Problem Statement:

Building a Ride-Sharing Platform with User and Admin Functionalities

Requirements:

3 types of Users:

1) Traveler
2) Traveler companion
3) Admin

Traveler:

Assuming, The one who has a vehicle to share with others(Traveler companion).
This type of user must have these requirements:

- Ability to create a Trip(basically a Ride) and share the details with others(Traveler companion)
- Can able to see the History of Trips he Created

Traveler Companion:

Assuming, The one who wants to be part of others(Traveler) Trip
This type of user must have these requirements:

- Ability to see available Trips
- To be a select one among available Trips and be a part of it
- Should get the nearby notification when the cab hits the geofence of the traveler drop.
- Share the feedback on the experience.

Admin:

This type of user must have these requirements:

- Admins have access to view all the rides shared by users.
- The Admin should be able to access the overall experience feedback of the users.

These are the requirements of a System.

To Solve the above Problem Statement fulfilling the above requirements.

Proposed Solution:

User:

```
class User {
protected:
    string userId;
    string name;
    string phoneNumber;
public:
    User(const string& name, const string& phoneNumber);
    virtual ~User() = default;

    string getName() const;
    string getPhoneNumber() const;
    string getUserId() const;
};
```

Contains Basic Attributes like Name, PhoneNumber.
And Getter Functions.

Traveler:

```
class Traveler : public User
{
    string carNumber;
    vector<string> tripHistory;
    public:
    Traveler(string name, string phoneNumber, string carNumber) : User(name, phoneNumber) {
        this->carNumber = carNumber;
    }
    string getCarNumber();
    void addToTripHistory(string tripId);
};
```

Inherited from User contains Extra Things like CarNumber, TripHistory

Traveler Companion:

```
class TravelerCompanion : public User
{
    vector<string> sharedRides;
    public:
```

```
    TravelerCompanion(string name, string phoneNumber) : User(name,
phoneNumber) {};
    void addToSharedRides(string tripId)
    {
        sharedRides.push_back(tripId);
    }
};
```

Trip:

```
class Trip
{
    string tripId;
    Traveler* traveler;
    vector<TravelerCompanion*> travelerCcompanions;
    TRIP_STATUS status;
    Location* startLocation;
    Location* endLocation;
    time_t startTime;
    vector<TripEvent*> events;
    vector<Feedback*> feedbacks;
    public:
    Trip(Traveler* driver, Location* startLoc, Location* endLoc);
    void addTravelerCompanion(TravelerCompanion* travelerCompanion);
    void setTripStatus(TRIP_STATUS status);
    void getTripDetails();
    void addEvent(TripEvent* event);
    void displayEvents();
    void displayFeedbacks();
    void addFeedback(Feedback* feedback);
};
```

Each Trip contains these attributes,

- **TripId:** Unique Identifier of type string for every Trip Object.
- **traveler**: The one who wants to share the car type Traveler*.
- **TravelerCompanions:** the group of travelerCompanions who are part of this trip stored as vector of TravelerCompanions*.
- **startTime:** the time the trip started.
- **Feedbacks:** The feedback of users who are part of this trip.
- **Events:** the events related to trip like
    - When the Trip Created
    - When the Trip Strarted

- When the Trip Completed
- Stored in form of vector of Events*

Status:

```
enum class TRIP_STATUS {
    CRAETED,
    IN_PROCESS,
    COMPLETED,
    CANCELLED
};
```

Different type of status possible for a trip.

- **Start and End Location:** the start and ending point proposed for a trip each of type Location

Location:

```
class Location {
    double latitude;
    double longitude;
public:
    Location(double pLat, double pLong) : latitude(pLat), longitude(pLong) {}
    double getLatitude() {
        return latitude;
    }
    double getLongitude() {
        return longitude;
    }
};
```

Methods:

```
Trip(Traveler* driver, Location* startLoc, Location* endLoc);
```

Constructor of a Trip.
The Following Methods are self-explanatory.

```
void addTravelerCompanion(TravelerCompanion*travelerCompanion);
void setTripStatus(TRIP_STATUS status);
void getTripDetails();
void addEvent(TripEvent* event);
void displayFeedbacks();
```

Feedback:

This is Feedback class,
Contains two attributes the user who wrote the review and message.
Methods: Constructor And display function to display feedback.

The advantage of writing separate feedback classes is we can add additional items like ratings, points of improvement, things you like the most without affecting (modifying) other class.

```
class Feedback
{

    User* user;
    string message;
    public:
    Feedback(User* fuser, string fMessage) :
user(fuser),message(fMessage) {};
    void displayFeedback();
};
```

TripEvent:

This is the TripEvent class,
Contains two attributes timestamp and message of an event like created, completed….
Methods: Constructor And display function to display Events.

The advantage of writing separate feedback classes is we can add additional items without affecting (modifying) other class.

```
class TripEvent
{
    time_t timeStamp;
    string message;
    public:
    TripEvent(string eMessage);
    void displayEvent();
};
```

Trip Manager:

```
class TripManager
{
    static TripManager* tripManagerInstance;
    static mutex mtx;
    NotificationMgr* notificationMgr;
    unordered_map<string, Trip*> notInProgressTrips;
    unordered_map<string, Trip*> inProgressTrips;
    TripManager() {
        NotificationMgr* notificationMgr =
NotificationMgr::getNotificationMgr();
    };
    void addUserForNotificationUpdates(string tripId, User* user);
    public:
    static TripManager* getTripManagerInstance();
    string createTrip(Traveler* travler, Location* startLoc,Location*
endLoc);
    void startTrip(string tripId);
    void addTravelerCompanion(string tripId, TravelerCompanion*
travelerCompanion);
    void completeTrip(string tripId);
    void notifyUpdatesToTravelerCompanions(string tripId,string message);
    Trip* getTrip(string tripId);
    vector<string> getAvailableTrips();
    vector<Trip*> getTrips();
    void addFeedbackToTrip(string tripId,User* user, string message);
    void showFeedbackToATrip(string tripId);
};
```

The most important class like a system:
User Interact with this to create, start, book, write feedback…
This is based on the **Singleton Design Pattern.**
Only one instance of TripManager is created for the whole project.
Attributes:

- **tripManagerInstance:** the pointer to a single object of TripManager.
- **Mtx:** the mutex used to ensure only one object is created of type TripManager even in multithreading envinormnet.
- **inProgressTrips:** map of tripId of type string as key and pointer to Trip as value. For the trips in progress.

- **notInProgressTrips:** map of tripId of type string as key and pointer to Trip as value. For the trips with are not in progress.
- **notificationMgr:** For handling Notification Service (discussed later).

Functions:

```
TripManager() {
        NotificationMgr* notificationMgr =
NotificationMgr::getNotificationMgr();
    };
```

The constructor of the TripManager class gets the notification manager instance.

```
static TripManager* getTripManagerInstance();
```

```
TripManager* TripManager::getTripManagerInstance()
{
    if(tripManagerInstance == nullptr)
    {
        mtx.lock();
        if(tripManagerInstance == nullptr) {
            tripManagerInstance = new TripManager();
        }
        mtx.unlock();
    }
    return tripManagerInstance;
}
```

Mtx ensures only once instance of TripManager is created.

```
string createTrip(Traveler* traveler, Location* startLoc,Location*
endLoc);
```

Travelers use this method to create a trip
Upon calling

- A new trip object gets created
- Unique tripid is created for a trip
- tripid along with a pointer to created a Trip is added to inProgressTrips
- The trip status is set to Created
- Event of Trip is Strated to event of trip

```
vector<string> getAvailableTrips();
```

TravellerComapnions use this method to get all the Avaliable trips
Upon calling

- All the trips in the inProgressTrips will be sent to TravellerComapnion

```
void addTravelerCompanion(string tripId, TravelerCompanion* travelerCompanion);
```

From the Available trips TravellerComapnion will choose one that is more suitable for them. And use to method to book the trip.
Upon calling,

- TravellerComapnion will be added to participants of the trip
- And TravellerComapnion will be added for Notification service of there preference.

```
void startTrip(string tripId);
```

Traveler use this method to start the trip
Upon calling,

- Trip Status will be changed
- Event will be added to EventLog

```
void notifyUpdatesToTravelerCompanions(string tripId,string message);
```

Traveler use this method to give updates to participates of the trip
Upon calling,

- Users will be Notified via Notification service.
- The event will be added to EventLog

```
void completeTrip(string tripId);
```

Travelers use this method to complete the trip
Upon calling,

- Trip Status will be changed
- The event will be added to EventLog
- Users will be notified via Notification service.
- The trip will be removed from the inProgressTrips map
- The trip will be added to the notInProgressTrips map

```
vector<Trip*> getTrips();
void addFeedbackToTrip(string tripId,User* user, string message);
void showFeedbackToATrip(string tripId);
```

Above methods are Self Explanatory

Notification Service:

This is Design using Observer Design Pattern.

```
class INotificationSender {
public:
    virtual void sendNotification(string userId, string message) = 0;
    virtual ~INotificationSender() = default;
};
```

This an Interface contains SendNotification as a method
In this Appliaction SMSNotificationSender and WhatsAppNotificationSender are Implemented it.
Advantage of this is we can able as many as other type of NotificationSenders without changing code logic runTime Polymorphism in Place.

```
class SMSNotificationSender : public INotificationSender {
public:
    void sendNotification(std::string userId, std::string message) override;
};
```

```
class WhatsAppNotificationSender : public INotificationSender {
public:
    void sendNotification(string userId, string message) {
        cout << "whatsapp Notification for "<< userId <<" is " << message << endl;
    }
};
```

Notification Manager:

```
class NotificationMgr {
    static NotificationMgr* notificationMgrInstance;
    static mutex mtx;
    unordered_map<string, vector<pair<string,INotificationSender*>>>
notificationSendersMap;
    NotificationMgr() {}
public:

    ~NotificationMgr();
    static NotificationMgr* getNotificationMgr();
```

```
    void addNotificationSender(string tripId, string userId,
INotificationSender* notificationSender);
    void removeNotificationSender(string tripId, string userId,
INotificationSender* notificationSender);
    void notify(string tripId, string message);
    void notifyParticularUser(string pUserId, string pMsg,
INotificationSender* sender);
};
```

Singleton Design is in place.

```
unordered_map<string, vector<pair<string,INotificationSender*>>>
notificationSendersMap;
```

This Contains for a tripId the observer and there preference of NotificationSender.

Working: Whenever we call the

```
void addUserForNotificationUpdates(string tripId, User* user);
```

In TripManager, the user is added to a particular TripId with this Preference of Sender.
Similarly for removing

```
void removeNotificationSender(string tripId, string userId,
INotificationSender* notificationSender);
```

```
void notify(string tripId, string message);
```

When some info like trip completed or updates to participates of trip.
This notify will send notifications to all members in the tripId.

Overview of UI-Based Implementation:

Implementation in SQLite:

```
class User(db.Model):
    id = db.Column(db.Integer, primary_key=True)
    username = db.Column(db.String(100), unique=True, nullable=False)
    email = db.Column(db.String(100), unique=True, nullable=False)
    password_hash = db.Column(db.String(128), nullable=False)
    role = db.Column(db.String(50), nullable=False)
    # Other user fields...

class Trip(db.Model):
    id = db.Column(db.Integer, primary_key=True)
    trip_id = db.Column(db.String(100), unique=True, nullable=False)
    traveler_id = db.Column(db.Integer, db.ForeignKey('user.id'),
nullable=False)
    driver_name = db.Column(db.String(100), nullable=False)
    driver_phone = db.Column(db.String(100), nullable=False)
    cab_number = db.Column(db.String(100), nullable=False)
    start_time = db.Column(db.DateTime, nullable=False)
    end_time = db.Column(db.DateTime)
    status = db.Column(db.String(50), nullable=False)
    companions = db.relationship('User', secondary='trip_companions',
backref='trips')

class TripCompanions(db.Model):
    id = db.Column(db.Integer, primary_key=True)
    trip_id = db.Column(db.Integer, db.ForeignKey('trip.id'),
nullable=False)
    companion_id = db.Column(db.Integer, db.ForeignKey('user.id'),
nullable=False)

class Feedback(db.Model):
    id = db.Column(db.Integer, primary_key=True)
    trip_id = db.Column(db.Integer, db.ForeignKey('trip.id'),
nullable=False)
    companion_id = db.Column(db.Integer, db.ForeignKey('user.id'),
nullable=False)
    feedback = db.Column(db.Text, nullable=False)
    timestamp = db.Column(db.DateTime, default=datetime.utcnow)

class RideRequest(db.Model):
```

```
    id = db.Column(db.Integer, primary_key=True)
    trip_id = db.Column(db.Integer, db.ForeignKey('trip.id'),
nullable=False)
    companion_id = db.Column(db.Integer, db.ForeignKey('user.id'),
nullable=False)
    status = db.Column(db.String(50), nullable=False,
default='pending')  # pending, accepted, rejected
    timestamp = db.Column(db.DateTime, default=datetime.utcnow)
```

End Points needed for this project:

Signup Endpoint

```
@auth.route('/signup', methods=['GET', 'POST'])
```

- **Purpose**: Allows new users to sign up for an account.
- **GET Request**: Displays the signup form.
- **POST Request**: Processes the signup form. Creates a new user with the provided details (username, email, password, role), hashes the password for security, adds the user to the database, logs them in, and redirects to the home page.

Login Endpoint

```
@auth.route('/login', methods=['GET', 'POST'])
```

- **Purpose**: Allows users to log in to their accounts.
- **GET Request**: Displays the login form.
- **POST Request**: Processes the login form. Checks the provided email and password against the database. If successful, log in to the user and redirect to the home page. Otherwise, it flashes an error message.

Logout Endpoint

```
@auth.route('/logout')
```

- **Purpose**: Log out the currently logged-in user.
- **GET Request**: Logs out the user and redirects them to the login page.

Index/Home Endpoint

```
@main.route('/')
```

- **Purpose**: Displays the home page with a list of trips.
- **GET Request**:
    - For travelers: Shows all trips created by the logged-in traveler.
    - For admins: Shows all trips in the system and the feedback
    - For companions: Users past trips, and trips shared by the travellers.
- **Login Required**: Only accessible to logged-in users.

Create Trip Endpoint

```
@main.route('/create_trip', methods=['POST'])
```

- **Purpose**: Allows travelers to create a shared ride
- **POST Request**:
    - Creates a new trip with the details provided in the form (trip ID, driver name, driver phone, cab number).
    - Adds the trip to the database and commits the transaction.
- **Role-Based Access**: Only accessible to users with the traveler role.

Search rides Endpoint

```
@main.route('/search_rides', methods=['GET'])
```

- **Purpose**: Allows traveler companions to search for the rides
- **GET Request**:
    - Gets all the rides matching the user's destination.
- **Role-Based Access**: Only accessible to users with the traveler companion role.

Request ride Endpoint

```
@main.route('/request_ride', methods=['POST'])
```

- **Purpose**: Allows traveler companions to request traveler for the ride
- **POST Request**:
    - Posts the data to the Ride Request table and notifies the traveler.
- **Role-Based Access**: Only accessible to users with the traveler companion role.

Accept companion Endpoint

```
@main.route('/accept_ride', methods=['POST'])
```

- **Purpose**: Allows travelers to accept a companion for sharing the ride
- **POST Request**:
    - Approve the request of the companion to join the ride and notify them.
- **Role-Based Access**: Only accessible to users with the traveler role.

Share ride details Endpoint

```
@main.route('/share_ride_details', methods=['GET'])
```

- **Purpose**: Allows travelers to share the details of the ride via WhatsApp etc
- **GET Request**:
    - Gets all the ride details in the form of text which can be shared with the companion
- **Role-Based Access**: Only accessible to users with the traveler role.

Track ride Endpoint

```
@main.route('/track_ride', methods=['GET'])
```

- **Purpose**: Allows companion to track his ride
- **GET Request**:
    - Gets all the details of the current ride
- **Role-Based Access**: Only accessible to users with the Admin role.

Complete Trip Endpoint

```
@main.route('/complete_trip/<int:trip_id>')
```

- **Purpose**: Marks a trip as complete.
- **GET Request**:
    - Check if the logged-in user is the traveler who created the trip.
    - Updates the trip status to 'completed' and sets the end time.
    - Commits the changes to the database.
    - Logic will notify companions that the trip is complete (it was not implemented in the code snippet but would be added here).
- **Role-Based Access**: Only accessible to the traveler who created the trip.

Feedback Endpoint

```
@main.route('/feedback', methods=['POST'])
```

- **Purpose**: Allows companions to provide feedback on a trip.
- **POST Request**:
    - Creates a new feedback entry with the details provided in the form (trip ID, feedback text).
    - Adds the feedback to the database and commits the transaction.
- **Role-Based Access**: Only accessible to users with the companion role.

Check Feedback Endpoint

```
@main.route('/check_feedback', methods=['GET'])
```

- **Purpose**: Allows Admin to see all the feedback from companions
- **GET Request**:
    - Gets all the feedback from the Feedback table and returns to the admin
- **Role-Based Access**: Only accessible to users with the Admin role.

Check Shared Rides Endpoint

```
@main.route('/check_shared_rides', methods=['GET'])
```

- **Purpose**: Allows Admin to see all the rides
- **GET Request**:
    - Gets all the rides shared by users from the Trip table
- **Role-Based Access**: Only accessible to users with the Admin role.