Technical Documentation

Project Name: Truman RideShare

Version: 1.0

Authors: Jake, Lanz, Matthew, and Umair

Date: 04/24/2025

Status: Draft / **In Progress**

Table of Contents

Introduction

Purpose

The main problem that the web app will try to solve is the lack of a service that helps transport people around Kirksville. This will target the Truman State community, those who don't have a consistent method of transportation. There are no current services like Uber and Lyft that can help with this problem. The idea stemmed from a story from a parent's point of view. There is a Truman parent Facebook group, where there are a great deal of comments relating to carpooling. The comments often go along the lines of, "My student needs a ride from Saint Louis to Kirksville this Sunday," or, "My student left something at home; is anyone leaving from Kansas City to Kirksville soon?" The problem not only stems from leaving/going to Kirksville but also within it. Small trips around Kirksville, as simple as getting groceries, may be difficult for some. For example, international students aren't able to go to the grocery store or the airport without having to find someone to take them. Currently, they have a GroupMe group chat that they use to find people; however, at the end of the day, it's only meant for basic communication. There aren't any features that would make it easier for the requester. There is also the human aspect of not wanting to be seen as annoying. When asking for help, once or twice wouldn't be a problem; however, asking all the time can put a mental burden on some. People don't want to be the cause of trouble for others, which is why we propose this type of solution.

Scope

The web app will be used to provide a ride-share service for students at Truman State University. Users will be able to arrange rides to and from Kirksville and within Kirksville. This aims to help students without a consistent method of transportation.

The app will focus on solving problems such as:

- Finding rides for long-distance trips (e.g., Saint Louis → Kirksville, Kansas City → Kirksville, etc).
- Coordinating local trips (e.g., grocery runs, airport trips, errands around town)
- Providing user-friendly features (like request posting, ride offers) that are missing from basic communication apps.

Target Audience

The primary target audience for this web app is the Truman State University community, specifically students, though staff and faculty can also use it. Anyone who needs reliable transportation within Kirksville or to and from surrounding cities like Saint Louis and Kansas City will find the web app to be useful. The app especially focuses on students without personal vehicles, such as international students, underclassmen, and others who may struggle with accessing grocery stores, airports, or traveling home during breaks.

System Overview

Homepage

The homepage provides basic information about the project, including what it is, why it matters, and some features of the solution. There is a button labeled “WEBSITE TERMS & SERVICE” that redirects the user to the terms and service.

Terms & Service

If the user decides to click “WEBSITE TERMS & SERVICE,” they will be redirected to see the User Agreement, which is the same as the Terms and Service. It has 10 different sections covering everything the user should be aware of when using Truman Ride-Share.

Navigation Bar

At the top of the page, it features a navigation bar that directs users to different sections of the website. Depending on whether the user is on mobile or desktop, the desktop view displays a list of links, while the mobile navigation presents a drop-down menu of these links. The navigation bar also varies based on the user's login status. If the user is not logged in, the navigation bar offers access to the Homepage, the Signup page, and the Login page. If the user is logged in, the navigation bar provides links to the homepage, profile page, dashboard page, and ride listing page, and includes a logout option.

Signup page

If the user is using the website for the first time, they will have to go to the signup page. On that page, there is a red button labeled “Sign up with Google.” It opens a pop-up, where the user must use their Truman email address to sign up. There is a note below the button letting the user know this.

Login page

If the user has signed up before, then they can head to the login page, where there is a red button that says "Login with Google," which will allow the user to sign in with their Truman email.

After signing up, the user will be redirected to the dashboard page, where the User Agreement pop-up will appear. It holds the same information as the Terms & Services page. They have to agree to the terms and conditions at the bottom of the pop-up before being allowed to continue. After doing so, another pop-up will be displayed, letting the user know that they will be redirected to complete their profile. Also, it informs them to know that if they want to offer a ride, they must register their vehicle. The user can click "Go to Next Page" to continue.

Setup Profile

The setup profile page asks the user to enter their name, contact type, and the contact value. The contact type includes phone number, email, and social media (Twitter, Instagram, Snapchat, etc.). The contact value is the phone number itself, the email address, the username of the social media, etc. The user can add as many contact types as they want, and they can delete all but one. They need to fill out the form and click "Save Profile" before being able to exit. After the user is done, they will be redirected to the profile page.

Register a Vehicle

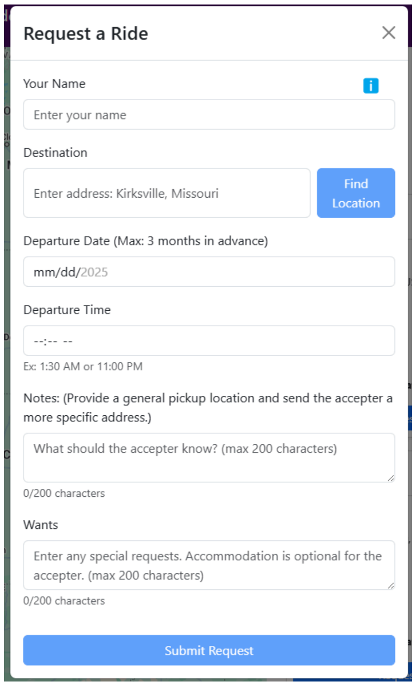
On the profile page, the user can register their vehicle. It asks for the color, make, and model of the vehicle. The user must fill out all the information before they can exit. After filling out the form, they can click "Save Vehicle" and "Exit" to be redirected back to the profile page.

Profile Page

The profile page includes information about the user: their name, email, contact information, vehicle info, if they've completed their profile, if they've completed the user agreement, and when their account was created. Below that, there are four buttons for them to use. The first two are for changing their profile or vehicle information. The third button leads them to the User Agreement/Terms and Services. The last button is for them to delete their account. If they click it, a warning appears before allowing them to proceed with the deletion.

Ride Listings

This page includes various features. The first functionality is the map, which displays the destinations users want to reach. Each pin color has a specific meaning: the blue pin corresponds to offers, green pins correspond to requests, and the lone purple pin indicates the location of Truman State. On the right side of the screen, in the desktop view, there are several buttons alongside the ride listings. At the top are the "request a ride" and "post a ride listing" forms. Here are pictures of them (some forms include an info button detailing how the form works):



Request a Ride

Your Name

Enter your name

Destination

Enter address: Kirksville, Missouri

Departure Date (Max: 3 months in advance)

mm/dd/2025

Departure Time

--:-- --

Ex: 1:30 AM or 11:00 PM

Notes: (Provide a general pickup location and send the accepter a more specific address.)

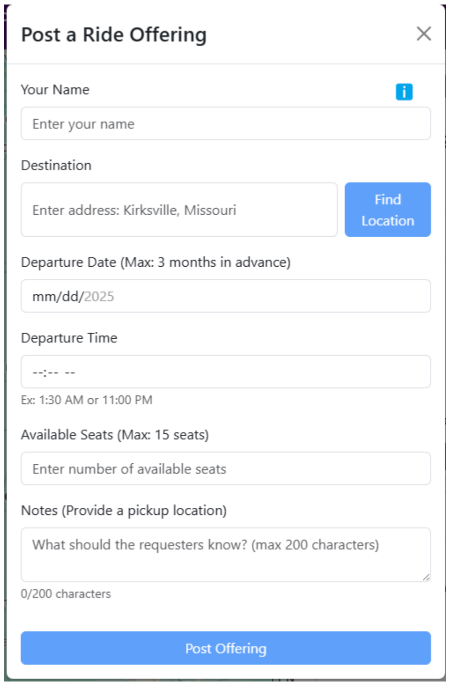
What should the accepter know? (max 200 characters)

0/200 characters

Wants

Enter any special requests. Accommodation is optional for the accepter. (max 200 characters)

0/200 characters



Post a Ride Offering

Your Name

Enter your name

Destination

Enter address: Kirksville, Missouri

Departure Date (Max: 3 months in advance)

mm/dd/2025

Departure Time

--:-- --

Ex: 1:30 AM or 11:00 PM

Available Seats (Max: 15 seats)

Enter number of available seats

Notes (Provide a pickup location)

What should the requesters know? (max 200 characters)

0/200 characters

Below those buttons is the filters button. It allows the user to change the view of the posts. They can choose to only show ride offerings, requests, favorites, or both. They can also sort the posts by the default view, earliest departure date, and latest departure date.

Posts

There are two different posts where the user can create a request and an offer. The posts show up in the ride listings and the dashboard pages.

Offer a ride

In the offering post, it displays the destination, the departure date, the departure time, notes from the creator, available seats, and the number of users waiting. Users can interact with the post in several ways. They can click the profile button to view information about the poster, favorite the post, and request a ride. If they request a ride, they can send a quick message to the offerer. When they do request a ride, the post interface updates slightly. It favorites the post for easier access. It indicates that they are on the waiting list and allows them to cancel the request to join. The available seats are also updated to reflect the current status of the accepted list.

Request a ride

The request post shows similar information: destination, date of departure, time of departure, notes, and wants. The user can also check the requester's profile and favorite post. The user can choose to accept the request. If they do accept the request, then the post will be favorited so they can easily find it again. The user can also click the unaccept button to retract their action.

Dashboard

The Dashboard has two sides, assuming desktop view, explained below.

Recent Listings

This part of the page has three different views that the user can choose from. “Both” is the default view, which shows the five most recent postings of the request and offering posts. The “Requests” view shows the five most recent request posts. The “Offerings” view shows the five most recent offering posts. The user can interact with the post as mentioned above.

Your Recent Post

This part of the page has all the posts that you have made. It includes your offerings, requests, both, and your favorite posts. Your offerings include two new actions. The first is management. If you click on it, it shows all the users on the waiting list for the post. The user can check the requester’s profile and/or accept. If the user decides to accept the request, the requester gets moved to the accepted users list. Then, to update the visuals of the post, click the “x” to refresh the page. In the accepted users tab, it shows who’s been accepted, their profile, and the option to remove them. In your requests, it shows the option to cancel—it deletes the post. If someone accepts the request, then it shows that information in a green box within the user’s request post. In favorites, it shows all the favorite posts. This can include the posts that the user has favorited, requested ride posts, and any requests that the user has accepted.

Other Functionality

Automatic Deletion

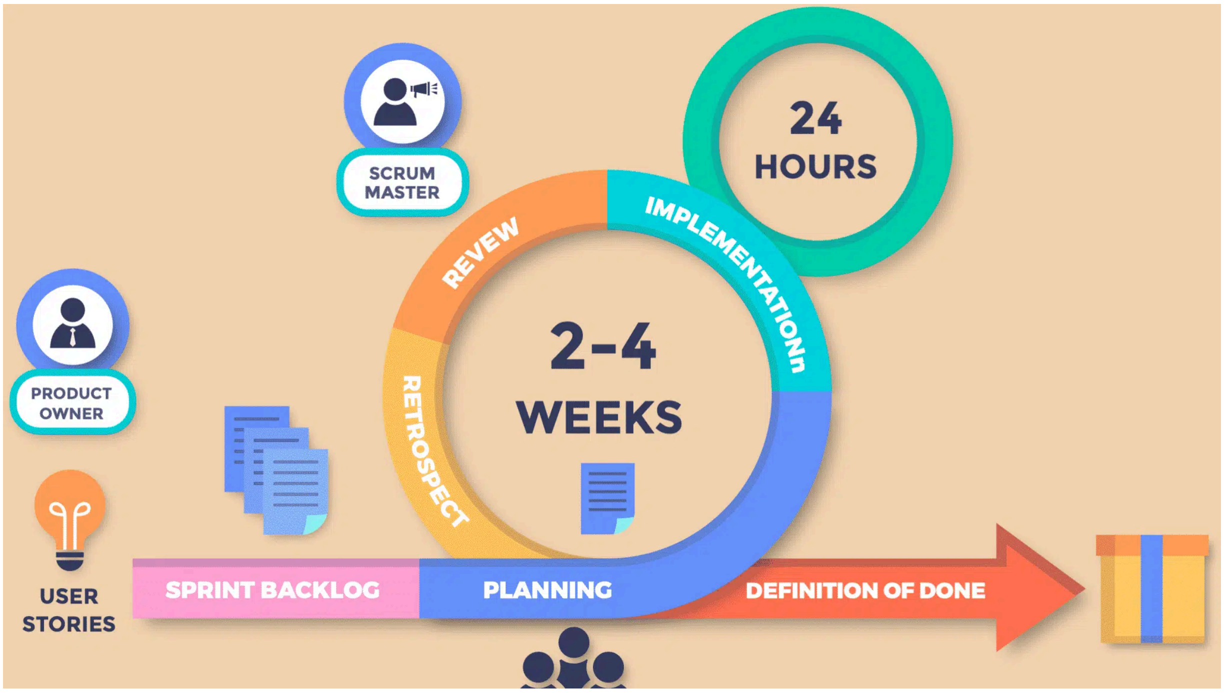
Automatic deletion happens in two cases: deleting posts and deleting users. When the departure date of a post is passed, at midnight, the post will be automatically deleted. Four years after the user has been created, the account will be automatically deleted because it is assumed that the user will have graduated. This is to help save space in the database.

Emails

In short, there are many cases in which the user will get an email. Some of those cases include signing up, deleting an account, creating a request, creating an offering, etc. Depending on the case, multiple users will receive an email with content relevant to the action that caused the email to be sent.

Agile Methodologies

Scrum



(Source)

In this project, we used the Agile methodology known as Scrum. Using Scrum, we divided the project timeline into sprints. Since the project needed to be completed by a deadline, we organized the work into roughly two-week sprints.

Scrum typically involves three key roles: Product Owner, Scrum Master, and Development Team. However, because our team only has four members—all serving as the development team—our use of Scrum isn't fully traditional. Our Product Owner role is filled by a family member of one of our group members, who has professional experience as a product manager.

As a group, we’ve held a few formal meetings (development team + “product owner”), but the group member in contact with the “product owner” has maintained more regular communication as the project progresses.

While there haven’t been many formal meetings, as a team, we’ve strived to have at least two meetings throughout the week. They are usually 30-60 minutes long. The content usually involves: what has each member done, delegating tasks, brainstorming implementation, etc. It was not realistic to have daily scrum meetings, so that is how we’ve adjusted.

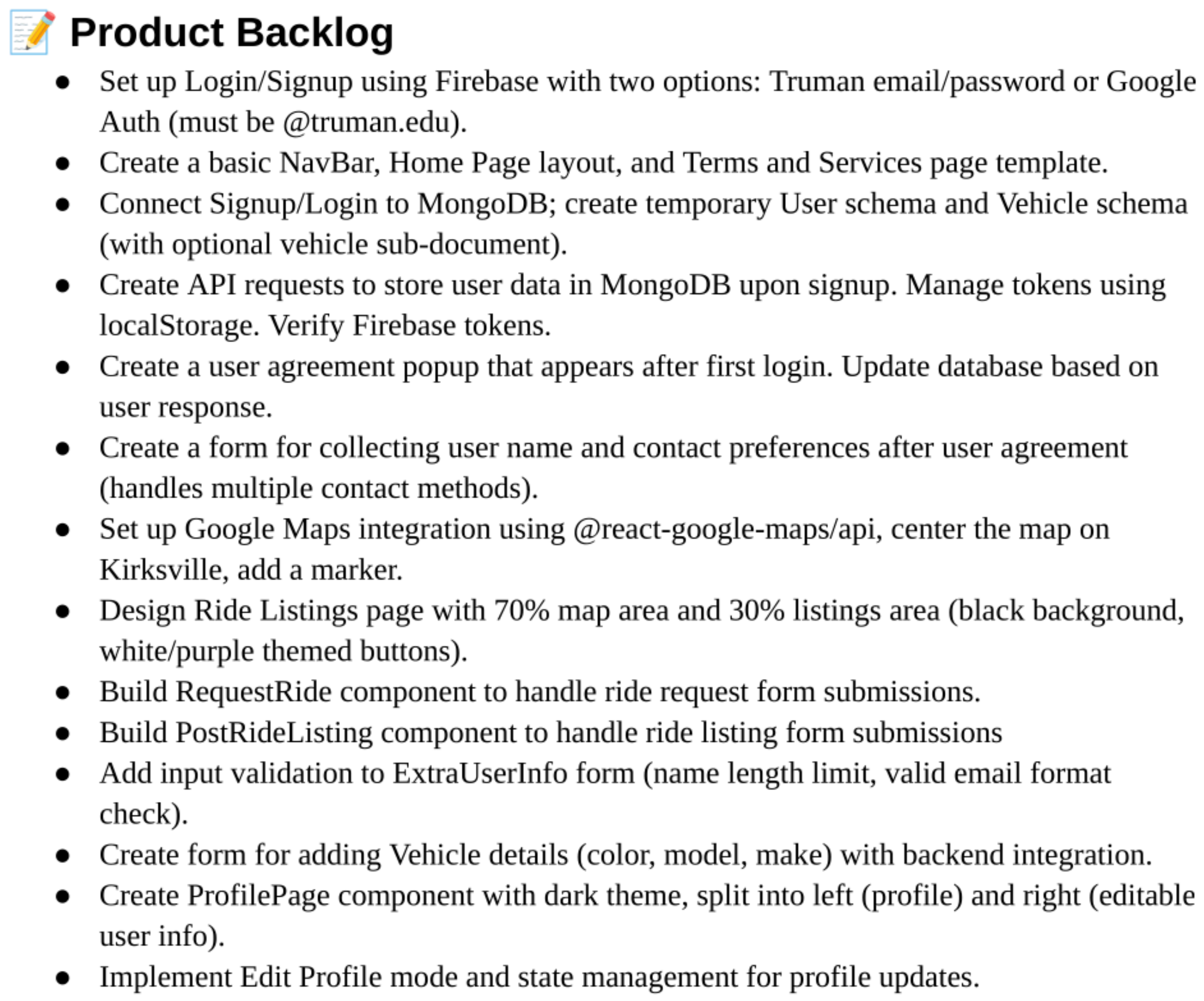
In terms of Sprint Review and Sprint Retrospective, we did not formally do them, due to time constraints. As previously mentioned, one of our group members would regularly keep in contact with the “project owner” so the Sprint Review’s objective to show stakeholders is still met. The Sprint Retrospective was not done at all, since the team lifespan ends at the end of the semester, so going over what was done well and what could be improved wouldn’t be helpful.

Sprints

Sprint 1 (Week 1-2):

The goal of sprint one is to lay a foundation for the basic needs of the web app. Before the start of the first sprint, we've arranged a few meetings to figure out what needs to be done.

The product log of this sprint is:

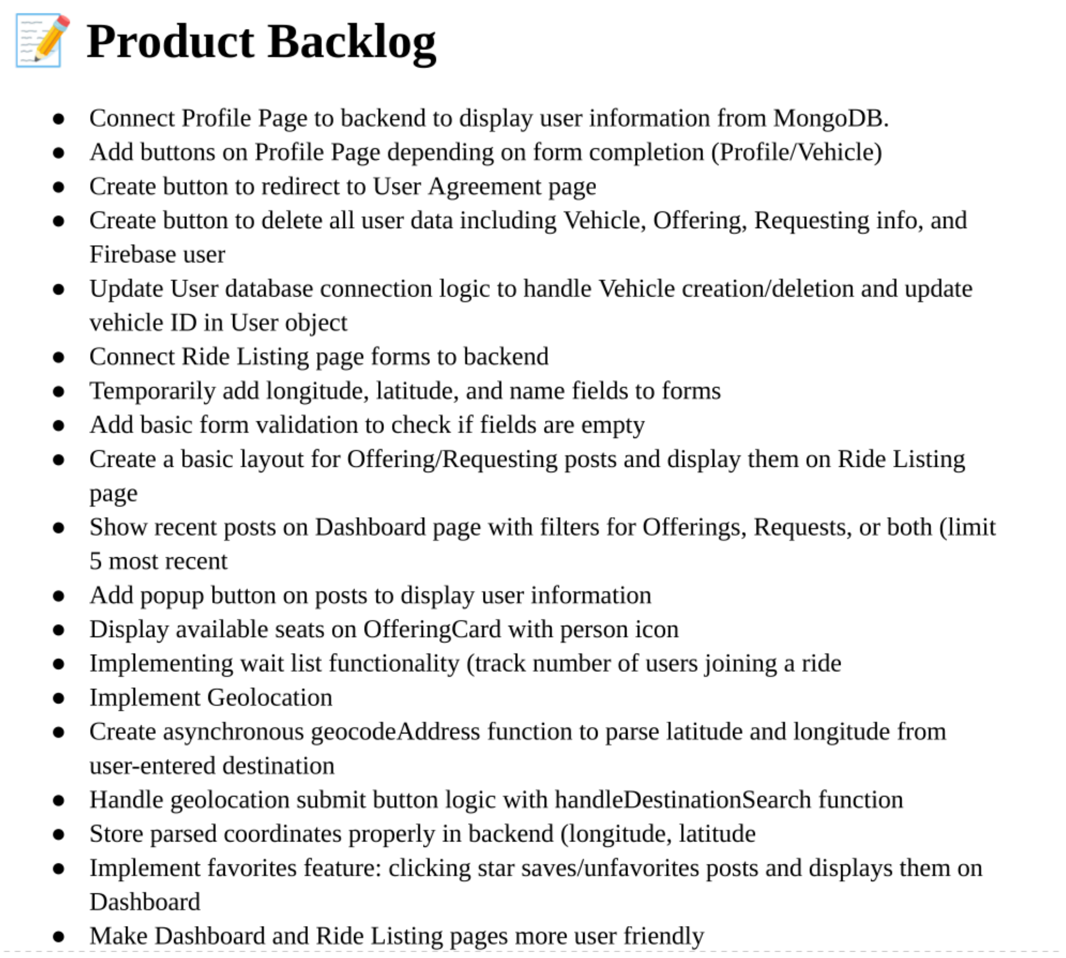


📝 Product Backlog

- Set up Login/Signup using Firebase with two options: Truman email/password or Google Auth (must be @truman.edu).
- Create a basic NavBar, Home Page layout, and Terms and Services page template.
- Connect Signup/Login to MongoDB; create temporary User schema and Vehicle schema (with optional vehicle sub-document).
- Create API requests to store user data in MongoDB upon signup. Manage tokens using localStorage. Verify Firebase tokens.
- Create a user agreement popup that appears after first login. Update database based on user response.
- Create a form for collecting user name and contact preferences after user agreement (handles multiple contact methods).
- Set up Google Maps integration using @react-google-maps/api, center the map on Kirksville, add a marker.
- Design Ride Listings page with 70% map area and 30% listings area (black background, white/purple themed buttons).
- Build RequestRide component to handle ride request form submissions.
- Build PostRideListing component to handle ride listing form submissions
- Add input validation to ExtraUserInfo form (name length limit, valid email format check).
- Create form for adding Vehicle details (color, model, make) with backend integration.
- Create ProfilePage component with dark theme, split into left (profile) and right (editable user info).
- Implement Edit Profile mode and state management for profile updates.

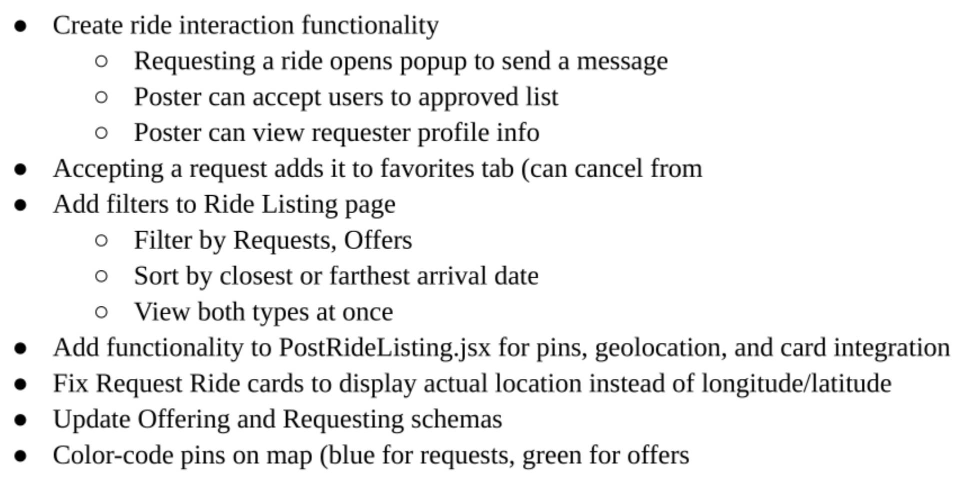
Sprint 2 (Week 3-4):

The goal of sprint 2 is to have the unique features of the web app: posts (offerings and requests) and map functionality.



📝 Product Backlog

- Connect Profile Page to backend to display user information from MongoDB.
- Add buttons on Profile Page depending on form completion (Profile/Vehicle)
- Create button to redirect to User Agreement page
- Create button to delete all user data including Vehicle, Offering, Requesting info, and Firebase user
- Update User database connection logic to handle Vehicle creation/deletion and update vehicle ID in User object
- Connect Ride Listing page forms to backend
- Temporarily add longitude, latitude, and name fields to forms
- Add basic form validation to check if fields are empty
- Create a basic layout for Offering/Requesting posts and display them on Ride Listing page
- Show recent posts on Dashboard page with filters for Offerings, Requests, or both (limit 5 most recent
- Add popup button on posts to display user information
- Display available seats on OfferingCard with person icon
- Implementing wait list functionality (track number of users joining a ride
- Implement Geolocation
- Create asynchronous geocodeAddress function to parse latitude and longitude from user-entered destination
- Handle geolocation submit button logic with handleDestinationSearch function
- Store parsed coordinates properly in backend (longitude, latitude
- Implement favorites feature: clicking star saves/unfavorites posts and displays them on Dashboard
- Make Dashboard and Ride Listing pages more user friendly



- Create ride interaction functionality
  - Requesting a ride opens popup to send a message
  - Poster can accept users to approved list
  - Poster can view requester profile info
- Accepting a request adds it to favorites tab (can cancel from
- Add filters to Ride Listing page
  - Filter by Requests, Offers
  - Sort by closest or farthest arrival date
  - View both types at once
- Add functionality to PostRideListing.jsx for pins, geolocation, and card integration
- Fix Request Ride cards to display actual location instead of longitude/latitude
- Update Offering and Requesting schemas
- Color-code pins on map (blue for requests, green for offers

- Clicking a pin highlights corresponding post on the side panel
- Add "Show All" button to reset side panel view
- Add sorting option by favorites on Ride Listing page.
- Create automatic deletion of posts and users
- Add email notification functionality

Sprint 3 (Week 5-6):

The goal of sprint 3 is to finalize the functionality of the web app. This is when we focused heavily on user testing and modified the existing program to do our best to accommodate all the changes requested/needed. At this point, on top of the needed changes, we also focused on making changes to the mobile view. Throughout the development process, most of the web app has been tested on a small screen; however, there wasn't deep testing. So during this time, more time was spent on testing it for mobile, and making changes to accommodate.

Functional Requirements

The functional requirements define the core features and behaviors that the web application must support. These requirements are expressed as user stories to ensure a focus on real user needs and experiences. They also include a test plan to help make sure they are implemented correctly. The requirements (user stories) are as follows:

Authentication

"As a student, I want to be able to easily log in and, for security purposes, use my Truman email."

Test plan:

- *Attempt to register using a Gmail account (Either through a form or Google's authentication).*
- *The system should either accept the Gmail account or display an error message stating that only Truman emails are allowed.*
- *Verify if the account is successfully created or not.*
- *Check if a confirmation email has been sent to the registered email.*

Request a Ride

"As a student, I want to request a ride to the grocery store."

Test plan:

- *Log in to the system as a student.*
- *Navigate to the "Request a Ride" section.*
- *Enter the destination (in this case, the grocery store)*
- *Enter the additional information, such as date, number of passengers, etc.*
- *Submit the request and verify if it appears in the ride listings.*

- *Ensure other users can see and respond to the request.*

Offer a Ride

"As a student, I want to offer a ride to Columbia, Missouri, on March 10."

Test plan:

- *Log in to the system as a student.*
- *Navigate to the "Offer a Ride" form.*
- *Enter the ride details, including destination and date, etc.*
- *Submit the ride offer and verify if it appears in the ride listings.*
- *Ensure other users can view and request the ride.*

Notification System

"As a student, I want a way to see all interactions with my posts, such as notifications for comments, ride acceptances, or messages."

Test plan:

- *Log in to the system as a student.*
- *Post a ride request or offer.*
- *Have another user interact with the post (accept, or message).*
- *Verify if a notification appears in the student's notification panel.*
- *Click the notification and ensure it redirects to the corresponding interaction.*

Edit Profile

"As a student, I want to be able to change my information whenever I see fit."

Test plan:

- *Log in to the system and navigate to the profile page.*
- *Edit profile details (name, phone number, ride preferences, etc.).*

- *Save the changes and refresh the page to verify updates persist.*
- *Log out and log back in to ensure changes are still saved.*

Responsive Navigation

"As a student, I want to easily navigate the website through a navigation bar or a hamburger menu on mobile devices."

Test plan:

- *Open the website on a desktop and check if the navigation bar is accessible.*
- *Open the website on a mobile device and verify if the hamburger menu appears.*
- *Click through multiple pages to check if the navigation is smooth and intuitive.*

Additional Ride Information

"As a student offering rides, I want to display additional personal notes or preferences along with my ride offer."

Test plan:

- *Navigate to the "Offer a Ride" section.*
- *Enter ride details along with additional information.*
- *Submit and verify if additional details appear in the listing.*
- *Have another user view the listing and confirm they can see the additional details.*

View User Profiles

"As a student searching for rides, I want to be able to view a driver's or rider's general profile information."

Test plan:

- *Click on a ride offer or request.*
- *Verify that a link to the user's profile is available.*

- *Click on the profile link and check for general information (name, preferences, ride history, etc.).*
- *Ensure privacy settings restrict sensitive information if needed.*

Search Filters

“As a student looking for rides, I want to use filters (ex., date) to narrow down my search results.”

Test plan:

- *Navigate to the "Find a Ride" section.*
- *Use filters such as date, number of passengers, and luggage space.*
- *Verify that only matching rides are displayed based on selected filters.*
- *Modify filters and ensure results update dynamically.*
- *Confirm that clearing filters resets the search results.*

Favorites System

“As a student, I want a way to favorite or bookmark ride offers that interest me so I can find them easily later.”

Test plan:

- *Navigate to the "Find a Ride" section.*
- *Click on a ride offer and select the "Favorite" or (star) option.*
- *Verify that the ride is saved to the user’s favorites list.*
- *Navigate to the favorites section and confirm that the saved ride appears.*
- *Remove the ride from favorites and ensure it disappears from the list.*

Non-functional Requirements

In addition to the functional features, the web application should also meet the following non-functional requirements, though not mandatory :

Performance: The application should load pages within a few seconds under normal network conditions.

Scalability: The system should be able to handle an increasing number of users as the Truman State community grows.

Security: Some user data, such as login credentials, must be securely stored.

Mobile Responsiveness: The application should be usable on both desktop and mobile devices.

Usability: The interface should be intuitive and easy to navigate for students of all technical skill levels.

Reliability: The application should maintain high availability, with minimal downtime.

Technology Stack

The technology stack used for this project is based on the MERN stack, which includes MongoDB, Express.js, React, and Node.js. To handle user authentication, Firebase Authentication was integrated to provide a simple and effective way to handle sign-up and login. The frontend was developed using HTML, CSS, JavaScript, and React, with additional styling provided by Bootstrap CSS for faster and more responsive user interface design.

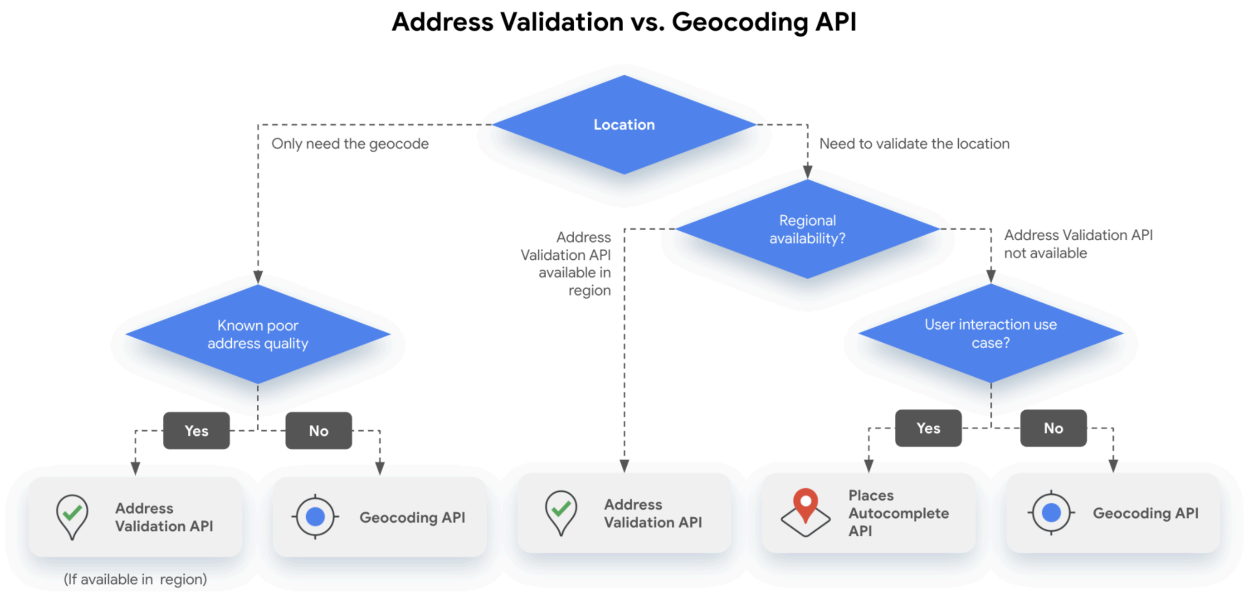
The backend was built using Node.js with Express.js to create a server and handle API requests. For the database, MongoDB Atlas, which is a cloud-hosted service was used to store the different data the project needs.

API Specifications

Google Maps JavaScript API

The Google Maps JavaScript API is used in the Ride Listings page to embed an interactive map directly within the application. Once the user's coordinates are retrieved using the Geolocation API, the JavaScript API is used to render a map centered on that location with a customizable zoom level and user marker. This integration provides full control over the map's behavior and appearance, enabling dynamic updates, marker placement, post handling, and more. The map enhances user experience by providing an interactive and responsive interface for navigation, ride visualization, or local point-of-interest display directly within the web application.

The figure below demonstrates how the Google Maps JavaScript API works in conjunction with adjacent APIs to perform supplemental actions. For example, when location data, regional availability, or user interaction is required, it incorporates results from the Geolocation API.


Address Validation vs. Geocoding API
Location
Only need the geocode
Need to validate the location
Regional availability?
Address Validation API available in region
Address Validation API not available
Known poor address quality
User interaction use case?
Yes
No
Yes
No
Address Validation API
(If available in region)
Geocoding API
Address Validation API
Places Autocomplete API
Geocoding API

[Figure Citation](#)

Google Maps Geolocation API

This project utilizes the Google Maps Geolocation API to estimate the user's current location based on available network information such as WiFi access points, cell towers, and IP addresses. A POST request is sent to the Geolocation API endpoint with either an empty or detailed JSON payload, depending on the available data. In our implementation, we used a basic empty payload ({}) to trigger an automatic location estimation, which returned the user's latitude and longitude. This geolocation data is used to dynamically place a pin on the map at the user's position, enhancing the map's interactivity.

The figure below illustrates the general flow of the Google Maps Geolocation API, from request to response.

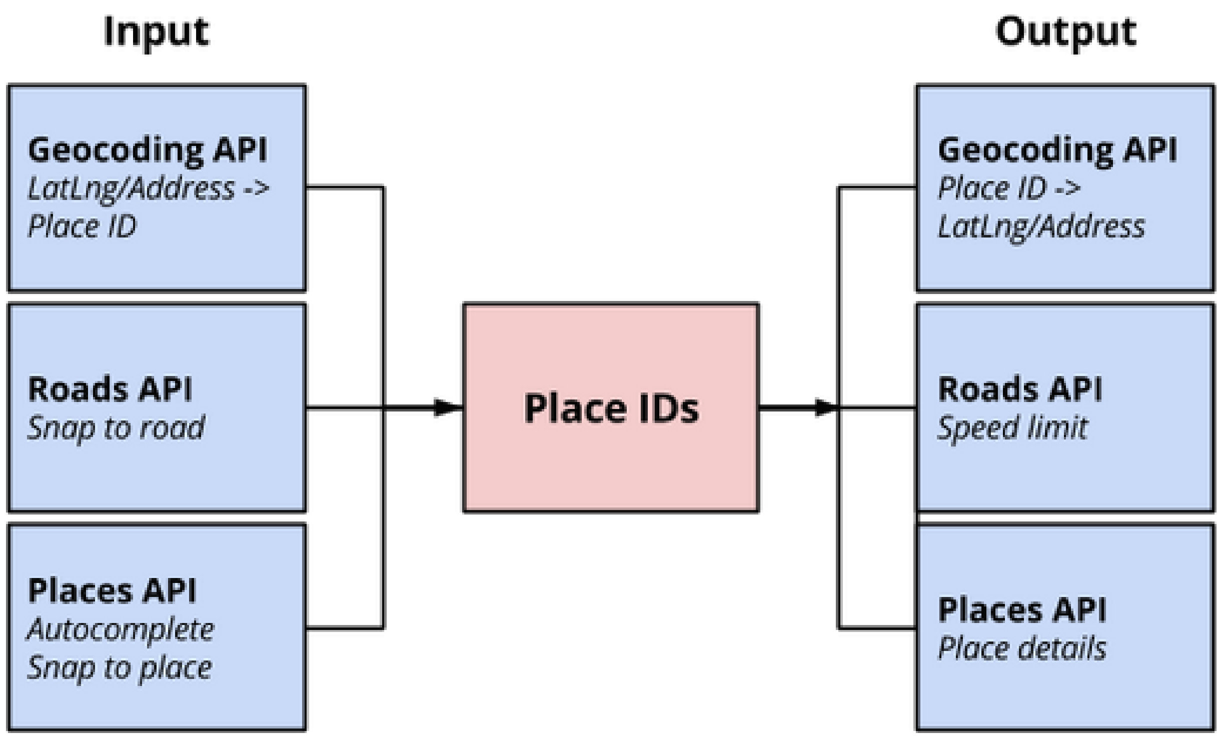
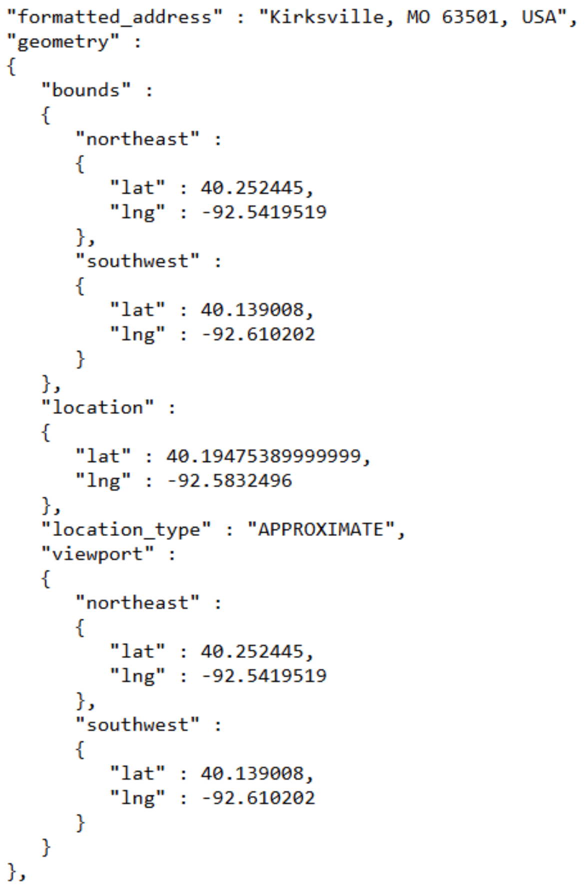


Figure Citation

Sample API Workflow

Suppose a user wants to request a ride to Kirksville, Missouri. The user inputs “Kirksville, Missouri” into the ride request form. Behind the scenes, the following API workflow occurs:

1. The Google Maps Geolocation API is called, returning a JSON file containing location-based results.
2. The program parses this file by accessing the geometry object, extracting the location element, and retrieving the associated lat and lng values.
3. These coordinates are used to place a marker on the interactive map via the Google Maps JavaScript API.
4. The same data is also used to update the user’s Dashboard for reference.



```
"formatted_address" : "Kirksville, MO 63501, USA",
"geometry" :
{
   "bounds" :
   {
      "northeast" :
      {
         "lat" : 40.252445,
         "lng" : -92.5419519
      },
      "southwest" :
      {
         "lat" : 40.139008,
         "lng" : -92.610202
      }
   },
   "location" :
   {
      "lat" : 40.19475389999999,
      "lng" : -92.5832496
   },
   "location_type" : "APPROXIMATE",
   "viewport" :
   {
      "northeast" :
      {
         "lat" : 40.252445,
         "lng" : -92.5419519
      },
      "southwest" :
      {
         "lat" : 40.139008,
         "lng" : -92.610202
      }
   }
},
```

API Security

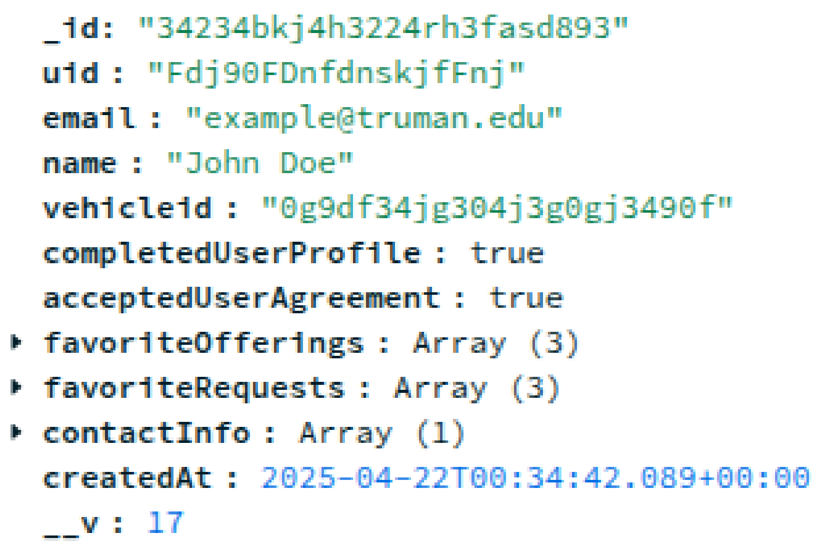
API keys used in this application are securely stored in environment variables and are never exposed publicly. Additionally, access to the enabled APIs is tightly controlled—usage is restricted to specific services and domains associated with Truman RideShare. This setup helps prevent unauthorized access and enhances the overall security of our application infrastructure.

Database

MongoDB

This web application uses MongoDB to store data about users and posts. It connects to MongoDB Atlas using an admin URI, which is securely stored in an environment file to protect sensitive information. Since the web app relies on an online service, there may be occasional instances when the database is temporarily unavailable. However, for this system, it remains a reliable solution. MongoDB Atlas offers a maximum storage limit of 512MB on the free tier. While this may seem limited, each document consumes very little space, so it should be sufficient for a long period. Additionally, the system includes automatic deletion functionality to remove posts and users after a certain amount of time, helping manage storage efficiently. There are four schemas, namely: user, vehicle, request, and offering.

User Schema:



```
  _id: "34234bkj4h3224rh3fasd893"
  uid : "Fdj90FDnfdnskjfFnj"
  email : "example@truman.edu"
  name : "John Doe"
  vehicleid : "0g9df34jg304j3g0gj3490f"
  completedUserProfile : true
  acceptedUserAgreement : true
▸ favoriteOfferings : Array (3)
▸ favoriteRequests : Array (3)
▸ contactInfo : Array (1)
  createdAt : 2025-04-22T00:34:42.089+00:00
  __v : 17
```

The picture above shows a sample user document containing 12 fields.

The first field is *_id*, which is an ObjectId automatically created by MongoDB to uniquely identify each document.

The second field is *uid*, a string provided by Firebase that serves as a unique identifier for each user and acts as a foreign key in other schemas.

The third field is *email*, which stores the user's email address, also retrieved from Firebase.

The fourth field is *name*, which stores the user's name.

The fifth field is *vehicleid*, a foreign key that references an ObjectId from the vehicle schema.

The sixth field is *completedUserProfile*, which stores a boolean indicating whether the user has completed their profile.

The seventh field is *acceptedUserAgreement*, another boolean indicating if the user has agreed to the terms of service.

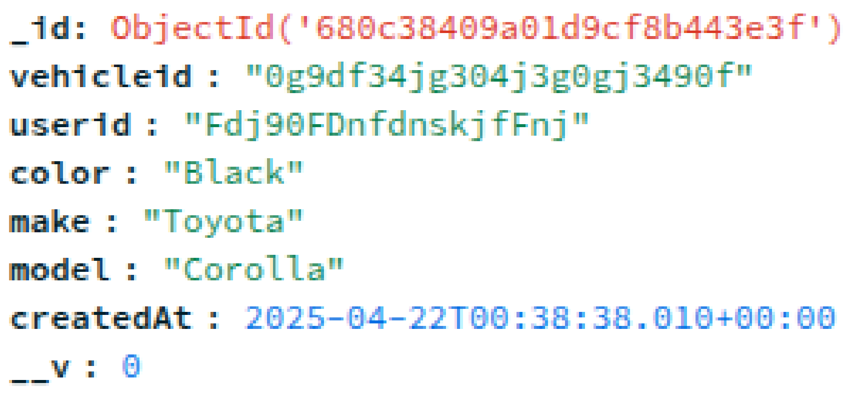
The eighth and ninth fields, *favoriteOfferings and favoriteRequests*, are both arrays that store ObjectIds of the user's favorite posts.

The tenth field is *contactInfo*, an array that contains objects. Each object includes three fields: *type*, which stores the type of contact information (e.g., "phone," "email," "social media"); *value*, which stores the actual contact detail (like a phone number or a social media username); and an internal *_id* automatically generated for each contact object.

The eleventh field is *createdAt*, which records the date and time the document was created.

The final field is __v, automatically added by MongoDB for version control; this number increases each time the document is updated.

Vehicle Schema:



```
_id: ObjectId('680c38409a01d9cf8b443e3f')
vehicleid : "0g9df34jg304j3g0gj3490f"
userid : "Fdj90FDnfdnskjfFnj"
color : "Black"
make : "Toyota"
model : "Corolla"
createdAt : 2025-04-22T00:38:38.010+00:00
__v : 0
```

The picture above shows a sample vehicle document that contains 8 fields.

The first field is *_id*, which is an ObjectId automatically created by MongoDB.

The second field is *vehicleid*, which stores the same _id to act as a primary key and is used as a foreign key in the users schema.

The third field is *userid*, a string that is referenced from the user schema.

The fourth field is *color*, a string that stores the color of the vehicle.

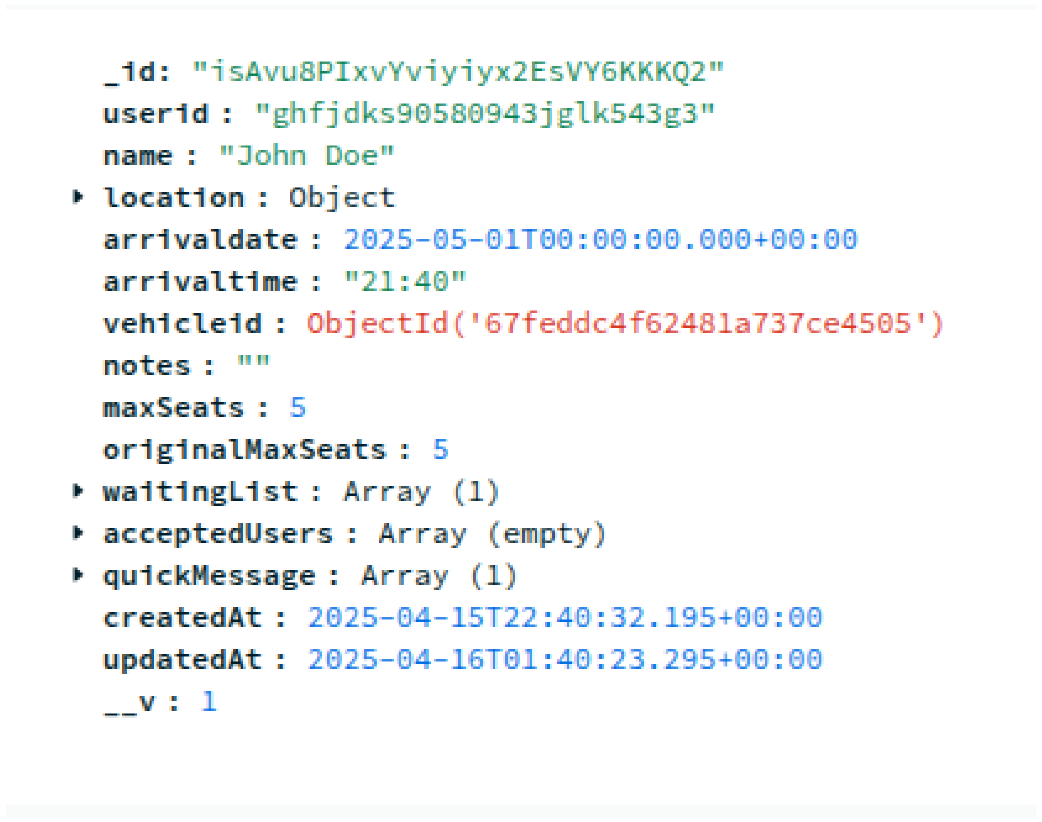
The fifth field is *make*, a string that stores the make (brand) of the vehicle.

The sixth field is *model*, a string that stores the model of the vehicle.

The seventh field is *createdAt*, which stores the date when the document was created.

The final field is *__v,* automatically added by Mongoose for version control; this number increments each time the document is updated.

Offering Schema:



```
  _id: "isAvu8PIxvYviyiyx2EsVY6KKKQ2"
  userid : "ghfjdks90580943jglk543g3"
  name : "John Doe"
▸ location : Object
  arrivaldate : 2025-05-01T00:00:00.000+00:00
  arrivaltime : "21:40"
  vehicleid : ObjectId('67feddc4f62481a737ce4505')
  notes : ""
  maxSeats : 5
  originalMaxSeats : 5
▸ waitingList : Array (1)
▸ acceptedUsers : Array (empty)
▸ quickMessage : Array (1)
  createdAt : 2025-04-15T22:40:32.195+00:00
  updatedAt : 2025-04-16T01:40:23.295+00:00
  __v : 1
```

The picture above shows a sample offering document that contains 16 fields. Each field is described below.

The first field is *_id*, which is an ObjectId automatically created by MongoDB to uniquely identify each offering.

The second field is *userid*, a string that is referenced from the user schema.

The third field is *name*, a string that stores the name of the person who posted the offering.

The fourth field is *location*, an object that contains the geolocation for where the offering is based. It stores the *coordinates* (longitude and latitude) and the name of the *formattedAddress*.

The fifth field is *arrivaldate*, which stores the date the offering is scheduled to occur.

The sixth field is *arrivaltime*, which stores the time the offering is scheduled to occur, saved as a string.

The seventh field is *vehicleid*, an ObjectId that links to the user's vehicle in the vehicle schema.

The eighth field is *notes*, a string where the offering creator can include any additional information.

The ninth field is *maxSeats*, a number representing how many seats are currently available.

The tenth field is *originalMaxSeats*, which stores the original number of seats when the offering was first created.

The eleventh field is *waitingList*, an array that stores user IDs of users waiting for a seat.

The twelfth field is *acceptedUsers*, an array that stores user IDs of users who have been accepted into the ride.

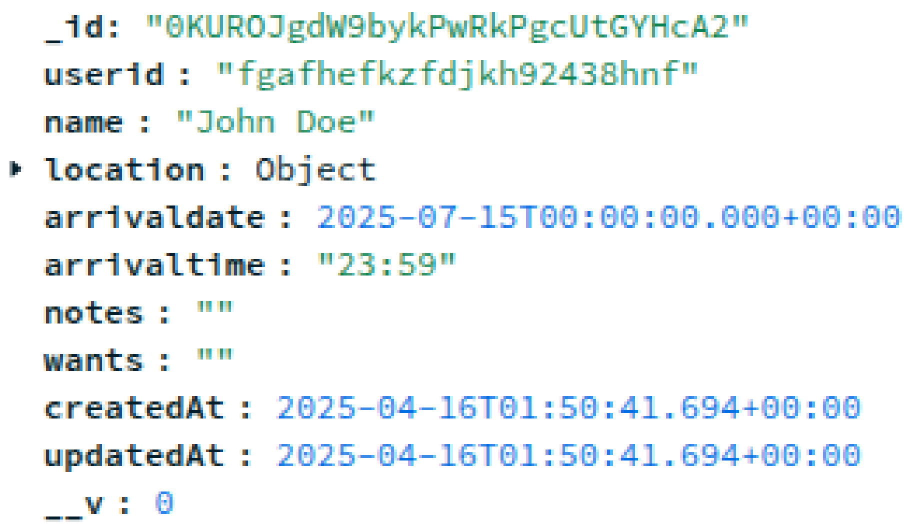
The thirteenth field is *quickMessage*, an array of objects containing messages related to the offering. Each message includes the message text, the sender's user ID, ObjectID, and the creation time.

The fourteenth field is *createdAt*, which records the date and time when the offering document was first created.

The fifteenth field is *updatedAt*, which records the most recent date and time when the offering document was updated.

The final field is *__v*, a version key automatically added by MongoDB for version control, increasing each time the document is updated.

Request Schema:



```
  _id: "0KUROJgdW9bykPwRkPgcUtGYHcA2"
  userid : "fgafhefkzfdjkh92438hnf"
  name : "John Doe"
▸ location : Object
  arrivaldate : 2025-07-15T00:00:00.000+00:00
  arrivaltime : "23:59"
  notes : ""
  wants : ""
  createdAt : 2025-04-16T01:50:41.694+00:00
  updatedAt : 2025-04-16T01:50:41.694+00:00
  __v : 0
```

The picture above shows a sample request document that contains 11 fields.

The first field is *_id*, which is an ObjectId automatically created by MongoDB to uniquely identify each request.

The second field is *userid*, a string that is referenced from the user schema.

The third field is *name*, a string that stores the name associated with the request.

The fourth field is *location*, an object that contains the geolocation for where the offering is based. It stores the *coordinates* (longitude and latitude) and the name of the *formattedAddress*.

The fifth field is *arrivaldate*, which stores the date the user wants to be picked up.

The sixth field is *arrivaltime*, a string that stores the time the user wants to be picked up.

The seventh field is *notes*, a string for any additional information the user wants to provide about the request.

The eighth field is *wants*, a string where users can specify any specific preferences or needs related to the ride.

The ninth field is *createdAt*, which records the date and time when the request document was first created.

The tenth field is *updatedAt*, which records the date and time when the document was last updated.

The eleventh field is __*v*, a version key automatically added by MongoDB for version control, which increases with each document update.

The final field is *userAccepted*, a string that stores the userid of the person who accepted the request.

Deployment & Environment Setup

Local Setup

Prerequisites:

- Node.js (v16+ recommended)
- npm or yarn package manager
- MongoDB (local installation or cloud-based, e.g., Atlas)
- Firebase Admin credentials (for authentication)
- Access to the Google Maps API key

Frontend Setup:

1. Navigate to the client directory.
2. Install dependencies:
   - npm install
3. Start the development server:
   - npm run dev
4. The application will connect to the backend at http://localhost:5000 (or a custom port specified in vite.config.js).

Backend Setup:

1. Navigate to the server directory.
2. Install dependencies:

   npm install

3. Create a .env file with necessary environment variables (see Environment Variables section).
4. Start the backend server:

   npm run dev
5. The backend API server runs on http://localhost:5000 but is configured locally on Vite to run at http://localhost:5173/ or a configured alternative port.

MongoDB Setup:

- If using MongoDB locally, ensure the MongoDB daemon is running.
- If using MongoDB Atlas, configure the .env file with the appropriate connection string.

Production Deployment

Our production deployment here is concerned with how our project will be accessible to the wider Truman campus population.

Frontend and Backend Deployment:

Hosted on sand.truman.edu with the top-level domain (https://rideshare.truman.edu)

pointing to sand.truman.edu, which will automatically redirect the user to our specific project (since multiple applications are running on the SAND virtual machines).

Since the client and server are running on SAND's virtual machines, the setup and build commands are the same within the Client and Server folders of our project, but within the purview of SAND itself:

`npm run dev` (in both client and server folders)

Deployment process:

- Retrieve the latest project version from our GIT source control platform.

- Configure the production domain and HTTPS (SSL) settings via Apache scripts.

Database Hosting:

- MongoDB Atlas is used for production-grade database hosting.
- Network Access is restricted to trusted IPs and backend server IPs only.
- MongoDB user permissions are scoped appropriately (e.g., no wildcard administrative access).

Firebase Authentication**:**

- Firebase Admin SDK is used for authentication.
- Private keys and service accounts are securely stored via environment variables.

Environment Variables

Our environment variables are used extensively to configure the application without hard-coding sensitive information into the codebase.

Typical Environment Variables:

Variable Name	**Purpose**
MONGO_URI	MongoDB connection string (Atlas or local)
VITE_API_KEY	Held our Vite API key
FIREBASE_ADMINSDK.JSON (Is not an env variable, but Firebase's env variables needed to be put in this file, and not a .env)	Held all information regarding Firebase. Included project_id, private_key, token_uri, auth_uri, etc.
VITE_GOOGLE_MAPS_API_KEY	API key for Maps integration
EMAIL_USER	Stored the username for our automated email notifications
EMAIL_PASS	The Nodecron email sender uses Google's 'app password' function, so the app password for our site is stored here

Our Practices for Environment Variables:

- .env files are **excluded from Git** via .gitignore to prevent leakage.
- Production environment variables are configured manually or through CI/CD pipelines (never hard-coded).
- Sensitive keys (such as Firebase's PRIVATE_KEY) must be carefully escaped to handle newline characters.

Testing Plan

Testing Process

Our testing process followed a comprehensive strategy that included both alpha and beta testing phases. Alpha testing was conducted internally by the development team in a controlled environment. During this phase, we focused on identifying and fixing functional bugs, UI inconsistencies, and integration issues across core features such as Firebase authentication, ride posting, Google Maps API interactions, and database transactions. This internal testing ensured that individual components were working correctly before moving on to more comprehensive full-stack evaluations.

Once alpha testing was complete, we initiated beta testing by recruiting several users to simulate real-world interaction with the application. These users were intentionally chosen to represent a mix of technical and non-technical backgrounds, helping us assess both functional capabilities and overall usability. Test cases during this phase included verifying that users could not accept their ride requests, confirming email notifications after ride postings, testing routing from the Terms of Service page, checking login/logout behavior across components, and ensuring rides could not be posted without a registered vehicle. Feedback from beta testers revealed several UX and logic issues, many of which were quickly addressed and re-tested. Some suggestions that fell outside the project's immediate scope were documented for potential future updates.

This dual-layered testing approach enabled us to confidently deliver a more reliable and user-friendly application. The table on the next page summarizes the key issues discovered during beta testing and how they were resolved.

User	CS History?	Issue(s) Discovered	Issue(s) Resolved
User one	No	Found it unclear how to save and add contact information. Found each page unclear	Merged the two buttons, causing confusion, into one. Added “i” icons to assist users with each page.
User two	Yes	The path route Homepage → TOS → Return to Profile resulted in an error. Requested a link to establish a vehicle after the error is thrown in Offer a Ride.	Resolved this error. Now the path is Homepage → TOS → Homepage. Added a link inside the error to add a vehicle.
User three	Yes	Requested to limit available seats (was infinite) Found that you could accept your request when you logically shouldn’t be able to. Requested a Time field when offering and requesting rides. Suggested extending the one-month in advance requirement to three months. Asked for a way to edit a post.	Limited the number of seats available to 15. Ensured you aren’t able to accept your request. Added a time field and updated the database accordingly. Extended the one-month in advance to three months when posting a ride. Outside of our project scope.
User four	Yes	Requested background images to enhance the UI Wanted more space for the cards.	Outside of our project scope. We decided this wasn’t necessary for the developers.
User five	Yes	Found a bug when a user selects “Do it later”, regarding the profile, they still have the ability to request to join a ride, and post both ride options.	Removed the ability to “Do it later” – regarding the Profile
User six	Yes	Found that if you request a ride, it shows up in the Favorites tab. Subsequently, the Dashboard → Favorites tab says “No favorite offerings”, as well as the favorited post.	This was not a problem in the end.
User seven	Yes	No major errors or bugs were found	

Testing Types

Throughout the development lifecycle, we employed utilized types of testing to ensure the functionality, stability, and usability of our application. Unit testing was performed on backend functions such as authentication, ride posting, and database interactions to verify individual pieces of logic. Integration testing followed, where we tested the interaction between modules like Firebase Authentication, MongoDB, and the Google Maps API to ensure proper communication between components. System testing was conducted on the main deployment branch to evaluate the application as a whole in its production-like environment. Additionally, we carried out alpha testing internally within the development team, simulating user behavior to catch critical issues early. This was followed by beta testing, which involved external users operating the system under real-world conditions and providing valuable feedback on both functionality and user experience.

Test Case Examples

Several specific test scenarios were designed and implemented to validate key functionalities. These included confirming that users could not accept their own ride requests, ensuring email notifications were sent in appropriate situations, and verifying that navigation between pages, such as from the Terms of Service to the user profile, functioned correctly. We also tested state persistence across the application, such as consistent login/logout behavior, and enforced constraints such as preventing ride posting without a registered vehicle. These test cases helped us detect both edge-case failures and day-to-day user flow issues.

CI/CD Integration

Our team utilized GitHub for version control and continuous integration/continuous deployment (CI/CD). Development was structured around seven dedicated branches, each with a specific focus, such as map functionality, email automation, and Firebase/database integration. The development team regularly pushed changes to their respective branches and created pull requests to merge into the main branch after review and testing. This workflow ensured that integration was smooth and allowed the team to detect and resolve merge conflicts or deployment issues early. The main branch served as the central integration point, reflecting the most stable and complete version of the application, ready for deployment.

Bug Tracking and Reporting Process

Bug tracking was handled collaboratively through GitHub Issues and team communication channels. During both alpha and beta testing, any identified bugs were documented in detail, including steps to reproduce, screenshots when necessary, and initial diagnoses. Developers were assigned to each issue and were responsible for resolving and pushing updates. Once fixes were implemented, regression testing was conducted to ensure the resolution did not introduce new bugs. The iterative process of reporting, fixing, and retesting allowed the development team to steadily improve the application's reliability and user experience throughout development.

GitHub Repository Instructions

Our GitHub Repository consists of seven branches. Below provides a general overview of each branch and its purpose.

1. **Map-functionality**
    a. This branch contains the true functionality for the map component of our web application. Inside this branch, you will find the code that creates the interactive functionality, for example, when a user selects a pin, the corresponding ride offer or request post will appear on the right-hand side of the screen. Other functionalities would include pin customization (i.e., color, size, and shape), ride request, and offer card display on the right-hand side of the screen, interactive favoriting, and filtering.
2. **Google-maps-integration**
    a. This branch contains the basic functionality of the Google Maps component of our web application. This branch was the original when integrating the Google Maps API into our web application and beginning the ride-listings page. The primary purpose of this branch was to get the basic functionality of the Google Maps API up and running.
3. **JakeEmail**
    a. This branch contains the functionality behind our automated email system. This automated email system utilizes nodemailer–an email-powered framework for backend development. Having this branch was essential in getting communication from the application to the user. A few examples of when a user would receive an

email would include a user signing up for the first time, a user posting a request or offer listing, accepting a user into their ride, and accepting a ride request, among others.

4. **JakeTimer**
    a. This branch consists of development that worked on automatically deleting a user's account functionality. This functionality, simply stated, is the idea that a user's account will be automatically deleted after four years past registration. This functionality allows us to maintain space in our database for future users.
5. **Database-+-Firebase-Authentication**
    a. This branch was a major part of the web application development. It includes core functionalities such as user authentication with Firebase and data management using MongoDB. Features implemented in this branch include login authentication, user creation, data storage, and related backend operations.
6. **JakeBackend**
    a. This branch was used as a way to initialize the database setup during our brainstorming phase. We used this branch to test different schemas within the purview of MongoDB to see how different information would be displayed and stored. This branch was then subsequently used as a backup during server-side development to safeguard against major errors or disruptions. It served as a stable fallback while implementing backend functionality.
7. **Main**
    a. This branch serves as the central integration point for our web application. It merges the work from all other branches, bringing together map functionality,

email automation, authentication, database management, and more. The Main branch represents the final and most complete version of our application, combining all components into a cohesive, fully functional system ready for deployment and use.

The repository also retains a **README** file, detailing specific technologies used, integrated services, documentation, technical highlights, and key features, among others.

Conclusion

The Truman RideShare project was built to address a real and pressing need within the Truman State University community—providing a reliable, user-friendly platform for students, faculty, and staff to organize transportation in and around Kirksville. Throughout development, our team focused on delivering a practical and scalable solution by leveraging modern technologies such as the MERN stack, Firebase Authentication, and Google Maps APIs.

By following Agile principles and adapting Scrum methodologies to fit the scope and timeline of our project, we were able to efficiently prioritize features, respond to feedback, and continuously improve the platform. Comprehensive testing, including development and alpha testing phases, helped us identify and resolve usability issues and functional bugs, ensuring the application is interactive, responsive, and stable across a variety of devices and use cases.

Key features such as ride requesting and offering, interactive map integration, user profile management, and automated account and post management were designed with the end-user in mind, promoting ease of use and reliability. Best practices for data management were incorporated to safeguard user information and maintain a sustainable system architecture.

Overall, Truman RideShare represents a significant step forward in creating a centralized, community-driven transportation solution for Truman State University. We are proud of the progress made and look forward to using this product in the community.