CAB432 Assignment 1

TrailSeeker Mashup

Nicholas Kress

n9467688

Semester 2, 2019

1. Introduction

This mashup, *TrailSeeker,* combines a database API with location, navigation and weather services to provide mountain bikers with updated information about mountain biking trails near them, with useful information such as current weather and driving time to each location. Users automatically provide their location to the service and in return receive a list of trails with all the necessary data.

APIs used

IPdata - https://ipdata.co
Provide an IP address to this API and receive information such as location data, network provider, user currency, time_zone and threat information.

Trails API - https://rapidapi.com/trailapi/api/trailapi
Takes latitude and longitude,with optional radius and number of items. Each trail found contains useful info such as description, image, rating, difficulty, directions and lat & lng.

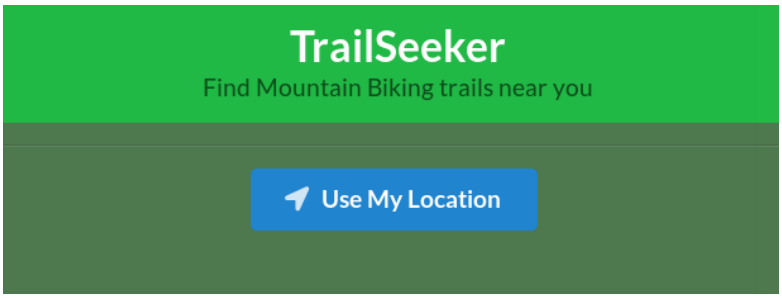
Google Maps JavaScript API
https://developers.google.com/maps/documentation/javascript/tutorial
Used to find transport information such as distance/time to each trail from user position to trail position.

DarkSky Weather API - https://darksky.net/dev
Takes location as latitude and longitude and returns the current weather. Returns current weather info as well as other useful info such as forecasts, text summary, sunrise/set time, humidity, wind speed etc.

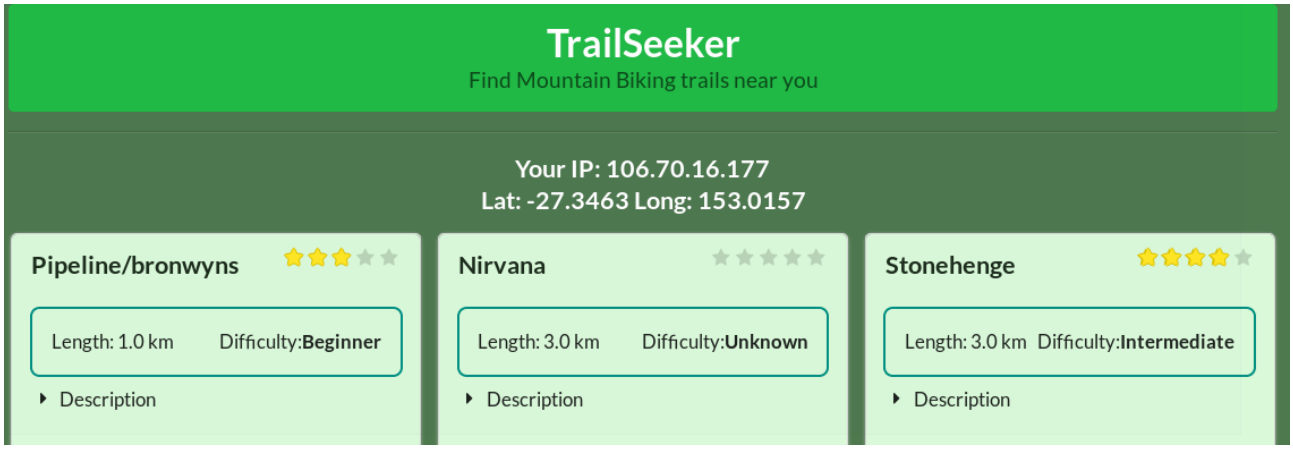
2. Mashup use cases and services

Use Cases

1. As a mountain biking enthusiast, I want to be able to find new mountain biking trails near my current location to ride.

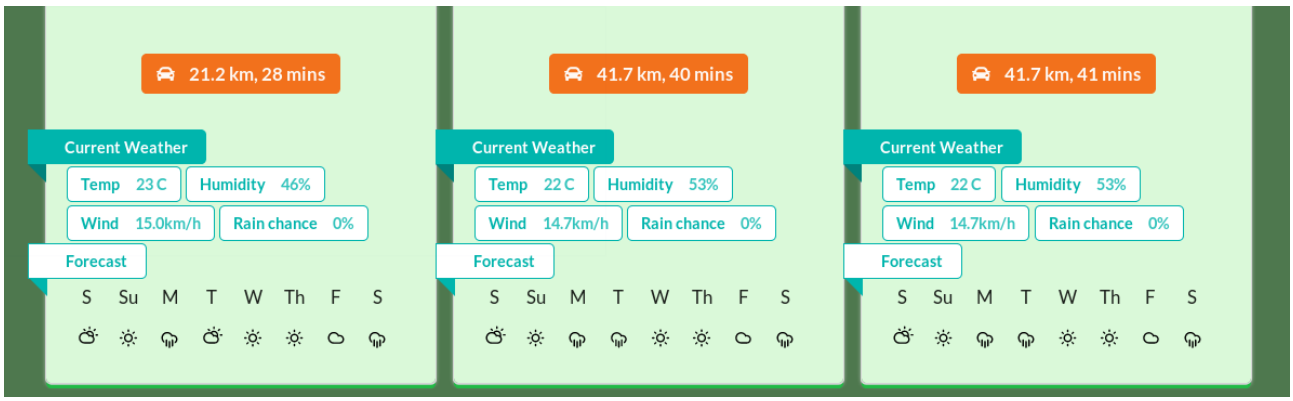


On the landing page of the app, users are simply shown the page title and the button to interact with the page and use their location. Location is gathered on the backend using the users IP, while this is not as accurate as using client side geolocation which can access GPS data, this still gives a general idea of the users location.



In *TrailSeeker*, Trails are displayed in a card layout with the trail information displayed at the top. Information includes the trail name, user rating, length from start to finish, difficulty and a description of the trail. The users IP and location latitude and longitude are also displayed.

2. As a mountain biking enthusiast who knows all the good trails, I want to see current weather information for each trail and the time it will take to get to them from my location with current traffic.



At the bottom of each trail card, the navigation information and weather data is shown. The navigation data is a click-able button that links to external driving directions on the Google Maps site.

Weather data gives specific current weather data such as current temperature, humidity, wind speed and chance of rain. Each trail also shows a weekly forecast with icons, the user can hover over any icon in the forecast to get the high/low temperature for that selected day. Note that in some locations weather icons may be the same due to trails being geographically close, but high/low temperatures should always vary somewhat.

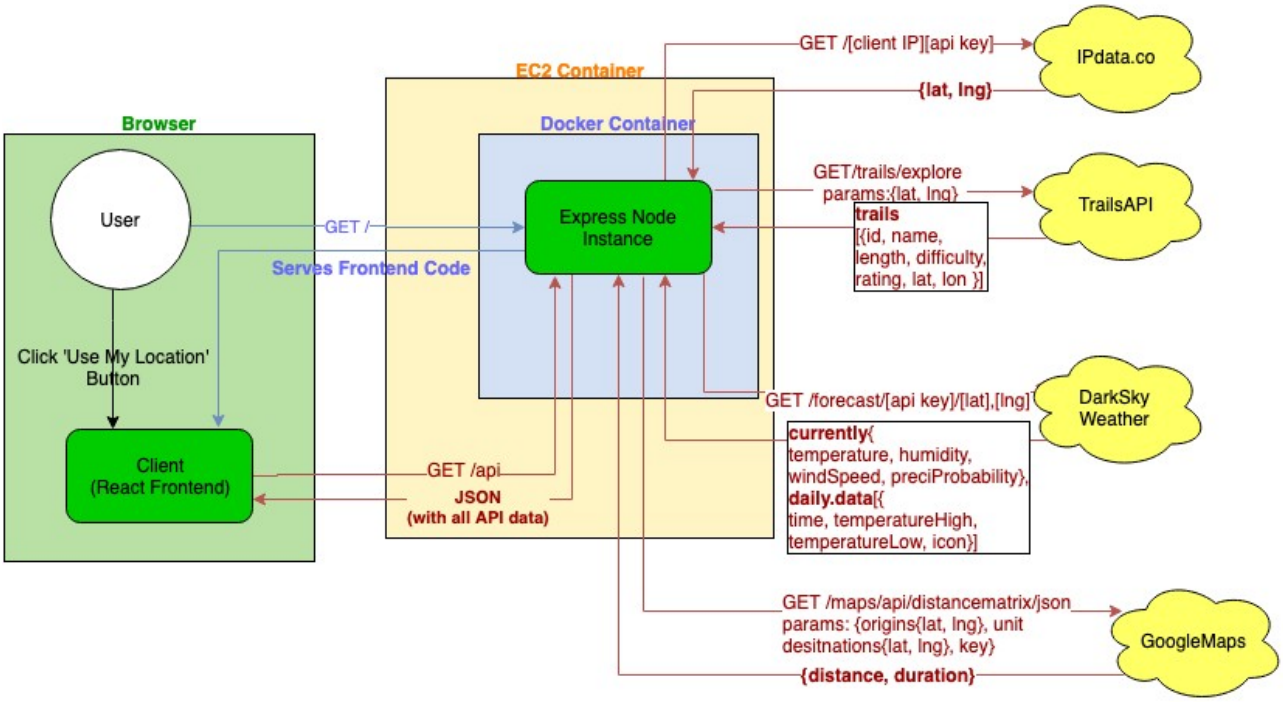
3. Technical Description

Breakdown

TrailSeeker runs on NodeJS using Express and React for the client side. A React production build is served to the client from the root(/) express route. The React client then makes a single request back to the server at the /api endpoint. The server handles all fetching and processing of necessary data, then returns a single JSON object to the client. All the client does is format the data for display, along with some visual error handling.

The single node instance runs in a Docker container running on an Amazon AWS EC2 instance with Ubuntu 18.04 as the OS.

This software architecture is showcased in the diagram below.



Note although each API returns more values for each request, only the values used are shown.

Operation Description

As shown in the diagram above:

1. Server serves React frontend to client via / endpoint
2. Client triggers request to server /api endpoint by pressing 'use my location' button
3. Server sends client IP address to Ipdata api to get location details `[lat, lng]`
4. Server uses location data to fetch list of trails from TrailsAPI `[id, name, length, description, difficulty, rating, lat, lon]`

5. For each trail using lat and lon, server requests data from Google Maps API and Dark Sky Weather API to get transport and weather info `[distance, duration, currentWeather{temperature, humidity, windSpeed, precipProbability}, weather{time, tenperatureHigh, temperatureLow, icon}]`
6. Server waits for all requests to complete, then returns the assembled data to the client as a JSON
7. Client receives data from server and formats for display it using React components

Data Handling (payload size reduction)

The APIs used provide a great more values and data then is needed to display. As this would be inefficient to collect and send to the client, the server only collects the variables needed for the TrailSeeker and can save the client and the server unnecessary network bandwidth.

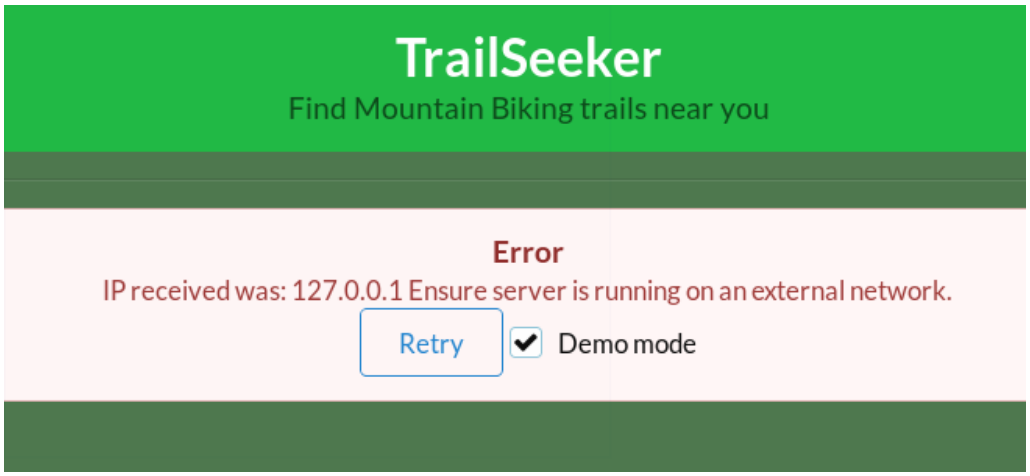
The original size of all requests (assuming 7 trails):

API	Stats
TrailsAPI	Status: 200 OK Time: 1298 ms Size: 5.54 KB
Dark Sky Weather	Status: 200 OK Time: 1111 ms Size: 24.2 KB x 7
Google Distance	Status: 200 OK Time: 545 ms Size: 972 B x 7
IPdata.co	Status: 200 OK Time: 223 ms Size: 1.68 KB
Total	**183.42KB**

Size of JSON sent to client: Status: 200 OK Time: 2277 ms Size: 8.24 KB

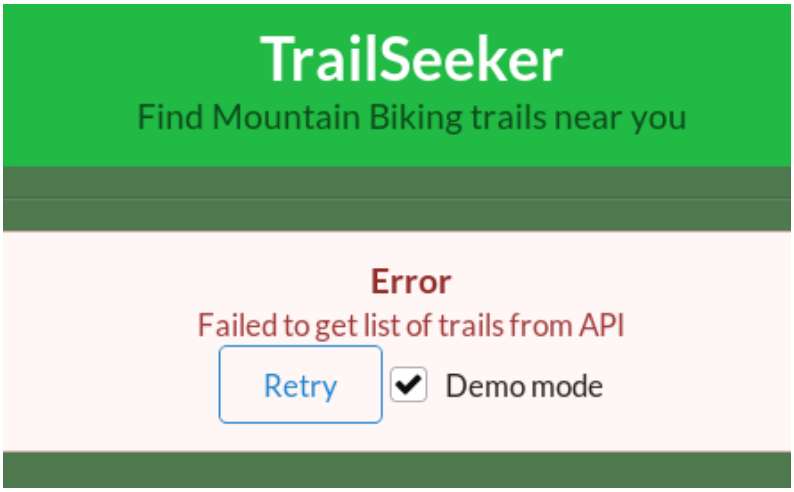
Error Handling

If the user tries to run the instance locally, in a Docker container or using node, the IP location lookup will fail as the localhost(127.0.0.1) IP will be supplied to the API.

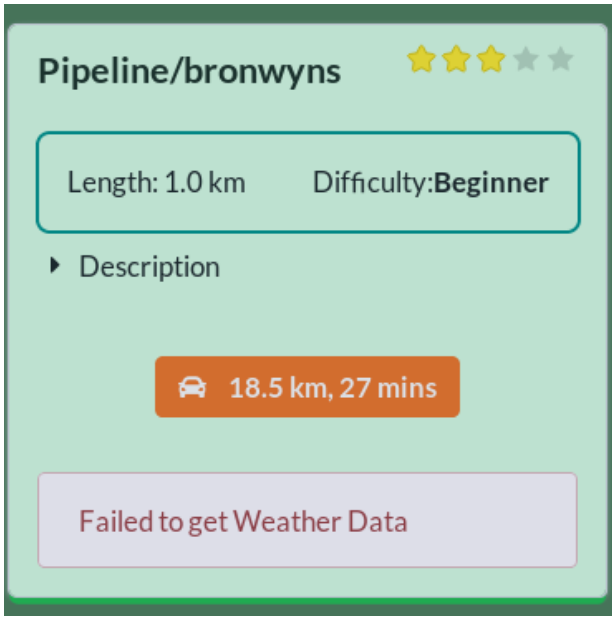


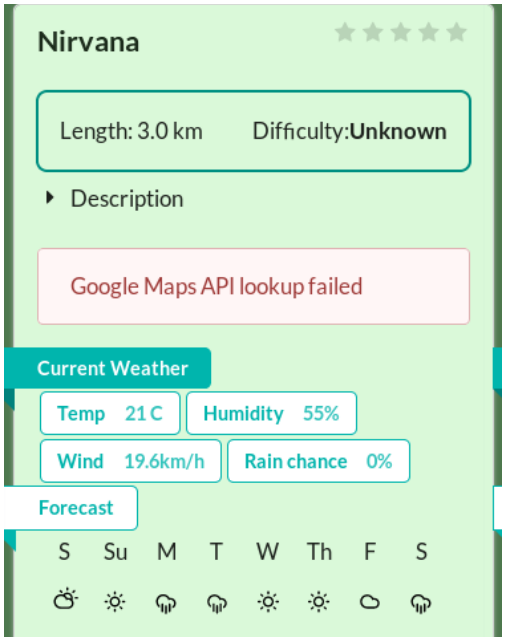
If this occurs, the user can select to retry in *demo mode*, where the server will skip the location lookup and use a pre-supplied latitude and longitude. In demo mode, the location cannot be changed and is used for demonstrating the applications capabilities only.

If the Trails API request fails, the user is informed of this information and given the option to retry.



If any other API requests fail, the user is notified but the successful retrieved data will be displayed.





Issues Encountered

An issue found early in development was the use of React possibly requiring two node instances to be running at once. Traditionally and in previous units such as CAB230, React was served on its own instance using node with pm2 or with another webserver. While it would be possible to Dockerize the React instance and the Express backend and have them run separately, I found it would be much simpler to have Express serve the React frontend so only one node process would have to run. This is done by building the static files from React and moving the build folder to the root of the Express app directory.

```
// Serve React static build
router.get('/', (req,res) =>{
    res.sendFile(path.join(__dirname+'/build/index.html'));
});
```

Express is then configured as above to serve the React client from the / endpoint. Any api calls needed to the server are then moved to the /api endpoint with React making calls to this endpoint. The server can then be more easily Dockerized as now only the server files directory must be included and executed.

4. Docker

```
FROM node:10
# create app directory
WORKDIR /usr/src/app
# use same package manager used in development
RUN npm install -g yarn
# copy source code from server folder to app
COPY ./src/server .
# install node_modules
RUN yarn
# open port 3000 in container
EXPOSE 3000
# start node instance
CMD ["yarn", "start"]
```

The Dockerfile uses the Node as the base image and simply copies all data from the server development directory to a new app folder in the image. Once the source is copied, dependencies using the *yarn* package manager. *Yarn* was instead of *npm* as this was the package manager used in development, it is a good idea to keep this consistent between development and production. The port that the express server uses is exposed and then the node application is started using the command `yarn start`

In the server folder, a .dockerignore file is included to ensure that no node_modules folders are copied to the docker container.

The image is run using:

```
docker run -d -p 80:3000 --restart unless-stopped
nkress/trailfinder:latest
```

This runs with the options:
-d : *Run container in background and print container ID*
-p : *specifies the port, mapping the host:80 to container:3000*
--restart unless-stopped : *restarts the container in case of an crash/error unless the container has been specifically stopped by the user*

5. Testing and Limitations

Action	Expected Outcome	Actual
Load application from root page.	TrailSeeker homepage is displayed Heading, subheading and button.	**Pass**
From homepage, click 'use my location' button	Page displays 'Loading' with spinner. After loading is finished, page is displayed with User IP, lat and long, and all trails are displayed.	**Pass**
Once page is loaded, click orange navigation button	A new browser tab opens to Google Maps with the location of the trail displayed	**Pass**
Once page is loaded. Hover over icon in a trails weather forecast	A small popup will display the high and low temperature for that day. These should be different for each day and trail	**Pass**
Once page is loaded. Check that weather data is different for each trail	Some trails have similar data because of close location but not all trails have the same weather numbers	**Pass**
Once page is loaded, view trail cards	Trails do not all have the same name, description, rating, length and difficulty.	**Pass**
Once page is loaded, verify IP data using a tool such as https://www.iplocation.net/	IP address matches that shown in TrailSeeker. Location data is not exactly the same but is close depending on which service used	**Pass**
If possible, use a proxy or VPN to test from different locations.	IP and location data are updated to that of VPN/proxy. A different list of trails are returned from search.	**The application works in most worldwide countries/areas tested. Some show many more trails then my current location. However, some areas such as *Israel*, result in the page loading until timeout.**

To test error-handling, the app has to be run locally in order to change varibles.

Action	Expected Outcome	Actual
Run server instance locally. Navigate to localhost:[port used]. Click button.	After button is clicked, the client will attempt to load but then respond with an error message with the received IP, a retry button and a demo mode checkbox.	**Pass**
On error screen, click retry.	The client attempts a retry, but shows an error again because demo mode was not checked.	**Pass**
On error screen, check demo mode checkbox, then click retry.	The page successfully returns a page with trails, with the users Lat: and Long: each being labeled with DEMO. All other functionality works.	**Pass**
Remove the last letter of the TrailsAPI key string in `src/server/src/requests function requestTrails headers - x-rapidapi-key` Restart the server. Run in demo mode	The page returns an error saying it failed to get a list of trails from the API with the option to retry	**Pass**
Remove the last letter of the Weather API key in `src/server/src/requests const WEATHER_API_KEY` Restart the server. Run in demo modeRemove the last letter of the Weather API key in `src/server/src/requests const WEATHER_API_KEY` Restart the server. Run in demo mode	The page loads but weather data is replaced with an error message. All other functionality remains working.	**Pass**
Remove the last letter of the Weather API key in `src/server/src/requests const GOOGLE_API_KEY` Restart the server. Run in demo mode	The page loads but the navigation button is replaced with an error message. All other functionality remains working.	**Pass**

Limitations

The main limitation of the app is location accuracy and running locally. Both of these are due to the users location being obtained via IP address rather than using client side geolocation. This was originally included in the app, however it was moved to the

backend as I did not want it to seem that too much functionality was being handled by the client side when it was meant to done by the server.

Another limitation is that of the weather data. Because in most locations, the trails are quite close to each other, the weather forecasts will likely be very similar. The API does return different high/low values and these can be seen in the application.

Some trails, when opened to the Google Maps link, may not appear to be exactly the right location for the trail. This is a limitation from the data from the TrailsAPI where some trails have the exact correct co-ordinates and others do not.

Some trails may show some data being *unkown* or *0.* This is also data missing from the TrailsAPI database.

6. Possible extensions

Client Side Geolocation
Client side geolocation such as Google Maps geolocation API or the HTML Geolocation API to improve accuracy of the users location. This would allow for better time and distance estimates from the users location to each trail. This was included in an early development version of this mashup, but was moved to the server-side later on to have all the processing done on the server-side.

Location selection
Currently TrailSeeker only allows the user to use their current location as the basis for their search. An improvement would be to use a map or address selector to allow the user to see other trails near a specific location.

Embed Maps
Instead of linking to the Google Maps site for directions, Google Maps could be embed directly into the page to show directions graphically so that the user would not have to leave the site.

7. References

Resources Used

Server:

- **Express** https://expressjs.com/
  Web framework for nodejs
- **Axios** https://github.com/axios/axios
  Promise based HTTP client for the browser and node.js
- **request-ip** https://www.npmjs.com/package/request-ip
  A tiny Node.js module for retrieving a request's IP address
- **helmet** https://github.com/helmetjs/helmet
  Help secure Express apps with various HTTP headers

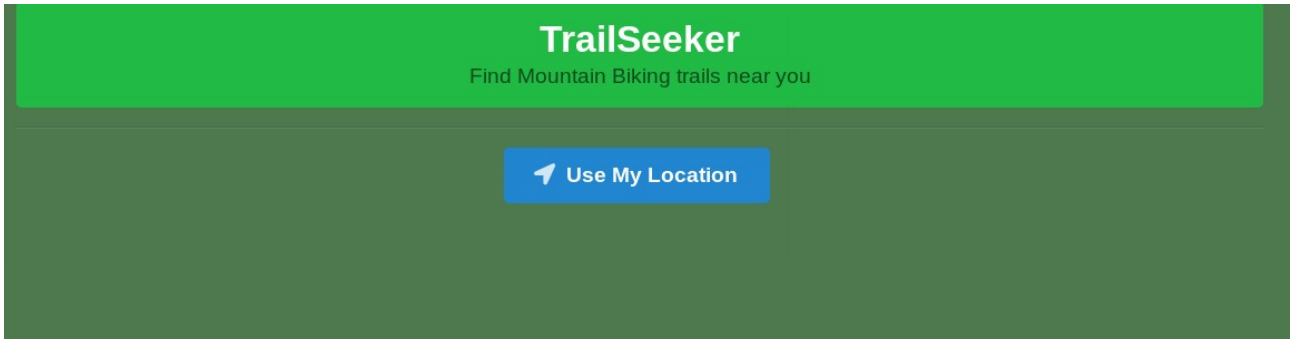
Client:

- **React** https://reactjs.org/
  A JavaScript library for building user interfaces
- **semantic-ui-react** https://react.semantic-ui.com/
  React integration for the Semantic UI CSS framework

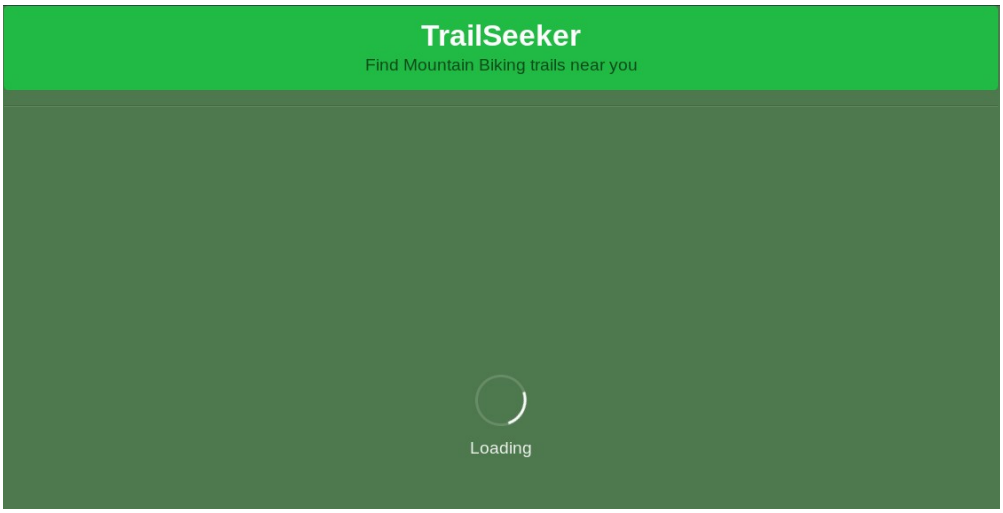
8. Appendix

User Guide

Navigate to the instances IP or hostname with correct port.

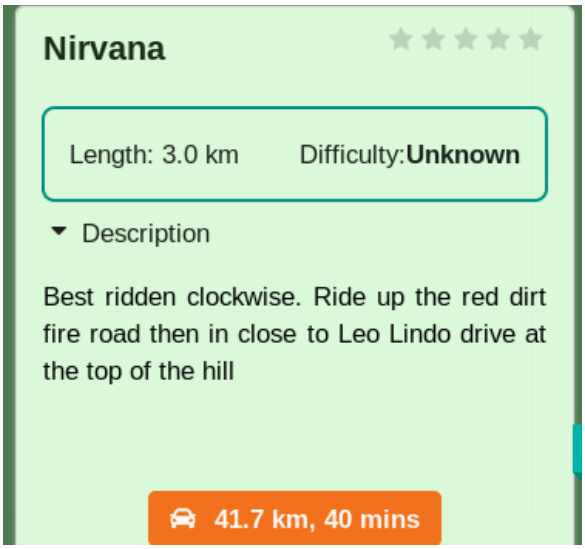


Click Use My Location to load trails.

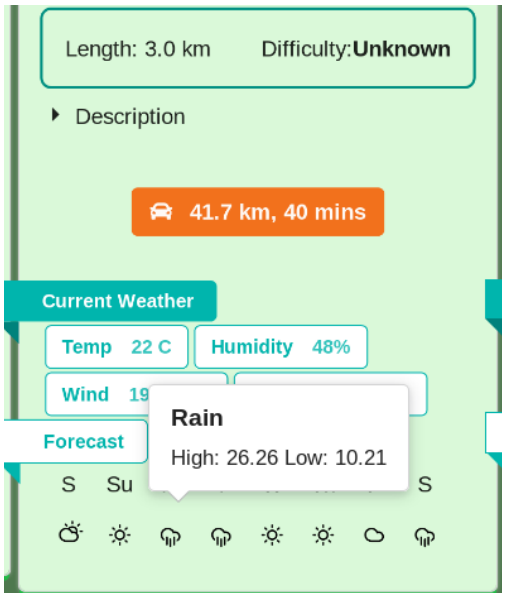


Wait for the page to load.

Click the description to view text.



Hover over weather icons to see high/low



Click on orange navigation button to be redirected to Google Maps with trail location.