Hola

Ride hailing app

REST API

Web App

TEAM

01FB15ECS**197** **Nitish** Makam

01FB15ECS**198** **Omkar** D

01FB15ECS**202** **Parashara** R

01FB15ECS**209** **Prajwal** Bolar

01FB15ECS**222** **Rahul** M

01FB15ECS**223** **Rahul** Pujari

01FB15ECS**224** **Rahul** Pillai

01FB15ECS**226** **Rahul** Thakur

1. Introduction

Hola is a web-based ride-hailing application similar to OLA/Uber. The objectives of this project are twofold -

i. Build a REST API for the rider-side app that
    a. Provides endpoints for all stages, from booking a cab till completing and rating the trip
    b. Allows only authenticated users to access the API
ii. Build rider-side Web application that
    a. Provides a simple & intuitive UI
    b. Communicates with the REST API
    c. Uses all endpoints of the API
    d. Simulates movement of cars from source to destination location

2. Requirements

The Software Requirements Specification describes all the capabilities a product must have in order to fulfill the business, stakeholder and user needs. This is useful for the following audience

I. Developers to build a workflow and divide work into logical sprints as well as gather and decide on the user interface and database requirements.
II. Stakeholders and users to refine and understand the functionality of the software.
III. Business teams to understand where the funding given will be utilized

2.1. Features

Backend

- Provide a REST API for the front end that provides endpoints for all states of a trip cycle
- Store multiple entities in the database which is essential for simulation.
- Provide token authentication for users for secure login.

Frontend

- Provide a map-based web application that visualizes the various steps in a trip cycle
- Provide a simple UI with subtle animations and bright colors to capture the attention of the user to important components on the UI
- Simulate movement of cabs on the map due to the absence of real cars

2.2. General Constraints

- The web application has to run on a widescreen browser environment as of this version
- The server backend database is relational and the application involves lots of joins thereby making it slower.

- The server running the REST API handles a limited load as it runs on a single machine.
- Due to the restriction on the number of allowed requests to the free tier of Mapbox API, the number of map loads and direction calculations is limited to 50000 per month.

2.3. Functional Requirements

1. **Login/Registration with token authentication**
   Register new users and login existing users to track their rides
2. **Select source and destination locations**
   Allow the user to set the required source and destination from an autocompletion based functionality
3. **Book a cab**
   Schedule a cab from an array of ‘available’ cars near source location
4. **Fetch details of car and driver**
   Get information about the car and its driver for identification by the user
5. **Rating and giving feedback for drivers**
   Rate the trip and give general feedback for the driver when the trip gets completed
6. **Cancel trip**
   Cancel a scheduled ride
7. **Payment option for the trip on completion**
   Pay the driver using a variety of payment options when the trip gets completed

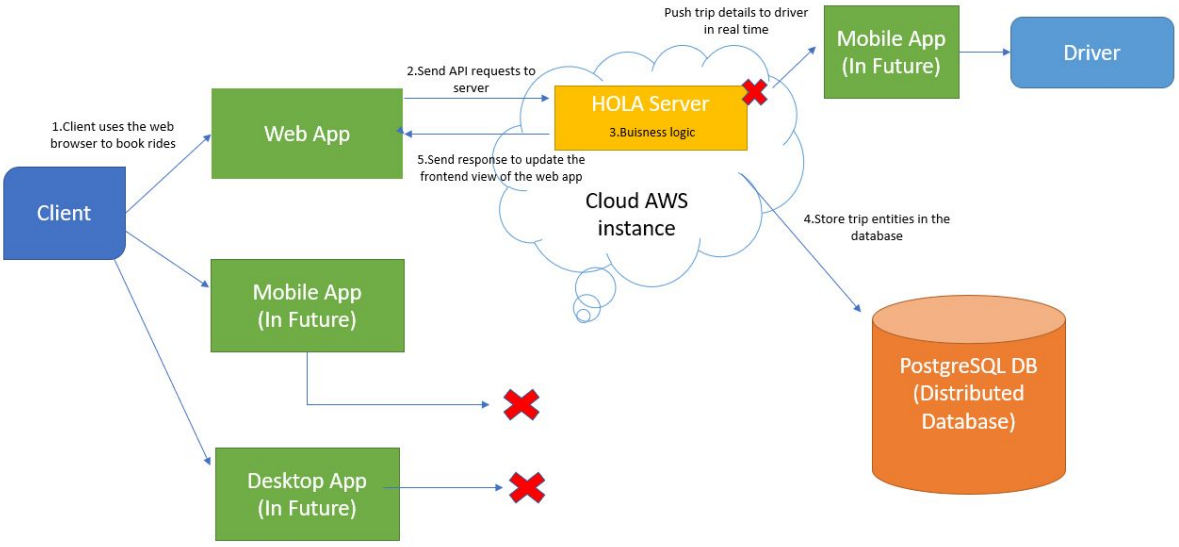
2.4. Non-functional Requirements

1. **Availability**
   The servers of both the REST API and the web app are required to have close to 100 percent availability from early morning till late at night.
2. **Efficiency and performance**
   Results of API calls should be returned within 1 second on a 4G or any equivalent bandwidth connection.
3. **Maintainability**
   Maintenance and updates to the system shall happen after late night when active users are negligible in number. Issue notices about the maintenance operation days in advance.
4. **Portability**
   The web app shall be compatible with all recent versions of web browsers in major operating systems
5. **Responsiveness**
   All UI components respond to user interaction instantly.
6. **Usability**
   Requirements for the web app state that users become familiar with the app’s functions by the end of the first trip.

3. High Level Design

We use layered architecture to split up architecture components into layers based on their functionality (UI, API, DB). We also use Client-Server architecture for communication between the web app and the REST API server.

3.1. Architecture Diagram



Architecture Diagram

3.2. Description

1. **Web Application (Front End)**
   When a user logs in or registers, the server checks the credentials of the user and returns a unique token (which is used to authenticate the session and is used in the header of all subsequent requests). The app also makes HTTP requests for data on behalf of the user to the server by sending data in JSON format.
2. **HTTP requests**
   These are essentially the interaction between the app and the server and are performed by the app through API routes set up on the server side. All requests except `logout/` send JSON data. In requests other than login/register, the token (which is stored in local storage after login/registration) is sent in the header to serve as a security measure that the web app is requesting only for data that is allowed to be accessed.
3. **Server**
   Our back-end code runs on the cloud and is written in Python using the Django framework. It performs the operations of controlling the API access, managing requests and also connecting to the database.

4. **PostgreSQL database**
   The database also runs on the cloud and when queries are sent from the server side, the response is sent from the database which is then communicated to the front end.
5. **Cloud**
   Both the server code and PostgreSQL instance are running on an AWS EC2 instance.

3.3. Languages and Frameworks

Frontend

- HTML, CSS, JS
- Mapbox GL JS
- Jquery
- Browserify

Backend

- Python
- Django, Django Rest Framework
- PostgreSQL with PostGIS
- Nginx server

4. Low Level Design

4.1. API Endpoints

Following are the supported REST API endpoint URIs and their corresponding HTTP methods, request and response formats in the server.

1. **Car Details**
URL: POST /hola/v1/cardetails
Input: Car ID

```
{
   "carId": 1
}
```

Output: Car & Car Driver details

```
{
   "car": {
   "carId": 1,
       "carType": 2,
       "carModel": "PUSHPAKA VIMANA HYBRID",
       "carLicense": "KA 41 M 1212"
   },
   "carDriver": {
   "driverId": 1,
       "name": "RAVANA",
       "phone": "9876543210"
   }
}
```

2. **Cars in Location**

URL: POST /hola/v1/carsinlocation

Input: User's current location

```
{
   "geoLocation": {
   "latitude": 12.918486,
       "longitude": 77.546386
   }
}
```

Output: Array of Car Statuses

```
{
   "carStatuses": [
   {
       "carStatus": {
           "carId": 1,
           "geoLocation": {
               "latitude": 12.913486,
               "longitude": 77.543386
           },
           "carAvailability": "CAR_AVAILABLE"
       }
   },
   {
       "carStatus": {
           "carId": 2,
           "geoLocation": {
               "latitude": 12.913486,
               "longitude": 77.542386
           },
           "carAvailability": "CAR_AVAILABLE"
       }
   }]
}
```

3. **Car Status**

URL: POST /hola/v1/carstatus

Input: car ID

```
{
   "carId": 1
}
```

Output: Current location & availability of the car

```
{
   "carStatus": {
       "carId": 1,
       "geoLocation": {
       "latitude": 77.998799,
           "longitude": 56.079089
       },
       "carAvailability": "CAR_AVAILABLE"
   }
}
```

4. **Complete Trip**

URL: PUT /hola/v1/complete_trip

Input: Trip ID, finishing location & a confirmation of the payment mode


```
{
   "tripId": 1,
   "finishLocation": {
   "latitude": 12.908486,
       "longitude": 77.536386
   },
   "paymentMode": 3
}
```


Output: Entire trip info as a confirmation

Note that endTimeInEpochs is now populated & tripStatus is updated to 4 (TRIP_STATUS_COMPLETED).

If the client doesn't receive this back or receives an error in completeTripStatus, it will retry the request.


```
{
   "completeTripStatus": 2,
   "trip": {
   "tripId": 1,
       "carId": 1,
       "driverId": 1,
       "sourceLocation": {
       "latitude": 12.908486,
           "longitude": 77.536386
   },
   "destinationLocation": {
       "latitude": 12.908486,
           "longitude": 77.536386
   },
   "startTimeInEpochs": 1512152782,
       "endTimeInEpochs": 1512170782,
       "tripPrice": 129.00,
       "tripStatus": 4,
       "paymentMode": 2
   }
}
```


5. **Fare Estimates**

URL:POST /hola/v1/fareestimates

Input: Source location & destination location


```
{
   "sourceLocation": {
   "latitude": 12.908486,
       "longitude": 77.536386
   },
   "destinationLocation": {
   "latitude": 12.908486,
       "longitude": 77.536386
   }
}
```

Output: Array of EstimateForCarType


```
{
   "estimatesForCarTypes": [
   {
       "estimateForCarType": {
           "carType": 1,
           "tripPrice": 129.00,
           "discountAmount": 0.00
       }
   },
   {
       "estimateForCarType": {
           "carType": 2,
           "tripPrice": 140.00,
           "discountAmount": 0.00
       }
   },
   {
       "estimateForCarType": {
           "carType": 3,
           "tripPrice": 181.00,
           "discountAmount": 0.00
       }
   }
]}
```


6. **Schedule Trip**

URL:POST /hola/v1/scheduletrip

Input: carType, tripPrice, start/end locations


```
{
   "carType": 1,
   "tripPrice": 129.00,
   "sourceLocation": {
   "latitude": 12.908486,
       "longitude": 77.536386
   },
   "destinationLocation": {
   "latitude": 12.908486,
       "longitude": 77.536386
   }
}
```


Output: Whether we were able to find cars & if so the trip details


```
{
   "scheduleTripStatus": 2,
   "trip": {
   "tripId": 1,
       "carId": 1,
       "driverId": 1,
       "sourceLocation": {
```

```
        "latitude": 12.908486,
            "longitude": 77.536386
    },
    "destinationLocation": {
        "latitude": 12.908486,
            "longitude": 77.536386
    },
    "startTimeInEpochs": 1512152782,
        "tripPrice": 129.00,
        "tripStatus": 2
    }
}
```


7. **Rate Trip**

URL:POST /hola/v1/ratetrip

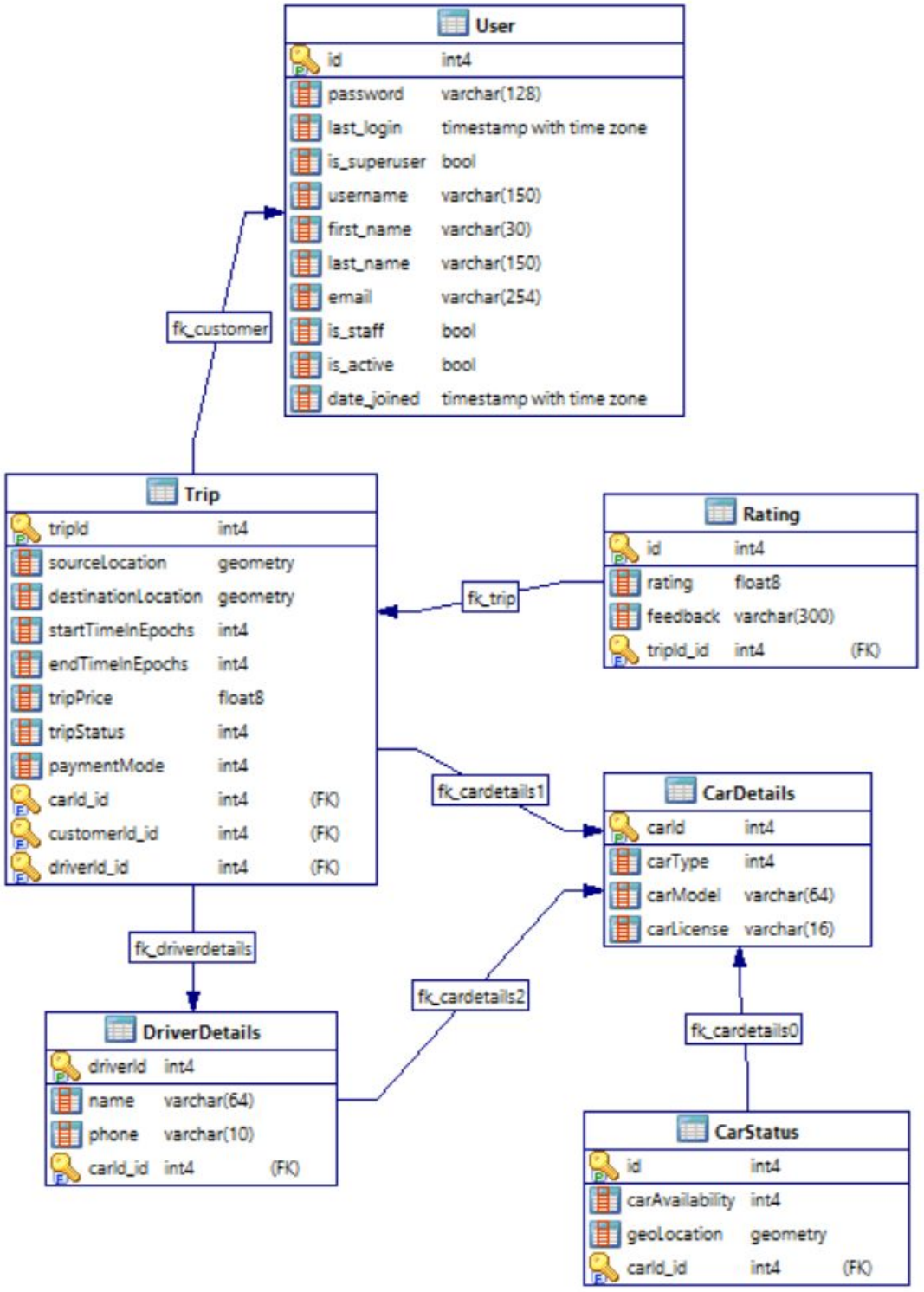
Input :


```
{
    "tripId": 6,
    "rating": 4,
    "feedback": "excellent driver! Comfortable ride!!"
}
```


Output:

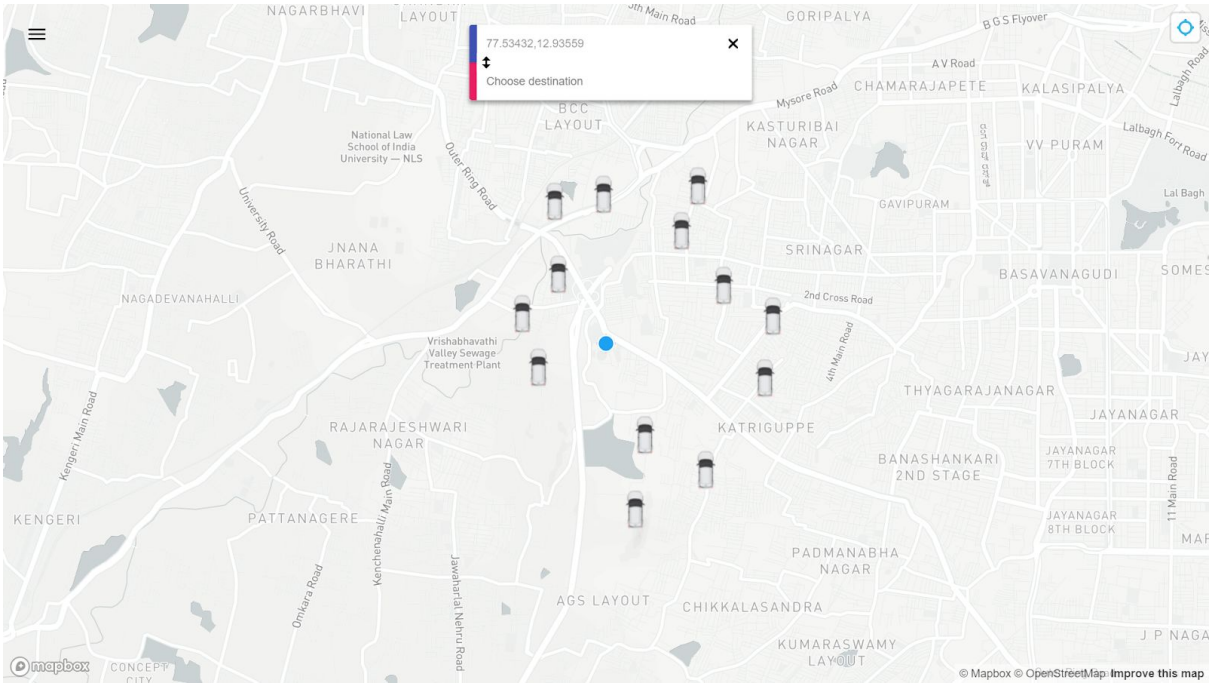

```
{
    "status": "successful saved the rating and feedback in the database"
}
```

4.2. Database Design

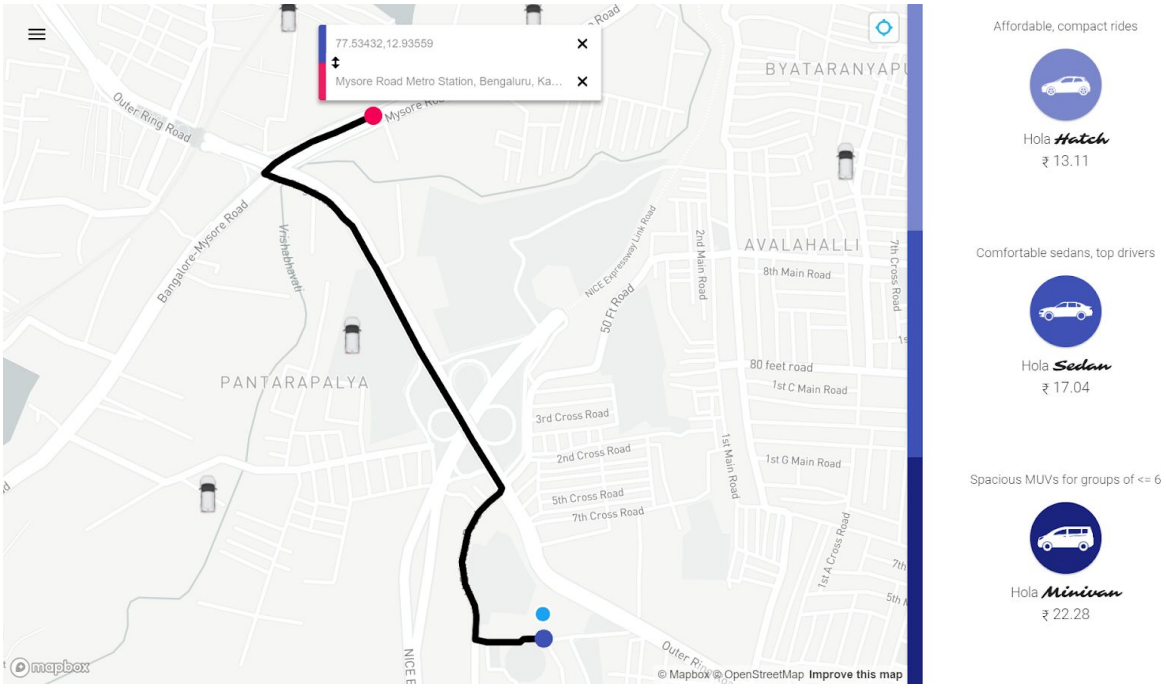


Database Tables

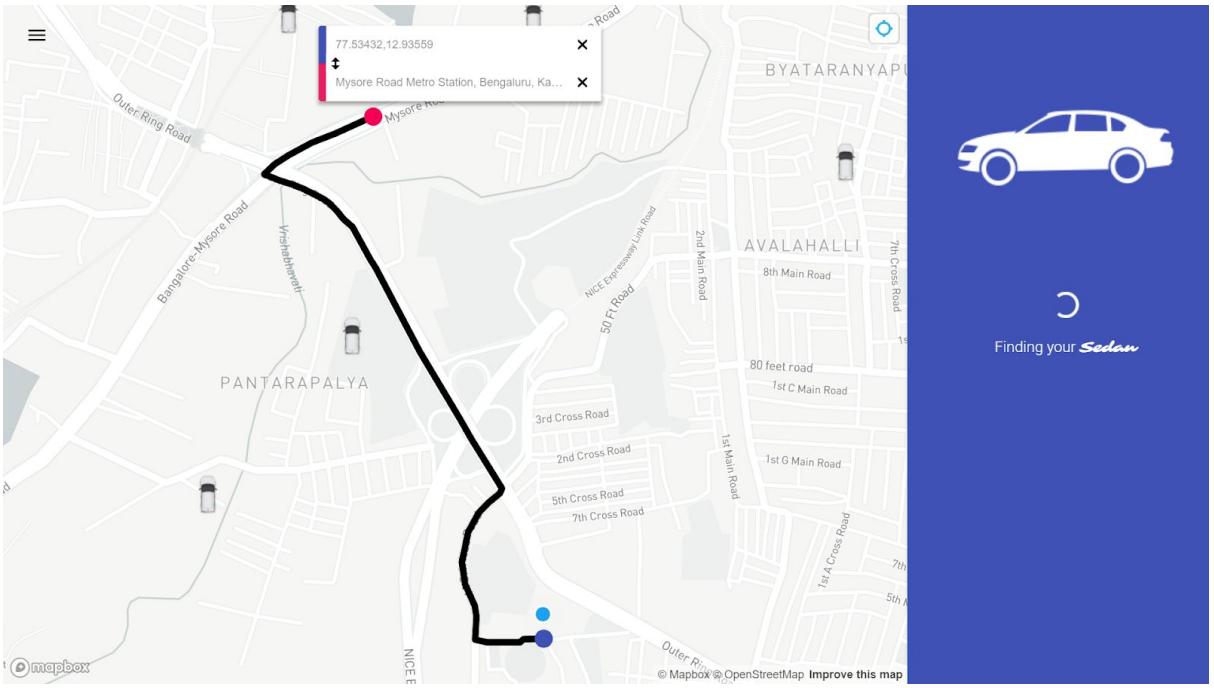
5 Screenshots



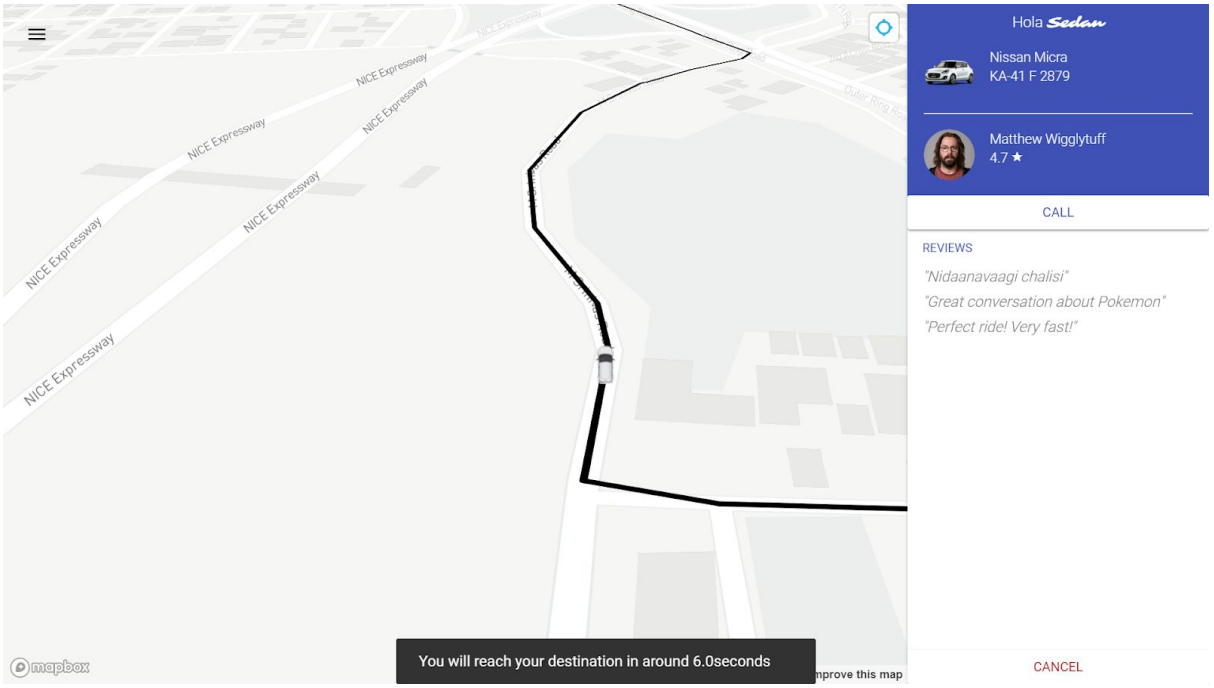
View cars around the user's current location



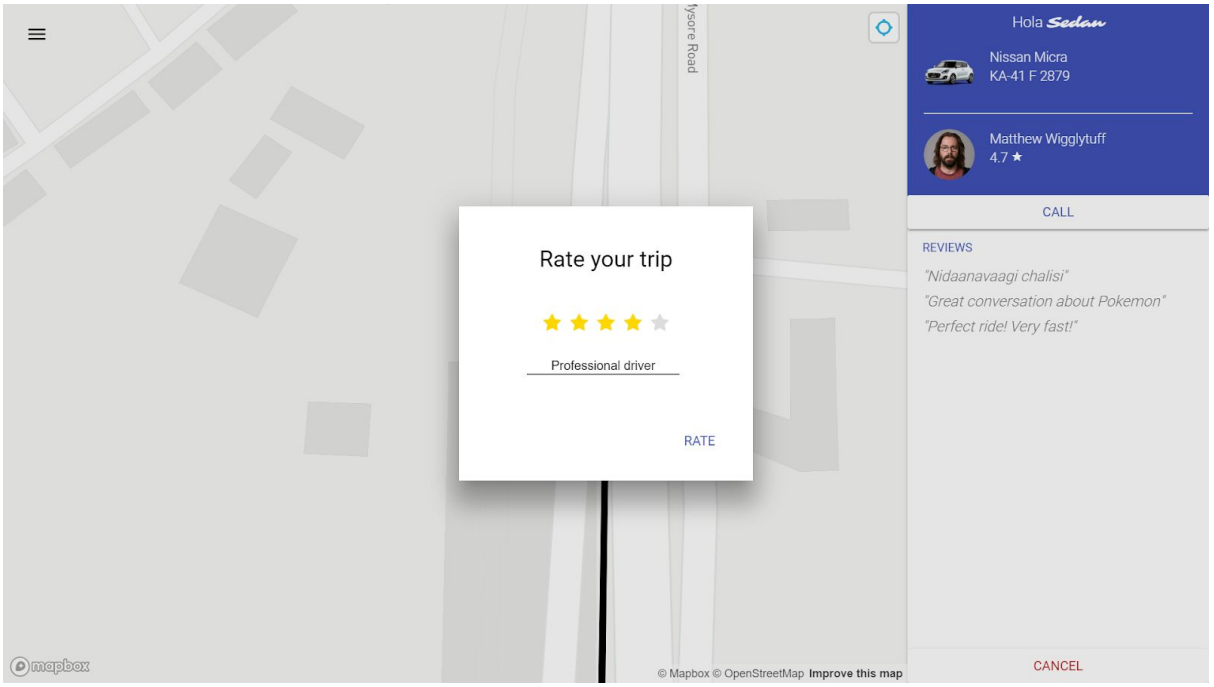
View route and fare estimates on different types of rides



An indication of scheduling a ride



Simulation of cab movement along a route and display of car and driver details



Rating trip after completion