To achieve your goal of finding two strangers with the same destination, whose start locations are close and whose routes overlap, you’ll need to develop a system that involves several components. Here’s a structured approach to help you design and implement this solution:

**1. System Overview**

1. **User Location and Destination Data**:
   * Each user provides their starting location and destination.
2. **Calculate Routes**:
   * For each user, calculate the route from their starting location to the destination using the Google Maps Directions API.
3. **Determine Proximity**:
   * Check if the starting locations of users are close to each other.
4. **Check Route Overlap**:
   * Compare the routes to determine if they overlap significantly.
5. **Match Users**:
   * Find pairs of users who meet the criteria and notify them.

**2. Detailed Steps**

**2.1. Collect User Data**

* Each user inputs their start location and destination.
* Example Data Structure:

json

Copy code

{

"userId": "user1",

"startLocation": { "lat": 37.422, "lng": -122.084 },

"destination": { "lat": 37.331, "lng": -122.031 }

}

**2.2. Calculate Routes Using Google Maps API**

* **Request Route Information**: Use the Google Maps Directions API to get the route information for each user.

**Example API Request:**

http

Copy code

https://maps.googleapis.com/maps/api/directions/json?origin=37.422,-122.084&destination=37.331,-122.031&key=YOUR\_API\_KEY

* **Response**: The response will include a polyline representing the route, which you’ll need to decode and analyze.

**2.3. Determine Proximity**

* **Distance Calculation**: Calculate the distance between the starting locations of two users using the Haversine formula or a similar method.

javascript

Copy code

function getDistance(lat1, lon1, lat2, lon2) {

const R = 6371; // Radius of the Earth in kilometers

const dLat = (lat2 - lat1) \* (Math.PI / 180);

const dLon = (lon2 - lon1) \* (Math.PI / 180);

const a = Math.sin(dLat / 2) \* Math.sin(dLat / 2) +

Math.cos(lat1 \* (Math.PI / 180)) \* Math.cos(lat2 \* (Math.PI / 180)) \*

Math.sin(dLon / 2) \* Math.sin(dLon / 2);

const c = 2 \* Math.atan2(Math.sqrt(a), Math.sqrt(1 - a));

return R \* c; // Distance in kilometers

}

* **Proximity Threshold**: Define a threshold distance to consider two starting locations as "close" (e.g., within 1 km).

**2.4. Check Route Overlap**

* **Decode Polylines**: Google Maps API returns encoded polylines which you need to decode into a series of latitude/longitude points.

javascript

Copy code

// Use a polyline library to decode

import polyline from '@mapbox/polyline';

const points = polyline.decode(encodedPolyline);

* **Route Comparison**: Compare the decoded routes for significant overlap. This can be complex and might involve spatial analysis techniques:
  + **Bounding Box Intersection**: Check if the bounding boxes of the routes overlap.
  + **Line Segment Intersection**: More detailed analysis of route segments to check for intersections.

**Libraries and Techniques**:

* + **Turf.js**: A JavaScript library for spatial analysis which can help with geo calculations and route comparison.

**2.5. Match Users**

* **Find Matching Pairs**: Create a matching algorithm that pairs users based on proximity and route overlap.
* **Notify Users**: Inform matched users that they can travel together. You can use push notifications, emails, or in-app messages.

**3. Example Workflow**

1. **User Registration**:
   * Users input their start location and destination.
   * Store this information in your database.
2. **Calculate and Compare**:
   * Periodically (or on-demand), calculate routes for all users.
   * Compare routes and starting locations based on your criteria.
3. **Notify Users**:
   * Send notifications to users who match the criteria.

**4. Technology Stack**

* **Frontend**: React or React Native for user interface.
* **Backend**: Node.js, Django, or another backend framework to handle API requests and logic.
* **Database**: Store user data and matching results (e.g., PostgreSQL, MongoDB).
* **Geospatial Analysis**: Use libraries like Turf.js for route analysis and spatial operations.
* **Notifications**: Implement notifications using Firebase Cloud Messaging or similar services.

**Summary**

1. **Collect user data** and calculate routes using the Google Maps Directions API.
2. **Determine proximity** between start locations.
3. **Analyze route overlap** using decoded polylines and spatial techniques.
4. **Match and notify** users who can travel together.

This approach involves integrating various technologies and techniques to ensure effective route and user matching. If you have any specific questions or need further clarification on any part of this process, feel free to ask!

Top of Form

mvn spring-boot:run

LK

Skip navigation

Top of Form



Bottom of Form

[Sign in](https://accounts.google.com/ServiceLogin?service=youtube&uilel=3&passive=true&continue=https%3A%2F%2Fwww.youtube.com%2Fsignin%3Faction_handle_signin%3Dtrue%26app%3Ddesktop%26hl%3Den%26next%3D%252Fwatch%253Fv%253Dqhz27TZYsYI%2526pp%253DygUmdHJhbnNhY3Rpb25hbCBkYXRhIHNhdmUgc3FsIHNwcmluZ2Jvb3Q%25253D&hl=en&ec=65620" \o "" \t ")

8:07 / 19:02

•

Service Layer

# Keep your Data CORRECT: How to use Transactions in Spring Boot:

<https://www.youtube.com/watch?v=qhz27TZYsYI>

# How FastAPI Handles Requests Behind the Scenes

<https://www.youtube.com/watch?v=tGD3653BrZ8>

# Introduction to Spring boot | Its Advantage over Spring MVC and Servlets based Web applications

<https://www.youtube.com/watch?v=TMX1kQ4W4hI>

# Spring AOP Tutorial

<https://www.youtube.com/watch?v=Ft29HgsePfQ>

# Spring boot @Transactional Annotation - Part1

<https://www.youtube.com/watch?v=Kf-gAW8hGQA&list=PL6W8uoQQ2c60g6_fcjDCLHSx1LBeVYqyZ&index=13>

<dependency>

<groupId>com.google.code.gson</groupId>

<artifactId>gson</artifactId>

<version>2.8.9</version>

</dependency>