Interconnecting Public Transport Information

Merlin Felix

Bachelor's Thesis
September 2024

Real-Time and Embedded Systems Group
Department of Computer Science
Kiel University

Advised by
Dr. Ing. Alexander Schulz-Rosengarten

Selbstständigkeitserklärung

Hiermit erkläre ich, dass ich die vorliegende Arbeit selbstständig verfasst und keine anderen als die angegebenen Quellen und Hilfsmittel verwendet habe.

Weiterhin erkläre ich, dass die digitale Fassung dieser Arbeit, die dem Prüfungsamt per E-Mail zugegangen ist, der vorliegenden schriftlichen Fassung entspricht.

Kiel,

Abstract

Public transport is an important aspect to a sustainable future of mobility [MS13]. Thus, research on innovative public transport options are gaining interest. To help new and innovative public transport projects to integrate with existing solutions, this thesis aims towards interconnecting public transport and mobility information from different sources. It is intended to provide an easy-to-integrate service providing all relevant data in a uniform way. The provided data is strived to be ready-to-use in traveler-facing public transport information systems without the need of further preprocessing or cleansing.

Acknowledgements

First, I would like to thank Prof. Dr. Reinhard von Hanxleden as the Head of the Real-Time and Embedded Systems Group for making it possible to write this thesis.

Further, I would like to thank my advisor Dr.-Ing. Alexander Schulz-Rosengarten for the guidance, feedback and ideas he provided.

Finally, I would like to thank Tokessa Hamann, Yorik Hansen and especially Finn Evers for provided feedback, ideas and inspiration as well as the good company during the process of writing this thesis.

Contents

List of Figures

List of Tables

List of Acronyms

HAFAS	HaCon Fahrplan-Auskunfts-System
NAH.SH	Nahverkehrsverbund Schleswig-Holstein
API	Application Programmable Interface
REST	Representational State Transfer
DB	Deutsche Bahn
GTFS	General Transit Feed Specification
GBFS	General Bike Feed Specification
NeTEx	Network Timetable Exchange
SIRI	Service Interface for Real Time Information
CSV	Comma Separated Values
DELFI	Durchgängige Elektronische Fahrgastinformation
SSE	Server-Side Event
SQL	Structured Query Language
UI	User Interface

List of Tables

HTTP	Hypertext Transfer Protocol
ID	Identifier
HATEOAS	Hypermedia as the Engine of Application State
JSON	JavaScript Object Notation
HTML	Hyper Text Markup Language
XML	Extensible Markup Language
CSS	Cascading Style Sheets
OSM	OpenStreetMap

Chapter 1

Introduction

Public transport consists of a variety of different networks, modes of transportation, and transportation providers. Utilizing only one of them might often be insufficient to reach a desired destination. For example, two cities might be directly connected via railway, but reaching a specific location within a city might require taking the bus or using an on-demand service in case of more remote places. As a consequence, traveling by public transport does often require the utilization of multiple different options or services.

Other than the binary parameter of reachability, different public transport options also differ in terms of time efficiency, frequency, comfort, etc. When planning a trip using public transport, all the different options, their availabilities and other circumstances such as schedules are usually investigated and taken into consideration.

This problem does not only exist regarding the planning process in advance, it persists during the trip when it comes to adapting to, e.g., delayed or cancelled connections. These deviations from the schedule are communicated using different apps or websites, further complicating traveling by public transport.

Although services gathering information from different public transport agencies already exist, these have some major limitations. They are often limited to fixed-route public transport services with fixed time schedules. On-demand public transport on the other hand is likely to become more popular with the increasing interest in autonomous vehicles. In the context of *REAKT DATA*[1], a project dedicated to reactivate disused railway tracks using on-demand autonomous trains, a well suited integration with other mobility options is of interest. While it is sometimes possible to submit own timetables as a mobility provider, easy and affordable ways to integrate information regarding available public transport and mobility options in another project are also very limited. Use cases for this include for instance displaying or announcing subsequent mobility options. Existing solutions also tend to restrict source formats to mostly one or very few. However, not all public transport and mobility information is available in the same format. The challenge arising from the utilization of various data sources is the high diversity of the data superficially in terms of representation, but also contentwise in quality, density and completeness. Different information sources might also overlap in as far as the same information might be contained in multiple sources.

[1] https://reakt.sh/

1.1 Problem Statement

The goal of this thesis is to conceptualize, implement and evaluate a service, which provides in a uniform way information regarding different public transport and mobility options. The gathered information should be traveler facing and is intended to be used by passenger information systems, such as mobile apps or embedded devices such as in vehicles or at stops.

In order to create a solution especially aimed towards small or regional mobility projects like REAKT DATA, publicly available data sources should be investigated and interconnected. Further, the proposed solution will be open-source. This way, it can easily be used to integrate the own mobility service with others. If a desired data-source is missing, it can just be implemented.

A broad coverage of available options should be aimed for, optimally eliminating the need of consulting any other information sources. For the scope of this thesis, it is limited to the geographical proximity to the *Malente-Lütjenburg* railway track.

1.2 Outline

The next chapter will conclude related work on the topic of public transport information and existing applications. Chapter 3 will introduce common data formats and information sources relevant to this thesis. In Chapter 4, methodology for acquiring, merging and processing public transport and mobility data is addressed. Chapter 5 shows how the developed concepts are implemented and which technology is used. Chapter 6 reflects on the developed concepts. Finally, Chapter 7 concludes this thesis.

Chapter 2

Related Work

This chapter provides a brief overview over similar existing software solutions and the research previously conducted on the field of public transport information.

2.1 Existing Traveler-Facing Public Transport Information

Existing services providing public transport information for travelers include various solutions by transport companies and associations, as well as mostly proprietary regional solutions commissioned by cities or similar representatives. The proprietary nature of these solutions make it difficult to integrate a small mobility project, especially when it comes to accessing information regarding other mobility options directly, rather than viewing it on a standalone website or app. This data has to be collected from various sources, constituting a major complication, as the sources vary in standards, availability, quality and completeness.

One of the most commonly used systems across Germany is the HaCon Fahrplan-Auskunfts-System (HAFAS). HAFAS provides both static timetable information and real-time deviations. It is used by, for example, the Deutsche Bahn and NAH.SH.

There are also many attempts to compile various public transport options into one service. For instance, *Google Transit* allows for the integration of public transport information with *Google Maps*. However, real-time data is missing for some routes and a few relevant information such as arrival or departure platforms are not displayed. While it is also possible to submit own timetables as a mobility provider, this requires operating on fixed-time schedules. This excludes on-demand services completely. Shared Mobility options are also not included. When a mobility project wants to integrate other mobility options by, e.g., displaying departures of adjacent public transport options, one can utilize the *Google Routes* API. This API however is not free, especially excluding very small local projects[1].

Other solutions integrating different mobility options include the *Liniennetzplan* NAH.SH[2], *SWL Mobil Planer*[3], *KVG Kiel Liniennetzplan*[4] and *Plön Mobil*[5], which are all developed by *Baumgardt Consultants*[6]. The UI is focused on nearby public mobility options as visible in Figure 2.1a using Plön Mobil. These solutions are also used for digital information kiosks

[1] https://mapsplatform.google.com/intl/de/pricing/

[2] https://www.liniennetz.nah.sh/

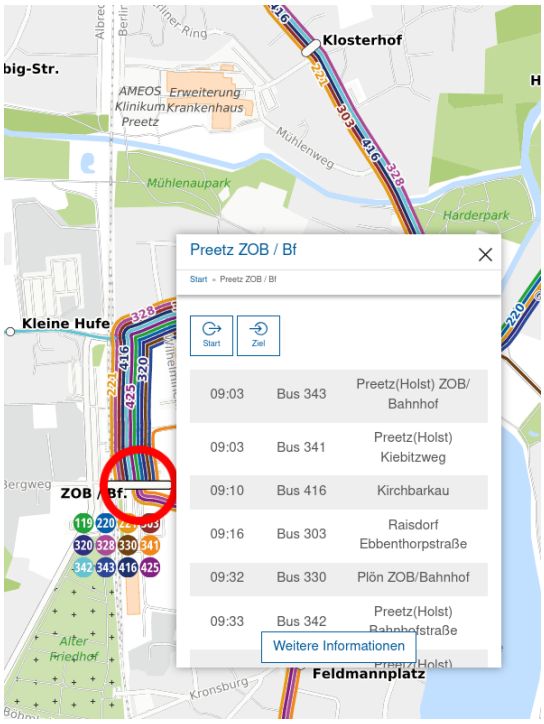
[3] https://netzplan.swhl.de/maps/tlnp

[4] https://www.netzplan-kiel.de/maps/tlnp-kiel

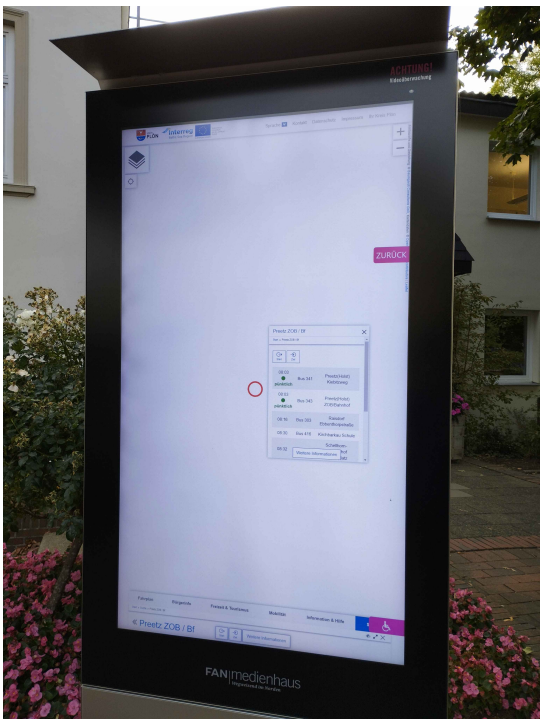
[5] https://www.ploen-mobil.de/

[6] https://www.baumgardt-online.de/

like Plön Mobil in Preetz, as shown in Figure 2.1b. In contrast to the strategy favored in this thesis, these solutions all just query the respective HAFAS instances from the frontend directly for displaying public transport information. This limits accessible public transport options to those available in the according HAFAS instance or requires connecting another API in the frontend. Shared mobility options like bike rental stations on the other hand are integrated via an own API.



(a) Plön Mobil website with the Bahnhof Preetz focused.



(b) Plön Mobil at a digital information kiosk in Preetz.

Figure 2.1. The Plön Mobil service.

Next to these proprietary solutions, there also exist a few open-source ones. For instance, Bahn Experte[7] is a web application, which provides detailed passenger information regarding German trains. Another example of a similar project is Kiel Live[8], which displays current positions of all buses operated by the Kieler Verkehrsgesellschaft. Catenary Maps[9] is a fairly new project, visualizing world-wide public transport. However, it mainly focuses on one source format and displays incorrect information in Germany. The UI can be seen in Figure 2.2, where multiple entries exist for the same lines, as well as a line called RE8, which does not exist in this form.

[7] https://bahn.expert/

[8] https://kiel-live.github.io/

[9] https://maps.catenarymaps.org

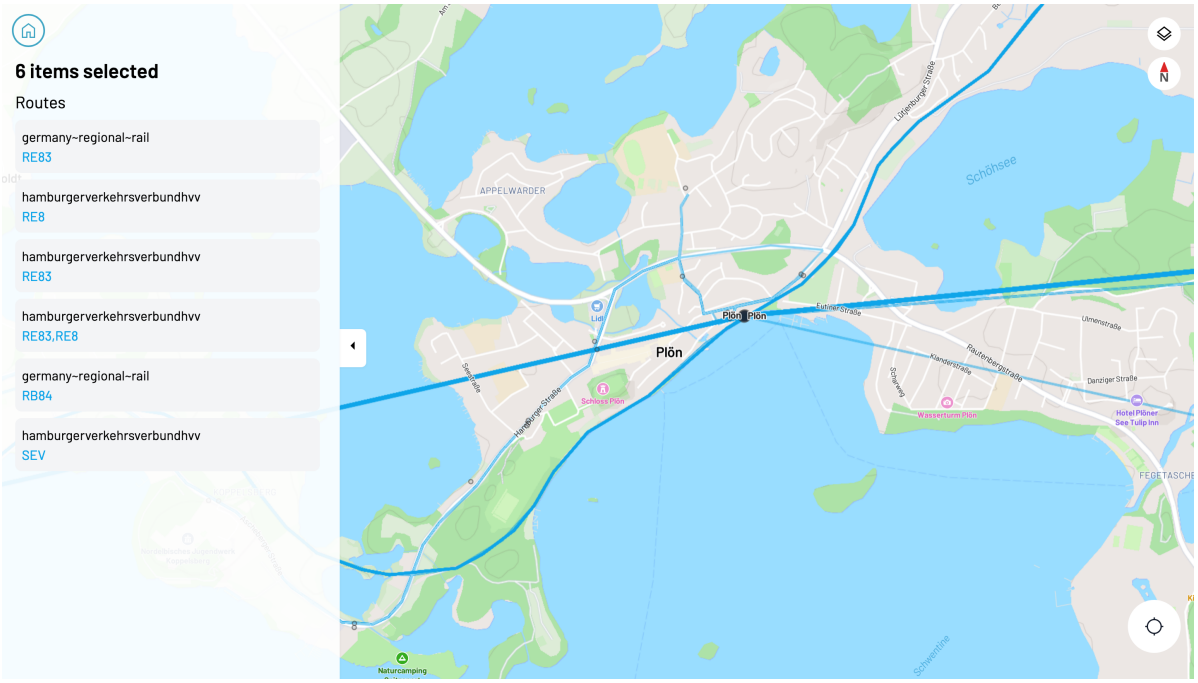


Figure 2.2. Catenary Maps with the wrongly displayed RE8 in Plön.

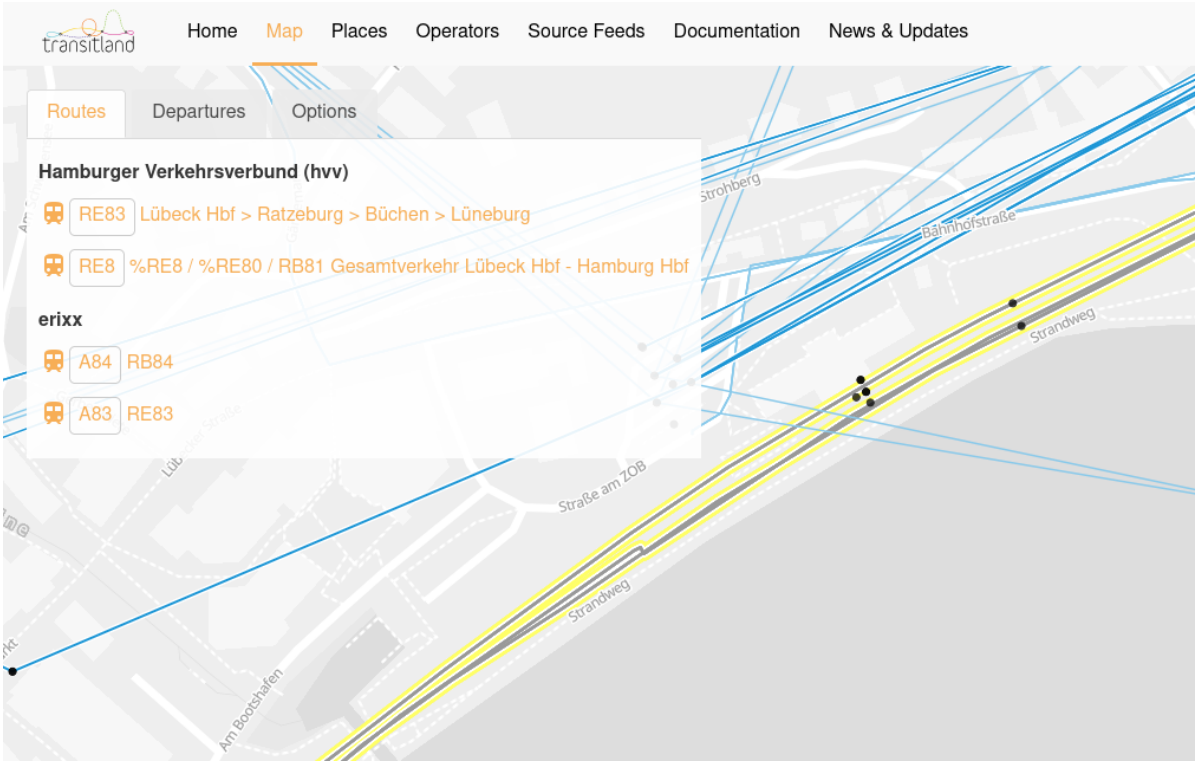


Figure 2.3. Transitland with the wrongly displayed RE8 in Plön.

Transitland[10] aggregates transit data from all over the world. This however is limited to open data standards, and proprietary APIs are excluded. Transitland features a download for the complete global feed as well as an API. It contains the same incorrect information as Catenary Maps as can be seen in Figure 2.3, since Catenary Maps uses Transitland as a source. Opposed to this thesis, Transitland seems to focus on global coverage rather than completeness within a regional area.

Lastly, ÖPNV Karte[11] displays public transport lines based on OpenStreetMap[12]. The displayed information however is limited to a list of stops for each line and the exact path. Further information such as arrival or departure times and schedules in general are not displayed.

2.2 Academic Research

Academic research on this topic is rather limited. However, research on similar topics has been conducted, including a world-wide visualization of vehicle positions based upon timetable information [BBS14]. Due to the bad quality of the available public transport schedule datasets for Germany, the authors decided to create a refined dataset, which is publicly available.

Another notable publication focuses on generating web scrapers for the public transport domain for the lack of publicly available information on accessibility [VCC+19]. This thesis however will not involve web-scraping, as many other data sources exist, which provide most of the desired information and are easier to work with. Nevertheless, web-scraping might be a suitable addition to the methods this thesis focuses on.

Lastly, research on the general performance of public transport information systems and routing exists [RVD+20; CVM17]. As this thesis' main focus is a broad, consistent and dense coverage of available mobility options, performance optimizations are beyond scope.

[10] https://www.transit.land/

[11] https://www.%C3%B6pnvkarte.de/

[12] https://www.openstreetmap.org/

Chapter 3

Preliminaries

This chapter gives a brief overview and introduction to the data specifications, data sets, and APIs used throughout this thesis.

Open data regarding public transport and mobility for the whole of Germany can be found in the *Mobilithek*[1], which is a service provided by the German *Bundesministerium für Digitales und Verkehr*. Datasets regarding public transport in Germany can also be found in the *Open Data ÖPNV*[2] portal. Although this service is provided by the *Verkehrsbund Rhein-Ruhr AöR*, datasets are for all of Germany are available. Datasets regarding public transport and mobility for the German state of Schleswig-Holstein can be found on the *Open-Data Schleswig-Holstein*[3] website, which is a service provided on behalf of the minister president of the state of Schleswig-Holstein.

3.1 Transmodel

Transmodel[4] provides a data model for the public transport domain. Based upon this are the specifications Network Timetable Exchange (NeTEx)[5] and Service Interface for Real Time Information (SIRI)[6]. NeTEx provides schedule data in the Extensible Markup Language (XML) format. SIRI extends this format by real-time information also via XML. This is a format mainly used in Europe including Germany. However, it is not used throughout this thesis.

3.2 General Transit Feed Specification Schedule

The GTFS[7] is an open standard and de facto standard for exchanging static information regarding public transport schedules. Google Transit[8] uses GTFS to integrate public transport information with Google Maps[9]. The *GeOps Live Train Tracker*[10] visualizes current positions

[1] https://mobilithek.info/

[2] https://www.opendata-oepnv.de/ht/de/willkommen

[3] https://opendata.schleswig-hosltein.de/dataset

[4] https://transmodel-cen.eu/

[5] https://transmodel-cen.eu/index.php/netex/

[6] https://transmodel-cen.eu/index.php/siri/

[7] https://gtfs.org/

[8] https://developers.google.com/transit

[9] https://www.google.com/maps

[10] https://mobility.portal.geops.io/com/world.geops.transit

of predominantly European and North American train lines on a map. These positions are estimated based on data in the GTFS format [BBS14]. This section provides a basic overview over the GTFS specification, but a detailed documentation can be found on the official GTFS website.

A GTFS feed consists of multiple text files in the Comma Separated Values (CSV) format. These files are usually bundled as a single zip file. The data is organized as a relational database, where each file represents a table. To gain a basic understanding of the GTFS structure, a selection of the most important files of a GTFS feed is listed:

- `stops.txt` contains names and geographic data for all stops in the feed.
- `trips.txt` contains all unique trips of the feed. Each trip is associated with a service. A trip is available when the service is available.
- `stop_times.txt` associates a list of stops and times of day with a trip. The order of stops within a trip is determined by a stop sequence.
- `routes.txt` contains all lines. Usually, `trips.txt` associates multiple trips with one line.
- `calendar.txt` defines the availability of services for each weekday within a start and end date.
- `calendar_dates.txt` specifies exceptions to the service availability specified by the `calendar_dates.txt`. It is also possible for service to have all available days defined in this file rather than `calendar_dates.txt`. At least one of `calendar.txt` or `calendar_dates.txt` must be present in the database.
- `shapes.txt` if present, describes the exact route a trip might take as sequence of geographic points.

3.2.1 General Transit Feed Specification Realtime

Live updates to a static GTFS feed are also possible using the GTFS Realtime specification. These updates may include deviations from the schedule or services, information about unscheduled trips and real-time vehicle positions. Like GTFS schedule, it is used by Google Transit and the GeOps Live Train Tracker. The updates are provided as a Protocol Buffer[11].

3.2.2 Relevant General Transit Feed Specification Datasets

A GTFS dataset for the whole of Germany is provided by the Durchgängige Elektronische Fahrgastinformation (DELFI). The GTFS schedule feed can be found in the Mobilithek[12] [13] and

[11] `https://protobuf.dev/`

[12] `https://mobilithek.info/offers/-2883874086141693018`

[13] `https://mobilithek.info/offers/552578819783815168`

OpenData ÖPNV[14] A matching GTFS Realtime feed is also available in the Mobilithek[15] This dataset is based upon the DELFI NeTEx dataset and preferred over the NeTEx dataset, as GTFS is a globally adopted format.

A dataset, which is developed upon the same base dataset, but aims towards a better quality, is the GTFS feed created by Patrik Brosi[16]. However, some information such as detailed paths of trips is omitted from the free version. This dataset also comes with a matching real-time feed.

Finally, a GTFS feed for NAH.SH can be found in the Mobilithek[17], OpenData ÖPNV[18] and Open-Data Schleswig-Holstein[19] A matching GTFS real-time feed can be found in the Mobilithek[20]. The geographical scope of this feed is Schleswig-Holstein.

3.3 Deutsche Bahn Timetables Application Programmable Interface

The *Deutsche Bahn (DB) Timetables API*[21] is a direct service of the Deutsche Bahn. It is one of the many APIs available at the API marketplace of the Deutsche Bahn[22]. While many of these APIs are potentially relevant to this thesis, most of them are not free to use. The DB Timetables API however comes with a free plan limited to 60 requests per minute. To use the DB Timetables API, a *BahnID* account is needed. When requesting data from the API, a previously generated client Identifier (ID) and client secret has to be present in the Hypertext Transfer Protocol (HTTP) headers.

Listing 3.1. Example response for the planned stops at the Bad Malente-Gremsmühlen station.

```
<?xml version='1.0' encoding='UTF-8'?>
<timetable station='Bad Malente-Gremsmühlen'>
    <s id="-5431105109679105522-2404212306-4">
        <tl f="D" t="p" o="X1" c="erx" n="21036"/>
        <ar pt="2404212334" pp="1" l="RE83" ppth="Lübeck Hbf|Bad Schwartau|Eutin"/>
        <dp pt="2404212335" pp="1" l="RE83" ppth="Plön"/>
    </s>
    <s id="8274242101897729034-2404212345-2">
        <tl f="D" t="p" o="X1" c="erx" n="21093"/>
        <ar pt="2404212353" pp="2" l="RB84" ppth="Plön"/>
        <dp pt="2404212353" pp="2" l="RB84"
```

[14] https://www.opendata-oepnv.de/ht/de/organisation/delfi/startseite?tx_vrrkit_view%5Baction%5D=details&tx_vrrkit_view%5Bcontroller%5D=View&tx_vrrkit_view%5Bdataset_name%5D=deutschlandweite-sollfahrplandaten-gtfs

[15] https://mobilithek.info/offers/755009281410899968

[16] https://gtfs.de/

[17] https://mobilithek.info/offers/766317902476267520

[18] https://www.opendata-oepnv.de/ht/de/datensaetze?tx_vrrkit_view%5Baction%5D=details&tx_vrrkit_view%5Bcontroller%5D=View&tx_vrrkit_view%5Bdataset_name%5D=fahrplandaten

[19] https://opendata.schleswig-holstein.de/dataset/fahrplandaten

[20] https://mobilithek.info/offers/766315425546817536

[21] https://developers.deutschebahn.com/db-api-marketplace/apis/product/timetables

[22] https://developers.deutschebahn.com/db-api-marketplace/

```
            ppth="Eutin|Pönitz(Holst)|Pansdorf|Bad␣Schwartau|Lübeck␣Hbf"/>
    </s>
    <s id="4353865067445023361-2404212315-2">
        <tl f="D" t="p" o="X1" c="erx" n="21041"/>
        <ar pt="2404212322" pp="2" l="RE83" ppth="Plön"/>
        <dp pt="2404212323" pp="2" l="RE83" ppth="Eutin|Bad␣Schwartau|Lübeck␣Hbf"/>
    </s>
    <s id="2175454902161277207-2404212228-6">
        <tl f="D" t="p" o="X1" c="erx" n="21084"/>
        <ar pt="2404212304" pp="1" l="RB84"
            ppth="Lübeck␣Hbf|Bad␣Schwartau|Pansdorf|Pönitz(Holst)|Eutin"/>
        <dp pt="2404212305" pp="1" l="RB84" ppth="Plön"/>
    </s>
</timetable>
```

The DB Timetables API provides timetables for individual stations. The main data structure is the `Timetable`, which consists of individual `TimetableStops`. An exemplary `Timetable` returned by the DB Timetables API can be seen in Listing 3.1. `TimetableStops` are denoted by the XML tag `<s></s>`. A `Timetable` is always assigned to a specific station. All contained `TimetableStops` are stops at that station. A `TimetableStop` consists of an arrival `Event` `<ar/>` and / or an departure `Event` `<dp/>`. The `Events` contain information like departure or arrival time, status of the stop, which can be either planned, added or cancelled, and the path of the trip before the arrival or past the departure. The `TimetableStop` also contain basic information regarding the trip they are part of, such as the trips' ID and the `TripLabel` denoted with `<tl></tl>`. The planned path (`ppth`) is provided via two strings of station names separated by pipe symbols. The first string provides the path of stations visited before the station of the `Timetable` and the second string contains all stations visited after the `Timetable`'s station. The fact that the path only contains station names rather than IDs makes it difficult to map those to the according `Timetables`, as these names are also not always consistent with names stored in the `Timetable`. `TimetableStops` also have a stop index as part of their ID, which can also be used to construct trip form `TimetableStops`. The problem hereby is, that the information regarding possibly missing stops get lost. Further, the documentation only states, that added stops get assigned a stop index above 100. It is not mentioned, how these indices are assigned in detail. Thus, it might not be possible to infer the actual position of an added stop within the path.

The DB Timetables API has one endpoint to provide scheduled timetable information for a specified station. This endpoint also requires specifying a date and an hour. It returns all scheduled stops at the station within the specified hour. This requires 24 requests per day per station to fetch, which is much considering the rate limiting.

The API has two endpoints for obtaining real-time changes to the schedule, one for all changes within a day and one for changes during past 60 seconds. The last endpoint should be used when calling the endpoint multiple times per minute. Just like the schedule endpoint, both endpoints are limited to one specified station.

Lastly, the DB Timetables API contains an endpoint for searching stations by name. This endpoint however does not work with German umlauts. The documentation does state how to deal with umlauts, but the proposed strategy does not work. Thus, stations containing umlauts in their name have to be obtained via the eva number or ril100/ds100 ID, which are internal identifiers used by the Deutsche Bahn. Multiple lists of ril100/ds100 IDs can be found online[23] [24], including one by the Deutsche Bahn[25].

The DB also provides an API for detailed station information called *StaDa*[26] in their API marketplace.

3.4 General Bike Feed Specification

General Bike Feed Specification (GBFS)[27] is the GTFS counterpart for shared mobility like bike rental stations. In contrast to GTFS, a GBFS feed consists of multiple JavaScript Object Notation (JSON) resources. Relevant resources in a GBFS feed include the following:

- `station_information.json` This resource contains static information for all stations in the feed. This includes the stations name, geographical coordinates and vehicle capacity.
- `station_status.json` This resource provides real-time status information for the available stations. This includes information such as number of available docks for returning vehicles, number of currently available vehicles and number of currently available vehicles of specific types. The latter can include ebikes, bikes, cargo bikes, etc. All existing vehicle types within a feed are specified in the resource `vehicle_types.json`.

In the scope of this thesis, one relevant GBFS feed exists for the *Kiel Region*[28] rental bikes called *SprottenFlotte*[29], which are operated by *Donkey Republic*[30]. Relevant stations for the Malente-Lütjenburg track exist in Kiel, Preetz, and Plön. The feed is available at `https://stables.donkey.bike/api/public/gbfs/2/donkey_kiel/gbfs`. The above-mentioned resources are available at:

- `https://stables.donkey.bike/api/public/gbfs/2/donkey_kiel/en/station_information.json`
- `https://stables.donkey.bike/api/public/gbfs/2/donkey_kiel/en/station_status.json`
- `https://stables.donkey.bike/api/public/gbfs/2/donkey_kiel/en/vehicle_types.json`

[23] `http://www.bahnseite.de/DS100/DS100_main.html`
[24] `https://ds100.frankfurtium.de/dumps/orte_de.html`
[25] `https://fahrweg.dbnetze.com/resource/blob/1359908/f9d782b88f2c1224ac1192e2d4b5f6ff/betriebsstellen-data.pdf`
[26] `https://developers.deutschebahn.com/db-api-marketplace/apis/product/stada/api/113501#/StaDaStationData_290/overview`
[27] `https://gbfs.org/`
[28] `https://www.kielregion.de/`
[29] `https://www.kielregion.de/mobilitaetsregion/sprottenflotte/`
[30] `https://www.donkey.bike/`

3.5 Other Sources

This section covers other available data sources, which were not used throughout this thesis due to missing time. In the geographical area surrounding the Malente-Lütjenburg track, e-scooters are available from multiple suppliers. These include *TIER*[31], *emmy*[32], and *Bolt*[33]. All of these have an own API. For the TIER API exists an official documentation[34]. Inofficial documentations for all above-mentioned e-scooter providers exist in a public GitHub repository[35] [36] [37].

The *Bürgerbus Malente* is run by a local association and provides two bus lines, which stop at multiple locations in and around Bad Malente-Gremsmühlen. The stops and live vehicle position are shown on a map on the website of the Bürgerbus Malente. The stops are obtained via a simple HTTP endpoint as JSON. Live vehicle position updates are pushed via a websocket connection. However, the Bürgerbus is out of service since the 11nth of September until this thesis was finalized. The websocket is not publicly documented and currently does not send any updates. Thus, the Bürgerbus is not further considered throughout this thesis.

A collection of public transport APIs can be found at *transport.rest*[38]. These include a wrapper around the HAFAS API of the DB and formerly also NAH.SH.

[31] https://www.tier.app/

[32] https://emmy-sharing.de/

[33] https://bolt.eu/

[34] https://api-documentation.tier-services.io/

[35] https://github.com/ubahnverleih/WoBike/blob/master/Bolt.md

[36] https://github.com/ubahnverleih/WoBike/blob/master/Emmy.md

[37] https://github.com/ubahnverleih/WoBike/blob/master/Tier.md

[38] https://transport.rest/

Chapter 4

Concepts

Traveler facing information regarding public transport is available at many different sources. Most of these sources are limited to information regarding public transport options which have a common denominator. This common denominator can be the transport agency, mode of transportation, spatial proximity or similar. In many cases, it is a combination of the aforementioned. Furthermore, many sources are not even complete within these scopes. For example, the DB Timetables API provides passenger information for most German train stations. According to the APIs description however, it is limited to stations operated by the DB Station&Service AG. Therefore, in the German state of Schleswig-Holstein, all stations operated by the Altona-Kaltenkirchen-Neumünster Eisenbahn GmbH are not included. Thus, multiple sources should be consulted in order to cover a broad variety of public transport options. Due to these circumstances, a software solution should be developed, which collects relevant public transport information from possibly various sources. The goal is to make the information accessible all at one place, ideally unified and interconnected as if all information were from the same source. The main challenges arising from this are attributed to the diversity of the data and the contained information as well as the different ways the data is made available. To address these issues, the following sections state the individual problems when gathering the information and propose a solution respectively.

4.1 Overall Architecture

In order to design a service, which interconnects, unifies and makes available information from different sources, two fundamental architectural concepts are considered.

The most trivial solution is to gather information from all sources directly in a frontend application, e.g., a mobile app. Information is queried from publicly available sources and directly displayed to the user. The benefits of this approach are obviously the simplicity and the independence. An application like this can be easily used without the need of setting up a complex infrastructure.

However, a major downside of this approach in the public transport domain is, that the provided data is in some cases not suitable for direct access to specific parts of information and might require some kind of preprocessing in order to efficiently obtain the desired information. For instance, retrieving all arrivals and departures at a specific station within a certain timeframe might require first extracting it from a schedule, as it is the case with the GTFS format. Not only must the data be present in some kind of database system to extract

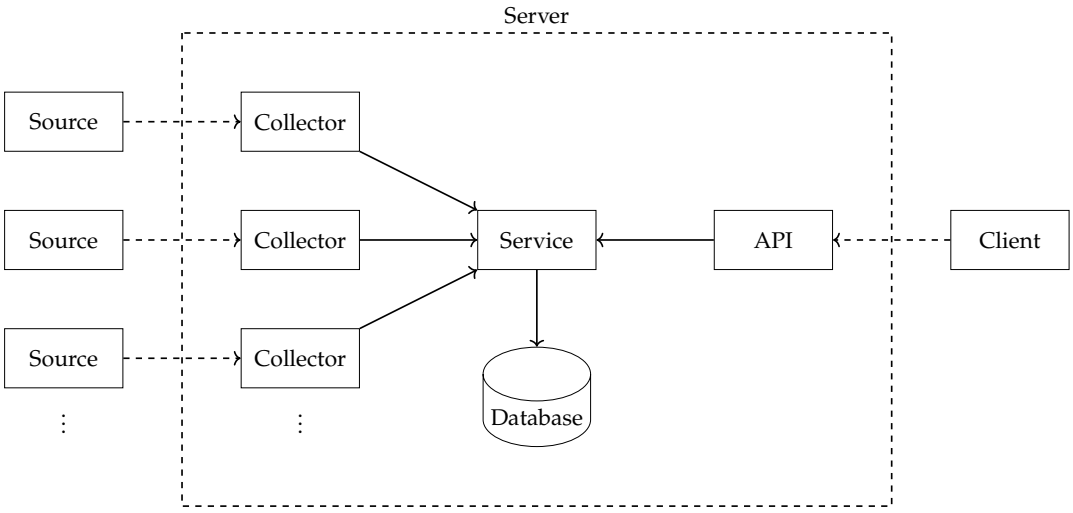


Figure 4.1. Architecture of the Application.

information with acceptable performance, in case of GTFS, the file size is also too large to download and process it on the spot. Thus, a persistence layer is required. Unfortunately, this requirement does not fit web-based solutions and rather requires a native application.

Another issue to deal with are sources that require an API key or a similar method of authentication. These keys should not be put of end-user applications. Besides that, those services tend to limit the requests per time and access key, which would also make multiple concurrent users accessing the service difficult. An example for this is the DB Timetables API. It is intended to be used in combination with a web server caching requests to the same resource. With the free plan, access is limited to 60 requests per minute per API key. Thus, each client should be required to obtain their own credentials for each source requiring authentication. This would however constitute an unwanted high barrier to using the service for the end-user.

Most importantly, this thesis focuses on making bundled mobility information available for virtually arbitrary use cases. End-user software such as mobile apps are in fact an important use case and are the main concern to the hereinafter presented concept, but other valid use cases may need to further process the information or to integrate them within other systems such as embedded devices for passenger information. Thus, the information should be provided with the uncertainty of the concrete use case in mind, which renders client-side solutions rather impractical.

The solution to the above-mentioned problems with the client side approach is to introduce a centralized server, which is accountable for aggregating information from all sources, performing any required processing of the data and providing it to client applications as a unified API. Another positive side effect to this is that adding, removing or replacing sources does not require changes to client code, as the client has no need to care about source formats and how to interpret and transform these formats. This particularly is an advantage when

multiple different clients are involved, like e.g., a mobile app and embedded passenger information device.

Due to these reasons, a client-server architecture is favored and implemented in this thesis. A frontend client is implemented as a simple demonstration to provide an easy user interface and just display the information demanded by the user as clear as possible. The client retrieves all information from the server via a minimal and uniform API. All complex tasks are left to the server. The architecture of the backend server can be seen in Figure 4.1 and consists of the following main components:

- Multiple *data collectors* gathering information from one specific source each. These sources include web APIs, external libraries, manually provided data, etc.
- A persistent database, where all information extracted by the different collectors flow together, such that relevant information can be easily queried.
- A central service, which manages the database and is used by both the API and the collectors to provide or obtain information.
- A web API enabling clients to query specific information.

The data collectors obtain data from their source, convert this data into the applications internal format and finally hand the result to the central service. The central service then processes the data semantically before pushing it into the database. On request, the web API consults the central service with the specific query. The central service then looks the desired information up in the database.

4.2 Data Acquisition

Different data sources are connected via different collectors. These specialized components exist for each data source or common format respectively. Collectors are free to obtain data in any way possible such as using an API, downloading and reading from files, scraping websites, utilizing external libraries or even by manually entered data. However, only APIs and downloaded files are implemented throughout this thesis.

A collector is responsible for regularly fetching data from the source it is responsible for. All collectors are required to implement a common protocol, so the rest of the application has no need to care for the specific collectors. This way, new collectors can be easily added the same way as existing ones can be removed or altered, without having to alter any other parts of the application.

This works for most of public transport and mobility data, but there are cases where it is not possible nor practical to collect and store all data in advance. While this could be due to technical limitations like restrictive rate limiting, there also exist sources for information impossible to fully collect in advance. This is when a source calculates its data based upon user specified parameters, where just the amount of possible parameter values is infinite or

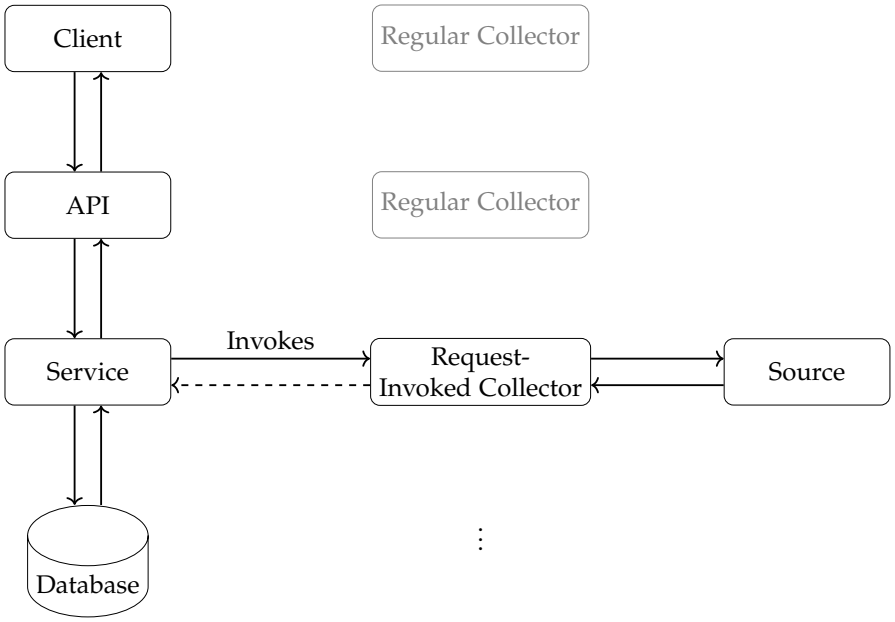


Figure 4.2. Invocation of Request-Invoked Collectors.

too large to preemptively request. An example for such sources are routing services like the Google Routes API[1] or similar APIs based on OpenStreetMap[2]. One solution to this problem would be to not utilize said sources at all and in case of calculate data, to self-calculate the data. However, this thesis aims to provide an easily extendable base for combined mobility data and strives for a broad coverage, it would be rather counterproductive to conceptually exclude sources like heavily rate-limited APIs. Although such sources are not implemented in this thesis, they might be required in other scopes and thus should be considered as well. Further, when complex algorithms are involved, own implementations might lag behind established services or re-implementation might constitute an unnecessary high expense. Thus, the proposed solution is to extend the architecture for *request-invoked data collectors*, which only get invoked on API requests as shown in Figure 4.2.

4.3 Public Transport and Mobility Domain Model

Regardless of source, format or the like, conventional traveler facing public transport information always consists of *stops*, *lines* and *trips*. A stop is anything where public transport vehicles stop at, like a bus stop, a train station or even a specific platform at a train station. A line (or route in GTFS) bundles similar trips under a short name conveying meaning to travelers. An example of this could be a train line with a name such as RE83, which serves the same route each hour. A trip is a concrete instance of such. It is assigned to a specific line and consists of

[1] https://developers.google.com/maps/documentation/routes

[2] https://wiki.openstreetmap.org/wiki/Routing

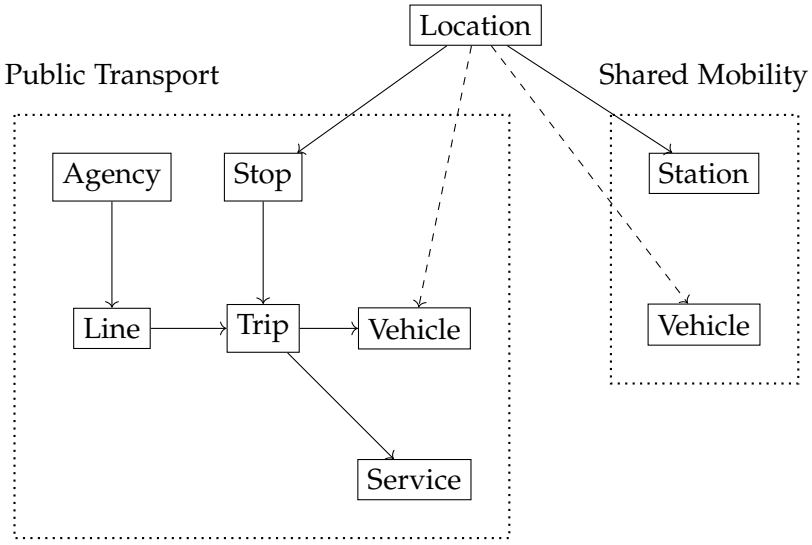


Figure 4.3. Public Transport and Mobility Domain Model.

a sequence of stops. Associated with each stop are the times of day the vehicle serving the trip is expected to arrive and depart according to plan. This is usually extended to real-time information, which contain information such as delay times, cancelations or the like.

The domain model used throughout this thesis is shown in Figure 4.3. It strongly resembles the GTFS model as it is a de facto standard, and designed to fit a broad variety of public transport options. Next to the core entities of stops, lines and trips, the developed model also includes *transport agencies* and *services*, which are similar to those in GTFS. Each line is operated by a transport agency. Each trip is assigned a service, which specifies the days a trip is available. Next to regular real-time information, a trip can also have a concrete vehicle assigned, which serves the trip. To also support shared mobility, this model is extended by shared mobility stations and vehicles. Shared mobility stations represent a fixed location, where vehicles like bikes can be rented. Shared mobility vehicles represent individual vehicles, which can be rented from the current position they are parked at, independent of any station. An example for this are e-scooters. Both stops and shared mobility stations are assigned fixed geographic coordinates, vehicles and shared mobility vehicles have dynamically updated positions.

In the context of this thesis, it is also differentiated between trips and concrete *trip instances*. Opposed to a trip, which itself is detached from specific dates of operation and more like a reusable pattern, a trip instance is a specific instance of such a trip at a specific day. This allows trip instances to also be enriched with possible real-time data. Trips are also optionally assigned to *shapes*. A shape specifies the exact geographical path a trip takes, with a way higher resolution than the path formed by the positions of each stop.

4.4 Uniform Internal Representation of the Data

Since multiple information sources by different providers are involved, the data naturally has a high diversity in representation, density and quality. Therefor, it is necessary to preprocess the obtained data accordingly in order to achieve uniformity and integrity. The overall goal of this is for all data of all sources to become compatible to each other semantically as well as in representation. In order to reach this goal, the data needs to be converted into a uniform structural representation first. This internal representation should be independent of the original data formats. Following the idea of making the main backend application unaware of the concrete source formats, this task has to be done by the collectors. Only when this is accomplished, the data can be handed to the central service for further processing and finally insertion into the database.

Simply converting the data into a uniform representation is rather straightforward opposed to the semantic harmonization, although it may involve converting concrete instances of trips into a schedule or vise versa. In this case, it was decided to store schedules rather than concrete trip instances for space consumption reasons.

4.5 Cross-Source Data Merging and Harmonization

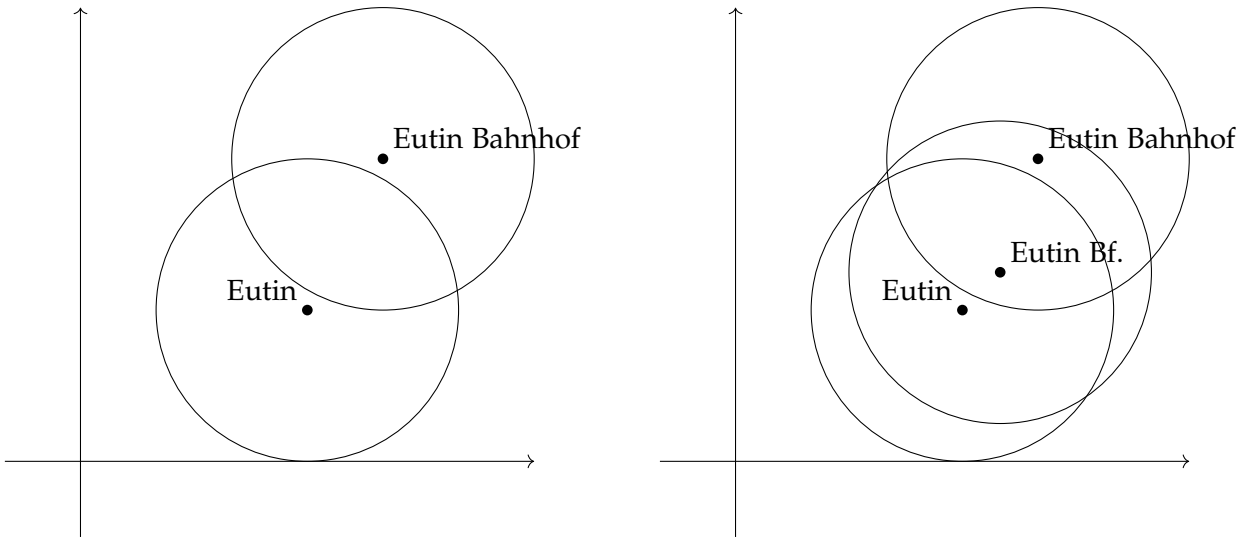
When considering multiple information sources, it may occur that these sources overlap. For instance, a bus stop or train station might be present in more than one sources. The challenge arising from this is to identify these overlapping subjects and to merge them in a sensible way. Therefor, it is distinguished between individual records and subjects, which each record has, but multiple records can also share the same subject. For instance, two records from different sources might exist for the train station Bad Malente-Gremsmühlen. In this case, both are individual records, but the subject of both is the Bad Malente-Gremsmühlen train station. In order to provide an easy way of tracing an information back to its origin and to make it easy for collectors to update their data without having to deal with other collectors' data, this separation is also reflected by the database. Thus, the identification of records with same subjects and the actual merging of said records is split.

4.5.1 Duplicate Identification on Insertion

Identification of records regarding the same subject can be done during query, periodically or on insert. Firstly, identification during query is the most flexible, as the criteria for when records are considered the same subject can easily be changed with each query. With this approach, identified subjects do not need to be present in the database. Also, the number of records to check is quite low, as they can be filtered to match the query first. On the other hand, this approach would negatively impact the query time. In order to keep the increased time low, advanced preprocessing like clustering has to be done. Moreover, the high flexibility of this approach is not that big of a benefit, as the criteria for what is e.g., considered the same stop is only expected to change during development to find suitable thresholds. Thus, this would

not really constitute a benefit to the end user. Secondly, periodically performed identifications are not impacting the query performance, but when new records are inserted, query results might be suboptimal until the next identification of same subjects. Lastly, identifying subjects on insertion affects only insertion performance, but not the query performance. Thus, it is not affecting the end user. In contrast to the periodically performed identifications, subjects are identified as soon as possible. This makes it less likely that two records, that should have been merged, are returned to the end user as two separate entities. This approach is favored in this thesis over duplicate identification during query or periodically performed identifications over the whole database due to the aforementioned reasons. A hybrid or cached approach is also possible, but was not chosen due to the increasing complexity.

The downside to this approach compared to all others is however, that not all records with potentially the same subject are known during insertion. When implementing this, one has to be aware, that the subjects of previously inserted records might change due to later inserted records. For instance, when inserting two records into the database, they might be found to be unrelated. Then, when a third record gets inserted, it might be found to regard the same subject as both the records inserted before. At this point, all three records are known to regard the same subject due to the transitivity of the equals-relation. To reflect this in the database, at least one of the existing records has to be updated.



(a) Two stops just too far apart to be considered equal.

(b) Insertion of a third stop, causing all three stops to be within radius to be considered the same.

Figure 4.4. Example of two stops considered distinct until insertion of another stop.

To give a practical example of this in the public transport domain, when inserting stops and identifying same stops only based on spatial distance, it could happen that two stops with just enough distance between to be found unrelated are inserted like shown in Figure 4.4a.

Here, first *Eutin* and *Eutin Bahnhof* are inserted from different origins. Both stops represent the same train station, but due to the relatively far distance between both stops, they are considered different stops. When a third stop is then inserted spatially in between, it is found to be the same stop as both the stops inserted before due to the short distance to these stops. Thus, all three stops should be marked as the same stop. This can be seen in Figure 4.4b, where the stop *Eutin Bf.* is inserted from yet another origin, which also represents the same train stations as both previously inserted stops. Let D be the set of all found duplicates including the one to insert. When all elements in D are from distinct origins, they can all be marked as the same subject. Otherwise, as it can be expected, that the same source does not include the same subject twice, it is not sensible to consider those the same.

In a bigger database, this could cascade through a lot of records. For simplicity, this is ignored during this thesis. Only the newly inserted element gets marked as the same subject as the found duplicate with the highest similarity. Note, that this might lead to suboptimal results in some cases. Also, it is possible due to this to get slightly different results based on the order of insertion. However, since subjects of records will not be changed after insertion, this also enables manual overriding subjects, without having to treat manually set subjects separately. Manual overriding is an important aspect to the developed concept, as the varying quality, density and completeness make it impossible to guarantee correct results when automatically identifying subjects. Thus, it is necessary to provide an easy option to override the detected subjects, without getting reset during each update. As this thesis targets a regionally limited area, these manual adjustments are feasible and subtle improvements to automatic processes might not be as useful. If the developed concept was extended to a wider scope, it might be worth to implement the above described behavior. With this simplified approach however, when a records subject is found to be duplicate upon insertion, the following cases apply:

- An existing record from the same source, which does exactly match gets replaced, as it is considered as an update.
- An existing record from the same source, which does not match exactly, is ignored and the new record gets inserted with its own subject. This is due to the fact, that two records regarding the same subject are not expected to be present in the same source, unless it is an updated version. In the latter case however, it can be expected, that the updated version can be identified exactly.
- Otherwise, the record is inserted next to the existing ones, but the newly inserted record gets assigned to the same subject.

The only exception to this is when two trips from the same origin are identified to be equal. If this applies, both trips are merged into one in the database. In this case, also the services assigned to both trips are merged. Two trips from the same origin can be detected as equal via a third trip from another source, if this is again equal to one of said trips.

4.5.2 Duplicate Identification Criteria

This section briefly points out the different characteristics of each entity in the public transport domain model suitable to identify records likely to regard the same subject. Note, that these identifications are mostly best-effort and correctness can not be guaranteed due to inaccuracies, errors or just slightly different naming in individual data sources. Thus, it might be necessary to manually correct individual records. In order to test two entities of the same kind for equal subjects, a similarity is computed for each considered characteristic. Those similarities are weighted according to their estimated significance, resulting in a single similarity between 0 and 1 for the tested entities. If this values surpasses a certain threshold, both entities are considered the same.

Stops In Figure 4.4, stops where used as an example on how identification of equal subjects is done in general. Therefor, the example used a simplified approach, where only spatial distance is considered to decide whether two stops are equal. However, this might lead to undesired results in reality, as two physically distinct stops might exist directly next to each other. For example, central bus stations are often located next to train stations. Also, geographical locations for the same larger central station provided by different sources might have a higher distance than the actual distance between two physically distinct bus stops. Thus, when only considering spatial distance, either the central station will be incorrectly identified as multiple distinct stations, or the two distinct bus stops will be incorrectly identified as the same. Thus, equal stops from different sources are identified by both geographical position and name. If the spatial distance between two stops from different sources is less than a few meters and their names are very similar, they are very likely to be the same stop. The similarity of names is determined using the Levenshtein distance, Wandelt et al. [WWS16] implemented a very similar approach in order to simplify a skeleton of the worldwide railway network obtained from OpenStreetMap (OSM). As differences in naming for the same stop follow common patterns for abbreviations and synonyms most of the time, the Levenshtein distance algorithm is supplemented by the recognition of those common patterns. For example, *Kiel Hauptbahnof* and *Kiel Hbf.* are considered to be exactly the same, as the abbreviation *Hbf.* for *Hauptbahnhof* is recognized.

It is however important to note, that different platforms within a station are also very likely to have a similar name and short distance in between, but these should not be considered equal. Thus, two stops are only considered equal, if both their platform codes - if set for at least on of them - are exactly the same.

Agencies Duplicate agencies are mainly identified using their name, as two agencies with the same name are very unlikely. If agency websites, email addresses or phone numbers are provided, those can be used to distinguish agencies with the same name, if ever needed. They can also help to identify agencies as the same, if the name varies too much.

Lines The easiest criteria for identifying same lines is the name of the line. But this alone is not sufficient, as different lines with different names exist. It is however very unlikely that two lines with the same name are operated by the same agency. Line names must match exactly, as just one different letter or digit will very likely describe a completely different line. This is due to the fact that line names are by design very short and often just 1 to 4-digit numbers. For instance, the train line RB84 with terminal station Lübeck Central Station via Bad Malente-Gremsmühlen is a completely different line as the RB85, which also has Lübeck Central Station as a terminal station, but does not stop in Bad Malente-Gremsmühlen. However, name matching is case-insensitive and all non-alphanumeric symbol such as spaces or dashes are ignored, as those typically do not convey any meaning. For example, in the GTFS dataset for the German state of Schleswig-Holstein, contains 98 lines with non-unique names, but there do not exist two lines with the same name, which are also operated by the same agency. In some special cases, it might be necessary to also consider concrete trip routes, but this is also more complex and unlikely to be necessary.

Trips Trips are the most complex to identify. The most meaningful identification characteristic for a trip is the path of stops and corresponding stop times. Considering possible deviations of different data sources, it might be useful to not compare paths for exact equality, but rather compute a similarity and decide whether it is above a certain threshold or not. This similarity might be computed based on - when not exactly the same - spatial distance between stops, stop names and most importantly stop times and order of stops. A practical example of when this might be useful are rail replacement buses. These are sometimes listed to stop at the railway station the replaced train usually stops at, but sometimes also at nearby bus stops. This might also vary between sources.

4.5.3 Merging during Query

Multiple data records found to regard the same subject are merged when queried based upon the previously identified subjects. This allows for excluding certain sources from the query. Also, not merging the data until queried allows the collectors to easily update the data, as they don't have to care about other collectors' data. The data is merged based on the subjects identified beforehand as described in Section 4.5.1.

As mentioned earlier, the only reason for having to merge data, is that different information sources might overlap. These sources might also have different levels of quality. For instance, one source might cover a very broad range of public transport services, but might have poor quality in that specific information such as platform codes might be missing or inaccurate. Another source however might be very specialized towards one public transport service, but have a high quality. Such circumstances make it sensible to prefer certain sources and use others just as a fallback. This allows to cover both a broad variety and also high quality information where available.

In order to implement this in the developed application, each source is assigned a priority. When merging data from different sources regarding the same subject, these priorities are

used to decide which information to keep. For each property, the value from the source with the highest priority, where the property is set, is kept in the resulting merged data. For example, when merging two stops which where identified as the same, but only the stop from the source with the lower priority has geographical coordinates assigned, these are kept. However, the name is present for both stops, thus the name of the stop from the source with the higher priority is kept.

4.6 User Story

The main use case of the developed service are traveler or passenger facing information systems of any kind, possibly including mobile apps, embedded devices at stations or on-board of vehicles which for instance display next departures or connection trips. The latter can be generalized as some kind of location-bound departure or arrival boards. In order to cover a broad range of mobility options, the user might find it helpful if these boards do not only include arrivals and departures at one specific stop, but also include those at nearby stops within walking distance. Displaying these arrival and departure boards is also a main use case to a possible mobile app. As a main goal is to provide small, regional, or experimental mobility and public transport projects a service to integrate information regarding adjacent mobility options, the average user can be expected to be mainly interested in mobility options near their current location or within a certain proximity to another mobility option.

Due to the aforementioned expected use-cases, the API is designed around geographic positions rather than stops or stations. This way, nearby mobility options can be discovered easily without having to search individual stops.

Following this decision, the main API endpoint returns all available mobility options at a specified geographical location within a certain radius and time frame. Also, the option to apply other filters might be useful. A response to a successful query of this endpoint might include nearby stops, lines regularly serving these stops, shared mobility options and most importantly trip instances. The information provided by this endpoint should be complemented by another endpoint providing real-time updates such as deviations from the schedule. The response should also link to the same endpoint at the location and time of arrival for each stop of each trip, so the user can easily navigate through trips and locations. This interaction is visualized in Figure 4.5.

When it comes to mobile apps, one of the most important features of such apps in the public transport domain is routing. Most of the time, users of such apps want to figure out how they can reach a certain destination from a starting point. Thus, it would be beneficial for such use-cases to provide an endpoint, which finds routes from a starting point to a desired destination, utilizing the available mobility options as efficient as possible. This however is beyond the scope of this thesis.

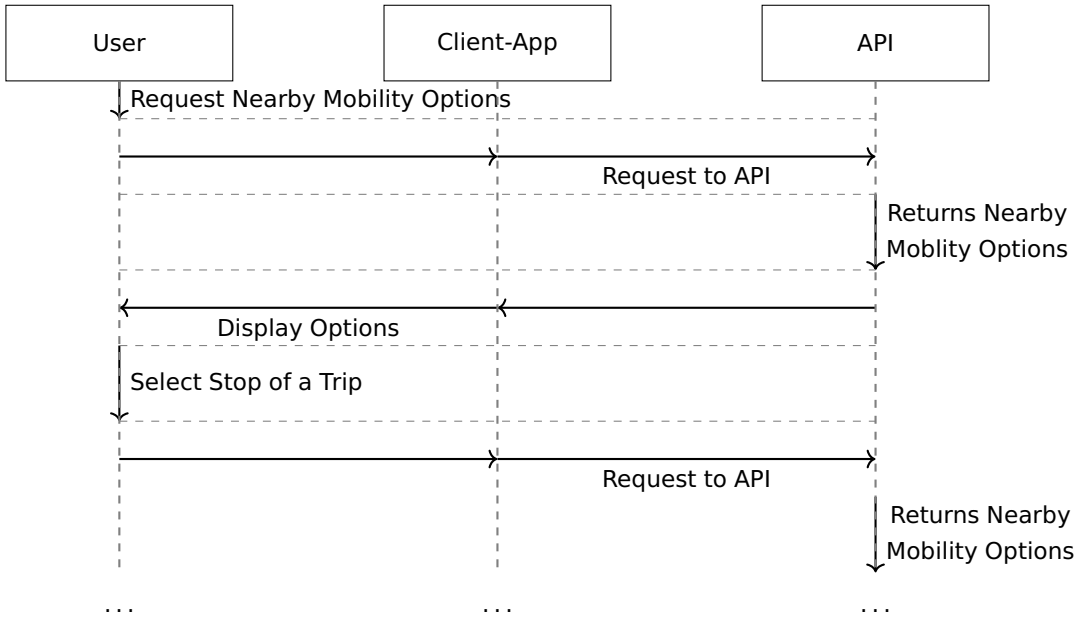


Figure 4.5. Interaction of a user with the API via a client app.

Chapter 5

Implementation

This chapter explains how the concepts discussed in the previous chapter where implemented.

5.1 Project Structure

The project mainly consists of a database server, the main backend web server, a map tile server for the frontend and lastly the web frontend itself as shown in Figure 5.1. For the database server, *PostgreSQL*[1] was chosen due to its performance and broad variety of useful features. It also has a fairly good documentation and is widely used. The frontend is directly hosted by the backend for simplicity. All components are orchestrated via *docker compose* and can be started using the `docker compose up` command. The frontend web application is a minimal exemplary consumer of the API provided by the backend server. It communicates with the backend via a Representational State Transfer (REST) API for static data and Server-Side Events (SSEs) for real-time updates.

[1] https://www.postgresql.org/

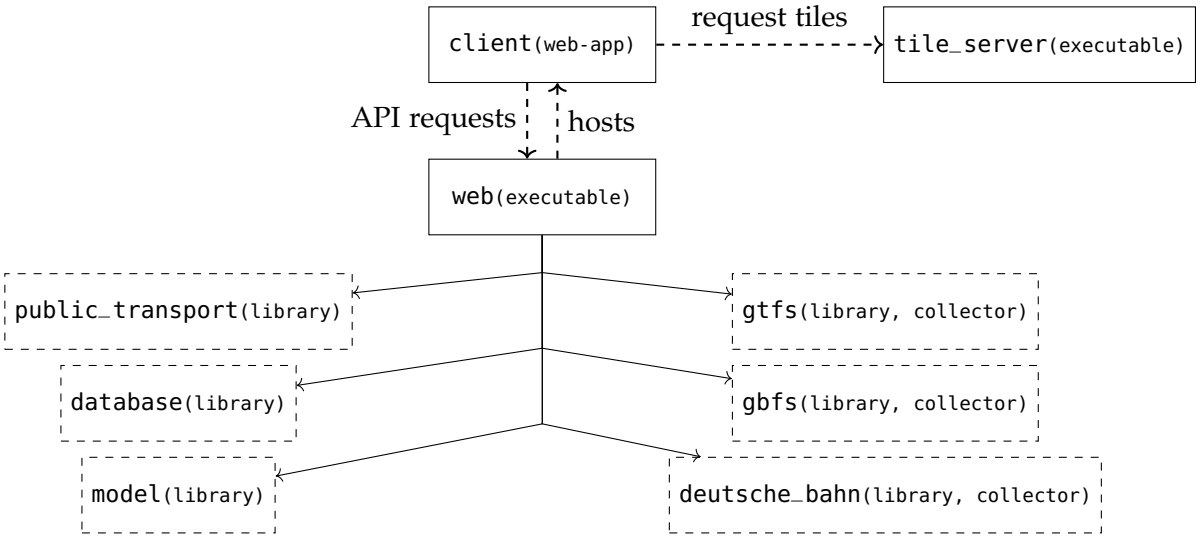


Figure 5.1. The application structure.

5.2 Backend

The backend server is implemented in the *Rust programming language*[2] with *axum*[3] for serving the API and *SQLx*[4] for accessing the PostgreSQL[5] database. Rust was chosen for its performance and static code analysis features, helping with the implementation of a fast and reliable service. It is important to the following, to understand some core aspects of rust. Rust traits are similar to interfaces in conventional object-oriented languages. Conventional classes do not exist in rust. However, rust has structs, which are somewhat like classes, but the pure definition of the data and the implementation of the associated method is a bit more separated.

5.2.1 Architecture

This section explains how the general architecture proposed in Chapter 4 was implemented in code.

Common struct definitions shared between backend and potential clients are implemented in the crate `model`. The API, which is basically a wrapper around the main backend library, is implemented in the crate `web`. This crate is also the main entry point of the backend application. The main backend library is implemented in the crate `public_transport`. It consists of the following components:

Client The *client* provides the functionality of the central service introduced in Chapter 4. While it is implemented decentralized for performance reasons, it coordinates database accesses like expected from the central service.

Database The database just consists of trait definitions. This way, the main backend is independent of a concrete database. The actual database implementation is externalized to the crate `database`.

Collectors The collectors are realized as a rust trait, which each collector has to implement. This way, the main backend application has no requirement to care about individual collectors. In order to add a new collector, it just has to implement the collector trait, an excerpt of which is shown in Listing 5.1. Besides the actual data gathering logic of the collector, it also provides methods to specify behavior for recovery on failure and on panic. By specifying the `State` type, the application knows how serialize and deserialize the collectors' state, to store it in the database as JSON. To activate an instance of this collector, the settings for the instance have to be inserted into the `collectors` table in the database. This way, multiple instances of the

[2] `https://www.rust-lang.org/`

[3] `https://github.com/tokio-rs/axum`

[4] `https://github.com/launchbadge/sqlx`

[5] `https://www.postgresql.org/`

same collector with different settings are possible at the same time to allow gathering data from e.g., multiple different GTFS sources.

Listing 5.1. Excerpt of the collector trait.

```rust
#[async_trait]
pub trait Collector {
    type Error: Debug;
    type State: Debug + /* ... */;

    /* ... */

    /// This method is regularly called and supposed to gather source data,
    /// convert it and push it to the database.
    async fn run<D: Database>(
        &mut self,
        client: &Client<D>,
        state: Self::State,
    ) -> Result<(Continuation, Self::State), Self::Error>;

    /// Specifies how long to wait between calls to the ‘run‘ method.
    fn tick(&self) -> Option<Duration> {
        Some(Duration::from_secs(10))
    }

    /* ... */

    /// Specifies the behavior for when the collector returns an error.
    fn on_error(&self, _error: Self::Error) -> SupervisionStrategy {
        SupervisionStrategy::Resume
    }

    /// Specifies the behavior for when the collector panics.
    fn on_panic(&self, _error: Box<dyn Any + Send>) -> SupervisionStrategy {
        SupervisionStrategy::Restart
    }
}
```

5.2.2 Database Design

As the chosen database server is PostgreSQL, the implemented database is a relational database. This fits the conceptional decision of using a very GTFS-like model well, as a GTFS feed also resembles a relational database.

Tables

Each entity from the model presented in Section 4.3 has its own table in the database. The central tables contain an identifier column and an origin column, hinting the source from which the record originates. The concept of records with possibly shared subjects introduced in Section 4.5 is implemented in the database by allowing multiple records with different origins to share the same identifier. Thus, the combination of the identifier and the origin makes the primary key of most tables. This way, to mark multiple records from different sources as the same subject, they just need to be assigned the same identifier. When a subject is queried from the database, it is queried only based on its identifier. Then, all returned records are merged with the priority of each origin.

For most tables, there also exist separate tables, in which identifiers used in the source datasets are mapped to the internal identifiers, in order to provide an option for collectors to obtain records in the database by the original identifiers. These can be used when translating references in the source dataset to the internal representation. The identifier mappings are also used to replace records with newer versions. By using the original identifier to match records on update, it is impossible, that a new record is created by accident.

The following provides an overview of the most important tables. All tables are shown in Figure 5.2.

- `origins`: Contains an entry for each data source.
- `collectors`: Contains all collector instances and settings as JSON.
- `agencies`: Contains all public transport agencies. The contained information includes the name, a website, an email address and the like.
- `stops`: Contains all stops. This table holds information such as a stops name, geographical location and kind (e.g, station, platform etc.)
- `lines`: Contains all lines. Lines have a name and reference an agency, which operates the line.
- `trips`: Contains all trips. Containes information such as the assigned line, thus references the `lines` table. It does however not contain any stops sequences or the like. Just as in GTFS, all stops visited are stored the `stop_times` table. The `trips` table references the `stop_times` table and via a specified service the `calendar_windows` and `calendar_dates` table.
- `stop_times`. Like in GTFS, this table contains the sequence of stops and the associated times for each trips.
- `calendar_windows` and `calendar_dates`. These tables specify the availability of a service, just like in GTFS.

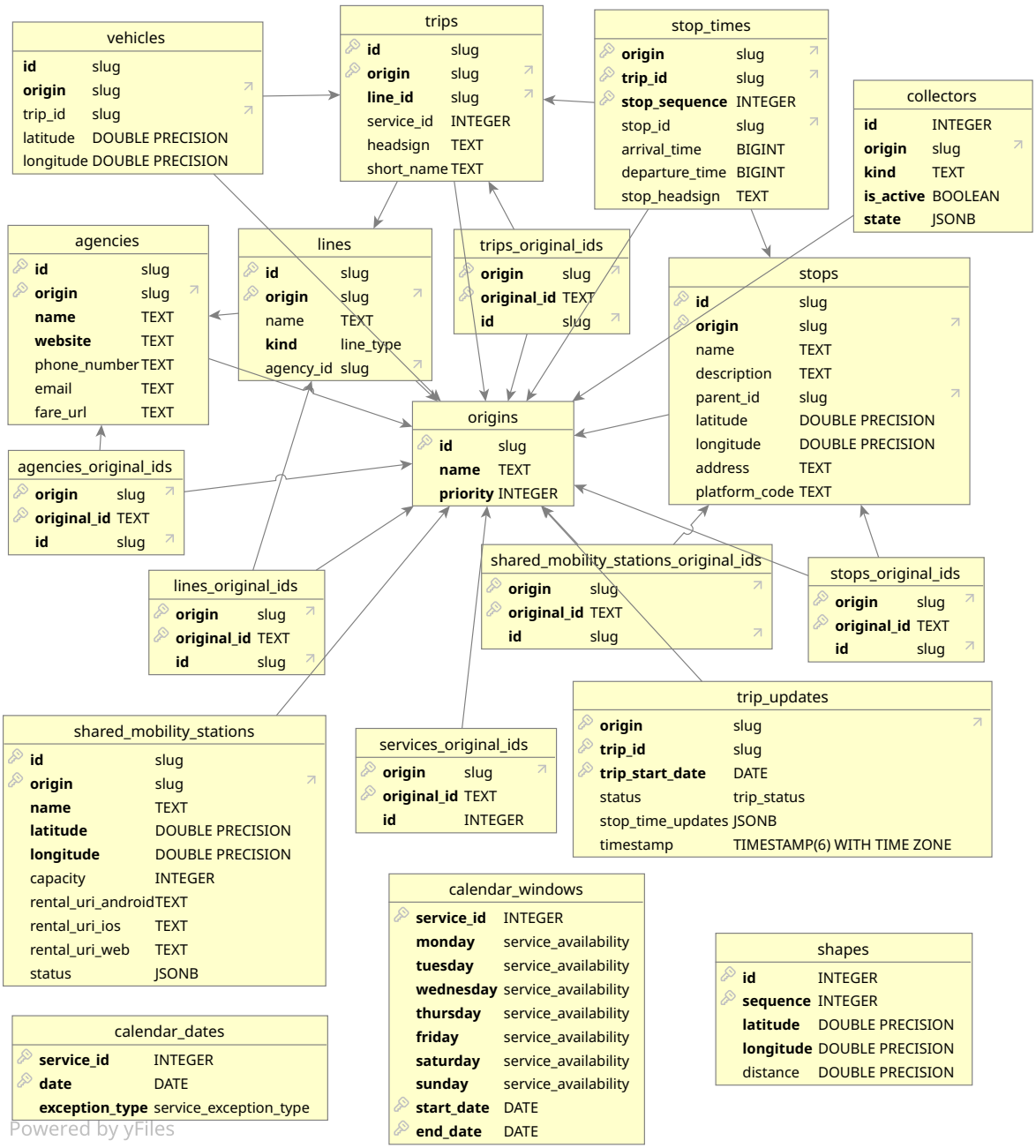


Figure 5.2. Diagram of all tables in the database.

Queries and Indexes

Next to primary keys, all columns, which are regularly used in queries, are indexed accordingly.

In order to provide an efficient text search for stations, the name column of the stations table is indexed using the PostgreSQL extension `pg_trgm`[6], which enables trigram search. This way, stations with similar names to specified text values can easily be queried and ordered by their according similarity. Trigram search is used to implement the stop search endpoint as specified in Section 4.6 as well as for fetching candidates for duplicate identification when inserting a new stop. When inserting a new stop, these candidates are also filtered on spatial distance between the geographic locations.

Geographical locations for e.g., stops are stored as two simple floating point values for the latitude and the longitude. Thus, they are indexed using a normal non-unique index. When searching for stops within a radius around a geographical point, this is entirely done in the Structured Query Language (SQL) query. Whether a stop is within the specified radius is determined by the haversine distance between the stops location and the radius' center point. Before performing the expensive computation for the haversine distance, the stops are filtered based on a bounding rectangle, as this is cheaper to compute. This approach works very well with the data used throughout this thesis. However, if even better performance for querying stops within a certain radius is required, the locations of the stops can be stored as a *PostGIS*[7] point and thus indexed using PostGIS. The stops can then be queried using the PostGIS functions.

5.2.3 Merging the Data

Each model definition in the `model` crate, which allows for sensible merging of instances, implements a respective trait. When multiple of those records are fetched from the database, they are returned in a `DatabaseFrame`. This data structure allows to individual access of those records, but also provides a method to merge the contained records if wanted. When calling this method, one can specify which origins to include. The order of the origins defines their priority. Examples for such models are stops, trips or lines. There are also models, which will not be merged. Instead, the value from the origin with the highest priority will be used, where the value is actually set. This includes shapes.

5.2.4 Instantiating Trips

In this thesis, trips are treated like a pattern. A trip mainly consists of a path of stops with associated stop times. Such a stop time is just a time of day, without a specific date. A trip is associated to a service, which specifies the concrete days a trip is available at. This way, a lot of storage is saved, as most trips follow a pattern and thus do not need to be stored for each

[6] `https://www.postgresql.org/docs/current/pgtrgm.html`

[7] `https://postgis.net/`

day. Users however are mostly interested in instances of such trips at specific dates. Thus, when trips are queried from the database, they must be instantiated to specific dates. This is done by fetching all possible trips within the requested time frame and iterating through the service dates, which fall into the specified time frame. For each of those days, a new instance of the trip is then created. During this process, it is also decided whether the trip really matches the query, as the database might return too many trips for performance reasons.

5.2.5 Serving the Application Programmable Interface

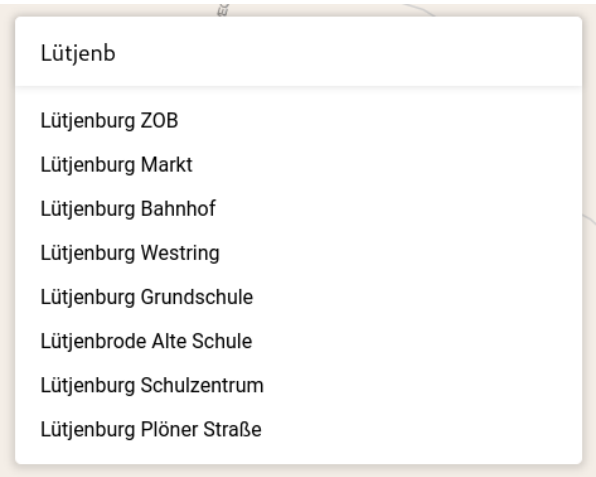
The API is implemented using axum. As described in Section 4.6, it consists of mainly one endpoint to discover everything near a certain point. Live updates are provided via SSEs. Only for this reason, the response of the main API endpoint features IDs. In order to fetch more detailed information for a specific thing or to e.g., navigate to a location a certain trip stops at, the response data contains direct links to the respective resources. This is done to avoid that the consumer has to deal with the API structure such as worrying which data is available at which endpoint other than the main endpoint. The concept used to implement this feature is called Hypermedia as the Engine of Application State (HATEOAS). The API is implemented in the `web` crate and is basically a wrapper around the main backend service. The most important endpoints are the following:

- `GET /api/v1/nearby`. This endpoint is the main endpoint to discover everything near a given point. The geographical location of this point can be specified using the query parameters `latitude` and `longitude`.

- `GET /api/v1/realtime/nearby`. This endpoint provides real-time updates for a nearby query. It works exactly as the `nearby` endpoint, but returns a SSE connection, over which real-time updates are pushed towards the client. The client has to match updates to the according trips using the IDs present in both the updates and the data obtained from the `nearby` endpoint.

- `GET /api/v1/stops/search/SEARCH_PATTERN`. This endpoint returns stations, where the name matches the specified pattern. The pattern can be a part of the stops name or the full name. It also has a tolerance for typos. `SEARCH_PATTERN` is a placeholder for this pattern. The results are sorted by similarity to the provided pattern. This endpoint is intended to be used for implementing e.g., a search box and also to provide autocomplete capabilities. An example of a search box utilizing this endpoint can be seen in Figure 5.3, where suggestions for the search pattern *Lütjenb* are shown.

All endpoints are automatically documented with JSON schemas. The JSON schemas are automatically inferred from the Rust types. Documentation comments in the Rust code are also included as field descriptions in the generated JSON schemas. For each endpoint, the JSON schema of the response is available by appending `/schema` to the address of the endpoint.



.

Figure 5.3. Suggestions provided by the stop search endpoint when searching for the pattern *Lütjenb*

5.3 Fronted Implementation

The frontend is a very minimalistic website, which just serves as an exemplary client of the provided API. It is implemented using Hyper Text Markup Language (HTML), Cascading Style Sheets (CSS) and JavaScript. The UI concept is very simple, as it is just a map, where the user can click on a certain location to view nearby departures, arrivals and shared mobility options. The UI also features a search bar, where the user can search for specific stations or bus stops. This can be seen in Figure 5.3. The map is rendered using *MapLibre*[8]. On startup, the user's actual location is fetched via the browser API and focused. When a location is focused, no matter whether by manually clicking on it, searching for it in the search bar or due to the location fetched on startup, the sidebar on the left shows all arrivals and departures of nearby public transport options as well as shared mobility. This is shown in Figure 5.4 for the example of Plön. The sidebar displays all lines stopping nearby, all arrivals and departures at nearby stops and the bike rental station at the train station. In Figure 5.4, one can also see that real-time information for the displayed train lines are available. These are displayed as bold text next to the scheduled arrival and departure times. The involved stops are also shown on the map. Users can also click on trips to see detailed information about the trip's path. This is shown in Figure 5.5. When the user clicks on a specific stop on the trip's path, the UI navigates to the location of that stop at the exact time of the planned arrival of the trip at that stop.

[8] https://maplibre.org/

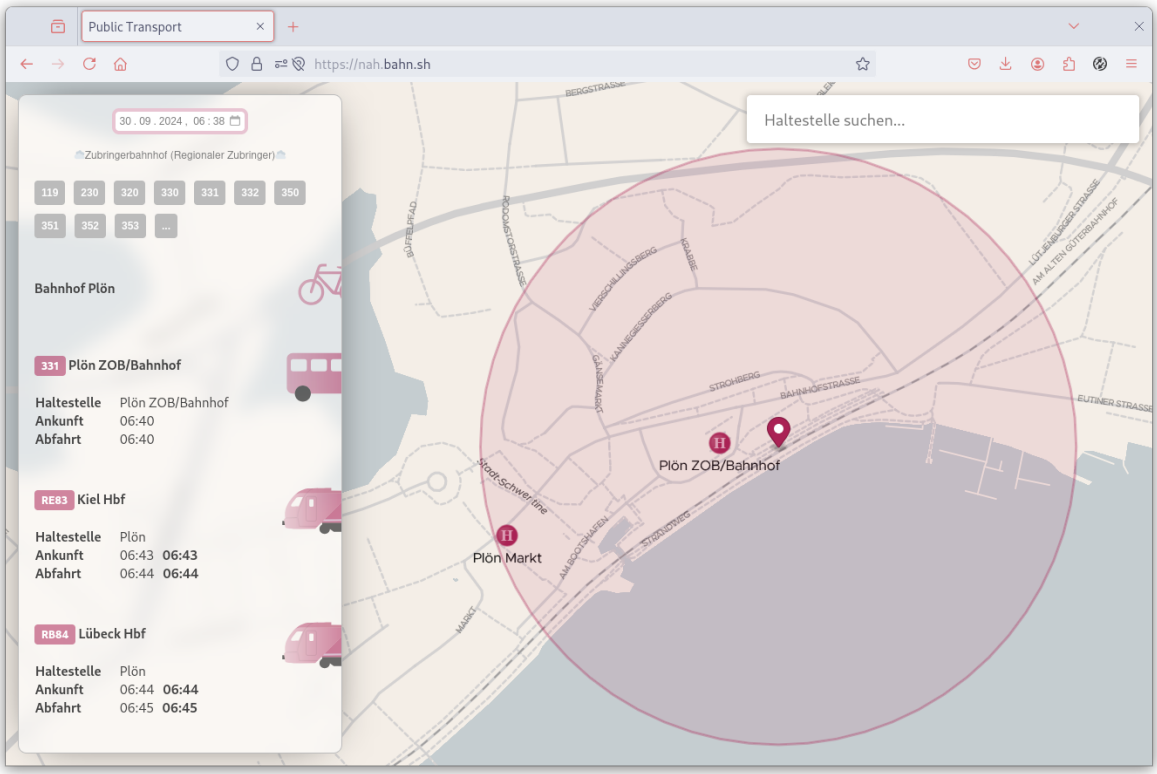


Figure 5.4. The UI with Plön focused.

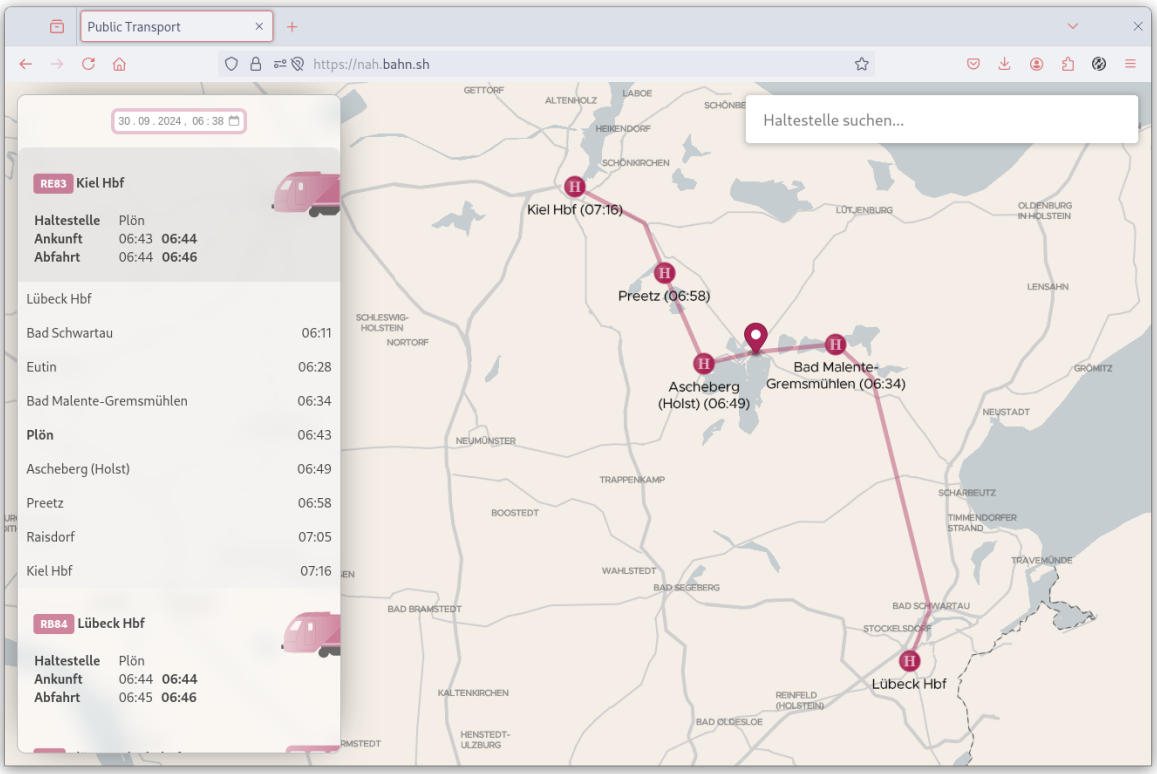


Figure 5.5. The UI with a trip of the train line *RE83* selected.

Chapter 6

Evaluation

This chapter evaluates the developed concept and implemented software. First, Section 6.1 compares the implemented application to existing solutions. Then, Section 6.2 presents some restrictions implied by the taken conceptual decisions. Section 6.3 briefly evaluates the source data and thus the implied quality of the information provided by the solution developed throughout this thesis. Lastly, Section 6.4 elaborates on the technical efficiency and performance of the implemented solution with a focus towards the main API endpoint.

6.1 Comparison with Existing Solutions

Compared to the broad variety of existing traveler facing public transport information systems, the features provided by the solution developed throughout this thesis are rather limited. Good routing features are probably the most important aspect of those systems and especially mobile apps, as they are perfectly suited to plan trips in advance. Most of the established solutions like the NAH.SH app and website come with a strong routing feature like shown in Figure 6.1. The implementation of such features was not the focus of this thesis and beyond scope. It is however perfectly possible to extend the developed software by such features. Most existing software will also display more detailed information such as ticket prices. This is, as long as provided by the consulted information sources, an easy addition.

6.2 Conceptual Limitations

While the client-server architecture constitutes an easy foundation for developing further clients utilizing the same data, it also comes with some limitations. A concrete example of this is the integration of renting a shared mobility vehicle or buying tickets directly in an end-user app utilizing the implemented API. Ideally, this would be directly possible via the API, as the advantage of the uniformity of the data provided by the API would get somewhat lost, when tasks like this have to be done by the client, as it would then require to implement separate solutions for each service in the client nevertheless. But integrating the purchase process within the API can be challenging for security reasons. To avoid directly sharing user credentials with the backend server, an access token would be required. Thus, all involved services must provide such an option. Further, even when sharing such an access token with a third-party server, the user suddenly has to trust the server not to use the access token in unintended ways.

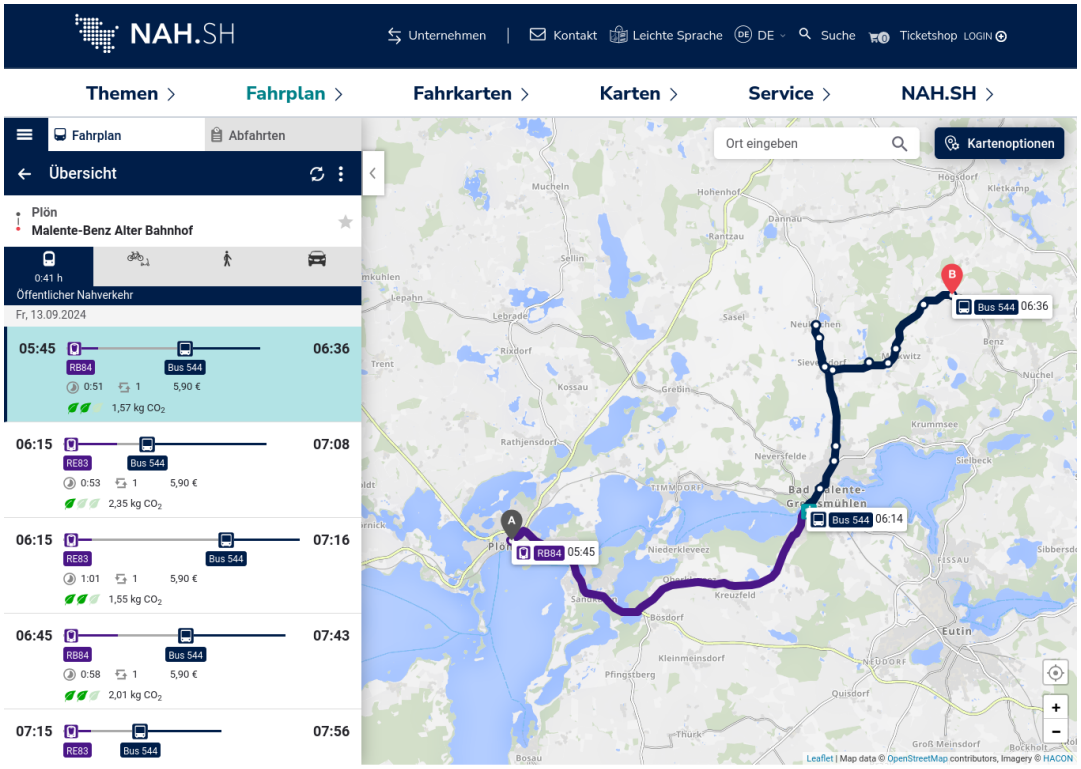


Figure 6.1. The routing feature of NAH.SH.

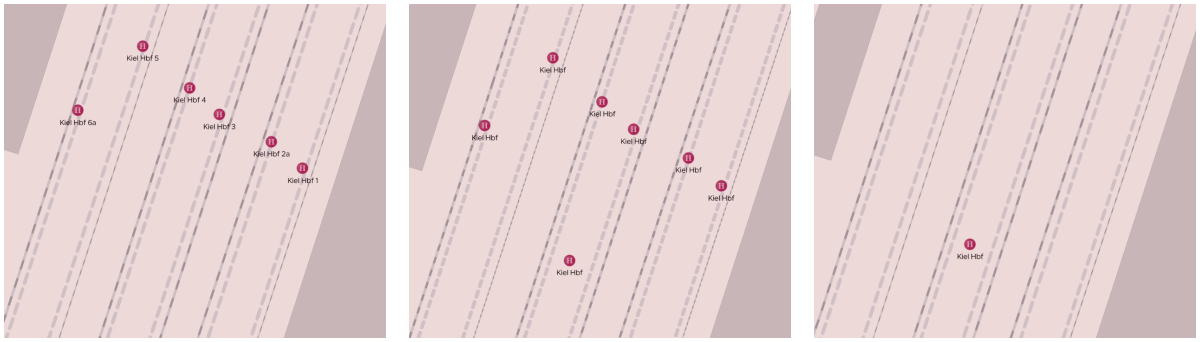
6.3 Quality and Consistency of the Source Data

The quality of the provided service is limited by the data it is built upon. In the process of writing the thesis, multiple information sources regarding public transport options in Germany and Schleswig-Holstein where considered. These vary a lot in quality and completeness.

When it comes to GTFS feeds, there exist three options covering the target region around the disused Malente-Lütjenburg track. These feeds are compared in Figure 6.2 and Figure 6.3 using two selected examples. First, there is the GTFS feed for the whole of Germany by the DELFI. According to a report by the *MITFAHR | DE | ZENTRALE*[1], validating different German GTFS feeds using the tool Mecatran GTFSVTOR[2], this dataset contains 420198 errors. When loaded into the application developed throughout this thesis, about the same number of rows could not even be parsed, mostly due to invalid enumeration variants. This feed contains a stop for each platform. For most platforms of train stations like Kiel Central Station, the platform codes are present in the data. This can be observed in Figure 6.2a Platform codes for the platforms of the Kiel Central Station bus stops are however missing as shown in Figure 6.3a. Because of the high amount of errors in this dataset, another GTFS feed for

[1] https://gtfs.mfdz.de/

[2] https://github.com/mecatran/gtfsvtor



(a) Kiel Central Station in the DELFI GTFS feed.

(b) Kiel Central Station in the GTFS feed by Patrick Brosi.

(c) Kiel Central Station in the NAH.SH GTFS feed.

Figure 6.2. Kiel Central Station in three different GTFS feeds.

Germany is published by Patrick Brosi. This feed has a much better quality and parses without errors. The free version lacks some information such as platform codes and detailed geographic trip paths, but each platform has its own stop. This can be observed in Figure 6.2b. In some exceptions however, such as the bus stops at Kiel Central Station, the platform codes are included in the stop names as seen in Figure 6.3b. Both feeds are based upon the DELFI NeTEx dataset. There also exists the NAH.SH GTFS feed for Schleswig-Holstein. This feed also parses without errors. Different from the two GTFS feeds for the whole of Germany, this feed does not contain separate stops for each platform at stations as shown in Figure 6.2c and Figure 6.3c. Thus, information about platforms are completely missing. This information however is quite important to passengers. Also, replacement buses are labeled as railway, which might be confusing to users. While the feed is very complete within the near proximity to Bad Malente-Gremsmühlen, some train lines such as the RE8 between Lübeck Central Station and Lübeck-Travemünde are missing. The NAH.SH feed contains 4869 errors according to the MITFAHR | DE | ZENTRALE.

When it just comes to the quality and reliability of the content, the data provided by the DB Timetables API seems to be the very reliable when compared to the information shown in e.g., the DB Navigator app, but it is limited to information regarding trains and does not even include all train stations in Germany let alone Schleswig-Holstein. The DB Timetables API also provides detailed real-time information for virtually all train lines included in the API. In contrast, the real-time feeds for the GTFS feeds seem to be pretty incomplete. Most real-time deviations shown by the NAH.SH app are not included.

One solution to the different quality and reliability of the sources might be to display the sources for each trip. This way, the user can decide how credible an information is. This way, poor quality information have less impact to the overall trustworthiness of all provided information.

With regard to Bad Malente-Gremsmühlen as the main point of interest for this thesis' implementation, the displayed information as seen in Figure 6.4 are nearly complete. However, information regarding the Bürgerbus Malente are missing due to the reasons mentioned

(a) Kiel Central Station bus stops in the DELFI GTFS feed.

(b) Kiel Central Station bus stops in the GTFS feed by Patrick Brosi.

(c) Kiel Central Station bus stops in the NAH.SH GTFS feed.

Figure 6.3. Kiel Central Station bus stops in three different GTFS feeds.

in Section 3.5. Visible in the screenshot are the buses and trains arriving and departing at the train station in Bad Malente-Gremsmühlen. An overview of stops near Bad Malente-Gremsmühlen, which are included in the implementation developed throughout this thesis, can be seen in Figure 6.5.

6.4 Performance

The main API endpoint provides all known public transport and mobility options near a specified geographical location and within a specified radius and time frame. Requests to this endpoint are slow when many public transport trips are stopping within the specified search radius. For example, requesting this endpoint for a radius of 300 meters around Kiel Central Station with a time frame of 1 hour during day, a client has to wait about 2 seconds for a response. This response includes 5 stops, 70 lines and 191 trip instances with a total size of 1.86 megabytes.

All performance measurements where conducted with the backend hosted on a vServer with 6 processor cores and 16 gigabytes of memory. Total request times are measured by a client requesting the resource. The durations for fetching trips from the database and the instantiation of trips where measured directly by the backend server. The results were sampled over 100 request for each point over a total duration of 2 hours and 9 minutes. The radius was set to 300 meters. The client script used to perform the measurements is provided in Listing A.1. The measurements performed in the backend application are all done within the code excerpt in Listing A.2. In order to prevent characteristics of different information sources to influence the results, only the NAH.SH GTFS dataset was consulted during the measurements. Since the DB Timetables API only provides trip schedules for short amount of days ahead of time and the amount of actually fetched schedule information is also influenced by the rate limiting, this could e.g., influence some measurements. In contrast,

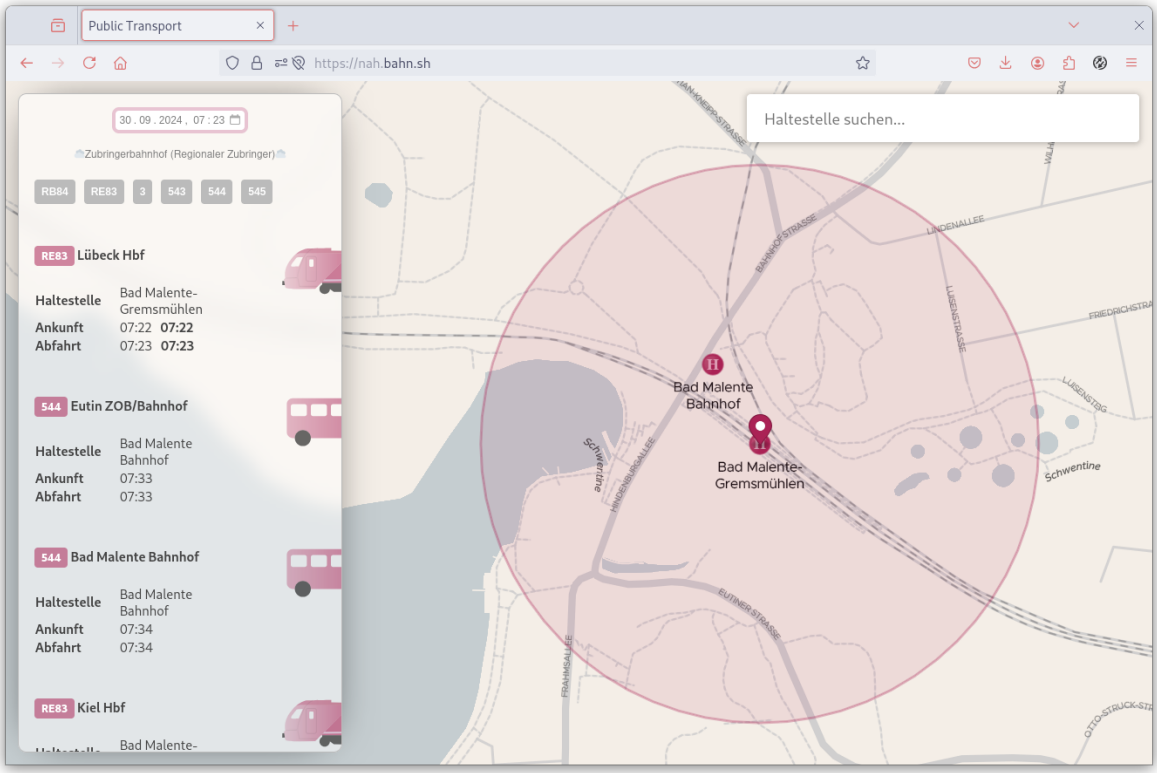


Figure 6.4. The UI of the developed tool in Bad Malente-Gremsmühlen.

with a GTFS dataset, all data is available and complete.

Response time measurements with the same parameters for all stations on the Kiel-Lübeck track and some stops close to the Malente-Lütjenburg track can be seen in Figure 6.6. These measurements are compared to the number of returned trip instances. A strong correlation between the number of trip instances returned and the total request time. Figure 6.7 also implies an equal correlation between the number of trips fetched from the database and the total request time. However, it is shown that much more trips are returned from the database than trip instances are returned for each stop.

Figure 6.8a shows the response times and returned trip instances for increasing time frames within a range expected as typical. The displayed measurements are performed for Kiel Central Station, since Figure 6.6 and Figure 6.7 show the highest response times for this stop. One can observe, that the number of returned trip instances does increase linearly, but the total response time does not change significantly with increasing time frames. So the response times do not really seem to depend on the number of returned trip instances. Further, it can be observed that the time taken to fetch the trips from the database does also stay pretty constant with increasing time frames. This is worth noting, as fetching the trips from the database seems to take most of the total response time, while the time taken to

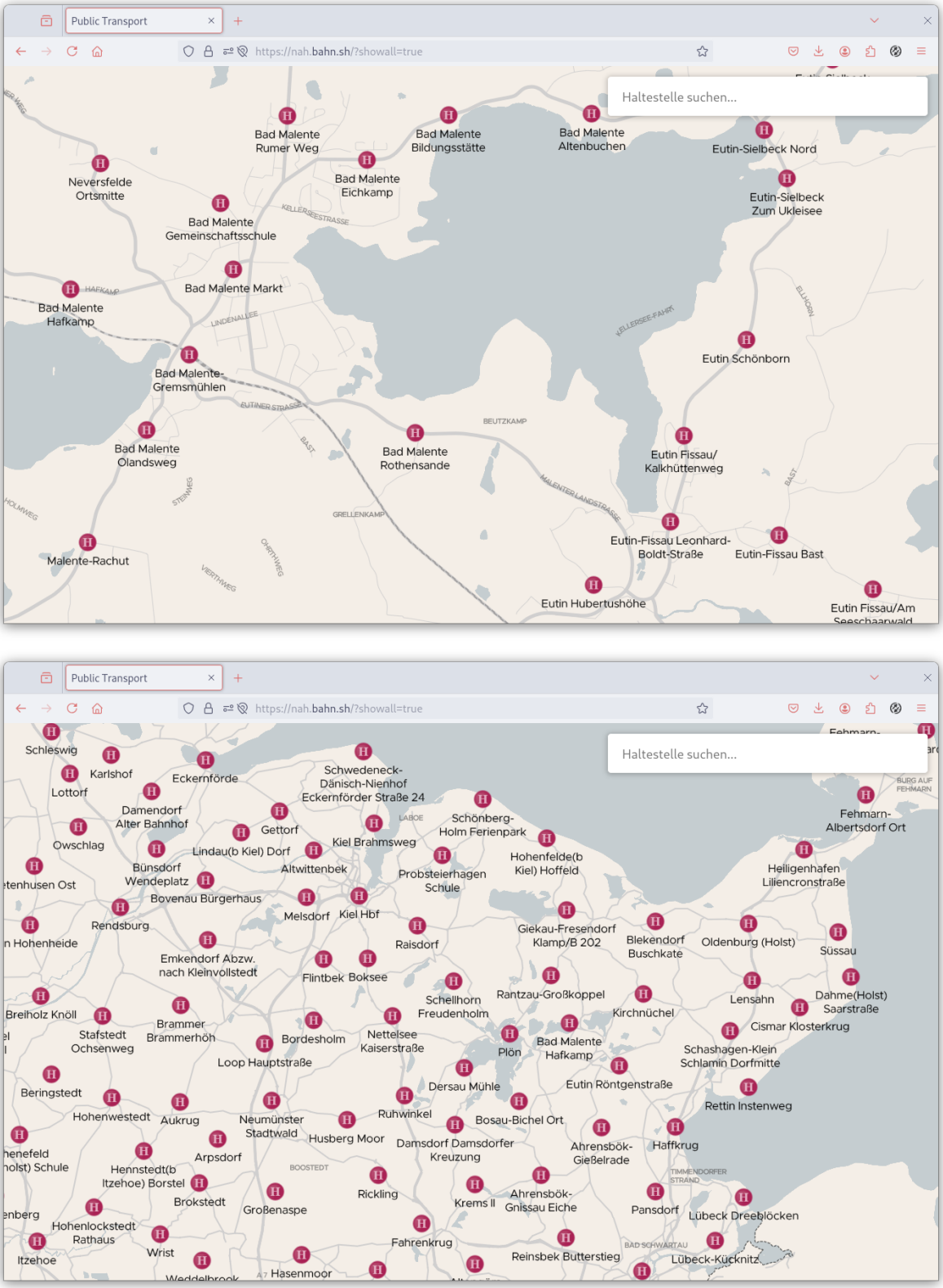


Figure 6.5. A selection of the available bus stops near Bad Malente-Gremsmühlen.

instantiate the trips is almost negligible.

The reason for this can be seen in Figure 6.8b, which shows the same information as Figure 6.8a, but for much larger time frames. Additionally, the number of trips fetched from the database is also shown. It can be observed, that for this very large span of time frames, ranging from 0 to 75 hours, the same amount of trips is queried from the database for each time frame. For time frames shorter than 24 hours, most of the trips returned from the database will be thrown away during instantiation. Only from 5 hours onwards, more trip instances are returned than trips are fetched from the database. Also, the total response time seems to indeed correlate with the number of returned trip instances, but not too much. In fact, this correlation seems especially low for typical time frames, as Figure 6.8a suggests. Since the trip instantiation time does not nearly increase equally to the overall request time and the time for fetching trips from the database does not increase at all, this correlation can probably be explained by the increasing transmission times caused by increasing file sizes.

In conclusion, the measurements imply that trips are poorly filtered by time when fetched from the database, which is indeed true. Although the trips are filtered by stops pretty accurately, time of day is not considered at all and filtering by service days only works for service exceptions and beyond the availability time span of a service, which typically is about a year. This is due to the complexity of the required logic for an adequate filtering in the database. Due to the way services are stored, it is difficult to decide whether a certain trip stops at a specific stop within a given timeframe or not in a SQL query. Therefor, too many trips are returned from the database rather than too few. It is then decided during instantiation, which of these trips are actually within the timeframe.

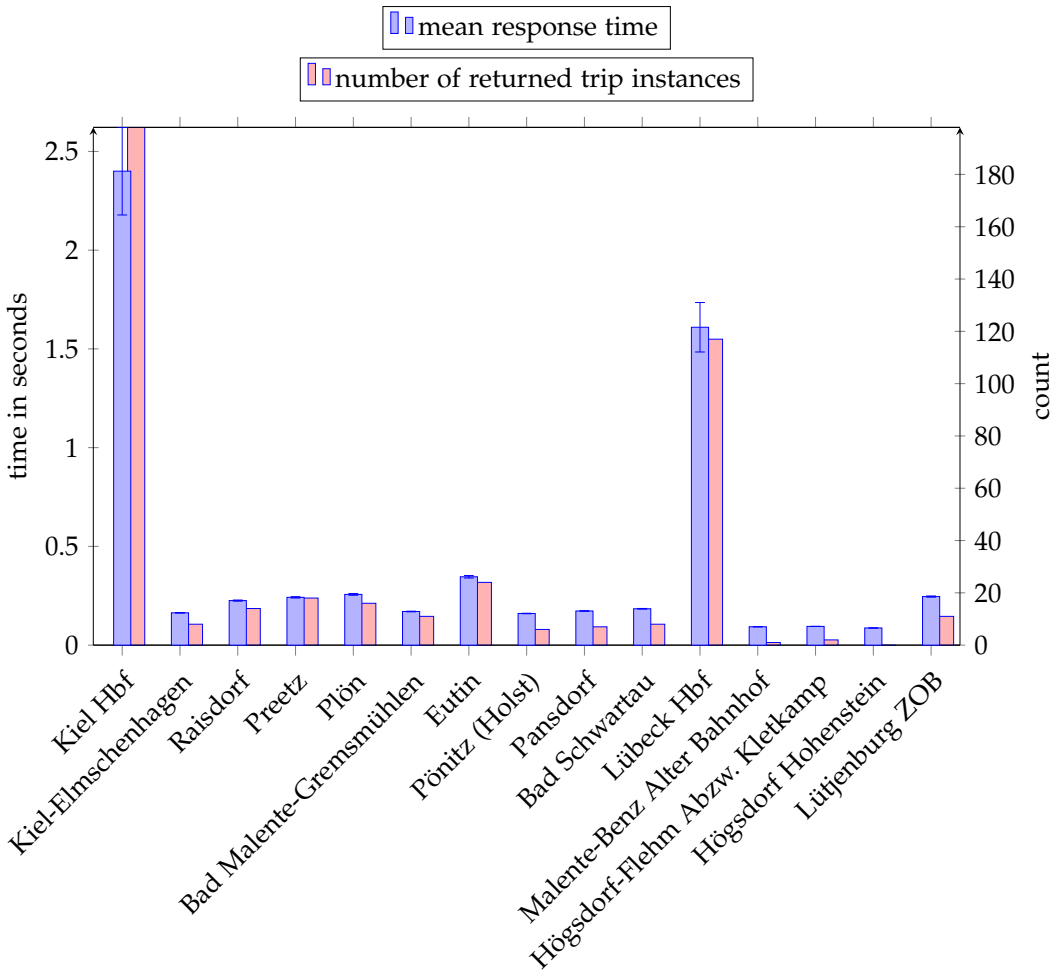


Figure 6.6. Mean response times and number of returned trips for all stations on the Kiel-Lübeck track and bus stops near the Malente-Lütjenburg track.

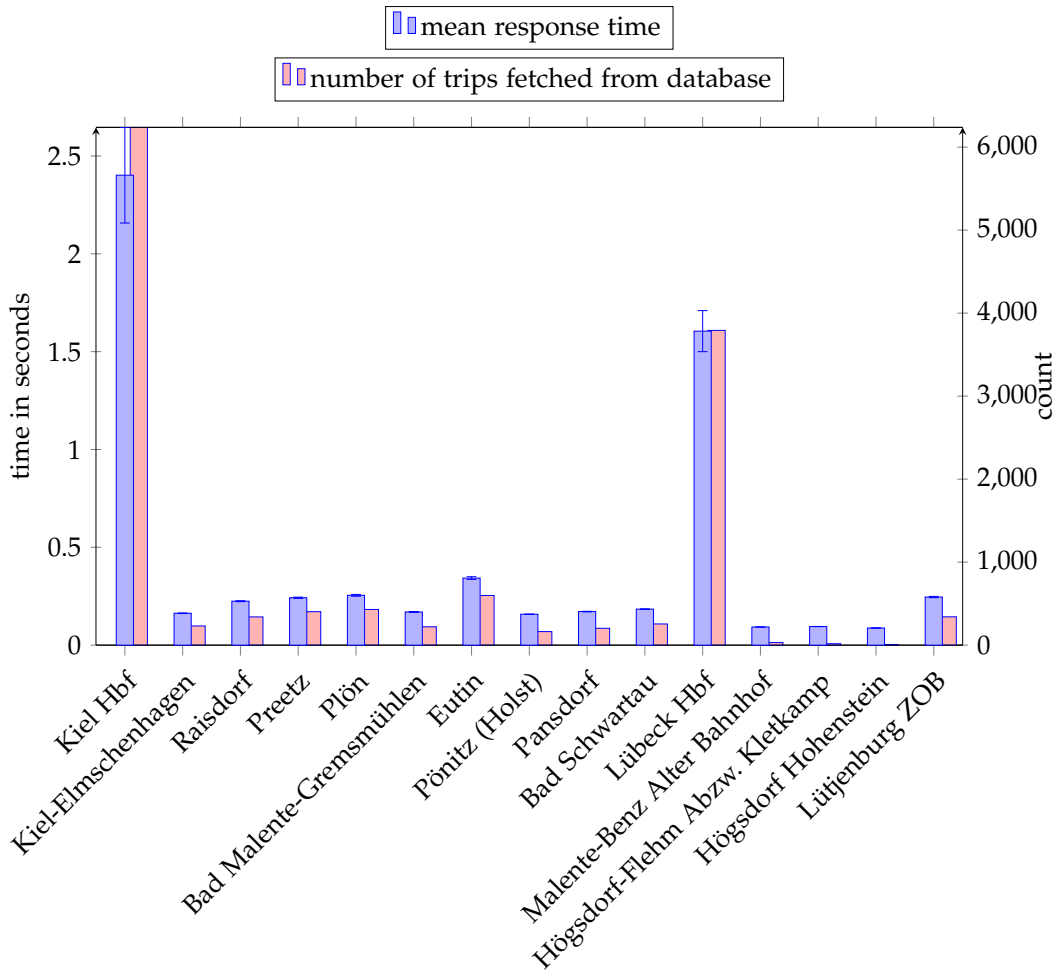
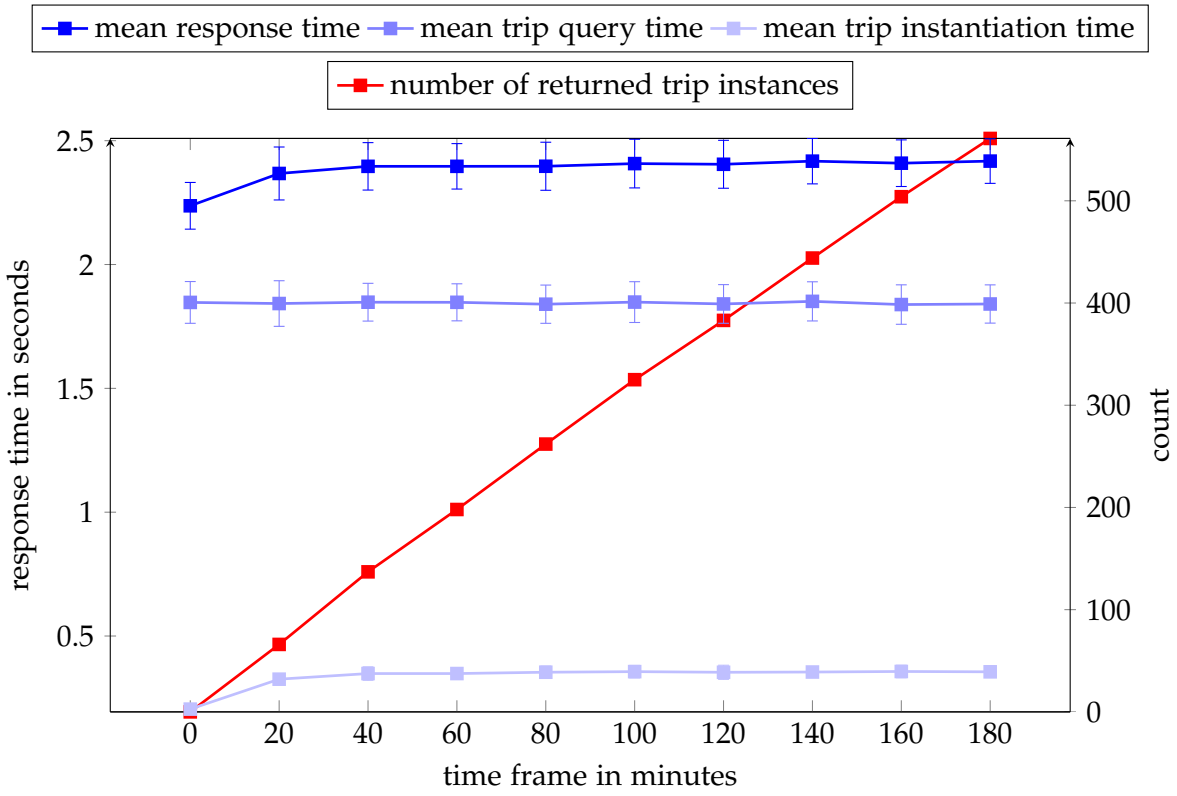
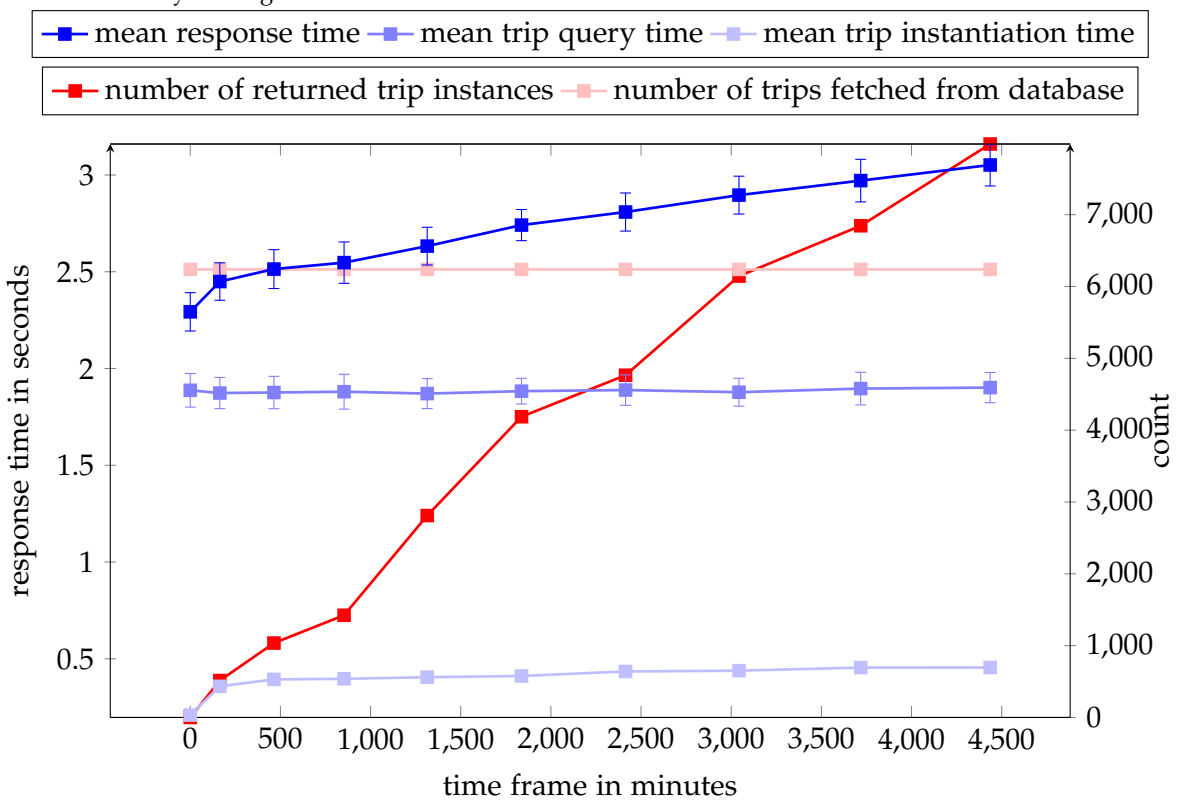


Figure 6.7. Mean response times and number of trips fetched from the database for all stations on the Kiel-Lübeck track and bus stops near the Malente-Lütjenburg track.



(a) Time frames expected to be requested in typical use-cases. Number of trips returned from database are omitted due to readability. See Figure 6.8b.



(b) Very large time frames.

Figure 6.8. Development of response times as a function of time frame size for Kiel Central Station.

Chapter 7

Conclusion

Concluding this thesis, Section 7.1 summarizes the findings and the developed solution. Section 7.3

7.1 Summary

In this thesis, existing solutions for passenger facing public transport solutions where presented. It was found, that existing solutions are either limited in scope and quality, or difficult for small projects to integrate with due to the proprietary nature. In order to develop a solution for this problem, available information sources for all different kinds of public transport and mobility with relevance to the REAKT DATA project where presented. As a possible solution, a concept was proposed and implemented throughout this thesis. Lastly, the developed solution was evaluated, revealing the weaknesses and unsolved problems. In order to provide reliable and complete public transport information, which is competitive with existing proprietary solutions such as the DB Navigator, there is a still work to be done regarding the developed concept and implementation. However, this thesis provides a starting point for implementing a more refined and feature-complete implementation. As it was part of the concept of this implementation to develop an open-source solution, the implemented software is published in a GitHub repository[1].

7.2 Encountered Problems

The high diversity of the different source data was expected to be a challenge from the beginning of this thesis. However, one particular source was especially challenging, is just the use of it in isolation was inconvenient. The DB Timetables API has some weaknesses already addressed in Section 3.3. Besides these, the available documentation is incomplete and contains incorrect information such as optional fields marked as required or wrong date formats. Further, the design of the API made it difficult to use it, as it is centered around stations and not trips. Thus, users of the API are required to build the trips from the stops fetched for each station themselves. In order to do so, one can either use the index of the stop in a trip, which is contained in the ID of the stop or use the provided path. The challenge when using the latter is that it is just a string of station names, not IDs. To make matters worse, these station names are not even consistent within the API. However, using the stop

[1] https://github.com/DerBusNachRaisdorf/OpenTransitAndMobility

indices to assemble trips from stops is also not ideal, since one can not easily know if stops are missing from the trip unless also utilizing the path string. Further, the documentation states that added stops get assigned a stop index above 100, it does however not mention how these indices are assigned. Thus, it is not possible to know the exact position of an added stop within a trip.

7.3 Future Work

As mentioned above, the approach developed throughout this thesis constitutes a basis for a combined public transport information system, but a lot of work has still to be done in order to compete with existing solutions. Thus, this section points out the unsolved problems and possible improvements.

7.3.1 Connect Other Sources

In Section 3.5, other information sources where mentioned, which are somewhat relevant to the scope of this thesis, but where not implemented. However, the e-scooter information sources are an interesting addition to the implemented shared mobility stations in the GBFS format, as e-scooters are not rented based on fixed-location stations. E-scooters can be rented anywhere someone left one and can be returned likewise. The Bürgerbus Malente is also an interesting service, as the service is quite unusual. Further, although the bus operates on fixed routes and with fixed schedules, the real-time updates only include vehicle positions and not estimated arrival and departure times. Thus, it is an interesting service for experimenting with predicting delays based upon real-time vehicle positions and schedules. Therefor, these information sources should be integrated with the developed application.

7.3.2 Distinguish Different Platforms

Especially at larger stations, platform codes are very important to users of a public transport information system. However, the current implementation shows all platforms of stations as one stop. This is mainly due to the choice of using the NAH.SH GTFS dataset, which does not include individual platforms. Replacing the dataset by e.g., the DELFI GTFS dataset is sufficient to display each platform individually and include platform codes in arrival and departures.

However, the implementation for the DB Timetables API collector must be extended to create an individual stop for each platform at each station. This is straightforward to implement, as the DB Timetables API returns all existing platform codes when station information are queried. The collector then to insert a stop with the same name as the station for each platform, where the field for the platform code is set respectively. The parent station of these stops should be set for best results, but it is not necessary. Geographical coordinates must not be set, as they are not known. The stops for the individual platforms will then successfully get matched with the GTFS stops for the corresponding platforms based upon station name,

platform code and optionally parent station. Thus, when merged on query, the geographical locations are obtained from the GTFS data.

Note, that using the free version of the GTFS dataset by Patrick Brosi will not work as intended, as the redundant information needed to match platforms is not present between this GTFS dataset and the information provided by the DB Timetables API. The free GTFS dataset by Patrick Brosi includes only the geographical locations of each platform, but not the platform codes, whereas the DB Timetables API provides only the platform codes, but not the geographical locations for each platform. The platform specific information of the GTFS dataset by Patrick Brosi and the platform specific information included in the DB Timetables API are disjunct,

7.3.3 Implement Routing

The features provided by the service are very basic at the moment. Filtering departures and arrivals by a desired destination or origin is not possible at the moment, but easy to implement. A routing feature on the other hand is much more complex to implement but crucial to the usefulness of the solution. Thus, possible solutions for routing should be researched and implemented. Beyond conventional public transport routing features as e.g., provided by HAFAS, including shared mobility like rental bikes or e-scooters might be a useful addition. When implementing such a feature, the user experience can be improved further by connecting other information services unrelated to public transport or mobility. This includes, for instance, favoring public transport options over certain shared mobility options despite worse availability or travel times over rental bikes or e-scooter based on current or predicted weather conditions.

7.3.4 Support for On-Demand Public Transport

On-demand public transport is especially useful in regions where the existing infrastructure is limited and demand for public transport is neither high enough nor stable to justify a conventional public transport line. An example for this is the service planned for the Malente-Lütjenburg track. Thus, on-demand public transport services should be supported by the implemented solution in order to meet the goal of supporting a broad variety of public transport and mobility with a focus on small and innovative mobility projects. However, an on-demand public transport service does currently not exist in the proximity to the Malente-Lütjenburg track, thus support for on-demand options where not implemented in this thesis. As the public transport domain model developed throughout this thesis is strongly based on the GTFS model, it should be possible to implement support for on-demand services without major changes to the existing solution. This is due to GTFS already supporting on-demand public transport to a certain degree.

7.3.5 Improve Implementation

Some features presented in the concept chapter currently have incomplete or poor implementations. Most notably, this applies to the duplicate identification and merging of trips. Thus, incomplete implementations, especially the merging and identification of trips, should be finalized. Further, features with poor implementations should be improved.

Besides missing or incomplete features, the implementation might also benefit from technical improvements. As mentioned in Chapter 6, the instantiation of trips at specified stations within a specified time frame is quite slow. To improve this, the SQL query fetching the trips from the database should filter the returned trips by time of day as well as service days.

Appendix A

Benchmark Script

Listing A.1. Python script used to benchmark the implemented API.

```python
import sys
import json
import requests
import numpy as np
from typing import Callable, Self
from datetime import datetime, timedelta

# helper functions for drawing tikz pictures
from eval.tikz import TikzPicture

def str_to_datetime(datetime_str: str) -> datetime:
    return datetime.strptime(datetime_str, "%Y-%m-%dT%H:%M:%S")

def datetime_to_str(dt: datetime) -> str:
    return dt.strftime("%Y-%m-%dT%H:%M:%S")

class Mean:
    def __init__(self, mean: float, err: np.floating):
        self.mean = mean
        self.err = err

def calc_mean(xs, selector: Callable | None = None) -> Mean:
    if selector is None:
        selector = lambda x: x
    mean = sum(selector(x) for x in xs) / len(xs)
    std = np.std(np.array(list(selector(x) for x in xs)))
    return Mean(mean, std)

class PointYMean:
    def __init__(self, x: float | int, y: Mean):
        self.x = x
        self.y = y

    def printed(self) -> Self:
        print(f"({self.x}, {self.y.mean}) +- (0, {self.y.err})")
        return self
```

```
37
38 def get(lat, lon, radius, start, end):
39     response = requests.get(
40         f'https://nah.bahn.sh/api/v1/nearby?latitude={lat}&longitude={lon}&radius={radius}&
               start={start}&end={end}',
41         allow_redirects=True,
42         headers={
43             "Accept": "application/json",
44             "Accept-Encoding": "gzip, deflate, br, zstd"
45         })
46
47     if response.status_code == 200:
48         return response
49     else:
50         print(f"Failed to retrieve data: {response.status_code}")
51         exit()
52
53 def increasing_timeframe(point, radius, start_time, step_minutes, num_steps, x_exp=None,
       count=10):
54     dt = str_to_datetime(start_time)
55     time_points = []
56     for i in range(num_steps):
57         minutes = i * step_minutes
58         if x_exp:
59             minutes = int(minutes**x_exp)
60         end_time = datetime_to_str(dt + timedelta(minutes=minutes))
61         time_points.append({
62             "minutes": minutes,
63             "start": start_time,
64             "end": end_time,
65             "responses": [],
66         })
67     for _ in range(count):
68         for time_point in time_points:
69             response = get(
70                 point["latitude"],
71                 point["longitude"],
72                 radius,
73                 time_point["start"],
74                 time_point["end"]
75             )
76             body = response.json()
77             time_point["responses"].append({
78                 "elapsed": response.elapsed.total_seconds(),
79                 "numTripInstances": len(body["trips"]),
80                 "benchmark": body["debugInfo"]["benchmark"]
81             })
```

```
            print(response.elapsed)
    results = []
    for time_point in time_points:
        elapsed = calc_mean(time_point["responses"], lambda x: x["elapsed"])
        fetchStops = calc_mean(time_point["responses"], lambda x: x["benchmark"]["
            fetchStopsSecs"])
        fetchLines = calc_mean(time_point["responses"], lambda x: x["benchmark"]["
            fetchLinesSecs"])
        fetchTrips = calc_mean(time_point["responses"], lambda x: x["benchmark"]["
            fetchTripsSecs"])
        instantiateTripsSecs = calc_mean(time_point["responses"], lambda x: x["benchmark"]["
            instantiateTripsSecs"])
        numberOfTripInstances = calc_mean(time_point["responses"], lambda x: x["
            numTripInstances"])
        numberOfTripsFetched = calc_mean(time_point["responses"], lambda x: x["benchmark"]["
            numTripsFetched"])
        results.append({
            "minutes": time_point["minutes"],
            "elapsed": {
                "totalRequest": elapsed,
                "fetchStops": fetchStops,
                "fetchLines": fetchLines,
                "fetchTrips": fetchTrips,
                "instantiateTrips": instantiateTripsSecs
            },
            "numberOfTripInstances": numberOfTripInstances,
            "numberOfTripsFetched": numberOfTripsFetched
        })
    return { "name": point["name"], "points": results, "count": count }

def plot_increasing_timeframe(result, picture: TikzPicture, include_database_trips=True):
    width = "14cm"
    height = "9cm"

    # count axis
    count_axis = picture.axis()
    count_axis.width(width)
    count_axis.height(height)
    count_axis.y_label("count")
    count_axis.y_axis_right()
    count_axis.x_tick(None)
    count_axis.legend_style(0.5, 1.05)

    # number of trips fetched plot
    trip_insances_plot = count_axis.plot("number␣of␣returned␣trip␣instances")
    trip_insances_plot.color("red")
    trip_insances_plot.mark("red")
```

```
123     trip_insances_plot.line_width("1pt")
124     trip_insances_plot.points_y_mean([
125         PointYMean(point["minutes"], point["numberOfTripInstances"])
126         for point in result["points"]
127     ])
128     trip_insances_plot.error()
129 
130     # number of returned trip instances plot
131     if include_database_trips:
132         trips_plot = count_axis.plot("number of trips fetched from database")
133         trips_plot.color("red!25")
134         trips_plot.mark("red!25")
135         trips_plot.line_width("1pt")
136         trips_plot.points_y_mean([
137             PointYMean(point["minutes"], point["numberOfTripsFetched"])
138             for point in result["points"]
139         ])
140         trips_plot.error()
141 
142     # time axis
143     time_axis = picture.axis()
144     time_axis.width(width)
145     time_axis.height(height)
146     time_axis.x_label("time frame in minutes")
147     time_axis.y_label("response time in seconds")
148     time_axis.y_axis_left()
149     time_axis.legend_style(0.5, 1.15)
150     time_axis.scaled_ticks(False)
151 
152     # total request time plot
153     req_time_plot = time_axis.plot("mean response time")
154     req_time_plot.mark("blue")
155     req_time_plot.line_width("1pt")
156     req_time_plot.points_y_mean([
157         PointYMean(point["minutes"], point["elapsed"]["totalRequest"]).printed()
158         for point in result["points"]
159     ])
160     req_time_plot.error()
161 
162     # trip query time plot
163     req_time_plot = time_axis.plot("mean trip query time")
164     req_time_plot.color("blue!50")
165     req_time_plot.mark("blue!50")
166     req_time_plot.line_width("1pt")
167     req_time_plot.points_y_mean([
168         PointYMean(point["minutes"], point["elapsed"]["fetchTrips"]).printed()
169         for point in result["points"]
```

```
    ])
    req_time_plot.error()

    # trip instantiation time plot
    req_time_plot = time_axis.plot("mean trip instantiation time")
    req_time_plot.color("blue!25")
    req_time_plot.mark("blue!25")
    req_time_plot.line_width("1pt")
    req_time_plot.points_y_mean([
        PointYMean(point["minutes"], point["elapsed"]["instantiateTrips"]).printed()
        for point in result["points"]
    ])
    req_time_plot.error()

def all_stops(points, radius, start_time, end_time, count):
    points = json.loads(json.dumps(points))
    for _ in range(count):
        for point in points:
            response = get(point["latitude"], point["longitude"], radius, start_time,
                end_time)
            elapsed = response.elapsed.total_seconds()
            data = response.json()
            print(elapsed)
            point["results"].append({
                "elapsed": elapsed,
                "numTripInstances": len(data["trips"]),
                "numLines": len(data["lines"]),
                "numStops": len(data["stops"]),
                "numSharedMobilityStations": len(data["sharedMobilityStations"]),
                "benchmark": data["debugInfo"]["benchmark"]
            })
    # evaluate
    results = []
    for point in points:
        elapsed = calc_mean(point["results"], lambda x: x["elapsed"])
        fetchStops = calc_mean(point["results"], lambda x: x["benchmark"]["fetchStopsSecs"])
        fetchLines = calc_mean(point["results"], lambda x: x["benchmark"]["fetchLinesSecs"])
        fetchTrips = calc_mean(point["results"], lambda x: x["benchmark"]["fetchTripsSecs"])
        instantiateTripsSecs = calc_mean(point["results"], lambda x: x["benchmark"]["
            instantiateTripsSecs"])
        numberOfTripInstances = calc_mean(point["results"], lambda x: x["numTripInstances"])
        numberOfTripsFetched = calc_mean(point["results"], lambda x: x["benchmark"]["
            numTripsFetched"])
        results.append({
            "stop": point["name"],
            "elapsed": {
                "totalRequest": elapsed,
```

```
                "fetchStops": fetchStops,
                "fetchLines": fetchLines,
                "fetchTrips": fetchTrips,
                "instantiateTripsSecs": instantiateTripsSecs
            },
            "numberOfTripInstances": numberOfTripInstances,
            "numberOfTripsFetched": numberOfTripsFetched
        })
    return { "points": results, "count": count }

def plot_all_stops(result, picture: TikzPicture, trip_instances=True):
    width = "14cm"
    height = "9cm"

    # count axis
    count_axis = picture.axis()
    count_axis.width(width)
    count_axis.height(height)
    count_axis.y_bar(width="7pt")
    count_axis.bar_shift("0.2cm")
    count_axis.y_axis_right()
    count_axis.y_label("count")
    count_axis.y_min(0)
    #count_axis.y_max(200)
    count_axis.x_min(0.5)
    count_axis.x_max(len(result["points"]) + 0.5)
    count_axis.x_tick(None)
    count_axis.legend_style(0.5, 1.05)

    if trip_instances:
        # number of trip instances plot
        trip_instances_plot = count_axis.plot("number␣of␣returned␣trip␣instances")
        trip_instances_plot.fill("red!30")
        trip_instances_plot.points_y_mean([
            PointYMean(idx+1, point["numberOfTripInstances"])
            for idx, point in enumerate(result["points"])
        ])
    else:
        # number of trips fetched plot
        trips_plot = count_axis.plot("number␣of␣trips␣fetched␣from␣database")
        trips_plot.fill("red!30")
        trips_plot.points_y_mean([
            PointYMean(idx+1, point["numberOfTripsFetched"])
            for idx, point in enumerate(result["points"])
        ])


```

```
    # response time axis
    time_axis = picture.axis()
    time_axis.width(width)
    time_axis.height(height)
    time_axis.y_bar(width="7pt")
    time_axis.y_axis_left()
    time_axis.y_label("time in seconds")
    time_axis.y_min(0)
    time_axis.scaled_ticks(False)
    time_axis.x_tick_labels([point["stop"] for point in result["points"]])
    time_axis.option("enlarge x limits={abs=0.5}")
    time_axis.legend_style(0.5, 1.15)

    # total response time plot
    req_time_plot = time_axis.plot("mean response time")
    req_time_plot.points_y_mean([
        PointYMean(idx+1, point["elapsed"]["totalRequest"])
        for idx, point in enumerate(result["points"])
    ])
    req_time_plot.error(relative=True)
    req_time_plot.bar_shift("-0.2cm")

RADIUS = 0.3
START_TIME = "2024-09-26T16:27:52"
END_TIME = "2024-09-26T17:27:52"
COUNT = 100

INCREASING_TIMEFRAME_STEP_MINUTES = 30
INCREASING_TIMEFRAME_STEPS = 10

def point(name, lat, lon):
    return {
        "name": name,
        "latitude": lat,
        "longitude": lon,
        "results": []
    }

POINTS = [
    point("Kiel Hbf", 54.314985, 10.131976),
    point("Kiel-Elmschenhagen", 54.287142, 10.180414),
    point("Raisdorf", 54.280937, 10.243694),
    point("Preetz", 54.233941, 10.275752),
    point("Plön", 54.159479, 10.422562),
    point("Bad Malente-Gremsmühlen", 54.167014, 10.55151),
    point("Eutin", 54.135343, 10.610123),
    point("Pönitz (Holst)", 54.045654,  10.671157),
```

```
    point("Pansdorf", 53.981034, 10.703056),
    point("Bad Schwartau", 53.916276, 10.702722),
    point("Lübeck Hbf", 53.867547, 10.669821),
    point("Malente-Benz Alter Bahnhof", 54.221874, 10.613542),
    point("Högsdorf-Flehm Abzw. Kletkamp", 54.240823, 10.63057),
    point("Högsdorf Hohenstein", 54.256625, 10.623575),
    point("Lütjenburg ZOB", 54.292405, 10.593936),
]

if __name__ == '__main__':
    args = sys.argv
    all = 'all' in args
    if all or 'all-stops-trip-instances' in args:
        results = all_stops(POINTS, RADIUS, START_TIME, END_TIME, COUNT)
        picture = TikzPicture()
        latex = plot_all_stops(results, picture, trip_instances=True)
        picture.render_to_file("thesis/graphics/6-performance-trip-instances.tex")
    if all or 'all-stops-trips-fetched' in args:
        results = all_stops(POINTS, RADIUS, START_TIME, END_TIME, COUNT)
        picture = TikzPicture()
        latex = plot_all_stops(results, picture, trip_instances=False)
        picture.render_to_file("thesis/graphics/6-performance-trips-fetched.tex")
    if all or 'increasing-timeframe-small' in args:
        results = increasing_timeframe(
            POINTS[0],
            RADIUS,
            START_TIME,
            20,
            10,
            count=COUNT
        )
        picture = TikzPicture()
        latex = plot_increasing_timeframe(results, picture, include_database_trips=False)
        picture.render_to_file("thesis/graphics/6-increasing-timeframe-small.tex")
    if all or 'increasing-timeframe-big' in args:
        results = increasing_timeframe(
            POINTS[0],
            RADIUS,
            START_TIME,
            INCREASING_TIMEFRAME_STEP_MINUTES,
            INCREASING_TIMEFRAME_STEPS,
            x_exp=1.5,
            count=COUNT
        )
        picture = TikzPicture()
        latex = plot_increasing_timeframe(results, picture)
        picture.render_to_file("thesis/graphics/6-increasing-timeframe-big.tex")
```

Listing A.2. Excerpt from the implementation of the API

```rust
#[derive(Serialize)]
#[serde(rename_all = "camelCase")]
struct NearbyBenchmark {
    fetch_shared_mobility_stations_secs: f64,
    fetch_stops_secs: f64,
    fetch_lines_secs: f64,
    fetch_trips_secs: f64,
    instantiate_trips_secs: f64,
    num_trips_fetched: usize,
}

async fn nearby(
    OriginalUri(original_uri): OriginalUri,
    State(WebState { transit_client, .. }): State<WebState>,
    Query(params): Query<TripsNearbyQuery>,
    Extension(base_url): Extension<Arc<BaseUrl>>,
) -> HateoasResult<NearbyDto> {
    let origins = transit_client.get_origin_ids().await?;
    let radius = params.radius.unwrap_or(0.05);
    let start = params.start.unwrap_or(Local::now());
    let end = params.end.unwrap_or(start + Duration::hours(1));

    // get shared mobility stations
    let now = Instant::now();
    let shared_mobility_stations = transit_client
        .find_nearby_shared_mobility_stations(
            params.latitude,
            params.longitude,
            radius,
            &origins,
        )
        .await
        .map_err(|why| {
            RouteErrorResponse::from(why)
                .with_method(&Method::GET)
                .with_message("Could not query nearby shared mobility stations.")
                .with_uri(original_uri.path())
        })?;
    let fetch_shared_mobility_elapsed = now.elapsed();

    // get stops
    let now = Instant::now();
    let stops = transit_client
        .find_nearby(params.latitude, params.longitude, radius, &origins)
        .await
        .map_err(|why| {
```

```
47              RouteErrorResponse::from(why)
48                  .with_method(&Method::GET)
49                  .with_message("Could not query nearby stops.")
50                  .with_uri(original_uri.path())
51          })?;
52      let fetch_stops_elapsed = now.elapsed();
53
54      // get lines and trips
55      let now = Instant::now();
56      let mut lines = vec![];
57      for stop in stops.iter() {
58          // get lines
59          lines.extend(
60              transit_client
61                  .get_lines_at_stop(&stop.content.id, &origins)
62                  .await
63                  .map_err(|why| {
64                      RouteErrorResponse::from(why)
65                          .with_method(&Method::GET)
66                          .with_message("Could not query lines at nearby stops.")
67                          .with_uri(original_uri.path())
68                  })?,
69          );
70      }
71      let fetch_lines_elapsed = now.elapsed();
72
73      // stop ids
74      let stop_ids = stops
75          .iter()
76          .map(|stop| &stop.content.id)
77          .collect::<Vec<_>>();
78
79      // get raw trips
80      // TODO: what to do with duplicate trips?
81      let now = Instant::now();
82      let trips = transit_client
83          .get_all_trips_via_stops(&stop_ids, start, end, &origins)
84          .await
85          .map_err(|why| {
86              RouteErrorResponse::from(why)
87                  .with_method(&Method::GET)
88                  .with_message("Could not query trips at nearby stops.")
89                  .with_uri(original_uri.path())
90          })?;
91      let fetch_trips_elapsed = now.elapsed();
92      let num_database_trips = trips.len();
93
```

```
    // instanciate trips
    let now = Instant::now();
    let mut instanciated_trips = transit_client
        .instanciate_trips_include(
            trips,
            DateTimeRange::new(start, end),
            Some(&stop_ids),
            true,
            true,
            true,
            &origins,
        )
        .await
        .map_err(|why| {
            RouteErrorResponse::from(why)
                .with_method(&Method::GET)
                .with_message("Could not instanciate trips at nearby stops.")
                .with_uri(original_uri.path())
        })?;
    let instantiate_trips_elapsed = now.elapsed();

    // sort trips
    TripInstance::sort(&mut instanciated_trips);

    // unique lines
    lines = lines
        .into_iter()
        .unique_by(|line| line.id.clone())
        .collect();

    let benchmark = NearbyBenchmark {
        fetch_shared_mobility_stations_secs: fetch_shared_mobility_elapsed
            .as_secs_f64(),
        fetch_stops_secs: fetch_stops_elapsed.as_secs_f64(),
        fetch_lines_secs: fetch_lines_elapsed.as_secs_f64(),
        fetch_trips_secs: fetch_trips_elapsed.as_secs_f64(),
        instantiate_trips_secs: instantiate_trips_elapsed.as_secs_f64(),
        num_trips_fetched: num_database_trips,
    };

    let nearby = NearbyDto {
        radius,
        latitude: params.latitude,
        longitude: params.longitude,
        start,
        end,
        stops: stops
```

```
            .into_iter()
            .map(|stop| stop_with_distance_hateoas(stop, base_url.clone()))
            .collect(),
        lines: lines
            .into_iter()
            .map(|line| line_hateoas(line, base_url.clone()))
            .collect(),
        trips: instanciated_trips
            .into_iter()
            .map(|trip| {
                trip_hateoas(
                    TripInstanceDto {
                        info: trip.info,
                        stops: trip
                            .stops
                            .into_iter()
                            .map(|stop_time| {
                                stop_time_hateoas(stop_time, base_url.clone())
                            })
                            .collect::<Vec<_>>(),
                        stop_of_interest: trip.stop_of_interest,
                        line: trip
                            .line
                            .map(|line| line_hateoas(line, base_url.clone())),
                        agency: trip
                            .agency
                            .map(|agency| agency_hateoas(agency, base_url.clone())),
                    },
                    base_url.clone(),
                )
            })
            .collect::<Vec<_>>(),
        shared_mobility_stations: shared_mobility_stations
            .into_iter()
            .map(|x| x.content.content)
            .collect(),
    };

    Ok(nearby_hateoas(nearby, base_url, Some(benchmark)).json())
}
```

Bibliography

[BBS14] Hannah Bast, Patrick Brosi, and Sabine Storandt. "Real-time movement visualization of public transit data". In: *Proceedings of the 22nd ACM SIGSPATIAL International Conference on Advances in Geographic Information Systems*. SIGSPATIAL '14. Dallas, Texas: Association for Computing Machinery, 2014, pp. 331–340. ISBN: 9781450331319. DOI: 10.1145/2666310.2666404. URL: https://doi.org/10.1145/2666310.2666404.

[CVM17] Pieter Colpaert, Ruben Verborgh, and Erik Mannens. "Public transit route planning through lightweight linked data interfaces". In: *Web Engineering*. Ed. by Jordi Cabot, Roberto De Virgilio, and Riccardo Torlone. Cham: Springer International Publishing, 2017, pp. 403–411. ISBN: 978-3-319-60131-1.

[MS13] Bhargab Maitra and Shubhajit Sadhukhan. "Urban public transportation system in the context of climate change mitigation: emerging issues and research needs in india". In: *Mitigating Climate Change: The Emerging Face of Modern Cities*. Ed. by Anshuman Khare and Terry Beckman. Berlin, Heidelberg: Springer Berlin Heidelberg, 2013, pp. 75–91. ISBN: 978-3-642-37030-4. DOI: 10.1007/978-3-642-37030-4_5. URL: https://doi.org/10.1007/978-3-642-37030-4_5.

[RVD+20] Julián Andrés Rojas, Dylan Van Assche, Harm Delva, Pieter Colpaert, and Ruben Verborgh. "Efficient live public transport data sharing for route planning on the web". In: *Web Engineering*. Ed. by Maria Bielikova, Tommi Mikkonen, and Cesare Pautasso. Cham: Springer International Publishing, 2020, pp. 321–336. ISBN: 978-3-030-50578-3.

[VCC+19] Belén Vela, José María Cavero, Paloma Cáceres, and Carlos E. Cuesta. "A semi-automatic data–scraping method for the public transport domain". In: *IEEE Access* 7 (2019), pp. 105627–105637. DOI: 10.1109/ACCESS.2019.2932197.

[WWS16] Sebastian Wandelt, Zezhou Wang, and Xiaoqian Sun. "Worldwide railway skeleton network: extraction methodology and preliminary analysis". In: *IEEE Transactions on Intelligent Transportation Systems* 18.8 (2016), pp. 2206–2216.