# Software Requirements Specification

**Group name - Kashikoi**
**Group members - Divya Sharma, Astha Baranwal, Rohan Acharya, Kavya Harlalka**
**Github link - https://github.com/kavyaharlalka/kashikoi-elena-navigation**

## Introduction

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.

The purpose of this SRS document is to describe and define the requirements of our project. It serves as a communication tool between stakeholders such as clients, developers, and testers, to instill a shared understanding of the goal of the software application.

The intended audience for our application consists of a mix of sports practitioners, tourists, bikers, hikers, runners, racers, etc. who have additional requirements over traditional navigation systems. If these customers want to optimize their travel or workouts by consuming more calories or selecting a certain type of mode of travel, they can do so using our application.

## Functional Requirements

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. The Floyd-Warshall algorithm is deemed to be computationally expensive hence

might not return a valid path within the given request timeout in which case a timeout message should be shown to the user.

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. Accuracy: The path that has been calculated by the application, based on the source and destination input by the user, has to be accurate. This should be in terms of the mode of elevation, the path limit as well as the shortest distance between the two points that are chosen by the user. The accuracy of the UI and how the optimal path is rendered on the map should also be taken into account.
4. Performance: 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. 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.

# Non Functional Requirements

**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.

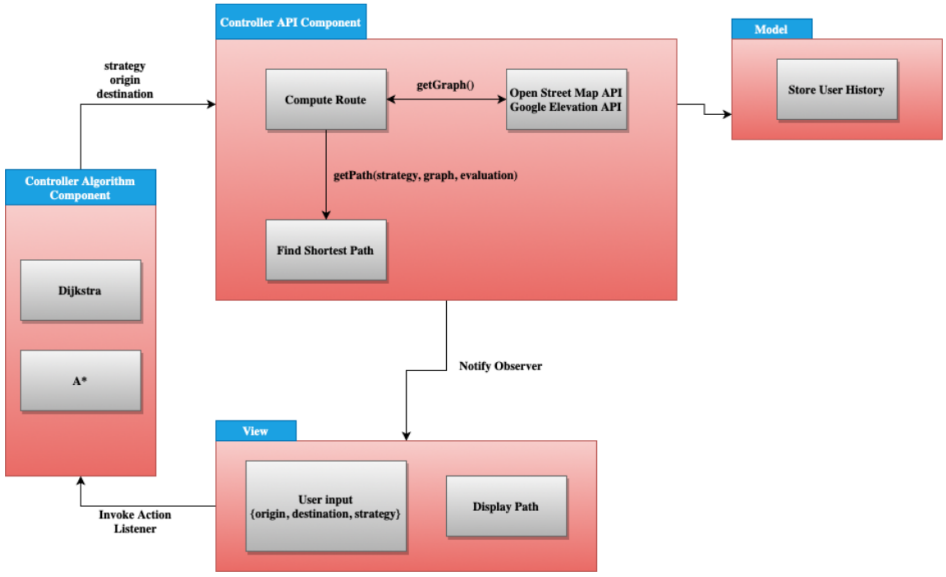
**Compatibility:** The application is compatible with all web browsers such as Google Chrome, Mozilla Firefox, Opera, Microsoft Edge etc.

**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.

# System Architecture




# External Interfaces

1. Google Maps API:

- Purpose: The application will interact with the Google Maps API to retrieve addresses and display paths on the user interface.

- Functionality:

a. Address Retrieval: The application will send requests to the Google Maps API to obtain addresses based on user input (source and destination addresses).

b. Path Display: The application will utilize the Google Maps API to render paths on the user interface, showing the calculated routes.

- Integration Requirements:

a. API Key: The application must have a valid API key to authenticate and access the Google Maps API services.

b. Request and Response Formats: The application should adhere to the specified request and response formats defined by the Google Maps API.

2. OpenStreetMap API:
- Purpose: The application will interact with the OpenStreetMap API to retrieve graphs, find nearest nodes, and calculate shortest distances.
- Functionality:
    a. Graph Retrieval: The application will send requests to the OpenStreetMap API to retrieve graph data for path calculations.
    b. Nearest Nodes: The application will utilize the OpenStreetMap API to find the nearest nodes to specific addresses or coordinates.
    c. Shortest Distance Calculation: The application will use the OpenStreetMap API to calculate the shortest distance between two nodes on the graph.
- Integration Requirements:
    a. API Access: The application should have proper access and authentication credentials to utilize the OpenStreetMap API services.
    b. Request and Response Formats: The application should follow the required request and response formats specified by the OpenStreetMap API.

# Use Cases

Given below are some of the high level use cases of the system :-

1. Plan Route with Elevation Consideration:
    - User enters the source and destination addresses.
    - User selects the desired mode of transportation (biking or walking) and any specific elevation preferences.
    - The Elevation Navigation System calculates the optimal route, considering elevation data and the user's preferences.
    - The system presents the recommended route, highlighting any elevation changes or constraints to the user.
2. Save and Retrieve Paths:
    - User selects a route and its associated elevation information.
    - The Elevation Navigation System saves the path as a pre-existing path.
    - The system stores the route and elevation data as a JSON string in the database for future retrieval.
    - Users can later retrieve their saved paths by selecting the corresponding pre-existing path, avoiding recalculation.

# User Stories

Given below are some of the high level use cases of the system :-

1. As the user, I want to be able to consider the elevation gain while planning a route between any two locations.
2. As the user, I want to be able to select a location from a dropdown box.

3. As the user, I want to be able to choose if I want to maximize or minimize elevation gain while planning a route between any two locations.
4. As the user, I want to be able to limit the paths to n% of the shortest path between any two points.
5. As the user, I want to be able to choose which algorithm I want to use.
6. As the user, I want to be able to choose what transportation mode I prefer to use.
7. As the user, I want the optimal route to be visible on the map.
8. As the user, I want the elevation gain and distance information for the optimal route to be visible.

# Data Requirements

These data requirements outline the specific entities, attributes, and relationships involved in storing and processing the user input and path results in the application's SQLite database.

1. Data Entities:
   a. User Input:
   - Source Address: The address provided by the user as the starting point.
   - Destination Address: The address provided by the user as the destination point.
   - Algorithm ID: An identifier specifying the algorithm to be used for path calculation.
   - Path Percentage: A numerical value representing the percentage of the path to be considered.
   - Elevation Mode: A mode indicating how elevation data should be taken into account during path calculation.
   - Travel Mode: A mode specifying the type of travel (e.g., walking, biking) for path calculation.

   b. Path Result:
   Path JSON: The final result of the path calculation stored as a JSON string. This JSON string can be used later to retrieve a pre-existing path.

2. Attributes and Relationships:
   a. User Input:
   - Source Address:
     - Attribute: Address string.
   - Destination Address:
     - Attribute: Address string.
   - Algorithm ID:
     - Attribute: Identifier (numeric).
   - Path Percentage:
     - Attribute: Numeric value.
   - Elevation Mode:
     - Attribute: Mode identifier ("min", “max”).
   - Travel Mode:

- Attribute: Mode identifier ( "walking", "biking").

b. Path Result:
- Path JSON:
- Attribute: JSON string containing the path information, such as waypoints, distances, and directions.

3. Data Storage:
- SQLite Database: The application will use a SQLite database to store the user input data and the corresponding path result. The database should support storing text (string) data, including the JSON string format.

## Assumptions

- Approximate paths: Even with approximate paths, the user is able to use the application and follow the shown path on bike or on foot.
- Weather conditions: The application does not take into consideration weather conditions in case certain routes are affected by weather. It assumes that users will check the weather conditions before embarking on a certain route.
- Route closure: The application is not aware of route closures due to any unforeseen reasons. It assumes that the user will check for any such circumstances before following a certain route.
- User Fitness and Ability: The application assumes that users have a reasonable level of fitness and ability to navigate their chosen mode of transportation, whether it is biking or walking.

## Constraints

- The application is currently limited to a 30km radius from 1039 N Pleasant St, Amherst.
- The application has a limited capability to show 25 waypoints which is a constraint of the Google Map API. If the number of nodes in the best path is more than 25, the path shown is approximated.
- The Floyd Marshall algorithm is computationally very expensive and hence it frequently might fail to give a response within the 30 seconds time constraint of a response timeout.

## StakeHolders

- End-users: Runners, bikers, hikers, tourists, fitness enthusiasts, event organizers, researchers, rescue teams.
- Developers: The team responsible for designing and developing the EleNa application.
- Administrators and IT Support: The team responsible for maintaining and updating the application.

# Glossary

- Elevation Navigation System: The software system that provides navigation functionality considering elevation data to assist users in selecting paths suitable for biking or walking.
- Elevation Mode: A user-selectable option that determines how elevation changes whether the aim is to minimize or maximize elevation.
- Route Planning: The process of determining the optimal path between a source and destination address, considering elevation data, user preferences, and mode of transportation.
- Algorithm: The “formula” for calculating the best route as per the constraints given as an input by the user.