Design Document

Group name - Kashikoi
Group members - Divya Sharma, Astha Baranwal, Rohan Acharya, Kavya Harlalka
Github link - https://github.com/kavyaharlalka/kashikoi-elena-navigation
Presentation link -
https://drive.google.com/file/d/1kYjDzBjgHvEUk565uJN3MAUknJmqa5kL/view?usp=share_link

Problem Statement

Navigation Systems typically provide the shortest path between any two given points. These are not optimal for a lot of scenarios where the user is interested in finding a path which has the least elevation gain. Moreover, some users might be interested in finding a path which has elevation gain so that they can partake in an intense and time-constrained workout. To extend navigation systems to solve the above problems, we have designed a software system that takes two points as input and finds the optimal route between them that maximizes or minimizes elevation gain while limiting the total path distance to n% of the shortest path between these two points.

For this system, we have considered the following functional requirements:

1. The system will take in a pair of points, one corresponding to the source and the other corresponding to the destination. The user of the system will also provide a choice between maximum or minimum elevation gain and a percentage of shortest path within which we will calculate the route. Finally, the user will select an algorithm out of 6 algorithms that are made available in our system. These are:
    a. Dijkstra's
    b. Bi-Directional Dijkstra's
    c. A*
    d. Bellman-Ford
    e. Goldberg-Radzik
    f. Floyd-Warshall

    The expected output of the system will be on the map showing the visual representation of the optimal route that we have determined for the user to travel from the source to the destination.

2. The system will check and validate the inputs that are entered into the application and will prompt the user in the case of unexpected input.
3. The performance of the system, in terms of how much delay there is from the point that the user presses "Go" till when the optimal path is rendered on the map was tested and satisfied.
4. We added an option to cache paths so that repeat search queries will perform much faster by retrieving data that is stored in the cache.
5. In case an API request fails or times out, the user should be appropriately notified.

6. The application stores the navigation history in a database for each request. When an API request is received on the system, it first queries the model to check if there is an already calculated route for the input parameters and returns the same if found.
7. The web-application should consist of a Help and About Page for better user experience.
8. The web-application consists of a reset button which allows the user to reset the form and the map in order to input a fresh query.

The non functional requirements that we have considered for the system are given below:

Understandability: The application has well documented and detailed user guide and functional documentation which enables a non-technical user to easily operate the application.

We have also auto-generated documentation using *Sphinx* for all the module docstrings with module description, input parameters, and return parameters. Sphinx makes it possible to create intelligent and streamlined documentations to increase understandability. The generated documentation is split into API and internal documentation so that developers can see exactly how the API will be used and how the modules are structured.

Readability: The code is properly supplemented with meaningful comments to make it lucid for developers and a README.md file is provided for further description on how to run the system.

Testability: The application has been thoroughly tested with unit as well as integration tests to test the functionality and any edge cases that can be encountered. Code coverage has also been evaluated to check what percentage of code has been tested.

Reliability: The application is responsive at all times and offers suggested routes within a reasonable time frame. It can handle multiple user requests and user load simultaneously without giving errors.

Compatibility: The application is compatible with all web browsers such as Google Chrome, Mozilla Firefox, Opera, Microsoft Edge etc.

Modularity: The backend code has been written in a modular design by creating separate folders for each of the model, controller and view in the MVC architecture.

Extensibility: The application has been developed in a manner that makes it easy to add support for enabling addition of more algorithms for finding the shortest route from origin to destination.

Usability: The UI has been made user-friendly and intuitive by making it easy for the user to navigate through the different options that we have provided in our application. The system is also usable in various locations and made available to all users, regardless of their locations.

Software Development Phases

Planning and Design Decisions

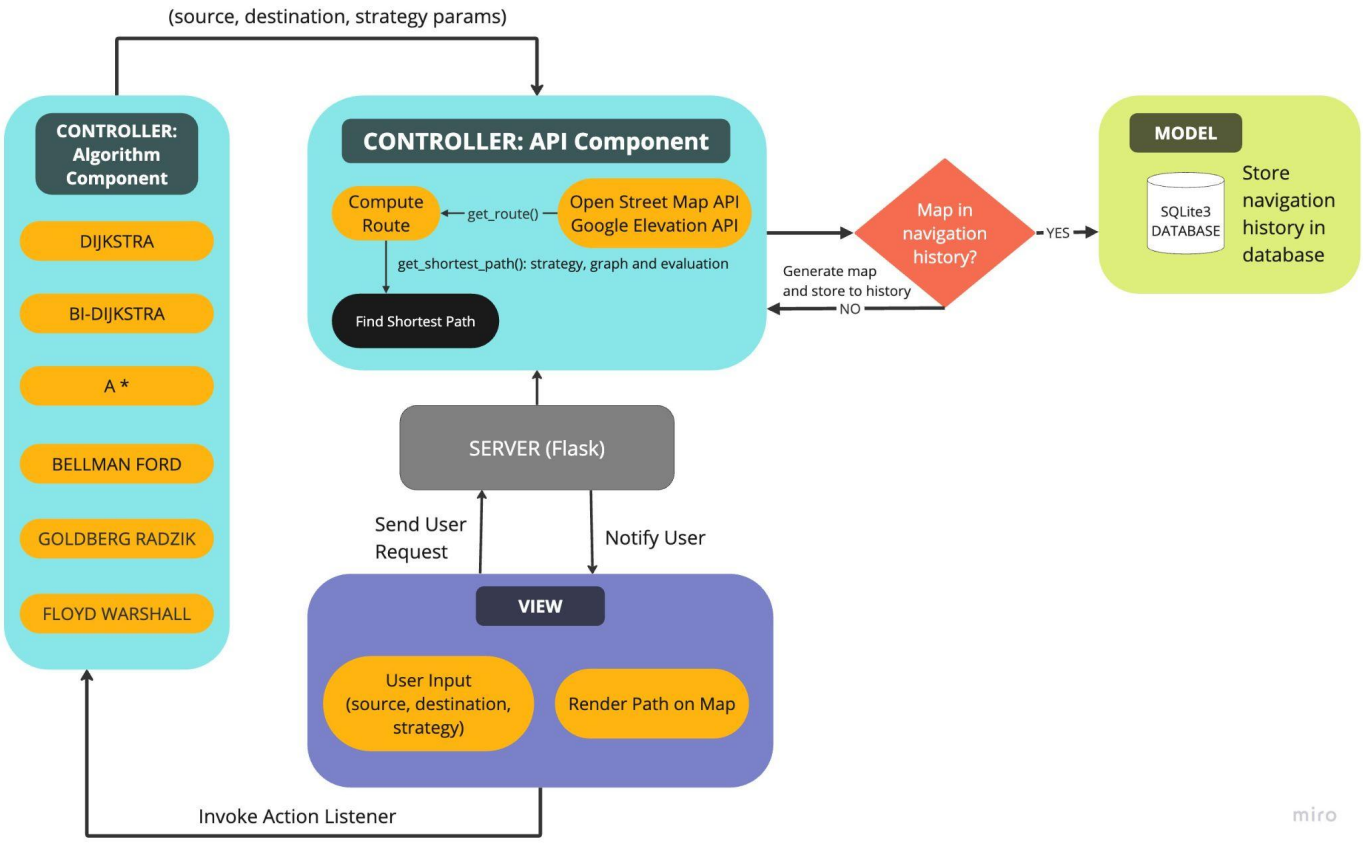
Choice of Server: Considering the scale of our application, we have chosen to use a flask server which is a lightweight and flexible framework for building web applications in Python.

Choice of API for getting Elevation Gain: The Google Maps Elevation API has been used for getting the elevation gain because of its accurate and comprehensive data, provided by Google's extensive data resources and algorithms.

Choice of API for Map Generation: The OSMnx library has been used for generating the map since it is a freely available and open source library which simplifies the process of accessing and analyzing OpenStreetMap data and is a valuable tool for map generation and network analysis. We have also used google.maps.DirectionsService and google.maps.DirectionsRenderer libraries for rendering paths on the map for UI.

Choice of Path finding Algorithms: We have chosen to present the user with an option to choose between Dijkstra's, Bidirectional Dijkstra's, A*, Bellman-Ford, Goldberg-Radzik, and Floyd-Warshall algorithms. The A* and Dijkstra's algorithms are primarily used for finding the shortest path from source node to a destination node, while Bellman-Ford, Goldberg-Radzik, and Floyd-Warshall algorithms are used for solving various forms of the shortest path problem for graphs. As we want to find an optimal path between two points, we have provided various algorithmic options for the user to try and visualize the different paths that are possible to be followed in order to make their final choice.

Design and Architecture



The above design has been proposed for our application. This follows both MVC (Model, View and Controller) and Client - Server architecture patterns.

Based on our requirements, we have determined that the optimal choice of languages are Python for the backend code and JavaScript, HTML and CSS for the frontend code.

Development and Implementation

Back-End

We have the following modules in our application:

1. Controller: The Controller uses the Google Maps Elevation API to retrieve data for the nodes with elevation characteristics and the Open Street API to create the graph. The Controller takes source, destination and route strategy details (like minimum/maximum elevation gain, algorithm, path limit percentage and transportation mode) from the user request. Then the coordinates of source and destination are computed, a graph object is created, and the shortest path is evaluated based on the cost functions between two

nodes using the route strategy parameters. Finally, the nodes of the best path route are obtained, and then the Controller sends the origin, destination and strategy data to the Model which in turn stores these in the navigation history database, if they are not already present in the database. Finally, the Controller notifies the view to show the appropriate route.

2. Model: The model checks if the map route entered is already present in the navigation history database. If it is not present, it stores the generated route in the SQLite navigation history database. We have used a sqlite database and modules to store our data. We are storing the final result as a json directly into the database so that we can fetch it and send it to the user directly. As soon as the request lands on the controller, it sends a query to the model seeking a record that matches the user inputted source, destination, algorithm id, path percentage, minimize elevation gain and transportation_mode. If such a record already exists, the model responds with the result of the said record so that the controller can directly send that to the user. If it does not exist, the model responds with an empty object letting the controller continue with the calculation of the best path. Once the calculation is done, the controller sends the same parameters again to the model this time with the result in a json format to store it. The model inserts the data into the database converting the json into text. Thus, it helps in saving computation cost, improving performance and reducing the memory usage of the application.
3. View:
   The View component will consist of the UI logic of the application and includes all logic of how the user of the Elena web-application will interact with the application.

    1. The UI consists of a Navigation bar for navigating to Home, About and Help page. Home Page consists of the main application consisting of the input form and a map to render the shortest elevated path on. Help and About page offer more information about the application and its usage manual.
    2. To get the shortest elevated path in the map, users will input the following details on the form provided:
        - The source and destination
        - Minimum or maximum elevation
        - Algorithm to be selected from the following options - Dijkstra's, Bi-directional Dijkstra's, A*, Bellman-Ford, Goldberg-Radzik, Floyd-Warshall
        - Path limit, which is the maximum percentage of the shortest route that we want to consider
        - Transportation mode: walk or bike

       Once the user clicks on the go button, it renders the map showing the path between source and destination which is computed by the controller.
    3. On clicking the reset button and in case of an error all the form fields and map are set to their default state.

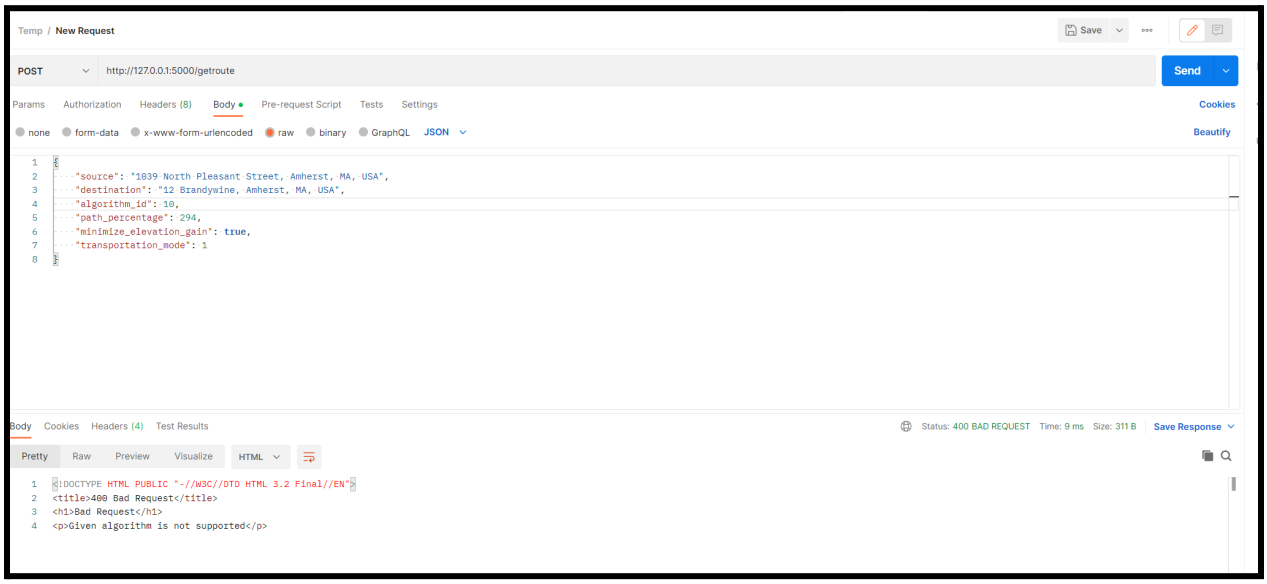
4. View module consists of two directories: static and template. Static Directory further consists of a css and js folder which consists of the css file for styling of the UI and script file for interactive behavior of the UI. Template consists of HTML files for the three pages ie. Home, About and Help on our web-application.

Error Handling

Back-End

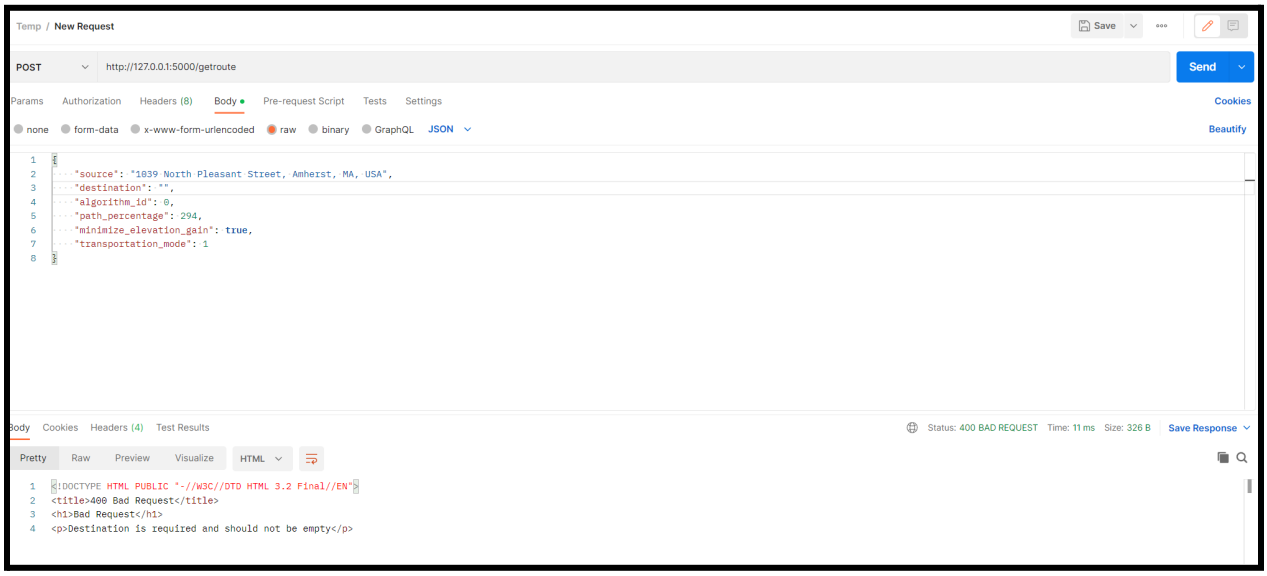
Controller:

- We are using multiple checks in the controller to ensure that before we fetch any shortest path data or query the model, we validate all the inputs coming into the controller.
- This is especially important since our controller is created not just for our internal usage but also as an API for users to use with their own UI.
- As soon as a request lands on our controller, we validate all the inputs - the inputs should not be empty or None, they should match the constraint type (for example, for source it should be string, for minimize elevation it should be boolean) and they should not have any invalid values (for example, the algorithm id can only range between 0 and 5 and the path percentage can only range between 100 and 500).



- If any of these validations fail, a Bad Request 400 error is immediately raised with the appropriate message for the API users to get to know what values they sent incorrectly.
- Also, all the functions/methods that the controller itself uses also have assertions for each of their arguments. This helps in ensuring that in case any error occurs in any of the functions, it can immediately be traced to that respective invalid value. It highly

impacts the debuggability of the application.



Model:

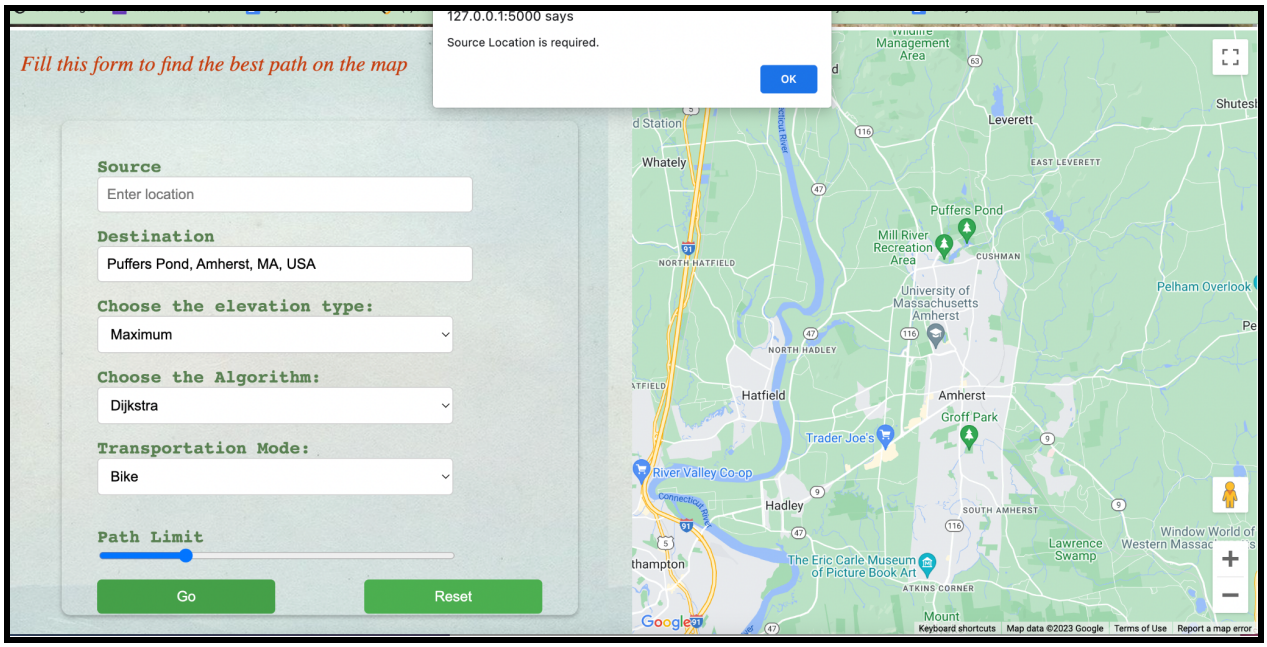
- It is essential to ensure that the database remains clean and does not have any garbage data in it
- To ensure that, we need to ensure that the data inserted into the database is constrained to their respective types and have valid values.
- Sqlite inherently provides a way to do this while creating the table, by mentioning the type of the column and any validation checks needed for it.
- Hence, we added validation checks for all the columns and in case any invalid value is sent to the database during insert query, it rejects the query with an error check letting the user know that the data is in an unexpected format.

Front-End

Along with the error handling in the back-end in API and Databases, we have added handlings of some of the error scenarios in the UI too as. Adding front-end validations complement the back-end validations provided by the API and contribute to better performance, user-experience, easier identification of errors and ensure more robust data processing.

Some of the Error cases handled :

1. Added validations in the Javascript that the source and/or destination location cannot be empty. If user presses go with empty locations, UI will show an informative alert and not proceed with calling the API.

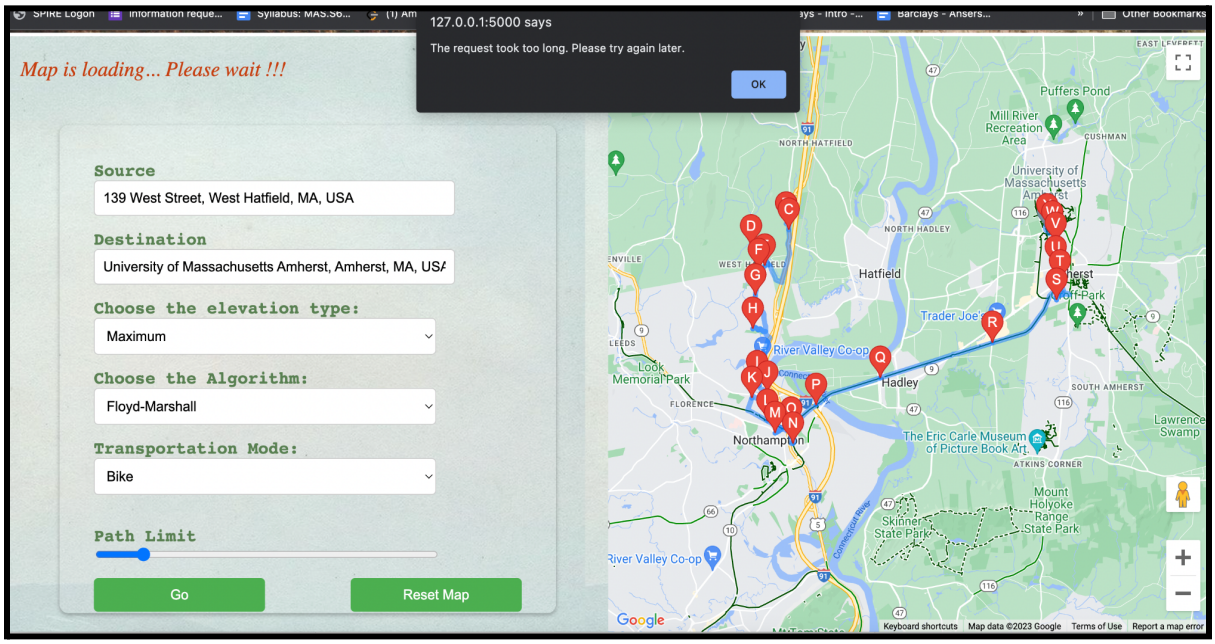


2. Ensured that input fields like Elevation type, Algorithm and Transportation mode are not empty by setting it to a default value in case the user doesn't select any other value.
3. For some reason if the Google.maps.DirectionsService and Google.maps.RendererService API fails to plot the coordinates obtained from the controller/api, because of unavailability or error, we have added all possible error case scenarios in the javascript to show the user specific error with the reason. We also reset the input and map after that for better user experience.
   For instance :

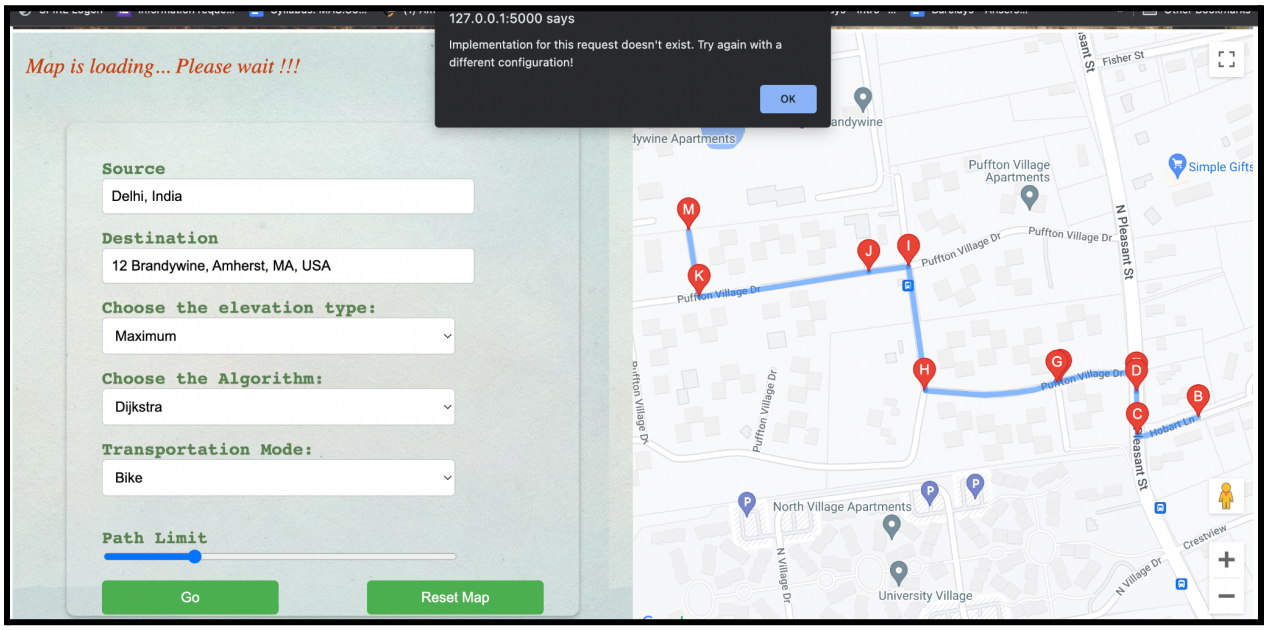
```
if (status === google.maps.DirectionsStatus.NOT_FOUND) {
    alert('Directions not found. Please check the provided locations.');
} else if (status === google.maps.DirectionsStatus.ZERO_RESULTS) {
    alert('No route could be found between the provided locations.');
} else if (status === google.maps.DirectionsStatus.MAX_WAYPOINTS_EXCEEDED) {
    alert('The maximum number of waypoints has been exceeded.');
} else if (status === google.maps.DirectionsStatus.INVALID_REQUEST) {
    alert('Invalid request. Please check the provided directions request.');
} else if (status === google.maps.DirectionsStatus.OVER_QUERY_LIMIT) {
    alert('The page has exceeded its query limit for the Directions API.');
} else if (status === google.maps.DirectionsStatus.REQUEST_DENIED) {
    alert('The page is not allowed to use the Directions API.');
} else if (status === google.maps.DirectionsStatus.UNKNOWN_ERROR) {
    alert('An unknown error occurred while requesting directions.');
} else {
    alert('An error occurred with the Directions API.');
}
console.error('Some error occurred via Google Maps API' + status);
reset();
```

4. Added the timeout feature in case the back-end api doesn't respond with the shortest path coordinates in time. This will ensure a good user experience. Based on our testing with different algos, locations and other scenarios, we have put a timeout after 30

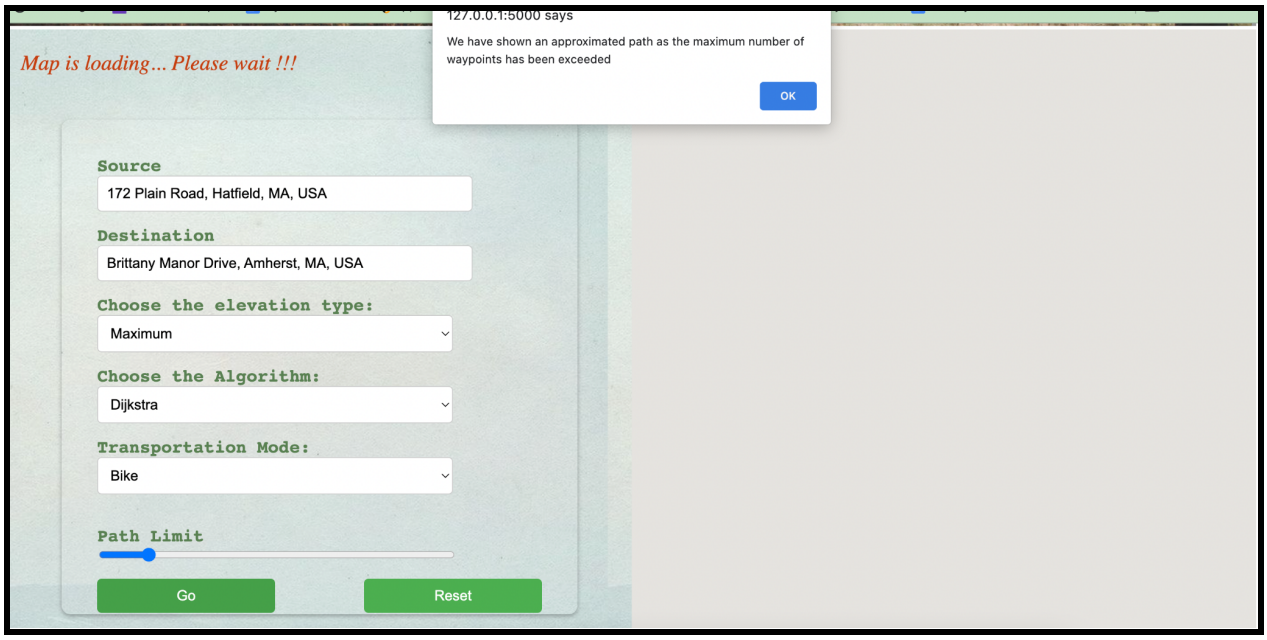
seconds of wait. In case the backend doesn't respond within this time-frame, we have provided the user with an alert saying "The request took too long. Please try again later."

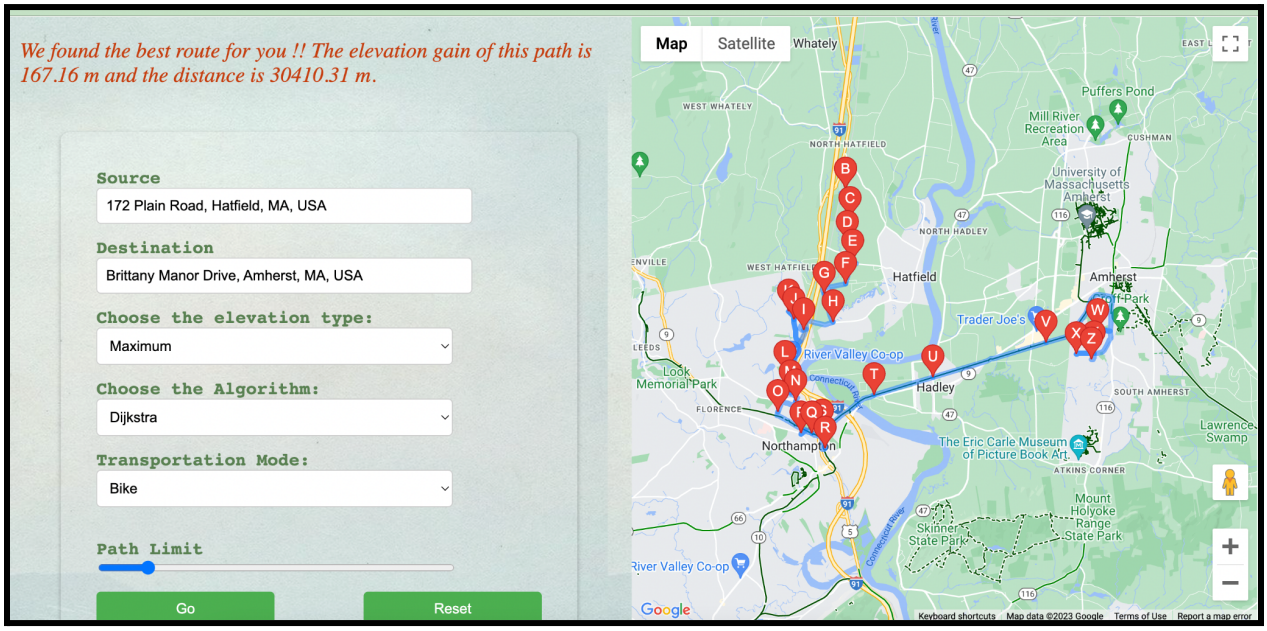


5. In case of any error from the back-end api, we are informing users via alerts and resetting the map and form fields once the user clicks okay on the alert.

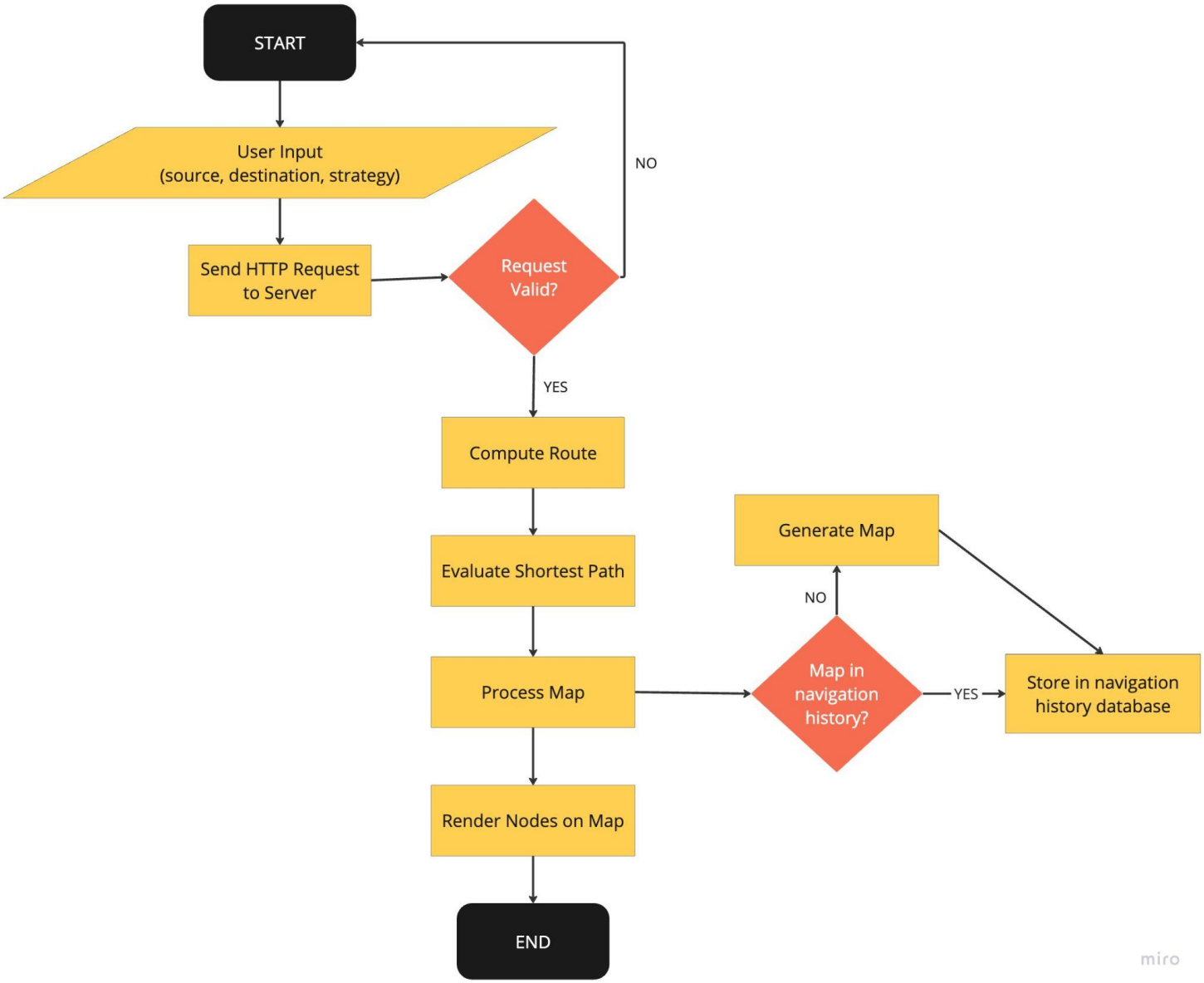


6. In case that the source and destination points entered are such that the number of nodes between them are very large (number of path coordinates >25), our application cannot calculate the exact path since we are using the free version of Google Maps Elevation API. In this case, we have handled this by computing an approximate path for this distance and added an alert for the user about the same!


127.0.0.1:5000 says
We have shown an approximated path as the maximum number of waypoints has been exceeded
OK
Map is loading... Please wait !!!
Source
172 Plain Road, Hatfield, MA, USA
Destination
Brittany Manor Drive, Amherst, MA, USA
Choose the elevation type:
Maximum
Choose the Algorithm:
Dijkstra
Transportation Mode:
Bike
Path Limit
Go
Reset


We found the best route for you !! The elevation gain of this path is 167.16 m and the distance is 30410.31 m.
Source
172 Plain Road, Hatfield, MA, USA
Destination
Brittany Manor Drive, Amherst, MA, USA
Choose the elevation type:
Maximum
Choose the Algorithm:
Dijkstra
Transportation Mode:
Bike
Path Limit
Go
Reset
Map
Satellite
Whately
Puffers Pond
Mill River Recreation Area
University of Massachusetts Amherst
Hatfield
Amherst
Trader Joe's
River Valley Co-op
Look Memorial Park
Hadley
Northampton
The Eric Carle Museum of Picture Book Art
Mount Holyoke Range State Park
Skinner State Park
Lawrence Swamp
Google

Control Flow Logic



- Client uses a REST API call to connect to the server sending a user request over a post request.
- Server checks if the data is invalid i.e., all the request parameters are in correct format, if there is any invalid data entered in any of the input fields etc
- If the request is valid, the route is computed between the source and destination nodes.
- The shortest path or the best path is evaluated based on cost functions between two nodes using the route strategy parameters like minimum/maximum elevation gain, algorithm, path limit percentage and transportation mode.

- The map is processed. If the map is not in navigation history, the map is generated and stored in history.
- The output nodes are populated on the map from the source node to the destination node.

Testing

Testability is important to keep our software smooth and free from bugs. We have tested the entire application to ensure that it works according to the specified requirements. We have written comprehensive tests to ensure that all the functions and modules are tested and all corner cases are covered.

1. UI Test: We have also performed UI Design Review and have attempted to incorporate good UI Design Principles like Schneiderman's 8 golden rules in order to make the application more understandable, user-friendly and aesthetically pleasing. A few key points that we have considered while designing the UI are maintaining consistency, error handling, providing visual feedback etc. More details on these are shared in the evaluation document.
2. Manual Test: We have tested edge cases and bugs manually. We have used Postman as well as the UI for testing various situations and source and destination points on the map. A few of the edge cases that we have covered are for scenarios where the source and destination locations are the same, when a user enters a location which is outside the scope of our application and when the server takes longer than 30 seconds to handle a request.
3. Performance Test: We have tested the performance of the APIs used in the application from the response-time perspective. We observed the response time taken by the API for different scenarios like response time for different algorithms with minimum and maximum elevation gain and response time for different destinations from a particular source and algorithm with minimum and maximum elevation gain.
4. Unit Test: Using pytest and pytest-cov, we have ensured 100% line and branch coverage for our controller and model code. Unit tests are used to test the individual units (methods) in the application. Hence, they are mainly used to test the helper methods and model methods. We have used integration tests to test the API methods.
5. Integration Test: We have tested the getroute API which returns information about the optimal path from the server. We have checked for positive cases as well as illegal cases where our application will not be able to display the optimal route between the source and destination points.

Challenges

We faced various challenges in the development of this project.

1. From the UI development point of view, plotting of the path coordinates between source and destination location was challenging as it involved a learning curve in understanding Google.maps.DirectionsService and Google.maps.RendererService APIs.
2. To increase the usability of the application to get the best elevated path between two points with more than 25 coordinates in the best path, we added the logic to show an approximate path. It was tough to write its implementation.
3. In order to make the UI more intuitive, user-friendly and visually appealing, we had to constantly make changes in the UI.
4. We had to optmize the algorithms like Bellman Ford to be able to give response time within the time limit of our request timeouts.
5. We faced the issue of long download times of initial graph if it did not already exist in the working directory, hence we decided to cache it and use the cached graph directly.

Limitations

- We are only able to generate the accurate shortest path between points that are reasonably close since the free version of Google Maps API is not as powerful. For paths with more than 25 coordinates and within the 30 km radius of Amherst, we have approximated the shortest path.
- We have used the OSMnx library to generate the graph which relies on OpenStreetMap (OSM) data. Since this is an open-source application, this may not be complete or up-to-date in all regions. This is primarily contributed by volunteers and hence the accuracy of the data is not reliable.

Future Work

- We can use a paid software for Google Map API to show the optimal routes for source and destination points that are very far from each other. This would benefit our application and increase its functionality significantly.
- User authentication and authorization for better security
- Use more advanced algorithms and reduce the time complexity of the existing implementation of the algos.