Silesian
University
of Technology

SILESIAN UNIVERSITY OF TECHNOLOGY

FACULTY OF AUTOMATIC CONTROL, ELECTRONICS AND COMPUTER SCIENCE

PROGRAMME: INFORMATICS

Final Project

Design and implementation of web application used for solving vehicle routing problems with time windows

author: Łukasz Kwiecień

supervisor: Tomasz Jastrząb, PhD

Gliwice, February 2022

Contents

Abstract

Transportation is one of the most critical activities in the supply chain. A good solution to the problem of vehicle routing is very important for improving logistics and its costs. The objective of this thesis is to design and implement a web application for solving the vehicle routing problem with time windows. The paper presents the process of creating a tool that allows the end user to easily define and solve the vehicle routing problem with time windows.

The paper describes the design and development process of the application, reviews the literature on vehicle routing problems and the different methods used to solve them. The tools and technologies used in the project are also presented, as well as an insight into the application architecture and its modules. Finally, the results achieved and the perspectives and possibilities for further development of the project are discussed.

Keywords: vehicle routing, vehicle routing problem with time windows, web application development

Chapter 1

Introduction

Transportation is one of the most critical activities in the supply chain. Its importance comes from the fact that the transportation costs can reach up to almost 40% of the total logistics costs of a manufacturing company [36]. For that reason, the companies need to transport the goods or persons efficiently. That goal can be achieved by either minimizing the distance traveled by the vehicles or reducing the number of routes required to visit all destinations. The Vehicle Routing Problem (VRP) describes the problem of assigning loads to the vehicles and sequencing the customers assigned to each vehicle to obtain optimal routes.

VRP is one of the most studied combinatorial optimization problems. Its popularity is due to the fact that VRP can be used in many real-life scenarios in the fields of distribution, collection, and logistics. In a VRP, the fleet of vehicles has to visit a set of customers starting from a given depot and deliver commodities to them, taking into consideration all given constraints. There are various constraints that could be introduced to the problem, for example, the time windows for the customers or the capacity of vehicles.

Depending on the problem variant, VRP is a combination of two or more NP-hard problems, which makes VRP also NP-hard. The fact that the problem is NP-hard makes the process of finding an exact solution very time-consuming. Therefore it is necessary to use a heuristic approach to get good solutions in an acceptable time.

1.1 Objective and scope of the thesis

In this thesis a web application solving Capacitated Vehicle Routing Problem with Time Windows (CVRPTW) is considered. The user defines the problem by picking waypoints on the map and determining the capacity and time window constraints. Application, if feasible, solves the problem and presents the results in a user-friendly way by visualizing all routes on the map. The problem is solved using Push Forward Insertion Heuristic and optimized using Local Search with λ interchange method. The fleet of vehicles is homogeneus, which means that every vehicle has the same amount of goods that can be transported in it. Every customer has its own time window within which the delivery must be made. The goal was to create a tool that will simplify the process of defining and solving the CVRPTW for the end user.

1.2 Content outline

The second chapter provides history and explores scientific background material for the Capacitated Vehicle Routing Problem with Time Windows. Moreover it provides formal and detailed definition of the problem and the solution methods based on the researched literature. Chapter 3 focuses on the design and implementation process of the project. It shows the functional and nonfuctional requirements as well as diagrams describing the system. The tools and metodologies used for the design and implementation process are also described there. The fourth chapter covers the hardware and software requirements for the application. It provides the step by step installation procedure along with the user manual. The usage examples and screenshots of the working application can also be found there. Chapter 5 discusses the software part of the project in detail. Description of the architecture, code and structure of the project is explained in this chapter. Furthermore, it specifies in detail the software-side of the entire process of solving the CVRPTW. Chapter 6 shows how the project was tested and verified if the requirements set during the design process were fulfilled. The last chapter summarizes the achieved results and describes the encountered difficulties.

Chapter 2

Problem analysis

2.1 Vehicle Routing Problem

The problem concerns a set of customers to whom products need to be delivered in such a way that the delivery cost is as low as possible. Transport is performed by a fleet of vehicles, each of which can carry a certain number of goods. The goal is to serve all customers with as few vehicles as possible and to keep the total distance travelled by all vehicles as short as possible within predefined constraints. The classic version of VRP has the following constraints:

1. Each vehicle's route starts and ends at a depot.
2. All goods must be delivered.
3. Customer must receive all goods at one time delivered by one vehicle.
4. The vehicle may not carry more goods than its capacity allows.

The VRP can be formally defined as a directed graph $G = (V, A)$, where $V = \{0, \ldots, n\}$ is the set of vertices representing customers and the depot, and A is the set of arcs (i, j) connecting these vertices. The set of vehicles is denoted by W. The depot is marked as vertex 0. The number of vehicles is denoted by m, and each vehicle has a capacity of Q. The cost of travel between vertices i and j is denoted by c_{ij}. The cost is the distance, duration or other costs that may occur during the travel between nodes. The demand of customer i is denoted by

q_i. The customer subset $S \subseteq V \setminus \{0\}$ can be served by a minimum of $r(S)$ vehicles. This number can be obtained by solving the Bin Packing Problem (BPP) with bin set S with capacities Q, but because BPP is an NP-hard problem, it can be approximated by its lower bound: $\lceil \sum_{i \in S} q_i / Q \rceil$ [10].

The Integer Linear Programming formulation of the problem is as follows [40]:

$$minimize \sum_{i,j \in V} c_{ij} x_{ij} \tag{2.1}$$

subject to:

$$\sum_{i \in V} x_{ij} = 1 \qquad \forall j \in V \setminus \{0\} \tag{2.2}$$

$$\sum_{j \in V} x_{ij} = 1 \qquad \forall i \in V \setminus \{0\} \tag{2.3}$$

$$\sum_{j \in V} x_{0j} = m \tag{2.4}$$

$$\sum_{i \in V} x_{i0} = m \tag{2.5}$$

$$\sum_{i \notin S} \sum_{j \in S} x_{ij} \geq \lceil \sum_{i \in S} q_i / Q \rceil \qquad \forall S \subseteq V \setminus \{0\}, S \neq \emptyset \tag{2.6}$$

$$x_{ij} \in \{0, 1\} \qquad \forall i, j \in V \tag{2.7}$$

In this formula, x_{ij} is a binary variable whose value indicates whether the arc between vertices i and j is traversed in the solution. If the connection between these vertices belongs to the solution, then the variable x_{ij} is equal to 1, otherwise it is equal to 0. The *indegree* Eq. (2.2) and *outdegree* Eq. (2.3) constraints guarantee that every customer is visited only once. Constraints (2.4) and (2.5) ensure that the solution has m routes, one for each vehicle (the number of routes does not exceed the number of vehicles). Constraints (2.6) are called *capacity-cut constraints* and refer to the minimum number of vehicles needed to serve a set of customers whose total demand is $\sum_{i \in S} q_i$. These constraints ensure the connectivity of the solution

and that there are no paths whose load exceeds the capacity of the vehicle. The last constraint (2.7) ensures that variables x_{ij} are binary. If x_{0j} is equal to 1 it means that j belongs to this route. Any route from the solution can be reconstructed in this way: the next customer in the route is customer i, for whom x_{ij} is equal to 1. The last customer is customer i for whom x_{i0} is equal to 1 [40].

The problem of vehicle routing was first introduced by Dantzig and Ramser in 1959 as "The truck dispatching problem" [14]. In the following years it became very popular, resulting in many studies, books and scientific publications exploring the topic. The problem is one of the most studied combinatorial optimisation problems since it offers many benefits for logistics and transport companies. Different implementations can save as much as 5% to 30% of transport-related costs [20].

2.2 Vehicle Routing Problem with Time Windows

The Vehicle Routing Problem with Time Windows (VRPTW) is a variant of VRP in which a constraint due to time windows is added. In this variant of the problem, each customer i has a time window $[a_i, b_i]$ in which it must be served. A customer cannot be served earlier than a_i and later than b_i. In case the vehicle arrives earlier than a_i it must wait until the window opens. Moreover, to each arc (i, j) the time t_{ij} is assigned. For each vertex i and vehicle k a decision variable s_{ik} is defined. This variable represents the time at which the vehicle k starts to serve the customer i [4]. This variant of VRP is also an NP-hard problem [32].

The VRPTW can be stated mathematically as multicommodity network flow formulation with time windows and capacity constraints [4]:

$$minimize \sum_{k \in W} \sum_{i,j \in V} c_{ij} x_{ijk} \tag{2.8}$$

subject to:

$$\sum_{k \in W} \sum_{i \in V} x_{ijk} = 1 \qquad \forall j \in V \setminus \{0\} \tag{2.9}$$

$$\sum_{i \in V \setminus \{0\}} q_i \sum_{j \in V} x_{ijk} \leq Q \qquad \forall k \in W \tag{2.10}$$

$$\sum_{j \in V} x_{0jk} = 1 \qquad \forall k \in W \tag{2.11}$$

$$\sum_{i \in V} x_{ihk} - \sum_{j \in V} x_{hjk} = 0 \qquad \forall h \in V \setminus \{0\}, \forall k \in W \tag{2.12}$$

$$\sum_{i \in V} x_{i0k} = 1 \qquad \forall k \in W \tag{2.13}$$

$$x_{ijk}(s_{ik} + t_{ij} - s_{jk}) \leq 0 \qquad \forall i, j \in V, \forall k \in W \tag{2.14}$$

$$a_i \leq s_{ik} \leq b_i \qquad \forall i \in V, \forall k \in W \tag{2.15}$$

$$x_{ijk} \in \{0, 1\} \qquad \forall i, j \in V, \forall k \in W \tag{2.16}$$

Objective function (2.8) minimizes travel cost. Constraints (2.9) and (2.10) ensure that each customer is visited once and vehicle cannot surpass its capacity. Equations (2.11), (2.12), and (2.13) indicate that each vehicle must leave the depot, once it has reached the customer it must travel to the next location and finally return to the depot. Inequality (2.14) shows the relationship between vehicle departure time and its successor. Inequalities (2.15) provide that the time windows are observed. Constraints (2.16) are the integrality constraints [4].

2.3 Solution approaches

There are two approaches to solving VRPs. The first is to solve using exact methods, which are guaranteed to find the optimal solution. However, these methods are very expensive in terms of memory and computation time, which makes them applicable only to problems of low complexity. The second approach is to use approximate methods (heuristics), which allow solving larger and more complex

problems in a reasonable time. However, heuristics do not guarantee finding the optimal solution.

2.3.1 Exact methods

The exact methods were divided by Laporte and Norbert's into three families: Direct Tree Search methods, Dynamic Programming and Integer Linear Programming [24].

The Direct Tree Search family includes branch and bound algorithms. The solution using this algorithm was first proposed by Christofides and Eilon in 1969 [5]. However, the algorithm was only able to solve problems up to 13 clients, due to time complexity. Over the years, the methods in this family have been improved, allowing solutions for problems with up to 25 [6] clients and even, in later years, 250 [23].

The Dynamic Programming solutions were, at first, able to solve problems up to 25 customers. Later, significant improvements were made that allowed to solve problems up to 50 customers.

The Integer Linear Programming methods produced some of the best exact solutions. This has been achieved by extending the set partitioning algorithm using the column generation method [15].

2.3.2 Heuristics

Heuristics do not guarantee the best solution—their aim is to obtain a solution close to the best, in a relatively short time. They make it possible to solve larger and more complex problems that would not be possible to solve with exact methods. For VRP, we can distinguish between two types of heuristics: construction and improvement. The process of solving a VRP usually can be divided into two steps: an initial solution is created using the construction heuristic, and then the improvement heuristics improves the solution.

Construction heuristics

One of the best known construction heuristics is Clarke and Wright's savings algorithm. In the first step, separate routes are created for each customer, even if the number of vehicles is insufficient. Then, the savings for all connections between edges (i, j) are calculated using the formula $s_{ij} = c_{i0} + c_{0j} - c_{ij}$. After that, the savings are sorted in the descending order and the algorithm tries to merge two routes that will reduce the total costs of travel. Routes are merged by connecting the customer from one route to the customer from the second route. A merge between i and j is only possible if they are the first or last vertices of their routes and the vehicle capacity constraint is not violated. The algorithm stops when there are no more possible connections.

Another type of Construction herustics is insertion heuristics. The Push forward insertion heuristics used in this thesis will be discussed in detail later in the chapter.

The last type of construction heuristics are two-phase heuristics. They divide the problem into two sub-problems and then solve each of them. One such method is cluster first, route second. In this approach, customers are clustered into a base set of routes. Each cluster is treated as a separate TSP instance. There are various methods of clustering customers, one of which is the sweep algorithm. The sweep algorithm builds routes by sweeping a ray, centered at the depot, clockwise adding each customer lying on the line of the circle to the cluster.

Improvement heuristics

Once an initial solution is found using the construction heuristics, it is improved using the improvement heuristics. One type of such heuristics is the local search heuristic, in which the best solution is selected from the neighbourhood until a stopping criterion is reached. Neighbourhood solutions are created from the current one by applying operators that modify the original solution.

Local search with λ interchange heuristic will be discussed later in the chapter. This improvement heuristic method was used in the project.

Simulated annealing is another example. This method replicates the process of heating up a material and then slowly lowering the temperature to reduce defects.

The main assumption of the algorithm is to infrequently accept worse solutions in order to avoid a local optimum. The solution is accepted with a probability calculated by means of a function depending on the difference in quality between the new and the current solution, and a parameter called the current temperature.

Another method is Tabu search. Based on the initial solution, neighbouring solutions are determined and the best of these solutions is selected as the new best solution. The previous solutions are stored in a tabu list, which has a predefined length, and when it is full, adding another solution removes the oldest one from the list. Such a mechanism avoids a local optimum and the return to previous solutions.

VRP can be solved using many other heuristics, including: Large Neighbourhood Searches, Genetic Algorithms or Ant Colonies.

2.3.3 Push forward insertion heuristic

In this thesis, the initial solution is created using Push Forward Insertion Heuristic. It was introduced by Solomon [35] in 1987 and it is an efficient method to insert customers into routes. The algorithm creates a new route by selecting the starting customer furthest from the depot and then inserts each unassigned customer into the route until the capacity or time constraints are met. Then new route is created and the process repeats until all customers are routed. The feasibility check ensures that all constraints are not violated. This method generates a good starting solution for later improvements, in a short time. For the pseudocode for PFIH algorithm for VPRTW see Figure 2.1 [37].

2.3.4 Local search with λ interchange

The λ interchange neighbourhood generation method was introduced by Osman and Christofides [30]. Local Search is performed by interchanging clients between routes. The interchanges are performed through operators whose number depends on the λ parameter. In this work, this parameter is set to 2, which means that the maximum number of interchanged clients between routes will be 2. For this parameter value, we can distinguish 8 interchange operators: (0, 1), (1, 0), (1, 1), (0, 2), (2, 0), (2, 1), (1, 2), (2, 2). Operator (1, 1) means that one customer will

```
1  Begin an empty route r_0 starting from the depot; i := 0;
2  Among all unassigned customers, select the customer
     farthest from the depot inserting it into the current
      route r_i;
3  if all customers are routed then goto line 13.
4  else
5      if the capacity Q of the vehicle k involved in the
         current route r_i is exceeded then goto line 12.
6      else
7          foreach unassigned customer
8              find the best insertion position in r_i without
                 violating any specified time window to
                 compare the cost of starting a route against
                  that of the best position found.
9  if there exists no feasible insertion position in the
     current route r_i then goto line 12.
10 else
11     Pick the customer with the greatest cost difference,
         insert it into r_i updating the capacity Q of the
         vehicle k involved. goto line 3.
12 Start a route r_{i+1} starting from the depot; i := i + 1; goto
     line 2.
13 Return the current solution
```

Figure 2.1: Push Forward Insertion Heuristic for Vehicle Routing Problem with Time Windows

be transferred from route R_i to R_j and also one customer will be transferred from route R_j to R_i. Operator (2, 1) means that two customers will be moved from route R_i to R_j and only one customer will be moved from route R_j to R_i. The other operators work on the same principle. If the solution has improved after the interchange of customers between routes, it is then accepted, otherwise such a solution is rejected [38].

To select between the candidate solutions a global-best strategy was chosen. The global-best strategy searches all solutions in the neighbourhood of the current solution for all operators and selects the one that gives the greatest cost reduction. The local search is stopped after a fixed number of iterations or other specified stopping criterion [38].

The algorithm can be summarised as follows: After generating a starting solution its entire neighbourhood (generated using the operators) is searched for a candidate solution. The candidate becomes the new current solution and its neighbourhood is searched to find a better solution. In case there are no more solutions in the neighbourhood that improve the current solution the algorithm is stopped.

Chapter 3

Requirements and tools

Establishing software requirements is an important part of the software development process. They allow to outline the final effect that the created project should achieve. Requirements help to speed up the software development process and help to deliver a better product.

A functional requirement describes what the system should do, while a non-functional requirement describes how the system should do it. In this project, the following functional and non-functional requirements were defined:

3.1 Functional requirements

1. The application should be accessible to any user without having to create an account.

2. The user should be able to define the problem using the world map.

3. There should be a simple user guide on how to use the main functionality.

4. The user should not be able to send incomplete or invalid data to the backend server.

5. The user should be able to freely modify data at any stage of the problem definition process.

6. The user should be able to go back and modify the problem, even if it has been resolved.
7. The solution should be visualized on the map.
8. Details of each route in the solution should be displayable.
9. The user should be able to upload and solve his own benchmark file.
10. Every uploaded benchmark file must be a ".txt" file with data in a correct format.
11. The user should be able to see the solutions calculated by the algorithm for each instance of Solomon Benchmark.
12. The user should be able to compare solutions from the algorithm with the best known Solomon Benchmarks solutions.
13. The algorithm should return the information if the problem is feasible or not.
14. The user should be able to navigate between application pages.

3.2 Non-functional requirements

1. The application should be compatible with any of today's most popular web browsers (Firefox, Edge, Safari, Chrome, Opera (or other Chromium-based browser)).
2. The application should work correctly on any desktop operating system.
3. The system should be able to solve large problems in a reasonable time.
4. System functionalities should be easy to understand and use.
5. The application should be time zone independent.
6. The application should display validation errors below HTML inputs.

7. The application should display notifications in a toast pop-ups.

8. The system should be integrated with the OSRM API to solve problems using real routes.

3.3 Tools and technologies

.NET

.NET is an open source software platform. It was created by Microsoft and allows developers to create many different types of applications. Applications developed using .NET are modern, flexible and cross-platform. The ASP.NET Framework, which is part of the .NET platform, offers a set of tools to help build web applications and APIs.

The backend project used the then latest version of the .NET 5 platform. The .NET platform contains all the necessary tools needed to develop web applications, which made the development process easier.

C#

C# is an open source, modern object-oriented programming language developed by Microsoft. The range of possibilities it offers is wide, and versatility is one of C#'s strongest points. Many different types of applications can be created using this language, such as web, mobile, desktop and cloud applications. The C# community is very large, which is why it is the 8th most popular programming language among professional programmers [29]. Of all the languages available on the .NET platform, C# is the most commonly selected one [18].

The backend project uses version 9 of the C# language, which was the latest version of the language at the time. The cross-platform aspect of the language was also very important, as the application was not to be designed for just one operating system.

NuGet

NuGet is the open source package manager for .NET plaftorm created by Microsoft. It contains almost 300 000 unique packages which can extend the standard .NET features [11]. It allows the developers to share and consume useful code easily.

Angular

Angular is an open-source JavaScript framework maintained by Google. As of 2021, Angular was the third most popular web framework among professional developers [29]. Angular provides a set of powerful features for efficient single-page application development. It allows developers to build web, mobile and even desktop applications.

Angular was chosen as the framework for the front-end application because of the possibilities it offers. An important aspect for this project was the support for maintaining data through services, which facilitates the sharing of data between components [26].

Typescript

Typescript is a strongly typed and object-oriented language that is a super set of JavaScript. It was introduced by Microsoft and extends JavaScript with e.g. types and interfaces [12]. The frontend client is powered by TypeScript as the primary language for Angular.

Node.js

Node.js was created by Google as a open-source backend JavaScript runtime environment. Node.js uses Chrome v8 engine to convert JavaScript code to machine code [27]. For this project, Node.js was one of the dependencies required by Angular.

npm

The Node package manager (npm for short) is the package manager for the Node JavaScript platform. Npm is responsible for installing and managing the dependencies used by Node.js. Angular applications depend on npm packages, which is why it was used in the project.

git

Every software project needs a version control system to improve workflow and make it easier to manage multiple versions of source files. Git, created by Linus Torvalds, is the most popular of the version control systems, which is why it was chosen for use on the project [29]. With the ability to store functionality in separate branches and control previous versions of each file, the entire software development process becomes more efficient.

Docker

Docker is a virtualization software which simplifies the process of building and running applications. It separates the software from the host infrastructure by placing the application in an isolated environment (container). Inside the environment, the same version of each dependency needed by the application will be provided each time [16]. As a result, only the installation of Docker on the host system is required to run the application.

PostgreSQL

PostgreSQL is an open source object-relational database system. This database offers a huge variety of features and capabilities. PostgreSQL is becoming more popular every year, according to the latest 2021 report PostgreSQL is the second most popular database management system [29]. This database system was chosen because of the capabilities it offers.

Chapter 4

External specification

4.1 Hardware and software requirements

To run and use the application a computer with Windows, Linux or MacOS operating system is needed. There is a possibility to use the application on mobile web browsers, but it can be unstable and some functionalities may not work as intended. Therefore the desktop device is preferred. The application requires an internet connection and can be used with Firefox, Edge, Safari, Chrome, Opera (or other Chromium-based web browser) browsers.

The application is containerised to not force the user to download and install dependent software and to make it easy to use. Docker is the only software that is needed to run and host the application.

4.2 Installation and activation

Due to containerisation, the installation procedure is very simple and requires no other software or dependencies than Docker. An internet connection is also required during installation. The information about required frameworks and dependencies is encapsulated within the Docker image. During initialisation, Docker downloads the necessary frameworks and dependencies, builds the web application, creates the database, its structure and inserts the necessary data.

Prerequisite

Docker installed and running.

Installation

To start the application the following command must be run from the main directory of the application (docker-compose.yml file location):

```
docker-compose up
```

The initialization of the application can be longer for the first time. On the first start-up, the application may take a few minutes to initialise and may restart several times (waiting for the database to initialise).

Once the application has been successfully initialised, it should be available at http://localhost:5000.

4.3 User manual

4.3.1 Navigation bar

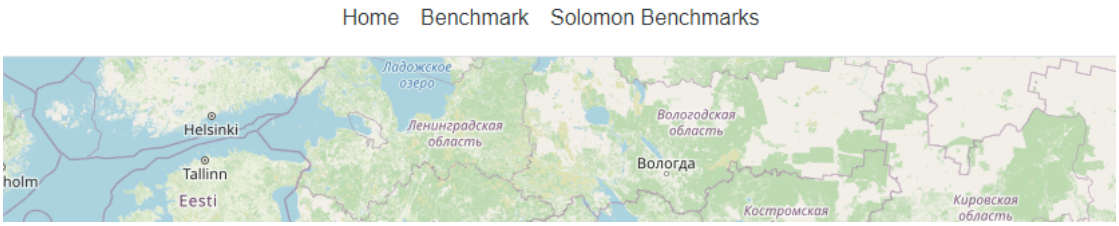

Figure 4.1: Navigation bar

The navigation bar (Fig. 4.1) is visible at the top of every page in the application, it contains 3 buttons:

- Home – redirects to the home page.
- Benchmark – redirects to a page where the user can upload the benchmark file and solve the problem found in the file.

- Solomon Benchmarks – redirects to a page with a table comparing OptiRoute (this project) solutions and the best known solutions for Solomon benchmark instances.

4.3.2 Home Page

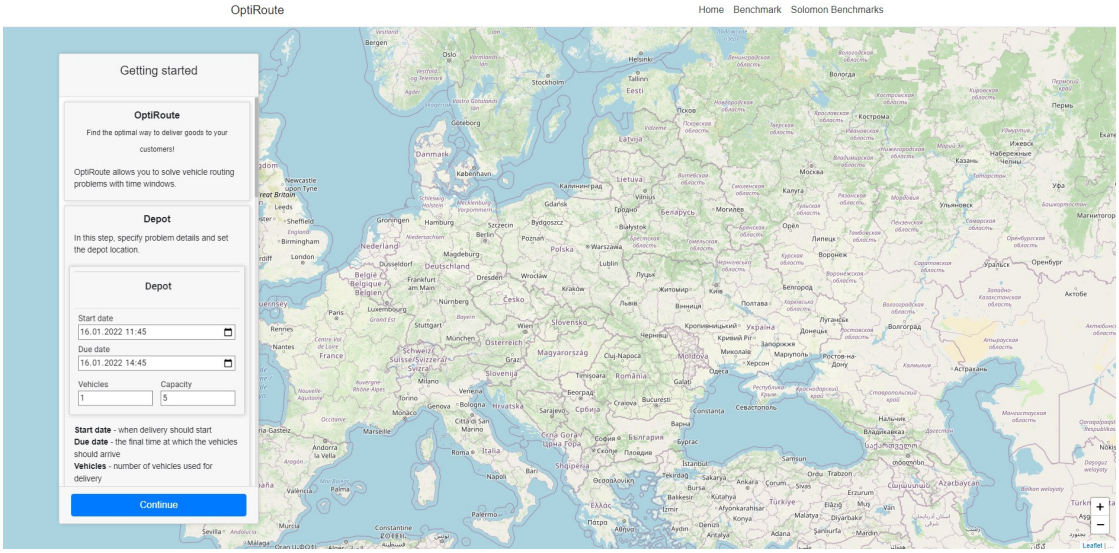

Figure 4.2: Home page

The home page (Fig. 4.2) consist of the map and the sidebar. The user can zoom in, zoom out, move freely around the map and add markers with a mouse click (Fig. 4.3). The sidebar consists of required inputs for data or details depending on the current step of the problem definition process.

4.3.3 Sidebar

The sidebar is where the user can define the details of the problem as well as see the details of the solution. The whole process is divided into individual steps, so the content of the sidebar depends on which stage the user is currently at.

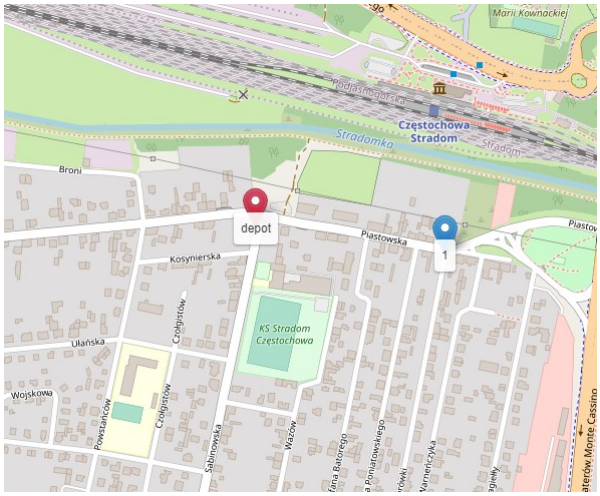

Figure 4.3: Markers placed on the map

Getting started

The “Getting Started” step in the sidebar (Fig. 4.4) provides a short user guide, explaining step-by-step how to use the application. Screenshots and short descriptions of the steps to follow make it easier for first-time users to use the application. After clicking the “Continue” button at the bottom of the sidebar, the contents of the sidebar will change to the "Depot" step.

Depot

In the “Depot” step, the user must place the depot marker by clicking on the desired location on the map and then specify the details in the input fields in the sidebar (Fig. 4.5).

At the bottom of the sidebar (Fig. 4.5) there are two buttons, “Back” and “Continue”, which allow the user to return to the previous step or move on to the next step.

The "Depot" form (Fig. 4.5) contains input fields:

- Start date – time when the delivery should start.
- Due date – the final time at which the vehicles should arrive at the depot.
- Vehicles – number of vehicles used for delivery.

- Capacity – the capacity of each vehicle.

The validation rules applied to the input data:

- Start date – is required and cannot be greater than or equal to the Due date.
- Due date – is required and cannot be less than or equal to the Start date.
- Vehicles – is required and must be an integer.
- Capacity – is required and must be an integer.

When the input data is invalid it will display appropriate message and disable the “Continue” button, preventing the user from proceeding to the next step (Fig. 4.6). The “Continue” button will also be inactive if the depot marker is not placed on the map.

Customers

The “Customers” section allows the user to place customer markers on the map and specify the details of each customer. Customer markers are placed in the same way as depot markers. When a location is selected, a separate section for that customer will appear in the sidebar, marked with the customer number in the left corner (Fig. 4.7).

At the bottom of the sidebar (Fig. 4.7) there are two buttons, “Back” and “Solve”, which allow the user to return to the "Depot" step or solve the problem and go to the presentation of the results.

Each customer form (Fig. 4.7) contains input fields:

- Ready time – the earliest date on which the customer can take delivery.
- Due date – final date on which the customer may take delivery.
- Service time – time taken to accept delivery.
- Demand – quantity of goods in delivery.

The validation rules applied to the input data:

- Ready time – is required, cannot be greater than or equal to the Due date and must be within the depot time window.
- Due date – is required, cannot be less than or equal to the Ready time and must be within the depot time window.
- Service time – is required.
- Demand – is required and must greater than 0.

As with the "Depot" form, when the data entered is invalid, it will display appropriate message and disable the "Solve" button, preventing the user from solving the problem (Fig. 4.8). The "Solve" button will also be inactive if there are no customers.

Solution

When the problem is feasible, the solution (routes) is drawn on the map, and the details of the entire solution and each route are shown in the sidebar (Fig. 4.9).

At the bottom of the sidebar (Fig. 4.9) there is "Back" button which allows the user to clear the routes from the map and come back to the previous step.

The user can highlight one of the routes on the map by hovering the mouse over the route details in the sidebar (Fig. 4.10).

4.3.4 Benchmark

The "Benchmark" page (Fig. 4.11) allows the user to solve a problem contained in a benchmark file. This file must be in .txt format and must match the Solomon Benchmark instance file structure [33]. The "Solve" button sends the problem to the backend server. If the format of the selected file is invalid, a message is displayed informing the user about the problem (Fig. 4.12). If the loaded text file does not match the Solomon Benchmark structure, a message is returned indicating the line where the validation error occurred (Fig. 4.13). When the file is successfully solved, the solution details are displayed along with a button to visualise the solution on the image (Fig. 4.14).

4.3.5 Solomon Benchmarks

The “Solomon Benchmarks” page contains a table with the best known solutions and the solutions generated by the application for the Solomon benchmark problem instances (Fig. 4.15). The user can compare the “OptiRoute” solutions for each instance with the best solution. The table can be sorted and contains 6 pages with a total of 56 records. Each row has two buttons “OptiRoute” and “Best”, which allow the user to visualise the solutions of the current row on the image.

4.3.6 Visualization

The “Visualisation” dialog, as mentioned earlier, can be displayed on the “Benchmark” and “Solomon Benchmarks” pages by clicking the appropriate button.

It contains a simple visualisation of the solution to the selected problem. Each customer is represented by a dot, customers belonging to the same route are connected by lines and have the same colour (Fig. 4.16).

4.3.7 Error

If an unexpected error occurs, a corresponding message is displayed in the top right corner of the page (Fig. 4.17). After a short time, the entire page reloads and is ready to be used again.

4.4 Security issues

The application is intended to be accessible to any user, so no authentication or authorisation functions have been implemented in it. Therefore, no user security issues were encountered during the development of the application. The application is safe from SQL Injection attacks, this vulnerability is protected by Entity Framework Core library.

Validation in the application takes place on both the server and client side. Proper multi-level validation prevents unwanted data from being sent to the server.

4.5 Usage and working scenarios

The software can be used by different types of users, such as a company that wants to streamline its logistics and plan the delivery of goods to customers or an individual who needs to pick up and transport multiple people from different locations to the same destination. In addition, the application can be used by someone who wants to test the algorithm on own benchmark instances.

Working scenarios can be differentiated according to the functionality that will be used:

- Main functionality with map (Home page)
- Benchmark functionality (Benchmark page)

4.5.1 Main Functionality

The procedure for using the main functionality is always the same, the user must define the problem in the "Depot" (Figs. 4.18 and 4.19) and "Customers" (Figs. 4.20 and 4.21) steps. The problem is then sent to the backend server and processed by the algorithm. When the problem is feasible, an appropriate message is displayed and the solution is visualised on a map and described in detail in the sidebar (Fig. 4.22). Otherwise the user is informed that the defined problem cannot be solved by an appropriate message appears on the screen (Fig. 4.23).

4.5.2 Benchmark functionality

User uploads the benchmark file (Figs. 4.24 and 4.25) and clicks the "Solve" button. The file is then sent to the backend server and processed by the algorithm. If the problem is solvable a corresponding message appears, details of the solution are displayed and a button to visualize the result (Fig. 4.26). If the problem cannot be solved an appropriate message is displayed (Fig. 4.27).

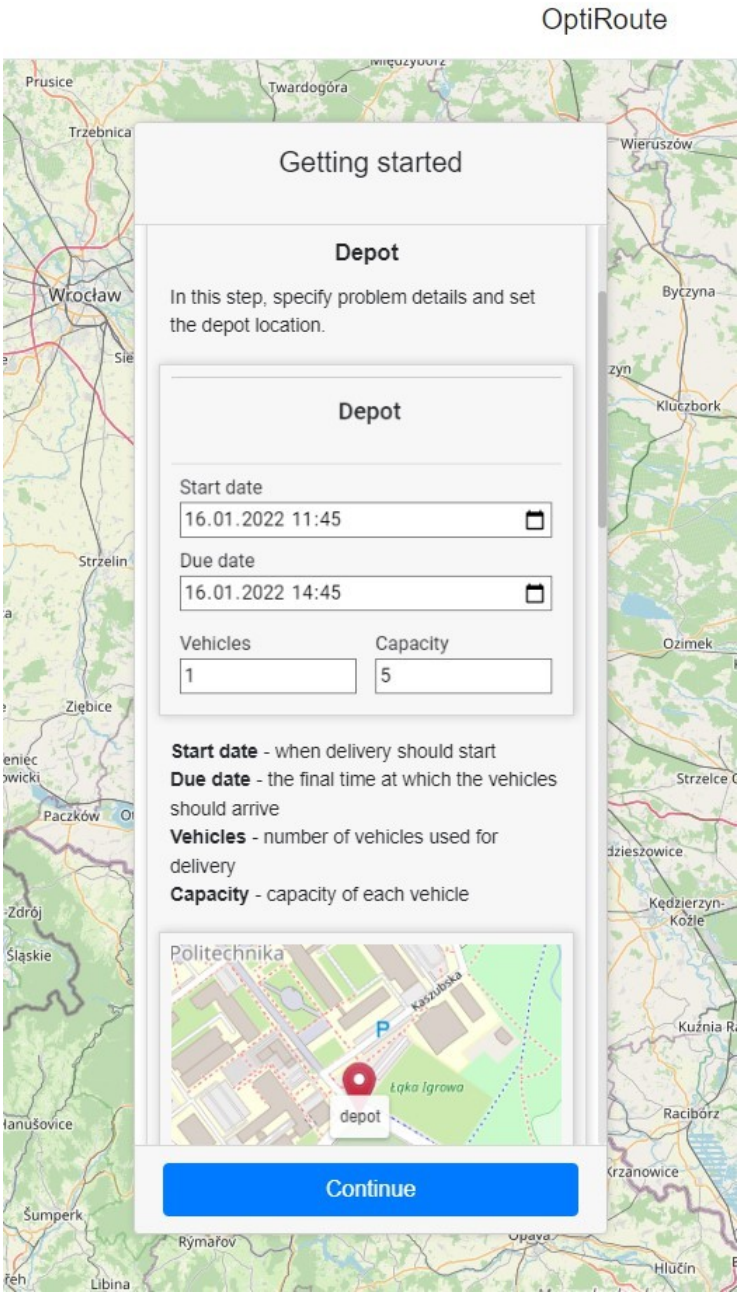

Figure 4.4: Getting started step in the sidebar

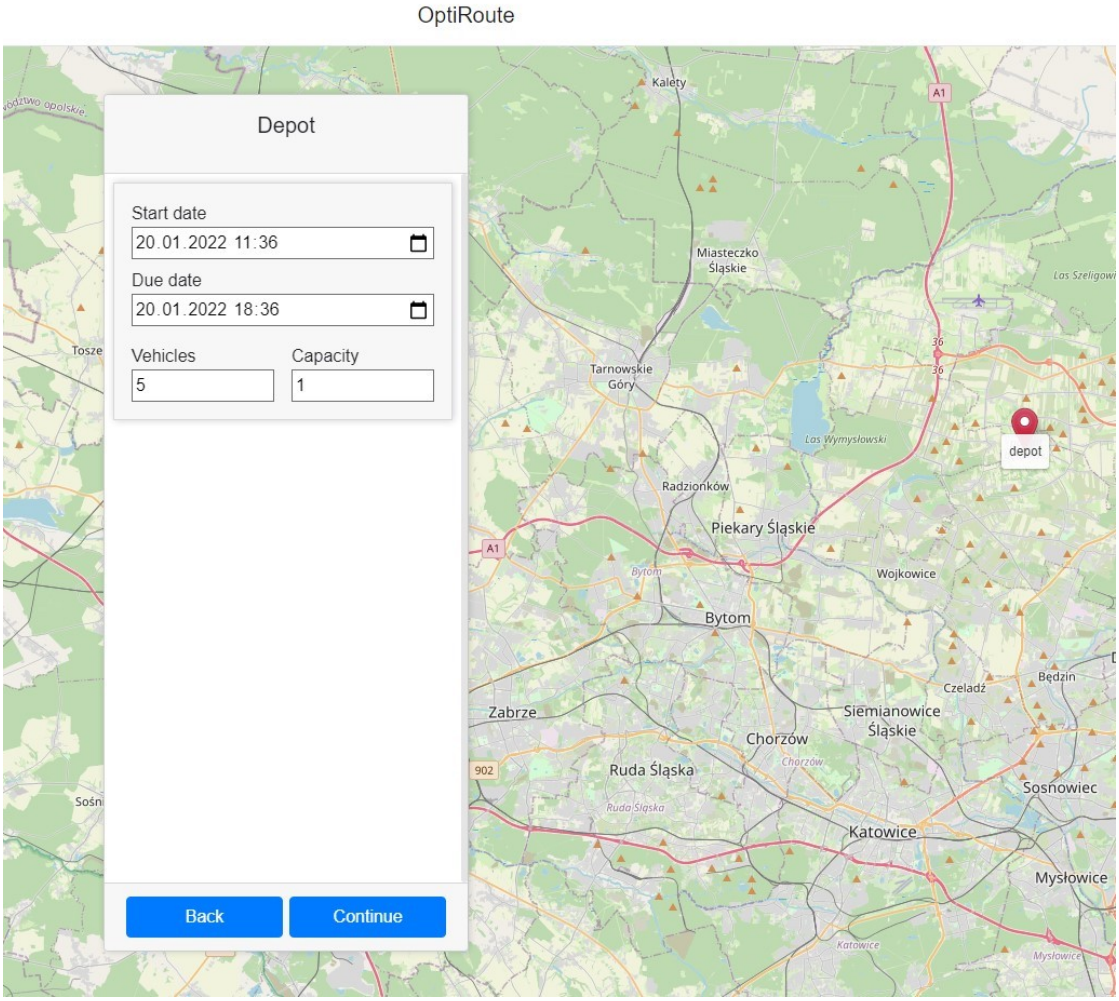

Figure 4.5: Depot step

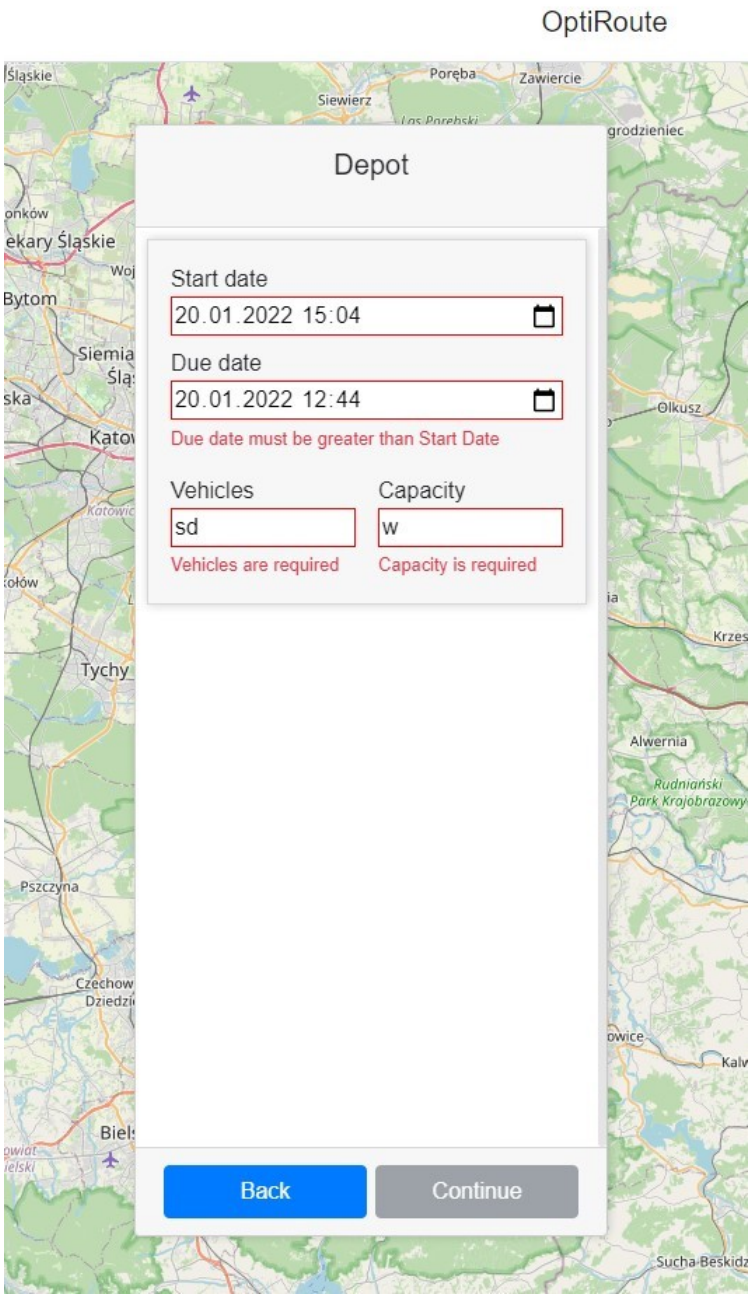

Figure 4.6: Depot form with invalid data

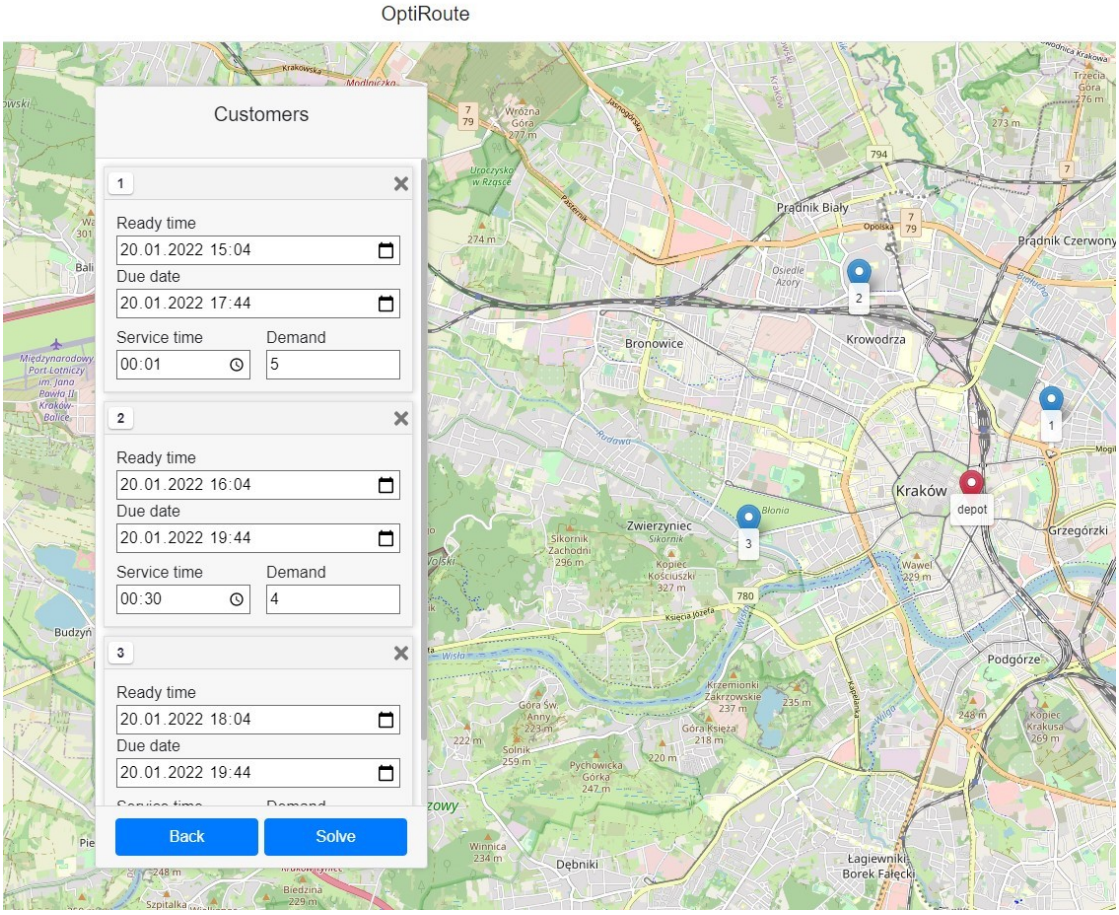

Figure 4.7: Customers step

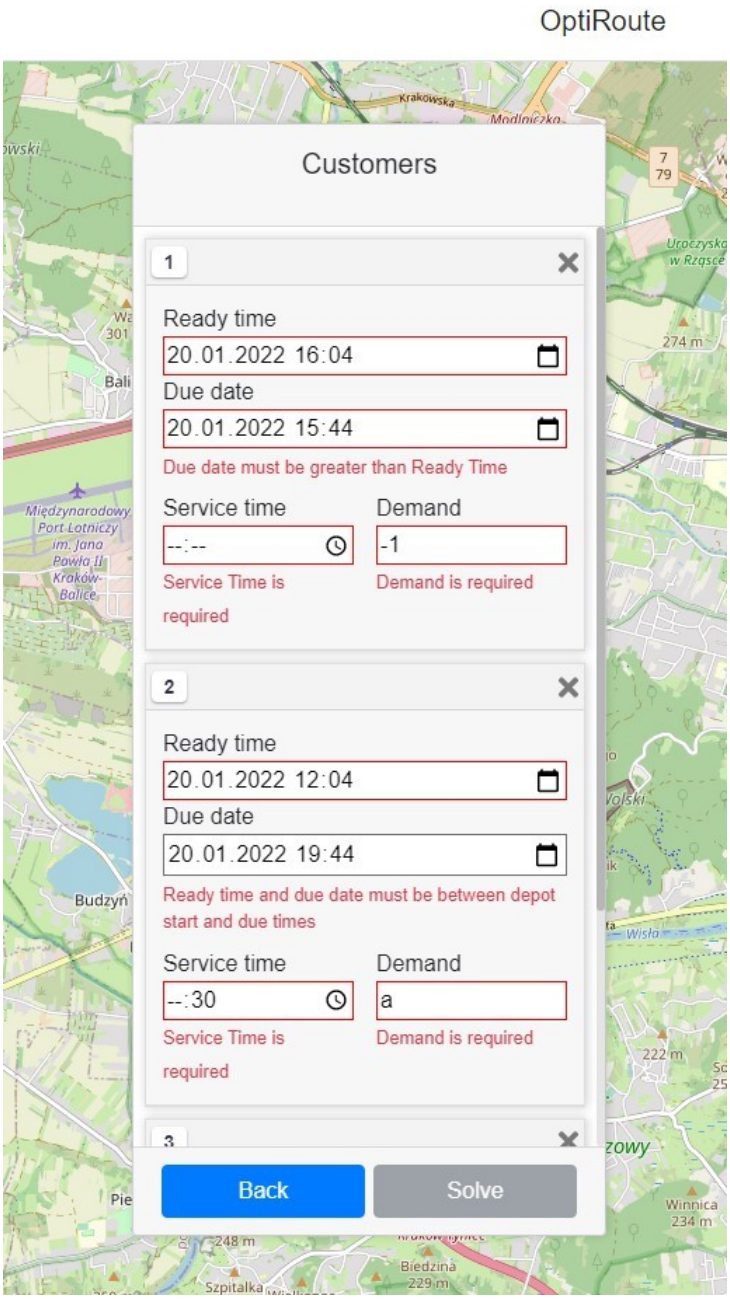

Figure 4.8: Customers forms with invalid data

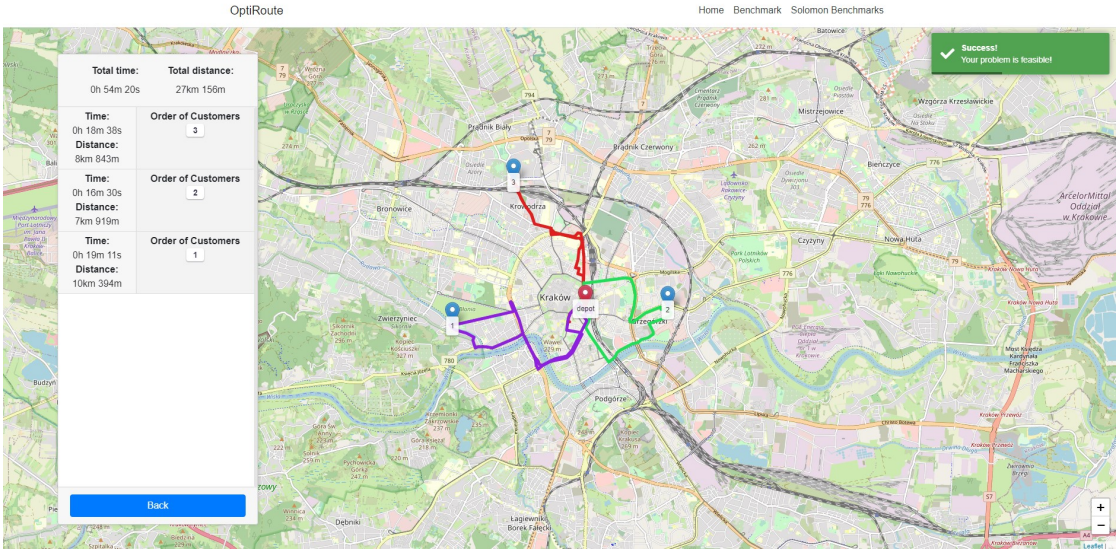

Figure 4.9: Solution

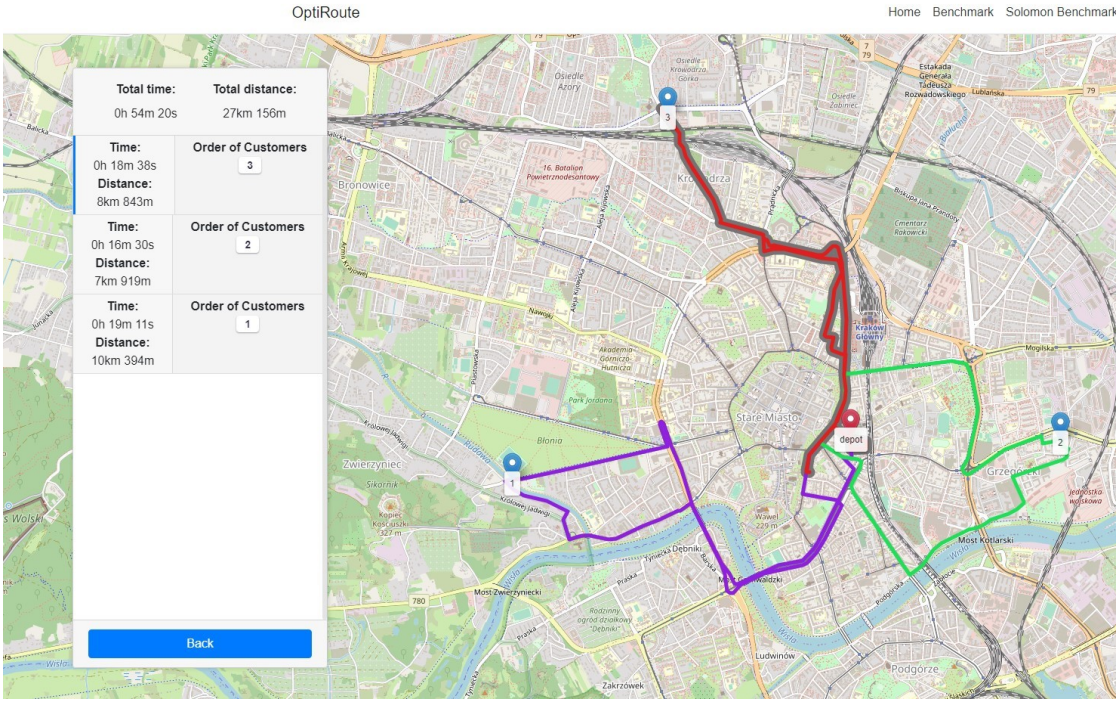

Figure 4.10: Highlighted route

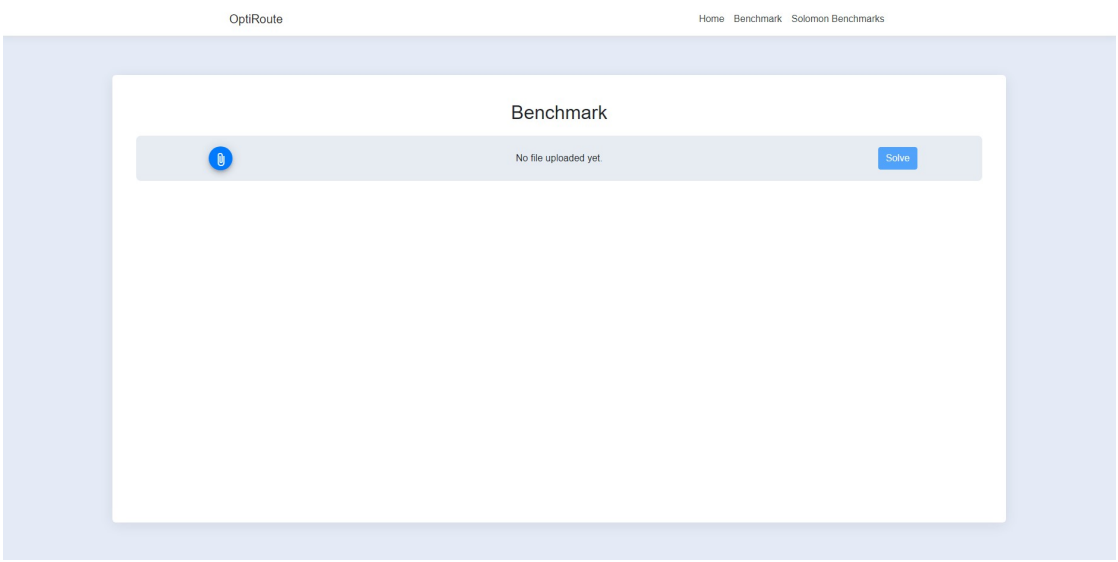

Figure 4.11: Benchmark page

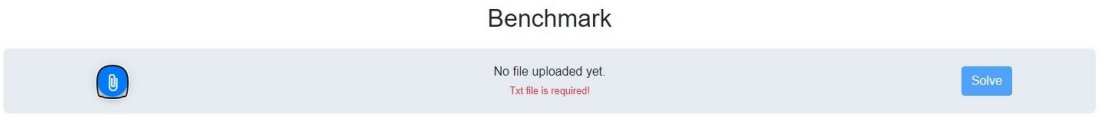

Figure 4.12: Invalid benchmark file format

Figure 4.13: Benchmark file validation error

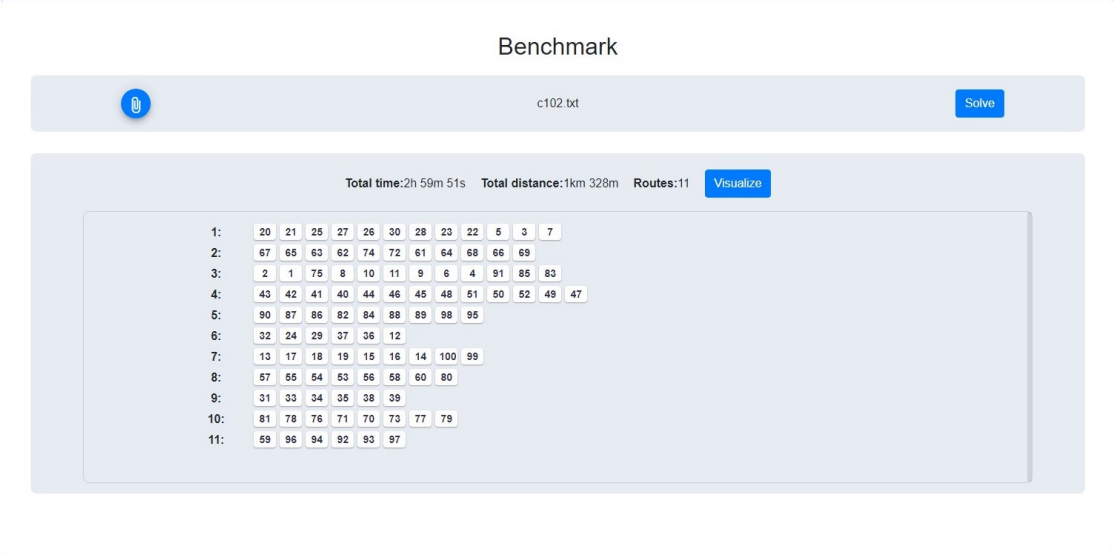

Figure 4.14: Details of the benchmark solution

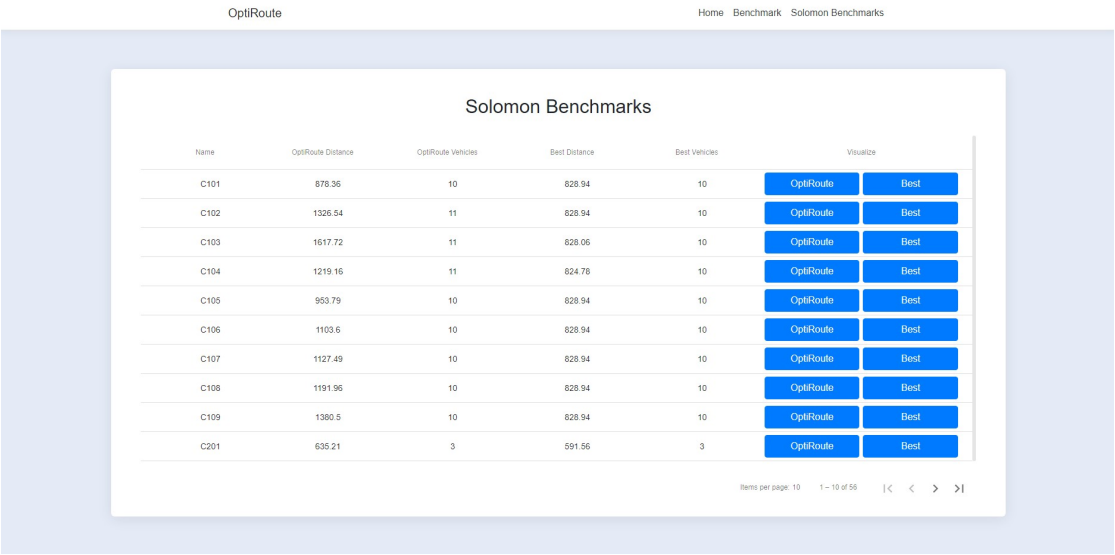

Figure 4.15: Solomon Benchmarks page

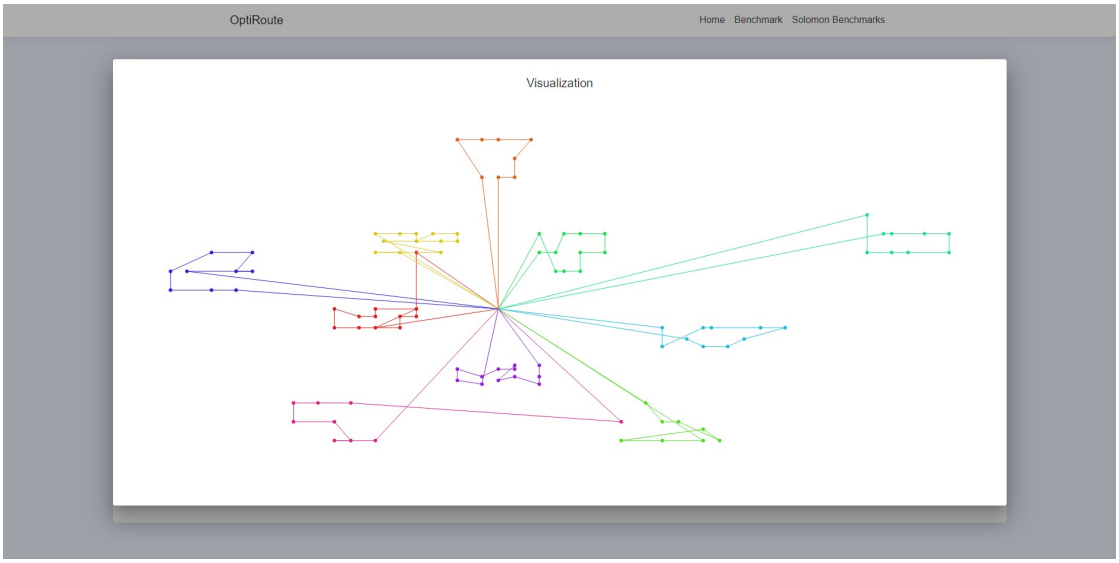

Figure 4.16: Visualization dialog

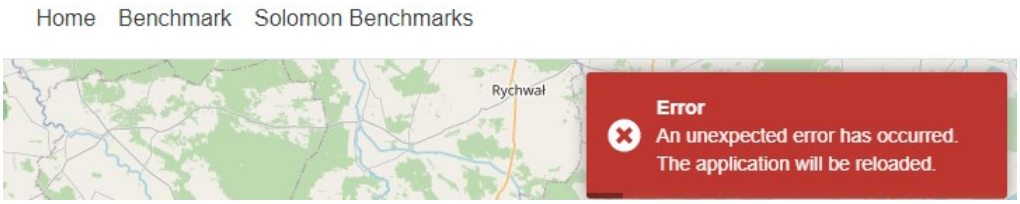

Figure 4.17: Error message

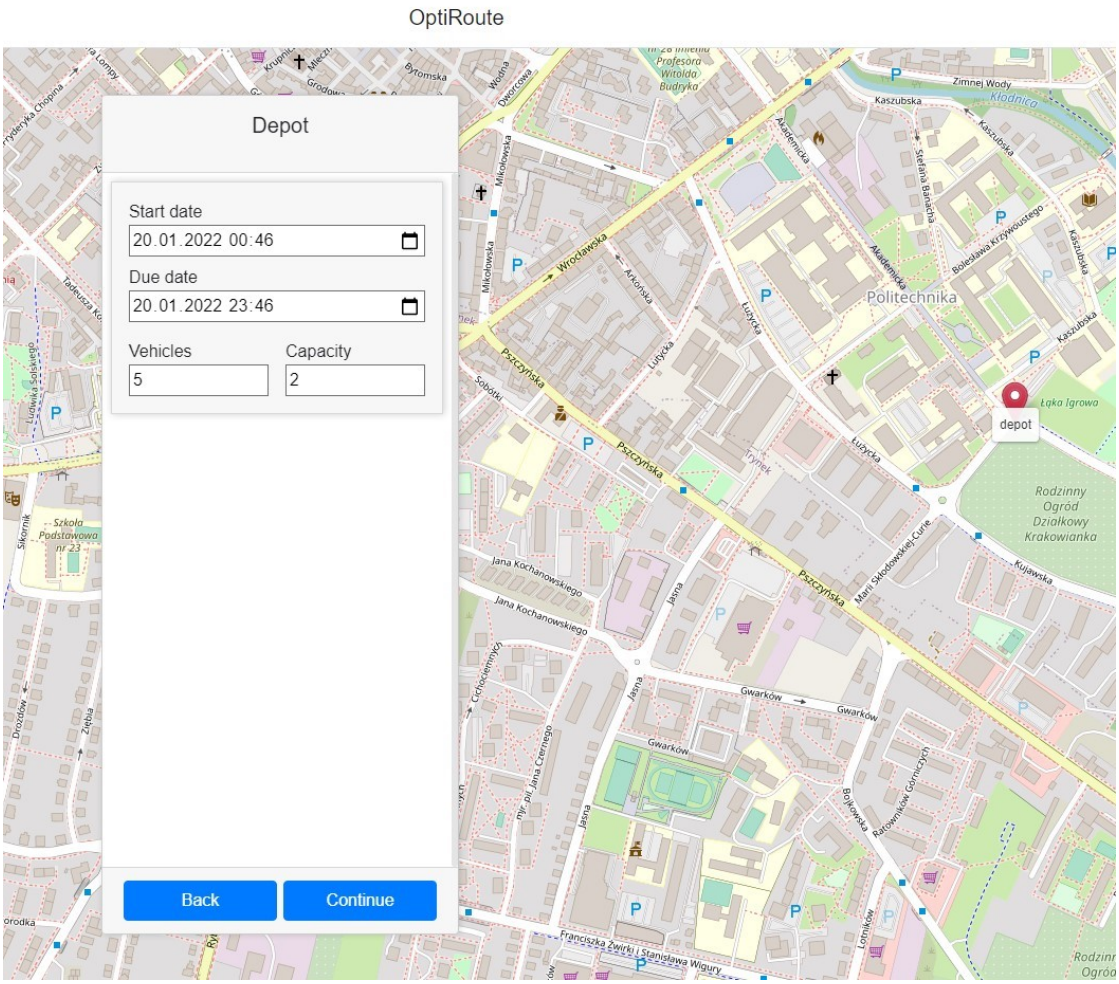

Figure 4.18: Usage example 1.1

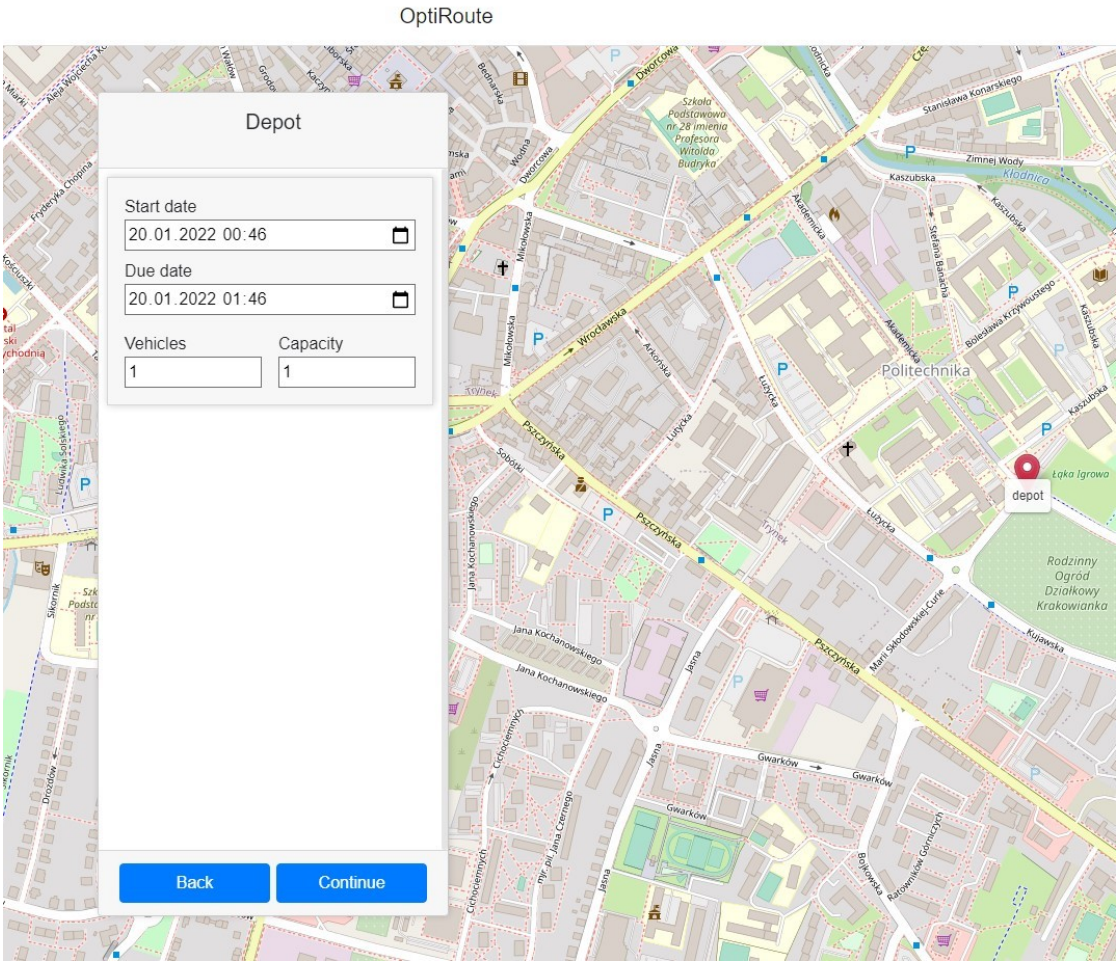

Figure 4.19: Usage example 2.1

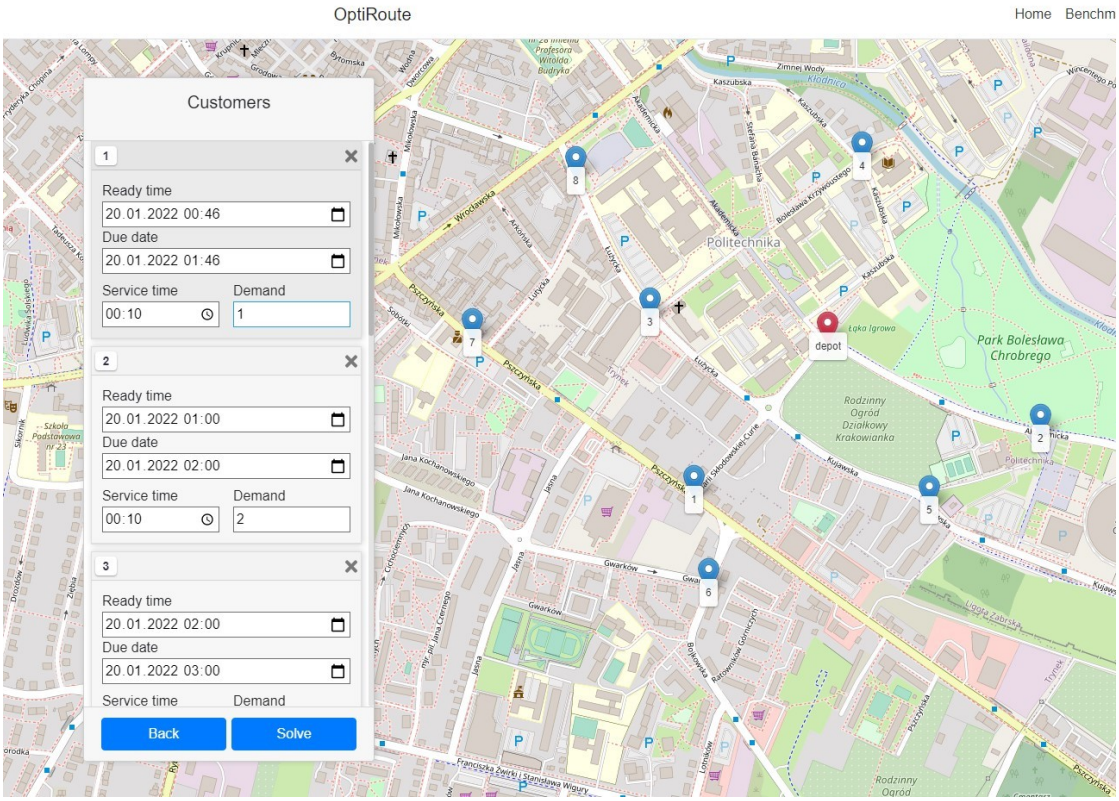

Figure 4.20: Usage example 1.2

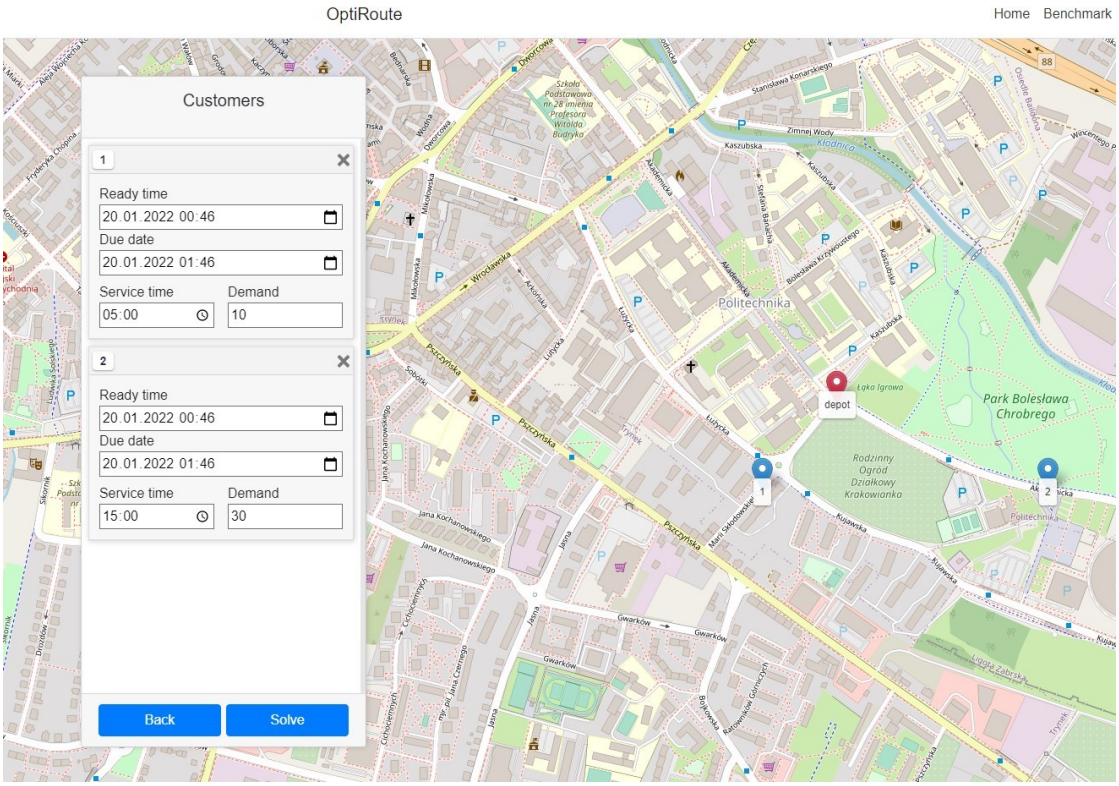

Figure 4.21: Usage example 2.2

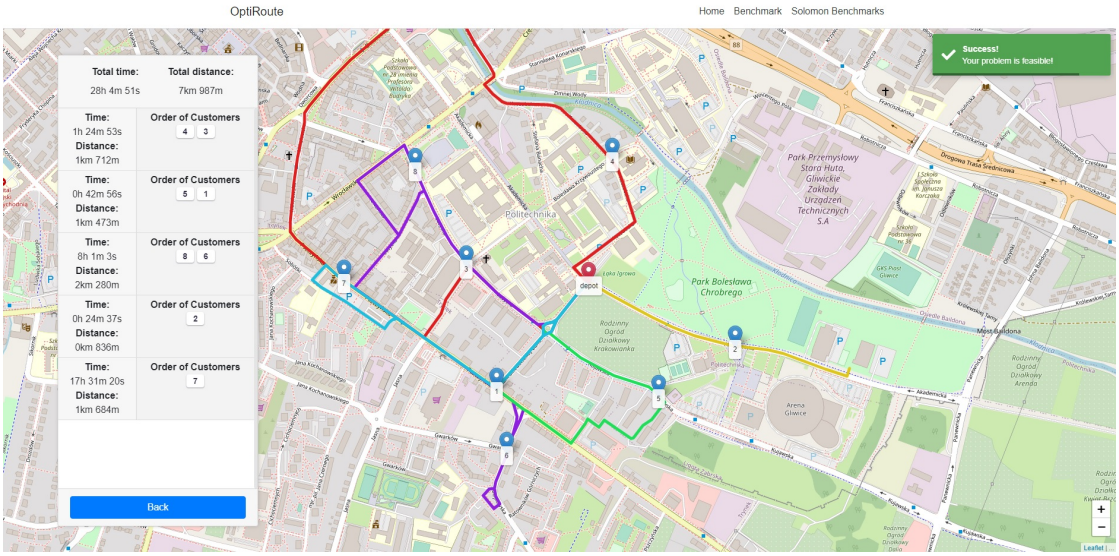

Figure 4.22: Usage example 1.3

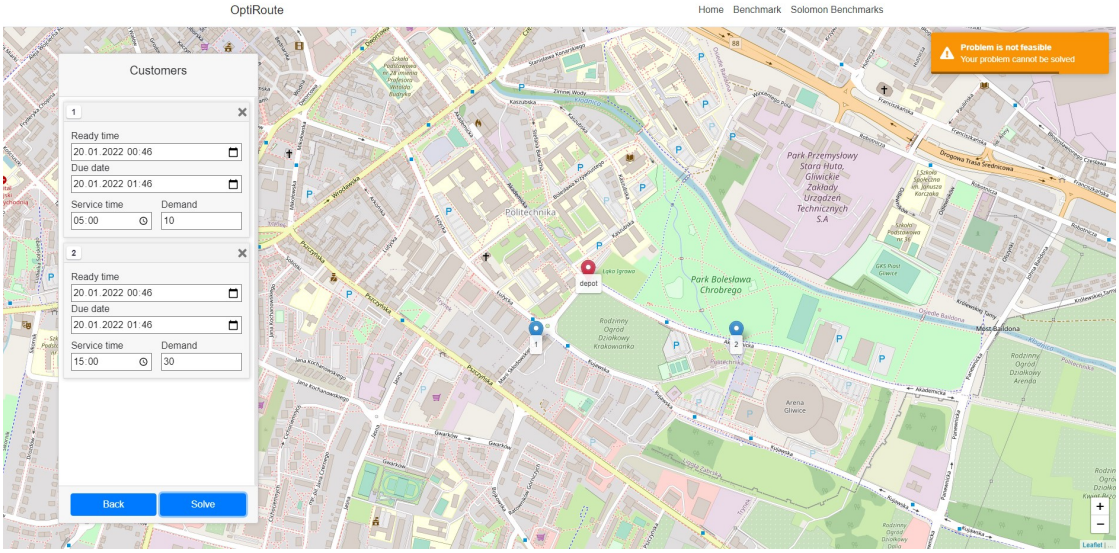

Figure 4.23: Usage example 2.3

C101

VEHICLE
NUMBER CAPACITY
25 200

CUSTOMER

CUST NO.	XCOORD.	YCOORD.	DEMAND	READY TIME	DUE DATE	SERVICE TIME
0	4000	5000	0	0	123600	0
1	4500	6800	10	91200	96700	9000
2	4500	7000	30	82500	87000	9000
3	4200	6600	10	6500	14600	9000
4	4200	6800	10	72700	78200	9000
5	4200	6500	10	1500	6700	9000

Figure 4.24: Usage example 3.1

C101

VEHICLE
NUMBER CAPACITY
25 5

CUSTOMER

CUST NO.	XCOORD.	YCOORD.	DEMAND	READY TIME	DUE DATE	SERVICE TIME
0	4000	5000	0	0	5000	0
1	4500	6800	10	91200	96700	9000
2	4500	7000	30	82500	87000	9000
3	4200	6600	10	6500	14600	9000
4	4200	6800	10	72700	78200	9000
5	4200	6500	10	1500	6700	9000

Figure 4.25: Usage example 4.1

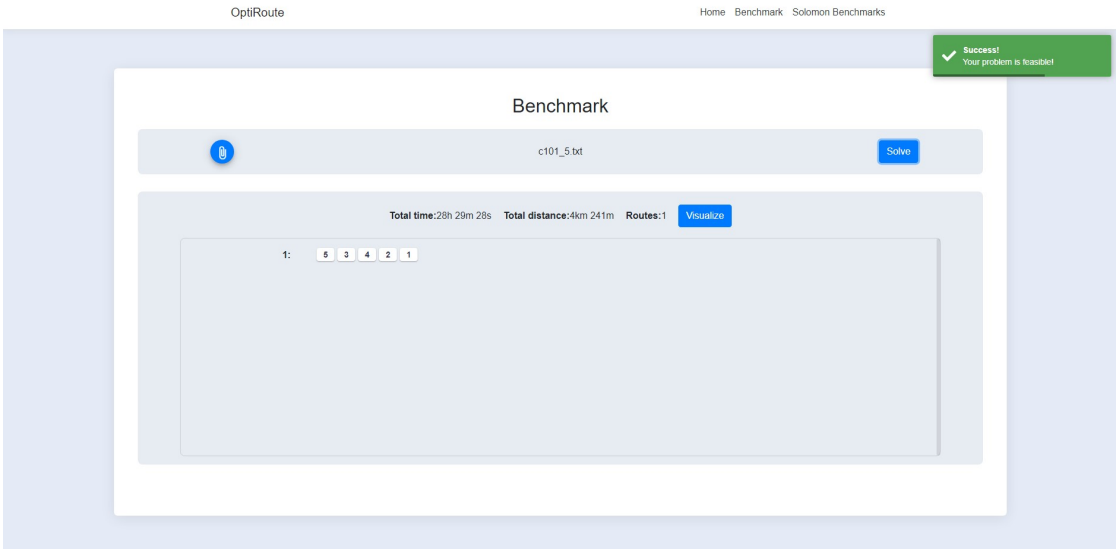

Figure 4.26: Usage example 3.2

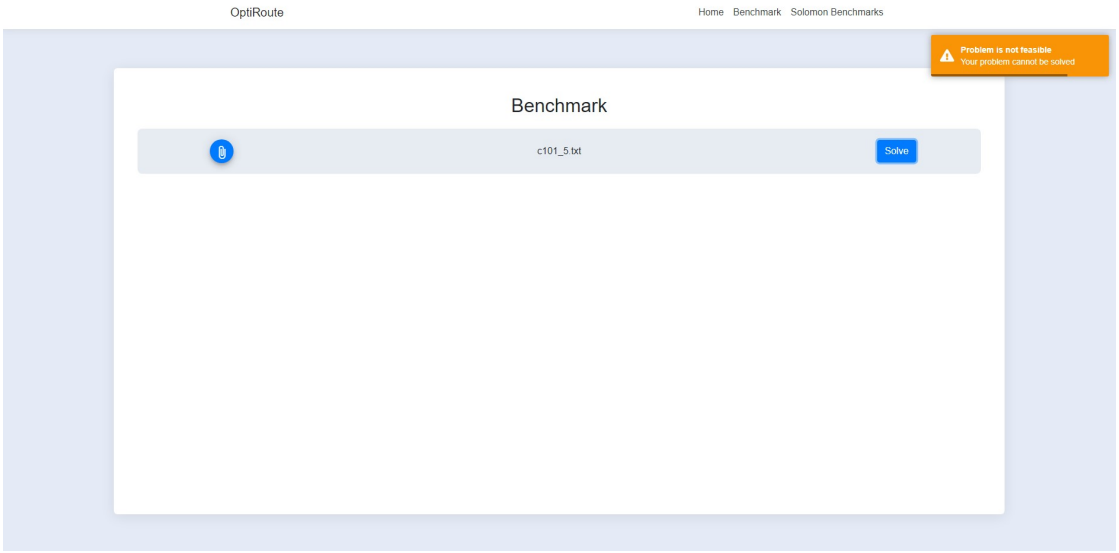

Figure 4.27: Usage example 4.2

Chapter 5

Internal specification

5.1 System architecture

The application implements a client-server architecture with containerised components. The overview of the architecture is presented in Fig. 5.1. Each component of the system is described in detail later in this chapter.

5.1.1 Design patterns

The system was implemented with the use of the three main design patterns: the clean architecture, the command and query responsibility segregation (CQRS) and the mediator pattern. All patterns were used in the backend part of the application.

Clean architecture

The backend server is designed according to the clean architecture pattern. The main goal of this architecture is to make the application logic and domain independent from any framework. This approach allows to change any external dependencies e.g. database, web framework or user interface (UI) while keeping the code of the project core intact. Thanks to the possibility of exchanging external dependencies, the application core can be easily reused in other systems. Moreover, such a system is easily testable, business rules can be tested without any external dependencies. The main benefits of this architecture are as follows [39]:

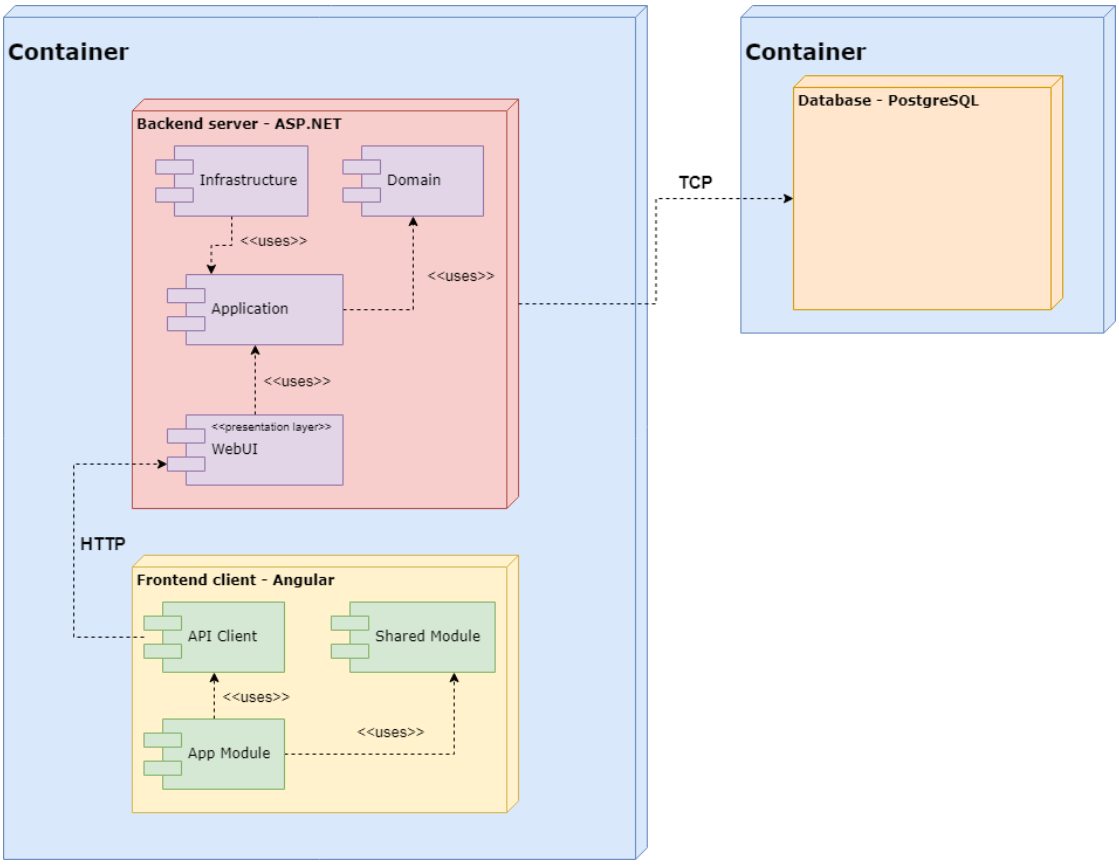

Figure 5.1: Architecture overview

- The system is independent of frameworks – the system can use libraries and frameworks as tools, rather than relying on them and adapting fully to them.

- Testable – the core of the application can be tested without the external dependencies.

- Independent of UI – the system may use, for example, a web-based or console UI, but any change to this can be made easily without changing the rest of the system code

- Independent of database – any database management system can be used, and changing such a system does not affect the core of the application

This architecture was chosen to design a system that is modern, scalable and easily expandable. The structure of the architecture is visible in Fig. 5.2.

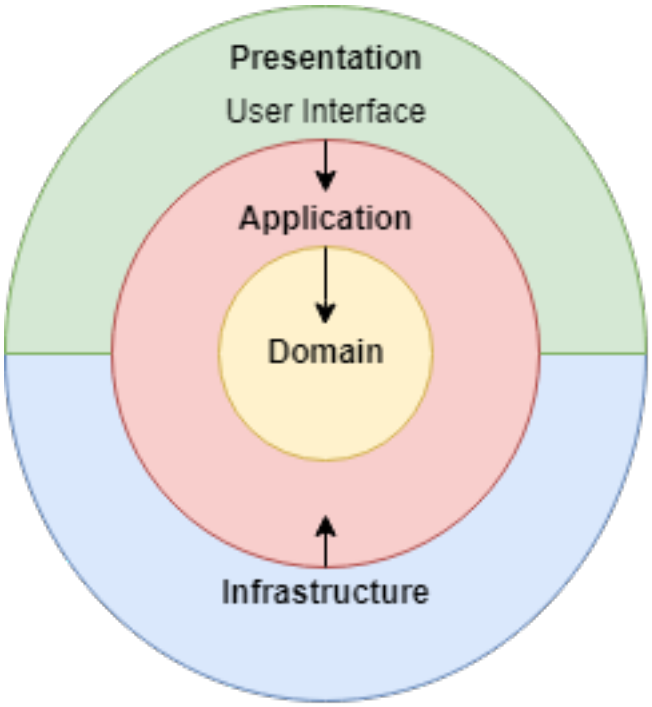

Figure 5.2: Clean architecture

CQRS

The command and query responsibility segregation pattern focuses mainly on the separation of data read and write operations. This pattern separates how data is read and written into two models:

- Queries – responsible for reading data.
- Commands – responsible for writing (updating) the data.

Furthermore, the pattern abstracts communication between layers and, using the mediator pattern, allows command/query parameters to be separated from their handlers. This approach simplifies the code as a result of reducing the amount of boilerplate code, but most importantly it creates an execution pipeline that allows error handling, logging or validation functionality to be injected [1].

The benefits of using CQRS pattern are as follows [1]:

- Scalability – read operations can be scaled independently of write operations
- Performance – separate reads and writes can be optimised independently of each other
- Security – separation ensures that the right class will perform the task
- Easy to maintain – when the number of available operations is large, each operation is stored in its own source file and is easy to find

Mediator

A mediator defines an object that encapsulates how a set of objects interact. Instead of communicating directly with each other, objects communicate through a special object called a mediator. The mediator centralises communication between objects, making them dependent only on the mediator, rather than on each object they wish to communicate with [13]. The main benefits of using the mediator design pattern include:

- Reduces the number of connections between classes

- Allows easier reuse of individual components
- The code is easier to maintain

The mediator pattern in combination with CQRS provides a set of powerful features and was therefore used in the project [28].

5.2 Backend server

5.2.1 Used libraries

Entity Framework

Entity Framework is an open-source object-relational mapping (ORM) framework supported by Microsoft. It allows the developer to work with POCO (Plain Old CLR Object) entities without focusing on the underlying database. Entity Framework takes care of creating database connections and executing commands. It also automatically parses the results of queries to the application entities. Another feature of this framework is the ability to track changes to an object and persist those changes to the database [41].

Entity Framework introduces the Context class, which implements the Unit Of Work and Repository patterns. The Context class represents a database session and allows various operations to be performed on it. It contains DbSet properties for each of the tables mapped to the application entities. DbSet is an implementation of the repository pattern [9].

A part of the ApplicationDbContext class used in the project is shown in Fig. 5.3.

MediatR

MediatR is an open-source library that is a simple implementation of the mediator pattern. The library allows in-process messaging without any dependencies. It separates requests from handlers, allowing the principle of single responsibility to be maintained in the code. MediatR also offers other functionality, such as sending notifications or injecting middlewares for requests called pipeline behaviours [21]. The main benefits from using the MediatR library are:

```
1         public class ApplicationDbContext : DbContext,
              IApplicationDbContext
2     {
3         private readonly IDateTime _dateTime;
4
5         public ApplicationDbContext(
6             DbContextOptions options,
7             IDateTime dateTime) : base(options)
8         {
9             _dateTime = dateTime;
10        }
11        public DbSet<BenchmarkResult> BenchmarkResults {
              get; set; }
12        public DbSet<BenchmarkInstance>
              BenchmarkInstances { get; set; }
13        public DbSet<Route> Routes { get; set; }
14        public DbSet<Customer> Customers { get; set; }
15        public DbSet<Depot> Depots { get; set; }
16        public DbSet<Solution> Solutions { get; set; }
```

Figure 5.3: Part of the ApplicationDbContext class.

- Less coupled code.
- Cleaner code because of the requests and handlers separation.
- Pipeline behaviours that allow data to be processed (e.g. logging) before or after a request is handled.

Example request and handler classes used in the project are shown in Fig. 5.4.

FluentValidation

FluentValidation is a library for building strongly typed validation rules. One of the most important features of fluent validation is the separation of validation logic from model and business logic. Each model that should be validated must have its own validator class with validation rules defined inside it [17].

The main advantages of using FluentValidation are:

- Easy to work with.

```
1   public class GetBestSolutionByBenchmarkResultIdQuery
        : IRequest<SolutionDto>
2   {
3       public int BenchmarkResultId { get; set; }
4   }
5
6   public class
        GetBestSolutionByBenchmarkResultIdQueryHandler :
        IRequestHandler<
        GetBestSolutionByBenchmarkResultIdQuery,
        SolutionDto>
7   {
8       private readonly IApplicationDbContext _context;
9       private readonly IMapper _mapper;
10
11      public
            GetBestSolutionByBenchmarkResultIdQueryHandler
            (IApplicationDbContext context, IMapper
            mapper)
12      {
13          _context = context;
14          _mapper = mapper;
15      }
16
17      public async Task<SolutionDto> Handle(
            GetBestSolutionByBenchmarkResultIdQuery
            request, CancellationToken cancellationToken)
18      {
19          var result = await _context.Solutions
20              .Include(x => x.Depot)
21              .Include(x => x.Routes)
22              .ThenInclude(x => x.Customers)
23              .Include(x => x.BestBenchmarkResult)
24              .FirstOrDefaultAsync(x => x.
                    BestBenchmarkResult.DbId == request.
                    BenchmarkResultId);
25
26          return _mapper.Map<SolutionDto>(result);
27      }
28  }
```

Figure 5.4: Example request and handler classes.

```
1  public class DepotDtoValidator : AbstractValidator<
       DepotDto>
2      {
3          public DepotDtoValidator()
4          {
5              CascadeMode = CascadeMode.StopOnFirstFailure
                   ;
6
7              RuleFor(v => v.Id)
8                  .NotNull()
9                  .GreaterThanOrEqualTo(0);
10
11             RuleFor(v => v.X)
12                 .NotNull()
13                 .GreaterThanOrEqualTo(0);
14
15             RuleFor(v => v.Y)
16                 .NotNull()
17                 .GreaterThanOrEqualTo(0);
18
19             RuleFor(v => v.DueDate)
20                 .NotEmpty();
21         }
22     }
```

Figure 5.5: Validator class for DepotDto type.

- Separation of validation rules and models.
- Clear syntax for validation rules.
- Speed of execution, which allows to increase the efficiency of the application.
- Easy to unit test.

Sample validator class is shown in Fig. 5.5

AutoMapper

AutoMapper is an object-to-object mapping library. The mapping is performed by transforming an input object of one type to an output object of another type.

```
1 profile.CreateMap<Route, RouteDto>()
2                 .ForMember(dest => dest.TotalTime, opt
                    => opt.MapFrom(src => src.Vehicle.
                    CurrentTime))
3                 .ForMember(dest => dest.TotalLoad, opt
                    => opt.MapFrom(src => src.Vehicle.
                    CurrentLoad))
4                 .ForMember(dest => dest.TotalDistance,
                    opt => opt.MapFrom(src => src.
                    TotalDistance));
```

Figure 5.6: Configuration of mapping from Route type to RouteDto type.

Mappings between types are fully configurable and allow the programmer to perform complex transformations between types. The library simplifies the object mapping process and separates the mapping configurations from the objects, which helps to maintain clean code. Furthermore, another advantage of the library is that it is fully testable [3]. An example mapping configuration is shown in Fig. 5.6.

NSwag

NSwag provides tools to generate OpenAPI specifications from within ASP.NET Web API controllers. Furthermore, NSwag can generate the TypeScript API client, reducing the amount of work required to integrate a frontend application with a backend server. The Swagger UI visual documentation (Fig. 5.7) is also generated by NSwag allowing the developer to test and interact with the API without manually writing implementation logic [19].

NSwag was used in the project to speed up the process of integrating the client application with the server and to provide a visual tool to interact and test the API in a convenient way.

NUnit

NUnit is an open-source unit testing framework for the .NET platform. The framework is easy to work with and provides various tools to create good and reliable unit tests for applications [22].

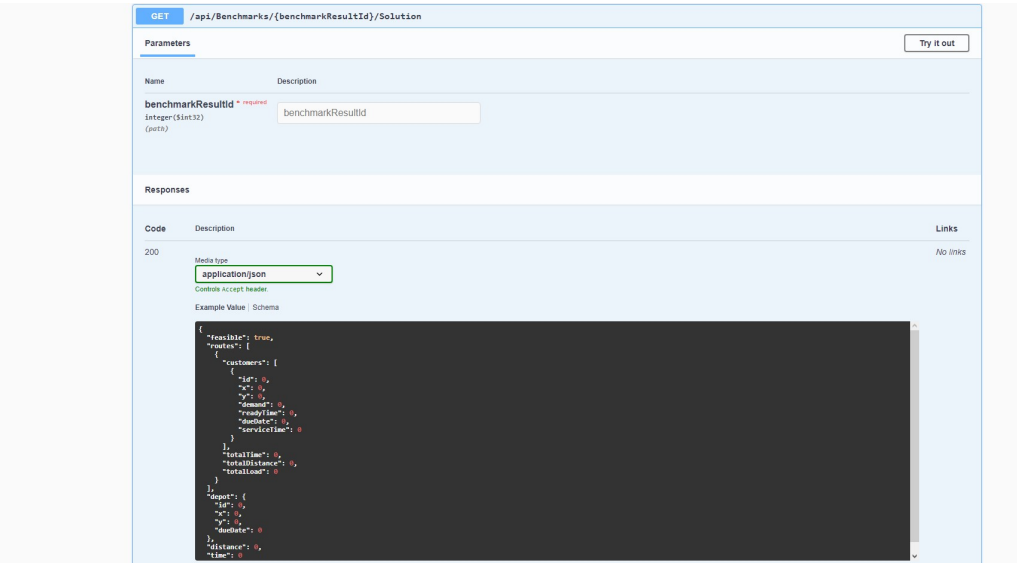

Figure 5.7: Swagger UI endpoint.

5.2.2 Domain

The project domain layer contains all the entities and their logic specific to the domain layer. The business domain of the application is modelled inside this layer. The term domain layer comes from the Domain Driven Design (DDD) approach to structuring software [8].

There are two main approaches when it comes to implementing domain entities:

- Anemic model – separates the logic from models. The logic is placed in separated classes (services, helpers etc.).
- Rich model – each model has its own behavioural logic encapsulated inside an entity. For example, the Cart entity will contain an AddProductToCart method inside the class.

The rich domain model approach was chosen for the project because it preserves one of the most important aspects of object-oriented programming (OOP), which is encapsulation. With a rich domain model, the code is cleaner and more secure (because sensitive data is only stored and modified inside the model) than with an anemic model. Furthermore, the domain is easier to maintain, and a rich model

prevents the application from creating invalid domain objects. The rich domain entities can be easily unit tested without the need to call any external services to apply business logic to them [2].

The domain project contains all the entities that are used in the application:

- BenchmarkInstance – contains information about the instance of the Solomon benchmark problem.
- BenchmarkResult – stores the BenchmarkInstance and the Solution generated by the application and the best known Solution .
- Customer – represents the customer in the CVRPTW. Includes methods for calculating the distance and duration of a travel to another Customer or Depot.
- Depot – represents the depot.
- Problem – defines the CVRPTW. It contains the details of the problem, the Depot, the Customer list and tables of distances and travel times between each location in the problem.
- Route – represents a route. It contains methods for i.a. adding or removing a Customer from the route, validating time and capacity limits (Fig. 5.8).
- Solution – the solution produced by the algorithm.
- Vehicle – represents the vehicle in the CVRPTW.

The project domain layer does not depend on any external library and can be reused in other projects as the core of the system.

5.2.3 Application

The application layer in the project depends only on the domain layer. It is responsible for the implementation of business rules for the application—in the case of this thesis it contains the algorithmic part of the system. Furthermore, besides the implementation of business rules, the application layer has a second

```
1  public class Route
2      {
3              public void AddCustomer(Customer customer)
4          {
5              if (Customers.Count > 0)
6              {
7                  CustomersDistance += this.Customers.Last
                       ().CalculateDistanceBetween(this.
                       Distances, customer);
8                  CustomersTime += this.Customers.Last().
                       CalculateTimeBetween(this.Durations,
                       customer);
9              }
10             this.Customers.Add(customer);
11
12             if (TimeFromDepot + CustomersTime +
                   WaitingTime < customer.ReadyTime)
13             {
14                 double waitingTime = customer.ReadyTime
                       - (TimeFromDepot + CustomersTime +
                       WaitingTime);
15                 this._waitingTime += waitingTime;
16                 this._waitingTimeDictionary.Add(customer
                       .Id, waitingTime);
17             }
18
19             CustomersTime += customer.ServiceTime;
20
21             this.Vehicle.CurrentLoad += customer.Demand;
22             this.Vehicle.CurrentTime = this.TotalTime;
23         }
24                 public bool IsFeasible()
25         {
26             if (CheckCapacityConstraints())
27                 return CheckTimeConstraints();
28
29             return false;
30         }
```

Figure 5.8: Part of the Route entity class.

responsibility, which is the abstraction of application concerns. This layer only contains the interfaces of external concerns, such as database or file system access, and the implementations of these interfaces are injected from the infrastructure layer.

Decoupling is one of the main advantages of a clean architecture. For example, when querying data, the application does not need to know where the data comes from, it can be a database or even a flat text file. This approach allows to focus only on business rules and provides the flexibility to easily change the data source. Furthermore, the application layer is testable and can be reused together with the domain layer in other projects [7].

The application project is structured as follows:

- **Benchmarks** – contains commands and queries connected with benchmarks. Each query and command folder contains request, handler, DTO and validator classes.
  - **Commands**
    - **SolveBenchmarkProblemCommand** – responsible for solving the problem from the given benchmark file.
  - **Queries**
    - **GetBenchmarkResultsQuery** – responsible for fetching the data from the BenchmarkResult table.
    - **GetBestSolutionByBenchmarkResultIdQuery** – returns the Solution object with the best known solution from the BenchmarkResult with the given identifier.
    - **GetSolutionByBenchmarkResultIdQuery** – returns the Solution object with the thesis solution from the BenchmarkResult with the given identifier.
- **Common** – collection of classes, methods and interfaces used across the application layer.
  - **Behaviours** – contains custom MediatR PipelineBehaviours used for triggering validators or exception handling.

  - **Enums** – enumerators used across the system.
  - **Exceptions** – custom exception classes used: NotFoundException and ValidationException .
  - **Extensions** – extension methods used to extend the functionality of a type.
  - **Interfaces** – interfaces used across the application layer.
  - **Mappings** – AutoMapper extensions and mapping configurations.
  - **Services** – algorithmic part of the application. It contains services responsible for solving the CVRPTW.
- **CVRPTW** – contains the command responsible for solving the CVRPTW.
  - **Commands**
    - **GetSolutionCommand** – contains request, handler, DTO and validator classes. Solves the CVRPTW defined in the request.

CVRPTW algorithm

The application defines 3 interfaces for the CVRPTW algorithm:

- ISolver (Fig. 5.9) – solves the CVRPTW by calling methods from the objects IMethod and IImprovement.
- IMethod (Fig. 5.10) – responsible for generating the initial solution.
- IImprovement (Fig. 5.11) – improves the initial solution.

The interfaces are implemented by the VRPTWSolver (ISolver), PFIHInitial (IMethod) and LocalSearchLambda (IImprovement) classes. The VRPTWSolver class implements only the methods from the interface. Inside each method the corresponding functions from IMethod or IImprovement objects are called.

PFIHInitial is an implementation of the Push Forward Insertion Heuristic algorithm. In addition to the methods implemented from the IMethod interface, this class introduces a set of private methods for constructing the solution according to the algorithm. The main PFIH Construct method is shown in Fig. 5.12.

```
1  public interface ISolver
2    {
3        IMethod Initial { get; set; }
4        IImprovement Improvement { get; set; }
5
6        Solution Solve(Problem problem);
7
8        Solution Create(Problem problem);
9
10       Solution Improve(Solution solution);
11   }
```

Figure 5.9: ISolver interface.

```
1  public interface IMethod
2    {
3        Solution Solve(Problem problem);
4    }
```

Figure 5.10: IMethod interface.

LocalSearchLambda is an implementation of local search with the λ interchange heuristic. It improves the initial solution by swapping customers between routes. Like PFIHInitial , this class also provides a set of private methods to improve the initial solution. The main LocalSearch function is presented in Fig. 5.13.

5.2.4 Infrastructure

The infrastructure layer implements the interfaces defined in the application layer. The infrastructure depends on external resources and libraries, e.g. Entity

```
1  public interface IImprovement
2    {
3        Solution Improve(Solution currentSolution);
4    }
```

Figure 5.11: IImprovement interface.

```
1         private void Construct(Problem problem, List<
              Customer> unroutedCustomers, List<Route>
              routes)
2         {
3             int counter = 0;
4             while (unroutedCustomers.Count > 0)
5             {
6                 Route currentRouteCustomerList = new
                      Route()
7                 {
8                     Id = counter,
9                     Vehicle = new Vehicle(counter,
                          problem.Capacity, 0, 0),
10                    Depot = problem.Depot,
11                    Distances = problem.Distances,
12                    Durations = problem.Durations
13                };
14                Customer seedCustomer = FindSeedCustomer
                      (unroutedCustomers);
15                currentRouteCustomerList.AddCustomer(
                      seedCustomer);
16                unroutedCustomers.Remove(seedCustomer);
17
18                InsertCustomers(unroutedCustomers,
                      currentRouteCustomerList);
19                routes.Add(currentRouteCustomerList);
20
21                counter++;
22            }
23        }
```

Figure 5.12: Construct method.

```
1         private double LocalSearch(List<Route> routeList
              )
2         {
3             double globalMaxDif = 0;
4             int globalFirstRouteIndex = 0;
5             int globalSecondRouteIndex = 0;
6             Route globalMinCostFirstRoute = null;
7             Route globalMinCostSecondRoute = null;
8
9             for (int i = 0; i < routeList.Count; i++)
10            {
11                Route firstRoute = routeList[i];
12                for (int j = i + 1; j < routeList.Count;
                      j++)
13                {
14                    Route secondRoute = routeList[j];
15                    Route localMinCostFirstRoute = null;
16                    Route localMinCostSecondRoute = null
                          ;
17                    double oldDistance = firstRoute.
                          TotalDistance + secondRoute.
                          TotalDistance;
18                    double newDistance = Interchange(
                          oldDistance, firstRoute,
                          secondRoute, ref
                          localMinCostFirstRoute, ref
                          localMinCostSecondRoute, Lambda);
19
20                    CheckNewDistance(ref globalMaxDif,
                          ref globalFirstRouteIndex, ref
                          globalSecondRouteIndex, ref
                          globalMinCostFirstRoute, ref
                          globalMinCostSecondRoute, i, j,
                          localMinCostFirstRoute,
                          localMinCostSecondRoute,
                          oldDistance, newDistance);
21                }
22            }
23            ChangeRoutes(routeList, globalMaxDif,
                  globalFirstRouteIndex,
                  globalSecondRouteIndex,
                  globalMinCostFirstRoute,
                  globalMinCostSecondRoute);
24
25            return Math.Round(routeList.Sum(x => x.
                  TotalDistance), 2);
26        }
```

Figure 5.13: LocalSearch method.

Framework is used to connect to the database. By separating the infrastructure implementation from the application layer, any of the external dependencies can be easily changed without changing the code in the application layer.

The infrastructure project contains implementations of file readers and file providers:

- BenchmarkBestFileReader – reads and processes the contents of the files with the best known solutions for Solomon benchmarks.
- BenchmarkInstanceFileReader – reads and parses the contents of the benchmark file into the Problem class.
- SolomonInstancesFileProviderService – used to access the file system to obtain Solomon benchmark files.

Database

The infrastructure layer uses Entity Framework to connect to and operate on the database. It also provides an implementation of ApplicationDbContext, separate classes for configuring Entity Framework and database tables, and a migrations folder containing SQL migration files generated by Entity Framework. Migrations, if required, are applied when a database connection is initiated, they ensure that the database schema is identical to that outlined in the code. The schema of the database is presented in Fig. 5.14.

5.2.5 Presentation

The presentation layer (WebUI) presents the data to the outside world. In the case of the thesis, this layer contains API controllers that are used to interact with the application layer of the system. The presentation project interprets the data entered by the user and calls the appropriate functionality in the application layer. The API endpoints exposed in the presentation layer as as follows:

- **POST api/Benchmarks**
  - sends SolveBenchmarkProblemCommand

- **GET api/Benchmarks**
  - sends GetBenchmarkResultsQuery
- **GET api/Benchmarks/benchmarkResultId/Solution**
  - sends GetSolutionByBenchmarkResultIdQuery
- **GET api/Benchmarks/benchmarkResultId/BestSolution**
  - sends GetBestSolutionByBenchmarkResultIdQuery
- **POST api/CVRPTW**
  - sends GetSolutionCommand

Each controller class is derived from the abstract class ApiControllerBase (Fig. 5.15), which contains an instance of the MediatR object ISender. In the controller, the HTTP request is converted into an appropriate command or query and sent via the ISender object to the application layer (Fig. 5.16).

The presentation layer also contains the ASP.NET classes Program and Startup for configuring and launching the application. The Startup class is responsible for injecting dependencies into the project. Dependencies are configured and injected by calling the appropriate methods in the Startup class. The application and infrastructure layers contain their own extension methods for configuring dependencies created and injected inside these layers, and these methods are called in the Startup class as part of the dependency injection configuration.

Inside the WebUI project are the source files of the frontend client application, so the backend server can serve static html, js and css files that result from compiling the frontend application, eliminating the need to host the frontend application on a separate server.

When any exception is thrown during request processing, the custom API filter ApiExceptionFilterAttribute class filters and handles exceptions. The filter handles exceptions by catching them and checking their types. When the exception comes from the application layer (e.g., ValidationException) it returns the corresponding HTTP response with the corresponding status code and an error

message. Otherwise, a response with an internal server error (status code 500) is returned.

5.3 Frontend client

5.3.1 Used libraries

Angular Material

Angular Material is a UI component library for Angular. The library contains a set of useful and reusable components for building responsive websites.

Bootstrap

Bootstrap is a free and open-source frontend development framework for creating responsive websites. Bootstrap contains a collection of CSS styles, UI components, layouts and JavaScript plugins.

Toastr

Toastr is a lightweight JavaScript library for displaying toast notifications. It allows to create simple and extendable pop-up notifications.

LeafletJS

LeafletJS is a lightweight, open-source JavaScript library for creating interactive maps. The library is designed to work efficiently on both desktop and mobile platforms. One of the main advantages of the LeafletJS library is that it can be extended with multiple plugins. The library is easy to use and has a well-documented API [25]. An example of creating a map using the library is shown in Fig. 5.17.

Leaflet Routing Machine

Leaflet Routing Machine is a library that extends LeafletJS map and adds routing to it. The library is easy to use and can be fully customised. It allows to draw a route from point A to point B on the map.

The library was used in the project to draw routes on the map using real roads so that the application can be used in real situations. The backend server returns a route, and Leaflet Routing Machine draws paths between clients on a map based on the route.

5.3.2 App module

The app module is the main module of the client application. The main configuration of the frontend application is done in this module. In addition, this module contains views and services, therefore it is responsible for the appearance and main functionality of the client.

The app module is structured as follows:

- **benchmark** – responsible for the "Benchmarks" page.
- **benchmark-results** – responsible for the "Solomon Benchmarks" page.
- **error-handler** – configures the error handling functionality which displays toast notifications.
- **interceptors** – HTTP requests interceptors (used for displaying the spinner while processing the request).
- **map** – responsible for displaying the map.
- **map-layout** – responsible for the "Home" page.
- **map-sidebar** – responsible for the sidebar on the "Home" page.
- **nav-menu** – responsible for the navigation bar.
- **services** – contains services used to transfer and process data.

5.3.3 Shared module

The shared module contains components, models and utility functions used throughout the application. It is designed to isolate those parts of the application that are reused in multiple places.

Components declared inside the shared module are, for example, route and marker details displayed in the sidebar on the home page.

Utility functions include custom date validators, pipes used to format distance and time values to a readable format, or functions used to get a random colour or remove an item from a list.

Models are custom interfaces that are only used inside the frontend application, for example to pass data between components using services. DTO and request objects are declared inside the API client.

A shared module has the following structure:

- **components** – contains the components reused across the application.
- **models** – collection of custom interfaces.
- **pipes** – utility functions in the form of custom Angular Pipes.
- **utils** – collection of utility functions.

5.3.4 API client

The client application integrates with the backend server using an API client generated by NSwag. NSwag creates a TypeScript file web-api-client.ts that contains the classes generated for each API controller.

The API client contains the interfaces (Fig. 5.18) for each controller and the classes that are implementations of those interfaces. Each controller class contains methods for sending requests to its endpoints.

DTO objects and requests are also generated by the NSwag tool. Each DTO has its own interface (Fig. 5.19) and a class implementing this interface. This class provides a set of useful methods to deserialize and serialize the object.

5.3.5 OSRM API

The frontend application is integrated with OSRM API. The API is used in the main functionality (home page) to get the distances and travel durations between selected locations on the map. As a result of the API call, two matrices are returned

(Fig. 5.20). The first matrix contains the distances between locations in meters and the second one holds the durations of travel in seconds [31].

The second call to the OSRM API is made by the Leaflet Routing Machine. When the problem is feasible, for each route from the solution returned from the backend server, the library calls the API to get the coordinates of the paths to draw on the map.

5.4 Application workflow

When the user defines the problem using the map and clicks the "Solve" button, the application will perform the following steps:

1. The problem data is collected into a single request object.
2. The client calls the OSRM API to obtain the duration and distance tables.
3. The complete request object (with distance and duration tables) is sent to the backend server.
4. The algorithm attempts to solve the problem and returns a solution object.
5. If the problem is feasible, Leaflet Routing Machine makes calls to the OSRM API to draw routes. Otherwise, an appropriate message is displayed.
6. Routes are drawn on the map and solution details are displayed in the sidebar.

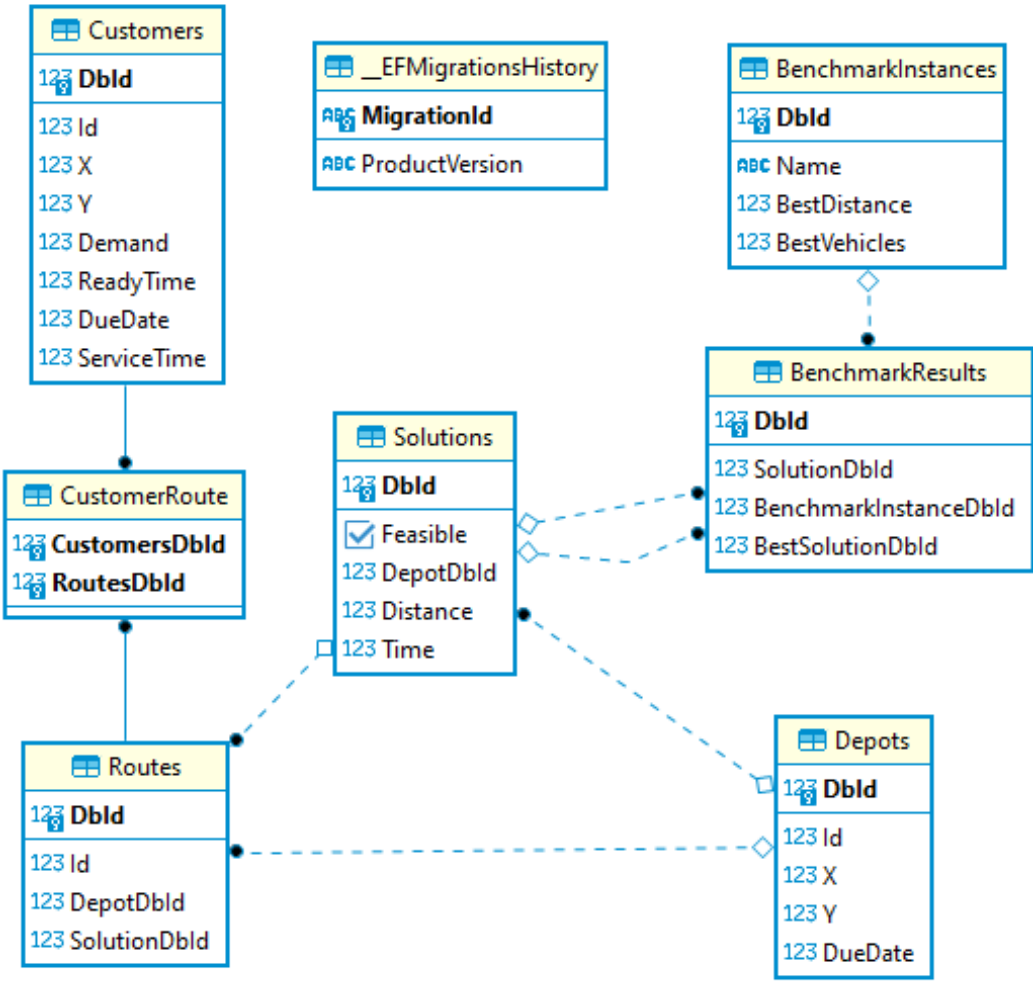

Figure 5.14: Database entity relationship diagram.

```
1   [ApiController]
2   [Route("api/[controller]")]
3   public abstract class ApiControllerBase :
        ControllerBase
4   {
5       protected ApiControllerBase()
6       {
7           CultureInfo newCulture = CultureInfo.
                CreateSpecificCulture("en-US");
8           Thread.CurrentThread.CurrentUICulture =
                newCulture;
9           Thread.CurrentThread.CurrentCulture =
                newCulture;
10      }
11
12      private ISender _mediator;
13
14      protected ISender Mediator => _mediator ??=
            HttpContext.RequestServices.GetService<
            ISender>();
15  }
```

Figure 5.15: The base controller.

```
1   public class CVRPTWController : ApiControllerBase
2   {
3       [HttpPost]
4       public async Task<ActionResult<SolutionDto>>
            GetSolution(ProblemDto problem)
5       {
6           return await Mediator.Send(new
                GetSolutionCommand() { Problem = problem
                });
7       }
8   }
```

Figure 5.16: The CVRPTWController.

```
var map = L.map('map').setView([51.505, -0.09], 13);

L.tileLayer('https://{s}.tile.openstreetmap.org/{z}/{x}/{y}.png', {
    attribution: '&copy; <a href="https://www.openstreetmap.org/copyright">OpenStreetMap</a> contributors'
}).addTo(map);
```

Figure 5.17: LeafletJS usage example.

```
export interface IBenchmarksClient {
    getSolution(file: FileParameter | null | undefined): Observable<SolutionDto>;
    getBenchmarkResults(): Observable<BenchmarkResultDto[]>;
    getSolutionByBenchmarkResultId(benchmarkResultId: number): Observable<SolutionDto>;
    getBestSolutionByBenchmarkResultId(benchmarkResultId: number): Observable<SolutionDto>;
}
```

Figure 5.18: Interface generated by NSwag for the BenchmarksController

```
export interface ISolutionDto {
    feasible?: boolean;
    routes?: RouteDto[] | undefined;
    depot?: DepotDto | undefined;
    distance?: number;
    time?: number;
}
```

Figure 5.19: Interface generated by NSwag for the SolutionDto

```
    "distances": [
        [
            0,
            818.3,
            416.5
        ],
        [
            411,
            0,
            827.6
        ],
        [
            416.5,
            1234.8,
            0
        ]
    ],
    "durations": [
        [
            0,
            110.7,
            37.3
        ],
        [
            40.1,
            0,
            77.4
        ],
        [
            37.3,
            148,
            0
        ]
    ]
```

Figure 5.20: Distances and durations tables returned by OSRM API.

Chapter 6

Verification and validation

6.1 Testing and validation

During the development process, different parts of the application were tested in different ways. The most difficult task was to test the algorithm and make sure that it gave correct solutions to the given problems.

6.1.1 Algorithm and backend

The correctness of the solutions was achieved by combining several testing and validation techniques.

Visualization tool

At the beginning of the application development process, when only the algorithmic part was under development, a simple visualisation tool was created. The purpose of this tool was to visually check that the resulting solutions were created in a meaningful and in some sense optimal way. The visualisation tool was necessary in the early stages of the application development because of the interpretability of the results generated by the algorithm. Detecting irregularities in the operation of the algorithm by reading the produced routes in text form is, for larger problems, an almost impossible task. The person validating the solution would have to remember, for example, the coordinates of 50 clients and the time

windows of each of them and check the routes considering each constraint of the problem. Thanks to the visualisation tool, the process of initial validation at that stage of development was significantly facilitated.

The tool was also useful after the heuristic part of the algorithm was completed. It was easy to observe the improvement in the solutions (Fig. 6.1).

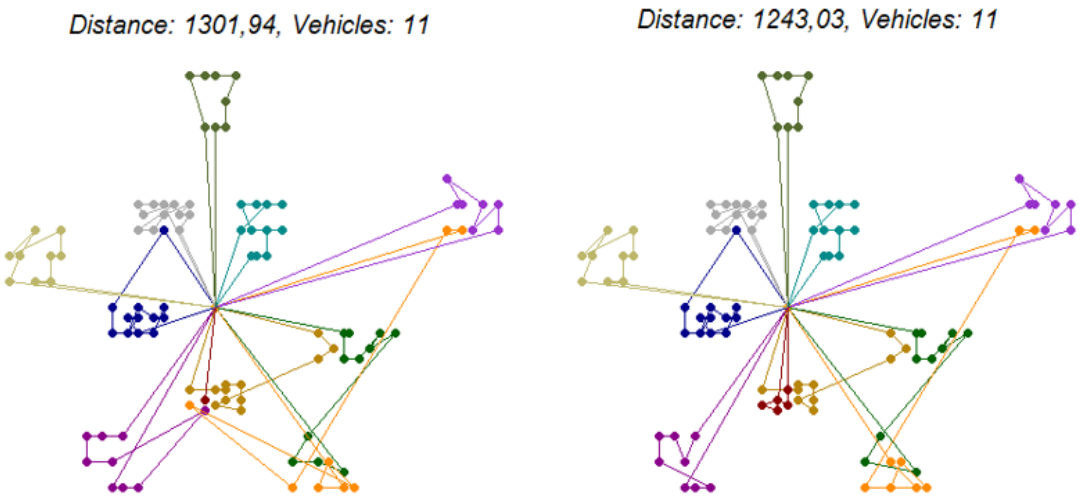

Figure 6.1: Initial and improved solutions visualized by the tool

Solomon Benchmarks

Solomon benchmark instances were used not only to evaluate the performance of the algorithm, but also to test and validate its operation. This was achieved by comparing the routes produced by the algorithm with the ones from the best known solutions. After comparison, it was observed that in most cases the routes were similar and differed by only a few customers. The difference in customers was reflected in the total distance and number of vehicles used in the solutions developed by the algorithm.

Unit tests

Unit tests played a key role in eliminating errors in the algorithmic part of the application. Each domain object has corresponding unit tests, and every functionality within each object is fully covered by unit tests. Unit tests have facilitated the software development process by allowing to check that all functionality works correctly after each change. Furthermore, during the development of the unit tests, it was discovered that the algorithm calculated the total time of each route without taking into account the waiting time before the time window. The error was quickly fixed, which would have been much more difficult and time-consuming without the help of unit tests.An example part of unit tests class for the Route object is shown in Fig. 6.2. An example run of domain unit tests is visible in Fig 6.3.

Manual tests

The algorithm and the entire backend part of the application were also tested manually. The API was tested using the Swagger UI (Fig. 6.4). During development, each endpoint was tested by sending various valid and invalid requests—by observing the responses it was possible to determine if the functionality was working correctly or not.

In a further stage of development, the algorithm was tested manually using a frontend client application. Using the frontend client, a number of problems were defined and solved, including a group of relatively small problems of low complexity, which made manual verification easier (small number of clients, vehicles and distances between points). This process made it possible to find and fix small errors and to prove the correctness of the algorithm. The summary times and distances for each route obtained by the algorithm were as expected. In addition, each API endpoint used in the frontend client was also manually tested along with the request data validation mechanisms.

6.1.2 Client

The specifics of creating a frontend application allows the developer to test functionality at every stage of its development, from the visual side to the logic

```
1     public class RouteTests
2 {
3                 [Test]
4         [TestCaseSource(typeof(EntityHelper), nameof(
              EntityHelper.PrepareCustomerCases))]
5         public void ShouldChangeLoadOfTheVehicle(List<
              Customer> customers)
6         {
7             Route route = EntityHelper.GetRouteCase(0,
                  5, 100);
8
9
10            foreach (var customer in customers)
11            {
12                customer.CalculateDepotTimesAndDistances
                      (route.Distances, route.Durations);
13                route.AddCustomer(customer);
14            }
15
16            route.Vehicle.CurrentLoad.Should().Be(
                  customers.Sum(x => x.Demand));
17        }
18
19        [Test]
20        [TestCaseSource(typeof(EntityHelper), nameof(
              EntityHelper.PrepareCustomerCases))]
21        public void ShouldNotChangeLoadOfTheVehicle(List
              <Customer> customers)
22        {
23            Route route = EntityHelper.GetRouteCase(0,
                  5, 100);
24
25
26            foreach (var customer in customers)
27            {
28                customer.CalculateDepotTimesAndDistances
                      (route.Distances, route.Durations);
29                route.AddCustomer(customer);
30                route.DeleteCustomer(customer);
31            }
32
33            route.Vehicle.CurrentLoad.Should().Be(0);
34        }
35 }
```

Figure 6.2: Part of the RouteTests class.

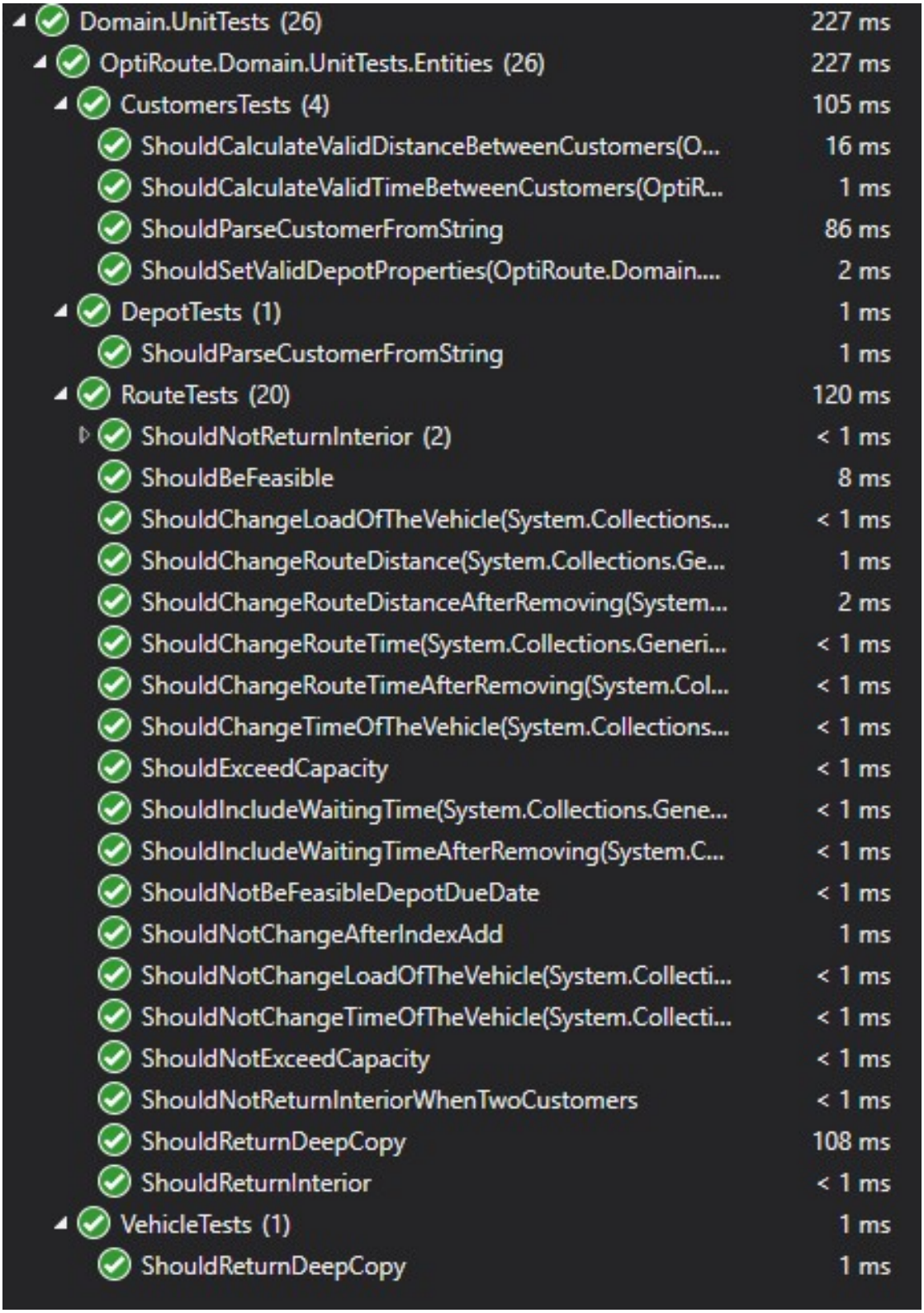

Figure 6.3: Domain unit tests

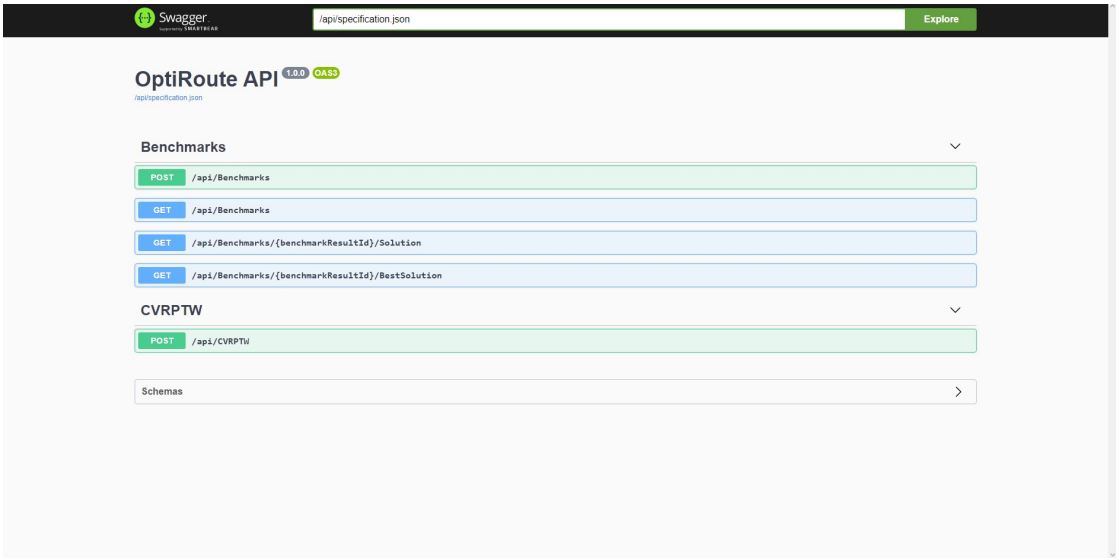

Figure 6.4: Swagger UI

of individual components. The client application was tested on different screen resolutions in order to achieve responsiveness. The request data is also validated on the client side of the application, so it was checked whether the information about the incorrect data is displayed correctly. In addition, toast pop-ups were tested to check the readability of messages and the correct implementation of the error handling mechanism.

6.2 Solomon benchmark results

In order to evaluate the performance of the algorithm, it was tested on all Solomon benchmark instances. A comparison of the best known solutions with the solutions obtained by the algorithm is shown in the Tables 6.1 and 6.2 [34].

Table 6.1: Solomon benchmark instances C101 - R111

	Thesis		Best	
Instance	Distance	Vehicles	Distance	Vehicles
C101	878.36	10	828.94	10
C102	1326.54	11	828.94	10
C103	1617.72	11	828.06	10
C104	1219.16	11	824.78	10
C105	953.79	10	828.94	10
C106	1103.6	10	828.94	10
C107	1127.49	10	828.94	10
C108	1191.96	10	828.94	10
C109	1380.5	10	828.94	10
C201	635.21	3	591.56	3
C202	1553.27	4	591.56	3
C203	1798.73	4	591.17	3
C204	1408.45	4	590.6	3
C205	903.72	4	588.88	3
C206	972.93	4	588.49	3
C207	1095.88	4	588.29	3
C208	974.34	3	588.32	3
R101	2181.6	22	1650.8	19
R102	2105.3	19	1486.12	17
R103	1851.02	17	1292.68	13
R104	1613.99	14	1007.31	9
R105	1931.59	17	1377.11	14
R106	1710.02	15	1252.03	12
R107	1825.93	16	1104.66	10
R108	1696.73	13	960.88	9
R109	1823.1	15	1194.73	11
R110	1864.64	14	1118.84	10
R111	1791.79	14	1096.72	10

Table 6.2: Solomon benchmark instances R112 - RC208

	Thesis		Best	
Instance	Distance	Vehicles	Distance	Vehicles
R112	1515.72	12	N/A	N/A
R201	2137.77	5	1252.37	4
R202	1950.89	5	1191.7	3
R203	1781.85	4	N/A	N/A
R204	1627.31	4	825.52	2
R205	2037.2	4	994.43	3
R206	1867.63	4	906.14	3
R207	1915.13	3	N/A	N/A
R208	1529.16	3	726.82	2
R209	1960.98	4	909.16	3
R210	1947.06	4	939.37	3
R211	1614.42	3	N/A	N/A
RC101	2192.4	17	1696.95	14
RC102	2281.54	17	1554.75	12
RC103	1975.79	15	1261.67	11
RC104	1772.02	13	1135.48	10
RC105	2210.6	18	1629.44	13
RC106	1969.33	15	1424.73	11
RC107	1828.77	14	N/A	N/A
RC108	1969.33	13	1139.82	10
RC201	2639.03	5	1406.94	4
RC202	2382.9	4	N/A	N/A
RC203	2098.92	4	N/A	N/A
RC204	1743.92	4	798.46	3
RC205	2395.17	5	1297.65	4
RC206	1886.68	4	1146.32	3
RC207	1942.84	4	1061.14	3
RC208	1772.11	3	828.14	3

Chapter 7

Conclusions

7.1 Achieved results

The primary objective of the work was to design and implement a tool that would simplify the process of defining and solving CVRPTW for the end user. This objective was successfully achieved and a modern and user-friendly web application was implemented. All functional and non-functional requirements that were identified in the early stages of development have been met. The implemented algorithm solves the VRP in a fast way and the results obtained can be considered satisfactory. The containerisation of the project makes it easy to run and deploy, which increases the usability of the application.

7.2 Further development

Although the application has met the requirements and expectations, there is room for improvement. One of the biggest areas with possibilities to extend the system is the VRP itself, there are many variations of the problem that take into account more constraints. In addition, the algorithmic part of the application could be extended to implement other heuristic and meta-heuristic methods that could potentially improve the performance of the algorithm.

The application could also be extended to include users and their management. Users would be able to save a problem, view the history of their previous problems

and solutions, or upload their own benchmark files and share them with other users. A mobile version of the application could also be a significant improvement. A portable version of the application would increase its usability.

There are still opportunities for further enhancements in the testing part of the application. Integration tests or end-to-end tests could be implemented to make the application more reliable and the development process even easier. The prospects for future development are broad and, thanks to the way the application has been designed, would not require rebuilding the existing code base and the system could be easily extended without much effort.

7.3 Encountered difficulties and problems

The most difficult problem encountered during the development process was adapting the LeafletJS and Leaflet Routing Machine libraries to the needs of the project. There is not much information on the Internet about redrawing multiple routes on the map or removing them. In the end, the desired effect was achieved and there was no need to change any of the libraries.

The process of containerising the application was very time consuming and required extensive research. The most difficult task was to create a correct docker file and to integrate the process of building both the server and client parts of the application.

The algorithmic part of the application was completed without any major problems, but the most difficult part of this stage of development was the correct calculation of time and distance when a client was added or removed from the route. However, thanks to unit testing, this functionality was completed in a reasonably short time.

Bibliography

[1] *A Brief Intro to Clean Architecture, Clean DDD, and CQRS*. https://blog.jacobsdata.com/2020/02/19/a-brief-intro-to-clean-architecture-clean-ddd-and-cqrs. [access date: 2022-01-27].

[2] *Anemic Domain Model vs Rich Domain Model – Co to*. https://www.bdabek.pl/anemic-domain-model-vs-rich-domain-model/. [access date: 2022-01-30].

[3] AutoMapper. *Getting Started Guide*. https://docs.automapper.org/en/stable/Getting-started.html. [access date: 2022-01-30].

[4] Kallehauge B. et al. "Vehicle Routing Problem with Time Windows." In: *Column Generation*. Boston, MA.: Springer, 2005, pp. 67–98.

[5] N. Christofides and S. Eilon. "An Algorithm for the Vehicle-dispatching Problem". In: *Journal of the Operational Research Society* 20.3 (1969), pp. 309–318.

[6] N. Christofides, A. Mingozzi and P. Toth. "Exact algorithms for the vehicle routing problem, based on spanning tree and shortest path relaxations". In: *Mathematical Programming* 20.1 (1981), pp. 255–282.

[7] *Clean Architecture - An Introduction*. https://www.dandoescode.com/blog/clean-architecture-an-introduction/. [access date: 2022-01-30].

[8] *Clean Architecture for ASP.NET Core Solution: A Case Study*. https://blog.ndepend.com/clean-architecture-for-asp-net-core-solution/. [access date: 2022-01-30].

[9] *Context Class in Entity Framework*. https://www.entityframeworktutorial.net/basics/context-class-in-entity-framework.aspx. [access date: 2022-01-30].

[10] J. F. Cordeau et al. "Vehicle Routing". In: *Management Science* 14 (2007), pp. 367–428.

[11] Microsoft Corporation. https://www.nuget.org. [access date: 2022-01-22].

[12] Microsoft Corporation. https://www.typescriptlang.org. [access date: 2022-01-22].

[13] *CQRS & Mediator in .NET Core — "A piece of cake"*. https://letienthanh0212.medium.com/cqrs-and-mediator-in-net-core-project-c0b477eab6e9. [access date: 2022-01-27].

[14] G. B. Dantzig and J. H. Ramser. "The Truck Dispatching Problem". In: *Management Science* 6.1 (1959), pp. 80–91.

[15] M. Desrochers, J. Desrosiers and M. Solomon. "A New Optimization Algorithm for the Vehicle Routing Problem with Time Windows". In: *Operations Research* 40.2 (1992), pp. 342–354.

[16] Inc. Docker. *Docker overview*. https://docs.docker.com/get-started/overview/. [access date: 2022-01-22].

[17] *Fluent Validation And Its Advantages*. https://www.c-sharpcorner.com/blogs/fluentvalidation-its-advantages. [access date: 2022-01-30].

[18] .NET Foundation. *.NET Foundation Survey - .NET Usage*. https://www.surveymonkey.com/stories/SM-GDSVMB2C/. [access date: 2022-01-22].

[19] *Generate TypeScript and C# clients with NSwag based on an API*. https://blog.logrocket.com/generate-typescript-csharp-clients-nswag-api/. [access date: 2022-01-30].

[20] Geir Hasle, Knut-Andreas Lie and Ewald Quak. *Geometric Modelling, Numerical Simulation, and Optimization: Applied Mathematics at SINTEF*. Berlin: Springer, 2007. ISBN: 978-3-540-68783-2.

[21] *How To Write Decoupled Code with MediatR: The Mediator Pattern*. https://hackernoon.com/how-to-write-decoupled-code-with-mediatr-the-mediator-pattern-u81231b0. [access date: 2022-01-30].

[22] *Introduction To NUnit Testing Framework*. https://www.c-sharpcorner.com/article/introduction-to-nunit-testing-framework/. [access date: 2022-01-30].

[23] G. Laporte, H. Mercure and Y. Nobert. "An exact algorithm for the asymmetrical capacitated vehicle routing problem". In: *Networks* 16.1 (1986), pp. 33–46.

[24] G. Laporte and Y. Nobert. "Exact Algorithms for the Vehicle Routing Problem." In: *Surveys in Combinatorial Optimization*. Ed. by S. Martello et al. Vol. 132. North-Holland Mathematics Studies, 1987, pp. 147–184.

[25] LeafletJS. https://leafletjs.com. [access date: 2022-01-30].

[26] Google LLC. *Managing data*. https://angular.io/start/start-data. [access date: 2022-01-22].

[27] Google LLC. *V8 JavaScript Engine*. https://v8.dev/docs. [access date: 2022-01-22].

[28] *Mediator Design Pattern*. https://sourcemaking.com/design_patterns/mediator. [access date: 2022-01-27].

[29] Prosus N.V. *Stack Overflow Developer Survey 2021*. https://insights.stackoverflow.com/survey/2021. [access date: 2022-01-22].

[30] I. H. Osman and N. Christofides. "Simulated annealing and descent algorithms for capacitated clustering problem". In: *Imperial College, University of London, Research Report* (1989).

[31] *OSRM API Documentation*. http://project-osrm.org/docs/v5.24.0/api. [access date: 2022-01-30].

[32] M. W. P. Savelsbergh. "Local search in routing problems with time windows". In: *Annals of Operations Research* 4 (1985), pp. 285–305.

[33] SINTEF. https://www.sintef.no/projectweb/top/vrptw/documentation2/. [access date: 2022-01-24].

[34] SINTEF. `https://www.sintef.no/projectweb/top/vrptw/100-customers/`. [access date: 2022-01-24].

[35] Marius M. Solomon. "Algorithms for the Vehicle Routing and Scheduling Problems with Time Window Constraints". In: *Operations Research* 35.2 (1987), pp. 254–265.

[36] Katarzyna Sukiennik. "Logistics Costs in Manufacturing Companies". In: *Zeszyty Naukowe Politechniki Częstochowskiej. Zarządzanie* 4 (2011), pp. 131–139.

[37] V. Tam and K.T. Ma. "An Effective Search Framework Combining Meta-Heuristics to Solve the Vehicle Routing Problems with Time Windows". In: *Vehicle Routing Problem*. Ed. by Tonci Caric and Hrvoje Gold. Rijeka: IntechOpen, 2008. Chap. 3.

[38] K. Tan et al. "Heuristic methods for vehicle routing problem with time windows." In: *AI in Engineering* 15 (Jan. 2001), pp. 281–295.

[39] *The Clean Architecture*. `https://blog.cleancoder.com/uncle-bob/2012/08/13/the-clean-architecture.html`. [access date: 2022-01-27].

[40] Paolo Toth and Daniele Vigo. *The Vehicle Routing Problem*. Society for Industrial & Applied Mathematics, 2002. ISBN: 978-0-89871-498-2.

[41] *What is Entity Framework?* `https://www.entityframeworktutorial.net/what-is-entityframework.aspx`. [access date: 2022-01-30].

Appendices

Index of abbreviations and symbols

VRP	Vehicle Routing Problem
VRPTW	Vehicle Routing Problem with Time Windows
CVRPTW	Capacitated Vehicle Routing Problem with Time Windows
V	Set of vertices representing customers and the depot
A	Arcs connecting vertices from V
W	Set of vehicles
Q	Capacity of vehicle
HTTP	Hypertext Transfer Protocol
DTO	Data Transfer Object
API	Application Programming Interface
SQL	Structured Query Language
TCP	Transmission Control Protocol
OSRM	Open Source Routing Machine
CQRS	Command Query Responsibility Segregation
UI	User Interface

POCO	Plain Old CLR Object
DDD	Domain Driven Design
OOP	Object-Oriented Programming
CSS	Cascading Style Sheets

List of additional files in electronic submission

Additional files uploaded to the system include:

- source code of the application
- test benchmark files
- a video showing how software developed for thesis is used

List of Figures

List of Tables