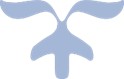
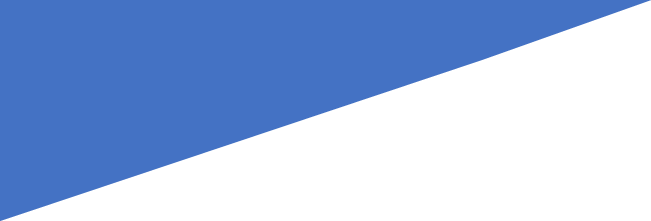


Advanced Databases

DB QUANTUM: Food Delivery Services

Guided by: Prof. Frank Hefter





**TEAM MEMBERS**

Vindhya Hegde

Sushant Huilgol

Kamal Sardiwal

Kalpesh Patil

Ambuj Solanki

09 May 2024

srh Hochschule hEIDELBERG

[1. Introduction 3](#_Toc886750426)

[2. Organization 3](#_Toc119034826)

[2.1. Roles & Responsibilities, use case handled 4](#_Toc240732741)

[2.2. Meetings 5](#_Toc884316860)

[2.3. Tools Used 8](#_Toc1148858874)

[3. User Story 8](#_Toc855672691)

[4. Details 9](#_Toc1842775753)

[4.1. Delivery Tracking and delivery (Sushant) 10](#_Toc1780766265)

[4.1.1. User Story 10](#_Toc753188022)

[4.1.2. Use Case Diagram 10](#_Toc1531574560)

[4.1.3. Actors 10](#_Toc1738807707)

[4.1.4. Descriptions of task with all interactions between actor and database 11](#_Toc1156691968)

[4.1.5. Data Flow 11](#_Toc1589197949)

[4.1.6. Databases 12](#_Toc1371847655)

[4.1.6.1. Databases Used 12](#_Toc413272022)

[4.1.6.2. Expressions 13](#_Toc581220953)

[4.2. Route Optimization for the rider (Vindhya) 13](#_Toc595174033)

[4.2.1. User Story 14](#_Toc2065432339)

[4.2.2. Use Case Diagram 14](#_Toc14585303)

[4.2.3. Actors 14](#_Toc1578934788)

[4.2.4. Descriptions of task with all interactions between actor and database 14](#_Toc1952886133)

[4.2.5. Data Flow 15](#_Toc348624475)

[4.2.6. Databases 15](#_Toc208563433)

[4.2.6.1 Databases Used 15](#_Toc1580365323)

[4.2.6.2 Expressions 16](#_Toc861173379)

[4.3. Enabling Fast Deliveries (Sushant & Vindhya) 16](#_Toc597043178)

[4.3.1. User Story 16](#_Toc310240835)

[4.3.2. Use Case Diagram 16](#_Toc1256673054)

[4.3.3. Actors 17](#_Toc547052089)

[4.3.4. Descriptions of task with all interactions between actor and database 17](#_Toc1719545336)

[4.3.5. Data Flow 18](#_Toc15177368)

[4.3.6. Databases 18](#_Toc1064720679)

[4.3.6.1. Databases Used 19](#_Toc1262341080)

[4.3.6.2. Expressions 19](#_Toc1797754086)

[4.4. Dynamically allocating the mode of transport based on various factors (Kalpesh Patil) 20](#_Toc90052128)

[4.4.1. User Story 20](#_Toc1377810475)

[4.4.2. UML: Use Case 20](#_Toc1117850165)

[4.4.3. Actors 20](#_Toc977666376)

[4.4.4. Description of tasks with all interactions between actor and database. 21](#_Toc896343384)

[4.4.5. Data Flow 21](#_Toc2002248345)

[4.4.6. Databases 22](#_Toc420006226)

[4.4.6.1 Databases Used 22](#_Toc912078246)

[4.4.6.2 Expressions used for this Use Case 22](#_Toc215460692)

[4.5. Boosting Restaurant Exposure: Leveraging Influencer Reviews (Kamal Kumar Sardiwal) 23](#_Toc775921306)

[4.5.1. User Story 23](#_Toc709580970)

[4.5.2. UML: Use Case diagram 24](#_Toc1223432184)

[4.5.3. Actors 24](#_Toc654423545)

[4.5.4. Description of tasks with all interactions between actor and database. 24](#_Toc1008259887)

[4.5.5. Data Flow 26](#_Toc1046309690)

[4.5.6. Databases 26](#_Toc455555055)

[4.5.7. Database Used 27](#_Toc238919407)

[4.5.6.2 Expressions used for this Use Case 27](#_Toc1729497790)

[4.6. Dynamic Pricing ( Ambuj Solanki ) 28](#_Toc1438328632)

[4.6.1. User Story 28](#_Toc831431274)

[4.6.2. UML Diagram 28](#_Toc1406247648)

[4.6.3. Actors 29](#_Toc199340720)

[4.6.4. Description of tasks with all interactions between actor and database. 29](#_Toc1614590395)

[4.6.5. Data Flow 30](#_Toc849310177)

[4.6.6. Databases 31](#_Toc90398094)

[4.6.6.1. Database used 31](#_Toc1621951581)

[5. Database 32](#_Toc1577998965)

[5.1. Overall structure 33](#_Toc1698357795)

[5.2. Data Model Overview of all Database 34](#_Toc1359360333)

[5.2.1. Mongo DB 35](#_Toc24752550)

[5.2.2. Neo4j 35](#_Toc1020939051)

[5.2.3. Redis 36](#_Toc2054459716)

[5.3. Used Expressions 37](#_Toc459138001)

[5.3.1. MongoDB 37](#_Toc1937850636)

[5.3.2. Neo4j 39](#_Toc2024846243)

[6. Application 40](#_Toc1108120610)

[6.1. Language Used & Why 41](#_Toc763224929)

[6.2. GitHub Path 41](#_Toc772475540)

[6.3. Methods/Functions 41](#_Toc2021627743)

[6.3.1. Calculate Shortest Path, Mode of Transport, Pricing 41](#_Toc198650180)

[7. API 42](#_Toc1188110327)

[7.1. Foreign API Description 43](#_Toc2045388687)

[7.1.1. Example Data 43](#_Toc307001559)

[8. Evaluation 44](#_Toc236979463)

[9. References 45](#_Toc1641755562)

[9.1. Books & Journals 46](#_Toc936861379)

[9.2. Webpages 46](#_Toc844256200)

# Introduction

Welcome to our exciting project! We're diving into the world of food delivery, aiming to create a service that's efficient and user-friendly. Our focus is on leveraging the strengths of different databases like MongoDB, Neo4J, and Redis to make it happen.

Imagine three main characters in our story: the Customer, the Restaurant, and the rider.

First up, the Customer. Picture someone craving their favorite dish but not wanting to leave the comfort of their home. With our service, they can hop online, choose a restaurant they love, and place an order, all while kicking back on their couch. Plus, they can keep tabs on their order every step of the way, from the moment it's placed to when it's in the hands of the rider.

Now, let's talk about the Restaurant. They're the wizards behind the delicious meals we crave. Through our platform, they can connect with hungry customers and showcase their menu in all its mouthwatering glory.

Last but not least, we have the rider. Picture them as the heroes zipping through the city streets, delivering happiness one meal at a time. With our system, they'll be equipped with the best routes, thanks to clever algorithms from Neo4j. These algorithms find the shortest and quickest paths, ensuring that your order arrives in record time.

But wait, there's more! We're not just stopping at speedy deliveries. We're also tapping into real-time data like weather and traffic to fine-tune the delivery experience. This means we can adjust delivery prices dynamically based on factors like how busy the roads are or if it's pouring rain outside.

And let's not forget about recommendations. We want to make it easy for our customers to find their new favorite spot. That's why we're serving up personalized recommendations based on things like user reviews, influencer endorsements, average costs, and different cuisines. It's like having a personal food critic in your pocket!

So, get ready to embark on a delicious journey with us as we revolutionize the way food gets from the kitchen to your doorstep. Let's eat!

# Organization

## Roles & Responsibilities, use case handled

|  |  |  |
| --- | --- | --- |
| **Name** | **Use case** | **Roles and responsibilities** |
| Sushant Huilgol | * Delivery Tracking and delivery * Enabling Fast Deliveries | * Make efficient use of Redis to store the order information and make sure the updated order information is available to the user at any given time. * With the help of the riders’ locations (who are idle) assign the nearest rider to deliver the order. |
| Vindhya Hegde | * Route Optimization for the rider * Enabling Fast Deliveries | * Utilizing Neo4j to import routes, locations, and addresses to construct maps. Using the shortest path algorithm to determine the quickest route between two points, while leveraging real-time traffic data via available open APIs to optimize and identify the fastest path. * With the help of location of rider and time of order placement, designing and implementing an algorithms to assign orders to riders, |
| Kamal Kumar Sardiwal | * Boosting Restaurant Exposure: Leveraging Influencer Reviews | * Representing relationships in Neo4j between influencers, restaurants, and users/ customers to drive efficient querying of graph data and derive insights about influencer recommendations, user connections, and restaurant popularity. * These insights include: Influencers' top picks, Top rated restaurants nearby within 10 kms, Cuisines with top rated restaurants, Top 5 budget-friendly restaurants nearby, Influencer reviews/ratings and customer reviews/ratings, and other restaurant details such as dishes, cuisine, average cost for two w.r.to each restaurant. |
| Kalpesh Patil | * Dynamically allocating the mode of transport based on various factors | * Using Redis to store the real-time traffic and weather updates from the API's and determine the best mode of transport to efficiently allocate delivery resources and MongoDB to handling the riders and maintain the delivery status. |
| Ambuj Solanki | * Dynamic Pricing | * Making use of MongoDB to calculate the delivery price based on various factors. |

## Meetings

|  |  |  |  |
| --- | --- | --- | --- |
| **Date** | **Topic** | **Outcome** | **Attendees** |
| 18.04.2024 | Project discussion | Feedback: To investigate interesting topics that can produce complex use cases. | Sushant Huilgol, Vindhya Hegde, Kamal Kumar Sardiwal, Kalpesh Patil, Ambuj Solanki |
| 22.04.2024 | Project & Use cases discussion | Few use cases were approved and rest were rejected as they were simple and we were asked to find challenging use cases w.r.t real world problems | Sushant Huilgol, Vindhya Hegde, Kamal Kumar Sardiwal, Kalpesh Patil, Ambuj Solanki |
| 23.04.2024 | Refined use cases. | Use cases were refined and presented and out of those use cases, 5 were approved and a few remarks were given on them. | Sushant Huilgol, Vindhya Hegde, Kamal Kumar Sardiwal, Kalpesh Patil, Ambuj Solanki |
| 26.04.2024 | Presenting POC on importing map data into Neo4j and live weather data in nodejs | A POC was shown on importing Map Data into Neo4j and a demo on finding the shortest path between 2 locations.  Another demo on getting live weather data into a nodejs application was also shown. | Sushant Huilgol, Vindhya Hegde, Kamal Kumar Sardiwal, Kalpesh Patil, Ambuj Solanki |
| 29.04.2024 | Simulation of object movement from source to destination and display of data flow diagram. | A simulation was shown where, after entering a source and destination, the shortest path was calculated and the marker moved from source towards destination. Also a Data Flow diagram was shown which was an integration of all use cases. A few suggestions were given w.r.t diagrams. | Sushant Huilgol, Vindhya Hegde, Kamal Kumar Sardiwal, Kalpesh Patil, Ambuj Solanki |
| 02.05.2024 | Project report progress.  AuraDB limitation  Exam clarification | * Write why the database was chosen in the project report. * Userstories clarification * Each UML diagram has been separated for individual use cases * AuraDB has limitations to 200000 nodes - huge demand for data due to maps * We need to add import statements into the tool section for documentation. * Transitioning to Google Cloud was decided * MongoDB and Redis Labs are set up, with documentation detailing their limitations and table record numbers. * During the demo, laptops will be switched according to the relevant use cases, if necessary, or the same screen will be used as per the project meeting outcome. * Presentation - Total time 30 min (5 mind or each) | Sushant Huilgol, Vindhya Hegde, Kamal Kumar Sardiwal, Kalpesh Patil, Ambuj Solanki |
| 08.05.2024 | Project report review | * Project report review. * Reviewed formatting, DFD and use case diagrams. | Sushant Huilgol, Vindhya Hegde, Kamal Kumar Sardiwal, Kalpesh Patil, Ambuj Solanki |
| 09.05.2024 | Project report Final review | * Reviewed updated MongoDB ER diagram and overall Database’s data flow diagram | Sushant Huilgol, Vindhya Hegde, Kamal Kumar Sardiwal, Kalpesh Patil, Ambuj Solanki |

## Tools Used

**Database:** Redis Lab Cloud, Mongo DB Atlas, Neo4j DB deployed on a VM in GCP

**UI:** Angular, Visual Studio Code, Leaflet for Map Simulation

**API:** NodeJS, Visual Studio Code

# User Story

As a hungry customer using the food delivery app, I want to have a seamless experience from selecting a restaurant to receiving my order, ensuring efficiency, accuracy, and satisfaction throughout the process. Upon opening the app, I aim to discover top-rated restaurants that align with my preferences and are recommended by influencers. This includes viewing influencers' top picks, nearby top-rated restaurants within a 10 km radius of my location, cuisine-based top-rated restaurants, and the nearby top 5 budget-friendly restaurants. I expect to see detailed information about each restaurant, including ratings and reviews from both customers and influencers, the cuisine offered, and the dishes associated with each restaurant.

Once I've selected a restaurant, I want to easily place an order to my address. After placing my order, I anticipate receiving a unique 4-digit code to ensure my order is delivered accurately. I also desire the ability to check the status of my order at any time and track the rider's location once my order is on its way. I appreciate knowing that the app optimizes the rider's route for each delivery, considering distance, real-time traffic conditions, and incidents along the way. This ensures timely deliveries and minimizes travel time and fuel consumption for the rider, enhancing overall customer satisfaction.

As a customer, I value transparency and fairness in pricing. I notice that the delivery fee adjusts based on factors like distance, weather conditions, and time of day. Despite slight fluctuations, I appreciate transparency and choose the delivery option that best suits my needs, feeling confident that I'm getting a fair deal for my delivery regardless of the circumstances.

Meanwhile, restaurant owners rely on the food delivery app to efficiently manage orders. They need to update the status of orders from preparing food to ready for pick-up, ensuring seamless coordination with delivery riders. They trust the app to intelligently select the most appropriate mode of transport based on factors such as distance, order size, types of food items, and current weather conditions. This dynamic allocation of transport modes ensures prompt and optimal deliveries, enhancing customer satisfaction and the restaurant's reputation.

Overall, the food delivery app strives to enhance the dining experiences of both customers and restaurant owners, providing a seamless, transparent, and efficient platform for ordering and delivering food.

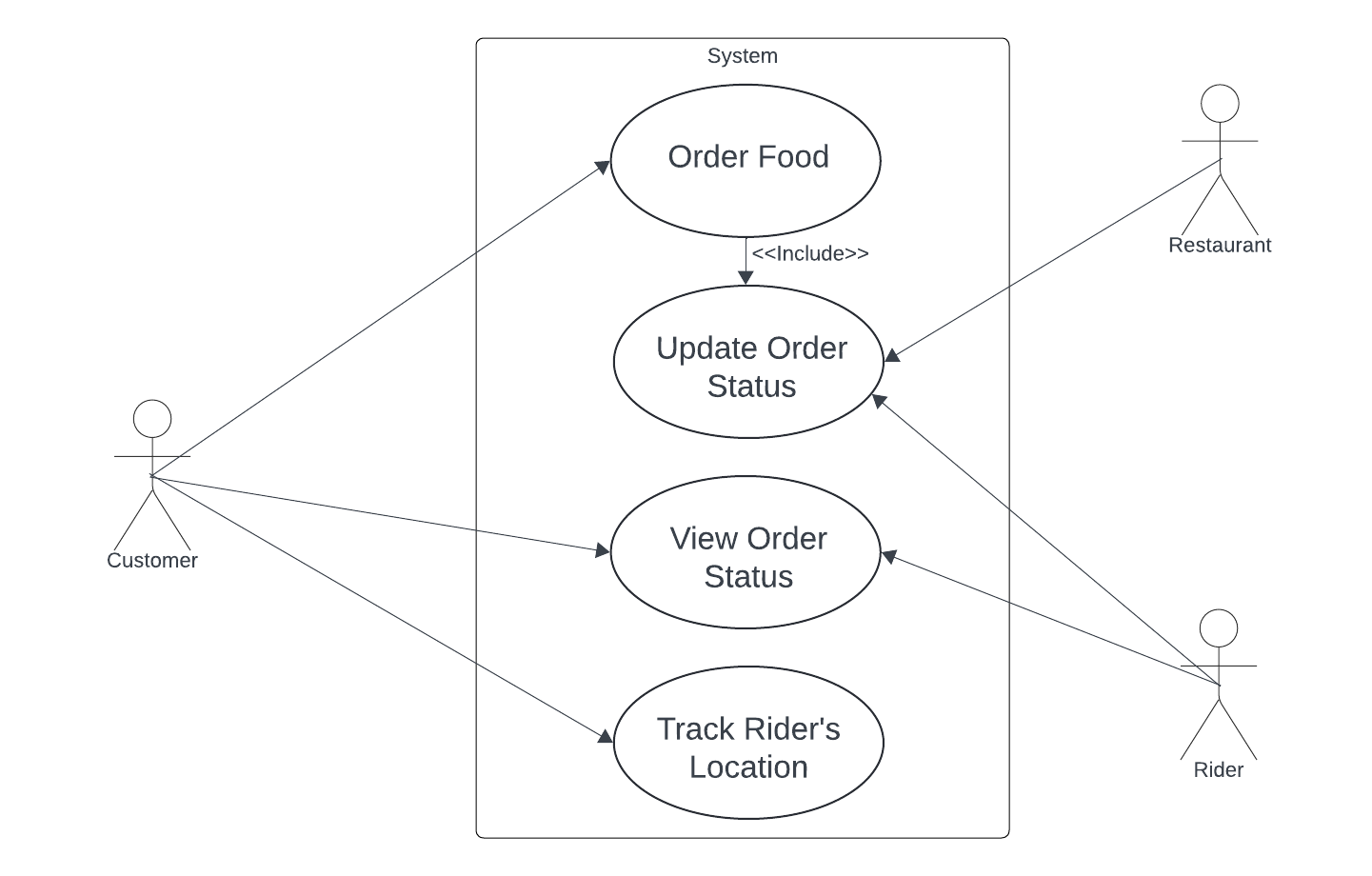
# Details

## Delivery Tracking and delivery (Sushant)

### User Story

As a hungry customer, I want to easily select a restaurant and place an order to my address. After placing my order, I expect to receive a unique 4-digit code to ensure my order goes to the right person. I also want to be able to check the status of my order whenever I want. Once the order is on its way, I'd like to track the rider's location. As a restaurant owner, I need to update the status of orders from preparing food to ready for pick-up. As a delivery rider, I want to be notified when an order is ready for pick-up. After picking up the order, I'll change the status to 'in transit.' When I receive the 4-digit code from the customer, I'll mark the delivery as completed.

### Use Case Diagram



### Actors

**Customer**: Represents individuals ordering food online. They select restaurants, place orders, track their order status, and provide a unique code for delivery confirmation.

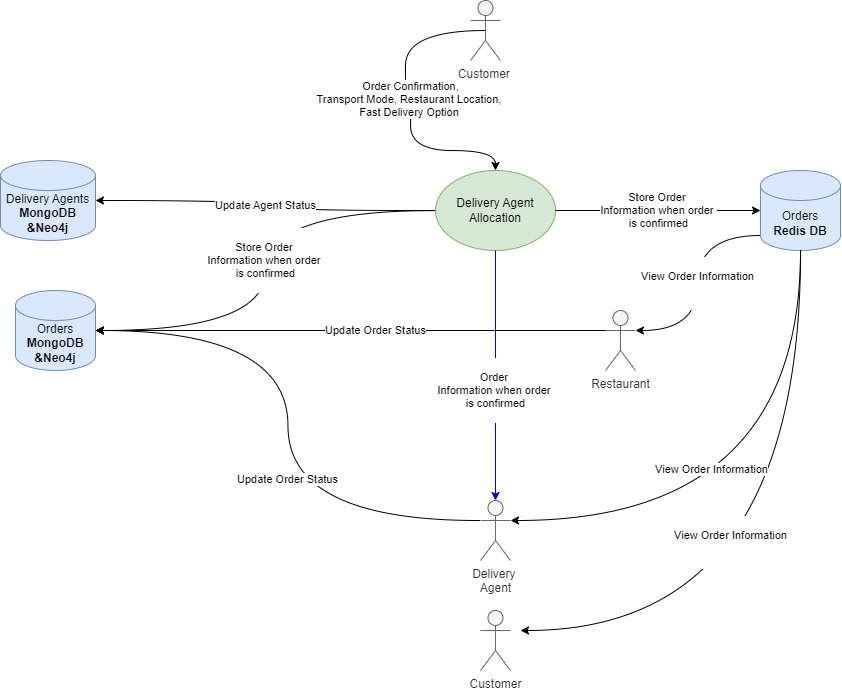
**Restaurant**: Represents establishments preparing and fulfilling food orders. They update order statuses, indicating when orders are ready for pick-up by delivery riders.

**Delivery Rider**: Represents individuals responsible for picking up orders from restaurants and delivering them to customers. They receive notifications when orders are ready for pick-up, update order statuses during transit, and confirm deliveries with customer-provided codes.

### Descriptions of task with all interactions between actor and database

* Customers place an order from a restaurant. This creates an entry into the mongodb database as an order created with a status as ‘order placed’ and also an entry in the redis database with necessary information for the customer.
* The object in the redis database is accessed by all the 3 actors associated to that order only. No other customer or restaurant or a rider can view it.
* The customer can only view the status to see what the current order status is. This will be fetched to the user from the redis database.
* The restaurant will be able to update the status to only ‘preparing dishes’ and ‘ready for pickup’. This change of status will be reflected only in the redis database.
* The rider will be able to pick-up the order and change the status to ‘in transit’. This will also only be reflected in the redis database.
* Once the order is delivered and the code is received from the Customer, the status will be updated to ‘delivered’. This will add the order object into the mongo db collection of order history and remove the entry from redis database.
* In order to simulate the rider’s live location to the customer through maps, the application will receive the path coordinates from neo4j and simulate it through interactive UI.

### Data Flow



Data Flow Diagram: Delivery Tracking

### Databases

#### Databases Used

* **Redis Database:** The redis database will be used to cache the order information from the moment it is placed to the time it will be delivered. This is because the customers would constantly keep accessing this data multiple times throughout the process and it would be efficient if it is accessed from cache memory instead of hitting the disk storage multiple times.
* **MongoDB:** MongoDB will be used to store the order in the orders database with a status as ‘order placed’, but will only be updated next when the order is delivered. This data acts as a dependency for other use cases such as dynamic pricing which takes orders with status as ‘orders placed’ as a factor to estimate demand and is also used in a queue to allocate a rider to those orders as the rider’s current locations will be set to those of

#### Expressions

const redisCustomerKey=(Buffer.from(customerDetails.name.toLowerCase().trim().replace(' ', '') + customerDetails.id).toString('base64'))

await redisClient.set(redisCustomerKey, JSON.stringify(customerDetails));

The above code is called when the user session is created. A unique user specific key is generated in Redis for that user.

redisKey = Buffer.from(customerName.toLowerCase().trim().replace(' ', '') + customerId).toString('base64');

customerObj = JSON.parse(await redisClient.get(redisKey));

if (deliveryCode !== customerObj.deliveryCode) {

return res.status(200).json({ message: "Delivery code invalid" })

}

customerObj.orderDetails = {};

await redisClient.set(redisKey, customerObj);

The above code is used to update the status of the order every time it is changed by the actors. For example, when the customer changes the order from preparing food to ready for pickup etc.

redisKey = Buffer.from(req.body.customerName.toLowerCase().trim().replace(' ', '') + req.body.customerId).toString('base64');

customerObj = JSON.parse(await redisClient.get(redisKey));

The above code is used to fetch the order details. When customer’s reload on their webpage, the data is accessed from this part of the code, which queries the Redis database for the current order data.

## Route Optimization for the rider (Vindhya)

### User Story

As a rider, I want the application to optimize my route for each delivery considering the distance, ensuring that I can reach the pick-up location and my delivery location, using the shortest possible path. This will help me minimize the travel time and fuel consumption, fulfill my deliveries promptly and enhance overall customer satisfaction. I also require the system to dynamically adjust my route based on real-time traffic conditions and incidents along the way. This feature will help me avoid congested areas, accidents, or road closures, enabling me to deliver orders as quickly as possible.

### Use Case Diagram

Use Case: Route Optimization

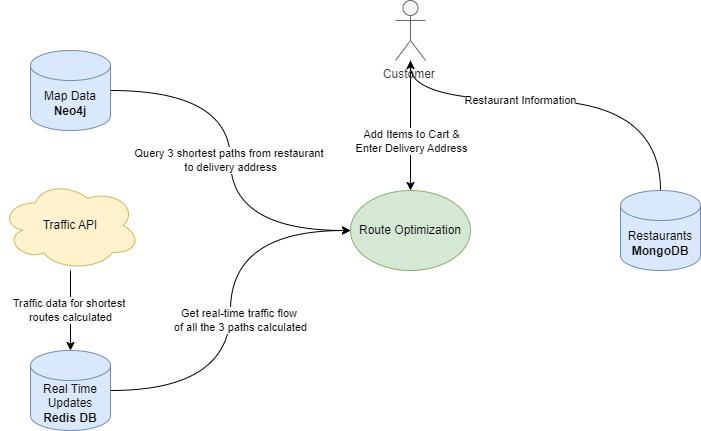
### Actors

**Rider:** The rider represents individuals tasked with collecting orders from restaurants and delivering them to customers. They utilize the application to guide them to the restaurant for order pickup and to the delivery location for swift delivery.

### Descriptions of task with all interactions between actor and database

* When an order is being placed, the customer enters the delivery location which is used to compute three shortest paths from restaurant to delivery location, with the help of maps imported in Neo4j.
* These routes are ordered based on the traffic condition in each of them which is used from TomTom Traffic API.
* The shortest and fastest route will then be used for dynamic pricing and allocating mode of delivery.
* All the three routes will also be stored in Redis for future rerouting considering real time traffic conditions.
* Once the order is placed and the rider is allocated, the application provides the agent with the shortest and quickest route, factoring in both traffic conditions and distance between their current location and the restaurant.
* After the order is picked up, the application computes the fastest route among the three routes stored in redis, based on the traffic conditions and presents it to the rider.

### Data Flow



Data Flow Diagram: Route Optimization

### Databases

#### 4.2.6.1 Databases Used

**Neo4j Database:** The Neo4j Graph Database is employed initially to establish a network between roads and intersections, effectively creating a map. This map serves as the basis for calculating the shortest routes between two or more points.

**Redis Database:** The Redis Database serves as a cache, storing the three shortest routes computed initially when an order is placed. These routes will later be utilized to reroute the rider, enabling them to use the fastest route considering the traffic conditions at that moment.

#### 

#### 4.2.6.2 Expressions

* Cypher query to calculate 3 shortest routes from restaurant to delivery location.

MATCH (to:Address{full\_address: $deliveryAddress})-[:NEAREST\_INTERSECTION]->(source:Intersection)

MATCH (from:Restaurant{name: $restaurantName})-[:NEAREST\_INTERSECTION]->(target:Intersection)

CALL gds.shortestPath.yens.stream('sanMateo',{sourceNode: source, targetNode: target, k: 3, relationshipWeightProperty: 'length'})

YIELD index, sourceNode, targetNode, totalCost, nodeIds, costs, path

RETURN index, totalCost,

[nodeId IN nodeIds | [ gds.util.asNode(nodeId).location.latitude, gds.util.asNode(nodeId).location.longitude]] AS path

ORDER BY index

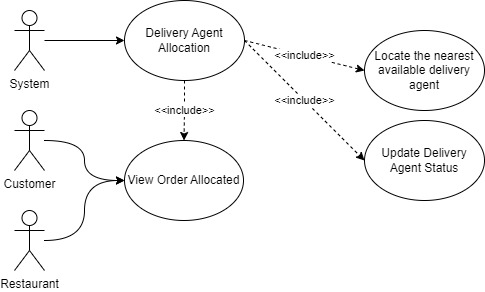
This Cypher query uses the Graph Data Science library to find the top 3 shortest paths between an address and a restaurant in the city of "sanMateo", utilizing intersections as nodes and road lengths as edge weights. It first matches the address and restaurant nodes, then computes the shortest paths using the Yen's algorithm, yielding information about the paths' indices, total costs or distance in meters, and coordinates, ultimately ordering the results by the index of the shortest paths.

## Enabling Fast Deliveries (Sushant & Vindhya)

### User Story

As a Customer, I would want the deliveries to be delivered to me as fast as possible. In order for that to happen it is necessary that a rider who is very close to the restaurant be associated with the restaurant rather than someone who is very far. As a Rider, I would want to be allocated to a restaurant close to me so that it saves my fuel, energy and I can make sure to deliver as many orders as possible in a day.

### Use Case Diagram



Use Case Diagram: Delivery Agent Allocation

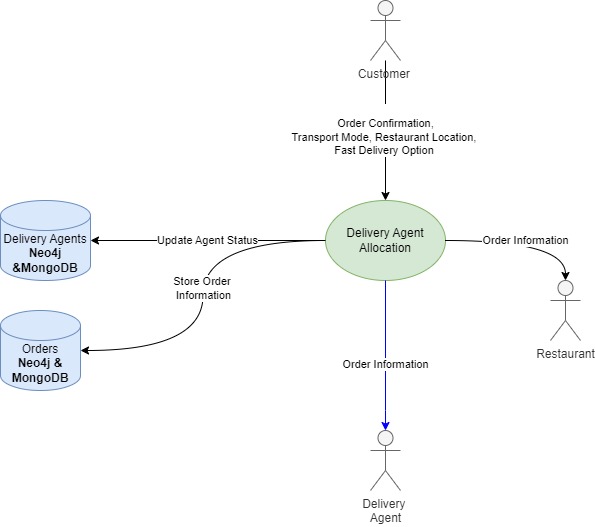
### Actors

**System**: Responsible for taking the current address of all the riders and finding the nearest rider to the restaurant. Takes in the input from the mode of transport process and filters the vehicles based on it.  
**Rider:** Once the order is allocated to a rider, the rider can view the order details, restaurant address and navigational route to restaurant.  
**Customer:** Once the order is allocated to a rider, the customer can view the updated status of the order.

### Descriptions of task with all interactions between actor and database

* When the order is placed by the customer, the system takes into consideration the mode of transport allocated for the order and checks the availability of riders, takes the location of all the riders from neo4j database and locates the nearest rider available to the restaurant and assigns him to the order.
* As soon as the order is allocated a rider, this is notified to customer, restaurant and the rider himself.
* The system also provides the rider with the route from the rider’s location to the restaurant using the powerful algorithms of neo4j.
* The status of that rider is changed to ‘busy’ in the MongoDB database as well as neo4j once an order is allocated to him. The rider is also provided with a code for order pick-up database.

### Data Flow



Data Flow Diagram: Rider Allocation

### Databases

#### Databases Used

* **Neo4j Database:** The neo4j database is used to get the location to the nearest rider using each rider’s latitude and longitude. This is then fed to the algorithms in neo4j to find the nearest rider.
* **MongoDB:** MongoDB will be used to update the status of the allocated rider to ‘busy’ or ‘idle’ once the rider is allocated an order or completes an order delivery. This will help other use cases as well to carry out their functionalities.

#### Expressions

* Cypher query to calculate the nearest delivery driver from the restaurant to enable fast delivery

MATCH (to:Restaurant{name:$restaurantName})-[:NEAREST\_INTERSECTION]-> (destination:Intersection)

WITH destination, to

MATCH (from:Rider{status: 'available', vehicleType: $vehicleType})- [:NEAREST\_INTERSECTION]->(source:Intersection)

WITH source, destination, collect(from) as sourceNodes, to

UNWIND sourceNodes as s1

CALL gds.shortestPath.dijkstra.stream('sanMateo', {

sourceNode: source,

targetNode: destination,

relationshipWeightProperty: 'length'

})

YIELD index, sourceNode, targetNode, totalCost, nodeIds, costs, path

RETURN

s1.id AS riderId,

s1.name AS riderName,

s1.location AS riderLocation,

to.address+", "+to.city+", "+to.zip AS restaurantAddress,

totalCost,

[nodeId IN nodeIds | [gds.util.asNode(nodeId).location.latitude, gds.util .asNode(nodeId).location.longitude]] AS path

ORDER BY totalCost

LIMIT 1

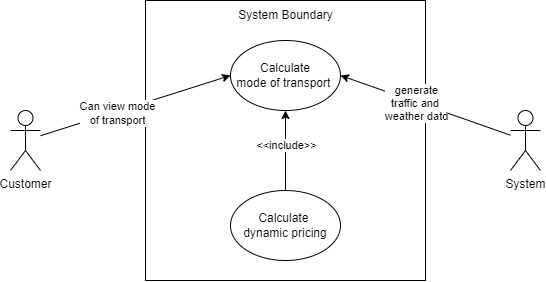
The above cypher query takes restaurant name for the destination input and the locations of all the riders who are associated with the mode of transport allocated and with status as idle as the source inputs. The query then locates the nearest intersection of the restaurant and riders and finds the shortest distance using Dijkstra's shortest path algorithm. The results are sorted in ascending order with respect to weights, which in this case is the distance in meters. The query results with the basic details of the rider who is nearest to the restaurant along with the path to the restaurant.

## Dynamically allocating the mode of transport based on various factors (Kalpesh Patil)

### User Story

Restaurants rely on the food delivery app to efficiently deliver orders to customers. They expect the app to intelligently select the most appropriate mode of transport based on factors such as the distance to the delivery address, the size of the order, the types of food items being delivered (with consideration for delicate items), and the current weather conditions. By dynamically allocating the mode of transport, the app ensures that orders are delivered promptly and in optimal condition, enhancing customer satisfaction and the reputation of the restaurant.

### UML: Use Case



Use Case Diagram: Calculating Mode of Transport

### Actors

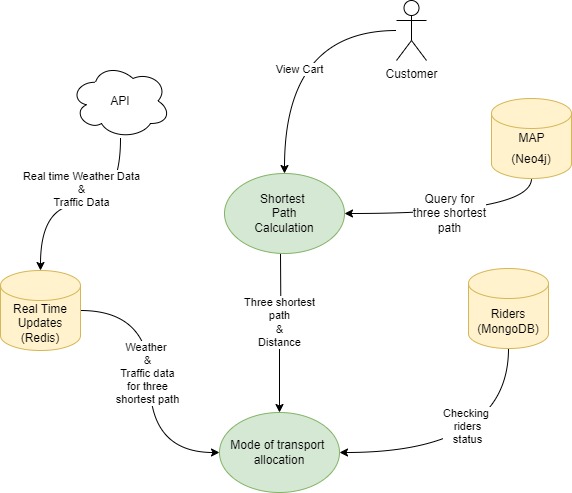
**Customer :** Represents individuals interacting with the system. Once the mode of transportation is selected, then they can see it on their screen.

**System:** System will gather factors such as traffic data and weather data from the APIs. Utilize this data, pass it to the process as input.

### Description of tasks with all interactions between actor and database.

* When the customer checks the cart, the shortest path will be calculated based on the user details and it will be passed to determine the mode of transport before riders pick up the order.
* The system continuously fetches real-time updates on weather conditions and traffic congestion through an API, storing this information in a Redis database for quick access.
* Following this, the system evaluates various factors including real-time traffic and weather data, distance between the customer and restaurant, and order type to determine the most suitable mode of transport.
* When the mode of transport is determined, the system will check the rider's database to see if the rider for the selected mode of transport is available and then allocates a delivery agent to safely deliver the order to the customer.
* The data in the Redis will be updated at some time interval to help riders to get the real time traffic data and weather data for the selected route to deliver the order.

### Data Flow



Data Flow Diagram: To determine the Mode of Transport

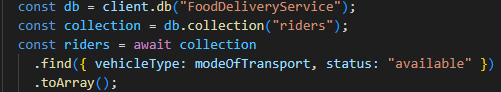
### Databases

#### 4.4.6.1 Databases Used

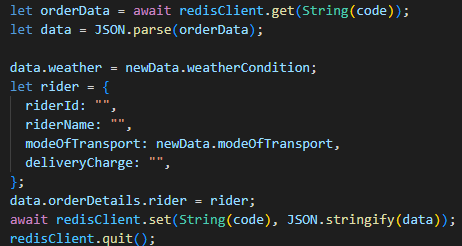
* **MongoDB:** MongoDB will be used to store rider information such as names, addresses, and contact details, as well as type of vehicle and availability. Its flexible document-based structure makes it suitable for managing dynamic rider information efficiently. This information later will be used to determine the rider based on the mode of transport.
* **Redis:** Redis will be utilized to store real-time updates on traffic conditions and weather. Its fast in-memory caching capabilities make it ideal for storing and retrieving frequently changing data, ensuring that the system has access to the most up-to-date information for optimizing delivery routes and determining the mode of transport.

#### 4.4.6.2 Expressions used for this Use Case

1. Query to fetch the data from the rider's database (MongoDB) to check the availability of riders during selection of mode of transfer.



2. Query to set the real time weather data, calculated mode of transport data and initialize the rider object to the Redis database. Further data will be used to calculate the dynamic pricing.



## Boosting Restaurant Exposure: Leveraging Influencer Reviews (Kamal Kumar Sardiwal)

### User Story

* As a user/ customer, I want to discover top-rated restaurants based on my preferences and those recommended by influencers to enhance my dining experiences.
* This includes viewing influencers' top picks, top-rated restaurants that are nearby, within a 10 kms radius of my location, cuisine-based top-rated restaurants, the nearby top 5 budget-friendly restaurants, ratings and reviews of both customers and influencers upon clicking on a restaurant, in addition to that I would like to see the cuisine offered, average cost for two and the dishes associated with each restaurant.

### UML: Use Case diagram



### Actors

* User (Customer): Engages with the application to discover top-rated restaurants based on preferences, location, budget, and Influencer recommendations.
* Influencer: Contributes expert reviews and recommendations for restaurants.
* System: Facilitates interactions between users and influencers, manages restaurant data, and presents reviews and recommendations to users.

### Description of tasks with all interactions between actor and database.

**1. User Interaction**: **Explore Top-Influencer 5 picked Restaurants:**

* + Interaction: The user interacts with the user interface (UI) by rendering the "Top 5 Influencer Picks" card.
  + System Action: The system receives the user's input and initiates a query to retrieve influencer recommendations from Neo4j.
  + Results are displayed to the user.

**2. User Interaction**: **View Restaurant Details:**

* + User clicks on a restaurant to view details.
  + The system retrieves detailed information from the database and displays it to the user which includes Name, Dishes, Cuisine, and Cost for two.

**3. User Interaction: View Customer & Influencer Reviews:**

* User clicks on a restaurant to view details.
* The system retrieves Influencer reviews by filtering based on followers and customer reviews alongside and displays it to the user.

**4. System calculates the average cost for two for each restaurant:**

* Interaction: The system processes the data retrieved from Neo4j.
* System Action: Utilizing Cypher query language and aggregation functions, the system calculates the average cost for two people at each restaurant.

**5. System displays nearby top 5 rated restaurants:**

* Interaction: The system initiates a query to Neo4j to retrieve the customer's address, designated as 'CustomerName', along with their location
* System Action: The system retrieves the customer's location from Neo4j and calculates the distance to each restaurant. Then, it aggregates the ratings of each restaurant and selects the top 5 highest-rated options within a 10km radius of the customer's location. Finally, it orders the results by average rating and displays them on the UI for the user.

**6. System displays the nearby top 5 budget-friendly restaurants:**

* Interaction: The system filters the list of restaurants based on the calculated average cost for two and the distance from the customer's location, retrieved from Neo4j and ensures they have been reviewed.
* System Action: Utilizing Cypher query language and aggregation functions, the system calculates the average cost for two people at each restaurant. After calculating the average cost for two, calculates the distance between the user/customer and the restaurant, and reviewing the restaurants, the system identifies the top 5 budget-friendly options within user’s proximity and prepares them for display on the UI, optimizing user experience.

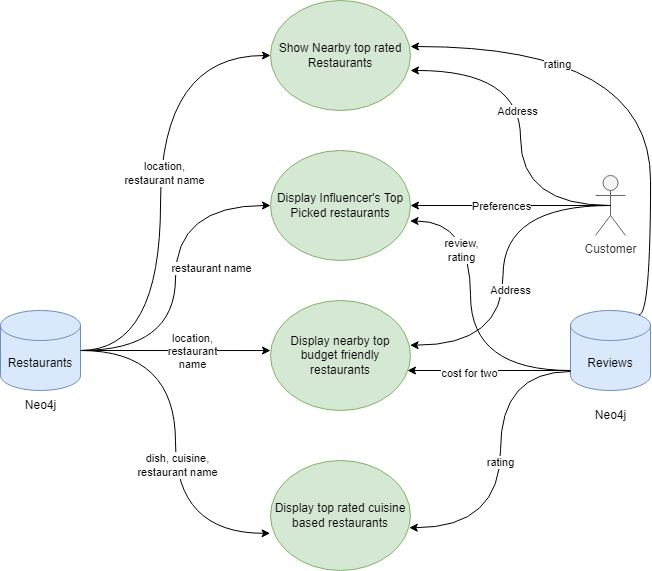
**7. System displays top-rated restaurants under each specific cuisine type:**

* + Interaction: The system organizes restaurants based on their associated cuisines and ratings.
  + System Action: After associating restaurants with cuisines, the system identifies the top-rated restaurants for each specific cuisine type using Cypher queries and prepares them for display on the UI.

**8. Database Interaction: Data Retrieval:**

* + System queries the database for restaurant information, including name, location, cuisines, reviews, ratings, and followers.

### Data Flow

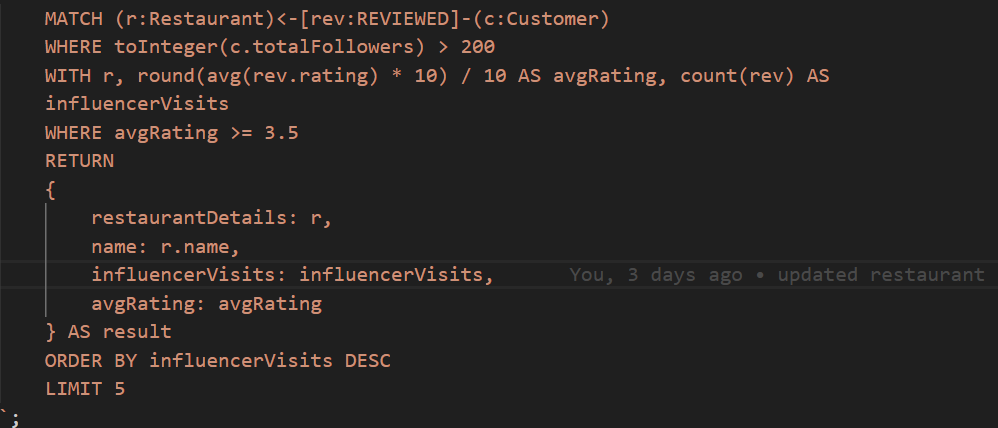
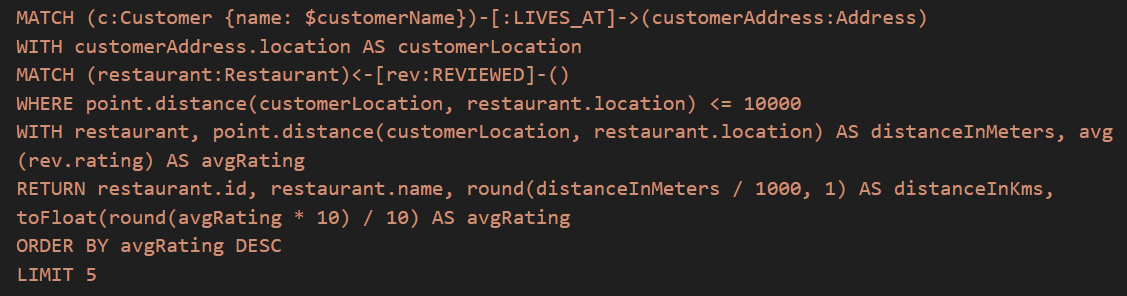


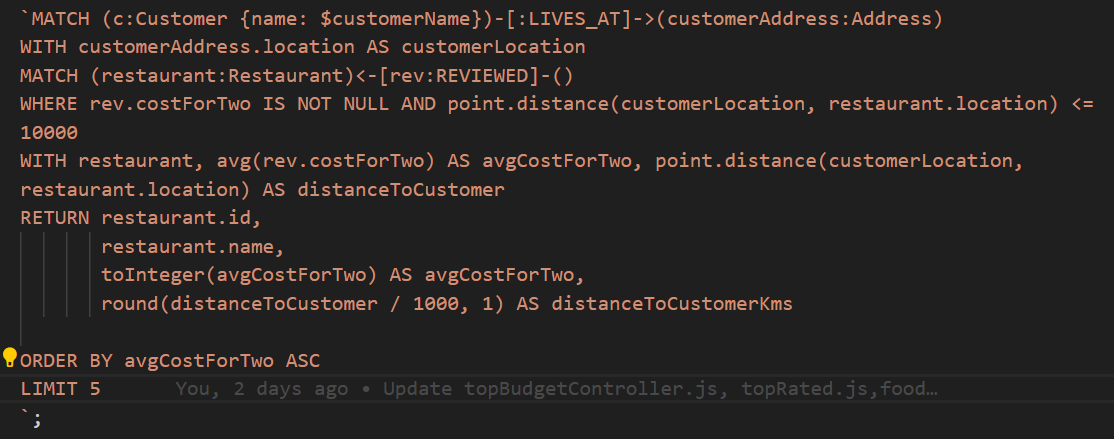
### Databases

#### Database Used

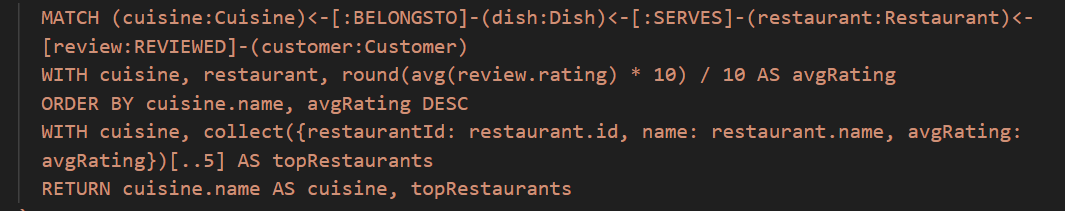
* Neo4j is chosen as the database management system for this use case due to its graph database model, which is well-suited for representing relationships between influencers, restaurants, and users. The flexibility and scalability of Neo4j enable efficient data retrieval and traversal, essential for implementing complex querying and analytical tasks in this scenario.

#### 4.5.6.2 Expressions used for this Use Casesus

1. Top 5 Influencer picks: This Cypher query retrieves restaurants that have been reviewed by customers with more than 200 followers, calculates their average rating rounded to one decimal place, and counts the number of influencer visits for each restaurant. It filters out restaurants with an average rating less than 3.5 and returns the top 5 restaurants ordered by the number of influencer visits.
2. Top 5 Rated restaurants nearby: This expression retrieves the customer's location and then matches restaurants within a 10-kilometer radius of that location. It calculates the distance between each restaurant and the customer in kilometers and calculates the average rating for each restaurant. Finally, it returns to the top 5 restaurants ordered by average rating. 
3. Top 5 Budget-friendly restaurants nearby: This Cypher query gets restaurants within 10 kilometers of a specified customer's location, calculates the average cost for two people, calculates the distance between the customer and the restaurant, converts the distance to kilometers rounded to one decimal place, and retrieves the restaurant name, average cost, distance to the customer, and detailed restaurant information. It then orders the results by average cost in ascending order and limits the output to the top 5 budget-friendly restaurants within the customer's proximity.



1. Top Restaurants based on cuisine: This Cypher query retrieves the top 5 restaurants for each cuisine based on their average ratings. It first matches restaurants reviewed by customers, calculates the average rating for each restaurant, orders the results by cuisine name and average rating, and then collects the top 5 restaurants along with their average ratings for each cuisine. Finally, it returns the cuisine name along with the list of top restaurants for each cuisine.

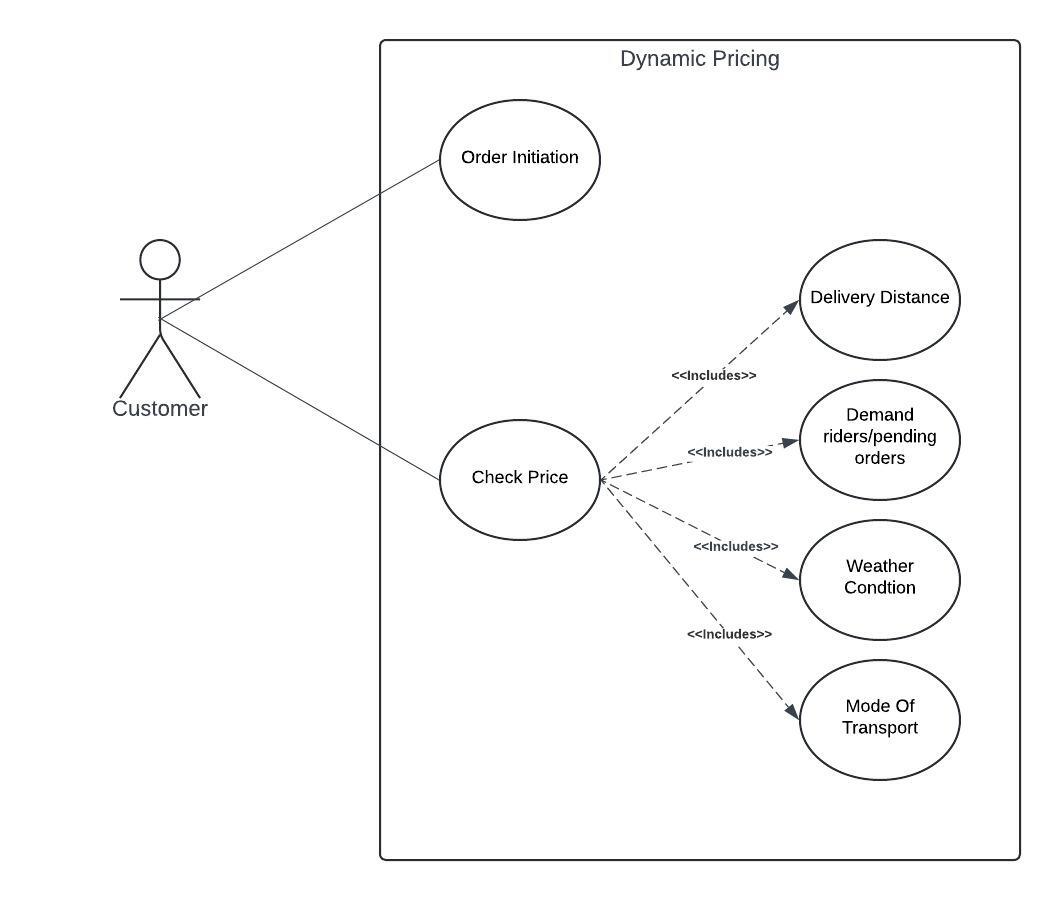


## Dynamic Pricing ( Ambuj Solanki )

### User Story

In a busy city, customers open their food delivery app to order dinner. Each of them notices that the delivery fee adjusts based on factors like distance, weather conditions, and time of day. Despite the slight fluctuations, they appreciate the transparency and choose the delivery option that best suits their needs. With dynamic pricing, they all feel confident knowing they're getting a fair deal for their deliveries, no matter the circumstances.

### UML Diagram



### Actors

**Customer** - They are the end-users interacting with the application after they decide on their order they proceed to check out page where they will be checking the delivery fees

### Description of tasks with all interactions between actor and database.

**Customer Proceeds to Checkout:**  
After adding items to the cart, the customer proceeds to checkout to finalize the order.

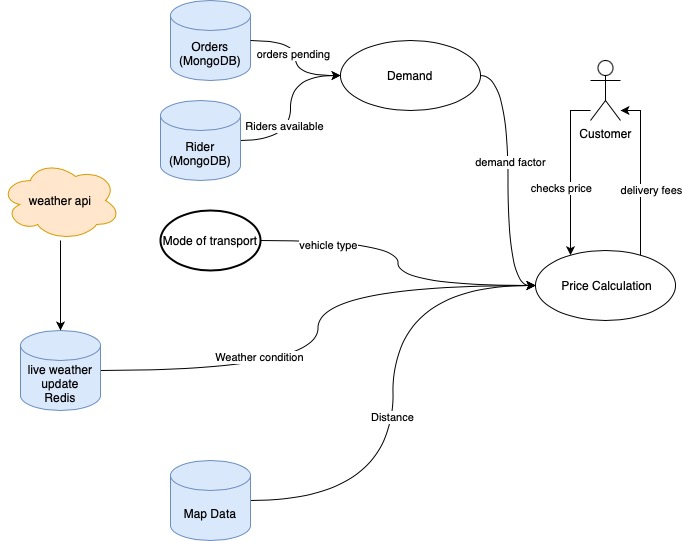
**Calculation of delivery fees:**  
The app needs to calculate the delivery fee based on several factors:

* Distance: The app calculates the distance between the selected restaurant and the customer's delivery address.
* Weather Conditions: The app checks the current weather conditions in the delivery area. If adverse weather conditions (such as rain, snow, or extreme temperatures) are detected, the delivery fee might be adjusted to account for potential delays
* Demand: The app analyzes the current demand for delivery services at a given time. If demand is high, the delivery fee might increase slightly to incentivize more delivery drivers to meet the increased demand.
* Mode of Transport: The app assesses the mode of transport available for delivery (e.g., bicycle, car, motorcycle) and the associated costs and efficiencies. Each mode of transport may have different operating costs and delivery times, which influence the delivery fee.

**Interaction with database:**  
The app retrieves all the factors mentioned above from the mongoDB to calculate the delivery fees

**Price Display:**  
After all the calculations the delivery fees are displayed to the customer

### Data Flow



### Databases

#### Database used

**Mongo DB:**

Using MongoDB for dynamic pricing ensures flexible storage of pricing data, scalability to handle growing datasets, real-time updates for accurate pricing information, and powerful aggregation capabilities for sophisticated pricing strategies in the food delivery app.

Redis:

Redis database is use to fetch the user specific weather and the dynamic price after the calculation is stored back in the redis with user specific.

* + - 1. Expression for this use case

1. Demand calculation

This expression retrieves the data based on the mode of transport from the mongoDB. It fetches from 2 collections orders and riders. All the orders that are pending and require the given mode of transport. Then from riders it fetches all the active driver who has that kind of specific mode of transport.

ordersCollection.countDocuments({

orderStatus: 'Pending',

"rider.modeOfTransport": {"$regex": mode.toString(), "$options": "i"}

});

ridersCollection.countDocuments({

status: { $ne: 'offline' },

vehicleType: { "$regex": mode.toString(), "$options": "i" }

});

1. Weather:

This expression retrieves the weather data specific to the user from the Redis database. Which will be used to calculate the weather factor for the dynamic pricing.

code = Buffer.from(

customerName.toLowerCase().trim().replace(" ", "") + customerId

).toString("base64");

redisClient.get(String(code));

1. Delivery fees:

This expression after calculation will be stored in the user specific data. The value is stored in the Redis database as cache memory.

code = Buffer.from(

customerName.toLowerCase().trim().replace(" ", "") + customerId

).toString("base64");

orderData = await redisClient.get(String(code));

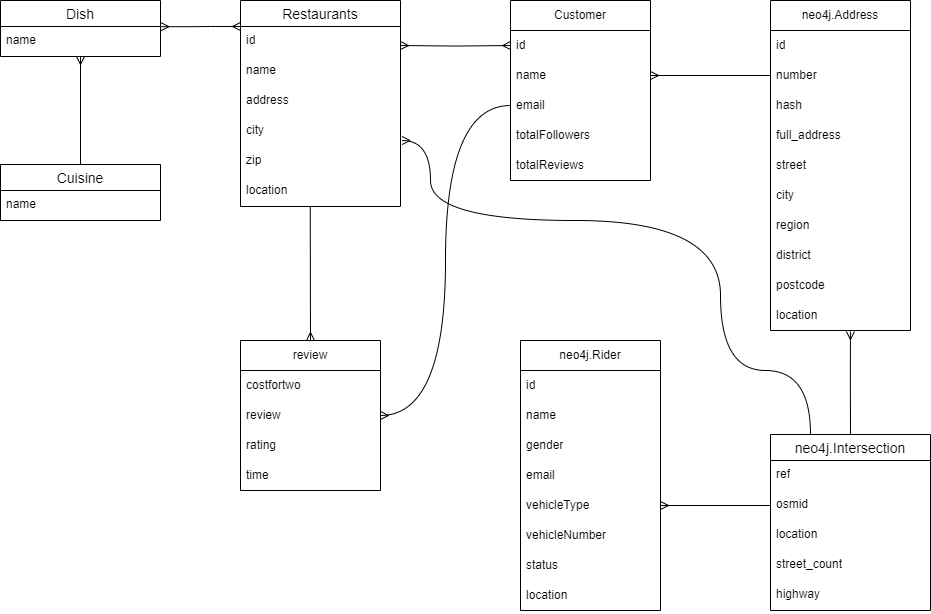
let data = JSON.parse(orderData);

data.orderDetails.rider.deliveryCharge = newCharge;

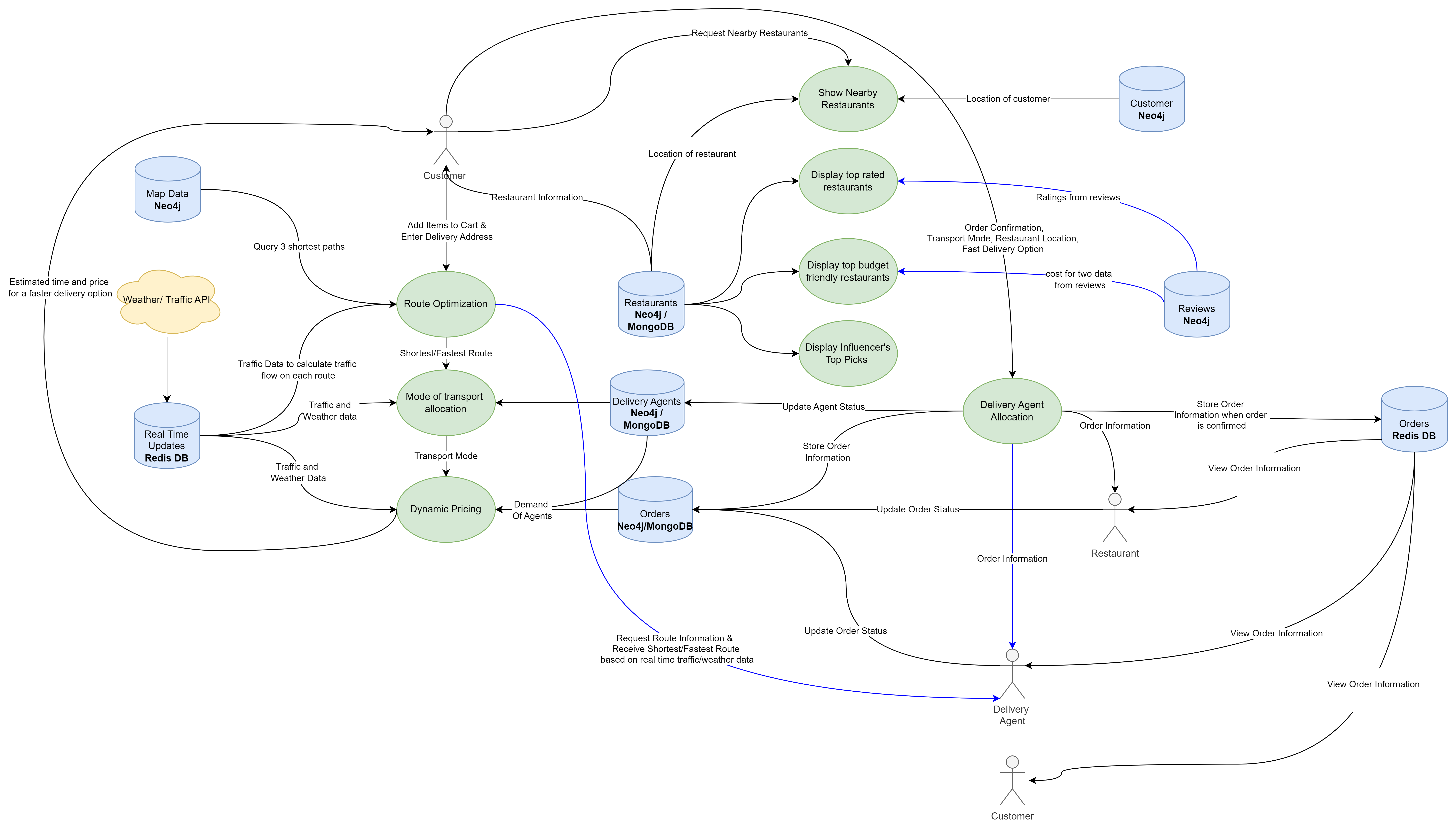
await redisClient.set(String(code), JSON.stringify(data));

# Database

## Overall structure



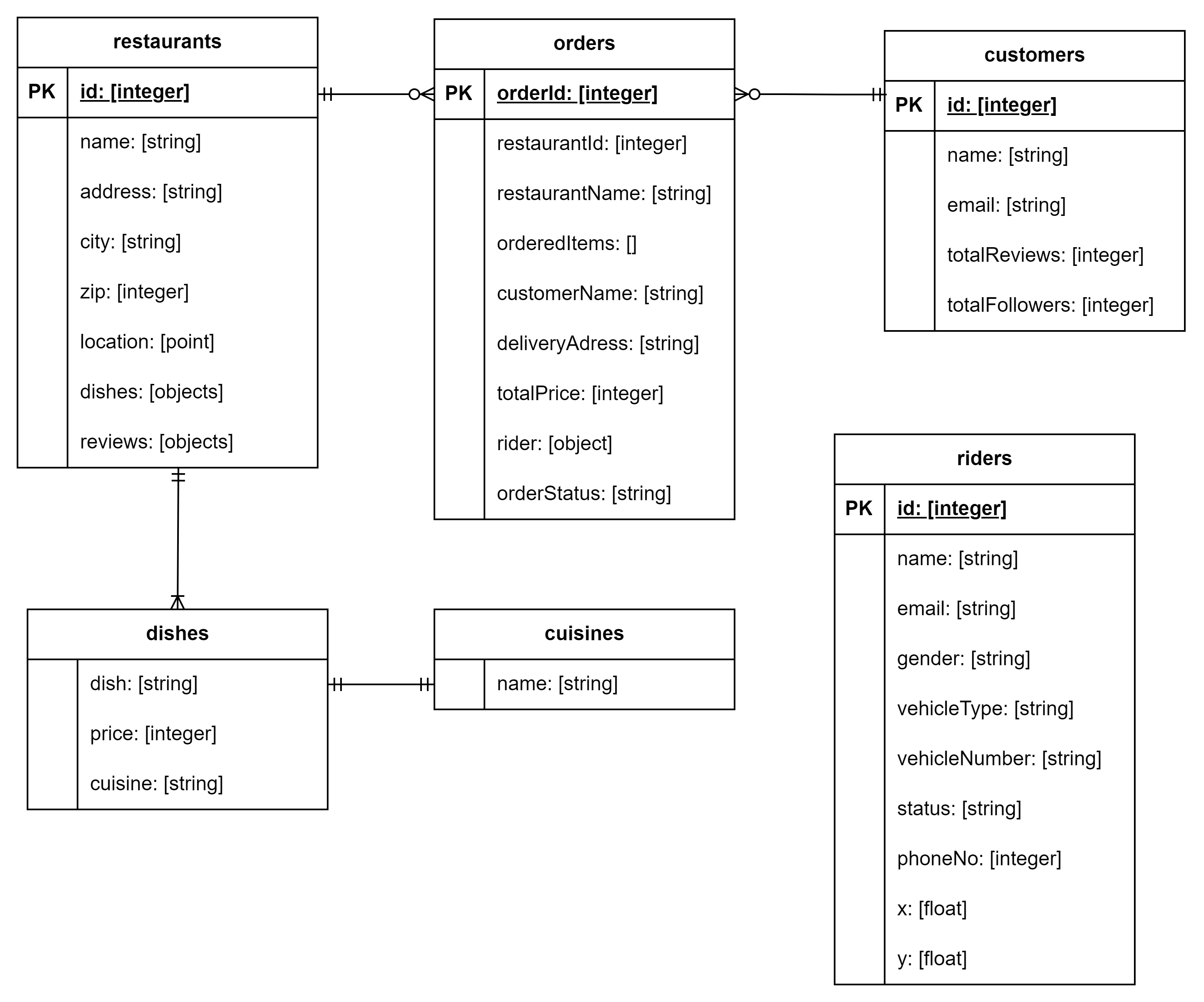
Overall Database ER Diagram



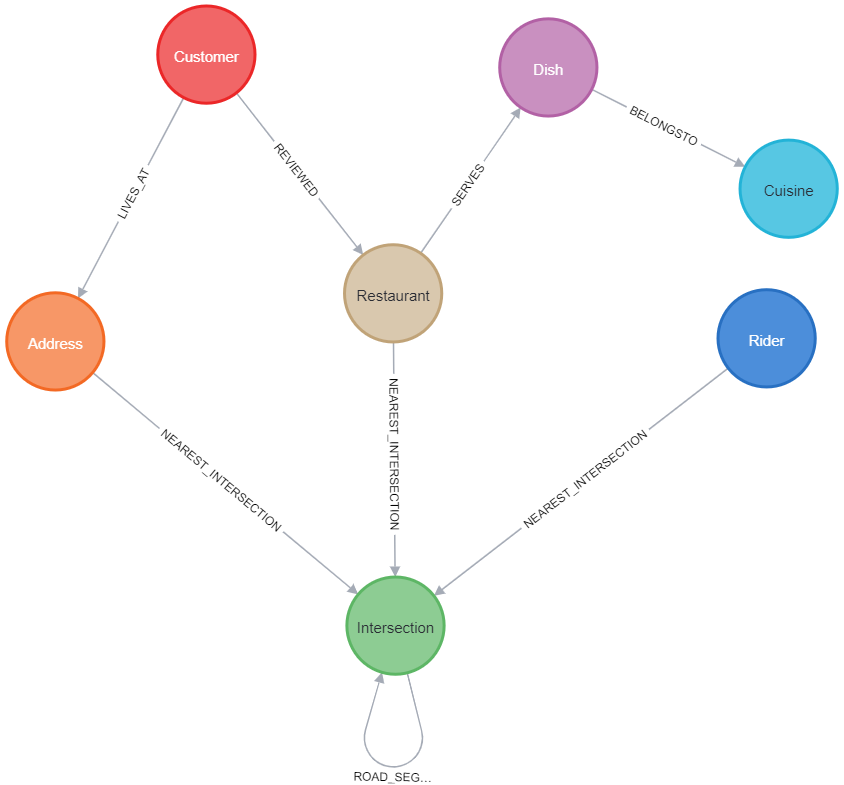
Overall Data Flow Diagram (DFD)

## Data Model Overview of all Database

### Mongo DB

Mongo DB: ER Diagram

### Neo4j

Neo4J: Schema Visualization

### Redis

* Order details, calculated routes, weather, traffic data specific to customer when the order is placed.

{

"shortestPaths": [

{

"path": [ [ 32.97689, -122.8979], [ 32.9999, -122.8999].....],

"distance": 2345,

"totalTravelTime": 30

},

{

"path": [ [ 32.97689, -122.8979], [ 32.9999, -122.8999].....], "distance": 2350,

"totalTravelTime": 33

},

{

"path": [ [ 32.97689, -122.8979], [ 32.9999, -122.8999].....],

"distance": 2545,

"totalTravelTime": 35

}

],

"weather": "snowy",

"orderDetails": {

"orderId": "123",

"restaurantId": 123,

"restaurantName": "Red Keep",

"orderedItems": [

{

"dish": "Waffles - choco",

"price": "123",

"quantity": 2

},

{

"dish": "Waffles - cherry",

"price": "123",

"quantity": 2

}

],

"customerName": "Jon Snow",

"deliveryAddress": "Belrade Ave Cross",

"totalPrice": 2300,

"rider": {

"riderId": "123",

"riderName": "Zoro Stark",

"modeOfTransport": "Car",

"deliveryCharge": "200",

"pickupCode": "1020",

},

"deliveryCode": "9090",

"orderStatus": "Pending"

}

}

## Used Expressions

### MongoDB

* When the order is confirmed, the order details are stored in mongo DB for future evaluation of demand, rider availability, order status maintenance. The JSON structure of order collection is shown below.

{

"shortestPaths": [

{

"path": [ [ 32.97689, -122.8979], [ 32.9999, -122.8999].....],

"distance": 2345,

"totalTravelTime": 30

},

{

"path": [ [ 32.97689, -122.8979], [ 32.9999, -122.8999].....], "distance": 2350,

"totalTravelTime": 33

},

{

"path": [ [ 32.97689, -122.8979], [ 32.9999, -122.8999].....],

"distance": 2545,

"totalTravelTime": 35

}

],

"weather": "snowy",

"orderDetails": {

"orderId": "123",

"restaurantId": 123,

"restaurantName": "Red Keep",

"orderedItems": [

{

"dish": "Waffles - choco",

"price": "123",

"quantity": 2

},

{

"dish": "Waffles - cherry",

"price": "123",

"quantity": 2

}

],

"customerName": "Jon Snow",

"deliveryAddress": "Belrade Ave Cross",

"totalPrice": 2300,

"rider": {

"riderId": "123",

"riderName": "Zoro Stark",

"modeOfTransport": "Car",

"deliveryCharge": "200",

"pickupCode": "1020",

},

"deliveryCode": "9090",

"orderStatus": "Pending"

}

}

### Neo4j

* Cypher Query used to import road network nodes as Intersection

UNWIND $rows AS row

WITH row WHERE row.osmid IS NOT NULL

MERGE (i:Intersection {osmid: row.osmid})

SET i.location =

point({latitude: row.y, longitude: row.x }),

i.ref = row.ref,

i.highway = row.highway,

i.street\_count = toInteger(row.street\_count)

RETURN COUNT(\*) as total

* Cypher query to import road network relationships connecting intersections as ROAD\_SEGMENT

UNWIND $rows AS road

MATCH (u:Intersection {osmid: road.u})

MATCH (v:Intersection {osmid: road.v})

MERGE (u)-[r:ROAD\_SEGMENT {osmid: road.osmid}]->(v)

SET r.oneway = road.oneway,

r.lanes = road.lanes,

r.ref = road.ref,

r.name = road.name,

r.highway = road.highway,

r.max\_speed = road.maxspeed,

r.length = toFloat(road.length)

RETURN COUNT(\*) AS total

* Cypher query to load addresses data of San Mateo downloaded from OpenAddresses.io

CALL apoc.periodic.iterate(

'CALL apoc.load.json("https://github.com/VindhyaKeshav/addresses/raw/main/source.geojson") YIELD value',

'MERGE (a:Address {id: value.properties.id})

SET a.location =

point(

{latitude: value.geometry.coordinates[1], longitude: value.geometry.coordinates[0]}),

a.full\_address = value.properties.number + " " + value.properties.street + " " + value.properties.city + ", CA " + value.properties.postcode

SET a += value.properties',

{batchSize:10000, parallel:true})

* Cypher query to connect each address node to network node at the nearest intersection by NEAREST\_INTERSECTION relationship

CALL apoc.periodic.iterate(

'MATCH (p:Address) WHERE NOT EXISTS ((p)-[:NEAREST\_INTERSECTION]->(:Intersection)) RETURN p',

'CALL {

WITH p

MATCH (i:Intersection)

USING INDEX i:Intersection(location)

WHERE point.distance(i.location, p.location) < 200

WITH i

ORDER BY point.distance(p.location, i.location) ASC

LIMIT 1

RETURN i

}

WITH p, i

MERGE (p)-[r:NEAREST\_INTERSECTION]->(i)

SET r.length = point.distance(p.location, i.location)

RETURN COUNT(p)',

{batchSize:1000, parallel:false})

# Application

## Language Used & Why

Following are the programming languages we used throughout the project for API and frontend:

* **JavaScript (NodeJS )**: We used it for writing code related to all the APIs as it doesnot need any additional setup. Our previous experience of writing APIs in JavaScript was one of the major factors behind selecting the language.
* **Python:** To load the data related to maps like roads, intersections, addresses we used python as it offers us powerful libraries that help us load and process the data quickly.
* **Typescript:** As we used Angular for developing the frontend application, we used typescript for UI related logic and routing.
* **HTML & CSS:** HTML and CSS were used for designing interactive UI web pages as they are the basic web page building languages and easy to write code in.

## GitHub Path

Following is the GitHub path which includes both client-side, server-side code, import queries for neo4j. It also includes a Readme file which includes instructions to set up the project onto your machine. The repository is set as public. Hence, anyone with the link can access and clone the repository.

Repository Link Source Code: <https://github.com/huilgolsushant1/FoodDeliveryServices>

## Methods/Functions

### Calculate Shortest Path, Mode of Transport, Pricing

const checkPrice = async (req, res) => {

try {

let reqObj = req.body;

let shortestPaths = await calculateShortestPath(

reqObj.restaurantName,

reqObj.deliveryAddress

);

shortestPaths = Array(shortestPaths).sort(

(a, b) => a.travelTime - b.travelTime

);

let modeAndWeather = await selectModeOfTransport(

shortestPaths[0][0],

reqObj.orderedItems,

reqObj.customerName,

reqObj.customerId

);

let deliveryCharge = calculateTotalPrice(shortestPaths[0].distance, reqObj.customerName, reqObj.customerId, modeOfTransport)

let customerDetails = JSON.parse(

await redisClient.get(

Buffer.from(

reqObj.customerName.toLowerCase().trim().replace(" ", "") +

reqObj.customerId

).toString("base64")

)

);

let response = {

orderDetails: reqObj

}

response.orderDetails.rider = modeAndWeather.orderDetails.rider;

response.orderDetails.weather = modeAndWeather.orderDetails.weather;

response.shortestPaths = shortestPaths;

response.orderDetails.rider.deliveryCharge = deliveryCharge;

res.status(200).json({

success: true,

message: "Price calculated",

data: response,

});

} catch (e) {

console.log(e);

res.status(500).json({

success: false,

message: e.message,

});

}

};

This function integrates Neo4j queries for calculating shortest paths, determining the mode of transport, and dynamically calculating the final price based on the route parameters. It returns the final price to the customer, combining these three use cases seamlessly.

# API

## Foreign API Description

We have included foreign APIs from TomTom for accessing real-time data related to traffic and weather. The weather data that we used is based on the city name and will be generic in all the areas. Traffic data will be calculated to paths based on the latitudes and longitudes. This data will be accessed regularly during delivery process to get traffic and weather information to find the fastest path, to select the mode of transport and to dynamically increase the price for delivery based on the weather condition and other factors.

* **Weather API**: The weather API gives us detailed weather information about the location which we pass in the parameters while fetching the data. Location can be provided either by name or by coordinates. From the detailed information we extract the weather conditions (e.g. rainy, snowy, sunny, etc.). This information is then used in two use cases i.e., Dynamic Pricing and Mode of Transport
* **Traffic API:** Tom Tom’s traffic route API calculates a route between an origin and a destination, passing through waypoints (if specified) and it aslo gives data to determine the traffic density. Based on which, the mode of transport and the total travel time for shortest path is being calculated.

### Example Data

* **Weather API:**

**Parameters:**

**location** = "heidelberg";

**apiKey** = 'f5b7ac4a88014420b2e163532240705';

**apiUrl** = `https://api.weatherapi.com/v1/current.json?key=${apiKey}&q=${location}`;

**Response**:  
{"weatherCategory":"sunny"}

* **Traffic-Route API:**

Parameters:

**API key** : eqi3q2O1zlkovRfuMnjXxg3Ev0GAkD5N

**sourceCoords**: 19.014473,72.835821:

**destinationCoords**:18955312091628045,72.83202161572633

**modeOfTransport: “**car**”**

**API URL**: `https://api.tomtom.com/routing/1/calculateRoute/${**sourceCoords**}:${**destinationCoords**}/json?key=${**apiKey**}&traffic=true&computeTravelTimeFor=all&routeType=fastest&language=en&instructionsType=tagged&travelMode=${**modeOfTransport**}`;

**Response:**

"summary": {

"lengthInMeters": 7492,

"travelTimeInSeconds": 871,

"trafficDelayInSeconds": 0,

"trafficLengthInMeters": 0,

"departureTime": "2024-05-09T03:43:45+05:30",

"arrivalTime": "2024-05-09T03:58:15+05:30",

"noTrafficTravelTimeInSeconds": 970,

"historicTrafficTravelTimeInSeconds": 970,

"liveTrafficIncidentsTravelTimeInSeconds": 871

},

# Evaluation

Throughout the course, we learned about the strengths of all the databases i.e., MongoDB, Neo4j and Redis database. The project helped us in understanding how to use which database in what situation and gave us a small glimpse of how these databases are used in solving real-world problems. Through this project, we aimed to create a food delivery application with several use cases that would make the best use of these databases and combine them with real-time data and simulate the solutions to those use cases.

As we look back, we learnt various path traversal algorithms like Dijkstra's algorithm and Yen's k-shortest path algorithm. We learnt how the map data is loaded into the database and how it would be useful when used together with these algorithms. We learnt about how and what data to store in Redis cache. We learnt why Neo4j is said to be beneficial in terms of recommendation engines through our Boosting Restaurant Exposure use case.

The lack of appropriate and adequate data was a reason we could not reach the extent of learning more about these databases. Hence, a possible extension to this database would be to solve even more complex issues connected to the real world along with the inclusion of modern technologies that have emerged like drones, robots, IoT etc.

# References

## Books & Journals

## Webpages

* [Leaflet Routing Machine](https://www.youtube.com/watch?v=6mAdRdwZihc&t=188s)
* [Documentation for Leaflet](https://github.com/iamtekson/Leaflet-Basic/tree/master/Basic)
* [Redis Caching Database](https://www.youtube.com/watch?v=oaJq1mQ3dFI&t=772s)
* [MongoDB Atlas Setup](https://www.youtube.com/watch?v=bBA9rUdqmgY&t=524s)
* [Initial Base for Front End Screens](https://github.com/mahmud8bd/Food_Delivery_System_SpringBoot_Angular/tree/main)
* [TomTom's Traffic Route API](https://developer.tomtom.com/routing-api/documentation/product-information/introduction)
* [Weather API](https://www.weatherapi.com/docs/)
* [OSMnx Python Library Documentation](https://osmnx.readthedocs.io/en/stable/)
* [Open Addresses](https://batch.openaddresses.io/data#map=0/0/0)
* [Neo4j Workshop to build Routing Web Application](https://neo4j.com/developer-blog/routing-web-app-neo4j-openstreetmap-leafletjs/)
* [Cypher Query Language Documentation](https://neo4j.com/docs/cypher-manual/current/introduction/)