratio)
            color = f'#{r:02x}{g:02x}{b:02x}'
            self.canvas.create_rectangle(0, i, width, i + step, fill=color, outline=color, tags="gradient")

    def _create_fallback_content(self):
        """Create fallback decorative content when image not found"""
        deco_frame = tk.Frame(self.left_frame, bg="#667eea")
        deco_frame.place(relx=0.5, rely=0.5, anchor="c")

        tk.Label(
            deco_frame,
            text="🚗",
            font=("Segoe UI", 80),
            bg="#667eea",
            fg="white"
        ).pack(pady=20)

        tk.Label(
            deco_frame,
            text="Welcome Back!",
            font=("Segoe UI", 20, "bold"),
            bg="#667eea",
            fg="white"
        ).pack()

        tk.Label(
            deco_frame,
```

main.py login_page.py ●

ui > login_page.py > LoginPage > _create_fallback_content

```
  8  class LoginPage(tk.Frame):
182      def _create_fallback_content(self):
204          deco_frame,
205          text="Login to continue your\njourney with us",
206          font=("Segoe UI", 12),
207          bg="#667eea",
208          fg="white",
209          justify="center"
210      ).pack(pady=10)
211
212      def _load_side_image(self):
213          """Load and display side image"""
214          try:
215              img = Image.open("assets/login_image.jpg")
216              height = self.left_frame.winfo_height() or 500
217              img = img.resize((350, height), Image.LANCZOS)
218              self.side_image = ImageTk.PhotoImage(img)
219              img_label = tk.Label(self.left_frame, image=self.side_image, bg="#667eea")
220              img_label.place(x=0, y=0, relwidth=1, relheight=1)
221          except:
222              pass
223
224      def handle_login(self):
225          username = self.username_entry.get().strip()
226          password = self.password_entry.get().strip()
227
228          if not username or not password:
229              messagebox.showerror("Error", "Please enter both username and password.")
230              return
231
232          user_service = self.controller.context.user_service
233          user = user_service.login(username, password)
234
235          if not user:
236              messagebox.showerror(
237                  "Login failed", "Invalid credentials or inactive account."
238              )
239              return
240
241          self.controller.set_current_user(user)
242          messagebox.showinfo("Login successful", "Login successful.")
243
244          role = user["role"]
245          if role == "customer":
246              customer_dash = self.controller.frames["CustomerDashboard"]
247              customer_dash.set_user(user)
248              self.controller.show_frame("CustomerDashboard")
249          elif role == "driver":
250              driver_dash = self.controller.frames["DriverDashboard"]
251              driver_dash.set_user(user)
252              self.controller.show_frame("DriverDashboard")
253          elif role == "admin":
254              admin_dash = self.controller.frames["AdminDashboard"]
255              admin_dash.set_user(user)
256              self.controller.show_frame("AdminDashboard")
257          else:
258              messagebox.showerror("Error", f"Unknown role: {role}")
```

register_page.py

main.py | login_page.py | register_page.py

ui > register_page.py > RegisterPage

```python
# ui/register_page.py

import tkinter as tk
from tkinter import ttk, messagebox
from PIL import Image, ImageTk


class RegisterPage(tk.Frame):

    def __init__(self, parent, controller):
        super().__init__(parent)
        self.controller = controller

        self.configure(bg="#764ba2")

        self.bind("<Configure>", self._on_resize)

        self.canvas = tk.Canvas(self, highlightthickness=0)
        self.canvas.place(x=0, y=0, relwidth=1, relheight=1)

        self.main_container = tk.Frame(self.canvas, bg="white")
        self.main_container.place(relx=0.5, rely=0.5, anchor="c", width=820, height=550)

        # ===== LEFT SIDE - FORM =====
        self.left_frame = tk.Frame(self.main_container, bg="white", width=500)
        self.left_frame.pack(side="left", fill="both", expand=False)
        self.left_frame.pack_propagate(False)

        # Inner container with padding
        form_outer = tk.Frame(self.left_frame, bg="white")
        form_outer.pack(fill="both", expand=True, padx=30, pady=20)

        # ===== TITLE =====
        title = tk.Label(
            form_outer,
            text="Create Account",
            font=("Segoe UI", 24, "bold"),
            bg="white",
            fg="#1a202c",
        )
        title.pack(pady=(0, 5))

        subtitle = tk.Label(
            form_outer,
            text="Join us today!",
            font=("Segoe UI", 11),
            bg="white",
            fg="#718096",
        )
        subtitle.pack(pady=(0, 15))

        # ===== SCROLLABLE FORM =====
        canvas_frame = tk.Frame(form_outer, bg="white")
        canvas_frame.pack(fill="both", expand=True)

        scrollbar = tk.Scrollbar(canvas_frame, orient="vertical")
        form_canvas = tk.Canvas(canvas_frame, bg="white", yscrollcommand=scrollbar.set, highlightthickness=0)
        scrollbar.config(command=form_canvas.yview)

        scrollbar.pack(side="right", fill="y")
        form_canvas.pack(side="left", fill="both", expand=True)

        # Form container inside canvas
        form_frame = tk.Frame(form_canvas, bg="white")
        form_canvas_window = form_canvas.create_window((0, 0), window=form_frame, anchor="nw")

        # Update scroll region when form changes size
        def configure_scroll(event):
            form_canvas.configure(scrollregion=form_canvas.bbox("all"))
            form_canvas.itemconfig(form_canvas_window, width=event.width)
```

main.py | login_page.py | register_page.py

ui > register_page.py > RegisterPage

```
class RegisterPage(tk.Frame):
    def __init__(self, parent, controller):

        form_frame.bind("<Configure>", configure_scroll)
        form_canvas.bind("<Configure>", lambda e: form_canvas.itemconfig(form_canvas_window, width=e.width))

        # Mouse wheel scrolling
        def _on_mousewheel(event):
            form_canvas.yview_scroll(int(-1*(event.delta/120)), "units")
        form_canvas.bind_all("<MouseWheel>", _on_mousewheel)

        # ===== FORM FIELDS =====
        self.entries = {}

        # Us
        tk.L  (variable) form_frame: Frame
            form_frame,
            text="User Type",
            font=("Segoe UI", 10, "bold"),
            bg="white",
            fg="#4a5568",
            anchor="w"
        ).pack(fill="x", pady=(8, 3), padx=5)

        self.user_type_var = tk.StringVar(value="customer")
        user_type_combo = ttk.Combobox(
            form_frame,
            textvariable=self.user_type_var,
            values=["customer", "driver", "admin"],
            state="readonly",
            font=("Segoe UI", 10),
            width=38
        )
        user_type_combo.pack(fill="x", ipady=6, padx=5)
        user_type_combo.bind("<<ComboboxSelected>>", self._on_user_type_change)

        # Container for dynamic fields
        self.dynamic_fields_frame = tk.Frame(form_frame, bg="white")
        self.dynamic_fields_frame.pack(fill="both", expand=True)

        # Create initial fields for customer
        self._create_fields_for_type("customer")

        # ===== BUTTONS =====
        btn_frame = tk.Frame(form_frame, bg="white")
        btn_frame.pack(fill="x", pady=(20, 10), padx=5)

        register_btn = tk.Button(
            btn_frame,
            text="Register",
            font=("Segoe UI", 11, "bold"),
            bg="#764ba2",
            fg="white",
            activebackground="#653a8f",
            activeforeground="white",
            relief="flat",
            cursor="hand2",
            command=self.handle_register,
            bd=0
        )
        register_btn.pack(fill="x", ipady=10, pady=(0, 10))

        back_btn = tk.Button(
            btn_frame,
            text="← Back to Login",
            font=("Segoe UI", 10),
            bg="#f7fafc",
            fg="#764ba2",
            activebackground="#edf2f7",
            activeforeground="#653a8f",
            relief="solid",
            bd=1
```

```
ui > register_page.py > RegisterPage > __init__
  8  class RegisterPage(tk.Frame):
 10      def __init__(self, parent, controller):
139              relief="solid",
140              bd=1,
141              cursor="hand2",
142              command=lambda: self.controller.show_frame("LoginPage"),
143          )
144          back_btn.pack(fill="x", ipady=8)
145
146          # ===== RIGHT SIDE - IMAGE SECTION =====
147          self.right_frame = tk.Frame(self.main_container, bg="#764ba2", width=320)
148          self.right_frame.pack(side="right", fill="both", expand=False)
149          self.right_frame.pack_propagate(False)
150
151          # Try to load decorative image
152          self.side_image = None
153          try:
154              img = Image.open("assets/register_image.jpg")
155              self._load_side_image()
156          except Exception:
157              # Fallback decorative content
158              self._create_fallback_content()
159
160      def _on_resize(self, event=None):
161          """Handle window resize to redraw gradient"""
162          self.after(10, self._draw_gradient)
163
164      def _draw_gradient(self):
165          """Draw a gradient background on canvas"""
166          self.canvas.delete("gradient")
167          width = self.canvas.winfo_width() or 900
168          height = self.canvas.winfo_height() or 600
169
170          # Gradient from purple to pink-purple
171          step = max(1, height // 100)
172          for i in range(0, height, step):
173              ratio = i / height
174              r = int(118 + (247 - 118) * ratio)
175              g = int(75 + (167 - 75) * ratio)
176              b = int(162 + (216 - 162) * ratio)
177              color = f'#{r:02x}{g:02x}{b:02x}'
178              self.canvas.create_rectangle(0, i, width, i + step, fill=color, outline=color, tags="gradient")
179
180      def _create_fallback_content(self):
181          """Create fallback decorative content when image not found"""
182          deco_frame = tk.Frame(self.right_frame, bg="#764ba2")
183          deco_frame.place(relx=0.5, rely=0.5, anchor="c")
184
185          tk.Label(
186              deco_frame,
187              text="🚗",
188              font=("Segoe UI", 70),
189              bg="#764ba2",
190              fg="white"
191          ).pack(pady=15)
192
193          tk.Label(
194              deco_frame,
195              text="Start Your",
196              font=("Segoe UI", 18, "bold"),
197              bg="#764ba2",
198              fg="white"
199          ).pack()
200
201          tk.Label(
202              deco_frame,
203              text="Journey",
204              font=("Segoe UI", 18, "bold"),
205              bg="#764ba2",
206              fg="white"
207          ).pack(pady=(0, 15))
208
```

```
ui > register_page.py > RegisterPage > _create_fallback_content
  8  class RegisterPage(tk.Frame):
180      def _create_fallback_content(self):
208
209          features = [
210              "✓ Easy booking",
211              "✓ 24/7 service",
212              "✓ Safe rides",
213              "✓ Best rates"
214          ]
215
216          for feature in features:
217              tk.Label(
218                  deco_frame,
219                  text=feature,
220                  font=("Segoe UI", 11),
221                  bg="#764ba2",
222                  fg="white",
223                  anchor="w"
224              ).pack(pady=5, padx=30, fill="x")
225
226      def _on_user_type_change(self, event=None):
227          """Handle user type selection change"""
228          user_type = self.user_type_var.get()
229          self._create_fields_for_type(user_type)
230
231      def _create_fields_for_type(self, user_type):
232          """Create form fields based on selected user type"""
233          # Clear existing fields
234          for widget in self.dynamic_fields_frame.winfo_children():
235              widget.destroy()
236
237          self.entries = {}
238
239          # Define fields based on user type
240          if user_type == "customer":
241              fields = [
242                  ("Full Name", False),
243                  ("Address", False),
244                  ("Phone", False),
245                  ("Email", False),
246                  ("Username", False),
247                  ("Password", True),
248                  ("Confirm Password", True),
249              ]
250          elif user_type == "driver":
251              fields = [
252                  ("Full Name", False),
253                  ("Address", False),
254                  ("Phone", False),
255                  ("Email", False),
256                  ("License Number", False),
257                  ("Vehicle Number", False),
258                  ("Username", False),
259                  ("Password", True),
260                  ("Confirm Password", True),
261              ]
262          else:  # admin
263              fields = [
264                  ("Full Name", False),
265                  ("Address", False),
266                  ("Phone", False),
267                  ("Email", False),
268                  ("Username", False),
269                  ("Password", True),
270                  ("Confirm Password", True),
271              ]
272
273          # Create fields
274          for label_text, is_password in fields:
275              # Label
276              tk.Label(
```

main.py | login_page.py | register_page.py

ui > register_page.py > RegisterPage > _create_fields_for_type

```
class RegisterPage(tk.Frame):
    def _create_fields_for_type(self, user_type):
        tk.Label(
            self.dynamic_fields_frame,
            text=label_text,
            font=("Segoe UI", 10, "bold"),
            bg="white",
            fg="#4a5568",
            anchor="w"
        ).pack(fill="x", pady=(8, 3), padx=5)

        # Entry
        entry = tk.Entry(
            self.dynamic_fields_frame,
            font=("Segoe UI", 10),
            relief="solid",
            bd=1,
            highlightthickness=2,
            highlightcolor="#764ba2",
            highlightbackground="#e2e8f0",
            show="•" if is_password else ""
        )
        entry.pack(fill="x", ipady=6, padx=5)
        self.entries[f"{label_text}:"] = entry

    def _load_side_image(self):
        """Load and display side image"""
        try:
            img = Image.open("assets/register_image.jpg")
            height = self.right_frame.winfo_height() or 550
            img = img.resize((320, height), Image.LANCZOS)
            self.side_image = ImageTk.PhotoImage(img)
            img_label = tk.Label(self.right_frame, image=self.side_image, bg="#764ba2")
            img_label.place(x=0, y=0, relwidth=1, relheight=1)
        except:
            pass

    def handle_register(self):
        user_type = self.user_type_var.get()
        full_name = self.entries.get("Full Name:", tk.Entry()).get().strip()
        address = self.entries.get("Address:", tk.Entry()).get().strip()
        phone = self.entries.get("Phone:", tk.Entry()).get().strip()
        email = self.entries.get("Email:", tk.Entry()).get().strip()
        username = self.entries.get("Username:", tk.Entry()).get().strip()
        password = self.entries.get("Password:", tk.Entry()).get().strip()
        confirm_password = self.entries.get("Confirm Password:", tk.Entry()).get().strip()

        # Validate common fields
        if not all([user_type, full_name, address, phone, email, username, password, confirm_password]):
            messagebox.showerror("Error", "All fields are required.")
            return

        if password != confirm_password:
            messagebox.showerror("Error", "Passwords do not match.")
            return

        user_service = self.controller.context.user_service

        try:
            # Register based on user type
            if user_type == "customer":
                user_id = user_service.register_customer(
                    full_name=full_name,
                    address=address,
                    phone=phone,
                    email=email,
                    username=username,
                    password=password,
                )
            elif user_type == "driver":
                # Get driver-specific fields
```

```
            license_number = self.entries.get("License Number:", tk.Entry()).get().strip()
            vehicle_number = self.entries.get("Vehicle Number:", tk.Entry()).get().strip()

            if not license_number or not vehicle_number:
                messagebox.showerror("Error", "License Number and Vehicle Number are required for drivers.")
                return

            user_id = user_service.register_driver(
                full_name=full_name,
                address=address,
                phone=phone,
                email=email,
                license_number=license_number,
                vehicle_number=vehicle_number,
                username=username,
                password=password,
            )
        elif user_type == "admin":
            user_id = user_service.register_admin(
                full_name=full_name,
                address=address,
                phone=phone,
                email=email,
                username=username,
                password=password,
            )

        messagebox.showinfo(
            "Success",
            f"Registration successful as {user_type}. You can now log in.\n(Your user id: {user_id})",
        )
        self.controller.show_frame("LoginPage")
    except ValueError as e:
        messagebox.showerror("Registration Error", str(e))
    except AttributeError as e:
        messagebox.showerror("Error", f"Registration method for {user_type} not implemented: {e}")
    except Exception as e:
        messagebox.showerror("Error", f"Unexpected error: {e}")
```

dataaccesslayer

base_dal.py

main.py | base_dal.py

dataaccesslayer > base_dal.py > BaseDAL

```python
from .db_connector import Database


class BaseDAL:
    """
    Base Data Access Layer class to be inherited by all DALs.
    Holds the shared Database instance and provides a helper for cursors.
    """

    def __init__(self, db: Database):
        self.db = db

    def _get_cursor(self, dictionary: bool = True):
        conn = self.db.get_connection()
        return conn.cursor(dictionary=dictionary)

```

booking_dal.py

main.py | booking_dal.py

dataaccesslayer > booking_dal.py > BookingDAL > cancel_booking

```python
# app/dataaccesslayer/booking_dal.py

from typing import Optional, List, Dict, Any
from datetime import datetime
from .base_dal import BaseDAL


class BookingDAL(BaseDAL):
    """
    Data Access Layer for 'bookings' table.
    """

    def create_booking(
        self,
        customer_id: int,
        pickup_location: str,
        dropoff_location: str,
        pickup_datetime: datetime,
        notes: Optional[str] = None,
    ) -> int:
        """
        Insert a new booking with status 'pending'.
        """
        query = """
            INSERT INTO bookings (
                customer_id,
                pickup_location,
                dropoff_location,
                pickup_datetime,
                status,
                notes
            )
            VALUES (%s, %s, %s, %s, 'pending', %s)
        """
        params = (
            customer_id,
            pickup_location,
            dropoff_location,
            pickup_datetime,
            notes,
        )

        cursor = self._get_cursor()
        cursor.execute(query, params)
        self.db.get_connection().commit()
        booking_id = cursor.lastrowid
        cursor.close()
        return booking_id

    def get_by_id(self, booking_id: int) -> Optional[Dict[str, Any]]:
        query = "SELECT * FROM bookings WHERE id = %s"
        cursor = self._get_cursor()
        cursor.execute(query, (booking_id,))
        row = cursor.fetchone()
        cursor.close()
        return row

    def list_by_customer(self, customer_id: int) -> List[Dict[str, Any]]:
        query = """
            SELECT
                id,
                customer_id,
                driver_id,
                pickup_location,
                dropoff_location,
                pickup_datetime,
                DATE(pickup_datetime) AS pickup_date,
                TIME(pickup_datetime) AS pickup_time,
                status,
                notes
            FROM bookings
            WHERE customer_id = %s
            ORDER BY pickup_datetime DESC
        """
        cursor = self._get_cursor()
        cursor.execute(query, (customer_id,))
        rows = cursor.fetchall()
        cursor.close()
        return rows

    def list_by_driver(self, driver_id: int) -> List[Dict[str, Any]]:
        query = """
            SELECT
                id,
                customer_id,
                driver_id,
```

```
                driver_id,
                pickup_location,
                dropoff_location,
                pickup_datetime,
                DATE(pickup_datetime) AS pickup_date,
                TIME(pickup_datetime) AS pickup_time,
                status,
                notes
            FROM bookings
            WHERE driver_id = %s
            ORDER BY pickup_datetime DESC
        """
        cursor = self._get_cursor()
        cursor.execute(query, (driver_id,))
        rows = cursor.fetchall()
        cursor.close()
        return rows

    def list_all(self) -> List[Dict[str, Any]]:
        query = """
            SELECT
                id,
                customer_id,
                driver_id,
                pickup_location,
                dropoff_location,
                pickup_datetime,
                DATE(pickup_datetime) AS pickup_date,
                TIME(pickup_datetime) AS pickup_time,
                status,
                notes
            FROM bookings
            ORDER BY pickup_datetime DESC
        """
        cursor = self._get_cursor()
        cursor.execute(query)
        rows = cursor.fetchall()
        cursor.close()
        return rows

    def cancel_booking(self, booking_id: int) -> None:
        """
        Mark a booking as 'cancelled'.
        """
        query = "UPDATE bookings SET status = 'cancelled' WHERE id = %s"
        cursor = self._get_cursor()
        cursor.execute(query, (booking_id,))
        self.db.get_connection().commit()
        cursor.close()

    def update_booking(
        self,
        booking_id: int,
        pickup_location: str,
        dropoff_location: str,
        pickup_datetime: datetime,
        notes: Optional[str] = None,
    ) -> None:
        """
        Update basic booking details.
        """
        query = """
            UPDATE bookings
            SET pickup_location = %s,
                dropoff_location = %s,
                pickup_datetime = %s,
                notes = %s
            WHERE id = %s
        """
        params = (
            pickup_location,
            dropoff_location,
            pickup_datetime,
            notes,
            booking_id,
        )

        cursor = self._get_cursor()
        cursor.execute(query, params)
        self.db.get_connection().commit()
        cursor.close()

    def assign_driver(self, booking_id: int, driver_id: int) -> None:
        """
```

```
        """
        Assign a driver and change status to 'assigned'.
        """
        query = """
            UPDATE bookings
            SET driver_id = %s, status = 'assigned'
            WHERE id = %s
        """
        cursor = self._get_cursor()
        cursor.execute(query, (driver_id, booking_id))
        self.db.get_connection().commit()
        cursor.close()

    def update_status(self, booking_id: int, status: str) -> None:
        """
        Update the status of a booking.
        """
        query = "UPDATE bookings SET status = %s WHERE id = %s"
        cursor = self._get_cursor()
        cursor.execute(query, (status, booking_id))
        self.db.get_connection().commit()
        cursor.close()

    def has_active_booking_for_driver(self, driver_id: int) -> bool:
        """
        Check if the driver already has any active booking
        (pending/assigned/ongoing). Used to prevent overlapping rides.
        """
        query = """
            SELECT COUNT(*) AS cnt
            FROM bookings
            WHERE driver_id = %s
              AND status IN ('pending', 'assigned', 'ongoing')
        """
        cursor = self._get_cursor()
        cursor.execute(query, (driver_id,))
        row = cursor.fetchone()
        cursor.close()
        return row["cnt"] > 0 if row else False

```

customer_dal.py

main.py | customer_dal.py

dataaccesslayer > customer_dal.py > CustomerDAL > delete_customer

```python
# app/dataaccesslayer/customer_dal.py

from typing import Optional, List, Dict, Any
from .base_dal import BaseDAL


class CustomerDAL(BaseDAL):
    """
    Data Access Layer for 'customers' table.
    """

    def create_customer(
        self,
        full_name: str,
        address: str,
        phone: str,
        email: str,
    ) -> int:
        """
        Insert a new customer and return the inserted ID.
        """
        query = """
            INSERT INTO customers (full_name, address, phone, email)
            VALUES (%s, %s, %s, %s)
        """
        params = (full_name, address, phone, email)

        cursor = self._get_cursor()
        cursor.execute(query, params)
        self.db.get_connection().commit()
        customer_id = cursor.lastrowid
        cursor.close()
        return customer_id

    def get_by_id(self, customer_id: int) -> Optional[Dict[str, Any]]:
        """
        Get a customer by ID.
        """
        query = "SELECT * FROM customers WHERE id = %s"
        cursor = self._get_cursor()
        cursor.execute(query, (customer_id,))
        row = cursor.fetchone()
        cursor.close()
        return row

    def get_by_email(self, email: str) -> Optional[Dict[str, Any]]:
        """
        Get a customer by email.
        """
        query = "SELECT * FROM customers WHERE email = %s"
        cursor = self._get_cursor()
        cursor.execute(query, (email,))
        row = cursor.fetchone()
        cursor.close()
        return row

    def list_all(self) -> List[Dict[str, Any]]:
        """
        Get all customers.
        """
        query = "SELECT * FROM customers"
        cursor = self._get_cursor()
        cursor.execute(query)
        rows = cursor.fetchall()
        cursor.close()
        return rows

    def update_customer(
        self,
        customer_id: int,
        full_name: str,
        address: str,
        phone: str,
        email: str,
    ) -> None:
        """
        Update a customer's details.
        """
        query = """
            UPDATE customers
            SET full_name = %s, address = %s, phone = %s, email = %s
            WHERE id = %s
        """
        params = (full_name, address, phone, email, customer_id)

        cursor = self._get_cursor()
```

```
86          cursor = self._get_cursor()
87          cursor.execute(query, params)
88          self.db.get_connection().commit()
89          cursor.close()
90
91      def delete_customer(self, customer_id: int) -> None:
92          """
93          Delete a customer. Bookings will cascade if FK has ON DELETE CASCADE.
94          """
95          query = "DELETE FROM customers WHERE id = %s"
96          cursor = self._get_cursor()
97          cursor.execute(query, (customer_id,))
98          self.db.get_connection().commit()
99          cursor.close()
100
```

Db_connector.py

main.py | db_connector.py

dataaccesslayer > db_connector.py > Database > __init__

```python
# app/dataaccesslayer/db_connector.py

import mysql.connector
from mysql.connector import Error
from config.settings import DB_CONFIG


class Database:
    """
    Handles MySQL connection and database/schema initialization.
    """

    def __init__(self):
        self.host = DB_CONFIG["host"]
        self.user = DB_CONFIG["user"]
        self.password = DB_CONFIG["password"]
        self.database_name = DB_CONFIG["database"]
        self.connection = None

        # Ensure DB exists, then connect to it
        self._ensure_database()
        self._connect_to_database()

    def _get_server_connection(self):
        """
        Connect to MySQL server WITHOUT specifying a database.
        Used for creating the database if it doesn't exist.
        """
        try:
            conn = mysql.connector.connect(
                host=self.host,
                user=self.user,
                password=self.password
            )
            return conn
        except Error as e:
            print(f"Error connecting to MySQL server: {e}")
            raise

    def _ensure_database(self):
        """
        Create the database if it does not exist.
        """
        try:
            conn = self._get_server_connection()
            cursor = conn.cursor()
            cursor.execute(f"CREATE DATABASE IF NOT EXISTS {self.database_name}")
            conn.commit()
            cursor.close()
            conn.close()
            print(f"Database '{self.database_name}' is ready.")
        except Error as e:
            print(f"Error ensuring database exists: {e}")
            raise

    def _connect_to_database(self):
        """
        Connect directly to the specific database.
        """
        try:
            self.connection = mysql.connector.connect(
                host=self.host,
                user=self.user,
                password=self.password,
                database=self.database_name
            )
            if self.connection.is_connected():
                print(f"Connected to database '{self.database_name}'.")
        except Error as e:
            print(f"Error connecting to database: {e}")
            raise
```

main.py | db_connector.py

dataacesslayer > db_connector.py > Database > __init__

```
class Database:
    def _connect_to_database(self):
            raise

    def get_connection(self):
        """
        Public method to get the active connection.
        """
        if not self.connection or not self.connection.is_connected():
            self._connect_to_database()
        return self.connection

    def init_schema(self):
        """
        Create all necessary tables if they do not exist.
        """
        conn = self.get_connection()
        cursor = conn.cursor()

        # 1. customers table (full customer details)
        cursor.execute(
            """
            CREATE TABLE IF NOT EXISTS customers (
                id INT AUTO_INCREMENT PRIMARY KEY,
                full_name VARCHAR(100) NOT NULL,
                address VARCHAR(255),
                phone VARCHAR(20),
                email VARCHAR(100) UNIQUE,
                created_at TIMESTAMP DEFAULT CURRENT_TIMESTAMP,
                updated_at TIMESTAMP DEFAULT CURRENT_TIMESTAMP
                    ON UPDATE CURRENT_TIMESTAMP
            ) ENGINE=InnoDB;
            """
        )

        # 2. drivers table (full driver details)
        cursor.execute(
            """
            CREATE TABLE IF NOT EXISTS drivers (
                id INT AUTO_INCREMENT PRIMARY KEY,
                full_name VARCHAR(100) NOT NULL,
                address VARCHAR(255),
                phone VARCHAR(20),
                email VARCHAR(100) UNIQUE,
                license_number VARCHAR(50),
                vehicle_number VARCHAR(50),
                status ENUM('available', 'busy', 'inactive') DEFAULT 'available',
                created_at TIMESTAMP DEFAULT CURRENT_TIMESTAMP,
                updated_at TIMESTAMP DEFAULT CURRENT_TIMESTAMP
                    ON UPDATE CURRENT_TIMESTAMP
            ) ENGINE=InnoDB;
            """
        )

        # 3. users table (auth + role; references customers/drivers)
        cursor.execute(
            """
            CREATE TABLE IF NOT EXISTS users (
                id INT AUTO_INCREMENT PRIMARY KEY,
                username VARCHAR(50) NOT NULL UNIQUE,
                password_hash VARCHAR(255) NOT NULL,
                role ENUM('customer', 'driver', 'admin') NOT NULL,
                customer_id INT NULL,
                driver_id INT NULL,
                is_active TINYINT(1) DEFAULT 1,
                created_at TIMESTAMP DEFAULT CURRENT_TIMESTAMP,
                updated_at TIMESTAMP DEFAULT CURRENT_TIMESTAMP
                    ON UPDATE CURRENT_TIMESTAMP,
                CONSTRAINT fk_users_customer
                    FOREIGN KEY (customer_id) REFERENCES customers(id)
                    ON DELETE SET NULL,
                CONSTRAINT fk_users_driver
```

main.py | db_connector.py

dataacesslayer > db_connector.py > Database > __init__

```
class Database:
    def init_schema(self):
                    ON UPDATE CURRENT_TIMESTAMP,
                CONSTRAINT fk_users_customer
                    FOREIGN KEY (customer_id) REFERENCES customers(id)
                    ON DELETE SET NULL,
                CONSTRAINT fk_users_driver
                    FOREIGN KEY (driver_id) REFERENCES drivers(id)
                    ON DELETE SET NULL
            ) ENGINE=InnoDB;
            """
        )

        # 4. bookings table
        cursor.execute(
            """
            CREATE TABLE IF NOT EXISTS bookings (
                id INT AUTO_INCREMENT PRIMARY KEY,
                customer_id INT NOT NULL,
                pickup_location VARCHAR(255) NOT NULL,
                dropoff_location VARCHAR(255) NOT NULL,
                pickup_datetime DATETIME NOT NULL,
                status ENUM('pending', 'assigned', 'ongoing', 'completed', 'cancelled')
                    DEFAULT 'pending',
                driver_id INT NULL,
                fare DECIMAL(10, 2) NULL,
                notes TEXT,
                created_at TIMESTAMP DEFAULT CURRENT_TIMESTAMP,
                updated_at TIMESTAMP DEFAULT CURRENT_TIMESTAMP
                    ON UPDATE CURRENT_TIMESTAMP,
                FOREIGN KEY (customer_id) REFERENCES customers(id)
                    ON DELETE CASCADE,
                FOREIGN KEY (driver_id) REFERENCES drivers(id)
                    ON DELETE SET NULL,
                INDEX idx_booking_driver_datetime (driver_id, pickup_datetime)
            ) ENGINE=InnoDB;
            """
        )

        conn.commit()
        cursor.close()
        print("Database schema initialised successfully.")
```

services

booking_service.py

main.py | booking_service.py

services > booking_service.py > BookingService > complete_ride

```python
# app/services/booking_service.py

from typing import Optional, List, Dict, Any
from datetime import datetime

from dataacesslayer.db_connector import Database
from dataacesslayer.booking_dal import BookingDAL
from dataacesslayer.driver_dal import DriverDAL


class BookingService:
    """
    Handles booking-related business logic:
    - customers: create/update/cancel/view bookings
    - admin: view all bookings, assign drivers (with overlap check)
    """

    def __init__(self, db: Database):
        self.db = db
        self.booking_dal = BookingDAL(db)
        self.driver_dal = DriverDAL(db)

    # ----------------- internal helpers -----------------

    def _parse_datetime(self, dt_str: str) -> datetime:
        """
        Parse a datetime string from user input.
        Expect format 'YYYY-MM-DD HH:MM'
        """
        try:
            return datetime.strptime(dt_str, "%Y-%m-%d %H:%M")
        except ValueError:
            raise ValueError("Invalid datetime format. Use 'YYYY-MM-DD HH:MM'.")

    # ----------------- customer-facing methods -----------------

    def create_booking_for_customer(
        self,
        customer_id: int,
        pickup_location: str,
        dropoff_location: str,
        pickup_datetime_str: str,
        notes: Optional[str] = None,
    ) -> int:
        """
        Customer books a taxi.
        """
        pickup_dt = self._parse_datetime(pickup_datetime_str)

        booking_id = self.booking_dal.create_booking(
            customer_id=customer_id,
            pickup_location=pickup_location,
            dropoff_location=dropoff_location,
            pickup_datetime=pickup_dt,
            notes=notes,
        )
        return booking_id

    def get_customer_bookings(self, customer_id: int) -> List[Dict[str, Any]]:
```

```python
    def get_customer_bookings(self, customer_id: int) -> List[Dict[str, Any]]:
        """
        Return all bookings for a given customer.
        """
        return self.booking_dal.list_by_customer(customer_id)

    def cancel_booking(self, booking_id: int, customer_id: int) -> None:
        """
        Customer cancels their own booking.
        """
        booking = self.booking_dal.get_by_id(booking_id)
        if not booking:
            raise ValueError("Booking not found.")

        if booking["customer_id"] != customer_id:
            raise PermissionError("You can only cancel your own bookings.")

        if booking["status"] in ("completed", "cancelled"):
            raise ValueError(f"Cannot cancel a booking with status '{booking['status']}'.")

        self.booking_dal.cancel_booking(booking_id)

    def update_booking(
        self,
        booking_id: int,
        customer_id: int,
        pickup_location: str,
        dropoff_location: str,
        pickup_datetime_str: str,
        notes: Optional[str] = None,
    ) -> None:
        """
        Customer updates a booking.
        """
        booking = self.booking_dal.get_by_id(booking_id)
        if not booking:
            raise ValueError("Booking not found.")

        if booking["customer_id"] != customer_id:
            raise PermissionError("You can only update your own bookings.")

        if booking["status"] != "pending":
            raise ValueError("Only 'pending' bookings can be updated.")

        pickup_dt = self._parse_datetime(pickup_datetime_str)

        self.booking_dal.update_booking(
            booking_id=booking_id,
            pickup_location=pickup_location,
            dropoff_location=dropoff_location,
            pickup_datetime=pickup_dt,
            notes=notes,
        )

    # ----------------- admin / driver methods -----------------

```

services > booking_service.py > BookingService > complete_ride

```python
class BookingService:
    # ----------------- admin / driver methods -----------------

    def get_all_bookings(self) -> List[Dict[str, Any]]:
        return self.booking_dal.list_all()

    # Backwards-compatible alias for existing UI code that calls list_all()
    def list_all(self) -> List[Dict[str, Any]]:

        return self.get_all_bookings()

    def get_bookings_for_driver(self, driver_id: int) -> List[Dict[str, Any]]:

        return self.booking_dal.list_by_driver(driver_id)

    # Backwards-compatible name for UI code that calls get_driver_bookings
    def get_driver_bookings(self, driver_id: int) -> List[Dict[str, Any]]:

        return self.get_bookings_for_driver(driver_id)

    def assign_driver_to_booking(
        self,
        booking_id: int,
        driver_id: int,
    ) -> None:
        """
        Admin assigns a driver to a booking.
        """
        booking = self.booking_dal.get_by_id(booking_id)
        if not booking:
            raise ValueError("Booking not found.")

        if booking["status"] in ("cancelled", "completed"):
            raise ValueError(f"Cannot assign driver to a booking with status '{booking['status']}'.")

        driver = self.driver_dal.get_by_id(driver_id)
        if not driver:
            raise ValueError("Driver not found.")
        # Check that driver has no other active booking
        if self.booking_dal.has_active_booking_for_driver(driver_id):
            raise ValueError(
                "This driver already has an active booking. "
                "They must complete the current ride before a new one can be assigned."
            )

        # If no active booking, assign driver
        self.booking_dal.assign_driver(booking_id, driver_id)

    # Backwards-compatible alias for UI code that calls assign_driver(...)
    def assign_driver(self, booking_id: int, driver_id: int) -> None:

        self.assign_driver_to_booking(booking_id, driver_id)

    # ----------------- ride lifecycle for drivers -----------------

    def start_ride(self, booking_id: int, driver_id: int) -> None:
        booking = self.booking_dal.get_by_id(booking_id)
        if not booking:
```

```
        booking = self.booking_dal.get_by_id(booking_id)
        if not booking:
            raise ValueError("Booking not found.")

        if booking.get("driver_id") != driver_id:
            raise PermissionError("You can only start rides assigned to you.")

        if booking.get("status") != "assigned":
            raise ValueError("Only 'assigned' bookings can be started.")

        # Update booking and driver statuses
        self.booking_dal.update_status(booking_id, "ongoing")
        self.driver_dal.update_status(driver_id, "busy")

    def complete_ride(self, booking_id: int, driver_id: int) -> None:
        """
        Driver completes a ride
        """
        booking = self.booking_dal.get_by_id(booking_id)
        if not booking:
            raise ValueError("Booking not found.")

        if booking.get("driver_id") != driver_id:
            raise PermissionError("You can only complete rides assigned to you.")

        if booking.get("status") != "ongoing":
            raise ValueError("Only 'ongoing' bookings can be completed.")

        # Update booking and driver statuses
        self.booking_dal.update_status(booking_id, "completed")
        self.driver_dal.update_status(driver_id, "available")

```

customer_service.py

main.py | booking_service.py | customer_service.py

services > customer_service.py > CustomerService

```python
from typing import List, Dict, Any, Optional

from dataacesslayer.db_connector import Database
from dataacesslayer.customer_dal import CustomerDAL


class CustomerService:

    def __init__(self, db: Database):
        self.db = db
        self.customer_dal = CustomerDAL(db)

    def list_all(self) -> List[Dict[str, Any]]:
        """Return all customers."""
        return self.customer_dal.list_all()

    def get_by_id(self, customer_id: int) -> Optional[Dict[str, Any]]:
        """Get a single customer by ID."""
        return self.customer_dal.get_by_id(customer_id)
```

driver_service.py

main.py | booking_service.py | customer_service.py | driver_service.py

services > driver_service.py > ...

```python
from typing import List, Dict, Any, Optional

from dataacesslayer.db_connector import Database
from dataacesslayer.driver_dal import DriverDAL


class DriverService:


    def __init__(self, db: Database):
        self.db = db
        self.driver_dal = DriverDAL(db)

    def list_all(self) -> List[Dict[str, Any]]:
        """Return all drivers."""
        return self.driver_dal.list_all()

    def list_available(self) -> List[Dict[str, Any]]:
        """Return drivers whose status is 'available'."""
        return self.driver_dal.list_available()

    def get_by_id(self, driver_id: int) -> Optional[Dict[str, Any]]:
        """Get a single driver by ID."""
        return self.driver_dal.get_by_id(driver_id)

    def update_status(self, driver_id: int, status: str) -> None:
        """Update driver status ('available', 'busy', 'inactive')."""
        self.driver_dal.update_status(driver_id, status)


```

user_service.py

main.py | booking_service.py | customer_service.py | driver_service.py | user_services.py

services > user_services.py > UserService > register_customer

```python
# app/services/user_service.py

from typing import Optional, Dict, Any
import hashlib

from dataacesslayer.db_connector import Database
from dataacesslayer.user_dal import UserDAL
from dataacesslayer.customer_dal import CustomerDAL
from dataacesslayer.driver_dal import DriverDAL


class UserService:
    """
    Uses DAL classes to communicate to the database.
    """

    def __init__(self, db: Database):
        self.db = db
        self.user_dal = UserDAL(db)
        self.customer_dal = CustomerDAL(db)
        self.driver_dal = DriverDAL(db)

    # -------------- internal helper ----------------

    def _hash_password(self, plain_password: str) -> str:

        return hashlib.sha256(plain_password.encode("utf-8")).hexdigest()

    # -------------- registration ----------------

    def register_customer(
        self,
        full_name: str,
        address: str,
        phone: str,
        email: str,
        username: str,
        password: str,
    ) -> int:

        # Check if email already exists as a customer
        existing_customer = self.customer_dal.get_by_email(email)
        if existing_customer:
            raise ValueError("A customer with this email already exists.")

        # Check if username already exists
        existing_user = self.user_dal.get_by_username(username)
        if existing_user:
            raise ValueError("Username is already taken.")

        # 1) create customer
        customer_id = self.customer_dal.create_customer(
            full_name=full_name,
            address=address,
            phone=phone,
            email=email,
        )

```

```
        )

        # 2) hash password and create user with role 'customer'
        password_hash = self._hash_password(password)

        user_id = self.user_dal.create_user(
            username=username,
            password_hash=password_hash,
            role="customer",
            customer_id=customer_id,
            driver_id=None,
        )

        return user_id

    def register_driver(
        self,
        full_name: str,
        address: str,
        phone: str,
        email: str,
        username: str,
        password: str,
        license_number: str,
        vehicle_number: str,
    ) -> int:
        """
        Register a new driver.
        Returns the new user ID.
        """

        existing_driver = self.driver_dal.get_by_email(email)
        if existing_driver:
            raise ValueError("A driver with this email already exists.")

        existing_user = self.user_dal.get_by_username(username)
        if existing_user:
            raise ValueError("Username is already taken.")

        # 1) create driver
        driver_id = self.driver_dal.create_driver(
            full_name=full_name,
            address=address,
            phone=phone,
            email=email,
            license_number=license_number,
            vehicle_number=vehicle_number,
            status="available",
        )

        # 2) create user with role 'driver'
        password_hash = self._hash_password(password)

        user_id = self.user_dal.create_user(
            username=username,
            password_hash=password_hash,
```

```
            role="driver",
            customer_id=None,
            driver_id=driver_id,
        )

        return user_id

    # -------------- login ----------------

    def login(self, username: str, password: str) -> Optional[Dict[str, Any]]:
        """
        Validate user credentials.
        """

        user = self.user_dal.get_by_username(username)
        if not user:
            return None

        if not user.get("is_active", 1):
            # user exists but is deactivated
            return None

        input_hash = self._hash_password(password)
        if input_hash != user["password_hash"]:
            return None

        role = user.get("role")

        if role == "customer" and user.get("customer_id"):
            # Fetch customer data and merge into user object
            customer = self.customer_dal.get_by_id(user["customer_id"])
            if customer:
                original_user_id = user.get("id")
                user.update(customer)
                user["id"] = user["customer_id"]
                user["user_id"] = original_user_id
        elif role == "driver" and user.get("driver_id"):
            # Fetch driver data and merge into user object
            driver = self.driver_dal.get_by_id(user["driver_id"])
            if driver:
                original_user_id = user.get("id")
                user.update(driver)
                user["id"] = user["driver_id"]
                user["user_id"] = original_user_id

        return user
```