CASE STUDY

- **Topic - Smart Cab Allocation System for Efficient Trip Planning**

- **NAME : RHYTHM BAGHEL**

Workflow

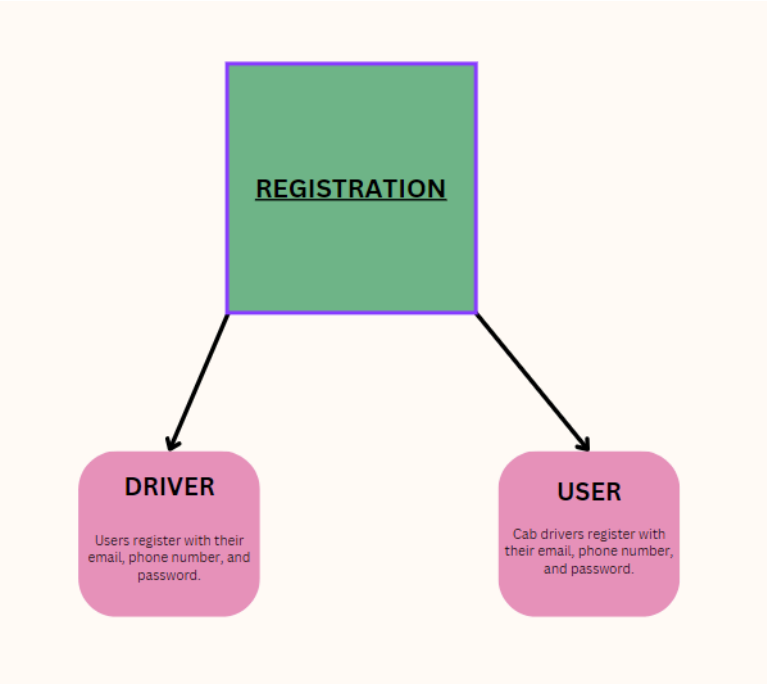
User Registration and Authentication

User Registration:

- Users register with their email, phone number, and password.

Cab Driver Registration:

- Cab drivers register with their email, phone number, and password.



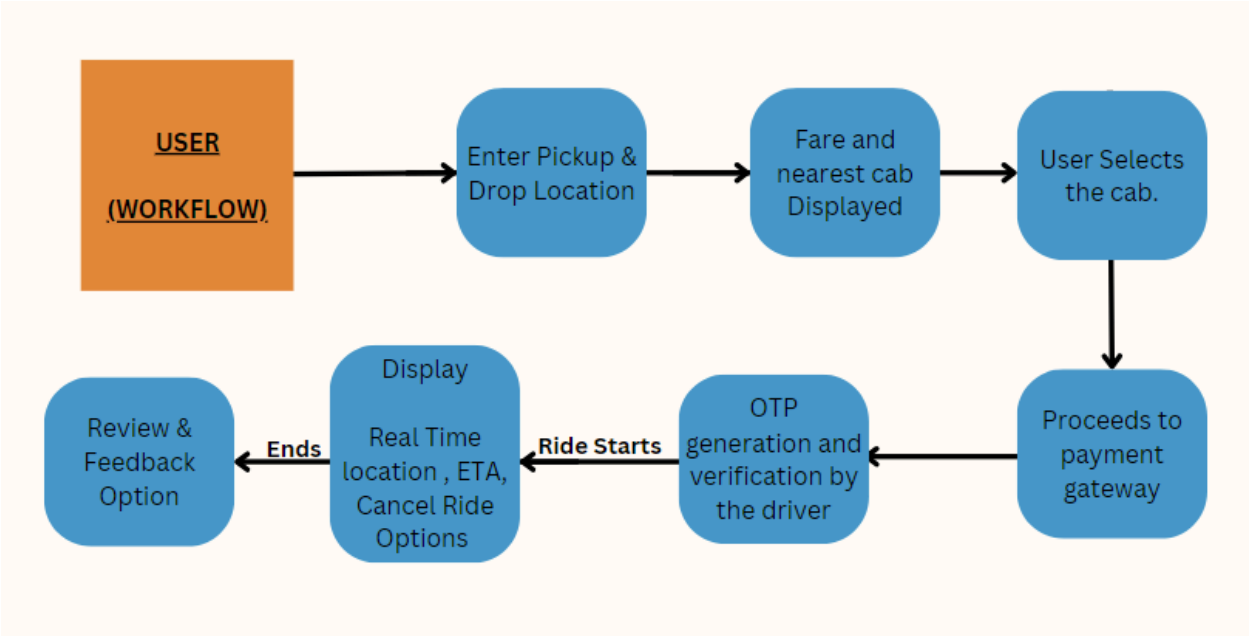
Log in / Sign up:

- Both users and cab drivers log in or sign up to access the application.

After Logging In

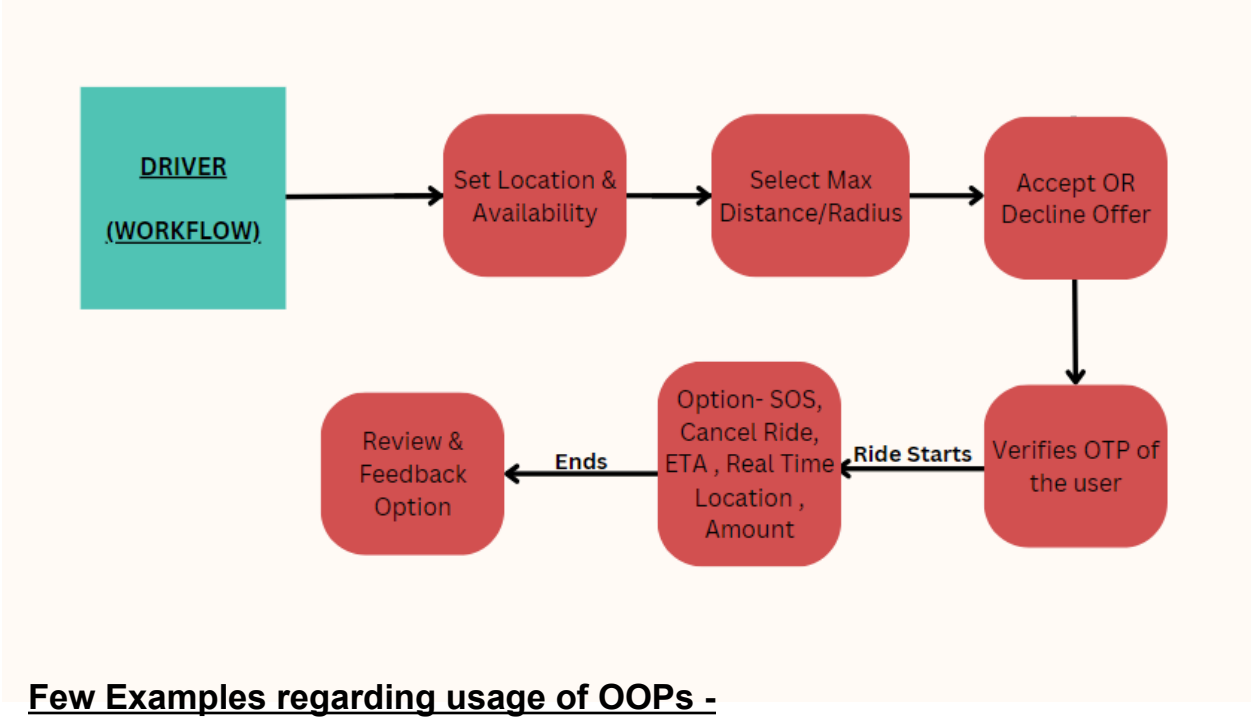
For Users:

1. Users can enter their location and request a ride (can book for now or later). They can also book a cab for someone else.
2. During booking, the app shows the nearest cabs and presents the fare and cancellation policies.
3. Users can select the type of cab they wish to ride, such as standard, premium, etc., and their respective price based on the algorithm should be displayed.
4. Once the user has selected their preferred type of ride, they proceed to the payment method (if it's an online payment).
5. The app generates a random OTP (One-Time Password) which needs to be verified by the driver.
6. Users have access to features like SOS button, ETA (Estimated Time of Arrival), fare amount, real-time location, and the option to cancel the ride.
7. After the ride is completed, users can provide reviews and feedback for the driver.



For Drivers:

1. Drivers can select the area they want to operate in and set their availability.
2. Drivers can set the maximum distance they'll cover to pick up users.
3. After receiving a ride request, drivers can accept or decline it.
4. Drivers also receive an OTP that must be verified by the user.
5. Drivers have access to features like cancel ride (SOS button), real-time location, ETA, and fare amount.
6. After the ride is completed, drivers can also receive reviews and feedback from users.



Few Examples regarding usage of OOPs -

Encapsulation:

- **Concept:** Each user's information (like their name, email, and phone number) and their actions (register, login, request a ride) are encapsulated within the User class.
- **Analogy:** This is like having a secure vault (the class) where you keep all important documents (data) and tools (methods) needed for specific tasks, ensuring that everything is organized and protected from outside interference.

Inheritance:

- **Concept:** Create a general Person class with common attributes and methods. The User and Driver classes inherit these attributes and methods but also have additional functionalities specific to their roles.
- **Analogy:** This is similar to inheriting a family recipe. The basic recipe (Person class) can be used as is, but each family member (User or Driver class) might add their own twist to it.

Polymorphism:

- **Concept:** The provide_feedback method can be used by both User and Driver, but the feedback content differs. This allows the system to handle feedback uniformly while letting each class define its specific behavior.
- **Analogy:** Think of a multi-tool with a single handle but multiple interchangeable heads (screwdriver, knife, bottle opener). The tool's purpose changes based on the head attached, yet it remains the same tool.

Key Functionalities:

1. **Booking and Cancellation:**Users can book and cancel rides.

2. **Matching Algorithm:**The app uses a matching algorithm to connect users with nearby drivers.

3. **Real-Time Tracking and Estimated Time of Arrival (ETA):**Provides real-time tracking of rides and calculates ETA.

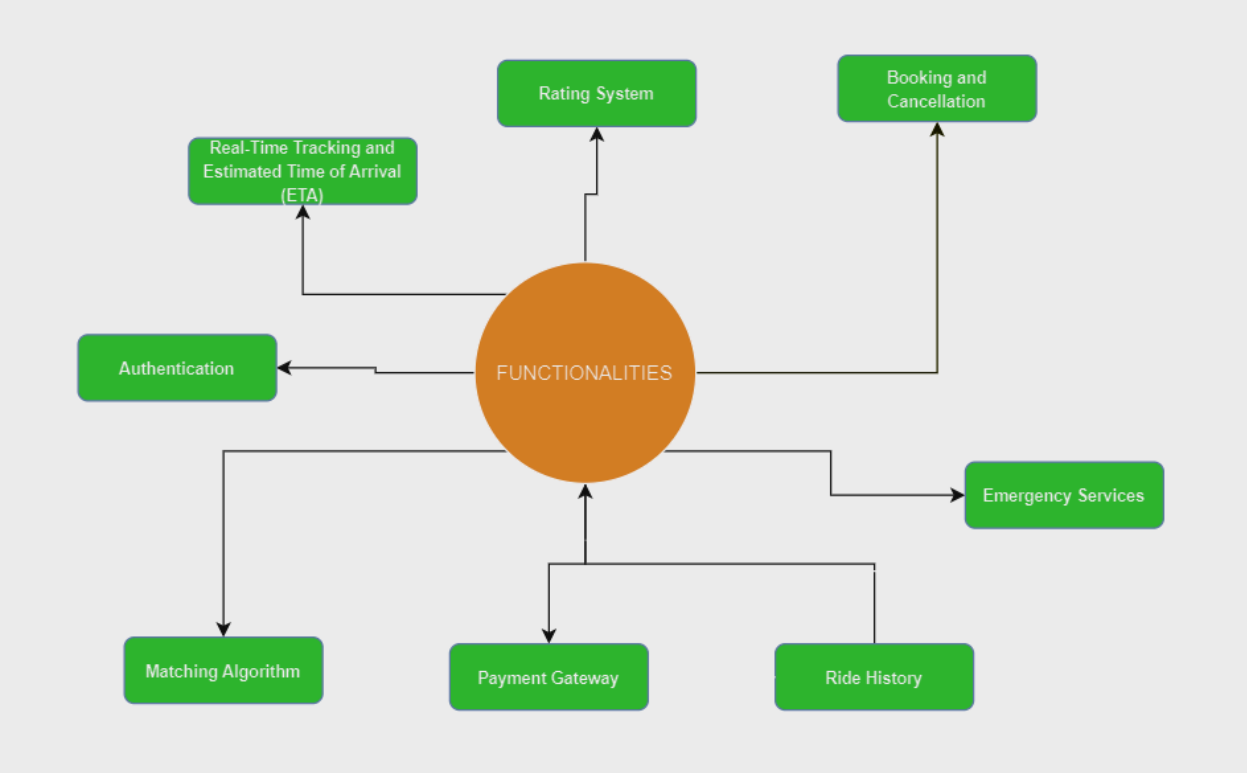
4. **Payment Gateway:**Integrates with a payment gateway for online payments.

5. **Ride History:**Users and drivers can view their ride history.

6. **Rating System:**Both users and drivers can rate each other.

7. **Authentication:**Includes various authentication mechanisms such as email verification, emergency contacts, and two-step authentication.

8. **Emergency Services:**Features an SOS button for emergencies.



Technical Aspects:

Software

- **Architecture:**Microservices architecture for scalability and maintainability.

- **Scalability:**Independent scaling of microservices based on demand.

- **Redundant Mechanisms for Failures:**Ensures high availability and fault tolerance.

- **Monitoring and Alerting:**Comprehensive monitoring with real-time dashboards.

- **Logging and Tracing:**Centralized logging for troubleshooting and performance monitoring.

Hardware

- **Microservices:** Deployed across multiple servers.

- **Load Balancing:**Round Robin load balancing for uniform request distribution.

Microservices Architecture

Overview

A microservices architecture offers several benefits for a cab booking application, such as scalability, maintainability, and the ability to deploy services independently.

1. User Service:

- Handles user registration, authentication, profile management, and user data.
- **Database:** MongoDB for user profiles and authentication data.

2. Driver Service:

- Manages driver registration, authentication, availability, and driver data.
- **Database:** MongoDB for driver profiles and availability data.

3. Ride Service:

- Manages ride requests, bookings, cancellations, ride history, and fare calculations.
- **Database:** PostgreSQL for transactional data related to rides.

4. Payment Service:

- Processes payments, handles transactions, manages payment methods, and generates invoices.
- Integrates with third-party payment gateways.
- **Database:** MySQL for payment transactions and invoices.

5. Notification Service:

- Sends notifications to users and drivers for ride status updates, OTPs, and alerts.
- Uses message queues (RabbitMQ in our case) for asynchronous communication.
- Integrates with SMS and email services.

6. Location Service:

- Provides real-time tracking of rides, calculates ETAs, and manages GPS data.
- **Database:** Redis for real-time location data.

7. Review Service:

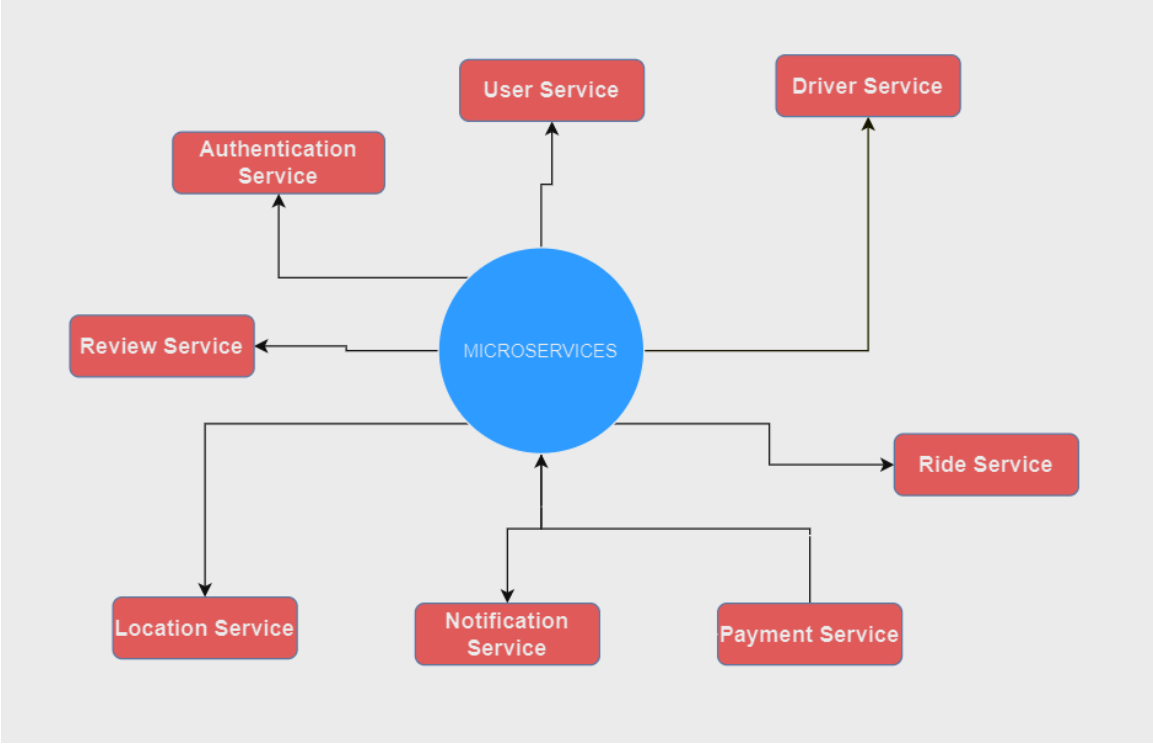
- Manages the review and rating system for users and drivers.
- **Database:** Elasticsearch for indexing and querying review data.

8. Authentication Service:

- Manages authentication mechanisms, including OTP generation, two-step verification, and security protocols.
- **Database:** MongoDB for security tokens and session data.

9. API Gateway:

- Acts as the entry point for all client requests.
- Routes requests to appropriate services, provides rate limiting, load balancing, and security features.



Load Balancing

Round Robin Load Balancing

Process:

Request Distribution:

- Incoming client requests are distributed in a cyclic order to each server instance.
- Each server receives a request, then the next server in the list, and so on, repeating the cycle.

Implementation:

- Configure the load balancer to use the round robin algorithm.
- Define backend server instances in the load balancer configuration.

Advantages:

- **Simplicity:**Easy to implement and understand.
- **Uniform Load Distribution:**Ensures all server instances receive an equal number of requests over time.
- **No Complex Configuration:**Does not require tracking the state or performance metrics of server instances.

Reasons for Not Using Other Strategies:

- **Least Connections:**
    - Requires monitoring active connections, which adds overhead.
    - Beneficial for services with highly variable processing times, which is not a primary concern for our application.

- **IP Hash:**
    - Ensures session persistence but can lead to uneven load distribution if the number of unique IPs is low or unevenly distributed.
    - More suitable for applications requiring sticky sessions, which is not a major requirement here.

Testing Methods

For our cab booking application, essential testing methods include:

1. Unit Testing:

- **Purpose:** Verify individual components and services function correctly in isolation.
- **Tools:** JUnit for Java-based services.
- **Example:** Testing the OTP generation function in the Authentication Service.

2. Integration Testing:

- **Purpose:** Ensure that different services interact correctly.
- **Tools:** Postman for API testing.
- **Example:** Testing the interaction between the User Service and the Ride Service when booking a ride.

3. Functional Testing:

- **Purpose:** Validate the application against the functional requirements.
- **Tools:** JUnit for backend functional tests.
- **Example:** Testing the complete ride booking flow, including user registration, ride request, OTP verification, and payment processing.

4. Performance Testing:

- **Purpose:** Assess the application's responsiveness, stability, and scalability under load.
- **Types:**
    - **Load Testing:** Simulates a high number of users to test system behavior under expected load.
    - **Stress Testing:** Pushes the system beyond normal operational capacity to identify its breaking point.
- **Tools:** Apache JMeter.
- **Example:** Simulating 10,000 concurrent ride requests to evaluate system performance.

5. Security Testing:

- **Purpose:** Identify and address vulnerabilities to ensure the application is secure.
- **Tools:** Burp Suite.
- **Example:** Conducting penetration tests to identify vulnerabilities in the authentication flow, such as SQL injection or cross-site scripting (XSS).

6. User Acceptance Testing (UAT):

- **Purpose:** Validate the system against user requirements and ensure it is ready for production.
- **Conducted By:** End-users or clients.
- **Example:** End-users testing the ride booking flow and providing feedback before the final release.

Trade-offs in the System

Performance vs. Scalability:

- **Decision:** We chose a microservices architecture to enhance scalability at the expense of some performance overhead due to inter-service communication.
- **Rationale:** This decision allows each service to be scaled independently based on demand, improving overall system flexibility and resilience.

Complexity vs. Maintainability:

- **Decision:** Using multiple microservices increases system complexity but improves maintainability.
- **Rationale:** Each service is easier to understand, test, and deploy independently, reducing the risk of introducing bugs when making changes.

Consistency vs. Availability:

- **Decision:** Implement eventual consistency in some parts of the system, such as ride status updates.
- **Rationale:** This approach improves availability and performance but might lead to temporary inconsistencies, which are acceptable for our use case.

Security vs. Usability:

- **Decision:** Implementing two-factor authentication for both users and drivers.
- **Rationale:** Enhances security but adds an extra step in the login process, slightly reducing usability.

Evaluate and Communicate the Rationale

Performance:

- Evaluate the impact of network latency and inter-service communication overhead in a microservices architecture.
- Communicate the benefits of independent scaling and fault isolation.

Scalability:

- Assess the ability to scale individual services horizontally to handle increased load.
- Highlight the ease of scaling specific parts of the application based on traffic patterns.

Maintainability:

- Consider the ease of updating, testing, and deploying services independently.
- Emphasize the reduced risk of changes affecting unrelated parts of the system.

System Monitoring

Metrics to Monitor:

- CPU and memory usage of each microservice.
- Request rates, error rates, and latency for each service.
- Database performance metrics, including query response times and throughput.

Utilize Real-time Dashboards and Logging Mechanisms

Real-time Dashboards:

- Create dashboards to visualize key performance indicators (KPIs) and system health.
- Example dashboards: service response times, request throughput, error rates.

Logging Mechanisms:

- Implement structured logging to collect and parse logs.
- To visualize and search through logs, making it easier to troubleshoot issues.

Caching

Integrate Caching Mechanisms

Tools:

- Use Redis for caching frequently accessed data.

Caching Strategies:

- Cache ride fare calculations, driver availability, and user session data to reduce database load.
- Use content delivery networks (CDNs) to cache static assets, such as images and scripts, for faster content delivery.

Implement Cache Eviction Policies

Eviction Policies:

- **Least Recently Used (LRU):** Evict the least recently used items when the cache reaches its maximum capacity.
- **Time-to-Live (TTL):** Set expiration times for cached items to ensure stale data is removed.
- **Custom Policies:** Implement custom eviction strategies based on specific application requirements, such as prioritizing eviction of less critical data

TASK -1 Admin's Cab Allocation Optimization:

```
function computeDistance(coord1, coord2) {
    const R = 6371.0;
    const dLat = (coord2.latitude - coord1.latitude) * Math.PI / 180.0;
    const dLon = (coord2.longitude - coord1.longitude) * Math.PI / 180.0;
    const a = Math.sin(dLat / 2) * Math.sin(dLat / 2) +
              Math.cos(coord1.latitude * Math.PI / 180.0) * Math.cos(coord2.latitude * Math.PI / 180.0) *
              Math.sin(dLon / 2) * Math.sin(dLon / 2);
    const c = 2 * Math.atan2(Math.sqrt(a), Math.sqrt(1 - a));
    return R * c;
}

async function findNearestCab(trip, availableCabs) {
    let minDistance = Infinity;
    let nearestCabId = -1;

    for (const cab of availableCabs) {
        await cab.mutex.runExclusive(async () => {
            const distance = computeDistance(trip.startLocation, cab.location);
            if (distance < minDistance) {
                minDistance = distance;
                nearestCabId = cab.id;
            }
        });
    }
    return nearestCabId;
```

TASK 2: Employee's Cab Search Optimization:

```
function calculateDistance(x1, y1, x2, y2) {
    return Math.sqrt((x2 - x1) ** 2 + (y2 - y1) ** 2);
}

function findNearbyCabs(cabs, empX, empY, maxDistance) {
    const nearbyCabs = cabs.filter(cab => cab.inUse && calculateDistance(empX, empY, cab.x, cab.y) <=
    maxDistance);
    nearbyCabs.sort((a, b) => calculateDistance(empX, empY, a.x, a.y) - calculateDistance(empX, empY, b.x, b.
    y));
    return nearbyCabs;
}
```