41. UBER

Let's design an Uber like ride-hailing service, similar to services like Lyft, OLA Cabs, etc.

What is Uber?

Uber is a mobility service provider, allowing users to book rides and a driver to transport them in a way similar to a taxi. It is available on the web and mobile platforms such as Android and iOS.

Requirements

Our system should meet the following requirements:

1. Functional requirements

We will design our system for two types of users: Customers and Drivers.

Customers

- Customers should be able to see all the cabs in the vicinity with an ETA and pricing information.
- Customers should be able to book a cab to a destination.
- Customers should be able to see the location of the driver.

Drivers

- Drivers should be able to accept or deny the customer-requested ride.
- Once a driver accepts the ride, they should see the pickup location of the customer.
- Drivers should be able to mark the trip as complete on reaching the destination.

2. Non-Functional requirements

- High reliability.
- High availability with minimal latency.

- The system should be scalable and efficient.

Extended requirements

- Customers can rate the trip after it's completed.
- Payment processing.
- Metrics and analytics.

Estimation and Constraints

Let's start with the estimation and constraints.

Note: Make sure to check any scale or traffic-related assumptions with your interviewer.

Traffic

Let us assume we have 100 million daily active users (DAU) with 1 million drivers and on average our platform enables 10 million rides daily.

If on average each user performs 10 actions (such as request a check available rides, fares, book rides, etc.) we will have to handle 1 billion requests daily.

100 million × 10 actions = 1 billion/day

What would be Requests Per Second (RPS) for our system?

1 billion requests per day translate into 12K requests per second.

$$\frac{1\ billion}{(24\ hrs \times 3600\ seconds)} =\sim 12K\ requests\ /\ second$$

Storage

If we assume each message on average is 400 bytes, we will require about 400 GB of database storage every day.

1 billion × 400 bytes = ~ 400GB /day

And for 10 years, we will require about 1.4 PB of storage.

400 GB × 10 years × 365 days = ~ 1.4 PB

Bandwidth

As our system is handling 400 GB of ingress every day, we will require a minimum bandwidth of around 5 MB per second.

$$\frac{400\ GB}{(24\ hrs \times 3600\ seconds)} =\sim 5\ MB\ /\ second$$

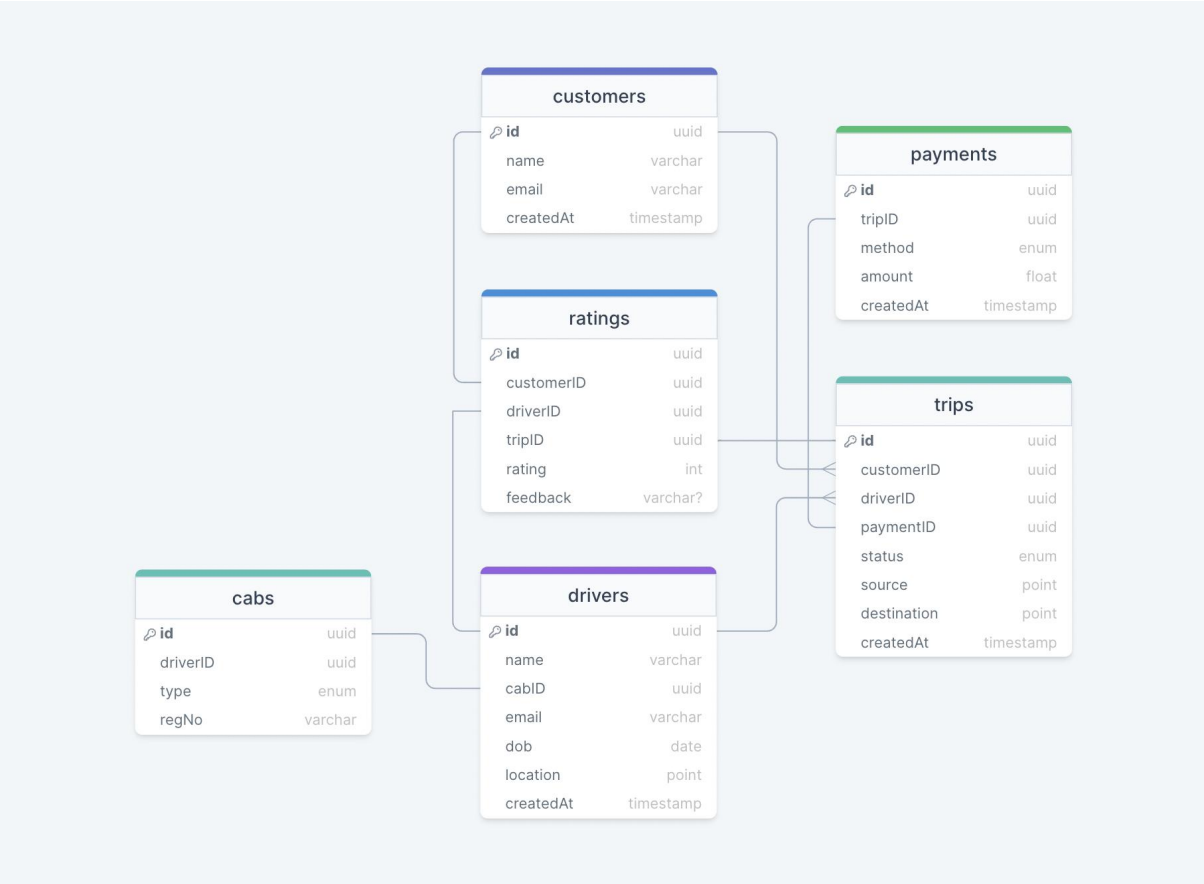
High-level estimate

Here is our high-level estimate:

Type	Estimate
Daily active users (DAU)	100 million
Requests per second (RPS)	12K/s
Storage (per day)	~400 GB
Storage (10 years)	~1.4 PB
Bandwidth	~5 MB/s

Data model design

This is the general data model which reflects our requirements.



We have the following tables:

customers

This table will contain a customer's information such as name, email, and other details.

drivers

This table will contain a driver's information such as name, email, dob and other details.

trips

This table represents the trip taken by the customer and stores data such as source, destination, and status of the trip.

cabs

This table stores data such as the registration number, and type (like Uber Go, Uber XL, etc.) of the cab that the driver will be driving.

ratings

As the name suggests, this table stores the rating and feedback for the trip.

payments

The payments table contains the payment-related data with the corresponding tripID.

What kind of database should we use?

While our data model seems quite relational, we don't necessarily need to store everything in a single database, as this can limit our scalability and quickly become a bottleneck.s

We will split the data between different services each having ownership over a particular table. Then we can use a relational database such as PostgreSQL or a distributed NoSQL database such as Apache Cassandra for our use case.

API design

Let us do a basic API design for our services:

Request a Ride

Through this API, customers will be able to request a ride.

```
requestRide(customerID: UUID, source: Tuple<float>, destination: Tuple<float>, cabType: Enum<string>, paymentMethod: Enum<string>): Ride
```

Parameters

- Customer ID (UUID): ID of the customer.
- Source (Tuple<float>): Tuple containing the latitude and longitude of the trip's starting location.
- Destination (Tuple<float>): Tuple containing the latitude and longitude of the trip's destination.

Returns

Result (Ride): Associated ride information of the trip.

Cancel the Ride

This API will allow customers to cancel the ride.

```
cancelRide(customerID: UUID, reason?: string): boolean
```

Parameters

- Customer ID (UUID): ID of the customer.
- Reason (UUID): Reason for canceling the ride (optional).

Returns

Result (boolean): Represents whether the operation was successful or not.

Accept or Deny the Ride

This API will allow the driver to accept or deny the trip.

```
acceptRide(driverID: UUID, rideID: UUID): boolean

denyRide(driverID: UUID, rideID: UUID): boolean
```

Parameters

- Driver ID (UUID): ID of the driver.
- Ride ID (UUID): ID of the customer requested ride.

Returns

Result (boolean): Represents whether the operation was successful or not.

Start or End the Trip

Using this API, a driver will be able to start and end the trip.

```
startTrip(driverID: UUID, tripID: UUID): boolean

endTrip(driverID: UUID, tripID: UUID): boolean
```

Parameters

- Driver ID (UUID): ID of the driver.
- Trip ID (UUID): ID of the requested trip.

Returns

Result (boolean): Represents whether the operation was successful or not.

Rate the Trip

This API will enable customers to rate the trip.

```
rateTrip(customerID: UUID, tripID: UUID, rating: int, feedback?: string): boolean
```

Parameters

- Customer ID (UUID): ID of the customer.

- Trip ID (UUID): ID of the completed trip.
- Rating (int): Rating of the trip.
- Feedback (string): Feedback about the trip by the customer (optional).

Returns

Result (boolean): Represents whether the operation was successful or not.

High-level design

Now let us do a high-level design of our system.

Architecture

We will be using microservices architecture since it will make it easier to horizontally scale and decouple our services. Each service will have ownership of its own data model. Let's try to divide our system into some core services.

Customer Service

This service handles customer-related concerns such as authentication and customer information.

Driver Service

This service handles driver-related concerns such as authentication and driver information.

Ride Service

This service will be responsible for ride matching and quadtree aggregation. It will be discussed in detail separately.

Trip Service

This service handles trip-related functionality in our system.

Payment Service

This service will be responsible for handling payments in our system.

Notification Service

This service will simply send push notifications to the users. It will be discussed in detail separately.

Analytics Service

This service will be used for metrics and analytics use cases.

What about inter-service communication and service discovery?

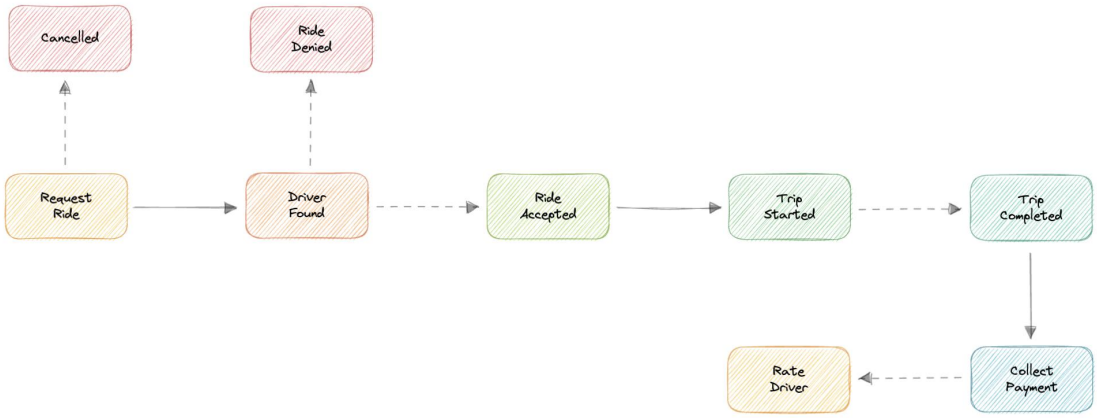
Since our architecture is microservices-based, services will be communicating with each other as well. Generally, REST or HTTP performs well but we can further improve the performance using gRPC which is more lightweight and efficient.

Service discovery is another thing we will have to take into account. We can also use a service mesh that enables managed, observable, and secure communication between individual services.

Note: Learn more about REST, GraphQL, gRPC and how they compare with each other.

How is the service expected to work?

Here's how our service is expected to work:



1. Customer requests a ride by specifying the source, destination, cab type, payment method, etc.
2. Ride service registers this request, finds nearby drivers, and calculates the estimated time of arrival (ETA).
3. The request is then broadcasted to the nearby drivers for them to accept or deny.

4. If the driver accepts, the customer is notified about the live location of the driver with the estimated time of arrival (ETA) while they wait for pickup.
5. The customer is picked up and the driver can start the trip.
6. Once the destination is reached, the driver will mark the ride as complete and collect payment.
7. After the payment is complete, the customer can leave a rating and feedback for the trip if they like.

Location Tracking

How do we efficiently send and receive live location data from the client (customers and drivers) to our backend? We have two different options:

Pull model

The client can periodically send an HTTP request to servers to report its current location and receive ETA and pricing information. This can be achieved via something like Long polling.

Push model

The client opens a long-lived connection with the server and once new data is available it will be pushed to the client. We can use WebSockets or Server-Sent Events (SSE) for this.

The pull model approach is not scalable as it will create unnecessary request overhead on our servers and most of the time the response will be empty, thus wasting our resources. To minimize latency, using the push model with WebSockets is a better choice because then we can push data to the client once it's available without any delay, given that the connection is open with the client. Also, WebSockets provide full-duplex communication, unlike Server-Sent Events (SSE) which are only unidirectional.

Additionally, the client application should have some sort of background job mechanism to ping GPS location while the application is in the background.

Note: Learn more about Long polling, WebSockets, Server-Sent Events (SSE).

Ride Matching

We need a way to efficiently store and query nearby drivers. Let's explore different solutions we can incorporate into our design.

SQL

We already have access to the latitude and longitude of our customers, and with databases like PostgreSQL and MySQL we can perform a query to find nearby driver locations given a latitude and longitude (X, Y) within a radius (R).

```
SELECT * FROM locations WHERE lat BETWEEN X-R AND X+R AND long BETWEEN Y-R AND Y+R
```

However, this is not scalable, and performing this query on large datasets will be quite slow.

Geohashing

Geohashing is a geocoding method used to encode geographic coordinates such as latitude and longitude into short alphanumeric strings. It was created by Gustavo Niemeyer in 2008.

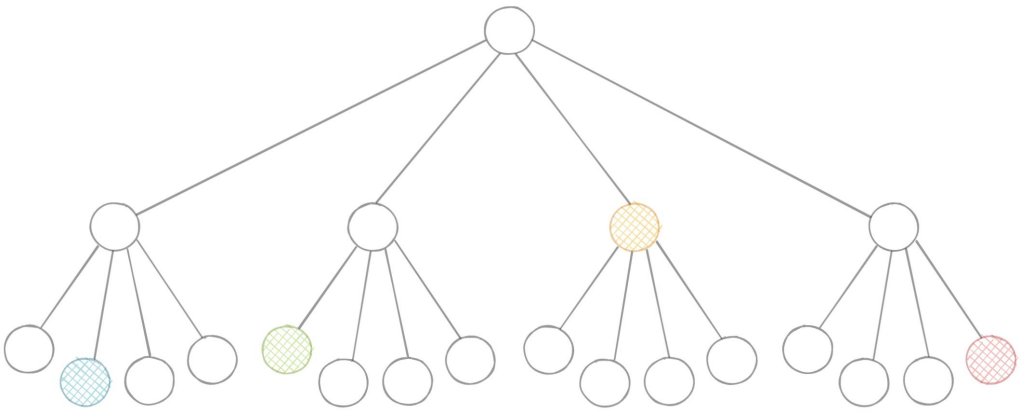
Geohash is a hierarchical spatial index that uses Base-32 alphabet encoding, the first character in a geohash identifies the initial location as one of the 32 cells. This cell will also contain 32 cells. This means that to represent a point, the world is recursively divided into smaller and smaller cells with each additional bit until the desired precision is attained. The precision factor also determines the size of the cell.

For example, San Francisco with coordinates 37.7564, -122.4016 can be represented in geohash as 9q8yy9mf.

Now, using the customer's geohash we can determine the nearest available driver by simply comparing it with the driver's geohash. For better performance, we will index and store the geohash of the driver in memory for faster retrieval.

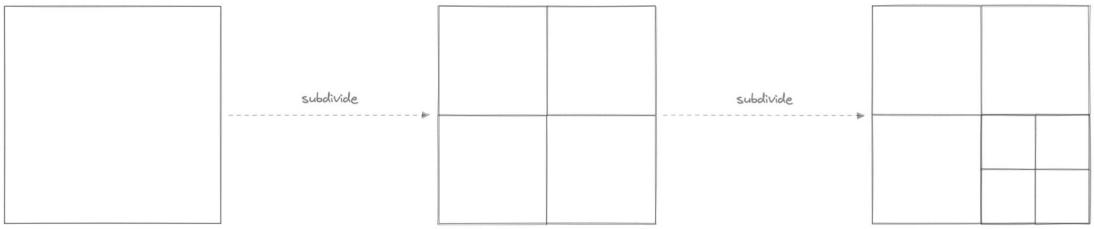
Quadtrees

A Quadtree is a tree data structure in which each internal node has exactly four children. They are often used to partition a two-dimensional space by recursively subdividing it into four quadrants or regions. Each child or leaf node stores spatial information. Quadtrees are the two-dimensional analog of Octrees which are used to partition three-dimensional space.



Quadtrees enable us to search points within a two-dimensional range efficiently, where those points are defined as latitude/longitude coordinates or as cartesian (x, y) coordinates.

We can save further computation by only subdividing a node after a certain threshold.



Quadtree seems perfect for our use case, we can update the Quadtree every time we receive a new location update from the driver. To reduce the load on the quadtree servers we can use an in-memory datastore such as Redis to cache the latest updates. And with the application of mapping algorithms such as the Hilbert curve, we can perform efficient range queries to find nearby drivers for the customer.

What about race conditions?

Race conditions can easily occur when a large number of customers will be requesting rides simultaneously. To avoid this, we can wrap our ride matching logic in a Mutex to avoid any race conditions. Furthermore, every action should be transactional in nature.

How to find the best drivers nearby?

Once we have a list of nearby drivers from the Quadtree servers, we can perform some sort of ranking based on parameters like average ratings, relevance, past

customer feedback, etc. This will allow us to broadcast notifications to the best available drivers first.

Dealing with high demand

In cases of high demand, we can use the concept of Surge Pricing. Surge pricing is a dynamic pricing method where prices are temporarily increased as a reaction to increased demand and mostly limited supply. This surge price can be added to the base price of the trip.

Payments

Handling payments at scale is challenging, to simplify our system we can use a third-party payment processor like Stripe or PayPal. Once the payment is complete, the payment processor will redirect the user back to our application and we can set up a webhook to capture all the payment-related data.

Notifications

Push notifications will be an integral part of our platform. We can use a message queue or a message broker such as Apache Kafka with the notification service to dispatch requests to Firebase Cloud Messaging (FCM) or Apple Push Notification Service (APNS) which will handle the delivery of the push notifications to user devices.

Detailed design

It's time to discuss our design decisions in detail.

Data Partitioning

To scale out our databases we will need to partition our data. Horizontal partitioning (aka Sharding) can be a good first step. We can shard our database either based on existing partition schemes or regions. If we divide the locations into regions using let's say zip codes, we can effectively store all the data in a given region on a fixed

node. But this can still cause uneven data and load distribution, we can solve this using Consistent hashing.

Metrics and Analytics

Recording analytics and metrics is one of our extended requirements. We can capture the data from different services and run analytics on the data using Apache Spark which is an open-source unified analytics engine for large-scale data processing. Additionally, we can store critical metadata in the views table to increase data points within our data.

Caching

In a location services-based platform, caching is important. We have to be able to cache the recent locations of the customers and drivers for fast retrieval. We can use solutions like Redis or Memcached but what kind of cache eviction policy would best fit our needs?

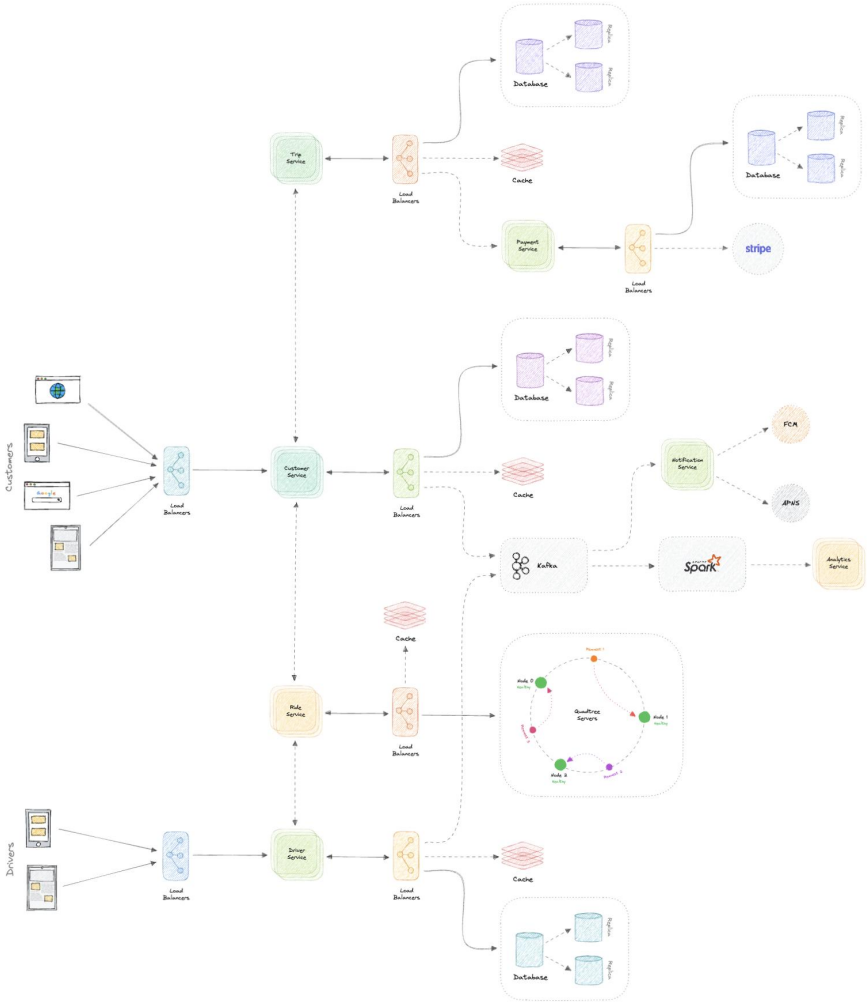
Which cache eviction policy to use?

Least Recently Used (LRU) can be a good policy for our system. In this policy, we discard the least recently used key first.

How to handle cache miss?

Whenever there is a cache miss, our servers can hit the database directly and update the cache with the new entries.

Identify and resolve bottlenecks



Let us identify and resolve bottlenecks such as single points of failure in our design:

- "What if one of our services crashes?"
- "How will we distribute our traffic between our components?"
- "How can we reduce the load on our database?"
- "How to improve the availability of our cache?"
- "How can we make our notification system more robust?"

To make our system more resilient we can do the following:

- Running multiple instances of each of our services.
- Introducing load balancers between clients, servers, databases, and cache servers.
- Using multiple read replicas for our databases.
- Multiple instances and replicas for our distributed cache.
- Exactly once delivery and message ordering is challenging in a distributed system, we can use a dedicated message broker such as Apache Kafka or NATS to make our notification system more robust.

Key Components:

- **Models and Data Structures:** Representing core entities like Customer, Driver, Ride, Trip, etc.
- **Microservices Design**: Each service will handle different functionalities (Customer, Driver, Ride, Payment, etc.).
- **Ride Matching Algorithm**: Efficiently finding nearby drivers using Geohashing or Quadtrees.
- **APIs for Communication:** For ride requests, cancellations, driver acceptance, etc.
- **Service Discovery:** Microservices will communicate via REST or gRPC.
- **Location Updates:** Efficiently sending and receiving GPS data from clients using WebSockets.

Let’s start with the basic Java classes and core APIs that form the foundation of this system.

1. Data Models

These are the basic entities for the application.

```
import java.util.UUID;

public class Customer {
	UUID customerId;
	String name;
	String email;
	double latitude;
	double longitude;

	// Getters and Setters
}

public class Driver {
	UUID driverId;
	String name;
	String email;
	double latitude;
	double longitude;
	boolean isAvailable;

	// Getters and Setters
}

public class Ride {
	UUID rideId;
	UUID customerId;
	UUID driverId;
	String source;
	String destination;
	String status; // e.g., "Pending", "In Progress", "Completed"
	double fare;
```

```
    // Getters and Setters
}

public class Trip {
    UUID tripId;
    UUID rideId;
    String status; // e.g., "In Progress", "Completed"
    double distance;
    double fare;

    // Getters and Setters
}

public class Payment {
    UUID paymentId;
    UUID rideId;
    double amount;
    String paymentMethod; // Credit Card, PayPal, etc.
    String status; // e.g., "Pending", "Completed"

    // Getters and Setters
}
```

2. Ride Request and Matching Algorithm

This is the critical functionality where customers request rides, and we find the nearest drivers.

```
import java.util.List;
import java.util.UUID;
import java.util.stream.Collectors;

public class RideService {
    List<Driver> availableDrivers; // This will be fetched from a DriverService or database
```

```
    // Customer requests a ride
    public Ride requestRide(Customer customer, String destination, String cabType, String paymentMethod) {
        // Match the ride request with the nearest available driver
        Driver nearestDriver = findNearestDriver(customer);

        // Create a new ride and associate the customer and driver
        Ride newRide = new Ride();
        newRide.rideId = UUID.randomUUID();
        newRide.customerId = customer.customerId;
        newRide.driverId = nearestDriver.driverId;
        newRide.source = customer.latitude + ", " + customer.longitude;
        newRide.destination = destination;
        newRide.status = "Pending";
        newRide.fare = calculateFare(customer, nearestDriver); // Implement fare calculation logic

        // Mark driver as unavailable
        nearestDriver.isAvailable = false;

        // Save the ride request (Could be saved in a RideRepository)

        return newRide;
    }

    // Find nearest driver using geohashing/quadtree (Simple placeholder for now)
    private Driver findNearestDriver(Customer customer) {
        // This could be a more complex geohashing/quadtree-based approach
        return availableDrivers.stream()
            .filter(Driver::isAvailable)
            .min((d1, d2) -> Double.compare(
                distance(customer.latitude, customer.longitude, d1.latitude, d1.longitude),
                distance(customer.latitude, customer.longitude, d2.latitude, d2.longitude)))
            .orElseThrow(() -> new RuntimeException("No available drivers"));
    }
```

```
    // Simple haversine distance calculation (for demonstration)
    private double distance(double lat1, double lon1, double lat2, double lon2) {
        final int R = 6371; // Radius of the earth in km
        double latDistance = Math.toRadians(lat2 - lat1);
        double lonDistance = Math.toRadians(lon2 - lon1);
        double a = Math.sin(latDistance / 2) * Math.sin(latDistance / 2)
                + Math.cos(Math.toRadians(lat1)) * Math.cos(Math.toRadians(lat2))
                * Math.sin(lonDistance / 2) * Math.sin(lonDistance / 2);
        double c = 2 * Math.atan2(Math.sqrt(a), Math.sqrt(1 - a));
        double distance = R * c; // Convert to km
        return distance;
    }

    // Fare calculation logic (simple placeholder)
    private double calculateFare(Customer customer, Driver driver) {
        double distance = distance(customer.latitude, customer.longitude, driver.latitude, driver.longitude);
        return distance * 10; // Assuming fare is 10 units per km
    }
}
```

3. Driver Accepting a Ride

This service allows drivers to accept or deny the ride requests.

```
public class DriverService {

    // Driver accepts a ride request
    public boolean acceptRide(UUID driverId, UUID rideId) {
        // Fetch the ride based on the rideId
        Ride ride = getRideById(rideId);

        if (ride == null || ride.status.equals("In Progress") || ride.status.equals("Completed")) {
            return false; // Ride cannot be accepted if it's already in progress or completed
        }
```

```
        // Change the status of the ride
        ride.status = "In Progress";
        // Update the driver availability (Mark the driver as unavailable)
        Driver driver = getDriverById(driverId);
        driver.isAvailable = false;

        return true; // Ride accepted successfully
    }

    // Driver denies the ride
    public boolean denyRide(UUID driverId, UUID rideId) {
        Ride ride = getRideById(rideId);

        if (ride == null || ride.status.equals("In Progress") || ride.status.equals("Completed")) {
            return false;
        }
        // Ride remains pending, driver availability is restored
        Driver driver = getDriverById(driverId);
        driver.isAvailable = true;

        return true;
    }

    private Ride getRideById(UUID rideId) {
        // Simulate fetching ride from database
        return new Ride(); // Placeholder
    }

    private Driver getDriverById(UUID driverId) {
        // Simulate fetching driver from database
        return new Driver(); // Placeholder
    }
}
```

4. Location Tracking (Push Model)

For real-time location tracking of drivers and customers, we can use WebSockets. Here's how it could look:

```
import javax.websocket.*;
import java.io.IOException;

@ClientEndpoint
public class LocationWebSocketClient {
    private Session session;

    @OnOpen
    public void onOpen(Session session) {
        this.session = session;
        System.out.println("Connected to server");
    }

    @OnMessage
    public void onMessage(String message) {
        // Handle incoming location updates from the server
        System.out.println("Received location update: " + message);
    }

    @OnClose
    public void onClose() {
        System.out.println("Connection closed");
    }

    @OnError
    public void onError(Throwable error) {
        error.printStackTrace();
    }

    public void sendLocation(double latitude, double longitude) {
        String message = "Location: " + latitude + ", " + longitude;
```

```
            try {
                session.getBasicRemote().sendText(message);
            } catch (IOException e) {
                e.printStackTrace();
            }
        }
}
```

5. Payment Service

Basic payment logic to handle trip payments.

```
public class PaymentService {

    // Process payment
    public boolean processPayment(UUID rideId, double amount, String paymentMethod) {
        Payment payment = new Payment();
        payment.paymentId = UUID.randomUUID();
        payment.rideId = rideId;
        payment.amount = amount;
        payment.paymentMethod = paymentMethod;
        payment.status = "Completed"; // Assume payment is successful

        // Save payment to the database (could be a repository)

        return true;
    }
}
```

6. Ride Rating (Post-Trip)

After the ride, customers can rate the trip.

```
public class RatingService {

    // Rate a trip
    public boolean rateTrip(UUID customerId, UUID tripId, int rating, String feedback) {
        // Check if trip is completed
        Trip trip = getTripById(tripId);

        if (trip == null || !trip.status.equals("Completed")) {
            return false; // Trip must be completed before rating
        }

        // Save the rating and feedback (could be saved in a RatingRepository)
        System.out.println("Rating received: " + rating);

        return true;
    }

    private Trip getTripById(UUID tripId) {
        // Simulate fetching trip details from a database
        return new Trip(); // Placeholder
    }
}
```

7. Final Integration

The final integration involves combining these microservices using service discovery, API calls, etc. Depending on traffic, scalability, and performance, the communication between these services will be optimized using REST/gRPC.