Case Study
Ride Sharing Platform

Introduction:

Our platform caters to the needs of Travelers, their Companions, and Administrators. We focus on seamless ride-sharing experiences, top-notch security, and scalability to meet the ever-growing urban transportation demands.

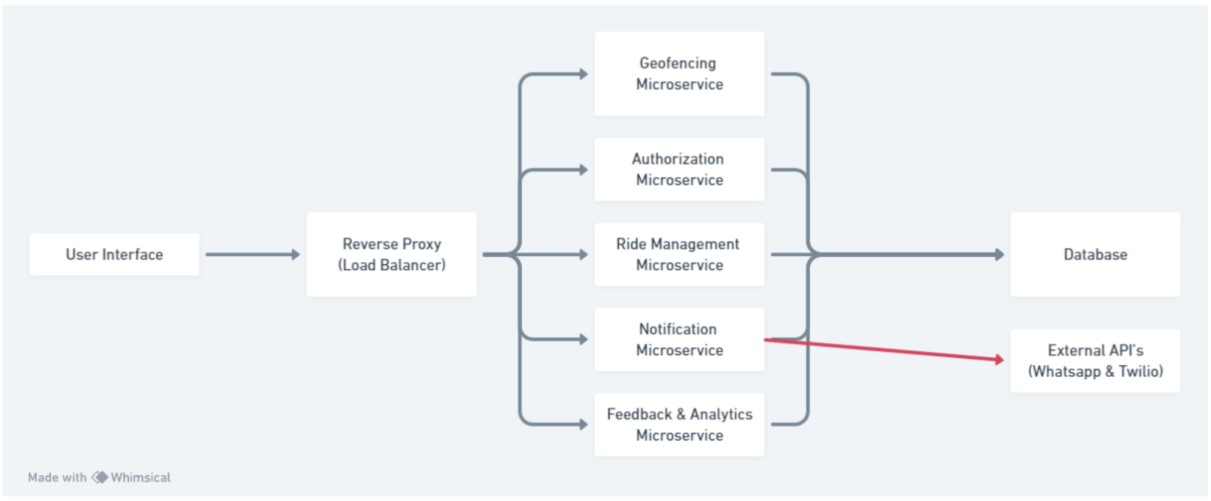
Project Overview:

The primary objective of this project is to create a seamless and user-friendly platform that facilitates efficient ride-sharing experiences while ensuring the highest standards of security, reliability, and scalability.

User Roles:

1. **Traveler:**
    - Empower travelers with the ability to share ride details during their journeys effortlessly.
    - This includes critical information such as Trip-Id, Driver Name, Driver Phone Number, and cab details through popular communication channels like WhatsApp or SMS.
    - Additionally, travelers should be able to review the audit trail of their shared rides for a comprehensive post-trip analysis.
2. **Traveler Companion:**
    - Enhance the travel experience for companions by tracking the traveler's real-time ride.
    - Traveler companions receive timely notifications, such as trip completion alerts and geofence triggers, when the cab approaches the traveller's drop location.
    - Moreover, the platform enables companions to share valuable feedback on the overall travel experience with the Admin.
3. **Admin:**
    - Empower administrators with comprehensive oversight and management capabilities.
    - Admins have the authority to review all rides users share on the platform.
    - Additionally, they can gain insights into the overall user experience through aggregated feedback, allowing for informed decision-making and continuous platform improvement.

System Architecture



User Interface
Reverse Proxy (Load Balancer)
Geofencing Microservice
Authorization Microservice
Ride Management Microservice
Notification Microservice
Feedback & Analytics Microservice
Database
External API's (Whatsapp & Twilio)
Made with Whimsical

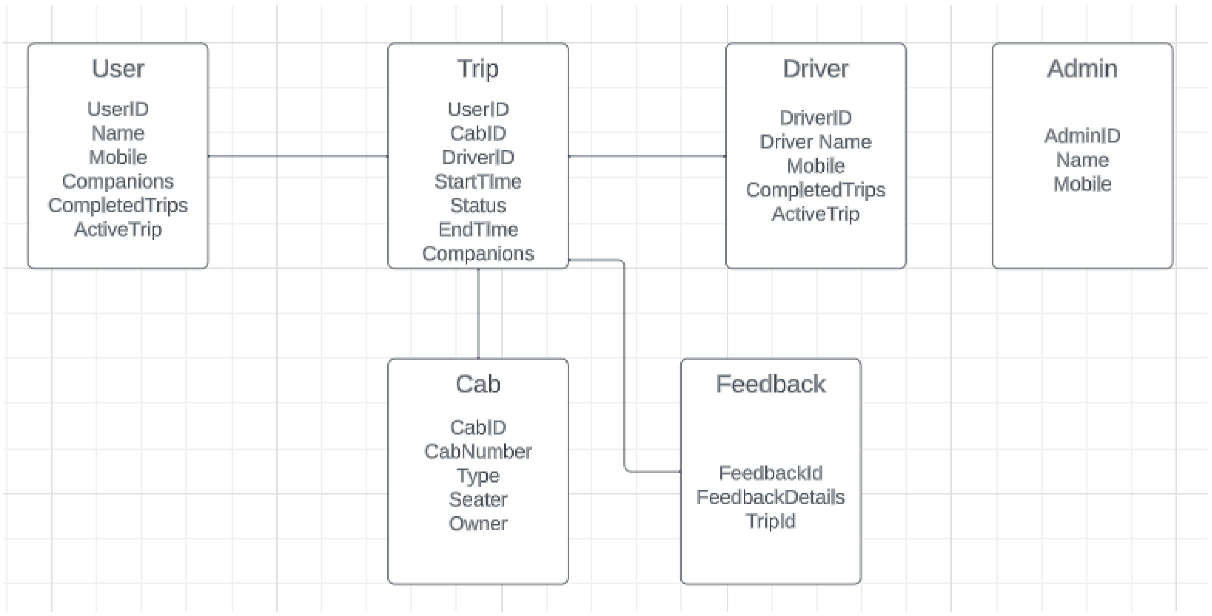
Microservices:

1. **Geofencing Microservice:**
    a. **Functionality:** Manages geofence-related operations, tracking the location of cabs and notifying relevant parties when they enter predefined geographical areas.
    b. **Key Responsibilities:** Geofence creation, monitoring, and triggering notifications based on cab movement.
2. **Authorization:**
    a. **Functionality:** Handles user authentication and authorization, ensuring secure access to the platform's features.
    b. **Key Responsibilities:** Authentication of users, authorization checks, and management of access tokens.
3. **Ride Management:**
    a. **Functionality:** Oversees the core ride-sharing operations, including trip creation, tracking, and completion.
    b. **Key Responsibilities:** Trip creation, real-time ride tracking, and managing the lifecycle of rides.
4. **Notification:**
    a. **Functionality:** Manages to send notifications to users and companions, including trip completion alerts and geofence triggers.
    b. **Key Responsibilities:** Notification generation, scheduling, and delivery based on various ride events.
5. **Feedback & Analytics:**
    a. **Functionality:** Gathers and analyses user feedback to enhance the platform's performance. Provides administrators with insights through analytics.
    b. **Key Responsibilities:** Collecting user feedback, analyzing trends, and presenting actionable insights to improve the ride-sharing experience.

External APIs

1. **WhatsApp API:**
    a. **Functionality:** Integrates with WhatsApp to facilitate communication between users and companions, sending ride details and updates.
    b. **Key Responsibilities:** Sending messages, sharing ride information, and engaging users through the WhatsApp platform.
2. **Twilio API (for SMS):**
    a. **Functionality:** Utilizes Twilio for sending SMS notifications to users and companions, ensuring timely communication.
    b. **Key Responsibilities:** Sending SMS alerts, notifications, and updates related to ride-sharing activities.

Database



Authorization

Authentication Implementation:

Ensuring the security of user accounts is paramount in our Ride-Sharing Platform. The implementation revolves around a robust authentication system that employs industry-standard protocols and best practices.

Authentication Protocol: OAuth 2.0 and JWT

We use OAuth 2.0 for secure user authentication because it's versatile and works well with different apps. OAuth 2.0 allows users to authorize access to their resources without revealing passwords. Within OAuth 2.0, we use JSON Web Tokens (JWT) to transmit information safely. JWT encodes user details, which are digitally signed to ensure integrity. This way, we keep user data secure during the authentication process.

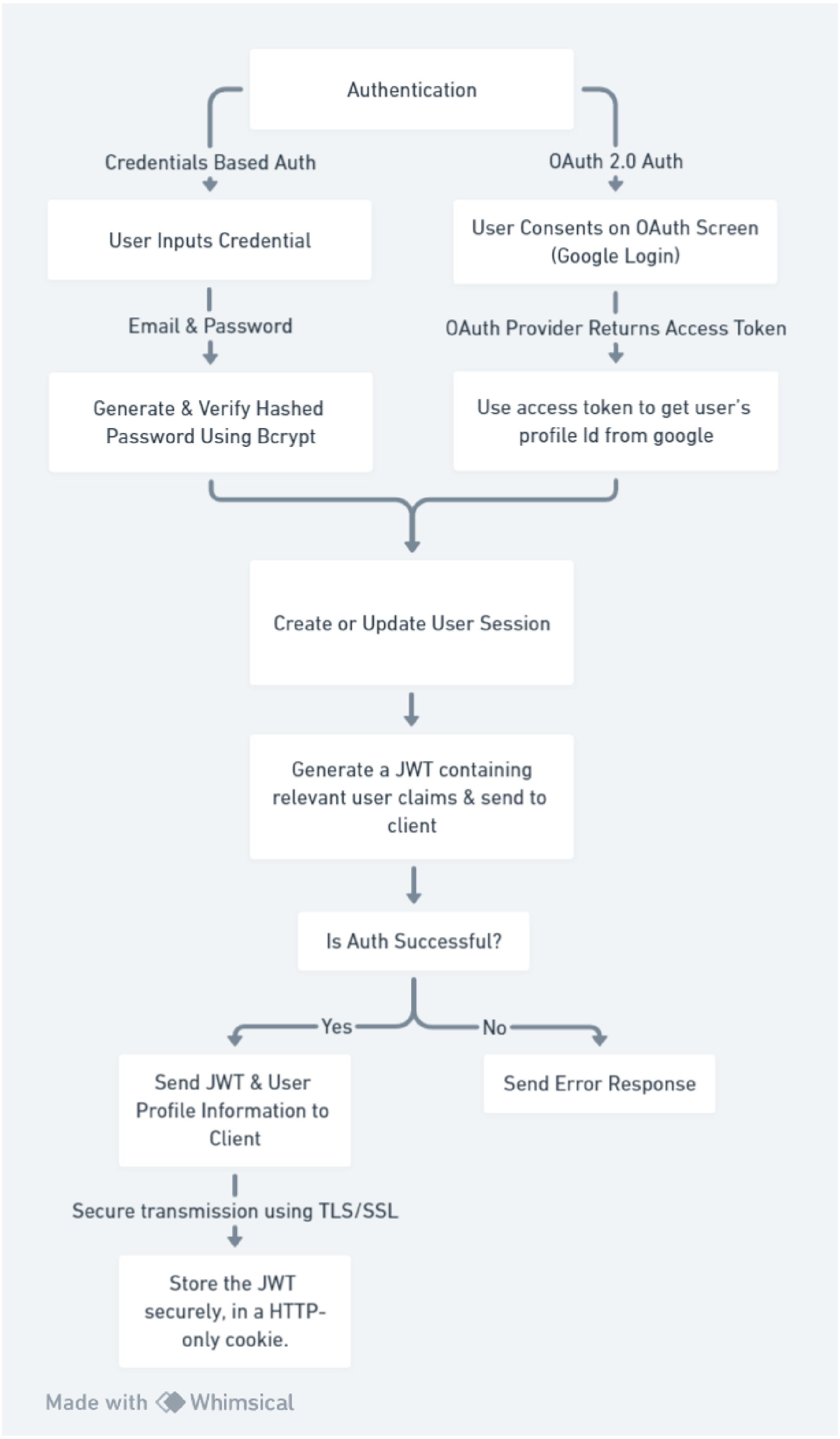
Secure Storage of User Credentials:

We store user credentials securely by hashing passwords with the robust Bcrypt algorithm. This ensures that plain-text passwords remain inaccessible even if there's a data breach. We add an extra layer of security by incorporating salt into the hashing process, making it difficult for attackers to use precomputed tables for password cracking.

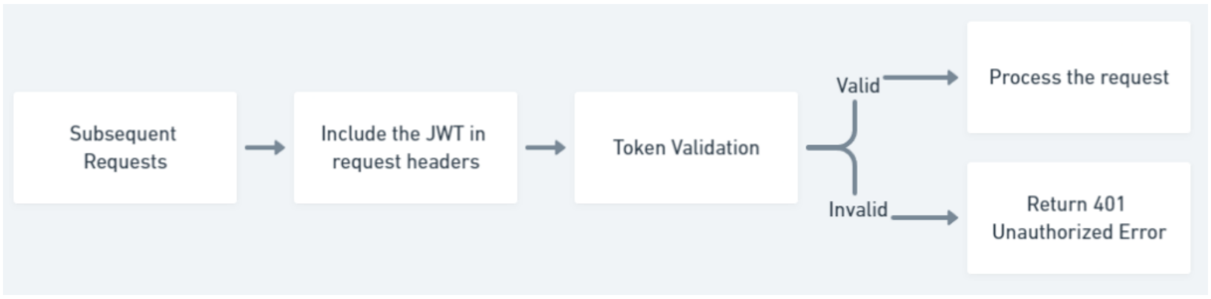
Encryption for Data Transmission:

We implement **Transport Layer Security** (TLS) across our communication channels to secure data transmission between the client and server. This encryption protocol safeguards sensitive information, including user credentials and session data, from potential eavesdropping and man-in-the-middle attacks.

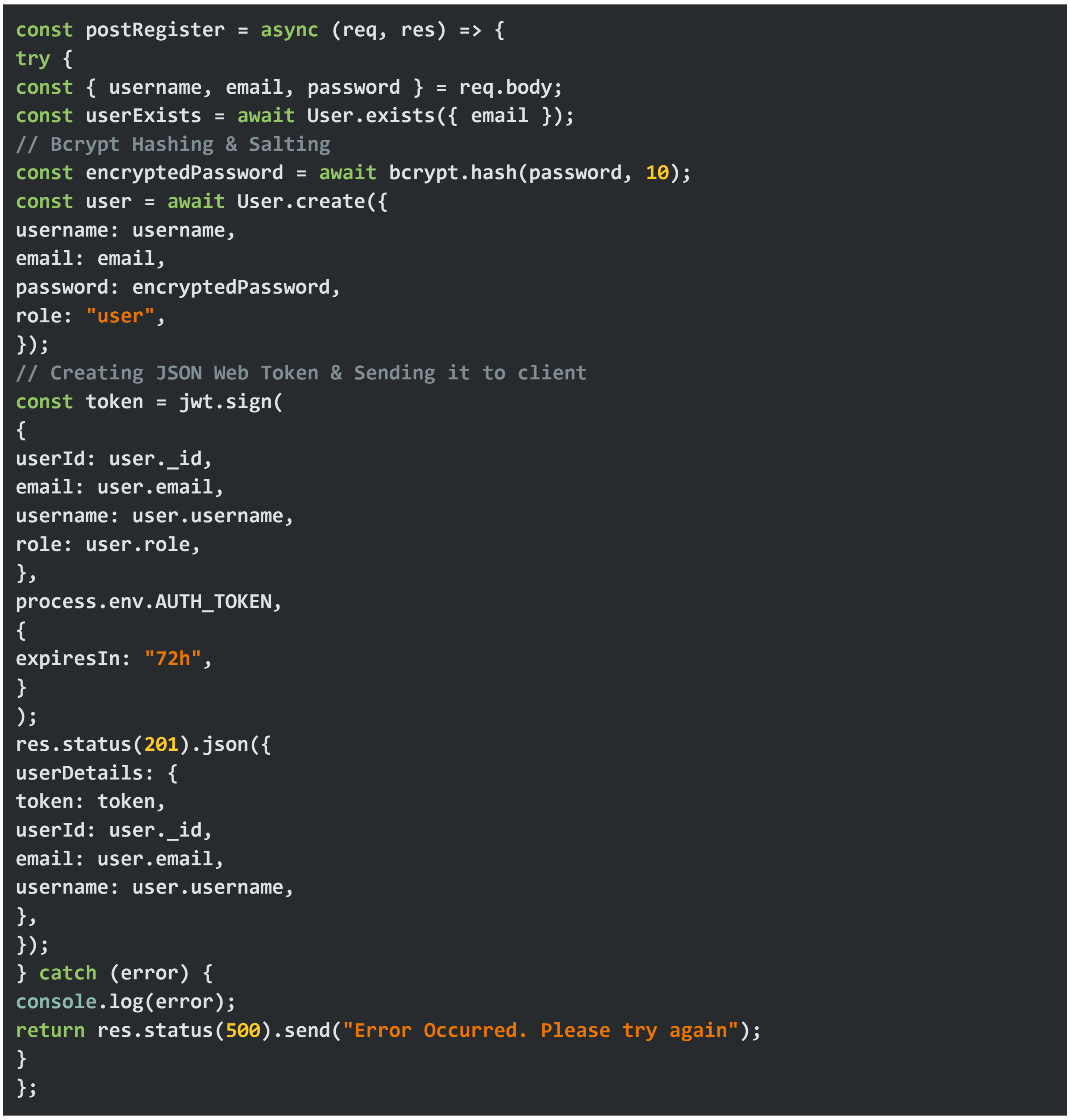
Authentication Flow:


Authentication
Credentials Based Auth
OAuth 2.0 Auth
User Inputs Credential
User Consents on OAuth Screen (Google Login)
Email & Password
OAuth Provider Returns Access Token
Generate & Verify Hashed Password Using Bcrypt
Use access token to get user's profile Id from google
Create or Update User Session
Generate a JWT containing relevant user claims & send to client
Is Auth Successful?
Yes
No
Send JWT & User Profile Information to Client
Send Error Response
Secure transmission using TLS/SSL
Store the JWT securely, in a HTTP-only cookie.
Made with Whimsical

Secure Requests:



Code Snippet:



```
const postRegister = async (req, res) => {
try {
const { username, email, password } = req.body;
const userExists = await User.exists({ email });
// Bcrypt Hashing & Salting
const encryptedPassword = await bcrypt.hash(password, 10);
const user = await User.create({
username: username,
email: email,
password: encryptedPassword,
role: "user",
});
// Creating JSON Web Token & Sending it to client
const token = jwt.sign(
{
userId: user._id,
email: user.email,
username: user.username,
role: user.role,
},
process.env.AUTH_TOKEN,
{
expiresIn: "72h",
}
);
res.status(201).json({
userDetails: {
token: token,
userId: user._id,
email: user.email,
username: user.username,
},
});
} catch (error) {
console.log(error);
return res.status(500).send("Error Occurred. Please try again");
}
};
```

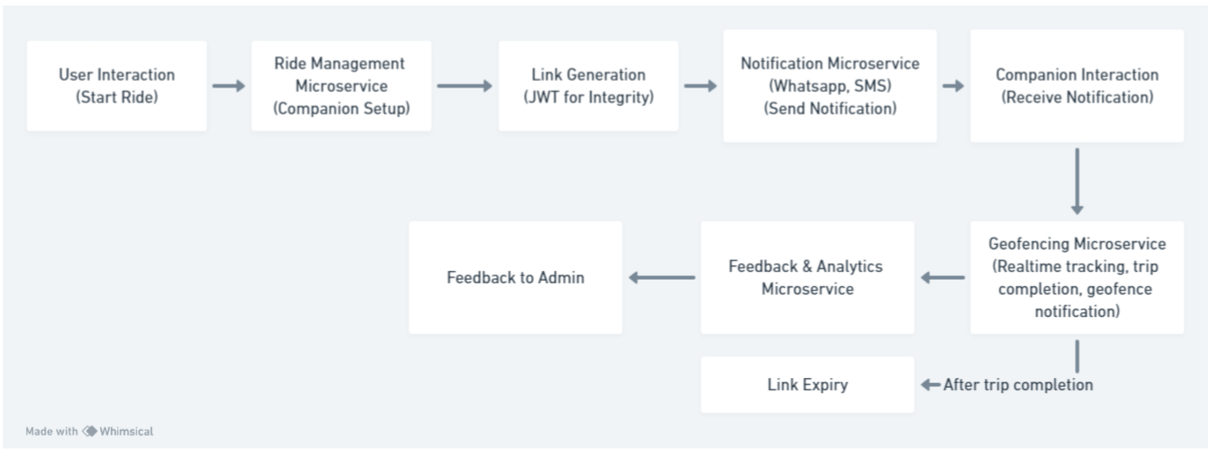
Ride Sharing Functionality

Assuming Ride has already been created, we are implementing the ride-sharing functionality.

1. **User Interaction:**
   The Traveler initiates the ride-sharing process through the user interface by selecting a companion and starting a ride.
2. **Ride Management Microservice:**
   The Ride Management microservice facilitates the ride-sharing process and interfaces with other components.
3. **Companion Setup:**
   Before starting a ride, the Traveler sets a companion by providing the companion's phone number through the Ride Management microservice.
4. **Link Generation:**
   When the Traveler starts a ride, the Ride Management microservice generates a unique link or token associated with that ride. This link contains details such as the trip ID and is signed using JWT for integrity.
5. **Notification Sending:**
   The Notification Microservice is triggered to notify the designated companion through WhatsApp or SMS. The notification includes the unique link and information about the ride.
6. **Companion Interaction:**
   The Traveler Companion receives the notification and opens the link, gaining access to real-time tracking and additional features.
7. **Real-time Tracking:**
   The link allows the Traveler Companion to track the ride in real-time using the Geofencing Microservice.
8. **Trip Completion Notification:**
   The Geofencing Microservice sends a notification to the Traveler Companion when the trip is complete, indicating a successful ride.
9. **Geofence Notification:**
   The Geofencing Microservice monitors the cab's location and sends nearby notifications to the Traveler Companion when the cab approaches the geofence of the traveller's drop location.
10. **Feedback Submission:**
    The Traveler Companion can share feedback about the ride experience through a feedback form submitted to the Feedback & Analytics Microservice.
11. **Link Expiry:**
    The link generated for ride-sharing has a time-limited validity. The link expires once the ride is complete, ensuring that access to ride details is restricted after the trip.
12. **Feedback to Admin:**
    The Feedback & Analytics Microservice records the feedback and forwards it to the Admin for analysis.

Security Measures
1. **JWT for Link Integrity:**
The link generated for ride-sharing is encoded as a JWT, ensuring the integrity of the information and preventing tampering.

2. **Secure Notification Delivery:**
Notifications containing links are sent securely through WhatsApp or SMS, utilising encryption protocols provided by these services.
3. **Link Expiry:**
The time-limited validity of the link ensures that even if intercepted, it becomes useless after the ride is complete.
4. **Transport Layer Security (TLS):**
The entire ride-sharing process, including link generation and companion interactions, is secured using TLS for encrypted data transmission.


User Interaction (Start Ride)
Ride Management Microservice (Companion Setup)
Link Generation (JWT for Integrity)
Notification Microservice (Whatsapp, SMS) (Send Notification)
Companion Interaction (Receive Notification)
Geofencing Microservice (Realtime tracking, trip completion, geofence notification)
Feedback & Analytics Microservice
Feedback to Admin
After trip completion
Link Expiry
Made with Whimsical

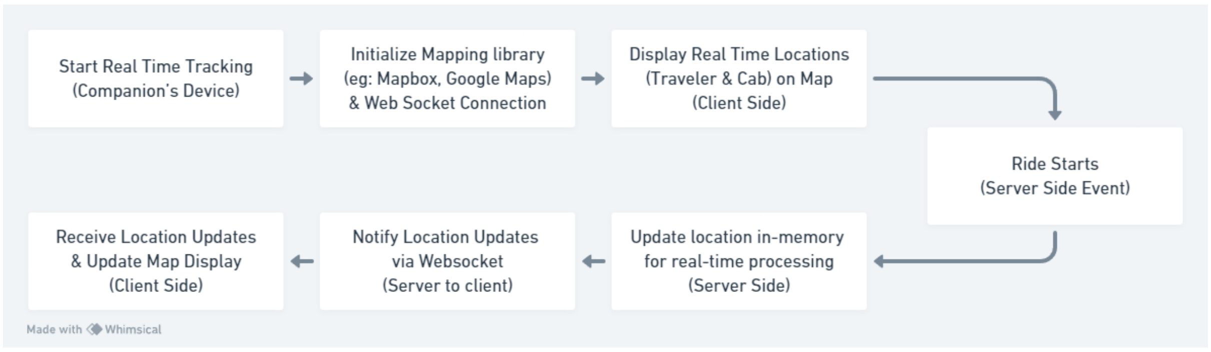
Real-time Tracking Functionality

Client Side:

- Use a mapping library (e.g., Mapbox, Google Maps) to display the traveller's and cab's real-time location on the Traveler Companion's device.
- Utilise a WebSocket connection to continuously update the traveller's and cab's location on the map.

Server Side:

- When the ride starts, update the traveller's and cab's locations on the server regularly.
- Use WebSocket to notify the Traveler Companion's device of location updates.



Start Real Time Tracking (Companion's Device)
Initialize Mapping library (eg: Mapbox, Google Maps) & Web Socket Connection
Display Real Time Locations (Traveler & Cab) on Map (Client Side)
Ride Starts (Server Side Event)
Update location in-memory for real-time processing (Server Side)
Notify Location Updates via Websocket (Server to client)
Receive Location Updates & Update Map Display (Client Side)
Made with Whimsical

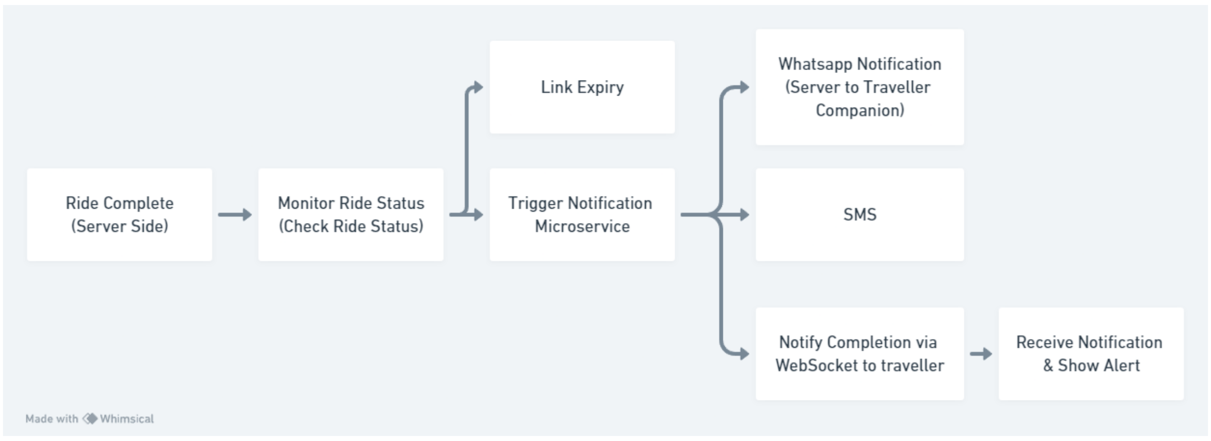
Trip Completion Notification:

Server Side:

- Monitor the ride's status, and when it's marked as complete, trigger a notification event.

Client Side:

- Use WebSocket to listen for trip completion events.
- When the event occurs, show a notification on the Traveler Companion's device indicating that the trip is complete.



Ride Complete (Server Side)
Monitor Ride Status (Check Ride Status)
Link Expiry
Trigger Notification Microservice
Whatsapp Notification (Server to Traveller Companion)
SMS
Notify Completion via WebSocket to traveller
Receive Notification & Show Alert
Made with Whimsical

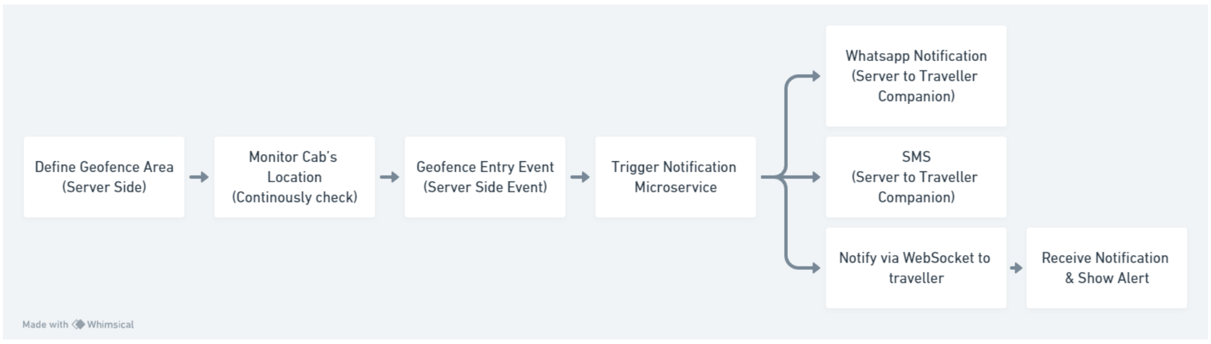
Geofence Notification:

Server Side:

- Define a geofence area around the traveller's drop location.
- Continuously monitor the cab's location.
- When the cab enters the geofence area, trigger a geofence entry event.

Client Side:

- Use WebSocket to listen for geofence entry events.
- When the event occurs, notify the Traveler Companion's device that the cab is nearby.


Define Geofence Area (Server Side)
Monitor Cab's Location (Continously check)
Geofence Entry Event (Server Side Event)
Trigger Notification Microservice
Whatsapp Notification (Server to Traveller Companion)
SMS (Server to Traveller Companion)
Notify via WebSocket to traveller
Receive Notification & Show Alert
Made with Whimsical

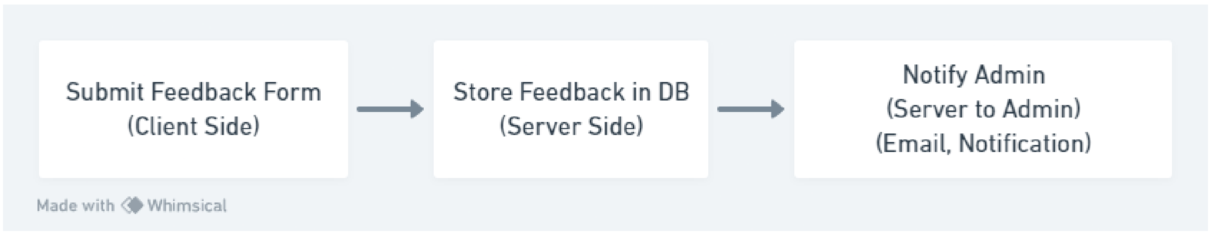
Feedback Functionality:

Server Side:

- Provide an API endpoint for submitting feedback.
- Fetch and serve feedback data from the database.
- The server forwards feedback to the Admin for analysis.

Client Side:

- Implement a feedback form in the client application.
- Users submit feedback about their ride experience.
- Collected feedback is sent to the server.


Submit Feedback Form (Client Side)
Store Feedback in DB (Server Side)
Notify Admin (Server to Admin) (Email, Notification)
Made with Whimsical

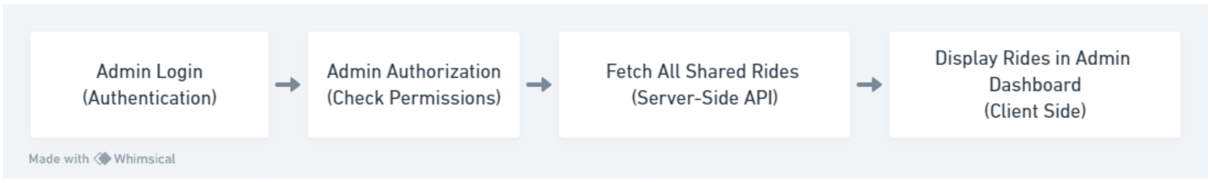
View All Shared Rides (Admin)

Server Side:

- Provide an API endpoint for admin authentication and authorization.
- Fetch and serve ride data from the database.

Client Side:

- Admins access the ride information through a secure admin dashboard.


Admin Login (Authentication)
Admin Authorization (Check Permissions)
Fetch All Shared Rides (Server-Side API)
Display Rides in Admin Dashboard (Client Side)
Made with Whimsical

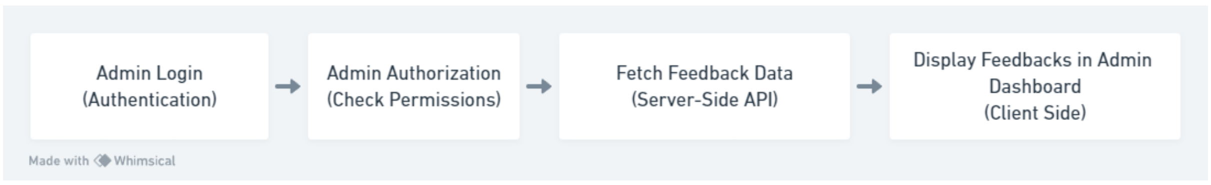
View All Feedbacks (Admin)

Server Side:

- Aggregate and process user feedback data.
- Make feedback data available through a secured API endpoint.

Client Side:

- Admins access the feedback overview through the admin dashboard.


Admin Login (Authentication)
Admin Authorization (Check Permissions)
Fetch Feedback Data (Server-Side API)
Display Feedbacks in Admin Dashboard (Client Side)
Made with Whimsical

Cost Estimation - Time and Space:

Dynamic Location Update Handling:

Real-time location updates are handled in memory to minimise frequent database writes. A combination of a traditional database for persistent storage and an in-memory database or caching solution for temporary data storage could be beneficial. On the other hand, traditional databases can be used for persistent storage.

- **Persistent Storage (Traditional Database):**
  - Database Choice: PostgreSQL or **MongoDB**
    - PostgreSQL: Offers substantial ACID compliance and supports complex queries, suitable for structured data.
    - MongoDB: Well-suited for scenarios with rapidly changing data structures and the need for flexible schema design.
- **In-Memory Storage (Caching Solution):**
  - Caching System: **Redis** or Memcached
    - Redis: Provides advanced data structures and persistence options, making it suitable for real-time data.
    - Memcached: Simplicity and speed make it practical for caching frequently accessed data.

In-Memory Storage for Location Updates (Time Complexity):

- Advantages: Updating in-memory storage is faster than frequent database writes, reducing time complexity.
- Considerations: Analyze the impact on write operations when managing updates in memory.

Scalability Strategies (Time & Space Complexity)

- Advantages: In-memory updates can enhance system scalability by reducing database load during peak times.
- Considerations: Plan for horizontal scaling of server resources to accommodate increased in-memory storage demands.

Conclusion:

In conclusion, our ride-sharing platform successfully integrates essential features such as robust user authentication, efficient ride-sharing functionalities, real-time tracking for traveler companions, and a streamlined feedback mechanism for administrators. The implementation of WhatsApp and SMS notifications enhances the user experience. Leveraging in-memory storage for location updates optimizes time and space complexity, ensuring swift and responsive performance. Scalability strategies have been systematically integrated, laying the foundation for accommodating an expanding user base. This comprehensive solution aligns with the project's goals, offering a secure, scalable, and user-centric ride-sharing experience.