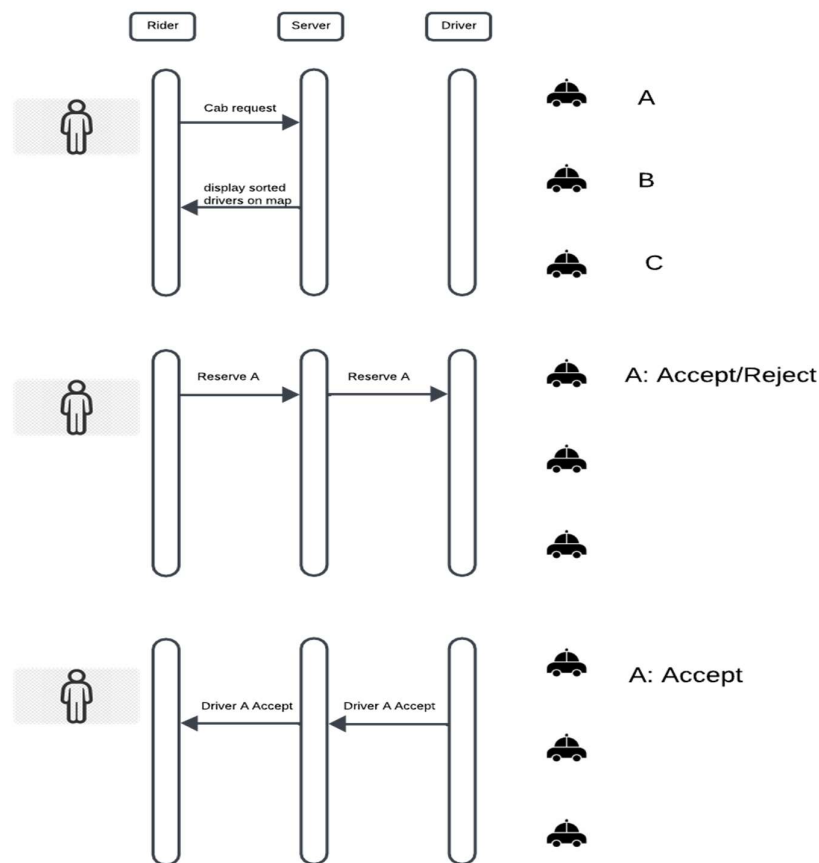


MoveInSync Case Study FTE

Objectives:

1. An employee in need of a cab ride should be able to query on nearest available cabs.
2. The cab drivers can publish the latitude and longitude so that their proximity can be calculated.
3. Once the closest cab driver is found, the employee sends a request to be picked up. cab driver in return can accept or reject the request.

The basic working of the app can be illustrated in the following diagram.



The requests involved in the design are:-

Employee: **GET** : An employee queries for all the available drivers near him/her. The result of the query is the list of all drivers with the latitude and longitude (sorted). A visual display will be shown.

Driver : **PUT** : Driver updates his/her latitude and longitude.

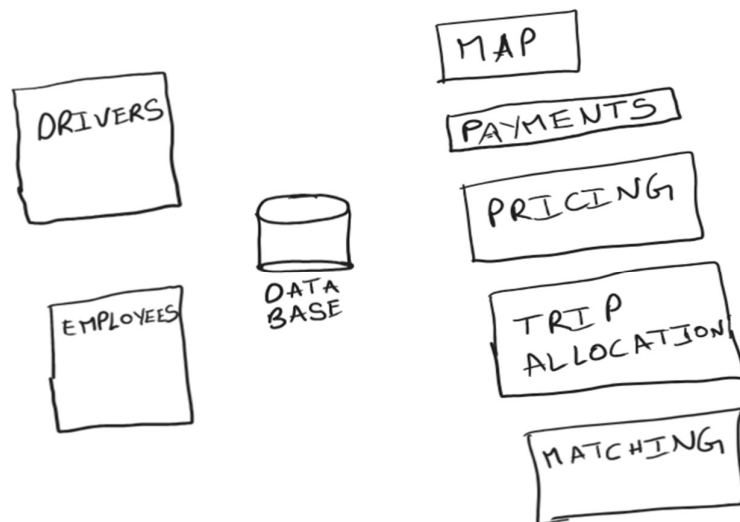
Employee : **POST** : request to accept/reject.

Driver : **POST** : accept/reject a request.

High level overview:

The application should consist of the following services to cater the needs of the employees

- 1) Drivers
- 2) Employees
- 3) Database
- 4) Map
- 5) Payments
- 6) Pricing
- 7) Trip allocation
- 8) Employee – Driver Matching



Event Bus:

Instead of services directly sending requests to one another, An event bus such as **kafka** can be utilized to streamline the data flow in the design. An event bus facilitates loose coupling between services.

Fault tolerance: In scenarios where a service might be temporarily unavailable, events can be queued in the event bus. Once the service is back online, it can process the queued events, ensuring that no data is lost during temporary outages.

Load Balancer:

By combining a load balancer for efficient request distribution and websockets for real-time data updates, the system can effectively handle a large number of users and deliver timely information to both the driver and employee applications.

Database:

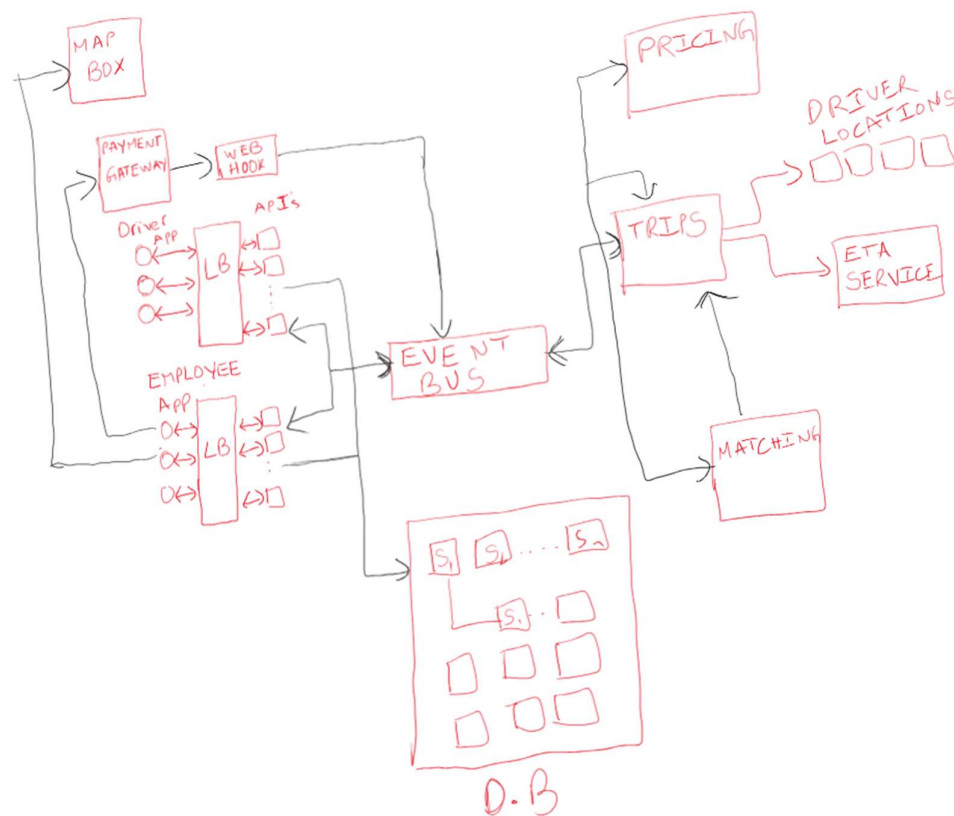
In order to accommodate the high traffic generated by constant location updates from drivers, horizontal scaling of the database is imperative. To achieve this scalability, we employ sharding as the chosen strategy. Sharding involves segmenting the database based

on key attributes such as driver id, employee id, and trip id. Additionally, the database is strategically organized, sorting data according to the status of the entries.

This approach offers several advantages:

Efficient Load Distribution: Sharding based on driver id, employee id, and trip id ensures that data is distributed across multiple shards, preventing any single shard from becoming a bottleneck and enabling a more even distribution of the workload.

Sequential Data Access: The sorting of the database according to status allows for sequential access to data. This proves beneficial in scenarios where processing data in a sequential order, such as updating the status of trips, is essential.



Maps

The integration of Mapbox API into our application empowers us to offer dynamic and customizable maps, geocoding services, and precise directions. By leveraging the Mapbox Geocoding API, user-provided addresses or location names are seamlessly converted into corresponding geographic coordinates, facilitating accurate mapping.

Requests:

1 Create / Update request

Insert a new record or update the latitude and longitude of a particular cab

HTTP Method	URL	Returns	Normal Response
PUT	/cabs/(cab_id)	No body	200 OK

Parameters

Required	Name	Type	Default	Description	Example
Yes	latitude	Float		GPS coordinate	37.799476
Yes	longitude	Float		GPS coordinate	-122.511635

2 Get request

Get the full details of the cab

HTTP Method	URL	Returns	Normal Response
GET	/cabs/(cab_id)	Cab	200 OK

Parameters

This action does not have any parameters.

3 Query request

Search for nearest cabs. The returned data does not have to be sorted and should be a list of cab records. Distance should be calculated based on the [Haversine for mula](#).

HTTP Method	URL	Returns	Normal Response
GET	/cabs	Cab	200 OK

Query Parameters

Required	Name	Type	Description	Example
Yes	latitude	Float	Client GPS coordinate	37.799476
Yes	longitude	Float	Client GPS coordinate	-122.511635
	limit	Integer	The total number of cabs to limit the response to	25
	radius	Float	The maximum distance (in meters) from the client location for a cab	1000

Sample Request

GET /cabs?latitude=37.763658&longitude=-122.427521=&radius=1000&limit=15

Sample Response(JSON)

```
[
  {
    "id": 23706134,
    "latitude": 37.788654783559,
    "longitude": -122.50747748978
  },
  {
    "id": 61344818,
    "latitude": 37.778952285851,
    "longitude": -122.43865835511
  },
  {
    "id": 19485186,
    "latitude": 37.778665475753,
    "longitude": -122.39094602609
  },
  ...
]
```

4 *Destroy request*

Destroy a cab

HTTP Method	URL	Returns	Normal Response
DELETE	/cabs/(cab_id)	No body	200 OK

Parameters

This action does not have any parameters.

5 *Destroy all request*

Destroy all cab records

HTTP Method	URL	Returns	Normal Response
DELETE	/cabs	No body	200 OK

Parameters

This action does not have any parameters.

Authentication:

Passport js stands as a dedicated authentication middleware for **Node js**, specifically designed to fulfill the singular purpose of authenticating requests. It underscores a crucial security practice of not storing user passwords as plain strings in the database. Instead, it is highly recommended to hash passwords before storage. A noteworthy convenience is offered by passport-local-mongoose, as it eliminates the need for manual password hashing using the crypto module. By incorporating **passport-local-mongoose**, this module autonomously generates salt and hash fields in the database, thereby simplifying the authentication process. Consequently, you no longer need a specific field for the password; instead, the database stores the auto-generated salt and hash, enhancing security practices within the authentication system.

```
npm install passport passport-local mongoose passport-local-mongoose --save
```

Create user schema:-

```
const mongoose = require('mongoose');
const Schema = mongoose.Schema;
const passportLocalMongoose = require('passport-local-mongoose');
```

```
const UserSchema = new Schema({
  email: {type: String, required:true, unique:true},
  username : {type: String, unique: true, required:true},
});

// passport-local-mongoose plugin
UserSchema.plugin(passportLocalMongoose);

// exporting userschema
module.exports = mongoose.model("User", UserSchema);
```

In app.js file:-

```
const User = require('./models/user');
app.use(passport.initialize());
app.use(passport.session());
passport.serializeUser(User.serializeUser());
passport.deserializeUser(User.deserializeUser());

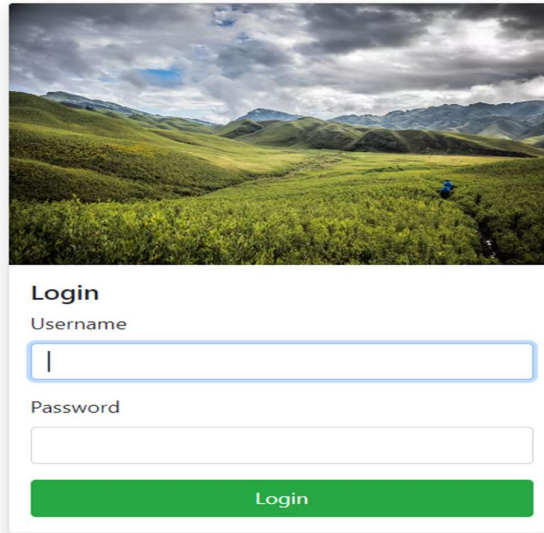
const LocalStrategy = require('passport-local').Strategy;
passport.use(new LocalStrategy(User.authenticate()));
```

The corresponding MongoDB data will be

```
{
  "_id" : ObjectId("7aa8b65835487f4c1e93c4k5"),
  "username" : "user_name",
  "email" : "mail@gmail.com",
  "salt" :
    "4c3b7dd2dbc6d4aaace01851c9687c8f6e009c4523b23553d9479e25254b266c",
  "hash" : "2257df4369e9d5ac7e2fc93b5b014260c6f8fbb01034b3f85ec
f11e086e9bf860f959d0e2104a1f825d286c99d3fc6f62505d1fde8601345c699ea08dcc071e55
47835c
16957d3830998a10762ebd143dc557d6a96e4b88312e1e4c51622fef3939656c992508e47ddc14
8696df
3152af76286d636d4814a0dc608f72cd507c617feb77cbba36c5b017492df5f28a7a3f3b7881ca
f6fb4a
9d6231eca6edbeec4eb1436f1e45c27b9c2bfcceccf3a9b42840f40c65fe499091ba6ebeb764b5d
815a43
d73a888fdb58f821fbe5f7d92e20ff8d7c98e8164b4f10d5528fddbccc7737fd21b12d571355cc6
05eb36
21f5f266f8e38683deb05a6de43114337de0df58406b25499b18f54229b9dd595b70e998fd02c7
866d06d2a6b25870d23650cdda829974a46e3166e535f1aeb6bc7ef182565009b1dcf57a64205b
5548f6974b77c2e3a3c6aec5360d55f9fcd3ffdf6fb99dce21aab021aced72881d3720f6a0975bf
ece4922282bb748e0412955e0afa2fb8c9
f5055cac0fb01a4a2288af2ce2a6563ed9b47852727749c7fe031b6b7fbb726196dbdfeeb6766d
5cba6a
```

```
055f66eeacce685daef8b6c1aed0108df511c92d49150efb6473ee71c5149dd06bfb4f73cb60f9
815af0
1e02fde8d8ed822bb3a55f040237cf80de0b1534de6bbafcb53f882c6eb03de4b4aa307828974e
b51261
661efb5155e68ad0e593c0f5fab7d690c",
  "__v" : 0
}
```

The salt and hash key are automatically generated by **passport-local-mongoose** module.



OOPs implementation:

OOPs can be implemented using programming languages like C++ and JavaScript

C++ code can be implanted in node environment using addons or node -ffi module.

JavaScript can be implemented in node environment using EJS(Embedded JavaScript).

Admin's Cab Allocation Optimization:

The cab allocation for a particular employee can be made easier using geospatial indexing by using open source models such as **H3 api** .

The H3 algorithm employs a hexagonal grid to partition the Earth's surface, allowing users to choose the level of detail by selecting from sixteen available zoom levels. Each zoom level corresponds to a different granularity, similar to zooming in on a map. As you progress to higher zoom levels, the hexagons become smaller, requiring more of them to cover the same geographical area. The efficiency of the H3 algorithm is notable, with most of its operations already achieving constant time complexity (**O(1)**) or linear time complexity (**O(n)**) for operations involving arrays of indexes. This design ensures that the algorithm maintains its effectiveness and performance across various levels of detail and geographical scales.

Here is a C++ code snippet to find h3 index from latitude and longitude.

```
#include <h3/h3api.h>
#include <iostream>

int main() {
    // Define the location coordinates (latitude and longitude)
    LatLng location;
    location.lat = degsToRads(40.689167);
    location.lng = degsToRads(-74.044444);

    // Define the resolution level for the H3 index
    int resolution = 10;

    // Get the H3 index for the specified location and resolution
    H3Index indexed;
    if (latLngToCell(&location, resolution, &indexed) != E_SUCCESS) {
        std::cout << "Failed" << std::endl;
        return 1;
    }
    std::cout << "The index is: " << std::hex << indexed << std::dec <<
std::endl;

    // Get the vertices of the H3 index
    CellBoundary boundary;
    if (cellToBoundary(indexed, &boundary) != E_SUCCESS) {
        std::cout << "Failed" << std::endl;
        return 1;
    }

    // Print the vertices of the H3 index
    for (int v = 0; v < boundary.numVerts; v++) {
        std::cout << "Boundary vertex #" << v << ": "
            << radsToDegs(boundary.verts[v].lat) << ", "
            << radsToDegs(boundary.verts[v].lng) << std::endl;
    }

    // Get the center coordinates of the H3 index
    LatLng center;
    if (cellToLatLng(indexed, &center) != E_SUCCESS) {
        std::cout << "Failed" << std::endl;
        return 1;
    }
    std::cout << "Center coordinates: " << radsToDegs(center.lat) << ", "
        << radsToDegs(center.lng) << std::endl;

    return 0;
}
```

Hexagons in close proximity possess a valuable characteristic, effectively approximating circular shapes within the framework of the grid system. The below code is a C++ implementation to find neighbor grids within **K = 5 units** distance from H3 indexed origin.

```
#include <h3/h3api.h>
#include <iostream>
#include <cstdint>
using namespace std;
int main() {
    H3Index indexed = 0x8a2a1072b59ffffL;
    // Distance away from the origin to find:
    int k = 5;

    int64_t maxNeighboring;
    maxGridDiskSize(k, &maxNeighboring);
    H3Index *neighboring = new H3Index[maxNeighboring];
    gridDisk(indexed, k, neighboring);

    cout << "Neighbors:" << endl;
    for (int i = 0; i < maxNeighboring; i++) {
        // Some indexes may be 0 to indicate fewer than the maximum
        // number of indexes.
        if (neighboring[i] != 0) {
            cout << hex << neighboring[i] << dec << endl;
        }
    }

    delete[] neighboring;

    return 0;
}
```

Employee's Cab Search Optimization:

The employee's cab search optimization can be performed using various open source ETA api's such as Distance Matrix API by google. Alternatively we can use Dijkstra's algorithm to find shortest path and minimum time between employee and driver. The real time location of driver can be updated to the api by using websocket.

The following is the pseudo code of Dijkstra's algorithm:

```
function Dijkstra(Graph, source):
    // Initialize distances to all nodes as infinity, except for the source
    // node.
    distances = map infinity to all nodes
    distances = 0
```

```

    // Initialize an empty set of visited nodes and a priority queue to keep
    track of the nodes to visit.
    visited = empty set
    queue = new PriorityQueue()
    queue.enqueue(source, 0)

    // Loop until all nodes have been visited.
    while queue is not empty:
        // Dequeue the node with the smallest distance from the priority queue.
        current = queue.dequeue()

        // If the node has already been visited, skip it.
        if current in visited:
            continue

        // Mark the node as visited.
        visited.add(current)

        // Check all neighboring nodes to see if their distances need to be
        updated.
        for neighbor in Graph.neighbors(current):
            // Calculate the tentative distance to the neighbor through the
            current node.
            tentative_distance = distances[current] + Graph.distance(current,
            neighbor)

            // If the tentative distance is smaller than the current distance
            to the neighbor, update the distance.
            if tentative_distance < distances[neighbor]:
                distances[neighbor] = tentative_distance

            // Enqueue the neighbor with its new distance to be considered
            for visitation in the future.
            queue.enqueue(neighbor, distances[neighbor])

    // Return the calculated distances from the source to all other nodes in
    the graph.
    return distances

```

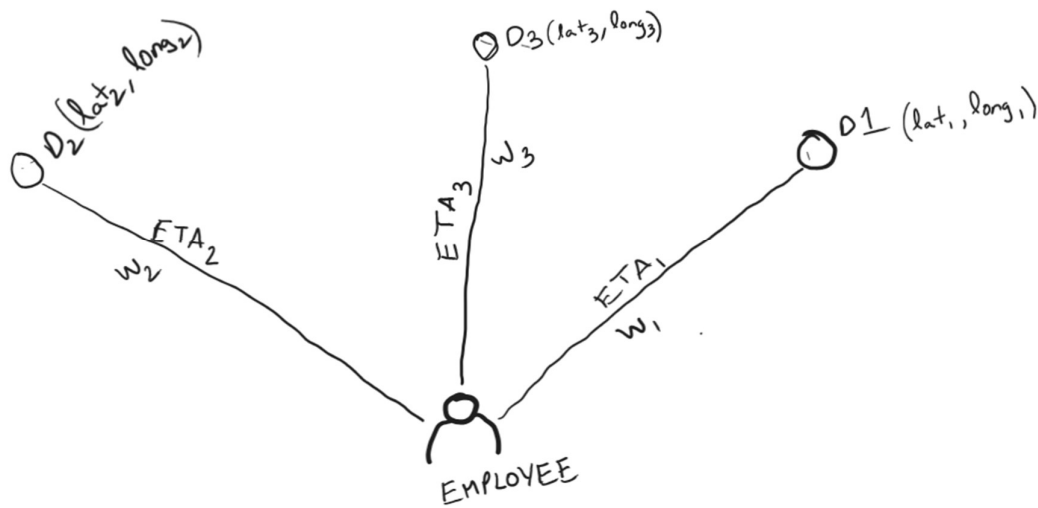
Time complexity: $O((V + E) \log V)$.

Space complexity: $O(V)$.

where V is the number of vertices and E is the number of edges.

Dijkstra's algorithm can be performed on graphs and grids with weighted edges , so by building a weight function for every route between employee and driver's real time location we can implement Dijkstra's algorithm.

Let **W** represent weight between two nodes. Then **W** could depend on distance between the nodes , traffic between the nodes and road condition between the nodes.



So **W** can be formulated to be:

$$\mathbf{W} = f(\text{distance, traffic, other factors})$$

Dijkstra's algorithm can be obtained over the obtained nodes from the employee position as the source node and drivers location as the destination node.

References:

- <https://www.uber.com/en-IN/blog/h3/>
- <https://developers.google.com/maps/documentation/distance-matrix>
- <https://github.com/uber/h3>
- <https://www.geeksforgeeks.org/>