

ABSTRACT

Public transportation remains a critical component of urban mobility, particularly in developing nations where a large segment of the population relies on affordable transit options such as auto-rickshaws and buses. However, the current systems in place are plagued by persistent inefficiencies that adversely affect both passengers and drivers. Chief among these challenges are the long waiting periods before vehicles depart, the tendency to delay trips until maximum occupancy is reached, and the frequent unscheduled stops in search of additional passengers. These operational flaws lead to extended travel times, commuter dissatisfaction, lost productivity, increased fuel consumption, and higher levels of air pollution due to prolonged idling.

This project presents a comprehensive solution to these issues through the development of an innovative, scalable, and user-centric Android application designed to enhance the efficiency, reliability, and convenience of public transportation services. The proposed system functions as a dual-interface platform: one interface for passengers and another for drivers. Passengers can register their intent to travel, specify routes, receive estimated times of arrival (ETA), and track vehicles in real time. Simultaneously, drivers gain access to real-time passenger information along their route, enabling them to make informed decisions, reduce idle times, and optimize their trips for maximum efficiency.

The application incorporates essential features such as live vehicle tracking, dynamic route management, digital payment integration, and real-time notifications, all aimed at minimizing delays and improving the commuting experience. It is also built with inclusivity in mind; while it operates primarily through a smartphone app, the system is designed to accommodate users with limited access to smart devices via potential future integration of SMS-based notifications or voice prompts.

Backed by a robust, cloud-based backend infrastructure, the system ensures high availability and the ability to manage high user loads, particularly during peak commuting hours. Additionally, the platform is designed with strong security and privacy protocols, including encrypted communication and secure authentication, to safeguard user data. Analytics and feedback mechanisms are integrated to continuously assess performance and user satisfaction, paving the way for iterative improvements.

Inspired by the efficiency of private ride-hailing services like Uber and Ola, this project seeks to bring similar technological advancements to public transport—without compromising affordability. By introducing real-time coordination, reducing operational inefficiencies, and enhancing the commuter experience, the project aims to make public transportation a more attractive and dependable alternative to personal vehicles and reserved cabs. In doing so, it supports broader goals such as reducing urban traffic congestion, lowering carbon emissions, and contributing to more sustainable and equitable urban development.

Ultimately, this project envisions a future where public transport is not only accessible and affordable but also smart, efficient, and environmentally responsible—benefiting passengers, drivers, and the city as a whole.

Contents

Contents	2
List of Figures	3
1. Introduction.....	4
1.1. Problem Statement.....	4
1.2. Inspiration.....	4
1.3. Proposed Solution.....	5
2. Literature Survey	7
3. User Stories and Requirement Gathering	8
4. Wireframes	9
5. Class UML Diagram	11
6. Initial Approach	12
7. Implementation	13
8. Top-to-down Implementation Approach.....	19
9. Virtualization of Public Transport Hubs:.....	24
9.1. Concept:	24
9.2. Implementation:.....	24
9.2.1. Route Registration Feature:	26
9.2.2. Insertion of Driver and Passenger in Registered Route:.....	29
9.2.3. Autocomplete Feature:	31
9.2.4. Live Tracking:.....	33
9.3. Implementation Architecture and System Design:	35
9.3.1. System Design Flowchart:.....	35
9.3.2. Database Schema:	35
9.3.3. Workflow:.....	36
10. The Intelligent Routing System:.....	41
10.1. Concept:	41
10.2. Implementation:.....	42
11. Future Scope and Implementation:	44
12. Conclusion:	44
13. Appendix:	46

List of Figures

1. Figure 4.1: Passenger End Wireframe.....	14
2. Figure 4.2: Driver End Wireframe	15
3. Figure 5.1: Class UML Diagram of the App.....	16
4. Figure 7.1: AWS EC2 Instance hosting the server	20
5. Figure 7.2: Google Cloud Console – Showing API statistics for Map and Directions APIs	21
6. Figure 8.1: Initial Location Plotting using Google Colab and Python	24
7. Figure 8.2: Live tracking w.r.t a reference	25
8. Figure 8.3: Live tracking w.r.t reference	26
9. Figure 8.4: The Passenger App UI	26
10. Figure 8.5: Updated Passenger App UI	27
11. Figure 8.6: Initial Route on Request	28
12. Figure 8.7: Updated Route	28
13. Figure 8.8: Driver End of the Application	28
14. Figure 9.1: Redis showing the SHA256 encoded Polyline	29
15. Figure 9.2: Redis Side showing the Sorted Set Route_ID : Start_Hub_ID : Passenger	30
16. Figure 9.3: Redis Side showing the In Transit Queue for Vehicles in In Transit mode	31
17. Figure 9.4: Routes Select Activity	31
18. Figure 9.5: Route Registration with information	32
19. Figure 9.6: Route Registered by the Driver	33
20. Figure 9.7: Redis view of Driver being added to the Stationary Queue	34
21. Figure 9.8: Redis view of Driver leaving Hub and being added to In-Transit Queue	35
22. Figure 9.9: Passenger Selecting the preferred route	36
23. Figure 9.10: Redis Client Side showing the Sorted Set with Key as City : Hub	37
24. Figure 9.11: Passenger Client-Side showing Autocomplete in action	37
25. Figure 9.12: Redis view of the Driver's Latitude and Longitude being sent for Live Tracking.....	38
26. Figure 9.13: Live Tracking between the Driver and Passenger starts	39
27. Figure 9.14: Live Tracking stops once the vehicle crosses the passenger	39
28. Figure 9.15: System Design Architecture Diagram	40
29. Figure 9.16: Database Schema for the application	40
30. Figure 9.17: API Gateway integrated with AWS API Gateway	41
31. Figure 9.18: Process RouteSelect in action	42
32. Figure 9.19: AWS Cloudwatch live tail showing Process RouteSelect payload	43
33. Figure 9.20: Hash set using key Hub : Hub ID	43
34. Figure 9.21: AWS DynamoDB	45
35. Figure 9.22: Redis in-memory database	45
36. Figure 9.23: Graph Database using Neo4j	46
37. Figure 10.1: Intelligent Routing System	47
38. Figure 10.2: Intelligent Routing Response	48

1. Introduction

1.1. Problem Statement

Public transport systems, particularly in densely populated urban towns and cities, have been long disturbed with inefficiencies that affect both passengers and drivers. The most prominent issue results from the waiting period before vehicles such as auto-rickshaws and buses initiate their journeys. These transport entities frequently delay departures until they have reached their full passenger capacity. This leads to unnecessary stagnation, increasing the time passengers spend waiting and contributing to traffic congestion.

Moreover, these vehicles tend to stop at various junctions for uncertain durations in hopes of filling up remaining seats, which creates an unpredictable and frustrating experience for commuters. This kind of inefficiency not only causes delays but also results in wasted fuel and increased air pollution due to prolonged idling. In areas with high passenger demand, such as business districts during peak hours, the delays become even more pronounced, compounding the inconvenience for commuters.

As urban populations grow and cities expand, the demand for reliable and punctual public transportation systems has surged. However, despite the increasing need, the system continues to falter under its outdated operational methods. The inefficiencies affect not just the passengers who rely on these services for daily commutes, but also the drivers whose income is contingent upon the number of completed trips. In some cases, drivers may waste significant time waiting for passengers, during which they could have completed multiple short trips, had the system been more organized.

A major repercussion of these inefficiencies is the shift in commuter preference from public transport to reserved cabs and other personalized modes of travel. Unlike auto-rickshaws and buses, reserved cabs often provide quicker and more predictable services. This shift not only reduces the usage of cost-effective public transport but also increases the number of private vehicles on the road, thereby adding traffic congestion and pollution.

1.2. Inspiration

The idea for this project was born out of observing the daily frustrations faced by both passengers and drivers using public transportation. In cities across the world, especially in developing countries, the inefficiencies in public transport systems have become a common grievance. Whether it's the long waiting times, much to be desired in terms of convenience and reliability.

For instance, many commuters find themselves at the mercy of bus schedules that are not adhered to or auto-rickshaws that refuse to leave until every seat is filled. These issues are not just inconvenient; they have broader social and economic implications. People end up being late for work, students miss classes, and essential appointments are delayed. This everyday struggle showcases a clear gap between what the public transport system offers and what commuters actually need.

Furthermore, drivers too are not immune to the inefficiencies. Their income is often hampered by the unpredictable nature of passenger availability and then time lost waiting at stands or making unnecessary stops. In the absence of a streamlined system, drivers are forced to make judgement calls that may or may not yield

passengers, resulting in wasted fuel and missed opportunities in earnings.

Another source of inspiration came from examining how private cab services like Uber and Ola have revolutionized urban transport. These services have effectively used technology to reduce waiting times, provide real-time updates and match supply with demand efficiently. However, their services come at a premium cost, which is not affordable for the average commuter who relies on public transportation.

This disparity highlights an opportunity to bring similar technological advancements to the public transit sector, making it more competitive while retaining its affordability. If public transport can incorporate some of the efficiencies seen in private cab services – such as real-time communication, dynamic routing, and passenger notifications – it can significantly enhance user satisfaction and operational effectiveness.

This disparity highlights an opportunity to bring similar technological advancements to the public transport sector, making it more competitive while retaining its affordability. If public transport can incorporate some of the efficiencies seen in private cab services—such as real-time communication, dynamic routing, and passenger notifications—it can significantly enhance user satisfaction and operational effectiveness.

The push for sustainable urban development also serves as a catalyst for this project. Cities worldwide are looking for ways to reduce carbon footprints and traffic congestion, and improving public transport is a crucial step in this direction. By optimizing how these systems function, we not only improve commuter experience but also contribute to environmental sustainability.

Finally, personal experiences and anecdotal evidence have reinforced the need for change. Numerous stories of missed appointments, long waits, and frustrating travel experiences have underscored the urgency to innovate. The realization that these problems are systemic and solvable through a well-designed application has fuelled the motivation behind this project.

In essence, the inspiration for this project stems from a combination of observed inefficiencies, comparative analysis with private transport models, and a strong desire to improve daily commuting for the common person.

1.3. Proposed Solution

To address the inefficiencies inherent in current public transportation systems, we propose a robust, scalable, and user-friendly Android application that serves both passengers and drivers. The application is designed to streamline communication and coordination, thereby enhancing the efficiency and reliability of public transport services.

At its core, the application functions as a two-pronged platform. One interface caters to the needs of passengers, while the other is tailored for drivers. The passenger interface allows users to notify their presence and intent to travel along a specific route. This notification is then relayed to the nearest and most suitable driver operating in that area, allowing for timely pickups and minimizing waiting time.

For drivers, the application provides real-time information about potential

passengers along their route. This enables them to plan their journey more effectively, reducing idle time and ensuring more efficient route management. With an optimized flow of information, drivers can avoid unnecessary stops, fuel wastage, and delays, thereby improving their earning potential.

Additionally, the app includes features such as estimated time of arrival (ETA), route tracking, and digital payment options to further enhance the user experience. Passengers can track their assigned vehicle in real-time, receive updates about delays or changes, and make payments seamlessly through the app. These features collectively contribute to a more predictable and hassle-free commuting experience.

Another key aspect of the application is its ability to adapt to varying levels of technological penetration. While the primary platform is a smartphone-based Android app, the system is designed to be scalable and can potentially integrate with other communication methods, such as SMS-based updates or voice call prompts, ensuring inclusivity for users who may not own smartphones.

The backend architecture of the application is built for high availability and scalability. Using cloud-based services, the platform can handle large volumes of data and user interactions, ensuring smooth performance even during peak hours. The system also employs data analytics to monitor performance, gather user feedback, and continuously improve the service.

Security and privacy are integral components of the solution. The application uses secure authentication methods and encrypted communication to protect user data. User feedback mechanisms are built in to report any issues or concerns, thereby maintaining a safe and trustworthy platform.

In a broader sense, the application aims to modernize public transportation by incorporating the benefits of digital technology. By reducing delays, optimizing routes, and improving communication, the app offers a sustainable alternative to private cabs, making public transport more appealing to a wider audience.

Ultimately, the goal is to create a system that not only solves existing problems but also lays the foundation for future improvements in urban mobility. With the successful deployment of this application, cities can expect reduced traffic congestion, better air quality, and a more satisfied commuting public, all while supporting the livelihoods of public transport drivers.

2. Literature Survey

One of the most crucial aspects of urban planning and management is the optimization of the public transportation system. The main aim of it is to enhance the efficiency, accessibility and sustainability of transportation networks. Over time, there has been numerous studies that have explored various methodologies and techniques to optimize public transport systems, accessing multimodal networks, and evaluation of public transport systems in urban areas.

Mandl (1980) takes one of the first historical approach with his study *Evaluation and Optimization of Urban Public Transportation Networks* [1], which was published in the *European Journal of Operational Research*. This article focuses on the evaluation of urban public transportation networks, with an emphasis on optimizing routes, schedules, and frequency of services. He introduced various operational researching techniques that could be used to optimize these identified parameters, resulting in improved overall performance of public transport. The research article presented several case studies that portray how different optimization models can give significant improvements both in operational efficiency and customer satisfaction.

Jansson (1984) in this monograph *Transport System Optimization and Pricing* [2] offers a foundational perspective on the economical and operational aspect of transportation systems. He emphasized the importance of optimization of public transportation systems for multiple reasons. Though his work is mostly on the pricing infrastructure of public transportation, it helps in understanding the behavior of the entities in a public transportation system and how one can utilize these behavioral patterns to one's ideas.

Zimmerman and Fang (2015) discuss the integration and optimization of public transport services in their paper *Public Transport Service Optimization and System Integration* [3]. This research focuses on the behavior of different component of a public transport system and how better integrated to improve service efficiency. The researchers have argued in their article that system integration is a major factor in achieving optimized transport services. By studying the interaction between different transportation entities and routes, Zimmerman and Fang explore methods for improving coordination, reducing waiting times, and increasing service reliability. The work is relevant especially in the fast-growing congestion of urban cities and helps in brainstorming different ways to overcome slowing down of traffic and improving the flow of traffic and public transport.

Over the last few years, performance of multimodal transportation systems have gathered a lot of emphasis by researchers. Tanwar and Agarwal (2024) provide a contemporary approach to accessing travel time performance in multimodal transportation networks using fuzzy-analytic hierarchy process (FAHP) [4]. Their case study on Bhopal City demonstrates how their fuzzy-based methodology can be applied to evaluate and compare travel time performance across different models of transportation, be it buses, trains, and private vehicles. The researchers emphasize the significance of considering uncertainty and subjectivity in evaluation of transportation systems, which is addressed effectively through the FAHP. This research is vital in understanding the metrics of performance in transportation systems and factoring leading to congestion in cities.

Together, these studies underscore the importance of strategic tools and sustainable platforms in public transport optimization.

3. User Stories and Requirement Gathering

3.1. User Stories

Primary Users:

- Passengers
- Drivers

Secondary Users:

- Developers
- Testers
- System Operators

3.2. Requirement Gathering

After identifying the user stories, we had to understand what are the requirements that need to be met by the application. Requirement gathering is the identification of the different functionalities from the perspective of the Primary user stories that should be included in the application.

- Passengers:
 - I must be able to notify my location to the public transport drivers who are operating on that very route.
 - I must be able to reserve a seat in case I am in a hurry.
 - I must be able to see the vehicle approaching me on the route in the maps fragment of the app.
- Drivers:
 - I should be notified by the application whenever a potential passenger is present in my operating route.
 - I should be able to get notified if any passenger is trying to reserve a seat in the vehicle.
 - I should be able to see the live passenger count of the potential number of passengers in the route up ahead.
 - I should be able to choose my route ID for a particular day.
 - I should be able to start my day's drive on clicking the 'Start Drive' button.

4. Wireframes

4.1. Application in perspective of Passenger

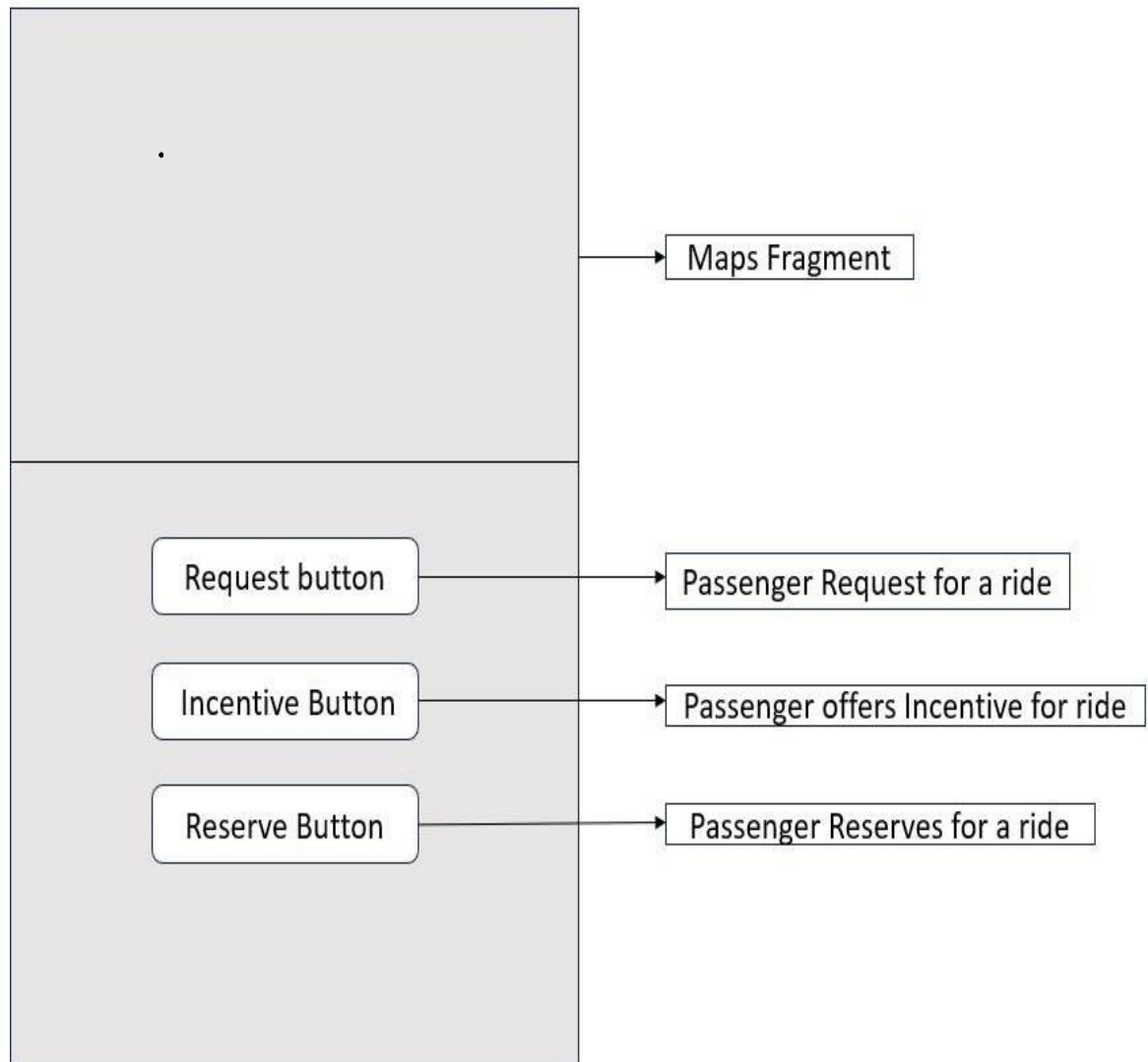


Figure 4.1: Passenger End Wireframe

4.2. Application in perspective of Driver

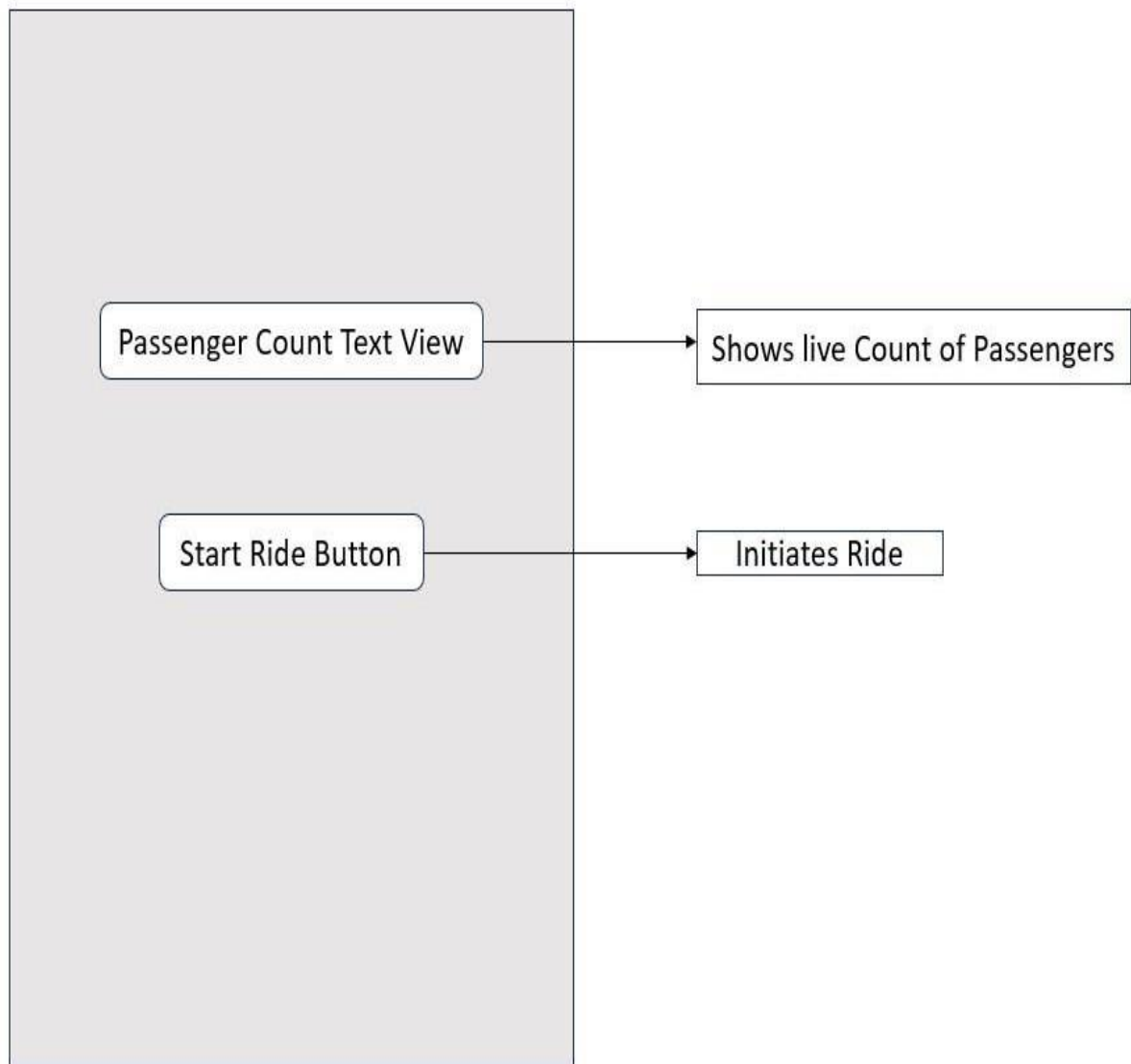


Figure 4.2: Driver End Wireframe

5. Class UML Diagram

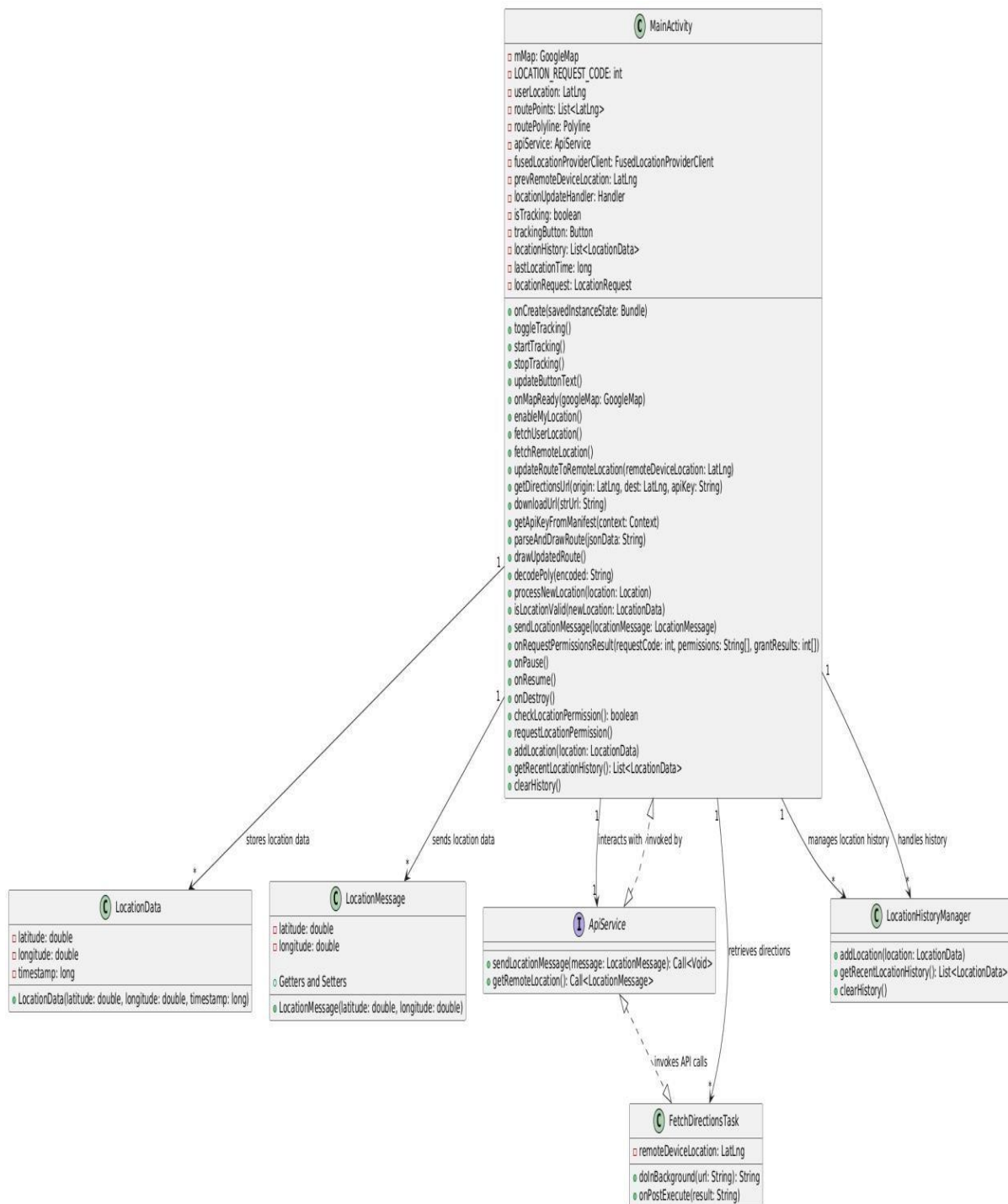


Figure 5.1: Class UML Diagram of the App

A Class UML Diagram provides the information about the relationships between the different components present in the application. It not just provides a high-level view of the interaction between the components and procedures, but also help us visualize the type of relationship two components/procedures share between them.

A Class UML Diagram is also useful in understanding the data and control flow in the application as to how the data flows through the application when it is being run on the server. This aspect of essential for developers as this provides them insights on to whether their work satisfies the requirements of the application or not, and accordingly they can plan the sprints to fix the application.

6. Initial Approach

6.1. From the Driver's Perspective

- *Auto/E-Rickshaw* generally tend to have a *chowk/stand* from where they collect passengers and go on to their different routes.
- Generally, by word-of-mouth, they follow an informal queue system to ensure fairness.
- We plan to utilize this queue system they have and virtualize it.
- The virtualization shall be implemented as follows:
 - The driver must be within 20 metres of the *chowk/stand* to be eligible for getting passengers from the application.
 - Once the vehicle is within 20 metres of the designated *chowk/stand*, the application logic will put that vehicle id into the queue. For example, If the vehicle is at position 5 in the queue, its driver will be able to see passengers only after the 5 vehicles before him have collected passengers or left the *chowk/stand*.
 - The driver can collect as many passengers as possible according to vehicle capacity at the stand. He will be able to see potential passengers in the route and will collect them up ahead, if vacancies are available, rather than waiting at the stand for longer time. Thus, saving time and increasing chances for assured income for the driver.
- If the passenger requests for reserving a seat, it will only be sent to the driver second in queue. The driver will have the option to accept or reject, with the request being forwarded to the next in queue in case the driver rejects the reserve request.

6.2. From Passenger's Perspective

- Due to their cheap cost, passengers prefer to use public transport. But uncertainty of their timing and availability, they steer towards reserved cabs, which can be costly.
- The passengers simply have to open the app on their smartphones and click on the **Request** button. This will notify the driver first in the queue about the existence of a potential passenger up ahead in their route.
- The passengers will be able to view the approaching vehicle on the map, thus increasing the assurance of transport.
- In case the passenger is in a bit of hurry, they can choose for **Reserve** at a higher incentive, which will further increase the chance of getting a transport.

7. Implementation

7.1. Driver End Application

7.1.1. Functional Requirements

- It sends its location information to the server using a queue data structure where it is stored.
- It shows the driver the count of the passengers ahead based on their unique id.

7.1.2. Non-functional Requirements

- The application considers the vehicle to be a legitimate one, if it travels at a speed lower than 100 km/hr.
- Abrupt change in location is avoided if the change in distance of the last sent location and to-be-sent new location varies extensively.

7.1.3. System Design

- Using Android Studio, we developed the Java XML framework that encapsulates the functionality of live streaming of driver's location to the server.
- For this, the app required permissions of device location which is asked to the user on the first installation. The *AndroidManifest.xml* file extracts all the required permissions for the application and that is called in the *MainActivity.java* file to verify the permissions.
- Upon successfully getting all permissions, *LocationServices* of Android provides functionality to get the device's current latitude and longitude.
- Retrofit was used for HTTP communication, with *GsonConverterFactory* for parsing the JSON responses. A custom *okHttpClient* was configured with optimized timeouts and retry policies for network resilience.

- A Textview was implemented to dynamically display passenger count that is being fetched by the server. A Handler has been used to schedule periodic API requests (once in every 10 seconds) for updating the count using the *fetchPassengerCount()*.
- A high-frequency location update system has been configured using *LocationRequest* with a 1-second interval and 0.5-second minimum update interval. A location validation logic has been put to filter the abrupt changes in location based on speed and distance thresholds.
- A *LocationData* class has been placed to store latitude, longitude and timestamps for each location update. The *Location.distanceBetween()* function calculates distances and derives speed for anomaly detection.
- Thus, a modular approach has been implemented, integrating various components for location tracking, REST API communication and UI updates.

7.2. Passenger End Application

7.2.1. Functional Requirements

- It sends a request to the driver end, when the passenger clicks on the **Request** button.
- In case the passenger clicks on the **Reserve** button, a reserve request is sent to the server.
- It fetches location of the driver from the server queue, where the driver location is updated periodically.
- It first fetches the route between the passenger and the driver and plots it on the dynamic maps fragment. Using polyline, this route is plotted on the maps fragment.
- This location of the driver is live tracked and the plot is updated accordingly.

7.2.2. Non-functional Requirements

- To reduce load on the server by consistent call of Google Directions API, the polyline is constantly updated such that the previously crossed locations by the driver are removed from the map resulting in lesser API calls. This results in an updated polyline, with lesser API calls.
- The polyline is updated every 5 seconds.

7.2.3. System Design

- Android Studio was used as a development environment. The language used was Java.
- Dependencies and external libraries managed using Gradle.

- Google Maps API was used for map visualization and routing, and Retrofit was used to handle REST API communication.
- An *OkHttp* component was added for network resilience with custom timeouts. For efficient and accurate location a *FusedLocationProviderClient* was added.
- Dynamic routing using Google Maps API to done to fetch routes dynamically based on user and remote device locations. The plotted polylines will be updated periodically as locations change from the driver end. Error handling has been done for failed API calls to ensure that the user experience is not compromised.
- Route optimization was incorporated via programming logic to adjust routes based on the proximity of user and remote locations. Caching has been used to cache the last route and only updating changes. This avoided a lot of redundant calculations.
- Map markers have been used to indicate user and remote locations with dynamic titles and icons.
- *Location.distanceBetween()* has been used to calculate speed and distance between consecutive updates, filtering out invalid data. Also a fallback mechanism have been implemented to handle location unavailability or abrupt changes.

7.3. GCP and AWS Server

7.3.1. AWS Server

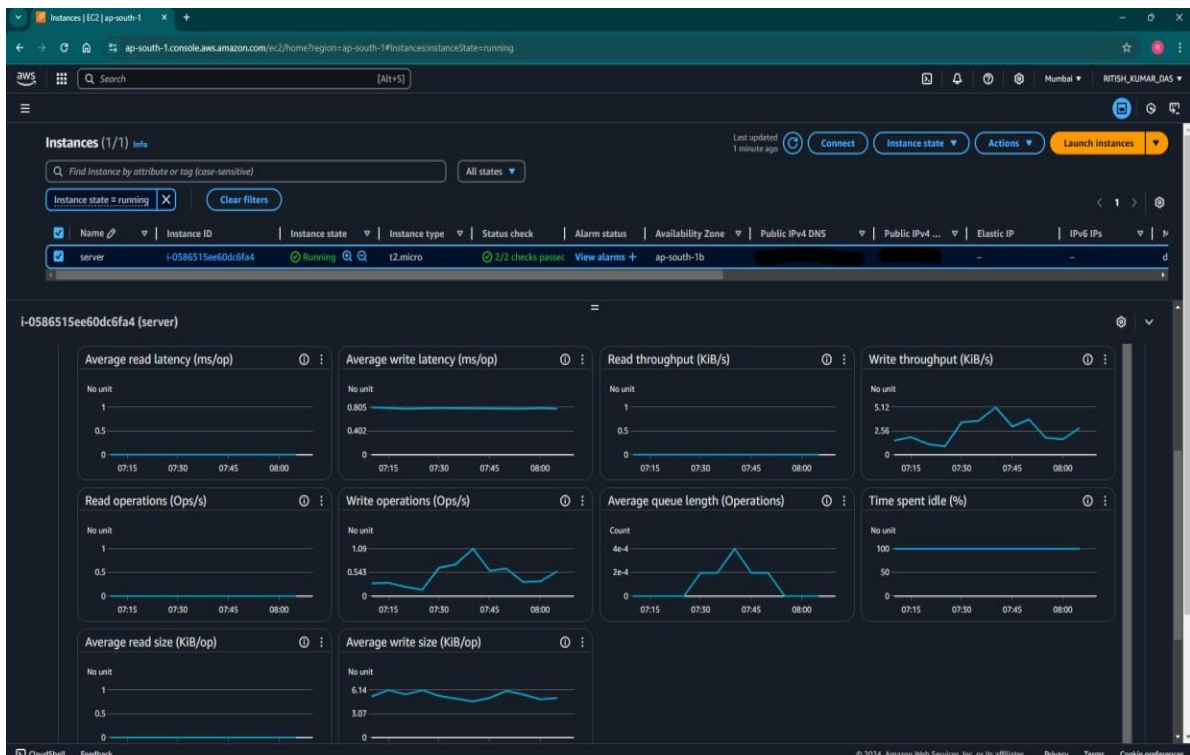


Figure 7.1: AWS EC2 Instance hosting the server

7.3.1.1. Functional Requirements

- The server has to process incoming POST requests containing user data and also manages a list of unique IDs.
- It adds the ID to an array *idArray*, if it is not already present.
- It also sets a timeout to remove the ID from *idArray* after 15 seconds.
- Upon successful processing, the server returns the current number of unique IDs in *idArray*.
- The server also adds a new item to a queue, with a maximum size of 5.
- The server also retrieves the latest item from the queue without removing it.

7.3.1.2. Non-functional Requirements

- Performance
 - The server handles concurrent requests, especially when adding and removing data from the queue or managing unique IDs.
 - Queue operations have been optimized by ensuring the queue size to 5.
- Scalability
 - The server can handle higher volume of requests.
- Reliability
 - Edge cases have been handled such as adding duplicate IDs, processing timeouts, and ensuring the queue never exceeds 5 times.
 - Timeout mechanism for IDs should reliably remove ID after the specified period.
- Data Integrity
 - The server ensures that the unique IDs are tracked correctly and removed after their timeout expires.
 - The queue should always return the latest item when queried and remove the oldest item when the size exceeds 5.

7.3.1.3. System Design

- Server Setup
 - Express.js has been used in the application, a lightweight framework for building web applications in Node.js
 - The server runs on port 8080 for now.

- Middleware
 - Body Parser: To parse the JSON request bodies, body-parser middleware has been used.
- Data Structures
 - Queue: used to store the data objects. The queue size is capped at 5 and the oldest item is removed when the queue exceeds this limit.
 - Unique ID Tracking: Array has been used to track the unique IDs. It ensures each ID in the array is unique and timeouts are used to remove IDs after 15 seconds.
- Timeout Management
 - Timeout (setTimeout): Each unique id has a timeout that removes the ID from the idArray after a 15-second delay.
 - Timer Storage (idTimers): This object stores the timers for each id. If an id is refreshed, its previous timeout is cleared using clearTimeout.

7.3.2. GCP APIs

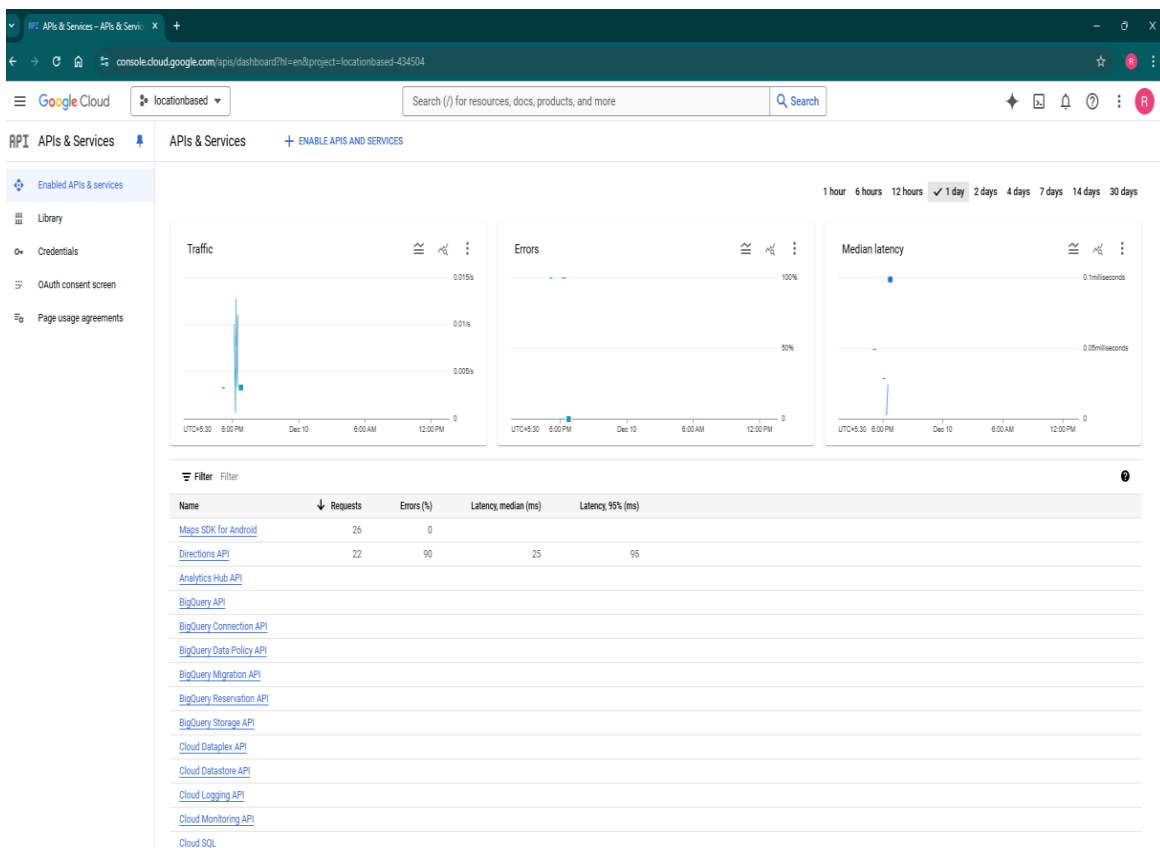


Figure 7.2: Google Cloud Console – showing API statistics for Map and Directions APIs

7.3.2.1. Functional Requirements

- The application integrates Maps and Directions API from Google Cloud Platform.
- The Maps API retrieves map-related data for a given address or location.
- The integration of these APIs helps in computation of travel time and distance between multiple origin-destination pairs.
- The Maps API converts a physical address into latitude and longitude coordinates using the GCP Maps API.
- The Directions API gives two locations (start and end points), calculates the best driving route between them using the GCP Directions API.

7.3.2.2. Non-functional Requirements

- The GCP APIs scale seamlessly to accommodate high traffic volume, when there is need to route and for distance matrix calculations.
- The integration is reliable, and backup mechanisms are in place in case of service disruptions.
- Latency is being minimized by the API whenever required.
- Implementation of caching mechanisms where feasible have been done to reduce repeated API calls for the same data.

8. Top-to-down Implementation Approach

8.1. Phase 1

- In phase 1, we tried static plotting of device location while it changes location throughout. This particular functionality is essential for the driver end of the application which continuously sends its location to the server. The plot in Figure 6 below, shows static plotting of a device as it changes location.

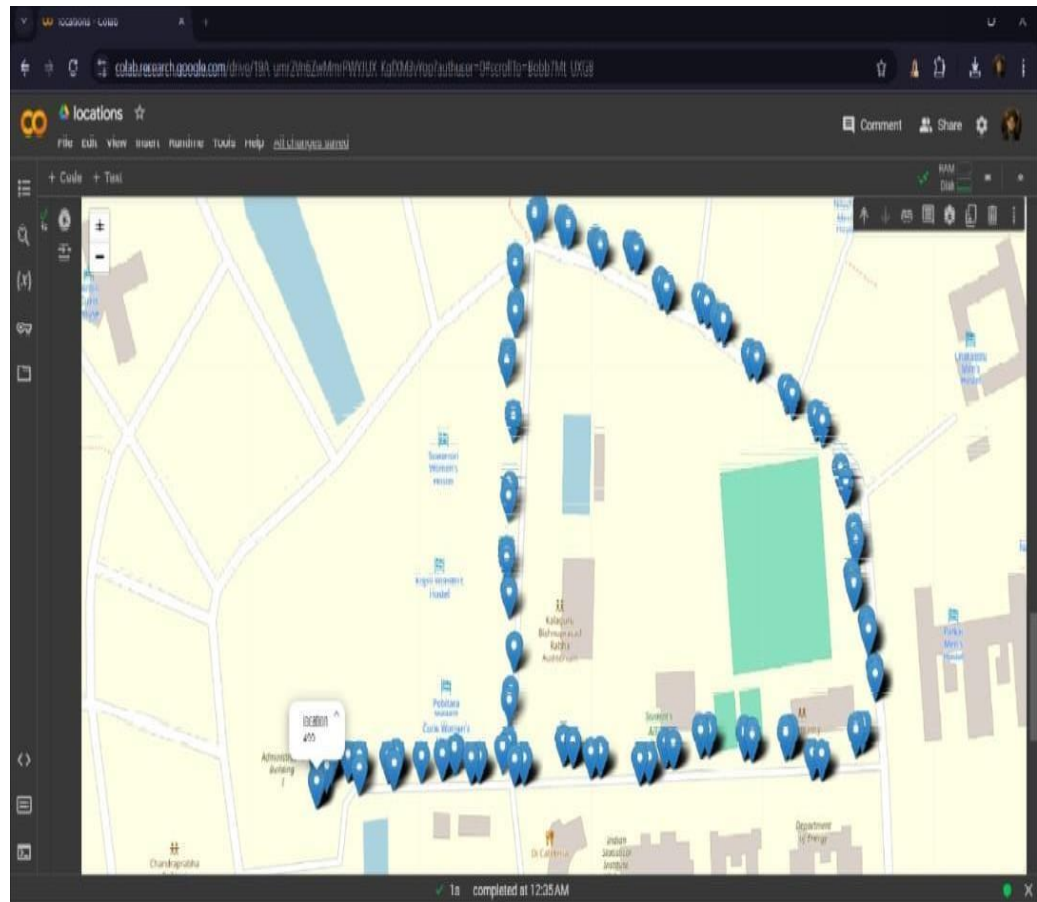


Figure 8.1: Initial Location Plotting using Google Colab and Python

- Location was received on the server side and converted to a .CSV file.
- The .CSV file was fed to the Python code using Geopandas plotting each coordinate on the static map.
- This step laid down the basic concept of tracking driver location through connection to server and its subsequent plotting helped us visualize how to implement this on a dynamic map using real-time updates.
- Next, to test live tracking, we fixed a point of reference.

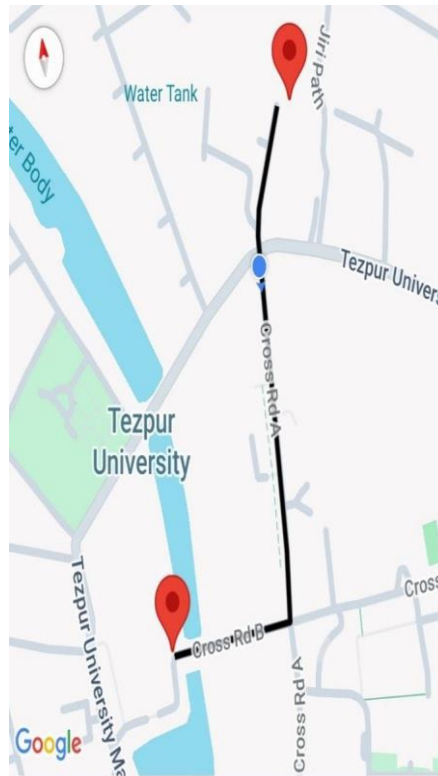


Figure 8.2: Live tracking w.r.t a reference

- A Directions API was called from the device's current location to the fixed location.
- After receiving the route polyline, it is plotted on the dynamic map.
- Live tracking on the route was done dynamically changing the route plot in accordance with changing remote device location.
- We needed this to establish dynamic changing of plotted route and tracking with changing location.
- This stage specifically uses current device location rather than fetching a remote device location from the server as this stage only encompasses manipulation of route plot on the map.
- Even though computed locally, this stage helped us get an idea of how to plot a route dynamically with changing location.

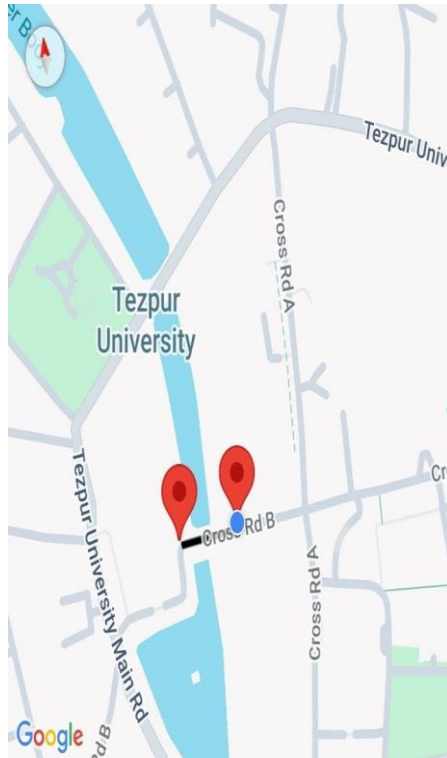


Figure 8.3: Live tracking w.r.t reference

- Next, the Request, Incentive and Reserve buttons were added to the UI and the Maps Fragment was integrated to the UI.

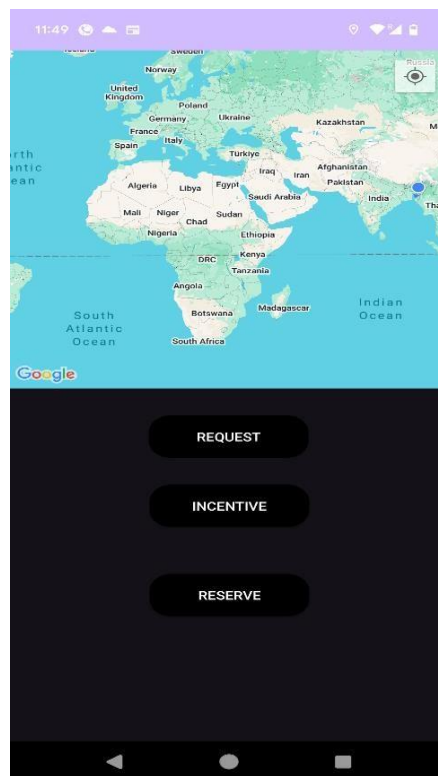


Figure 8.4: The Passenger App UI

8.2. Phase 2

- Fixed the route between user and driver entities on click of **Request** button.

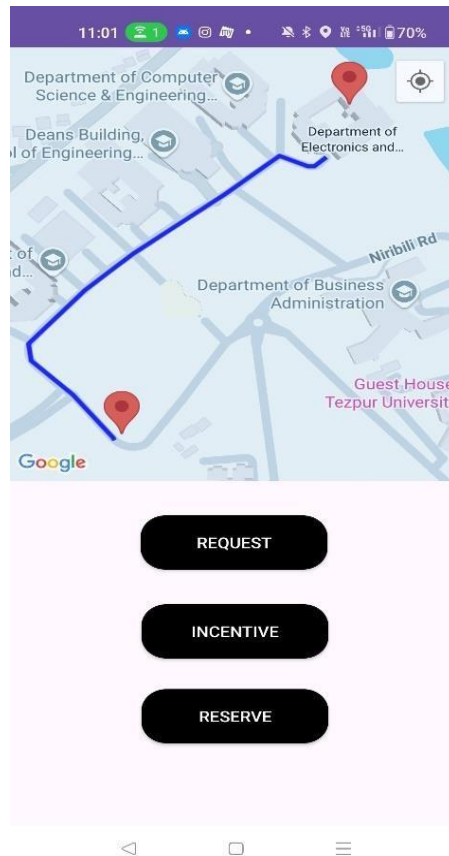


Figure 8.5: Updated Passenger App UI

- This stage establishes connection between remote device of driver and passenger.
- We fetch the current driver location from the server and get the current passenger location.
- A Directions API call is made between these two points plotting a route polyline between the two entities.
- Periodically, the updated driver location is fetched from the server by the passenger end of the application.
- Introducing dynamic routing between passenger and driver by re-plotting the route between the two. As the driver location changes, the route plotted changes.

- We can see the dynamic update of the route on the map shown in Figure 8.6 and Figure 8.7 below.

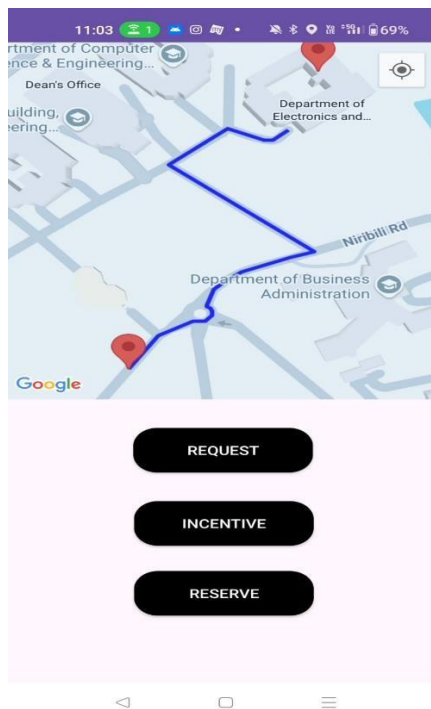


Figure 8.6: Initial Route on Request

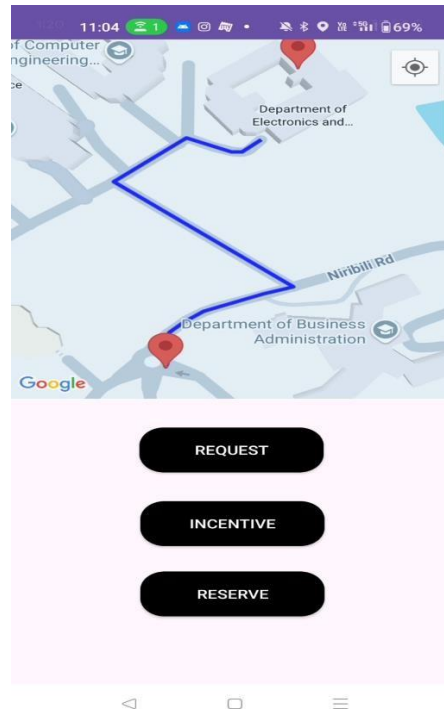


Figure 8.7: Updated route

- Next, we designed the Driver End of the application that shows the live count.



Figure 8.8: Driver End of the Application

9. Virtualization of Public Transport Hubs:

9.1. Concept:

- Every city has different chowks/hubs in between which public transport entities traverse. It is an interconnected mesh of routes that we aim to virtualize.
- In present practice, there is an unsaid queuing system as to First Come First Serve amongst the drivers. Whichever driver comes first, he gets added to the first in the queue, and he can load up passengers first.
- These autos move from one hub to another on a dedicated path to and forth. If we could give each hub a unique ID and simultaneous route connecting them with a unique ID as well, we could picture a virtualized public transportation where each driver is a part of a well-defined virtual system.
- Each driver could select/be a part of those routes and hubs which they would generally take to pick up customers.
- Establishing a unique ID for each driver, the queuing process starts when one driver comes in proximity to a unique Hub which they have already registered for.
- A passenger on a route could pick a start hub and end hub and a connection between the driver enrolled into the unique Hub and the passenger in the unique route will be established, notifying presence of a potential passenger in the route to the driver.

9.2. Implementation:

- Driver creates a route. In this process he enters the starting hub, ending hub and selects one route out of possible many routes between these two hubs.

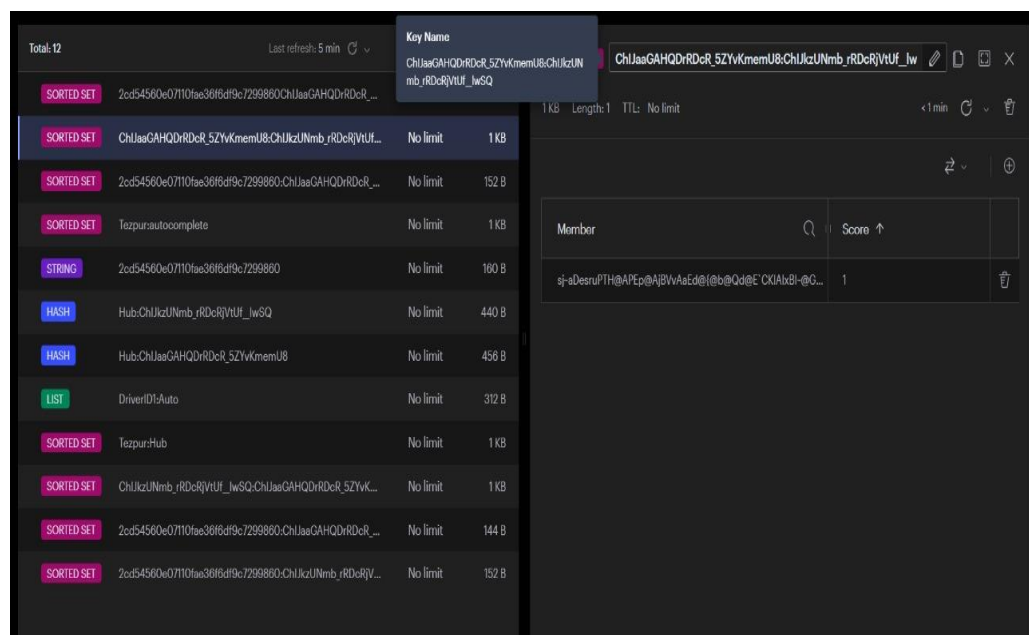


Figure 9.1: Redis showing the SHA256 encoded Polyline

- The starting hub and the ending hub are initialized with the value of the **Google placesId** which is unique for every place. The route ID is a SHA256 encode of route points (**Polyline**).
- Every route has two directional incoming traffic. From Point A to B and Point B to A. To create a unique ID, for queueing a combination of route ID : start hub ID is used for queueing of drivers.
- When a driver comes in proximity of 100 meters and clicks **Start Drive**, they get added to the queue. Also, its location is added to the in-memory (Redis) database for tracking by the passenger.
- When a passenger selects a start hub and an end hub, different registered routes is displayed on a dynamic map which they can select.
- After selecting a route from the passenger's side, we have start hub ID and end hub ID. The passenger gets added to a sorted set route ID : start hub ID : passenger.

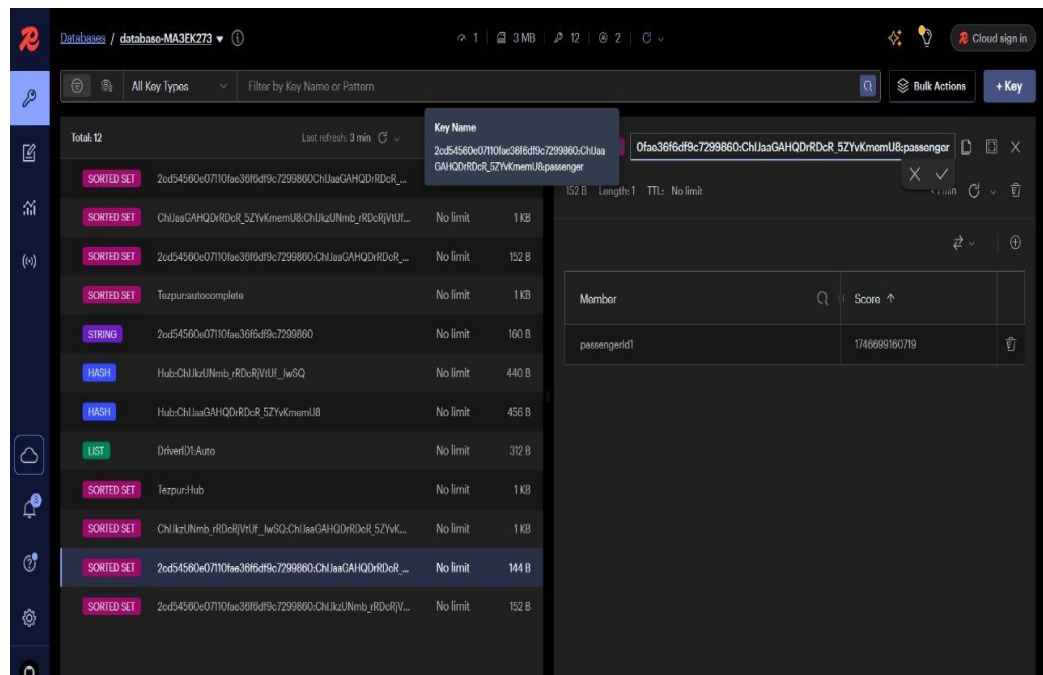


Figure 9.2: Redis Side showing the Sorted Set Route_ID : Start_Hub_ID : Passenger

- Both driver and passenger are bound to the route within 100 meters distance on each side of the path using **BoundsBuilder Containment**.
- The driver first in queue with selected route ID and start hub ID in which it is queued gets to see the number of customers ahead in the route.
- Once the driver moves 200 meters away from the hub and inside the bounds of the path, the driver is removed from the stationary queue which is Route ID : Start Hub ID and is moved to In Transit Queue, which is Route ID : Start Hub ID:inTransit.

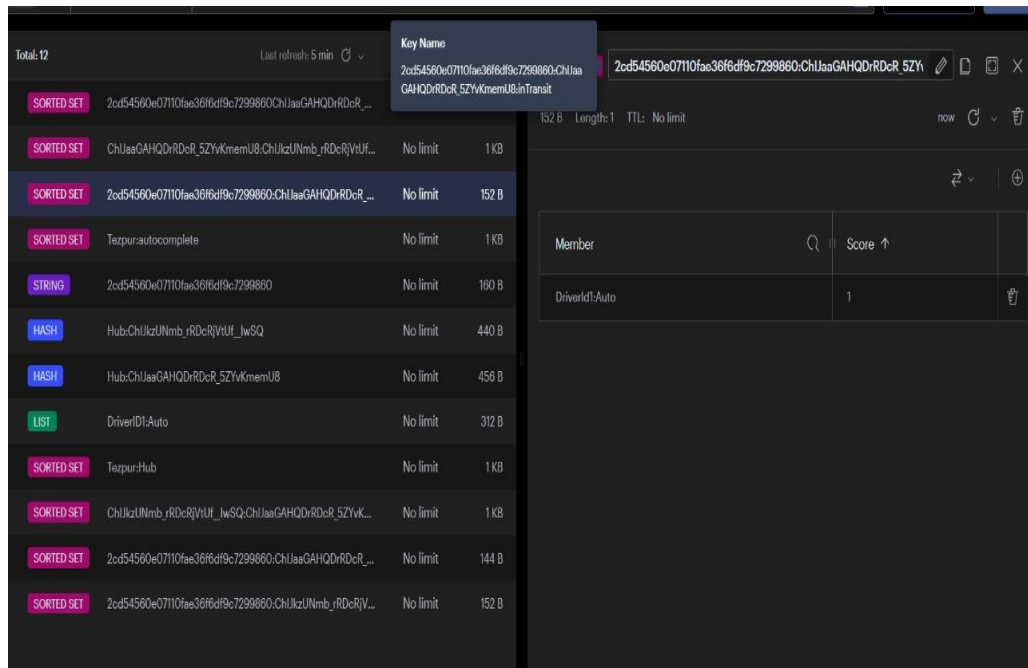


Figure 9.3: Redis Side showing the In Transit Queue for Vehicles in In Transit mode

- The passenger, once logged in and selects the route and start hub, first gets to see all the drivers in transit through live tracking on map. If no drivers are present in transit, they could request a driver from the hub/stationary queue and would be able to track them.

9.2.1. Route Registration Feature:

- Every driver gets to choose/create the route it generally would traverse on a daily basis.

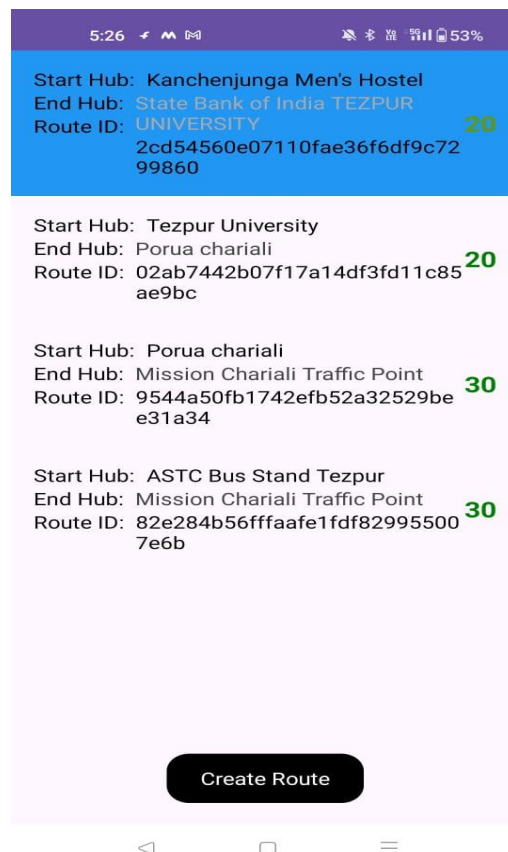


Figure 9.4: Routes Select Activity

- The Driver can choose any of the pre-registered routes or create a new one by clicking “**Create Route**”.

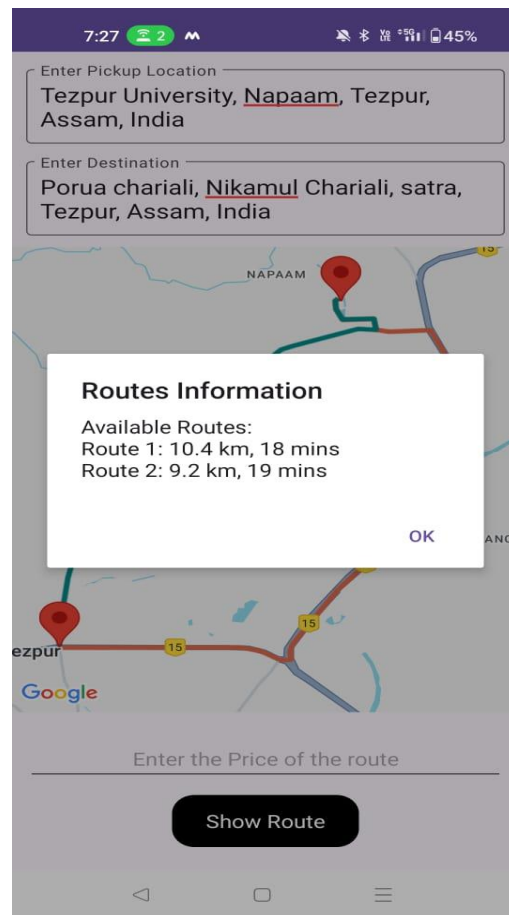


Figure 9.5: Route Registration with information

- The Driver can see the statistics of all the possible routes between the chosen start and end hubs.
- The Driver can then choose any of the routes and register themselves to the same.
- The Driver is supposed to choose one of the routes and enter the corresponding fare that it might take for the whole trip.
- This is essential to create a database of routes that is queried by the passengers to connect with the drivers as well as further on essential for the Intelligent Routing System that we will see up ahead.

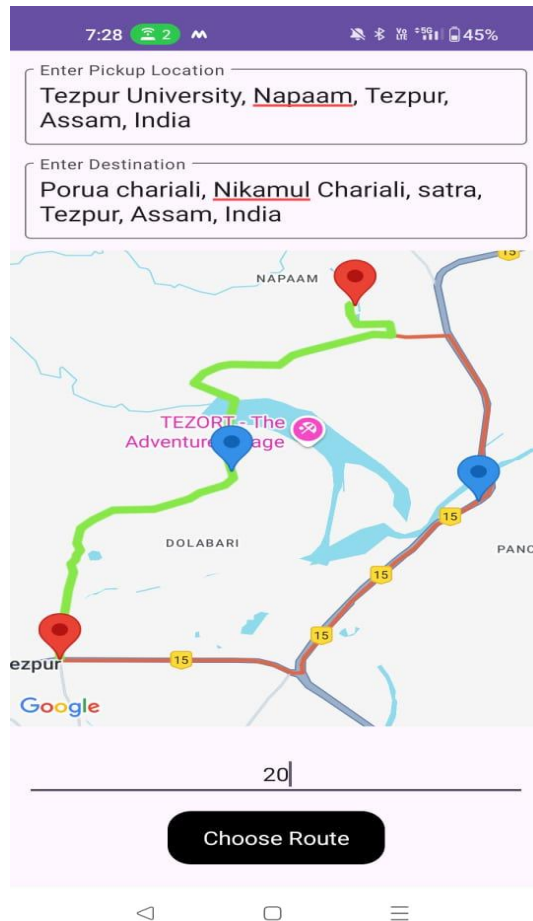


Figure 9.6: Route Registered by the Driver

- After the Driver chooses the route, it gets updated into the databases and also is stored locally to be selected from the **Route Select Activity** shown in Figure 16.
- This functionality will allow the governing authority to create a database as the usage of the application spreads out, the micro-information about each hub, route will become a part of the database as drivers across multiple cities keep on adding the routes to the database.
- This helps in discovering new possible routes, which further helps in intra-city traversal for passengers via the Intelligent Routing System.

9.2.2. Insertion of Driver and Passenger in Registered Route:

9.2.2.1. Driver Trip Lifecycle and Queue Management Workflow:

- After choosing a route from the **Select Route Activity**, the Driver can now opt for their day's trip by clicking on **Start Drive** button after coming in proximity of 100 meters from the Start Hub.
- Once the Driver is near the Start Hub, the Driver is added to the stationary queue of that Start Hub.

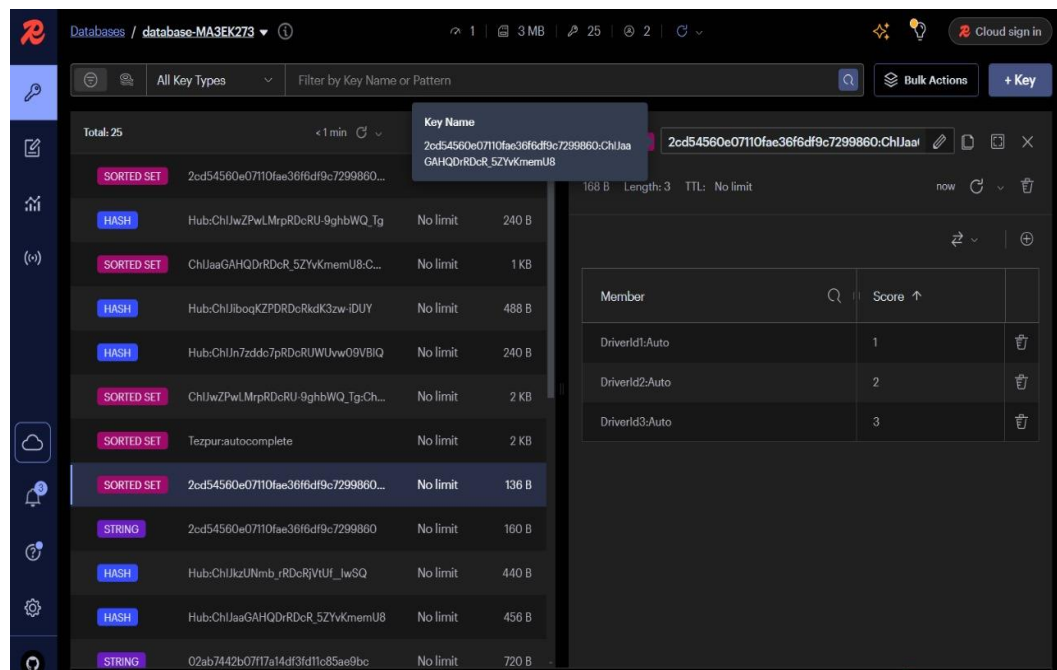


Figure 9.7: Redis view of Driver being added to the Stationary Queue

- The first driver in the queue gets to see all the potential passengers up ahead in the registered selected route who have requested a ride.
- Others in the queue are more susceptible to reservation of rides by a passenger or are made to wait until they move up the queue position.
- Once the Driver first leaves 200 meters and within the bounds of the route, he is moved from the stationary queue to In-Transit Queue.

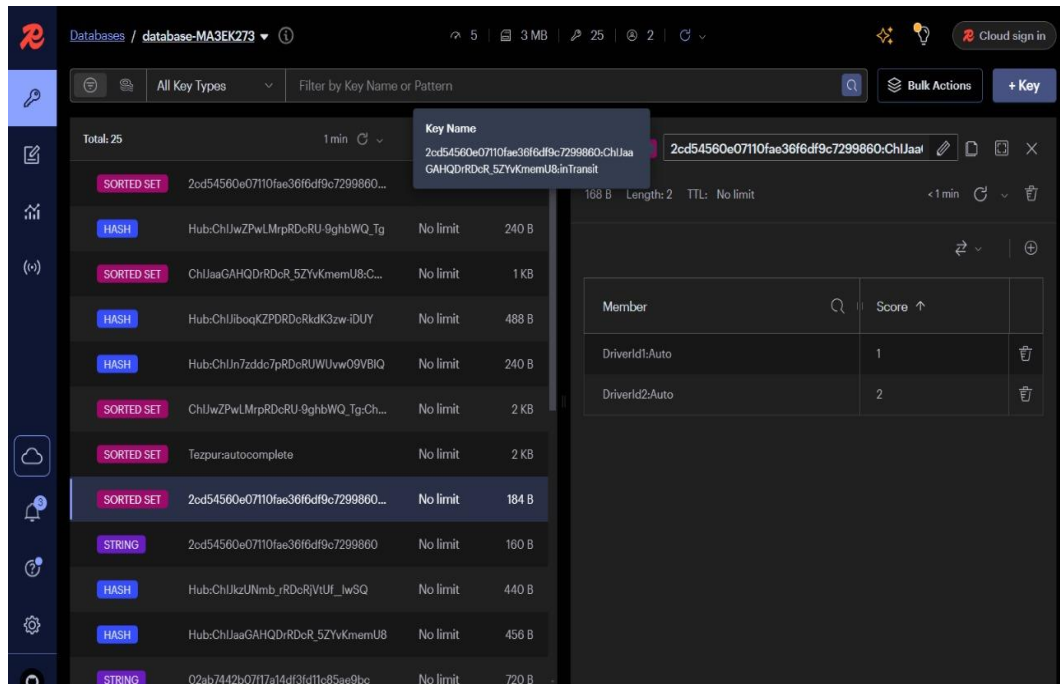


Figure 9.8: Redis view of Driver leaving Hub and being added to In-Transit Queue

- The Driver stays in transit until it reaches 200 meters of the End Hub after which the Start Hub and End Hub are switched so that the Driver can perform a return trip.

9.2.2.2. Passenger Lifecycle and Workflow:

- The Passenger first logs in to their app and selects the Start Hub and End Hub.
- The Passenger is shown all possible registered routes that they might want to take.
- After selecting the preferred route, if they are within the bounds of the route, they get a list of all drivers in transit which they can effectively track. If no driver is in transit, they are prompted to **Request** from the stationary queue of drivers at the Start Hub.
- After requesting, the first Driver ID in the queue is sent to the client-side passenger app which effectively tracks movement of the driver and notifies of existence of a passenger to the same.

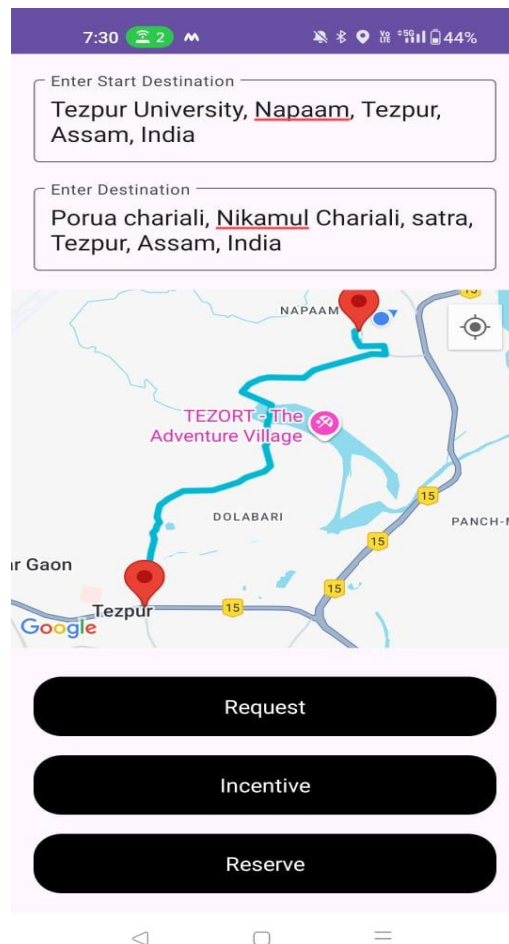


Figure 9.9: Passenger Selecting the preferred route

- If the Driver crosses the passenger, the passenger is again prompted to request the next Driver ID in the queue.

9.2.3. Autocomplete Feature:

- Every time a driver registers a route, a microservice checks if the unique hub and route is already present in the database (AWS DynamoDB) and in-memory (Redis) database or not. If not, then it adds the hubs and routes to the respective databases.
- For each city, the autocomplete feature stores names of hubs in a sorted set with Key as **City : Hub**.
- Redis provides LEXRANGE command which is used to retrieve a range of elements from a sorted set based on their lexicographical order.

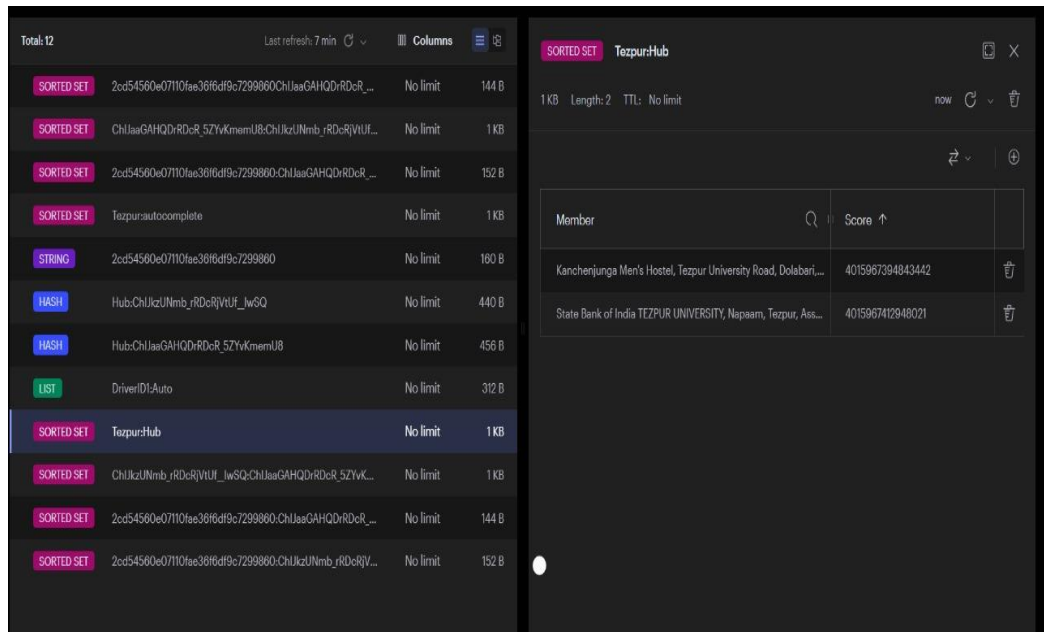


Figure 9.10: Redis Client Side showing the Sorted Set with Key as City : Hub

- On the client-side the passenger is prompted with autocomplete filling up the start hub and end hub using the **LEXRANGE** feature in the backend.

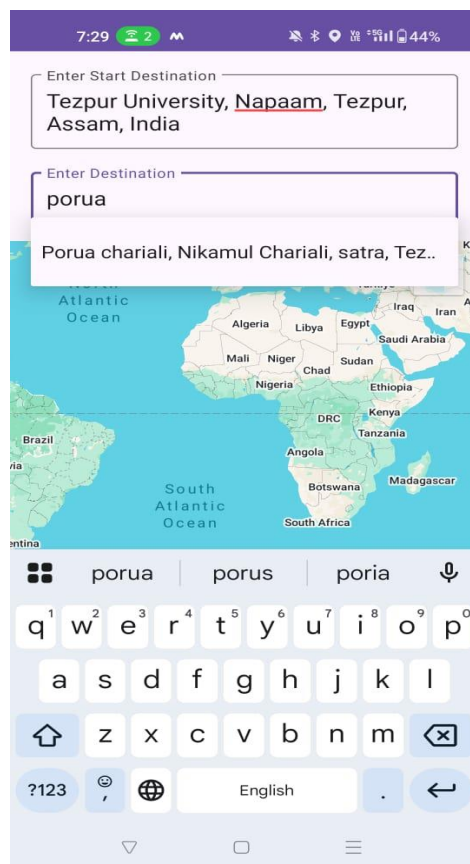


Figure 9.11: Passenger Client-Side showing Autocomplete in action

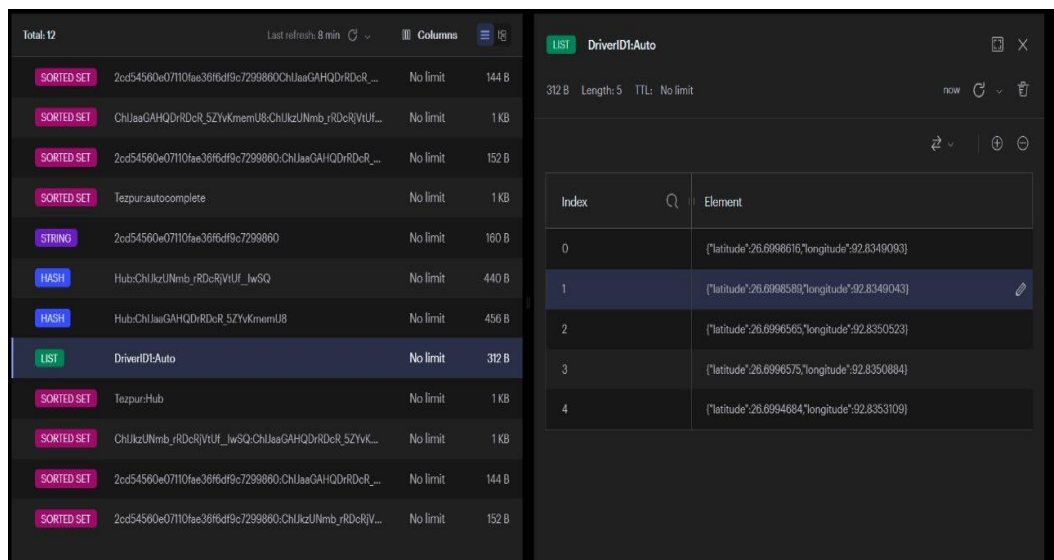
- A function of **GeoSearch** to recommend all hubs within a 5 kilometre radius is also provided at the beginning when no lexicons have been typed.
- The Value stored with respect to Key **City : Hub** is **Prefix : Hub Name : Hub**

ID. When autocomplete is enacted, and the passenger selects a hub name, Hub ID is also extracted from the Key.

- This feature has been implemented for usefulness in cases where passengers are unsure about the exact hub names. The auto filling feature helps them identify the desired location hassle-free and also select start hub and end hub IDs for plotting all possible routes to be selected by user.
- Selecting a route helps generate **Route ID : Hub ID** which is essential for communication between passenger and driver.

9.2.4. Live Tracking:

- The Live Tracking feature has been integrated in the application for the passengers to track all vehicles in route and their movements.
- Each Driver's ID acts as a key to store the latitude and longitude of the Driver.



The image shows a Redis CLI interface. On the left, a list of keys is displayed with their types and sizes. The 'DriverID:Auto' key is highlighted. On the right, a detailed view of the 'DriverID:Auto' key is shown, displaying a list of 5 elements, each containing a JSON array of latitude and longitude coordinates.

Index	Element
0	["latitude":26.6998616,"longitude":92.8349093]
1	["latitude":26.6998589,"longitude":92.8349043]
2	["latitude":26.6996565,"longitude":92.8350523]
3	["latitude":26.6996575,"longitude":92.8350884]
4	["latitude":26.6994884,"longitude":92.8353109]

Figure 9.12: Redis view of the Driver's Latitude and Longitude being sent for Live Tracking

- Once a passenger logs in and selects his/her route, the live locations of the in-transit vehicles on that corresponding route are shown on a dynamic map.
- In case there are no such in-transit vehicles, the passenger is prompted to click on the **Request** button, which then allows the passenger to live track the first vehicle in the Start Hub's Stationary Queue.
- The first step in Live Tracking is to plot the route polyline on the dynamic map. After which, Driver's location is fetched.
- After fetching Driver's location, the marker is moved to the current remote location positioned in the map connecting it to the closest route point.
- The tracking subsequently moves from one route point to another connected to remote location until it crosses the closest route point to the passenger and the distance between the passenger and the remote location also exceeds 30 meters.

- It effectively transitions between tracking in-transit vehicles and/or tracking of vehicle allocated to the passenger through the **Request** being first one in the stationary queue.

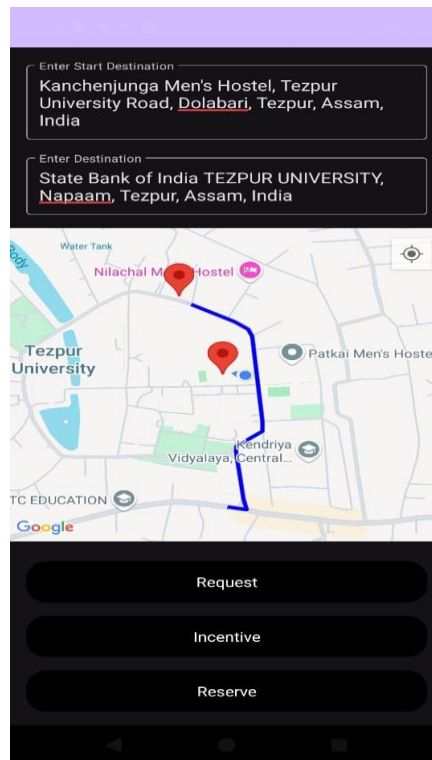


Figure 9.13: Live Tracking between the Driver and Passenger starts

- The feature uses the unique Driver ID of each driver to track its live location and continuously updates the dynamic map on the passenger side.
- If a driver surpasses the passenger, the application tries to find other vehicles which shall be a potential ride for the passenger.

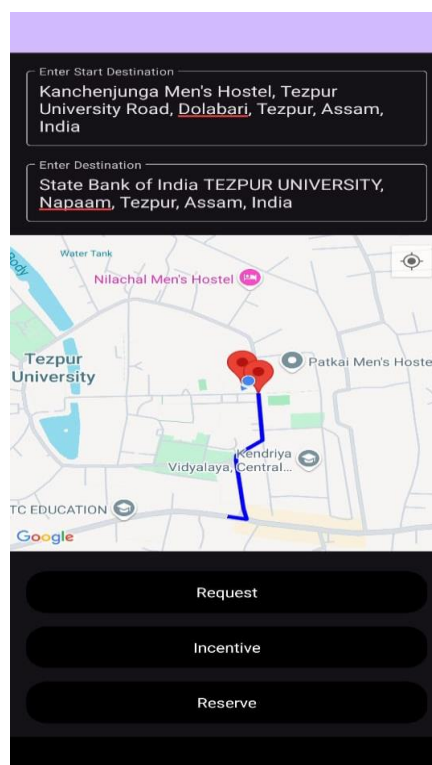


Figure 9.14: Live Tracking stops once the vehicle crosses the passenger

Figure 9.16: Database Schema for the application

9.3.3. Workflow:

- The Application has been developed by implementing various tools such as:
 - Google Cloud Platform (GCP)
 - Android Studio
 - Redis (Redis.io)
 - Neo4j (Graph Database)
 - Amazon Web Services (AWS)
 - EC2 Instance
 - Lambda
 - DynamoDB
 - API Gateway
- AWS Lambda has been used to implement the APIs as or when needed using the logic of 3 lambda functions defined in it. These are:
 - **Process RouteSelect:** This lambda function enables the driver to select an existing route from the database or create a new one. It handles multiple errors effectively and authorizes any new route update or creation successfully.
 - **Route InfoFetch:** This lambda function fetches all the information of the routes required for many microtasks done at the Redis level. Also, this lambda function, along with Redis, plays a major role in the Autocomplete feature and Hub information.
 - **Intelligent Routing:** The most important part of the software, the intelligent routing system's backend logic has been defined in this lambda function which enables a passenger to travel through unknown places effectively – both in time and money.

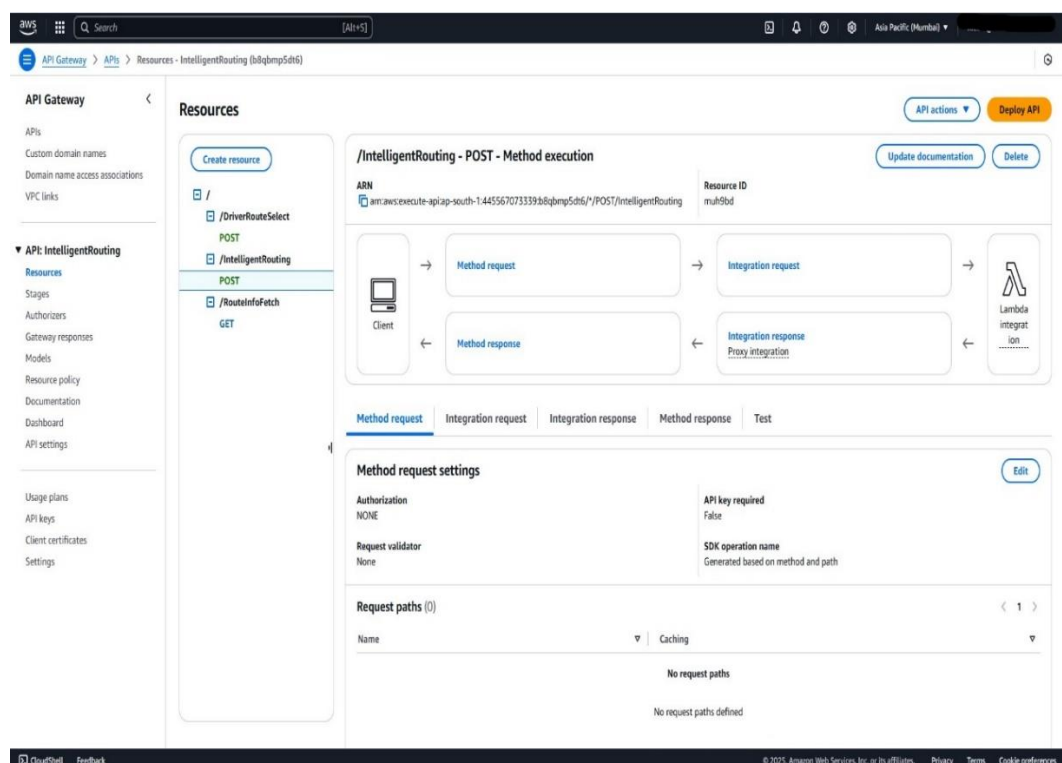


Figure 9.17: API Gateway integrated with AWS API Gateway

- The Route Registration begins by first making a Places API request to Google Cloud Platform, every time the driver changes the Start Hub or End Hub fulfilling a suggestion of predictions from which the driver can select from the provided dropdown list.
- After the Start Hub and End Hub is decided, information about both the hubs is fetched. For example: placesId, Latitude & Longitude of respective hubs.
- Then, on click of **Show Route** button, a Directions API call is made from the Start Hub to End Hub with all alternative routes being true.
- The received response from the Directions API is a list of polylines with distance and duration, each route is marked with incrementing value 1, 2, n for ease of distinguishing and selecting routes by the driver.
- Each polyline is decoded and plotted on the map with information associated with it as discussed before at the midpoint of each route.
- These routes are plotted with different colors. When the driver clicks on any one of the routes for selection, a unique color of light green is set to distinguish between selected and unselected routes.

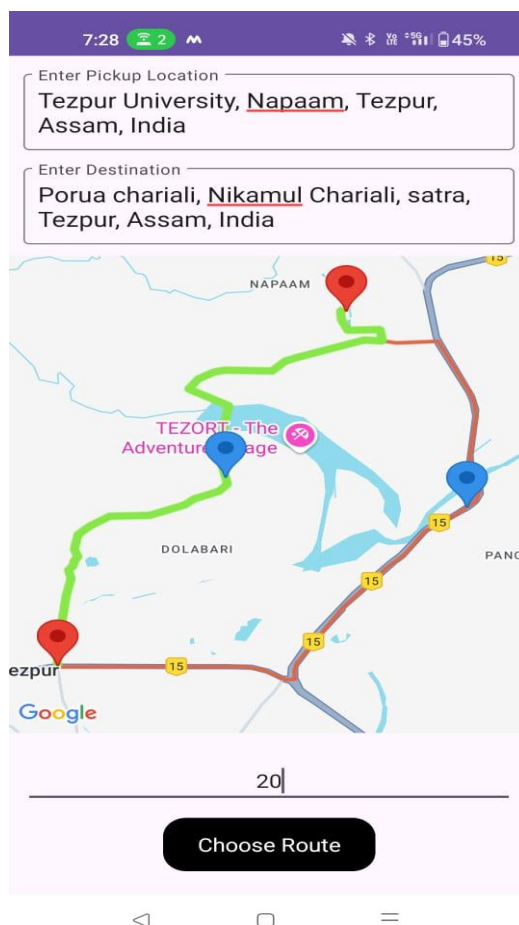


Figure 9.18: Process RouteSelect in action

- After selection, the driver is supposed to enter the fare for the whole ride and then click on **Choose Route** to register themselves for that particular route.
- Lambda function **Process RouteSelect** of API Gateway processes this information, checks that the driver has not registered for the route before, and if

the route exists in the database or not.

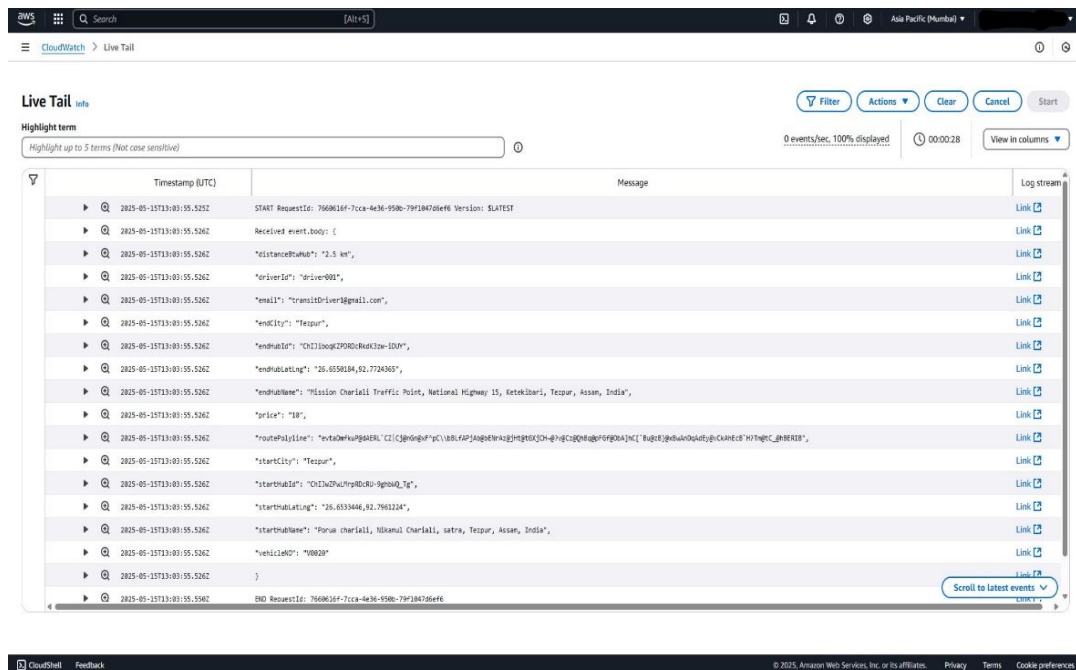


Figure 9.19: AWS Cloudwatch live tail showing Process RouteSelect payload

- If the route doesn't exist, the route is created and is added to DynamoDB, Redis and Graph database.
- The autocomplete using lexicographically sorted sets is stored in the key **City : Hub**.
- **Start Hub : End Hub** key is created/updated containing the polylines by the lambda function, **Process RouteSelect**.
- Also, if the hub does not exist the hash set of all the values corresponding to the hub being latitude, longitude, hubs connected and other useful information about the hub is and can be stored with the key **Hub : Hub ID**.

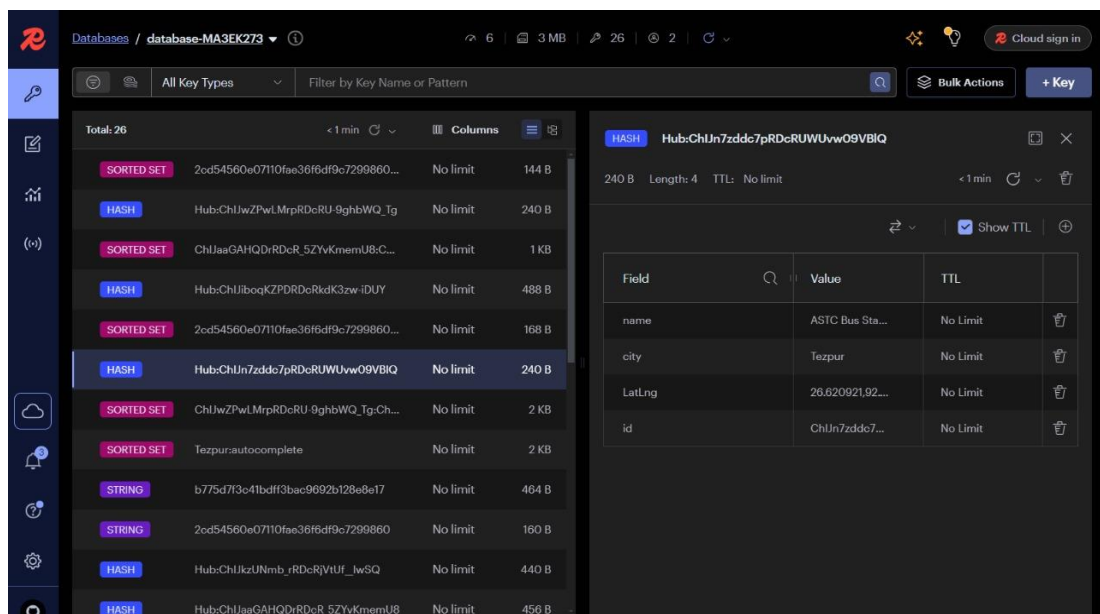


Figure 9.20: Hash set using key Hub : Hub ID

- The created route by the driver is stored locally with all the values in room db local storage.
- Whenever the driver selects the route to traverse and then clicks **Start Drive**, a web socket connection is established which recursively emits driver location to the **EC2 Instance** corresponding to the key Driver ID.
- At the passenger end, when the user starts typing the name of the starting destination or end destination, then the **Route InfoFetch** lambda is triggered which fetches values from Redis.
- First, it assists in autocomplete for the hub names. The response is **Hub Name : Hub ID**. The dropdown list is populated with hub names.
- After the user selects any of the provided hub name suggestions, Hub ID is extracted from the autocomplete payload.
- Now another request is made with the key **Start Hub ID : End Hub ID** which returns all the possible polylines between the two stations/hubs.
- The polylines are decoded and plotted on the map on the passenger end of the app.
- The passenger is supposed to select his/her desired route within whose bounds they lie as well.
- After selecting a route, the **SHA256 encoded string** of the polyline becomes the **Route ID**.
- A request is made to the EC2 Instance with the payload **Route ID** and **Start Hub ID**. The EC2 Instance fetches all Driver IDs with the key **Route ID : Start Hub ID : inTransit** and returns the list of Driver IDs to the passenger.
- If no driver is present in-transit, the passenger is prompted to click **Request**.
- On click of the **Request** button a similar payload is sent to the EC2 Instance asking for first driver in the stationary queue.
- The EC2 Instance returns the first value in the key **Route ID : Start Hub ID** from Redis. After getting the ID, the passenger recursively fetches the remote location of the driver by sending requests to the EC2 Instance with Driver ID as payload.
- The continuously changing remote location is plotted on the map with respect to the plotted polylines effectively tracking the driver on the route.
- If the driver crosses the passenger, the passenger is prompted to request again.
- These lambda functions effectively communicate with the database systems such as:
 - AWS DynamoDB: stores static values in table format. It is designed for high scalability.

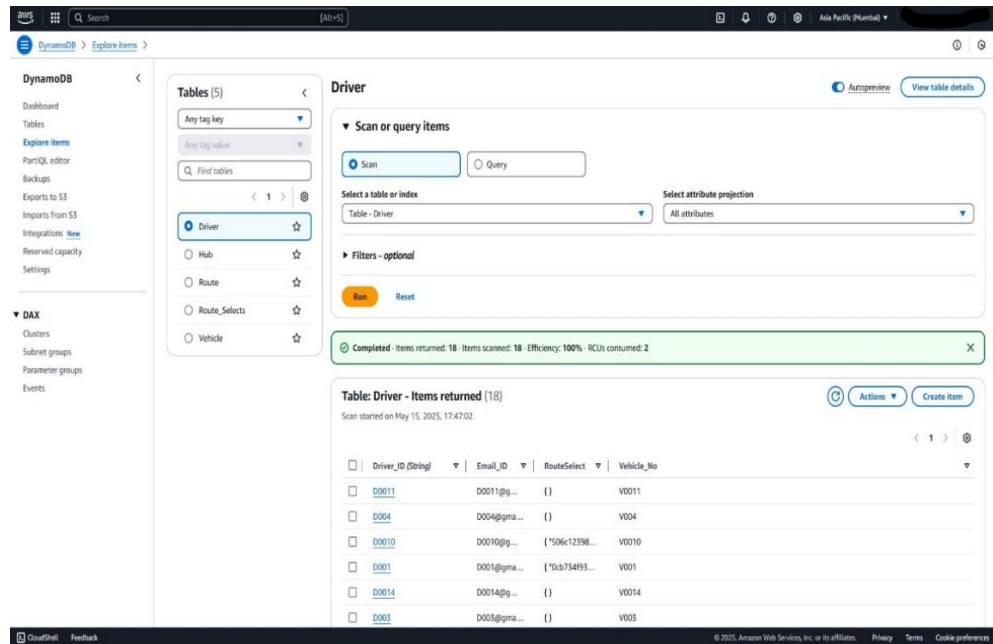


Figure 9.21: AWS DynamoDB

- Redis: This is the in-memory database. This has been implemented for faster access, scheduling tasks and storing session and dynamic variables.

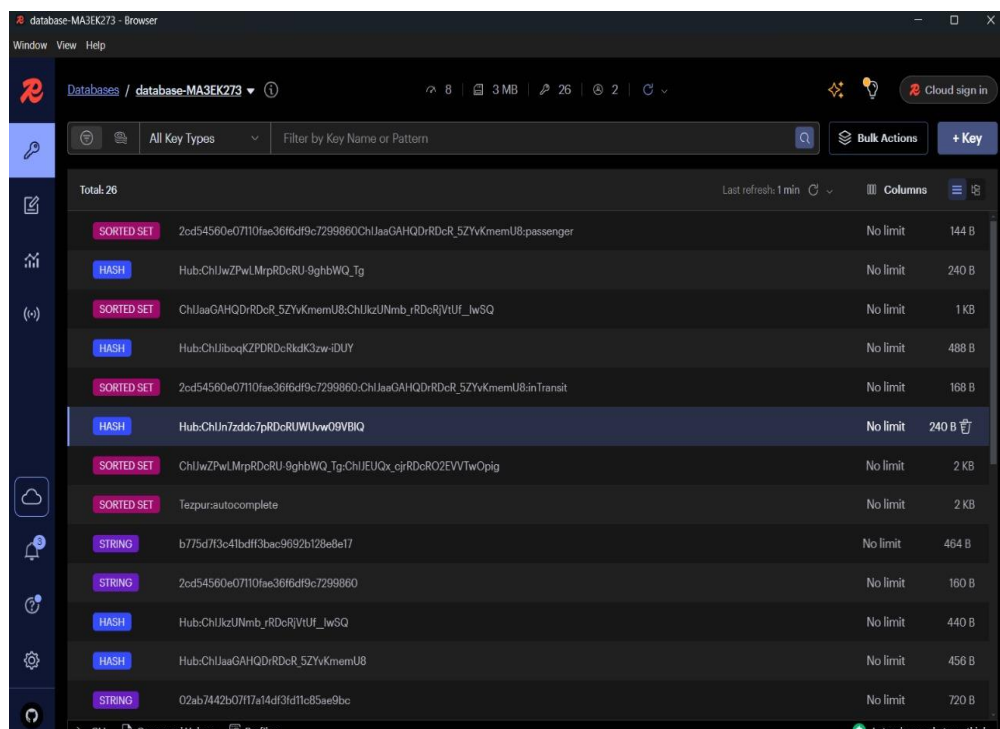


Figure 9.22: Redis in-memory database

- Graph Database: The graph database has been designed using Neo4j. This has been implemented to store relationships between hubs and routes and the interconnections within.

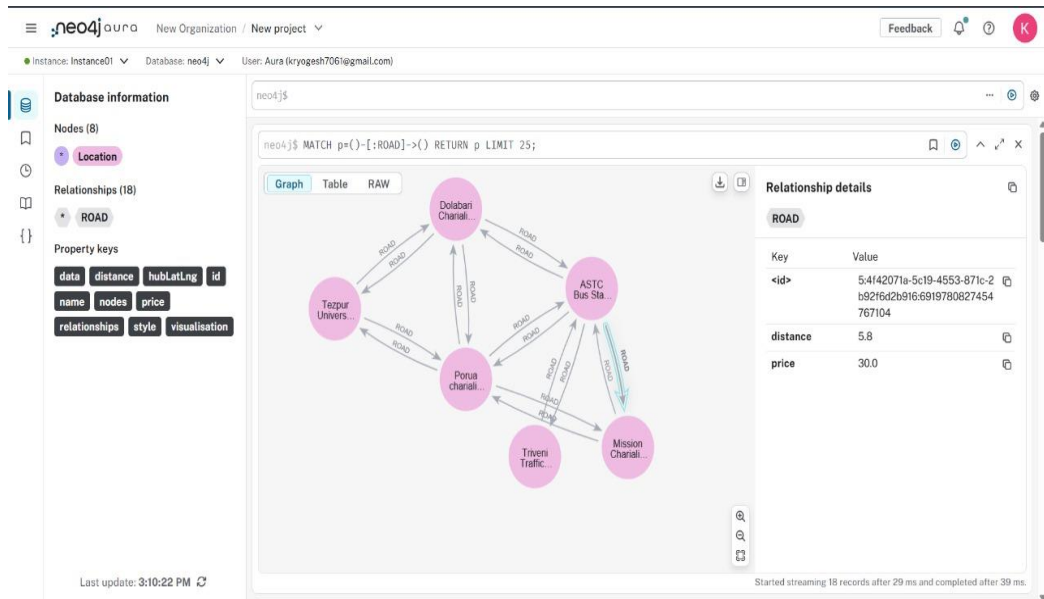


Figure 9.23: Graph Database using Neo4j

10. The Intelligent Routing System:

10.1. Concept:

- Every time a driver registers and the Graph database is updated, it adds to the mesh of networks between nodes which are hubs in this case. Looking carefully it is evident that travelling between these nodes require simple graph traversal algorithms to find shortest possible paths from one hub to another based on weights being distance, cost or time taken.
- It is also useful for finding interconnected routes or a possible connected route even though they might be far apart spaced by different nodes in between.
- From the already existing values of hubs if the passenger could choose any of the hubs as starting or ending locations, the routing system could query and apply algorithms on the already existing database to get desired output for the connected routes.
- The inspiration for the development of this unique Intelligent Routing System arises from the emphasis of uberization that has sprung up in many tier-2 and tier-3 cities in India. Using reserved cabs may not be an option for every individual. It is essential that they are given a hassle-free option of intra-city travelling.
- It will help a new user whichever city he/she might be unbeknownst to the way public transport system operates in the city. Unaware of the nitty-gritty of hub changes and traversal plans, they might need to take to reach from one location to another.
- Think of going to any city. Entering the starting point and ending point; getting back all possible hubs connected by routes with their location, total fare and total distance attached to the traversal.
- This effectively provides the cheaper public transport alternative to individuals against the reservation constraint system that has sprung up due to uberization.

10.2. Implementation:

- A new activity is designed on the passenger client side of the application called **IntelligentRouting**.

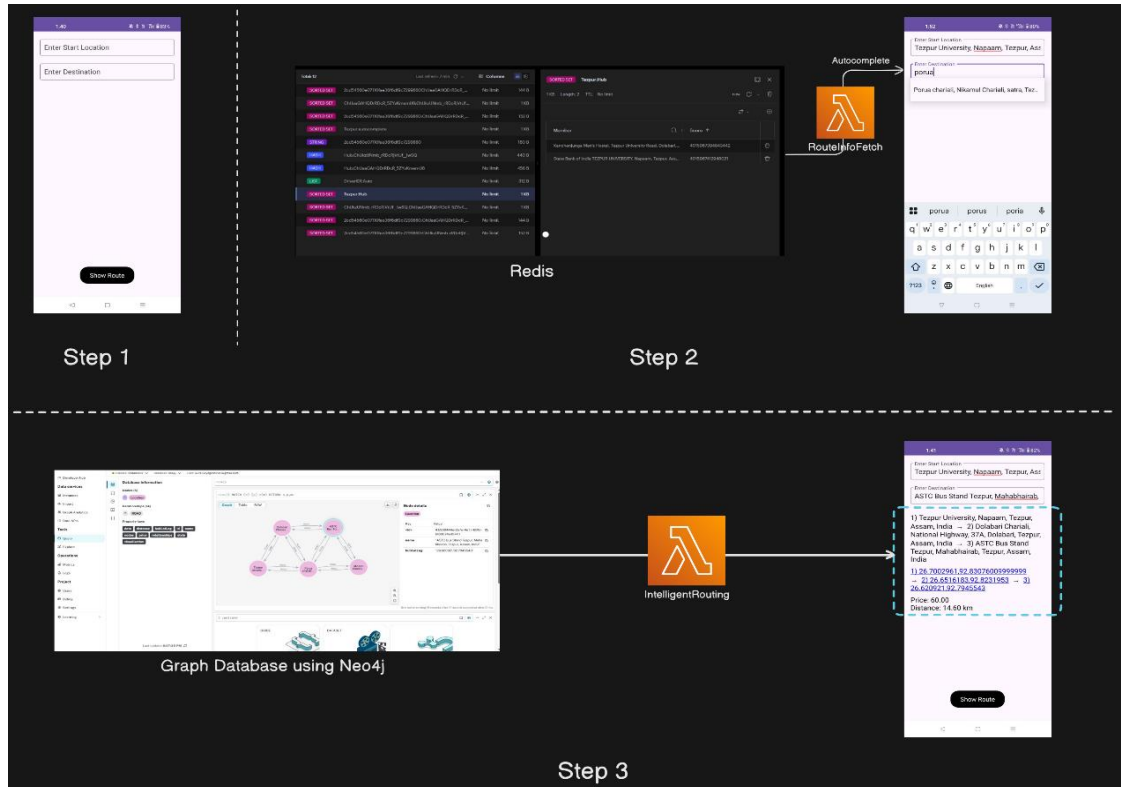


Figure 10.1: Intelligent Routing System

- IntelligentRouting** activity contains two autocomplete text view respectively for start and end destinations.
- Every time the autocomplete text view is triggered by the typing of user, it makes a request to **Route InfoFetch** to fulfill and provide suggestions for autocomplete from the already existing in-memory dataset from Redis.
- After the passenger selects from the drop-down list the start and end destinations, the user is supposed to click **Show Routes** button.
- On click of **Show Routes** button, a request is made to **IntelligentRouting** lambda function connected through the API Gateway.
- The lambda function receives the request payload of start hub and end hub and queries the graph database applying Dijkstra's algorithm to the dataset, finding the shortest interconnecting hub-routes with individual hub location path route-wise and Estimated Time of Travel.
- After receiving a valid value from the database, it sends in response the payload to the passenger end application.

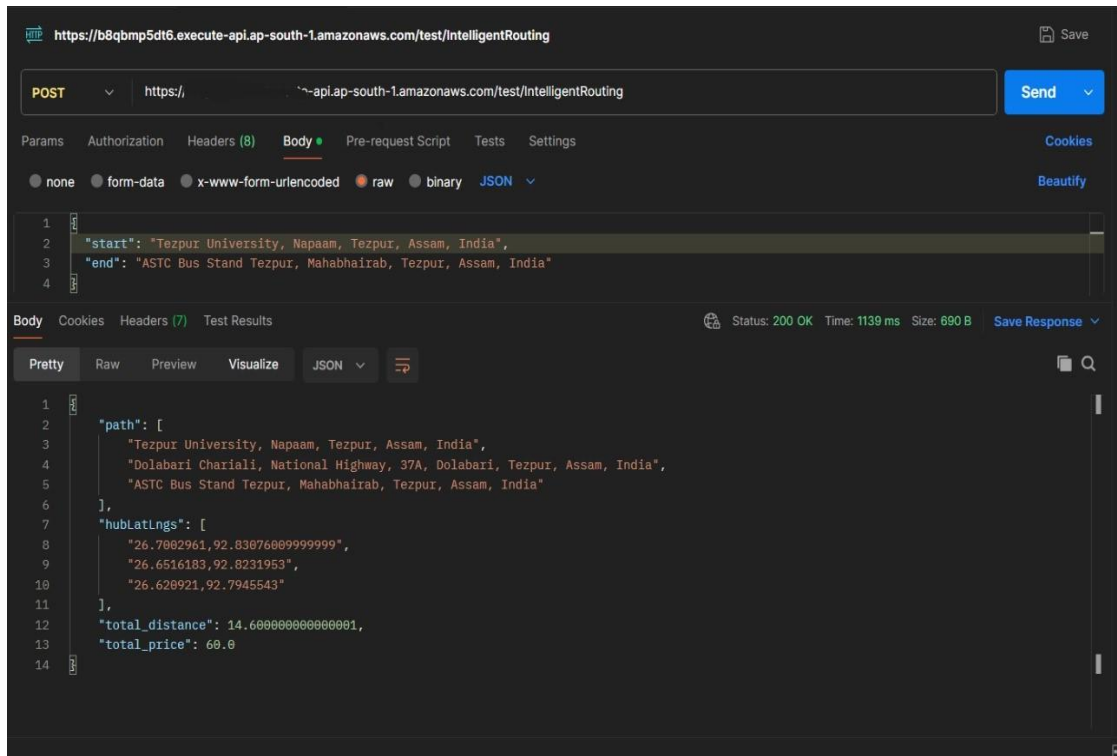


Figure 10.2: Intelligent Routing Response

- The passenger end extracts the information, namely the path, hub latitude, longitude, total fare and total distance.
- Sequential Hub transition is shown by the respective sequential values in the path field of the response received by numbering them 1 to n with arrows separating and notifying transition between hubs.
- Sequential hub latitude, longitude transition is also shown by respective sequential values in the **hubLatLng** field. Similarly, numbered according to their parent hub name in the path field.
- The latitude and longitude are clickable instances, that opens Google Maps pinpointing the corresponding coordinates.
- Total distance and total fare are also displayed with respect to their received fields of **total_distance** and **total_price**.

11. Future Scope and Implementation:

This public transport optimization project has been devised keeping in mind the real time problems associated with public transport for daily commuters in urban cities. We had identified some of the major problems we face as public transport users throughout this country and the inefficiencies presented in front of us.

There is no denying that there are still multiple problems that make public transportation inefficient on a grass-roots level. Identifying more such weaknesses and finding a single platform solution is definitely the most important scope of this project.

The project was proof of concept that our idea on virtualization and implementing the Intelligent Routing System is credible and implementable. But scaling it on a city or a nation level would require significant infrastructure backing. Scaling the system to such infrastructure would indeed be a formidable task.

There are many fields in which improvements can be made:

- **Additional public transport options, such as buses:**
 - In case of Buses, the vehicle is not related to the driver in a permanent sense. Hence the vehicle becomes the sole primary entity in the system.
 - The Driver End of the mobile application can be replaced by an **IoT device** in this case with an attached QR Code.
 - On initialization, this unique QR Code would be scanned by the operator, following the due process of registration of the vehicle and its associated route.
 - The IoT device will send location information to the server, and it will be duly processed by our existing architecture.
- **Intelligent Routing System:**
 - Implementation of a scoring system for the routes, so that segregation of active and inactive routes can be done. This will help with database maintenance and keeping the Graph database up to date.
 - To prevent unnecessary heavy load on the database, zoning of the database in accordance with respective levels would be a good starting point to optimize the querying and storage efficiency.

12. Conclusion:

The project successfully attempts to optimize the way public transportation services are provided to the common public. The virtualization of the *chowks/stands* is not just to help the passengers, but the drivers as well, establishing a real-time connection between them. The key takeaway points to be considered are as follows:

- **Easier access to generic public transport:** With the application, common public will be able to access public transport in a more convenient manner. With the reduction of waiting time, the transport experience will be smoother, efficient and more streamlined.
- **Optimized Handling of Public Transport:** With the drivers getting information about potential passengers up ahead, stagnation of transport systems will be reduced to a great extent, as the flow of traffic will always be at a constant pace.
- **Better income opportunities:** With passenger count always on the smartphone screen, drivers can have more opportunities of income and wouldn't have to go off-route in search of passengers for their daily income. The wastage of fuel, by going off-route in search of passengers would be avoided as well, saving fuel and money.
- **Focus on the common public:** With the virtualization of public transport, the public will no longer be limited to reserving cabs every time they commute. The uncertainty of getting public transport will be reduced to a great extent, thus, providing the passenger enough information to choose whether to travel in a bus/auto or book a cab.

The Intelligent Routing System is a novel idea that will map the whole public transportation system of a nation which can be made accessible to general public. The data extracted will act as a foundation for further enhancements possible/implementable by the governing authorities in the public transportation sector. The map created by the data extracted will surface out the inefficiencies and the corners of improvement on which authorities can work on and optimize it further on ground in the interest of public well-being.

Overall, the project paves the way for better passenger experience when it comes to the uncertain public transport. Whether it is big cities or small towns, the right to information for a passenger is what the application strives to provide. With virtualization of hubs, guaranteeing the systematic queuing at the stands, the time spent in unnecessary hassle of collecting passengers is being reduced, thus saving valuable time and money for both the passengers and drivers.

13. Appendix:

Technical Terminologies

1. Google Cloud Platform (GCP):

Google Cloud Platform (GCP) provides a robust and scalable infrastructure for leveraging Google Maps APIs. Through GCP, developers can seamlessly integrate the power of Google Maps into their applications and services. GCP's reliable network and global infrastructure ensure high availability and performance for applications utilizing these location-based services.

- **Directions API:** This API is a service that provides directions for various modes of transportation—such as driving, walking, bicycling, and transit—between multiple locations. It can calculate optimal routes, taking into account real-time traffic, road closures, and other conditions. The API supports features like waypoints, alternative routes, and toll avoidance, making it useful for logistics, delivery apps, or navigation services. Polygons are key outputs of this API, used to visualize the calculated routes on a map.
- **Places API:** This API provides access to a vast database of information about places worldwide. Users can search for places based on keywords, proximity or type, and retrieve detailed information such as names, addresses, phone numbers, user ratings, reviews and photos. The Place Details functionality within this API allows the user to get even more specific information about a particular place using its unique ID.
- **Maps API:** GCP's Dynamic Maps API is a service that allows developers to embed interactive, customizable maps into web or mobile applications. These maps support user interactions like zooming, panning, and clicking, and can display real-time data layers such as traffic or transit routes. Unlike static maps, dynamic maps are rendered on the client side and update in real time based on user input. The API also supports adding markers, shapes, and custom overlays, making it ideal for applications requiring live map interaction, such as ride-sharing, location tracking, or asset management platforms.

2. Polygons:

In the context of mapping and spatial data, a polygon is a sequence of connected line segments represented by a series of coordinate pairs (latitude and longitude). Polygons are commonly used to visually depict routes, paths, boundaries, and other linear features on a map. Their encoded forms are often used to efficiently store and transmit geographical line data.

3. Amazon Web Services (AWS):

Amazon Web Services (AWS) is a leading cloud platform providing a vast array of on-demand computing services over the internet. It offers a flexible and scalable infrastructure, allowing users to access computing power, storage, databases, and various other IT resources without the need for physical hardware. This pay-as-you-go model enables businesses and individuals to build and run applications, from simple websites to complex

enterprise systems, with high reliability, security, and cost-efficiency.

- **AWS Elastic Compute Cloud (EC2) Instance:** Virtual server in Amazon's cloud. Provides scalable computing capacity, allowing users to rent virtual machines with varying operating systems and configurations to run their applications. EC2 instances offer flexibility in terms of processing power, memory, storage, and networking, making them a fundamental building block for deploying and managing applications and services on the AWS infrastructure.
- **AWS DynamoDB:** Fully managed NoSQL key-value and document database that delivers single-digit millisecond performance at any scale. It's a serverless database, meaning AWS handles the underlying infrastructure. DynamoDB is designed for high availability and scalability, making it suitable for applications with large amounts of data and high traffic requirements. It uses a flexible schema, allowing for the storage of diverse data structures.
- **AWS Lambda:** Serverless compute service that lets you run code without provisioning or managing servers. You only pay for the compute time you consume. Lambda functions are event-driven, meaning they execute in response to specific triggers, such as changes to data in an S3 bucket or a request from API Gateway. This serverless approach simplifies application development and deployment, allowing developers to focus on writing code.
- **AWS API Gateway:** Fully managed service that makes it easy for developers to create, publish, maintain, monitor, and secure APIs at any scale. It acts as a "front door" for applications to access data, business logic, or functionality from backend services, such as AWS Lambda functions or EC2 instances. API Gateway handles tasks like request routing, authentication, authorization, rate limiting, and API versioning.

4. Redis:

Redis is an in-memory data structure store, used as a distributed, in-memory key-value database, cache and message broker. It supports various data structures such as strings, hashes, lists, sets, sorted sets.

- **Sorted Sets (LEXRANGE):** Data structures which each member has a score, allowing elements to be ordered. LEXRANGE is a specific command that enables querying elements in a sorted set based on lexicographical (alphabetical) range, useful for tasks like autocomplete or indexing strings.
- **Geosearch:** Powerful feature in Redis that allows the user to store and query geospatial data (latitude and longitude coordinates) efficiently, enabling proximity-based searches.

5. SHA256 Encoding:

SHA256 (Secure Hash Algorithm 256-bit) is a cryptographic hash function that produces a fixed-size 256-bit (32-byte) hash value. This encoding process is one-way, meaning it's computationally infeasible to reverse the process and obtain the original input from the hash. SHA256 is widely used for data integrity verification, password storage (by hashing the password instead of storing it in plain text), and in blockchain technologies.

6. BoundsBuilder:

Utility class or object used to programmatically define a rectangular geographical area (viewport or bounding box) that encompasses a set of geographical coordinates. By adding latitude and longitude points to a BoundsBuilder, it automatically calculates the minimum and maximum latitude and longitude values required to contain all the specified points.

7. Neo4j (Graph Database):

Graph database management system. Unlike relational databases that store data in tables, Neo4j stores data in a network of nodes (entities) and relationships (connections between entities). This graph-based structure makes it highly efficient for querying and analyzing highly connected data, such as social networks, navigation systems, and knowledge graphs. Relationships in Neo4j are first-class citizens, meaning they have properties and direction, making it easy to traverse and understand the connections between data points.

