Routable AI

Fleet Visualizer

- Mikhail Khayretdinov, `M.Khayretdinov001@umb.edu`
- Xianbin Zhen, `xianbin.zhen001@umb.edu`
- Cole Maxwell, `cole.maxwell001@umb.edu`
- Sam Kellman-Wanzer, `sam.kellmanwanzer001@umb.edu`
- Ryan Dang, `Ryan.Dang001@umb.edu`
- Menno van der Zee, `menno@routable.ai`

Revision History

- v1 02/28/2020
- v2 03/27/2020
- v3 05/15/2020
- v4 05/22/2020

1 Introduction

1.1 Purpose

The project provides a map-based visualization via web application to the AI-powered technology for dispatching large fleets of vehicles. The target audience of the application is fleet operators and related personnel who might elevate their performance and the efficiency of their fleets through accessing an insightful data provided by our interactive visualization tool.

1.2 Scope

The name of the project is the *Fleet Visualizer* and it is fully accessible through internet via a web browser. The essential functionality of the application is built around the control by fleet operators and related personnel of the interactive map. This map allows them to observe and analyze fleets and its performance, indicating problem areas and potential for operational improvements. Users of the *Fleet Visualizer* are offered a variety of tools for accessing and collecting all the significant information either about their whole fleet or about each vehicle in particular.

Main features:

- The application tracks the current route of the particular vehicle with its particular location on the interactive map of the area where it operates.
- It reports location and time of each departure of any particular vehicle
- It reports location and time of each arrival of any particular vehicle.
- It is possible to see the route plan for the upcoming rides of each vehicle.
- Play / pause is present to enable/disable the visualization.
- Slider is present to manipulate the current time of the visualisation.
- Several map themes are offered to choose from.

Features that are not supported:

- The project is a visualisation tool only, therefore it does not provide algorithms of fleet dispatching optimization. This technology is provided by Routable AI.
- The user cannot contact the driver via the application.
- The user cannot adjust or manipulate the route of a vehicle. They can only observe and analyze it.

1.3 Overview

Fleet visualizer is a web-based application. The data collected by Routable AI will be visualized and interactive. Visualizing and interacting with this data sheds incredible insight into how operational efficiency can be improved and where other problems exist within the system. The Fleet visualizer show the street map (in several different Map Themes), as well as the movement and events (picking someone up, dropping someone off, etc) over time.

1.3.1 Existing Work

Here, we present some existing and similar software products.

Waze Waze(formerly FreeMap Israel) is a GPS navigation software app owned by Google. It works on smartphones and tablet computers that have GPS support. It provides turn-by-turn navigation information and user-submitted travel times and route details, while downloading location-dependent information over a mobile telephone network. Waze describes its app as a community-driven GPS navigation app, which is free to download and use. `https://www.waze.com/`

Google Maps Google Maps is a web mapping service developed by Google. It offers satellite imagery, aerial photography, street maps, 360° interactive panoramic views of streets (Street View), real-time traffic conditions, and route planning for traveling by foot, car, bicycle and air, or public transportation. In 2020, Google Maps was used by over 1 billion people every month. `https://www.google.com/maps`

1.3.2 Limitations

- **L1** Time based: Client can only view a limited period of historical data.
- **L2** Time based: Generating a representation of future data is difficult and limited by the data points generated by Routable's routing algorithm.

2 Requirements

- **R1** The visualizer displays vehicles moving over time to pick up and drop off riders.
- **R2** The visualizer displays the historical path of each vehicle within the fleet.
- **R3** The visualizer dynamically displays the future path of each vehicle within the fleet.
- **R4** The visualizer displays vehicle data for a time period specified by the user.
- **R5** Users can interact with the visualizer by playing, pausing, and jumping to any time within their requested time period.
- **R6** The visualizer displays vehicle locations within a 3D model.

3 Specifications

- **S1** In the visualization, green circles will represent pickup locations, red circles will represent drop off locations, and an arc between the two circles will represent the route requested.
- **S2** The visualizer displays historical vehicle data, which is retrieved from a Routable AI server through an HTTP request from the Routable API.
- **S3** The visualizer displays future vehicle data using pickup, drop off, and vehicle location requests, and future paths are based on a well-researched optimization algorithm and Routable AI's mobility-as-a-service backend.
- **S4** The user provides their unique fleet key, the start time and end time they wish to view data for, and their request will be specified as URL query parameters.
- **S5** The user interface has a play/pause button to start and stop the visualization, and a slider to manipulate the current time.
- **S6** Based on vehicle location coordinates, the deck.gl framework provides visual layers to generate a 3D model of the route and vehicle representation.

4 Design

4.1 Use Cases

The Fleet Visualizer allows the fleet manager to display route data that is requested from the Routeable API. This gives the fleet manager a visualization of what route the vehicle(s) can take, have taken, or be a projection of the routes it may take. The user can select what type of map to view, and what vehicles they wish to view. They can also view the vehicle(s) in different places in time with a slider.

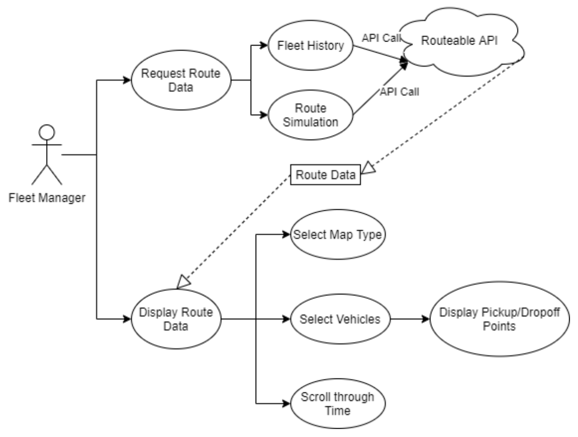


Figure 1: Use case diagram for the Fleet Visualizer

4.2 Architecture

Deck.gl is a web based visualization tool for large collections of data layered on top of a map. It was recommended for us to use and is ideal for this type of project

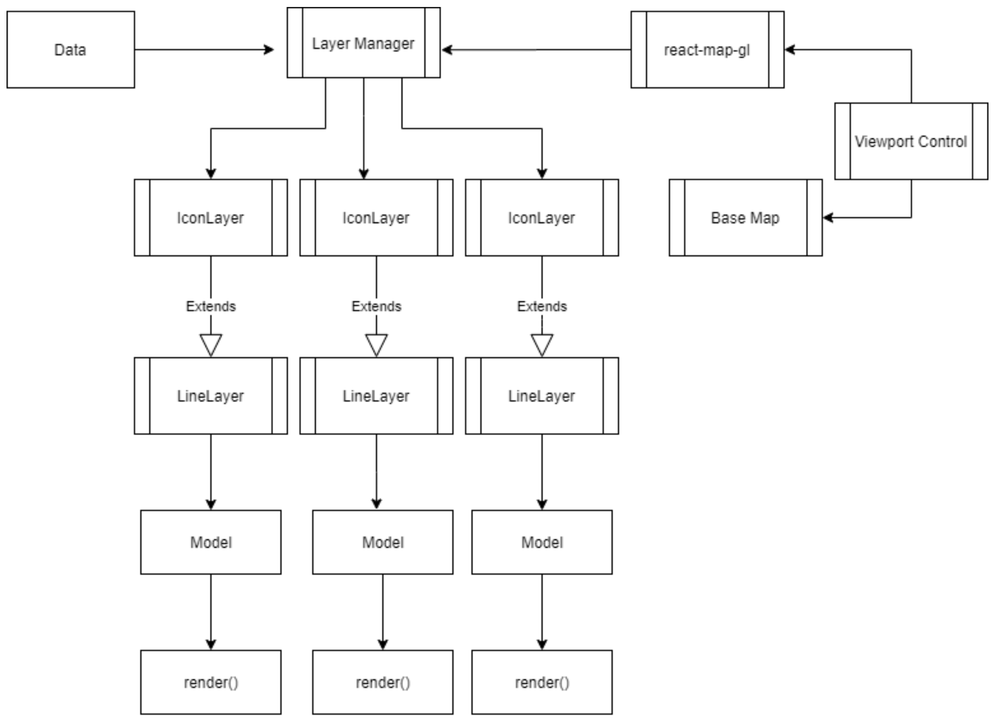


Figure 2: UML Diagram

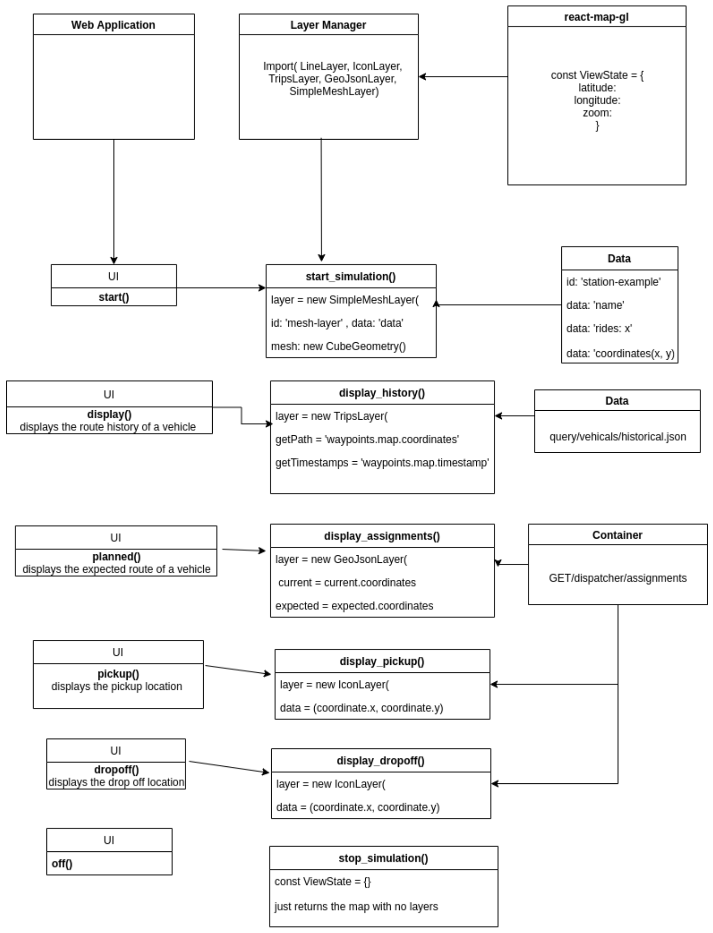

Web Application
Layer Manager
Import(LineLayer, IconLayer, TripsLayer, GeoJsonLayer, SimpleMeshLayer)
react-map-gl
const ViewState = { latitude: longitude: zoom: }
UI
start()
start_simulation()
layer = new SimpleMeshLayer(
id: 'mesh-layer' , data: 'data'
mesh: new CubeGeometry()
Data
id: 'station-example'
data: 'name'
data: 'rides: x'
data: 'coordinates(x, y)
UI
display()
displays the route history of a vehicle
display_history()
layer = new TripsLayer(
getPath = 'waypoints.map.coordinates'
getTimestamps = 'waypoints.map.timestamp'
Data
query/vehicals/historical.json
UI
planned()
displays the expected route of a vehicle
display_assignments()
layer = new GeoJsonLayer(
current = current.coordinates
expected = expected.coordinates
Container
GET/dispatcher/assignments
UI
pickup()
displays the pickup location
display_pickup()
layer = new IconLayer(
data = (coordinate.x, coordinate.y)
UI
dropoff()
displays the drop off location
display_dropoff()
layer = new IconLayer(
data = (coordinate.x, coordinate.y)
UI
off()
stop_simulation()
const ViewState = {}
just returns the map with no layers

Figure 3: Class Diagram

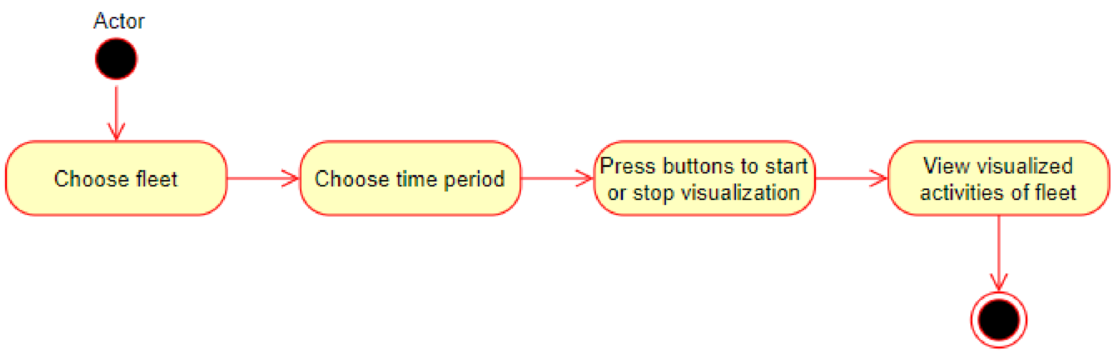
Here is the update in final version of our project. We introduced the following Deck.gl layers: TripsLayer(show the movement of each vehicle), IconLayer(use icon to indicate pickup and dropoff locations), GeoJsonLayer(dynamically show the future path of each vehicle), SimplyMeshLayer(use a car model to represent the current location of each vehicle) and React UI components(control the behavior of layers).
We were not familiar with react at the beginning of this project, so we change our class structure a lot. Our main class is App which extends react Component. App has several life cycle methods: constructor(), componentDidMount(), componentWillUnmount(), render(). During the mounting methods(constructor and componentDidMount), we set up the properties of our App and fetch the data from an URL. The updating method(render) is called when a component is being re-rendered as a result of changes to either its props or state. Method aimate() is responsible to change the state of our App. We update the data providing to each layer in the function animate() so to cause App to re-render. As a result, it shows animation in the App. We also render some static html and react component as our UI. The unmounting method(componentWillUnmount) is call when a component is being removed from the DOM. We simply stop the animation request.

4.3 Workflows

The workflows of our Fleet Visualizer can be described with the activity diagram in Figure 4. The actor, which is our fleet manager, enters the fleet key, start time, and end time to specify the information to visualize. The user then presses play / pause button to start or stop the visualization.

4.4 Technologies and Implementation Details

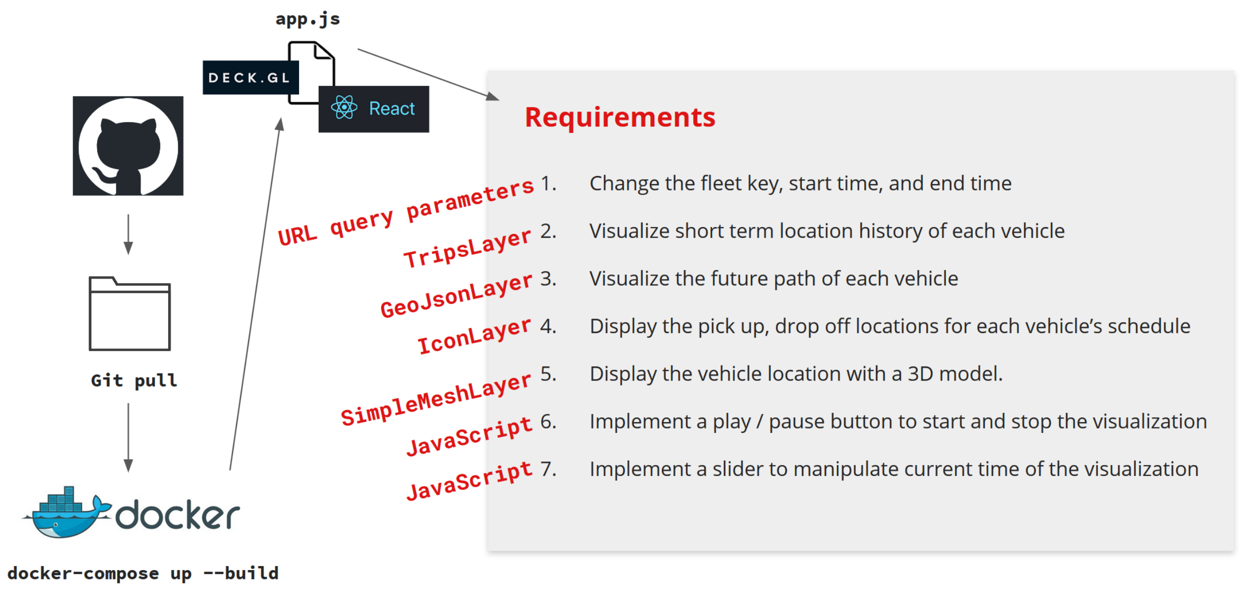
The backend is supported by our client, therefore our project focuses on the frontend aspects of the application. To implement, the following technological stack was utilized:



Actor
Choose fleet
Choose time period
Press buttons to start or stop visualization
View visualized activities of fleet

Figure 4: Activity Diagram for Fleet Visualizer

- HTML/CSS for structuring the layout of the web application.
- Javascript and React.js specifically to implement the dynamic parts of the application.
- Routable AI provides API (Application Programming interface) that will allow us to access key events information (picking up a passenger or dropping someone off etc.) and the state of the vehicle at a set interval as they operate. We specify this data as URL query parameter via some of the React.js libraries.
- The graphical tool of out project is deck.gl, which provides several layers of visualization outlined below:



app.js
DECK.GL
React
Git pull
docker
docker-compose up --build
Requirements
URL query parameters 1. Change the fleet key, start time, and end time
TripsLayer 2. Visualize short term location history of each vehicle
GeoJsonLayer 3. Visualize the future path of each vehicle
IconLayer 4. Display the pick up, drop off locations for each vehicle's schedule
SimpleMeshLayer 5. Display the vehicle location with a 3D model.
JavaScript 6. Implement a play / pause button to start and stop the visualization
JavaScript 7. Implement a slider to manipulate current time of the visualization

Figure 5: Technologies Workflow

Slack for daily communication
Google Hangouts for meetups
Github for development collaboration

Figure 6: Collaboration Tools

4.5 Team Structure and responsibilities

- Xianbin Zhen - Technology Lead
- Cole Maxwell - Client Lead
- Mikhail Khayretdinov - GeoJsonLayer
- Ryan Dang - SimpleMeshLayer
- Sam Kellman-Wanzer - IconLayer

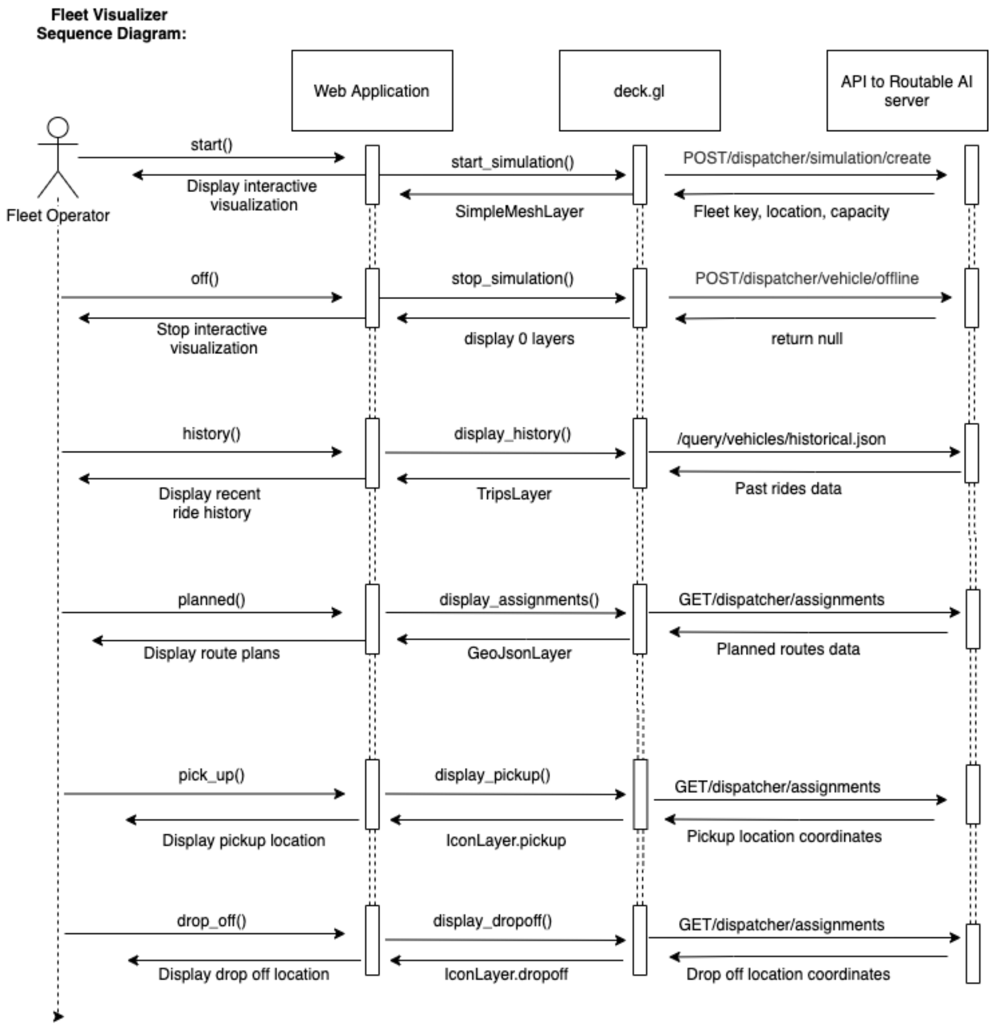

Fleet Visualizer Sequence Diagram:
Web Application
deck.gl
API to Routable AI server
Fleet Operator
start()
Display interactive visualization
start_simulation()
SimpleMeshLayer
POST/dispatcher/simulation/create
Fleet key, location, capacity
off()
Stop interactive visualization
stop_simulation()
display 0 layers
POST/dispatcher/vehicle/offline
return null
history()
Display recent ride history
display_history()
TripsLayer
/query/vehicles/historical.json
Past rides data
planned()
Display route plans
display_assignments()
GeoJsonLayer
GET/dispatcher/assignments
Planned routes data
pick_up()
Display pickup location
display_pickup()
IconLayer.pickup
GET/dispatcher/assignments
Pickup location coordinates
drop_off()
Display drop off location
display_dropoff()
IconLayer.dropoff
GET/dispatcher/assignments
Drop off location coordinates

Figure 7: Sequence Diagram

Note, that to get planned routes, pick ups and drop off locations we make the same API call: GET/dispatcher/assignments. Below is the sample of returned information about two vehicles, one of which has an active request and another one doesn't. It starts with the information about active vehicles, 'vehs'. In our sample case we have two active vehicles, one with id 42 and another one with id 89. The vehicle with id 42 has an active request which means it has accepted and is performing a transportation of passengers and thus it must have a starting point of the journey which is the pickup location and time and the endpoint of the journey, which is the drop off location and time. Both of these points are listed in the 'events' subsection related to the vehicle. Since vehicle 89 doesn't have any accepted requests it has no 'events' subsection related to it. Comments in bold outline each block of code.

```
{
   ”vehs”: [
               {
               /*** This block of code provides information regarding the vehicle with id 42. It is executing one
               of the requests and therefore has information concerning a pickup and a drop off locations ***/
               ”veh_id”: 42,
               ”location”: {
                  ”lat”: 40.75315839577503,
                  ”lng”: -73.98957383947436
               },
               ”assigned”: true,
               ”req_ids”: [
                  31
               ],
               ”events”: [
               /*** This block of code provides information regarding the pickup for vehicle 42. ***/
                  {
                     ”req_id”: 31,
                     ”location”: {
                        ”lat”: 40.75466940037548,
                        ”lng”: -73.99382114410399
                     },
                     ”time”: ”2019-08-06T00:16:10Z”,
                     ”event”: ”pickup”
                  }, /*** This block of code provides information regarding the
                   drop off time and location for vehicle 42 ***/
                     {
                     ”req_id”: 31,
                     ”location”: {
                        ”lat”: 40.74465591168391,
                        ”lng”: -73.98643970489502
                     },
                     ”time”: ”2019-08-06T00:34:10Z”,
                     ”event”: ”dropoff”
                  } ] },
   {           /*** Information about the vehicle with id 89. It has no assigned
               destinations and therefore lacks pickup and drop off blocks. ***/
               ”veh_id”: 89,
               ”location”: {
                  ”lat”: 40.74135093859483,
                  ”lng”: -73.99630284736294 },
               ”assigned”: false,
               ”req_ids”: [ ],
               ”events”: [ ]
        }
],
/*** Information about the requests and notifications will
not be directly accessible in our application and therefore is omitted ***/
   ”reqs”: [...]
   ”notifications”: [...]    }
```

4.6 User Interface

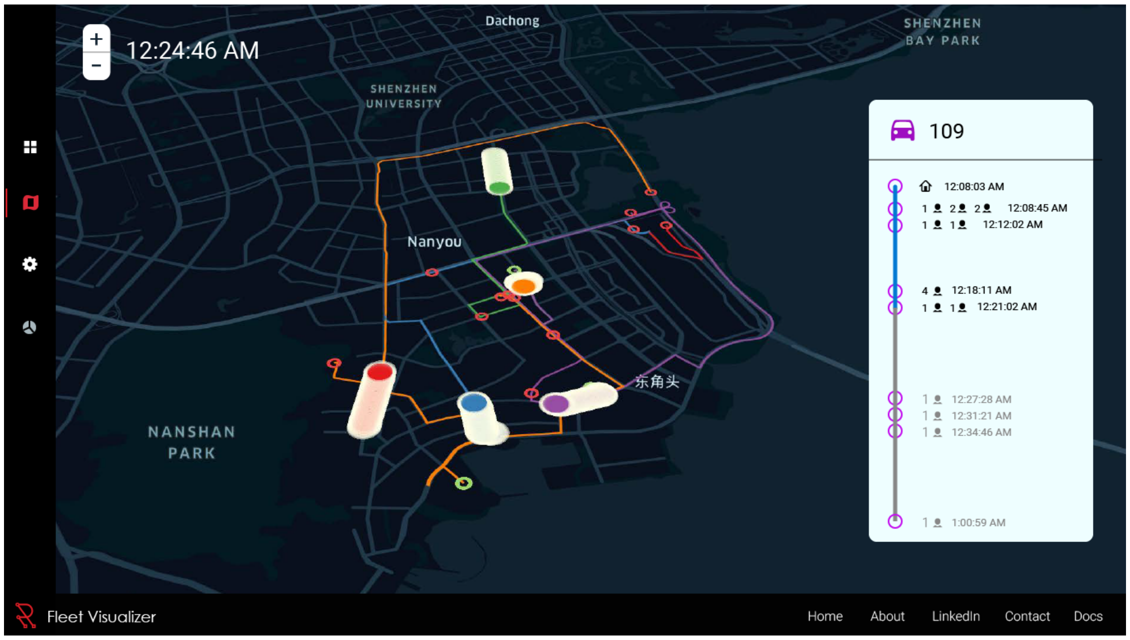
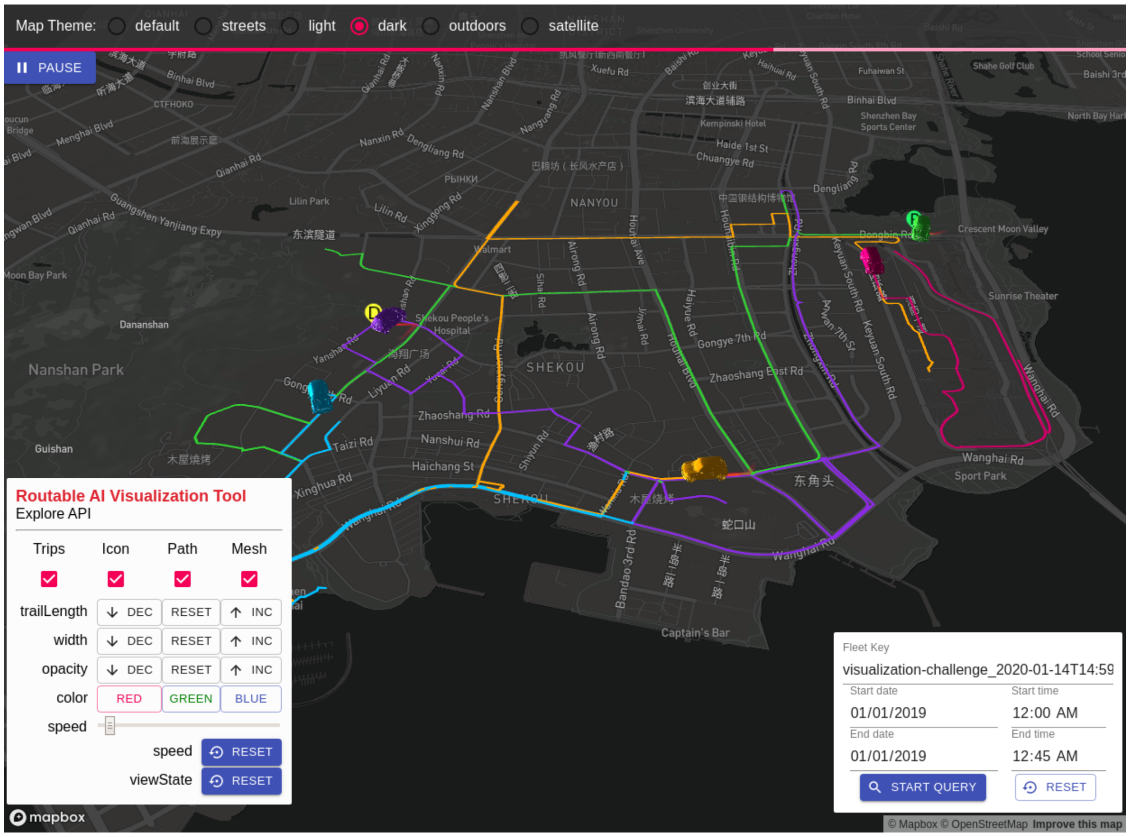


Figure 8: UI mockup



Map Theme: default streets light [✓ dark] outdoors satellite

Fleet Key: visualization-challenge_2020-01-14T14:59

Start date	Start time
01/01/2019	12:00 AM
End date	End time
01/01/2019	12:45 AM

Figure 9: UI Final (Dark Map Theme)

5 Timeframe and Milestones

Timeline

- 2/28/2020 - Project Proposal Draft
- 3/27/2020 - Revised Project Proposal
- 3/27/2020 to 5/4/2020 - Work on Project (Coding)
- 05/15/2020 - Project Presentations
- 5/22/2020 - Documentation

Milestones

- Request Data from API and if needed parse them in a usable way
- Rendering the Map
- Render route data
- Scrolling through route's path

6 Installation and Use

To run the Fleet Visualizer on your local machine, complete the following steps:

1. Download and install Docker.
2. Run the development server using docker with the command **docker-compose up**. Note: If you change package.json or webpack.config.js, you will need to run **docker-compose up –build**.
3. Navigate to **localhost:8081** to view the visualization.

7 Testing

Since we only had one set of data points for this project to get started we initially had to test each layer with the sample data-sets provided by deck.gl, just to make sure that they were setup correctly and are able to display data at all. Overall since the main objective was visualization, the bugs were mostly immediately obvious and the testing process was mainly based on visual observation of how each particular feature and layer behaves compared to how it was expected to act.

8 Conclusion

Overall, the project was very successful and a great learning experience. We were able to connect with our client Menno early on, who provided us with a clear background on the company, the project, and his expectations. He provided us with a basic version of the project in a ZIP to get us started. After familiarizing ourselves with the project files (as well as getting up to speed on deck.gl and React), we scheduled a couple more meetings with Menno for some back-and-forth and clarification on our specific tasks. From here, we were able to divide the work amongst the team and begin working toward a quality project.

Our biggest challenge was loading fleet data in a form that could be used by all the different deck.gl layers. From the project requirements, we knew we had to allow the fleet key, start time, and end time to be specified as URL query parameters, but loading this data was a challenge. Additionally, we needed to ensure that we visualized the data in real time. The following steps were taken to solve these problems. First, we needed to fetch the data from the repository and store it in a JSON array. Then, we defined layer parameters from this data. Finally, we took the data that corresponds to a timestamp and push the new data each frame. The project called for the development of new skills and we ended up with a project we are proud of.