Final Report

Link: https://github.com/HowardHuang1/Rideshare.git

Work Distribution Table

Rohil Kalra	Web Scraper, Email Sender, Nearby Ride Optimization Algorithm
Howard Huang	Create-Ride Page, Ride Stack, Static Map Image, Join Ride Button
Parthiv Nair	Backend APIs, Profile/Create-ride Styling, Calling Google-APIs, connected Search Ride API, Modals
Satvik Eltepu	Profile-Page, Connecting Update / Leave Ride API, Pop-up alert, Create/Search Ride Modal
Aidan Robinson	Login/Sign-up Page, Website Styling, Input Validation, Alerts
Dhruva Sankhe	Home page with instructional details, navbar, and routing

Project Functionality Summary

Bruin Cruisin is an application that allows college students to organize carpools and rideshares with fellow students to popular destinations like the airport, downtown, and concerts in the area. Bruin Cruisin features the ability to create a ride, join a ride, modify a ride, and leave a ride as well as capabilities like filtered search for rides within a 5 mile radius. Past rides are displayed on the profile page in the ride history section. Bruin Cruisin also has 3 special features: Real-time webscraped prices from popular ride sharing platforms, email notification that notifies users with a confirmation message whenever they have just created a new ride as well as update emails whenever a new user joins the ride or the ride has been updated, and a ride price optimization algorithm that suggests new pickup locations in the nearby area that offer better rates.

Routing

Howard made the navbar component as well as the routing to the home page, create ride page, and the profile page.

Home Page
The home page, worked on by Dhruva, is the first page that the user lands on after opening the application. It involves four main components: navbar, herosection, redirecting button, and a footer. Dhruva put together these components and styled the home page, while uploading appropriate images in the 3 divs on the page. The herosection, which contains all the text on the home page, has a brief description of the main purpose of the application, followed by descriptive steps on how to use the features on the other two pages (namely Create Ride and Profile pages). The button under the title routes users to the signup/login page created by Aidan, and if the user is already logged in, it will route them to the create ride page.

User Authentication
The User Authentication is a nicely styled Hook form embedded into imported Chakra UI components that nicely define behavior for error notification in the form. Aidan made the Signup and Login pages, as well as implementing a logged-in state for the app. The signup form requires inputs that are longer than 3 characters. These inputs are: a full name, a unique username, an email address, a password, and password confirmation. Login is similar, but only requires the username and a correct password.

Error checking is handled in 2 ways. Syntax errors, such as inputs that are too short and a “confirm password” value not matching the “password” value, are handled on the front end. This means if one of these errors are present in the form, these authentication components won't even make an api call– the login page will simply notify the user that they cannot submit until they fix the syntax errors. This was implemented by Aidan. Logical errors, such as registering a username that already exists or inputting an incorrect password, are handled backend-side. The API call will be made, and will return error messages about what went wrong, and the front end displays these errors. The backend checking was done by Rohil, and the front-end display of the errors is done by Aidan

Once a user logs in, the login Component lets the rest of the app know that they are logged in. Inside the main App component that renders everything, there is a “login” state that stores the username, initialized to “null” at first. The function to update this is passed down into the login, signup and profile components (since you logout through the profile page), and whenever the login info changes, it calls this function. The “login” state is then passed to all other components, so that they can perform actions specific to the user that is currently logged in. The Login-state handling and rendering is created by Aidan

Create Ride Page

The Create Ride Page provides the main functionality of the program, posting, retrieving, and searching for rides from the ride database. In here, a user creates a ride based off of modal. The first version of the modal was created by Howard and Dhruva did the styling. Aidan and Dhruva worked on additional modal features. The modal is a Hook form that takes in a Pickup & Drop-Off location, Date & Time, and number of riders (4 or 6). The modal has 2 options, it allows users to either create a new ride or search for existing rides. The drop down buttons specify whether the ride is open or closed, and whether the time is in AM or PM, were created by Dhruva. Howard and Parthiv worked together to implement the API call to the server for creating new rides based on the input passed into the form on the modal. To improve the modal's UI, Dhruva added apt icons in each search bar using font-awesome, which is a CSS based icon toolkit designed to be used with inline elements. As for the display, there are 3 parts: A vertical stack displaying rides, a map of a selected ride's path, and the "Create Ride" modal.

Howard created the vertical ride stack that displays all active rides on the left side of the screen. The ride stack is composed of individual ride card components that fetch ride data from the backend and display it to the user including cost of the ride, date, time, pickup and dropoff locations, as well as icons displaying the number of riders currently in the ride. Parthiv implemented the axios get requests to the backend to fetch the data.

Howard implemented the map that displays on the right side of the screen showing the shortest path distance between pickup spot and destination. Howard and Parthiv worked together to implement the axios get requests to the backend to fetch the rendered map from the routes API implemented on the backend by Rohil.

Create/Search Ride Modal

Satvik programmed the create/search ride modal, which is what pops up when the user attempts to create or search for a ride. The modal requires 6 values, which are pickup location, destination, date of ride, time of ride, and number of riders. The user then has the option to click either the create ride or search button, both of which Satvik created. If the create ride button is clicked, the frontend calls the GET: search-ride API endpoint. If a similar ride exists in the database, an alert modal pops up, which allows the user to potentially save money by joining that ride. If the user clicks 'x' from this alert modal, they are taken back to the create/search ride page in which they see the similar ride(s) and can join them. This is one way data from the database is searched. If the user selects continue, rather, a new ride is created anyway. If a similar ride does not exist in the database when the create ride button is clicked, a new ride is created. This is how data is uploaded to the database. There is also validation checking for whether the

locations are invalid. If the user selects search directly, however, the GET: search-ride API endpoint is called and a data search is performed, showing similar rides.

Profile Page

Satvik created the profile page of the BruinCruisin' app. This page is split into two sections, profile information and ride history. Both of these sections consist of tables that display dynamic data to the user. The profile information section makes use of the GET: user-data API endpoint in order to retrieve the full name, email, username, money saved, carbon saved, and number of rides. The ride history section makes use of the GET: rides-for-user API endpoint in order to retrieve the entire list of rides that the user has and will be on. However, these rides are not sorted by whether they are in the past or future. So, Satvik created a function that takes the entire list of rides and sorts it into existing and past rides by comparing the ride's time to the current time. Additionally each existing ride has two buttons, update ride and leave ride, both of whose functionalities Satvik coded. The update ride button, only when the user is the creator of the ride, opens a modal that allows the user to enter in new time and number of riders values to update the ride. The Put: update-ride API endpoint then successfully updates the ride's information in the database. The leave ride button, when clicked, calls the POST: leave-ride API endpoint, allowing the user to remove the ride from their list of rides in the database. Lastly, the logout button at the bottom of the page allows the user to log out of their account.

Backend APIs

Our Backend utilized 10 APIs in order to connect the frontend of the website with the backend. These are organized by user-focused and ride-focused.

The following are the user-focused APIs:

1. POST: Create-user
    - Takes in a full name, username, password, and email.
    - Performs input validation checks and saves the passwords as encrypted in the database.
    - Adds the user to the MongoDB user collection
2. POST: Login
    - Takes in a username and password.
    - Returns true if usernames and password match correctly in the database, false if bad password, and null if not found
3. GET: User-data
    - Takes in a username

    - Returns name, email, money saved, carbon saved, and number of rides user is in
4. GET: Rides-for-user
    - Takes in a username
    - Returns an array of rides that the user is in. (Filters all rides found in Mongo)

The following are the ride-focused APIs:

1. POST: create-ride
    - Takes in username, data, time, AM/PM, locationFrom, locationTo, numRidersAllowed, search
    - Search is a boolean where if true, will first search if similar rides exist. Returns an error message if so or will continue to create ride
    - Calls Google Maps APIs to get address info, ride distance, ride duration, ride price, etc…
    - MongoDB initialize a rideID and provides it to the ride
    - All information is saved in the rides collection of MongoDB
2. POST: join-ride
    - Takes in username, RideID
    - Simply adds the username to the username array of the ride object with the given rideID
3. POST: leave-ride
    - Takes in username, RideID
    - Simply removes the username from the username array of the ride object with the given rideID
4. PUT: update-ride
    - Takes in username, rideID, time, AM, numRidersAllowed
    - First checks if username is the first element of the ride object's username array (only ride creator can update rides)
    - Edits the ride in the database with the updated information
5. GET: search-ride
    - Takes in locationFrom, locationTo, date, time, AM
    - Searches for rides within 1 mile from the pickup and dropoff location, starting 15 minutes earlier or later
    - Returns an array of the relevant rides from the search
6. GET: ride-image
    - Take in rideID
    - Outputs link for google static map route image

Special Feature 1: Web Scraper

In the project specification, it states that we must implement “three more distinct features. Points will be given according to how creative and useful the features are.” The first special feature is a web scraper

Rohil implemented a web scraper using Puppeteer to fetch fare values for ridesharing services from the website ride.guru. The main function, scrapeFareValues, takes two locations and the number of riders allowed as parameters. It utilizes Puppeteer to launch a browser, navigate to the specified URL with the provided locations, and wait for the page to load. Using techniques like filtering and string manipulation, it extracts the fare value from the HTML content after specific elements have rendered on the page. The extracted fare is then processed based on the number of riders allowed and returned as the result. In case of any errors during the scraping process, they are caught, logged, and null is returned to indicate the failure.

The implementation showcases the use of asynchronous functions and await statements to handle promises, ensuring that the necessary page elements are accessible before proceeding. By leveraging Puppeteer's automated browser capabilities, the web scraper interacts with the web page, simulating user actions like button clicks and retrieving the required data. The scraped fare value can be further utilized within the application for various purposes, such as comparing prices or estimating fares. To prevent significant slowdowns, Rohil has set a time limit of 30 seconds for the scraping process, ensuring efficient execution within the backend server.

Special Feature 2: Email Client

Rohil has implemented a function that utilizes the Nodemailer library to send emails. The function takes various parameters, including the sender's email, password, recipient's email, subject, and message. It creates a transporter using the SMTP settings for Gmail, setting up the necessary configuration. Once the transporter is ready, the function defines the email options, specifying the sender, recipient, subject, and text of the email. It then attempts to send the email using the transporter and logs a success message with the message ID if the email is sent successfully. In case any errors occur during the process, they are caught and logged for error handling and debugging purposes.

Furthermore, Rohil has expanded on this general email-sending function by implementing additional functions that generate email content with specific parameters.

For instance, the createEmailSender function constructs the email body by including details such as the pickup location, destination, date, and estimated ride price. These specialized functions allow for flexible and dynamic email content generation based on different scenarios. Additionally, Rohil has also developed an updateEmailSender function, which sends email notifications to all users containing relevant information when a ride has been updated. A particular challenge encountered during this implementation was ensuring the proper formatting of the email body. Careful attention was given to writing clean and organized code to avoid generating messy and difficult-to-debug email content.

Special Feature 3: Nearby Ride Price Optimization Algorithm

Rohil has developed an algorithm that enhances the user experience of finding rides by identifying nearby alternatives that offer significantly lower prices. The algorithm he implemented utilizes a greedy randomized approach, combined with the utilization of geolocation and distance matrix APIs, to achieve this goal. The algorithm begins by leveraging the geolocation API to obtain the coordinates of the original ride's pickup and drop-off locations.

Once the coordinates are obtained, the algorithm utilizes the distance matrix API to calculate the distances between these locations and potential alternative ride options. By considering the proximity and distance factors, the algorithm systematically searches for rides that provide a cost advantage. It explores nearby areas, identifying rides that are in close proximity but offer more favorable pricing due to factors like surge pricing or discounts. This approach maximizes the chances of finding cost-effective alternatives, providing users with better choices and potential savings.

To collect the necessary data for implementing the algorithm, Rohil ran a data collection script on a Google Cloud Platform (GCP) virtual machine (VM). This script performed approximately 2000 web-scrapes to gather a comprehensive dataset of ride information from various ridesharing services. Running the script on a VM provided efficient and parallel execution of web-scraping operations, utilizing the VM's computing power and network connectivity for faster data acquisition.

Difficulties Faced/Reflections

We faced lots of challenges on the create ride page with connecting the apis via axios requests to the backend and had to play around with the data types of the arguments that were required in the get requests. We addressed this issue by frequently writing documentation so that the frontend wouldn’t have to look through complicated backend code.

We also had difficulty with prop drilling and passing props and useState functions down multiple layers of components. This required us to frequently lift and lower state, making rather inefficient code at times. Next time, we should consider using React Redux for statement management

We had troubles with state-based rendering. For example, when a user goes to update a ride in their profile page, they should only see an update ride modal if they update a ride they created. Otherwise, it should show a model telling the user they cannot update the ride. What's hard is that often, these issues have their own unique solution, rather than a one size fits all solution for these problems.

In this project, our backend code loaded the entire database into the frontend for every GET request and then proceeded to filter out specific data. In the future, we should use Mongoose functionality for filtering data more efficiently.

Discussion of Improvements

Additional Features:

1. Price Comparison between Uber and Lyft using Uber and Lyft APIs
2. Directly booking rides by implementing a Paypal payment system using the Paypal API and creating a 2nd web scraper that automates the placing of payment and booking info into 3rd party ride sharing sites like Uber and Lyft
3. Creating a finance dashboard on the profile screen with the amount of money and carbon emissions saved plotted over a period of time
4. Adding a messaging feature that allows carpoolers to directly message and chat with people in their ride