UNIVERSITY OF BUCHAREST

FACULTY OF
MATHEMATICS AND
COMPUTER SCIENCE

SPECIALIZATION ARTIFICIAL INTELLIGENCE

Master's Thesis

MAP NAVIGATION APP

Graduate
Mihai-Cristian Popa

Scientific coordinator
Lect.dr. Florentina Suter

Bucharest, September 2025

Abstract

We developed a full-stack, cross-site web application that computes an optimized round-trip route through user-selected tourist attractions. Built with HTML, CSS, JavaScript, and Node.js, the system integrates LeafletJS and OpenStreetMap for interactive mapping, and uses cloud-hosted MongoDB for caching and authentication. The core challenge—and the centerpiece of this project—was route optimization. We engineered a custom search framework capable of handling up to 24 way-points efficiently. To this end, we designed a sandbox environment for testing different algorithms, implemented three distinct approaches (brute force, nearest neighbour, and 2-opt), and evaluated them with real use cases. Our final algorithm combines brute force for up to five way-points and a 2-opt heuristic seeded with the nearest neighbour solution for larger sets—balancing optimality with speed. Location data and routing are supported by geocoding and matrix APIs from Mapbox and LocationIQ, chosen dynamically based on usage limits. Additional data is retrieved from Overpass and Wikidata to enrich the user experience. Secure cookie-based authentication ensures session integrity.

Rezumat

Am implementat o aplicație web cross-site care calculează un itinerar circular optim pentru atracțiile selectate de către utilizator. Construit cu HTML, CSS, JavaScript și Node.js, sistemul integrează o hartă interactivă cu ajutorul LeafletJS și OpenStreetMap și folosește MongoDB hostat în cloud pentru autentificare. Optimizarea de rute a fost sarcina principală a proiectului. Am implementat o abordare de căutare capabilă să gestioneze eficient până la 24 de atracții turistice. În acest sens, am creat un mediu de testare a diferiților algoritmi, am implementat trei abordări distincte (abordarea brută, euristica celui mai apropiat vecin, euristica 2-opt) și le-am evaluat folosind date reale. Algoritmul nostru final combină abordarea brută, în cazul în care sunt până la 5 atracții, și euristica 2-opt folosind ca tur inițial rezultatul euristicii celui mai apropiat vecin în cazul în care trebuie integrate în itinerar peste 5 atracții. Astfel, am obținut un echilibru între optimizare și viteză. Coordonatele, matricile și rutele sunt obținute folosind API-uri de la Mapbox și LocationIq. Informații adiționale sunt extrase folosind Overpass și Wikidata pentru a îmbunătăți experiența utilizatorilor. Integritatea sesiunii este asigurată prin autentificare bazată pe cookie-uri securizate.

Contents

1 Introduction **7**

2 Used Technologies **9**

2.1 Front-end 9

2.2 Back-end 10

2.3 External APIs 11

3 Application Architecture **13**

4 Implementation **17**

4.1 Front-end 17

4.1.1 Structure 17

4.1.2 HTML 18

4.1.3 JavaScript 18

4.1.4 Styling 21

4.2 Back-end 22

4.2.1 Structure 22

4.2.2 Middleware 23

4.2.3 Authentication 23

4.2.4 Search and Route 25

4.2.5 Search Solution Optimization 27

4.2.6 Testing Endpoints 31

4.3 Database 32

4.4 Deployment 33

4.4.1 Front-end 33

4.4.2 Back-end 34

4.4.3 Adjustments 37

5 Usage Scenarios **39**

5.1 On page opening 39

5.2 Authentication 40

5.3 Search and Route 40
5.4 Reset Map and Logout 42

6 Conclusions **44**

Bibliography **46**

A Logging **50**

B Search Solvers Comparison **52**

List of Figures

3.1 Conceptual Architecture Diagram 14
3.2 User opens up the client. 16

4.1 User opens up the client. 32
4.2 Front-end GitHub Repository 33
4.3 GitHub Pages setup 33
4.4 Deployed Application 34
4.5 Setting up the initial Render deployment: (Top) Settings for Building and Running the Back-end, (Bottom) The Project View 35
4.6 Render Environment Variables 35
4.7 Render Logging Capability 35
4.8 Connecting Render Logs with New Relic: (Top) Created the New Relic API Key for Render, (Bottom) New Relic set as default logger for Render 36
4.9 New Relic Logging Capability 37

B.1 The global optimal route obtained with the brute force algorithm with the duration of 00:59:40. 55
B.2 An optimal route obtained with both 2-opt and 2-optNN heuristics with the duration of 01:02:16. 56
B.3 The route obtained with the nearest neighbour heuristic with the duration of 01:03:44. 56
B.4 The route obtained with Nearest Neighbour for 15 way-points with the duration of 01:46:22. 57
B.5 The route obtained with Nearest Neighbour for 24 way-points with the duration of 02:38:48. 57
B.6 The route obtained with 2-opt for 15 way-points with the duration of 01:40:18. 58
B.7 The route obtained with 2-optNN for 15 way-points with the duration of 01:38:01. 58
B.8 The route obtained with 2-opt for 24 way-points with the duration of 02:29:44, maximum iterations of 1000 reached. 59
B.9 The route obtained with 2-optNN for 24 way-points with the duration of 02:22:13, maximum iterations of 1000 reached. 59

List of Source Code

1 JavaScript event handled using method from module in main.js front-end file. 20
2 JavaScript event handled using method from main.js front-end file. 21
3 CSS setting up the full screen map background. 22
4 Recursive search algorithm. 29
5 HTTP file used for testing the geocode endpoint. 31
6 Resolving the backend origin for API requests. 37
7 loggerHelper methods. 50
8 Usage of the loggerHelper methods in the controllers. 51

Chapter 1

Introduction

Efficiency is one of the most important words today. Most people strive to be as efficient as possible in their daily life. For people who want to take efficiency to the next level during vacation, we want to provide an application that can create a closed tour with the attractions that a user wants to see. The purpose of the application is for the user to input some points of interest, click a button, and then to find out the order in which he can visit those attractions, the total driving time to reach all the way-points and the time needed for each of the complete route steps. Moreover, for each of the attractions on the map, the user will be able to see some useful information, such as: a representative image of the attraction, the opening hours, the official web-site and contact details. Earlier, we have mentioned that the application generates a closed tour, which means that the first way-point added will be treated as both the starting and ending point of the generated route. The user also has the possibility to share his current location, and that can be treated as the starting point of his journey.

The approach for this task has been to build a web application, with the purpose in mind to have this tool available across different devices. To have the application available for different users, we needed some form of deployment. The building blocks of the application are:

- Map display;
- Finding out the coordinates of the attraction based on the name;
- Allowing the user to select attractions;
- Showing the selected attraction;
- Computing and displaying the efficient route;
- Fetching and displaying extra details about the attraction;

For some of the building blocks we are using some external *APIs* which have some usage limits that we have to consider. For this reason, before deploying the backend, we had to add checks to see if the request is made by an authenticated user. This was needed to protect the connections exposing the external *APIs* by applying some per user and total limits.

After getting the coordinates of the attractions and the matrix of durations between the way-points (by using back-end proxies created around *LocationIq or Mapbox, Geocoding and Matrix APIs*), we are going to use our own search algorithms to find the most efficient closed tour.

For optimizing our search approach we have created a sandbox for testing different search approaches, implemented three search algorithms (the brute force approach, 2-opt and nearest neighbours), and compared their behaviour under similar circumstances as the ones needed in the application. Based on the knowledged gained from these tests we have chosen the best combination of the algorithms, such that users can get an optimal tour, fast and with up to 24 way-points. Our search approach involves using the brute force search for requests with 5 or less way-points, and the 2-opt heuristic seeded with the tour resulted from the nearest neighbours approach.

After ordering the way-point coordinates we make another request (to *Mapbox Directions*) for obtaining the LineString *GeoJSON* that will be displayed on the map as the route.

For each of the attractions, extra details will be fetched, using the open-data resources provided by *Overpass and Wikidata*.

The task of finding the efficient route between a number of input points is a popular computer science task which carries the name of **Travelling Salesman Problem** (TSP). Obtaining the best route in a fast manner every time becomes increasingly more difficult with the growing number of input points. This is why in practice, a good enough route is chosen over the perfect route, as computing the perfect route would take too much time [1].

For our use case, the maximum number of way-points that can be handled is twenty-four (as the *APIs* can handled at most twenty-five, but the first way-point will be added again as the twenty-fifth). Still most users will probably never input even half of that, as the scope of the application is for users to create a tour that can be handled in at most a day, so our main focus, in terms of user experience, is for the smaller way-point counts, but we have provided support in terms of search algorithms for the maximum count of way-points.

The following topic will be **Chapter 2 (Used Technologies)**, where we are going to present the technology components that will be referenced later in this work. **Chapter 3 (Application Architecture)** comes next with an overview of the interactions between the different components of the application. After that we have added **Chapter 4 (Implementation)**, where we deep dive in the technical details of the application, followed by **Chapter 5 (Usage Scenarios)** where the main features of the application are presented from a user point of view. The last topic from this work is **Chapter 6 (Conclusions)**, where we take a look at what is done and how this project can be improved in the future.

Chapter 2

Used Technologies

Visual Studio Code (VSCode)[1] is a lightweight code editor that can be used for a wide range of programming language. It comes with integrated Git support, the possibility of adding a variety of extension with ease to customize your experience, it provides an integrated terminal and powerful debugging tools. Behind the scenes, VSCode runs a core that is open source [2].

Git is an open source distributed version control system designed to handle both small and large projects efficiently [3].

GitHub is a tool providing code hosting by using *Git* commands. GitHub allows you to create repositories which are storages that can be used for development projects [4].

HTTP (Hyper Text Transfer Protocol) is used for the communication between client computers and web servers. Clients make HTTP requests and receive HTTP responses [5].

HTTP cookies represent a piece of data that is sent from the server via a *HTTP* response to the web browser of the user. Web cookies can be used to check the user authentication status [6].

GeoJSON is a standardized way of encoding geographic data structures [7].

API (Application Programming Interface), is a set of rules that creates a bridge that can be used to connect two different systems [8].

2.1 Front-end

HTML (HyperText Markup Language) is the programming language used for defining the structure of a web page [9].

CSS (Cascading Style Sheets) is a stylesheet language that is used to describe how the content of a web page, defined in *HTML* should behave [10].

JavaScript is a prototype-based, garbage-collected, dynamic typed programming

[1] https://code.visualstudio.com/

language which supports multiple paradigms such as imperative, functional and object-oriented [11]. Moreover, JavaScript is an umbrella term with multiple components, where the core functionality is actually ECMAScript, which can be used in non-browser environments such as *Node.js*, but also includes DOM (Document Object Model) *APIs* [12].

ES Modules, represent pieces of exportable JavaScript code, that can be imported into other ES Modules. The import/export functionality is very helpful for separation of concerns in *JavaScript* developments. It is especially helpful in front-end development as this kind of functionality was already there in *Node.js* [13].

Leaflet is the leading open-source *JavaScript* tool for interactive maps. It has good support accross platforms be it desktop or mobile, and it is very customizable [14].

OpenStreetMap (OSM) is an open data tool which provides map data for thousands of websites mobile apps and hardware devices. OpenStreetMap provides the possibility of displaying a map using pre-rendered tiles, as described in [15]. The map tiles from OpenStreetMap can be used alongside *Leaflet* for displaying a functional map.

FontAwesome is an open source provider of icons for web-development [16].

LiveServer is a *VSCode* extension, which can be used for serving a local server from a static or dynamic web page with live reload [17].

GitHub Pages is a tool used for hosting static pages, by extracting the *HTML*, *CSS*, and *JavaScript* code from an associated repository, than optionally running a build process and finally publishing the website. There can be only one deployment of this kind for a repository [18].

2.2 Back-end

Node.js is an open-source and cross-platform *JavaScript* runtime environment which allows developers to create servers and web applications [19].

Npm is a very important tool that comes alongside the *Node.js* installation. It is the largest registry of software, containing over 800 thousand packages of code. Npm is also free to use and it comes with a command line interface used for installing packages that aid in *Node* development[20].

CORS (Cross-Origin Resource Sharing) is a mechanism using *HTTP* headers to allow at the server level requests from indicated origins [21].

The used *npm* packages, are as follows:

- **express** is a framework for building web servers with *Node* [22];
- **cors** can be used alongside express as a middleware that sets up some *CORS* configuration [23];
- **cookie-parser** provides a middleware that extracts the *cookies* from the header of the request [24];
- **dotenv** loads the environment variables declared inside the .env file [25];

- **winston** is a flexible logging tool that can be used for logging in the console but also for saving logs in a local file or on a remote database [26];
- **axios** can be used to make *HTTP* requests [27];
- **bcrypt** can be used for hashing passwords [28];
- **country-state-city** can be used for extracting the world's countries with their associated ISO[2] country code [29];
- **simplify-geojson** can be used to reduce the dimensionality (based on a tolerance parameter) of a GeoJSON Feature Collection. This might be needed when facing different API limitations [30].

NoSQL databases are non-relational, so there are no tables that need to be linked. This kind of approach provides the flexibility needed for data storage in rapid changing development scenarios.

MongoDB is a scalable emphNoSQL database, which uses a document based model for storing its data, which also makes it very flexible [31].

MongoDB Atlas is a service that can be used for creating a cloud *MongoDB* kind of database [32].

Render is a cloud provider that can be used for building and deploying applications [33].

New Relic is a cloud provider that can be used for storing logs and querying logs and can be connected to *Render* for extracting application logs [34].

REST (Representational State Transfer) is an architecture used for defining *APIs*, where the *HTTP* protocol is used for communication between components.

REST Client is a tool that can be used for simulating *HTTP* requests and reviewing the *HTTP* responses, or even reusing parts of a previous response as an input to another request [35].

2.3 External APIs

External *API* Providers used are as follows:

- **LocationIq**, with the following endpoints:
  -> **Search / Forward Geocoding** which can be used for converting attraction names or addresses to coordinates [36].
  -> **Matrix** computes the duration of the fastest route for all the pairs of coordinates sent as input, and returns the durations matrix, distances matrix can also be returned [37];
- **Mapbox**, with the following endpoints:
  -> **Optimization** returns the duration optimized route for the input coordinates. The *API* provides some optional parameters that can be used to for setting

[2]International Organization for Standardization.

the first pair of coordinates as start and end point. The maximum number of input coordinates is 12 [38].

-> **Matrix** similar to the *LocationIq Matrix API*, but with a mentioned limit of maximum 25 coordinate pairs that can be sent as input [39];

-> **Directions** provides a routing for the inputted coordinates that can be displayed on a map. The *GEOJson* response of this *API* is compatible with the *Leaflet* map. This endpoint can handle at most 25 input coordinates [40];

-> **Geocoding** is similar to *LocationIq Search / Forward Geocoding*, but for this one we can send as extra input a country code for limiting the search with query to a specific country [41];

-> **Static Images** can be used for creating static map images, which can be enhanced with GeoJSON Feature Collections. This way search algorithms can be tested, and their outputs visualised [42].

- **Overpass** is a read-only *API* that can be used for extracting specific data from the *OSM* map data [43]. Generally, important touristic points from *OSM* also have an *Wikidata* id attached;
- **Wikidata** is a free knowledge base, from which we can read information about points of interest [44];

Chapter 3

Application Architecture

This Map Navigation App is a cross-site application, meaning that we have developed a user-facing client and a back-end server and these two will be deployed on different domains. The high-level overview of the interaction between the components can be seen in Figure 3.1. Zooming in on the diagram, there are a few notes to be added:

- The *HTTP* requests between the front-end and the back-end are being checked with a *CORS* middleware;
- For each of the requests the database needs to be up. This is checked using a middleware developed by us;
- For each of the requests (other than the log-in and registration) the user needs to be authenticated (to contain a valid session id attached to the cookie header). This is checked using a middleware developed by us;
- All requests processed inside our back-end component are being logged at least once;
- Before making the requests from the Back-end to the *External APIs* we have implemented checks to see whether the user or the total limit for the requested resource has been crossed;

Next up we are going to detail the integration between components:

First, the flow of opening the client is presented in Figure 3.2. Whenever we check whether the user is authenticated, a back-end request is made where a lookup in the database takes place to find a log-in session with the id extracted from the cookie header of the request.

Second, there are the authentication flows, registration, log-in, log-out, and the authentication check, for each of these there is an endpoint defined at the back-end side, which is requested whenever a user does the associated action from the client.

Third, there are the application specific flows, as follows:

- Searching for an attraction: User adds a query and clicks on search, the back-end ***geocode endpoint*** is requested with the query set as parameter. A result might be cached or a request is made to *LocationIq Search/Forward Geocoding*. If the endpoint fails in providing suggestions, then the *geocode-fallback approach* is used.

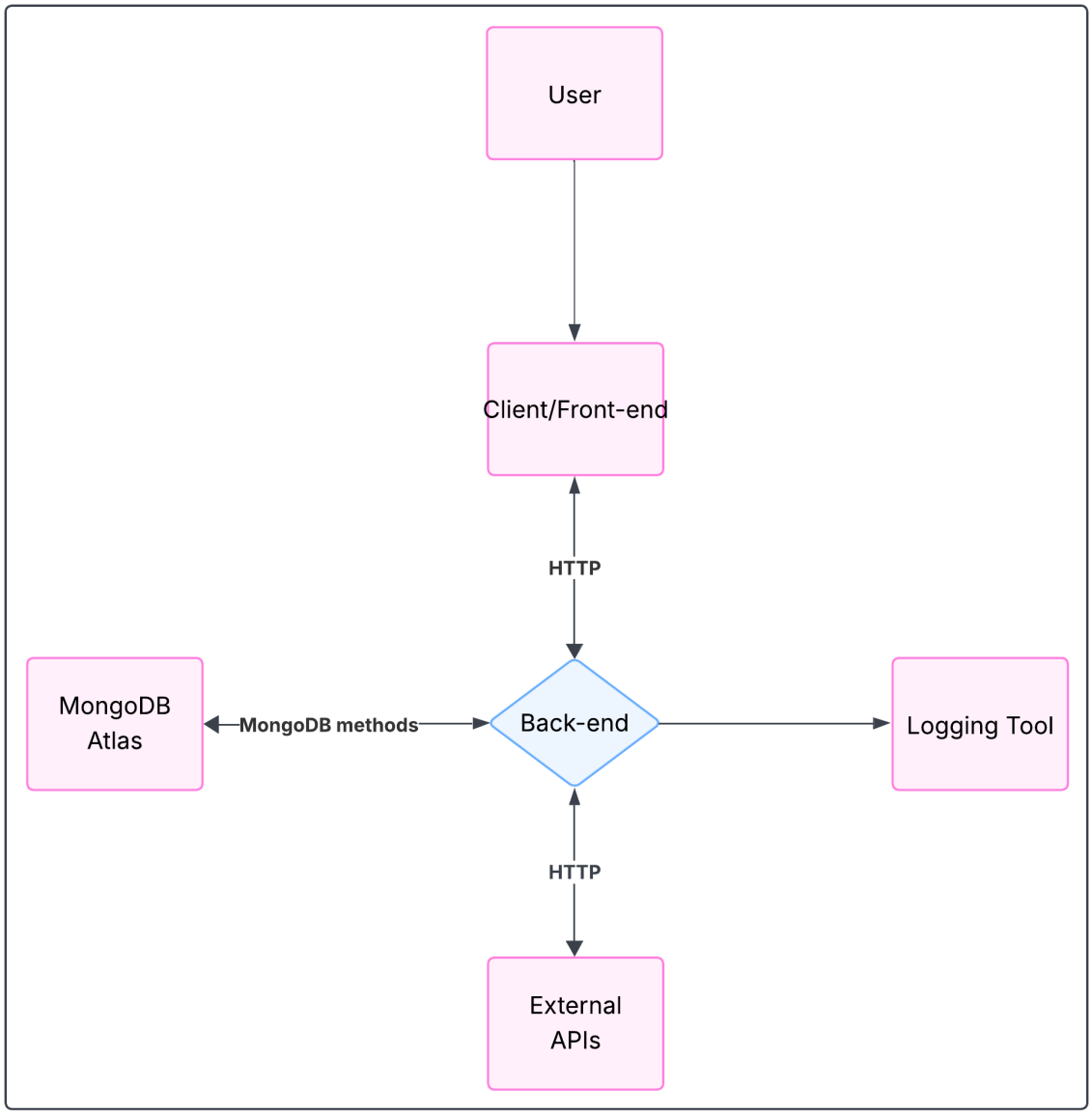

Figure 3.1: Conceptual Architecture Diagram

- Selecting one of the suggestions: User already searched for an attraction and was provided a list of possible matches for his attraction, here there are two possible cases, as follows:

  -> If one of the suggestions is a match with the query, after the user selects it, a back-end request is made to the ***extra-details endpoint*** with either the *OSM* id and type (when selected suggestion is from *LocationIq Search/Forward Geocoding*) or *Wikidata* id (when selected suggestion is from *Mapbox Geocoding*) as parameters, where data is requested in sequence from *Overpass* and then *Wikidata*, if not found in the database for the respective query.

  -> If none of the suggestions is a match and the user clicks on the "None of the above" option, then the *geocode-fallback approach* is used.

- The geocode-fallback approach: At the client level, the user is prompted to select the country where the attraction he is looking for is to be found. For the list of countries to be displayed a request is made to the backend ***countries endpoint***, which fetches the countries details from the *country-state-city npm package*. Then the ***geocode-fallback endpoint*** is requested with the previous query and the country code as parameters, and a request to *Mapbox Geocoding* is made.
- Whenever the user clicks on Get Route, a request is made to the ***optimization endpoint*** with a coordinate list (longitude first, latitude second, separated by comma and the coordinate pairs separated by semi-colon) and a way-point name list (with the names separated by comma) set as parameters. Here based on the *API* usage there are a couple of scenarios:
  -> A request is made to either *Mapbox Matrix* or *LocationIq Matrix*, then that matrix is processed with an own search algorithm and finally *Mapbox Directions* is called.
  -> A request is made to *Mapbox Optimization*.

  From a deployment point of view, we have two scenarios:
- Development:
  -> Run the *Node.js* server and open the front-end using *LiveServer*.
  -> If changes are done to only one of the components (front-end / back-end), then we also allow testing with a local component and with the current production version of the other component;
  -> When the back-end component is ran locally the logs are also maintained locally, otherwise they are maintained on the *New Relic* cloud;
- Production, the front-end component is deployed to *GitHub Pages*, the back-end component to *Render*, the logs are sent to the *New Relic* cloud.

In both scenarios we are using the *MongoDB Atlas* cloud database.

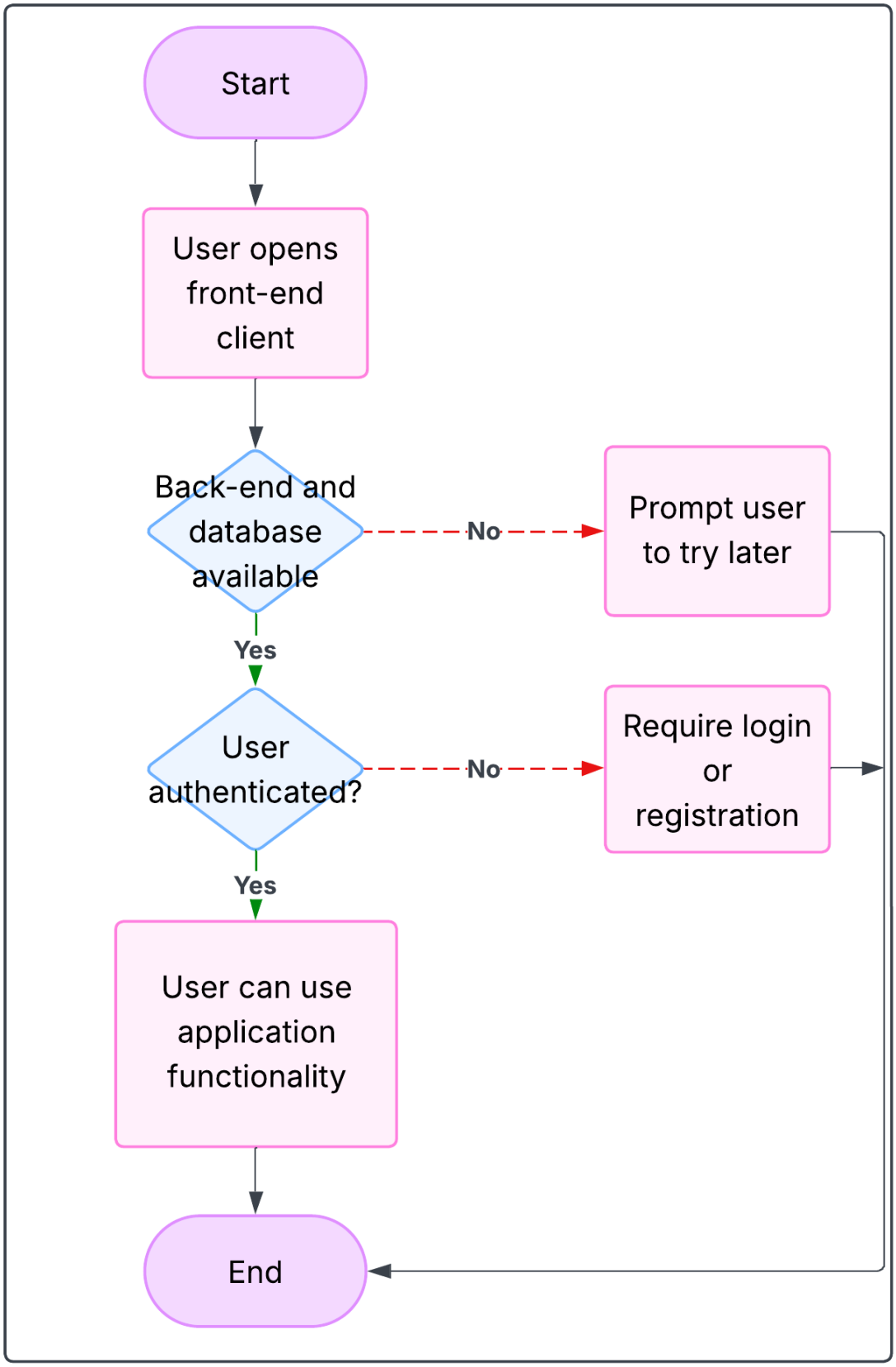

Figure 3.2: User opens up the client.

Chapter 4

Implementation

The code for the entire implementation is available in the Navigation_App GitHub repository.

4.1 Front-end

4.1.1 Structure

The starting point of the front-end component is the index.html file. Here we load the *LeafletJs*[1] script and stylesheet, which aid us in displaying the *OSM*[2] map tiles, and our own *CSS* and main *JavaScript* file.

For the *HTML* part, we have added the minimal components needed to provide a user interface(UI) which fits our purpose of solving the *Traveling Salesman Problem(TSP)*. Since we are working with a Single Page Application(SPA) we had to work with quite a few toggles for hiding and showing different objects.

For the *JavaScript* logic we are using *ES modules*, which allow us to create separate handlers for the different *HTML* components that we create. For example, to handle the elements that we are adding to the map we have implemented a MapManager class, that we are instantiating in the main.js file attaching it to the map object defined in the *HTML*. This approach will be further detailed in the **Subsection 4.1.3 (JavaScript)**.

For the styling part we have used *CSS*. We wanted to be able to have as much control as possible on the application styling. The main point of the styling was to have all the elements float nicely over the map which acts as background.

[1] https://leafletjs.com/index.html
[2] https://www.openstreetmap.org/

4.1.2 HTML

The elements defined in the *HTML* section are as follows:

- Map component;
- Authorization panel, including the login and registration flows;
- User panel, showing the authenticated user details and the logout button;
- Removal of route popup, in case a user wants to remove one of the current way-points when a route has already been computed and is currently on display;
- Geocoding fallback popup, which is used in case the first geocoding request fails, or the user does not find his response between the suggested options and click on none of the above, the aforementioned popup appears and the user can select the country of the way-point that he is looking for to aid in searching;
- Message panel displaying the information regarding the use of the application and next steps that the user should take;
- Route panel used for displaying the summary of the current route, which involves presenting the step by step driving time between each of the way-points in the order obtained via the routing and the total driving time of the complete route;
- The controls panel, which contains the location status message, the search container which includes the search input box and the list of suggestions displayed as a drop-down after a succesful geocoding, and the attractions section, where attractions can be removed, or set as starting point and the user can request to compute the efficient route or to reset the map;

By default we also set all the components other than the map as hidden, and we toggle the visibility based on the different scenarios available in the application. This will be analyzed more in depth in **Chapter 5 (Usage Scenarios)**.

Other than the *LeafletJs* stylesheet and script and our own stylesheet and script we have also loaded *FontAwesome* which we have used for the search button starting from the following resource [45].

4.1.3 JavaScript

The entry point for the *JavaScript* logic is the main.js file, where we are instantiating our own modules and attaching the *HTML* objects to the modules we defined for promoting code separation based on concerns.

The *ES modules* we have created are, as follows:

- AttractionManager, which handles all the flows regarding selected way-points: addition, removal to/from the list displayed inside the search container. It also provides functionalities for fetching the way-points' coordinates and names needed for the routing request;
- MessageManager, which has the purpose of changing the messages in the two main

message areas, the one from the middle of the screen which displays general information about the status of the app, and the one from the search container which displays information about the geolocation status;

- authService, handles the requests for: checking the authentication status, registration, login and logout. Moreover, it defines an initialization function used when the user opens the application (so in the main.js), where the backend origin is fetched, and then a check to see if the user is authenticated happens;
- AuthUI is the layer used to handle the visibility of the different *HTML* authentication components, based on the result of the requests made using authService;
- ServiceUri, builds the requests made to our own back-end for geocoding, routing, fetching the countries, getting extra details about the attractions;
- MapManager, will be discussed with more details below;
- constants, this file contains contains different user friendly messages, and custom events names which were used to provide an easier communication between the *ES modules*, without direct instantiation of the objects. We will be providing an example of using a *custom event below*;
- RouteSummaryHandler, this handles the route summary area which is overlapping the main message area of the app. Whenever the route is computed, the main message area is hidden and the route summary is displayed. This also works the other way. Here we mostly structure the information coming from the backend to be able to display it in a nicely formatted way;

MapManager

MapManager loads the map component using *LeafletJs* library, version 1.9.4 loaded as a script in the index.html and *OSM* tiles.

Other tools used from *Leaflet*, all inside the MapManager, are:

- The locate function, whenever the user connects to the web application, he is prompted to allow access to his location;
- The location related events (success, failure), which are propagated through to the main JavaScript module;
- The Markers, all the interactions that a user has with the way-points, changes the state of the two marker arrays that are handled in the MapManager;

In case of successful identification of the user location, from the main *JavaScript* file, the location is added to the way-point array from the search container, and a success message is displayed using the Message Area Handler, while in case of error, the user is informed that his current location has not been fetched so he must input a different starting point.

As previously mentioned, in the MapManager, which is an own *ES module*, we are handling two marker arrays. The markers from one array are displaying permanent labels,

and the markers from the other array set an icon on the map, which has a popup attached containing extra details about the attraction, if those are available.

The scenarios of markers are added to the map are the following:

- A user allows access to his location;
- A user searches an attraction using the search container;
- A user clicks on the map and a dropped pin is added to the map;

The only scenario where the popup contains extra details about an attraction is when the attraction is selected using the Search Container. This is because in the other cases scenarios (when adding way-points based on location or dropped pins) we do not have access to *Overpass* or *Wikidata* ids, so we do not have access to the needed inputs for the requesting the extra details.

Custom Events

Regarding the use of Custom Events, a nice example is when clicking to remove an attraction, this triggers the handler from the AttractionManager. Then we forward the event to the MapManager, where we check whether there is a route displayed, if so the user is prompted whether he wants to remove the route. If the user decides that he wants to remove both the route and that specific way-point, the associated markers from the MapManager get deleted, alongside the route and then an event is sent back to the AttractionManager, and the way-point gets deleted from the list. If there is no route displayed, then no other input from the user is needed, but the event interaction between the two modules stays the same as we have to delete both the two markers from the map and the way-point from the attractions list.

Continuing with the analysis of the main.js file, after instantiation of the aforementioned modules, we are calling the initialization logic from the authService, which checks which backend instance is available and provides the user with the appropriate flow, based on his authentication status. Also in the main.js file we have defined most of the event listeners. For some of the events we call as handlers methods from the other modules, an example being Source Code 1, while for other events we have implemented the handlers directly in the main file, as it can be seen in Source Code 2.

```
document.addEventListener(EVENTS.ADD_ATTRACTION_ON_CLICK, (e) => {
  manageSelectedAttractions.addAttractionToContainer(e.detail);
});
```

Source Code 1: JavaScript event handled using method from module in main.js front-end file.

One of the other things to be mentioned for the main.js file is the way the countries provided as option in the form of the popup of the fallback geocoding are being handled.

```
document.getElementById("search-attraction-form").addEventListener("submit",
 ↪  (e) => {
  e.preventDefault();
  performSearch();
})
async function performSearch() {
  const currentQuery = searchInput.value.trim();
  if (!currentQuery) return
   ↪  manageTextAreas.showNoQueryFoundErrorMessage("attraction");
  clearSuggestions(false);
  lastQuery = currentQuery;
  try {
    lastSearchResponse = await
     ↪  geocodingRequestManager.fetchSuggestions(lastQuery);
    searchedQueries.push(currentQuery);
    const items = lastSearchResponse.options;
    if (!items.length) return manageTextAreas.showNoSuggestionMatch(lastQuery);
    processGeocodingSearchResults(items);
  } catch (err) {
    console.error(err);
    await showLocationSelectionDialog();
  }
}
```

Source Code 2: JavaScript event handled using method from main.js front-end file.

Whenever the geocode-fallback flow is called and the popup where the user can select the country of the attraction that he is looking for is opening up, there is a check to see whether there are countries already available in the session. If the countries are not available, a check is done to see whether the countries are stored in the session storage, which should still be available after a refresh, and if neither is available, then the countries are requested from the back-end.

4.1.4 Styling

For the styling part we started by doing a *CSS* reset, following the tips from [46]. The most important parts from the styling are as follows:

- Having the map cover the entire area;
- Having the elements float over the map;

To obtain these we have two very important styling settings, that can be seen in Source Code 3. For having the map as background we used z-index with a minimal value, and in opposition we have set a high z-index for the floating elements. For having the map cover the entire view-port we set the width and height to 100%. Finally to make sure there are no empty margins, we set an absolute position of the map starting from the top-left corner. For the floating-panel class we also set the position as absolute, but the specific

position is defined by each of the elements that use this class individually.

```
#map {
  width: 100%;
  height: 100%;
  position: absolute;
  top: 0;
  left: 0;
  z-index: 1;
}
.floating-panel {
  position: absolute;
  z-index: 1000;
  background: rgba(255, 255, 255, 0.95);
  backdrop-filter: blur(10px);
  border-radius: 12px;
  box-shadow: 0 8px 32px rgba(0, 0, 0, 0.15);
  padding: 1rem;
  max-width: 90vw;
}
```

Source Code 3: CSS setting up the full screen map background.

4.2 Back-end

4.2.1 Structure

The starting point of our back-end structure is the server.js file. Here we are setting up the *Express* server, connecting to the *MongoDB Atlas* database instace, setting the *CORS middleware*, making the server *parse cookies*, creating the health endpoint (which informs the front-end whether the back-end instance and the database connection are available, and is only protected by the *CORS middleware*) and setting up the authentication, routing and search routes.

Then we have the routes folder, where we have two separate files containing the authentication routes and the routing and search routes. In here we define the endpoints that will be used from the front-end component to request data and add the custom middlewares we have implemented. The authentication routes are: register, login, logout, and me (used for checking whether the user is logged in already). All these routes are protected by the *database middleware* as the connection to the database is needed for the authentication process. The routing and search endpoints are: geocode, geocode-fallback,

countries, v2/optimize, and extra-details. All of them use both the *database middleware* and the *authentication middleware*.

Each of the routes is connected to an associated controller from the controllers folder.

Each of the controllers is using the logger.js and the helper functions from logObj.js for logging purposes as it can be viewed more in depth in the **Appendix A (Logging)**. Some of the controllers are using one or more services. The services represent the layer which interacts with the *MongoDB Atlas* instance.

We are using the *dotenv npm package* for loading the *API* keys. We created a config object that we import in the controllers that use keys for external *API* requests.

4.2.2 Middleware

For this application we are using three middlewares:

- The one provided by the ***cors npm package***, used for allowing only requests that come from the deployed front-end origin or from a local instance of the front-end and for setting a property to not omit the credentials of requests.
- The one used for **checking if the database is up** by verifying a variable attached to the express app locals property once the database connection request is finished.

- The one used to **see if the user is authenticated**. For this we first extract the *cookie* from the header of the request coming from the front-end. If a user is authenticated on a browser, then a the session cookie should come with all the requests. If no session id cookie is found in the request, it means that the user is not authenticated. If there is a session id attached, then we check whether the id of the session is in the expected format which should match the default id generator from *MongoDB Atlas*. Next, we try to find the login session, based on its id extracted from the request, in the sessions collection from the authentication database. If a login session is found based on the id, then we check to see if the session is expired. If all checks are successful then we can say that the user is authenticated, in the other cases if there is a session id found, but it fails one of the checks, then we send to the front-end the status unauthorized and we remove the session cookie from the browser and the login session from the database.

4.2.3 Authentication

For the authentication process we have defined the following services:

- **userService**, where we have defined two methods for the authentication database, users collection, one for finding the user based on the email address (using the findOne method of *MongoDB's* Collection) called getUserByEmail and the other

for storing a user during the registration process (using the insertOne method of *MongoDB's* Collection) called registerUser;

- **sessionService**, where we have defined the create (called createLoginSession), read using the session id as key (called getLoginSession) and delete using the session id as key (called deleteLoginSession using the deleteOne method of *MongoDB's* Collection) methods for the authentication database, sessions collection;
- **sessionCookieHandler**, is the place where cookie handling is centralized, here we have defined the options for the cookie (name, expiration date defined as twenty-four hours, sameSite as none because the application is cross-site, and finally, for security reasons we have set httpOnly and secure to true), and we have created some external methods for:
  -> setting and clearing the cookie from the browser;
  -> fetching the session id from the request;
  -> checking if the session cookie is expired;
  -> creating a *MongoDB* clearing index for the authentication database, sessions collection, based on the creation time field, which automatically removes the entries in the collection after twenty-four hours;

For the authentication process we have defined four endpoints as follows:

- The **me** endpoint, where we just wrap the successful result (if that is the case) of the *authentication middleware* to a JSON response;
- The **register** endpoint, where we first check if user and email have been added to the request body, then we call getUserByEmail from userService, to see if the email has been used before for registering, and then finally if the email has not been used, we hash the password using the *bcrypt npm package* and store the user details with the hashed password in the authentication database, users collections from *MongoDB Atlas* using registerUser method from userService;
- The **login** endpoint, where we first check if user and email have been added to the request body, then we call getUserByEmail from userService, to see if a user with the respective email address is registered, next we check whether the password inputted by the user is the same as the hashed password from the database by using *bcrypt's* compare method. If all checks have passed, we create a clearing index, using the method from sessionCookieHandler, for the sessions collection from the authentication database, only if the expiration time of the session cookies has been changed. Then we store the login session to the database using createLoginSession method of the userService and finally we set the cookie to the browser using the method import from sessionCookieHandler;
- The **logout** endpoint, where we first fetch the session id using the method imported from sessionCookieHandler, then we call the deleteLoginSession method from userService and we always clear the cookie from the browser, even in case of errors;

4.2.4 Search and Route

For the search and route process we have defined the following services:

- **attractionService**, where we have defined the create (called insertAttraction) and read using the attraction search name as key (called getAttractionOptions) for the geocoding results database, attractions collection;
- **extraDetailsService**, where we have defined the create (called insertExtraAttractionDetails) and read using a search object containing either the *OSM* id and type or the *Wikidata* id (called getExtraAttractionDetails) for the geocoding results database, attractions extra details collection, where we have defined the create and read;
- **userApiRequestService**, where we have defined the following methods for the authentication database, user api requests collection:
  -> incrementRequestCount, where we maintain the user details, the API provider, the used endpoint, the timestamp and a count (by default is 1, but there is the *Matrix API from Mapbox* which needs a different count, which is computed as source way-point times destination way-points);
  -> getDailyApiRequestCount, where we are able to count how many requests have been done today using a specific API provider (*LocationIq* shares the count of requests accross the different endpoints), we are able to do per endpoint filtering, we can also do per user filtering and finally we can count the number of requests either by using the count field (in the *Mapbox Matrix API* scenario) or directly count the number of documents from the collection from today using the countDocuments method from Collection with the filtering options containing today's date;

For the endpoints that make requests to the external *APIs* we have defined both per user limits and total limits as follows:

- For *LocationIq* we have set the daily limits as 5000 in total, and 500 for each user.
- For *Mapbox* we have set the daily limits as 3300 in total, and 330 for each user.

For the search and route processes we have defined the following endpoints:

- The **geocode** endpoint, where we first check whether an attraction has been set as parameter and then we check whether the LocationIq API key is available in the config file connected to the environment variables. Then we call the getAttractionOptions from attractionService with the current attraction name to see if the geocoding suggestions for this attraction have been cached, if so the cached result is sent to the front-end. Otherwise, we check whether the API limit for *LocationIq Geocoding* has been reached by calling getDailyApiRequestCount twice for the LocationIq API provider: once filtering with the user and once without. This way we obtain the count of requests of the user and the total count of requests from the

current day. If neither of the limits has been reached we call: *LocationIq Search / Forward Geocoding* and incrementRequestCount from userApiRequestService to track the request. The final steps are: to create the object to be cached (which will contain the attraction search name, the options obtained from the geocoding requests and the creation time), to call insertAttraction method from the attractionService and to send to the frontend the object created for caching;

- The **geocode-fallback** endpoint, where we first check whether an attraction and a country code have been set as parameters and then we check whether the Mapbox API key is available in the config file connected to the environment variables. Then we check whether the *Mapbox Geocoding API* limits have been reached (same approach as for the *LocationIq Geocoding*, but here we also provide the endpoint name as filtering option). If not we make the external request where we change the maximum limit of geocoding suggestions from the default 5 to 10. Finally we process the response, to reconstruct the geocoding suggestions to have similar fields as the ones obtained from *LocationIq Geocoding*;
- The **countries** endpoint, where we use the *country-state-city npm package* when calling the getAllCountries method from the Country object, then we extract for each country the name and the ISO code and we wrap this in a JSON object and send it to the front-end;
- The **extra-details** endpoint, where we first check whether the OSM id and type or the Wikidata id have been provided as parameters, then based on the provided parameters there are two separate handlers:

  -> When **OSM id and type** are provided, we first check the cache by calling getExtraAttractionDetails method from extraDetailsService with the OSM details bundled in a JSON object, then if there is no result cached we request data from OSM via the *Overpass API*. Next if there is a match for the OSM details, we try extracting from the tags the Wikidata id (and if that is available we will try fetching some Wikidata results as it will be described for the next scenario, from which we are interested in the image), the official website, the opening hours, the contact details (phone number and email address). Finally we build an object which contains the OSM id and type as key and the other fetched information, then we send it to the front-end and cache it by passing it as a parameter to insertExtraAttractionDetails method from extraDetailsService.

  -> When **Wikidata id** is provided, we first check the cache by calling getExtraAttractionDetails method from extraDetailsService with the Wikidata id wrapped in a JSON object, then if there is no result we make a *Wikidata SPARQL* request looking for the image and the official website associated to the Wikidata id, and finally we build an object that is both to be sent to the front-end and cached with the information found and the Wikidata id

as key. The caching is done by calling insertExtraAttractionDetails from extraDetailsService with the aforementioned object as parameter.

- The **v2/optimize** endpoint, where we first check whether the waypoint name list and the coordinate list have been set as parameters. Then we check the API limits as follows: we bundle the request counts for the Mapbox Directions and Matrix APIs; if the *Mapbox Directions* limits are surpassed then the user will get an error, otherwise if the user did not surpass the limits and did not surpass the *Mapbox Matrix* limit then the processing would continue with these two APIs. In the case where the Directions limits were not surpassed, but the *Mapbox Matrix* reached its limit, we would check the LocationIq request count, and if that did not surpass the limit we would use the *LocationIq Matrix* alongside the *Mapbox Directions*. The last possible scenario is when neither of the Mapbox and LocationIq Matrix APIs are available, and in that case we have a different implementation which calls the *Mapbox Optimization*. After this API selection is done, we check whether the Mapbox or LocationIq API key is available in the config file connected to the environment variables and we call the associated API alongside the incrementRequestCount method from the userApiRequestService, with the sole difference that the count for the *Mapbox Matrix* is the count of coordinates squared. The response in each of the two cases returns both the directions and durations matrices. The next step is to call our own searching algorithm to find the optimal route. The algorithm, using as input the durations matrix and the waypoint names, is detailed in **Subsection 4.2.5 (Search Solution Optimization)**. The next step is to call the *Mapbox Directions API* with the coordinates in the order obtained from our search algorithm, and finally we process the response and the route result obtained from our search algorithm to build the output that needs to be sent to the front-end. The content that are needed at the front-end level are: the *GeoJSON* to be displayed on the map, the array of durations which is formatted with a UI friendly way from seconds, an array with indices from the initial list in the order of visitation, and the way-point names in the order of the visitation. In the scenario where we call the *Mapbox Optimization API*, the approach is similar to the other implementations, the significant part is the processing of the API's response, where we build an object similar to the one from the other scenario to be able to use the same front-end process;

4.2.5 Search Solution Optimization

For obtaining an efficient algorithm for our use cases we have created the **test-optimization** endpoint, that is not connected to the front-end component. This was used for running optimization tests by repeatedly calling the different search algorithms with varied number of way-points (from 2 to 25). To create variety we shuffled on each

test the 25 way-points that we extracted as testing base. We have compared the speeds, but also the quality by plotting the resulting tours using *Mapbox Static Images and Directions APIs*. For more details on how and what was tested, please refer to **Appendix B (Search Solvers Comparison)**.

Brute Force

Initially we have implemented a brute force search algorithm that has the purpose to compute all the possible routes, and then at the end to compare the routes by their total cost (in this case duration or distance) and to return the minimal cost route. The function we have implemented takes as parameters the way-point names and either the distances or durations matrix (by default durations). The output is the efficient route which starts and ends with the first way-point from the list. First we create an array of possible steps based on the durations matrix, removing the source destination pairs from the main diagonal which have the distance as 0 since they are the same way-point. Then we call the recursiveIterate method from Source Code 4. The recursion ends whenever we have went through as many recursive rounds as there are way-points in the input list. In the first round we set up the initial routes, by adding all possible routes which start from the first way-point. Also, for each of these initial routes, we compute the next possible steps, by filtering the initially defined possible steps, to have the source way-point match the current way-point in the route. Then, on each subsequent recursive steps, we extend the routes list, by adding a new route for each of the route's current possible steps. When adding a new step to the route (while creating the new route to be added to the route list) we remove the attraction that is being visited from the route's attractionsToSee array and we compute the new possible steps of the route. While computing the new possible steps of the route, there are three scenarios: when all the way-points have been visited we return an empty list, if it is the final step then return the step which starts from the current location and has the destination the starting way-point, in the other cases we look for the possible steps which have the source being the current location and the destination has to be part of the attractionsToSee array, but it must not be the starting way-point as that is to be visited last.

The brute force approach works fine for a small number of attractions, but as this approach has a complexity of O(n!), when the number of attractions starts growing the limits of this algorithm become noticeable. On the local server we have seen that the implementation can handle at most 9 way-points, from 10 onwards the algorithm fails due to low memory, from the too large count of recursive steps.

```
function recursiveIterate(routes, round) {
    if (round === route_max_steps) return routes;
    let newRoutes = [];
    for (const route of routes) {
        for (const possibleStep of route.nextPossibleSteps) {
            let newRoute = JSON.parse(JSON.stringify(route));
            newRoute.stepCount += 1;
            addStepToRoute(newRoute, possibleStep);
            newRoutes.push(newRoute);
        }
    }
    return recursiveIterate(newRoutes, round + 1);
}
return getFastestRoute(recursiveIterate(getInitialRoutes(), 1));
```

Source Code 4: Recursive search algorithm.

Nearest Neighbour

Then we have implemented the nearest neighbour heuristic[3] which is the fastest solver for this problem. The idea behind this solver is the following:

- We start visiting nodes with the first way-point (start and end point);
- Then for the next step we check the costs of all the available variants, and we choose the cheapest one (smallest duration / distance between attractions);
- We do this iteratively until we have visited all the way-points in the list, but since the start way-point appears only once at the beginning, to include also the step where we are returning to it, we add an extra step, with the source being the last point from the way-point list and the destination being the first way-point;

The complexity of this heuristic is $O(n^2)$, since we are iterating once to account for the number of remaining attractions to visit, and there is a nested loop for finding the minimal cost node starting from the point we are currently at.

2-opt

The 2-opt heuristic is not as fast as the Nearest Neighbour approach, but it provides solutions that are way more reliable. The approach with this solution is based on viewing the cyclic tour as an ordered graph and on having a tour before even starting running the heuristic. At each step, we are trying to minimize the total cost of the tour by doing a 2-change, which means:

- Choosing 2 edges, that do not share any nodes (so there are 4 different nodes involved), and removing them;
- Now we are going to have two disconnected graphs;

[3] A heuristic provides a good enough solution, while an algorithm returns the optimal solution.

- Now we need to connect the 4 nodes differently; Theoretically there are two options, but in practice there is only one that recreates a cyclical tour, while the other approach creates two non-cyclical graphs so not a viable tour for our problem;
- We compute the cost modification that comes with switching the edges, which is computed by summing the new two edges (the two durations of the new roads between the attractions);
- If the result is an improvement, then we update the tour;

In our implementation, we have provided two approaches for setting the initial tour, that we represent as an array:

- By default, we start with the way-points in the order they are inputted by the user, and we create an index array from 0 to n-1 where n is the number of way-points (we refer to this approach as 2-opt);
- Otherwise, an initial tour can be provided as parameter. The purpose of having this option is to allow the use of other faster heuristics to provide an initial tour, that is closer to the optimal one, and in this way to converge to the solution faster. We have used as starting tour, the result for the Nearest Neighbour (we refer to this approach as 2-optNN).

The complexity for this heuristic is structured as follows:

- In general the 2-move has a complexity of O(n^2), but in our specific implementation, in case a a move has been found, we break out of the nested loops of the 2-move to re-start from the begining of the indices array.
- Then, we have to consider that we iterate for at most 1000 times to find a 2-move, as we have set a max iteration limit. So the algorithm with the current implementation, in the worst case scenario O(n^2 * max_iterations).

In the **Appendix B (Search Solvers Comparison)** we have detailed how we have built the testing structure to compare the search algorithms and we have presented the comparative results, both with statistics and figures. As earlier mentioned, we have observed that the brute force method can handle at most 9 way-points, that the nearest neighbour heuristic is great for its speed, but not standalone, and that between the 2-opt with default start and the 2-optNN (2-opt seeded with the nearest neighbour tour) it seems that the 2-opt is slightly faster, but the other one converges to better solutions, and as both of them are fast enough for our use case we are going to choose the one providing a slightly better tour.

The implementation that is currently connected to the front-end provides a mix of the approaches, by using the brute force algorithm if the number of way-points is lower or equal to five, and the 2-opt heuristic seeded by the nearest neighbour tour for larger way-point counts.

4.2.6 Testing Endpoints

For testing the back-end endpoints we have installed the *REST Client VSCode* extension.

When building the controller of a new endpoint, it was helpful to define also a *HTTP* file where to simulate a request to that specific endpoint. This was helpful for GET requests, but it was a very important tool for the POST requests. Analyzing Source Code 5 we can see in the first section that some testing variables are being set. To be able to properly do the testing for the *geocode endpoint*, we first need to be logged in, since there is a middleware attached to all the backend requests that checks whether the user is authenticated. The different requests are separated by the ### delimiter. This is why we have also added a login request. After sending the login request, the cookie is set to a variable, that can be reused for multiple requests. We have also created both a scenario where the request fails due to the place query parameter not being set. In the end, after the testing is done, we can use the logout request so that the login session gets deleted from the database.

In a similar way we have implemented *HTTP* files for testing the authentication endpoints (registration, login, logout) and for the other search and route endpoints (optimization, fetching countries and fetching extra attraction details).

```
@baseUrl = http://localhost:3000
@query = Arcul+de+Triumf
# @name login
POST {{baseUrl}}/authentication/login HTTP/1.1 content-type: application/json
{
    "email": "user5@gmail.com",
    "password": "somepass"
}
###
@fullCookie = {{login.response.headers.set-cookie}}
### 2. Successful Get suggestions request
GET {{baseUrl}}/api/geocode?place={{query}} HTTP/1.1 Cookie: {{fullCookie}}
### 3. Failed request due to missing parameter
GET {{baseUrl}}/api/geocode? HTTP/1.1 Cookie: {{fullCookie}}
### 4. Use session for logout
POST {{baseUrl}}/authentication/logout HTTP/1.1 Cookie: {{fullCookie}}
```

Source Code 5: HTTP file used for testing the geocode endpoint.

4.3 Database

For the database we have used *MongoDB Atlas* for two reasons: the flexibility of *MongoDB* document based model and the fact that by using this service we got a production ready cloud database. For this *MongoDB* project we have whitelisted all the IPs. In *MongoDB* terms we created two databases, and for each we created collections, as follows:

- The authentication database with the following collections:
  -> user_api_requests with the fields: user_id, email, apiProvider, endpoint, version, timestamp ,and date;
  -> users with the fields: email_address, password (the hashed password), created_at, and date;
  -> sessions with the fields: user_id, email_address, login_time; For this collection we created a twenty four hours clearing index;
- The geocoding_results database with the following collections:
  -> attractions with the following fields: attraction_search_name, options (the array of suggestions provided by *LocationIq Geocoding*, a sample is provided in Figure 4.1), created_at and date;
  -> attractions_extra_details which contains two kind of entries (based on how the extra details are being requested); when the request is made with osmId and osmType other maintained fields are: osmName, osmEnglishName, osmWebsite, openingHours, phone, email, osmImage, wikidataId, created_at, and date; when the request is made with wikidataId, the document contains also: osmImage, osmWebsite, name, description, created_at and date; For the scenario where wikidataId is used all the details come from *Wikidata*, but we kept the names that were used at the front-end side constant (osmImage); Moreover the image in both scenarios comes from *Wikidata*;

Each collection has a field automatically added named _id and its value is automatically generated. We have also used its value for the session id when creating the log-in cookie and for referencing the users.

Figure 4.1: User opens up the client.

4.4 Deployment

4.4.1 Front-end

For the deployment of the front-end component we have used *GitHub Pages*, following the documentation [47]. We created a *GitHub* repository, where we added the front-end related files from the main GitHub repository, as it can be seen in the Figure 4.2.

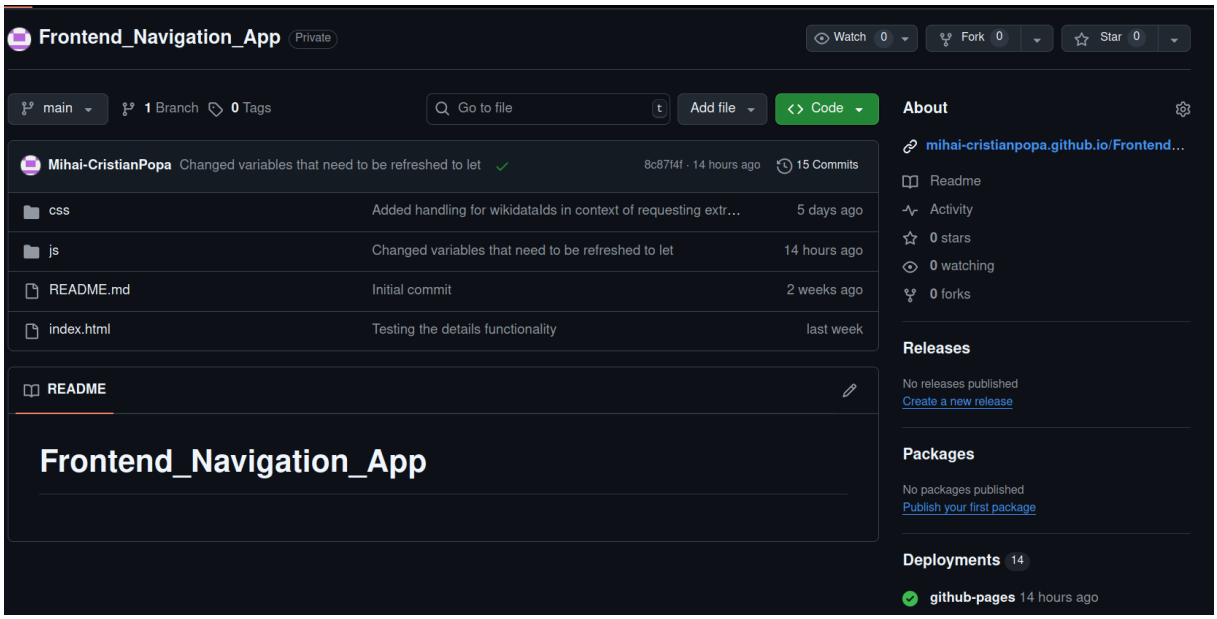

Figure 4.2: Front-end GitHub Repository

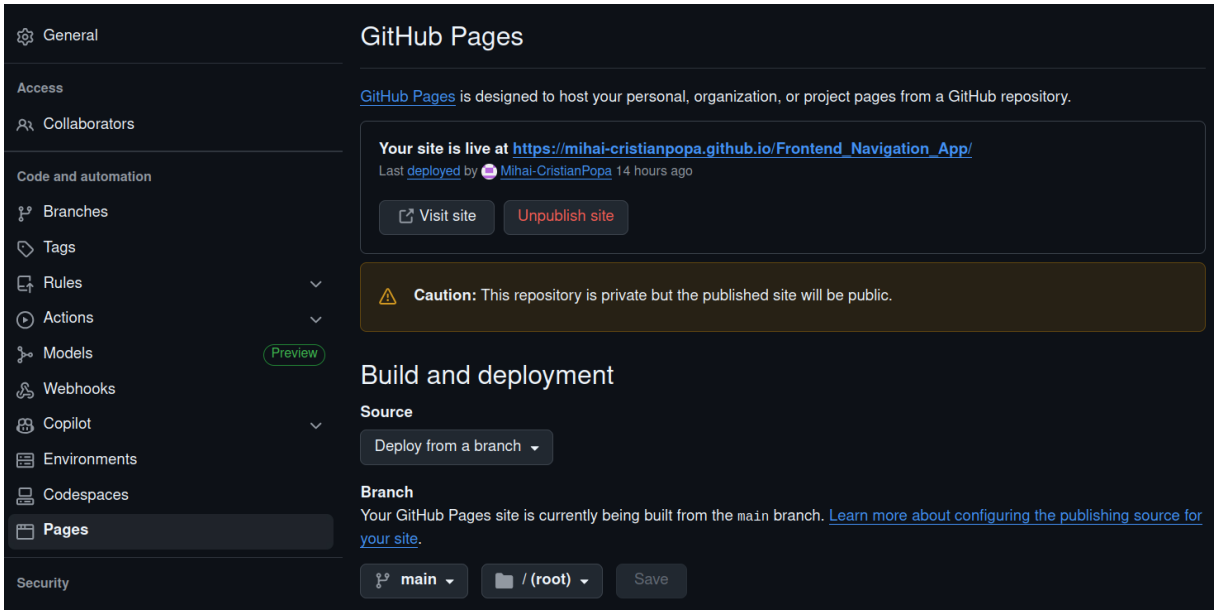

Figure 4.3: GitHub Pages setup

Then, for that repository we have created a new *GitHub Pages* page, tracking the main branch, as it can be seen in Figure 4.3. The application is available at the following

link: https://mihai-cristianpopa.github.io/Frontend_Navigation_App/. The result of the front-end deployment can be seen in Figure 4.4.

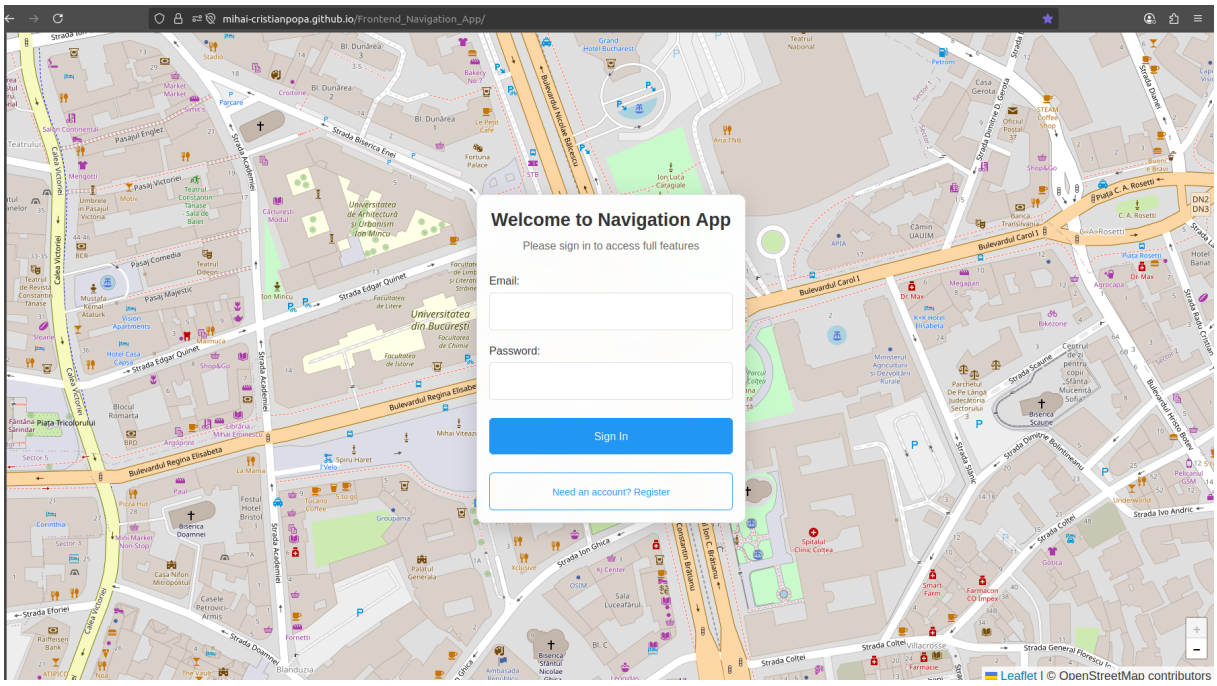

Figure 4.4: Deployed Application

4.4.2 Back-end

For the deployment of the back-end component we used *Render*. We have created a new *GitHub* repository, where we have added all the files related to the back-end from the main *GitHub* repository, the same way we did for the front-end component. Then we have connected to *Render*, with the *GitHub* account, selected the Backend_Navigation_App repository, main branch, as source for the deployment. Alongside choosing the repository, we had to make some other settings regarding the commands used for building and running the project. After this, the *Render* Project has been created. These steps can be visualised in Figure 4.5. The back-end deployment can be checked at the following link: https://backend-navigation-app.onrender.com/health .

Most of the connections with external resources need some *API* keys/tokens, so to handle that we added the environment variables needed in *Render*, as depicted in Figure 4.6.

There is also a logging capability provided by *Render*, as it can be seen in Figure 4.7, but the logs are maintained for only seven days.

For this reason we decided to connect *Render* to an external logging tool. We have found the most appropriate for our use case to use *New Relic*.

To achieve the connection between *Render* and *New Relic*, we have:

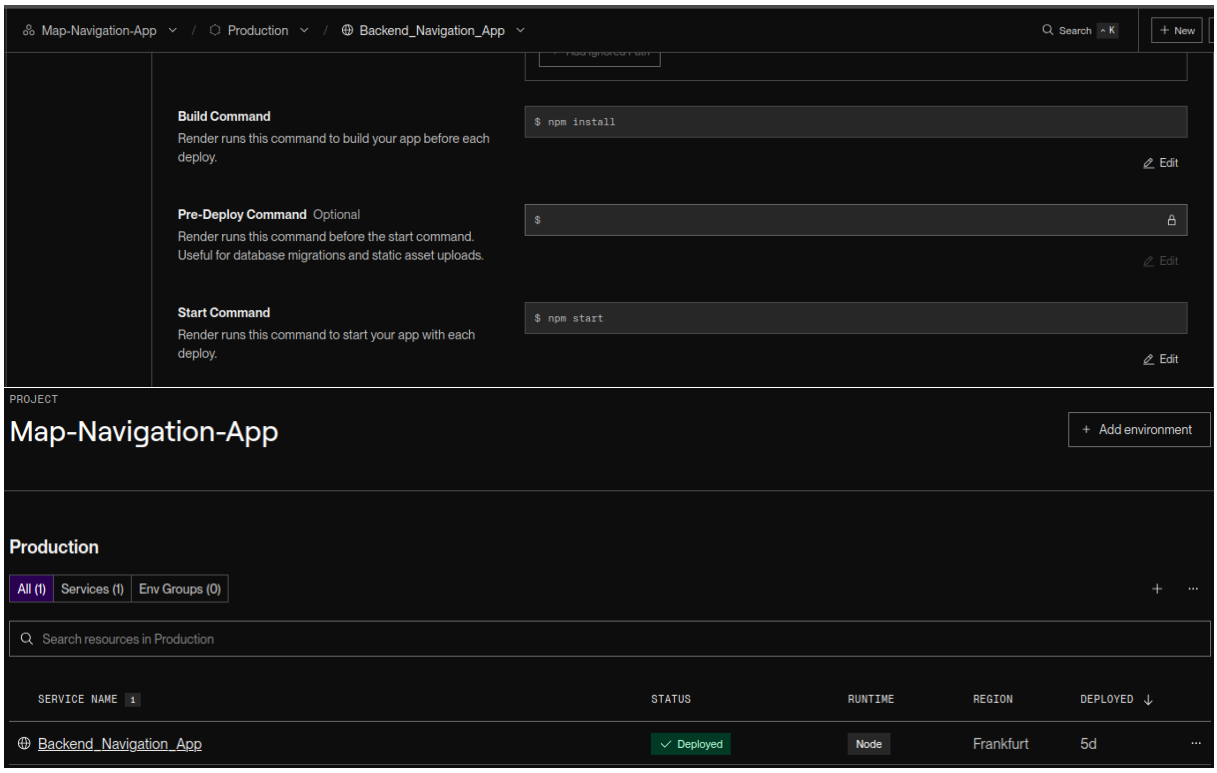

Figure 4.5: Setting up the initial Render deployment: (Top) Settings for Building and Running the Back-end, (Bottom) The Project View

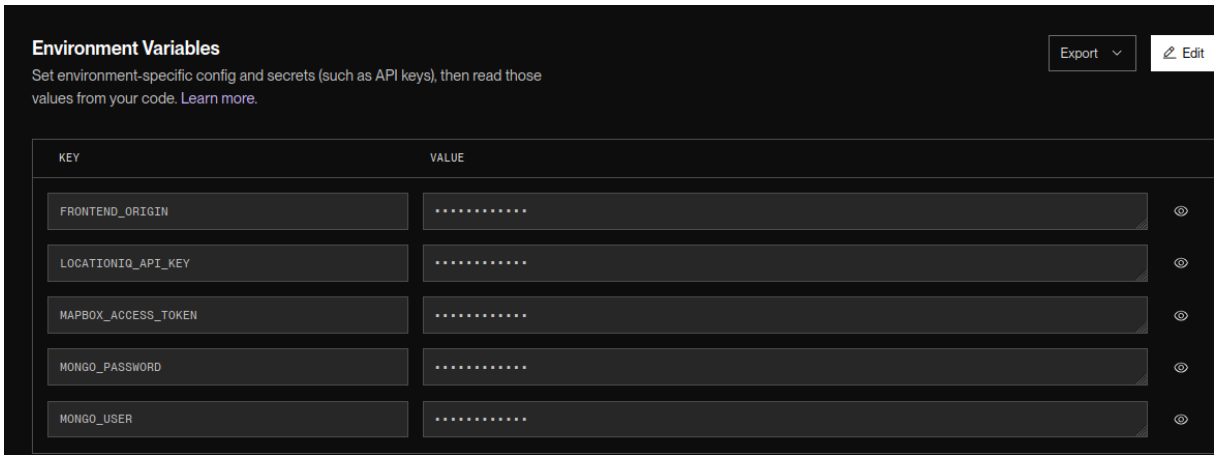

Figure 4.6: Render Environment Variables

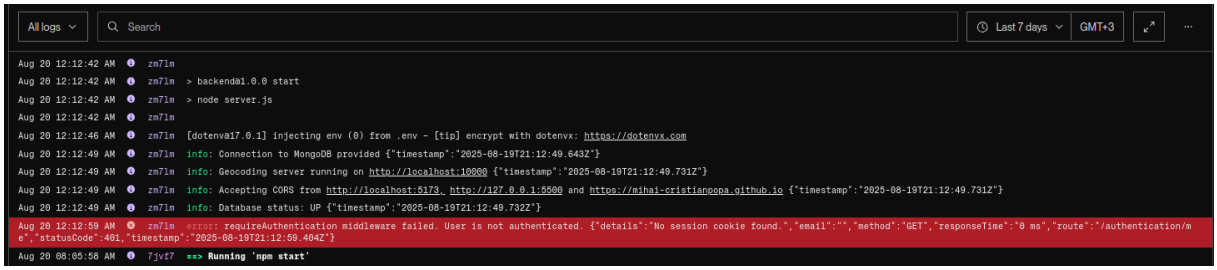

Figure 4.7: Render Logging Capability

- Created a new *API* key inside of *New Relic* specifically for the *Render* connection, following the documentation [48];
- Connected inside of *Render's* observability section[4] the default log stream destination to be the European *New Relic* TCP[5] endpoint[6], as it can be seen in the documentation [49].

The connection steps are depicted in Figure 4.8.

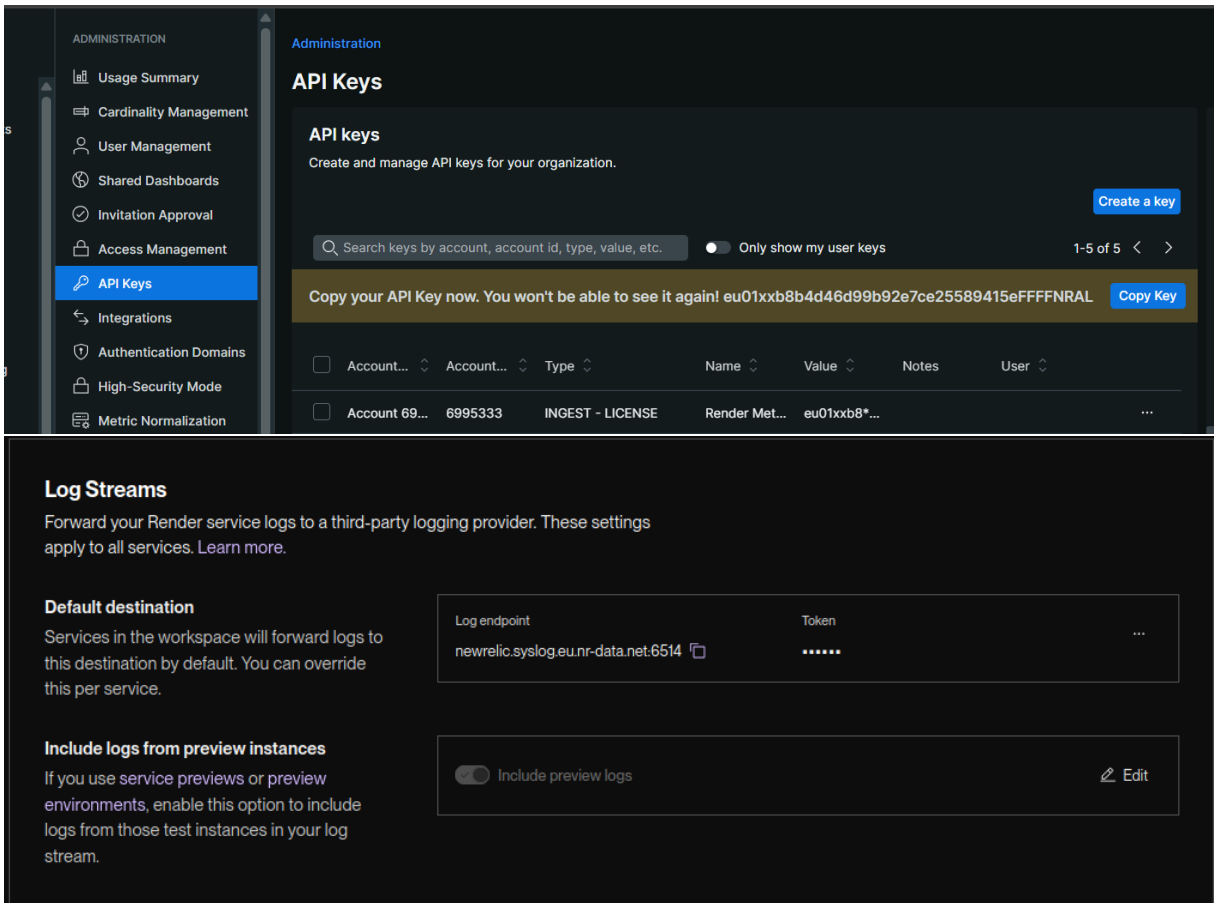

Figure 4.8: Connecting Render Logs with New Relic: (Top) Created the New Relic API Key for Render, (Bottom) New Relic set as default logger for Render

The logging dashboard from *New Relic* can be seen in Figure 4.9.

Render limits: The deployment can be used for 750 hours per month. That means if there are 10 users, each of them can use the application for 2 hours and a half per day.

New Relic limits: Logs are kept for 30 days, and there are 100gb that can be used for storage.

[4]https://dashboard.render.com/observability
[5]Transmission Control Protocol is an important protocol of the Internet.
[6]newrelic.syslog.eu.nr-data.net with port 6514

Figure 4.9: New Relic Logging Capability

4.4.3 Adjustments

Adjustments made so that both the deployed components work correctly together: First, we had to make sure to add this new origin of requests[7] to the allowed list that we are setting in the back-end using the *cors npm package*.

```
export default async function resolveBackendOrigin() {
    const localOrigin = "http://localhost:3000";
    const remoteOrigin = "https://backend-navigation-app.onrender.com";
    const origins = [localOrigin, remoteOrigin];
    const healthEndpoint = "health";
    for (const origin of origins) {
        try {
            const res = await fetch(`${origin}/${healthEndpoint}`);
            const data = await res.json();
            if (res.ok && data.database === "UP") {
                return origin;
            }
            console.warn(`Health check at ${origin} returned ${res.status}`);
        } catch(error) {
            console.warn(`Network error checking ${origin}:`, error);
        }
    }
    return "";
}
```

Source Code 6: Resolving the backend origin for API requests.

Second, we had to make sure to add to the front-end component a request to the health endpoint of the back-end deployment. We need to know what back-end origin

[7]https://mihai-cristianpopa.github.io

to use for sending the requests. So, whenever the front-end is opened, either locally or in deployment, it first checks the health endpoint from the local back-end. If the local backend is not available then it checks the deployed back-end. Based on which back-end works the origin of the requests is set for the front-end session, as it can be seen in the Source Code 6.

Chapter 5

Usage Scenarios

The actor in our application will always be an authenticated user, with the only exception being the use cases from **Section 5.2 (Authentication)**.

In the following sections we are going to refer to use cases as UC and to the user interface as UI.

5.1 On page opening

UC-01a — App readiness & authentication **Goal:** Verify availability and gate access behind login. **Pre:** Application loaded in the browser. **Trigger:** Page is opened.
Main:

1. Check back-end health;
2. If authenticated, show the map UI;

Alt/Errors: A1: If unauthenticated, show authentication UI and wait for successful login; E1: back-end/DB unavailable → outage banner, actions disabled. **Post:** User is authenticated or blocked by outage.

UC-01b — Location setup **Goal:** Choose initial centering and (optionally) create the start way-point. **Pre:** UC-01a (App readiness & authentication) completed; map UI visible. **Trigger:** On first render.
Main:

1. Browser requests geolocation permission;
2. If granted and no start exists, center on current position and add *Current Location* as start;

Alt/Errors: A1: If denied/unavailable, center on default and show the first attraction added will be the start/end point of the tour warning message. **Post:** Map centered; start way-point defined or deferred.

5.2 Authentication

UC-02 — Registration **Goal:** Create a user account with email and password. **Pre:** App loaded; back-end reachable; user is unauthenticated. **Trigger:** User clicks *Register*.
Main:

1. Show registration form (email, password, confirm password);
2. User fills the form; client validates format/strength/match;
3. Submit to back-end; back-end creates account;
4. On success, navigate to Login screen with email pre-filled;

Alt/Errors: E1: failure during this process → show inline errors and allow retry. **Post:** Account exists; user is ready to log in.

UC-03 — Login **Goal:** Obtain an authenticated session and reveal the main UI. **Pre:** User has an account; back-end reachable. **Trigger:** User submits *Login* (email + password).
Main:

1. Back-end validates credentials; on success sets a secure *HTTP session cookie*;
2. Hide authentication UI; show floating panels: search/start controls, message area, user email + logout;
3. Center the map according to UC-01b (Location setup): Current Location if permitted; otherwise default (University of Bucharest);

Alt/Errors: E1: failure during this process → show error and allow retry; A1: if geolocation resolves, after login add *Current Location* as first way-point. **Post:** Session active; main application UI is visible and ready for search and routing.

5.3 Search and Route

UC-04 — Search & Select Attraction **Goal:** Add a way-point for routing. **Pre:** Authenticated; back-end reachable. **Trigger:** User types and clicks *Search*, then selects a suggestion.
Main:

1. Back-end returns suggestion list below the search box;
2. User selects the intended attraction;
3. Map re-centers; marker and way-point are added; extra details for the popup content are requested;

Alt/Errors: A1: clicking outside collapses the list; A2: clicking inside the search bar re-opens the drop-down; E1: no results → UC-05 (Fallback "None of the above" or no suggestions found) is called; E2: No query typed → message area will display the error. **Post:** Way-point added and ready for routing.

UC-05 — Fallback "None of the above" or no suggestions found **Goal:** Refine search with country code. **Pre:** Authenticated; initial search was not satisfactory. **Trigger:** User clicks *None of the above*, or first geocoding request fails providing suggestions.

Main:

1. Dialog opens; user selects country;
2. System queries a different geocoding API with the country code;
3. New suggestions appear; on selection, marker and way-point are added

Alt/Errors: A1: user cancels dialog → no changes; E1: still no useful results → guidance message (try alternative map/tool/name). **Post:** Way-point added via refined search or the user is informed about alternatives.

UC-06 — Drop pin manually **Goal:** Add a way-point at an exact map position. **Pre:** Authenticated; map UI visible. **Trigger:** User clicks on the map.

Main:

1. Create a marker at the clicked coordinates;
2. Add it to the way-point list; treat identically in routing to other way-points;

Alt/Errors: —. **Post:** Way-point added by coordinates.

UC-07 — View attraction details (popup) **Goal:** Reveal useful details for decision-making. **Pre:** Marker exists on the map. **Trigger:** User clicks the marker.

Main:

1. Show full name and feature type;
2. If available, show image, opening hours, contact, and official website (from cache or fetched on demand);

Alt/Errors: E1: enrichment fetch fails → keep basic content. **Post:** User sees concise, actionable information about the attraction.

UC-08 — Set way-point as Start **Goal:** Define the starting (and ending) way-point. **Pre:** At least two way-points exists. **Trigger:** User clicks *Set as start* on a way-point that is not the first already.

Main:

1. Promote the chosen way-point to position #1 (start);
2. If a route is currently shown, to see the route with the newly promoted way-point as first, recompute by applying UC-10 (Compute Route);

Alt/Errors: A1: when moving *Current Location* away from start, show a warning that another way-point becomes start/end. **Post:** New start is defined.

UC-09 — Delete way-point **Goal:** Remove an attraction from the routing list. **Pre:** UC-04 (Search & Select Attraction) completed; At least one way-point exists. **Trigger:** User clicks *Delete* on a way-point.
Main:

1. If a route is drawn, prompt: removing this way-point will clear the current route;
2. On confirmation, remove the way-point and marker; clear the route overlay and order badges;

Alt/Errors: A1: user cancels confirmation → no changes; A2: If the deleted way-point was the start, choose a new start (first in list); A3:If the deleted way-point was the *Current Location*, show a warning message. **Post:** Way-point list updated; route cleared, user may recompute via UC-10 (Compute Route)).

UC-10 — Compute Route (TSP) **Goal:** Obtain the most efficient closed tour and display it. **Pre:** at least two way-points; start defined. **Trigger:** User clicks *Get Route*.
Main:

1. Back-end requests the distances and durations matrices;
2. Back-end solves TSP (brute force);
3. Returns order, per-leg durations, total;
4. Updates markers with order badges, renders A→B→C line with times, and draws the poly-line;

Alt/Errors: E1: < 2 way-points → error message; E2: matrix/directions failure → show error and keep prior state if any. **Post:** Route visible; summary displayed in the message panel.

5.4 Reset Map and Logout

UC-11 — Reset Map **Goal:** Return the application to the post-login initial state. **Pre:** Authenticated; map UI visible. **Trigger:** User clicks *Reset Map*.
Main:

1. Remove all markers, poly-lines, the route summary;
2. Clear the routing list and any computed order;

Alt/Errors: A1: Based on UC-01b (Location setup): Either re-add the Current Location way-point and center the map on it, or center on default and add the note of missing current location. **Post:** Clean map; no route/way-points; centered per location policy.

UC-12 — Logout **Goal:** End the session and return to the authentication screen. **Pre:** Authenticated; back-end reachable. **Trigger:** User clicks *Logout*.
Main:

1. Invalidate the server session (clear the *HTTP session cookie*);

2. Clear the client state similar to UC-11 (Reset Map);

3. Hide the main UI and show the authentication screen;

Alt/Errors: E1: failure during the process → clear local state and login session and show auth UI. **Post:** Session terminated; next open in the same browser starts unauthenticated.

Chapter 6

Conclusions

The Map Navigation App is a tool that solves the *Travelling Salesman Problem* for touristic attractions. The application provides a user interface that works well on the laptop and integrates smoothly multiple external *APIs* from different providers: *LocationIq*, *Mapbox*, *Overpass*, *Wikidata*. The external resources have been used both to cover different topics, as fallbacks for one another, to cover for certain limitations of the *APIs* (for example, in case that the *LocationIq geocoding*fails, a new request is made to the *Mapbox geocoder* with the same query and also the country). We have handled the authentication process, the deployment, the logging process and also deployed the logs to an external service. We have implemented caching for the geocoding results to increase the efficiency of our application and to reduce the load on the external *APIs*.

Moreover, we have created a sandbox for testing different search approaches, implemented three search algorithms (the brute force approach, 2-opt and nearest neighbours), and compared their behaviour under similar circumstances as the ones needed in the application. Based on the knowledged gained from these tests we have chosen the best combination of the algorithms, such that users can get an optimal tour, fast and with up to 24 way-points. Our search approach involves using the brute force search for requests with 5 or less way-points, and the 2-opt heuristic seeded with the tour resulted from the nearest neighbours approach.

We have tried improving the user experience, by testing out the application for real use cases, and while there are certain limitations which come with the *APIs* used, we have tried mitigating them. For example, since we have discovered there are certain attractions that can not be found using our current geocoders we have implemented the possibility to add a way-point to the map by clicking. The reason some attractions can not be found be found using the *LocationIq geocoding*, is because the data is extracted from *OpenStreetMap* and there might be the case that nobody mapped the point of interest yet. Another issue that comes up is related to language, it depends how a point of interest is mapped in *OpenStreetMap*. Sometimes, it has only the name in the local language, other times it has the name in English, sometimes it has the name in multiple languages. What can

happen is that even though you are searching for an existing attraction, if your query is in a language that the attraction does not have a label for, then you will not find it. Moreover, for *LocationIq geocoding* adding extra details in the query such as the city separated by a comma helps, but for the *Mapbox geocoding* that we use as fallback, it happened that whenever an attraction was not found the center of the city was returned as a match. One other thing that could be done better is the way we cache the geocoding results. Right now we do the caching directly based on the user's query. But two people could look for the same attraction with one of them using the articulated name and the other using the non-articulated name. Another scenario would be two people looking up the same attraction but with the name in different languages.

Future improvements:

- Caching the geocoding results with the query being normalized. For this we would also need to identify the language in which the query is written. To be able to have the same cache also for multiple languages we would need to extract the labels in the different languages from *Overpass* and *Wikidata*, normalize them and store them in the cache. Finally, whenever the user would look up the same attraction, the user's query would be normalized and the cache would be searched with the filters being the language of the query and the normalized version of it.
- The authentication process, by adding an email confirmation, as with the current approach users could create multiple accounts with emails they do not have access to.
- Testing and improving the responsiveness of the application based on device size.
- Improving the cookie storage policy, so that cookies set on incognito browsers are stored for a shorter period of time.

Bibliography

[1] *Algorithms for the Travelling Salesman Problem — routific.com*. https://www.routific.com/blog/travelling-salesman-problem#three-popular-travelling-salesman-problem-algorithms. [Accessed 31-08-2025].

[2] Dominykas Jasiulionis. *What is VS Code: an overview of the popular code editor and its features — hostinger.com*. https://www.hostinger.com/tutorials/what-is-vs-code. [Accessed 26-08-2025].

[3] *Git — git-scm.com*. https://git-scm.com/. [Accessed 26-08-2025].

[4] *What is GitHub — w3schools.com*. https://www.w3schools.com/whatis/whatis_github.asp. [Accessed 26-08-2025].

[5] *What is HTTP — w3schools.com*. https://www.w3schools.com/whatis/whatis_http.asp. [Accessed 26-08-2025].

[6] *Using HTTP cookies - HTTP | MDN — developer.mozilla.org*. https://developer.mozilla.org/en-US/docs/Web/HTTP/Guides/Cookies. [Accessed 27-08-2025].

[7] *GeoJSON — geojson.org*. https://geojson.org/. [Accessed 31-08-2025].

[8] *What is an API (Application Programming Interface) - GeeksforGeeks — geeksforgeeks.org*. www.geeksforgeeks.org/software-testing/what-is-an-api/. [Accessed 31-08-2025].

[9] *HTML: HyperText Markup Language | MDN — developer.mozilla.org*. https://developer.mozilla.org/en-US/docs/Web/HTML. [Accessed 25-08-2025].

[10] *CSS: Cascading Style Sheets | MDN — developer.mozilla.org*. https://developer.mozilla.org/en-US/docs/Web/CSS. [Accessed 25-08-2025].

[11] *JavaScript | MDN — developer.mozilla.org*. https://developer.mozilla.org/en-US/docs/Web/JavaScript. [Accessed 25-08-2025].

[12] *JavaScript technologies overview - JavaScript | MDN — developer.mozilla.org*. https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/JavaScript_technologies_overview. [Accessed 25-08-2025].

[13] *JavaScript modules - JavaScript | MDN — developer.mozilla.org.* https://developer.mozilla.org/en-US/docs/Web/JavaScript/Guide/Modules. [Accessed 25-08-2025].

[14] *Leaflet — an open-source JavaScript library for interactive maps — leafletjs.com.* https://leafletjs.com/. [Accessed 25-08-2025].

[15] *Tiles - OpenStreetMap Wiki — wiki.openstreetmap.org.* https://wiki.openstreetmap.org/wiki/Tiles. [Accessed 25-08-2025].

[16] *Font Awesome — fontawesome.com.* https://fontawesome.com/. [Accessed 25-08-2025].

[17] *Live Server - Visual Studio Marketplace — marketplace.visualstudio.com.* https://marketplace.visualstudio.com/items?itemName=ritwickdey.LiveServer. [Accessed 26-08-2025].

[18] *What is GitHub Pages? - GitHub Docs — docs.github.com.* https://docs.github.com/en/pages/getting-started-with-github-pages/what-is-github-pages. [Accessed 26-08-2025].

[19] *Node.js — Run JavaScript Everywhere — nodejs.org.* https://nodejs.org/en. [Accessed 26-08-2025].

[20] *What is npm — w3schools.com.* https://www.w3schools.com/whatis/whatis_npm.asp. [Accessed 26-08-2025].

[21] MozDevNet. *Cross-Origin Resource Sharing (CORS) - HTTP | MDN — developer.mozilla.org.* https://developer.mozilla.org/en-US/docs/Web/HTTP/Guides/CORS. [Accessed 21-08-2025].

[22] *Express - Node.js web application framework — expressjs.com.* https://expressjs.com/. [Accessed 27-08-2025].

[23] *cors — npmjs.com.* https://www.npmjs.com/package/cors. [Accessed 27-08-2025].

[24] *cookie-parser — npmjs.com.* https://www.npmjs.com/package/cookie-parser. [Accessed 27-08-2025].

[25] *dotenv — npmjs.com.* https://www.npmjs.com/package/dotenv. [Accessed 27-08-2025].

[26] *winston — npmjs.com.* https://www.npmjs.com/package/winston. [Accessed 27-08-2025].

[27] *axios — npmjs.com.* https://www.npmjs.com/package/axios. [Accessed 27-08-2025].

[28] *bcrypt — npmjs.com*. https://www.npmjs.com/package/bcrypt. [Accessed 27-08-2025].

[29] *country-state-city — npmjs.com*. https://www.npmjs.com/package/country-state-city. [Accessed 27-08-2025].

[30] *simplify-geojson — npmjs.com*. https://www.npmjs.com/package/simplify-geojson. [Accessed 31-08-2025].

[31] *MongoDB - Working and Features - GeeksforGeeks — geeksforgeeks.org*. https://www.geeksforgeeks.org/mongodb/what-is-mongodb-working-and-features/. [Accessed 27-08-2025].

[32] *What is MongoDB Atlas? - Atlas - MongoDB Docs — mongodb.com*. https://www.mongodb.com/docs/atlas/. [Accessed 27-08-2025].

[33] *Cloud Application Platform | Render — render.com*. https://render.com/. [Accessed 27-08-2025].

[34] *Monitor, Debug and Improve Your Entire Stack — newrelic.com*. https://newrelic.com/. [Accessed 27-08-2025].

[35] *REST Client - Visual Studio Marketplace — marketplace.visualstudio.com*. https://marketplace.visualstudio.com/items?itemName=humao.rest-client. [Accessed 27-08-2025].

[36] *Search / Forward Geocoding — docs.locationiq.com*. https://docs.locationiq.com/docs/search-forward-geocoding. [Accessed 27-08-2025].

[37] *Matrix API — docs.locationiq.com*. https://docs.locationiq.com/docs/matrix-api. [Accessed 27-08-2025].

[38] *Optimization API v1 | API Docs — docs.mapbox.com*. https://docs.mapbox.com/api/navigation/optimization-v1/. [Accessed 27-08-2025].

[39] *Matrix API | API Docs — docs.mapbox.com*. https://docs.mapbox.com/api/navigation/matrix/. [Accessed 27-08-2025].

[40] *Directions API | API Docs — docs.mapbox.com*. https://docs.mapbox.com/api/navigation/directions/. [Accessed 27-08-2025].

[41] *Geocoding API | API Docs — docs.mapbox.com*. https://docs.mapbox.com/api/search/geocoding/. [Accessed 27-08-2025].

[42] *Static Images API | API Docs — docs.mapbox.com*. https://docs.mapbox.com/api/maps/static-images/. [Accessed 31-08-2025].

[43] *Overpass API - OpenStreetMap Wiki — wiki.openstreetmap.org*. https://wiki.openstreetmap.org/wiki/Overpass_API. [Accessed 27-08-2025].

[44] *Wikidata:SPARQL query service - Wikidata — wikidata.org*. https://www.wikidata.org/wiki/Wikidata:SPARQL_query_service. [Accessed 27-08-2025].

[45] *W3Schools.com — w3schools.com*. https://www.w3schools.com/howto/howto_css_search_button.asp. [Accessed 24-08-2025].

[46] Josh W. Comeau. *A Modern CSS Reset • Josh W. Comeau — joshwcomeau.com*. https://www.joshwcomeau.com/css/custom-css-reset/. [Accessed 24-08-2025].

[47] GitHub. *Quickstart for GitHub Pages - GitHub Docs — docs.github.com*. https://docs.github.com/en/pages/quickstart. [Accessed 21-08-2025].

[48] Render. *Streaming Render Service Metrics — render.com*. https://render.com/docs/metrics-streams. [Accessed 21-08-2025].

[49] New Relic. *Use TCP endpoint to forward logs to New Relic | New Relic Documentation — docs.newrelic.com*. https://docs.newrelic.com/docs/logs/log-api/use-tcp-endpoint-forward-logs-new-relic/. [Accessed 21-08-2025].

Appendix A

Logging

For logging we have defined a logger object, created using the*winston npm package*, for which we have defined settings for: console display, transporting to local files (a combined file containing all the logs, an error log, and daily log files), and a way of processing objects added for logging, to be able to add our own defined errors.

Then, to adapt to streamline the logs from our controllers we have created a loggerHelper module, which contains the methods displayed in Source Code 7.

```
const defaultLog = (req, startTime) => {
    const log = {};
    if (req) {
        log.route = req?.originalUrl;
        log.method = req?.method;
        log.email = req?.user?.email || "";
    }
    if (startTime) {
        log.responseTime = `${Date.now() - startTime} ms`;
    }
    return log;
}
export const infoLog = (req, startTime, message = "") => {
    const log = defaultLog(req, startTime);
    log.statusCode = 200;
    if (message) log.message = message;
    logger.info(log);
}
export const errorObj = (req, startTime, error) => {
    const log = defaultLog(req, startTime);
    log.statusCode = error?.statusCode || 500;
    log.message = error?.message || "An error occurred";
    if (error?.details) log.details = error.details;
    return log;
}
```

Source Code 7: loggerHelper methods.

For all the logs we are adding the timestamp, and we try extracting as much information as possible from the req object. Then for the informational and warning logs we have a similar approach where we call the logger directly inside the loggerHelper. For the errors, the approach is different due to to the fact that the error codes are more diverse than the success codes, and there are scenarios where we log our own error objects and scenarios where we log the error objects that we catch during the execution as it can be seen in the Sample Code 8.

```
const METHOD_FAILURE_MESSAGE = "geocodeController failed.";
// Scenario 1: Logging an error created by us
if (!place) {
err = ERROR_OBJECTS.BAD_REQUEST("place");
logger.error(METHOD_FAILURE_MESSAGE, errorObj(req, startTime, err));
return res.status(err.statusCode).json(err);
}
// Where ERROR_OBJECTS is defined in constants as:
export const ERROR_OBJECTS = {
    BAD_REQUEST: (missingField) =>  {
        return {
          statusCode: 400,
          message: "Missing required inputs. Ensure that all necessary fields
          ↪  are included or calculated correctly.",
          details: `Missing field: ${missingField}`
        }
    },
    FRONTEND_INTERNAL_SERVER_ERROR: {
      statusCode: 500,
      message: "Internal server error. Please try again later.",
    },
// Scenario 2: Logging the caught error
  } catch (error) {
    logger.error(METHOD_FAILURE_MESSAGE, errorObj(req, startTime, error));
    return res.status(500).json(ERROR_OBJECTS.FRONTEND_INTERNAL_SERVER_ERROR);
  }
```

Source Code 8: Usage of the loggerHelper methods in the controllers.

Appendix B

Search Solvers Comparison

For being able to properly compare the different algorithms we have implemented, to properly choose the most appropriate for our use case we have created the following testing structure.

We have extracted 25 attractions (the maximum number of attractions for which we can compute the durations matrix) from Bucharest with their coordinates and we have stored them in a constants file.

Then, to create different scenarios on each of the runs, we are shuffling the 25 attractions, then extracting some of them (based on a query parameter) and then we call one of the APIs which provides us the directions matrix.

The next step is to call one of the search algorithms we implemented (based on another query parameter), providing as parameter the directions matrix.

The last step is to create an image based on the result of the search algorithm. Call the Mapbox Directions API for obtaining the LineString geometry that is connecting the coordinates through the road, then add markers with order labels and add them to the GeoJSON.

Finally call *Mapbox Static Images* with the GeoJSON object and save the resulting image locally.

Whenever we were trying to create images for the maximum number of way-points (24), we noticed that the *Mapbox Static Images API* was throwing an error that the request was too large. The problem was that the GeoJSON object got too large, so we had to add *simplify-geojson npm-package* to reduce the dimensionality (based on a tolerance parameter). This resulted in the issue being solved, but less qualitative routes being displayed.

To not affect all the images, we have handled separately the error code thrown in case of too large of a request. Moreover, even in the cases where the initial request fails, we are increasing the tolerance incrementally and checking whether the parameter length has been shortened enough, until making the second request, this way maximizing the quality of the route.

The comparison results can be seen in Table B.1.

Nodes	Solver	Avg Exec Time (ms)	Avg Duration (Cost)	Ex.T. St Dev	Num of Trials
5	BF	0.933	**00:40:42**	0.506	3
5	NN	0.076	00:41:55	**0.026**	3
5	2-OPT	**0.057**	00:40:54	0.028	3
5	2-OPTNN	0.650	00:41:09	0.699	3
9	BF	1306.721	**01:15:37**	88.420	3
9	NN	**0.053**	01:29:15	**0.037**	3
9	2-OPT	0.115	01:17:40	0.041	3
9	2-OPTNN	0.125	01:17:21	0.052	3
15	NN	**0.036**	01:59:21	**0.025**	10
15	2-OPT	0.112	**01:45:38**	0.096	10
15	2-OPTNN	0.697	01:46:40	1.160	10
24	NN	**0.087**	02:41:15	**0.098**	10
24	2-OPT	2.280	02:29:35	3.776	10
24	2-OPTNN	3.233	**02:26:31**	5.016	10

Table B.1: Comparison of search solvers over different waypoint counts, showing average execution time, duration (cost), and execution time standard deviation. Best values per group are highlighted in bold.

What we have noticed when increasing the number of trials, as it can be seen in Table B.2, is that for all the algorithms the the average running time went down significantly. For example 2-OPTNN had an average for 24 almost 5 times smaller on 100 runs, than on 10 runs. We were not able to observe a large discrepancy between 2-OPT, and 2-OPTNN, but the tour durations were marginally lower when using 2-OPTNN, so we have decided to use that in our implementation.

In the tables below we are making the following associations:

- Avg Exec Time (ms) = the average execution time in milliseconds taken to finish the algorithm execution, and the average represents the arithmetic mean, considering the values from all the trials; one caveat here, for 2-OPTNN we summed up the runtime of both NN and 2-OPT algorithms, to provide a fair comparison;
- Avg Duration (Cost) = the arithmetic mean of durations (initially in seconds), then formatted in hh:mm:ss format for better comparison;

- Ex.T. St Dev = the standard deviation of the algorithm execution time;
- Num of Trials = the number of trials
- BF = the brute force algorithm;
- NN = the nearest neighbour heuristic;
- 2-OPTNN = the 2-opt heuristic seeded by the resulting tour of nearest neighbour;

Nodes	Solver	Avg Exec Time (ms)	Avg Duration (Cost)	Ex.T. St Dev	Num of Trials
5	BF	0.622	**00:42:22**	0.203	100
5	NN	0.013	00:44:38	**0.006**	100
5	2-OPT	**0.011**	00:43:29	0.011	100
5	2-OPTNN	0.028	00:43:33	0.013	100
15	NN	**0.024**	01:52:16	**0.009**	100
15	2-OPT	0.076	01:42:42	0.175	100
15	2-OPTNN	0.141	**01:41:44**	0.223	100
24	NN	**0.029**	02:38:41	**0.017**	100
24	2-OPT	0.599	02:27:36	2.026	100
24	2-OPTNN	0.794	**02:24:59**	2.199	100

Table B.2: Comparison of search solvers over different waypoint counts with 100 trials each, showing average execution time, duration (cost), and execution time standard deviation. Best values per group are highlighted in bold.

What can be noticed from the Figures B.1, B.2, and B.3 is that there is a larger discrepancy in terms of duration between the global optimal result and the local optimal result provided by the 2-opt approaches, than between the 2-opt approaches and the nearest neighbour heuristic result. But in terms of how the GeoJSON LineString is drawn they are quite similar to the brute force algorithm result. But, from these runs and the result from the table we have decided for up to 5 it is worth using the brute force algorithm as the average runtime is not that high, and the users receive the best possible solution for their scenario.

For nine way-points the discussion is drastically changed, the brute force approach is around 10000 times slower than the 2-OPT approaches and we also know that from 10 way-points onwards the brute force is unusable.

Something interesting that can be seen by comparing the two tables, is that even though the average went down drastically, with the number of trials for 2-OPT approaches, the standard deviation, especially in the scenario with the maximum number of way-points remained very high.

So while the results of are promising, we see that the influence of the specific scenario that needs to be mapped is very important.

In Figures B.8, B.9 it is visible that the algorithm did not reach its local optimum, and probably a larger number of iterations would have helped in this case. Still, the solutions are better than the one from Figure B.5.

Regarding Figures B.6, and B.7, these look like great routes, with 8 and 6 minutes faster than the route from Figure B.4.

Figure B.1: The global optimal route obtained with the brute force algorithm with the duration of 00:59:40.

Figure B.2: An optimal route obtained with both 2-opt and 2-optNN heuristics with the duration of 01:02:16.

Figure B.3: The route obtained with the nearest neighbour heuristic with the duration of 01:03:44.

Figure B.4: The route obtained with Nearest Neighbour for 15 way-points with the duration of 01:46:22.

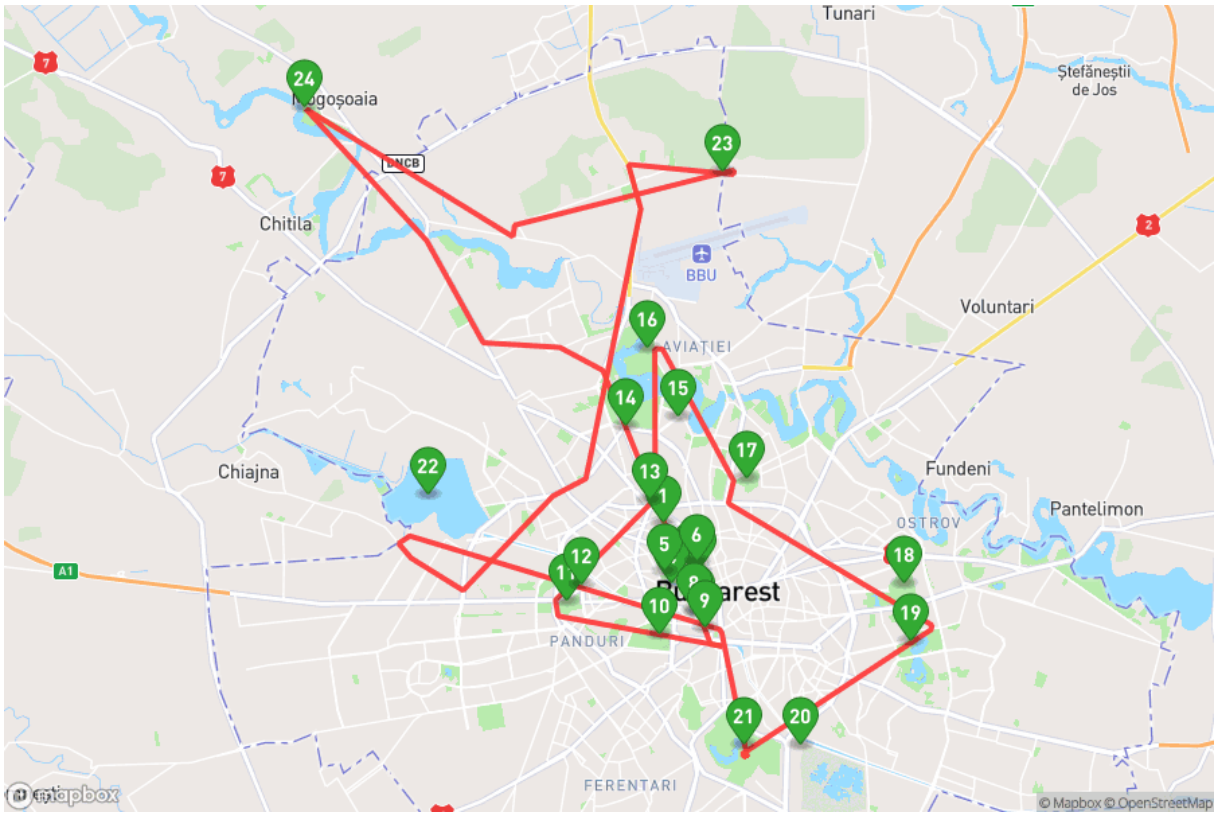

Figure B.5: The route obtained with Nearest Neighbour for 24 way-points with the duration of 02:38:48.

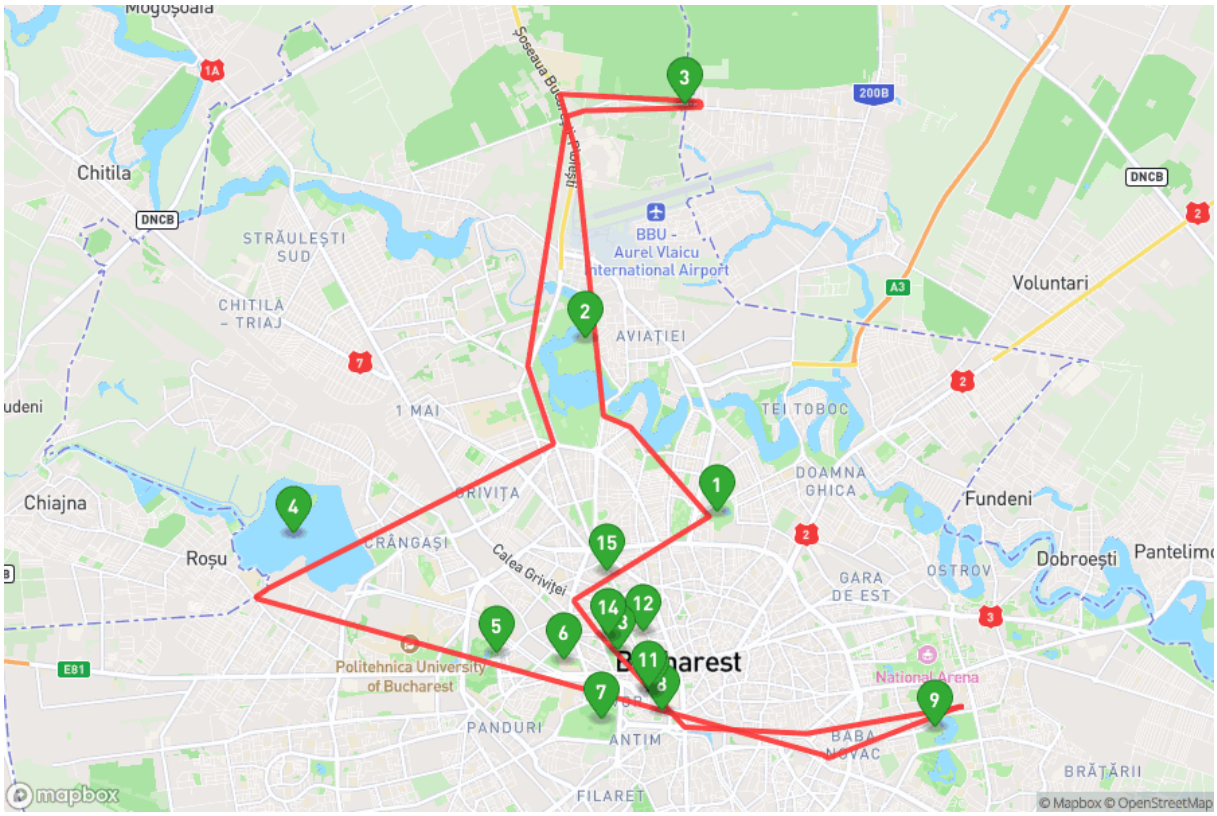

Figure B.6: The route obtained with 2-opt for 15 way-points with the duration of 01:40:18.

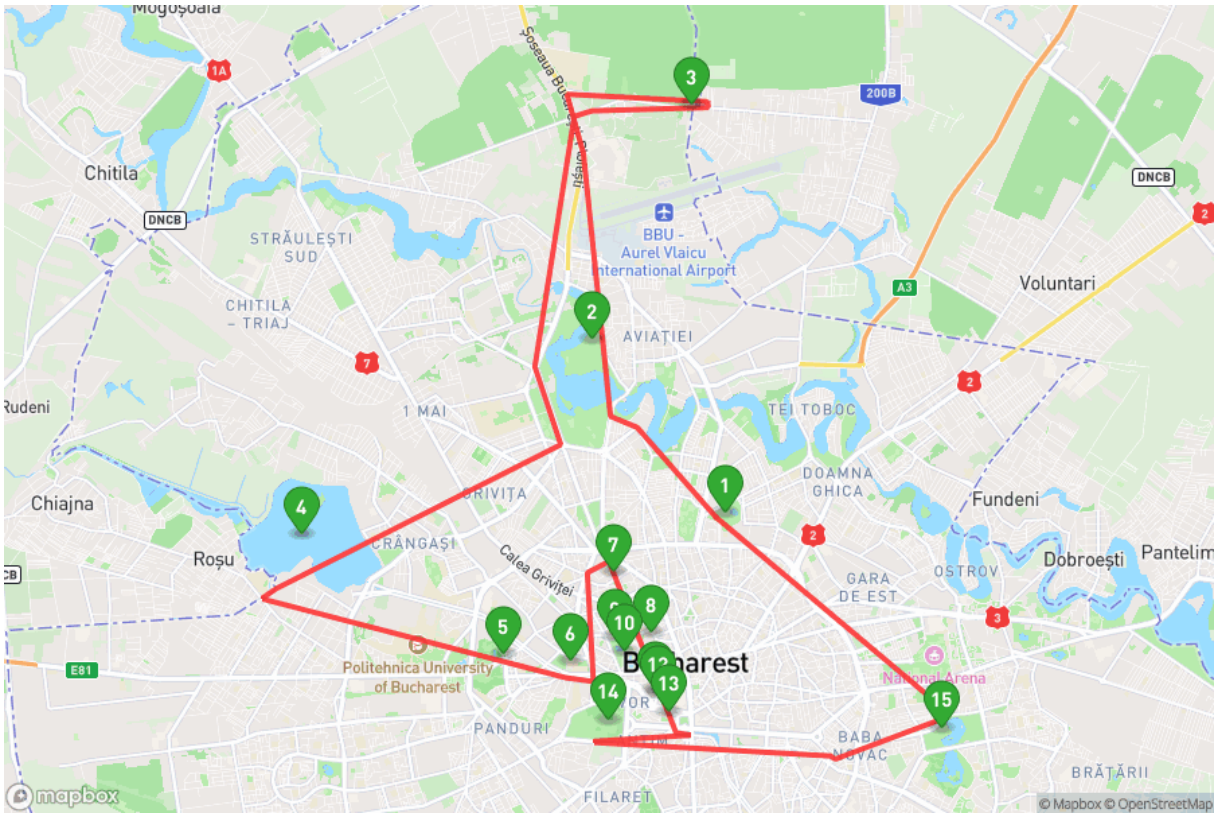

Figure B.7: The route obtained with 2-optNN for 15 way-points with the duration of 01:38:01.

Figure B.8: The route obtained with 2-opt for 24 way-points with the duration of 02:29:44, maximum iterations of 1000 reached.

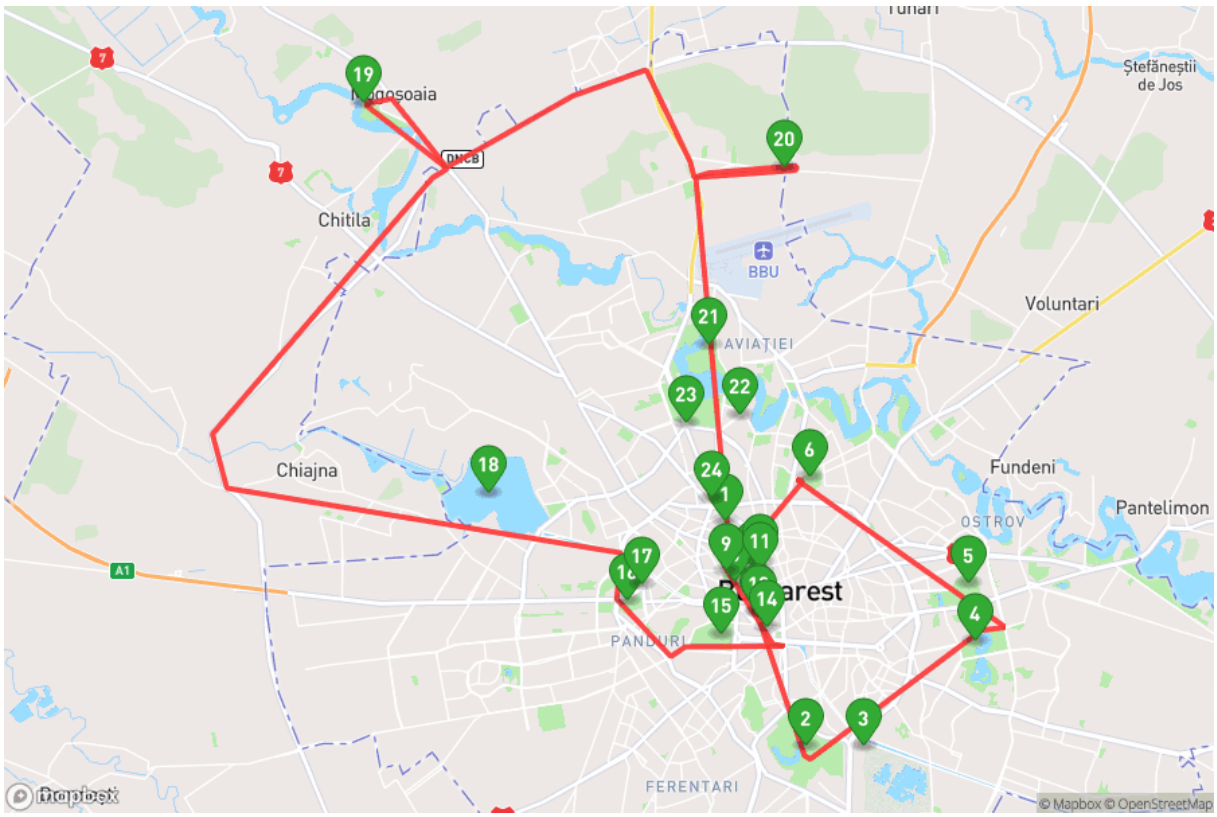

Figure B.9: The route obtained with 2-optNN for 24 way-points with the duration of 02:22:13, maximum iterations of 1000 reached.