Instituto Tecnológico y de Estudios Superiores de Monterrey.

Campus Puebla.

Laboratory of Web Application Development

Final Project:
Safe Ride

Team members:

Alejandro Jordan Mayorga Constantino *A01172971*
Daniel Saúl Chávez García *A01324491*
Jesús Enrique Librado Polanco *A01551367*

Professor:

Luciano García Bañuelos

General Description

Our project consists of a web application called “Safe Ride”. It consists on a carpooling system available only to students from ITESM campus Puebla. It allows students to offer carpooled trips, using own vehicles, to other students going in similar directions out of the university.

Most current similar solutions (e. g., Uber, DiDi, etc.), specialize in individual, “non-cooperative” transportation; i. e., the “taxi” model. In contrast to existing solutions, this carpooling based system attempts to offer the following advantages:

- Decrease gas pollution made by vehicles.
- Decrease in traffic jams during peak hours around the university.
- Accessibility through a cheaper service. The cost of the trip consists of a simple commission for the driver, plus estimated gas expenses. The service remains cheap, and the motivation for the drivers remains relatively altruistic, instead of incentivizing as many (polluting, traffic jamming) trips as possible.
- Last but not least, it’s a much safer option for both passengers and drivers, as they both know everyone is a fellow student member of the campus.

Solution elements

Architecture

Basic structure

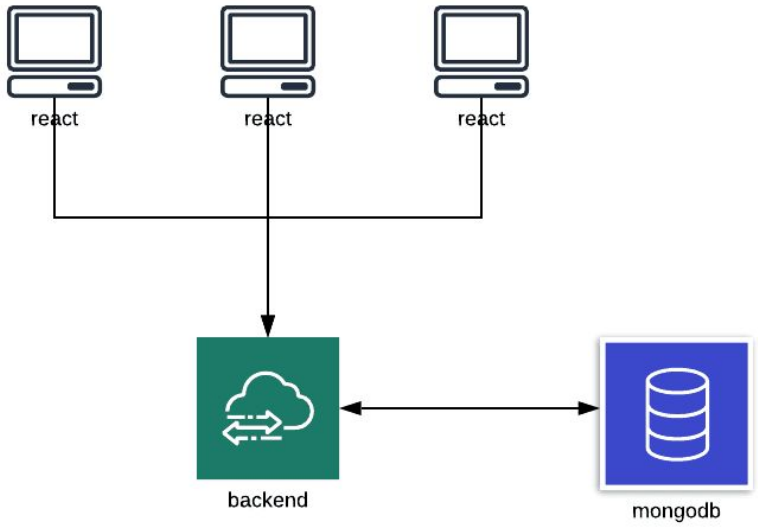

Figure 1: Diagram of the basic structure of the architecture

As usual for web applications, the project is divided in two main components: the backend and the frontend.

The backend was built with the Express JavaScript framework, using the conventions of an MVC architecture. It was decided that all of the functionality should be exposed through a REST API, based on HTTP requests and responses. Requests send information either through URL or standard form-encoded parameters, and answers include the requested information (or notice of errors) in JSON. This design allows future extensibility and flexibility, e. g., if we were to make an mobile app, or migrate the frontend to another technology. The database is implemented with MongoDB, making use of its seamless integration with Express through the module Mongoose and making use of its geospatial capabilities.

Express proved to be a powerful, quick, and easy to use framework. Sadly, Few of us had any experience at all in JavaScript beyond simple DOM manipulation in handmade webpages, let alone big-scale projects in the asynchronous model of NodeJS. We felt a lot of time was wasted simply searching for gotchas and obscure error messages. These are all great technologies for this use-case, but we now realise one should not underestimate the difficulties of learning entire new programming paradigms, especially when balancing with other projects.

The frontend used directly by the clients, was designed with the React framework. At the moment, two versions of the frontend are used: one for normal students browsing trips to join, and another for drivers (previously registered as students), to manage their trips, including accepting or rejecting new students for their travels in an asynchronous way, using the Pusher third-party service.

Many of the points made with JavaScript and NodeJS apply equally to React. We found that simply trying to use code snippets when appropriate, without understanding the React ecosystem in depth, led to costly errors down the road. However, a deep understanding of React could takes entire courses, if online resources are to be believed. Again, we conclude it is an excellent technology for this use-case, definitely beating handmade HTML or even templating (e. g., EJS), but should be approached with time and care.

Other third party service used extensively was Mapbox. It provides forward and reverse geocoding, autocompletion for textboxes asking for geographical data, and map rendering.

One last mentioned should be made about Storybook, a system for testing of UI components, compatible with React. It simplified much of the frontend visual development, an area most of us have little to no experience.

Database model

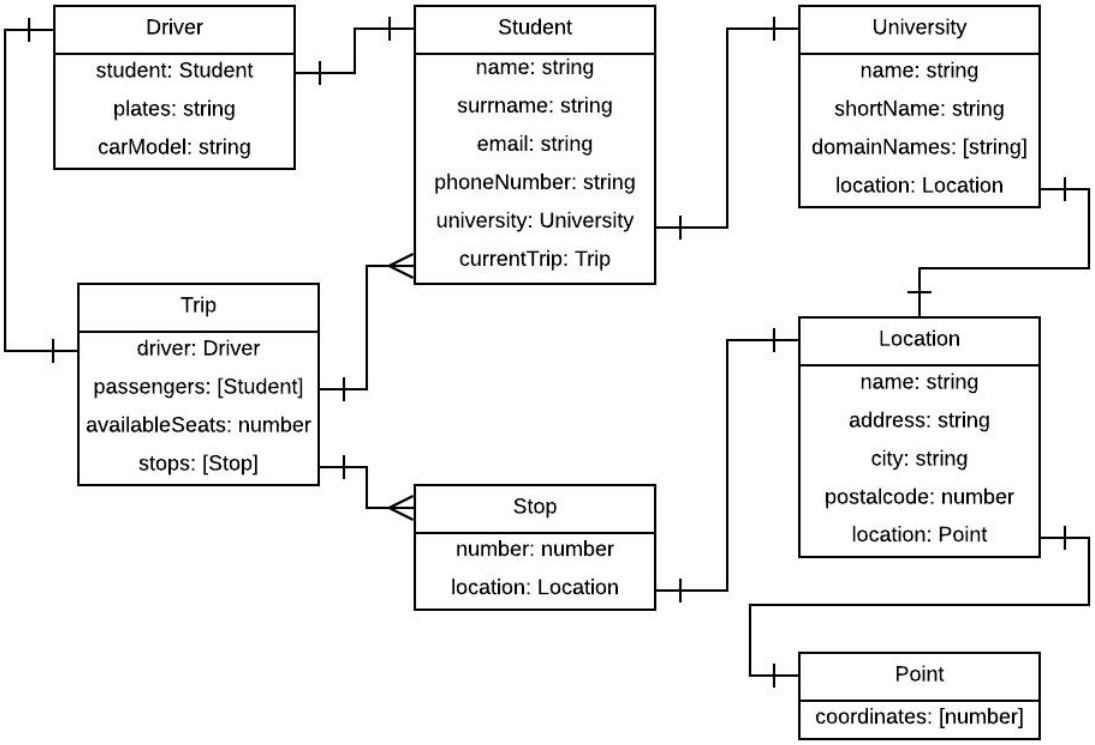

Figure 2: Diagram of the database

As mentioned, the database is implemented in the MongoDB document-oriented database.

The decision to use MongoDB was a difficult one. What ultimately won us over was, mainly, its geospatial capabilities. That being said, this was the first time many of us used this technology and paradigm in a real project.. Many design decisions made at the beginning were suboptimal, and many redesigns and implementations needed to be made after the fact. We especially regretted the lack of transactions as used in relational alternatives, which we weren't aware at the time and made us scratch lots of functionality at the last second. Even with all these problems, we believe this project has given us a deeper understanding of when NoSQL, and MongoDB specifically, is more appropriate to a given task. We would probably still choose it for a real project, with more experience and proper training under our belts.

The current design depends on seven identities, implemented with Mongoose models: Student, Driver, University, Location, Trip, Stop and Point. Students represent a student registered in the service. Once a student exists, the system can register it as a Driver; this ensures the requirements of ensuring all users are students. Location is a geographical location, including geospatial data contained in Point. This being a service mostly directed at university students, we found it useful to add an special University sublocation; right now only one exists (ITESM Campus Puebla), but with the possibility of expansion, should other

universities desired to join. Trip is a trip scheduled by a Driver, which requires passing through advertised Stops.

Components used

- Backend:
    A. Folders are separated by functionality
        a. **Controllers**: they fulfill the requests that the routes receive, they are javascript exported classes. They are separated by entities
        b. **Helpers**: JavaScript functions. They are separated by entities which basically fulfill requests made directly to the database, but the purpose of them is to work as any kind of external async or sync function.
        c. **Models**: entities models created with Mongoose.
        d. **Routes**: routes balancers separated by entities.
    B. Scalability
        a. REST API design guidelines. It allows for a standard, logical organization of backend functionality and easy integration of new features.
        b. In code, functionality is spreaded by folders in order to improve coding time and feature implementation.
        c. The problem with the structure and architecture so far is that as entities increase, there will be a big amount of files in the same folder, making it harder to implement a new feature.
    C. Improvements
        a. Some asynchronous operations should be handled in the background while the backend responds with a temporary *loading* statement to bring proper feedback to the user.
        b. Implement transactions to avoid model declaration before the existence of an entity is proven.
    D. Mongoose and MongoDB
        a. Mongoose is used as an ORM to connect with a locally hosted instance of MongoDB.
        b. Mongoose uses asynchronous functions to make requests to the database.
        c. As mentioned, poor to no support for transactions as used by traditional relational models.
- Frontend:
    A. Components are stored in *sr/components* where they are just listed with their filename representing a reusable component.
    B. Storybook is used to program and test components individually while working on them independently inside the same workspace.
    C. We were going to use React Routers but the concepts were a little bit confusing for the level we have so far. Specially with route params.
    D. We used React component hooks instead of classes to reduce code and take advantage of the improvements that hooks have over normal JSX classes. Specifically with params and life cycle functions like *useState* and *useEffect.*
    E. To make the designs and USX, we used React material which made tables, maps, cards and textfields a lot easier to handle, even when it comes to reactive autocomplete inputs.

F. We used Mapbox API to get the forward and reverse geocoding and autocomplete of places based in the name and coordinates.

Interaction types

- Asynchronous
    - All requests made to the backend through the API are responded with a promise already implemented by Express.
    - Requests made by the backend to MongoDB through Mongoose respond with a promise as well.
    - In order to avoid errors during asynchronous operations, *try and catch* blocks are implemented to respond with an error message if something is not found.
    - Pusher is used to accept and reject students from joining a trip.

Resources of the software development process

Repository:
https://bitbucket.org/web_projects_2020/final_project_labweb20/src/master/

Trello Board:
https://bitbucket.org/web_projects_2020/final_project_labweb20/addon/trello/trello-board

Personal Conclusions

Jesus

I personally enjoyed the experience of developing the app, specially the backend part, which I don't generally focus on, but I definitely liked thinking about the design and improvements that could be done so that the frontend developers find it easier to make useful requests which is the main problem of REST API's: get more or less data than what it is needed. When I started with the implementation design, the first thing that came to my mind was scalability and how to make it easier for anybody to add a new feature or fix a minor problem if found. The problem was that I didn't have a lot of experience with Express or React, so that made it harder and, to be honest, I don't think the tools we were given during class time were enough to build an app, which goes beyond the sum of multiple technologies and fancy framework names.

Daniel

Of course, I will not attempt to pretend that the project went excellent. Many mistakes were made. Many can be attributed due to lack of time and stresses brought to us individually by the current situation but, as in most projects, it was mostly due to miscommunication, lack of

organization, and underestimation of the difficulties of the project and the technologies involved.

Personally, I've been involved in bigger, more ambitious projects with more traditional technologies. The difficulty with this project was, in my estimation, not in any way limited by the abilities of members of our team. It was purely organizational, personal, and stress-related. In a way, I'm happy I had this experience. I managed to notice failures in my habits, and now I know how they react with high-stress situations as the current ones. Better that it happened here, than when actually employed.

I'm excited to put to test my new strategies to cope with situations like these. Not to mention that the technologies and paradigms here studied will be invaluable, and I look forward to learning looking deeper into them as soon as I have actual time to do it.

Alejandro

This project ended up being more complicated than we originally thought, when designing and making choices for how the app would work, we never considered the limitations of our knowledge and our inexperience in working with new stuff, there is not enough time in class to learn about every instance, problem and application, so a lot of the investigation had to be on our own, as the material in class was just the base, MongoDB for example ended up causing us troubles while trying to apply transactions in the app, but the investigation and tries ended up most of the time in dead ends, which made the development itself harder than it was already; despite all this, it really was a good learning experience that summarize all the subjects seen in this class.