NUS

National University of Singapore

CS2102 Project Report

Carpooling App - RideShare

By:

Karnati Sai Abhishek (A0184397M)

Marc Phua Hsiao Meng (A0183219A)

Suther David Samuel (A0182488N)

Priyan Rajamohan (A0187872L)

Table of Contents

1. Introduction

Rideshare is a carpooling app that enables people to register as a user and be either a passenger or a driver. Drivers can advertise their trips and passengers can bid for them. Drivers have the luxury of solely accepting whichever bids they want without automated matching to allow for a fast and seamless process.

1.1 Aim

The main aim of RideShare is to allow drivers and passengers to have a quick and efficient way of clocking a high mileage of rides and getting fast and frequent matches respectively. This is done through a seamless integration of an easy to use user interface such as a login page where people can sign up for an account. In addition, users will be able to select if they want to be a passenger or a driver. Users will be redirected to a passenger or driver page depending on their choice. Drivers must select a vehicle to advertise their rides with a start location and an end location with a stipulated arrival time. Passengers will be able to key in their start location and see relevant advertisements before bidding for a ride. Drivers will be able to see passengers bid and will be given a choice to either accept or reject them.

1.2 Design

The design is based upon the 3-tier architecture: frontend web-page, database server and the database design. The app was designed solely on the web. For our Frontend, we used HTML, CSS and Javascript. For our backend server , we used SQL to store all our data and entities with the schema drafted from our database design. Our database design was done using an Entity-Relationship Diagram to capture constraints that we wanted to enforce. The framework we used to link the frontend and backend was NodeJS. The type of database server we used to store our information was using local Postgresql.

1.3 Roles and Responsibilities

We split the work evenly amongst the 4 of us and managed to get it done with equal contribution from each member. The way the roles were split are as follows:

Marc was in charge doing the ER diagram conceptualisation and constraints design. He also helped out with translating the ER diagram to the backend SQL schema and did up the triggers to enforce the constraints in our project along with generating test cases.

Abhishek was in charge of integrating the backend to the frontend and did up all the queries and complex queries to allow filtering and sorting in our design. He integrated the complex queries into the frontend for both the passenger and driver analytics. He was responsible for integrating the pooling queries for both the passenger and driver.

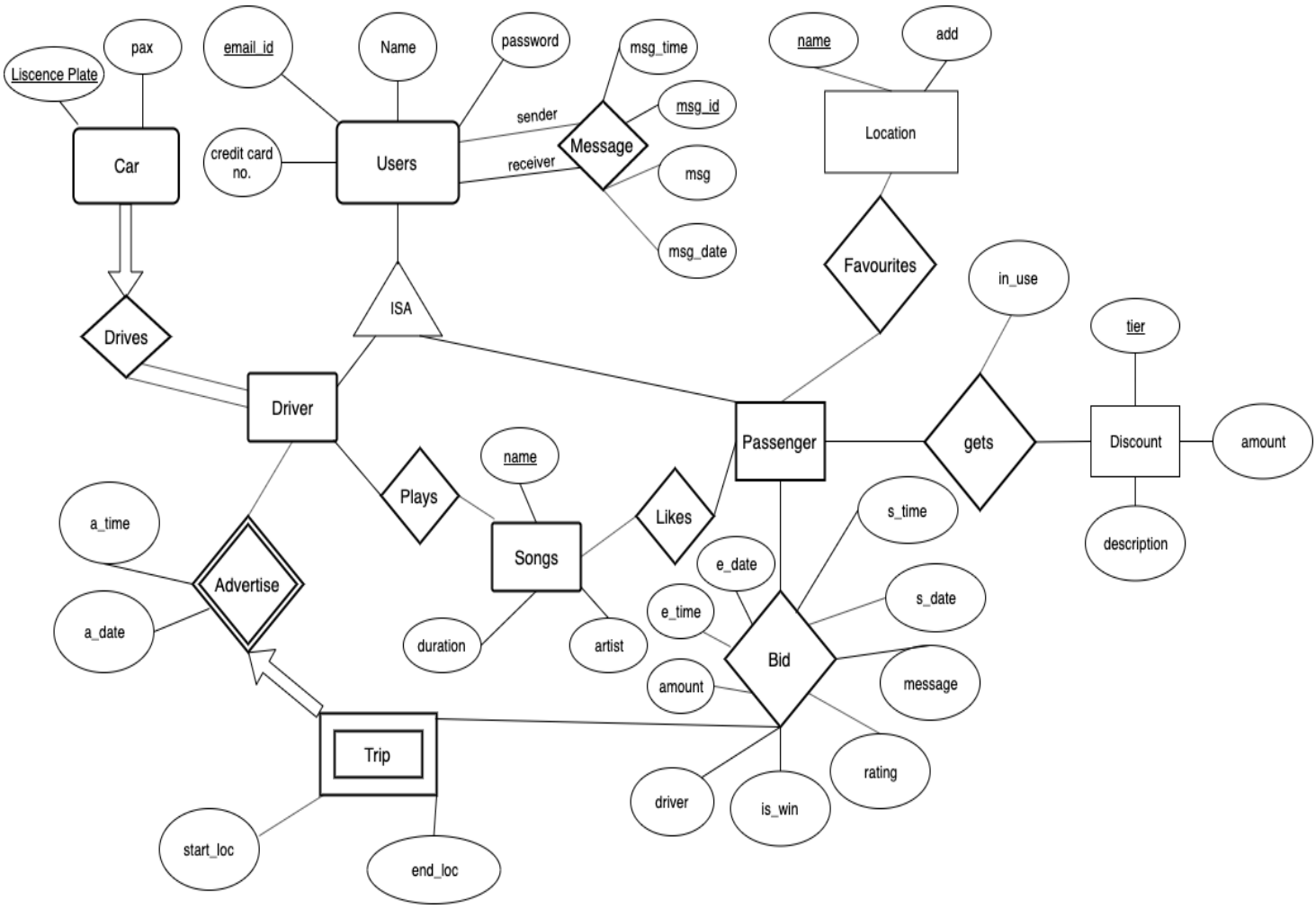
Samuel was in charge of the frontend and did up the NodeJS parts of the code. More specifically, he configured the signup, login and session of the user. He helped link the frontend to the backend for

various pages such as analytics. He also helped out with conceptualising the ER diagram and translating it into the schema.

Priyan was in charge of doing up the UI for all the pages as well as integrating the favourites and messages feature into our project. He helped link the frontend and backend for the those features as well.

2. Database Design

2.1 Entity-Relationship Diagram



2.2 Constraints

2.2.1 ER Diagram Constraints

- Each Passenger can be a driver
- Each vehicle can be driven by exactly one driver
- Each driver can drive one or more cars
- A Trip is a weak entity set
- Trip entities have a many to one relationship with a driver entity
- Each Passenger can bid for 0 or more trips
- Each user can send 0 or more messages to any other user
- Each Passenger can favourite 0 or more locations
- Each location can be favorited by 0 or more passengers
- Each passenger can obtain 0 or more discounts
- Each discount can be owned by 0 or more passengers
- Each driver can play 0 or more songs
- Each song can be liked by 0 or more passengers
- Each song can be played by 0 or more drivers

2.2.2 Non-ER Diagram Constraints

- When a user logs into a session, he can only either be a passenger or a driver. This is to ensure that he cannot simultaneously advertise and bid for rides while in the same session as it will cause bugs in our website.
- A registered car can only be owned by one registered driver. This is to prevent discrepancies between passengers and drivers.
- A driver can only advertise a trip once at any particular start date, start time and start location. This is to prevent multiple advertisements of the same type showing up on the passengers dashboard.
- A passenger cannot accept a bid from his own driver account. This is to prevent a user from exploiting the systems flaws and getting a higher overall rating.
- A passengers bid for a certain location, at a certain date and time can only be accepted by one driver. This is to prevent multiple drivers from picking up the same passenger at the same instant.
- A user cannot message himself from both his passenger and driver account. This to prevent overload of the system and ensure only meaningful information is disseminated between two parties.
- Driver cannot start his trip without advertising his trip. A driver has to wait for an accepted bid before a proper trip can happen to prevent exploitation of the system where the driver can post high earnings and high ratings through starting trips on his own.
- Driver cannot start his trip without accepting any bids. The driver needs a passenger first before he can start his trip.
- Driver cannot start a trip without advertising his vehicle. This is to ensure that the driver has a registered vehicle that is capable of carrying a passenger and completing the trip.

- Driver cannot advertise any trips with locations not in the locations relation. This is because our application only allows trips between locations in the locations relation.
- Driver cannot accept more bids than the current capacity of his car. The driver cannot carry more people than required during any point in the journey.
- Passenger cannot search for locations that are not inside the locations relation. Our app does not support travelling outside of the desired spots.
- Passenger cannot unlock more than one discount tier after every upgrade. The user will not be able to unlock two consecutive tiers at one go as the tier system follows a strict order before unlocking.
- Tiers unlocked by a passenger is always increasing. The passenger gets more rewards as he spends more on his rides.
- A passenger can use a discount from a tier he unlocked only once. This is to prevent users from abusing the app by using discounts repeatedly.
- A bid value for a trip has to be less than $100. This is to prevent users from exploiting the system and bidding very high amounts which can skew driver analytics.
- A passenger has a maximum of up to 5 tiers he can claim. This is to prevent unfair advantage to passengers who have more spending power that wish to spend more in order to get infinitely more discounts.

2.3 Schema

```
create table passenger(
    email varchar(256) primary key,
    name varchar(100) not null,
    password varchar(100) not null,
    credit_card_num varchar(100) not null
);
create table driver(
    email varchar(256) primary key references passenger(email)
);
create table vehicles(
    license_plate varchar(50) primary key,
    pax integer not null
    check (pax >= 1 and pax <= 6)
);
create table drives(
    email varchar(256) references driver(email) not null,
    license_plate varchar(50)references vehicles not null,
    primary key (email, license_plate)
);
create table message(
    sender_email varchar(256) references passenger(email) not null,
```

```
    receiver_email varchar(256) references passenger(email) not null,
    msg varchar(1024) not null,
    msg_time time,
    msg_date date,
    check (sender_email <> receiver_email),
    primary key (msg_time, msg_date, sender_email, receiver_email)
);
create table location(
    loc_name varchar(256) primary key,
    loc_add varchar(256) not null
);
create table favouriteLocation(
    email_passenger varchar(256) references passenger(email),
    loc_name varchar(256) references location (loc_name),
    primary key(email_passenger, loc_name)
);
create table advertisesTrip(
    start_loc varchar(256) not null references location(loc_name),
    end_loc varchar(256) not null references location(loc_name),
    email varchar(256) not null,
    vehicle varchar(50) not null,
    a_date date not null,
    a_time time not null,   --time the driver will start his trip
    foreign key(email, vehicle) references drives (email, license_plate),
    primary key(email, vehicle, start_loc, a_date, a_time)
);
create table bid(
    is_win boolean default false,
    amount float not null,
    start_loc varchar(256) not null,
    end_loc varchar(256) not null,
    email_bidder varchar(256) references passenger(email),
    email_driver varchar(256) not null,
    vehicle varchar(50) not null,
    s_date date not null,
    s_time time not null,
    e_date date,
    e_time time,
    review varchar(1024),
    rating numeric,
```

```
    CHECK (((is_win is true and ((e_time > s_time and e_date = s_date) or (e_date > s_date)))
or ((is_win is false and e_time is null and e_date is null and review is null and rating is
null))) and email_bidder <> email_driver),
    primary key(email_bidder, email_driver, start_loc, s_date, s_time)
);
create table discount(
    description varchar(256),
    tier numeric not null,
    amount float not null,
    primary key(tier)
);
create table gets (
    email varchar(256) references passenger(email),
    tier numeric references discount(tier),
    is_used boolean default false,
    primary key(email, tier)
);
create table songs (
    name varchar(256) primary key,
    duration time,
    artist varchar(256)
);
create table likes (
    email varchar(256) references passenger(email),
    name varchar(256) references songs(name),
    primary key(email, name)
);
create table plays (
    email varchar(256) references driver(email),
    name varchar(256) references songs(name),
    primary key(email, name)
);
```

2.4 3NF/BCNF Satisfiability

3NF Condition:
Every functional dependency in set (a → A) must be either:
1. Trivial
2. a is a Superkey
3. A is a prime attribute

BCNF Condition:
For every one of the entities functional dependencies; $X \rightarrow Y$, at least one of the following conditions hold

- $X \rightarrow Y$ is a trivial functional dependency (Y ⊆ X),
- X is a superkey for schema R.

The messages entity is not in 3NF/BCNF as a primary key of (sender_email, receiver_email) cannot uniquely identify every entity in the table as a user could send a message to the same receiver across a different date and time. Therefore, all attributes need to be considered in the primary key on the message entity.

The other tables are in 3NF/BCNF as they consist of attributes that depend on the primary key of the entity. The vacuously true case is considered when there are no other attributes in the table except for the primary key, in this case the table is still in 3NF/BCNF as no attribute depends on the primary key of the entity.

2.5 Complex SQL Queries

We implemented several complex queries in order to send and retrieve complex sets of information to and fro from the database. The following is an example of one such query.

2.5.1 Advertised Trips Complex Query

We used this query to retrieve and display all information regarding the trips advertised by the driver on the driver's home page. We obtained this information by querying results from several subqueries. Following are the various sets of information retrieved using this query:

1. Start location: This is the start location of the advertised trip.
2. End Location: This is the end location of the advertised trip.
3. Start date: This is the date the driver aims to start the advertised trip.
4. Start time: This is the time the driver aims to start the advertised trip.
5. Driver Email: This is the email of the driver which is required to display all the advertised trips posted by that particular driver.
6. Vehicle: This is the vehicle license plate number which the driver will use to carry out the trip.
7. Current capacity: This is the current capacity of the vehicle. It is calculated by subtracting the number of bids which the driver has accepted for the advertised trip from the total capacity of the vehicle.

All this information helps the driver make an educated guess of how many more bids he can accept and also when he should start the trip.

```
select distinct A.start_loc, A.end_loc, A.a_date, A.a_time, CP.email_driver, CP.vehicle,
CP.current_pax
from advertisestrip A,
    (select distinct T.email_driver, T.vehicle, (T.pax - O.occupancy) as current_pax
        from (select distinct D.email as email_driver, D.license_plate as vehicle, V.pax
                from drives D, vehicles V
                where D.license_plate = V.license_plate) T,
            ((select distinct email as email_driver, license_plate as vehicle, 0 as occupancy
                from drives
                where (email, license_plate)
                    not in (select distinct Q1.email_driver, Q1.vehicle
                                from
                                    (select distinct email_driver, vehicle, count(*)
                                    from bid
                                    where e_date is null
                                    group by email_driver, vehicle) Q1
                                left join
                                    (select distinct email_driver, vehicle, count(*)
                                    from bid
                                    where is_win is true
                                    and e_date is null
                                    group by email_driver, vehicle) Q2
                                on  Q1.vehicle = Q2.vehicle
                                and Q1.email_driver = Q2.email_driver
                                group by Q1.email_driver, Q1.vehicle))
            union
            (select distinct Q1.email_driver, Q1.vehicle, coalesce(sum(Q2.count),0) as
occupancy
                from
                    (select distinct email_driver, vehicle, count(*)
                    from bid
                    where e_date is null
                    group by email_driver, vehicle) Q1
                left join
                    (select distinct email_driver, vehicle, count(*)
                    from bid
                    where is_win is true
                    and e_date is null
                    group by email_driver, vehicle) Q2
                on  Q1.vehicle = Q2.vehicle
                and Q1.email_driver = Q2.email_driver
                group by Q1.email_driver, Q1.vehicle)) O
```

```
            where T.email_driver = O.email_driver and T.vehicle = O.vehicle) CP
where A.email = CP.email_driver
and A.vehicle = CP.vehicle
and A.email = $1
order by A.a_date desc, A.a_time desc;
```

2.6 Triggers

We created triggers to enforce constraints on our implementation to ensure that there were no violations that would degrade the performance of the website.

2.6.1 Trigger #1

The following trigger checks if there is a corresponding advertisement for every bid that the passenger makes. We enforced this because we do not want to insert an entry in the database where the passenger bids for a ride that does not exist. This is equivalent to enforcing a foreign key constraint on the bid entity. We enforced the constraint using a trigger because we used the advertisesTrip entity to capture all the live advertisements that have not had a winning bidder yet. Therefore, we did not want the bid entity to depend on the advertisesTrip entity, hence the enforcement using a constraint.

```
--checks if there is a corresponding advertisement for the bid
create or replace function advertise()
returns trigger as $$
declare count NUMERIC;
BEGIN
   select COUNT(*) into count from AdvertisesTrip A
   where A.start_loc = NEW.start_loc
   and A.email = NEW.email_driver
   and A.a_date = NEW.s_date
   and A.a_time = NEW.s_time;
   IF count = 1 THEN
       RETURN NEW;
   ELSE
       RETURN NULL;
   END IF;
END;
$$ language plpgsql;

create trigger check_bid_trip
before insert
on bid
```

```
for each row
execute procedure advertise();
```

2.6.2 Trigger #2

The following trigger checks if there is a driver has accepted a bid for a particular advertisement before deleting it. This is to ensure that no live advertisement gets deleted without a winning bid which would result in the driver having an unfair disadvantage.

```
--checks if a bid has been accepted by the driver before deleting the advertisement
create or replace function won()
returns trigger as $$
declare count NUMERIC;
BEGIN
  select COUNT(*) into count from bid B
  where B.start_loc = OLD.start_loc
  and B.email_driver = OLD.email
  and B.is_win
  and B.s_date = OLD.a_date
  and B.s_time = OLD.a_time;
  IF count = 1 THEN
     RETURN OLD;
  ELSE
     RETURN NULL;
  END IF;
END;
$$ language plpgsql;
```

2.6.3 Trigger #3

This trigger checks if the new discount tier obtained by the passenger is more than all his previous tiers to prevent the passenger from getting the same discount twice. We implement a strict tier system such that no user get more promotions over the other unfairly. We also want the user to get a discount that is proportional to his expenditure. Therefore, after every time he hits a certain amount of expenditure he will get a tier that his higher than his current tier. This enforces a strict hierarchy and captures the constraint of the application properly.

```
--check if the new tier obtained by the user is greater than any of them in the gets entity
create trigger check_delete_trip
before delete
```

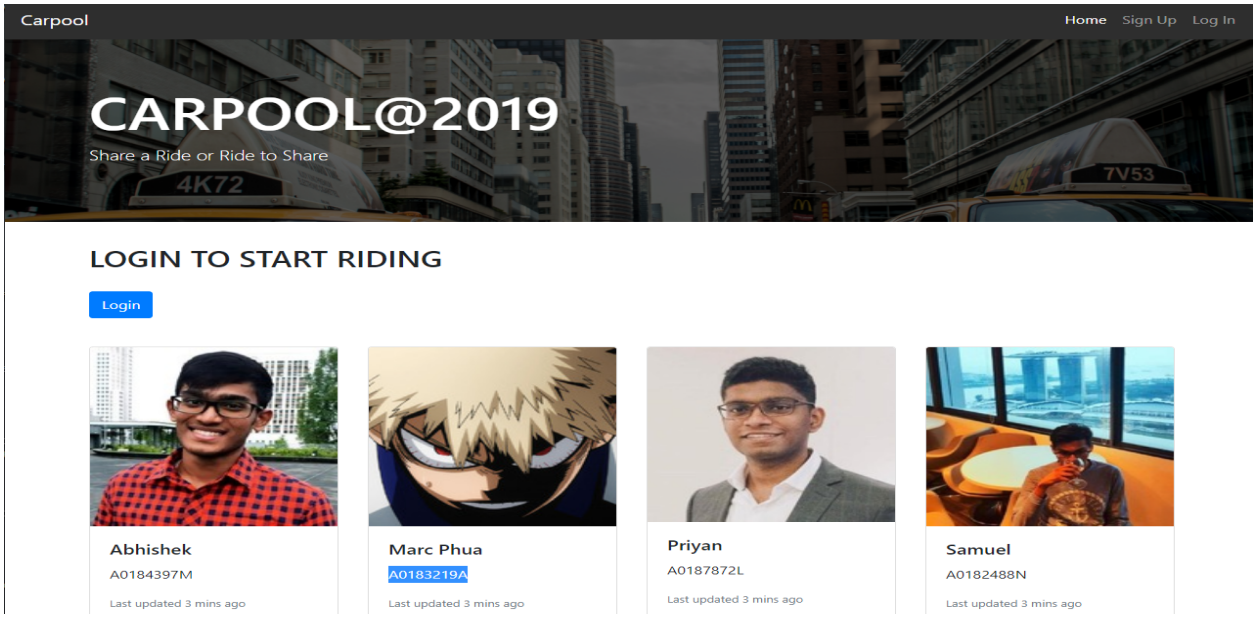
```
on AdvertisesTrip
for each row
execute procedure won();

create or replace function uses()
returns trigger as $$
BEGIN
   IF NEW.tier > ALL (SELECT G1.tier from gets G1 where G1.email = NEW.email) THEN
       return NEW;
   ELSE
       return NULL;
   END IF;
END;
$$ LANGUAGE plpgsql;

create trigger check_highest_tier
before INSERT on gets
for each row
execute procedure uses();
```

3. Web Page Design

3.1 Before Login

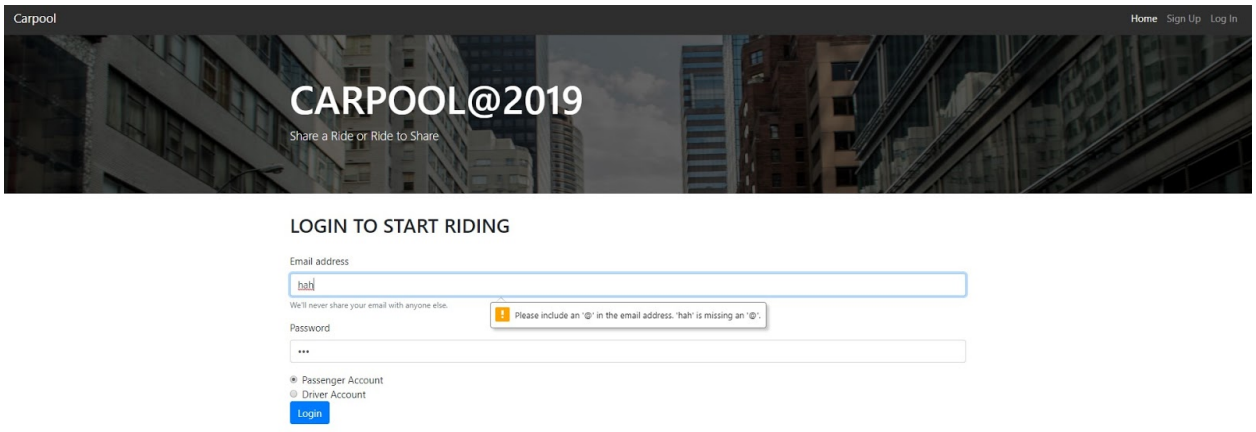
3.1.1 Landing Page

The following figure below shows the landing page of the Carpool application. It clearly indicates the purpose of the app which is to “Share a Ride or Ride to Share”. The call for action is the Login button which prompts the user to Login to use the services. The Menu bar clearly indicates that the pages accessible by a user who has just visited the site are “Home, Sign Up and Login”.



3.1.2 Login Page

As shown in the figure below, the Login button leads to the Login page. If a user does not have an existing account, login will not be successful and user will be redirected to the login page. Hence user will have to sign up for an account.



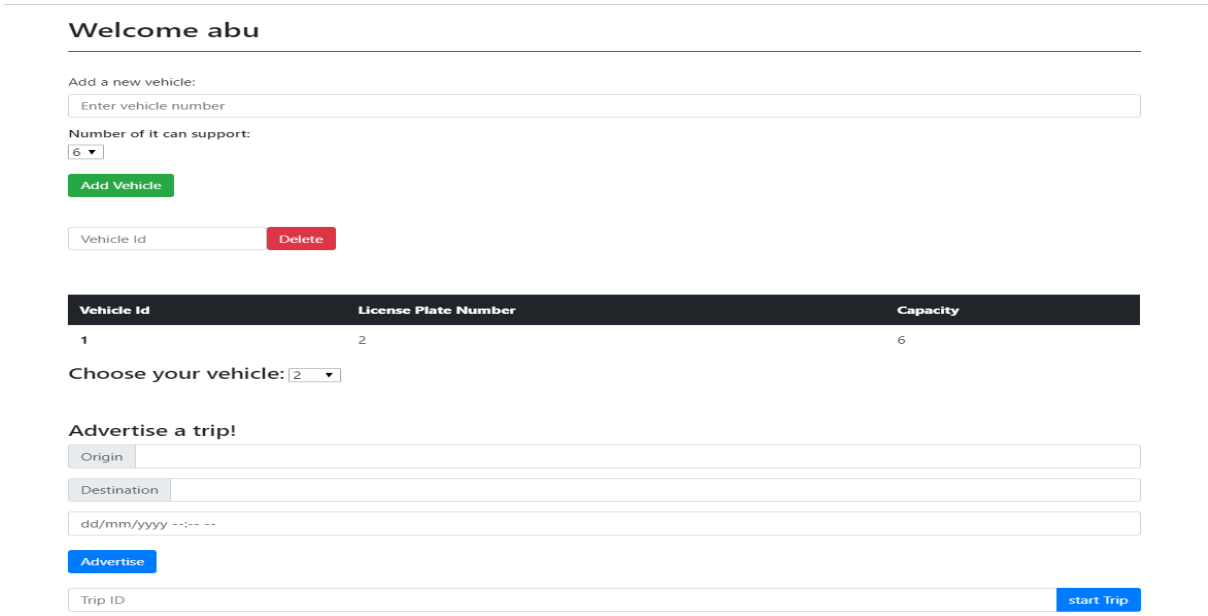
3.1.3 Sign Up Page

It is mandatory for the user to fill in all of the required fields in Sign Up page which will be used to register he or she as a user. For security purposes, the passwords are hashed using *bcrypt* and stored in our database.

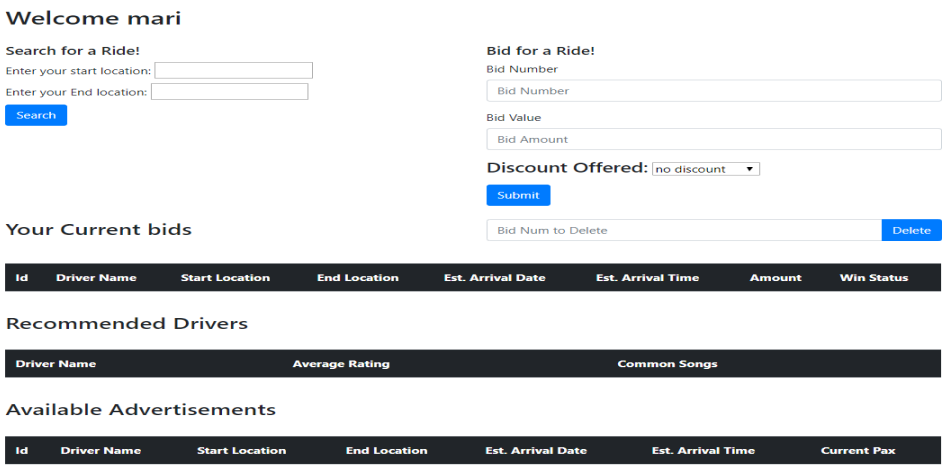
Upon successful creation of an account with our application, user is redirected to the login page. If an existing user tries to sign up for an account again, he or she will be redirected to the login page as well.

3.2 After Login

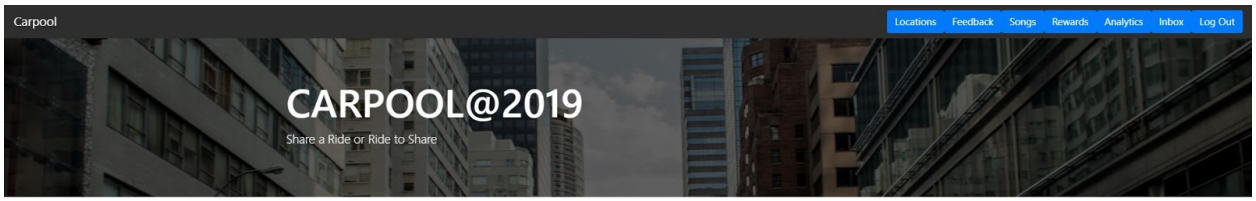
3.2.1 Passenger Dashboard



If user has logged in to their passenger account, the application redirects the user to the Passenger Dashboard as shown in the picture below.

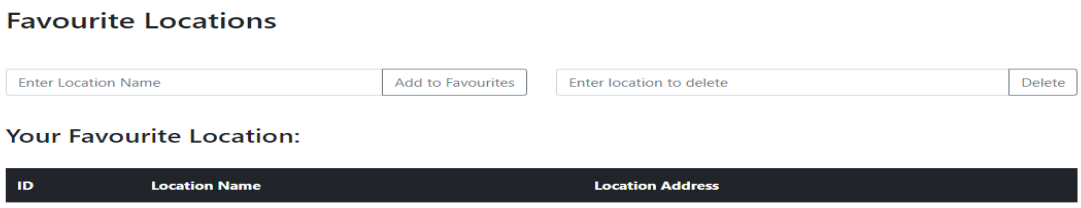


The passenger dashboard offers a user-friendly environment for the user to look for potential rides. The user will have to first enter their start location and end location. Upon that by pressing the button search, the two tables are displayed - namely the Recommended Drivers and the Available advertisements will be populated.
As a passenger, the user is able to view a number of options which are tied with aims of providing unique features which support the application. These options are as shown below in blue.



3.2.1.1 Locations

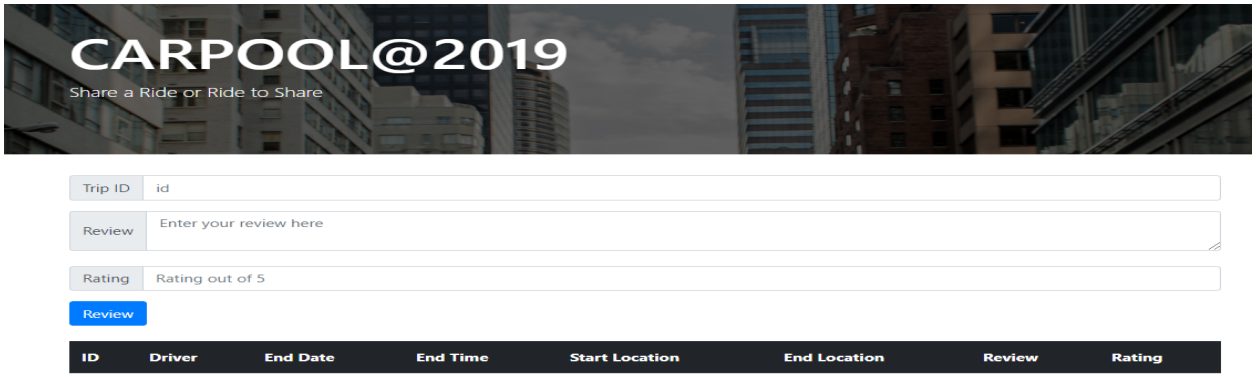
Pressing on Locations leads the user to the following page :



A passenger is able to add in his favourite locations Instantly, the favourite locations are displayed on the table, just below the input fields based on the query as shown above.

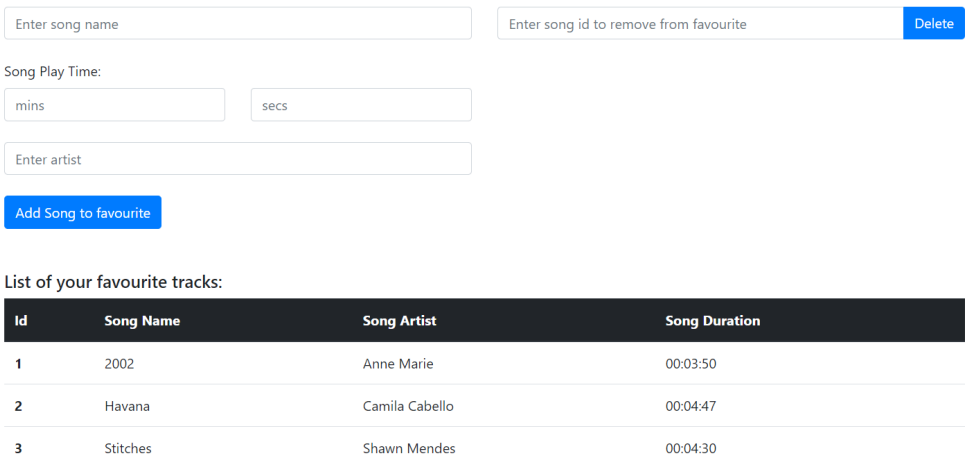
3.2.1.2 Feedback

By clicking on feedback, the user is able to provide feedback on the ride which he or she took.



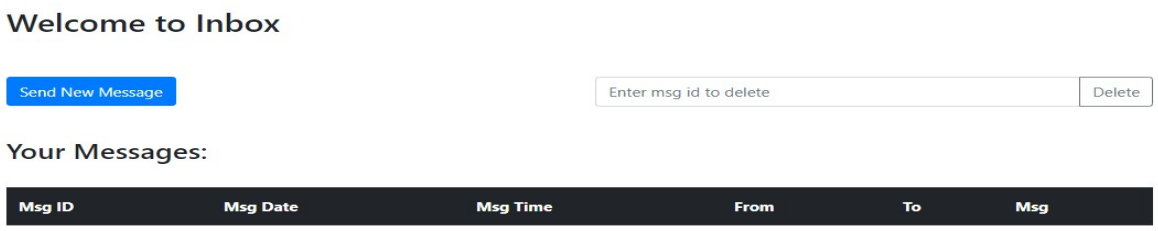
3.2.1.3 Songs

A unique feature which has been implemented, is that a passenger will be able to add their favourite songs to the database after entering the input fields in the songs page.

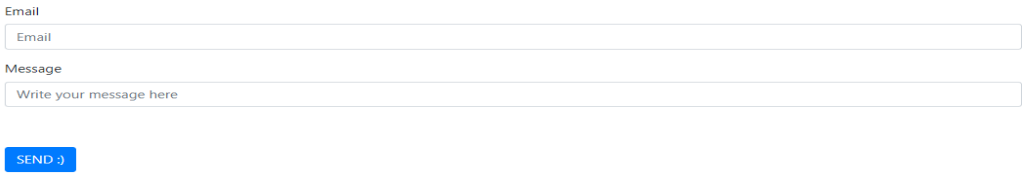


3.2.1.4 Messages

The passenger is able to send messages to any other fellow users of the application through the message interface. The following image displays the interface of the message interface.



The Your Messages table shows the messages which have been sent to the passenger who has been currently logged in.
Clicking on Send New Message allows the user to write a message for a fellow user. This redirects the user to the following page as shown below.



Once the send button has been clicked, the message is then sent to the recipient.

3.2.1.5 Analytics

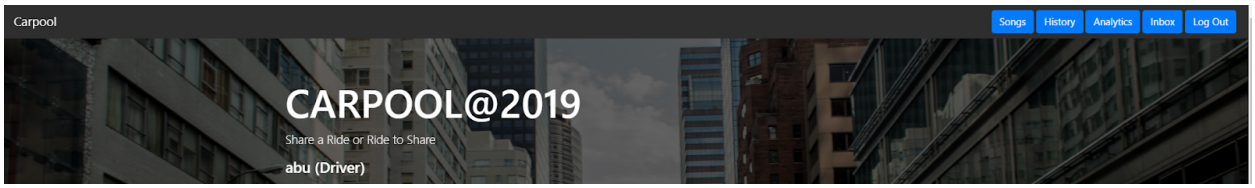
Clicking on Analytics redirects the user to the following page :

The analytics portion of our website allows passengers to see the hotspots of the most popular locations at any time of the day where they are able to filter by time. They are also able to see the chance of success and average bid done within that time period.

3.2.2 Driver Dashboard

If user has opted to login to their Driver account, the application redirects the user to the Driver Dashboard page which is as shown in the image below.

The Driver is able to navigate through various options presented below:



At the driver dashboard, the driver is allowed to choose which vehicle to drive and there are also input parameters displayed for driver to advertise a trip. The following is an excerpt from the various queries existent to relay information between the driver and passenger.

4. Challenges Faced

1. We were inexperienced with NodeJS and SQL and had to spend a considerable amount of time learning the syntax and how to integrate SQL and NodeJS together.
2. Time constraint was a significant problem as we had other modules as well. We also spent a considerable time conceptualising the ER diagram and had to rush a lot of the coding as a result.
3. The logic behind the ER diagram was time consuming as well as we had to understand what was going on in the lecture notes before applying it to the project.
4. We had trouble thinking of additional features for the project such as analytics and discounts. We spent a lot of time thinking of how to implement them before starting on the actual code.
5. Trying to figure out how to use Passport Local took up a lot of time
6. We initially used ReactJS and were very unsure on how to integrate them into our application as we could not access our psql database. We tried debugging but eventually switched the entire frontend to NodeJS.