Final Project - Uber
FA24: DATA-236 Sec 11 - Distributed Systems

Professor:

Dr. Simon Shim, Ph.D.

Group-1 (Members):

Soumya Challuru Sreenivas

Prithvi Elancherran

Hamsalakshmi Ramachandran

---

1. Introduction

Uber, founded in 2009, revolutionized urban transportation by connecting passengers with drivers through an on-demand mobile application. This project simulates an Uber-like system, incorporating essential features such as real-time ride tracking, dynamic pricing, proximity-based ride assignment, and efficient management of drivers, customers, rides, and billing. Built with a three-tier architecture, the system emphasizes scalability, reliability, and efficient data management using a robust relational database. The user-friendly interface and innovative design choices ensure a seamless experience for both administrators and users. This report details the system's design, development process, and performance evaluation, showcasing a comprehensive solution for modern ride-hailing applications.

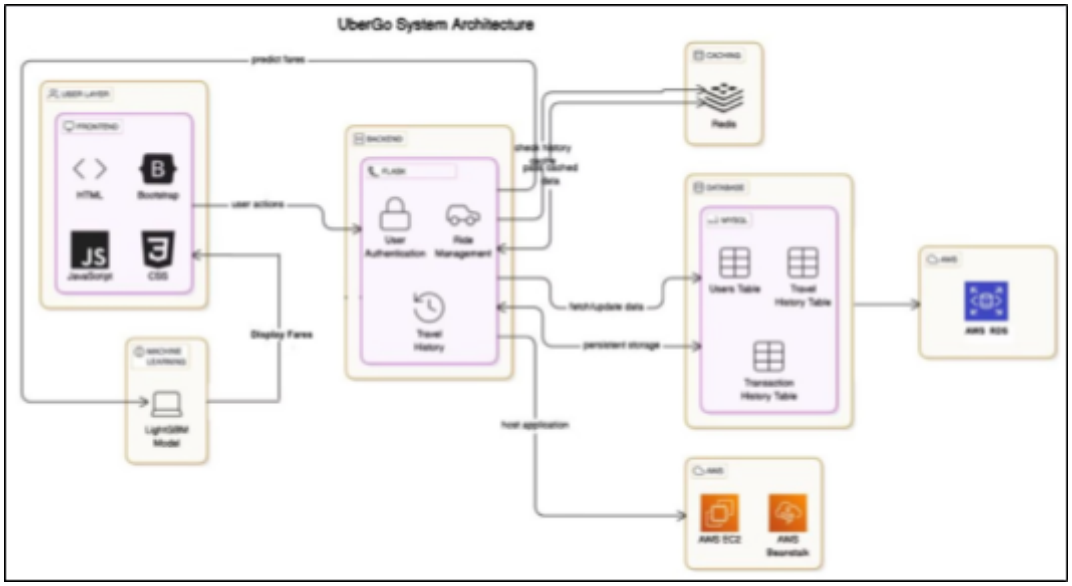
2. Motivation & Objective

The motivation for this project stems from the growing reliance on ride-hailing platforms like Uber and the need to understand the complexities behind such systems. In order to investigate real-world issues like dynamic pricing, effective resource allocation, and scalable system design, we simulated an Uber-type solution. Through this project, theoretical ideas in database administration, distributed systems, and API integration could be applied to create a workable, useful application.

3. Technological Stack

Category	Tools and Framework
Backend Framework	Flask (RESTful APIs), Flask-CORS (Cross-Origin Resource Sharing), Flask-JWT-Extended (JWT-based authentication)
Database Management	MySQL, mysql-connector-python (Flask integration)
Caching	Redis (session caching, travel history caching)
Machine Learning	LightGBM (Dynamic pricing model), Pandas and
Frontend	NumPy
Map	HTML, CSS, JavaScript,Bootstrap (User interface for booking rides, ride assignments, and admin analytics)
Date and Time Handling	Mapbox API (Visualizing routes, pickup, and drop-off points)
Performance Testing	Datetime (Managing timestamps for rides and fare calculations)
Geospatial Calculations	Apache JMeter (Traffic simulation and system load testing with 100 threads)
	Haversine formula (Accurate distance calculations for ride pricing)

4. System Architecture Diagram



The diagram above illustrates our system architecture that includes three layers: User Layer (Frontend), Backend Layer (Application Logic), and Database Layer (Data Storage), with added support for caching and machine learning.

1.1 User Layer (Frontend)

HTML, CSS, JavaScript, Bootstrap is used to build the client-side of the application to provide a user-friendly interface for interacting with the system. We have implemented Mapbox integration that enables user to select pickup and drop-off locations visually on the map. It also displays routes, live driver locations, and estimated time of arrival (ETA). We have kept our interface simple and intuitive. It displays and fetches results from the backend for fare calculations, driver allocation, and ride status.

The interface allows user to:

- Log-in or Sign-in
- Book rides.
- View ride history
- View user profile and update information.

1.2 Backend Layer (Application Logic)

- Flask framework is used to manage backend logic and APIs to manage user authentication (login, registration and sessions), ride management (ride creation, driver allocation, tracking) and travel history (caching for faster retrieval) etc.
- Lightweight Gradient Boosting Machine Regressor (LGBM Regressor) machine learning model is used for dynamic price prediction based on inputs like distance, time of the day, event etc.
- Redis key-value pair in-memory NoSQL database is used for caching frequently accessed data such as login sessions and travel history. This helps in improving performance by reducing the backend query traffic to the database.

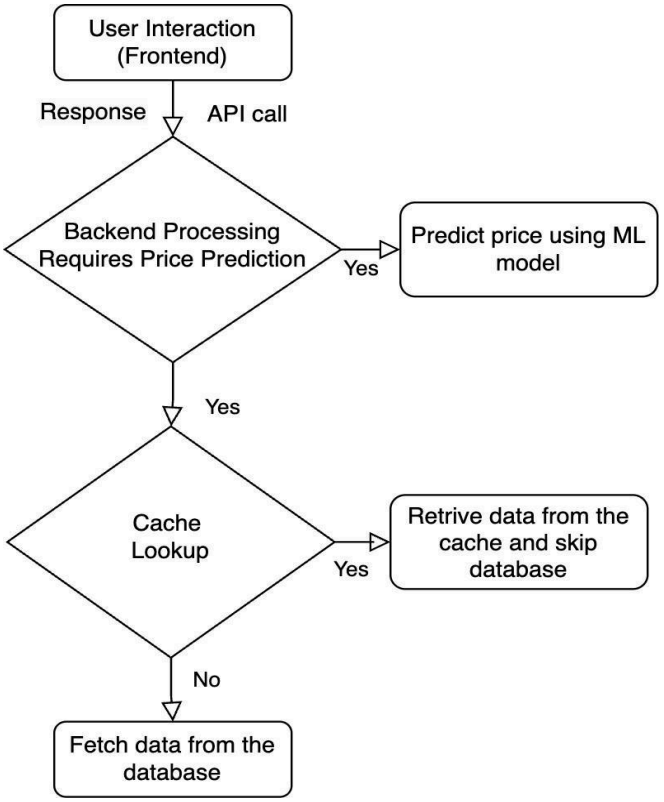
1.3 Database Layer

- MYSQL is used for persistent storage of the data and also AWS RDS is used for scalability, reliability, availability and performance.

1.4 Hosting and Deployment (AWS)

- AWS EC2 and Beanstalk simplifies application deployment, scaling, monitoring and hosting the backend application, enabling scalability and efficient processing of the user requests. (screenshots attached in Section 6)

1.5 General Workflow



User Request:
The user interacts with the frontend, sending a request via an API call.

Backend Decision:
If the request involves fare prediction, the backend uses a Machine Learning model to calculate the fare dynamically.

Cache Lookup:
The backend checks if the required data is in the cache

If Found:
Data is retrieved from the cache, bypassing the database.

If Not Found:
The backend queries the database for the required information.

Database Access:
Data retrieved from the database can be stored in the cache for future requests.

Response to User:
The backend processes and sends the final result (e.g., fare, ride details) to the frontend for user display.

5. Uber Application Design and Implementation

1.6 Frontend

Our frontend serves as the client side part of the application. It is structured to cater to different roles such as Admin, Driver and User modules, each offering distinct functionalities. The client application handles the user interface and interaction written in HTML, CSS Bootstrap, JavaScript.

A. Admin Module :

The purpose of this module is to handle administrative tasks like manage and monitor the system. The administrator is responsible for tasks such as overseeing managing and monitoring tasks and to ensure support and security.

- Below is the list of some of the important files unde Admin module:

dashboard.html	Displays key metrics like total rides, earnings, and active users, often using charts or graphs.
all_customer.html	Lists all registered customers, with options to view, edit, or delete user accounts.
all_driver.html	Similar to all_customer.html, but for managing drivers.
add_balance.html	Add funds to a user or driver account.
debit_balance.html	Deduct funds from a user or driver account.
transaction_history.html	View a log of financial transactions.
fund_history.html	Displays the admin's history of adding and debiting funds.
ride_history.html	View the history of completed rides, including user and driver

	details, date, and fare
support.html	Interface for managing support tickets from users and drivers.
help.html	Provides resources or FAQs for admins.
login.html	Admin login page.
logout.html	Handles secure logout.
change_password.html	Page for updating the admin's password
forget_password.html & forgotpassword.html	Password recovery interfaces.
index.html	serves as a welcome page or a redirect to the dashboard.

B. Driver Module:

The driver module is designed for drivers using the applications. It allows them to manage their rides, view earnings, handle their profiles, and access support resources.

- Below is the list of some of the important files under Driver module:

dashboard.html	Displays driver -specific metrics and notifications.
ride_history.html	Lists rides completed by the driver.
login.html	Handles driver authentication.
register.html	Allows drivers to manage their personal information.
user_profile.html	Provide drivers with access to help and support.
help.html& support.html	Displays safety guidelines for drivers.
safety.html	Shows a record of payments and transactions.
transaction_history.html	Allows drivers to update their password.
change_password.html	Tracks ride requests made by users.
request_history.html	Tracks ride requests made by users.
logout.html	Handles secure logout functionality.

C. User Module:

The User Module in our application focuses on providing a seamless experience for users/customers. These users interact with our platform to book rides, view their history, manage their profiles, and interact with the system through features like payments and support.

- Below is the list of some of the important files under User module:

ride_history.html	Show past rides with details like date, time, pickup/drop-off locations, fare, and payment status.
user_profile.html	Allow users to manage their personal details like name, email, and phone number.
login.html	For existing users to log in.
register.html	For new users to create an account.
forget_password.html	For password recovery.

transaction_history.html	Provides add payment methods and displays all payments made by the user (successful, pending, or failed payment)
support.html	Provide a channel for users to raise queries or issues related to rides, payments, or accounts. Allows submitting ticket for ride-related issues and track them.
index.html	Landing page for the application.

1.7 Backend

Our backend is handled mostly by application.py script**.** It implements a Flask-based backend application with the following core functionalities

1.7.1 Handling Heavy Weight Resources

In any application, effective handling of resource-intensive tasks is crucial for maintaining system performance and ensuring a smooth user experience. Our application's backend, while lightweight, needs to manage several complex and resource-heavy processes. To address these heavy-weight resources efficiently, we implemented the following strategies.

A. API Development:

The application uses Flask to define RESTful APIs for handling client requests to interact with frontend, database and cache. It supports JSON based requests and responses using Flask's request and jsonify. Our backend logic handles and defines more than twenty five APIs.

- Following are the API endpoints:

Endpoint	Method	Purpose
/register	POST	Registers a new user.
/login	POST	Authenticates a user.
/logout	POST	Logs out a user.
/getprofile	POST	Retrieves user profile information.
/change_password	POST	Allows users to change their password.
/checksession	POST	Checks the validity of a user session.
/gettravelhistory	POST	Retrieves travel history for users or drivers.
/inittravel	POST	Initiates a new travel request.
/driver_accept	POST	Marks a travel request as accepted by a driver.
/otp_verify	POST	Verifies an OTP for a travel action.
/when_reached	POST	Marks the trip status as 'reached.'
/afterpayment	POST	Updates the trip status after payment.
/adddriver	POST	Adds a new driver to the system.
/getdriverinfo	POST	Retrieves driver information.
/cehckdriverallo	POST	Checks driver allocation status.
/riding_request	POST	Handles ride requests.
/ridehistoryuser	POST	Retrieves a user's ride history.
/gettransactionuser	POST	Retrieves a user's transaction history.
/get_driver_riding_history	POST	Retrieves a driver's ride history.
/update_profile	POST	Updates the user's profile.
/admin_login	POST	Authenticates admin users.

/ridehistory	POST	Retrieves ride history for admin.
/transaction	POST	Retrieves transaction details for admin.
/incominfo	POST	Provides income statistics.
/usersinfo	POST	Retrieves user information for admin.
/addbalance	POST	Allows admin to credit user accounts.
/debitbalance	POST	Allows admin to debit user accounts.
/getvehicleinfo	POST	Retrieves vehicle details and calculates fares.

B. Dynamic Pricing Algorithm (Lightweight Gradient Boosting Mechanism Regressor):

Dynamic pricing is a strategy where prices for a product or services are adjusted based on various factors like demands, supply, time, or events. We have used LightGBM Regressor (LGBM) is used to implement dynamic pricing for prediction ride fares dynamically.

```
@application.route('/predict_fare', methods=['POST'])
def predict_fare():
try:
data = request.get_json()
fare = dynamic_pricing_logic(data)
    return jsonify({'status': 'Success', 'fare': fare}), 200
  except Exception as e:
return jsonify({'status': 'Failed', 'message': f'Error occurred: {str(e)}'}), 500
```

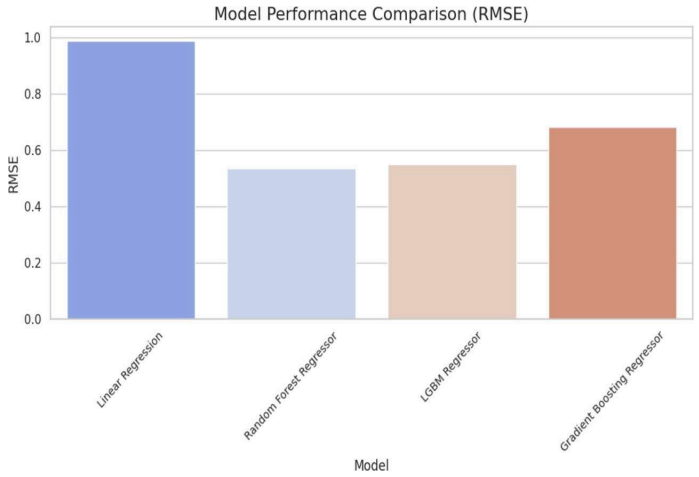
Data collection and model training:

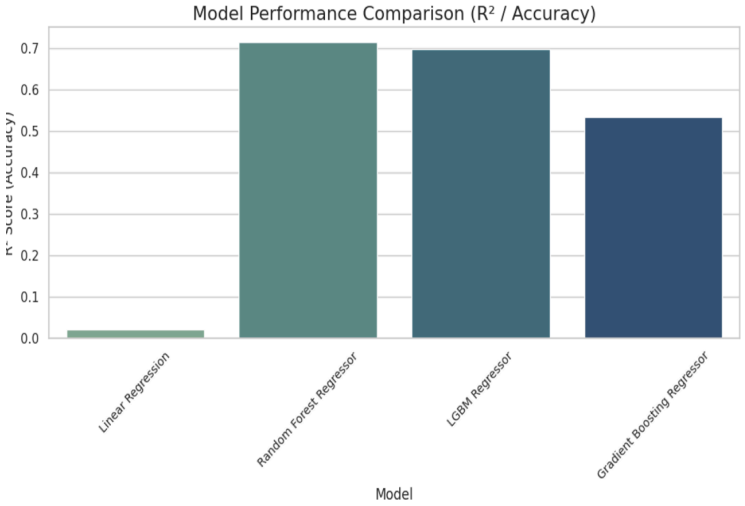
We used twitter dataset from Kaggle (Link) to train and compare different machine learning models for selecting the best one for our application. Following are some of our findings that helped us in finalizing upon Light GBM Regressor as our preferred model for dynamic prising. Although XGBoost Regressor has slightly higher but comparable accuracy compared to LGBM, LGBM has a much faster testing time than all the other models.

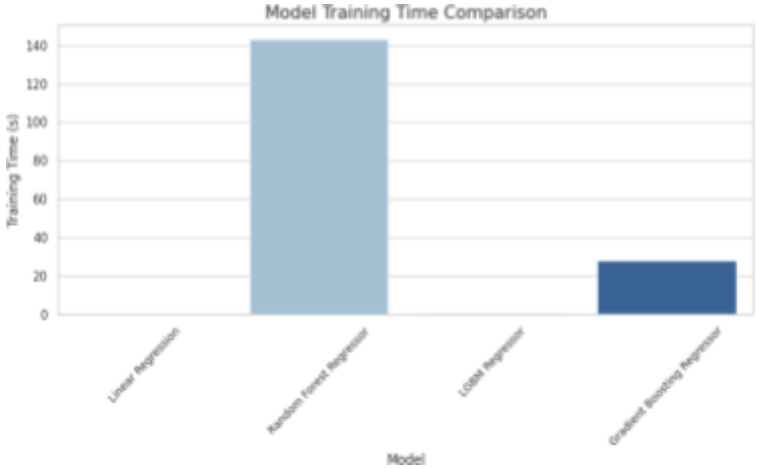
Model Performance Comparison

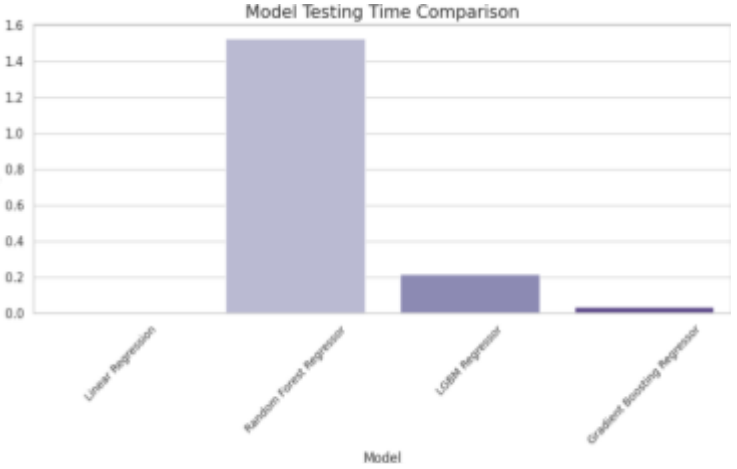
	Model	RMSE	MAE	R^2	Training Time (s)	Testing Time (s)
0	Linear Regression	0.9886	0.7820	0.0207	0.0335	0.0026
1	Random Forest Regressor	0.5341	0.3775	0.7141	143.2995	1.5274
2	LGBM Regressor	0.5488	0.3972	0.6983	0.8244	0.2149
3	Gradient Boosting Regressor	0.6819	0.5184	0.5341	28.1442	0.0370

<ipython-input-9-7038fbf31297>:101: FutureWarning:









We choose the LightGBM Regressor as it is the best choice for implementing the dynamic pricing algorithm due to its optimal balance between predictive performance and testing time, given our analysis. With an R^2 of 0.6983, it demonstrates strong predictive capability, while its testing time of 0.2149 seconds ensures it can handle real-time pricing efficiently. Additionally, the LightGBM Regressor has relatively low RMSE and MAE values, indicating higher prediction accuracy compared to the Gradient Boosting Regressor. These qualities make it a well-suited model for dynamic pricing applications where both accuracy and speed are critical.

Lightweight Gradient Boosting Mechanism Regressor (LightGBM Regressor):

It is a gradient boosting algorithm optimized for efficiency and speed is trainde using Kaggle dataset. After predicting the base fare, additional multipliers are applied to account for real-time conditions like event multiplier.

- **Event Multiplier** - Adjusts the fare during special events or holidays. We have used the holidays() method of the USFederalHolidayCalendar to retrieve a list of holidays in that range. If the ride date is a holiday, the multiplier is applied (e.g., 1.2 for a 20% increase).

Code Snippet of LGBM Regressor Model:

- **Data Cleaning :**

```
data = data.dropna()
data = data[(data['fare_amount'] > 0) & (data['fare_amount'] < 500)]
data = data[(data['pickup_longitude'] >= -180) & (data['pickup_longitude'] <= 180)]
data = data[(data['pickup_latitude'] >= -90) & (data['pickup_latitude'] <= 90)]
data = data[(data['dropoff_longitude'] >= -180) & (data['dropoff_longitude'] <= 180)]
```

- `data = data[(data['dropoff_latitude'] >= -90) & (data['dropoff_latitude'] <= 90)]`

- **Feature Engineering :**

```
data['pickup_datetime'] = pd.to_datetime(data['pickup_datetime'])
data['hour'] = data['pickup_datetime'].dt.hour
data['day_of_week'] = data['pickup_datetime'].dt.dayofweek
data['month'] = data['pickup_datetime'].dt.month
# Compute distance using the Haversine formula
def haversine_distance(lat1, lon1, lat2, lon2):
  R = 6371 # Radius of Earth in kilometers
  phi1, phi2 = np.radians(lat1), np.radians(lat2)
  dphi = np.radians(lat2 - lat1)
  dlambda = np.radians(lon2 - lon1)
  a = np.sin(dphi/2)*2 + np.cos(phi1) * np.cos(phi2) * np.sin(dlambda/2)*2
  return R * 2 * np.arctan2(np.sqrt(a), np.sqrt(1 - a))

data['distance'] = haversine_distance(
  data['pickup_latitude'], data['pickup_longitude'],
  data['dropoff_latitude'], data['dropoff_longitude']
)
```

- **Model Training and Testing :**

```
# Train a LightGBM model
model = LGBMRegressor(n_estimators=100, learning_rate=0.1, random_state=42)
model.fit(X_train, y_train)

# Evaluate the model
y_pred = model.predict(X_test)
rmse = np.sqrt(mean_squared_error(y_test, y_pred))
#print(f'RMSE: {rmse:.2f}')

def predict_fare(pickup_lat, pickup_lon, dropoff_lat, dropoff_lon, datetime, passenger_count,
is_event):
  distance = haversine_distance(pickup_lat, pickup_lon, dropoff_lat, dropoff_lon)
  hour = datetime.hour
  day_of_week = datetime.weekday()
  month = datetime.month

  # Base feature engineering
  input_features = pd.DataFrame({
    'distance': [distance],
    'hour': [hour],
    'day_of_week': [day_of_week],
    'month': [month],
    'passenger_count': [passenger_count]
  })
  # Predict base fare
  base_fare = model.predict(input_features)[0]
```

- **Calculating Final Fare:**

```
# Event Multiplier
event_multiplier = 1.2 if is_event else 1.0
# Final fare calculation
adjusted_fare = base_fare * event_multiplier
```

Some of the other advantages LGBM Regressor over other models are as follows:

- **Fast Training-** LightGBM uses a histogram-based algorithm to bucket continuous feature values, significantly speeding up training and testing.
- **Efficient Memory Usage-** Its implementation is optimized for low memory consumption, making it suitable for large datasets.
- **Flexibility-** works well with both numerical and categorical data. Supports a wide range of loss functions, including custom ones, making it adaptable to various regression problems. Natively supports categorical features without needing one-hot encoding, reducing memory usage and training time.

C. Error Handling:

Error handling is critical aspect of our application logic codebase, ensuring that our system can efficiently handel unexpected issues and provide meaningful feedback to its users and developers. Following is brief summary of how we have implemented error and exception handling in our code.

- **Structured try-except blocks for general exception handling:** our application uses these structured blocks to manage exception during critical operations. This ensures that any errors encountered during execution are caught and handled without crashing the application.
- **Database Error Handling:** the application uses MySQL- specific handling to manage the database related errors, such as connection failures, invalid queries, or missing data.
- **Input Validation:** to ensure robustness, our application validates all incoming requests before processing. Missing or invalid parameters or arguments leads to 400 Bad Request response with a clear and precise error message.
- **HTTP Error Responses:** our application provides JSON error messages with appropriate HTTP status codes to inform clients of the issue.
- **Session and Authorization Errors:** session validation is a key component of error handling. The application checks for a valid session (uuid) before executing operations. Invalid or expired sessions return a 401 unauthorized error.
- **Resource Cleanup:** The application ensures proper cleanup of resources, such as database connections and cursors, even in the event of an error. This is achieved using finally blocks and it helps in preventing resource leaks and ensures consistent application behaviour.

D. Driver Proximity Logic:

The driver proximity logic in our application is designed to optimize the allocation of drivers to ride requests. By leveraging geospatial calculations and dynamic filterring, the system ensures efficient and fair matching of drivers to users. A driver is visible to a user if they are within a 10-mile radius. The logic incorporates the Haversine formula to calculate the distance menasurements on the Earth's spherical surface. The logic ensures efficient and fair driver allocation within a reasonable range, improbving the overall reliability and user satisfaction of the application.

```
def haversine_distance(lat1, lon1, lat2, lon2):
    R = 6371 # Earth's radius in kilometers
    phi1, phi2 = np.radians(lat1), np.radians(lat2)
```

Haversine Implementation:

```
dphi = np.radians(lat2 - lat1)
dlambda = np.radians(lon2 - lon1)
a = np.sin(dphi / 2)**2 + np.cos(phi1) * np.cos(phi2) * np.sin(dlambda / 2)**2
return R * 2 * np.arctan2(np.sqrt(a), np.sqrt(1 - a))
```

The driver proximity logic handles:

- Identify the closest available drivers to a user's pickup location.
- Allocate the nearest driver based on real-time geospatial data.
- Minimize user wait times and reduce driver idle times.

E. Caching Logic in Login and Travel History:

Caching is an integral part of our application for improving performance and scalability by reducing the need for repetitive and resource intensive database queries. We have utilised Redis for caching as it is a high-performing in-memory key-value store.

Our application connects to a Redis server using the StrictRedis class from the redis library:

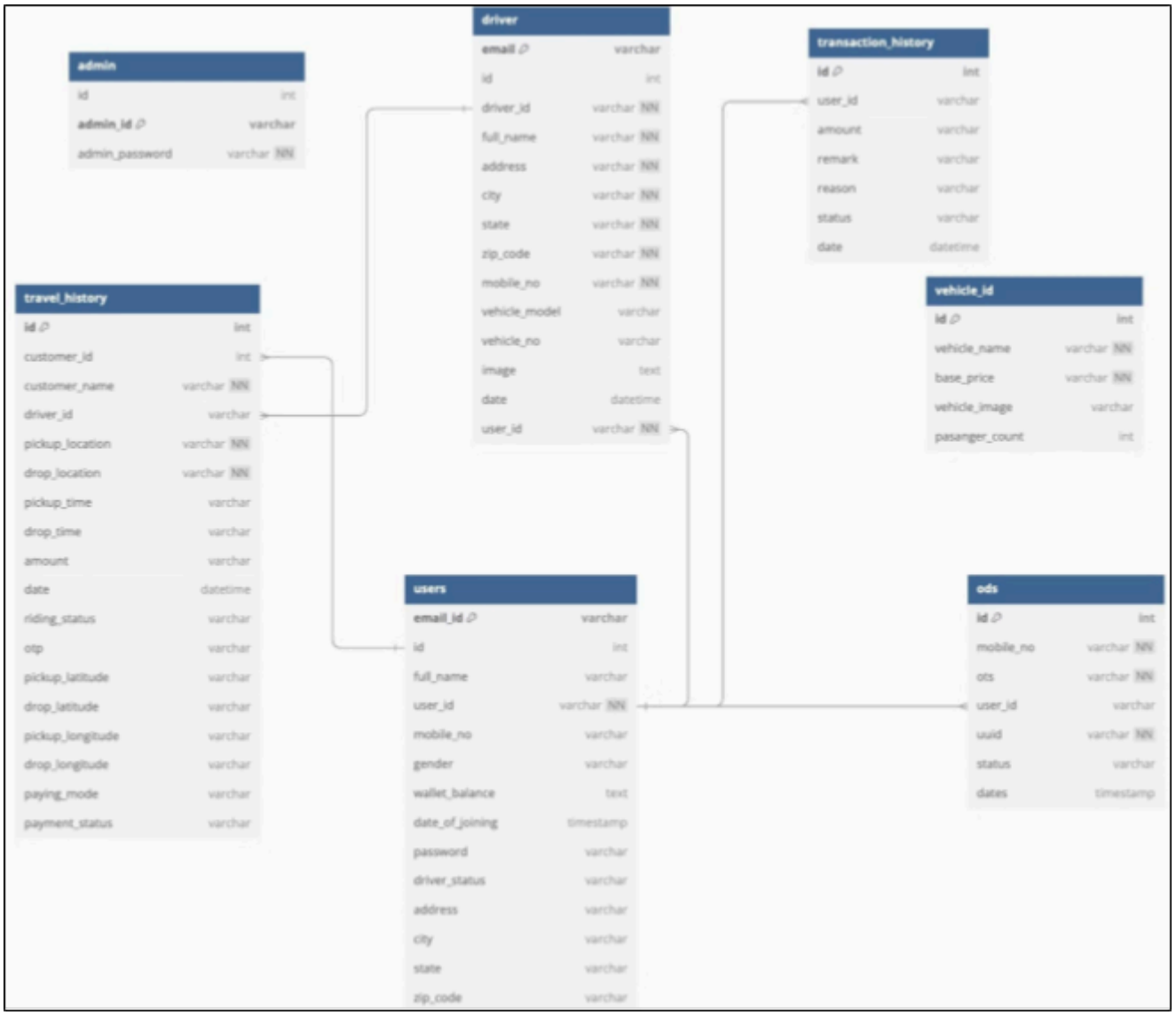
```
from redis import StrictRedis


# Redis configuration
redis_cache = StrictRedis(host='localhost', port=6379, db=0, decode_responses=True)
```

Redis caching is used in frequent tasks, like login states or travel history, to lessen database demand. Furthermore, while outdated session data is periodically erased, data retention standards guarantee that important information, like as transactions and ride details, are kept for reporting and compliance.

1.8 Database Design and Object Management:

Scalability, dependability, and effectiveness in data management across the program are the main goals of the database design and object management policy. The main relational database system, MySQL, is made to store and retrieve information about drivers, users, transactions, and ride histories. Primary keys are used to optimize each table, including users, travel_history, transaction_history, and drivers, in order to preserve data integrity and improve query performance. With the help of Redis caching for quicker, temporary data access and the ODS database for durable session data, user sessions are maintained using a hybrid approach. Session information is verified using the uuid and session status fields. Non-critical data like cached sessions leverage a lazy write approach with a short TTL (Time to Live) of five minutes. When it comes to controlling database operations, this organized approach guarantees a balance between scalability, consistency, and performance.

Database Schema:



This database schema is designed to handle and manage the core functionalities of our application. It is divided into several interconnected tables that store key information about users, drivers, trips, transactions, and vehicles. Below is an explanation of the tables and their relationships.

1. Admin: The admin table stores information about the administrators managing the platform. It includes.

id	A unique identifier for each admin.
admin_id	A unique login identifier for the admin.
admin_password	The password for admin authentication in the system.

2. Travel_history: The travel_history table records details of every trip made by the users on the platform. Each record in this table is linked to a customer and a driver, capturing information about the ride's details, location, and payment.

id	A unique identifier for each travel record.
customer_id	Foreign key referencing the users table, identifying the customer who took the trip
customer_name	The name of the customer.

driver_id	Foreign key referencing the driver table, identifying the driver of the trip.
pickup_location	The location where the customer was picked up
drop_location	The location where the customer was dropped off.
pickup_time and drop_time	The start and end times of the trip.
amount	The fare charged for the trip.
date	The date when the trip took place.
riding_status	The status of the ride (e.g., completed, canceled).
payment_status	The payment status for the ride (e.g., paid, pending).

3. Driver: the driver table contains information about the drivers operating on the platform.

email_id	A unique email address for the driver.
full_name	The full name of the driver
address	The address of the driver.
vehicle_model	The model of the vehicle used by the driver.
vehicle_number	The registration number of the vehicle.
image	A photo or profile picture of the driver.
date	The date the driver joined the platform.

4. Transaction_history: the transaction_history table tracks financial transactions associated with users. It records information such as payment amounts and statuses for each transaction.

id	A unique identifier for each transaction.
user_id	Foreign key referencing the users table, identifying the user involved in the transaction.
amount	The amount of the transaction.
remark	Additional notes or details regarding the transaction.
status	The status of the transaction (e.g., completed, failed).
date	The timestamp when the transaction was made.

5. Vehicle_id: the vehicle_id table stores data about the vehicles used by drivers in the system.

id	A unique identifier for each vehicle.
vehicle_name	The name of the vehicle.
basic_price	The base price associated with the vehicle for fare calculation.
vehicle_image	An image of the vehicle.
passenger_count	The number of passengers the vehicle can accommodate.

6. Users: The users table stores information about the customers who use the platform for rides.

email_id	A unique email address for the user.
full_name	The full name of the user
gender	The gender of the user.
wallet_balance	The available balance in the user's wallet for making payments.
date_of_joining	The date when the user registered on the platform.
address, city, state, and zip_code	The user's residential details.

7. ods (Operational Data Store): the ods table collects operational data related to users.

id	A unique identifier for each record in the operational data.
mobile_no	The mobile number of the user.
oss	A reference to the source system or platform from where the data originated.
user_id	Foreign key linking to the users table, identifying the user.
status	The current status of the user (e.g., active, inactive)
dates	A timestamp indicating when the record was updated

Relationships:

- The **users** and **travel_history** tables have a one-to-many relationship, where each user can have multiple rides (travel records).
- The **driver** and **travel_history** tables have a one-to-many relationship, where each driver can be associated with multiple rides.
- The **vehicle_id** and **driver** tables likely have a one-to-one relationship, with each driver operating a specific vehicle.
- The **transaction_history** table has a one-to-many relationship with the **users** table, as a user can have multiple financial transactions.
- The **ods** table stores supplementary data related to users, maintaining operational details like user status and mobile numbers.

Code snippets for Database Design Implementation:
The database access class is responsible for managing database connections and executing queries. Below is an example implementation

```
import mysql.connector

def create_connection():
  return
    mysql.connector.connect(
    host="localhost",
    user="root",
    password="password",
    database="ride_sharing"
```

To prevent SQL injection, parameterized queries are used. Values are passed as parameters instead of being concatenated into the query string.

```
cursor.execute("SELECT * FROM users WHERE email_id = %s", (email,))
cursor.execute("INSERT INTO users (full_name, email_id) VALUES (%s, %s)", (full_name, email))
```

The application likely uses transactions to ensure data integrity. Transactions group multiple operations into a single unit of work that either fully succeeds or fails.

```
try:
 connection.start_transaction()
 cursor.execute("UPDATE users SET wallet_balance = %s WHERE user_id = %s", (balance, user_id))
    cursor.execute("INSERT INTO transaction_history (user_id, amount) VALUES (%s, %s)", (user_id,
amount))
 connection.commit() # Commit the transaction
except Error as e:
 connection.rollback() # Rollback if any error occurs
 print("Transaction failed:", e)
```

Code listing of your server implementations for the entity objects

These snippets include the relevant server implementations for managing the User entity, focusing on session validation, user registration, and authentication. Let me know if you need further details or snippets for other entities or functionalities.

1. **Check user session**
   This function checks the session status of a user based on their UUID.

```
def checkusersession(uuid):
  try:
    cursor.execute("SELECT status FROM ods WHERE uuid = %s", (uuid,))
    user = cursor.fetchone()
    if user:
      return user[0] # Return the status of the session
    return None
  except Exception as e:
    print(f"Error in checkusersession: {e}")
    return None
```

2. **Get User ID by UUID**
   Retrieves a user ID by their UUID, with caching for optimized performance.

```
def getuseridbyuuid(uuid):
  try:
    cached_userid = redis_cache.get(f"user_id:{uuid}")
    if cached_userid:
      return cached_userid
    cursor.execute("SELECT user_id FROM ods WHERE uuid = %s", (uuid,))
    user = cursor.fetchone()
```

```
    if user:
        user_id = user[0]
        redis_cache.setex(f"user_id:{uuid}", 300, user_id) # Cache for 5 minutes
        return user_id
    return None
except Exception as e:
    print(f"Error in getuseridbyuuid: {e}")
    return None
```

3. **User Registration**
   Registers a new user and stores their details in the database.

```
@app.route('/register', methods=['POST'])
def register_user():
  try:
    data = request.json
    username = data.get('full_name')
    emailid = data.get('email_id')
    mobile = data.get('mobile_no')
    password = data.get('password')
    gender = data.get('gender')
    address = data.get('address')
    city = data.get('city')
    state = data.get('state')
    zip_code = data.get('zip_code')

    if not username or not password:
        return jsonify({'status': 'Failed', 'message': 'Missing required fields'}), 400

    userid = 'CSH' + str(random.randint(10000, 99999))

    # Check if user already exists
    cursor.execute(
        "SELECT email_id FROM users WHERE email_id=%s OR mobile_no =%s",
        (emailid, mobile)
    )
    if cursor.fetchone():
        return jsonify({'status': 'Failed', 'message': 'User already registered'}), 400

    # Insert user into the database
    cursor.execute(
        """
```

```
        INSERT INTO `users`(`full_name`, `email_id`, `user_id`, `mobile_no`, `gender`,
        `blocking_status`, `wallet_balance`, `date_of_joining`, `password`, `driver_status`,
        `address`, `city`, `state`, `zip_code`)
        VALUES (%s, %s, %s, %s, %s, %s, %s, NOW(), %s, %s, %s, %s, %s, %s)
        """,
        (username, emailid, userid, mobile, gender, '0', '0', password, '0', address, city, state, zip_code)
    )
    connection.commit()
    return jsonify({'status': 'Success', 'message': 'User registered successfully'}), 201
except Exception as e:
    print(f"Error in register_user: {e}")
    return jsonify({'status': 'Failed', 'message': 'Error during registration'}), 500
```

Other code listing (utility classes, etc)

4. **Redis Configuration and Utility**
   The application uses Redis for caching frequently accessed data to improve performance.

```
def checkusersession(uuid):
    # Check Redis cache first
    cached_status = redis_cache.get(f"session:{uuid}")
    if cached_status:
        return cached_status == 'verified'

    # Fetch from database if not cached
    conn = create_connection()
    cursor = conn.cursor()
    cursor.execute("SELECT status FROM ods WHERE uuid = %s", (uuid,))
    user = cursor.fetchone()
    if user:
        status = user[0]
        redis_cache.setex(f"session:{uuid}", 300, status) # Cache for 5 minutes
        return status == 'verified'
```

5. **Redis-Based Caching for User ID Retrieval**
   A utility to retrieve and cache user IDs for optimized performance.

```
def getuseridbyuuid(uuid):
    # Check Redis cache first
    cached_userid = redis_cache.get(f"user_id:{uuid}")
    if cached_userid:
        return cached_userid

    # Fetch from database if not cached
```

```
conn = create_connection()
cursor = conn.cursor()
cursor.execute("SELECT user_id FROM ods WHERE uuid = %s", (uuid,))
user = cursor.fetchone()
if user:
    user_id = user[0]
    redis_cache.setex(f"user_id:{uuid}", 300, user_id) # Cache for 5 minutes
    return user_id
return None
```

6. **Test Connection Endpoint**
   An API route to test the database connection.

```
@app.route('/test_connection')
def test_connection():
try:
conn = create_connection() cursor =
    conn.cursor()
cursor.execute("SELECT DATABASE();") current_db =
    cursor.fetchone()
    return jsonify({'status': 'Success', 'message': f'Connected to database {current_db[0]}'}), 200
  except Exception as e:
return jsonify({'status': 'Failed', 'message': f'Error: {e}'}), 500
```

Policies used to decide when to write data to the database:

Our database write policies are designed to ensure secure, consistent, and efficient data management. Writes are triggered by specific user actions, system events, or conditional logic, such as user registration, order creation, status updates, or transaction logging. To maintain data integrity, all write operations adhere to primary and foreign key constraints and are executed only when necessary conditions are met, avoiding redundancy. Data consistency is achieved through the use of transactions, ensuring that all operations within a transaction are either fully completed or rolled back in case of errors. Additionally, parameterized queries are used to prevent SQL injection, enhancing the security of database interactions. This robust approach ensures that every write operation is purposeful, error-free, and aligned with the system's relational structure.

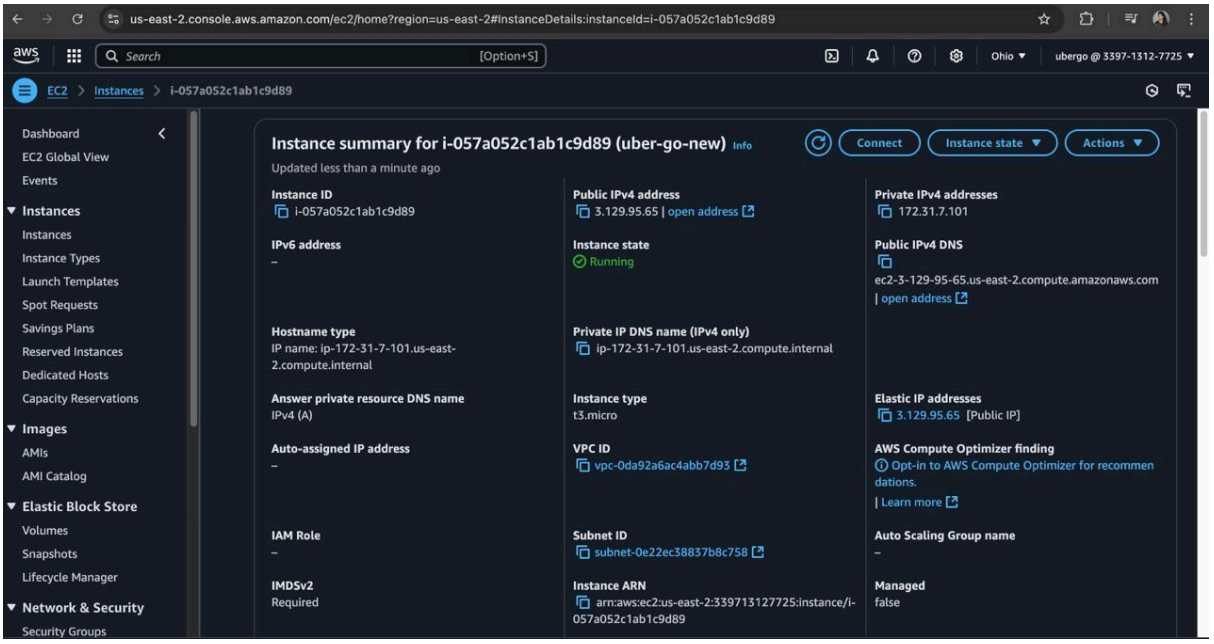
- User Registration
- Order Creation
- Updates to wallet balance or status
- Updates to travel history as the status changes.
- Transaction logging - Every transaction is logged in transaction_history
- Event driven writes - Writing to travel_history when a new ride is initiated. Logging transactions after wallet updates
- Begin a transaction, write data, and commit only when all steps are successful.

6. Deployment

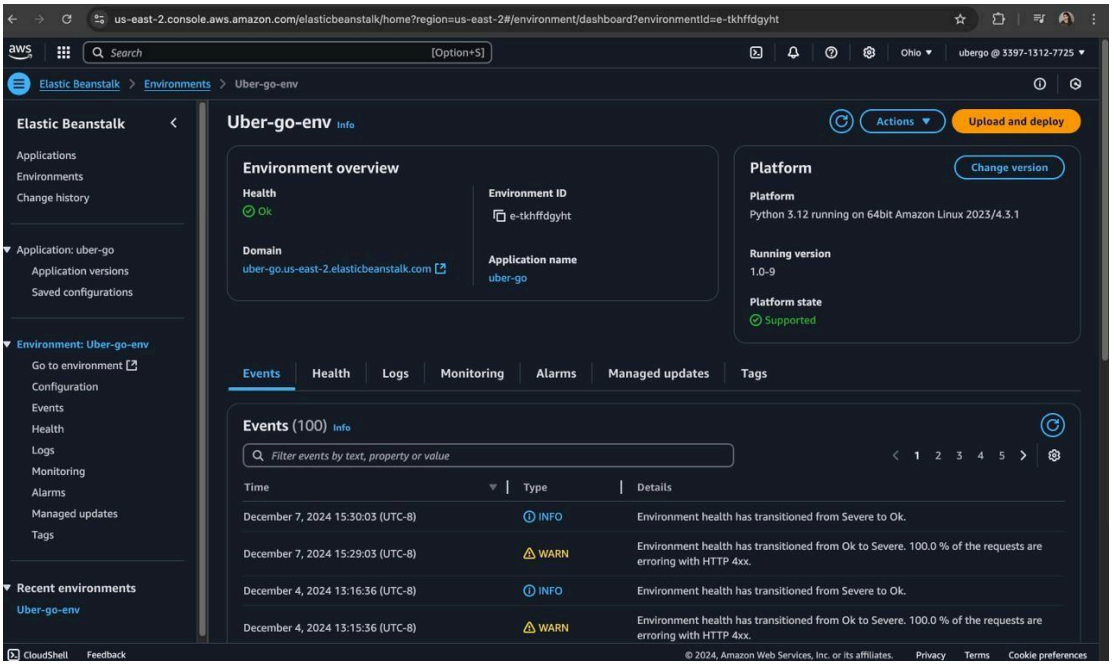
User : http://3.129.95.65/user/login.html
Driver: http://3.129.95.65/driver/login.html
Admin:
http://3.129.95.65/admin/login.html

Our Uber project leverages a robust deployment and database management infrastructure through AWS services, including EC2, Elastic Beanstalk, and RDS (MySQL). The EC2 instance serves as the backbone for computing resources, hosting the backend APIs and enabling real-time processing for requests like ride assignments, fare predictions, and user sessions. Elastic Beanstalk simplifies application deployment, scaling, and monitoring by managing the environment hosting our backend application, ensuring high availability and seamless updates. Meanwhile, Amazon RDS with MySQL provides a scalable, reliable database to store critical data such as user profiles, ride details, and transaction history. With automated backups, secure connections, and fault tolerance, RDS ensures the smooth functioning of data-intensive operations like dynamic pricing and transaction management. Together, these services form an integrated ecosystem that ensures reliability, scalability, and efficiency in delivering a seamless experience.

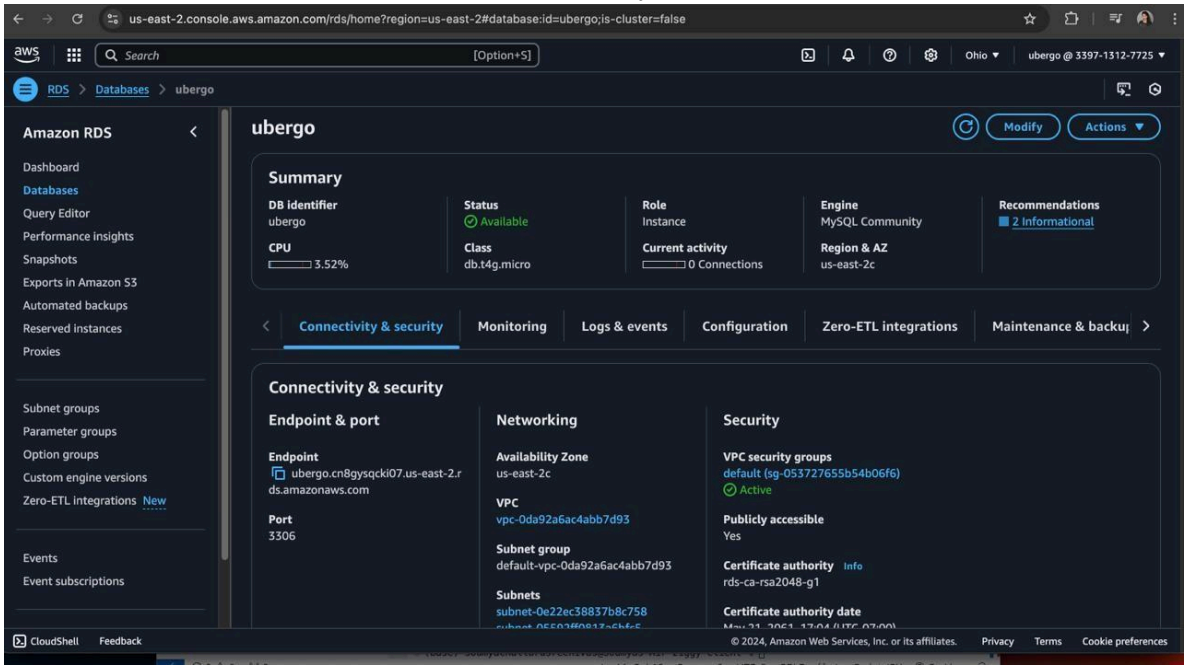
EC2 Instance:



Elastic Beanstalk:



AWS RDS - MySQL :

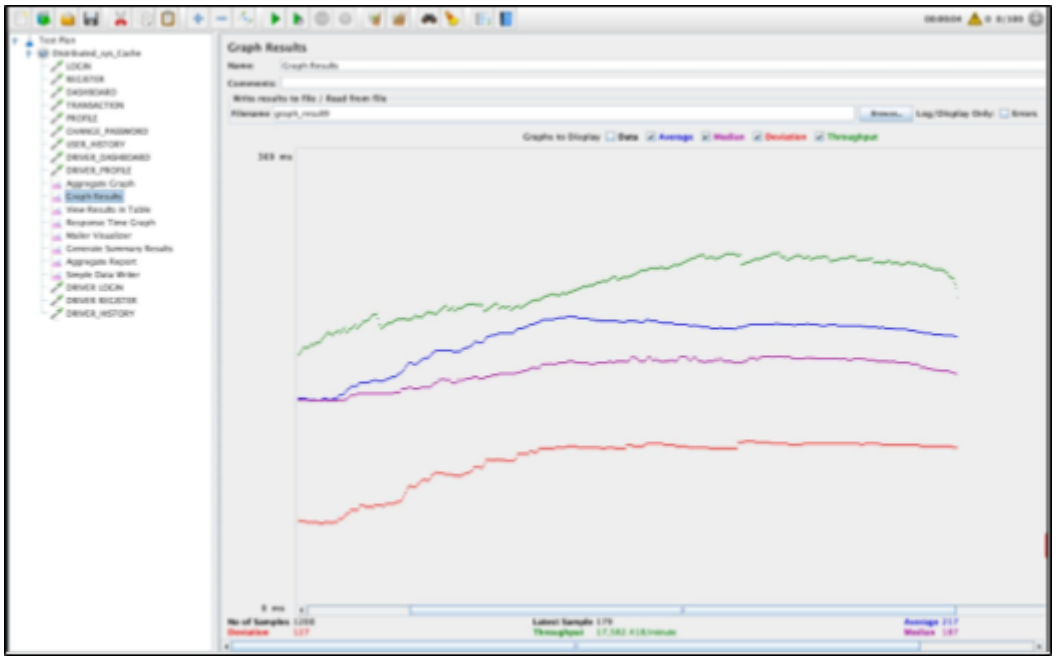


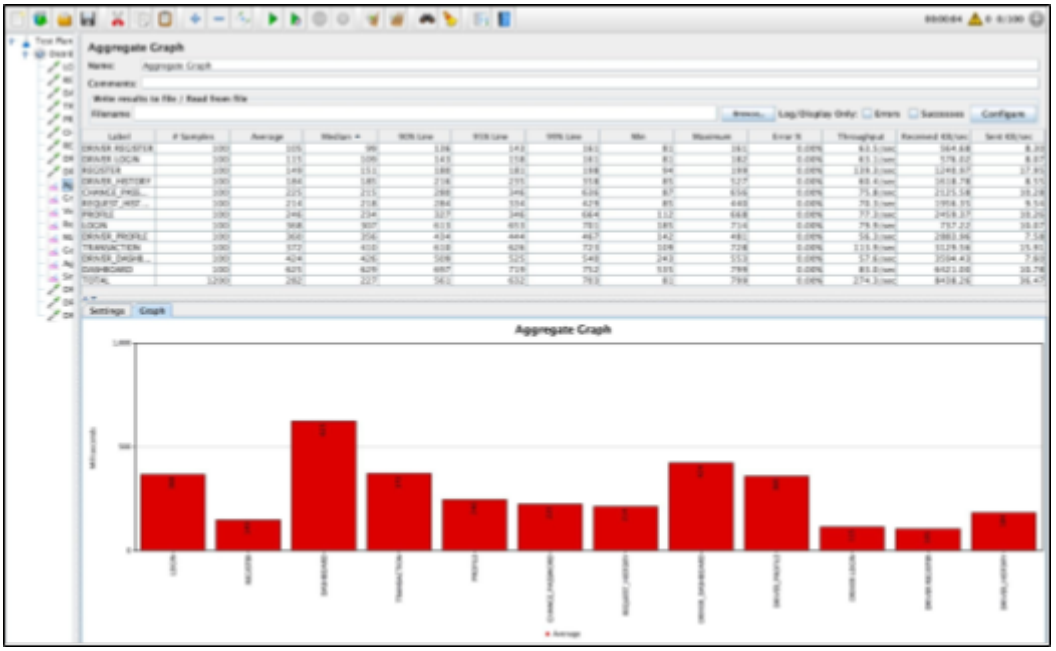
7. Load Testing, Stability and Scalability (Apache JMeter graphs)

The objective of this performance testing exercise was to evaluate the scalability, stability, and efficiency of the application under concurrent user load by analyzing key metrics such as response times, throughput, and error rates across multiple critical endpoints.

Testing Scope : The test plan included key functionalities of the system, such as:

- **Endpoints Tested:** LOGIN, REGISTER, DASHBOARD, TRANSACTION, PROFILE, DRIVER LOGIN, DRIVER REGISTER, etc.
- **Metrics Collected:** Average response time, median response time, percentile response times (90%, 95%, 99%), throughput, and error rates.
- **Number of Samples:** 100 requests per endpoint, with a total of 1200 requests processed during the test.





Findings

- **Throughput**
  The system demonstrated an excellent overall throughput of ~17,582 requests per minute, indicating its ability to handle high user concurrency.

- **Response Times**

  - **Fast Endpoints -** Endpoints like DRIVER REGISTER (105 ms) and DRIVER LOGIN (115 ms) showed highly efficient response times.

  - **Bottlenecks -** The DASHBOARD endpoint had the highest average response time (625 ms) and a 99th percentile time of 752 ms, identifying it as a potential bottleneck.Overall average response time across all endpoints was 282 ms, which is reasonable for most real-world applications.

- **Stability -** Consistent response times were observed for the majority of endpoints, with minimal variability (deviation ~127 ms).No errors were reported during the test, highlighting the robustness of the system.

- **High Percentile Delays -** Some endpoints, such as DASHBOARD and TRANSACTION, exhibited high 95th and 99th percentile response times, indicating occasional delays for certain requests.

Overall, the performance testing results indicate that the system is well-optimized for handling concurrent user requests, with excellent throughput and stable performance for most endpoints. The test evaluates multiple endpoints , twelve to be exact ensuring coverage of critical system functionality. The consistent response time and high throughput suggests the system maintains stability under load. No errors were reported, indicating robust error handling and backend logic.

---

8. Kafka Implementation

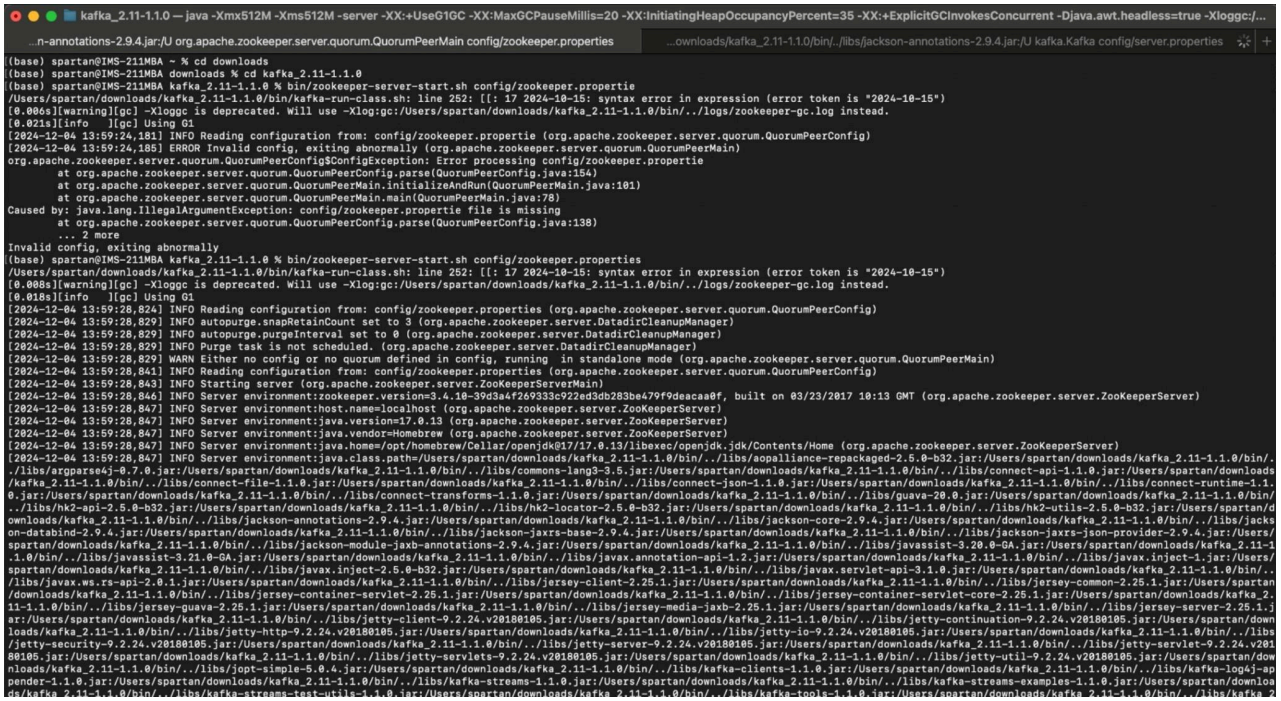
In our project, we incorporated Apache Kafka to enable real-time streaming of ride requests and status updates. The idea was to use Kafka's high throughput and fault tolerance capabilities to process events from two topics: ride_request and ride_status_update. The uber_consumer.py script was designed to consume these events, retrieve ride and passenger details, update ride statuses, and log the relevant information. While our setup included a functional Kafka consumer and producer, we faced significant challenges in ensuring consistent data flow and connectivity between Zookeeper, Kafka, and the consumer.

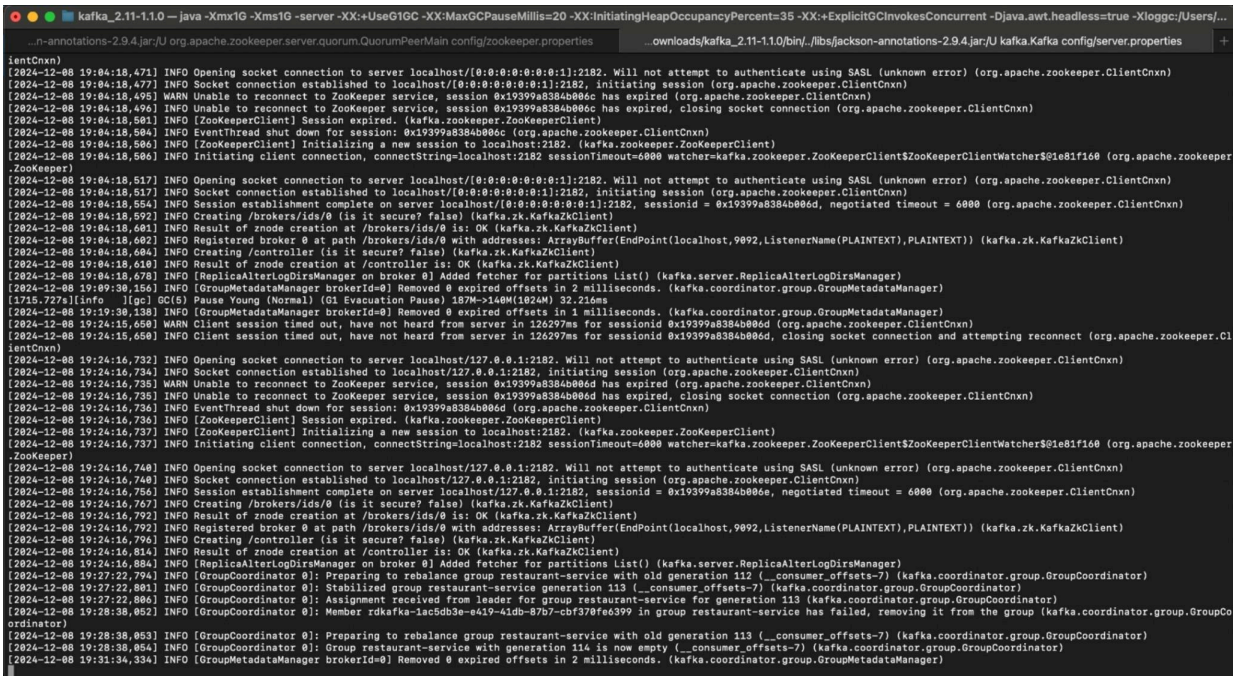
Proof of Work

We implemented a Kafka consumer (uber_consumer.py) that was capable of polling data from the ride_request and ride_status_update topics. The consumer fetched and sanitized ride and passenger data, updated statuses, and logged

event details. This functionality demonstrated our understanding of Kafka's producer-consumer model and laid the groundwork for future improvements. Screenshots attached to this document show the Zookeeper and Kafka setup logs, the consumer script, and an example of the terminal output where the consumer processed a ride request, highlighting our partial progress in implementing Kafka.

Respective Snippets:





uber_consumer.py > ...

```python
def process_ride_request_event(event_data):
    try:
        ride_id = event_data.get('ride_id')
        passenger_id = event_data.get('passenger_id')

        # Fetch the ride and passenger details
        ride = Ride.objects.get(id=ride_id)
        passenger = Passenger.objects.get(id=passenger_id)

        ride.distance = sanitize_decimal(ride.distance)
        ride.estimated_fare = sanitize_decimal(ride.estimated_fare)

        print(f"Ride Request Received: Ride ID {ride_id}, Passenger {passenger.name}")
        print(f"Pickup Location: {ride.pickup_location}")
        print(f"Drop-off Location: {ride.dropoff_location}")
        print("Ride Details:")
        print(f" - Distance: {ride.distance} miles")
        print(f" - Estimated Fare: ${ride.estimated_fare}")
        print(f"Ride Status: '{ride.ride_status}'")

        # Save ride details to ensure any modifications are persisted
        ride.save()

    except Passenger.DoesNotExist:
        print(f"Error: Passenger with ID {passenger_id} not found.")
    except Ride.DoesNotExist:
        print(f"Error: Ride with ID {ride_id} not found.")
    except Exception as e:
        print(f"Error processing ride request: {str(e)}")
```

uber_consumer.py > ...

```python
def process_ride_status_update_event(event_data):
    try:
        ride_id = event_data.get('ride_id')
        new_status = event_data.get('status')

        # Fetch the ride details
        ride = Ride.objects.get(id=ride_id)

        # Sanitize and update the ride status
        ride.ride_status = new_status
        ride.save()

        print(f"Ride Status Update: Ride ID {ride_id}")
        print(f" - New Status: {new_status}")

    except Ride.DoesNotExist:
        print(f"Error: Ride with ID {ride_id} not found.")
    except Exception as e:
        print(f"Error processing ride status update: {str(e)}")
```

uber_consumer.py > ...

```python
def consume_ride_events():
    consumer = kafka_consumer()
    consumer.subscribe(['ride_request', 'ride_status_update'])

    print("Listening for ride events...")
    while True:
        message = consumer.poll(1.0)
        if message is None:
            continue
        if message.error():
            print(f"Consumer error: {message.error()}")
            continue

        topic = message.topic()
        event_data = json.loads(message.value().decode('utf-8'))

        if topic == 'ride_request':
            process_ride_request_event(event_data)
        elif topic == 'ride_status_update':
            process_ride_status_update_event(event_data)

        consumer.commit()
```

Limitations

Despite our efforts, we encountered several obstacles that prevented us from fully integrating Kafka into our project:

- **Zookeeper Configuration Issues:** Errors in the zookeeper.properties file delayed the initial setup. While the syntax issues were resolved, session timeout warnings persisted, leading to unstable Kafka-Zookeeper communication.
- **Consumer Reliability:** Although the uber_consumer.py script could process messages from Kafka topics, connectivity issues often disrupted its functionality, resulting in inconsistent event handling.

Future Work

Successful Kafka integration will greatly enhance the scalability and efficiency of our project by enabling real-time processing of ride requests and updates. This will lead to faster response times, better load handling, and improved user experience, ultimately making our system more robust and production-ready. To realize the full potential of Kafka in our project, we plan to address the limitations and integrate it successfully in the future. This will involve:

- Resolving Connectivity Issues: Fine-tuning Kafka and Zookeeper configurations, such as adjusting session timeouts and improving network stability, to ensure seamless producer-consumer communication.
- Optimizing Event Processing: Enhancing the consumer script to handle high volumes of data with fault tolerance and implementing a retry mechanism for failed messages.
- Improving Scalability: Using Kafka's partitioning and replication features to scale event processing and provide redundancy.

---

9. Observations and Lessons Learned

Throughout the course of this project, we encountered several valuable learning experiences and challenges that have deepened our understanding of building scalable, efficient, and real-world applications.These experiences not only helped us successfully build a functional prototype but also highlighted areas where we can grow and improve. This project gave us a deeper understanding of the complexities of real-world system design and implementation, preparing us for future challenges.

1. **Learning New Technologies**: Working with Flask, Redis, and Mapbox taught us how to design scalable systems that can manage real-time data efficiently. These technologies gave us hands-on experience in building an integrated system that mimics a real-world application.

2. **Dynamic Pricing**: Implementing dynamic pricing using LightGBM helped us understand how machine learning models are applied to real-world scenarios. By carefully balancing the trade-off between prediction accuracy and testing time, we were able to create a practical and effective solution.

3. **Collaboration and Teamwork**: Collaborating on key modules like session management and fare prediction emphasized the importance of teamwork and communication. It was a rewarding experience to see how different contributions came together to shape the project.

4. **Overcoming API Challenges**: While integrating the frontend and backend, we frequently faced CORS errors during API calls. Troubleshooting these issues taught us the importance of properly configuring headers and managing cross-origin requests effectively.

5. **Accurate Distance Calculations**: Early attempts at calculating distances using the Haversine formula occasionally gave incorrect results. Validating latitude and longitude data to ensure they were within valid ranges proved to be an essential step in improving accuracy.

6. **Deployment Hurdles**: Deploying the system on AWS presented challenges, particularly with misconfigured environment variables, which resulted in database connection errors such as "Access denied for user 'admin'." This taught us the importance of meticulous attention to detail during deployment.

7. **Error Logging and Debugging**: We learned the value of detailed error logging and robust debugging practices. For instance, logs were crucial in identifying and resolving issues like Redis desynchronization and database connection errors, saving us time and effort in troubleshooting.
8. **Challenges with Kafka Integration**: We attempted to integrate Apache Kafka for real-time data streaming of ride requests and status updates. While we were able to set up a Kafka consumer and producer, configuration

issues like Zookeeper session timeouts and inconsistent connectivity prevented us from achieving full functionality. This remains an area for future improvement.

9. **Insights from Consumer Implementation**: Our Kafka consumer script (uber_consumer.py) successfully processed ride-related events during testing, fetching and logging data as intended. However, the instability in event handling underscored the need for more reliable configurations and testing.

10. **Model Testing and Evaluation**: Choosing the LightGBM model for fare prediction was a valuable experience in evaluating models for real-world use. Its balance between predictive performance and testing time made it ideal for a system with real-time constraints.

11. **System Scalability**: Conducting load tests with Apache JMeter showed that our system performed well under significant concurrent user loads. However, testing under extreme scenarios revealed some bottlenecks that would need to be addressed in future iterations.

10. Conclusion

Our Uber simulation application successfully combines advanced technologies to build a scalable and efficient platform. The implementation of dynamic pricing using the LightGBM model ensures accurate and fair pricing, while AWS deployment enhances the system's reliability and scalability. This project offered us practical experience with tools such as Flask, Redis, and Mapbox, allowing us to design and implement an effective system. Overcoming challenges like debugging errors, optimizing configurations, and ensuring smooth integration helped deepen our understanding of system architecture and cloud-based deployment. Overall, this project was a rewarding learning experience that enhanced our technical skills and prepared us to address complex challenges in system development and deployment.

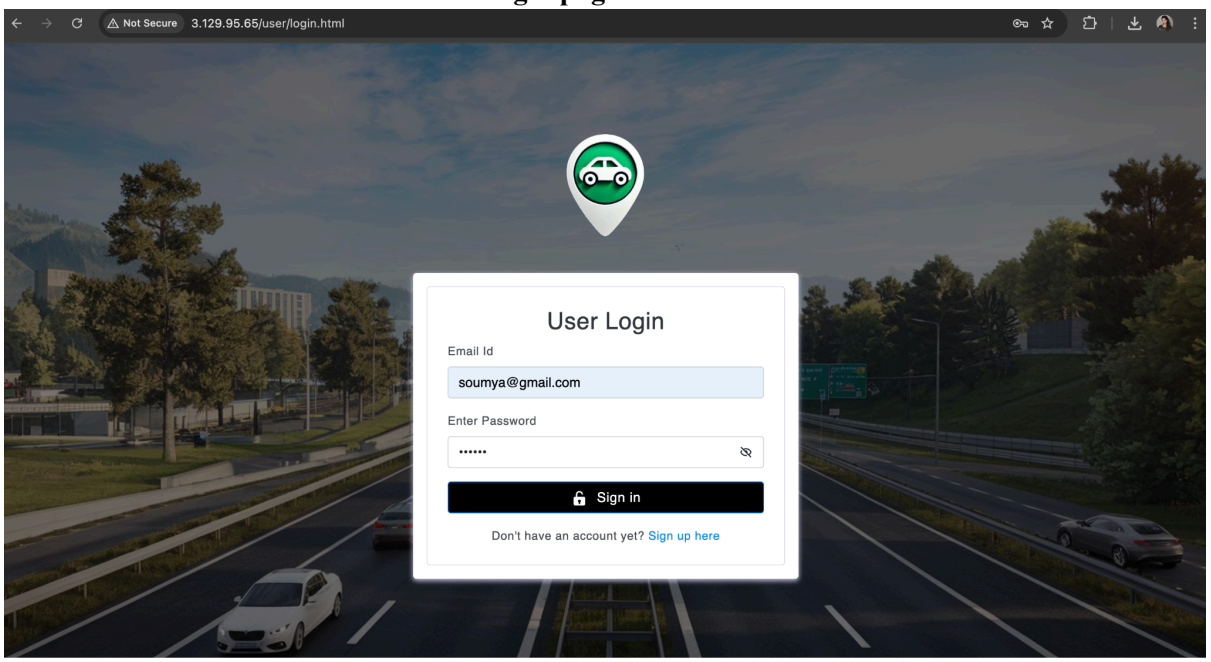
11. References

- **System Design of Uber App | Uber System Architecture**. GeeksforGeeks. Available at: https://www.geeksforgeeks.org/system-design-of-uber-app-uber-system-architecture/.
- **Ride-sharing with Redis**| Martin Joo. Available at: https://martinjoo.dev/ride-sharing-with-redis.
- **Design the Uber Backend: System Design Walkthrough** | Educative Blog. Available at: https://www.educative.io/blog/uber-backend-system-design.
- **System Design of Uber | Full System Design Walkthrough** |YouTube. Available at: https://www.youtube.com/watch?v=umWABit-wbk.

12. Application Screenshots

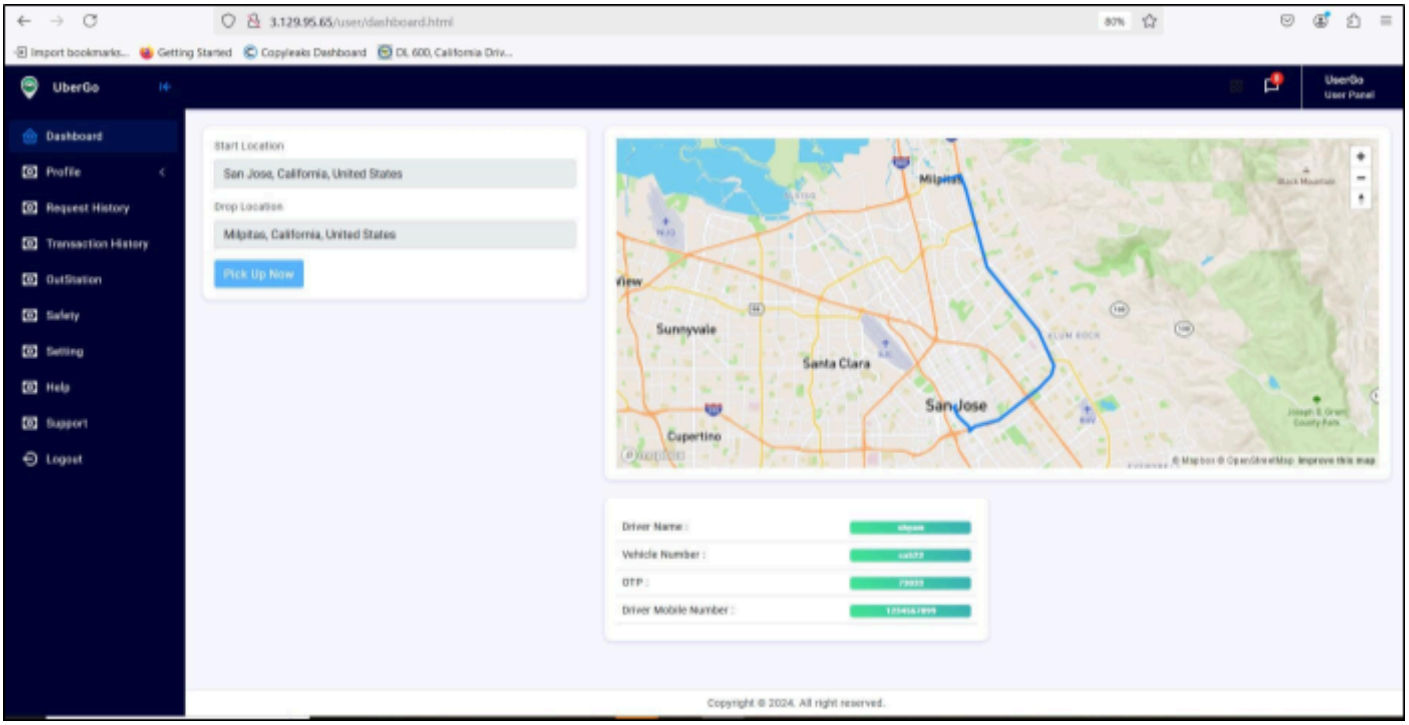
Customer Portal

Login page:

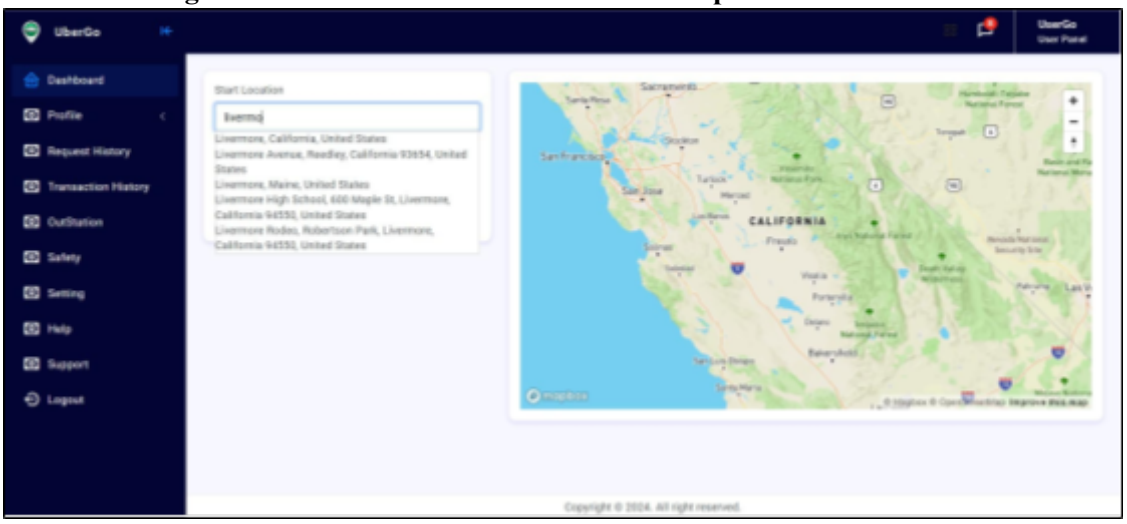


Register page:

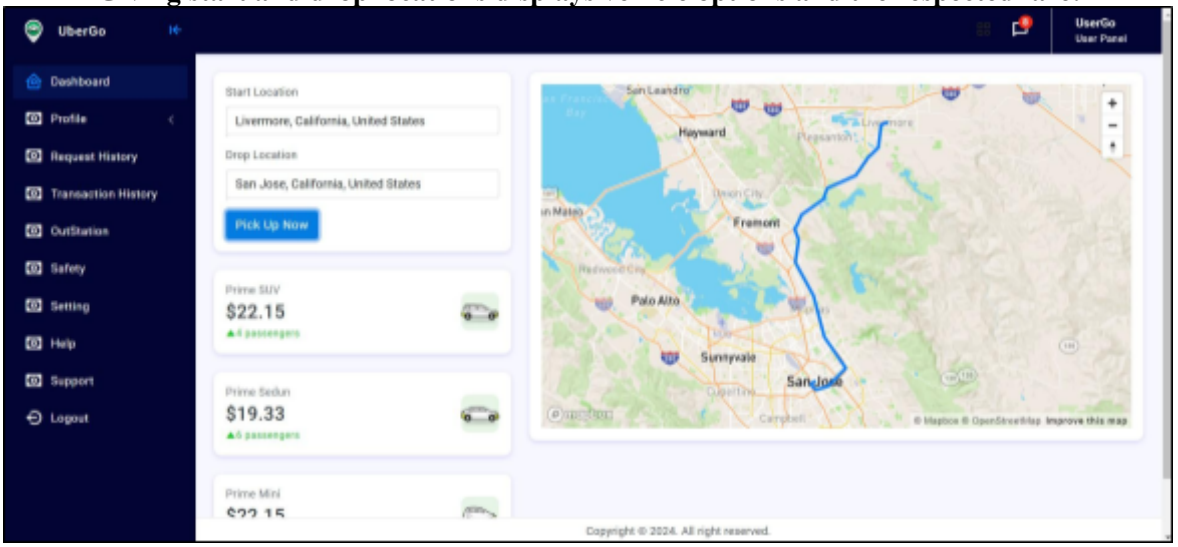
Home page:



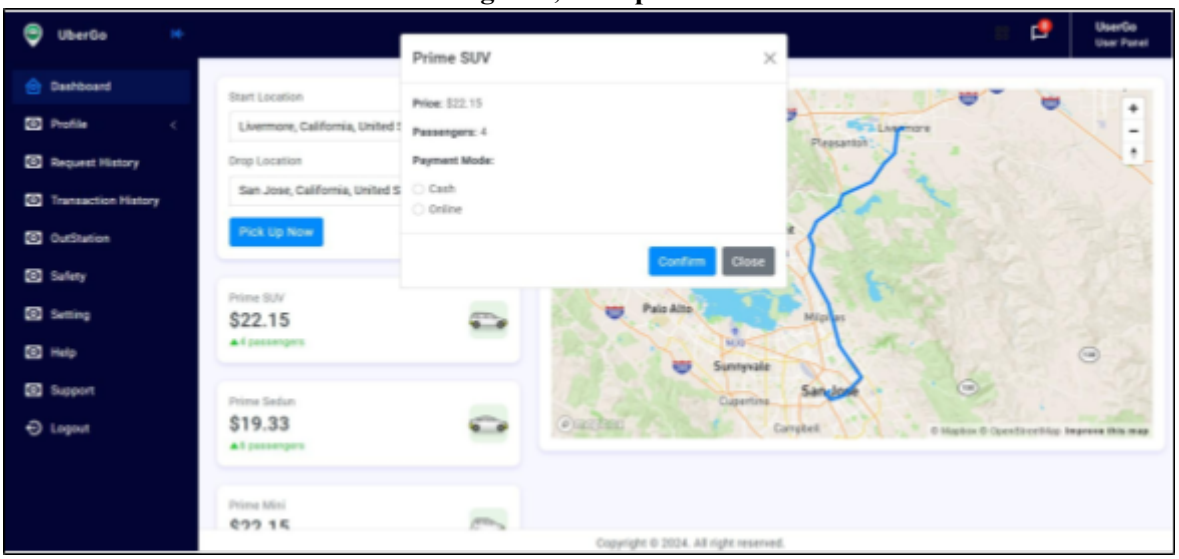
Entering a location for Start Location and the drop down shows the results:



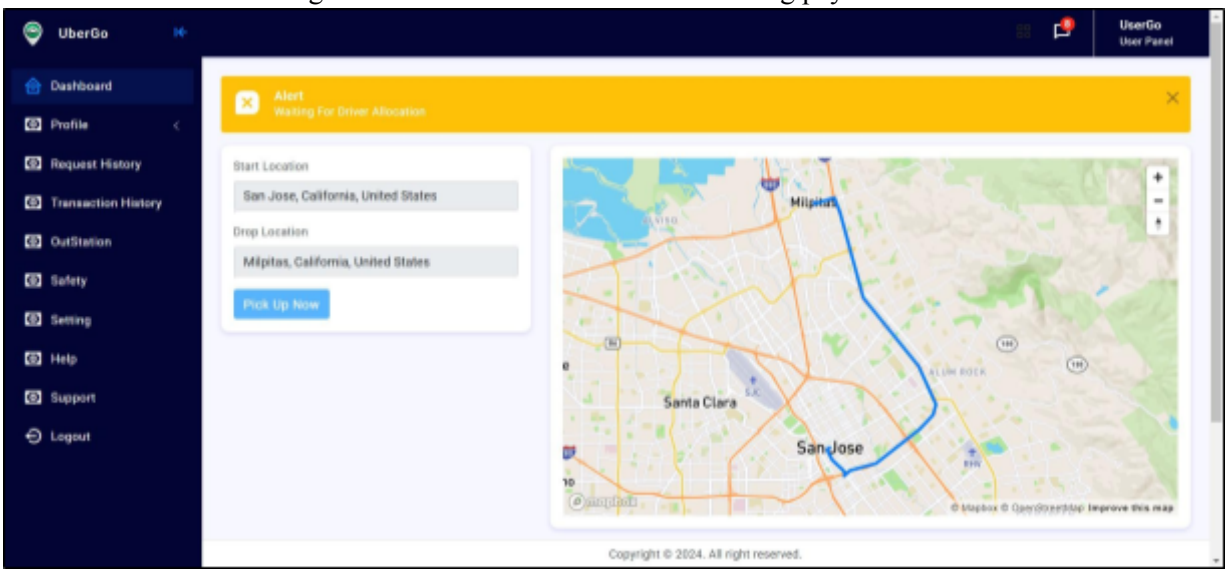
Giving start and drop locations displays vehicle options and the respected fare:



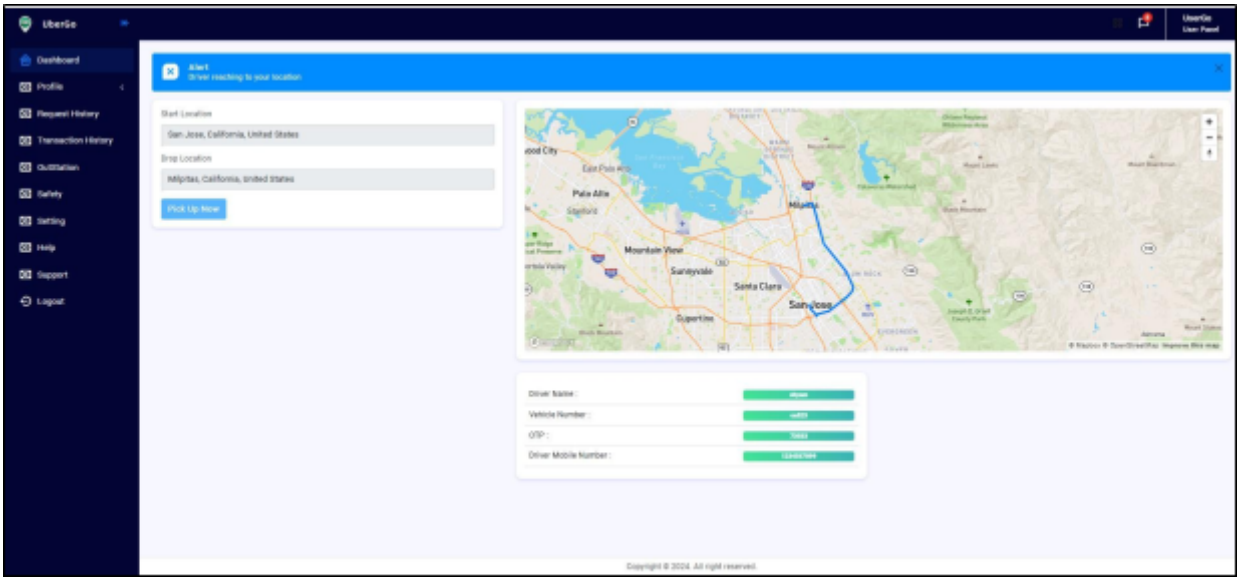
On choosing one , example: Prime SUV:



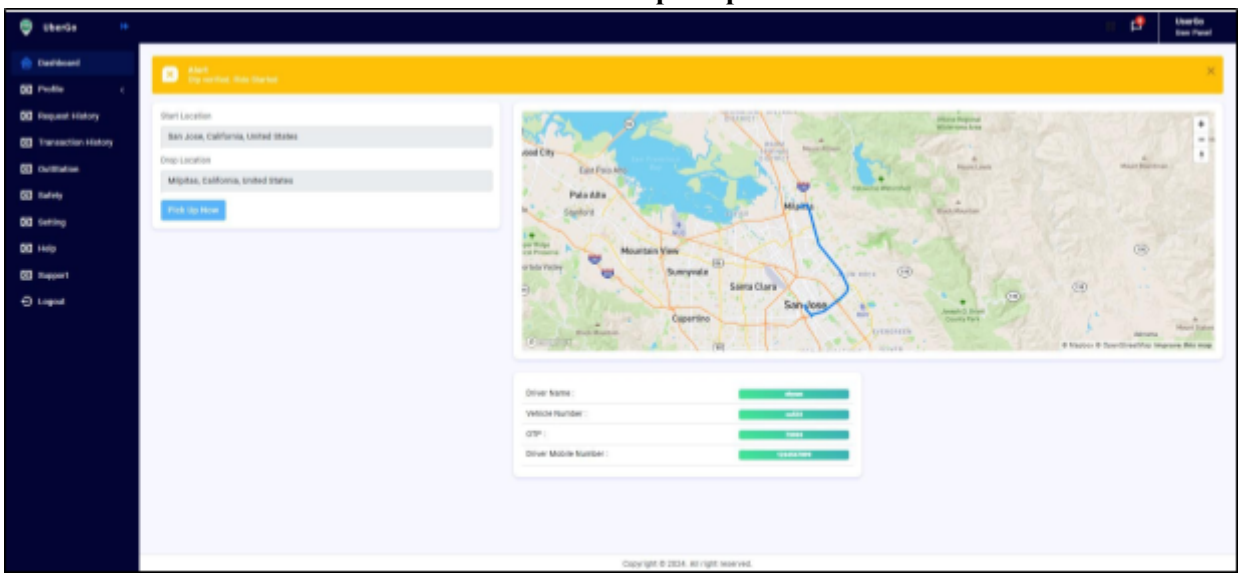
Waiting for driver to be allocated after choosing payment method



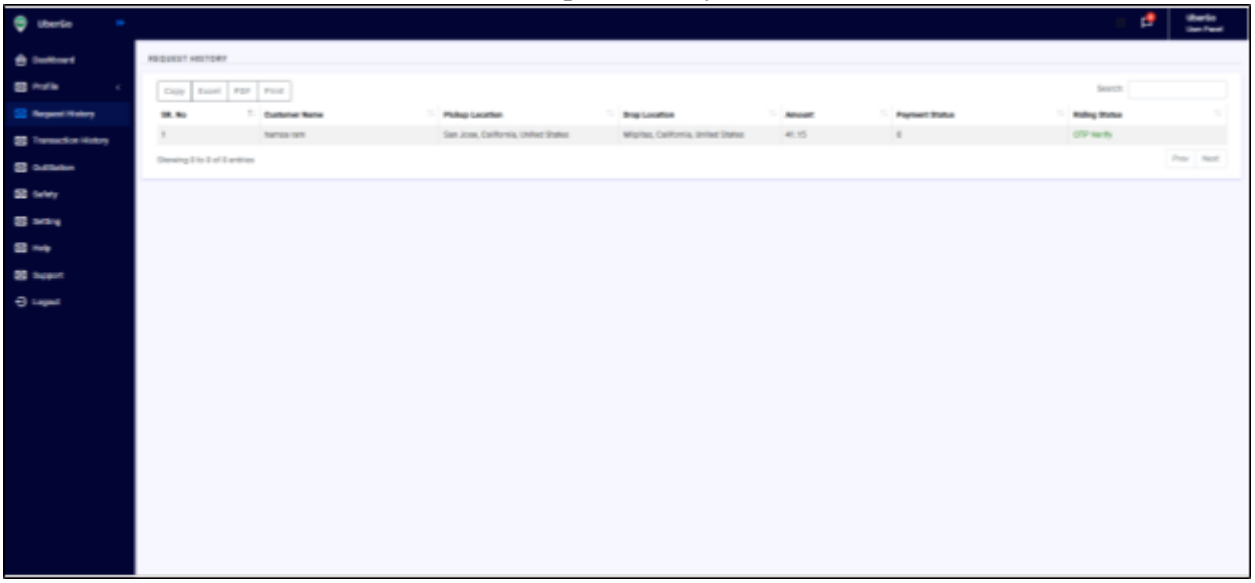
When a driver accepts the ride



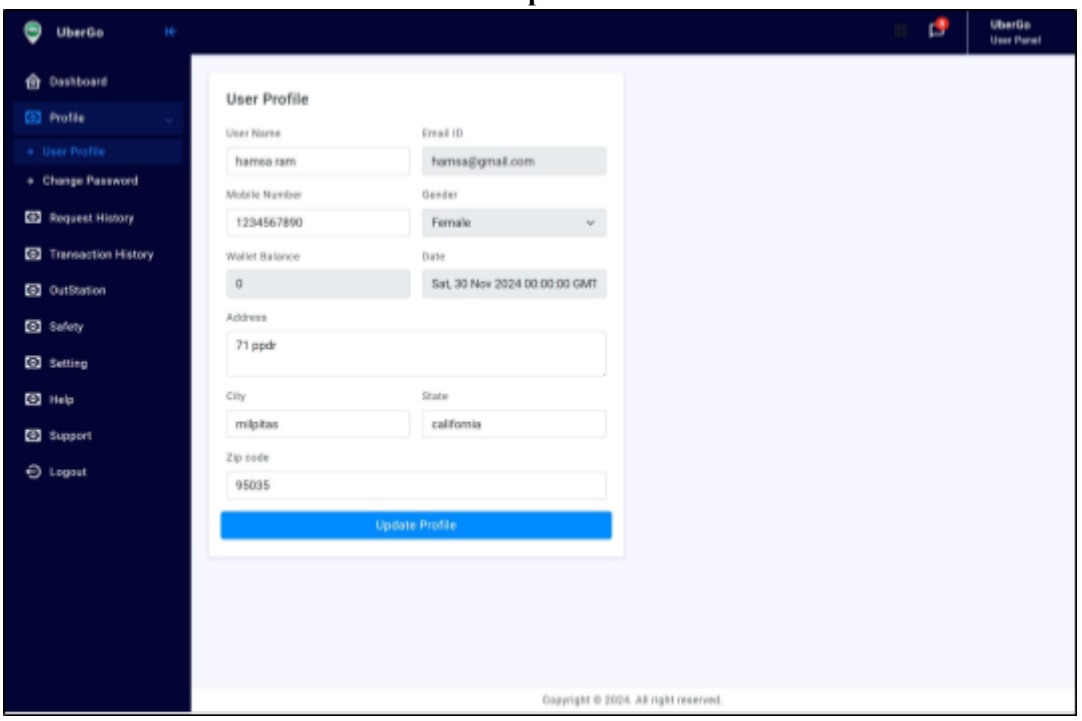
After driver reaches pickup location



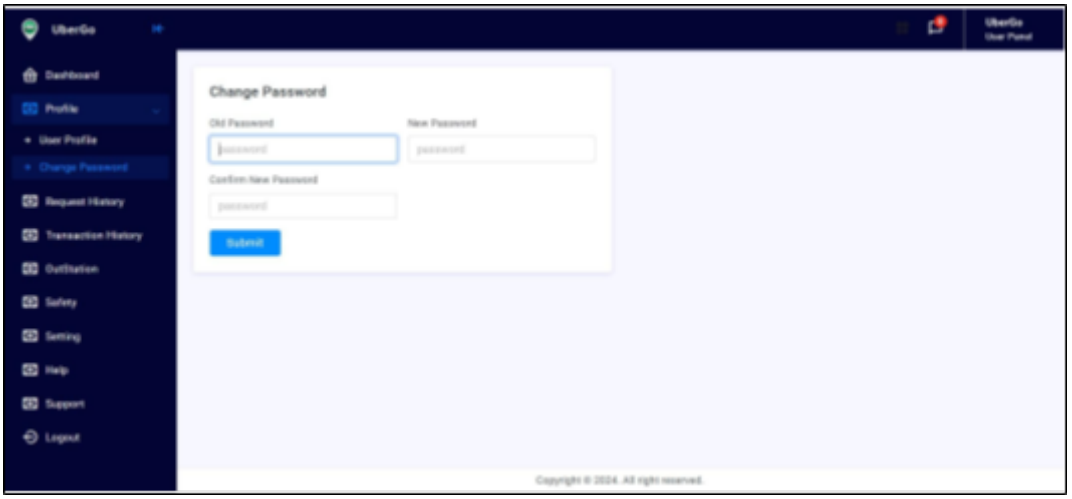
Request history:



User profile:

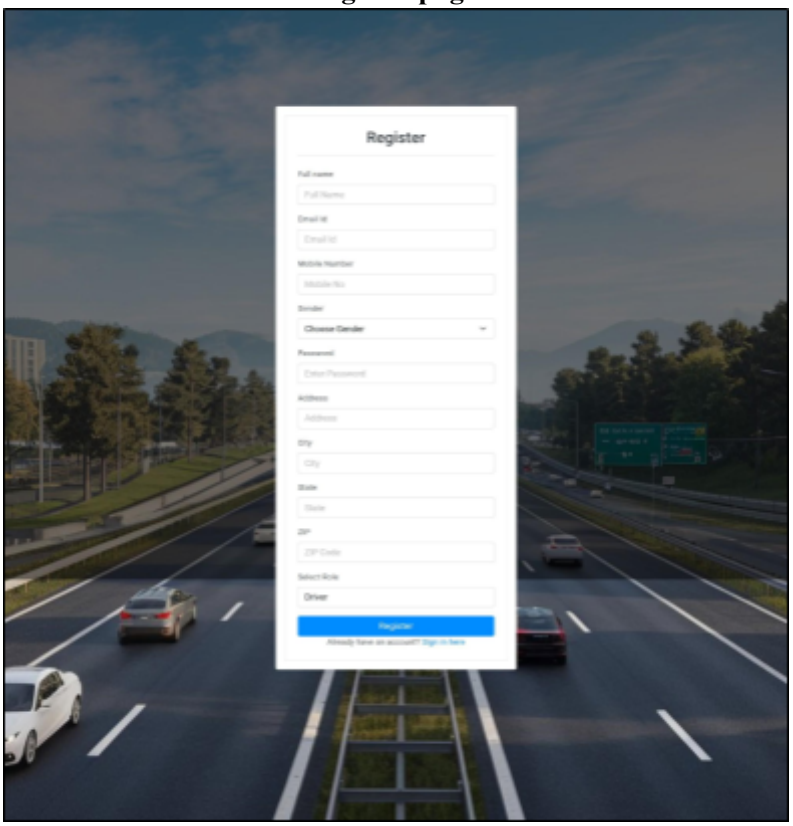


Change password page:

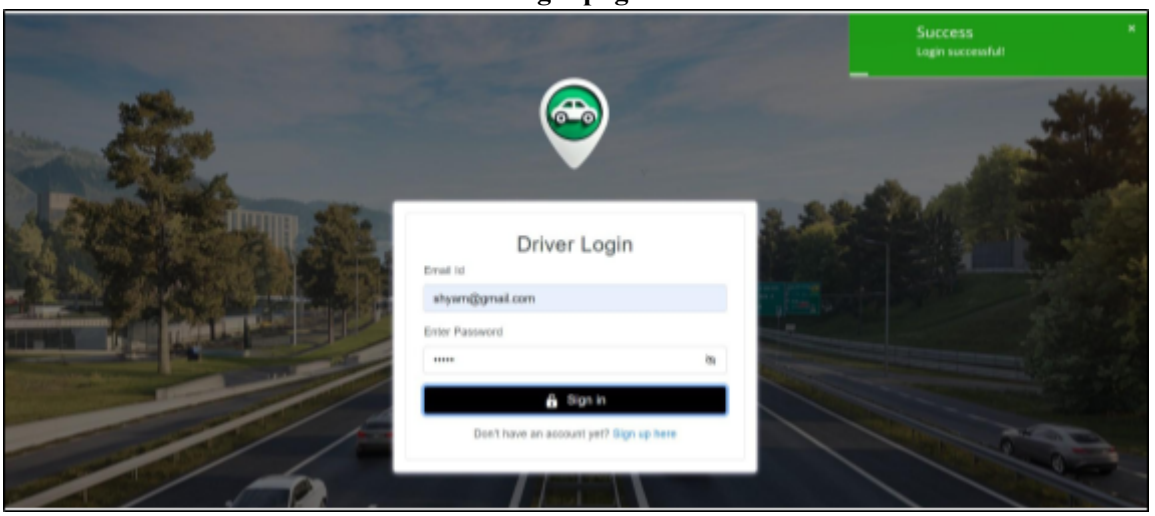

UberGo
Dashboard
Profile
User Profile
Change Password
Request History
Transaction History
Outstation
Safety
Setting
Help
Support
Logout
UberGo
User Panel
Change Password
Old Password
New Password
Confirm New Password
Submit
Copyright © 2024. All right reserved.

Driver Portal

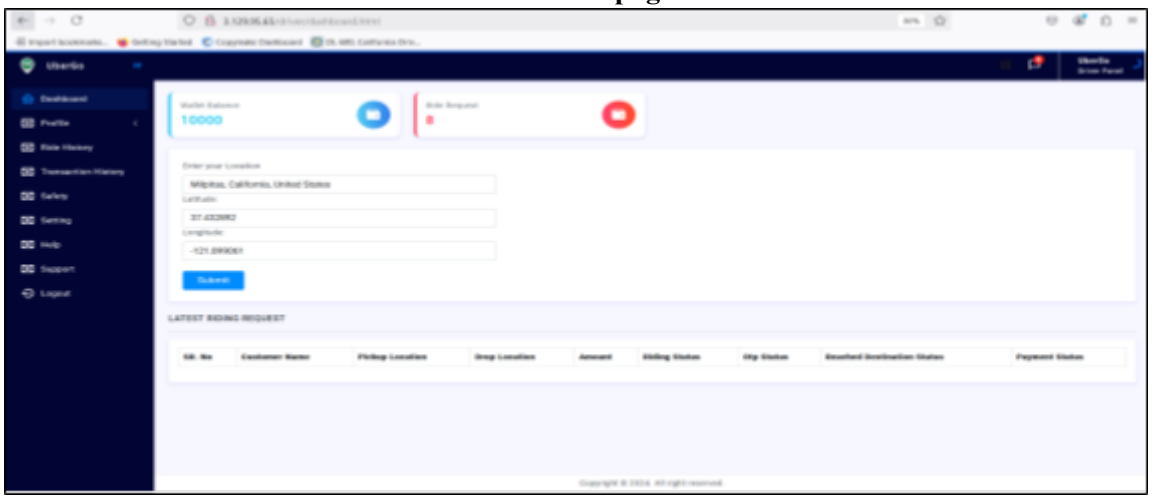
Register page:



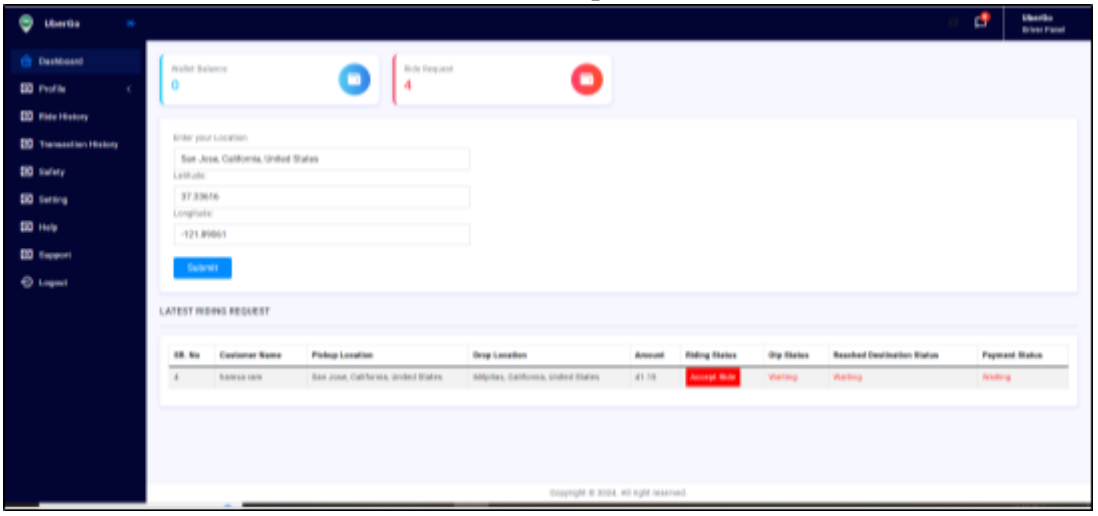
Login page:

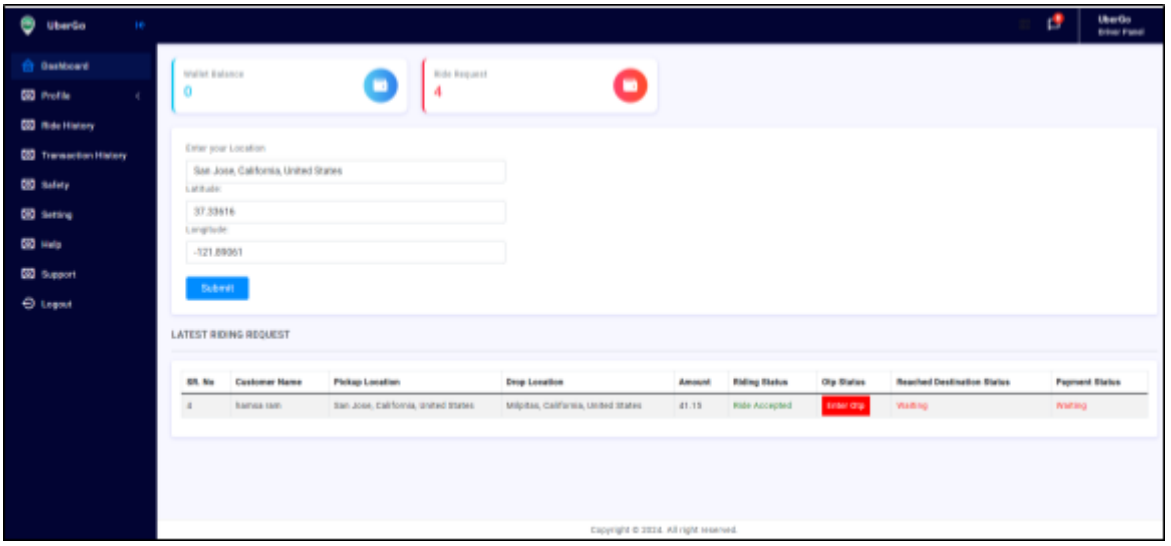


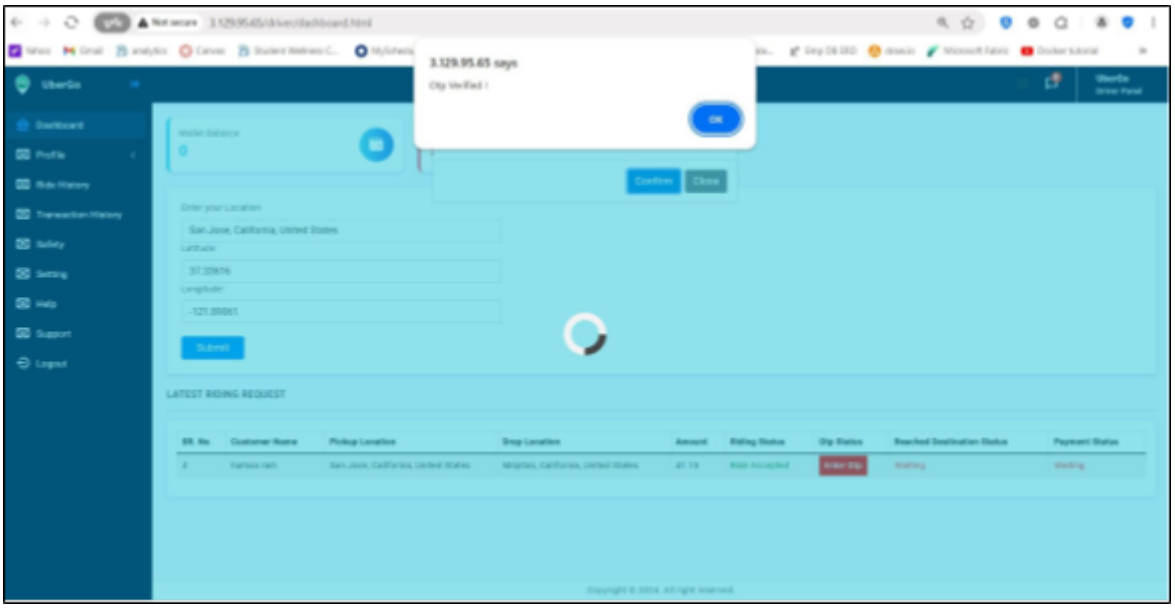
Dashboard page:

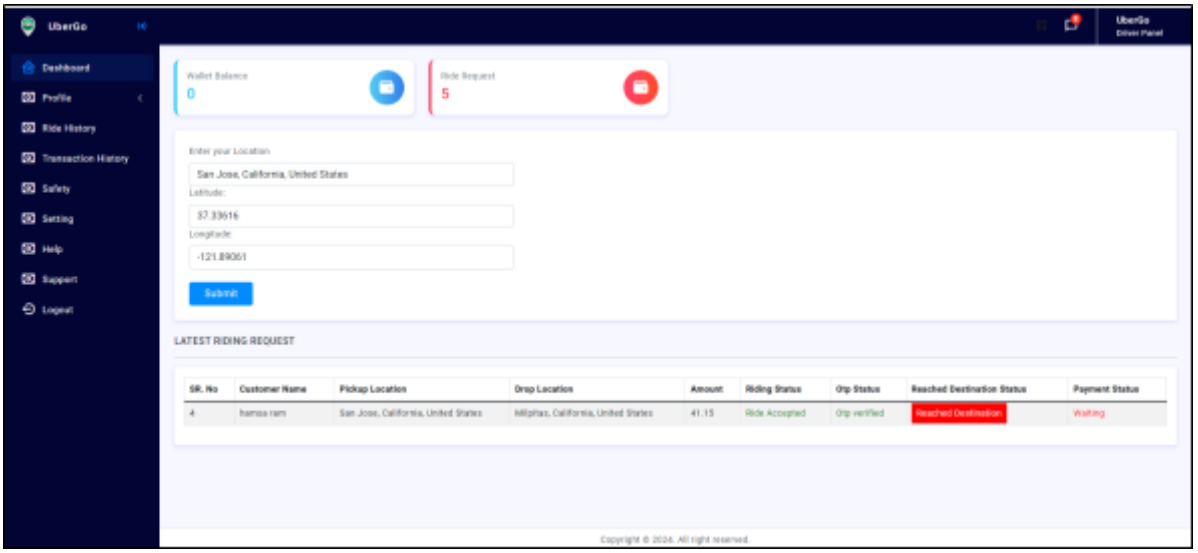


New ride request:

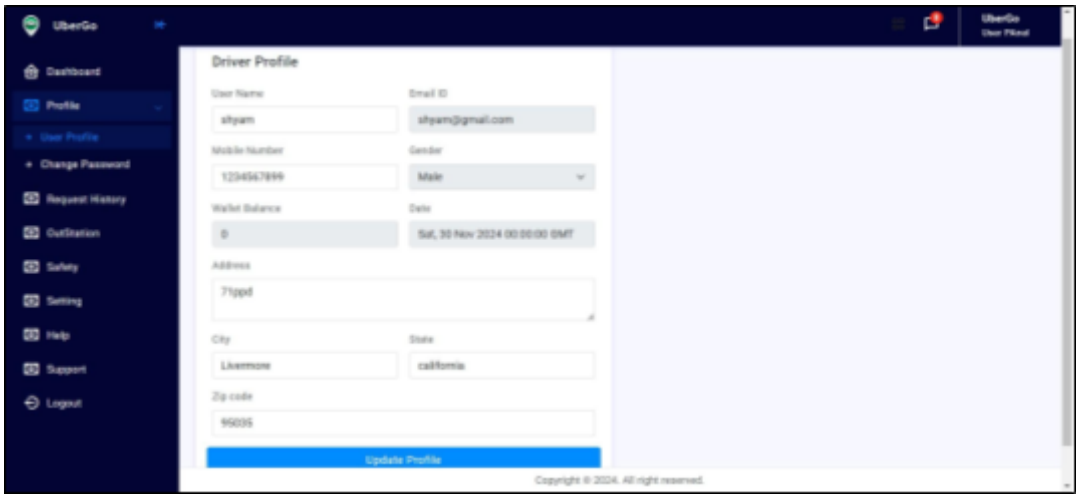




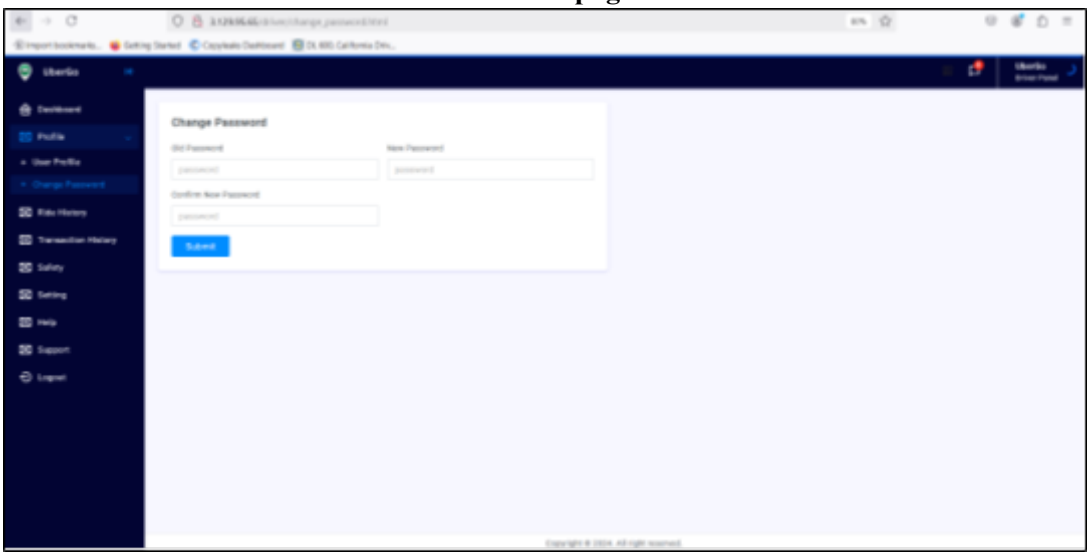




Profile page:

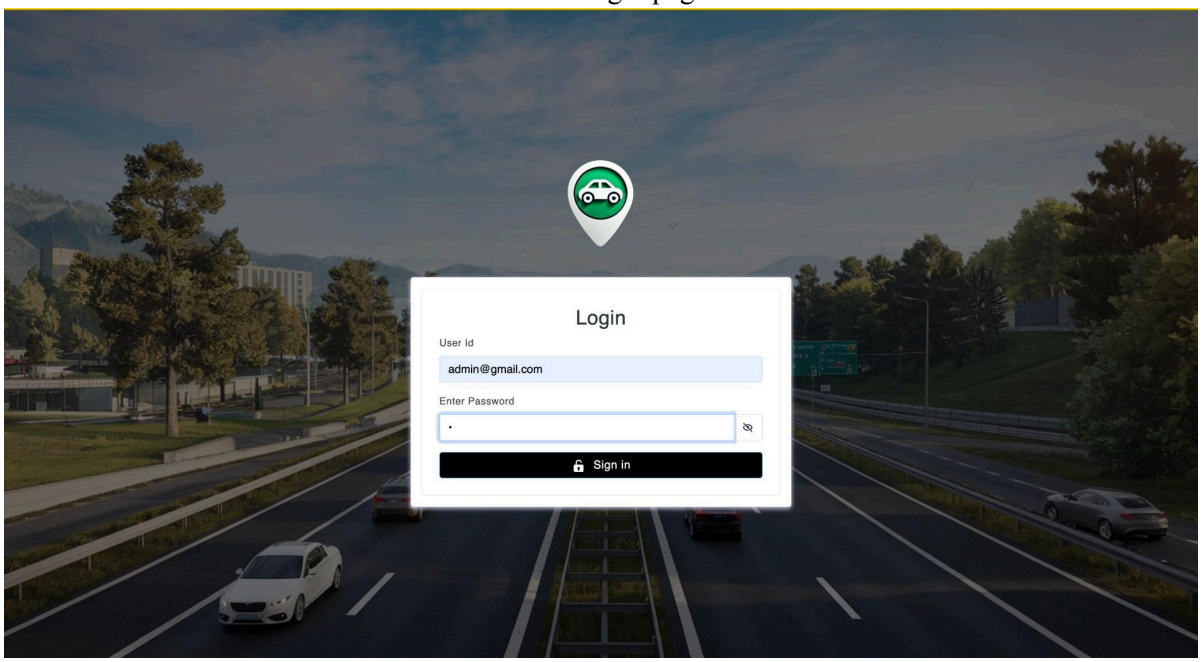


Password page:

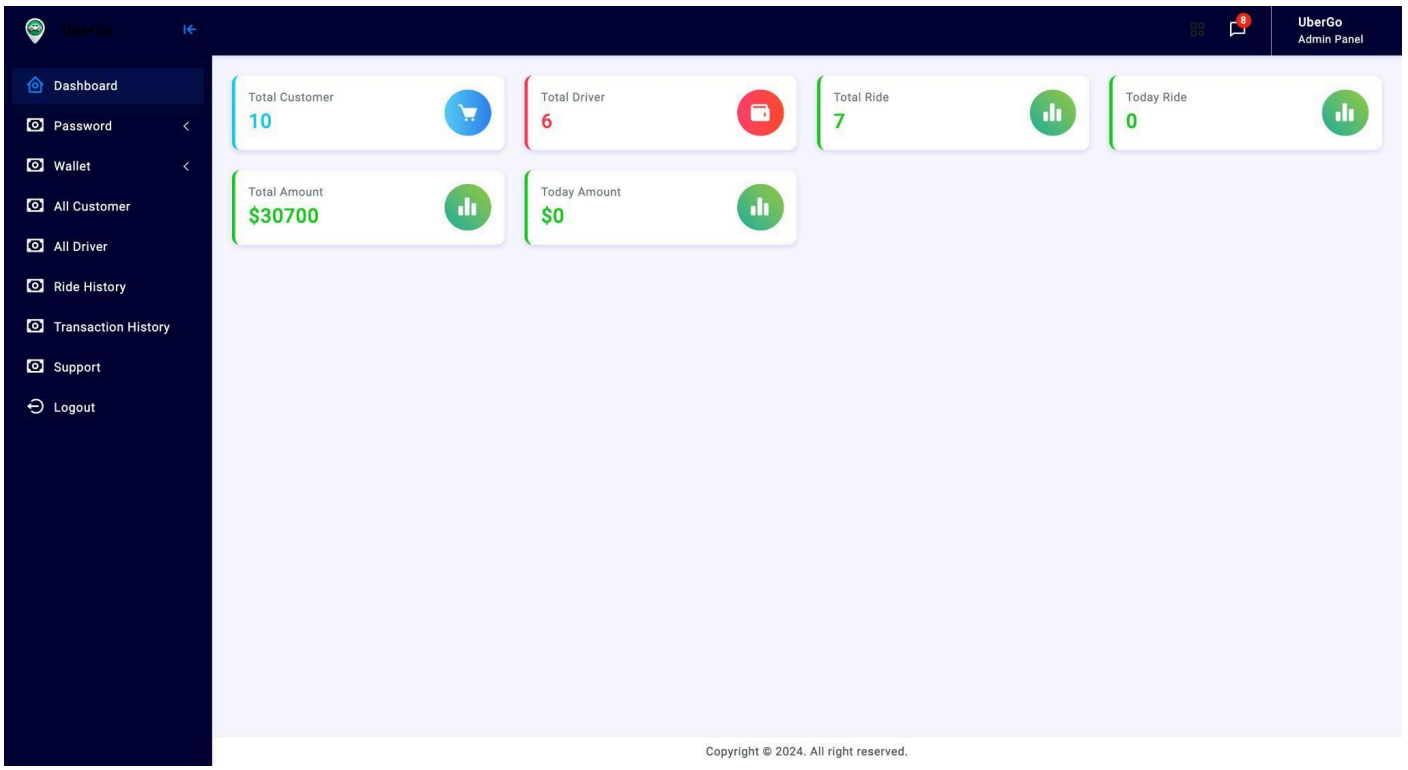


Admin Portal

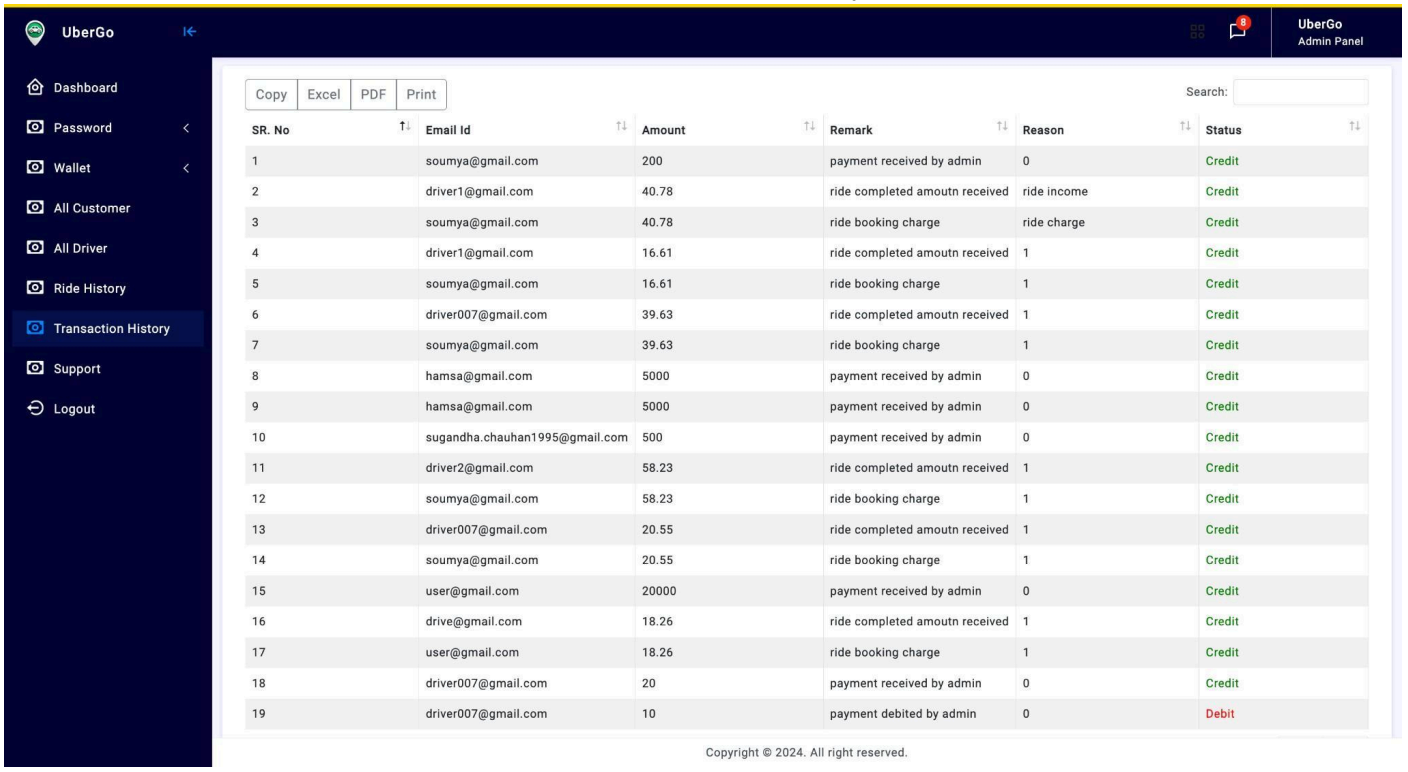
Admin login page:



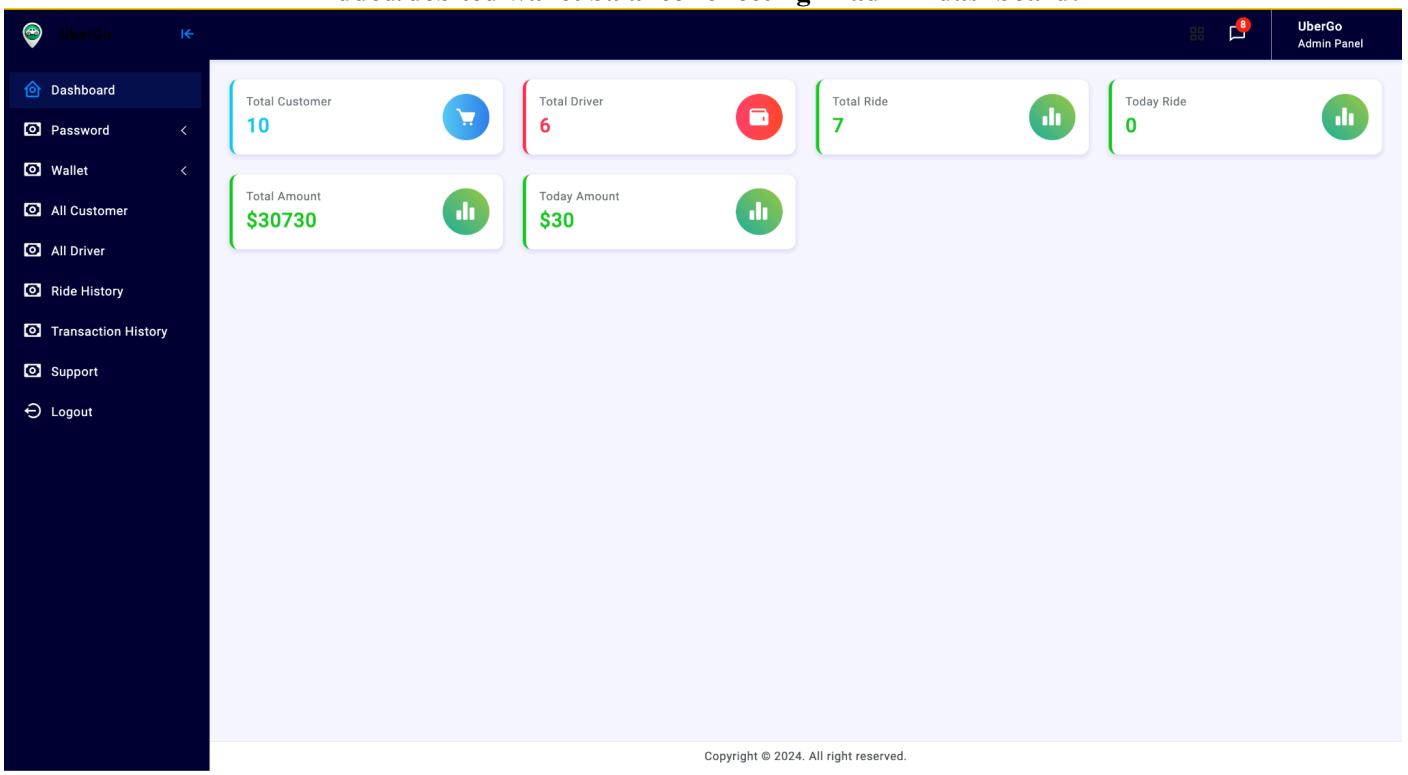
Admin Dashboard:



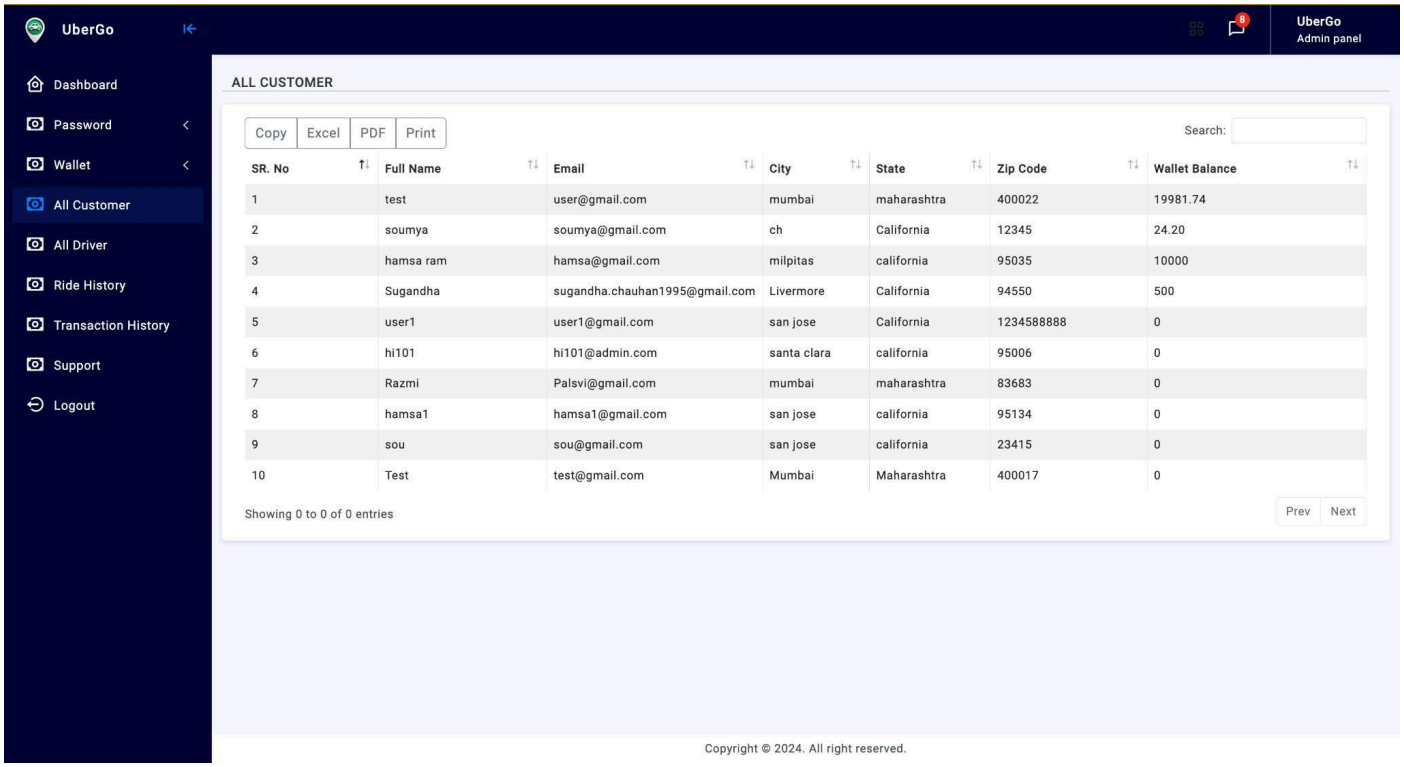
Admin Transaction History:



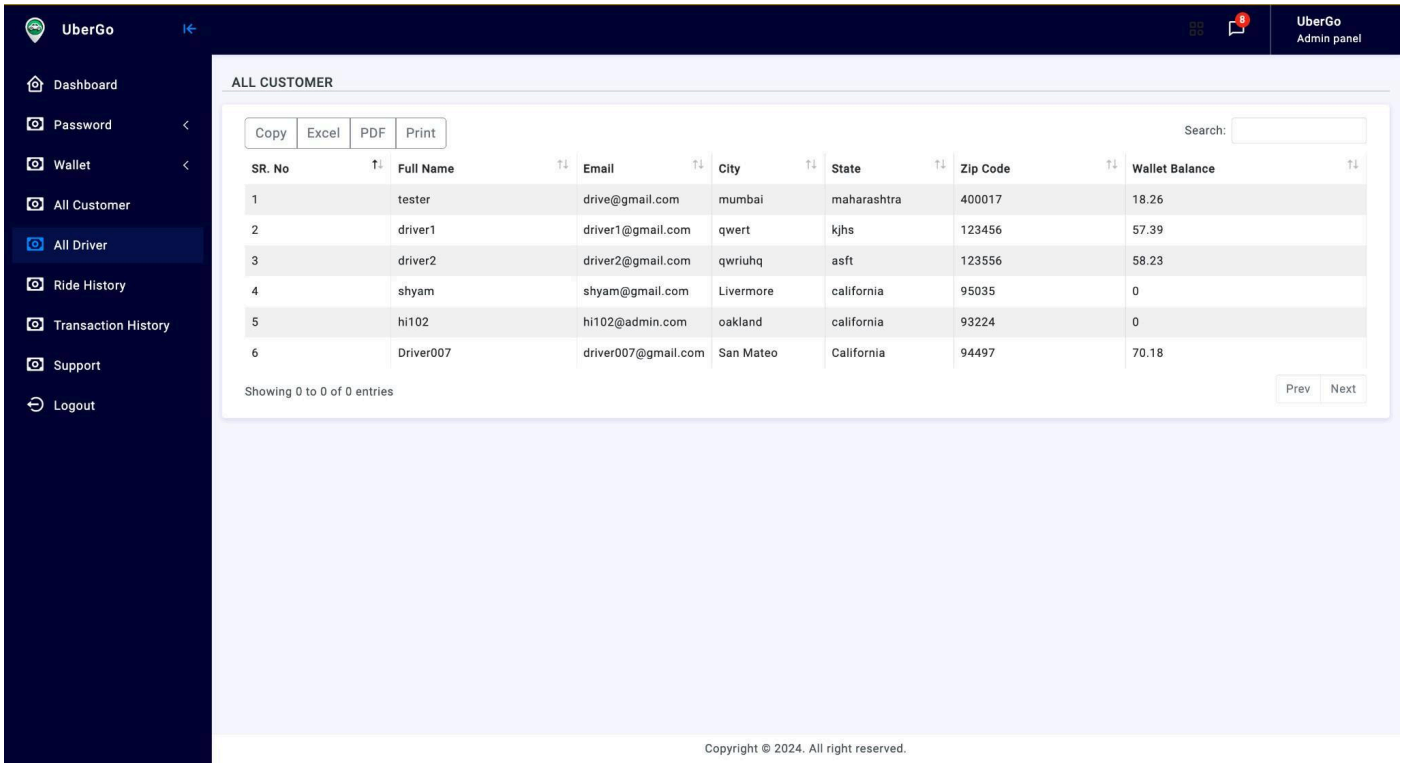
Added/debited wallet balance reflecting in admin dashboard:



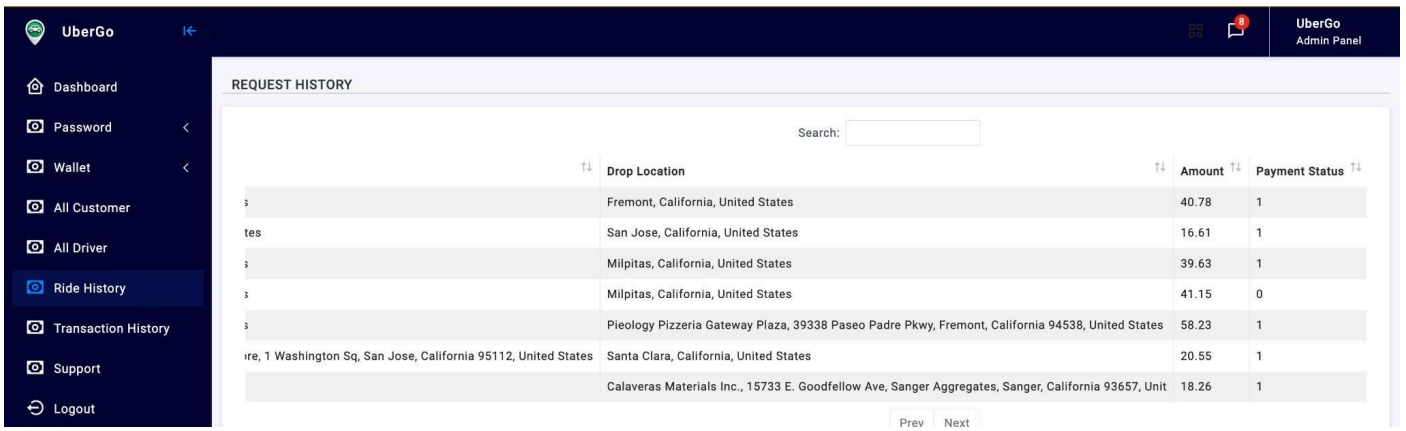
Customers list



Drivers list:



Ride History:



Code Snippets

Users Table

Insert User (Registration)

```python
@application.route('/register', methods=['POST'])
def register():
    try:
        # Get data from request
        data = request.get_json()
        username = data.get('full_name')
        emailid = data.get('emailid')
        mobile = data.get('mobile')
        gender = data.get('gender')
        password = data.get('password')
        driverstatus = data.get('driverstatus')
        address = data.get('address')
        city = data.get('city')
        state = data.get('state')
        zip = data.get('zip')

        if not username or not password:
            return jsonify({'status':'Failed','message': 'Missing Perameters'}), 400

        userid = 'CSH' + str(random.randint(10000, 99999))
        # Create a cursor and insert user into the database
        conn = create_connection()
        cursor = conn.cursor()
        cursor.execute("SELECT email_id FROM users WHERE email_id=%s OR mobile_no =%s",(emailid,mobile))
        check = cursor.fetchone()
        if not check:
            cursor.execute("INSERT INTO `users`(`full_name`, `email_id`, `user_id`, `mobile_no`, `gender`, `blocking_status`, `wall
        (username, emailid,userid,mobile,gender,'0','0',date.today(),password,driverstatus,address,city,state,zip))
            conn.commit()

            cursor.close()
            conn.close()
        else:
            return jsonify({'status':'Failed','message': 'User Already Registered'}), 400

        return jsonify({'status':'Success','message': 'User registered successfully!'}), 201
```

Update User Profile

```python
@application.route('/update_profile', methods=['POST'])
def updateprofile():
    try:
        # Get data from request
        data = request.get_json()
        uuid = data.get('uuid')
        name = data.get('name','')
        mobile = data.get('mobile','')
        address = data.get('address','')
        city = data.get('city','')
        state = data.get('state','')
        zip = data.get('zip','')
        gender = data.get('gender','')
        userid = getuseridbyuuid(uuid)
        if checkusersession(uuid):
            if not uuid or not name or not address or not mobile or not gender or not state or not zip or not state or not city:
             return jsonify({'status':'Failed','message': 'Details are required!'}), 400


            # Create a cursor and insert user into the database
            conn = create_connection()
            cursor = conn.cursor()
            cursor.execute("SELECT user_id FROM users WHERE user_id=%s",(userid,))
            pas = cursor.fetchone()

            if pas[0] == userid:
                cursor.execute("UPDATE users SET full_name=%s,mobile_no=%s,gender=%s,address=%s,city=%s,state=%s,zip_code=%s WHERE
                conn.commit()

                cursor.close()
                conn.close()
                return jsonify({'status':'Success','message': 'Profile Updated Success'}), 201
            else:
                return jsonify({'status':'Failed','message': 'Invalid Session'}), 401
```

travel_history Table
Insert Ride Details

```python
@application.route('/inittravel', methods=['POST'])
def inittravels():
    try:
        # Get data from request
        data = request.get_json()
        uuid = data.get('uuid')
        pickup_location = data.get('pickup_location','')
        drop_location = data.get('drop_location','')
        pickup_lati = data.get('pickup_lati','')
        pickup_longi = data.get('pickup_longi','')
        drop_lati = data.get('drop_lati','')
        drop_longi = data.get('drop_longi','')
        amount = data.get('amount','')
        paying_mode = data.get('paying_mode','')

        if checkusersession(uuid):
            if not pickup_location:
                return jsonify({'status':'Failed','message': 'Details are required!'}), 400


            opt = str(random.randint(10000, 99999))
            # Create a cursor and insert user into the database
            conn = create_connection()
            cursor = conn.cursor()
            customer_id = getuseridbyuuid(uuid)
            cursor.execute("SELECT riding_status FROM travel_history WHERE riding_status !='4' AND customer_id=%s",(customer_id,))
            check = cursor.fetchone()
            if not check:
                    cursor.execute("SELECT wallet_balance FROM users WHERE  user_id=%s",(customer_id,))
                    blc = cursor.fetchone()
                    if float(blc[0]) >= float(amount):
                        cursor.execute("INSERT INTO travel_history (customer_id, customer_name, driver_id, pickup_location, drop_lo
                        ('',pickup_location,drop_location,'','',amount,'0',opt,pickup_lati,drop_lati,pickup_longi,drop_longi,paying

                        conn.commit()
                        cursor.close()
                        conn.close()
```

Fetch Ride History

```python
# Route for Ride History
@application.route('/ridehistoryuser', methods=['POST'])
def ridehistoryss():
    try:
            # Get data from request
            data = request.get_json()
            uuid = data.get('uuid')

            if not uuid:
                return jsonify({'status':'Failed','message': 'Session Id Invalid'}), 400

            # Create a cursor and check the user in the database
            if checkusersession(uuid):
                conn = create_connection()
                userid = getuseridbyuuid(uuid)
                cursor = conn.cursor()
                cursor.execute("SELECT * FROM `travel_history` WHERE customer_id =%s",(userid,))
                rows = cursor.fetchall()

                cursor.close()
                conn.close()

                if rows:
                    column_names = [desc[0] for desc in cursor.description]
                    users = [dict(zip(column_names, row)) for row in rows]
                    return jsonify(users), 200
                else:
                    return jsonify({'status':'Failed','message': 'No Data Found'}), 400
            else:
                return jsonify({'status':'Failed','message': 'Invalid session'}), 401

    except Exception as e:
        return jsonify({'status':'Failed','message': 'We Are Facing Some Technical Issue Please try Again '+str(e)}), 500
```

driver Table
Insert Driver Details

```python
# Route for add driver
@application.route('/adddriver', methods=['POST'])
def adddrivers():
    try:
        # Get data from request
        data = request.get_json()
        uuid = data.get('uuid')
        driver_id = getuseridbyuuid(uuid)
        vehicle_model = data.get('vehicle_model')
        vehicle_no = data.get('vehicle_no')
        image = data.get('image')


        if checkusersession(uuid):

            if not vehicle_model or not vehicle_no or not image:
                return jsonify({'status':'Failed','message': 'Details are required!'}), 400

            # Create a cursor and insert user into the database
            conn = create_connection()
            cursor = conn.cursor()
            cursor.execute("SELECT u.driver_status, d.driver_id FROM users AS u LEFT JOIN driver AS d ON u.user_id = d.driver_id WH
            isdriver = cursor.fetchone()
            if isdriver[0] == '1':
                if isdriver[1] is None:
                    cursor.execute("INSERT INTO `driver`(`driver_id`, `full_name`, `address`, `city`, `state`, `zip_code`, `mobile_
                    conn.commit()

                    cursor.close()
                    conn.close()
```

transaction_history Table
Add Transaction

```python
# Route After Payment
@application.route('/afterpayment', methods=['POST'])
def afterpayment():
    try:
        # Get data from request
        data = request.get_json()
        uuid = data.get('uuid')
        id = data.get('id','')
        amount = data.get('amount','')

        if checkusersession(uuid):
            if not uuid and not id or not amount:
                return jsonify({'status':'Failed','message': 'Details are required!'}), 400

            conn = create_connection()
            cursor = conn.cursor()
            useri = getuseridbyuuid(uuid)
            cursor.execute("SELECT wallet_balance FROM users WHERE user_id=%s", (useri,))
            blc = cursor.fetchone()
            if blc:
                if Decimal(blc[0]) >=Decimal(amount):
                    newblc = Decimal(blc[0]) - Decimal(amount)
                    cursor.execute("SELECT driver_id FROM travel_history WHERE id=%s", (id,))
                    driver = cursor.fetchone()
                    if driver:
                        driver_id = driver[0]
                        cursor.execute("SELECT wallet_balance FROM users WHERE user_id=%s", (driver_id,))
                        dblc = cursor.fetchone()
                        if dblc:
                            newdblc = Decimal(dblc[0])+Decimal(amount)
                            cursor.execute("UPDATE travel_history SET payment_status=%s,riding_status=%s WHERE id=%s ;", ('1','4',i
                            conn.commit()
                            cursor.execute("UPDATE users SET wallet_balance=%s WHERE user_id=%s ;", (newblc,useri))
                            conn.commit()
                            cursor.execute("UPDATE users SET wallet_balance=%s WHERE user_id=%s ;", (newdblc,driver_id))
                            conn.commit()
```

Code Listing: Server Implementation for the Session Object: The session object is responsible for managing user sessions, including login, logout, and session validation. In your project, it likely interacts with Redis for caching session data and MySQL for persistent storage.

Redis:

```python
# checking for usersession with caching
def checkusersession(uuid):
    try:
        # Check Redis cache first
        cached_status = redis_cache.get(f"session:{uuid}")
        if cached_status:
            return cached_status == 'verified'

        # Fallback to database query
        conn = create_connection()
        cursor = conn.cursor()
        cursor.execute("SELECT status FROM ods WHERE uuid = %s", (uuid,))
        user = cursor.fetchone()
        cursor.close()
        conn.close()

        if user:
            status = user[0]
            # Cache the result in Redis for 5 min
            redis_cache.setex(f"session:{uuid}", 300, status)
            return status == 'verified'
        else:
            return False
    except Exception as e:
        print(f"Error in checkusersession: {e}")
        return False
```

```python
# Function to get user ID by UUID (with caching)
def getuseridbyuuid(uuid):
    try:
        # Check Redis cache first
        cached_userid = redis_cache.get(f"user_id:{uuid}")
        if cached_userid:
            return cached_userid

        # Fallback to database query
        conn = create_connection()
        cursor = conn.cursor()
        cursor.execute("SELECT user_id FROM ods WHERE uuid = %s", (uuid,))
        user = cursor.fetchone()
        cursor.close()
        conn.close()

        if user:
            user_id = user[0]
            # Cache the user ID in Redis for 5 min
            redis_cache.setex(f"user_id:{uuid}", 300, user_id)
            return user_id
        else:
            return None
    except Exception as e:
        print(f"Error in getuseridbyuuid: {e}")
        return None
```

```python
@application.route('/gettravelhistory', methods=['POST'])
def gettravelhistory():
    try:
        # Get data from request
        data = request.get_json()
        uuid = data.get('uuid')

        if not uuid:
            return jsonify({'status': 'Failed', 'message': 'Session Id Invalid'}), 400

        # Check session validity
        if checkusersession(uuid):
            # Check Redis cache for travel history
            cached_travel_history = redis_cache.get(f"travel_history:{uuid}")

            if cached_travel_history:
                # Return cached data if available
                return jsonify({'status': 'Success', 'message': 'Fetch Success', 'travel_history': eval(cached_travel_history)}),

            # Fallback to database query if not in cache
            conn = create_connection()
            cursor = conn.cursor()
            cursor.execute(
                "SELECT u.* FROM travel_history u JOIN ods o ON u.customer_id = o.user_id WHERE o.uuid = %s AND o.status = 'verifie
                (uuid,)
            )
            user = cursor.fetchone()

            cursor.close()
            conn.close()
```

```python
            # Cache the result in Redis for 5 minutes
            redis_cache.setex(f"travel_history:{uuid}", 300, str(travel_history))

            # Return the result
            return jsonify({'status': 'Success', 'message': 'Fetch Success', 'travel_history': travel_history}), 200
        else:
            return jsonify({'status': 'Failed', 'message': 'No Data Found'}), 400
    else:
        return jsonify({'status': 'Failed', 'message': 'Invalid session'}), 401

except Exception as e:
    return jsonify({'status': 'Failed', 'message': f'We Are Facing Some Technical Issue Please try Again {e}'}), 500
```

Main Server Code Listing: The main server code is the entry point for your backend application. It initializes the Flask app, sets up routes, configures middleware, and starts the server.

backend > application.py > ...

```python
from flask import Flask, request, jsonify
from flaskext.mysql import MySQL
from redis import StrictRedis
from flask_cors import CORS
from datetime import date,datetime
```

```python
cal = USFederalHolidayCalendar()

application = Flask(__name__)
CORS(application)

# Redis configuration
redis_cache = StrictRedis(host='localhost', port=6379, db=0, decode_responses=True)
```

```python
DB_CONFIG = {
    "host": os.getenv("DB_HOST", "localhost"),
    "user": os.getenv("DB_USER", "root"),
    "password": os.getenv("DB_PASSWORD", ""),
    "database": os.getenv("DB_NAME", "ubergo_db")
}

# Function to create a database connection
def create_connection():
    try:
        connection = mysql.connector.connect(**DB_CONFIG)
        if connection.is_connected():
            return connection
    except Error as e:
        print("Error in create connection")
        print(f"Error: {e}")
        return None
```

```python
@application.route('/test_connection')
def test_connection():
    try:
        # Establish the connection
        conn =  create_connection()
        cursor = conn.cursor()

        # Optional: Execute a simple query
        cursor.execute("SELECT DATABASE();")  # Get the current database
        current_db = cursor.fetchone()

        # Clean up
        cursor.close()
        conn.close()

        return jsonify({'status': 'Success', 'message': f'Connected to database {current_db[0]}'}), 200
    except Exception as e:
        return jsonify({'status': 'Failed', 'message': str(e)}), 500
```

```python
if __name__ == "__main__":
    application.run(port=5000, debug=True)
```

Code Listing: Database Access Class: The database access class is responsible for interacting with the database. It abstracts the database queries and connections, making it easier to perform CRUD (Create, Read, Update, Delete) operations and maintain code modularity.

```python
DB_CONFIG = {
    "host": os.getenv("DB_HOST", "localhost"),
    "user": os.getenv("DB_USER", "root"),
    "password": os.getenv("DB_PASSWORD", ""),
    "database": os.getenv("DB_NAME", "ubergo_db")
}
```

```python
# Function to create a database connection
def create_connection():
    try:
        connection = mysql.connector.connect(**DB_CONFIG)
        if connection.is_connected():
            return connection
    except Error as e:
        print("Error in create connection")
        print(f"Error: {e}")
        return None
```

Appendix

System Architecture Diagram

- User Layer (Frontend)
- Backend Layer (Application Logic)
- Database Layer
- Hosting and Deployment (AWS)
- Workflow

Uber Application Design and Implementation

Frontend

- Admin Module
- Driver Module
- User Module

Backend

- Handling Heavy Weight Resources
- API Development
- Dynamic Pricing Algorithm (Lightweight Gradient Boosting Mechanism Regressor)
- Error Handling
- Driver Proximity Logic
- Caching Logic in Login and Travel History
- Database Design and Object Management

Database Design and Object Management

- Visual Diagram
- Tables and their relationships
- Code snippets for Database Design Implementation
- Code listing of your server implementations for the entity objects
- Other code listing (utility classes, etc)
- Policies used to decide when to write data to the database

Load Testing, Stability and Scalability (Apache JMeter graphs)

Kafka Implementation (Limitations and Future Work)

Observations and Conclusion

Application Screenshots

- User Portal
- Driver Portal
- Admin Portal