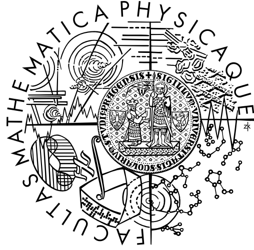

FACULTY
OF MATHEMATICS
AND PHYSICS
Charles University

BACHELOR THESIS

Dmitry Zhukov

Web application for keyword-aware walking route search

Department of Software Engineering

Supervisor of the bachelor thesis: doc. Mgr. Martin Nečaský, Ph.D.
Study programme: Computer Science
Study branch: Programming and Software Development

Prague 2024

I declare that I carried out this bachelor thesis independently, and only with the cited sources, literature and other professional sources. It has not been used to obtain another or the same degree.

I understand that my work relates to the rights and obligations under the Act No. 121/2000 Sb., the Copyright Act, as amended, in particular the fact that the Charles University has the right to conclude a license agreement on the use of this work as a school work pursuant to Section 60 subsection 1 of the Copyright Act.

In .... date ....

Author's signature

I am sincerely grateful to doc. Mgr. Martin Nečaský, Ph.D. for the time spent guiding me through this thesis, all the great advice, and many fruitful discussions that led to a better result.

Title: Web application for keyword-aware walking route search

Author: Dmitry Zhukov

Department: Department of Software Engineering

Supervisor: doc. Mgr. Martin Nečaský, Ph.D., Department of Software Engineering

Abstract: Most mainstream web mapping applications implement location-based direction search. The typical workflow involves constructing an explicit sequence of places to visit. In this thesis, we aim to develop a web application that lets users formulate search queries in terms of categories, each composed of a keyword and attribute filters. A resulting route passes through at least one place from each category. The search procedure is formalized as a variant of the generalized Traveling Salesman Problem and solved with the help of polynomial-time heuristics.

The application follows the three-tier architecture pattern. The frontend is implemented as a single-page application written in TypeScript using the React library, while the backend is programmed using the ASP.NET framework. We utilize the OpenStreetMap dataset and two knowledge graphs, Wikidata and DBPedia, as the basis for the conceptual model. Data is preprocessed and stored in MongoDB, which also serves as an efficient index. The OSRM routing engine helps calculate shortest paths and estimate network distances.

Last but not least, the application stores user data in a decentralized way, either in IndexedDB or a Solid pod. The former is a standardized in-browser database, while the latter is part of an emerging technology that gives users control over the physical location of their data and access rights.

Keywords: spatial queries route search personal data Solid open data

Název práce: Webová aplikace pro vyhledávání pěších tras s ohledem na klíčová slova

Autor: Dmitry Zhukov

Katedra: Katedra softwarového inženýrství

Vedoucí bakalářské práce: doc. Mgr. Martin Nečaský, Ph.D., Katedra softwarového inženýrství

Abstrakt: Většina mainstreamových webových mapových aplikací nabízí vyhledávání tras založené na poloze. Uživatel zadává konkrétní místa a určuje jejich pořadí. Na základě těchto vstupů systém naplánuje cestu. V předložené práci se věnujeme vývoji webové aplikace, která umožní uživatelům formulovat vyhledávací dotazy pomocí kategorií, z nichž každá se skládá z klíčového slova a atributových filtrů. Nalezená cesta nutně prochází alespoň jedním místem z každé kategorie. Vyhledávací procedura je pak formalizována jako varianta zobecněného problému obchodního cestujícího a je řešena pomocí několika heuristik s polynomiální časovou složitostí.

Aplikace využívá třívrstvou architekturu. Frontend je implementován jako jednostránková webová aplikace psaná v jazyce TypeScript s použitím knihovny React. Backend je navržen za pomocí ASP.NET frameworku. Používáme datovou sadu OpenStreetMap a dva znalostní grafy, konkrétně Wikidata a DBPedia, jako podklad pro konceptuální model. Data jsou předzpracována a uložena do databáze MongoDB, která zároveň slouží pro efektivní dotazování. OSRM routovací služba pomáhá s výpočtem nejkratších cest a odhadem vzdáleností.

Aplikace ukládá uživatelská data decentralizovaným způsobem, a to buď do IndexedDB nebo Solid podu. První možnost představuje databázi integrovanou do webového prohlížeče, zatímco ta druhá je součástí nově vznikající technologie, která svým uživatelům poskytuje úplnou kontrolu nad fyzickým umístěním jejich dat a přístupovými právy.

Klíčová slova: prostorové dotazy vyhledávání tras osobní data Solid otevřená data

Contents

Introduction **3**

1 Concepts **6**
1.1 Linked Data 6
1.2 Reclaiming data 8
1.3 Solid Project 9

2 Analysis **10**
2.1 Definitions 10
2.2 Requirements 11
2.3 User stories 15
2.4 Roles 16
2.5 Use cases 16
2.6 Data sources 19
2.7 Conceptual model 22
2.8 Existing solutions 24

3 Design **30**
3.1 User interface 30
3.2 Architecture 39
3.3 Data preparation 44
3.4 Routing algorithms 46

4 Implementation **54**
4.1 Prerequisites 54
4.2 Single-page application 54
4.3 Web API application 60
4.4 Querying with MongoDB 63
4.5 Data pipelines 64

5 Testing **65**
5.1 Automated testing 65
5.2 Performance testing 67
5.3 Usability testing 72

Conclusion **74**

Bibliography **75**

List of Tables **80**

List of Figures **81**

List of Abbreviations **83**

A Attachments **84**
A.1 Documentation 84
A.2 Prerequisites 84
A.3 Use cases 87
A.4 Results of usability testing 91
A.5 Electronic attachment 92

Introduction

Maps have undoubtedly played an important role in human history, helping to satisfy our innate urge to explore and navigate the surrounding world. Cartography, the science of mapmaking, has always evolved simultaneously with the progress in other fields of knowledge, gradually enhancing its methods. The invention of the World Wide Web has revolutionized numerous areas of our life. In particular, many web mapping platforms emerged at that time, pushing forward the frontiers of cartography.

Plewe [1] described four generations of web maps that appeared on the market up to 2007, ranging from simple static pictures to more advanced ones with dynamic elements. He pointed out that not only had they become available to the public, but users had also got a tool for creating and sharing content. Tsou [2] has speculated that these services would grow into more user-centered products with ubiquitous access via mobile devices, crowdsourced by amateurs and part-timers. A decade later, geocoding, satellite-based navigation, and real-time traffic information, to name a few, are all examples of advanced services that we use on a daily basis.

Let us consider one specific task that the majority of active travelers have tackled at least once. Suppose a person has just arrived at the train station of an unfamiliar city. Due to a late check-in, they might decide to visit several places of different kinds (gallery, museum, etc.) on the way to the hotel. There are no other limitations on where the points of interest reside, provided that the total distance is not too large. While initial and terminal locations are known upfront, extra effort needs to be made to define waypoints. A typical user interaction with mainstream applications would involve an iterative process of building a route that includes the following steps applied for each waypoint:

1. Searching a set of places that might satisfy imposed constraints.
2. Appending one of them to the sequence, with possible manual reordering.
3. New route is recalculated and presented to the user right after any change is introduced into the sequence configuration.

The main advantage of this procedure is the ability of the user to decide which points will appear on the route. We may also observe that once a place is added to the sequence, it loses the connection to the search context. The place is then treated as a simple point on the map. This leads to two significant drawbacks. Firstly, the found routes become increasingly difficult to revisit and reason about as time passes. The far more important problem is that all three steps must be repeated for each waypoint, leading to a poor user experience.

Another issue we would like to address is related to user data management. Modern web applications often function as centralized authorities, allowing third parties to access their functionality through personal profiles. Typically, a user agreement specifies that the service provider is responsible for storing and processing data, transferring genuine data ownership from the signee.

Below, we give a non-exhaustive list of unpleasant situations that a subscriber of such a service might encounter in practice.

- The service provider has decided to sell or grant access to user data to other companies for profit.
- The user wants to access the same data from different applications or transfer data to another provider offering better conditions.
- The service provider has permanently shut down servers without notifying the client base.
- The user has accomplished all their goals and requested the deletion of their profile along with the data. Instead, the service provider has deactivated it until the user returns.

The goals of the thesis

The main goal of the thesis is to design, develop and test a web application that attempts to deal with both concerns. The final solution shall incorporate the following subgoals.

G1 Propose an alternative view on geographical data and devise a search procedure based on exact categorical matching that reduces user input while keeping search context and results together.

G2 Manage personal data in a decentralized manner by decoupling them from the application, allowing users to have full control over their physical location and access rights.

G3 Provide a user experience similar to other mainstream applications.

Several aspects of the system are intentionally simplified as proper implementation would introduce non-trivial complexity with little gain for the thesis.

- Support for various commuting profiles (walking, driving, public transport, hybrid, etc.) is limited to walking mode only.
- The system implements neither direct nor reverse geocoding.

Document overview

The thesis is divided into *five* chapters, evolving from an in-depth understanding of the domain to the application of testing techniques.

Chapter **1** focuses on standards and principles for data organization and publishing on the modern Web. After that, we provide a brief overview of Solid technology, which stores user data in an external pod.

Chapter **2** analyzes the system requirements that the application should comply with and the use cases it should support. Next, we explore available sources of geodata and propose a conceptual model. In the final part, we compare existing applications with similar capabilities based on eight criteria.

Chapter **3** discusses the user interface, architecture, the technology stack used during implementation, and how the selected data sources should be queried. Last

but not least, the route search procedure is formalized, and efficient heuristics are selected.

Chapter **4** outlines notable implementation details, libraries, limitations, and potential pitfalls we addressed to achieve the desired functionality.

Chapter **5** explains the application of testing techniques, including automated, performance, and usability tests, for improving the quality of the application, with the results presented in the form of graphs and tables.

In addition, the Administrator's guide in Attachment **A.1** describes a step-by-step procedure for preparing a dataset and running the application on a personal computer.

1. Concepts

The purpose of this chapter is to clarify the conceptual difference between our and other mainstream web applications. We start with standards and principles related to data organization and publishing on the modern Web. Then, different kinds of decentralized storage are covered concerning operational risks. Finally, we give an overview of Solid technology — our choice for storing personal data.

1.1 Linked Data

The World Wide Web has gradually evolved into a dynamic and heterogeneous environment for social interactions, collaboration, and innovation. Consequently, the amount of available data has increased tremendously. It is safe to assume that no human, without the help of a machine, can organize them and uncover their hidden potential.

Suppose a user has published a photo without any attached metadata. Perhaps their friends and relatives would be able to understand the context of the picture and its value, but the same task does not appear so easy for an intelligent agent. The *Semantic Web* is an extension of the standard "Web of documents" to give data additional dimension and enable machines to navigate resources, read and reason about them. This term was introduced by Tim Berners-Lee, the creator of the Web, long before the exponential data growth became an issue.

At first, this task may seem very ambitious, yet achievable if communicated through standards and recommendations, allowing other participants to contribute. As one might expect, the topic is indeed vast. Therefore, we discuss only a subset relevant to this thesis. So, how can we make the Web machine-readable?

The initial step in the right direction is to provide a tool for describing data. The Resource Description Framework (RDF) [3] is an abstract syntax for representing information on the Web. It is well-suited for modeling structures similar to directed, labeled multigraphs. Every edge of a graph is a *statement* — a triple consisting of a subject, predicate, and object — applicable in a given context. An RDF document is a collection of RDF graphs.

For example, the sentence *"Alice knows Bob."* can be expressed by the statement illustrated in Figure 1.1, provided that additional information about entities is available upon dereferencing the corresponding links.

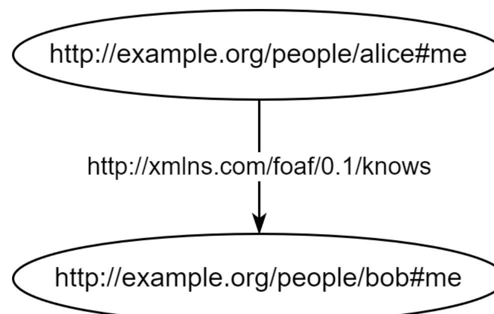

Figure 1.1: An RDF graph with one triple.

There are multiple options to choose from while serializing RDF data. Let us demonstrate two very different examples.

The most obvious ones are N-Triples [4] and N-Quads [5], that is to write each triple (optionally with a graph name) on a separate row. Reading one triple at a time is especially beneficial in case an RDF document is large enough so that it does not fit into the main memory. An example of an RDF statement serialized as N-Triple follows.

```
<http://example.org/people/alice#me>
↪  <http://xmlns.com/foaf/0.1/knows>
↪  <http://example.org/people/bob#me> .
```

A JSON-based Serialization for Linked Data (JSON-LD) [6] is another option based on a well-known and widely used JavaScript Object Notation (JSON) data format. Triples are contained within the `@graph` property. The `@context` defines a mapping between properties of a JSON object and their corresponding links.

Our running example serialized into JSON-LD is shown in the code snippet below, but it is important to note that this serialization is much more versatile.

```
{
  "@context": {
    "knows": {
      "@id": "http://xmlns.com/foaf/0.1/knows",
      "@type": "@id"
    }
  },
  "@graph": [
    {
      "@id": "http://example.org/people/alice#me",
      "knows": "http://example.org/people/bob#me"
    }
  ]
}
```

We have learned how to model data and represent them in a machine-readable form. The SPARQL Protocol and RDF Query Language (SPARQL) [7] enables searching and modifying RDF data via *basic graph pattern matching*, *property paths* and other advanced techniques.

The following query searches for all the people that Alice knows. If executed on our running example, the result would be a table with the only cell containing a link to Bob's web page.

```
PREFIX foaf: <http://xmlns.com/foaf/0.1/>

SELECT * WHERE {
  <http://example.org/people/alice#me> foaf:knows ?o .
}
```

Finally, we state four slightly adapted principles, so-called *Linked Data*, proposed by Tim Berners-Lee [8]. If properly followed, there is a chance that the Web will evolve into a standardized global-scale database.

1. Use Internationalized Resource Identifier (IRI) as names for things.
2. Use HTTP(S) IRIs so that people can look up those names.
3. When someone looks up an IRI, provide useful information in RDF.
4. Include links to other IRIs, so that they can discover more things.

1.2 Reclaiming data

Earlier, we mentioned operational risks that arise from centralized data management. The common reason behind all of them is that data are in the possession of a service provider. Let us discuss three very different kinds of *decentralized storage* — those not belonging to a service provider — applicable in web development that could help to mitigate or even avoid these risks altogether.

Device storage

The simplest approach is to use the file system of the device used to access a web page. The latest version of the application is fetched on page refresh. Then, it behaves more like being installed on a desktop computer.

Such storage provides control over where data are stored and ensures privacy, but it makes collaboration hard. There is no simple answer to how to access data from multiple devices either.

Kleppmann et al. [9] formulated seven principles of *local-first software*. Their vision was to harness the benefits of the software-as-a-service business model but with the assumption that the application makes a local copy of all data and treats it as a primary replica. The user does not need to wait for actions to finish or be online all the time; changes made locally are eventually propagated to the cloud.

Peer-to-peer storage

BitTorrent[1], IPFS[2], SAFE Network[3], and PingER [10] are examples of technologies based on peer-to-peer file sharing and communication protocols. Data are split into chunks and replicated among agents available on the network; there is no single point of failure.

It is worth noting that p2p-storages are decentralized from the user's perspective as well, which brings additional challenges. Published data are hard or even impossible to alter. If not encrypted, distributed chunks may potentially be used in an unsolicited way. Therefore, additional measures for data protection might need to be considered upfront.

[1] https://www.bittorrent.org/
[2] https://ipfs.tech/
[3] https://primer.safenetwork.org/

Personal storage

The last group in our classification comprises technologies that enable users or system administrators to create a secure space for their data, decide the exact physical location of data, grant and revoke permissions to access resources, and so on. Individual spaces are interconnected and establish a federated network.

Mastodon[4], Diaspora[5], and Hubzilla[6] are representatives of rather specialized software for social interactions, aiming to meet the needs of groups. Solid[7] stands out as a technology for storing the personal data of individuals.

Assuming that the service application does not perform unintended operations or act maliciously, personal storage offers the most flexibility concerning the risks we address. Furthermore, we do not attempt to implement a social network but focus on providing a novel way to search routes. Hence, Solid will be our choice for storing user data.

1.3 Solid Project

Social Linked Data (Solid) [11] is a specification built on top of the existing infrastructure and standards, and any Solid-compliant implementation acts as a personal storage we described in the previous section.

Started as a research project by Tim Berners-Lee and other collaborators in response to the improper use of the Web [12], the technology later developed into the startup Inrupt[8] to turn Solid into a market-ready product.

Within Solid, user data are stored in personal online data stores called *pods*, following the principles of *Linked Data*. It distinguishes between structured RDF datasets and unstructured binary blobs (text, images, etc.). Pods reside on a *pod server* and are accessed via a well-defined HTTP API. Servers are also responsible for authentication, authorization, access control, and request handling [13].

The concept of decentralization using Solid is shown in Figure 1.2. The owner of a pod can grant and revoke permissions based on unique user identifiers. Agents attempt to access a resource through Solid apps. Agent 1 is allowed, but Agent N is not because their identifier does not appear in the list.

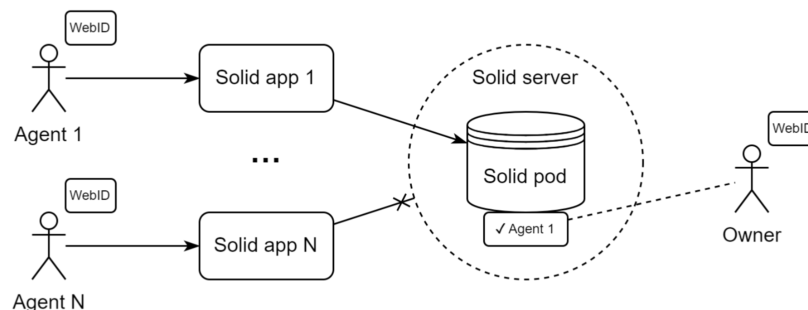

Figure 1.2: The concept of decentralization using Solid.

[4] https://mastodon.social/
[5] https://diasporafoundation.org/
[6] https://hubzilla.org/
[7] https://solidproject.org/
[8] https://www.inrupt.com/

2. Analysis

At the beginning of this chapter, we define abstract concepts useful for discussion and formalizing the problem at hand. Following that, the requirements and use cases are stated, both of which are fundamental for proper system design. Next, possible sources of open geodata available on the Web are described, and a conceptual model for the application is proposed. Lastly, we analyze and compare competing applications tangibly related to our goals, aiming to identify the set of features essential for a reasonable user experience.

2.1 Definitions

To simplify further communication, we give definitions used throughout the thesis and then illustrate some of them in a real-life situation.

Public storage is a read-only collection of entities accessible to all users via a predefined set of methods.

Private storage is a collection of entities accessible only to a specific user and other actors authorized by the user.

Client is an integral part of the application, allowing user interaction with both public and private storage.

Keyword is a string having an "instance of" relationship with a given place.

This place is an instance of a museum.

Attribute is a singleton or key-value pair having a "has a" relationship with a given place.

This museum has a capacity of 100 people.

Attribute filter is a constraint imposed on a certain attribute. Only museums with a phone number and a large enough capacity would be selected by the query from the example.

Place is any location recognized by the public or private storage and associated with at least one keyword and a (possibly empty) set of attributes.

Named location is a distinguished point on the map recognized by the private storage. Possible candidates are locations with special meaning for the user, such as place of work, home, and so on.

Path is a composite structure consisting of a starting point, destination, optional waypoints in between, and a polygonal chain.

Direction is a path resulting from a search query based on an ordered sequence of locations.

Category is a composite structure consisting of a keyword and attribute filters.

Categorical matching is the process of recognizing whether a place belongs to a given category. Formally, we speak about the application of a binary function $m : V \times C \to \{0, 1\}$, where V is the set of places, and C is the set of categories. Its value is equal to 1 if a place has the keyword of a category and satisfies all attribute filters, and 0 otherwise.

Route is a path resulting from a search query based on a set of categories. It also comprises the context in which the search has been conducted.

Suppose a group of N tourists wants to plan an excursion, starting at the hotel, finishing in the city center, and visiting a castle and a museum. Not every facility is suitable due to the group size. Furthermore, it is advisable to make a reservation upfront by phone or other means.

The organizer marks two terminal points in a *client* and configures two *categories* similar to the one below that *match* only specific castles and museums.

$$(\underbrace{\mathsf{museum}}_{\textit{keyword}}, \underbrace{\{\mathsf{phone}, \mathsf{capacity} \geq N\}}_{\textit{attribute filters}})$$

Once the query form is prepared, a request is issued. The application operates over *places* stored in the *public storage* and suggests several *routes*. The organizer saves one of them in their *private storage* for later use.

2.2 Requirements

In this section, we specify the properties that our solution (hereinafter referred to as *SmartWalk*) should possess by expressing functional and non-functional requirements.

Functional requirements are statements about the system that clarify the services and features it should provide to users and external systems, as well as how it should behave with respect to the surrounding environment.

On the contrary, non-functional requirements, also called quality attributes, define characteristics that improve the ability of the system to deliver functionality and meet stakeholders' expectations [14].

2.2.1 Functional requirements

Here, we outline the features our system implements, some implied by the goals of the thesis and others derived from the analysis of existing solutions in Section 2.8.

Place search

F1 The system allows users to search places around some center point within a crow-fly (or great circle) distance of at most *15* km.

F2 The user can choose a center point on the map or from the stored options.

F3 The user is allowed to adjust a maximum distance (radius of a circle).

F4 The user can provide a list of categories for place matching. If the list remains empty, the system retrieves all places within the bounding area.

F5 The result is paginated (5, 10, 20, or 50 places per page) and sorted by the distance in ascending order.

F6 The user is able to filter the result by category. Only places associated with at least one active category are listed.

Route search

F7 The application enables users to search routes leading from a starting point to a destination with walking distance of at most *30* km.

F8 The user can choose a starting point and destination on the map or from the stored options. Furthermore, points are swappable and might be distinct.

F9 The user is allowed to adjust a maximum walking distance.

F10 The user is required to provide at least one category for place matching.

F11 The user can define an order in which categories should be visited by configuring a set of *arrows*. Arrows have the same meaning as the word "before" and are not allowed to form cyclic dependencies. Given categories $\{1, 2, 3\}$ and the only arrow $(1 \to 2)$, the following orders are valid:

$$3 \to 1 \to 2, \qquad 1 \to 3 \to 2, \qquad 1 \to 2 \to 3.$$

F12 The result contains only routes satisfying the following conditions:

- a route starts at the starting point, ends at the destination, and visits at least one place from each category,
- the "before" relation is preserved for all arrows,
- the distance of the route is less than or equal to the maximum.

F13 The result is paginated (one route per page) and sorted by the distance in ascending order.

F14 The user can (un-)hide places associated with a selected category.

Direction search

F15 The system allows users to search directions for a given sequence of locations; a result contains those passing through all points in the given order.

F16 The user can extend the sequence by selecting a point on the map, choosing from the stored options, or appending from a detailed view.

F17 The user can move points of the following entities to the sequence:

- routes from the result of a search query,
- directions and routes in the private storage.

F18 The application enables the user to rearrange points of the sequence one by one and reverse them.

F19 The result is paginated (one direction per page) and sorted by the distance in ascending order.

Entity management

F20 Search queries consider only entities of the public storage and ignore those stored privately.

F21 The user can create named locations within their private storage.

F22 For every place, the system maintains a unique persistent identifier and provides a detailed view of all information available in the public storage.

F23 The detailed view of a place contains an embedded JSON-LD representation of that place. The object should include only specific predetermined fields.

F24 The user can save a *partial copy* of a place linked to the original object and redefine the name. Its detailed view indicates the existence of that copy.

F25 The system generates routes and directions on demand without granting identifiers. Search queries for these kinds always yield "new" objects. The user can save them in their original state.

F26 If an entity of the public storage has a unique identifier, only one copy of that object may exist in a given private storage.

F27 The application permits the user to view, edit, and delete routes, directions, copies of places, and named locations in their private storage.

F28 The application enables the user to authenticate against a Solid server and activate an available pod as private storage.

F29 The system supports all use cases without extra effort from the user regarding entity management and the concept of decentralization.

*Rationale: To ensure the user has a gentle learning curve. Essentially, this requirement mandates the use of two interchangeable storages. Please refer to Requirement **N4** for implementation details.*

User interface

F30 The system provides users the option to request their current location.

F31 When configuring a category, the system suggests possible keywords based on a prefix. Once a keyword is selected, the system provides the user with the corresponding keyword-specific attribute filters and their bounds.

F32 The removal of a category also resets the list of arrows.

F33 The system enables users to clean up request forms and alter entered items, including points, categories, and arrows.

F34 The result panels contain information regarding the number of objects found and summarize the input from the request form.

2.2.2 Non-functional requirements

The quality of a software system can be evaluated in different dimensions, including performance, testability, and modifiability. Below, we list the essential attributes that our application possesses. The reasons behind these requirements and the decisions made to meet them are discussed in the following chapters.

N1 The application follows the three-tier architectural pattern and consists of the frontend, backend, and infrastructural nodes.

N2 The frontend is designed as a single-page application and written in TypeScript using the React library and Material UI components.

N3 The frontend implements a panel-centric layout and provides the same level of user experience on both desktop and mobile devices.

N4 The frontend employs IndexedDB as device storage and a Solid pod as external storage. Both of them are instances of private storage.

N5 The frontend prevents users from entering invalid input while interacting with panels and the map.

N6 The map performs marker clustering if the number of drawn objects exceeds a predefined limit.

N7 The backend is designed as a Web API application written in C# using the ASP.NET Core framework and asynchronous primitives.

N8 From the algorithmic point of view, the implementation of a route planner prioritizes efficiency and a variety of results over optimality.

N9 The system uses MongoDB for both storing entities and facilitating search queries.

N10 The application utilizes the OSRM backend as a routing engine to calculate paths and distance matrices.

N11 The application maintains an in-memory data structure to index keywords and lists of corresponding attributes and their bounds.

N12 The frontend communicates with the backend via API based on the HTTP protocol and the JSON data format.

N13 The response time of reasonably large place search queries does not exceed *1* second on average.

N14 The response time of reasonably large route and direction search queries does not exceed *2* seconds on average.

N15 The system integrates information from at least three data sources: OpenStreetMap datasets, Wikidata, and DBPedia.

N16 The data ingestion strategy supports filtering based on bounding boxes and enables incremental updates.

N17 The architecture enables horizontal scaling as the number of active users or data volume increases.

N18 The solution provides a cross-platform deployment procedure optimized for running on a stand-alone personal computer.

N19 The frontend is supported in Mozilla Firefox, Google Chrome, and Microsoft Edge browsers.

N20 The source code depends on the existing open-source software and is published on GitHub as a monorepo, enabling easy collaboration with other developers.

N21 The source code is testable, reusable, and decently documented. The solution uses standard techniques for code organization.

2.3 User stories

To justify the relevance of our application and understand the target audience, we list several examples of actors which would benefit from using it. All examples are given in the form of user stories, following the template "*As a <role>, I can <capability> so that/to <receive benefit>.*"

- As a self-guided tourist, I can create an itinerary providing a unique local experience to harness the full potential of my journey.
- As an international student, I can explore my local area to fulfill everyday duties, including doctor, shop, or leisure.
- As a family, we can visit the zoo and include a restaurant on the way home so that no one feels hungry after a long day.
- As a tourist guide, I can create routes that pass through numerous categories to satisfy the interests and needs of different people.
- As a person waiting for the train, I can find a short route through a souvenir shop and an ATM to ensure I return before the train departs.
- As an information point, I can answer questions from my clients so that I always send them in the right direction.
- As an entrepreneur, I can identify missed opportunities in a specific area to expand my business and increase my chances of success.
- As a person looking for an apartment, I can learn more about services within walking distance to choose an option that fits my needs best.
- As an urban planner, I can reflect on the development of areas I have been responsible for to help me improve my expertise.
- As a culture vulture, I can look for art galleries, theaters, cinemas, and more in one request so that I do not have to explore each category separately.

2.4 Roles

The system primarily caters to the needs of individuals and legal entities with diverse backgrounds that do not know their local area well or seek insights. It also does not support collaborative scenarios or interactions other than searching and storing results.

Therefore, we define only one role within the system, which is a **user**. Due to **F29**, users authenticated against a Solid server behave the same way as if they were guests.

2.5 Use cases

In addition to the requirements, we provide the essential use cases that users might need to perform within the system. Each consists of up to five parts: the initial state of the system, the normal flow, the final state, and optional alternatives and extensions. To grasp how use cases are organized, it may be beneficial to review Figures 2.1, 2.2, 2.3, and 2.4 before delving into implementation details. Their detailed descriptions are provided in Attachment A.3.

The following use cases guarantee the ability to search for and save entities.

- *UC01: Select point*
- *UC02: Add category*
- *UC03: Add arrow*
- *UC04: Search routes*
- *UC05: Show detailed view of a place*
- *UC06: Save place*
- *UC07: Search directions*
- *UC08: Modify route*

Users are allowed to view, delete, and edit stored entities. The process of deletion is similar to editing metadata and is therefore excluded from consideration.

- *UC09: View entity*
- *UC10: Edit entity*

The following use cases explain the interplay between device storage and Solid pod so that user data is never lost or compromised.

- *UC11: Activate Solid pod*
- *UC12: Deactivate Solid pod*

While designing scenarios, the following principles have been applied to simplify human-computer interaction and make application behavior predictable.

- Actions that modify the application state, such as saving entities or adding arrows, are always bound to the corresponding dialog. The user is allowed to close the dialog as long as the action has not been confirmed. Otherwise, they should wait until the request is resolved.
- Actions initiating a search prevent users from leaving the panel until the request is resolved.
- Errors that appear within a dialog or panel do not corrupt its state. In other words, information entered before the error has occurred remains intact.

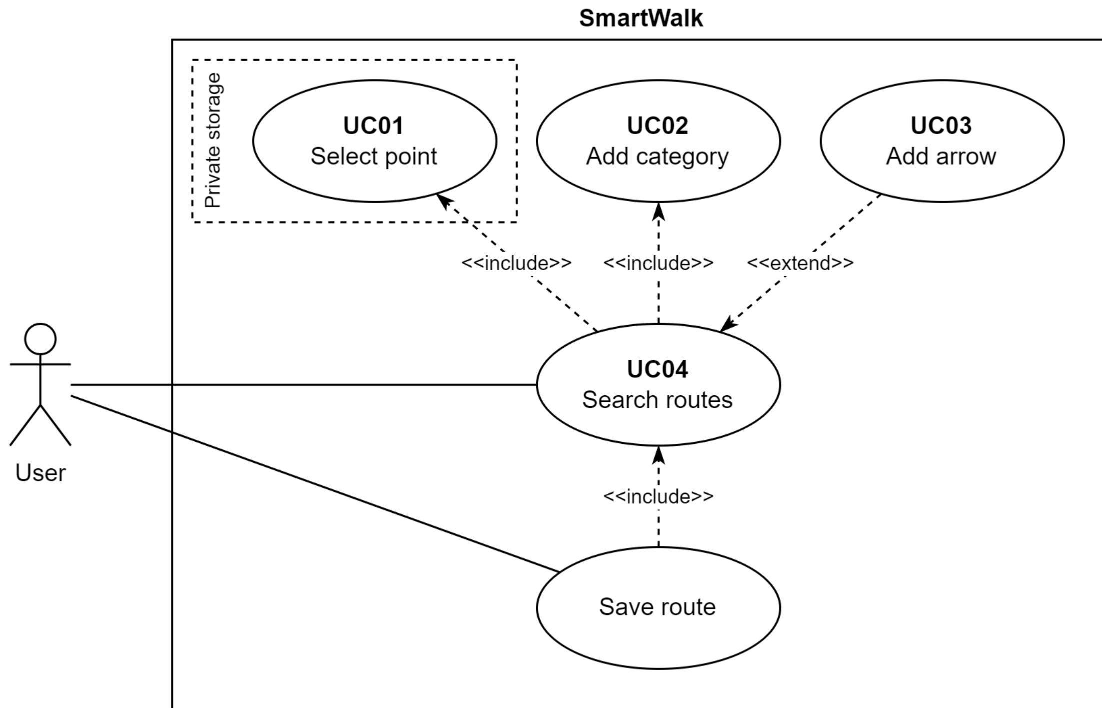

Figure 2.1: The use cases related to searching routes.

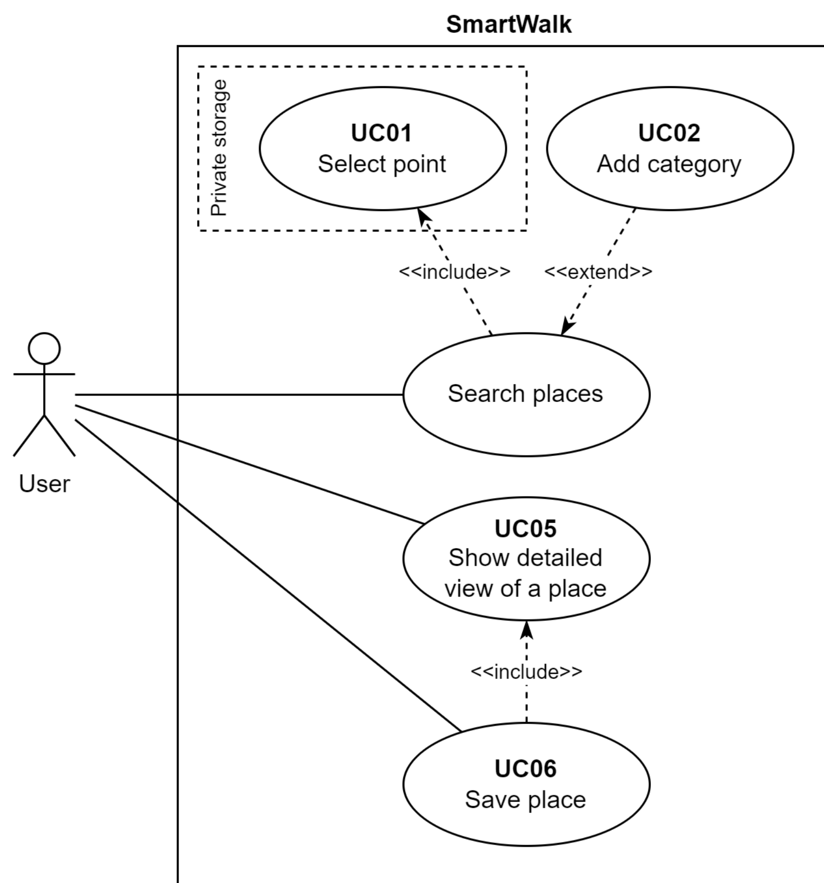

Figure 2.2: The use cases related to searching places.

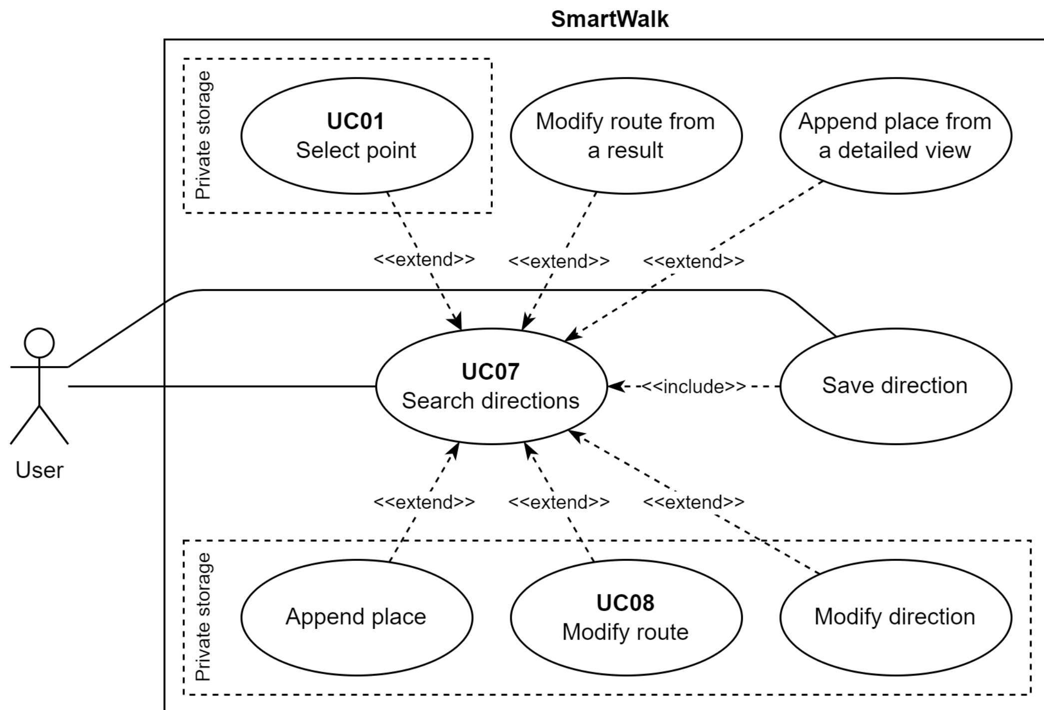

Figure 2.3: The use cases related to searching directions.

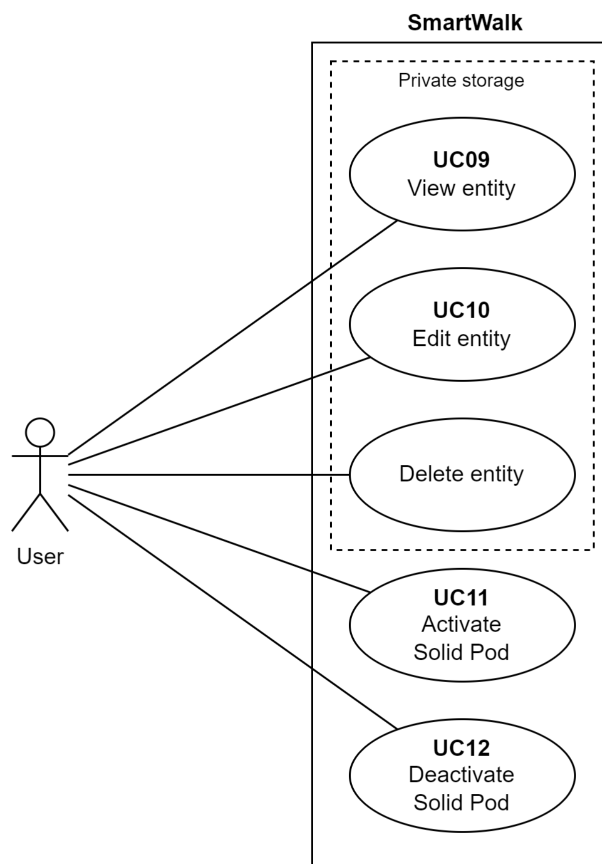

Figure 2.4: The use cases related to entity and storage management.

2.6 Data sources

We have introduced abstract concepts that form the foundation of our system and have outlined the procedures it will support. However, data is the key element that connects ideas to the real world and makes applications usable for end users. The goal of this section is to discuss the properties and internal organization of the selected sources of open geographic data. Their actual usage is tightly related to the conceptual model and is covered later in Section 3.3.

We will deal with two types of datasets: semi-structured and structured. The former usually has some internal organization but does not explain the meaning of individual items or constrain them. The latter is completely defined by schema or abstract model. For example, recall the code snippet with JSON-LD serialization from Section 1.1. The meaning of the property `knows` is precise in the presence of `@context` and becomes ambiguous without it.

All sources of structured data that we consider further are represented by large *knowledge graphs*. Although this term is rather general [15], our experience will be limited to publicly available HTTP endpoints capable of processing SPARQL queries.

2.6.1 OpenStreetMap

OpenStreetMap (OSM)[1] is a community-driven initiative to create freely available global-scale geographic data. The project was founded in 2004 and, over the years, has become the largest dataset of its kind available on the Web, thanks to volunteers and regular contributors.

Perhaps one of the main reasons OSM has taken off so well and attracted the attention of thousands of people is that everyone can get started with minimal effort. Its conceptual data model defines the following three types of elements.

- *Nodes* are point-like objects that either represent standalone real-world entities or are used as building blocks for other composite elements.
- *Ways* are ordered sequences of nodes suitable for modeling line features or area features, depending on whether the terminal points are distinct.
- *Relations* are the most generic elements designed to aggregate other primitives, including relations, and describe new meanings and behavior. Typical examples are polygons with holes, transportation routes, or even administrative boundaries.

Furthermore, each OSM entity could have a set of key-value pairs (*tags*) attached. Both key and value are *free-form text* entries. A key could occur within the same entity only once. Tag statistics are collected and published on Taginfo[2].

An example of a widely used pair is `tourism=museum`. One might argue that museums have a broader meaning and are not necessarily associated with tourism. That is another significant disadvantage of the OSM data model, making it hard to deal with in real applications.

[1] https://www.openstreetmap.org/
[2] https://taginfo.openstreetmap.org/

To fix the problem, we could try to map tags to some *ontology* or a schema. Sophox[3] and WorldKG[4] attempt to create knowledge graphs from OSM data, but none of the projects help to convert textual values to structured ones.

There are multiple options for how to obtain the OSM dataset. Full and partial dumps for selected regions are available on Geofabrik[5] in binary and textual data formats. More customized queries are realizable via Overpass API[6].

Despite incompleteness and heterogeneity, OSM remains a significant source of geographic data for our application. We were able to extract about 115000 distinct entities, including polygons and relations, within a bounding box of Prague, the capital of the Czech Republic. Wikidata discussed just below contained about 35000 point-like objects for the same area.

2.6.2 Wikidata

Wikidata[7] is one of the largest publicly accessible sources of general-purpose structured data on the Web. The original idea behind this project was to create a database shared among other Wikimedia projects to deduplicate and interlink already existing information. The repository is maintained and extended by editors and robots. In addition, it integrates knowledge from various datasets with compatible licenses while keeping links to originals.

In Wikidata, anything — a real-world entity or abstract concept — can be represented by an *item* with a label, description, and aliases. Each item is granted a unique, persistent identifier starting with `Q`. For example, `Q188112` denotes the National Museum in Prague.

Facts about items are expressed using property-value pairs called *statements*. In the following example, the property `P31` is translated to English as "instance of," and `Q33506` stands for the concept of "museum" as an institution:

$$\texttt{Q188112}^{\text{(item)}} \;\rightarrow\; \texttt{P31}^{\text{(property)}} \;\rightarrow\; \texttt{Q33506}^{\text{(value)}}.$$

To put it simply, a property describes a relation between an item and a value. Moreover, it restricts the data type of the value and possibly its range so that facts have a predictable structure and are easy to work with.

The data model of Wikidata is built upon RDF but extends it with internal conventions. Please refer to Figure 2.5 for a detailed view of its structure as presented in a web browser.

The purpose of a *qualifier* is to provide additional context about a statement, such as the timestamp when it started being true.

More than one value may exist for a given item and property. For example, an object could be an instance of a "museum" and "gallery" simultaneously. *Ranks* help to order values by relevancy.

One of the characteristics Wikidata mentions about itself in the product statement is being a secondary database. *References* are intended to point to the source of information, making statements verifiable.

[3]https://sophox.org/

[4]https://www.worldkg.org/

[5]https://download.geofabrik.de/

[6]http://overpass-api.de/

[7]https://www.wikidata.org/

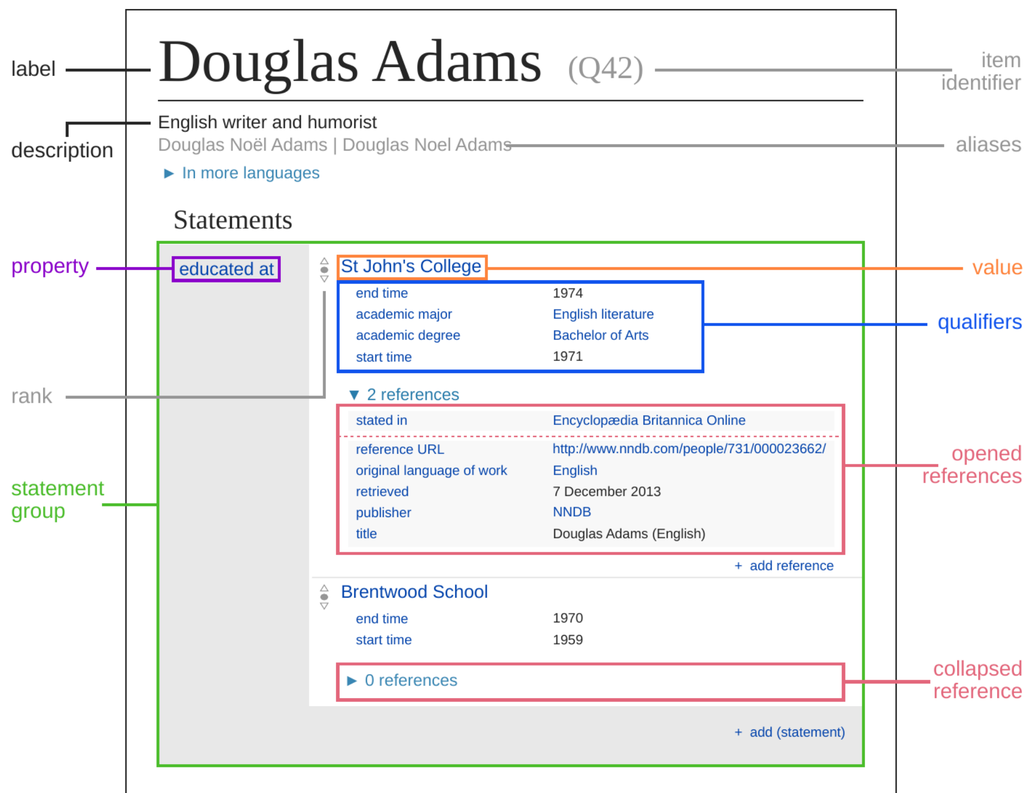

Figure 2.5: Simplified data model of Wikidata [16].

Wikidata follows the principles of *Linked Data*; entity representation can be requested in different data formats, including JSON-LD, using content negotiation. The database can also be accessed via the SPARQL-based Wikidata Query Service (WDQS)[8]. Furthermore, WDQS offers service extensions, with the most relevant ones in the context of this thesis being geospatial `around` and `box`. The following snippet demonstrates how to retrieve all instances of museums with their locations within the bounding box of Prague.

```
SELECT ?wikidataId ?location WHERE {
  ?wikidataId wdt:P31/wdt:P279* wd:Q33506. # an instance of a museum
  SERVICE wikibase:box {
    ?wikidataId wdt:P625 ?location.
    bd:serviceParam wikibase:cornerSouthWest
      "Point(14.18 49.90)"^^geo:wktLiteral.
    bd:serviceParam wikibase:cornerNorthEast
      "Point(14.80 50.20)"^^geo:wktLiteral.
  }
}
```

2.6.3 DBPedia

DBPedia[9] is another project in the field of *Linked Data* that aims to convert Wikipedia into a structured form. Its core component, the information extraction framework initially developed by Auer et al. [17], parses articles into an abstract

[8]https://query.wikidata.org/
[9]https://www.dbpedia.org/

syntax tree, performs the algorithm, and transforms the structure into an RDF graph. This workflow implies that the resulting dataset is read-only and the only meaningful way to update it is through repeated extraction.

Although both DBPedia and Wikidata are related to Wikipedia, these projects differ in many aspects [18]. For example, DBPedia utilizes a more general RDF as its data model. Fortunately, if the range of the property is defined for a given triple, the extraction framework accepts only valid values.

Information from DBPedia is accessible through methods similar to Wikidata, including dumps, content negotiation, and the SPARQL endpoint. Due to the generality of the extraction procedure, certain data items might be challenging to interpret and work with. Therefore, we will use this dataset as an auxiliary source to enrich existing entities.

2.7 Conceptual model

We have reached the point where we can interconnect abstract ideas with the available datasets. Let us show the UML class diagram of the conceptual model, depicted in Figure 2.6. It mostly incorporates the entities from Section 2.1 that we have already covered. Nonetheless, several aspects deserve extra attention.

All concepts evolve alongside `Keyword`, which is essential in the context of this thesis. `Place` represents the simplest form of a geographic entity that is sensible to consider. Certain locations, such as tourist attractions, are inherently data-rich. `ExtendedPlace` helps accommodate additional information, where the `attributes` data field can be understood as a collection of key-value pairs. Moreover, we assume that each key clearly defines the data type of its value, akin to properties in Wikidata.

To search for routes, a user must provide at least one `Category`. In the initial version of the application, we implement *six* types of attribute filters, formally defined in Table 2.1. The variable v holds the result of accessing `attributes`[`key`]. Values for other variables are supposed to be set by the user.

Filter	**Variables**	**Predicate**
Existential	v : `any`	defined(v)
Boolean	v, b : `boolean`	$v = b$
Numeric	v, a, b : `number`	$a \leq v \leq b$
Textual	v, t : `string`	v.contains(t)
Set (include)	$v, s \neq \emptyset$: $\langle$`string`$\rangle$	$\exists w \in s : w \in v$
Set (exclude)		$\forall w \in s : w \notin v$

Table 2.1: Attribute filters.

The first type, existential, is applicable when we do not care about the shape of the value, such as an email or phone number. The meaning of the next three rows is obvious, whereas the last two filters are less common. Suppose the user wants to have dinner in a restaurant with Czech or Slovak cuisine. Any instance of a restaurant with the attribute `cuisine` that includes `czech`, `slovak` or both would work. The last filter is the negation of the previous one.

To enable effective communication between the user and system and reduce the number of meaningless queries, bounds within an `AdviceItem` restrict ranges of their corresponding filters. The user can select only values valid for at least one place in the public storage while configuring a category. Bounds are essential for numeric and set filters.

A careful reader might have noticed that named locations do not appear in the drawing. Instances of this type have the same structure as `Place` but are not associated with any `Keyword`. Routes and directions could still visit them.

The diagram highlights some of the elements. The green boxes are concepts of the global state and exist in the public storage. The blue entities are generated by the application in response to queries, and users can choose to store them.

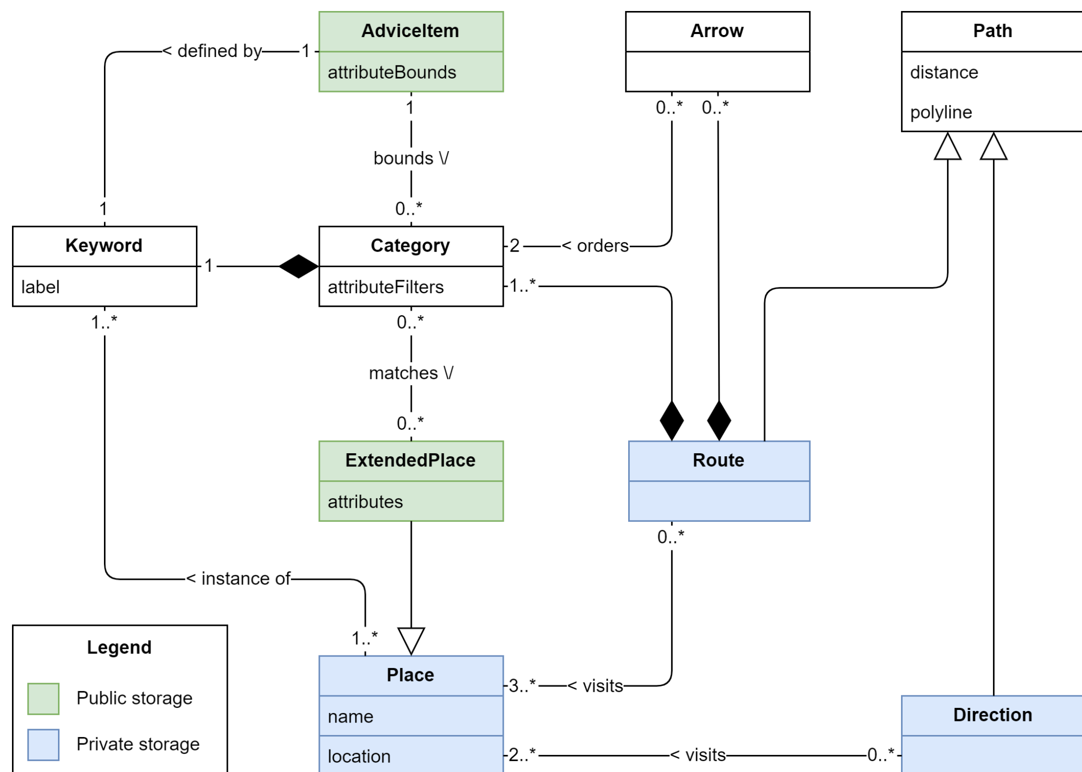

Figure 2.6: A UML class diagram of the conceptual model.

The last point we should discuss here, which eventually impacts the user interface and the technology stack, is how to define `attributes`. Since the selected data sources offer different experiences in terms of data quality, the preprocessing phase is necessary. There are at least two general approaches to this issue.

We could explicitly define keys, data types, value constraints, and mapping between our attributes and items of an external data source. This representation enables the extraction of parsable data from the OSM dataset. Moreover, resulting entities would be easier to work with for tasks like configuring categories or presenting detailed views. However, the internal structure of knowledge graphs would be underutilized.

Another method takes advantage of the fact that both knowledge graphs define the notion of a data type. For instance, property `P2043` in Wikidata and predicate `length` in DBPedia both describe the "length" of an object. The former expresses values in terms of `Quantity`, while the latter uses simple doubles.

In principle, we could ingest only property-value pairs with reasonable data types, such as boolean or double, while preparing our attributes. This approach is more generic than the previously discussed and, at the same time, problematic in several aspects.

- The OSM dataset remains beyond the scope; we still have to parse it.
- A user might encounter too many attribute filters while configuring a search query.
- A detailed view of a place would lose appeal if it had to implement a generic layout.
- General ranges could potentially complicate the user interface. Recall the scenario from *Definitions*, where the group of people was looking for a museum to visit with a specific capacity requirement. In Wikidata, values of the property `P1083` are constrained within the range of 0 and 9×10^5. Consequently, using a slider to obtain user input is no longer feasible. Instead, entered values must be validated, and errors should be reported.

Taking into account the goals of the thesis, we prioritize usability over generality. Hence, our final decision is to choose the first option and implement a fixed set of attributes.

2.8 Existing solutions

Web maps have been around for almost three decades [1]. As a result, there are implicit expectations regarding the functionality a typical application should offer and the behavior of its user interface.

In this section, we consider three types of products and compare them with our solution: commercial platforms backed by large technology companies, less popular applications implementing innovative features, and research-oriented prototypes. Table 2.2 summarizes the differences and commonalities in *eight* criteria.

While most websites are usable out of the box, extended capabilities might become accessible upon login. We assume that authentication has been performed.

2.8.1 Mapy.cz

Our starting point is Mapy.cz[10], a web mapping platform that originated in the Czech Republic. It is undoubtedly one of the most popular projects in the local market, integrating numerous industry best practices as well as several distinctive features. We use Mapy.cz as a reference point for other alternatives.

The opening page contains a vertical panel on the right side with three tabs: Search, Directions, and My Maps; the rest is filled with raster tiles containing interactive points of interest. If the width of the viewport becomes sufficiently small, the layout switches to the mobile version. The panel moves to the bottom but keeps its layout unchanged.

[10]https://en.mapy.cz/

The Search tab includes a search bar and category buttons. Clicking on a button has the same effect as if we had entered its label into the input. Based on provided text, autocomplete could suggest two kinds of objects that act very differently. Place items lead to the detailed view. Category items initiate search and return a list of relevant places with basic refinement filters. Filter configuration seems identical for all categories. The result is relative to the visible part of the map; shifting or zooming causes immediate recalculation. Different states of this tab are presented in Figure 2.7.

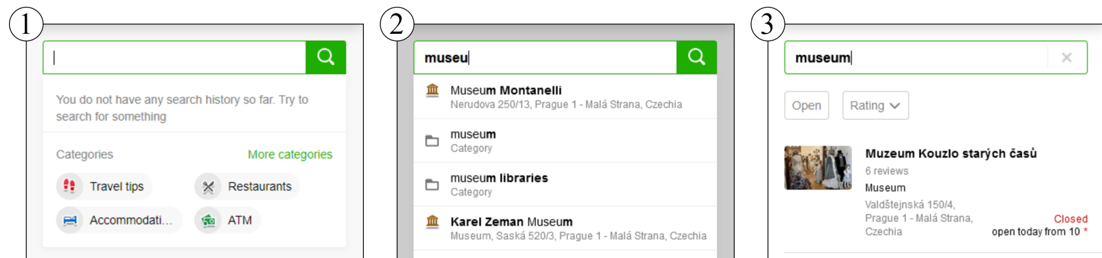

Figure 2.7: Mapy.cz – Search tab: ① initial view, ② autocomplete options with places and categories, ③ the results of a categorical search.

The Directions tab allows a user to construct a path in iterations. The sequence can be extended and customized in various ways; users can insert or append new points, rearrange points through drag-and-drop, and even reverse the order. The result is calculated for every sequence configuration (see Figure 2.8 for an example). Nevertheless, this workflow suffers from all the drawbacks mentioned in the *Introduction*.

Figure 2.8: Mapy.cz – Directions tab: ① initial view, ② a list of directions.

Mapy.cz also supports automatic route planning, addressing one of our concerns. Given a point on the map and a maximum distance, the Circuit Route Planner[11] generates a round trip and a list of tourist attractions nearby. Even though user input is reduced to two items, we can identify several drawbacks of their approach:

- the route is eventually circular,
- found points might lie far from the polygonal chain,
- the user has no control over what categories will be included in the list.

[11]https://napoveda.seznam.cz/en/circuit-route-planner-biking-and-walking/

An integral part of flawless user experience is the ability to store and reuse previously obtained assets. The application enables storing all types of entities and accessing them through the My Maps tab later. When saving a place, users can assign it any desired name. Paths in the storage are editable. Users have the option to duplicate a path by resaving it as a new entity. To enhance personalization, users can also label any position on the map and treat it as a distinct place in their collection.

2.8.2 Komoot

Komoot[12] is a web application aimed at active individuals who enjoy outdoor sports such as hiking, cycling, or running. The startup page contains two sections: Discover and Route Planner.

The Discover section offers a community-driven tour recommendation system with filtering capabilities (duration, difficulty, elevation, etc.); routes are generated by an algorithm based on user activities[13].

If none of the routes meet all the requirements, users have the option to modify a discovered path or create a new one. In contrast to *Mapy.cz*, a total of *22* hard-coded category filters are available to users and activated by corresponding checkboxes while manually constructing a sequence of points, as shown in Figure 2.9. The resulting direction can be saved and updated later. The storage functionality allows filtering by location, date, sport, or searching by name.

Outdooractive[14] is another website with similar objectives and features.

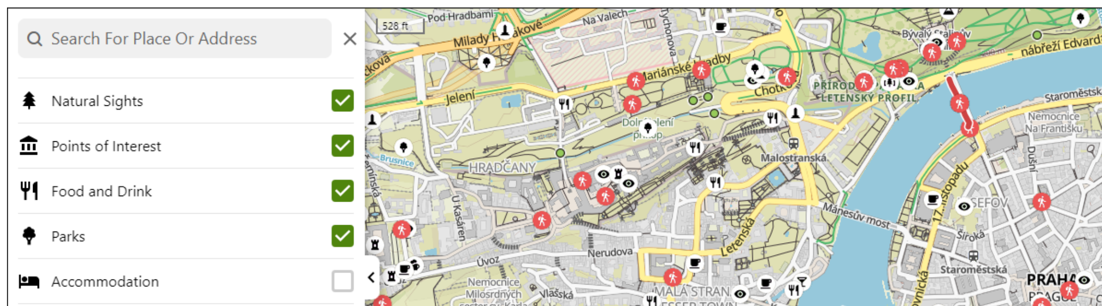

Figure 2.9: Komoot: category filters.

2.8.3 Kurviger

Kurviger[15] is a web application intended to cover the needs of motorcyclists. It shares several similarities with the aforementioned web pages, such as the ability to filter places by category (up to *10*), store places and routes, and rename places.

Additionally, users are able to generate randomized round trips based on a starting point, compass direction, and tortuosity (refer to Figure 2.10 for an example). Repeated searches with the same parameters yield distinct results.

[12] https://www.komoot.com/
[13] https://support.komoot.com/hc/en-us/articles/360058879211
[14] https://www.outdooractive.com/en/
[15] https://kurviger.de/about/

Applications Naviki[16] and cycle.travel[17] offer a similar user experience for cyclists.

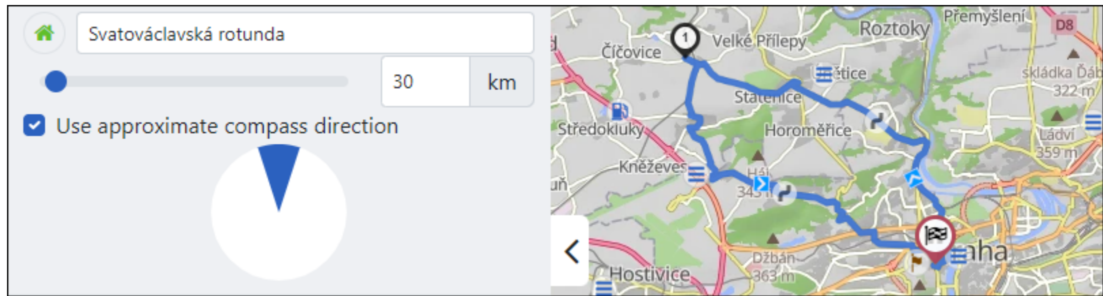

Figure 2.10: Kurviger: a randomized round trip.

2.8.4 City Trip Planner

Until now, none of the considered applications have sufficiently addressed the goals related to route search, mainly due to limited customization capabilities.

A different approach was demonstrated by Vansteenwegen et al. in [19], where they introduced an expert system, City Trip Planner, capable of planning tourist trips based on personal preferences.

In their system, users begin by entering trip constraints such as the duration of the visit, terminal points, and lunch breaks along with their interests specified through groups of keywords. Using this information, the server performs calculations and generates a multi-day trip itinerary. Additionally, places of high significance that have not been selected are presented alongside it. In subsequent iterations, users have the flexibility to improve the route by adding or removing points and request recalculations as needed.

The web page was organized as a multi-step questionnaire. The authors reported positive user feedback on the application, highlighting that the layout of the user interface posed no problems for the majority.

It is worth noting that although the City Trip Planner might no longer be available, recent projects have adopted similar principles and algorithms [20, 21].

2.8.5 WISER

Friedman et al. [22] developed and presented WISER, an experimental mobile application for iterative route search over probabilistic geospatial datasets.

We should acknowledge that some of the *SmartWalk* requirements and the current algorithmic implementation of the search procedure were inspired by WISER and other articles written by its authors.

Both applications allow users to enter keywords and a set of arrows. In each iteration, WISER presents only the next place to be visited. The rationale behind this approach was that a calculated path might include waypoints the user would find irrelevant upon visiting. Instead, the system builds the route based on

[16] https://www.naviki.org/en/
[17] https://cycle.travel/map

provided feedback. A negative response triggers a repeated calculation, while positive feedback removes the keyword from the list. The process continues as long as at least one keyword or candidate to visit is present.

2.8.6 Feature comparison

We have discussed different solutions that are freely available on the market. Table 2.2 compares *SmartWalk* with them based on eight criteria. In the list below, we elaborate on three less obvious ones.

[**Place search**] *Mapy.cz* has the most expressive mechanism for searching places out of other alternatives. The set of keywords is unbounded, and a user can apply filters to the result (post-filtering). *SmartWalk* improves its approach by allowing the user to search for multiple categories and embed filters into a query (pre-filtering).

[**Route search**] The purpose of *City Trip Planner* was to provide tourists with a somewhat imprecise solution for navigation with no guarantees to satisfy all entered constraints. *SmartWalk*, on the other hand, strives to fulfill a given task precisely. Unlike *City Trip Planner*, *SmartWalk* does not define a notion of time, as its perception is highly individual and, therefore, difficult to capture accurately.

[**Saving places**] An attached copy maintains a link to the original, whereas a detached one does the opposite.

Feature	**Mapy.cz**	**Komoot**	**Kurviger**	**City Trip Planner**	**WISER**	**SmartWalk**
Entity search						
Place search	Geocoding, unbounded set of keywords, post-filters	Geocoding, bounded set of keywords	Geocoding, bounded set of keywords	✗	✗	Categorical
Direction search	✓	✓	✓	✗	✗	✓
Route search	Round, via nearby attractions	Pre-calculated by an algorithm	Randomized, round, compass direction	Personalized, time-/keyword-aware	Ordered, keyword-aware	Ordered, categorical
Entity management						
Data ownership	Service provider	Service provider	Service provider	✗	✗	User
Creating named locations	✓	✗	✓	✗	✗	✓
Saving places	Link to an original, user-defined name	Link to an original	Detached partial copy	✗	✗	Attached partial copy
Saving routes and directions	✓	✓	✓	✗	✗	✓
User interface						
Layout	Panel	Panel	Panel	Questionnaire	Minimal, task-oriented	Panel

✓ denotes "implements", and ✗ denotes "does not implement."

Table 2.2: Feature comparison.

3. Design

The objective of this chapter is to lay the foundations for future implementation. The user interface is designed based on use cases and communicated with the help of simplified wireframes. Then, we discuss the main parts of the system, including applied design principles and the technology stack. Next, we explain how to acquire data from the selected sources. The last section establishes a theoretical framework that enables us to select appropriate routing algorithms.

3.1 User interface

The user interface of *SmartWalk* will be panel-based, following the approach used in many industrial solutions. The proposed navigation schema[1], depicted in Figure 3.1, consists of ten interconnected panel views. The layout of the "Entity Viewer" is similar to that of other panels; therefore, we omit it for brevity.

It is worth mentioning that the schema enforces a specific workflow focused on task completion. While on "Place Result," a user can only access "Place Search" and "Detailed View." This relation is expressed in terms of direct accessibility.

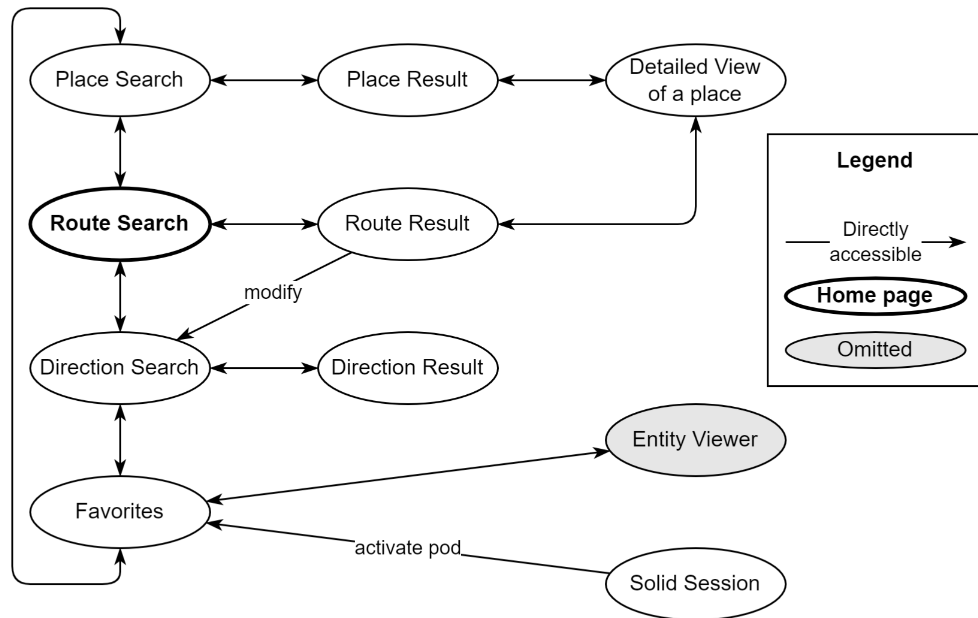

Figure 3.1: Navigation schema.

The home page of the application is designed to carry out route searches. To fill out the request form, users need to provide a starting point, destination, arrows, and at least one category. Figures 3.2, 3.3, and 3.4 illustrate wireframe prototypes for the respective dialogs. Subsequently, the panel might reach the state shown in Figure 3.5. Please note that the starting point is yet to be defined.

After issuing a search query, the user is presented with the results, as demonstrated in Figure 3.6. Waypoint names are hyperlinked to their respective places. Additionally, users have the option to center the map on a specific waypoint and

[1] If not stated otherwise, the pictures were created with Draw.io drawing software.

access the list of categories it belongs to by clicking the pin object within the route sequence.

Panels for searching places and directions utilize the same supporting dialogs and have only minor differences compared to route search. Their wireframes are represented in Figures 3.7, 3.8, 3.9, and 3.10.

The only thing we have not discussed regarding directions is how the system allows users to rearrange the sequence mentioned in **F18**. Its points are draggable by elements consisting of six dots, and the "Rv" button reverses their order.

Each place in the public storage is assigned a detailed view showing all available information in a standard form, as depicted in Figure 3.11. If present, the exact geometry is drawn under the pin. One interesting aspect of this wireframe is that the place is already saved with a different name. The application detects the entity by its identifier and informs the user via the message box. Furthermore, the "Save" button is deactivated. This functionality fulfills Requirement **F24**.

Figure 3.12 and 3.13 shed light on how to log in against a Solid server and activate an available pod. After entering an address and clicking the "Log in" button, the user is redirected to an external web page where they complete the authentication process. The appearance and actual procedure depend on the provider.

After pod activation, the user is redirected to the panel where the collection of entities is maintained. The layout of this panel view is rather straightforward; Figure 3.14 defines three separate sections for places, routes, and directions in the given order. All operations on entities assumed by **F27** are accessible via menu buttons. The user is also informed about the type of decentralized storage used. In particular, three states are possible: device storage, Solid storage, and emergency in-memory storage as a fallback for an outdated web browser.

Up to this point, we should have gained the impression that having more than one pin drawn on the map is a common situation. All markers in the presented wireframes were drawn as circles colored with shades of grey. To enhance visual perception and improve the user experience with the application, we propose the schema listed in Table 3.1, which will determine pin colors in a given context.

Pin color	Description
■ #2aad27	Starting point
■ #cb2b3e	Destination
■ #797979	Center point
■ #2a81cb	Not stored places
■ #9c2bcb	Stored places

Table 3.1: Possible colors of pins on the map and their meaning.

The meaning of the first three rows is evident; let us concentrate on the rest. The main objective is to visually separate places in the result of a query based on whether they appear in the private storage or not. According to **F22**, every point of interest in the public storage is assigned a unique identifier. Therefore, we can incorporate these identifiers into partial copies and use them while analyzing the results.

Please note that wireframes are intended for design simplification. For details on the actual user interface and its behavior, refer to Attachments A.1 and A.3.

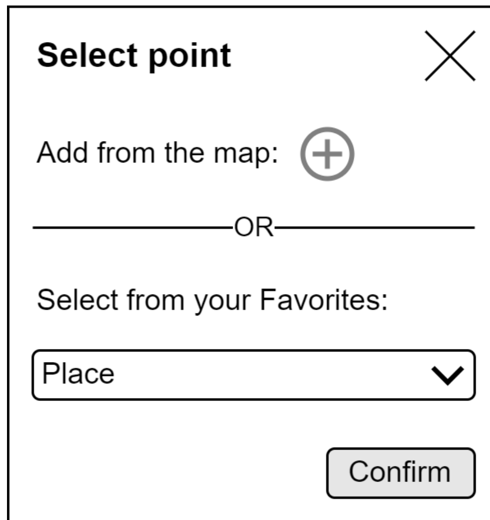

Figure 3.2: Wireframe with the dialog for selecting a point.

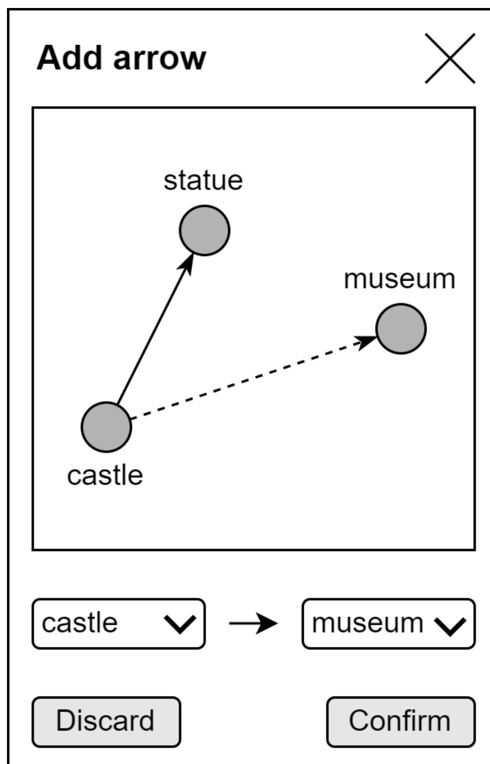

Figure 3.3: Wireframe depicting the state of the dialog for configuring arrows. The arrow with a solid line represents a “confirmed” arrow, and the one with a dashed line represents a “not confirmed” arrow.

Figure 3.4: Wireframe depicting the state of the dialog for configuring categories after selecting a keyword, including all five types of attribute filters.

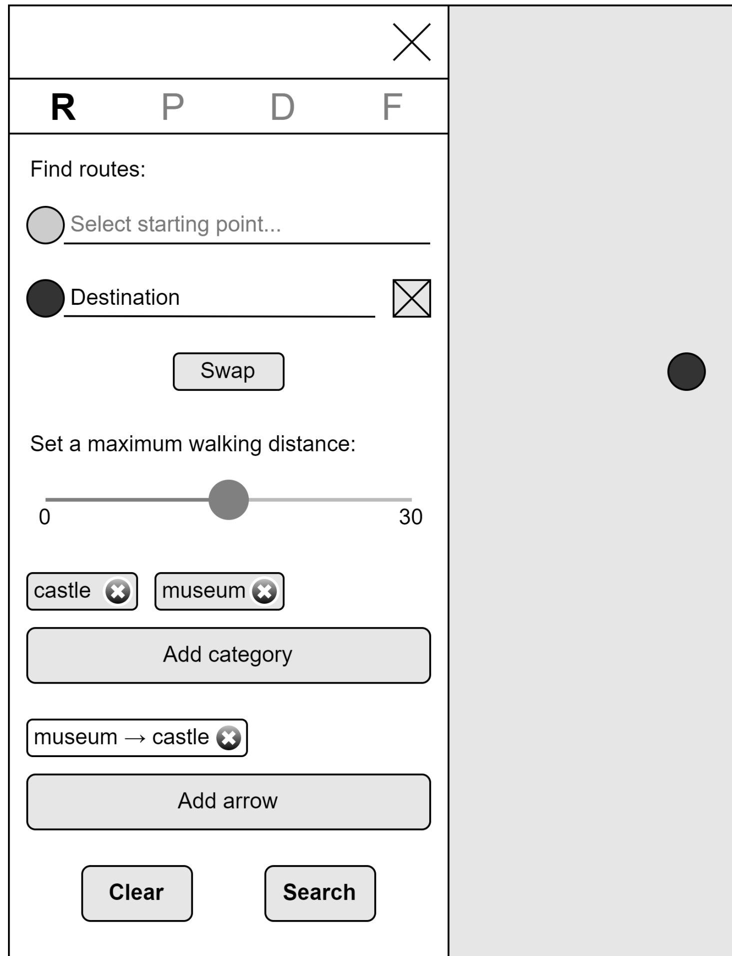

Figure 3.5: Wireframe with the panel for searching routes.

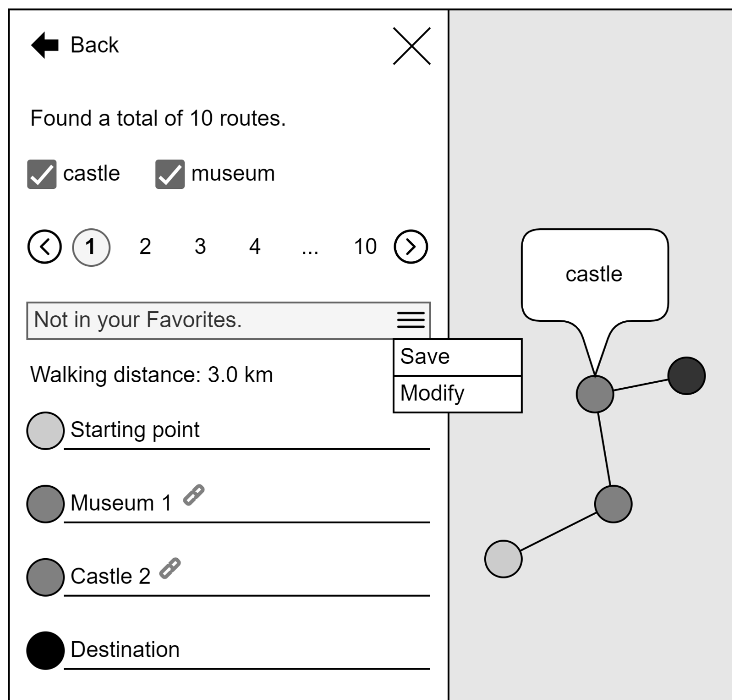

Figure 3.6: Wireframe with the panel showing the results of a route search.

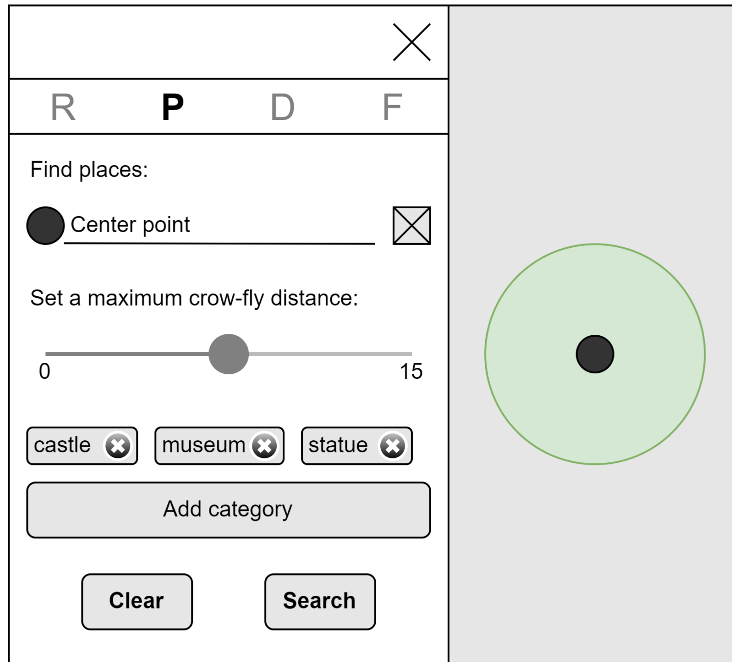

Figure 3.7: Wireframe with the panel for searching places.

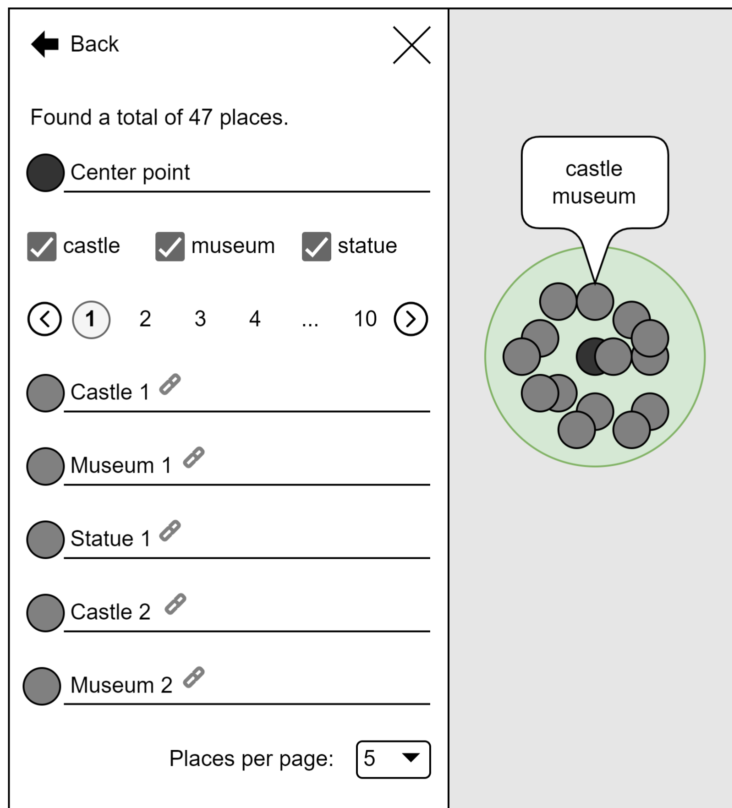

Figure 3.8: Wireframe with the panel showing the result of a place search.

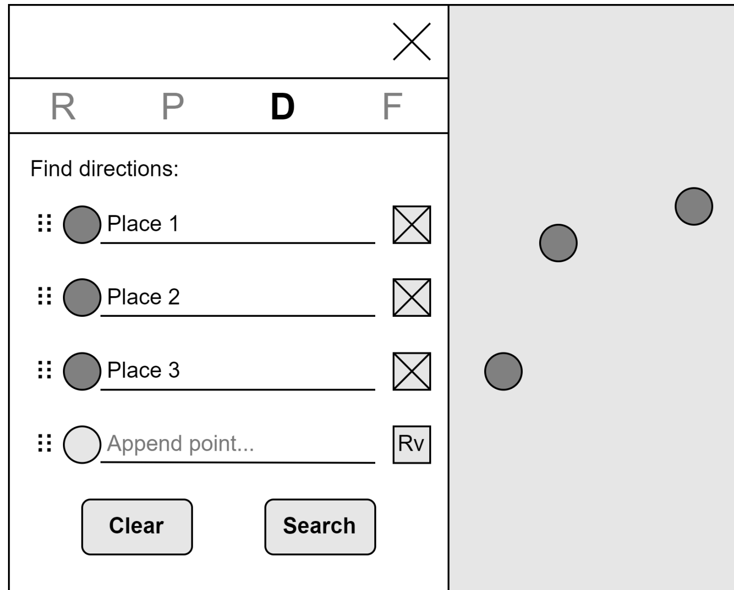

Figure 3.9: Wireframe with the panel for searching directions.

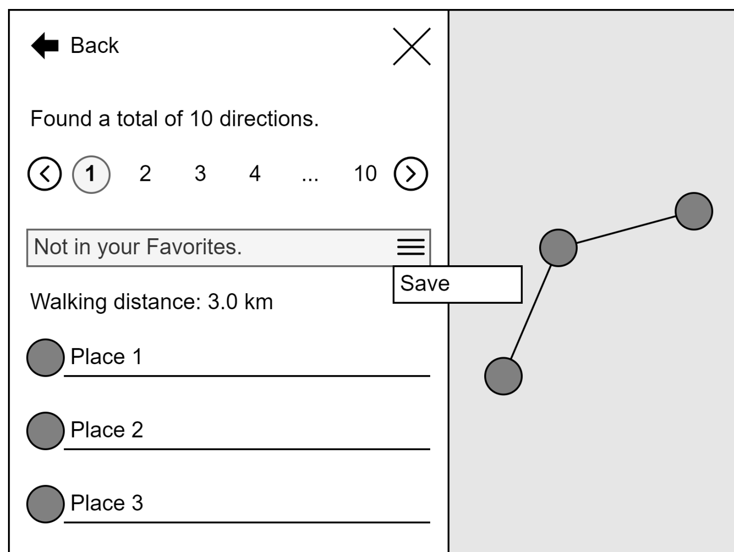

Figure 3.10: Wireframe with the panel showing the results of a direction search.

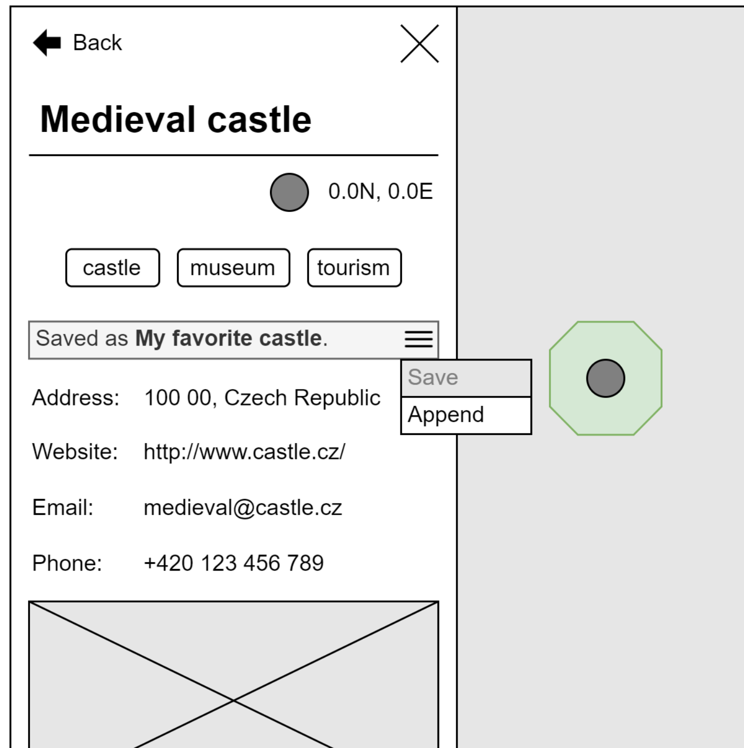

Figure 3.11: Wireframe with the detailed view of a place.

Figure 3.12: Wireframe with the Solid login dialog.

Figure 3.13: Wireframe with the panel for activating a Solid pod.

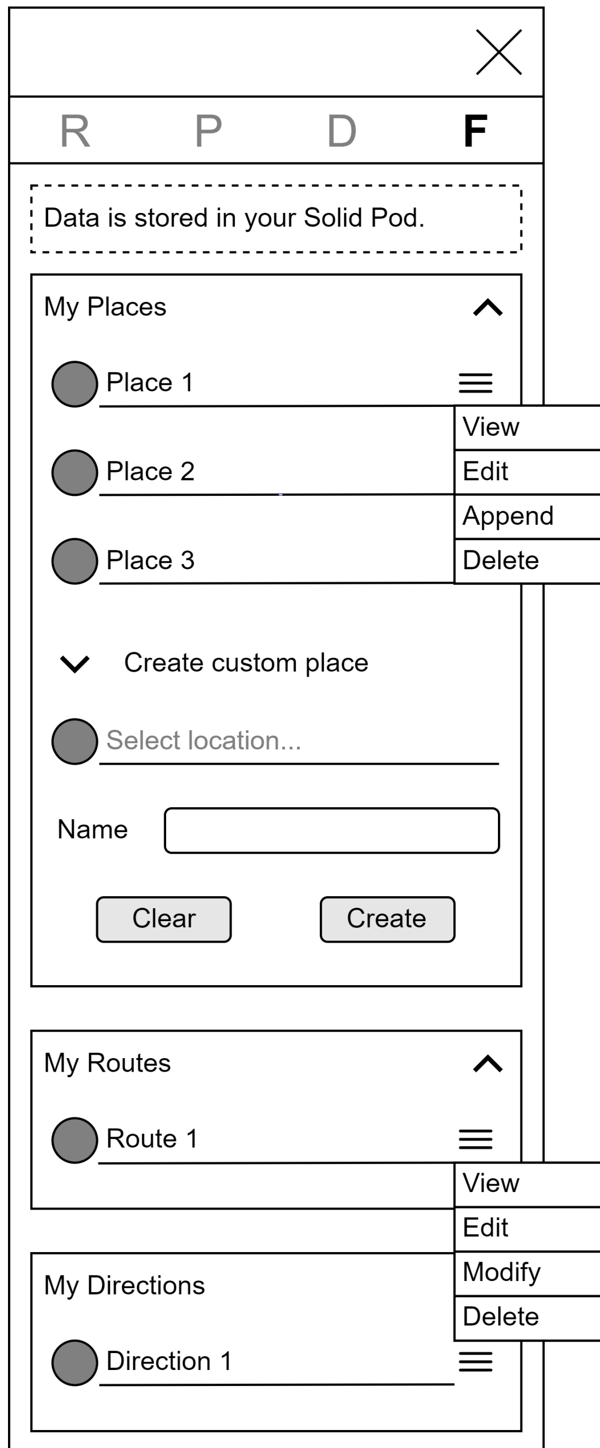

Figure 3.14: Wireframe with the panel containing stored (favorite) entities.

3.2 Architecture

Solving algorithmic problems in the presence of modern web technologies requires dozens of smaller parts glued together to perform specific tasks efficiently. Their integration would be challenging without the proper level of abstraction. For this reason, the architecture of the *SmartWalk* solution is demonstrated using the *C4 model* [23], which defines the following hierarchy of elements.

- A *software system* brings value and makes sense to consider in isolation, as is the case with *SmartWalk*.
- A *container* is a runnable part of a software system (database, mobile application that communicates over the network, web server, or even script).
- A *component* represents a non-deployable implementation of an interface used by a container.
- *Code* describes the internal organization of a component through the use of UML classes or similar notation.

Although the model defines four types of diagrams into which blocks are gathered, we only utilize two of them — container and component views — as others do not provide the desired level of detail.

The architecture of our solution comprises *six* essential parts, as illustrated in Figure 3.15. It can be viewed as an implementation of the *three-tier* architectural pattern with the adjustment that users store their data externally. Moreover, the *backend* (or service provider) lacks access to user data, thus addressing **G2**.

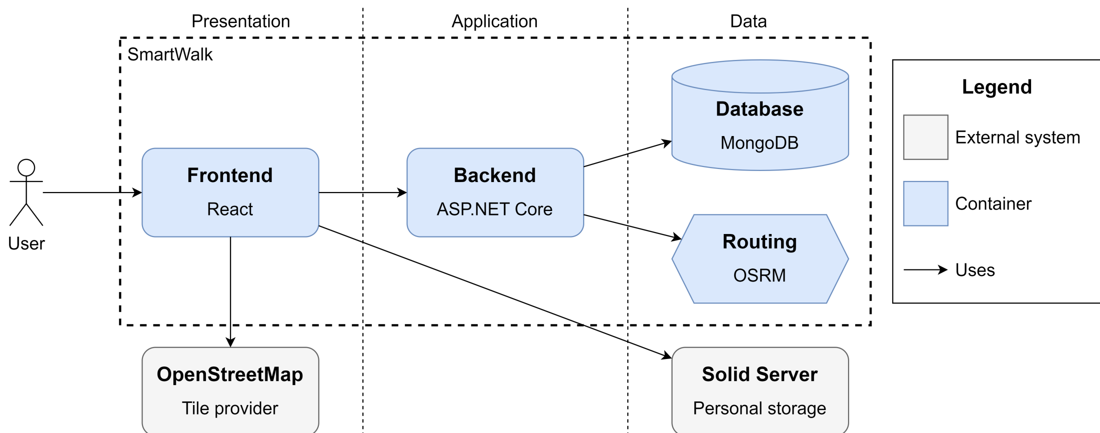

Figure 3.15: C4 container diagram of the SmartWalk software system.

The *frontend* serves as an entry point to the application, offering a rich user experience. It communicates with the application tier and personal storage through a well-defined Application Programming Interface (API).

All searching and planning functionality resides within the *backend*. Since user data are stored elsewhere, we assume that only read operations, such as generating new paths or fetching places, should be supported.

The *database* and *routing* are containers that supply business logic with actual data. The former acts as an entity store and search index, whereas the latter is a specialized routing engine.

3.2.1 Frontend

The *frontend* is a part of the presentation tier that is exposed to a user via a web browser, providing all intended functionality. We further decompose it into five smaller components drawn in Figure 3.16.

`PanelDrawer` mimics the user interface and navigation schema shown in Section 3.1. `SessionProvider` ensures the login dialog and handles the proper switch over to the "Solid Session" panel. Other parts make possible interaction with the environment. In particular, `Map` is responsible for loading tiles, drawing markers and vector geometries on the map. `SmartWalkAPI` provides a set of functions for retrieving data from the *backend*. Finally, `Storage` is an abstraction that unifies methods for accessing both device and personal storages.

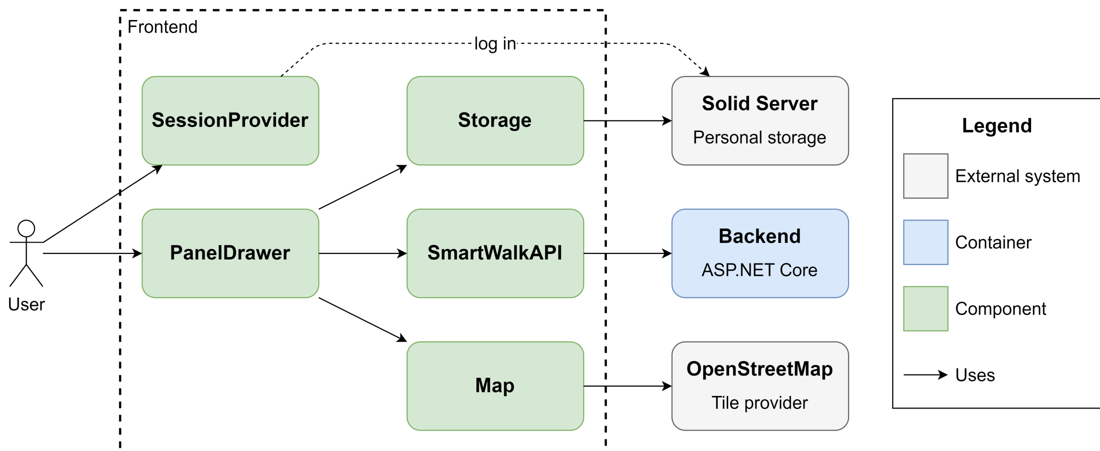

Figure 3.16: C4 component diagram of the Frontend container.

If the frontend were programmed as a classic multi-page application, navigating between pages would trigger a reload of the entire map. Hence, it is designed as a *single-page application*; once loaded, the state is altered via small updates in response to user actions, and tiles are left intact. In the remainder, we discuss the technology stack employed in later implementation.

JavaScript is a natural choice for programming web solutions. However, its weak typing could make development challenging as a project grows, and catching potential errors becomes crucial. To address this issue and improve the maintainability and clarity of the codebase, we choose TypeScript[2], a statically typed JavaScript dialect with built-in type inference.

It is evident from the functional requirements that our use cases on the client side do not require extensive data processing and are limited to sending search queries and showing results. Since performance is not a top priority, the *frontend* employs the React[3] library. Despite not being the most efficient compared to other libraries and frameworks, it offers several advantages: a composable component-based architecture, a declarative syntax with JSX[4] extension similar to HTML markup, and an intuitive unidirectional data flow where changes are propagated from parents to their nested components.

[2] https://www.typescriptlang.org/
[3] https://react.dev/
[4] https://react.dev/learn/writing-markup-with-jsx

3.2.2 Backend

The next container, the *backend*, is related to the application tier. Its internal structure, revealed in Figure 3.17, loosely follows the Action Domain Responder (ADR) pattern [24]. ADR is an alternative to the classic Model View Controller (MVC), offering better abstractions for HTTP-based services with distinct client and server sides.

`TController` receives a request object and performs validation or parsing to reject malformed input early, similar to the `Action` in ADR. Well-formed data is then handed over to the corresponding domain-level `THandler`, a realization of a targeted use case. `TResponder` is responsible for completing the response object based on three possible outcomes: a valid result of a calculation, an internal server error, or failed validation. Finally, `Gateways` are entities of the data access layer that implement abstract interfaces.

Please note the letter `T` in the names of some elements, indicating the generic nature of the diagram. A total of *five* distinct pipelines are implemented following the same concepts. The actual routing and invocation of the proper controller are details delegated to the framework.

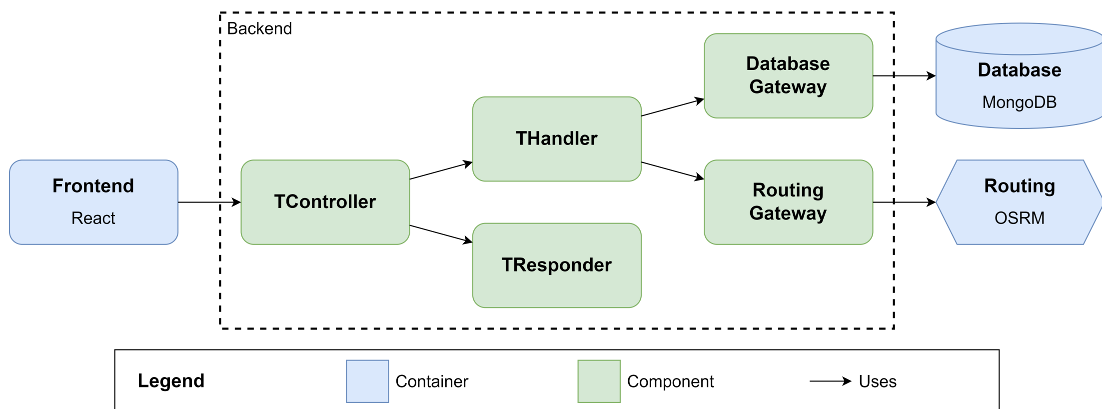

Figure 3.17: C4 component diagram of the Backend container.

Another design approach we apply while writing source code, which positively influences code modifiability and testability, is the Onion Architecture [25]. The pattern relies on a more general Dependency Inversion Principle (DIP) and places the domain model at the top of the hierarchy. In our solution, the entities defined in Section 2.7 and planning algorithms are positioned at the center. Use cases are implemented with the help of specific *handlers* that depend on the core primitives, and instances of infrastructural *gateways* are injected into *controllers*.

As in the previous section, we provide an overview of the technologies used later. Let us begin with a discussion of the programming language. The *backend* is expected to perform computationally intensive tasks. Reasonable assumptions regarding its runtime include a managed environment with garbage collector, support for asynchronous operations, native multi-threading, and accessibility to other developers. The most suitable languages appeared to be Java and C#[5], and we choosed the latter due to the author's prior experience.

The next step is to select a technology for creating the API. The most obvious approach is to implement separate endpoints over HTTP protocol for each query

[5]https://learn.microsoft.com/en-us/dotnet/csharp/

type. An advantage is that, since the *backend* is essentially *stateless*, calculated results could be effectively cached by the server and intermediaries. Other options, such as GraphQL[6], SOAP[7], and gRPC-Web[8], were considered but deemed too advanced or atypical for our use cases.

A standard framework for building web applications in C# is ASP.NET Core[9]. It offers configurable request processing pipelines and out of the box dependency injection container.

There are two main approaches to creating an HTTP interface with ASP.NET Core: minimal and controller-based APIs[10]. The first approach skips most of the boilerplate code; endpoints are defined in terms of lambda functions directly attached to respective routes. Despite its simplicity, minimal API lacks support for model validation, among other things. Thus, we select the second, which enforces design patterns and conventions for code organization.

3.2.3 Database

Data collected from various sources ends up in the *database*, a container within the data tier. This segment of the architecture should ultimately behave as public storage from Section 2.1 accessible through the *backend*. Based on the requirements analysis, we have divided the capabilities into two semantically independent subsets, each designed to accomplish certain tasks.

The first group enforces the *database* to act as a *store*. Those include fetching places by an identifier and the ability to accommodate arbitrarily large amounts of possibly incomplete entities (the presence of keys in the `attributes` is not stable across the collection).

The second role is an *index* with a focus on facilitating search queries. In some sense, these capabilities are orthogonal to the previously mentioned ones but are equally important, as they directly affect the system's ability to fulfill Requirements **N13** and **N14**. We expect this container to be able to:

1. suggest the k most relevant keywords for a given prefix,
2. find all places around some center point ordered by the crow-fly distance or within an arbitrary polygon with no distance constraints,
3. and have a mechanism to express all six types of attribute filters.

A commonly applied practice is to implement these roles as two services: the store with entities and the index holding only the information to be queried. Together with the routing engine, this separation would make the system challenging to run on a personal computer, as the setup would require allocating substantial hardware resources. Therefore, the goal is to consolidate all functionality into one container without compromising performance.

We should note that a relational database with ACID properties would not be a good fit, mainly due to the unbounded size of the place collection and its

[6]https://graphql.org/
[7]https://www.w3.org/TR/soap12-part1/
[8]https://grpc.io/docs/platforms/web/
[9]https://learn.microsoft.com/en-us/aspnet/core/?view=aspnetcore-6.0
[10]https://learn.microsoft.com/en-us/aspnet/core/fundamentals/apis?view=aspnetcore-6.0

heterogeneity. At the same time, spatial queries are well-supported by PostGIS[11] extension for PostgreSQL, and the database itself is used by the OSM project.

The alternative is to consider a NoSQL database with support for spatial queries, a weak schema, and horizontal scaling. To simplify the decision-making process, we refer to the article by Guo and Onstein [26], where they reviewed ten of the most popular NoSQL solutions. Taking into account their overview and the fact that places, as defined by the *Conceptual model*, exist as isolated bundles, we conclude that MongoDB[12], a document-oriented database, successfully addresses our needs.

The last observation is that keyword suggestions are likely to be the most frequent requests, primarily because place and route search queries depend on their results. Answering them without disturbing the *database* is highly desirable. We assume that the collection of keywords is small enough to fit in the main memory, and such requests can be carried out by the PruningRadixTrie[13], a high-performance data structure written in C#.

3.2.4 Routing

According to Figures 3.6 and 3.10, the application should present any route or direction to the user using points of interest and a polygonal chain drawn on the map. The segments of this chain should overlap with the actual street network to achieve the desired level of precision. Since most of our locations come from OSM, it is also rational to use this dataset as the basis for finding traversals.

Implementing a routing engine would have a scale infeasible for one developer. For this reason, we rely on one of the most popular open-source projects in this field. Please refer to the overview [27] for more details.

Due to the focus on efficiency and variety of results stated in **N8**, we select Open Source Routing Machine (OSRM)[14] written in optimized C++. To achieve high performance in calculations, this engine builds hierarchies of pre-computed shortcuts that enable early pruning of candidate vertices that would fail to contribute to the final result. As a tradeoff, OSRM is quite resource consuming, but we have already freed up sufficient computation power and memory by merging store and index in one container.

3.2.5 Personal storage

As specified in Requirement **N4**, both device and personal storages should be used interchangeably. In principle, only a Solid pod would be enough, but several drawbacks led us to incorporate a fallback.

- Solid is a promising technology with far-reaching implications concerning data ownership, and it requires an understanding of the related concepts. However, not many users are willing to invest additional effort.

[11] https://postgis.net/
[12] https://www.mongodb.com/
[13] https://github.com/wolfgarbe/PruningRadixTrie
[14] https://project-osrm.org/

- The Solid protocol does not have a finalized version. According to the documentation: *"This document may be updated, replaced or obsoleted by other documents at any time. It is inappropriate to cite this document as other than work in progress [11]."*
- There is currently no market-ready Solid implementations or providers with guaranteed quality of service.

In this thesis, we target only Inrupt PodSpaces[15] with instances of Enterprise Solid Server (ESS)[16]. We use Inrupt JavaScript Client Libraries[17] to authenticate against servers and perform data access operations. The frontend has also been tested on Community[18] and Node[19] Solid Servers, yielding positive outcomes.

Solid allows for storing both structured and unstructured data. The former would be advantageous if we were to expose this information to other agents, but this is not the case. As a simplification, it is assumed that entities are saved as separate JSON files in the *hardcoded* folder `${storage-root}/smartwalk/`.

3.2.6 Tile provider

Results of queries are presented on the map together with square-shaped tiles, visually representing the corresponding local area. Tiles are typically distributed in two forms: raster pictures rendered on the server side and vector graphics rendered by a client. As smartphones are among the supported platforms, pictures fit better due to their optimized power consumption pattern.

We use OSM standard tile provider. It is permitted[20] as long as the application does not generate too much traffic. Tiles are downloaded using the following link, where `s` stands for a subdomain (optional parameter), `x` and `y` point to a rectangle in the grid, and `z` is a zoom level:

`https://${s}.tile.openstreetmap.org/${z}/${x}/${y}.png`.

3.3 Data preparation

Potential data sources and distribution methods were discussed in Section 2.6, but not all of them were equally good for our needs. We utilize only the following *five* distributions in the prescribed way.

OSM dump files, published in PBF format, are used to build graph structures for OSRM and populate the database. The first procedure is done automatically by standard tools, with details given in Attachment A.1, Administrator's guide. The other part is our responsibility.

PBF files can be handled as a stream of elements using the OsmSharp[21] library written in C#. Unfortunately, the OSM data model does not restrict the shape of tags. Hence, we need to implement custom parsers.

[15] https://docs.inrupt.com/pod-spaces/
[16] https://docs.inrupt.com/ess/latest/
[17] https://docs.inrupt.com/developer-tools/javascript/client-libraries/
[18] https://github.com/CommunitySolidServer/
[19] https://github.com/nodeSolidServer/
[20] https://operations.osmfoundation.org/policies/tiles/#requirements
[21] http://www.osmsharp.com/

The community recognized this problem and established naming conventions and guidelines. Taginfo helps to identify the most popular keys and *frequencies* of values associated with them. This information allows us to filter out infrequent strings by setting a threshold and develop finer-grained extractors. The service exposes an HTTP endpoint so that the statistics for a given key can be retrieved via a simple `GET` request.

Determining a location for *relation*-type OSM elements is not straightforward. We must retain all references to nodes and ways with coordinates in the main memory, and the task becomes too large to accomplish on an ordinary computer. Instead, these smaller requests are delegated to Overpass API. The service precomputes the *centers* of multi-polygonal bodies, which we retrieve in bulk using the following query sent in the `data` parameter of an HTTP `GET` request:

```
/* s, w, n, e should form a valid bounding box */
[out:json];
relation(${s},${w},${n},${e})[type=multipolygon];
out center;
```

We mentioned earlier that Wikidata has a predictable internal structure and can be considered an independent source of geographic data. In contrast, triples in DBPedia are inherently more abstract. Hence, it makes sense to separate the process of ingesting data from knowledge graphs into two self-contained steps: creating simple stubs with georeferences from Wikidata that do not yet exist and enriching (or updating) places.

To obtain locations, we utilize a slightly modified version of the query mentioned in Section 2.6.2. In particular, matching by the property path `P31/P279*` is a time-consuming operation and should be omitted.

The following query serves a generic template to handle the *enrichment* step.

```
PREFIX my: <http://www.example.com/#>
PREFIX owl: <http://www.w3.org/2002/07/owl#>

CONSTRUCT {
  ?wikidataId
    my:dbpedia ?dbpediaId;
    my:param ?param.
}
WHERE {
  VALUES ?wikidataId {
    ${l} # expand a list of Wikidata identifiers
  }
  ?dbpediaId owl:sameAs ?wikidataId.
  OPTIONAL {
    ?dbpediaId ${p} ?param. # replace by a predicate
  }
  ... more OPTIONAL blocks follow ...
}
```

The result of this query is a new RDF graph due to the `CONSTRUCT` clause. To narrow the search and focus solely on relevant entities, the `VALUES` block prescribes possible values that the `wikidataId` variable can hold. This block is populated with Wikidata identifiers available in our database. Some targeted properties might be missing, and we should consider partial matches as valid results. The `OPTIONAL` block encloses optional parts of the graph pattern. Furthermore, the predicate `owl:sameAs` establishes an equality relation between entities from different knowledge graphs.

Wikidata and DBPedia are queried via HTTP endpoints that accept percent-encoded SPARQL queries in the `query` parameter. To simplify response handling, we employ content negotiation in the way described below.

1. Request data with the `Accept` header set to `application/n-triples` for DBPedia and `application/n-quads` for Wikidata.

2. Parse response string into JSON-LD, and compact the graph using injected `@context`. This step is carried out by the jsonld[22] library.

3. We receive a plain JavaScript object with the known property configuration as the output.

Eventually, a place could have up to *40* distinct attributes. Some of these attributes are aggregated into larger objects, resulting in only *29* that are queryable.

Below, we note two more rules that should be followed to ensure the feasibility of data preparation procedures and the correctness of the input.

- Routines extracting or fetching places have a *bounding box* as a compulsory input parameter.

- As a matter of fact, MongoDB uses coordinates given in the *WGS84* Coordinate Reference System (CRS) to index spatial objects. Fortunately, all the information repositories we consider comply with that CRS. Otherwise, location references would need to be translated before writing to the database.

3.4 Routing algorithms

The final piece of information we need before moving to the next chapter is how to map the route search defined by Requirements **F7** to **F12** onto a theoretical framework. A better understanding of the underlying mathematics positively affects our reasoning abilities and later implementation.

First, various formalizations are described, culminating in one that best aligns with our needs. Subsequently, two polynomial-time heuristics are presented, capable of finding "valid" routes from **F12**.

We expect thorough knowledge of the basic courses on discrete mathematics and theory of computation, typically covered in a standard Computer Science curriculum. Please refer to [28, 29, 30] for a refresher, or feel free to skip this section if theoretical aspects are not of interest to you.

[22] https://www.npmjs.com/package/jsonld

3.4.1 Notation

Imagine a map of some city as a collection of objects, where business centers, pharmacies, and shops are interconnected by sidewalks, highways, and roads. A *finite graph* is an abstract model that accurately describes structures of this kind. To avoid misunderstanding, we provide formal definitions of concepts used in this section; most of them can be found in the literature referred to just above.

A *graph* G is an ordered pair (V, E), where V is a set of *vertices* (points drawn on the map), and E is a set of *edges* (roads and sidewalks). We do not consider multigraphs, and thus, $E \subseteq V^2$. $|V|$ is the *order* of a graph, and $|E|$ is its *size*. If all edges of a graph satisfy the property $(u, v) \in E \Leftrightarrow (v, u) \in E$, the graph is said to be *undirected* and *directed* otherwise. *Complete* graphs are those having all possible edges, that is, $E = V^2$.

A *walk* is an alternating sequence of vertices and edges $(v_0, e_1, v_1, \ldots, v_n)$, such that $v_i \in V$ and $e_i = (v_i, v_{i+1}) \in E$. A *trail* is a walk with distinct edges. A *path* is a walk with distinct vertices. A *Hamiltonian path* is a path containing all vertices of a graph. Similarly, a *closed walk*, *closed trail*, *cycle*, and *Hamiltonian cycle* are defined by setting $v_0 = v_n$. We occasionally use s to denote a starting vertex and t to represent a target.

Distances between vertices are determined by a non-negative *distance function* $d : E \to \mathbb{R}_{\geq 0}$. The total distance of a (closed) walk is calculated as the sum of the distances of all visited edges and, if necessary, is bounded above by $D_{\max}$.

Please recall the function $m : V \times C \to \{0, 1\}$ for categorical matching introduced in Section 2.1; its definition remains the same. The existence of an arrow between any two categories is given by an *arrow function* $a : C^2 \to \{0, 1\}$ whose value $a(c_1, c_2) = 1$ if a place matched by c_1 must precede another place matched by c_2 on a path, and 0 if this is not the case.

3.4.2 Formalization

The goal is to gradually generalize problem statements until we find a suitable one. Figure 3.18 summarizes our endeavor. Let us start with the definition of a particular *decision problem*; a proof of its NP-completeness can be found in [31].

Definition 1 (HamCycle). *Does a graph G have a Hamiltonian cycle?*

Even though this problem is NP-complete, and the existence of an efficient algorithm solving it is unlikely, there is no way to express the distance of a cycle, among other limitations. Therefore, we propose the following extension: the well-known Traveling Salesman Problem (TSP).

Definition 2 (TSP). *How long is the shortest Hamiltonian cycle in a complete graph G with a distance function d?*

A careful reader might argue that there is a decision variant of the TSP. One interesting aspect of Definition 2 is that it seeks the shortest cycle, representing the best possible or optimal solution for the given settings. Moreover, the HamCycle is reducible to the TSP by setting values of the distance function to 1 for all $e \in E$, and 2 otherwise. Then, we ask whether the shortest Hamiltonian cycle in G has the length of $|V|$. This shows the NP-hardness of the TSP.

All route search formalizations targeted in this thesis, including the TSP, are *NP-optimization problems* defined over possibly large but finite graphs. In principle, we could solve them by enumerating all configurations. Unfortunately, this idea becomes impractical for sufficiently large instances. In the TSP, the number of cycles grows factorially with the order of G.

Considering the current state of knowledge, we cannot hope to obtain an optimal solution in a reasonable time, even for the simplest problems, leaving aside categories and arrows. Sometimes, a *near-optimal* solution is enough if computed efficiently.

Definition 3 (α-approximation algorithm [32])**.** *An* α-approximation algorithm *for an NP-optimization problem is a polynomial-time algorithm that for all instances of the problem produces a solution whose value is within a factor of* α *of the value of an optimal solution.*

Sahni and Gonzalez [33] showed that an α-approximation for the TSP, with no further restrictions on the distance function, implies the solvability of the HamCycle in polynomial time. The existence of such an algorithm is unlikely.

The reason HamCycle could be solved is that the distance function does not always respect the triangle inequality $\forall u, v, w \in V : d(u, w) \leq d(u, v) + d(v, w)$. Together with this inequality, d becomes a *quasimetric*. In practice, one might encounter two cases of the TSP equipped with such a function: the Asymmetric TSP (ATSP) and the Symmetric TSP (STSP).

The asymmetric alternative covers everyday needs better because transportation networks typically disregard symmetry. However, this generalization appears particularly difficult to deal with. Only recently, Traub and Vygen [34] designed a highly non-trivial $(22 + \varepsilon)$-approximation algorithm based on the results of many other researchers.

Fortunately, STSP is approximable by polynomial-time heuristics with a much better constant ratio. Christofides [35] employed a composition of minimum spanning tree and minimum-weight perfect matching to devise a 3/2-approximation. Since then, only subtle progress has been made towards achieving a lower factor.

Till now, we have been discussing the TSP that aims to obtain a Hamiltonian cycle, but certainly, finding a *path* would reflect Requirement **F12** better. Again, the best approximation algorithms achieve ratios of $43 + \varepsilon$ for the ATSP due to Traub and Vygen [34] and 3/2 for the STSP due to Zenklusen [36].

Taking into account the fundamental difficulty of the ATSP, we conclude that the STSP is the most appropriate formalization to consider when selecting or designing more capable algorithms. This was also one of the reasons we support only the walking mode, since sidewalks are assumed to be passable in both directions, ensuring that the distance function of a problem instance is symmetric.

In the remainder, we discuss more relevant problem statements without details on known approximation ratios. Instead, we leave pointers for further exploration.

Let us proceed with an extension of the TSP, the Orienteering Problem (OP). Its input also includes a starting point s and destination t, non-negative function $score : V \to \mathbb{R}_{\geq 0}$, and upper bound $D_{\max}$. An instance of the OP seeks for a path from s to t with a length no longer than $D_{\max}$ that visits a subset of V, maximizing the total collected score [37].

Although we have addressed the distance, it is not clear how to map a category to the set of real numbers. The *City Trip Planner* used a variant of the OP, the Tourist Trip Design Problem (TTDP), and accomplished this task by converting user input into the sum of type, category, and keyword search scores [19].

The OP enables us to reason about places in terms of *relevancy*, but our categories match them *precisely*. The Generalized TSP (GTSP) offers a toolset that captures the discrete nature of keywords and attribute filters.

Definition 4 (GTSP [38])**.** *Given a complete graph G whose vertices are divided into a given number of mutually exclusive clusters, denoted by $L_1, L_2, \ldots, L_k$, and a distance function d, the GTSP searches for the shortest cycle visiting a collection of vertices with the property that at least one vertex from each cluster is visited.*

Li et al. [39] proposed the Trip Planning Query (TPQ), a path version of the GTSP with a symmetric distance function. Cao et al. [40] studied approximation algorithms and heuristics for the Keyword-Aware Optimal Route Query (KOR), a modification of the GTSP in which the total distance is bounded above. In both TPQ and KOR, clusters are expressed by means of distinct keywords.

It is worth mentioning that our places could satisfy more than one category at once; a simple reduction to an instance of the GTSP is to define V as a set of tuples (place, category).

Finally, we need to provide users with a way to *order* clusters. The Precedence Constrained GTSP (PCGTSP) [38] incorporates arrows and enforces specific arrangements using directed acyclic graphs. Similar to the GTSP, there are less abstract interpretations of the same problem focused on real-world applications. In particular, the Multi-Rule Partial Sequenced Route (MRPSR) [41] and Optimal Sequenced Route (OSR) [42] cover *partial* and *linear* orders, respectively.

In this thesis, we consider the GTSP and PCGTSP as baseline formalizations for unordered and ordered instances, respectively, around which all algorithms and heuristics should evolve. Both lack a mechanism for limiting the distance of a path. As we will see in Section 3.4.3, calculating a *network distance* is a time and resource consuming process, leading us to opt for estimation. Hence, satisfying this constraint at all costs becomes meaningless.

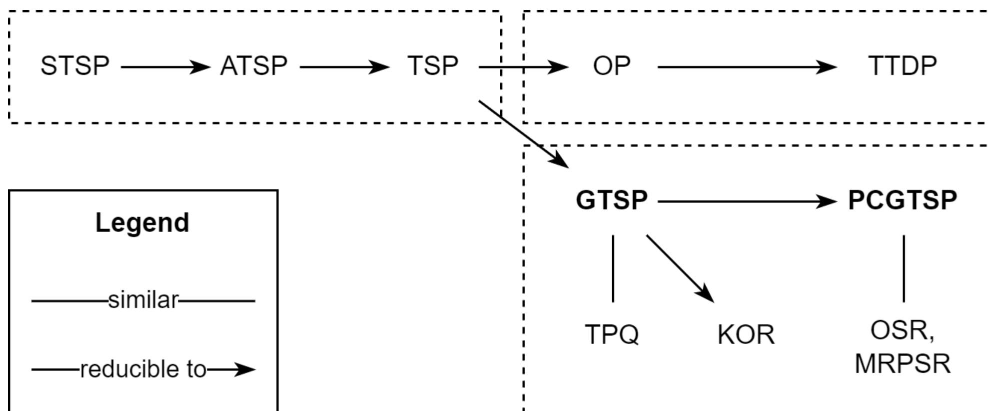

Figure 3.18: An overview of the related NP-optimization problems.

3.4.3 Finding a sequence

Two polynomial-time heuristics *without* performance guarantees are described at the beginning of this section, one for the GTSP and another for the PCGTSP. Relaxed requirements on optimality are aligned with **N8**. Finally, the geometric justification of the place retrieval process is given, and two realizations of a distance function are considered.

Unordered categories

To solve unconstrained instances, we utilized Infrequent-First Heuristic (IFH) designed by Kanza et al. [43]. This iterative procedure unfolds as follows.

Algorithm 1 Infrequent-First Heuristic.

---

function IFHEURISTIC($V, C = \{c_1, \ldots, c_k\}, m, d, s, t$)
  $\mathcal{S} \leftarrow [s, t]$ ▷ *Initialize a sequence*
  for $c_i \in C$ **do**
    $M_i \leftarrow \{v : v \in V \wedge m(v, c_i) = 1\}$ ▷ *Find matching places*
  end for
  $\mathcal{M} \leftarrow [M_1, \ldots, M_k]$ sorted by cardinality in ascending order
  for all M of $\mathcal{M}$ in a given order **do**
    $v, i \leftarrow$ SELECTBEST($\mathcal{S}, M, d$)
    $\mathcal{S} \leftarrow$ INSERTAT($\mathcal{S}, v, i$)
  end for
  return $\mathcal{S}$
end function

The SELECTBEST determines a place in the given set M and an index within the current sequence $\mathcal{S}$ that, if inserted, minimizes the impact on the length. The selection criterion is illustrated in Figure 3.19, where the blue dot is the candidate place, two blue edges with "+" contribute to the total distance, and the one with "–" is removed from the configuration.

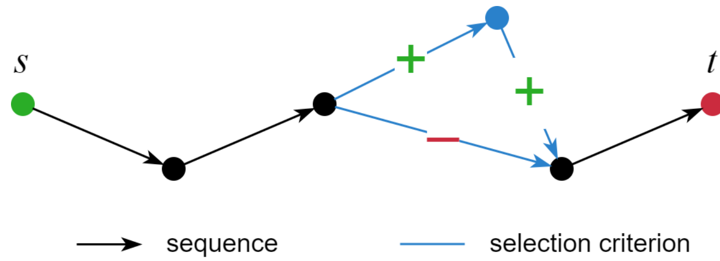

Figure 3.19: Infrequent-First Heuristic, selection criterion.

The time complexity of this algorithm is dominated by the **for**-cycle over $\mathcal{M}$ resulting in $O(|V||C|^2)$.

After finding a feasible route, we could try to improve the sequence via local search. One of the most popular and simple heuristics is called 2-Opt. For every pair of edges, $\{u_1, v_1\}$ and $\{u_2, v_2\}$, we redefine them as $\{u_1, u_2\}$ and $\{v_1, v_2\}$ and accept change if the total length decreases. The procedure is repeated as long as such a pair exists.

Even though 2-Opt has a very concise description, there are graphs proposed by Englert, Röglin, and Vöcking [44] on which it can make an exponential number of steps. Despite this, we include it as a subroutine because our paths are typically short.

Ordered categories

Kanza et al. [45] developed the Oriented Greedy Heuristic (OGH) to address precedence constraints, the implementation details are shown in Algorithm 2.

Algorithm 2 Oriented Greedy Heuristic.

function OGHEURISTIC($V, C = \{c_1, \ldots, c_k\}, m, d, a, s, t$)
  $\mathcal{S} \leftarrow [s, t]$ ▷ *Initialize a sequence*
  for $c_i \in C$ **do**
    $M_i \leftarrow \{(v, i) : v \in V \wedge m(v, c_i) = 1\}$ ▷ *Find matching tuples*
  end for
  $\mathcal{M} \leftarrow \{M_1, \ldots, M_k\}$
  while $\mathcal{M} \neq \emptyset$ **do**
    $U \leftarrow \bigcup\{M_i : M_i \in \mathcal{M} \wedge (\nexists M_j \in \mathcal{M} : a(c_j, c_i) = 1)\}$ ▷ *Find candidates*
    $(v, i') \leftarrow$ SELECTBEST($\mathcal{S}, U, d$)
    $\mathcal{M} \leftarrow \mathcal{M} \setminus \{M_{i'}\}$
    $\mathcal{S} \leftarrow$ INSERTAT($\mathcal{S}, v, |\mathcal{S}|$)
  end while
  return $\mathcal{S}$
end function

The semantics of the SELECTBEST here are slightly different than that of the corresponding function in IFH, as illustrated in Figure 3.20. The candidate vertex is always inserted between t and the one just before it. Furthermore, only the blue (+)-edges participate in decision-making. The criterion, defined as the sum of the distances, ensures that places are selected as close to the straight line connecting s and t as possible and that the sequence does not progress too fast towards the target. The blue edges with arrow tips are added to the final configuration, and the black one marked "×" is removed.

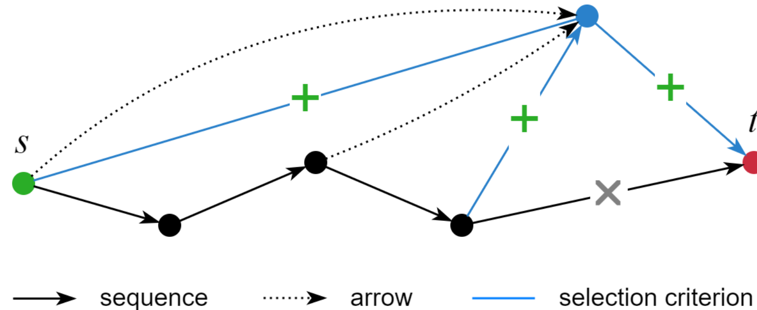

Figure 3.20: Oriented Greedy Heuristic, selection criterion.

Since each M_i can contain at most $|V|$ tuples, an iteration of the **while**-cycle has a time complexity of $O(|V||C|)$. Therefore, this heuristic runs in $O(|V||C|^2)$, which is as fast as IFH.

Bounding shape

The next aspect directly affecting the search speed is the cardinality of V. We should agree that places lying too far from s or t are of little interest to us due to an upper bound on the length of a path and should be eliminated before entering an algorithm. However, it is not entirely clear how to measure distances.

First, we need an abstract geometric shape to model the surface of the Earth. The WGS84[23] *geographic* coordinate system uses an *oblate spheroid*, as specified in the EPSG public registry. Calculations on an ellipsoidal body achieve higher precision but are more computationally demanding. As a result, most web maps, including OSM, have adopted the Web Mercator[24] as a baseline *projected* coordinate system for translating WGS84 points into two-dimensional space, assuming they are drawn on a *sphere*. Thus, we make a similar assumption.

Once the geometry is known, a straight line distance between two given points, denoted by d_H, can be expressed using the *Haversine formula*. As an alternative, we could also query the OSRM to determine a network distance d_N. Clearly, the relation $d_H \leq d_N$ holds for all pairs of vertices.

In order to filter out points lying too far from s or t and eventually reduce $|V|$, we should retrieve only those within some *bounding shape*. The most appropriate approach is to utilize the mathematical properties of an *ellipse*. We set foci F_1 and F_2, major axis a, and focal distance c as follows, provided that $d_H(s,t) < D_{\max}$:

$$F_1 = s, \qquad F_2 = t, \qquad a = \frac{D_{\max}}{2}, \qquad c = \frac{d_H(s,t)}{2}.$$

As $d_H(s,v) + d_H(v,t) \leq D_{\max}$ holds for any $v \in V$ inside the ellipse or on the boundary, we may safely skip points outside, especially if final distances are determined by d_N.

Our use cases assume that a starting point and destination need not lie on a horizontal line, and a basic ellipse almost always requires rotation and translation. The major obstacle related to drawing figures on a sphere is that its *parallels* have different lengths. The following subroutine handles transformation properly.

1. Calculate the coordinates of the midpoint o between s and t (in *degrees*).

2. Define an ellipse e with its center at the origin $(0,0)$, F_1 at $(-c,0)$, and F_2 at $(c,0)$, where c is measured in *meters*.

3. Approximate e by a polygon p whose points lie precisely on the boundary.

4. Rotate p about the origin so that it has the same orientation as the straight line connecting actual s and t.

5. Transform x- and y-components of points of p to *degrees* with respect to the parallel at the latitude of o, which defines the cost of one radian along the x-axis. The cost of a radian along the y-axis remains constant.

6. Translate p so that its center coincides with o.

[23]https://epsg.io/4326: WGS84 – World Geodetic System 1984, used in GPS
[24]https://epsg.io/3857: WGS84 / Pseudo-Mercator – Spherical Mercator

Please note that this is only an approximation that works well on small scales. An example of a rotated and translated ellipse is presented in Figure 3.21.

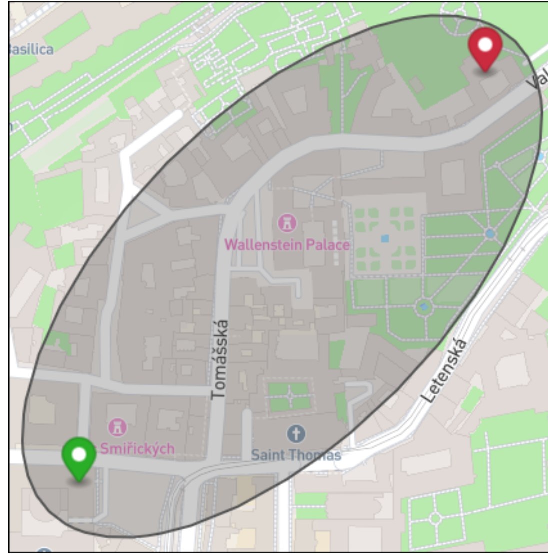

Figure 3.21: Geojson.io: rotated and translated ellipse drawn on the map.

Distance function

If we carefully review the bodies of the aforementioned heuristics, neither IFH nor OGH explicitly bounds the total distance of a constructed sequence. This behavior is intentional, and the reason is that obtaining the length of the shortest path between two arbitrary points is a costly operation.

The OSRM has `route` and `table` services suitable for our needs. The former focuses on finding a full representation of the shortest path for a given sequence of points, while the latter performs bulk distance calculations and can be used to fulfill the second step in the following scenario.

1. Fetch the $|V|$ most *relevant* places within a bounding ellipse.
2. Request an $\mathbb{R}^{|V|\times|V|}$ matrix containing distances between all pairs of places.
3. Use this matrix as a distance function, $d = d_N$.

Recall that category matching is a boolean-valued function, implying that places cannot be sorted by relevancy. If we were to accept all matches, large values of $|V|$ would inevitably increase CPU usage and memory consumption.

The only viable option, which does not require additional effort regarding data preparation, is to set $d = d_H$, let an algorithm construct *any* sequence preserving arrows, find the shortest path for this sequence using the `route` service, and discard those with distances larger than $D_{\max}$.

4. Implementation

High-level architectural decisions and a brief overview of the technology stack have already been covered in the previous chapter. This part of the thesis clarifies notable implementation details and the reasons behind them.

4.1 Prerequisites

To acquire the tools necessary for setting up the development environment, preparing the dataset, and writing and testing source code, follow the steps outlined in Attachment A.2. The source code of the application is published at

`https://github.com/zhukovdm/smartwalk`.

4.2 Single-page application

The frontend is a *single-page application* written in modern React, with *components* and *hooks* serving as its basic building blocks. Components are functions that return renderable structures, while hooks are functions that provide limited access to React global state and other features.

The corresponding source code is located in the `./app/frontend/` folder.

4.2.1 Toolchain

To streamline the configuration process, *SmartWalk* was bootstrapped with Create React App (CRA)[1]. The initial template is a small, production-ready application that includes a transpiler, development server with hot reload, bundler, and various other tools. It is worth noting that CRA is no longer recommended by the React team, as their focus has shifted towards React-based frameworks.

4.2.2 Visual components

Branding is unnecessary for *SmartWalk* at this phase of development. Hence, Material UI[2] has been added as an external dependency. This advanced library consists of visually appealing components designed for desktop and mobile devices and the collection of icons. Moreover, navigation `Drawer` and `Autocomplete` with asynchronous data fetching were found particularly useful and less common across other popular React-based libraries.

The only functionality we missed in Material UI was a special handling for `img` HTML elements. Since the detailed view of a place might point to a high-quality image, it could take time for a browser to load it. We have employed mui-image[3] to indicate that the picture is loading by displaying a spinner that reserves space and prevents unexpected content shifts.

[1] https://create-react-app.dev/
[2] https://mui.com/material-ui/
[3] https://github.com/benmneb/mui-image

4.2.3 Client-side routing

Navigation between panels is realized by means of the React Router[4] library. The `BrowserRouter` wraps the entire application and makes the desired functionality accessible within the nested components. The `Routes` component acts as a selector, ensuring the proper panel is rendered based on the current path.

The code snippet below is a simplified showcase from the codebase; the content of a panel should appear within the `Drawer`, assuming that the `PanelDrawer` is a child of a `BrowserRouter`.

```
function PanelDrawer() {
  return (
    <Drawer>
      <Routes>
        <Route path={"/search/routes"} element={<SearchRoutesPanel />} />
        ... more Route instances follow ...
      </Routes>
    </Drawer>
  );
}
```

A total of *12* distinct paths were defined to accommodate the panels proposed in Section 3.1.

- `/search` + `/routes`, `/places`, `/direcs` (search panels)
- `/result` + `/routes`, `/places`, `/direcs` (result panels)
- `/entity/places/{smartId}` (detailed view of a place)
- `/favorites` (view into the private storage)
- `/viewer` + `/route`, `/place`, `/direc` (entity viewer)
- `/session/solid` (Solid session)

Hooks `useNavigate` and `useParams` are used to navigate between panels and extract `smartId` from the path, respectively.

4.2.4 State management

Suppose a user has configured a couple of categories with many attributes for a route search. Before submitting the query, they decide to check whether a route with similar preconditions exists. The panel is then switched to the "Favorites" view and back to the request form.

Changing a path also entails removing (or unmounting) elements associated with the previous panel from the DOM tree of the browser. If categories were kept in the component's local state, they would disappear just before the new content is rendered. We need a systematic approach to avoid this undesired behavior so that the state of a panel is restored as long as a hard reset or page reload has not been performed.

[4] https://reactrouter.com/

Redux Toolkit[5] is a library for modeling and managing an application state composed of *slices*. *Reducers* are functions that, given the previous state and an input value, unambiguously define how to transition to the next state. Properties of a slice can be modified directly due to the Immer[6] library used in the background; the slice is an immutable proxy. A simple example is shown in the code snippet below.

```
const slice = createSlice({
  name: "slice",
  initialState: { item: 0 },
  reducers: {
    setItem: (state, action) => { state.item = action.payload; }
  }
});
```

A distinct slice is implemented for each panel, see files ending with `Slice.ts` in the `./src/features/` folder.

The state is accessed and manipulated through custom hooks `useAppDispatch` and `useAppSelector` as follows. A new value of `item` appears in the `h1` upon each increment automatically.

```
function Component() {
  const dispatch = useAppDispatch();
  const { item } = useAppSelector((state) => state);
  return (
    <h1>{item}</h1>
    <button onClick={() => { dispatch(setItem(item + 1)); }}>
      Increment
    </button>
  );
}
```

Unfortunately, Redux Toolkit is not recommended for keeping non-serializable data[7], such as class instances. For this reason, we store `Map`, `Storage`, and cached backend responses (keyword suggestions based on prefixes and full representations of retrieved places) using the standard Context API[8]. In principle, only this container would be enough to achieve the same functionality. Nonetheless, there are two hidden drawbacks: components cannot subscribe for a part of the value, and we would need to write custom logic for modifying nested objects.

4.2.5 Device storage

The `DeviceStorage` class is an implementation of the `Storage` abstraction for storing entities on the user's device. The open question is which technology we should utilize. Saving files within the file system of an operating system brings additional overhead for users, as their addresses must be determined manually. Alternatively, we can employ one of the storages that come with a web browser.

[5]https://redux-toolkit.js.org/

[6]https://immerjs.github.io/immer/

[7]https://redux.js.org/faq/organizing-state

[8]https://react.dev/learn/passing-data-deeply-with-context

The `DeviceStorage` is a wrapper over IndexedDB [46], a standardized database for storing (un-)structured data with support for transactions and indexing. We distinguish the following two consecutive phases of its lifecycle.

1. Initialization of the database via creating three *object stores* for each entity type: `routes`, `places`, and `direcs`. If it fails, the instance internally falls back to the `InmemStorage` and informs the user via the message box at the top of the "Favorites" panel, similar to the one in Figure 3.14.

2. **C**reate, **R**etrieve, **U**pdate, and **D**elete operations performed on the entities. Failed attempts raise exceptions that are eventually reported to the user.

One disadvantage that many programmers may encounter when working with IndexedDB is that its API does not support modern *promises* but instead accepts event handlers. The `DeviceStorage` injects `resolve` and `reject` into the bodies of the corresponding callbacks.

4.2.6 Solid pod session

The `SolidStorage` class maintains a connection between the frontend and a Solid pod. This class implements the same `Storage` interface; using it does not make any difference from the programmer's perspective compared to other storages. However, the lifecycle of a Solid pod session is more involved and includes *four* steps.

4.2.6.1 Authentication

First, the application should authenticate against a Solid server and acquire an identity. There are two mechanisms to achieve the same result: Solid-OIDC (recommended) and WebID-TLS [11, Authentication].

We do not delve deep into the subject, as our concerns are fulfilled by merely calling `login` from the solid-client-authn[9] library and waiting until the browser opens an external page with an authentication form. Once the user enters credentials and allows the application to read the identity and access pods, the browser redirects back and opens the "Solid Session" panel depicted in Figure 3.13.

4.2.6.2 Initialization

An activated pod should go through the process of initialization. Recall Section 3.2.5 where we have assumed that all data will be stored on a certain address. Thus, the procedure performs the following actions.

1. Attempt to create a destination folder for each entity type. Failures are not reported, and existing resources are not affected.

2. Retrieve the folder as a `SolidDataset` (prove the existence).

3. Verify that this dataset is of type `BasicContainer`.

[9]https://github.com/inrupt/solid-client-authn

4.2.6.3 CRUD operations

Entities are created and altered using primitives from the solid-client[10] library, such as asynchronous `overwriteFile`, `getFile`, and `deleteFile`. Data is stored in the form of binary `Blob` objects with the `application/json` media type.

4.2.6.4 Logging out

After a user clicks the "Log out" button, `logout` is called on the Solid default session, application's state resets, and the storage switches to a `DeviceStorage` instance.

4.2.7 Custom hooks

Standard React hooks were found to be insufficient to cover the needs of our application. The following custom hooks were implemented to hide the complexity of stateful logic.

`useFavorites` loads entities from the private storage into the Redux-based state container so that reading queries are resolved locally.

`useStoredSmarts` constructs a set of places with defined `smartId` that appear in the private storage.

`usePlaces` merges places in the result of a search and those in the private storage (with `name` replacement).

`useSmartPlace` retrieves the cached representation of a place for a given `smartId` or fetches the object from the backend in case of a cache miss.

`useKeywordAdvice` retrieves the cached keyword advice for a given prefix or calls the backend API, similar to the previous hook.

There are a couple more supporting hooks. See source files whose names end with `Hooks.ts` in the `./src/features/` folder.

4.2.8 Calling API

The `SmartWalkAPI` abstraction is realized by the collection of functions defined in the `smartwalk.ts` file. The communication protocol is hidden in the bodies of these functions. A query could resolve with a value or fail with an exception.

4.2.9 Marker clustering

The map is one of the essential visual parts of the application, without which navigation through the road network would be hard to imagine. We use Leaflet[11], a minimalistic library, for managing map state, loading tiles, and drawing markers and geometries. The basic functionality of this library is extended via plugins.

[10] https://github.com/inrupt/solid-client-js
[11] https://leafletjs.com/

One interesting observation about *SmartWalk* is that, according to Requirement **F4**, place search queries are bounded only by a circle. There is a good chance that a response might return thousands of places. Since Leaflet mounts each pin as a separate node to the DOM tree, rendering a large number of elements might cause the browser to hang or even crash.

The current implementation imports the markercluster[12] plugin as an external dependency to cluster markers on the client side in the main thread. We should admit that this solution is still not ideal; the user interface might hang for a while with large inputs. There are two possible approaches to improve clustering.

- Render clusters by a dedicated Web Worker on the client side. For example, supercluster[13] is a library independent of Leaflet with suitable primitives.
- Render clusters on the server side. It is certainly an interesting option that simplifies the programming of new frontends. However, we might need to introduce additional containers into our architecture to facilitate this type of spatial query.

As we encountered performance issues with the clustering functionality later during implementation, improvements are deferred for future development.

4.2.10 JSON-LD representation

Earlier, we stated the principles of *Linked Data* and their positive impact on data discoverability and interoperability. This section is dedicated specifically to Requirement **F23**.

The detailed view of any place from the public storage includes a JSON-LD object with the following properties: name, longitude, latitude, keywords, and links to the original entities with the `owl:sameAs` semantics. These properties are guaranteed to be present due to the way data is prepared.

Since the frontend is a single-page application, and representations are fetched on demand by `useSmartPlace`, it is obvious that this object should be generated dynamically. `EntityPlaceHelmet`, a custom component, injects a `script` element with the `application/ld+json` media type into the `head` of the page once the hook succeeds in retrieving a record by `smartId`.

One might argue that the resulting JSON-LD object has only a small number of properties, and the potential of knowledge graphs is underutilized. We neglect RDF because the majority of records originate from the OSM dataset, where the meaning of keys is not defined.

Another disadvantage of our approach is that JavaScript code should be executed to construct the JSON-LD representation. This directly impacts the ability of intelligent agents to understand the content. Perhaps a better approach would be to incorporate the `@context` into a backend response.

Thus, the current implementation offers rather limited capabilities regarding data friendliness. These concerns and potential improvements in the way data is stored in a Solid pod are mentioned as directions for future research.

[12]https://github.com/Leaflet/Leaflet.markercluster
[13]https://github.com/mapbox/supercluster

4.3 Web API application

The backend is a .NET solution that consists of the following *four* projects.

`SmartWalk.Core` defines entities, algorithms, solvers, and core-level abstract interfaces used across the application.

`SmartWalk.Application` prescribes the shape of query objects, provides domain-level input parsers and validators, along with separate handlers for each type of supported queries.

`SmartWalk.Infrastructure` implements gateways to the containers discussed in Section 3.2.2.

`SmartWalk.Api` serves as an entry point to the application with HTTP endpoints, middlewares, and controllers.

The source code is located in the `./app/backend/` folder. The toolchain required for building, running, testing, and publishing the application is simplified to just one command-line utility named `dotnet`.

4.3.1 HTTP endpoints

We define *five* HTTP-based endpoints with the supported `application/json` media type to facilitate the needs of the frontend. Words written in an emphasized typewriter ***`font`*** symbolize query parameters.

`GET /api/advice/keywords`

Obtain the ***`count`*** most relevant keywords starting with ***`prefix`*** and their attribute bounds.

`GET /api/search` + `/routes`, `/places`, `/direcs`

Three endpoints for handling search queries, with the only ***`query`*** parameter set to a serialized and percent-encoded JSON object.

`GET /api/entity/places/{smartId}`

Parameterless request fetching the full representation of a place by `smartId`.

The first and last endpoints are trivial. However, there are multiple options for passing complex nested objects via the Hypertext Transfer Protocol (HTTP).

The first idea that we may come up with is to place the object in the request's body. The `POST` method accepts bodies, but such a request could introduce side effects and alter the state of the server, ruling out caching. `GET` requests with bodies are less supported by standard libraries and are harder to send and cache. Nonetheless, the well-known search engine Elasticsearch defines APIs that accept this kind of request[14] to perform search queries.

Thus, we opted for the solution adopted by Wikidata and DBPedia, where a percent-encoded SPARQL query is expected in the ***`query`*** parameter.

[14] https://www.elastic.co/guide/en/elasticsearch/reference/8.11/api-conventions.html

Our API is documented using the Swashbuckle[15] library, a toolset compatible with the OpenAPI[16] specification, and published at

`https://app.swaggerhub.com/apis/zhukovdm/smartwalk/`.

4.3.2 Controllers

In ASP.NET Core, every HTTP request goes through a series of *middleware components*, collectively forming a pipeline. A middleware performs an intended action on the input and invokes the next middleware or short-circuits the request.

As part of the pipeline, the framework routes a request and calls an appropriate method on a custom `TController`, distinct for each HTTP endpoint defined in Section 4.3.1. All implemented controllers have internal structures similar to the one shown below.

```
class TController : ControllerBase {
  public async Task<ActionResult<T>> Action([FromQuery] TRequest request) {
    if (!new TValidator(...).Validate(request)) {
      return new TResponder().Invalid();
    }
    try {
      return new TResponder().Respond(await new THandler().Handle(request));
    }
    catch (...) {
      return new TResponder().Failure();
    }
  }
}
```

Request-level and domain-level validations occur before a query object reaches the corresponding handler. The former is a standard middleware hidden in the pipeline before the `TController`, parsing the content of a request into a schema-constrained JSON object. The latter, executed by the `TValidator`, checks if the parsed object meets constraints imposed by the conceptual model. For example, ensuring the acyclicity of an arrow configuration is a domain-level concern.

A malfunctioning database or routing engine may lead to an exception in the `THandler`. The **catch**-block gracefully manages unexpected failures.

Errors are communicated to the frontend in the form of JSON objects, with the corresponding HTTP status set by the `TResponder`. The reason for a failure is included, as shown in the code snippet below.

```
{
  "type": "https://tools.ietf.org/html/rfc7231#section-6.5.1",
  "title": "One or more validation errors occurred.",
  "status": 400,
  "traceId": "...",
  "errors": {
    "query": [ "Cycle 0 → 1 → 2 → 0 detected." ]
  }
}
```

[15] https://github.com/domaindrivendev/Swashbuckle.AspNetCore

[16] https://www.openapis.org/

4.3.3 Gateways

As a result of Section 3.2, we identified *four* abstractions vital for handling all types of queries. Hence, their implementations are proposed based on the selected technology stack.

The `TrieKeywordAdvicer` is a simple wrapper over PruningRadixTrie for finding the k most relevant keywords. Please note that relevancy is measured based on keyword occurrence; a higher value indicates that the number of places with this keyword is greater compared to other keywords. An instance of this abstraction is populated with precalculated data from the database upon application start.

The `MongoEntityStore` and `MongoEntityIndex` are gateways to the database, each with a distinct set of methods discussed in Section 3.2.3. Data is retrieved using the `MongoClient` from the MongoDB.Driver[17] library, which internally maintains a thread-safe[18] connection pool.

Data is fetched from an instance of OSRM through simple HTTP `GET` requests. The `OsrmRoutingEngine` is a stateless component that constructs a destination URL from the input, calls the API [47], and converts the response into an output object.

Infrastructural gateways are added to the dependency injection container as singletons during application start in the source file `Program.cs` and then injected into a controller's constructor.

```
builder.Services.AddSingleton<IContext>(new TContext()
{
  Gateway = new Gateway()
});

class TController : ControllerBase {
  public TController(IContext ctx) {
    this.ctx = ctx;
  }
}
```

4.3.4 Handlers

The purpose of handlers is to calculate results given the current context and valid user input. There is a one-to-one correspondence between HTTP endpoints, controllers, and handlers. One might observe that all of them, with the exception of the `SearchRoutesHandler`, have trivial implementations. Let us focus on this particular example.

To calculate routes, the `SearchRoutesHandler` performs the following steps.

1. Retrieve places matched by at least one category and lie within a bounding ellipse.

2. Given places, create an instance of the `HaversineDistanceFunction`.

[17] https://www.nuget.org/packages/MongoDB.Driver

[18] https://www.mongodb.com/docs/drivers/csharp/v2.21/faq/#std-label-csharp-faq-connection-pool

3. Create an instance of the `SolverFactory` that accepts a distance function, arrow configuration, source, and target. Since the input parameters are the same for all routes, only one instance of the factory is necessary.

4. Repeat unless the time has expired or there are no more places left.

   (a) Calculate a sequence of places using a solver instance provided by the factory. The `FloatSolver` solves the unconstrained variant, while the `ArrowSolver` is designed for the PCGTSP.

   (b) Find the shortest path connecting the points of this sequence.

   (c) Include this path into the result if its distance is less than or equal to the upper bound; otherwise, disregard it.

   (d) Filter out places that have appeared in the sequence.

4.4 Querying with MongoDB

Places and keywords are stored in MongoDB, a document-oriented NoSQL database with support for horizontal scaling and geospatial queries [48]. This section addresses two concerns: indexes for achieving adequate performance and the query language for modeling attribute filters.

4.4.1 Indexes

In NoSQL databases, indexes might be expensive due to possible sharding or the size of a collection. Therefore, only the `2dsphere` spatial index applied to the mandatory `location` field for each place is actively utilized.

The indexed field is required to hold a GeoJSON object or a legacy coordinate pair[19]. Since GeoJSON points contain arrays of coordinates, we adopt the latter approach, setting longitude to `lon` and latitude to `lat`. Using explicit fields makes them easier to access and improves the readability of generated documentation.

There are a few more key-based indexes, but they are not interesting from an implementation perspective.

4.4.2 Operators

We claimed that MongoDB has the capability to express all types of attribute filters. The following operators of the MongoDB Query Language are used exactly for this reason.

- `$exists` checks whether an attribute is defined on an object.

- `$eq` compares a boolean-valued attribute with a given value.

- `$gte` and `$lte` bound numeric attributes above and below, respectively.

- `$in` matches strings using a regular expression. When applied to an array, the same operator checks whether an element is present.

[19]https://www.mongodb.com/docs/v4.4/core/2dsphere/#2dsphere-indexed-field-restrictions

- Individual filters are concatenated by the operator `$and`.

The geospatial operators listed below are utilized to perform search queries.

- `$nearSphere` retrieves a set of places lying within a bounding circle. The set is *ordered* by the distance from the center.
- `$geoWithin` returns places within an arbitrary polygon, with no assumed ordering.

4.5 Data pipelines

Besides the application source code, there are small task-oriented programs in the `./data/` folder to carry out the data preparation phase outlined in Section 3.3.

`taginfo/` loads key statistics from Taginfo into key-specific `.json` files.

`osm/` combines information stored in Taginfo files, OSM binary files, and fetched from Overpass API to create new places or update existing ones.

`wikidata-create/` creates simple stubs for places that do not exist yet.

`wikidata-enrich/` updates the current dataset with the latest information from the Wikidata knowledge graph.

`dbpedia/` does the same action as `wikidata-enrich/` but for DBPedia.

`advice/` collects statistics about keywords and attributes across the dataset and recreates advice items.

`dump/` dumps places and keywords into `.txt` files.

`restore/` restores place and keyword collections from `.txt` dump files.

These programs are organized using one of the following general patterns. The way they are executed is described in Attachment A.1, Administrator's guide.

The first type of program iterates over entities from a source, processing them one by one. Records are transformed and loaded into a target individually. This approach is applied whenever there is a risk that a dataset is larger than the main memory.

Another type of program includes an explicit "**E**xtract, **T**ransform, and **L**oad" (ETL) pipeline in its main function. It loads the dataset into the main memory, transforms raw objects, and loads them into a provided target. The code snippet below captures the internal structure of these programs.

```
{
  const e = await pipeline.e(new Source());
  const t = await pipeline.t(e);
  const l = await pipeline.l(new Target(), t);
}
```

5. Testing

This chapter is dedicated to various testing techniques applied during implementation to improve the quality of the application in multiple dimensions.

5.1 Automated testing

Numerous *unit* and *integration* (without infrastructural dependencies) tests were written for both the frontend and backend to verify that individual modules and their assemblies meet functional requirements. Life containers were replaced by failing, stalling, or ordinary mocks.

Please note that *end-to-end* tests were not conducted, mainly because interactions among parts of the system involve straightforward scenarios. If necessary, frontend integration tests may be converted to end-to-end tests using the *Playwright*[1] framework or similar.

Frontend

The frontend was tested with the Jest[2] framework and React Testing Library[3]. Test files reside in `__tests__/` folders close to the corresponding components and functions.

In total, *676* tests were implemented, covering a range of tasks from simple rendering to simulating *user actions*, such as advanced search, entity saving and removal, and navigation between panels. Test suites have the following structure:

```
// Component.test.tsx
describe("<Component />", () => {
  it("should allow users to search for routes", () => {
    const { getByRole, getByText } = render(...);
    fireEvent.click(getByRole("button", { name: "Search" }));
    expect(getByText("Found a total of")).toBeInTheDocument();
  });
  ... more tests follow ...
});
```

The `describe` function forms a test suite. Test recipes are passed as lambdas in the parameters of `it` functions. The Jest test runner discovers them automatically. HTML elements are accessed via `getByRole` and `getByText`. The `expect` function asserts whether a test condition is met. More primitives are used in the actual tests; however, their descriptions are omitted for brevity.

We estimate the test coverage to be between 60–80%, depending on one's perception. Please note that this estimation is somewhat subjective. While writing tests, we focused on black-box testing, examining large chunks of code, as well as white-box testing, targeting individual components. Since tests were written for each component separately, adding new ones is straightforward.

[1] https://playwright.dev/
[2] https://jestjs.io/
[3] https://testing-library.com/docs/react-testing-library/intro/

Because we keep the state in Redux and Context API containers, components that access them cannot be tested in isolation, and some sort of mocking is required. The `renderWithProviders` function helps to solve this issue by wrapping a component instance in standard providers. Then, rendering is customized via `props` and `options` parameters.

```
function render(props = getProps(), options = getOptions()) {
  return renderWithProviders(<Component {...props} />, options);
}
```

Furthermore, the testing library recommends[4] accessing elements of the DOM tree by roles and names. The frontend provides a reasonable level of *accessibility* so that all tested elements can be reached and identified without referring to the `data-testid` attribute.

To execute tests, navigate to the `./app/frontend/` folder and type in:

```
$ npm run tests
```

Backend

Tests are located in projects whose names end with `.Test`. The MSTest[5] test framework was our choice to describe and run individual test cases.

The layout of the backend is much simpler compared to the frontend; requests are processed by dedidated handlers in isolation. As a result, the number of tests is only *135*. We focused on modeling typical scenarios with malfunctioning infrastructure, individual handlers returning expected objects, and randomized tests for heuristics. By applying the DIP throughout the application, we were able to inject custom implementations of abstract interfaces defined by the `Core` into our tests. Test suites are expressed in the form of classes discoverable by the runner:

```
// ComponentTests.cs
[TestClass]
public class ComponentTests {
  [TestMethod]
  public void ShouldAssertNull() {
    Assert.IsNull(null);
  }
  ... more tests follow ...
}
```

Similar to the previous section, we estimate the test coverage to a value between 50–70%. Since each request invokes the corresponding pipeline that defines methods and functions to be called, overly detailed tests are unnecessary. The ability of the backend to provide a valid response is partly covered by performance tests that check whether HTTP responses contain the status code `200`.

To run tests, navigate to the `./app/backend/` folder and enter:

```
$ dotnet test
```

[4] https://testing-library.com/docs/queries/about#priority

[5] https://github.com/microsoft/testfx

5.2 Performance testing

The purpose of this testing phase is to justify that the system can meet Requirements **N13** and **N14**. Given that we have only one personal laptop, our focus is solely on the *response times* of individual requests rather than on the system's throughput.

The term "reasonably large queries" was mentioned in the requirements. We assume that only a minority of people are interested in casual walks longer than *five* kilometers.

It is also specified that waiting 1 second for a place search and 2 seconds for a route search is deemed acceptable for the majority of users. This conclusion is drawn from [49], where the author claims that a response time under 1 second ensures a user stays uninterrupted. Requests lasting between 1 and 10 seconds should indicate progress. Nonetheless, we should not overly restrict route search queries, as more time spent on calculations often leads to better results.

The parameters of the performance test environment are listed in Table 5.1. There are two more assumptions we should state explicitly:

- responses were not rendered in a web browser,
- and no network delay was included in the calculations.

The dataset for performance tests consisted of objects from the 10 largest cities of the Czech Republic, resulting in around 3×10^5 places and 817 keywords. It should be noted that the data are skewed; tourist cities often have a much higher number of places with no dependency on population.

Since we are interested only in a general trend, the results are presented in two types of graphs. The first type is a box-and-whisker diagram, whose interquartile range (IQR) is set to the distance between the lower quartile covering 25% of the dataset, and the upper quartile covering 75% of the dataset. The whiskers extend up to $1.5 \times \text{IQR}$, and measurements outside of this range are considered outliers. The second type is a simple scatter plot.

Parameter	**Value**
Machine	Lenovo ThinkPad E580
Processor	Intel(R) Core(TM) i5-8250U CPU @ 1.60GHz, 1801 Mhz 4 Core(s), 8 Logical Processor(s)
Primary memory	DDR4, 8GB, 2.40GHz
Secondary memory	LENSE30256GMSP34MEAT3TA PCIe NVMe SSD, 256GB
Operating system	Microsoft Windows 11 Home, ver. 10.0.22621
WSL version	2.0.9.0
Kernel version	5.15.133.1

Table 5.1: Parameters of the performance test environment.

The source code for all performance tests is in the `./misc/perf/` folder, along with instructions on how to run them. The dataset is included in the electronic attachment for the thesis, as described in Attachment A.5.

Let us present the results of the tests, distinct for each HTTP endpoint. Please note that the orange lines inside the boxes are *medians*.

GET /api/advice/keywords

The input for this test consists of all keyword labels and the probability mass function determined by their frequencies. A trial randomly selects a keyword and then its prefix of length between 1 and 5. An API call is issued for a fixed ***`count`*** and this prefix.

Four different values of the ***`count`*** parameter were considered, as shown along the x-axis of Figure 5.1. For each of them, 100 trials were performed.

These queries typically respond in ≤ 4 ms. Outliers took at most 6 ms.

GET /api/entity/places/{smartId}

In one trial of this test, a place identifier is randomly selected from all possibilities, and an API call is made. The script performs 100 consecutive trials.

Figure 5.2 illustrates that requests for retrieving places by `smartId` are about $1.5\times$ longer than those for keywords.

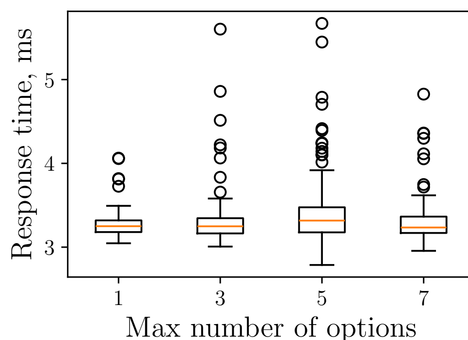

Figure 5.1: Response times for fetching keywords by prefix.

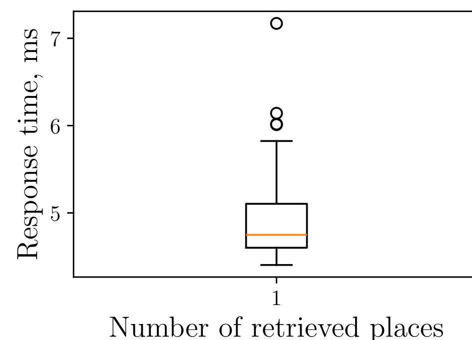

Figure 5.2: Response times for retrieving places by `smartId`.

GET /api/search/direcs

As part of the route search procedure, the `SearchRoutesHandler` class finds the shortest path connecting a sequence of places. Hence, direction search queries require a different approach because we are also interested in concrete numbers.

We want to test how the number of places visited by the shortest path and its total length affect the response time of a request. The following pseudocode captures the main ideas of the iterative testing procedure.

```
for count in [2, 3, 5, 7]:
  for city in 10 imported cities:
    for trial in [1, ..., 50]:
      locs <- SampleLocations(count, city)
      time, distance <- SearchDirecs(locs)
      result.append(count, time, distance)
```

The graphs in Figure 5.3 illustrate the results. There is a strong tendency to complete a query in ≤ 30 ms, even though distances may vary up to 200 km. The explanation for how OSRM achieves such performance is that it starts returning less detailed polygonal chains for very long paths.

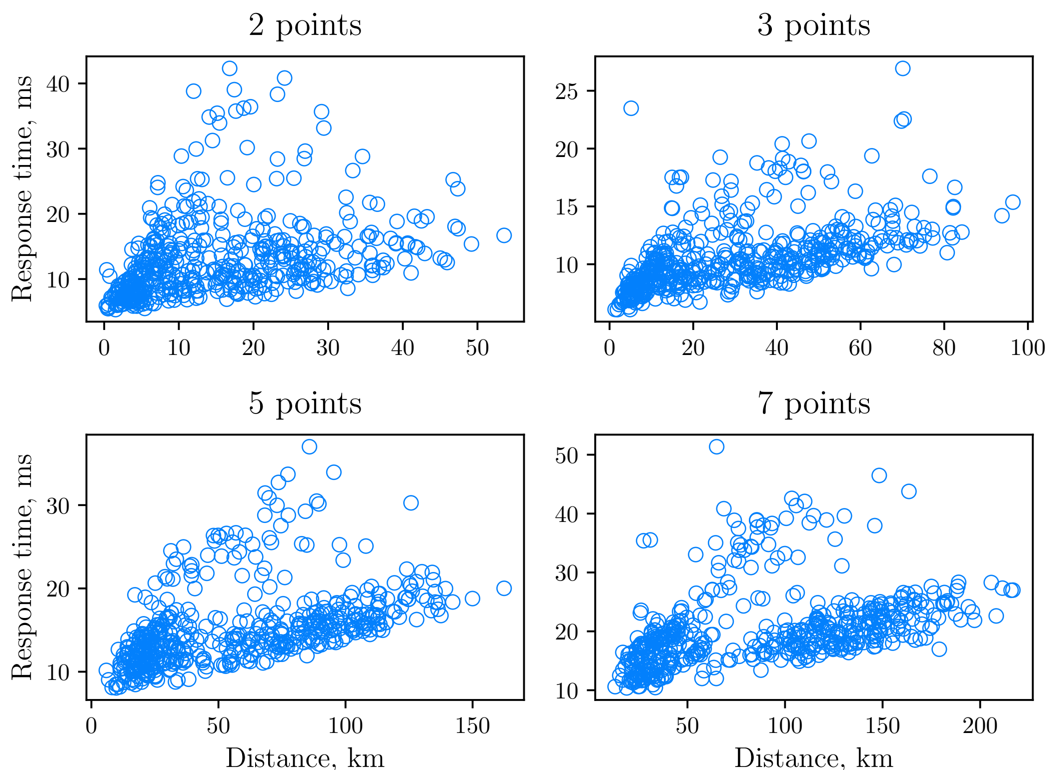

Figure 5.3: Response times for direction search queries.

GET /api/search/places

Place search requests were tested for various radii and different numbers of categories. Categories were represented by randomly selected keywords using a probability mass function similar to the first test; no attributes were configured.

When the number of categories is equal to ∞, the script sends an empty array instead of sampling, as specified in **F4**.

```
for count in [1, 2, 3, ∞]:
  for radius in [1, 3, 5, 7, 10, 15]:
    for trial in [1, ..., 50]:
      center <- SampleLocation()
      categories <- SampleKeywordsWithPmf(count)
      time <- SearchPlaces(center, radius, categories)
      result.append(count, radius, time)
```

Figure 5.4 confirms that radii up to 5 km are manageable within 1 s, but larger values may result in longer request times.

We may find the current level of performance acceptable. However, clustering delegated to the server or a Web Worker becomes especially relevant for the ability of the system to provide a seamless user experience. Hence, this extension should be prioritized in later development.

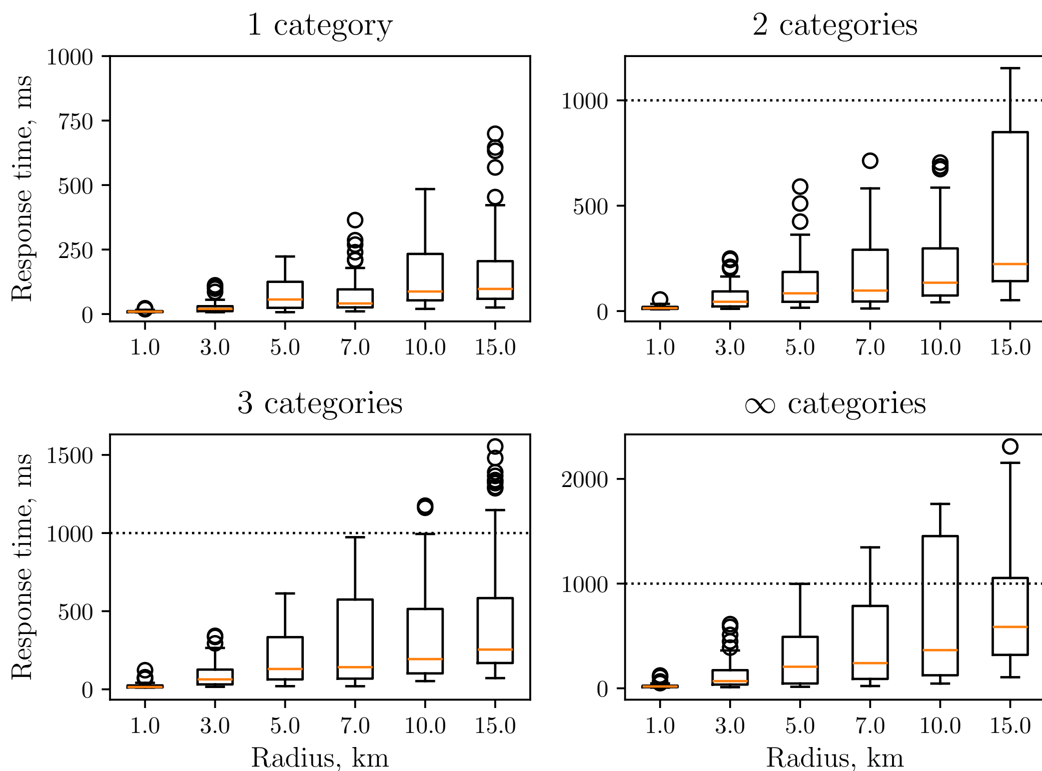

Figure 5.4: Response times for place search queries.

GET /api/search/routes

This test has a structure similar to the previous one; please refer to the code snippet below for details. The **for**-cycle iterates over distances rather than radii. The source and target are set to the same location so that the generated routes are circular. The presence of arrows does not make a difference because the IFH and OGH have the same worst-case time complexity, $O(|V||C|^2)$.

```
for count in [1, 2, 3, 5]:
  for distance in [1, 3, 6, 10, 15, 30]:
    for trial in [1, ..., 50]:
      source <- SampleLocation()
      categories <- SampleKeywordsWithPmf(count)
      time <- SearchRoutes(source, distance, categories)
      result.append(count, distance, time)
```

According to Figure 5.5, finding a route is a more computationally demanding task compared to the previous ones. For distances ≤ 6 km, all requests were answered in ≤ 2 s. Rendering is not a problem either, as only one route at a time is displayed on the map.

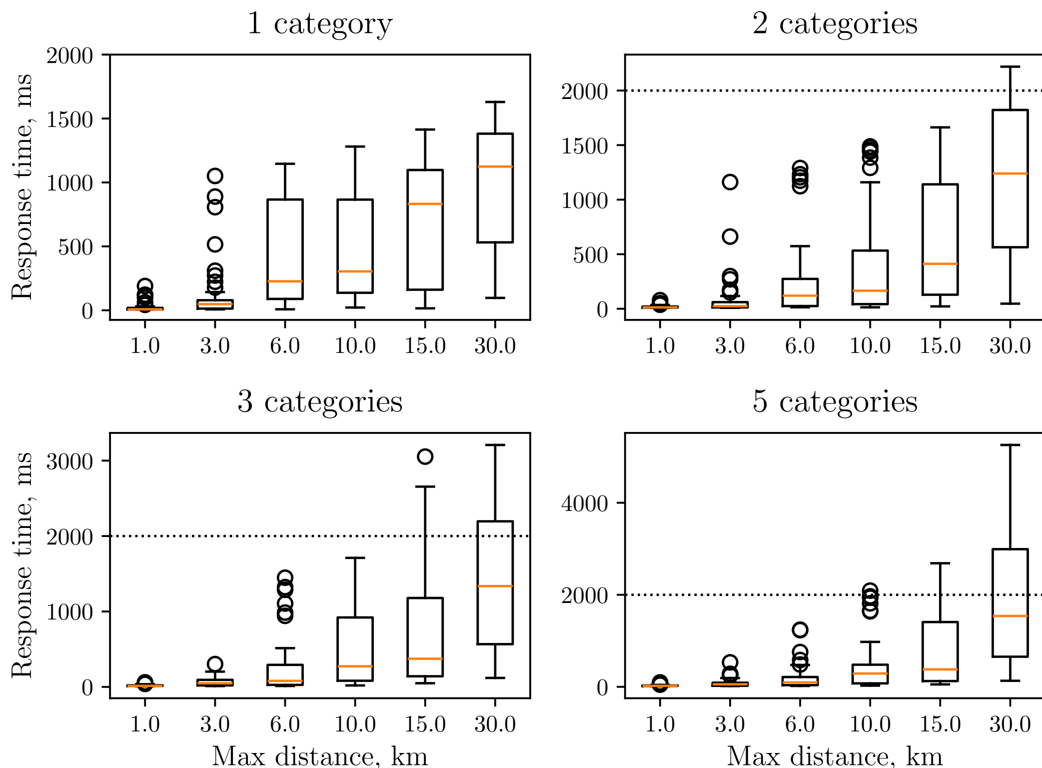

Figure 5.5: Response times for route search queries.

5.3 Usability testing

The usability of the user interface was measured using the System Usability Scale (SUS) questionnaire, as proposed by Brooke [50]. Respondents were asked to perform the following tasks in the given order on an attribute-rich dataset with places extracted from a bounding box around Prague[6].

1. Find routes between two arbitrary points on the map with a distance at most 5 km that visits a castle and museum. Save any of the found routes.
2. Create a custom place, such as your favorite attraction, home, work, etc.
3. Find all restaurants with internet access and Italian cuisine around your favorite place. Navigate to the detailed view of any found restaurant and save it.
4. Find directions connecting your favorite place and the saved restaurant. Save any direction that appears in the result.
5. Delete the entities you have created.

Before attempting to complete tasks, respondents were given a brief explanation of the application's purpose without significant showcases. If they got stuck, we guided them by directing their attention to clues left in the test descriptions. Right after the tests, the respondents were asked to provide feedback by evaluating the statements from Attachment A.4.

We successfully contacted four participants from diverse backgrounds. The first respondent was somewhat familiar with the concepts addressed in this thesis but had never independently accomplished any task. The second participant had a deep understanding of the theory behind algorithms and web development in general. The third respondent possessed practical experience in developing commercial web applications. Finally, the fourth respondent lacked specific knowledge or experience.

The results presented in Table A.1 indicate that the average SUS score for all participants is **85.00**. A score equal to 68 out of 100 is considered average, with higher scores indicating a more intuitive and usable application [51].

Below, we state several valuable conclusions drawn from the survey evaluation and follow-up discussions.

1. The user interface was easy to learn but seemed cumbersome at first impression. Several participants expressed concerns about the structure of the dialog for adding a category as being too long or ill-aligned.
2. The conceptual difference between routes and directions was not immediately clear.
3. The test cases were not well-structured, particularly in the transition between the first and second steps. Also, starting with a test case other than searching for routes might be more effective.

[6] This dataset is not included in the electronic attachment due to license conditions. Please refer to https://wiki.openstreetmap.org/wiki/Wikidata#Importing_data for more information.

4. The dialog for selecting a point might contain one more option that allows filtering based on address or arbitrary keywords, similar to *Mapy.cz*.

5. When searching for routes, the system checks if the starting point and destination are not too far from each other at the time of clicking the “Search” button. Some participants found it more convenient to be informed upfront.

6. In the results of a place search, sorting by the distance from the center point did not play a significant role. It was suggested to include a link to the corresponding detailed view and a “Save” button in each place’s popup.

7. Some participants pointed out that three-dotted menu buttons should be considered to hide extra functionality, and separating the “Save” button might be beneficial. In contrast, other participants found the menu layout for managing results and favorite entities consistent.

Despite all the efforts to create an intuitive user interface for accomplishing **G3**, there is still room for improvement. At the same time, the results showed that the application is easy to learn and get used to.

Conclusion

The primary goal of this thesis was to address the iterative nature of explicit location-based routing implemented by most mainstream web mapping applications. We designed, developed, and tested the web application that lets users to formulate route search queries in terms of categories, each composed of a keyword and attribute filters. A resulting route passes through at least one place from each category. The search procedure is formalized as a variant of the generalized Traveling Salesman Problem.

We analyzed existing solutions and derived the unique set of requirements and properties that our application should possess. Furthermore, we justified its relevancy by providing user stories based on real-life situations.

The application follows the three-tier architectural pattern. The frontend implements a panel-based layout compatible with both desktop and mobile devices. The backend, database, and routing engine together form an efficient solver. We integrate information from several publicly available semi-structured and structured data sources to facilitate search queries. User data is stored in a decentralized way, with an in-browser database and Solid pod being used interchangeably.

The application was tested using a variety of techniques. The performance tests confirmed that the technology stack was sufficient to meet the requirements, while the usability tests provided directions for improving the user experience.

We may conclude that the application satisfies all the requirements stated in the analysis phase; however, it cannot be considered complete. First, we should recall the performance drain caused by rendering a large number of markers on the client side in the main thread. Thus, delegating this task to a Web Worker or the backend should be prioritized.

Another possible extension briefly mentioned in relation to Requirement **F23** is to make the application follow the principles of *Linked Data*. In simpler terms, the backend should generate RDF for all types of entities. Once the application data is structured, we will be able to harness the full potential of Solid pods. For example, Van de Wynckel and Signer [52] recently presented a Solid-based architecture with interoperable location data for an indoor positioning system and a prototype application. Their experience and results can serve as a valuable starting point.

In addition, we identify four possible directions for future development that might be interesting from both theoretical and practical points of view.

- Enhance the user experience by designing a richer system of metadata and advanced use cases for entities in private storage, such as searching, tagging, grouping, filtering, etc.
- Enable collaboration via Solid pods. Allow users to share and comment on entities.
- Experiment with path-finding algorithms and heuristics to strike a new balance between the variety of search results and optimality.
- Apply advanced data mining and keyword extraction techniques to improve the quality of application data.

Bibliography

[1] Brandon Plewe. "Web Cartography in the United States". In: *Cartography and Geographic Information Science* 34.2 (2007), pp. 133–136. DOI: 10.1559/152304007781002235.

[2] Ming-Hsiang Tsou. "Revisiting Web Cartography in the United States: the Rise of User-Centered Design". In: *Cartography and Geographic Information Science* 38.3 (2011), pp. 250–257. DOI: 10.1559/1523040638250.

[3] Richard Cyganiak, David Wood, and Markus Lanthaler. *RDF 1.1. Concepts and Abstract Syntax*. W3C Recommendation. 2014. URL: http://www.w3.org/TR/2014/REC-rdf11-concepts-20140225/ (visited on 22/10/2023).

[4] Gavin Carothers and Andy Seaborne. *RDF 1.1. N-Triples. A line-based syntax for an RDF graph*. W3C Recommendation. 2014. URL: http://www.w3.org/TR/2014/REC-n-triples-20140225/ (visited on 22/10/2023).

[5] Gavin Carothers. *RDF 1.1. N-Quads. A line-based syntax for an RDF graph*. W3C Recommendation. 2014. URL: http://www.w3.org/TR/2014/REC-n-quads-20140225/ (visited on 22/10/2023).

[6] Gregg Kellogg, Pierre-Antoine Champin, and Dave Longley. *JSON-LD 1.1. A JSON-based Serialization for Linked Data*. W3C Recommendation. 2020. URL: https://www.w3.org/TR/2020/REC-json-ld11-20200716/ (visited on 22/10/2023).

[7] The W3C SPARQL Working Group. *SPARQL 1.1. Overview*. W3C Recommendation. 2013. URL: http://www.w3.org/TR/2013/REC-sparql11-overview-20130321/ (visited on 22/10/2023).

[8] Tim Berners-Lee. *Linked Data. Personal View*. 2006. URL: https://www.w3.org/DesignIssues/LinkedData.html (visited on 23/10/2023).

[9] Martin Kleppmann, Adam Wiggins, Peter van Hardenberg, and Mark McGranaghan. "Local-first software: you own your data, in spite of the cloud". In: *Proceedings of the 2019 ACM SIGPLAN International Symposium on New Ideas, New Paradigms, and Reflections on Programming and Software*. ACM, 2019, pp. 154–178. ISBN: 9781450369954. DOI: 10.1145/3359591.3359737.

[10] Saqib Ali, Guojun Wang, Bebo White, and Roger Leslie Cottrell. "A blockchain-based decentralized data storage and access framework for PingER". In: *2018 17th IEEE International Conference On Trust, Security And Privacy In Computing And Communications/ 12th IEEE International Conference On Big Data Science And Engineering (TrustCom/BigDataSE)*. 2018, pp. 1303–1308. DOI: 10.1109/TrustCom/BigDataSE.2018.00179.

[11] Sarven Capadisli, Tim Berners-Lee, Ruben Verborgh, and Kjetil Kjernsmo. *Solid Protocol. Version 0.10.0 (work in progress)*. Solid Technical Report. 2022. URL: https://solidproject.org/TR/2022/protocol-20221231 (visited on 24/10/2023).

[12] Tim Berners-Lee. *Socially Aware Cloud Storage. Personal View*. 2009. URL: https://www.w3.org/DesignIssues/CloudStorage.html (visited on 24/10/2023).

[13] Essam Mansour, Andrei Vlad Sambra, Sandro Hawke, Maged Zereba, Sarven Capadisli, Abdurrahman Ghanem, Tim Berners-Lee, and Ashraf Aboulnaga. "A Demonstration of the Solid Platform for Social Web Applications". In: *Proceedings of the 25th International Conference Companion on World Wide Web*. WWW '16 Companion. International World Wide Web Conferences Steering Committee, 2016, pp. 223–226. ISBN: 9781450341448. DOI: 10.1145/2872518.2890529.

[14] Martin Glinz. "On Non-Functional Requirements". In: *15th IEEE International Requirements Engineering Conference (RE 2007)*. 2007, pp. 21–26. DOI: 10.1109/RE.2007.45.

[15] Aidan Hogan, Eva Blomqvist, Michael Cochez, Claudia d'Amato, Gerard de Melo, Claudio Gutiérrez, Sabrina Kirrane, José Emilio Labra Gayo, Roberto Navigli, Sebastian Neumaier, Axel-Cyrille Ngonga Ngomo, Axel Polleres, Sabbir M. Rashid, Anisa Rula, Lukas Schmelzeisen, Juan F. Sequeda, Steffen Staab, and Antoine Zimmermann. *Knowledge Graphs*. English. Synthesis Lectures on Data, Semantics, and Knowledge 22. Springer, 2021. ISBN: 9783031007903. DOI: 10.2200/S01125ED1V01Y202109DSK022. URL: https://kgbook.org/.

[16] Charlie Kritschmar. *Graphic representing the datamodel in Wikidata with a statement group and opened references*. June 2016. URL: https://www.mediawiki.org/wiki/File:Datamodel_in_Wikidata.svg (visited on 2/11/2023).

[17] Sören Auer, Christian Bizer, Georgi Kobilarov, Jens Lehmann, Richard Cyganiak, and Zachary Ives. "DBpedia: A Nucleus for a Web of Open Data". In: *The Semantic Web*. Berlin, Heidelberg: Springer Berlin Heidelberg, 2007, pp. 722–735. ISBN: 978-3-540-76298-0. DOI: 10.1007/978-3-540-76298-0_52.

[18] Ali Ismayilov, Dimitris Kontokostas, Sören Auer, Jens Lehmann, and Sebastian Hellmann. "Wikidata through the eyes of DBpedia". In: *Semant. Web* 9.4 (2018), pp. 493–503. ISSN: 2210-4968. DOI: 10.3233/SW-170277.

[19] Pieter Vansteenwegen, Wouter Souffriau, Greet Vanden Berghe, and Dirk Van Oudheusden. "The City Trip Planner: An expert system for tourists". In: *Expert Systems with Applications* 38.6 (2011), pp. 6540–6546. ISSN: 0957-4174. DOI: 10.1016/j.eswa.2010.11.085.

[20] Daniel Herzog, Christopher Laß, and Wolfgang Wörndl. "Tourrec: a tourist trip recommender system for individuals and groups". In: *Proceedings of the 12th ACM Conference on Recommender Systems*. RecSys '18. Association for Computing Machinery, 2018, pp. 496–497. ISBN: 9781450359016. DOI: 10.1145/3240323.3241612.

[21] Daniel Herzog, Sherjeel Sikander, and Wolfgang Wörndl. "Integrating route attractiveness attributes into tourist trip recommendations". In: *Companion Proceedings of The 2019 World Wide Web Conference*. Association for Computing Machinery, 2019, pp. 96–101. ISBN: 9781450366755. DOI: 10.1145/3308560.3317052.

[22] Roi Friedman, Itsik Hefez, Yaron Kanza, Roy Levin, Eliyahu Safra, and Yehoshua Sagiv. "WISER: a web-based interactive route search system for smartphones". In: *Proceedings of the 21st International Conference on World Wide Web*. Association for Computing Machinery, 2012, pp. 337–340. ISBN: 9781450312301. DOI: 10.1145/2187980.2188043.

[23] Simon Brown. *The C4 Model for Software Architecture*. June 2018. URL: https://www.infoq.com/articles/C4-architecture-model/ (visited on 12/11/2023).

[24] Paul M. Jones. *Action Domain Responder*. URL: https://pmjones.io/adr/ (visited on 14/11/2023).

[25] Jeffrey Palermo. *The Onion Architecture: part 1*. July 2008. URL: https://jeffreypalermo.com/2008/07/the-onion-architecture-part-1/ (visited on 14/11/2023).

[26] Dongming Guo and Erling Onstein. "State-of-the-Art Geospatial Information Processing in NoSQL Databases". In: *ISPRS International Journal of Geo-Information* 9.5 (2020). ISSN: 2220-9964. DOI: 10.3390/ijgi9050331.

[27] Nils Nolde. *Open Source Routing Engines And Algorithms – An Overview*. Dec. 2020. URL: https://gis-ops.com/open-source-routing-engines-and-algorithms-an-overview/ (visited on 18/11/2023).

[28] Jiří Matoušek and Jaroslav Nešetřil. *Invitation to Discrete Mathematics*. 2nd ed. Oxford: Oxford University Press, 2008. ISBN: 978-0-19-857043-1.

[29] Reinhard Diestel. *Graph Theory*. 5th Electronic Edition. 2016. URL: http://diestel-graph-theory.com/ (visited on 21/11/2023).

[30] Michael Sipser. *Introduction to the Theory of Computation*. 3rd ed. Boston: Course Technology Cengage Learning, 2013. ISBN: 978-1-133-18779-0.

[31] Richard M. Karp. "Reducibility among Combinatorial Problems". In: *Complexity of Computer Computations. The IBM Research Symposia Series*. Boston, MA: Springer US, 1972, pp. 85–103. ISBN: 978-1-4684-2001-2. DOI: 10.1007/978-1-4684-2001-2_9.

[32] David P. Williamson and David B. Shmoys. *The Design of Approximation Algorithms*. 2010. URL: https://www.designofapproxalgs.com/ (visited on 23/12/2022).

[33] Sartaj Sahni and Teofilo Gonzalez. "P-Complete Approximation Problems". In: *J. ACM* 23.3 (1976), pp. 555–565. ISSN: 0004-5411. DOI: 10.1145/321958.321975.

[34] Vera Traub and Jens Vygen. "An improved approximation algorithm for ATSP". In: *Proceedings of the 52nd Annual ACM SIGACT Symposium on Theory of Computing*. Association for Computing Machinery, 2020, pp. 1–13. ISBN: 9781450369794. DOI: 10.1145/3357713.3384233.

[35] Nicos Christofides. *Worst-Case Analysis of a New Heuristic for the Travelling Salesman Problem.* Technical report 388. Carnegie Mellon University, 1976.

[36] Rico Zenklusen. “A 1.5-Approximation for Path TSP”. In: *CoRR* (2018). DOI: 10.48550/arXiv.1805.04131.

[37] Aldy Gunawan, Hoong Chuin Lau, and Pieter Vansteenwegen. “Orienteering Problem: A survey of recent variants, solution approaches and applications”. In: *European Journal of Operational Research* 255.2 (2016), pp. 315–332. ISSN: 0377-2217. DOI: 10.1016/j.ejor.2016.04.059.

[38] Petrică C. Pop, Ovidiu Cosma, Cosmin Sabo, and Corina Pop Sitar. “A comprehensive survey on the generalized traveling salesman problem”. In: *European Journal of Operational Research* (2023). ISSN: 0377-2217. DOI: 10.1016/j.ejor.2023.07.022.

[39] Feifei Li, Dihan Cheng, Marios Hadjieleftheriou, George Kollios, and Shang-Hua Teng. “On Trip Planning Queries in Spatial Databases”. In: *Advances in Spatial and Temporal Databases.* Berlin, Heidelberg: Springer Berlin Heidelberg, 2005, pp. 273–290. ISBN: 978-3-540-31904-7. DOI: 10.1007/11535331_16.

[40] Xin Cao, Lisi Chen, Gao Cong, and Xiaokui Xiao. “Keyword-aware Optimal Route Search”. In: *Proc. VLDB Endow.* 5.11 (2012), pp. 1136–1147. ISSN: 2150-8097. DOI: 10.14778/2350229.2350234.

[41] Haiquan Chen, Wei-Shinn Ku, Min-Te Sun, and Roger Zimmermann. “The Multi-Rule Partial Sequenced Route Query”. In: *Proceedings of the 16th ACM SIGSPATIAL International Conference on Advances in Geographic Information Systems.* GIS ’08. New York, NY, USA: Association for Computing Machinery, 2008. ISBN: 9781605583235. DOI: 10.1145/1463434.1463448.

[42] Mehdi Sharifzadeh, Mohammad Kolahdouzan, and Cyrus Shahabi. “The Optimal Sequenced Route Query”. In: *The VLDB Journal* 17.4 (2008), pp. 765–787. ISSN: 1066-8888. DOI: 10.1007/s00778-006-0038-6.

[43] Yaron Kanza, Eliyahu Safra, Yehoshua Sagiv, and Yerach Doytsher. “Heuristic Algorithms for Route-Search Queries over Geographical Data”. In: *Proceedings of the 16th ACM SIGSPATIAL International Conference on Advances in Geographic Information Systems.* GIS ’08. New York, NY, USA: Association for Computing Machinery, 2008. ISBN: 9781605583235. DOI: 10.1145/1463434.1463449.

[44] Matthias Englert, Heiko Röglin, and Berthold Vöcking. “Worst Case and Probabilistic Analysis of the 2-Opt Algorithm for the TSP”. In: *Algorithmica* 68.1 (2014), pp. 190–264. ISSN: 1432-0541. DOI: 10.1007/s00453-013-9801-4.

[45] Yaron Kanza, Roy Levin, Eliyahu Safra, and Yehoshua Sagiv. “Interactive Route Search in the Presence of Order Constraints”. In: *Proc. VLDB Endow.* 3.1–2 (2010), pp. 117–128. ISSN: 2150-8097. DOI: 10.14778/1920841.1920861.

[46] Ali Alabbas and Joshua Bell. *Indexed Database API 2.0*. W3C Recommendation. 2018. URL: https://www.w3.org/TR/2018/REC-IndexedDB-2-20180130/ (visited on 7/12/2023).

[47] Project OSRM. *OSRM API Documentation. Version 5.24.0*. URL: https://project-osrm.org/docs/v5.24.0/api/ (visited on 14/12/2023).

[48] MongoDB Inc. *MongoDB Documentation. Version 4.4*. URL: https://www.mongodb.com/docs/v4.4/ (visited on 7/12/2023).

[49] Jakob Nielsen. *Response Times: The 3 Important Limits*. Jan. 1993. URL: https://www.nngroup.com/articles/response-times-3-important-limits/ (visited on 16/12/2023).

[50] John Brooke. "SUS: A 'Quick and Dirty' Usability Scale". In: *Usability Evaluation In Industry*. 1st ed. London: CRC Press, June 1996, pp. 189–194. ISBN: 9780429157011. DOI: 10.1201/9781498710411.

[51] Jeff Sauro. *Measuring Usability with the System Usability Scale (SUS)*. Feb. 2011. URL: https://measuringu.com/sus/ (visited on 26/12/2023).

[52] Maxim Van de Wynckel and Beat Signer. "A Solid-based Architecture for Decentralised Interoperable Location Data". In: *Proceedings of IPIN 2022 (WiP), 12th International Conference on Indoor Positioning and Indoor Navigation, Beijing, China, September 2022*. Vol. 3248. CEUR Workshop Proceedings, 2022, pp. 1–15. URL: http://www.ipin-conference.org/2022/.

List of Tables

2.1 Attribute filters. 22
2.2 Feature comparison. 29

3.1 Possible colors of pins on the map and their meaning. 31

5.1 Parameters of the performance test environment. 67

A.1 Results of usability testing. 92

List of Figures

1.1 An RDF graph with one triple. 6
1.2 The concept of decentralization using Solid. 9

2.1 The use cases related to searching routes. 17
2.2 The use cases related to searching places. 17
2.3 The use cases related to searching directions. 18
2.4 The use cases related to entity and storage management. 18
2.5 Simplified data model of Wikidata [16]. 21
2.6 A UML class diagram of the conceptual model. 23
2.7 Mapy.cz – Search tab: ① initial view, ② autocomplete options with places and categories, ③ the results of a categorical search. . 25
2.8 Mapy.cz – Directions tab: ① initial view, ② a list of directions. . 25
2.9 Komoot: category filters. 26
2.10 Kurviger: a randomized round trip. 27

3.1 Navigation schema. 30
3.2 Wireframe with the dialog for selecting a point. 32
3.3 Wireframe depicting the state of the dialog for configuring arrows. The arrow with a solid line represents a “confirmed” arrow, and the one with a dashed line represents a “not confirmed” arrow. . . 32
3.4 Wireframe depicting the state of the dialog for configuring categories after selecting a keyword, including all five types of attribute filters. 33
3.5 Wireframe with the panel for searching routes. 34
3.6 Wireframe with the panel showing the results of a route search. . 34
3.7 Wireframe with the panel for searching places. 35
3.8 Wireframe with the panel showing the result of a place search. . . 35
3.9 Wireframe with the panel for searching directions. 36
3.10 Wireframe with the panel showing the results of a direction search. 36
3.11 Wireframe with the detailed view of a place. 37
3.12 Wireframe with the Solid login dialog. 37
3.13 Wireframe with the panel for activating a Solid pod. 37
3.14 Wireframe with the panel containing stored (favorite) entities. . . 38
3.15 C4 container diagram of the SmartWalk software system. 39
3.16 C4 component diagram of the Frontend container. 40
3.17 C4 component diagram of the Backend container. 41
3.18 An overview of the related NP-optimization problems. 49
3.19 Infrequent-First Heuristic, selection criterion. 50
3.20 Oriented Greedy Heuristic, selection criterion. 51
3.21 Geojson.io: rotated and translated ellipse drawn on the map. . . . 53

5.1 Response times for fetching keywords by prefix. 68
5.2 Response times for retrieving places by `smartId`. 68
5.3 Response times for direction search queries. 69
5.4 Response times for place search queries. 70
5.5 Response times for route search queries. 71

A.1 The panel for searching routes with configured input parameters. . 85
A.2 The results of a route search. 86

List of Abbreviations

ADR	Action Domain Responder
API	Application Programming Interface
ATSP	Asymmetric TSP
CRS	Coordinate Reference System
DIP	Dependency Inversion Principle
ESS	Enterprise Solid Server
GTSP	Generalized TSP
HTTP	Hypertext Transfer Protocol
IFH	Infrequent-First Heuristic
IRI	Internationalized Resource Identifier
JSON	JavaScript Object Notation
JSON-LD	A JSON-based Serialization for Linked Data
KOR	Keyword-Aware Optimal Route Query
MRPSR	Multi-Rule Partial Sequenced Route
OGH	Oriented Greedy Heuristic
OP	Orienteering Problem
OSM	OpenStreetMap
OSR	Optimal Sequenced Route
OSRM	Open Source Routing Machine
PCGTSP	Precedence Constrained GTSP
RDF	Resource Description Framework
Solid	Social Linked Data
SPARQL	SPARQL Protocol and RDF Query Language
STSP	Symmetric TSP
SUS	System Usability Scale
TPQ	Trip Planning Query
TSP	Traveling Salesman Problem
TTDP	Tourist Trip Design Problem
WDQS	Wikidata Query Service

A. Attachments

A.1 Documentation

The project documentation comprises the following *three* parts.

User's documentation gives an overview of how to use the application and accomplish basic tasks, such as searching for and managing entities.

Please note that this is a simplified version of Attachment A.3 with screenshots. Examples of the user interface are depicted in Figures A.1 and A.2.

Programmer's guide brings clarity into the application architecture and code organization.

Please note that this is a condensed version of Section 3.2 and Chapter 4.

Administrator's guide provides instructions for preparing a dataset, running the application in development or production mode on a personal computer, and troubleshooting potential issues.

The version of the documentation at the time of submitting the thesis is available in the `./docs/` folder of the electronic attachment, and the latest version is hosted at

`https://zhukovdm.github.io/smartwalk-docs/`.

A.2 Prerequisites

SmartWalk is essentially cross-platform. However, Unix utilities simplify certain aspects of system maintenance. We assume that the application will run on *Unix-like* environments, such as Linux or Windows Subsystem for Linux[1].

Please ensure that the following programs are installed on the target system. If mentioned, preserve versions of packages due to library dependencies.

- Docker
- .NET SDK, v6.0
- Git
- GNU Bash, Make, Tar, and Wget
- Node.js, v18.x (install via nvm[2])

Clone the repository with *submodules* and navigate to its root folder:

```
$ git clone --recurse-submodules https://github.com/zhukovdm/smartwalk.git
$ cd ./smartwalk/
```

[1]https://learn.microsoft.com/en-us/windows/wsl/about
[2]https://github.com/nvm-sh/nvm#install–update-script

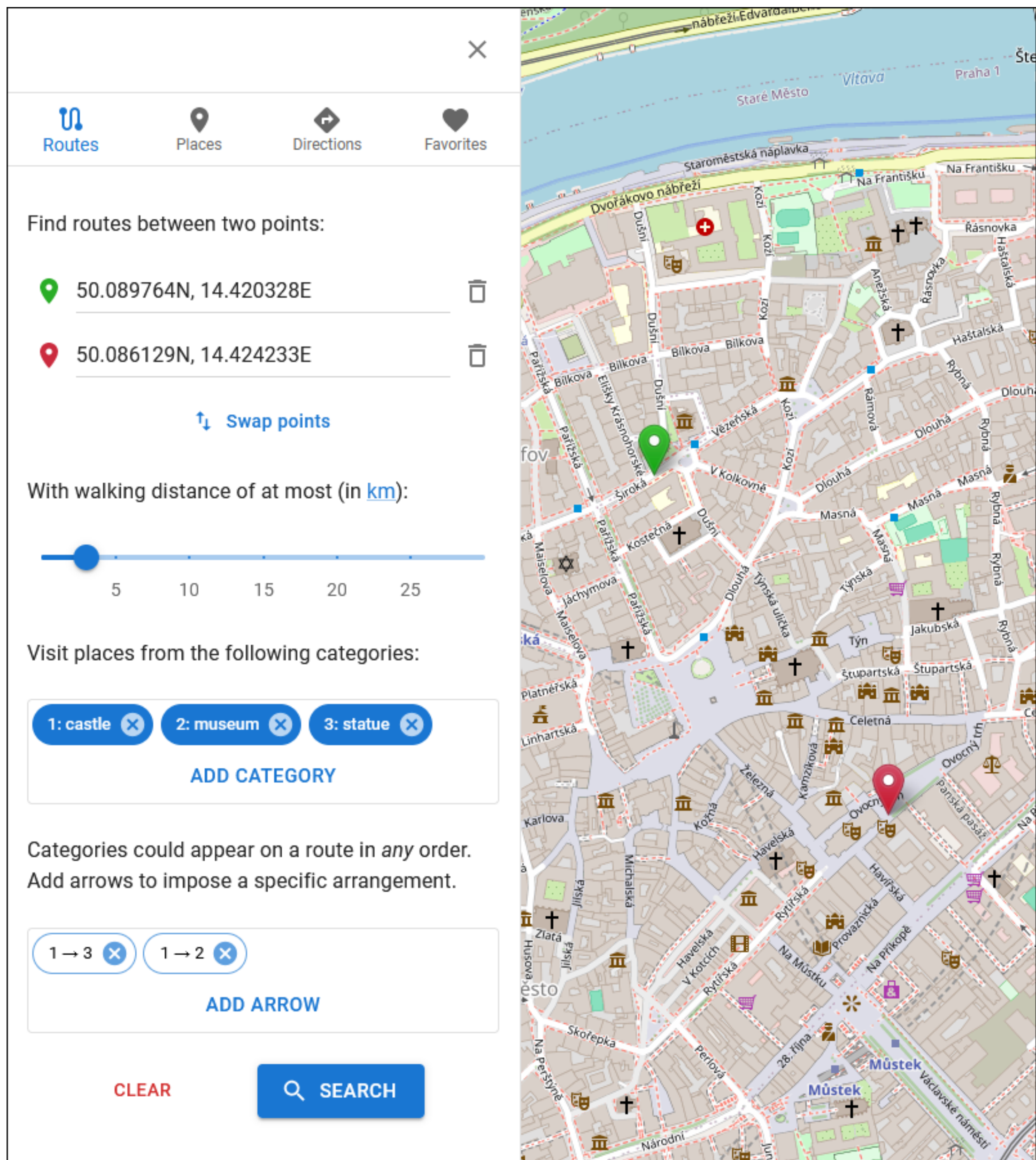

Figure A.1: The panel for searching routes with configured input parameters.

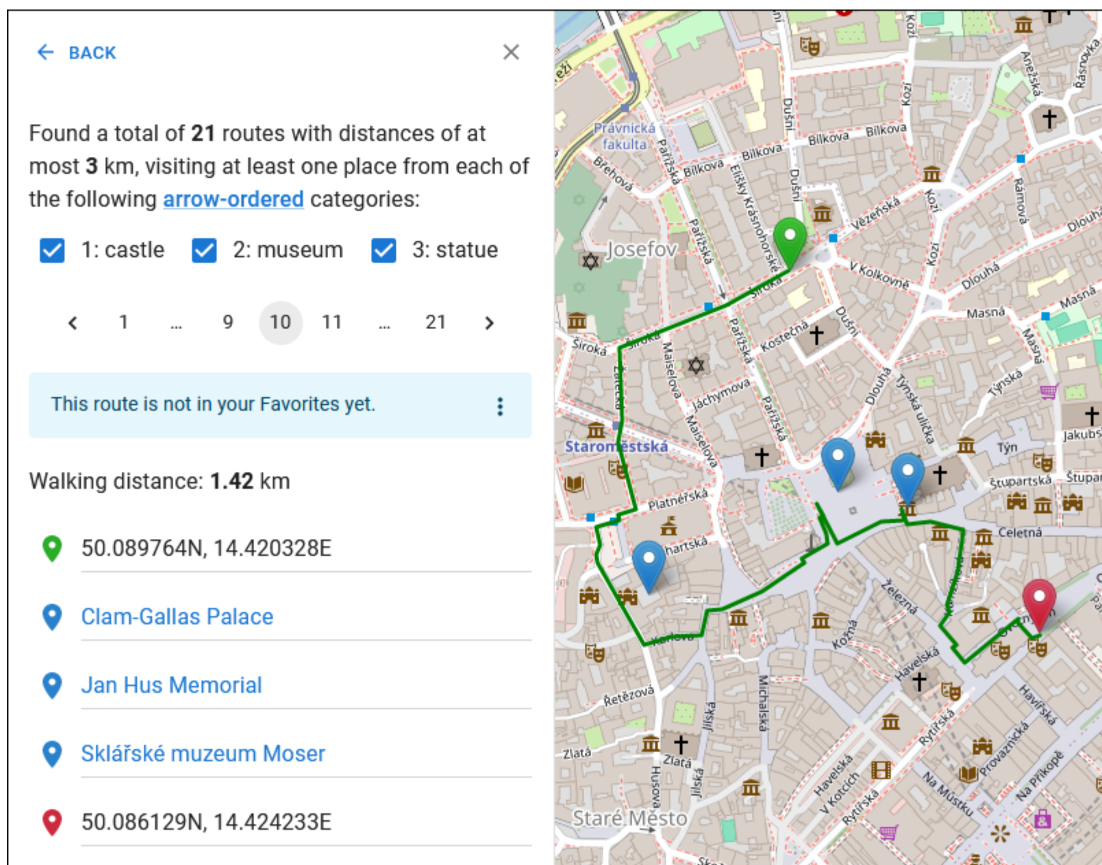

Figure A.2: The results of a route search.

A.3 Use cases

A.3.1 UC01: Select point

Initial state

- A user has opened a panel for searching routes, places, or directions.

Normal flow

1. The user clicks the "Select point" button to fill the corresponding slot.
2. The system shows a dialog with two options:
   (a) select a location on the map,
   (b) select a place from the list of stored places.
3. The user performs one of the following workflows.
   - They vote for option (a).
     i. The user clicks the button "Select location".
     ii. The system hides the panel and dialog, presenting only the map.
     iii. The user selects a location by clicking on the map.
   - They vote for option (b).
     i. The user selects a place from the list of stored places and confirms their choice.

Final state

- The panel appears with the selected entity set as required.

A.3.2 UC02: Add category

Initial state

- A user has navigated to a panel for searching routes or places.

Normal flow

1. The user clicks on the "Add category" button.
2. The application opens the dialog and proposes to type in a keyword.
3. The user starts typing. For every prefix, the system suggests options.
4. The user clicks on one of the options confirming their choice.
5. The system shows keyword-specific attribute filters below the input field.
6. The user activates attribute filters relevant to their search and sets bounds.
7. The user clicks on the "Confirm" button once the desired configuration has been achieved.

Final state

- The newly created category is appended to the list.

Alternatives

- The system raises an error upon a failed attempt to suggest options.

Extensions

- The configuration of the attribute filters of a category can be altered at any time by clicking on the category element and applying changes.

A.3.3 UC03: Add arrow

Initial state

- A user has navigated to the panel for searching routes.

Normal flow

1. The user clicks on the "Add arrow" button.
2. The application opens the dialog, shows the current precedence graph, and proposes to add a new arrow.
3. The user selects the left and right counterparts of the arrow and clicks on the "Confirm" button.

Final state

- The newly created arrow is appended to the list.

Alternatives

- The system raises an error if the user attempts to create a cyclic dependency or repeats an existing arrow.

A.3.4 UC04: Search routes

Initial state

- A user has navigated to the panel for searching routes.

Normal flow

1. The user performs the following steps:
   - Select a starting point and a destination following *UC01: Select point*.
   - Set a maximum walking distance using the corresponding slider.
   - Configure a non-empty list of categories applying *UC02: Add category* repeatedly.
   - Configure an optional list of arrows applying *UC03: Add arrow* repeatedly.
2. The user clicks the "Search" button, finalizing their request.
3. The system navigates the user to the panel with a paginated list of routes.

Alternatives

- The system rejects the query if the crow-fly distance between the starting point and destination exceeds the maximum limit.
- If the system fails to find routes, it proposes to alter search parameters.
- If the system fails to complete the query, it informs the user via an error.

A.3.5 UC05: Show detailed view of a place

Initial state

- The user has navigated to the detailed view of a place using a link or entered the link directly into the browser.

Normal flow

1. The system shows a progress spinner and renders the view once the object is ready to be presented.

Alternatives

- An object with the given identifier does not exist, or an error has occurred while retrieving the object. Then, the system informs the user via an alert.

A.3.6 UC06: Save place

Initial state

- The user has obtained a detailed view of a place that has not been saved.

Normal flow

1. The user clicks the "Menu" button and then the "Save" menu item.
2. The system shows a dialog, proposes to enter metadata of the place and informs that a personal copy is about to be created.
3. The user enters a name and confirms their choice by clicking the "Confirm" button.

Final state

- The system updates the private storage, hides the dialog, and marks the place as saved.

Alternatives

- The system fails to update the private storage and shows an alert.

A.3.7 UC07: Search directions

This use case has many similarities with *UC04: Search routes*. Instead of providing its full description, let us cover various options on how to extend a sequence of points depicted in Figure 2.3. Most of them are implemented to enhance the user experience.

The sequence can be extended by selecting a point directly on the map or from the private storage using standard *UC01: Select point*. This use case implies that the user interface shows the panel for searching directions.

Nevertheless, the user might want to browse through stored entities to recall the content. The system enables them to append places and move points of paths directly from the storage.

To avoid the overhead of storing and navigating, the user could also modify a route that has appeared in the result of a search query or append a place from its detailed view.

A.3.8 UC08: Modify route

Initial state

- A user has navigated to the panel with stored entities and sees the route to be modified.

Normal flow

1. The user clicks the "Menu" button and then the "Modify" menu item.
2. The system shows a dialog with the message informing that points of this route will replace those configured in the panel for searching directions.
3. The user clicks the "Confirm" button.

Final state

- The system redirects the user to the panel for searching directions. Points of the selected path have replaced old ones.

A.3.9 UC09: View entity

Initial state

- A user has navigated to the panel with stored entities and sees the entity to be viewed.

Normal flow

1. The user clicks the "Menu" button and then the "View" menu item.

Final state

- The system immediately redirects the user to the viewer panel.

A.3.10 UC10: Edit entity

Initial state

- A user has navigated to the panel with stored entities and sees the entity to be edited.

Normal flow

1. The user clicks the "Menu" button and then the "Edit" menu item.
2. The system shows a dialog with editable fields containing current metadata.
3. The user sets new values for selected items and clicks the "Save" button.

Final state

- The system hides the dialog, indicating that the object has been updated.

Alternatives

- The system fails to update the object and shows an alert.

A.3.11 UC11: Activate Solid pod

Initial state

- A user has loaded the initial web page.

  A Solid pod can be activated at any time unless another one is active.

Normal flow

1. The user clicks the "Log In" button outside the panel drawer and then the "Solid" menu item.

2. The system shows a dialog with an input field and asks to enter the URL of a Solid pod provider.
3. The user enters the URL or selects from the list and then clicks the "Log In" button.
4. The system redirects to the web page of the selected pod provider so that the user can enter their credentials and allow the application to access pods.
5. The user is redirected to the initial web page, and the "Solid" panel opens up shortly after.
6. The user selects a pod from the list of pods associated with their account and clicks the "Activate Pod" button.

   Before this step, the application still uses entities from the device storage.
7. The system redirects to the panel with stored entities, and loads the content from the Solid pod.

Final state

- The user ends up in the panel with entities stored in the activated pod.

Alternatives

- The system fails to activate pod and shows an error. No redirection to the panel with stored entities occurs.
- The system fails to load the content of the activated pod. It shows an error, and the loading spinner remains forever. The user should refresh the web page so that the application can load entities from the device storage.

A.3.12 UC12: Deactivate Solid pod

Initial state

- The user has activated a pod following *UC11: Activate Solid pod*.

Normal flow

1. The user clicks the "Solid" button outside the panel drawer so that the system redirects them to the "Solid" panel.
2. The user clicks the "Log Out" button.
3. The system redirects to the panel with stored entities, and loads the content from the device storage.

Final state

- The user ends up in the panel with entities stored in the device storage.

A.4 Results of usability testing

The respondents were asked to express their opinions on the following statements, which first appeared in [50].

S1 I think that I would like to use this system frequently.

S2 I found the system unnecessarily complex.

S3 I thought the system was easy to use.

S4 I think that I would need the support of a technical person to be able to use this system.

S5 I found the various functions in this system were well integrated.

S6 I thought there was too much inconsistency in this system.

S7 I would imagine that most people would learn to use this system very quickly.

S8 I found the system very cumbersome to use.

S9 I felt very confident using the system.

S10 I needed to learn a lot of things before I could get going with this system.

Responses were evaluated on a scale of 1 (strongly disagree) to 5 (strongly agree). Individual scores were calculated using the formula

$$2.5 \cdot \left(\sum_{i \in \{1,3,5,7,9\}} (S_i - 1) + \sum_{j \in \{2,4,6,8,10\}} (5 - S_j) \right)$$

and then averaged.

Resp.	**S1**	**S2**	**S3**	**S4**	**S5**	**S6**	**S7**	**S8**	**S9**	**S10**	**Score**
1	5	1	5	1	5	1	5	2	4	1	95.00
2	4	2	5	1	3	2	4	1	5	2	82.50
3	3	2	3	4	5	2	5	2	3	1	70.00
4	5	1	4	2	5	1	5	1	4	1	92.50
											85.00

Table A.1: Results of usability testing.

A.5 Electronic attachment

The electronic attachment to the thesis follows the structure below.

`code/` contains source code of the application (tag `v1.0.0` on GitHub) and documentation (tag `v1.0.0` on GitHub).

`docs/` contains generated documentation for the tag `v1.0.0` on GitHub. The file `./index.html` represents the main page.

`perf/` accommodates the dataset, two files `place.txt` and `keyword.txt`, for performance tests we mentioned in Section 5.2.