HitchRide

Carpool Management System

07.05.2020

Group 2

ES16BTECH11019 - Siddharth Goel
CS16BTECH11027 - Abhishek Pawar
CS16BTECH11035 - Siddharth Shrivastava
CS16BTECH11030 - Saahil Sirowa

Overview

Car-Pooling Management System (CMS) is intended to help the user to share car rides with other users traveling on the same route. The user may intend to share his car or else ride with another user who is willing to share.

The carpooling software is designed and developed in the Django framework using the GoogleMaps API.

CMS is aimed toward a person who is a frequent traveler and is looking for a cheap and comfortable mode of transport. It will prove beneficial for office commuters who are headed on a common route and are willing to share the travel cost. Anyone can opt to provide a drive, thus reducing his/her expense.

Please Refer HitchRide Documentation for details.

Requirements Implemented

- **Web APP** for the user(driver) who is going to its destination and wants to share the ride.
- **Web APP** for the user(Rider) who wants to go from pickup to destination.

There are two categories of users:

1. **Driver** - A user who wants to share his ride with other people along the same route or is a full-time driver.
2. **Rider** - A user(other than driver) sharing a ride. He can book in realtime and will be assigned a driver from the pool of drivers available.

**Please Refer to the attached screenshot for all the features details.*

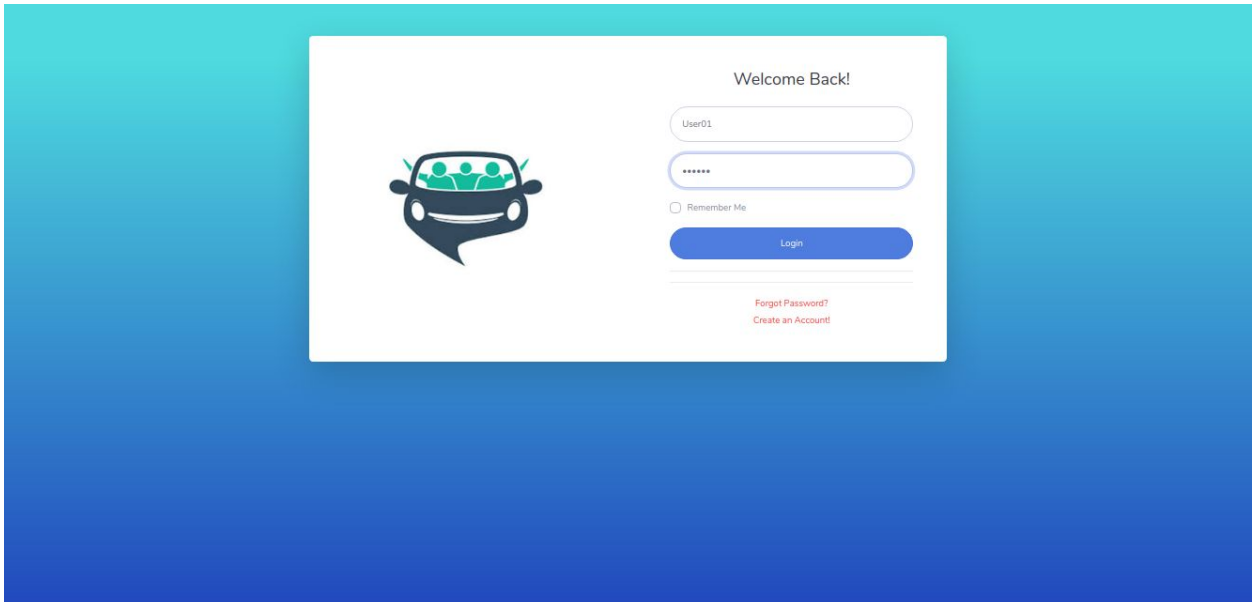
Requirements Not Implemented

- All **necessary requirements** mentioned in SRS **are implemented**. But there are some small functionality like **user feedback**, **Ride history of a user which** are not implemented .

Working & Screenshots

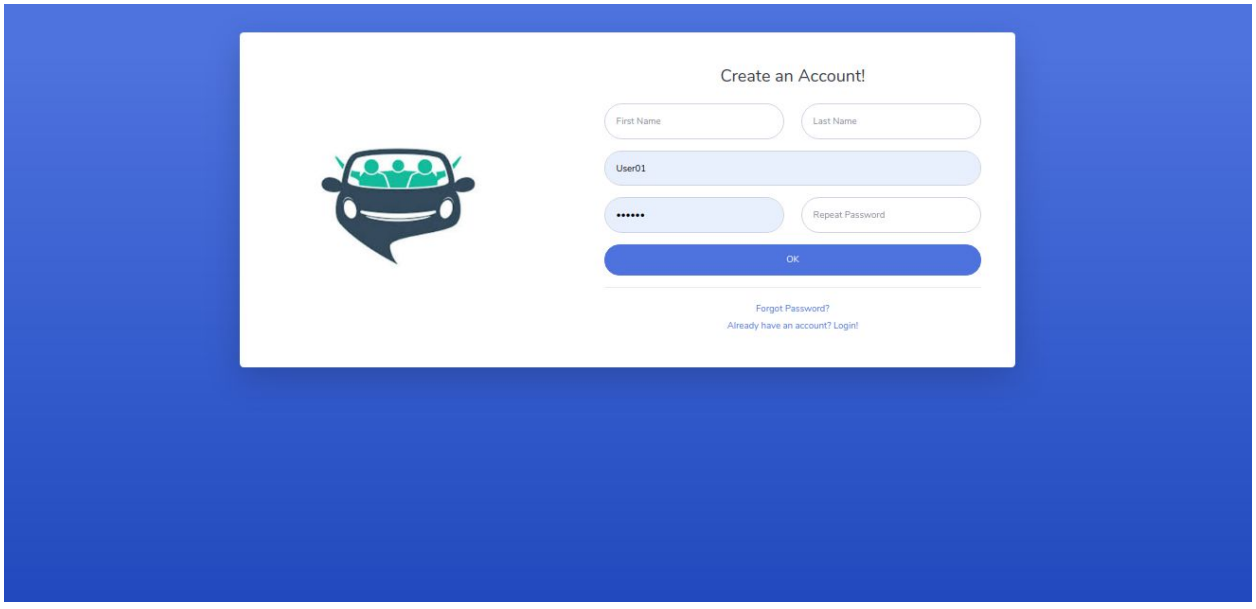
Login Page

- Authenticate the user with userId and password.



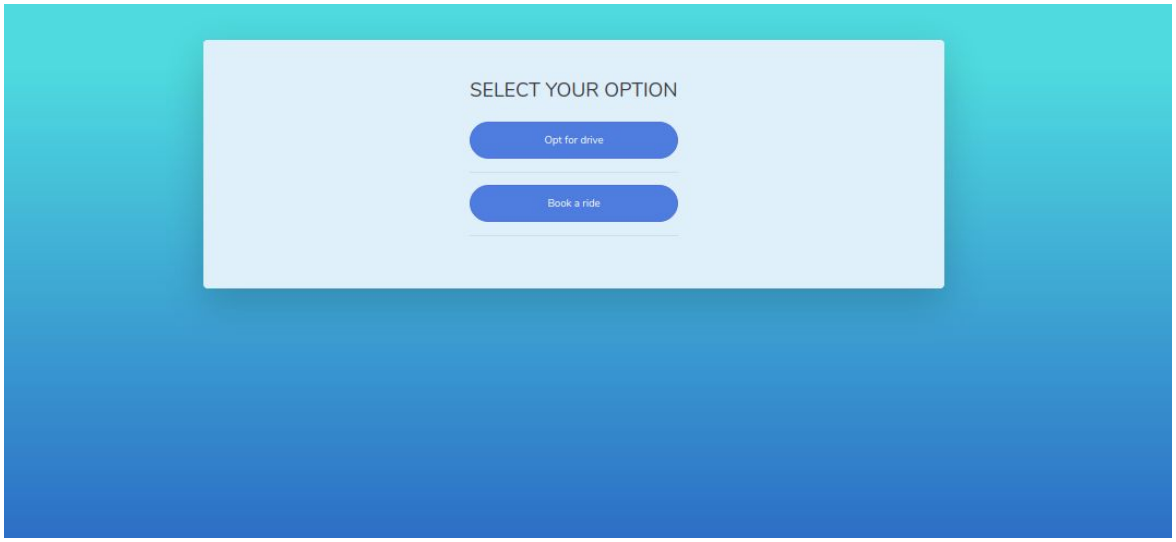
Registration Page

- Allow users to create new accounts with prefered user id and password (Min 6 char).



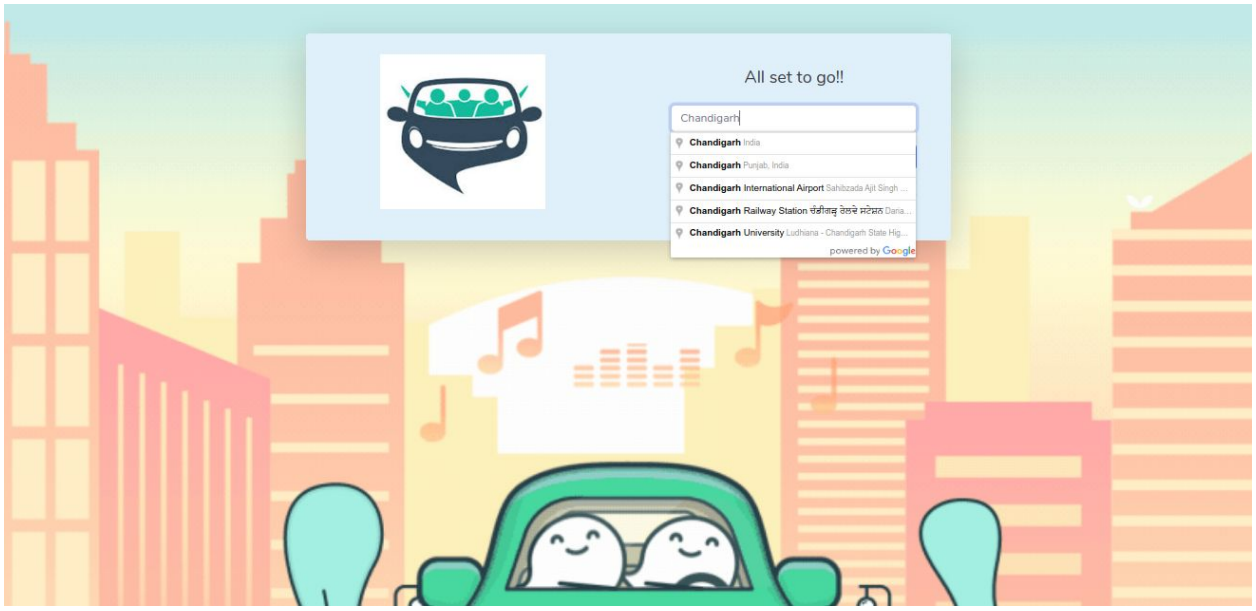
Drive or Ride! (Page)

- After login comes to this page which gives the user the option of **selection to drive or ride!**



Driver Flow

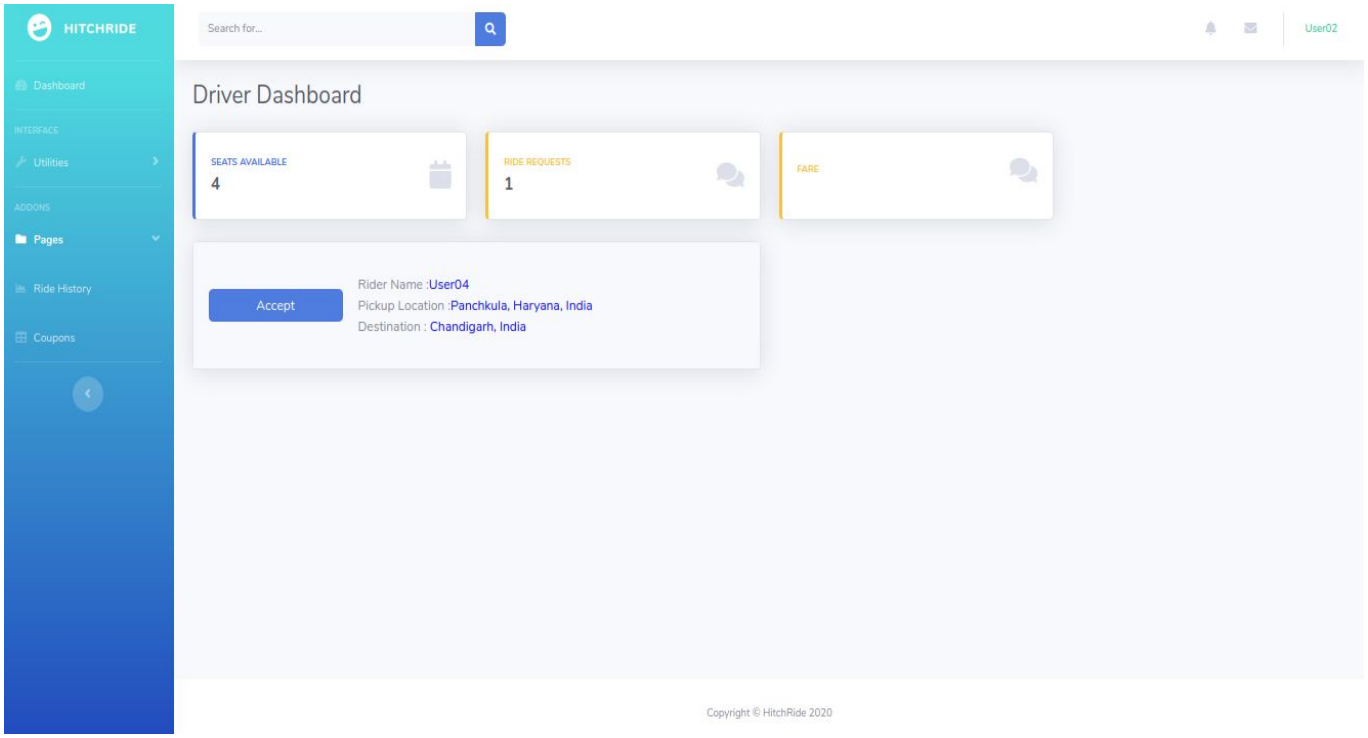
Once the user selects the Driver option the page asks for the destination. We have used the **Google places API** which is shown in the below screenshot. It **auto-suggest** the driver for its destination. In the background it also captures the current location of the driver using **Google GeoLocation API**.



Driver Dashboard

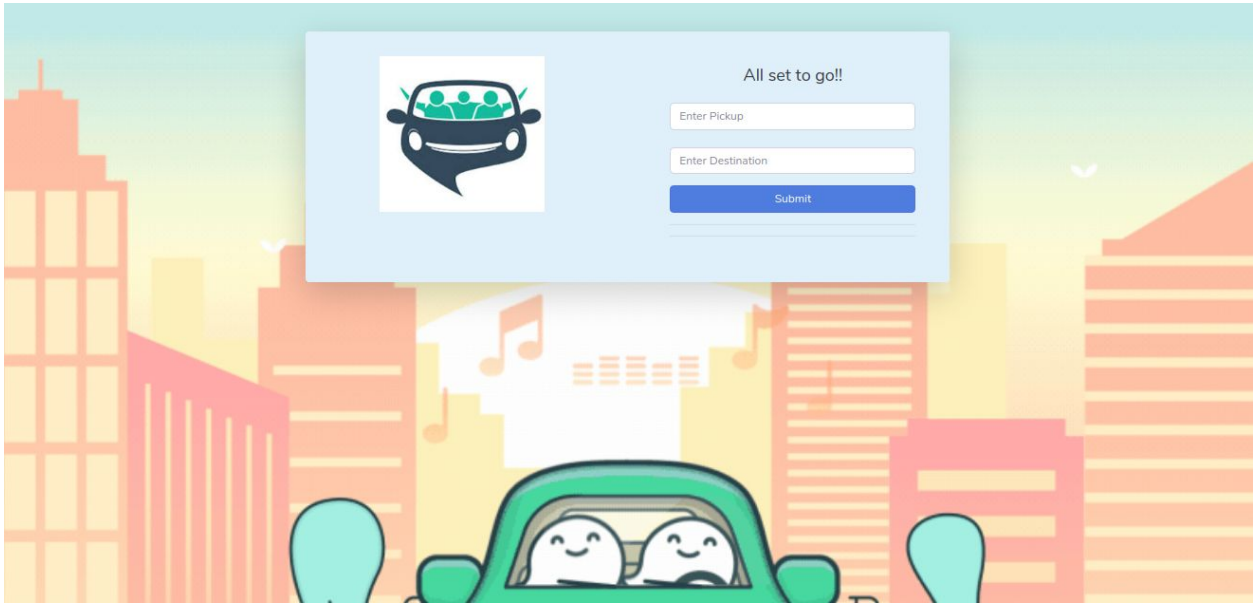
Once the driver submits its destination point it is redirected to the driver dashboard where all the **available riders will be visible** to him. In the below screenshot we can see one available rider in his dashboard.There is an accept button for each rider that gives the rider the **freedom to choose the rider** based on his location (PickUP and Destination).

As you can see We also have features for **Available seats, Current Ride Requests and Displaying Fare(once ride ends).**

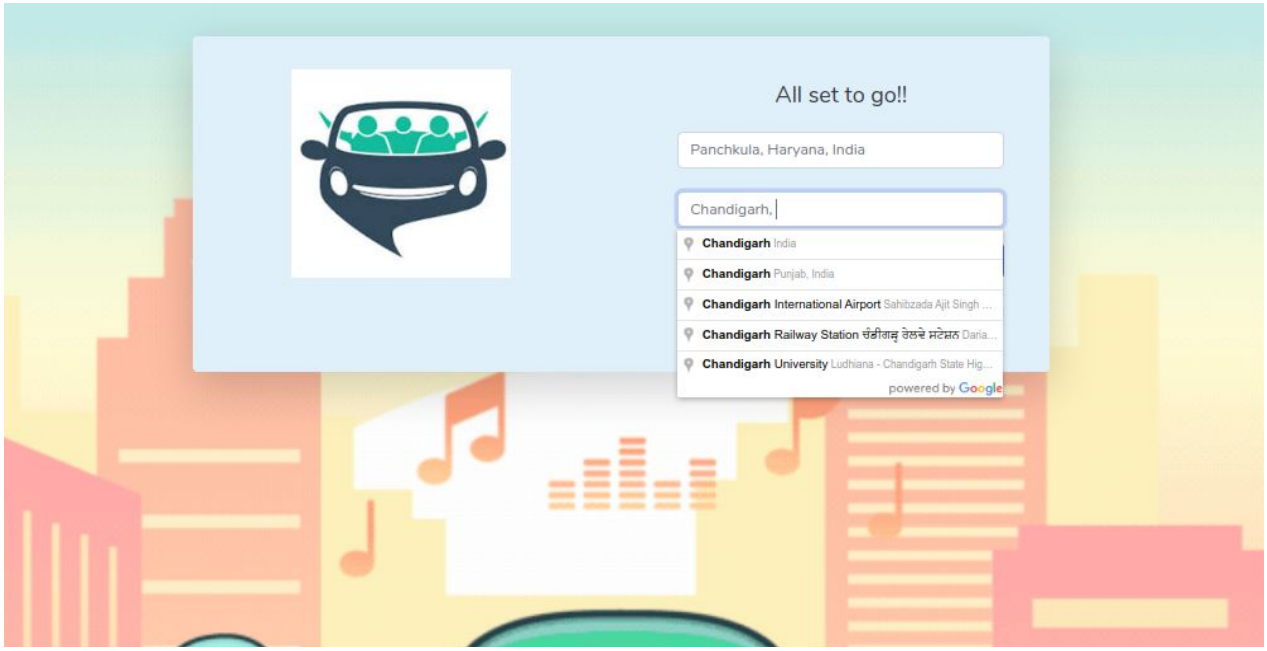


Rider Flow

If the user selects **"Book a Ride"** option, the following page appears for him, where he is asked for Pickup and Destination Location using **Google's Places API**.



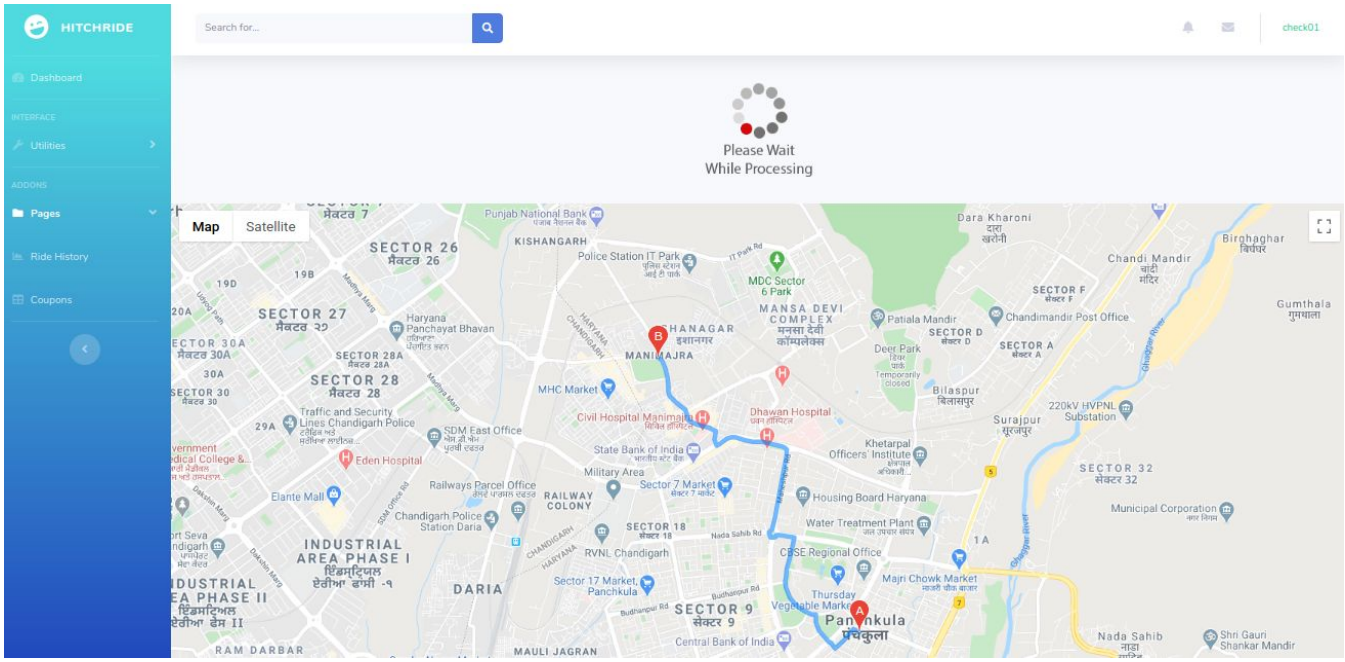
Here the rider is **auto suggested places** using the **Google Places API**



Rider Dashboard

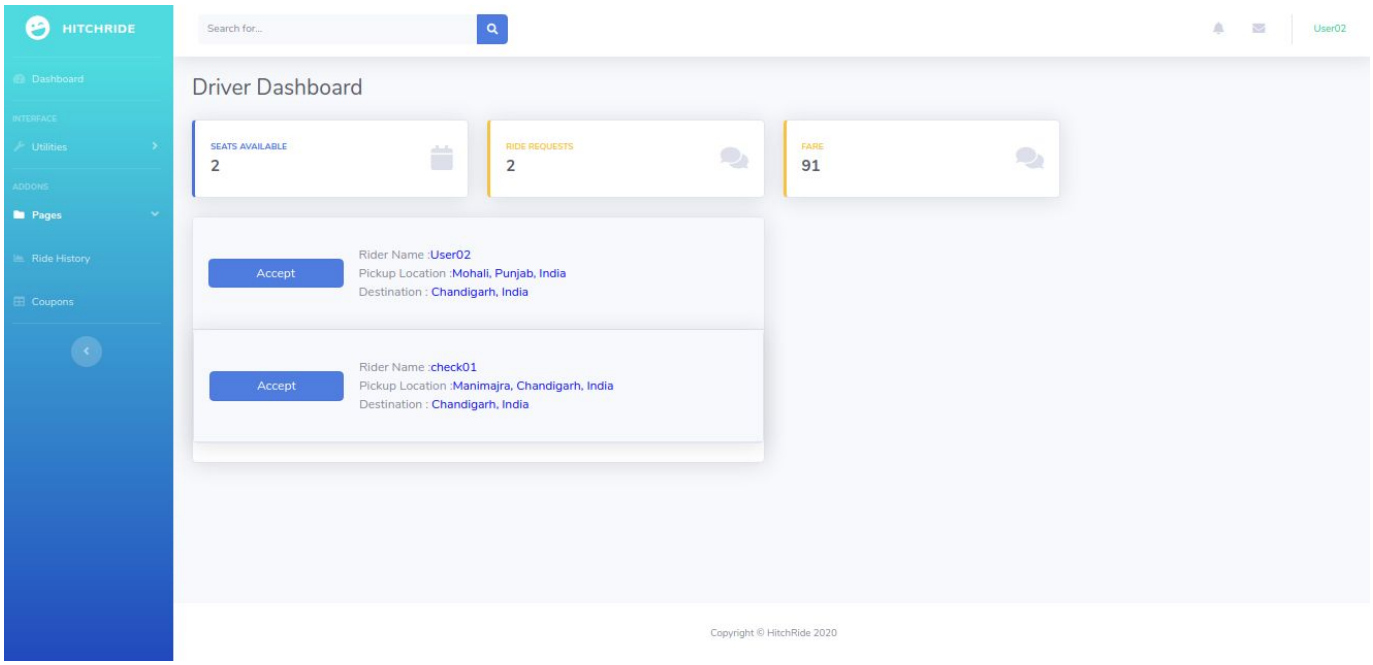
Once rider enters pickup and destination, in the background our app :-

- Collects the rider's current location using **Geo-Location API**, and
- While the ride request is being processed, **shows user Directions to Pick Up point** using **Google Maps Direction API.**

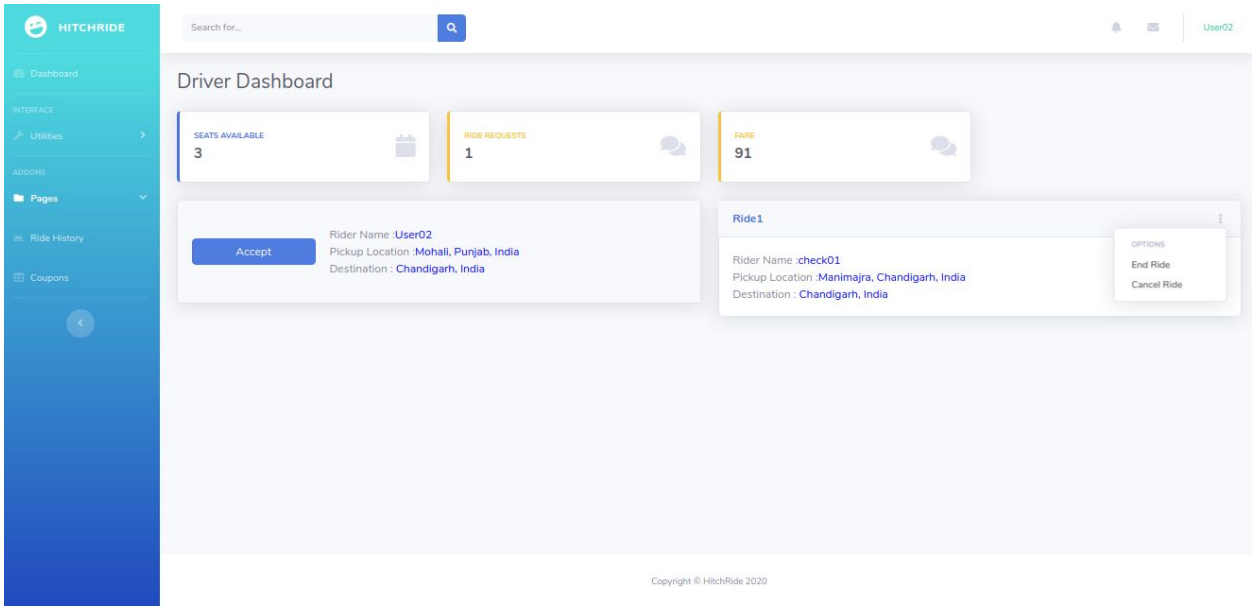


Driver Dashboard And Accept rider Request

1. In the **Previous driver dashboard only one ride** was visible but when the above rider requested a ride it's dashboard was **updated and now it's showing two riders.**



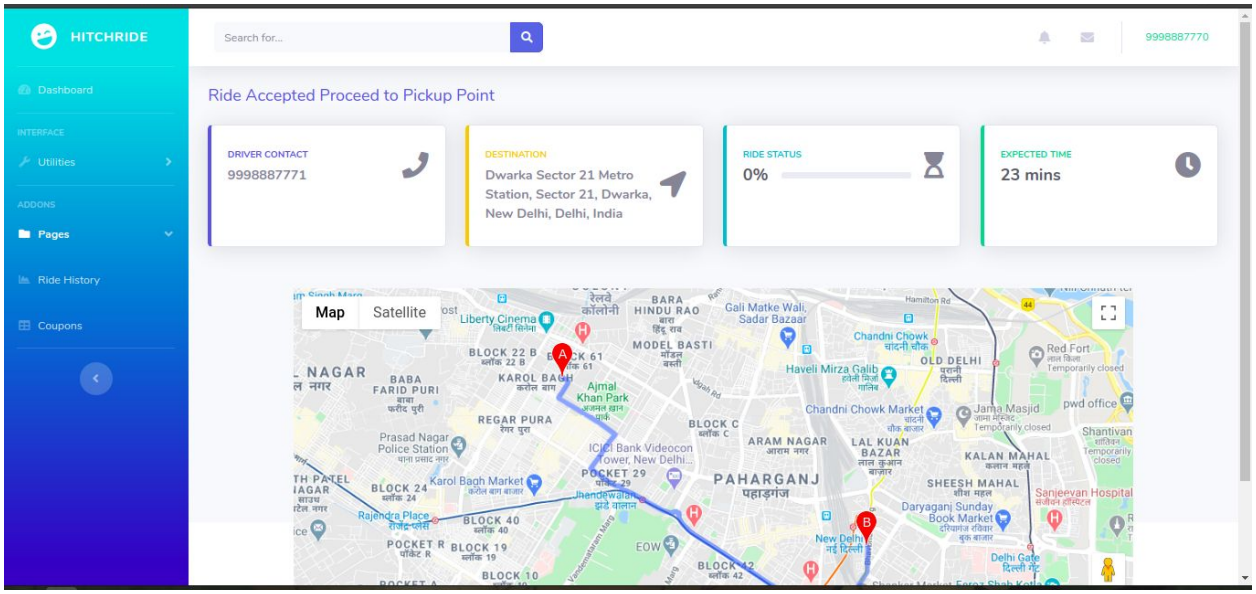
2. Once the **driver accepts the rider** request the **accepted ride is now added to the right side of the screen** which shows all the accepted rides.



Rider dashboard after accepting ride

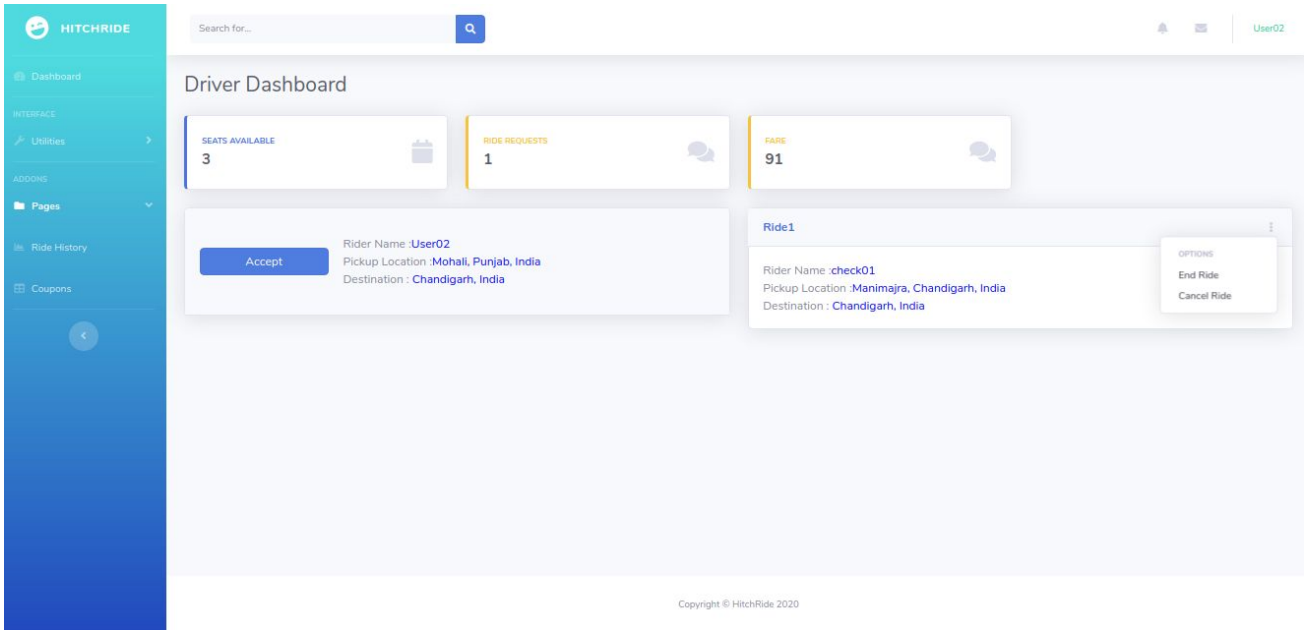
One the ride of user “check01” has been accepted, **the rider dashboard changes and now has 4 floting cards** which :-

1. Shows the Driver contact and Destination details.
2. Shows the ride completeness Status.
3. Estimated time to reach.



Driver Dashboard and End Ride

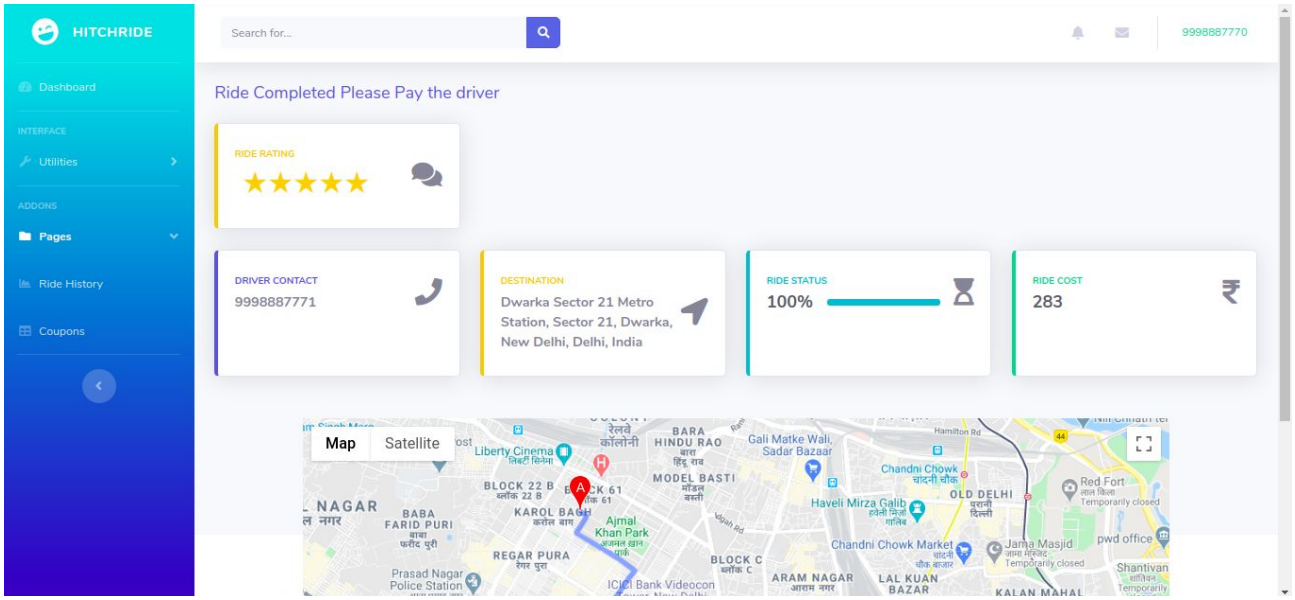
- For each selected ride the **driver gets an option to end the ride.**
- If the driver ends the ride then the **Fare and capacity of the driver car gets updated**.



Rider Dashboard After End Ride

Once the driver ends the ride from his dashboard the following is reflected on rider dashboard :-

1. **Actual Fare gets updated** in place of expected fare.
2. **Ride Status** also gets 100%.



Software Testing

For our Application we followed **Agile software development techniques where in. requirements, programming, and testing are often done concurrently.**

To ensure our application is properly developed and tested we used the following **Testing Cycle**:

1. Meets the requirements that guided its design and development and the general result its stakeholders desire.

   Cross verified all the use cases identified in SRS documents are implemented and working correctly.

   Asked our friends and family to use the application to ensure the general idea of the application is easy to use (without any external help) and fulfils all the necessary features required for a carpool management system.

   Also, we made sure all the code base is properly tested and documented for anyone to take it up in future.

2. Responds correctly to all kinds of inputs.
   - **Static Testing**

Static testing is basically the verification, reviews, walkthroughs, or inspections etc. Before committing/shiping each section of our code, we made sure it is being reviewed by at least one other member of our teams, and appropriate comments were given for any required changes.

To ensure this was properly handled we used github as our version control system, which provides us a platform to handle above efficiently.

- **Dynamic Testing**

  We ensured **proper logging statements** are printed on terminal or console so that any type of **runtime errors could be easily debugged**.

- **Unit Testing**

  For each of the functions(views.py) we wrote in our django environment, we wrote unit tests for all of these using the **pythons unittest module**. When you run your tests, the default behavior of the test utility is to find all the test cases (that is, subclasses of unittest.TestCase) in any file whose name begins with test, **automatically build a test suite out of those test cases, and run that suite**.

  We also ensure the code coverage of these unit tests is close to 90% to avoid any failures in our application.

3. Performs its functions within an acceptable time.

- **Functional Testing**

  Functional testing refers to activities that verify a specific action or function of the code. Functional tests tend to answer the question of "can the user do this" or "does this particular feature work."

  It was ensured all specific services (like google Maps API, Payment calculation services) were working correctly independently before they were integrated with our codebase.

- **Integration Testing**

  Integration testing works to expose defects in the interfaces and interaction between integrated components (modules).

  Before and While pushing the code for different modules (API's and other services (such as payment)), it was made sure that their interactions do not break any existing functionality and are seamlessly integrated with our system.

4. It is sufficiently usable.

- **Software Performance Testing**

Performance testing is generally executed to determine how a system or sub-system performs in terms of responsiveness and stability under a particular workload. It can also serve to investigate, measure, validate or verify other quality attributes of the system, such as scalability, reliability and resource usage.

Using Load Testing verified this by creating multiple user instances (both rider and driver) and made sure our application can handle the load properly, and perform just as efficiently as with less number of users.

Using Stress Testing verified that each of the important modules are working properly even under unexpected amounts of traffic. And using Stability testing ensured that software can continuously function well in or above an acceptable period.

5. Can be installed and run in its intended environments.

- **Installation Testing**

  Software installation instruction, as mentioned in the project readme, was followed on multiple Operating systems, and there was no issue following the necessary steps.

- **Compatibility Testing**

  A common cause of software failure (real or perceived) is a lack of its compatibility with other operating systems (or operating system versions, old or new), or target environments that differ greatly from the original (such as a terminal or GUI application intended to be run on the desktop now being required to become a Web application, which must render in a Web browser).

  Therefore, it was ensured all modules are working properly by running applications across multiple Operating Systems(Ubuntu, Windows or MacOS) and Web Browsers(Chrome, FireFox or Safari).