

Design Document

Problem Statement -

Navigation systems optimize for the shortest or fastest route. However, they do not consider elevation gain. Let's say you are hiking or biking from one location to another. You may want to literally go the extra mile if that saves you a couple thousand feet in elevation gain. Likewise, you may want to maximize elevation gain if you are looking for an intense yet time-constrained workout. The high-level goal of this project is to develop a software system that determines, given a start and an end location, a route that maximizes or minimizes elevation gain, while limiting the total distance between the two locations to $x\%$ of the shortest path.

The functional requirements that we have considered for this system are as follows:

- The application must provide a valid route between 2 points in such a way that it maximizes or minimizes elevation gain under the constraint that the total distance outputted by the system cannot be more than $x\%$ of the shortest path.

The inputs to the application are -

1. Source address
2. Destination address
3. A choice between maximizing or minimizing elevation gain between the source and destination
4. The acceptable percentage within which the system can provide the path length

The expected output of the application is a numerical value for the path length and a map showing the visual representation of the exact route chosen by the system from source to destination.

- The system validates the inputs and instructs the user in case of invalid input values.

The non-functional requirements that we expect the system to fulfill are:

Understandability, Readability, Testability, Portability, Usability, Modularity. These are further explained below:

Understability: Well documented user manual tailored to the needs of a non-technical user, well documented functional documentation explaining the flow and functionality of each method used.

Readability: Commented code to provide clarity to developers. Detailed description to run the system in the README.md.

Testability: To ensure that our system indeed satisfies the expected functionalities and to test any edge cases we have written unit and integration tests. To ensure that adequate testing is done we have also evaluated the code coverage through testing.

Portability: We have dockerized our application to make it machine independent and free of any setup procedures. This makes our system ready to use on the fly by accessing the image.

Modularity: Our backend server code is modularized into 2 main modules namely: map generation and path finder.

Usability: The UI created is intuitive with significant directions to the user to use the applications.

Software Development Phases

Planning and Design Decisions :

Choice of Server - A flask server was chosen for this application given after analyzing the scale of the application. Flask is a micro server which is easy to set up and provides inbuilt method invocations that accelerate the development process.

Choice of API for getting Elevation Gain - **Open elevation API** was selected as it is a free and open source alternative to Google's elevation API.

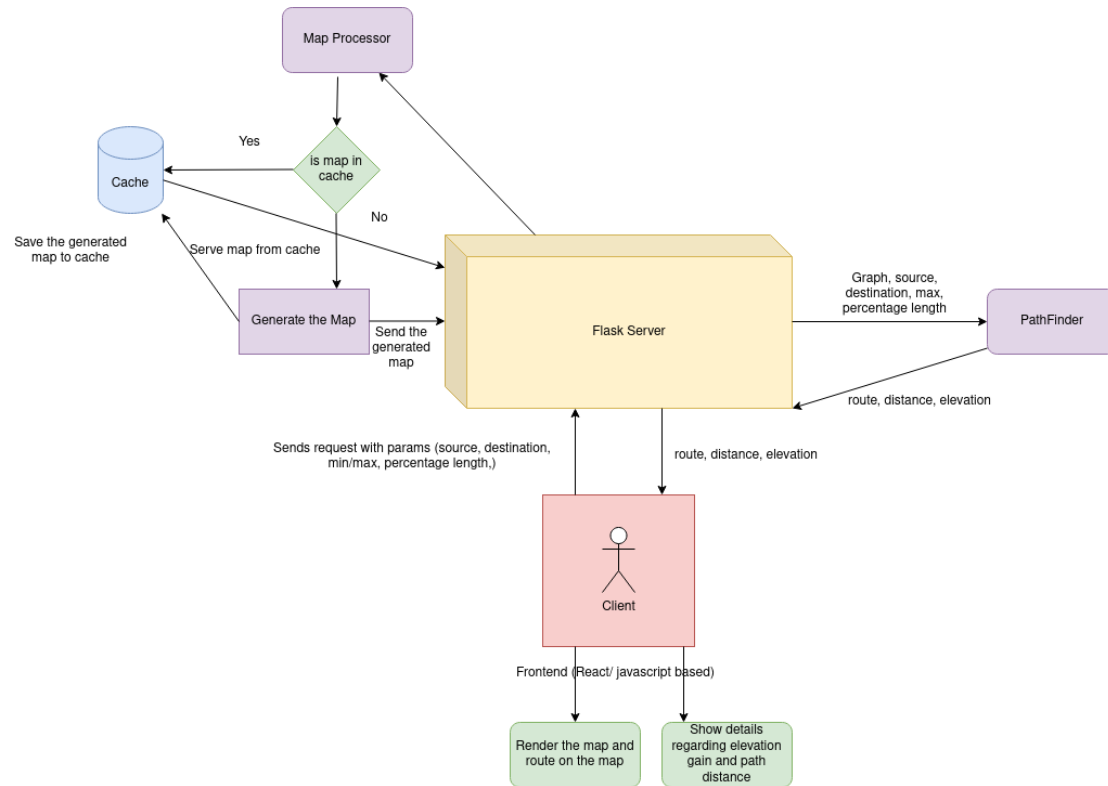
Choice for map generation - **OSMnx library** was chosen as it is an open source map generation library that abstracts the process of converting maps to networkX graphs. The generated graphs can use all functionalities of networkX graphs.

Choice of cache method - The graphs generated are way too large and would occupy a huge amount of space if stored as it is. Hence while caching, they have been dumped as **.pkl files** which compress the content when storing.

Choice of path finding algorithm- We chose 2 algorithms to implement - Dijkstra's algorithm and A* path finding algorithm. Dijkstra's ensures that we find the shortest path between two points while A* which is a heuristic based algorithm that can provide a different output. As our requirement specification aims to find an optimized path

including elevation as an added factor we chose these algorithms and have analyzed their outputs before providing the user with the most suitable output.

Design and Architecture:



We have proposed the above design for the application. The application follows both Client- Server and MVC architecture patterns.

After analyzing the requirements we concluded that the language of choice for implementing the application would be python for backend and JavaScript for frontend.

Development and Implementation:

The modules of the application are described below:

- **Client** - Our client is written in JavaScript. It consists of a form that takes input from the user and sends form data to the server. Our source and destination input is integrated with Google's Autocomplete Places API that facilitates entering location for the user. The user can enter source , destination in the form, choose either minimum/maximum elevation gain and choose percentage to limit shortest

distance to. When the client receives a response from the server, we pass the route information to Google's Direction API which plots the route on the generated map.

- **Backend Server** - A Flask based backend server. It receives the input from the client and calls the map generator module to generate a map for the specified city. It will then call the pathfinder module to generate the most optimized route for the given user input. The generated map and optimized route along with elevation gain info will be sent back to the client for frontend rendering.
- **Map Generator** - Map generator module will be responsible for fetching latitude and longitude for source and destination using Geocoding API from Google Maps Platform and generating map of input city using OSMnx library. Osmnx provides a method which attaches elevation data to each node of the generated graph using Google's elevation API. Since Google's elevation API is not free, we have modified this method to use Open-elevation API which is an open source version of Google's elevation API.

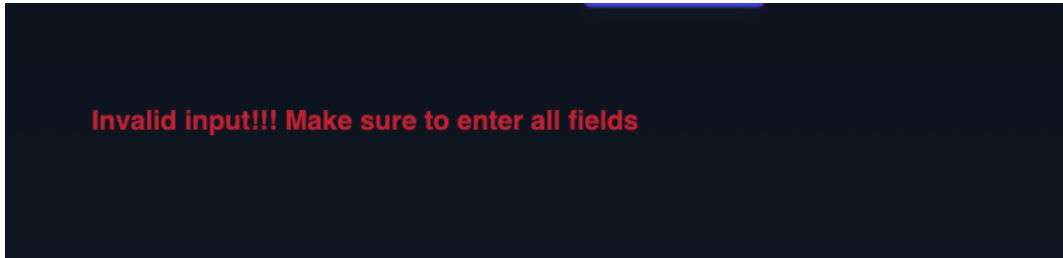
Using the Open-elevation API is a time consuming process, thus to improve the response time of the application we have introduced a cache. This module caches any newly generated maps to avoid redundant calls to the API.

- **PathFinder** - PathFinder module is responsible for finding an optimized path for a given user input. The optimized path is found through a 2 step process. In the first step, it finds the path elevation based route using 2 algorithms - Dijkstra's and A*. A* is a heuristic based algorithm and hence in certain situations both algorithms might generate different results. Hence in the second step, the module's algorithm picker analyzes the output of both the algorithms and chooses the one which is most aligned with the user's preferences.

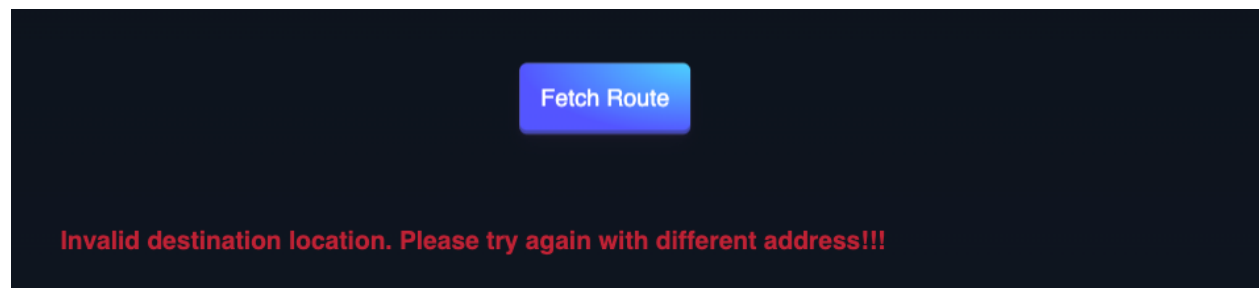
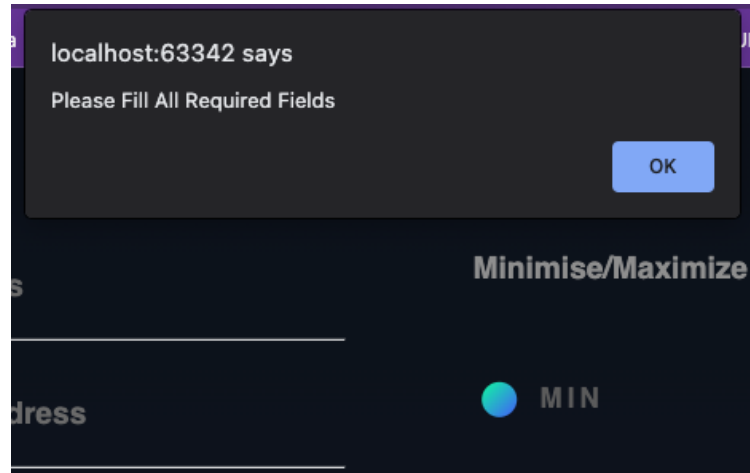
Error Handling-

- **Server side** - We have handled error handling in the server side by sending explicit status codes and detailed error messages along with the response. The error cases handled are - invalid addresses, location not in the same city, response timeout. Exceptions are also handled in our code. For logging we have used python's logger module.
- **Client side** - We have checked for input validation in the UI, and handled errors in cases of Directions API request failing. We display all response errors received from the server on the UI.

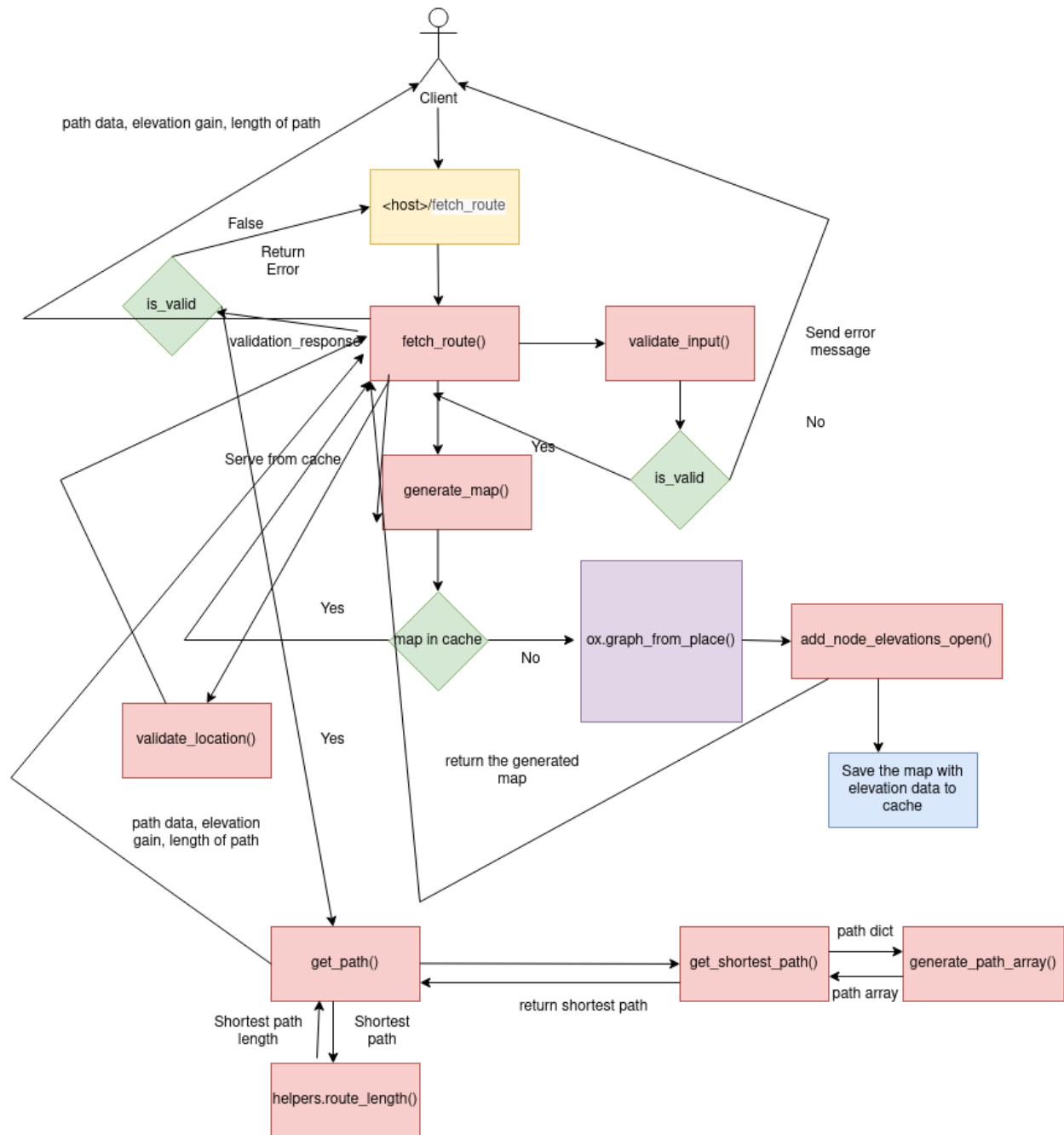
Examples :



Invalid input!!! Make sure to enter all fields



Control flow logic:



- Client connects to the server using a REST API call sending the user request over a post request.
- Server verifies the data and returns if the data is invalid (not in the required

format).

- Server forwards the request to map_generator module and fetches coordinates for the source and destination address.
- The city of the address is used to generate the map using osmnx map generator. The generator function first checks if the map for a particular city is already available or not. If map in cache serves from cache else generate a new map and cache it.
- Server fetches the nearest node for the source and destination coordinates and finds source and destination nodes nearest to the given location in the graph.
- Now the server calls algorithm selector to select the most optimized path finding algorithm for current input.
- The result (optimized path, elevation_gain, path length) is returned back to the server which returns it back to the client.
- Once the client gets the result it plots the coordinates on the map using google directional API.
- Any error occurred is captured by server and propagated all the way back to client

Testing:

The testing is done to ensure that the entire application works according to the requirements. We have added an exhaustive test suite to make sure that our application performs as expected and all corner cases are covered properly.

- Unit tests- server
 - The map generator, helper and pathfinder modules were unit tested for edge cases for both happy and broken flow. Map generator functionality was verified by testing if the application outputs a valid graph with correct number of nodes and edges, considering the google map values as the absolute truth. For the pathfinder module we tested the length and elevation of the output path to check if it's within an acceptable range. We have also tested the helper module functionality for valid address fields and route length and other functionalities.
- Integration tests- server
 - We have used a postman test runner to write and run integration tests for the server.
- UI testing:
 - For UI testing, we made 2 frontends, and asked our peers to try our application and give feedback on the UI elements. We also asked them

about our design choices and if they liked the way our UI looked. A friend of ours suggested the slider percentage changer which we changed after the feedback, it was a number field before.

- Manual Testing:
 - We tested our server using Postman using different locations and scenarios.
 - We tested edge cases and error situations manually as well.

Challenges

- Since the address formats for different locations are different, we had to try many publicly available geocoding APIs for fetching Latitude and longitude coordinates for a given location. After evaluating the results for multiple address formats, we found out that the Geocoding API provided by Google Maps Platform was most efficient and hence we stuck with this for our final implementation.
- Also, we observed that when max is selected for elevation gain, the response time for our server was pretty high and on further analysis, we found out that our path finder algorithm was taking a lot of time as it was exploring all possible paths without taking percentage length criterion into consideration. Hence, we have modified our algorithm to prune any path as soon as it exceeds the percentage length criterion without any further exploration.

Limitations

- One of the major limitations of our application is that we use osmnx map_from_place module to generate the graph. As osmnx is an open source application and runs on commodity servers, generally it fails to generate the map of an entire state like Massachusetts. Due to this our application is limited only to work if both source and destination are in the same city.

Future Scope and improvements

- Use a paid version of Google map api to generate graphs from place as google maps API is much more powerful and fast enough to generate graphs of entire cities in reasonable time.
- We can also try some kind of implementation for generating partial maps of source and destination to avoid the high costly operation of generating the entire state of huge cities.
- We can also add Grafana dashboards for monitoring the status of our application in real time.