Taxi Booking System

Mathematics and Concepts for Computational Thinking (CIS093-1)

AN ASSIGMENT SUBMITTED IN THE PARTIAL FULFILLMENT OF THE REQUIREMENTS FOR THE DEGREE OF BACHELOR OF SOFTWARE ENGINEERING

Submitted By:

Nirajan Dhungel (2425686)

Submitted To:

Department of Computer Science

University of Bedfordshire

Date: 18 December, 2024

Table of Contents

Table of Figures

Table of Tables

Introduction

A complete desktop app, Taxi Booking System using Python programming language for core logics, Custom Tkinter for GUI (Graphical User Interface) and My SQL for data storage is developed in order to facilitate transportation services.

The work began with proper planning of the system's structure, defining the roles and functionality of respective roles. A clean structure of the system was designed using UML diagrams, Front-End developed using Custom Tkinter and Python for backend functionality, managing user data and handling booking process. The system ensures dynamic changes of UI components through database communication, smooth navigation between different user interfaces.

Displaying correct details of users after successfully login, along with smooth navigation and dynamic changes of UI components was a major challenge. Additional challenges arose when extra features such as editing user's credentials, delete drivers, implementing hashed password etc. were added to the system. To fix these issues, techniques like step-by-step debugging, checking error messages and handling database errors were used.

After all the issues were handled, the system finally met the defined requirements. All three user roles (drivers, passengers and admins) were able to perform their specific tasks with better user experience and without any major issues. The application of decomposition, pattern recognition, abstraction, and algorithm design helped to make the system work more effectively and significantly eased the development process.

Task Description

Objective

The primary goal of this task is to develop a taxi booking system using Python, applying principles of computational thinking to manage user interaction and system operations and designing the system with UML diagrams.

System Features

The system should include three types of users (Customers, Drivers and Administrators) and all the users should have access to the system after successfully login. Depending upon the role of users, users should have different features.

- The system shall provide features related to Customers, Drivers, and Administrators, role-specific, with role-based access post-login.
- The user interface should be intuitive and easy to navigate, using text-based menus or Tkinter, allowing users to complete tasks effortlessly.
- Management of data should ensure safekeeping of customer, driver, and booking information in text files, SQLite, or MySQL databases.
- Design should be guided by computational thinking: task decomposition, pattern recognition for reusable code, data abstraction of essentials, and development of algorithms.
- UML diagrams, such as Use Case, Activity, and Class diagrams, shall depict user interactions, booking processes, and system design graphically.
- Testing and validation should verify all functionalities, such as user registration, login, and bookings, ensuring the system operates reliably under different scenarios.

Project Planning

1. Requirements analysis:

Property	Details
Description	Understand the requirements of project: easy navigation, secure login, role-based dashboard, easy booking process, and driver management.
Tasks	-Gather functional requirements. -Define Technologies: Python, Custom Tkinter, and My SQL.
Days	Day 1-2
Priority	Medium

Table 1: Requirements analysis

2. System Design:

Property	Details
Description	Create the system architecture and design components.
Tasks	-Design UI components using Figma. - Develop Use Case, Activity, and Class Diagrams for system structure and interactions.
Days	Day 3-5
Priority	High

Table 2: System Design

3. Development:

Property	Details
Description	Develop the application based on design.
Tasks	-Code the GUI using Custom Tkinter. -Develop backend functionality e.g. user authentication, booking management etc. -Integrate My SQL with Python for database interactions.
Days	Day 6-13
Priority	High

Table 3: System development

4. Testing and Debugging:

Property	Details
Description	Conduct testing to ensure functionality and resolve issues
Tasks	-Black box testing for individual modules. -White box testing for integrated modules.
Days	Day 14-15
Priority	High

Table 4: Testing and Debugging

Requirement Analysis

Functional Requirements:

1. User Roles

This system empowers customers, drivers, and administrators to perform some of their tasks with greater effectiveness. The passenger can sign up by providing details. Passenger can book a taxi in a specified pickup location, dropping location, date, and time. Besides, the passengers can view their bookings along with the driver's name and phone number, cancel or mark a booking as complete, and access other features like viewing booking history and editing their profile. The driver can request registration by filling in full name, email, phone number, address, gender, vehicle number, license number, and password to be sent to the administrator for approval. They can thus view assigned trips with passenger details, with options to view trip history and edit their profiles. The administrator's role is then major in approving or rejecting driver registrations, assigning drivers to trips, viewing the history of rides, and managing drivers by deleting their profiles if need be.

2. Core System Features

The system should have secure login for all user roles and access to their respective dashboard with history. In the case of booking management, passengers can book, view, and cancel or complete trips. The administrator is responsible for managing the bookings and assigning drivers to them, ensuring proper coordination. Drivers, on the other hand, receive a clear schedule of trips if they are assigned any, enabling them to manage time and trips effectively.

4. Graphical User Interface

A well-designed, graphical user interface is crucial for better user experience. The system should have a friendly layout that will easily guide a user through various functionalities. By using Custom Tkinter for customizing the interface, the system can be developed based on the design. This will not only make the system easily accessible but also enhance user interactions, whereby booking a taxi or dealing with user accounts can easily be done without any mess.

3. Data Management

The system should store important data, like passenger information, booking details, and driver information, in a MySQL database. In this way, the data can be managed efficiently and can be retrieved or updated whenever needed with ease to maintain accuracy and consistency throughout the application.

Non-Functional Requirements

1. Performance

The system should support multiple user interactions at the same time without deteriorating in performance.

2. Scalability

The database structure should be such that it will be able to hold more and more users and their bookings. This means the system will easily scale up or down depending on demand.

3. Security

The system shall implement strong security measures regarding data privacy. Authentication mechanisms should also be provided for all users in a secure way to protect sensitive information.

5. Reliability

The system should minimize errors regarding booking assignments; all information should be handled accurately, with reliability to maintain data integrity.

Usability Requirements

Usability requirements are very important in this system because there are a variety of user roles that interact with the system differently, including customers, drivers, and administrators. Ease of use, error prevention, and feedback are some of the aspects which helps to avoid mistakes and provide satisfying experiences for all kinds of users. Strong usability design reduces the learning curve, enhances task efficiency, and develops overall user satisfaction, making the system more practical and reliable.

- Users should easily operate the system-for instance, registering or booking a taxi-within a few minutes, with prompts and examples provided where necessary.
- Common tasks such as booking a taxi or assigning a driver must be fast and no longer than three steps. Data should load promptly within two seconds.
- Inputs should be validated against errors, and users should receive clear error messages with suggestions for corrections.
- Successful operations will return confirmation messages, such as "Booking confirmed," and failure notifications on why the operation failed.

System Design

The system should be designed in such a way that that would be easy for any user to understand and operate. It will include UML diagrams such as Use Case Diagrams to show the interaction of users, Activity Diagrams to outline steps for booking, and Class Diagrams to illustrate the structure of the system.

- The system must provide role-based features for Customers, Drivers, and Administrators after successful login.
- The user interface should be simple and easy to navigate. Custom Tkinter will be in use to enable users to take care of every activity efficiently.
- Data management shall securely store information related to customers, drivers, and bookings in a MySQL database.
- Computational thinking shall guide the design process through task breakdown, reuse of patterns, simplification of key information, and creation of efficient algorithms.

UML Diagrams

UML diagrams (Use Case, Activity, and Class) will visually represent user interactions, booking processes, and system design.

Activity Diagram

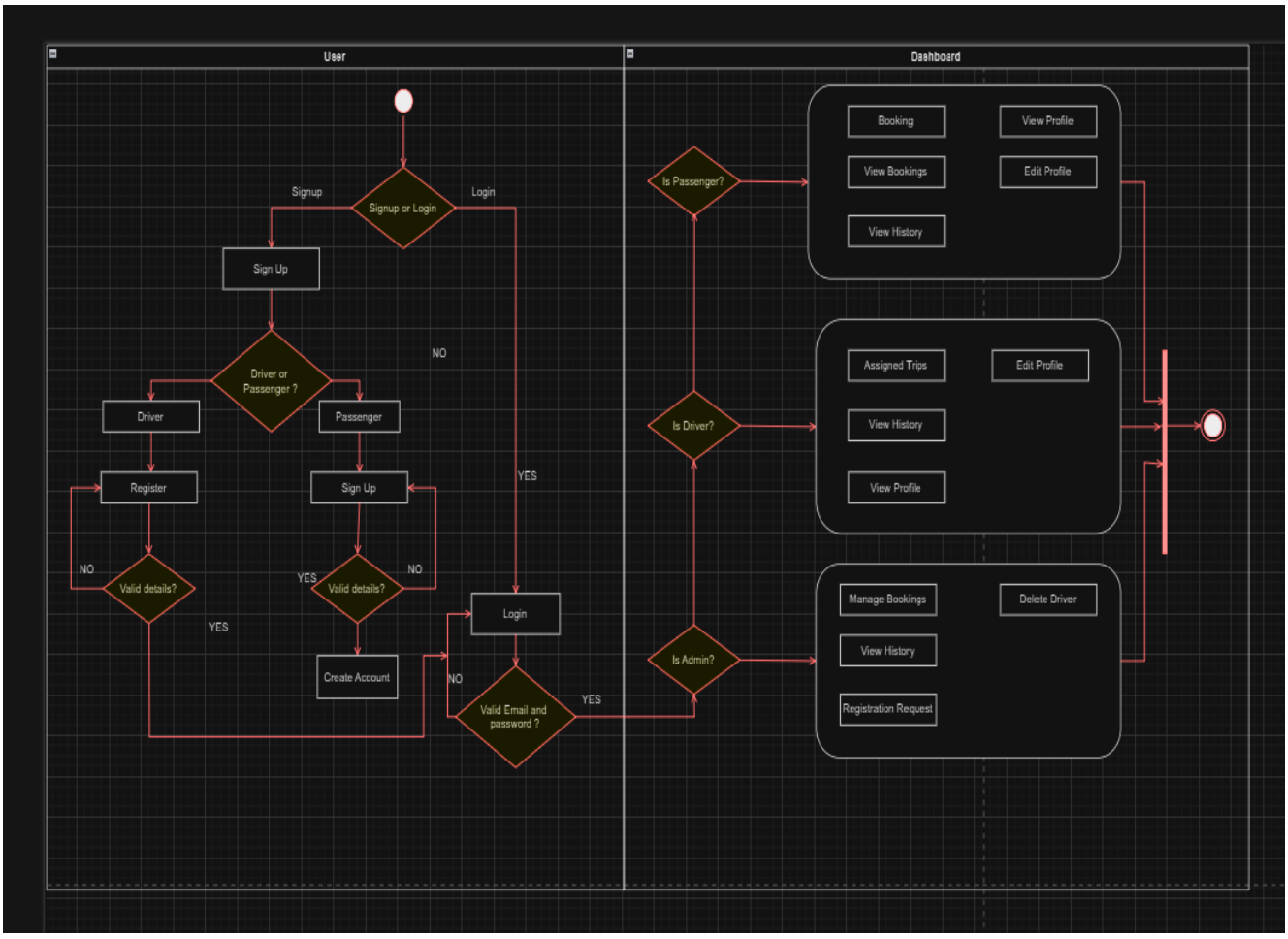


Figure 1: Activity Diagram of taxi booking system

The above activity diagram shows the flow of a taxi booking system for passenger, the driver, and the admin. A user can either sign up or log in. At the time of signing up, the user has to specify his/her role and valid details for sign-up, whereas at login, it requires valid credentials to proceed. After login, the system redirects the users to the role-based dashboards. Passenger users will see the options to book rides, view bookings and trip history, and edit their profile. Drivers can view assigned trips, trip history, and edit their profiles. Admins manage bookings, view trip histories, handle registration requests, and can delete drivers if necessary, ensuring a smooth and organised workflow.

Use Case Diagrams

Sea Level:

Sea level in UML diagrams shows the main interactions and processes at the system-wide level. It is about core use cases and shows the key functionality of the system without going into technical details.

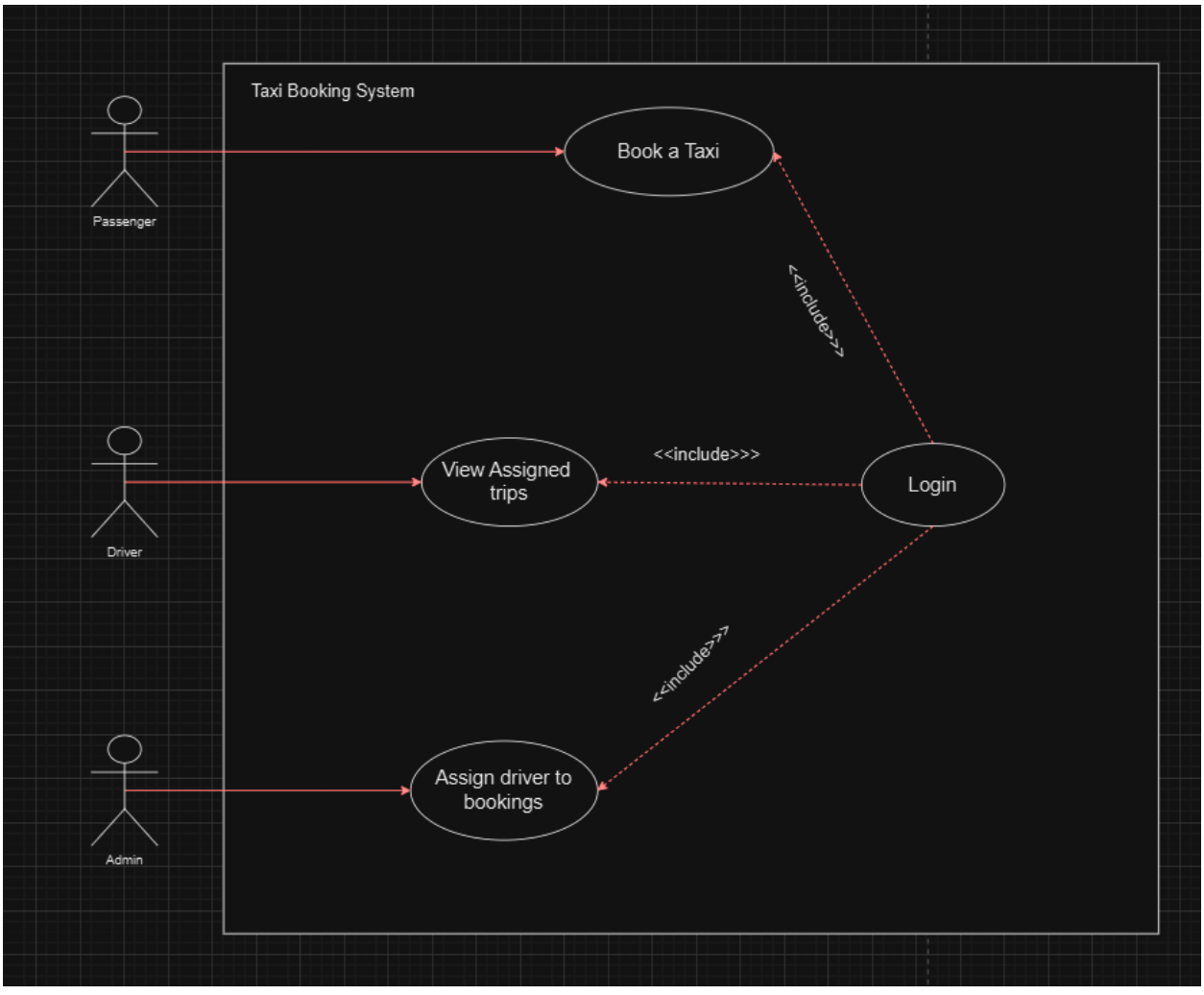


Figure 2: Sea level Use Case Diagram of taxi booking system

The above diagram displays some required functions of the Taxi Booking System, where there are three main actors (Passenger, Driver, and Admin). Each of these actors has to log in in order to access their functional capabilities, namely booking a taxi (Passenger), viewing the assigned trips (Driver), and assigning drivers to bookings (Admin). The <<include>> indicates that the other actions depend on logging first.

Fish Level

For the Taxi Booking System, this level is crucial to understand the relationships between key functionalities such as booking, managing trips, and driver assignment.

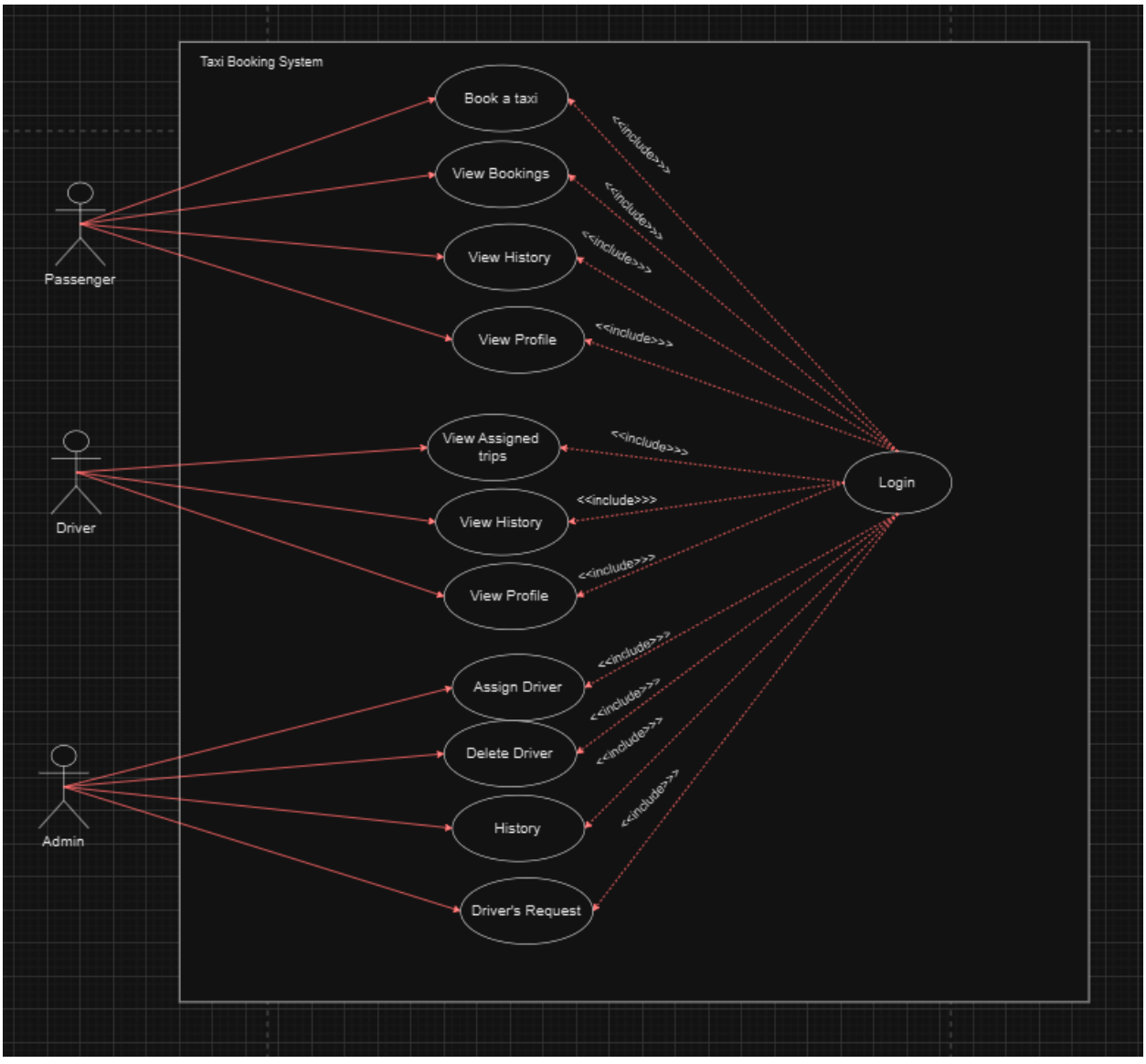


Figure 3: Fish level Use Case Diagram of taxi booking system

This use case diagram shows major functionalities of the taxi booking system. A passenger can log in, book a taxi, view details about the booking, and cancel or complete the booking. Drivers log in to see assigned trips, while Admin manages the bookings by assigning drivers. Each action is interrelated through the "Login" action, with actions like "View bookings" having sub-actions such as "Assign driver".

Clam Level:

1. Clam level use case Diagram of Passenger

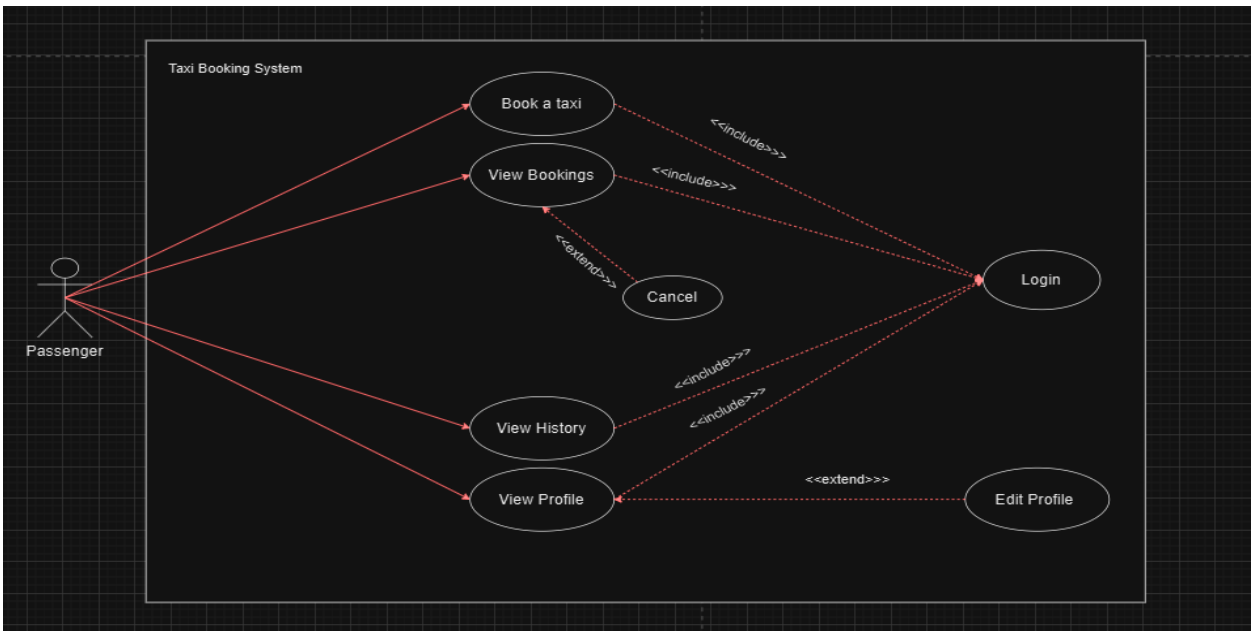


Figure 4:Passenger's Clam level use case Diagram

In this, passengers can log in to book taxis, view bookings and history, edit profiles, and complete or cancel bookings.

2. Clam level use case diagram of Driver

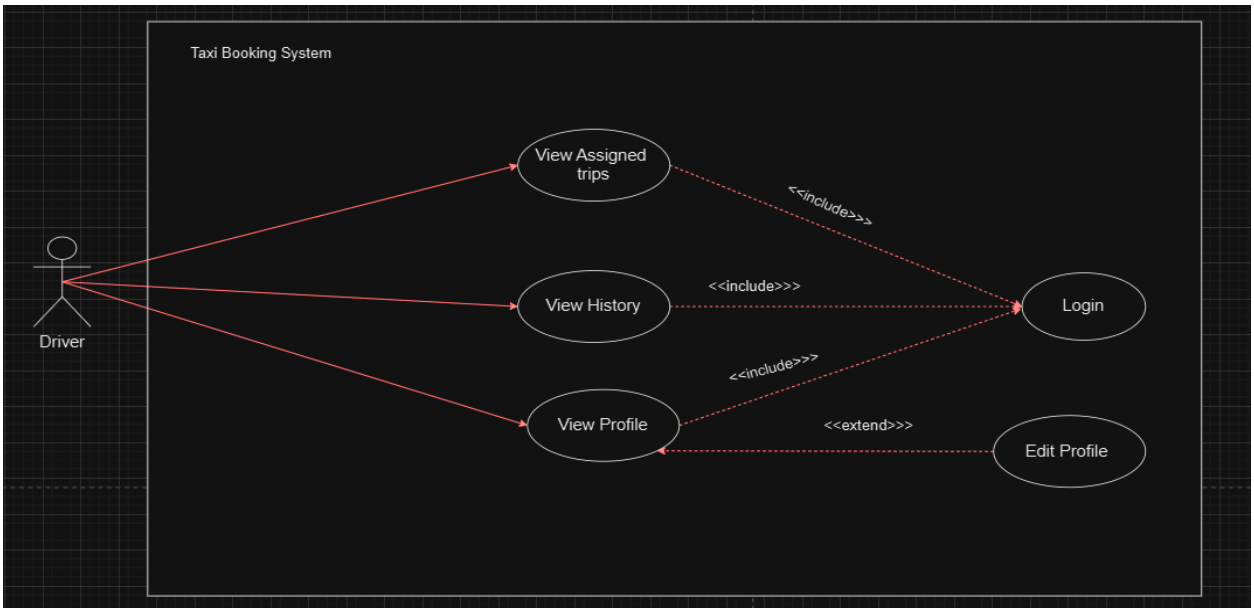


Figure 5: Driver's Clam level use case diagram

The drivers have somewhat similar functionalities, focusing mainly on viewing assigned trips and history, and managing their profiles.

2. Clam level use case diagram of Driver

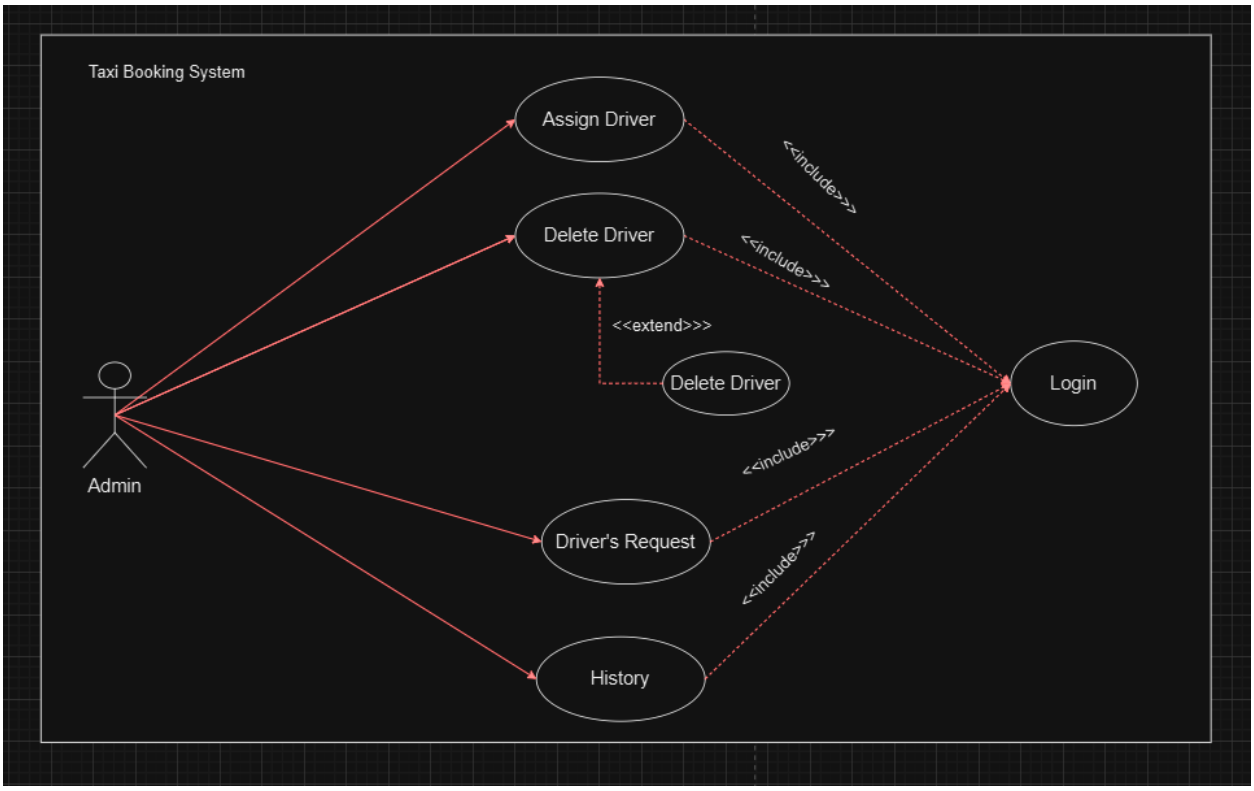


Figure 6: Admin's Calm level use case Diagram

This is a clam-level use case diagram which elaborates on the detailed functionalities of the taxi booking system for the actors. Admins handle tasks such as assigning drivers, deleting drivers, and viewing the system's history. Each of these actions is based on the "Login" process, underlining how central it is to everything.

Class Diagram

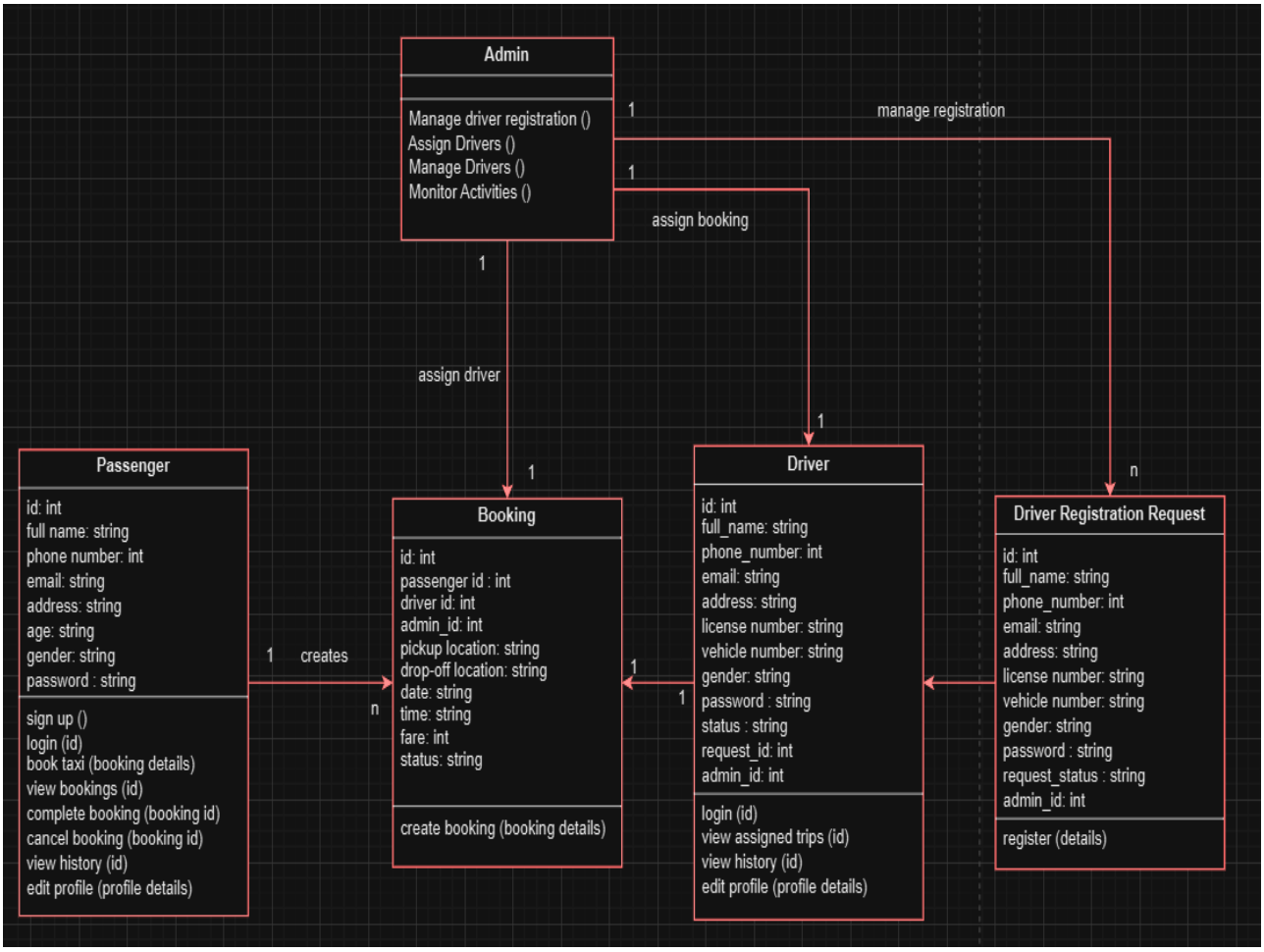


Figure 7: Class Diagram of taxi booking system

The above class diagram represents a Taxi Booking System with four major entities: Admin, Passenger, Driver, and Booking. The passenger can register themselves, log in, book taxis, view bookings, check history, and edit their profile. One passenger can book many bookings. Drivers manage the registration, view the assigned trips, check history, and update the profile of themselves. Admins take care of driver registration and assign bookings to them while monitoring activities. The Booking class links passengers and drivers, maintaining information such as pickup and drop-off locations, fare, and status. Admins oversee the entire system to ensure that passengers and drivers are functioning accordingly. This diagram effectively illustrates the core functionality and the relationships of the system.

Database Design

ER Diagram:

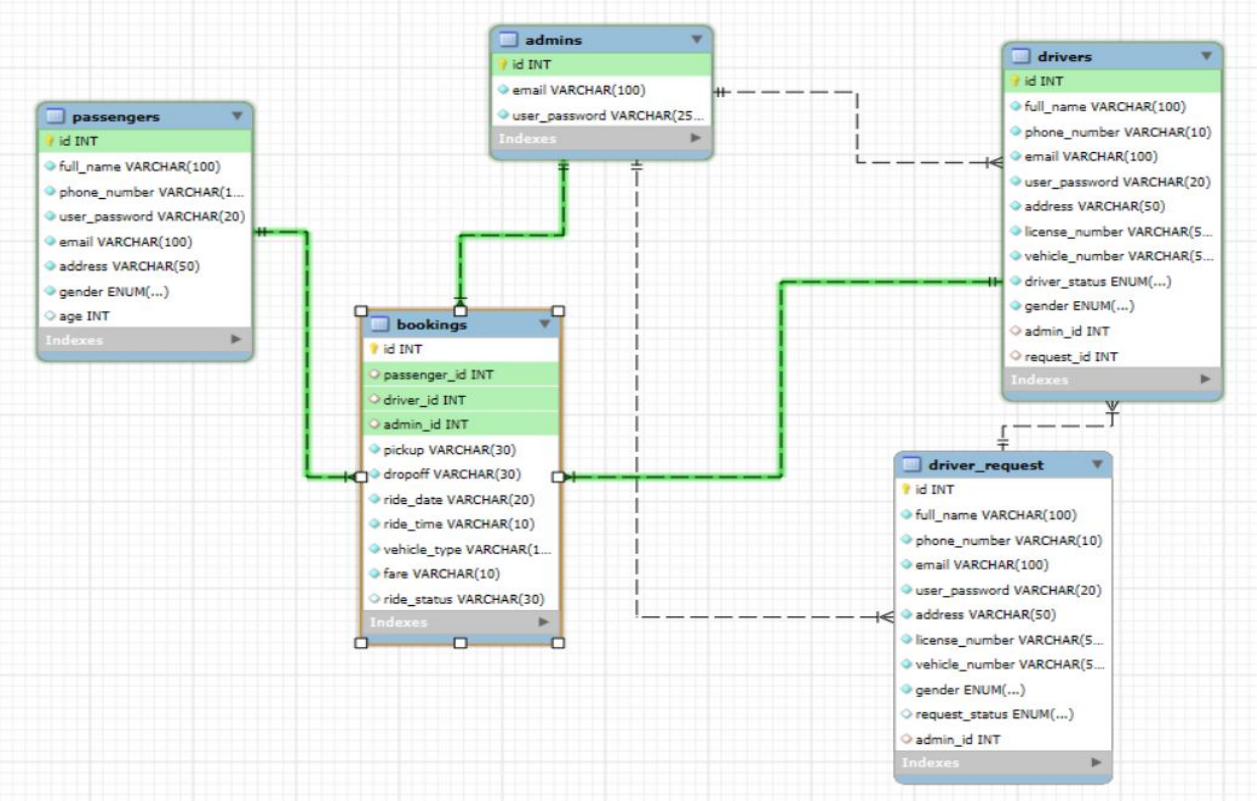


Figure 8: ER Diagram of taxi booking system

This ER diagram represents the taxi booking system with five interconnected entities: passengers, drivers, driver_request, bookings, and admins. The passengers table stores passenger information such as full_name, phone_number, and email. The drivers table captures driver-specific details, including license_number, vehicle_number, and their current driver_status. Before becoming active drivers, the drivers provide their information in the driver_request table, which has similar attributes but also includes a request_status column that stores status of request and the table is linked to an admin for approval through admin_id. The bookings table is the central entity for ride management, connecting passengers, drivers, and admins through passenger_id, driver_id, and admin_id, respectively, while storing the trip details like pickup, dropoff, fare, and ride_status. The admins table contains admin credentials and oversees approvals and system management. Foreign keys, such as admin_id and driver_id, create the relationships between tables to keep users, drivers, and system administrators in contact.

Skeleton Tables

1. Passengers

Field	Type	Null	Key	Default	Extra
id	int	NO	PRI	NULL	auto_increment
full_name	varchar(100)	NO		NULL	
phone_number	varchar(10)	NO		NULL	
user_password	varchar(20)	NO	UNI	NULL	
email	varchar(100)	NO	UNI	NULL	
address	varchar(50)	NO		NULL	
gender	enum('Male','Female','Others')	NO		NULL	
age	int	YES		NULL	

Table 5:Skeleton table of passengers

2. Driver Requests

Field	Type	Null	Key	Default	Extra
id	int	NO	PRI	NULL	auto_increment
full_name	varchar(100)	NO		NULL	
phone_number	varchar(10)	NO		NULL	
email	varchar(100)	NO	UNI	NULL	
user_password	varchar(20)	NO	UNI	NULL	
address	varchar(50)	NO		NULL	
license_number	varchar(50)	NO		NULL	
vehicle_number	varchar(50)	NO		NULL	
gender	enum('Male','Female','Others')	NO		NULL	
request_status	enum('Approved','Requested','Rejected')	YES		NULL	
admin_id	int	YES	MUL	NULL	

Table 6:Skeleton Table of registration request Table

2. Drivers

Field	Type	Null	Key	Default	Extra
id	int	NO	PRI	NULL	auto_increment
full_name	varchar(100)	NO		NULL	
phone_number	varchar(10)	NO		NULL	
email	varchar(100)	NO	UNI	NULL	
user_password	varchar(20)	NO	UNI	NULL	
address	varchar(50)	NO		NULL	
license_number	varchar(50)	NO		NULL	
vehicle_number	varchar(50)	NO		NULL	
driver_status	enum('Online','Offline')	NO		NULL	
gender	enum('Male','Female','Others')	NO		NULL	
admin_id	int	YES	MUL	NULL	
request_id	int	YES	MUL	NULL	

Table 7:Skeleton Table of drivers Table

3. Admins

Field	Type	Null	Key	Default	Extra
id	int	NO	PRI	NULL	auto_increment
email	varchar(100)	NO	UNI	NULL	
user_password	varchar(255)	NO		NULL	

Table 8:Skeleton Table of admins Table

4. Bookings

Field	Type	Null	Key	Default	Extra
id	int	NO	PRI	NULL	auto_increment
passenger_id	int	YES	MUL	NULL	
driver_id	int	YES	MUL	NULL	
admin_id	int	YES	MUL	NULL	
pickup	varchar(30)	NO		NULL	
dropoff	varchar(30)	NO		NULL	
ride_date	varchar(20)	NO		NULL	
ride_time	varchar(10)	NO		NULL	
vehicle_type	varchar(10)	NO		NULL	
fare	varchar(10)	NO		NULL	
ride_status	varchar(30)	YES		NULL	

Table 9:Skeleton Table of bookings Table

The passenger table manages user details like full_name, phone_number, email, and other personal data, having id as the primary key. The driver_request table collects the initial requests of the drivers, including their details and request_status, which can be one of Approved, Requested, or Rejected, with id acting as the request identifier. Once approved, the data goes to the drivers table, where request_id in drivers maps to id in driver_request, ensuring that requests are connected to active drivers. The admin table is used for administrators and their credentials for access control. Finally, the bookings table is the core of the system, as it connects passengers, drivers, and admins through passenger_id, driver_id, and admin_id, respectively, and records the details of every trip, including pickup, dropoff, ride_date, and fare.

- Data Flow: The driver requests are first validated in the driver_request table and handled by admins before being added to the system.
- Bookings Table: It connects passengers, drivers, and admins for trip management.
- Admins' Role: Admins oversee driver approvals and manage system integrity using admin_id in various tables.
- Relationships are made between tables through foreign keys like passenger_id, driver_id, and admin_id.

Data Dictionary

1. Passengers

Field Name	Datatype	Length	Index	Null	Default	Validation	Description
id	int	10	PRIMARY KEY	No	NULL		Uniquely identifier for customer with auto_increment.
full_name	varchar	255		No	NULL		Stores customer's name.
email	varchar	255		No	NULL	Must be in email format containing @gmail.com.	Stores Customer's email.
user_password	varchar	255		No	NULL		Password for login
phone_number	varchar	10		No	NULL	Must be 10 digits and numeric.	Stores customer's phone number.
gender	Enum			No	NULL		Stores customer's gender.

Table 10: Data dictionary of passengers

- **Index:**

Key name	Type	Unique	Column	Null
PRIMARY KEY	BTREE	Yes	id	No

Table 11: Index of passengers

2. Driver

Field Name	Datatype	Length	Index	Null	Default	Validation	Description
id	int	10	PRIMARY KEY	No	NULL		Auto_increment Uniquely identifies drivers.
full_name	varchar	255		No	NULL		Stores driver's full name.
user_password	varchar	255		No	NULL	must be of length between 6-18	Password for login.
phone_number	varchar	10		No	NULL	Must be 10 digits and numeric.	Driver's phone number.
email	varchar	255	UNIQUE	No	NULL	Must be in email format containing @gmail.com.	Stores driver's email.
license_number	varchar	10		No	NULL		Stores vehicle license number.
vehicle_number	varchar	10	UNIQUE	No	NULL		Stores vehicle registration number.
driver_status	enum	15		No	Null		Online /Offline status of diver.
admin_id	int	10		No	NULL		admin id who approves registration
request_id	int	10		No	NULL		request id from request table

Table 12: Data dictionary of drivers

- **Index:**

Key name	Type	Unique	Column	Null
PRIMARY KEY	BTREE	Yes	id	No
FOREIGN KEY	BTREE	Yes	admin_id	No
FOREIGN KEY	BTREE	Yes	request_id	No

Table 13: Index of drivers

3. Driver's request

Field Name	Datatype	Length	Index	Null	Default	Validation	Description
id	int	10	PRIMARY KEY	No	NULL		Auto_increment Uniquely identifies request of drivers.
full_name	varchar	255		No	NULL		Stores driver's full name.
user_password	varchar	255		No	NULL	must be of length between 6-18	Password for login.
phone_number	varchar	10		No	NULL	Must be 10 digits and numeric.	Driver's phone number.
email	varchar	255	UNIQUE	No	NULL	Must be in email format containing @gmail.com.	Stores driver's email.
license_number	varchar	10		No	NULL		Stores vehicle license number.
vehicle_number	varchar	10	UNIQUE	No	NULL		Stores vehicle registration number.
request_status	enum	15		No	Null		Online /Offline status of diver.
admin_id	int	10		No	NULL		admin id who approves registration

Table 14: Data dictionary of driver's request table

- **Index:**

Key name	Type	Unique	Column	Null
PRIMARY KEY	BTREE	Yes	id	No
FOREIGN KEY	BTREE	Yes	admin_id	No

Table 15: Index of driver's request table

4. Admin

Field	Datatype	Length	Index	Null	Default	Validation	Description
id	int	10	PRIMARY KEY	No	NULL		Auto_increment Uniquely identifies admin.
email	varchar	255					Stores admin's name.
password	varchar	355		No	NULL		Stores hashed password

Table 16: Data dictionary of Admins

- **Index:**

Key name	Type	Unique	Column	Null
PRIMARY KEY	BTREE	Yes	id	No

Table 17: Index of admin

5. Bookings

Field	Datat ype	Length	Index	Null	Default	Validation	Description
id	int	10	PRIMARY KEY	No	NULL		Uniquely identifier for bookings with auto_increment.
passenger_id	int	10		No	NULL		Stores passenger's id
driver	int	10		No	NULL		Stores driver's id if assigned
admin_id	varch ar	10		No	NULL		Stores admin's id if admin assigns
pickup	varch ar	255		No	NULL	Must be 10 digits and numeric.	Stores pickup location.
dropoff	Enum	255		No	NULL		Stores drop off location.
ride_data	varch ar	255		No	NULL		date of ride
ride_time	varch ar	255		No	NULL		time of ride
fare	varch ar	255		No	NULL		fare of ride
vehicle_type	varch ar	255		No	NULL		vehicle type of ride
ride_status	varch ar	255		No	NULL		status of ride

Table 18: Data dictionary of bookings

- **Index:**

Key name	Type	Unique	Column	Null
PRIMARY KEY	BTREE	Yes	id	No

Table 19: Index of bookings

Use Interfaces

- Starting page of the system

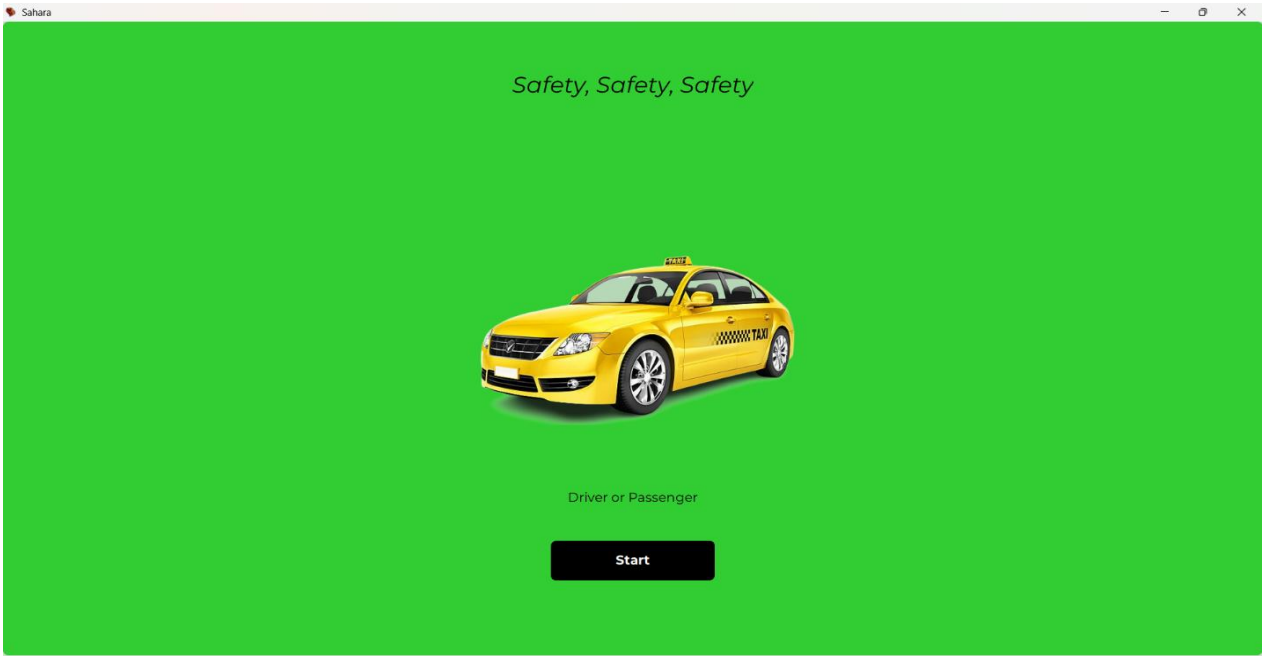


Figure 9: Starting page

- This page directs to login page and new account page

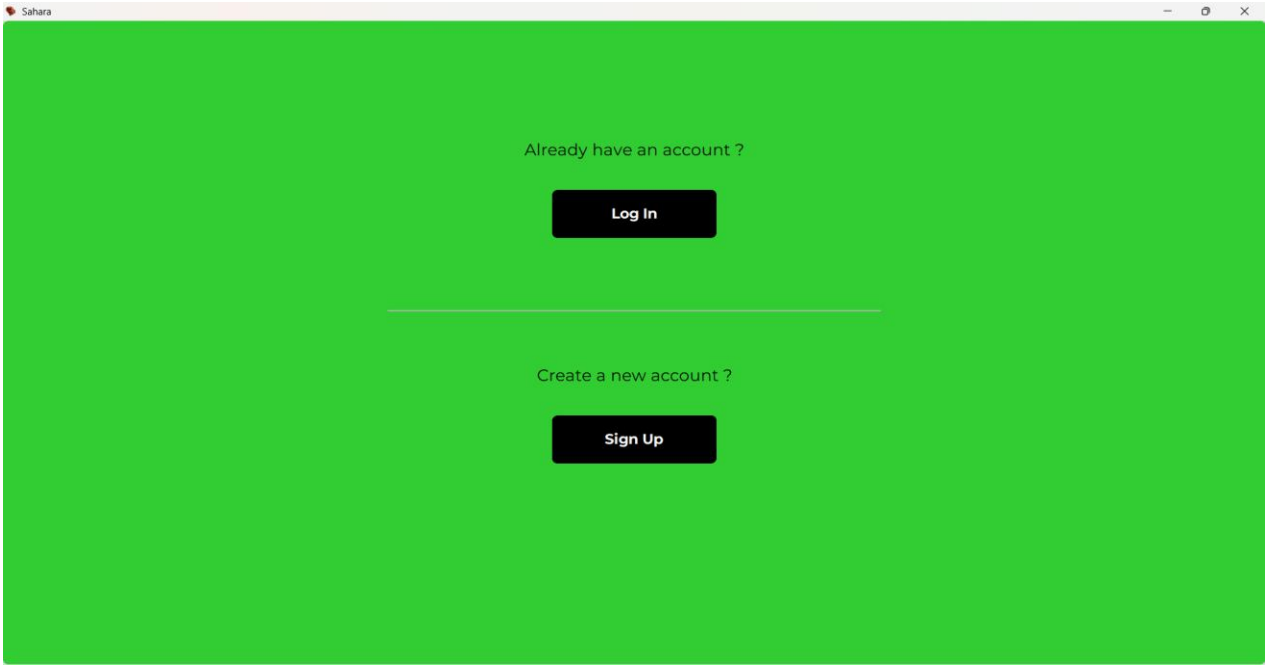


Figure 10: Option for login or signup

- Login page of the system

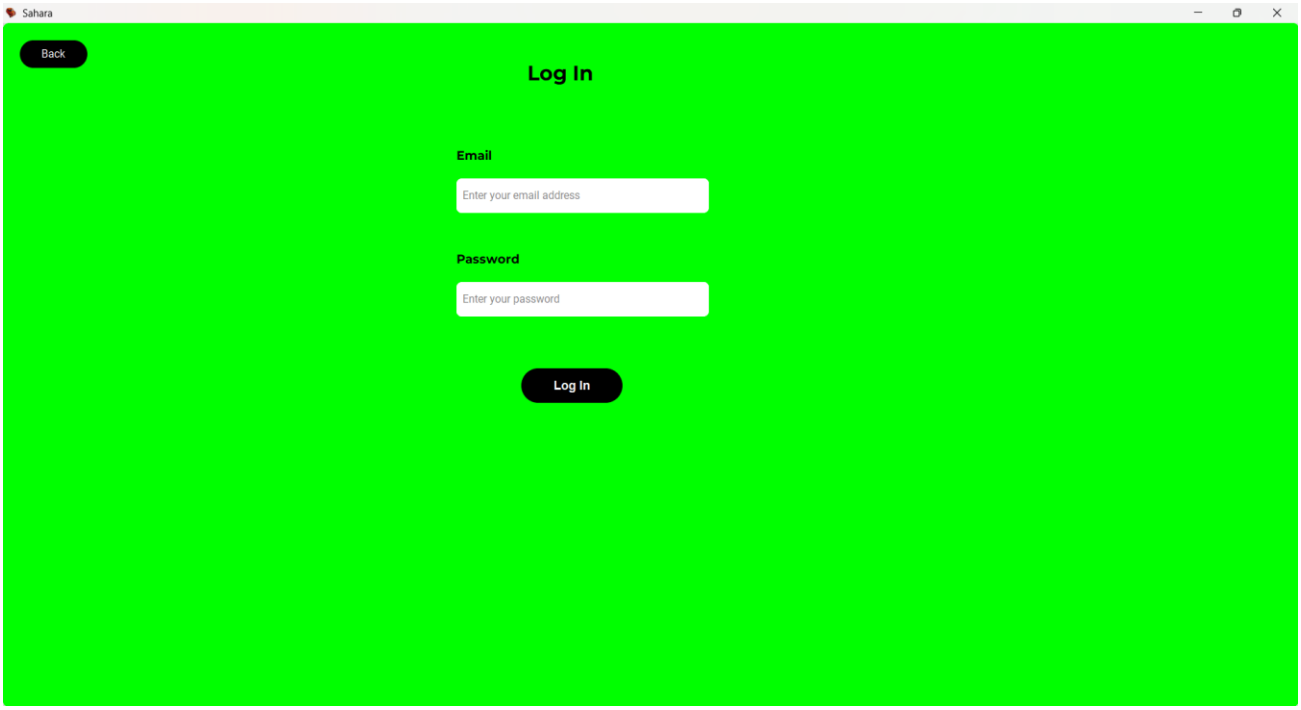


Figure 11: Login page

- This page divides driver and passenger to create a new account

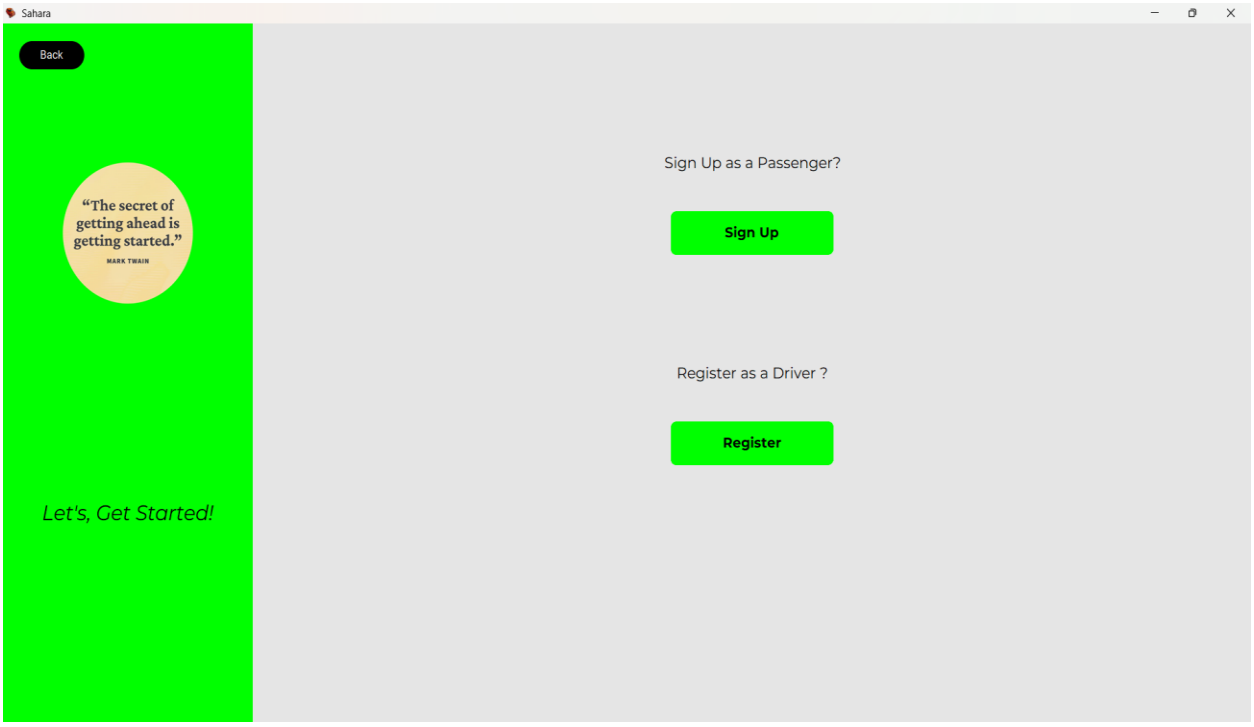


Figure 12: Signup and register options

- Passenger's signup page

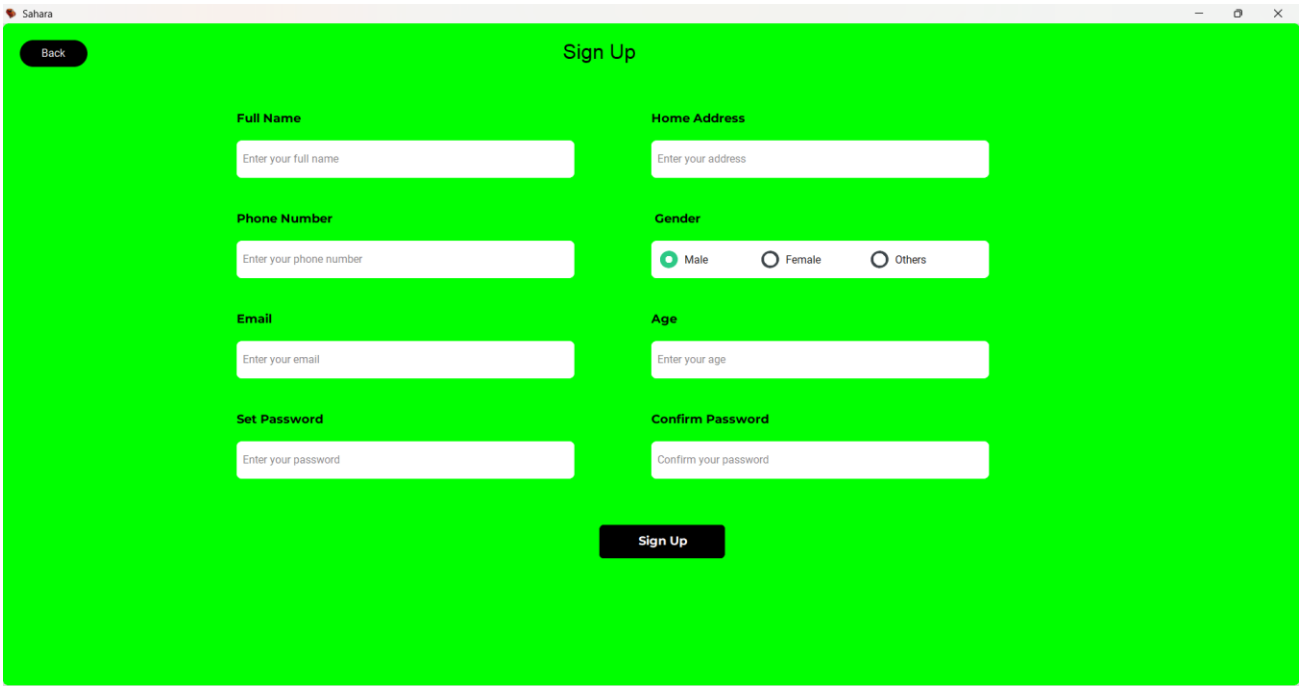


Figure 13: Signup Page for Passengers

- Driver's Registration page

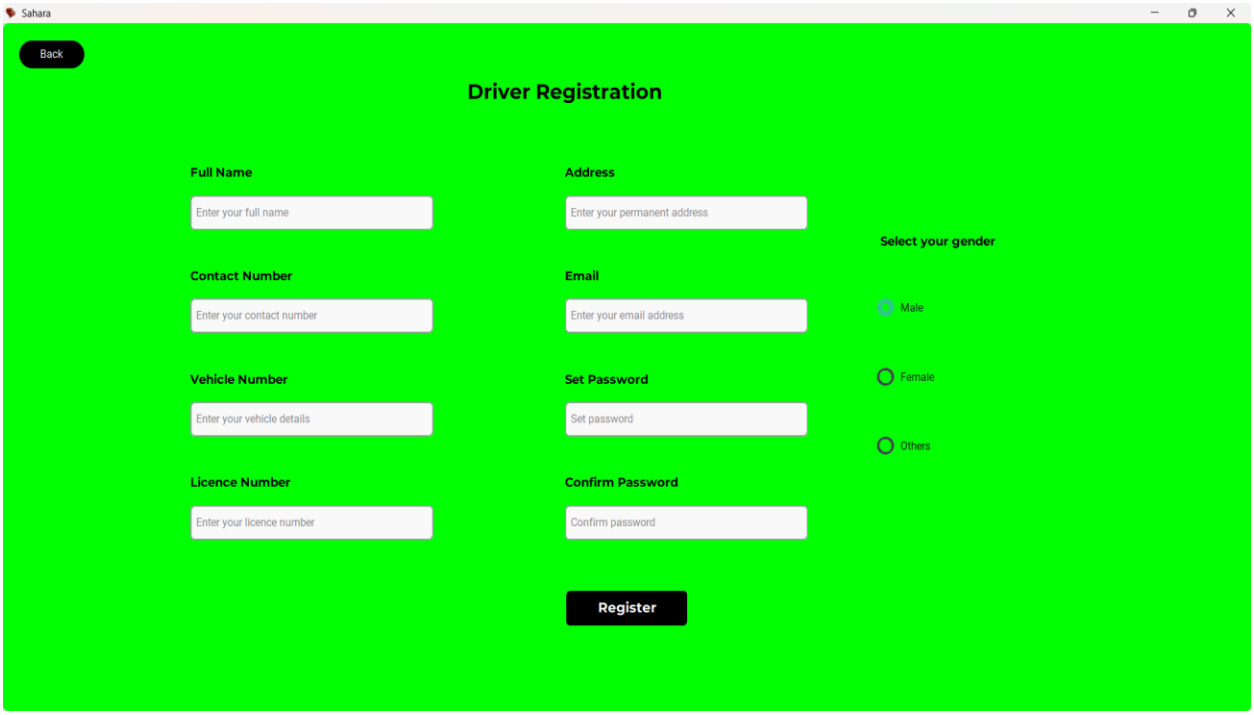


Figure 14: Registration page for drivers

- Dashboard of Passenger

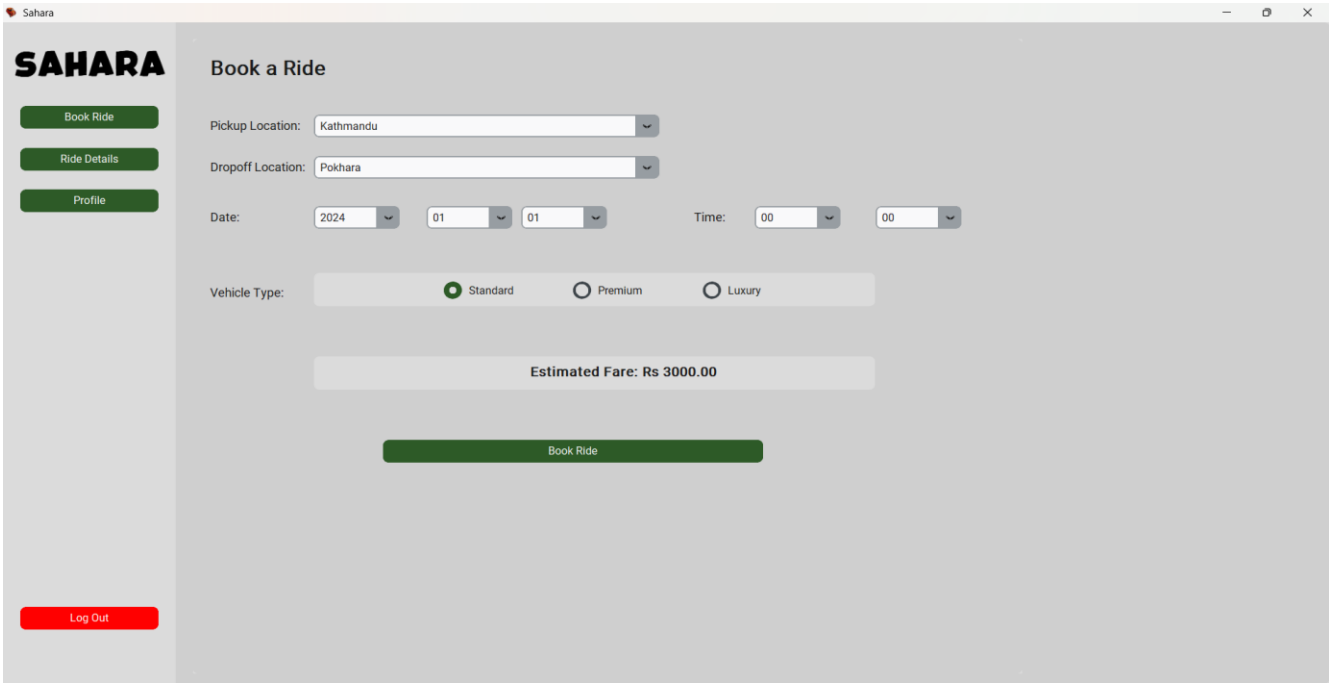


Figure 15: Passenger's Dashboard

- Ride History of Passenger

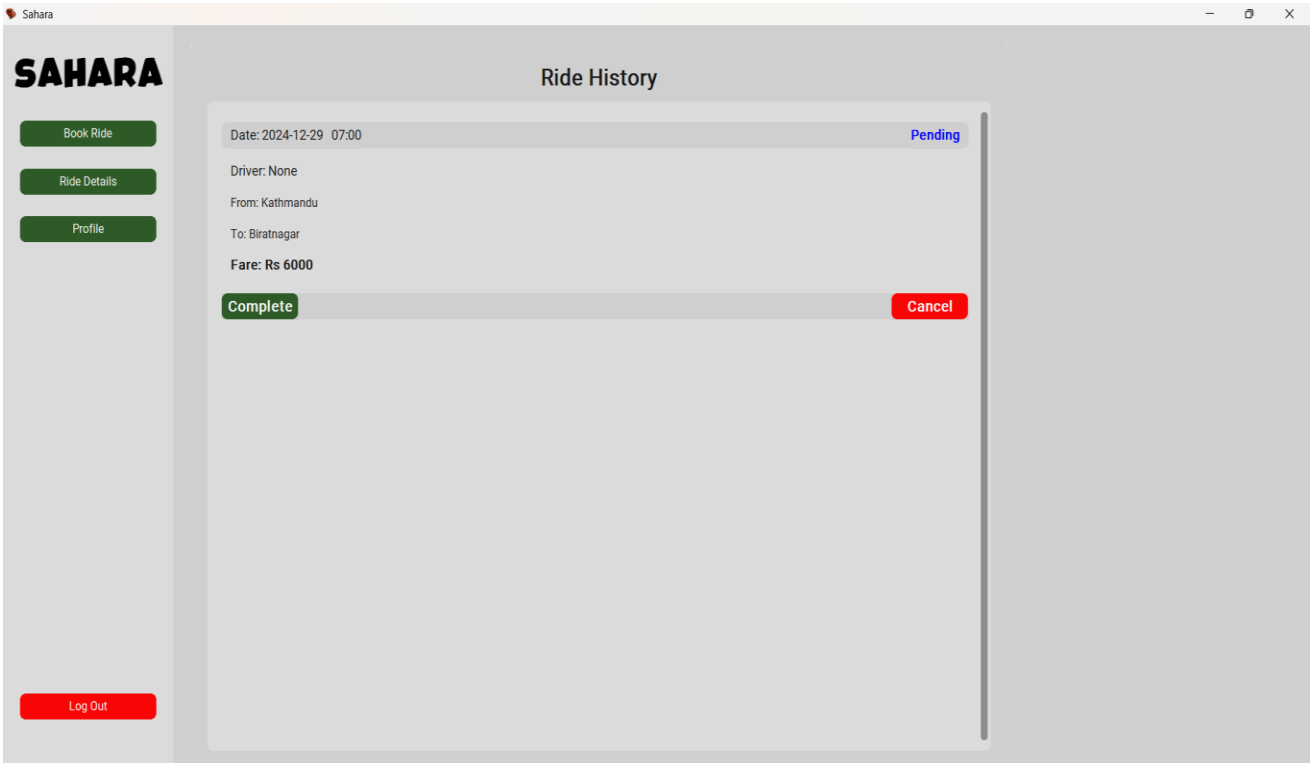


Figure 16: Passenger's History

- Profile of Passenger

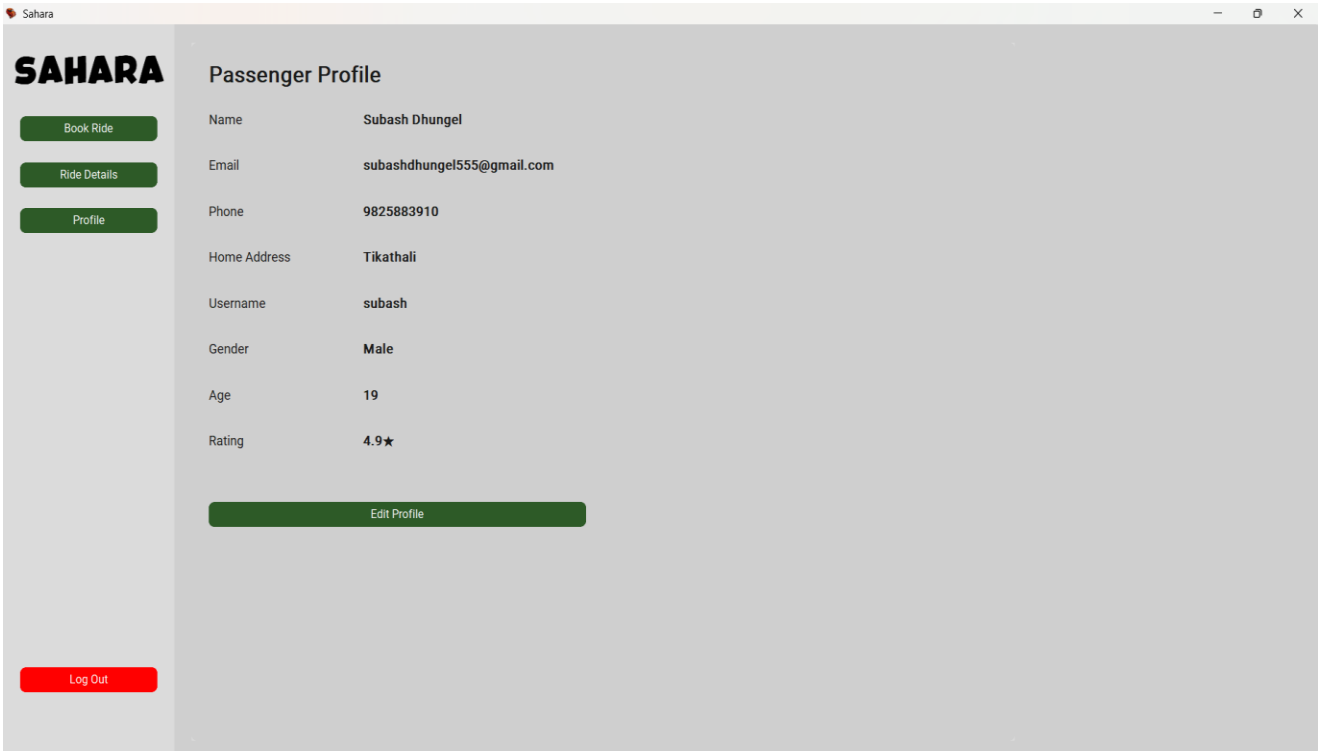


Figure 17: Passenger's Profile

- To edit profile of the passenger

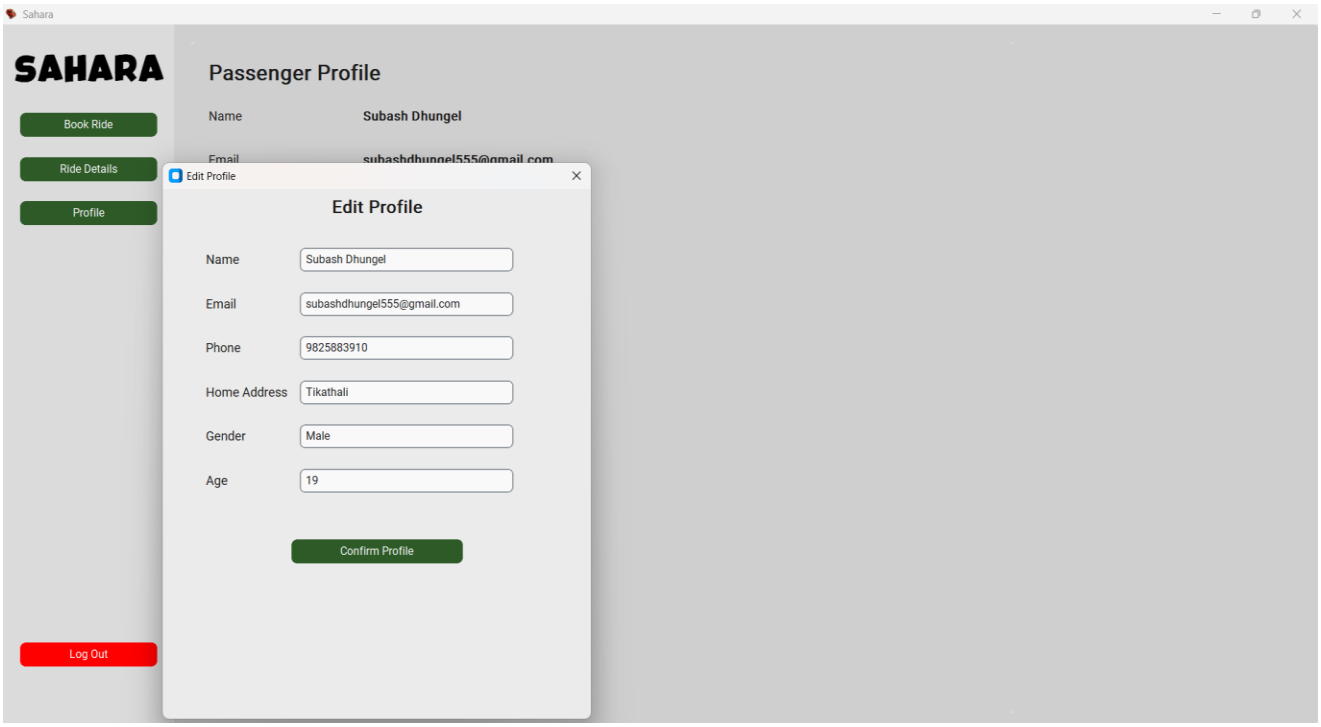


Figure 18: Edit Profile window for Passenger

- Admin Dashboard for booking management

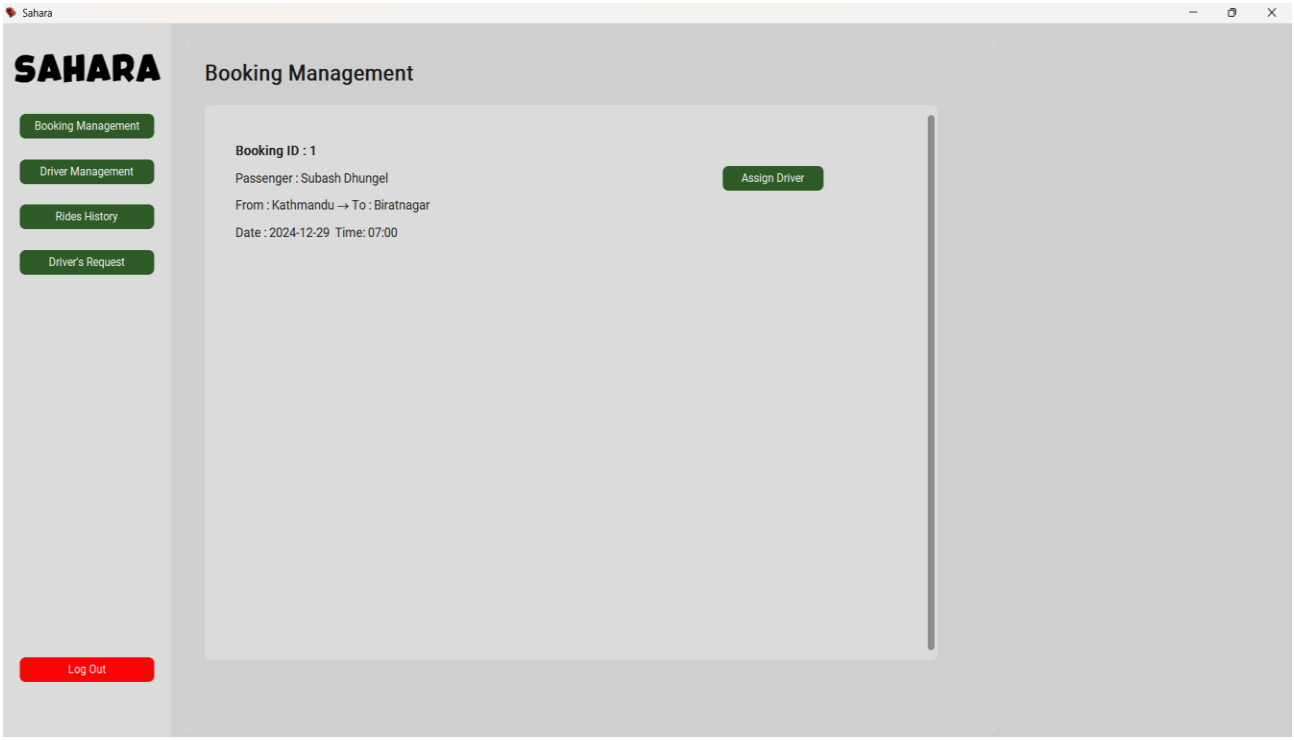


Figure 19: Admin's Dashboard

- To delete driver from the system

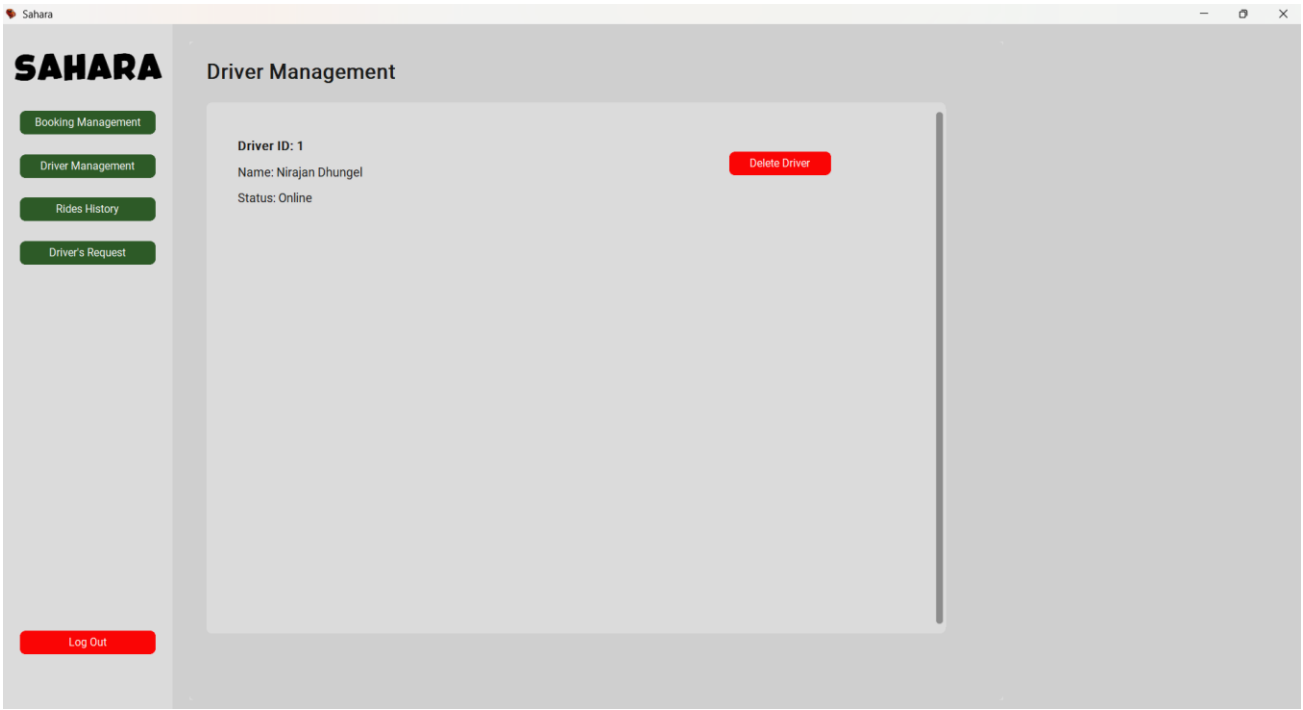


Figure 20: Driver Management for Admin

- History of all rides

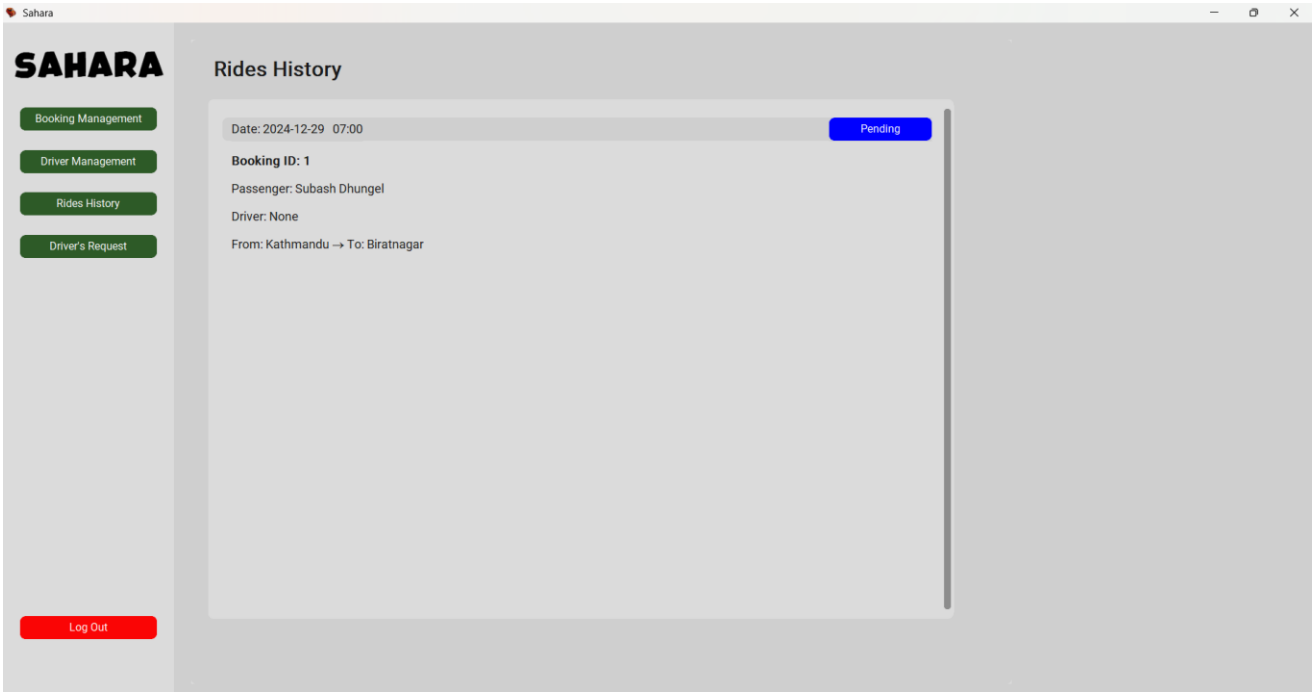


Figure 21: List of Rides History with status

- Requests of Driver for registration to the system

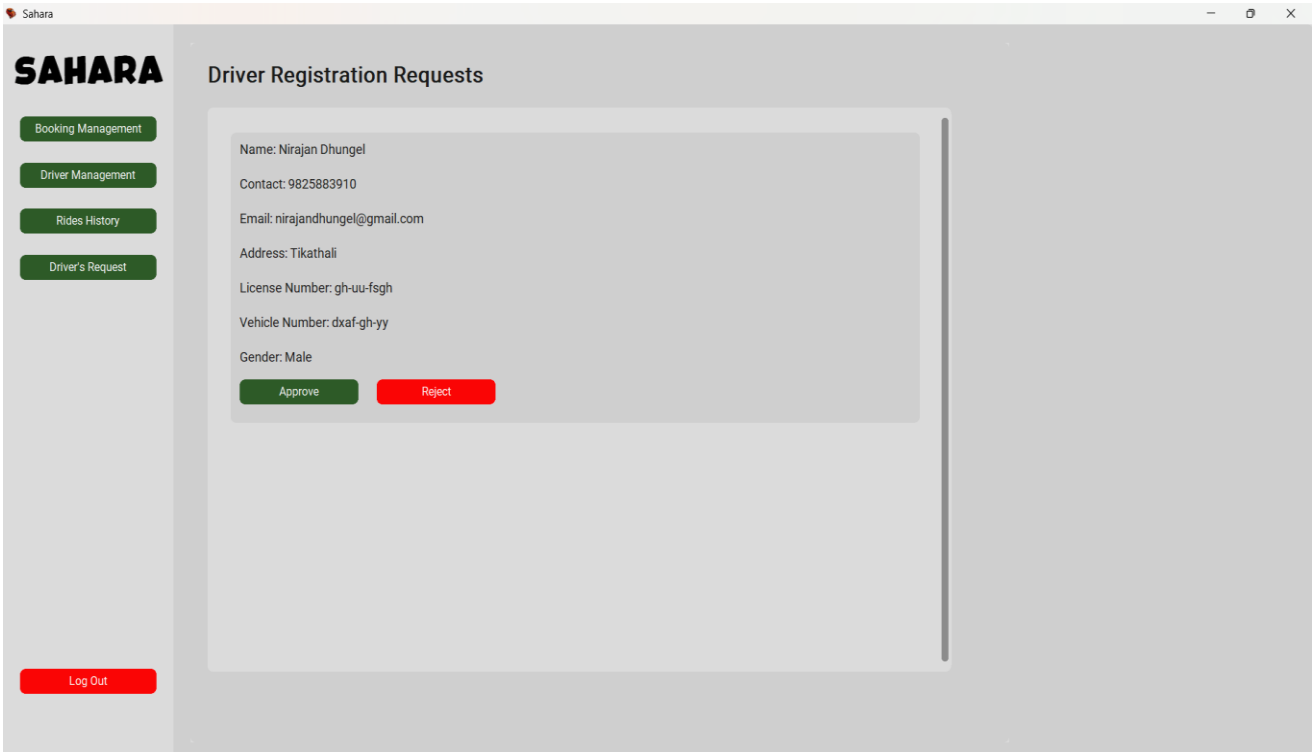


Figure 22:Driver's request on system

- Driver Dashboard

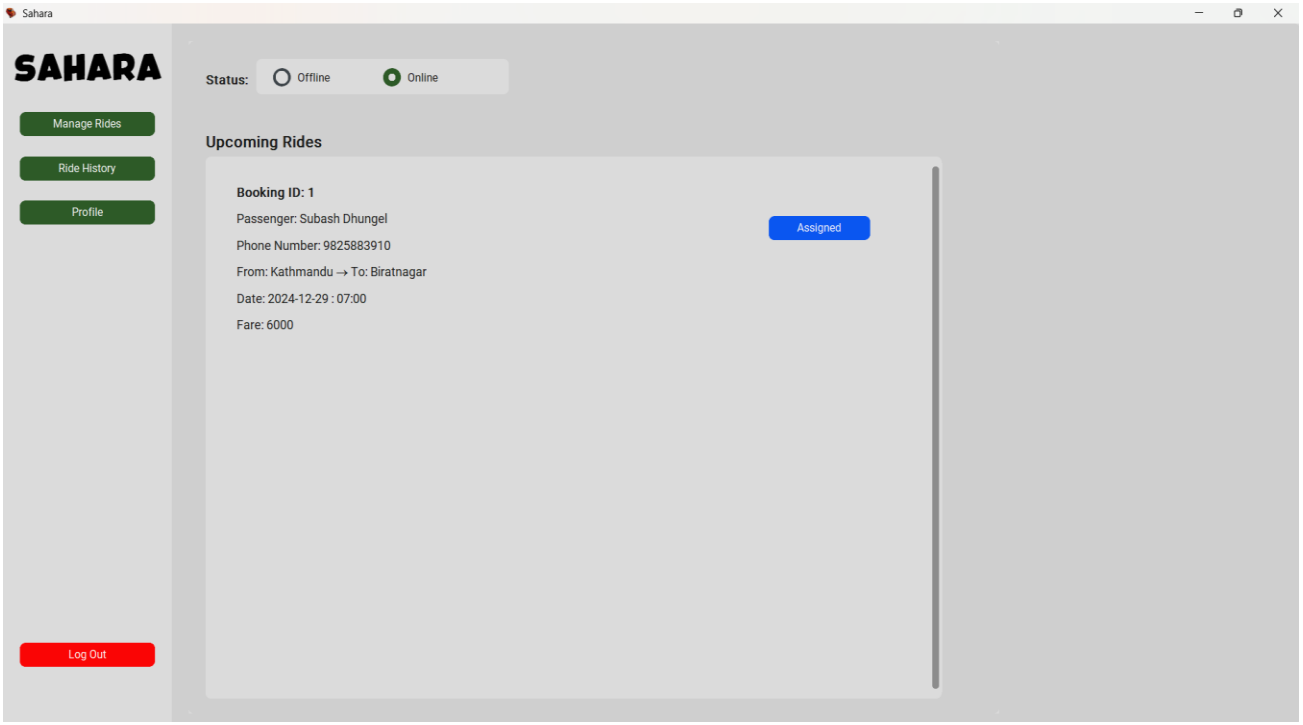


Figure 23:Assigned Trips on driver dashboard

- History of rides

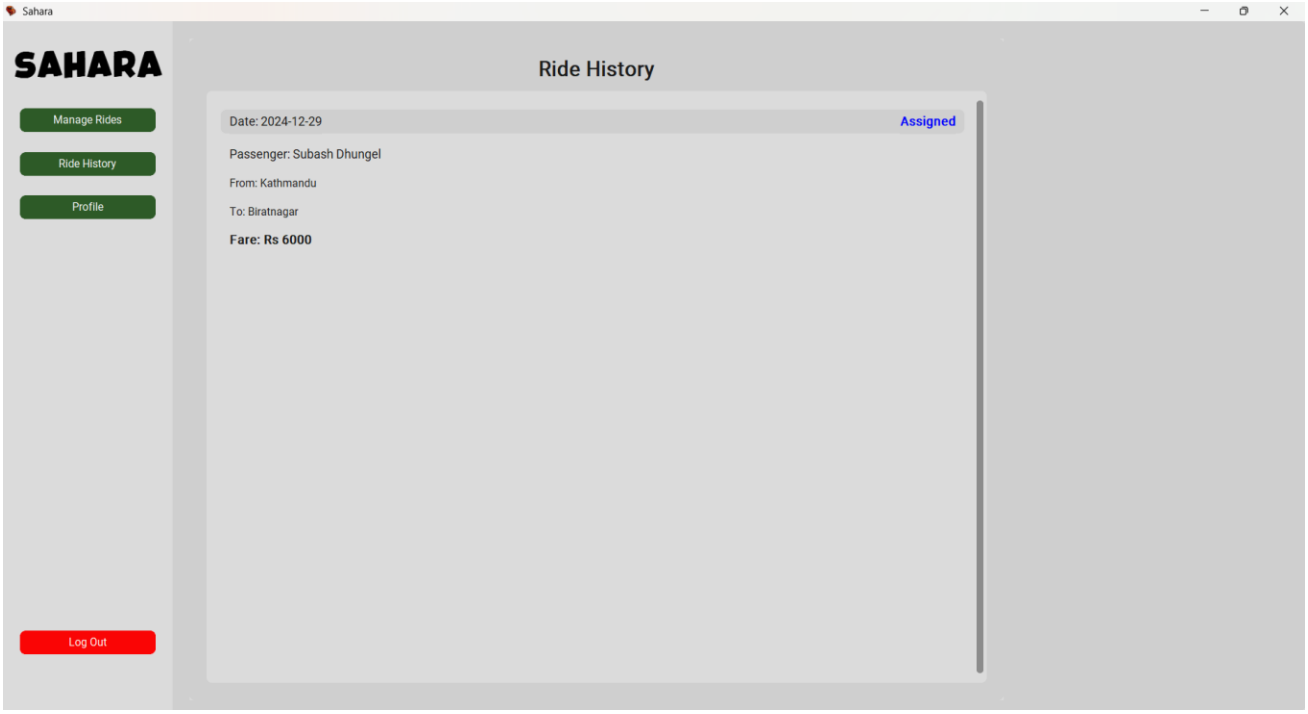


Figure 24: ride's history

- Driver's Profile

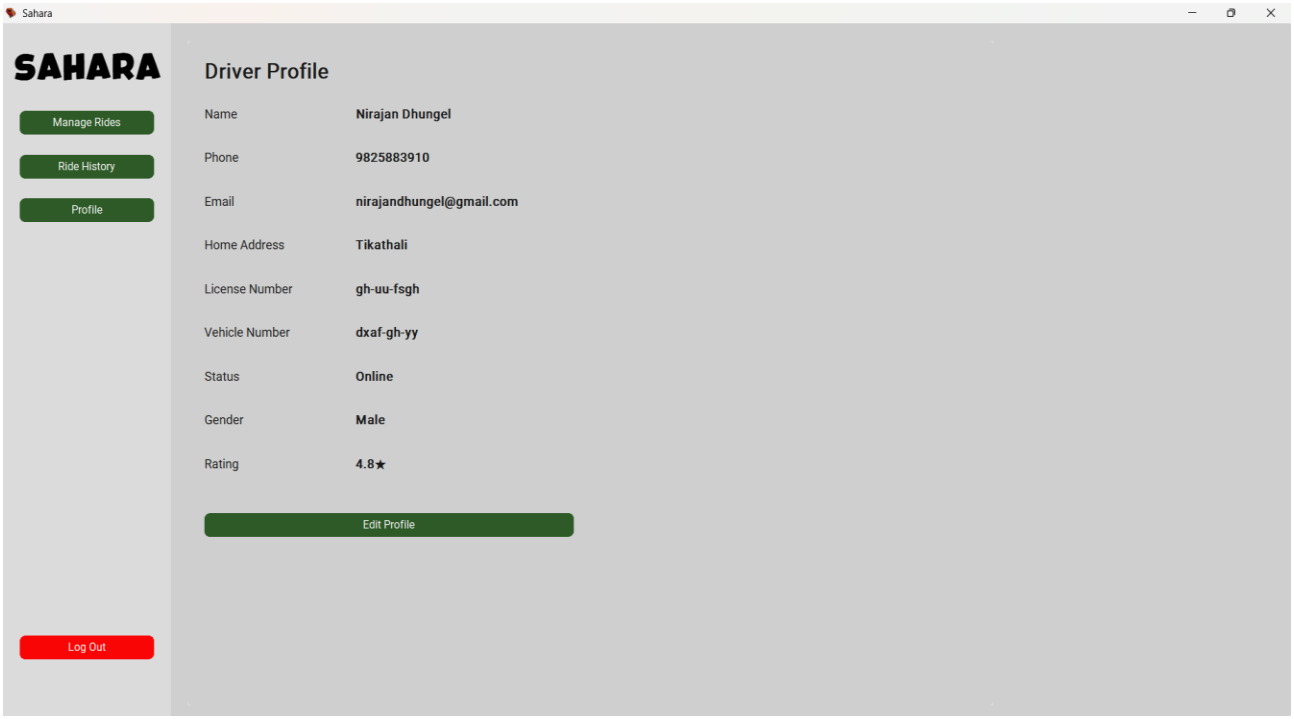


Figure 25: Profile of Driver

- To edit profile of driver

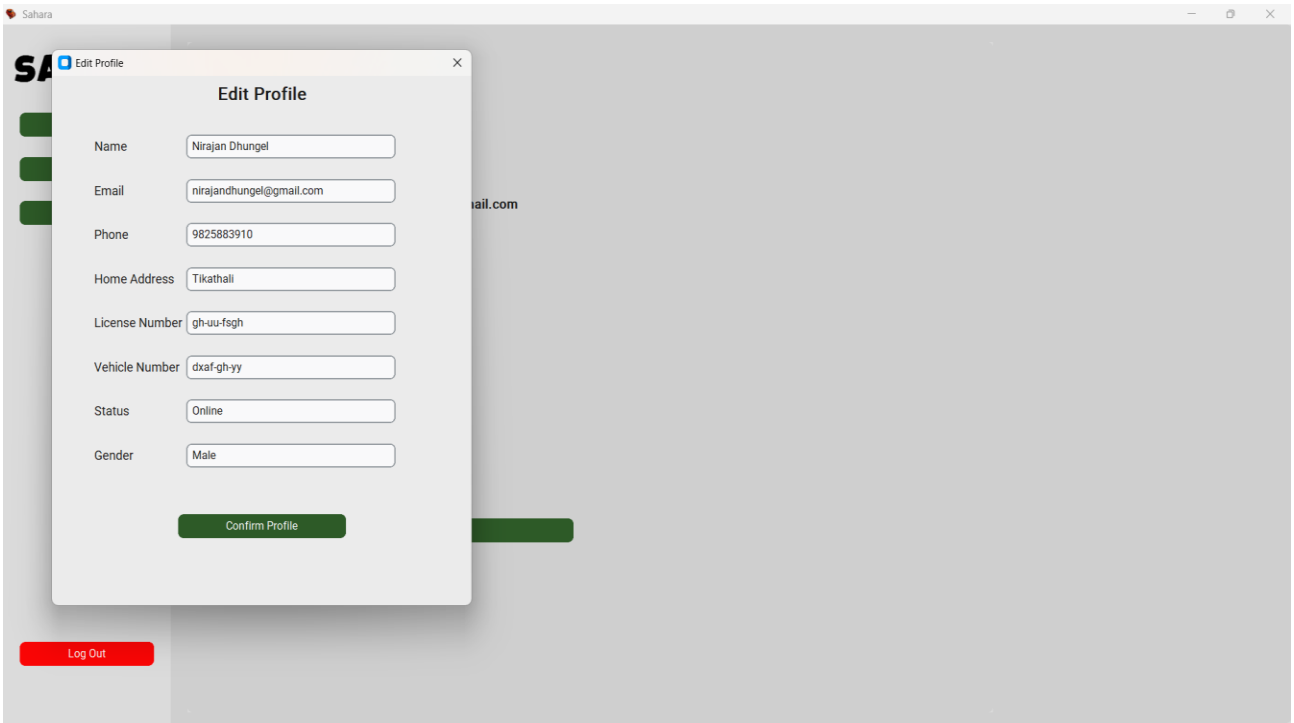


Figure 26: Edit window of driver's profile

Implementation

The taxi booking system, named "**SAHARA**," was a solo development carried out in Python with the CustomTkinter library to provide a user-friendly graphical interface. The project used object-oriented programming (OOP) principles and was built in Visual Studio Code because of its flexibility, ease of use, and debugging features. MySQL was used for database, storing key information such as user details, trip records, and driver availability. To keep things simple, SQL queries were embedded directly in the application; this eliminated the need for a complex server setup.

As the sole developer, I carefully planned and managed my time by breaking the project into smaller tasks i.e. designing the interface, coding the backend, and connecting the database. I first focused on the essential features, like booking rides and fare calculation, before adding extra features to the system. Challenges included fixing database errors and handling schema changes during testing, but I resolved these by debugging and maintaining regular database backups.

Throughout the project, I applied modular design with reusable code, which made the system easier to maintain and extend. The use of CustomTkinter made the interface modern and sleek-looking while keeping the application lightweight. This project tested not only my problem-solving skills but also enhanced my knowledge of Python, database integration, and user-centered design, thus laying a very strong foundation for future projects that are more advanced.

- Python was used for core logics, CustomTkinter for graphical interface, and Visual Studio Code were used for the development of system.
- Being a solo developer, all coding responsibilities were handled independently, with disciplined time management and proper planning ensuring success.
- MySQL has been used for storing user data and trips, with embedded SQL queries for simplicity and without setting up a server.
- A modular design with reusable classes and functions was applied for better maintainability and scalability.
- The experience has enhanced my knowledge of Python, database integration and designing laying a strong foundation for future projects.

Computational Thinking

Computational thinking is the process of identifying a clear, step-by-step solution to a complex problem. It involves the breakdown of a problem into smaller pieces, recognizing patterns, and eliminating all that is extraneous. Then, one can create a step-by-step solution that can be replicated (Team, 2024).

1. Decomposition

Development of the taxi booking system was simplified by breaking it down into smaller, manageable parts. It is mainly divided into four sections: secure login for all user roles, booking a taxi, assigning driver, and completing/cancelling the ride. For this, the system is decomposed into multiple files and folders.

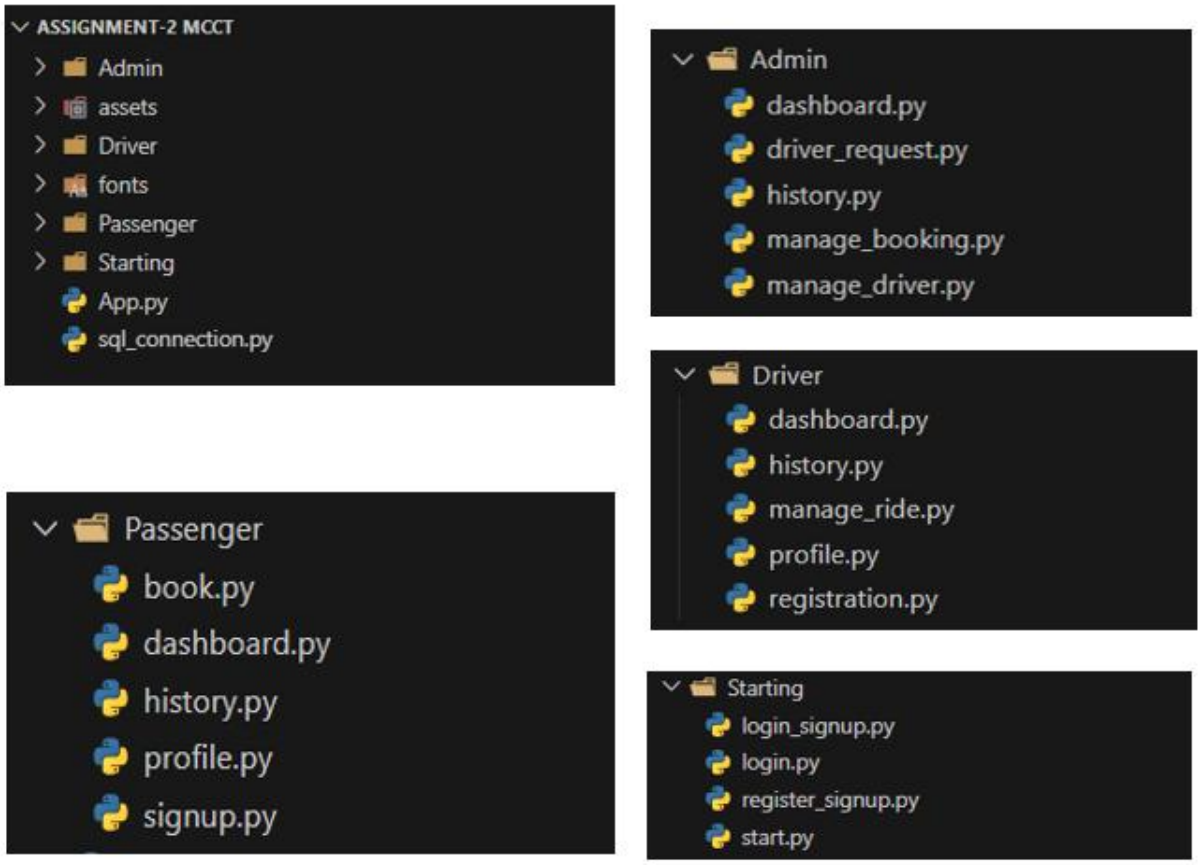


Figure 27: Decomposition of Taxi Booking System

- The Admin folder contains admin specific features such as dashboard for admin, history, manage bookings (assign driver), manage driver registration requests and delete drivers.

- Driver folder includes tasks related to managing rides, profiles, and registrations, handled in files manage_ride.py, profile.py, and registration.py.
- The Passenger folder develops the user side-logic to book taxis, show history, and the ability to sign up via book.py, dashboard.py, and signup.py.
- The Starting folder, containing login.py, is supposed to handle log in-signup for passengers; and register_signup.py would supposedly be for the registered user sign-up.

Again, the files are decomposed into classes with distinct attributes and methods by applying the principles of OOP. This further helps in improving modularity, code reusability, and simplifying the debugging and maintenance process.

Figure 28: Decomposition of files of the system

2. Pattern Recognition

Repeated patterns in functionalities, such as login for all user roles, and database connection are just created once and use multiple times as shown in the below images.

a. Database connection as a pattern recognition:

In the system, the database is created along with the tables and one connection is created but reused across multiple files of the system like login, sign up, registration,booking etc.

sql_connection.py > DatabaseConnection > connection

```python
import mysql.connector as conn
class DatabaseConnection:
    message = ""
    try:
        @staticmethod #doesnt contain any dynamic methods like self.connection()
        def connection():
            # Initial connection without specifying a database
            connect = conn.connect(···
            if connect.is_connected():···

            # Reconnect and specify the database now that it exists
            connection = conn.connect(
                host='localhost',
                user='root', # username
                password='2005subash0910', # password
                database='nirajan_taxi' # name of the database
            )
            return connection

    except Exception as e:
        print(e)
```

```python
class Login(ctk.CTkFrame):
    def __init__(self, parent, controller,shared_data):
        super().__init__(parent)
        self.controller = controller
        self.shared_data = shared_data
        self.configure(fg_color="#00ff00")
        self._conn_=DatabaseConnection.connection() # Protected
        self.login_ui()
```

```python
class PassengerSignUp(ctk.CTkFrame):
    def __init__(self, parent, controller,shared_data):
        super().__init__(parent)
        self.controller = controller
        self.shared_data = shared_data
        self.configure(fg_color="#00ff00")
        self._conn_= DatabaseConnection.connection() # Protected
        self.signup_ui()
```

Figure 29: Pattern recognition in database connection

The connection method is called multiple times within different files with a protected access specifier to ensure that access to the database is done securely and uniformly throughout the project. This avoids redundancy, keeping the connection of the database centralized and efficient.

b. Pattern recognition in login

All the user roles share the same login page as they all have to login with email and password before entering into the system.

```
App.py    login.py M ×
Starting > login.py > Login > __init__
  5  class Login(ctk.CTkFrame):
 42      def login(self):
 50              messagebox.showerror("Error", "All fields are required!")
 51              return False # Exit the function if validation fails
 52          try:
 53              self._conn_=DatabaseConnection.connection() # connection for dynamic changes
 54
 55              # Attempt to connect to the MySQL database
 56              with  self._conn_.cursor() as cursor:
 57                  query1 = "SELECT id FROM passengers WHERE email = %s AND user_password = %s"
 58                  cursor.execute(query1, (email, password))
 59                  self.customer = cursor.fetchone() # Fetch the result of the query (if any)
 60                  query2 = "SELECT id FROM drivers WHERE email = %s AND user_password = %s"
 61                  cursor.execute(query2, (email, password))
 62                  self.driver = cursor.fetchone() # Fetch the result of the query (if any)
 63                  query3 = "SELECT id FROM admins WHERE email = %s AND user_password = %s"
 64                  cursor.execute(query3, (email, password))
 65                  self.admin = cursor.fetchone() # Fetch the result of the query (if any)
 66
 67          except Exception as e:# If any other error occurs, show a generic error message
 68              messagebox.showerror("Error", f"An unexpected error occurred: {e}")
 69
 70
 71          if self.admin:
 72              self.shared_data["admin_id"] = self.admin[0] # Set the passenger_id in shared_data for later use on passenger dashboard
 73              self.controller.show_frame("AdminDashboard") # show_frame is an example of abstarcion, it is changing the frames without showing the codes
 74              messagebox.showinfo("Success", "Successfully Logged In!")
 75          elif self.customer:
 76              self.shared_data["passenger_id"] = self.customer[0] # Set the passenger_id in shared_data for later use on passenger dashboard
 77              self.controller.show_frame("PassengerDashboard") # show_frame is an example of abstarcion, it is changing the frames without showing the codes
 78              messagebox.showinfo("Success", "Successfully Logged In!")
 79          elif self.driver:
 80              self.shared_data["driver_id"] = self.driver[0] # Set the passenger_id in shared_data for later use on passenger dashboard
 81              self.controller.show_frame("DriverDashboard") # show_frame is an example of abstarcion, it is changing the frames without showing the codes
 82              messagebox.showinfo("Success", "Successfully Logged In!")
 83          else:
 84              messagebox.showerror("Error", "Invalid email or password!") # error message for invalid inputs
 85
```

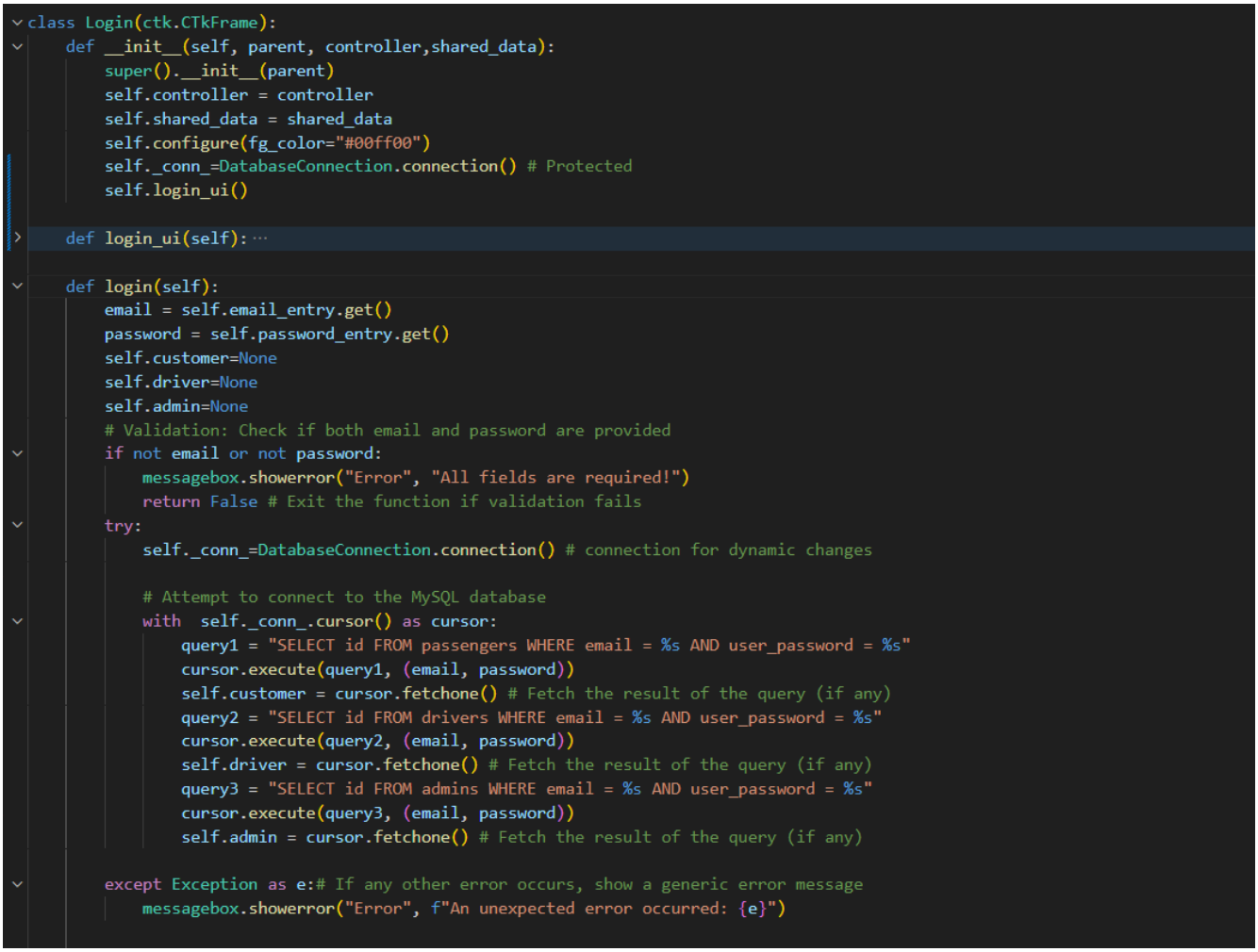
Figure 30: Pattern recognition in login

When the user logs in with their email and password, the system first checks in the Admin table. If this doesn't work, the system will check the Passengers table and finally the Drivers table. Once the match is found, the system automatically redirects the user to his or her respective dashboard according to their role, Admin, Passenger, or Driver, and displays their credentials. In this way, it ensures a smooth and unified login for all users.

3. Abstraction

Abstraction is the part of computational thinking that removes complexity of program by showing relevance and clarity on a higher level. It involves filtering out the unnecessary details and highlighting what is most important and interrelating to the decomposed elements (McVeigh-Murphy, 2020).

Irrelevant details like database connections is hidden from the main logic using reusable classes and functions. For instance, in the login file, only the core logics for checking user credentials and handling the login are shown, while the database connection and UI elements are kept hidden. This simplifies the main logic by making it more readable, manageable, and maintainable as shown in the below figure.



```
class Login(ctk.CTkFrame):
    def __init__(self, parent, controller,shared_data):
        super().__init__(parent)
        self.controller = controller
        self.shared_data = shared_data
        self.configure(fg_color="#00ff00")
        self._conn_=DatabaseConnection.connection() # Protected
        self.login_ui()

    def login_ui(self):···

    def login(self):
        email = self.email_entry.get()
        password = self.password_entry.get()
        self.customer=None
        self.driver=None
        self.admin=None
        # Validation: Check if both email and password are provided
        if not email or not password:
            messagebox.showerror("Error", "All fields are required!")
            return False # Exit the function if validation fails
        try:
            self._conn_=DatabaseConnection.connection() # connection for dynamic changes

            # Attempt to connect to the MySQL database
            with  self._conn_.cursor() as cursor:
                query1 = "SELECT id FROM passengers WHERE email = %s AND user_password = %s"
                cursor.execute(query1, (email, password))
                self.customer = cursor.fetchone() # Fetch the result of the query (if any)
                query2 = "SELECT id FROM drivers WHERE email = %s AND user_password = %s"
                cursor.execute(query2, (email, password))
                self.driver = cursor.fetchone() # Fetch the result of the query (if any)
                query3 = "SELECT id FROM admins WHERE email = %s AND user_password = %s"
                cursor.execute(query3, (email, password))
                self.admin = cursor.fetchone() # Fetch the result of the query (if any)

        except Exception as e:# If any other error occurs, show a generic error message
            messagebox.showerror("Error", f"An unexpected error occurred: {e}")
```

Figure 31: Abstraction in login page

Another example of abstraction is use of different colors just by importing Colors class and using it on different files without defining colors again and again as shown in below figure.

```
import customtkinter as ctk
from sql_connection import DatabaseConnection
from fonts.colors import Colors
```

```
fg_color=Colors.GREEN_BUTTON,
hover_color=Colors.GREEN_BUTTON_HOVER,
```

Figure 32: Colors as a part of abstraction

4. Algorithm Design

a. Algorithm of fare calculation

Algorithms were designed for the calculation of fares, checking the validity of login credentials, and retrieval of data from the database. Example: Fare calculation algorithm dynamically changes as per the distance and time and provides correct results.

```
def update_fare(self, *args):
    # Fare calculation based on vehicle type
    base_fare = {
        "Standard": 1000,
        "Premium": 2000,
        "Luxury": 3000
    }

    self.selected_fare = base_fare[self.vehicle_var.get()] # store taxi-type fare in selected_fare

    if self.pickup_entry.get() and self.dropoff_entry.get():# Add distance-based fare if locations are selected

        # storing pickup and dropoff location
        pickup = self.pickup_entry.get()
        dropoff = self.dropoff_entry.get()

        #Fare calculations
        if pickup == dropoff:
            self.selected_fare = 0.00
        elif( pickup == "Kathmandu" and dropoff=="Pokhara") or ( dropoff == "Kathmandu" and pickup=="Pokhara"):# 2000 Rs added to base_fare for this distance
            self.selected_fare = self.selected_fare + 2000
        elif( pickup == "Kathmandu" and dropoff=="Lalitpur") or ( dropoff == "Kathmandu" and pickup=="Lalitpur"):# 500 Rs added to base_fare for this distance
            self.selected_fare = self.selected_fare + 500
        elif( pickup == "Kathmandu" and dropoff=="Bhaktapur") or ( dropoff == "Kathmandu" and pickup=="Bhaktapur"):# 500 Rs added to base_fare for this distance
            self.selected_fare = self.selected_fare + 500
        elif( pickup == "Kathmandu" and dropoff=="Biratnagar") or ( dropoff == "Kathmandu" and pickup=="Biratnagar"):# 3000 Rs added to base_fare for this distance
            self.selected_fare = self.selected_fare + 3000
        elif( pickup == "Pokhara" and dropoff=="Lalitpur") or ( dropoff == "Pokhara" and pickup=="Lalitpur"):# 2000 Rs added to base_fare for this distance
            self.selected_fare = self.selected_fare + 2000
        elif( pickup == "Pokhara" and dropoff=="Bhaktapur") or ( dropoff == "Pokhara" and pickup=="Bhaktapur"):# 2000 Rs added to base_fare for this distance
            self.selected_fare = self.selected_fare + 2000
        elif( pickup == "Pokhara" and dropoff=="Biratnagar") or ( dropoff == "Pokhara" and pickup=="Biratnagar"):# 2000 Rs added to base_fare for this distance
            self.selected_fare = self.selected_fare + 2000
        elif( pickup == "Lalitpur" and dropoff=="Bhaktapur") or ( dropoff == "Lalitpur" and pickup=="Bhaktapur"):# 500 Rs added to base_fare for this distance
            self.selected_fare = self.selected_fare + 500
        elif( pickup == "Lalitpur" and dropoff=="Biratnagar") or ( dropoff == "Lalitpur" and pickup=="Biratnagar"):# 3000 Rs added to base_fare for this distance
            self.selected_fare = self.selected_fare + 3000
        elif( pickup == "Bhaktapur" and dropoff=="Biratnagar") or ( dropoff == "Bhaktapur" and pickup=="Biratnagar"):# 3000 Rs added to base_fare for this distance
            self.selected_fare = self.selected_fare + 3000

    self.fare_label.configure(text=f"Estimated Fare: Rs {self.selected_fare:.2f}")
```

Figure 33: Algorithm of fare calculation

b. Algorithm to validate signup details

This algorithm validates user inputs in a form using sequential checks. It checks whether all fields are filled, the contact number is exactly 10 digits, and the email is in a valid format. The algorithm verifies that the password matches the confirm password, is 6–18 characters long, and checks if the age is numeric and within a valid range of 5–100. If any validation fails, an error message is shown; otherwise, the input is considered valid.

```
def validate(self):
    # Validation of user inputs
    name = self.name_entry.get()
    contact = self.phone_entry.get()
    email = self.email_entry.get()
    password = self.password_entry.get()
    age = self.age_entry.get()
    confirm_password = self.confirm_password_entry.get()

    if not name or not contact or not email or not password:
        messagebox.showerror("Error", "All fields are required!")
        return False

    if not re.match(r"^[0-9]{10}$", contact):
        messagebox.showerror("Error", "Invalid contact number!")
        return False

    if not re.match(r"[^@]+@[^@]+\.[^@]+", email):
        messagebox.showerror("Error", "Invalid email format!")
        return False

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

    if len(password) < 6 or len(password) > 18:
        messagebox.showerror("Error", "Password must be between 6 and 18 characters!")
        return False

    if not age.isdigit():
        messagebox.showerror("Error", "Invalid age!")
        return False

    age = int(age)
    if age < 5 or age > 100:
        messagebox.showerror("Error", "Invalid age!")
        return False

    return True
```

Figure 34: Validation algorithm

Testing

1. Signup for passenger

Property	Details
Date	13th December
Reason	Check addition of new user
Action	Invalid inputs at first and valid input at last
Expected Result	Error and Success message
Actual Result	Error and success message displayed
Result	Pass

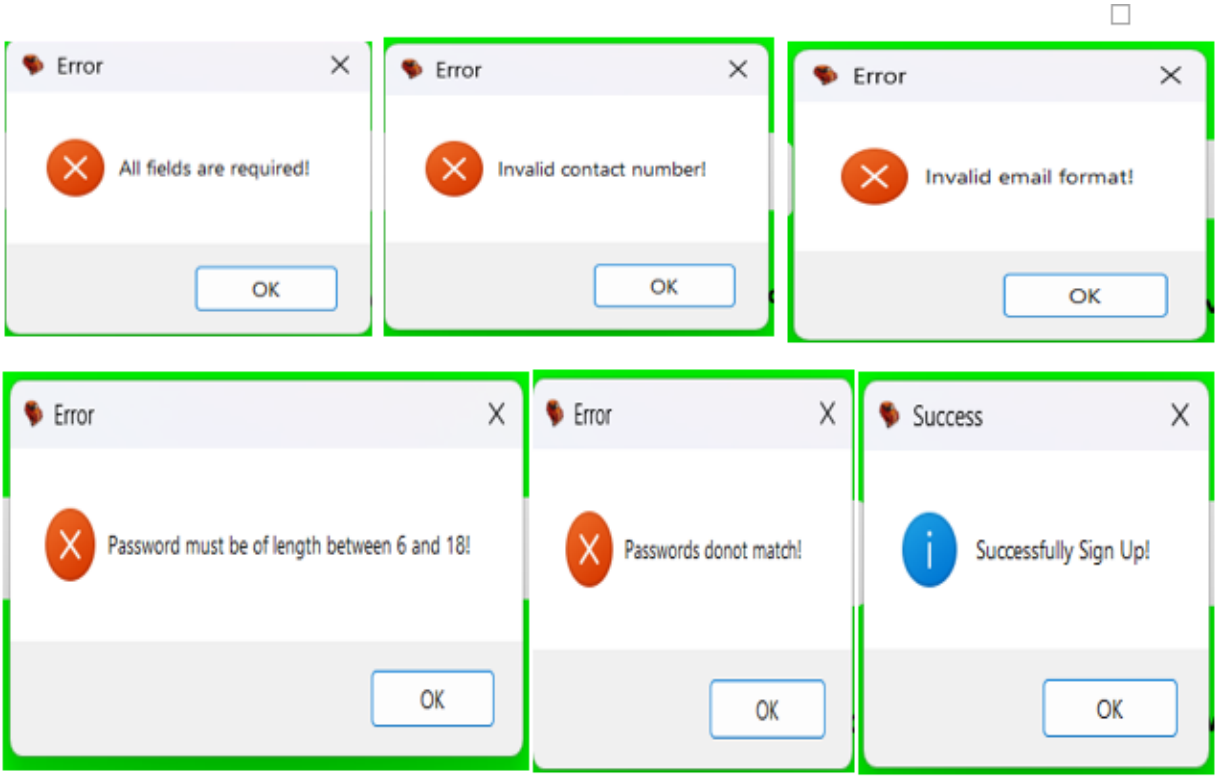


Figure 35: Testing of signup

2. Login

Property	Details
Date	13th December
Reason	Check Login for all users
Action	Invalid inputs at first and valid input at last
Expected Result	Error and Success message
Actual Result	Error and success message displayed
Result	Pass

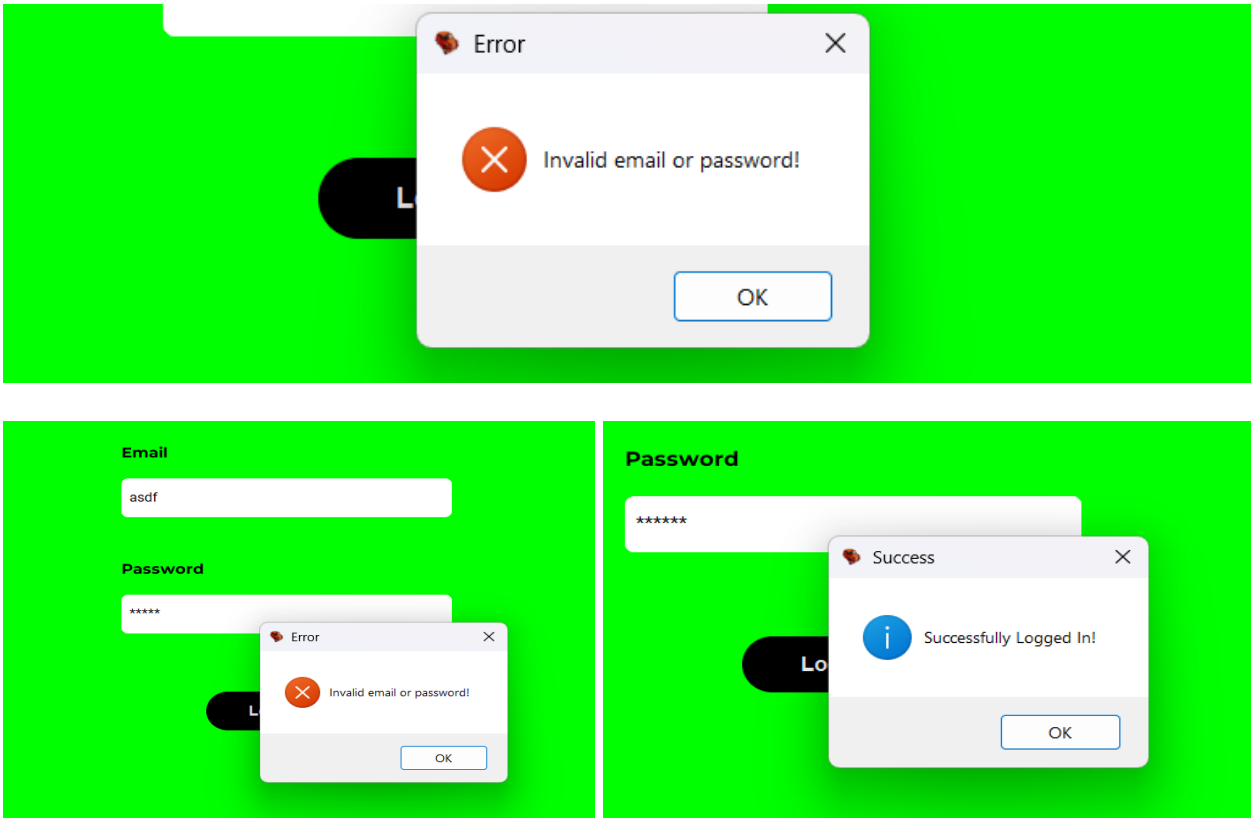


Figure 36:Testing of login

3. Booking for passengers

Property	Details
Date	13th December
Reason	Test valid and invalid bookings
Action	Invalid inputs at first and valid input at last
Expected Result	Error and Success message
Actual Result	Error and success message displayed
Result	Pass

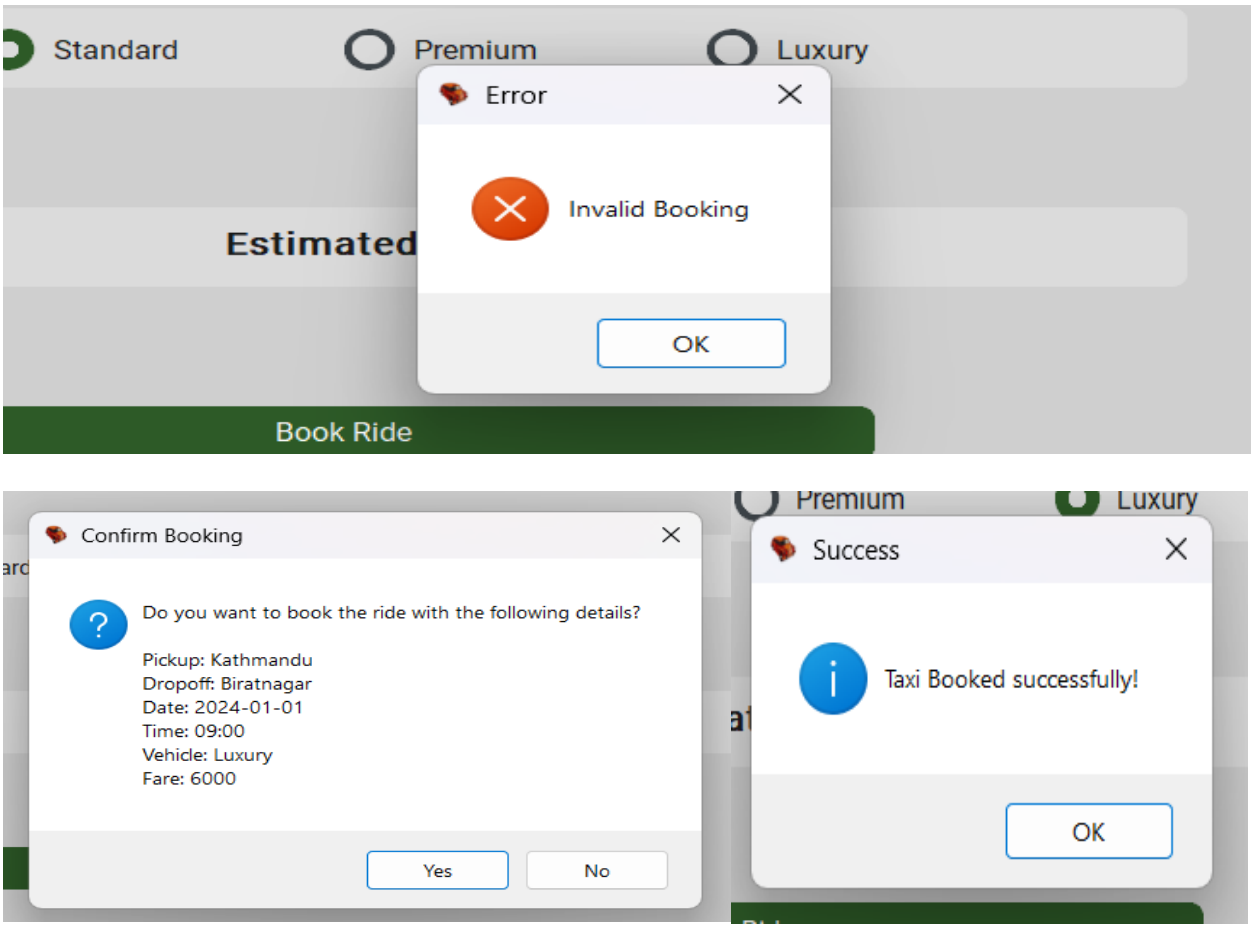


Figure 37: Testing of booking

4.Complete ride for passengers

Property	Details
Date	13th December
Reason	Check completion of ride
Action	Invalid inputs at first and valid input at last
Expected Result	Error and Success message
Actual Result	Error and success message displayed
Result	Pass

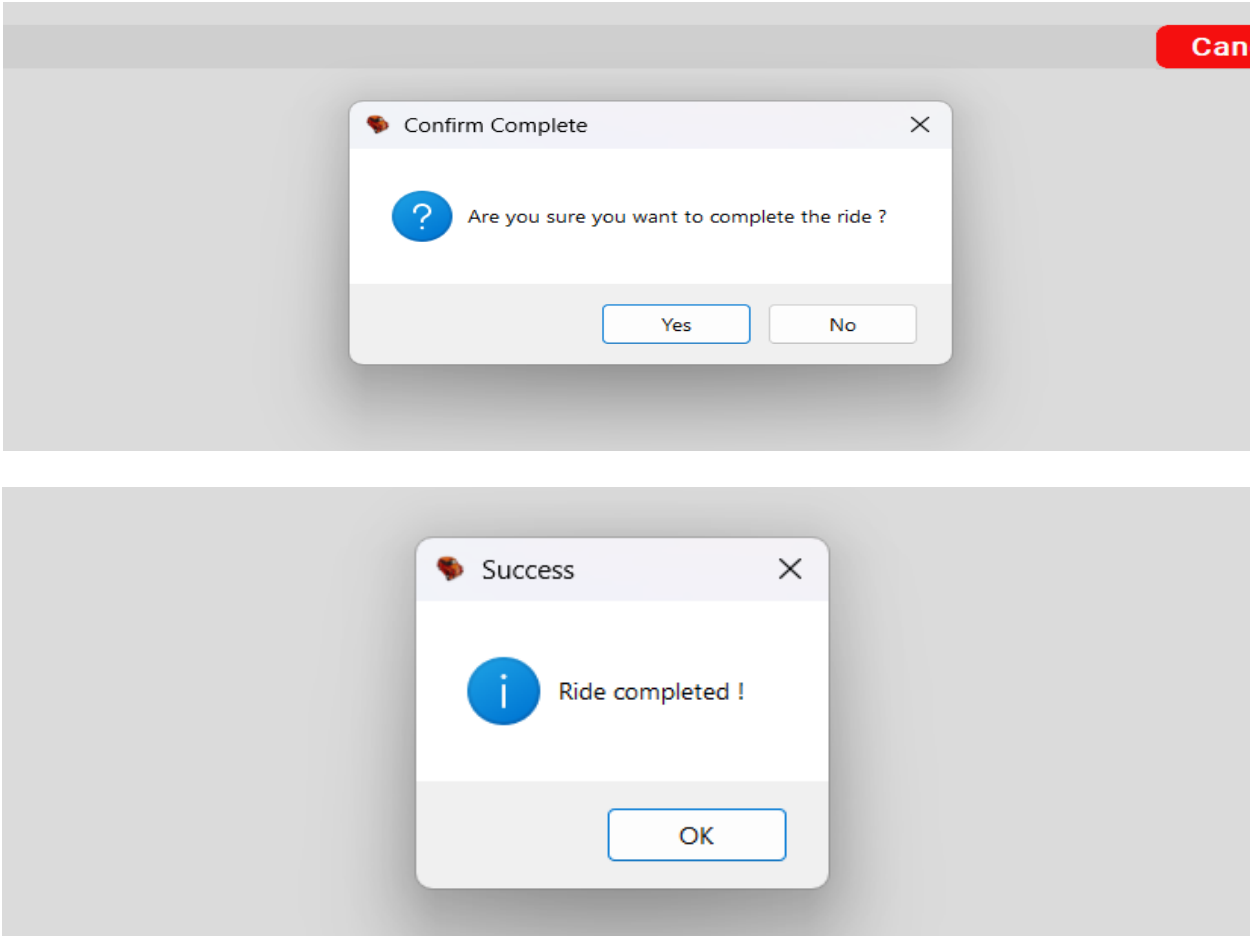


Figure 38: Testing of ride completion

5. Assign Driver for Admins

Property	Details
Date	13th December
Reason	Assign driver to the bookings
Action	Test driver assignment
Expected Result	Success message
Actual Result	Success message displayed
Result	Pass

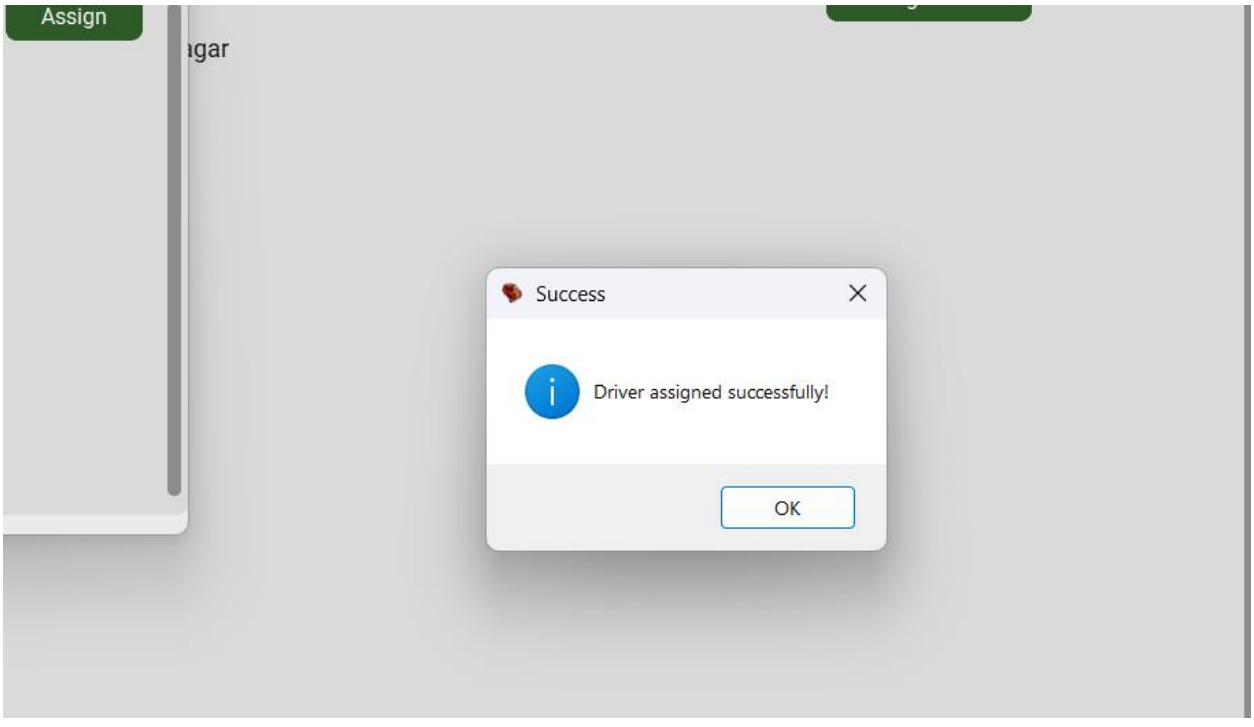


Figure 39: Testing of driver assignment

6. Approve/Reject Driver Request

Property	Details
Date	13th December
Reason	Check approve and reject function
Action	assign driver
Expected Result	Success message
Actual Result	Success message displayed
Result	Pass

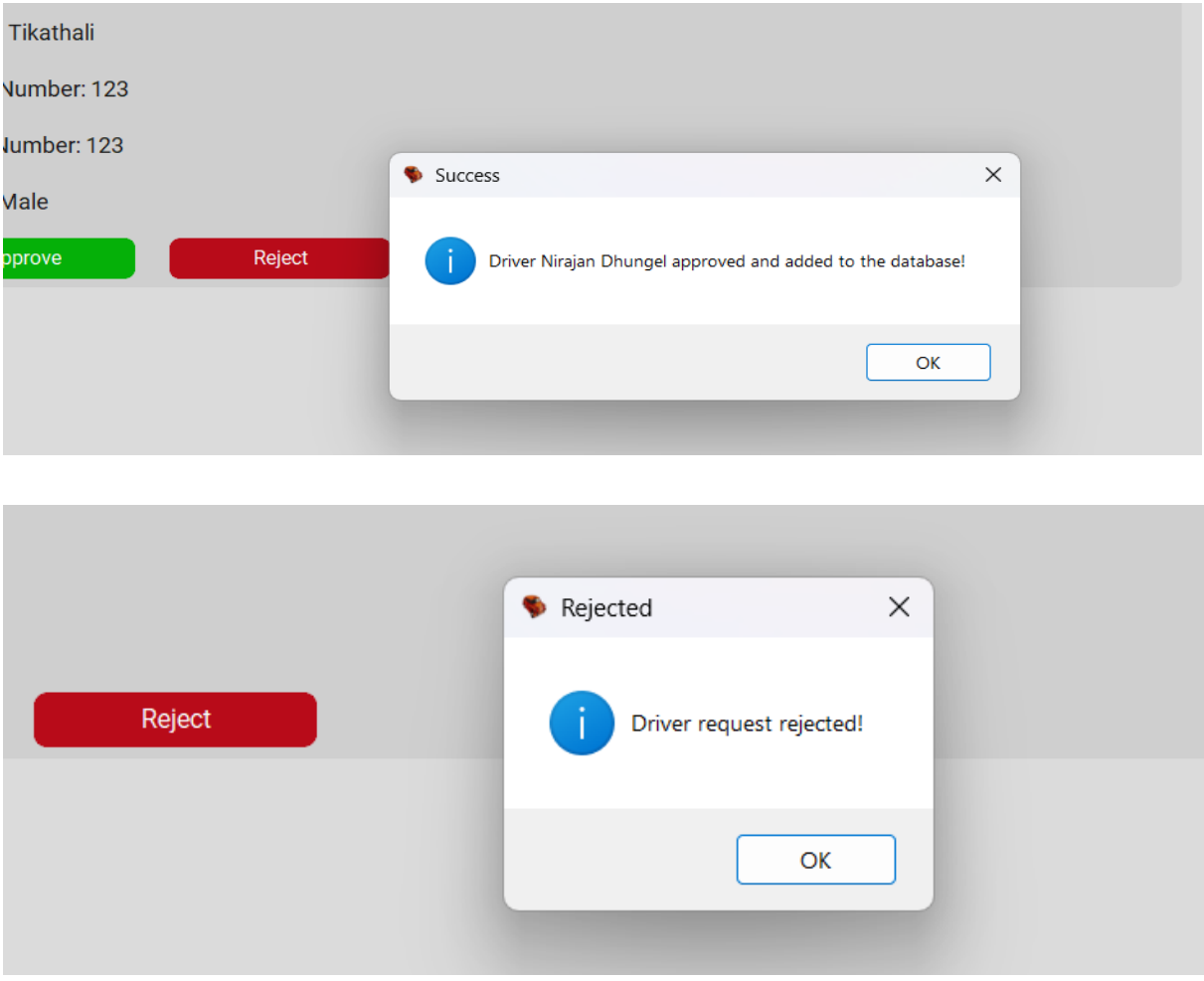


Figure 40:Testing of driver request approve or reject

Discussion

This project has been a great experience in which I got to work on the development of a taxi booking system. In the process, I got to practice my programming skills, manage my time better, and most important I got to communicate with teachers frequently for guidance.

First of all, I was able to realize most of the specifications that were in the assignment brief. However, along the way, there were a few challenges. Adding features, such as implementing user login by passing user credentials to the user's dashboard and dynamically updating UI components based on the database, was not as straightforward as I had thought it would be. I overcame this by researching online for solutions, breaking down the problem into smaller parts, and testing thoroughly.

Time management was a bit challenging, yet it was a learning process. I did plan enough time for coding and testing, but some of the bugs were more difficult to iron out than I had imagined. That taught me a valuable lesson in leaving room for unexpected issues when estimating project timelines in the future. Being the sole developer of this project, I had to do everything-myself, from designing the UI to writing the backend. This gave me in-depth knowledge of how all of these components work together; it also taught me to plan and prioritize tasks effectively. It also taught me to remain focused, even working on challenges.

This project really helped me get a better understanding of Python. I gained more confidence in concepts such as user role management, including drivers, passengers, and admin, and handling interactions between the frontend and backend. It has also given me the motivation to explore Python further and look into other programming languages for furthering my programming skills.

If I had to do this project again, I would have spent more time at the beginning planning and breaking down tasks. This would avoid having to rush through some parts later on. I would also have added a feature to integrate maps into the system using Google Maps, which would make it easier to visualize locations and routes. In addition, I would include ride tracking, and live location updates for both drivers and passengers. These enhancements will make the system more user-friendly and realistic.This assignment was a great learning experience for me because it helped me to improve my technical skills and also gave a better understanding of how to manage a project.

Conclusion

The requirements of assignment are successfully met with the development of system by including user credential management, administrative control, and data organisation. The system design has ensured functionality, reliability, and ease of use expected at a high standard. While there were various challenges involved in this process, such as dynamic updating and debugging of complex interaction, these were overcome through extensive research, hard testing, and problem-solving techniques applied. The resulting system demonstrates a well-balanced combination of practicality with technical efficiency.

The project has been one huge learning curve, in which the learning did not solely relate to the technical aspects, but it included planning, management, and attention to details. Translation of theoretical knowledge to a working system reinforced critical thinking and flexibility. This was coupled with maintaining structured workflows, ensuring tasks were done without sacrificing time. Insights developed through the research process and acquired skills are surely going to provide substantial assistance in further studies and professional pursuits.

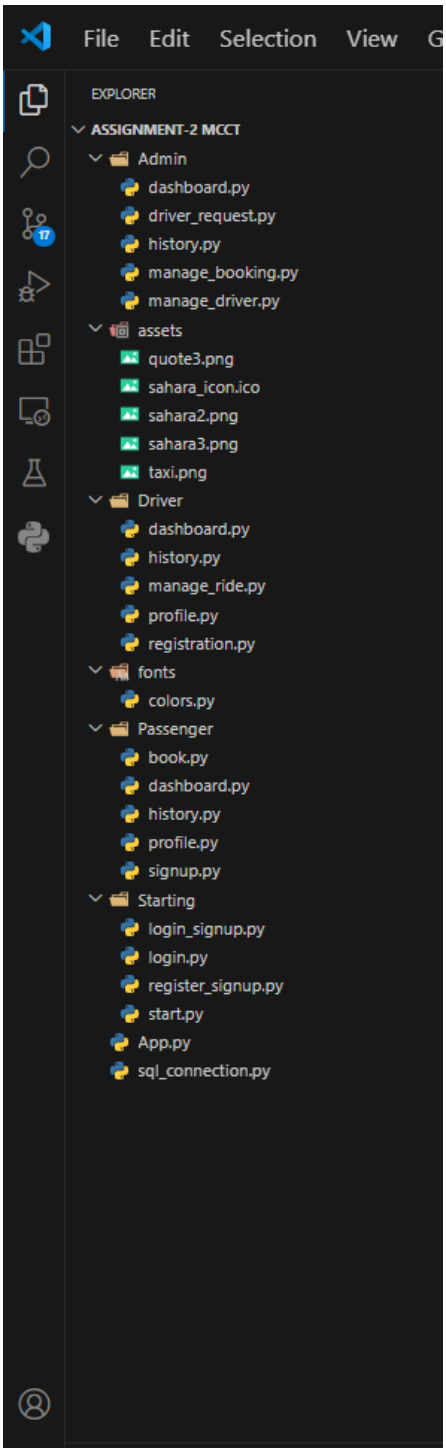
References

McVeigh-Murphy, A., 2020. *Abstraction in Computational Thinking | Learning.com.* [Online]
Available at: https://equip.learning.com/abstraction-computational-thinking
[Accessed 15 December 2024].

Team, L., 2024. *Defining Computational Thinking.* [Online]
Available at: https://www.learning.com/blog/defining-computational-thinking/
[Accessed 14 December 2024].

Appendix

List of files



1. sql_connection.py

2. App.py

3. Starting

- start.py
- login_signup.py
- login.py
- register_signup.py

4. Passenger

- signup.py
- dashboard.py
- book.py
- history.py
- profile.py

5. Admin

- dashboard.py
- driver_request.py
- history.py
- manage_booking.py
- manage_driver.py

6. Driver

- registration.py
- dashboard.py
- history.py
- manage_ride.py
- profile.py

7. assets

- images and icon

4. fonts

- colors.py

1. sql_connection.py

sql_connection.py

sql_connection.py > DatabaseConnection > connection

```python
import mysql.connector as conn
import bcrypt
class DatabaseConnection:
    message = ""
    try:
        @staticmethod #doesnt contain any dynamic methods like self.connection()
        def connection():
            # Initial connection without specifying a database
            connect = conn.connect(
                host='localhost',
                user='root', # change username here and at the return statement
                password='2005subash0910'  #  change password here and at the return statement
            )
            if connect.is_connected():
                DatabaseConnection.message = "Connected"
                cursor = connect.cursor()

                # Create the database if it doesn't exist
                cursor.execute("CREATE DATABASE IF NOT EXISTS sahara")
                cursor.execute("USE sahara")

                # Create admin table
                cursor.execute('''
                    CREATE TABLE IF NOT EXISTS admins (
                        id INT AUTO_INCREMENT PRIMARY KEY,
                        email VARCHAR(255) UNIQUE NOT NULL,
                        user_password VARCHAR(255) NOT NULL
                    );
                ''')

                # Create user_register table
                cursor.execute('''
                    CREATE TABLE IF NOT EXISTS passengers (
                        id INT AUTO_INCREMENT PRIMARY KEY,
                        full_name VARCHAR(255) NOT NULL,
                        phone_number VARCHAR(10) NOT NULL,
                        email VARCHAR(255) UNIQUE NOT NULL,
                        user_password VARCHAR(255) NOT NULL,
                        address VARCHAR (255) NOT NULL,
                        gender ENUM('Male', 'Female', 'Others') NOT NULL,
                        age INT
                    );
                ''')


                    # Create driver_request table
                    cursor.execute('''
                        CREATE TABLE IF NOT EXISTS driver_request (
                            id INT AUTO_INCREMENT PRIMARY KEY,
                            full_name VARCHAR(255) NOT NULL,
                            phone_number VARCHAR(10) NOT NULL,
                            email VARCHAR(255) UNIQUE NOT NULL,
                            user_password VARCHAR(255) NOT NULL,
                            address VARCHAR (255) NOT NULL,
                            license_number VARCHAR (255) UNIQUE NOT NULL,
                            vehicle_number VARCHAR (255) UNIQUE NOT NULL,
                            gender ENUM('Male', 'Female', 'Others') NOT NULL,
                            request_status ENUM('Approved','Requested','Rejected'),
                            admin_id INT ,
                            FOREIGN KEY (admin_id) REFERENCES admins(id)
                        );
                    ''')
```

```
                # Create driver table
                cursor.execute('''
                    CREATE TABLE IF NOT EXISTS drivers (
                        id INT AUTO_INCREMENT PRIMARY KEY,
                        full_name VARCHAR(255) NOT NULL,
                        phone_number VARCHAR(10) NOT NULL,
                        email VARCHAR(255) UNIQUE NOT NULL,
                        user_password VARCHAR(255) NOT NULL,
                        address VARCHAR (255) NOT NULL,
                        license_number VARCHAR (255) NOT NULL,
                        vehicle_number VARCHAR (255) NOT NULL,
                        driver_status ENUM('Online','Offline') NOT NULL,
                        gender ENUM('Male', 'Female', 'Others') NOT NULL,
                        admin_id INT ,
                        request_id INT ,
                        FOREIGN KEY (admin_id) REFERENCES admins(id),
                        FOREIGN KEY (request_id) REFERENCES driver_request(id)
                    );
                ''')

                # Create bookings table
                cursor.execute('''
                    CREATE TABLE IF NOT EXISTS bookings (
                        id INT AUTO_INCREMENT PRIMARY KEY,
                        passenger_id INT ,
                        driver_id INT ,
                        admin_id INT ,
                        pickup VARCHAR(255) NOT NULL,
                        dropoff VARCHAR(255) NOT NULL,
                        ride_date VARCHAR(255) NOT NULL,
                        ride_time VARCHAR(255) NOT NULL,
                        vehicle_type VARCHAR(255) NOT NULL,
                        fare VARCHAR(255) NOT NULL,
                        ride_status VARCHAR(255),
                        FOREIGN KEY (passenger_id) REFERENCES passengers(id),
                        FOREIGN KEY (admin_id) REFERENCES admins(id),
                        FOREIGN KEY (driver_id) REFERENCES drivers(id)
                    );
                ''')
```

```

                hashed_password1 = bcrypt.hashpw('admin'.encode('utf-8'), bcrypt.gensalt()).decode('utf-8')# Convert password to bytes and hash it
                hashed_password2 = bcrypt.hashpw('pcpscollege'.encode('utf-8'), bcrypt.gensalt()).decode('utf-8')# Convert password to bytes and hash it


                # Insert default admin if not already present
                cursor.execute('''INSERT IGNORE INTO admins (id, email, user_password) VALUES (%s, %s, %s), (%s, %s, %s)'''
                               , (1, 'admin@gmail.com', hashed_password1,2, 'pcpscollege@gmail.com', hashed_password2))
                # Commit the transaction to save changes
                connect.commit()
                cursor.close()
                connect.close()

            # Reconnect and specify the database now that it exists
            connection = conn.connect(
                host='localhost',
                user='root', # username
                password='2005subash0910', # password
                database='sahara' # name of the database
            )
            return connection

    except Exception as e:
        print(e)

```

2. App.py

App.py

App.py > MainApp > __init__

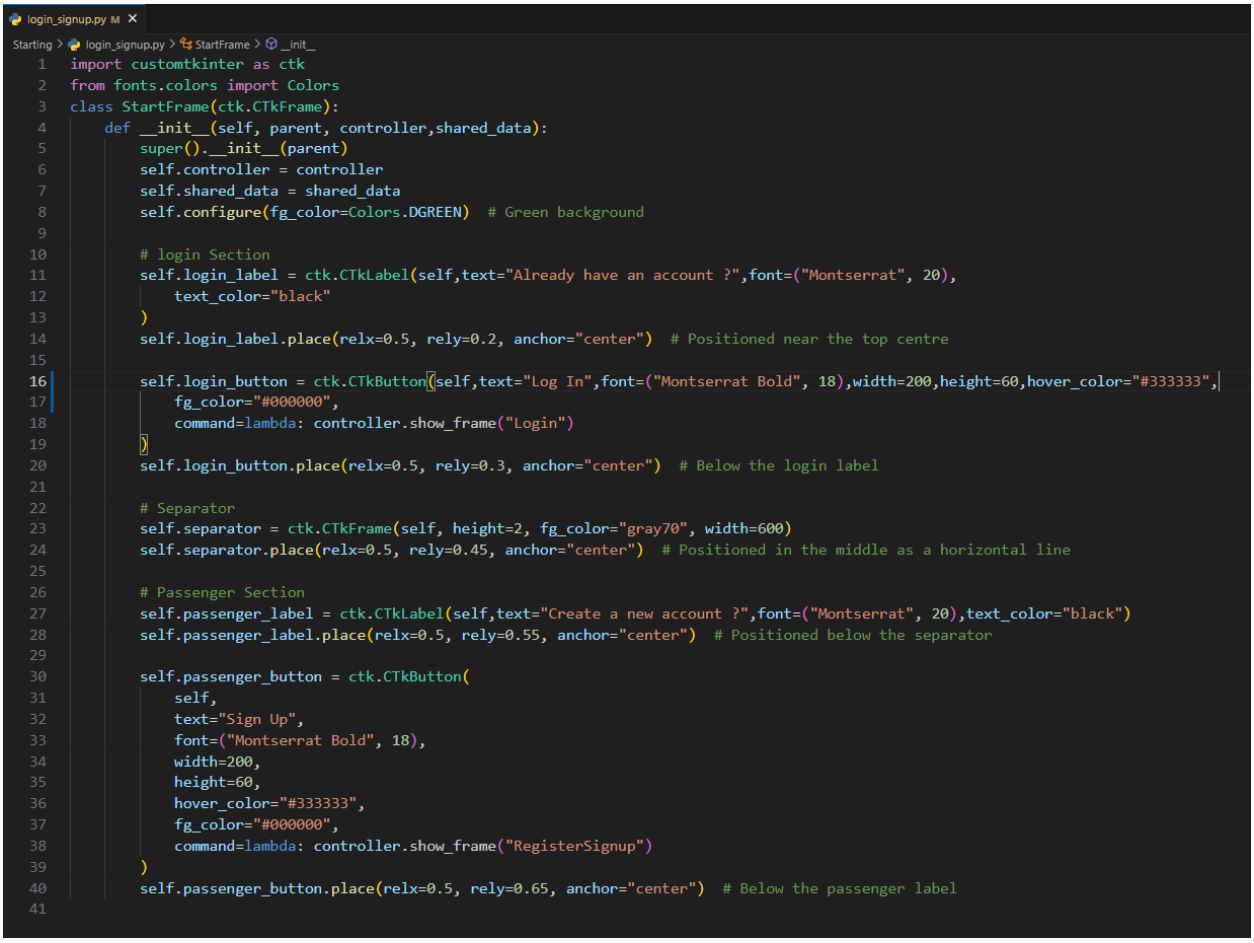
```python
import customtkinter as ctk
from sql_connection import DatabaseConnection
from Starting.start import Start
from Starting.login_signup import StartFrame
from Starting.register_signup import RegisterSignup
from Passenger.signup import PassengerSignUp
from Driver.registration import DriverRegistration
from Passenger.dashboard import PassengerDashboard
from Driver.dashboard import DriverDashboard
from Starting.login import Login
from Admin.dashboard import AdminDashboard
import os


ctk.set_appearance_mode("light")  # Appearance mode
ctk.set_default_color_theme("green")  # Theme color

class MainApp(ctk.CTk):
    def __init__(self):
        super().__init__() #Calls the initializer of the parent class (ctk.CTk)
        self.title("Sahara")
        self.geometry("800x600")  # Default size
        self.minsize(800, 600)   # Minimum size to avoid shrinking
        # Load the icon
        icon_path = os.path.join(os.path.dirname(__file__), "assets", "sahara_icon.ico")
        self.iconbitmap(icon_path)  # Set the custom icon
        DatabaseConnection.connection()

        # Data dictionary to store id when logged in
        self.shared_data = {
            "passenger_id": None,
            "driver_id": None,
            "admin_id": None,
        }
        print(f'{self.shared_data}, In main.py')

        # Create container frame for all screens
        self.container = ctk.CTkFrame(self)
        self.container.pack(fill="both", expand=True)

        # Configure row/column weights for responsiveness
        self.container.grid_rowconfigure(0, weight=1)
        self.container.grid_columnconfigure(0, weight=1)

        self.frames = {}  # Dictionary to store frames
        self.initialized_frames = set()  # Track initialized frames

        # Frame classes (lazy loading setup)
        self.frame_classes = {
            "Start": Start,
            "StartFrame": StartFrame,
            "RegisterSignup": RegisterSignup,
            "PassengerSignUp": PassengerSignUp,
            "DriverRegistration": DriverRegistration,
            "PassengerDashboard": PassengerDashboard,
            "DriverDashboard": DriverDashboard,
            "Login": Login,
            "AdminDashboard": AdminDashboard
        }

        self.show_frame("Start")  # Shows the initial frame

```

```
#The show_frame method abstracts away the complexity of switching between frames.

    def show_frame(self, page_name):
        """Display the specified frame, initializing it if necessary."""
        if page_name not in self.initialized_frames:
            # Initialize the frame and pass shared data
            frame_class = self.frame_classes[page_name]  # Initializse the frame by fetching the frame class from the frame_classes dictionary
            frame = frame_class(parent=self.container, controller=self, shared_data=self.shared_data)
            # Create an instance of the frame class, passing the container (parent), controller,
            #controller refers to the current instance of the class
            self.frames[page_name] = frame
            # Store the initialized frame in the frames dictionary with the page_name as the key
            frame.grid(row=0, column=0, sticky="nsew")
            # east west north south
            self.initialized_frames.add(page_name) #Marks the frame as initialized by adding the page_name to the initialized_frames set.
        frame = self.frames[page_name]
        # If showing the ProfileFrame, update the passenger_id
        if page_name == "PassengerDashboard" and "passenger_id" in self.shared_data:
            frame.update_id(self.shared_data["passenger_id"])
        # If showing the ProfileFrame, update the passenger_id
        if page_name == "DriverDashboard" and "driver_id" in self.shared_data:
            frame.update_id(self.shared_data["driver_id"])
        # If showing the ProfileFrame, update the passenger_id
        if page_name == "AdminDashboard" and "admin_id" in self.shared_data:
            frame.update_id(self.shared_data["admin_id"])

        frame.tkraise()  # Bring the frame to the front
#Different types of frames can be displayed using the same show_frame method, demonstrating the ability to handle different classes through a single interface.

if __name__ == "__main__":
    app = MainApp()
    app.mainloop()

```

3.1 start.py

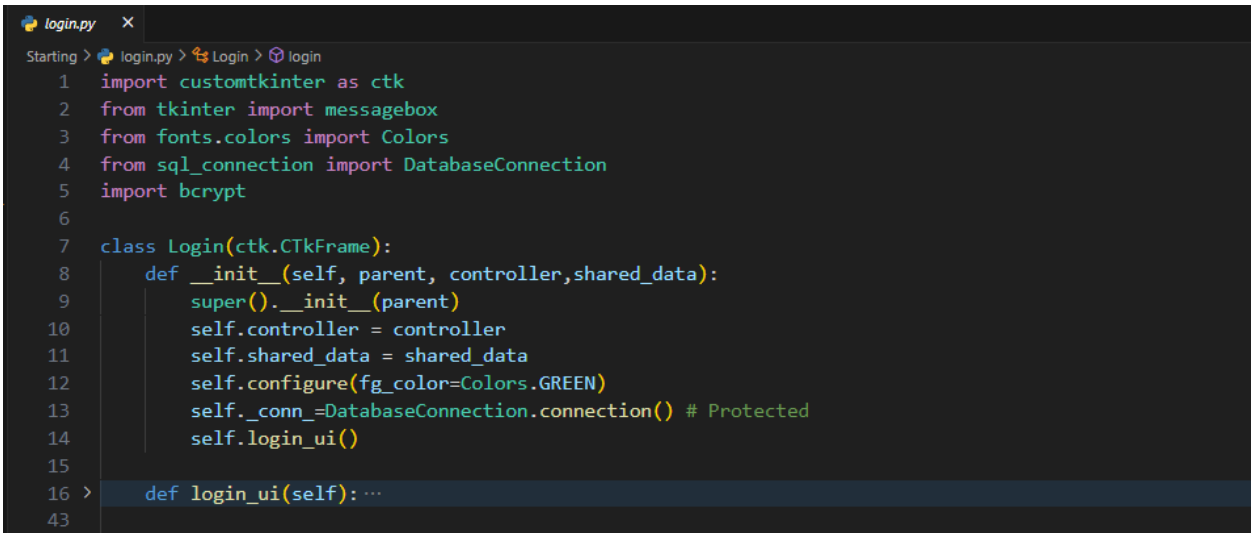
```
start.py
Starting > start.py > Start > __init__
import customtkinter as ctk
from fonts.colors import Colors  # Importing custom colour definitions
from PIL import Image, ImageTk  # Importing Pillow for image handling
import os  # for file path handling

class Start(ctk.CTkFrame):
    def __init__(self, parent, controller, shared_data):
        super().__init__(parent)
        self.controller = controller  # Reference to the main application controller
        self.shared_data = shared_data  # Shared data across frames
        self.configure(fg_color=Colors.DGREEN)  # Setting the background colour to dark green

        # Title text
        title_label = ctk.CTkLabel(self, text="Safety, Safety, Safety", font=("Montserrat", 28, "italic"), text_color="black")  # Creating a label for the title
        title_label.place(relx=0.5, rely=0.1, anchor="center")  # Positioning

        # Taxi image
        taxi_path = os.path.join(os.path.dirname(__file__), "..", "assets", "taxi.png")
        # Path to the taxi image file
        taxi_image = Image.open(taxi_path)  # Opening the taxi image
        self.taxi_photo = ImageTk.PhotoImage(taxi_image)  # Converting the image for Tkinter
        self.taxi_label = ctk.CTkLabel(self, image=self.taxi_photo, text="")
        # Creating a label to display the image
        self.taxi_label.place(relx=0.5, rely=0.5, anchor="center")  # Positioning the image label

        # Start label
        button_label = ctk.CTkLabel(
            self,
            text="Driver or Passenger",
            font=("Montserrat", 16),
            text_color="black"
        )  # Label to prompt user choice
        button_label.place(relx=0.5, rely=0.75, anchor="center")  # Positioning the prompt label

        # Start button
        start_button = ctk.CTkButton(self,text="Start",  font=("Montserrat Bold", 16),  width=200,  height=50,  fg_color="black",
            hover_color="#333333",
            command=lambda: controller.show_frame("StartFrame")
            # Command to switch to the "StartFrame"
        )
        start_button.place(relx=0.5, rely=0.85, anchor="center")  # Positioning the start button

```

3.2 login_signup.py



```python
import customtkinter as ctk
from fonts.colors import Colors
class StartFrame(ctk.CTkFrame):
    def __init__(self, parent, controller,shared_data):
        super().__init__(parent)
        self.controller = controller
        self.shared_data = shared_data
        self.configure(fg_color=Colors.DGREEN)  # Green background

        # login Section
        self.login_label = ctk.CTkLabel(self,text="Already have an account ?",font=("Montserrat", 20),
            text_color="black"
        )
        self.login_label.place(relx=0.5, rely=0.2, anchor="center")  # Positioned near the top centre

        self.login_button = ctk.CTkButton(self,text="Log In",font=("Montserrat Bold", 18),width=200,height=60,hover_color="#333333",
            fg_color="#000000",
            command=lambda: controller.show_frame("Login")
        )
        self.login_button.place(relx=0.5, rely=0.3, anchor="center")  # Below the login label

        # Separator
        self.separator = ctk.CTkFrame(self, height=2, fg_color="gray70", width=600)
        self.separator.place(relx=0.5, rely=0.45, anchor="center")  # Positioned in the middle as a horizontal line

        # Passenger Section
        self.passenger_label = ctk.CTkLabel(self,text="Create a new account ?",font=("Montserrat", 20),text_color="black")
        self.passenger_label.place(relx=0.5, rely=0.55, anchor="center")  # Positioned below the separator

        self.passenger_button = ctk.CTkButton(
            self,
            text="Sign Up",
            font=("Montserrat Bold", 18),
            width=200,
            height=60,
            hover_color="#333333",
            fg_color="#000000",
            command=lambda: controller.show_frame("RegisterSignup")
        )
        self.passenger_button.place(relx=0.5, rely=0.65, anchor="center")  # Below the passenger label

```

3.3 login.py



```python
import customtkinter as ctk
from tkinter import messagebox
from fonts.colors import Colors
from sql_connection import DatabaseConnection
import bcrypt

class Login(ctk.CTkFrame):
    def __init__(self, parent, controller,shared_data):
        super().__init__(parent)
        self.controller = controller
        self.shared_data = shared_data
        self.configure(fg_color=Colors.GREEN)
        self._conn_=DatabaseConnection.connection() # Protected
        self.login_ui()

    def login_ui(self):···

```

```python
    def login(self):
        email = self.email_entry.get()
        password = self.password_entry.get()
        self.customer=None # initializing attributes
        self.driver=None
        self.admin=None
        # Validation: Check if both email and password are provided
        if not email or not password:
            messagebox.showerror("Error", "All fields are required!")
            return False # Exit the function if validation fails
        try:
            self._conn_=DatabaseConnection.connection() # connection for dynamic changes

            # Attempt to connect to the MySQL database
            with self._conn_.cursor() as cursor:
                # Check for passenger login
                query1 = "SELECT id, user_password FROM passengers WHERE email = %s"
                cursor.execute(query1, (email,))
                result = cursor.fetchone()
                if result and bcrypt.checkpw(password.encode('utf-8'), result[1].encode('utf-8')):
                    self.customer = result[0]  # Passenger ID

                # Check for driver login
                query2 = "SELECT id, user_password FROM drivers WHERE email = %s"
                cursor.execute(query2, (email,))
                result = cursor.fetchone()
                if result and bcrypt.checkpw(password.encode('utf-8'), result[1].encode('utf-8')):
                    self.driver = result[0]  # Driver ID

                # Check for admin login
                query3 = "SELECT id, user_password FROM admins WHERE email = %s"
                cursor.execute(query3, (email,))
                result = cursor.fetchone()
                if result and bcrypt.checkpw(password.encode('utf-8'), result[1].encode('utf-8')):
                    self.admin = result[0]  # Admin ID

        except Exception as e:
            # Log the error if needed (optional)
            print(f"Error during login: {e}")
            messagebox.showerror("Error", "An unexpected error occurred. Please try again.")
            return False  # Exit the function if an error occurs

        # Handle login results
        if self.admin:
            self.shared_data["admin_id"] = self.admin  # Store admin ID for later use
            messagebox.showinfo("Success", "Successfully Logged In!")
            self.controller.show_frame("AdminDashboard")
        elif self.customer:
            self.shared_data["passenger_id"] = self.customer  # Store passenger ID for later use
            messagebox.showinfo("Success", "Successfully Logged In!")
            self.controller.show_frame("PassengerDashboard")
        elif self.driver:
            self.shared_data["driver_id"] = self.driver  # Store driver ID for later use
            messagebox.showinfo("Success", "Successfully Logged In!")
            self.controller.show_frame("DriverDashboard")
        else:
            # If no matches were found
            messagebox.showerror("Error", "Invalid email or password!")

```

3.4 register_signup.py

register_signup.py M

Starting > register_signup.py > ...

```python
import customtkinter as ctk
from fonts.colors import Colors
from PIL import Image, ImageTk
import os

class RegisterSignup(ctk.CTkFrame):
    def __init__(self, parent, controller,shared_data):
        super().__init__(parent)
        self.controller = controller
        self.shared_data = shared_data

        # Set background colour for the entire frame
        self.configure(fg_color="#E5E5E5")  # Light grey
        # Back Button

        # Sidebar
        self.sidebar = ctk.CTkFrame(self, fg_color=Colors.GREEN, corner_radius=0)
        self.sidebar.place(relx=0, rely=0, relwidth=0.2, relheight=1)  # Sidebar takes 20% width of the window

        # Add sidebar content
        self.add_sidebar_content()

        # Main content area
        self.main_frame = ctk.CTkFrame(self, fg_color="#E5E5E5", corner_radius=0)
        self.main_frame.place(relx=0.2, rely=0, relwidth=0.8, relheight=1)  # Main content takes 80% width of the window

        # Add main content
        self.add_main_content()

    def add_sidebar_content(self): ···

    def add_main_content(self):
        # Already have an account label
        account_label = ctk.CTkLabel(
            self.main_frame,
            text="Sign Up as a Passenger?",
            font=("Montserrat", 18),
            text_color="black",
        )
        account_label.place(relx=0.5, rely=0.2, anchor="center")

        # Log In button
        login_button = ctk.CTkButton(
            self.main_frame,
            text="Sign Up",
            font=("Montserrat", 16, "bold"),
            fg_color=Colors.GREEN,
            hover_color="#00CC00",
            width=200,
            height=50,
            text_color="black",
            command=lambda: self.controller.show_frame("PassengerSignUp")
        )
        login_button.place(relx=0.5, rely=0.3, anchor="center")

        # Create a new account label
        create_label = ctk.CTkLabel(self.main_frame,text="Register as a Driver ?",font=("Montserrat", 18),text_color="black",
        )
        create_label.place(relx=0.5, rely=0.5, anchor="center")

        # Sign Up button
        signup_button = ctk.CTkButton(
            self.main_frame,
            text="Register",
            font=("Montserrat", 16, "bold"),
            fg_color=Colors.GREEN,
            hover_color="#00CC00",
            width=200,
            height=50,
            text_color="black",
            command=lambda: self.controller.show_frame("DriverRegistration")
        )
        signup_button.place(relx=0.5, rely=0.6, anchor="center")

```

4. Passenger

4.1 signup.py

signup.py

Passenger > signup.py > PassengerSignUp > signup_ui

```python
import customtkinter as ctk
from tkinter import messagebox
from fonts.colors import Colors  # Importing custom colour definitions
import re  # Importing regex module for validation
from sql_connection import DatabaseConnection  # Importing custom database connection
import mysql.connector  # Importing MySQL connector for database interactions
import bcrypt  # Importing bcrypt for password hashing

class PassengerSignUp(ctk.CTkFrame):
    def __init__(self, parent, controller, shared_data):
        super().__init__(parent)
        self.controller = controller  # Reference to the main application controller
        self.shared_data = shared_data  # Shared data across frames
        self.configure(fg_color=Colors.GREEN)  # Setting the frame's background colour
        self._conn_ = DatabaseConnection.connection()  # Establishing a protected database connection
        self.signup_ui()  # Calling the UI setup method

    def signup_ui(self):
        # Back Button
        self.back_button = ctk.CTkButton(
            self,
            text="Back",
            width=80,
            height=32,
            corner_radius=16,
            fg_color="black",
            hover_color="#333333",
            command=lambda: self.controller.show_frame("RegisterSignup")
        )  # Button to navigate back to the registration screen
        self.back_button.place(x=20, y=20)  # Positioning at the top-left corner

        # Title
        self.title = ctk.CTkLabel(
            self,
            text="Sign Up",
            font=("Montserrat Bold", 24),
            text_color="black"
        )  # Label for the title
        self.title.place(relx=0.46, y=20, anchor="n")  # Centering the title

        # Full Name Entry
        self.name_label = ctk.CTkLabel(
            self,
                text="Full Name",
                font=("Montserrat Bold", 14),
                text_color="black"
            )  # Label for the name field
            self.name_label.place(relx=0.18, y=100)

            self.name_entry = ctk.CTkEntry(
                self,
                width=400,
                height=45,
                border_width=0,
                fg_color="white",
                placeholder_text="Enter your full name"
            )  # Input field for the full name
            self.name_entry.place(relx=0.18, y=140)

```

```
60          # Phone Number Entry
61          self.phone_label = ctk.CTkLabel(
62              self,
63              text="Phone Number",
64              font=("Montserrat Bold", 14),
65              text_color="black"
66          )  # Label for the phone number field
67          self.phone_label.place(relx=0.18, y=220)
68
69          self.phone_entry = ctk.CTkEntry(
70              self,
71              width=400,
72              height=45,
73              border_width=0,
74              fg_color="white",
75              placeholder_text="Enter your phone number"
76          )  # Input field for the phone number
77          self.phone_entry.place(relx=0.18, y=260)
78
79          # Email Entry
80          self.email_label = ctk.CTkLabel(
81              self,
82              text="Email",
83              font=("Montserrat Bold", 14),
84              text_color="black"
85          )  # Label for the email field
86          self.email_label.place(relx=0.18, y=340)
87
88          self.email_entry = ctk.CTkEntry(
89              self,
90              width=400,
91              height=45,
92              border_width=0,
93              fg_color="white",
94              placeholder_text="Enter your email"
95          )  # Input field for the email
96          self.email_entry.place(relx=0.18, y=380)
97
```

```
        # Password Entry
        self.password_label = ctk.CTkLabel(
            self,
            text="Set Password",
            font=("Montserrat Bold", 14),
            text_color="black"
        )  # Label for the password field
        self.password_label.place(relx=0.18, y=460)

        self.password_entry = ctk.CTkEntry(
            self,
            width=400,
            height=45,
            border_width=0,
            fg_color="white",
            placeholder_text="Enter your password",
            show="*"
        )  # Input field for the password with masked input
        self.password_entry.place(relx=0.18, y=500)

        # Address Entry
        self.address_label = ctk.CTkLabel(
            self,
            text="Home Address",
            font=("Montserrat Bold", 14),
            text_color="black"
        )  # Label for the address field
        self.address_label.place(relx=0.5, y=100)

        self.address_entry = ctk.CTkEntry(
            self,
            width=400,
            height=45,
            border_width=0,
            fg_color="white",
            placeholder_text="Enter your address"
        )  # Input field for the address
        self.address_entry.place(relx=0.5, y=140)

```

```
        # Gender Selection
        self.gender_label = ctk.CTkLabel(
            self,
            text=" Gender",
            font=("Montserrat Bold", 14),
            text_color="black"
        )  # Label for gender selection
        self.gender_label.place(relx=0.5, y=220)

        self.gender_var = ctk.StringVar(value="Male")  # Default gender selection variable
        gender_frame = ctk.CTkFrame(self, width=400, height=45, fg_color="white")
        # Frame for radio buttons
        gender_frame.place(relx=0.5, y=260)

        self.male_radio = ctk.CTkRadioButton(
            gender_frame,
            text="Male",
            value="Male",
            variable=self.gender_var
        )  # Radio button for male gender
        self.male_radio.place(x=10, y=12)

        self.female_radio = ctk.CTkRadioButton(
            gender_frame,
            text="Female",
            value="Female",
            variable=self.gender_var
        )  # Radio button for female gender
        self.female_radio.place(x=130, y=12)

        self.others_radio = ctk.CTkRadioButton(
            gender_frame,
            text="Others",
            value="Others",
            variable=self.gender_var
        )  # Radio button for other genders
        self.others_radio.place(x=260, y=12)

            # Age Entry
            self.age_label = ctk.CTkLabel(
                self,
                text="Age",
                font=("Montserrat Bold", 14),
                text_color="black"
            )  # Label for the age field
            self.age_label.place(relx=0.5, y=340)

            self.age_entry = ctk.CTkEntry(
                self,
                width=400,
                height=45,
                border_width=0,
                fg_color="white",
                placeholder_text="Enter your age"
            )  # Input field for the age
            self.age_entry.place(relx=0.5, y=380)

```

```
194         # Confirm Password Entry
195         self.confirm_password_label = ctk.CTkLabel(
196             self,
197             text="Confirm Password",
198             font=("Montserrat Bold", 14),
199             text_color="black"
200         )  # Label for confirm password field
201         self.confirm_password_label.place(relx=0.5, y=460)
202 
203         self.confirm_password_entry = ctk.CTkEntry(
204             self,
205             width=400,
206             height=45,
207             border_width=0,
208             fg_color="white",
209             placeholder_text="Confirm your password",
210             show="*"
211         )  # Input field for confirming the password
212         self.confirm_password_entry.place(relx=0.5, y=500)
213 
214         # Sign Up Button
215         self.signup_button = ctk.CTkButton(
216             self,
217             width=150,
218             height=40,
219             text="Sign Up",
220             font=("Montserrat Bold", 14),
221             text_color='white',
222             corner_radius=5,
223             fg_color="black",
224             hover_color="#006400",
225             command=self.sign_up
226         )  # Button to submit the form
227         self.signup_button.place(relx=0.46, y=600)
228 
229     def sign_up(self):
230         # Validate the form data
231         if not self.validate():
232             return
```

```
233
234         try:
235             cursor = self._conn_.cursor()
236             name = self.name_entry.get()
237             number = self.phone_entry.get()
238             email = self.email_entry.get()
239             password = self.password_entry.get()
240             hashed_password = bcrypt.hashpw(password.encode('utf-8'), bcrypt.gensalt()).decode('utf-8')
241             # Hash the password
242             gender = self.gender_var.get()
243             age = self.age_entry.get()
244             address = self.address_entry.get()
245             query = """INSERT INTO passengers (full_name,phone_number,email,user_password,gender,age,address)
246                        VALUES (%s,%s,%s,%s,%s,%s,%s)"""
247             cursor.execute(query, (name, number, email, hashed_password, gender, age, address))
248             self._conn_.commit()  # Commit the changes to the database
249             print("Successfully Inserted Data in database")
250             messagebox.showinfo("Success", "Successfully Signed Up!")
251             self.reset_form()  # Reset the form fields
252         except mysql.connector.Error as e:
253             messagebox.showerror("Database Error", f"An error occurred: {e}")
254         finally:
255             self.controller.show_frame('Login')  # Navigate to login frame
256
257     def reset_form(self):
258         # Clear all form fields
259         self.name_entry.delete(0, "end")
260         self.phone_entry.delete(0, "end")
261         self.email_entry.delete(0, "end")
262         self.password_entry.delete(0, "end")
263         self.confirm_password_entry.delete(0, "end")
264         self.age_entry.delete(0, "end")
265         self.address_entry.delete(0, "end")
266
267     def validate(self):
268         # Validation of user inputs
269         name = self.name_entry.get()
270         contact = self.phone_entry.get()
271         email = self.email_entry.get()
272         password = self.password_entry.get()
273             age = self.age_entry.get()
274             confirm_password = self.confirm_password_entry.get()
275
276             if not name or not contact or not email or not password:
277                 messagebox.showerror("Error", "All fields are required!")
278                 return False
279
280             if not re.match(r"^[0-9]{10}$", contact):
281                 messagebox.showerror("Error", "Invalid contact number!")
282                 return False
283
284             if not re.match(r"[^@]+@[^@]+\.[^@]+", email):
285                 messagebox.showerror("Error", "Invalid email format!")
286                 return False
287
```

```
288         if password != confirm_password:
289             messagebox.showerror("Error", "Passwords do not match!")
290             return False
291 
292         if len(password) < 6 or len(password) > 18:
293             messagebox.showerror("Error", "Password must be between 6 and 18 characters!")
294             return False
295 
296         if not age.isdigit():
297             messagebox.showerror("Error", "Invalid age!")
298             return False
299 
300         age = int(age)
301         if age < 5 or age > 100:
302             messagebox.showerror("Error", "Invalid age!")
303             return False
304 
305         return True
306 
```

4.2 dashboard.py

dashboard.py M

Passenger > dashboard.py > PassengerDashboard > __init__

```
1   import customtkinter as ctk
2   from PIL import Image,ImageTk
3   import os
4   from Passenger.book import BookingFrame
5   from Passenger.profile import ProfileFrame
6   from Passenger.history import RideHistoryFrame
7   from fonts.colors import Colors
8   
9   class PassengerDashboard(ctk.CTkFrame):
10      def __init__(self, parent, controller, shared_data=None):
11          super().__init__(parent)
12          self.shared_data = shared_data
13          self.controller = controller
14  
15          self.passenger_id = self.shared_data.get("passenger_id", None)
16          # self.update_id(self.passenger_id)
17          # Get initial passenger ID
18          print(f"Passenger ID after update id in Passenger Dashboard : {self.passenger_id}")  # Debugging
19  
20          # Sidebar
21          self.sidebar = ctk.CTkFrame(self, width=200, height=800, corner_radius=0)
22          self.sidebar.place(x=0, y=0)
23  
24          # Taxi image
25          # Load and resize the image
26          sahara_path = os.path.join(os.path.dirname(__file__), "..", "assets", "sahara3.png")
27          sahara_image = Image.open(sahara_path)
28          resized_sahara_image = sahara_image.resize((250, 85))  # Resize to 100x100 pixels (adjust as needed)
29  
30          # Convert the resized image to PhotoImage
31          self.sahara_photo = ImageTk.PhotoImage(resized_sahara_image)
32  
33          # Create and place the label
34          self.sahara_label = ctk.CTkLabel(self.sidebar, image=self.sahara_photo, text="")
35          self.sahara_label.place(x=110, y=50, anchor="center")
36  
```

```
        # Navigation buttons
        nav_items = ["Book Ride", "Ride Details", "Profile"]
        self.nav_buttons = []

        for i, item in enumerate(nav_items):
            button = ctk.CTkButton(
                self.sidebar, text=item, width=160, fg_color=Colors.GREEN_BUTTON,
                hover_color=Colors.GREEN_BUTTON_HOVER, command=lambda x=item: self.switch_frame(x)
            )
            button.place(x=20, y=100 + i * 50)
            self.nav_buttons.append(button)

        # Logout Button
        logout = ctk.CTkButton(
            self.sidebar, text='Log Out', width=160, fg_color=Colors.RED,
            hover_color=Colors.HOVER_RED, command=lambda: controller.show_frame('Login')
        )
        logout.place(x=20, y=700)

        # Main content area
        self.main_frame = ctk.CTkFrame(self, width=960, height=760)
        self.main_frame.place(x=220, y=20)

        # Initialise frames
        self.frames = {
            "Book Ride": BookingFrame(self.main_frame,self.passenger_id),
            "Ride Details": RideHistoryFrame(self.main_frame, self.passenger_id),
            "Profile": ProfileFrame(self.main_frame, self.passenger_id)
        }

        # Show default frame
        self.current_frame = self.frames["Book Ride"]
        self.current_frame.place(x=0, y=0, relwidth=1, relheight=1)

    def update_id(self, passenger_id=None):
        """Update the passenger_id and refresh the user profile."""
        self.passenger_id = passenger_id
        print(f"Passenger id inside update id {self.passenger_id}")
        # Reinitialize frames with the updated passenger_id
        self.frames = {
            "Book Ride": BookingFrame(self.main_frame, self.passenger_id),
            "Ride Details": RideHistoryFrame(self.main_frame, self.passenger_id),
            "Profile": ProfileFrame(self.main_frame, self.passenger_id)
        }

        # Update the currently displayed frame
        if self.current_frame:
            self.current_frame.place_forget()

        # Default frame to show after update
        self.current_frame = self.frames["Book Ride"]
        self.current_frame.place(x=0, y=0, relwidth=1, relheight=1)

    def switch_frame(self, frame_name):
        # Hide current frame
        self.current_frame.place_forget()

        # Show selected frame
        self.current_frame = self.frames[frame_name]
        # Refresh the frame if it has a refresh method
        if hasattr(self.current_frame, "connection_and_history"):
            self.current_frame.connection_and_history()
        self.current_frame.place(x=0, y=0, relwidth=1, relheight=1)
```

4.3 book.py

book.py M

Passenger > book.py > BookingFrame > _init_

```python
import customtkinter as ctk
from datetime import datetime
from tkinter import messagebox
from sql_connection import DatabaseConnection

class BookingFrame(ctk.CTkFrame): # Define the 'BookingFrame' class, which is a subclass of 'CTkFrame'
    def __init__(self, parent,passenger_id=None):

        #The constructor (__init__) method is called when an instance of the class is created.
        #It initializes the frame and stores the passenger_id if provided.
        self.passenger_id = passenger_id
        self._conn_=DatabaseConnection.connection()
        super().__init__(parent, width=parent.winfo_width(), height=parent.winfo_height())

        # Make the frame expand to fill the parent
        self.pack_propagate(False)
        self. booking_ui()

    def booking_ui(self):···

    def update_fare(self, *args):
        # Fare calculation based on vehicle type
        base_fare = {
            "Standard": 1000,
            "Premium": 2000,
            "Luxury": 3000
        }

        self.selected_fare = base_fare[self.vehicle_var.get()] # store taxi-type fare in selected_fare

        if self.pickup_entry.get() and self.dropoff_entry.get():# Add distance-based fare if locations are selected

            # storing pickup and dropoff location
            pickup = self.pickup_entry.get()
            dropoff = self.dropoff_entry.get()

        #Fare calculations
        if pickup == dropoff:
            self.selected_fare = 0.00
        elif( pickup == "Kathmandu" and dropoff=="Pokhara") or ( dropoff == "Kathmandu" and pickup=="Pokhara"):# 2000 Rs added to base_fare for th
            self.selected_fare = self.selected_fare + 2000
        elif( pickup == "Kathmandu" and dropoff=="Lalitpur") or ( dropoff == "Kathmandu" and pickup=="Lalitpur"):# 500 Rs added to base_fare for t
            self.selected_fare = self.selected_fare + 500
        elif( pickup == "Kathmandu" and dropoff=="Bhaktapur") or ( dropoff == "Kathmandu" and pickup=="Bhaktapur"):# 500 Rs added to base_fare for
            self.selected_fare = self.selected_fare + 500
        elif( pickup == "Kathmandu" and dropoff=="Biratnagar") or ( dropoff == "Kathmandu" and pickup=="Biratnagar"):# 3000 Rs added to base_fare
            self.selected_fare = self.selected_fare + 3000
        elif( pickup == "Pokhara" and dropoff=="Lalitpur") or ( dropoff == "Pokhara" and pickup=="Lalitpur"):# 2000 Rs added to base_fare for this
            self.selected_fare = self.selected_fare + 2000
        elif( pickup == "Pokhara" and dropoff=="Bhaktapur") or ( dropoff == "Pokhara" and pickup=="Bhaktapur"):# 2000 Rs added to base_fare for th
            self.selected_fare = self.selected_fare + 2000
        elif( pickup == "Pokhara" and dropoff=="Biratnagar") or ( dropoff == "Pokhara" and pickup=="Biratnagar"):# 2000 Rs added to base_fare for
            self.selected_fare = self.selected_fare + 2000
        elif( pickup == "Lalitpur" and dropoff=="Bhaktapur") or ( dropoff == "Lalitpur" and pickup=="Bhaktapur"):# 500 Rs added to base_fare for t
            self.selected_fare = self.selected_fare + 500
        elif( pickup == "Lalitpur" and dropoff=="Biratnagar") or ( dropoff == "Lalitpur" and pickup=="Biratnagar"):# 3000 Rs added to base_fare fo
            self.selected_fare = self.selected_fare + 3000
        elif( pickup == "Bhaktapur" and dropoff=="Biratnagar") or ( dropoff == "Bhaktapur" and pickup=="Biratnagar"):# 3000 Rs added to base_fare
            self.selected_fare = self.selected_fare + 3000


            self.fare_label.configure(text=f"Estimated Fare: Rs {self.selected_fare:.2f}")

      # on clicking the book button this function is called
```

```
    def book_ride(self):
        # setting up all the entries
        pickup = self.pickup_entry.get()
        dropoff = self.dropoff_entry.get()
        date = f"{self.year_entry.get()}-{self.month_entry.get()}-{self.day_entry.get()}"
        time = f"{self.hour_entry.get()}:{self.minute_entry.get()}"
        vehicle = self.vehicle_var.get()
        fare=self.selected_fare
        ride_status="Pending"
        id=self.passenger_id
        # Display a confirmation dialog to the user
        confirmation_message = f"Do you want to book the ride with the following details?
        \n\nPickup: {pickup}\nDropoff: {dropoff}\nDate: {date}\nTime: {time}\nVehicle: {vehicle}\nFare: {fare}"

        if(fare>0):
        # Ask for user confirmation
            response = messagebox.askyesno("Confirm Booking", confirmation_message)
            if response:
                # mysql connection and sending dadta to bookings table
                try:
                    with  self._conn_.cursor() as cursor:
                        query = "INSERT INTO bookings (passenger_id, pickup,dropoff,ride_date,ride_time,vehicle_type,fare,ride_status) VALUES (%s,%s,%s,%s,%s,%s,%s,%s)"
                        cursor.execute(query,(id,pickup,dropoff,date,time,vehicle,fare,ride_status))
                        self._conn_.commit()

                except Exception as e:
                    messagebox.showerror("Error", f"An unexpected error occurred: {e}")
                finally:
                    messagebox.showinfo("Success", "Taxi Booked successfully!")
        else:
            messagebox.showerror("Error","Invalid Booking")

```

4.4 history.py

history.py

Passenger > history.py > RideHistoryFrame > connection_and_history

```
import customtkinter as ctk
from tkinter import messagebox
from fonts.colors import Colors
from sql_connection import DatabaseConnection

class RideHistoryFrame(ctk.CTkFrame):
    def __init__(self, parent,passenger_id=None):
        super().__init__(parent, width=parent.winfo_width(), height=parent.winfo_height())
        self.passenger_id = passenger_id
        self._conn_=DatabaseConnection.connection()
        self.connection_and_history()

    def connection_and_history(self):
        # Clear existing widgets
        for widget in self.winfo_children():
            widget.destroy()

        # Initialize ride details list
        self._conn_=DatabaseConnection.connection()
        self.pending_ride_details = []
        self.other_status_ride_details = []
        cursor = self._conn_.cursor()
        query="""SELECT
                    b.id,
                    d.full_name AS driver_name,
                    b.pickup,
                    b.dropoff,
                    b.ride_date,
                    b.ride_time,
                    b.vehicle_type,
                    b.fare,
```

```
                    b.ride_status
                FROM
                    bookings b
                LEFT JOIN
                    drivers d
                ON
                    b.driver_id = d.id -- Correct join condition
                WHERE
                    b.passenger_id = %s;
            """

        cursor.execute(query,[self.passenger_id])
        self.dets = cursor.fetchall()

        # Header
        self.header = ctk.CTkLabel(self,text="Ride History",font=ctk.CTkFont(size=24, weight="bold"))
        self.header.pack(pady=(20, 10))  # Space around the header

        # Create scrollable frame for ride history
        self.scrollable_frame = ctk.CTkScrollableFrame(self,width=int(self.winfo_width() * 0.95),height=600)
        self.scrollable_frame.pack(padx=20, pady=(0, 20), fill="both", expand=True)

        # Process fetched data into ride_details format
        if self.dets:
            for ride in self.dets:
                ride_dict = {
                    "id":ride[0],
                    "driver": ride[1],  # Assuming column 4 is the driver name
                    "from": ride[2],  # Assuming column 5 is the 'from' location
                    "to": ride[3],  # Assuming column 6 is the 'to' location
                    "date": ride[4],  # Assuming column 7 is the date
                    "time": ride[5],  # Assuming column 8 is the time
                    "vehicle": ride[6],  # Assuming cloumn 9 is vehicle type
                    "fare": f"Rs {ride[7]}",  # Assuming column 10 is the fare
                    "status": ride[8]  # Assuming column 11 is the ride status
                }
                # Check the status of the ride and categorize it
                if ride_dict["status"] == "Pending" or ride_dict["status"] == "Assigned" :
                    self.pending_ride_details.append(ride_dict)  # Add to pending rides
                else:
                    self.other_status_ride_details.append(ride_dict)  # Add to other status rides
        else:
                self.no_ride_history()


        # Sample ride history
        for i, ride in enumerate(self.pending_ride_details):
            self.pending_ride_card(ride)
        for i, ride in enumerate(self.other_status_ride_details):
            self.other_status_ride_card(ride)

    def no_ride_history(self):
        self.scrollable_frame.pack(fill="both", expand=True)  # Make parent frame fill the entire window.
        frame = ctk.CTkFrame(self.scrollable_frame)
        frame.pack(fill="both", expand=True)  # Make the frame fill the parent container.
        ctk.CTkLabel(frame, text=f"No Ride history").pack(side="left", padx=10)

    def pending_ride_card(self, ride):
        # Card container
        card = ctk.CTkFrame(self.scrollable_frame,width=int(self.scrollable_frame.winfo_width() * 0.95),height=160)
        card.pack(pady=10, padx=10, fill="x")  # Add padding and fill horizontally

```

```
        # Date and status (placed at the top of the card)
        header_frame = ctk.CTkFrame(card, height=30)
        header_frame.pack(pady=5, fill="x")

        ctk.CTkLabel(header_frame,text=f"Date: {ride['date']}   {ride['time']}",font=ctk.CTkFont(size=14)).pack(side="left", padx=10)

        status_color = "#0000ff"  # Blue for pending

        ctk.CTkLabel(header_frame,text=ride["status"],text_color=status_color,font=ctk.CTkFont(size=14, weight="bold")).pack(side="right", padx=10)
        # Driver information
        ctk.CTkLabel(card,text=f"Driver: {ride['driver']}",font=ctk.CTkFont(size=14)).pack(pady=(5, 0), anchor="w", padx=10)

        # Route information
        ctk.CTkLabel(card,text=f"From: {ride['from']}",).pack(pady=(5, 0), anchor="w", padx=10)
        ctk.CTkLabel(card,text=f"To: {ride['to']}").pack(pady=(5, 0), anchor="w", padx=10)

        # Fare information
        ctk.CTkLabel(card,text=f"Fare: {ride['fare']}",font=ctk.CTkFont(size=14, weight="bold")).pack(pady=(5, 10), anchor="w", padx=10)

        # Buttons placed at the top of the card
        footer_frame = ctk.CTkFrame(card, height=30)
        footer_frame.pack(pady=5, fill="x")
        ctk.CTkButton(footer_frame,text="Complete",hover_color=Colors.GREEN_BUTTON_HOVER,
                      width=90,fg_color=Colors.GREEN_BUTTON,font=ctk.CTkFont(size=16, weight="bold") ,
                      command=lambda: self.complete_ride(ride['id'],ride['driver'])).pack(side="left")
        ctk.CTkButton(footer_frame,text="Cancel",hover_color=Colors.HOVER_RED,
                      width=90,fg_color=Colors.RED,font=ctk.CTkFont(size=16, weight="bold") ,
                      command=lambda: self.cancel_ride(ride['id'])).pack(side="right")
    def complete_ride(self,id,driver):
        print(driver)
        response = messagebox.askyesno("Confirm Complete", "Are you sure you want to complete the ride ?")
        if response:
            if not driver:
                messagebox.showerror("Error", "Can't complete ride without driver ! ")

            else:
                cursor=self._conn_.cursor()
                cursor.execute("UPDATE bookings SET ride_status='Completed' WHERE id = %s",(id,))
                self._conn_.commit()
                    self.connection_and_history()# for dynamic changes
                    messagebox.showinfo("Success", "Ride completed ! ")
    def cancel_ride(self,id):
        response = messagebox.askyesno("Confirm ", "Are you sure you want to cancel the ride ?")
        if response:
            cursor=self._conn_.cursor()
            cursor.execute("UPDATE bookings SET ride_status='Cancelled' WHERE id = %s",(id,))
            self._conn_.commit()
            self.connection_and_history() # for dynamic changes
            messagebox.showinfo("Success", "Ride cancelled ! ")


    def other_status_ride_card(self, ride):
        # Card container
        card = ctk.CTkFrame(self.scrollable_frame,width=int(self.scrollable_frame.winfo_width() * 0.95),height=160)
        card.pack(pady=10, padx=10, fill="x")  # Add padding and fill horizontally

        # Date and status (placed at the top of the card)
        header_frame = ctk.CTkFrame(card, height=30)
        header_frame.pack(pady=5, fill="x")

        ctk.CTkLabel(header_frame,text=f"Date: {ride['date']}   {ride['time']}",font=ctk.CTkFont(size=14)).pack(side="left", padx=10)
        status_color = None
        status_color = Colors.GREEN_BUTTON if  ride["status"] == "Completed" else Colors.RED
        ctk.CTkLabel(header_frame,text=ride["status"],text_color=status_color,font=ctk.CTkFont(size=14, weight="bold")).pack(side="right", padx=10)
        # Driver information
        ctk.CTkLabel(card,text=f"Driver: {ride['driver']}",font=ctk.CTkFont(size=14)).pack(pady=(5, 0), anchor="w", padx=10)
        # Route information
        ctk.CTkLabel(card,text=f"From: {ride['from']}",).pack(pady=(5, 0), anchor="w", padx=10)
        ctk.CTkLabel(card,text=f"To: {ride['to']}").pack(pady=(5, 0), anchor="w", padx=10)
        # Fare information
        ctk.CTkLabel(card,text=f"Fare: {ride['fare']}",font=ctk.CTkFont(size=14, weight="bold")).pack(pady=(5, 10), anchor="w", padx=10)

```

4.5 profile.py

profile.py

Passenger > profile.py > ProfileFrame > user_profile

```python
import customtkinter as ctk  # Importing CustomTkinter for enhanced UI components
from tkinter import messagebox  # Importing messagebox for error and success dialogs
from sql_connection import DatabaseConnection  # Importing the custom database connection module
from fonts.colors import Colors  # Importing custom colour constants

class ProfileFrame(ctk.CTkFrame):
    """
    A frame to display and manage passenger profiles.
    """
    def __init__(self, parent, passenger_id=None):
        super().__init__(parent, width=parent.winfo_width(), height=parent.winfo_height())
        self.passenger_id = passenger_id  # Storing the passenger ID for later use
        print(f"Passenger id in profile frame: {self.passenger_id}")

        self._conn_ = DatabaseConnection.connection()  # Establishing a database connection
        self.user_profile()  # Display the profile UI

    def user_profile(self):
        """
        Displays the passenger's profile information.
        """
        # Clear existing widgets
        for widget in self.winfo_children():
            widget.destroy()

        self._conn_ = DatabaseConnection.connection()  # Refreshing the database connection
        try:
            with self._conn_.cursor() as cursor:
                query = "SELECT * FROM passengers WHERE id = %s"
                cursor.execute(query, (self.passenger_id,))
                self.details = cursor.fetchone()  # Fetching the user's details
        except Exception as e:
            messagebox.showerror("Error", f"An unexpected error occurred: {e}")

        # Header
        self.header = ctk.CTkLabel(self, text="Passenger Profile", font=ctk.CTkFont(size=24, weight="bold"))
        self.header.place(x=20, y=20)

        # User details
        self.user_data = [
            ("Name", self.details[1]),
            ("Phone", self.details[2]),
            ("Email", self.details[3]),
            ("Home Address", self.details[5]),
            ("Gender", self.details[6]),
            ("Age", self.details[7]),
            ("Rating", "4.9★")
        ]

        # Display the user information
        for i, (label, value) in enumerate(self.user_data, start=1):
            ctk.CTkLabel(self, text=label, font=ctk.CTkFont(size=14)).place(x=20, y=20 + i * 50)
            ctk.CTkLabel(self, text=value, font=ctk.CTkFont(size=14, weight="bold")).place(x=200, y=20 + i * 50)

        # Edit Profile button
        self.edit_button = ctk.CTkButton(
            self,
            text="Edit Profile",
            width=440,
            fg_color=Colors.GREEN_BUTTON,
            hover_color=Colors.GREEN_BUTTON_HOVER,
            command=self.edit_profile
        )
        self.edit_button.place(x=20, y=500)

    def edit_profile(self):
        """
        Opens the Edit Profile window.
        """
        self.edit_window = EditProfileWindow(self, self.passenger_id)
```

```python

class EditProfileWindow(ctk.CTkToplevel):
    """
    A window for editing the user's profile information.
    """
    def __init__(self, parent, id=None):
        super().__init__(parent)
        self.id = id  # Storing the passenger ID
        self.parent = parent  # Reference to the parent frame
        self._conn_ = DatabaseConnection.connection()  # Establishing a database connection
        self.title("Edit Profile")
        self.geometry("500x600")
        self.resizable(False, False)  # Prevent resizing the window

        try:
            with self._conn_.cursor() as cursor:
                query = "SELECT * FROM passengers WHERE id = %s"
                cursor.execute(query, (self.id,))
                self.details = cursor.fetchone()  # Fetching the user's details
        except Exception as e:
            messagebox.showerror("Error", f"An unexpected error occurred: {e}")

        # Title Label
        title_label = ctk.CTkLabel(self, text="Edit Profile", font=ctk.CTkFont(size=20, weight="bold"))
        title_label.place(x=250, y=20, anchor="center")

        # Editable user fields
        self.user_data = [
            ("Name", self.details[1]),
            ("Email", self.details[4]),
            ("Phone", self.details[2]),
            ("Home Address", self.details[5]),
            ("Gender", self.details[6]),
            ("Age", self.details[7])
        ]

        # Creating input fields for user data
        self.inputs = {}
        y_position = 80
        for label, value in self.user_data:
                # Label
                label_widget = ctk.CTkLabel(self, text=label, font=ctk.CTkFont(size=14))
                label_widget.place(x=50, y=y_position, anchor="w")

                # Input field
                entry_widget = ctk.CTkEntry(self, width=250)
                entry_widget.insert(0, value)
                entry_widget.place(x=160, y=y_position, anchor="w")
                self.inputs[label] = entry_widget

                y_position += 50

          # Confirm button
          self.confirm_button = ctk.CTkButton(
              self,
              text="Confirm Profile",
              command=self.validate_and_update_profile,
              fg_color=Colors.GREEN_BUTTON,
              hover_color=Colors.GREEN_BUTTON_HOVER,
              width=200
          )
          self.confirm_button.place(x=250, y=y_position + 30, anchor="center")

          # Position the window relative to the parent
          self.transient(parent)
          self.grab_set()  # Restrict interactions with the parent until this window is closed

```

```python
def validate_and_update_profile(self):
    """
    Validates user input and updates the profile information in the database.
    """
    # Extract updated data from input fields
    updated_data = {label: entry.get() for label, entry in self.inputs.items()}

    # Basic validation
    if not all(updated_data.values()):
        messagebox.showerror("Error", "All fields are required!")
        return
    if "@" not in updated_data["Email"]:
        messagebox.showerror("Error", "Invalid email address!")
        return
    if not updated_data["Phone"].isdigit() or len(updated_data["Phone"]) != 10:
        messagebox.showerror("Error", "Invalid phone number!")
        return

    try:
        with self._conn_.cursor() as cursor:
            # Update query
            query = """UPDATE passengers SET full_name = %s, email = %s, phone_number = %s, address = %s, gender = %s, age = %s WHERE id = %s"""
            cursor.execute(
                query,
                (
                    updated_data["Name"],
                    updated_data["Email"],
                    updated_data["Phone"],
                    updated_data["Home Address"],
                    updated_data["Gender"],
                    updated_data["Age"],
                    self.id
                )
            )
            self._conn_.commit()  # Save changes
            messagebox.showinfo("Success", "Profile updated successfully!")
    except Exception as e:
                messagebox.showerror("Database Error", f"An error occurred: {str(e)}")
                return

            # Close the window and refresh the parent frame
            self.destroy()
            self.parent.user_profile()
```

5. Admin

5.1 dashboard.py

dashboard.py M

Admin > dashboard.py > AdminDashboard > __init__

```python
import customtkinter as ctk
from PIL import Image,ImageTk
import os
from Admin.manage_booking import BookingManagementFrame
from Admin.manage_driver import DriverManagementFrame
from Admin.driver_request import DriverRequestFrame
from Admin.history import HistoryRidesFrame
from fonts.colors import Colors

#It represents the dashboard view for the admin user.
class AdminDashboard(ctk.CTkFrame):
    def __init__(self, parent, controller, shared_data):
        super().__init__(parent)
        self.shared_data = shared_data
        self.controller = controller

        # Get initial passenger ID
        self.admin_id = self.shared_data.get("admin_id", None)
        print(f"Admin ID at the  Dashboard (initial): {self.admin_id}")  # Debugging

        # Sidebar setup
        self.sidebar = ctk.CTkFrame(self, width=200, height=800, corner_radius=0)
        self.sidebar.place(x=0, y=0)

```

5.2 driver_register.py

```
        # Load and resize the image
        sahara_path = os.path.join(os.path.dirname(__file__), "..", "assets", "sahara3.png")
        sahara_image = Image.open(sahara_path)
        resized_sahara_image = sahara_image.resize((250, 85))  # Resize to 100x100 pixels (adjust as needed)

        # Convert the resized image to PhotoImage
        self.sahara_photo = ImageTk.PhotoImage(resized_sahara_image)

        # Create and place the label
        self.sahara_label = ctk.CTkLabel(self.sidebar, image=self.sahara_photo, text="")
        self.sahara_label.place(x=110, y=50, anchor="center")

        # Navigation buttons
        nav_items = ["Booking Management", "Driver Management", "Rides History", "Driver's Request"]
        self.nav_buttons = []

        # self.frames = {}
        for i, item in enumerate(nav_items):
            button=ctk.CTkButton(
                self.sidebar,
                text=item,
                width=160,
                fg_color=Colors.GREEN_BUTTON,
                hover_color=Colors.GREEN_BUTTON_HOVER,
                command=lambda x=item: self.switch_frame(x)
            )
            button.place(x=20, y=100 + i * 50)
            self.nav_buttons.append(button)

        # Logout button
        ctk.CTkButton(self.sidebar,text="Log Out",width=160,fg_color=Colors.RED,hover_color=Colors.HOVER_RED,
                      command=lambda: controller.show_frame("Login")).place(x=20, y=700)

        # Main content area
        self.main_frame = ctk.CTkFrame(self, width=960, height=760)
        self.main_frame.place(x=220, y=20)

        # Mapping frame names to their respective classes
        self.frames= {
            "Booking Management": BookingManagementFrame(self.main_frame,self.admin_id),
            "Driver Management": DriverManagementFrame(self.main_frame),
            "Rides History": HistoryRidesFrame(self.main_frame),
            "Driver's Request": DriverRequestFrame(self.main_frame)
        }

        # Load and display the default frame
        self.current_frame = self.frames["Booking Management"]
        self.current_frame.place(x=0, y=0, relwidth=1, relheight=1)

    def update_id(self, admin_id=None):
        """Update the passenger_id and refresh the user profile."""
        self.admin_id = admin_id

        # Reinitialize frames with the updated passenger_id
        self.frames= {
            "Booking Management": BookingManagementFrame(self.main_frame,self.admin_id),
            "Driver Management": DriverManagementFrame(self.main_frame),
            "Rides History": HistoryRidesFrame(self.main_frame),
            "Driver's Request": DriverRequestFrame(self.main_frame)
        }
        # Update the currently displayed frame
```

```
        if self.current_frame:
            self.current_frame.place_forget()

        # Default frame to show after update
        self.current_frame = self.frames["Booking Management"]
        self.current_frame.place(x=0, y=0, relwidth=1, relheight=1)

    def switch_frame(self, frame_name):
        """Switches to the selected frame and updates its content dynamically."""
        # Destroy the current frame if it exists
        self.current_frame.place_forget()

        self.current_frame = self.frames[frame_name]
        # Refresh the frame if it has a refresh method
        if hasattr(self.current_frame, "load_bookings"):
            self.current_frame.load_bookings(self.admin_id)
        if hasattr(self.current_frame, "load_history"):
            self.current_frame.load_history()
        if hasattr(self.current_frame, "load_driver_requests"):
            self.current_frame.load_driver_requests(self.admin_id)
        if hasattr(self.current_frame, "load_drivers"):
            self.current_frame.load_drivers()

        self.current_frame.place(x=0, y=0, relwidth=1, relheight=1)
```

5.3 history.py

history.py M

Admin > history.py > ...

```
import customtkinter as ctk
from tkinter import messagebox
import mysql.connector
from sql_connection import DatabaseConnection
from fonts.colors import Colors

#  class of history rides whuch shows all the booking histories
class HistoryRidesFrame(ctk.CTkFrame):
    def __init__(self, parent):
        super().__init__(parent, width=parent.winfo_width(), height=parent.winfo_height())
        self.pack_propagate(False)
        self._conn_=DatabaseConnection.connection()

        # Header
        self.header = ctk.CTkLabel(self, text=" Rides History", font=ctk.CTkFont(size=24, weight="bold"))
        self.header.place(x=20, y=20)

        # Create scrollable frame for bookings
        self.scrollable_frame = ctk.CTkScrollableFrame(self, width=850, height=600)
        self.scrollable_frame.place(x=20, y=70)

        self.load_history()
    # loading all history
    def load_history(self):
        try:
            # Clear existing bookings
            for widget in self.scrollable_frame.winfo_children():...

```

```
            # connection and sql queries
            self._conn_=DatabaseConnection.connection()
            cursor = self._conn_.cursor()
            query="""SELECT
                        b.ride_date,
                        b.ride_time,
                        b.id,
                        p.full_name AS passenger_name,
                        d.full_name AS driver_name,
                        b.pickup,
                        b.dropoff,
                        b.vehicle_type,
                        b.fare,
                        b.ride_status
                    FROM
                        bookings b
                    LEFT JOIN
                        drivers d
                        ON b.driver_id = d.id -- Correct join condition
                    LEFT JOIN
                        passengers p
                        ON b.passenger_id = p.id -- Correct join condition;
            """
            cursor.execute(query)
            bookings = cursor.fetchall()

            # if data is empty on database
            if not bookings:
                self.no_history_rides()

            # else showing booking details
            else:
                for booking in bookings:
                    self.create_booking_card(booking)
        # handling mysql error
        except mysql.connector.Error as e:
            messagebox.showerror("Database Error", f"An error occurred: {e}")
    # at last
# function of  no any history
def no_history_rides(self):
    card = ctk.CTkFrame(self.scrollable_frame, height=150)
    card.pack(pady=10, padx=10, fill="x")
    ctk.CTkLabel(card, text=f"No History of rides . Check back later!").pack(side="left", padx=10)

def create_booking_card(self, booking):
    card = ctk.CTkFrame(self.scrollable_frame,width=int(self.scrollable_frame.winfo_width() * 0.95), height=250)
    card.pack(pady=10, padx=10, fill="x")

    # Date and status (placed at the top of the card)
    header_frame = ctk.CTkFrame(card, height=30)
    header_frame.pack(pady=5, fill="x")

    # Booking details
    ctk.CTkLabel(header_frame,text=f"Date: {booking[0]}   {booking[1]}",font=ctk.CTkFont(size=14)).pack(side="left", padx=10)
    ctk.CTkLabel(card, text=f"Booking ID: {booking[2]}", font=ctk.CTkFont(size=14, weight="bold")).pack(pady=(5, 0), anchor="w", padx=10)
    ctk.CTkLabel(card, text=f"Passenger: {booking[3]}", font=ctk.CTkFont(size=14)).pack(pady=(5, 0), anchor="w", padx=10)
    ctk.CTkLabel(card, text=f"Driver: {booking[4]}", font=ctk.CTkFont(size=14)).pack(pady=(5, 0), anchor="w", padx=10)
    ctk.CTkLabel(card, text=f"From: {booking[5]} → To: {booking[6]}", font=ctk.CTkFont(size=14)).pack(pady=(5, 0), anchor="w", padx=10)


    status_colors = Colors.RED if booking[9] == "Cancelled"  else Colors.GREEN if booking[9] == "Completed" else Colors.BLUE
    # Assign driver button
    pending_button = ctk.CTkButton(header_frame,text=booking[9],width=120,fg_color=status_colors,hover_color=status_colors)
    pending_button.pack(side="right")

```

5.4 manage_booking.py

manage_booking.py

Admin > manage_booking.py > AssignDriverWindow

```python
import customtkinter as ctk
from tkinter import messagebox
import mysql.connector
from sql_connection import DatabaseConnection
from fonts.colors import Colors

class BookingManagementFrame(ctk.CTkFrame):# Define the 'BookingFrame' class, which is a subclass of 'CTkFrame'
    def __init__(self, parent,admin_id=None):
        #The constructor (__init__) method is called when an instance of the class is created.
        #It initializes the frame
        self.admin_id = admin_id
        super().__init__(parent, width=parent.winfo_width(), height=parent.winfo_height())
        self.pack_propagate(False)#This disables the "propagation" of the frame's size to its children widgets when using the pack()
        self._conn_ = None
        self._conn_=DatabaseConnection.connection()

        #This calls the constructor of the parent class (CTkFrame,

        # Header
        self.header = ctk.CTkLabel(self, text="Booking Management", font=ctk.CTkFont(size=24, weight="bold"))
        self.header.place(x=20, y=20)

        # Create scrollable frame for bookings
        self.scrollable_frame = ctk.CTkScrollableFrame(self, width=850, height=600)
        self.scrollable_frame.place(x=20, y=70)

        # call load bookings method
        self.load_bookings(self.admin_id)

    def load_bookings(self,admin_id):
        try:
            # Clear existing bookings
            for widget in self.scrollable_frame.winfo_children():
                widget.destroy()
                # connection and sql queries

            self._conn_=DatabaseConnection.connection()
            cursor = self._conn_.cursor()
            query = """SELECT
                    b.id,
                    p.full_name AS passenger_name,
                    b.pickup,
                    b.dropoff,
                    b.ride_date,
                    b.ride_time,
                    b.fare
                FROM
                    bookings b
                LEFT JOIN
                    passengers p
                ON
                    b.passenger_id = p.id -- Correct join condition
                WHERE
                    b. ride_status='Pending';
                """
            cursor.execute(query)
            bookings = cursor.fetchall()

            # if data is empty on database
            if not bookings:
                self.no_pending_rides()

```

```
            # else showing booking details
            else:
                for booking in bookings:
                    self.create_booking_card(booking,admin_id)

        except mysql.connector.Error as e:
            messagebox.showerror("Database Error", f"An error occurred: {e}")


    def no_pending_rides(self):
        # self.scrollable_frame.pack(fill="both", expand=True)  # Make parent frame fill the entire window.
        card = ctk.CTkFrame(self.scrollable_frame, height=150)
        card.pack(pady=10, padx=10, fill="x")

        ctk.CTkLabel(card, text=f"No bookings found !").pack(side="left", padx=10)

    def create_booking_card(self, booking,admin):
        card = ctk.CTkFrame(self.scrollable_frame, height=150)
        card.pack(pady=10, padx=10, fill="x")

        # Booking details
        ctk.CTkLabel(card, text=f"Booking ID : {booking[0]}", font=ctk.CTkFont(size=14, weight="bold")).place(x=20, y=20)
        ctk.CTkLabel(card, text=f"Passenger : {booking[1]}", font=ctk.CTkFont(size=14)).place(x=20, y=50)
        ctk.CTkLabel(card, text=f"From : {booking[2]} → To : {booking[3]}", font=ctk.CTkFont(size=14)).place(x=20, y=80)
        ctk.CTkLabel(card, text=f"Date : {booking[4]}  Time: {booking[5]}", font=ctk.CTkFont(size=14)).place(x=20, y=110)

        # Assign driver button
        assign_button = ctk.CTkButton(card,text="Assign Driver",width=120,fg_color=Colors.GREEN_BUTTON,
                                      hover_color=Colors.GREEN_BUTTON_HOVER,
                                      command=lambda: self.assign_driver(booking[0],admin))
        assign_button.place(x=600, y=50)

    def assign_driver(self,booking_id,admin_id):
        self.assign_window = AssignDriverWindow(self,admin_id, booking_id,booking_frame=self)


class AssignDriverWindow(ctk.CTkToplevel):
    def __init__(self, parent,admin_id, booking_id,booking_frame):
        super().__init__(parent)
        self._conn_ = None
        self._conn_=DatabaseConnection.connection()

        self.title("Assign Driver")
        self.geometry("400x400")
        self.booking_id = booking_id
        self.booking_frame = booking_frame # Store the reference to BookingManagementFrame

        # Position the window in front of the parent
        self.transient(parent)
        self.grab_set()  # Disable interactions with the parent until this window is closed


        ctk.CTkLabel(self, text="Available Drivers", font=ctk.CTkFont(size=18, weight="bold")).pack(pady=20)

        # Create scrollable frame for available drivers
        self.scrollable_frame = ctk.CTkScrollableFrame(self, width=350, height=400)
        self.scrollable_frame.pack(pady=10)

        self.load_available_drivers(admin_id)

    def load_available_drivers(self,admin):
        try:
            cursor = self._conn_.cursor()
            query = "SELECT * FROM drivers WHERE driver_status = 'Online'"
            cursor.execute(query)
            drivers = cursor.fetchall()
            print(f"Driver in drivers")
            if not drivers:
                self.no_driver_available()
            else:
                for driver in drivers:
                    self.create_driver_option(driver,admin)
```

```
        except mysql.connector.Error as e:
            messagebox.showerror("Database Error", f"An error occurred: {e}")


    # function to create frames of drivers

    def create_driver_option(self, driver,admin):
        self.scrollable_frame.pack(fill="both", expand=True)  # Make parent frame fill the entire window.

        frame = ctk.CTkFrame(self.scrollable_frame)
        frame.pack(fill="both", expand=True)  # Make the frame fill the parent container.

        ctk.CTkLabel(frame, text=f"{driver[1]} ({driver[0]})", font=ctk.CTkFont(size=14)).pack(side="left", padx=10)

        assign_button = ctk.CTkButton(frame,text="Assign",width=80,fg_color="#2D5A27",hover_color="#1E3D1A",command=lambda: self.assign_driver(driver[0],admin))
        assign_button.pack(side="right", padx=10)

    # function if no any driver is availanle
    def no_driver_available(self):
        self.scrollable_frame.pack(fill="both", expand=True)  # Make parent frame fill the entire window.
        frame = ctk.CTkFrame(self.scrollable_frame)
        frame.pack(fill="both", expand=True)  # Make the frame fill the parent container.
        ctk.CTkLabel(frame, text=f"No Drivers Available").pack(side="left", padx=10)

    # function to assign driver to the booking
    def assign_driver(self, driver_id,admin_id):
        try:
            cursor = self._conn_.cursor()
            print(self.booking_id)

            # Update booking with assigned driver
            update_booking_status = "UPDATE bookings SET driver_id = %s,admin_id=%s, ride_status = 'Assigned' WHERE id = %s"
            cursor.execute(update_booking_status, [driver_id,admin_id, self.booking_id])
            self._conn_.commit()
                    messagebox.showinfo("Success", "Driver assigned successfully!")

                    # Refresh the bookings list in the parent frame
                    if self.booking_frame:
                        print('hello')
                        self.booking_frame.load_bookings(admin_id)
                        self.destroy()

                except mysql.connector.Error as e:
                    messagebox.showerror("Database Error", f"An error occurred: {e}")

```

5.5 manage_driver.py

```
manage_driver.py
Admin > manage_driver.py > DriverManagementFrame > create_driver_card
import customtkinter as ctk
from tkinter import messagebox
import mysql.connector
from sql_connection import DatabaseConnection
from fonts.colors import Colors

class DriverManagementFrame(ctk.CTkFrame):
    def __init__(self, parent):
        super().__init__(parent, width=parent.winfo_width(), height=parent.winfo_height())
        self.pack_propagate(False)
        self._conn_=DatabaseConnection.connection()


        # Header
        self.header = ctk.CTkLabel(self, text="Driver Management", font=ctk.CTkFont(size=24, weight="bold"))
        self.header.place(x=20, y=20)

        # Create scrollable frame for drivers
        self.scrollable_frame = ctk.CTkScrollableFrame(self, width=850, height=600)
        self.scrollable_frame.place(x=20, y=70)

        self.load_drivers()

    def load_drivers(self):

```

```
        try:
            # Clear existing widgets
            for widget in self.scrollable_frame.winfo_children():
                widget.destroy()
            self._conn_=DatabaseConnection.connection()
            cursor = self._conn_.cursor()
            query = "SELECT * FROM drivers"
            cursor.execute(query)
            drivers=[]
            drivers = cursor.fetchall()
            cursor.close()


        except mysql.connector.Error as e:
            messagebox.showerror("Database Error", f"An error occurred: {e}")
        if drivers:
            for driver in drivers:
                self.create_driver_card(driver)
        else:
            self.no_drivers_found()

    def no_drivers_found(self):
        card = ctk.CTkFrame(self.scrollable_frame, height=150)
        card.pack(pady=10, padx=10, fill="x")
        ctk.CTkLabel(card, text="No drivers found !").pack(side="left", padx=10)



    def create_driver_card(self, driver):
        card = ctk.CTkFrame(self.scrollable_frame, height=120)
        card.pack(pady=10, padx=10, fill="x")

        # Driver details
        ctk.CTkLabel(card, text=f"Driver ID: {driver[0]}", font=ctk.CTkFont(size=14, weight="bold")).place(x=20, y=20)
        ctk.CTkLabel(card, text=f"Name: {driver[1]}", font=ctk.CTkFont(size=14)).place(x=20, y=50)
        ctk.CTkLabel(card, text=f"Status: {driver[8]}", font=ctk.CTkFont(size=14)).place(x=20, y=80)

```

```
            # Delete button
            delete_button = ctk.CTkButton(card,text="Delete Driver",width=120,
                                          fg_color=Colors.RED,hover_color=Colors.HOVER_RED,
                                          command=lambda: self.delete_driver(driver[0]))
            delete_button.place(x=600, y=40)

        def delete_driver(self, driver_id):
            if messagebox.askyesno("Confirm Delete", "Are you sure you want to delete this driver?"):
                try:
                    cursor = self._conn_.cursor()
                    query = "DELETE FROM drivers WHERE id = %s"
                    cursor.execute(query, [driver_id])
                    self._conn_.commit()
                    messagebox.showinfo("Success", "Driver deleted successfully!")
                    self.load_drivers()  # Refresh the list
                except mysql.connector.Error as e:
                    messagebox.showerror("Database Error", f"An error occurred: {e}")

```

6. Driver

6.1 registration.py

registration.py M

Driver > registration.py > DriverRegistration > __init__

```python
import customtkinter as ctk
from tkinter import messagebox
from fonts.colors import Colors
import re
import mysql.connector
from sql_connection import DatabaseConnection
import bcrypt
# from driver_request import driver_requests

class DriverRegistration(ctk.CTkFrame):
    def __init__(self, parent, controller,shared_data):
        super().__init__(parent)
        self.controller = controller
        self.shared_data = shared_data
        self.configure(fg_color=Colors.GREEN)  # Green background
        self._conn_ = DatabaseConnection.connection()

        # Back Button
        self.back_button = ctk.CTkButton(self, text="Back", width=80, height=32, corner_radius=16, fg_color="black",
                                         hover_color="#333333",command=lambda: self.controller.show_frame("RegisterSignup"))
        self.back_button.place(x=20, y=20)

        # Title
        self.title_label = ctk.CTkLabel(self, text="Driver Registration", font=("Montserrat Bold", 24, "bold"), text_color="black")
        self.title_label.place(relx=0.45, rely=0.1, anchor="center")


        """Create individual input fields with labels."""
        # Full Name
        full_name_label = ctk.CTkLabel(self, text="Full Name", font=("Montserrat Bold", 14), text_color="black")
        full_name_label.place(relx=0.15, rely=0.2)
        self.name_entry = ctk.CTkEntry(self, width=300, height=40, placeholder_text="Enter your full name")
        self.name_entry.place(relx=0.15, rely=0.25)
        # Contact Number
        contact_label = ctk.CTkLabel(self, text="Contact Number", font=("Montserrat Bold", 14), text_color="black")
        contact_label.place(relx=0.15, rely=0.35)
        self.contact_entry = ctk.CTkEntry(self, width=300, height=40, placeholder_text="Enter your contact number")
        self.contact_entry.place(relx=0.15, rely=0.4)
        # Vehicle Number
        vehicle_label = ctk.CTkLabel(self, text="Vehicle Number", font=("Montserrat Bold", 14), text_color="black")
        vehicle_label.place(relx=0.15, rely=0.5)
        self.vehicle_entry = ctk.CTkEntry(self, width=300, height=40, placeholder_text="Enter your vehicle details")
        self.vehicle_entry.place(relx=0.15, rely=0.55)
        # Licence Number
        licence_label = ctk.CTkLabel(self, text="Licence Number", font=("Montserrat Bold", 14), text_color="black")
        licence_label.place(relx=0.15, rely=0.65)
        self.licence_entry = ctk.CTkEntry(self, width=300, height=40, placeholder_text="Enter your licence number")
        self.licence_entry.place(relx=0.15, rely=0.7)
        # Address
        address_label = ctk.CTkLabel(self, text="Address", font=("Montserrat Bold", 14), text_color="black")
        address_label.place(relx=0.45, rely=0.2)
        self.address_entry = ctk.CTkEntry(self, width=300, height=40, placeholder_text="Enter your permanent address")
        self.address_entry.place(relx=0.45, rely=0.25)
        # Email
        email_label = ctk.CTkLabel(self, text="Email", font=("Montserrat Bold", 14), text_color="black")
        email_label.place(relx=0.45, rely=0.35)
        self.email_entry = ctk.CTkEntry(self, width=300, height=40, placeholder_text="Enter your email address")
        self.email_entry.place(relx=0.45, rely=0.4)
        # Set Password
        set_password_label = ctk.CTkLabel(self, text="Set Password", font=("Montserrat Bold", 14), text_color="black")
        set_password_label.place(relx=0.45, rely=0.5)
        self.set_password_entry = ctk.CTkEntry(self, width=300, height=40, placeholder_text="Set password", show="*")
        self.set_password_entry.place(relx=0.45, rely=0.55)
```

```
        # Confirm Password
        confirm_password_label = ctk.CTkLabel(self, text="Confirm Password", font=("Montserrat Bold", 14), text_color="black")
        confirm_password_label.place(relx=0.45, rely=0.65)
        self.confirm_password_entry = ctk.CTkEntry(self, width=300, height=40, placeholder_text="Confirm password", show="*")
        self.confirm_password_entry.place(relx=0.45, rely=0.7)
        # Gender Entry
        self.gender_label = ctk.CTkLabel(self,text=" Select your gender",font=("Montserrat Bold",14),text_color="black")
        self.gender_label.place(relx=0.7,rely=0.3)

        # Variable to store the selected gender
        self.gender_var = ctk.StringVar(value="Male")

        self.male_radio = ctk.CTkRadioButton(self,text="Male",value="Male",variable=self.gender_var)
        self.male_radio.place(relx=0.7,rely=0.4)

        self.female_radio = ctk.CTkRadioButton(self,text="Female",value="Female",variable=self.gender_var)
        self.female_radio.place(relx=0.7,rely=0.5)

        self.others_radio = ctk.CTkRadioButton(self,text="Others",value="Others",variable=self.gender_var)
        self.others_radio.place(relx=0.7,rely=0.6)


        # Register Button
        self.register_button = ctk.CTkButton(self, text="Register", font=("Montserrat Bold", 16), width=150, height=40,corner_radius=5,
                                             fg_color="black", hover_color="#006400",command=self.register_driver)
        self.register_button.place(relx=0.5, rely=0.85, anchor="center")

    def register_driver(self):
        print("Register driver method called!")

        """Register the driver."""
        if not self.validate_driver():
            return
        print("Register driver method called after validaion!")
        # Collect data

        name= self.name_entry.get()
        number= self.contact_entry.get()
        email= self.email_entry.get()
        password= self.confirm_password_entry.get()
        hashed_password = bcrypt.hashpw(password.encode('utf-8'), bcrypt.gensalt()).decode('utf-8')# Convert password to bytes and h

        address= self.address_entry.get()
        license_number= self.licence_entry.get()
        vehicle_number= self.vehicle_entry.get()
        gender= self.gender_var.get()

        # Add to database
        try:
            with self._conn_.cursor() as cursor:
                query="INSERT INTO driver_request (full_name ,phone_number ,email ,user_password ,address ,license_number ,vehicle_n
                cursor.execute(query,(name,number,email,hashed_password,address,license_number,vehicle_number,gender,"Requested"))
                self._conn_.commit()
                # Show success message
                messagebox.showinfo("Success", "Driver registration submitted for approval!")

                # Reset the form (assuming you have a `reset_form` method)
                self.reset_form()

        except mysql.connector.Error as e:
            # Handle database connection or execution errors
            messagebox.showerror("Database Error", f"An error occurred: {e}")

        except Exception as ex:
            # Handle any other unexpected errors
            messagebox.showerror("Error", f"An unexpected error occurred: {ex}")

```

```
131         finally:
132             self.reset_form()
133             self.controller.show_frame('Login')
134 
135     def reset_form(self):
136         """Reset all input fields and attributes."""
137         self.name_entry.delete(0, "end")
138         self.contact_entry.delete(0, "end")
139         self.vehicle_entry.delete(0, "end")
140         self.licence_entry.delete(0, "end")
141         self.address_entry.delete(0, "end")
142         self.email_entry.delete(0, "end")
143         self.set_password_entry.delete(0, "end")
144         self.confirm_password_entry.delete(0, "end")
145         self.gender_var.set("")
146 
147     def validate_driver(self):
148         """Validate driver input fields."""
149         name = self.name_entry.get()
150         contact = self.contact_entry.get()
151         email = self.email_entry.get()
152         password = self.set_password_entry.get()
153         confirm_password = self.confirm_password_entry.get()
154         gender=self.gender_var.get()
155 
156         if not name or not contact or not email or not password or not gender :
157             messagebox.showerror("Error", "All fields are required!")
158             return False
159 
160         if password != confirm_password:
161             messagebox.showerror("Error", "Passwords do not match!")
162             return False
163 
164         if len(password) < 6 or len(password) > 18:
165             messagebox.showerror("Error", "Password must be of length between 6 and 18!")
166             return False
167 
168         if not re.match(r"^[0-9]{10}$", contact):
169             messagebox.showerror("Error", "Invalid contact number!")
170             return False
171 
172         if not re.match(r"[^@]+@[^@]+\.[^@]+", email):
173             messagebox.showerror("Error", "Invalid email format!")
174             return False
175         return True
176 
```

6.2 dashboard.py

dashboard.py M

Driver > dashboard.py > DriverDashboard > __init__

```python
import customtkinter as ctk
from fonts.colors import Colors
from PIL import Image,ImageTk
import os
from Driver.profile import ProfileFrame
from Driver.history import RideHistoryFrame
from Driver.manage_ride import ManageRidesFrame


class DriverDashboard(ctk.CTkFrame):
    def __init__(self, parent, controller, shared_data):
        super().__init__(parent)
        self.shared_data = shared_data
        self.controller = controller

        # Get initial Driver ID
        self.driver_id = self.shared_data.get("driver_id",None)

        print(f'Driver shared data at the top (initial): {self.shared_data}')  # Debugging statement
        print(f'Driver ID at the top (initial): {self.driver_id}')  # Debugging statement

        # Create sidebar
        self.sidebar = ctk.CTkFrame(self, width=200, height=800, corner_radius=0)
        self.sidebar.place(x=0, y=0)

        # Load and resize the image
        sahara_path = os.path.join(os.path.dirname(__file__), "..", "assets", "sahara3.png")
        sahara_image = Image.open(sahara_path)
        resized_sahara_image = sahara_image.resize((250, 85))  # Resize to 100x100 pixels (adjust as needed)

        # Convert the resized image to PhotoImage
        self.sahara_photo = ImageTk.PhotoImage(resized_sahara_image)

        # Create and place the label
        self.sahara_label = ctk.CTkLabel(self.sidebar, image=self.sahara_photo, text="")
        self.sahara_label.place(x=110, y=50, anchor="center")

        # Navigation buttons
        nav_items = ["Manage Rides", "Ride History", "Profile"]
        self.nav_buttons = []

        for i, item in enumerate(nav_items):
            button = ctk.CTkButton(self.sidebar,text=item,width=160,fg_color=Colors.GREEN_BUTTON,hover_color=Colors.GREEN_BUTTON_HOVER,
                                   command=lambda x=item: self.switch_frame(x))
            button.place(x=20, y=100 + i*50)
            self.nav_buttons.append(button)

        # Logout button
        logout = ctk.CTkButton(self.sidebar,text='Log Out',width=160,fg_color=Colors.RED,hover_color=Colors.HOVER_RED,
                               command=lambda: controller.show_frame('Login'))
        logout.place(x=20, y=700)

        # Main content area
        self.main_frame = ctk.CTkFrame(self, width=960, height=760)
        self.main_frame.place(x=220, y=20)

        # Initialize frames
        self.frames = {
            "Manage Rides": ManageRidesFrame(self.main_frame,self.driver_id),
            "Ride History": RideHistoryFrame(self.main_frame,self.driver_id),
            "Profile": ProfileFrame(self.main_frame, self.driver_id)
        }
```

```
63
64          # Show default frame
65          self.current_frame = self.frames["Manage Rides"]
66          self.current_frame.place(x=0, y=0, relwidth=1, relheight=1)
67      def update_id(self, driver_id=None):
68          """Update the passenger_id and refresh the user profile."""
69          self.driver_id = driver_id
70          # Reinitialize frames with the updated passenger_id
71          self.frames = {
72              "Manage Rides": ManageRidesFrame(self.main_frame,self.driver_id),
73              "Ride History": RideHistoryFrame(self.main_frame,self.driver_id),
74              "Profile": ProfileFrame(self.main_frame, self.driver_id)
75          }
76
77          # Update the currently displayed frame
78          if self.current_frame:
79              self.current_frame.place_forget()
80
81          # Default frame to show after update
82          self.current_frame = self.frames["Manage Rides"]
83          self.current_frame.place(x=0, y=0, relwidth=1, relheight=1)
84
85      def switch_frame(self, frame_name):
86          # Hide current frame
87          self.current_frame.place_forget()
88
89          # Show selected frame
90          self.current_frame = self.frames[frame_name]
91          self.current_frame.place(x=0, y=0, relwidth=1, relheight=1)
92
```

6.3 history

history.py M

Driver > history.py > RideHistoryFrame > __init__

```
1   import customtkinter as ctk
2   from sql_connection import DatabaseConnection
3   from fonts.colors import Colors
4
5   class RideHistoryFrame(ctk.CTkFrame):
6       def __init__(self, parent,driver_id=None):
7           super().__init__(parent, width=parent.winfo_width(), height=parent.winfo_height())
8           self.driver_id=driver_id
9           self._conn_=DatabaseConnection.connection()
10          # Initialize ride details list
11          ride_details = []
12          try:
13              cursor = self._conn_.cursor()
14              query="""SELECT
15                      c.full_name AS customer_name,
16                      b.pickup,
17                      b.dropoff,
18                      b.ride_date,
19                      b.ride_time,
20                      b.vehicle_type,
21                      b.fare,
```

```
                    b.ride_status
                FROM
                    bookings b
                LEFT JOIN
                    passengers c
                ON
                    b.passenger_id = c.id -- Correct join condition
                WHERE
                    b.driver_id = %s;
            """
            cursor.execute(query,[self.driver_id])
            self.dets = cursor.fetchall()


            # Header
            self.header = ctk.CTkLabel(self,text="Ride History",font=ctk.CTkFont(size=24, weight="bold"))
            self.header.pack(pady=(20, 10))

            # Create scrollable frame for ride history
            self.scrollable_frame = ctk.CTkScrollableFrame(self,width=int(self.winfo_width() * 0.95),height=600)
            self.scrollable_frame.pack(padx=20, pady=(0, 20), fill="both", expand=True)

            # Process fetched data into ride_details format
            if not self.dets:
                self.no_ride_history()

            else:
                for ride in self.dets:
                    ride_dict = {
                        "passenger": ride[0],  #  column 2 is the Passenger name
                        "from": ride[1],  #  column 5 is the 'from' location
                        "to": ride[2],  #  column 6 is the 'to' location
                        "date": ride[3],  #  column 7 is the date
                        "time": ride[4],  #  column 8 is the time
                        "vehicle": ride[5],  #  cloumn 9 is vehicle type
                        "fare": f"Rs {ride[6]}",  #  column 10 is the fare
                        "status": ride[7]  #  column 11 is the ride status
                    }
                    ride_details.append(ride_dict)  # Add ride dictionary to the list
        finally:
            print(self.dets)

        # Sample ride history
        for i, ride in enumerate(ride_details):
            self.create_ride_card(ride)

    def no_ride_history(self):
        self.scrollable_frame.pack(fill="both", expand=True)  # Make parent frame fill the entire window.
        frame = ctk.CTkFrame(self.scrollable_frame)
        frame.pack(fill="both", expand=True)  # Make the frame fill the parent container.
        ctk.CTkLabel(frame, text=f"No Ride history").pack(side="left", padx=10)

    def create_ride_card(self, ride):
        # Card container
        card = ctk.CTkFrame(self.scrollable_frame,width=int(self.scrollable_frame.winfo_width() * 0.95),height=160)
        card.pack(pady=10, padx=10, fill="x")

        # Date and status
        header_frame = ctk.CTkFrame(card, height=30)
        header_frame.pack(pady=5, fill="x")

        ctk.CTkLabel(header_frame,text=f"Date: {ride["date"]}",font=ctk.CTkFont(size=14)).pack(side="left", padx=10)

```

```
        status_color = None
        if ride["status"] == "Completed":
            status_color = Colors.GREEN  # Green for completed
        elif ride["status"] == "Cancelled":
            status_color = Colors.RED  # Red for cancelled
        elif ride["status"] == "Assigned":
            status_color = Colors.BLUE  # Blue for pending

        ctk.CTkLabel(header_frame,text=ride["status"],text_color=status_color,font=ctk.CTkFont(size=14, weight="bold")).pack(side="right", padx=10)

        # Passenger information
        ctk.CTkLabel(card,text=f"Passenger: {ride['passenger']}",font=ctk.CTkFont(size=14)).pack(pady=(5, 0), anchor="w", padx=10)

        # Route information
        ctk.CTkLabel(card,text=f"From: {ride['from']}").pack(pady=(5, 0), anchor="w", padx=10)

        ctk.CTkLabel(card,text=f"To: {ride['to']}").pack(pady=(5, 0), anchor="w", padx=10)

        # Fare information
        ctk.CTkLabel(card,text=f"Fare: {ride['fare']}",font=ctk.CTkFont(size=14, weight="bold")).pack(pady=(5, 10), anchor="w", padx=10)

```

6.4 manage_ride.py

manage_ride.py M

Driver > manage_ride.py > ManageRidesFrame > __init__

```
import customtkinter as ctk
from tkinter import messagebox
from sql_connection import DatabaseConnection

class ManageRidesFrame(ctk.CTkFrame):
    def __init__(self, parent,driver_id=None):
        super().__init__(parent, width=parent.winfo_width(), height=parent.winfo_height())
        self._conn_=DatabaseConnection.connection()
        self.driver_id = driver_id
        print(f"Driver id me {self.driver_id}")

        # Make the frame expand to fill the parent
        self.pack_propagate(False)

        # Status selector
        ctk.CTkLabel(self,text="Status:",font=ctk.CTkFont(size=14,weight="bold")).place(x=20, y=30)
        self.status_var = ctk.StringVar(value="Online")  # Default value set to "Offline"
        status_frame = ctk.CTkFrame(self, width=300, height=40)
        status_frame.place(x=80, y=20)

        # Create radio buttons
        self.offline_radio = ctk.CTkRadioButton(status_frame, text="Offline", variable=self.status_var,
                                                value="Offline", fg_color="#2D5A27", command=self.update_status)
        self.offline_radio.place(x=20, y=10)
        self.online_radio = ctk.CTkRadioButton(status_frame, text="Online", variable=self.status_var,
                                               value="Online", fg_color="#2D5A27", command=self.update_status)
        self.online_radio.place(x=150, y=10)

        ctk.CTkLabel(self,text="Upcoming Rides",font=ctk.CTkFont(size=18, weight="bold")).place(x=20, y=100)

        # Upcoming rides section
        self.create_upcoming_rides_section()
    def create_upcoming_rides_section(self):

        self.upcoming_frame = ctk.CTkScrollableFrame(self, width=850, height=600)
        self.upcoming_frame.place(x=20, y=130)

```

```
        # Sample upcoming rides
        try:
            # Clear existing bookings
            for widget in self.upcoming_frame.winfo_children():
                widget.destroy()

            cursor = self._conn_.cursor()
            query = """SELECT
                    b.id,
                    c.full_name AS customer_name,
                    c.phone_number AS customer_number,
                    b.pickup,
                    b.dropoff,
                    b.ride_date,
                    b.ride_time,
                    b.fare
                FROM
                    bookings b
                LEFT JOIN
                    passengers c
                ON
                    b.passenger_id = c.id -- Correct join condition
                WHERE
                    b.driver_id = %s AND ride_status =%s ;"""

            print(self.driver_id)
            print(f"Driver id me {self.driver_id}")

            cursor.execute(query,(self.driver_id,"Assigned"))
            bookings = cursor.fetchall()
            print(bookings)
            print(f"Driver id me {bookings}")


            if not bookings:
                self.no_pending_rides()

            else:
                for booking in bookings:
                    self.create_booking_card(booking)
    finally:
        print("Upcoming rides")

def no_pending_rides(self):
    card = ctk.CTkFrame(self.upcoming_frame, height=150)
    card.pack(pady=10, padx=10, fill="x")

    ctk.CTkLabel(card, text=f"No Rides are assigned").pack(side="left", padx=10)

def create_booking_card(self, booking):
    card = ctk.CTkFrame(self.upcoming_frame, height=200)
    card.pack(pady=10, padx=10, fill="x")

    # Booking details
    ctk.CTkLabel(card, text=f"Booking ID: {booking[0]}", font=ctk.CTkFont(size=14, weight="bold")).place(x=20, y=10)
    ctk.CTkLabel(card, text=f"Passenger: {booking[1]}", font=ctk.CTkFont(size=14)).place(x=20, y=40)
    ctk.CTkLabel(card, text=f"Phone Number: {booking[2]}", font=ctk.CTkFont(size=14)).place(x=20, y=70)
    ctk.CTkLabel(card, text=f"From: {booking[3]} → To: {booking[4]}", font=ctk.CTkFont(size=14)).place(x=20, y=100)
    ctk.CTkLabel(card, text=f"Date: {booking[5]} : {booking[6]}", font=ctk.CTkFont(size=14)).place(x=20, y=130)
    ctk.CTkLabel(card, text=f"Fare: {booking[7]}", font=ctk.CTkFont(size=14)).place(x=20, y=160)

```

```
        # Assign driver button
        pending_button = ctk.CTkButton(card,text="Assigned",width=120,fg_color="#0a56f0",)
        pending_button.place(x=650, y=50)

    def update_status(self):
        status = self.status_var.get()

        driver_id=self.driver_id
        # Update status in database
        try:
            with  self._conn_.cursor() as cursor:
                query="UPDATE drivers SET driver_status = %s WHERE id = %s "
                cursor.execute(query,(status,driver_id))



        except Exception as e:
            messagebox.showerror("Error", f"An unexpected error occurred: {e}")
        finally:
            print("Booking Data inserted into database")

    def start_ride(self):
        messagebox.showinfo("Ride Started", "Ride has been started successfully!")

    def complete_ride(self):
        messagebox.showinfo("Ride Completed", "Ride has been completed successfully!")

```

6.5 profile.py

profile.py

Driver > profile.py > ProfileFrame > user_profile

```
import customtkinter as ctk
from fonts.colors import Colors
from tkinter import messagebox
from sql_connection import DatabaseConnection

class ProfileFrame(ctk.CTkFrame):
    def __init__(self, parent, driver_id=None):
        #The constructor method is called when an instance of ProfileFrame is created.
        super().__init__(parent, width=parent.winfo_width(), height=parent.winfo_height())
        self.driver_id = driver_id
        self._conn_=DatabaseConnection.connection()
        print(f'Driver Id = {self.driver_id} in Profile frame ')
        self.user_profile()
    def user_profile(self):
        # Clear existing widgets
        for widget in self.winfo_children():
            widget.destroy()
        self._conn_=DatabaseConnection.connection()
        try:
            with  self._conn_.cursor() as cursor:
                query = "SELECT * FROM drivers WHERE id = %s"
                cursor.execute(query, [self.driver_id])
                self.details = cursor.fetchone()
        except Exception as e:
            messagebox.showerror("Error", f"An unexpected error occurred: {e}")

```

```
        # Header
        self.header = ctk.CTkLabel(self,text="Driver Profile",font=ctk.CTkFont(size=24, weight="bold"))
        self.header.place(x=20, y=20)
        # Driver information
        self.user_data = [
            ("Name", self.details[1]),
            ("Phone", self.details[2]),
            ("Email", self.details[3]),
            ("Home Address", self.details[5]),
            ("License Number", self.details[6]),
            ("Vehicle Number", self.details[7]),
            ("Status", self.details[8]),
            ("Gender", self.details[9]),
            ("Rating", "4.8★")
        ]

        # Display user information
        for i, (label, value) in enumerate(self.user_data, start=1):
            ctk.CTkLabel(self,text=label,font=ctk.CTkFont(size=14)).place(x=20, y=20 + i * 50)
            ctk.CTkLabel(self,text=value,font=ctk.CTkFont(size=14, weight="bold")).place(x=200, y=20 + i * 50)

        # Edit profile button
        self.edit_button = ctk.CTkButton(self,text="Edit Profile",width=440,fg_color=Colors.GREEN_BUTTON,
                                         hover_color=Colors.GREEN_BUTTON_HOVER,command=self.edit_profile)
        self.edit_button.place(x=20, y=540)

    def edit_profile(self):# edit profile window
        self.edit_window = EditProfileWindow(self,self.details)

class EditProfileWindow(ctk.CTkToplevel):
    """
    EditProfileWindow is a subclass of CTkToplevel which provides a separate window for editing user profiles.
    """
    def __init__(self, parent, dets=None):
        super().__init__(parent)
        self.parent=parent
        self.dets = dets
        self.id=self.dets[0]
        self._conn_ = DatabaseConnection.connection()
        self.title("Edit Profile")
        self.geometry("500x600")
        self.resizable(False, False)

        # Title Label
        title_label = ctk.CTkLabel(self, text="Edit Profile", font=ctk.CTkFont(size=20, weight="bold"))
        title_label.place(x=250, y=20, anchor="center")

        self.user_data = [
            ("Name", self.dets[1]),
            ("Email", self.dets[3]),
            ("Phone", self.dets[2]),
            ("Home Address", self.dets[5]),
            ("License Number", self.dets[6]),
            ("Vehicle Number", self.dets[7]),
            ("Status", self.dets[8]),
            ("Gender", self.dets[9])
        ]

        # Inputs for user credentials
        self.inputs = {}
        y_position = 80  # Starting position for the labels and entries
        for label, value in self.user_data:
            # Create and place label
```

```
            label_widget = ctk.CTkLabel(self, text=label, font=ctk.CTkFont(size=14))
            label_widget.place(x=50, y=y_position, anchor="w")

            # Create and place entry
            entry_widget = ctk.CTkEntry(self, width=250)
            entry_widget.insert(0, value)
            entry_widget.place(x=160, y=y_position, anchor="w")
            self.inputs[label] = entry_widget

            y_position += 50  # Increment y-position for the next label and entry

        # Confirm Profile Button
        self.confirm_button = ctk.CTkButton(self, text="Confirm Profile", command=self.validate_and_update_profile,
                                            fg_color=Colors.GREEN_BUTTON, hover_color=Colors.GREEN_BUTTON_HOVER, width=200)
        self.confirm_button.place(x=250, y=y_position + 30, anchor="center")

        # Position the window in front of the parent
        self.transient(parent)
        self.grab_set()  # Disable interactions with the parent until this window is closed

    def validate_and_update_profile(self):
        """Validate the user inputs and update the user data."""
        # Extract user inputs
        updated_data = {label: entry.get() for label, entry in self.inputs.items()}

        # Perform basic validation
        if not all(updated_data.values()):
            messagebox.showerror("Error", "All fields are required!")
            return
        if "@" not in updated_data["Email"]:
            messagebox.showerror("Error", "Invalid email address!")
            return
        if not updated_data["Phone"].isdigit() or len(updated_data["Phone"]) != 10:
            messagebox.showerror("Error", "Invalid phone number!")
            return

        try:
            # Update the database
            cursor = self._conn_.cursor()
            # SQL query to update user data
            query = """ UPDATE drivers SET full_name = %s, phone_number = %s, email = %s, address = %s,
                        license_number = %s, vehicle_number=%s, gender = %s WHERE id = %s """


            # Execute the query
            cursor.execute(query, (updated_data["Name"], updated_data["Phone"], updated_data["Email"],
                                   updated_data["Home Address"], updated_data["License Number"],
                                   updated_data["Vehicle Number"], updated_data["Gender"], self.id))
            # Commit the changes
            self._conn_.commit()
            messagebox.showinfo("Success", "Profile updated successfully!")

        except Exception as e:
            messagebox.showerror("Database Error", f"An error occurred: {str(e)}")
            return
        finally:
            cursor.close()



        # Close the Edit Profile window
        self.destroy()
        # After updating the profile, call user_profile on the ProfileFrame instance (the parent)
        self.parent.user_profile()  # Call user_profile on the parent instance (ProfileFrame)

```

7. assets

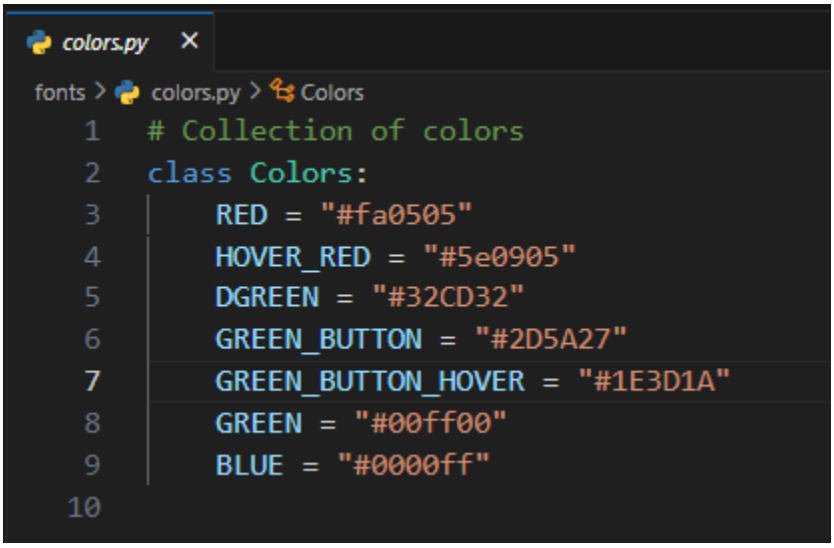
The assets folder contains a collection of images which are used on the system.

sahara_icon.ico

SAHARA

8. fonts
8.1 colors.py

It contains a class named “Colors”



This class contains the collection of colors which are used on multiple files of the system.