Hochschule
Kaiserslautern
University of
Applied Sciences

Implementierung des A*-Algorithmus unter Verwendung von OpenStreetMap

Bachelorthesis von

Alexander Wiltz

an der Fakultät für Informatik und Mikrosystemtechnik
Zweibrücken

Gutachter:	Prof. Dr. Jörg Hettel
Zweitgutachter:	Dipl.-Inf. Christian Endler

20.08.2024 - 23.12.2024

Ehrenwörtliche Erklärung

Hiermit erkläre ich, Alexander Wiltz, geboren am 06.03.1987 in Saarlouis, ehrenwörtlich, dass ich meine Bachelorthesis mit dem Titel: „Implementierung des A*-Algorithmus unter Verwendung von OpenStreetMap“ selbstständig und ohne fremde Hilfe angefertigt und keine anderen als in der Abhandlung angegebenen Hilfen benutzt habe. Die Übernahme wörtlicher Zitate aus der Literatur sowie die Verwendung der Gedanken anderer Autoren an den entsprechenden Stellen habe ich innerhalb der Arbeit gekennzeichnet. Ich bin mir bewusst, dass eine falsche Erklärung rechtliche Folgen haben kann.

Überherrn, 7. Dezember 2024

. . . . . .

(Alexander Wiltz)

Zusammenfassung

Diese Arbeit setzt sich mit der Implementierung des A*-Algorithmus unter Verwendung des Kartenmaterials von OpenStreetMap auseinander.

Im Mittelpunkt der Arbeit steht eine JAVA-Integration des A-Stern-Algorithmus als Web-Service. Dieser Web-Service ist an eine Datenbank angeschlossen, die die Geodaten eines ausgewählten Kartenbereichs hält.

Ein einfach gehaltenes User-Interface in Form eines Web-Frontends bietet die Möglichkeit eine Startadresse und eine Zieladresse einzugeben, die an den Web-Service per RESTful-API übergeben wird und im Falle eines möglichen Weges, eine optimale Route zurückliefert, die schlussendlich angezeigt wird.

Die Datenbasis bilden exportierte Geodaten aus OpenStreetMap. Design- und Architekturentscheidungen werden beleuchtet und begründet. Weiterhin sind Probleme und Herausforderungen die eine Rolle für die Integration der Daten und des Codes spielen gesondert erklärt.

Abstract

This paper deals with the implementation of the A*-algorithm using map material from OpenStreetMap.

The focus is on a JAVA integration of the A-star-algorithm as a web service. This web service is connected to a database that holds the geodata of a selected map area.

A simple user interface integrated as a web front end offers the possibility to enter a start address and a destination address, which is transferred to the web service via RESTful-API. In the case of a possible route the interface of the web-service returns an optimal route that is finally displayed.

Exported geodata from OpenStreetMap form the database. Design and architecture decisions are analysed and justified. Furthermore, problems and challenges that are important in the integration of data and code are explained separately.

Inhaltsverzeichnis

1 Einleitung

In der digitalisierten Welt spielt die Routenplanung eine wichtige Rolle. Sei es für die Navigation im täglichen Straßenverkehr, in Logistiknetzwerken oder bei der Planung von Reisen. Kürzeste-Wege-Algorithmen sind dabei ein wichtiges Werkzeug, um optimale Routen zu berechnen. Ein Beispiel hierfür ist der A*-Algorithmus, der mittels Heuristik besonders schnell zur besten Lösung gelangt. Dieser Algorithmus kombiniert die Dijkstra-Suche mit einer Schätzung der verbleibenden Distanz und findet dadurch eine kürzeste Route.

Eine besonders spannende Herausforderung bei der Implementierung solcher Algorithmen stellt die Nutzung von Geodaten dar. OpenStreetMap, ein Projekt zur Erfassung von Geodaten, bietet eine umfangreiche und stetig wachsende Datenbank, die Straßen, Wege und andere infrastrukturelle Informationen weltweit abbildet. Diese offenen Daten ermöglichen es Entwicklern, reale Karteninformationen in Softwareanwendungen zu integrieren um Routenplanungen zu realisieren.

Die Implementierung des A*-Algorithmus unter Verwendung von OpenStreetMap, bietet viele interessante Anwendungsmöglichkeiten. Sie ermöglicht einerseits die Berechnung der optimalen Route zwischen zwei Punkten auf realen Karten, andererseits eröffnet sie auch die Möglichkeit für weiterführende Optimierungen, wie die Berücksichtigung von Verkehrsinformationen, Sperrungen oder besonderen Präferenzen bei der Streckenwahl.

1.1 Ziel der Arbeit

Ziel ist es, mittels Implementierung eines A*-Algorithmus in JAVA als Web-Service, einem passenden Datenmodell und einer entsprechenden Architektur mit dem freien Kartenmaterial von OpenStreetMap, kürzeste Wege innerhalb eines festgelegten Areals berechnen zu lassen.

Den Einstieg bildet ein User-Interface als Webfrontend, dass mittels RestAPI an ein Backend angeschlossen ist, mit dem die kürzesten Wege zwischen zwei übergebenen Adressen bestimmt werden. Das Ergebnis wird dann an das Frontend zurückgeliefert und entsprechend angezeigt.

Die Herausforderung dabei ist, dass die Datenbasis Geokoordinaten sind und die Applikation unabhängig und mit generellen Daten aus OpenStreetMap lokal arbeiten soll. Die Daten aus OpenStreetMap werden demnach exportiert und im Rahmen der Applikation weiterverarbeitet, aber nicht verändert.

1.2 Motivation

Die Motivation sich mit dieser Thematik zu beschäftigen ist einerseits das Interesse an den mathematischen Hintergründen, als auch die Möglichkeit derartige Anwendungen als Software zu implementieren. Ein großes Interesse an Web-Services und Schnittstellenprogrammierung mit möglichst generellen Modellen durch aktuelle Frameworks, wie Spring Boot, kombinieren genau die Themen und Werkzeuge die zur Umsetzung dieser Arbeit notwendig sind.

Nicht ganz unerheblich bei der Themenfindung ist die Faszination für Navigation, Navigationssysteme und Routenberechnungen. Die Bestimmung von möglichen Routen oder Alternativen, um möglichst optimal zu einem Ziel zu gelangen, ist reizvoll und begeisternd.

1.3 Aufbau der Arbeit

Im nächsten Kapitel werden die Grundlagen zur Problemstellung aufgegriffen und zum Verständnis der nachfolgenden Themen beschrieben.
In dem danach folgenden Kapitel wird der verwendete Algorithmus vorgestellt, ebenso die Modellierung und Implementierung in JAVA beschrieben und in einem weiteren Kapitel die gewählte Architektur behandelt. Dabei werden für alle Teile der Architektur theoretische Grundlagen beschrieben, sowie die Entscheidung zur Verwendung begründet.

In einem gesonderten Teil werden alle Erkenntnisse die sich während der Problembehandlung ergeben haben nochmals aufgegriffen und die gefundene Lösung erörtert.

2 Grundlagen

In folgendem Kapitel werden die notwendigen Grundlagen zum Verständnis der weiteren Themen umrissen. Auf weitere Informationen zu den Begriffen wird in den jeweiligen Abschnitten verwiesen oder durch Angabe entsprechender Informationsquellen im anhängenden Quellenverzeichnis.

2.1 Openstreetmap

OpenStreetMap (kurz: OSM) ist eine freie, offene und weltweite Karte, die von einer Gemeinschaft von Freiwilligen erstellt wird. Die Idee ist, dass jeder helfen kann, eine Karte zu verbessern, indem Informationen über Straßen, Gebäude, Flüsse und viele andere Dinge hinzugefügt werden. So entsteht eine ständig wachsende und detaillierte Karte, die jeder kostenlos nutzen und bearbeiten kann.

OpenStreetMap ist Kartenmaterial, dass für viele verschiedene Zwecke verwendet werden kann. Im Gegensatz zu kommerziellen Anbietern wie Google Maps ist OpenStreetMap kostenlos und die Daten stehen jedem zur Verfügung. Das macht OpenStreetMap vor allem nützlich für Projekte, die keine teuren Lizenzgebühren zahlen wollen oder für Regionen, die in anderen Karten schlecht abgedeckt sind.

Mit OpenStreetMap können zum Beispiel Routen geplant und eigene Karten erstellt werden, geografische Daten können analysiert werden und Programme entwickelt, die auf Karten und Ortsdaten basieren.

Da die Daten öffentlich sind, können Entwickler OpenStreetMap nutzen, um ihre eigenen Anwendungen zu erstellen.

2.2 Karte

Das benutzte Kartenmaterial ist ein Auszug eines Bereichs, das Daten aus einem Wohngebiet enthält. Bei dem Wohngebiet handelt es sich um den Überherrner Ortsteil ”Wohnstadt”, das im Saarland in Deutschland liegt. Die Koordinaten für den Ausschnitt spannen sich über folgenden Bereich auf:
Süden: ”49.2304”, Westen:”6.6994”, Norden: ”49.2408” und Osten: ”6.7125”.
Besonderheiten für die Bestimmung der Wege sind einerseits die Vielschichtigkeit an Wegarten: Landstraßen, Ortsdurchfahrten, Fuß- und Radwege. Teile der Häuser stehen in Wegen, die ausschließlich zu Fuß zu erreichen sind. Teilweise sind die Wege mit den Häusern nur auf einer Seite gebaut, teilweise beidseitig und zum Teil unregelmäßig angeordnet.

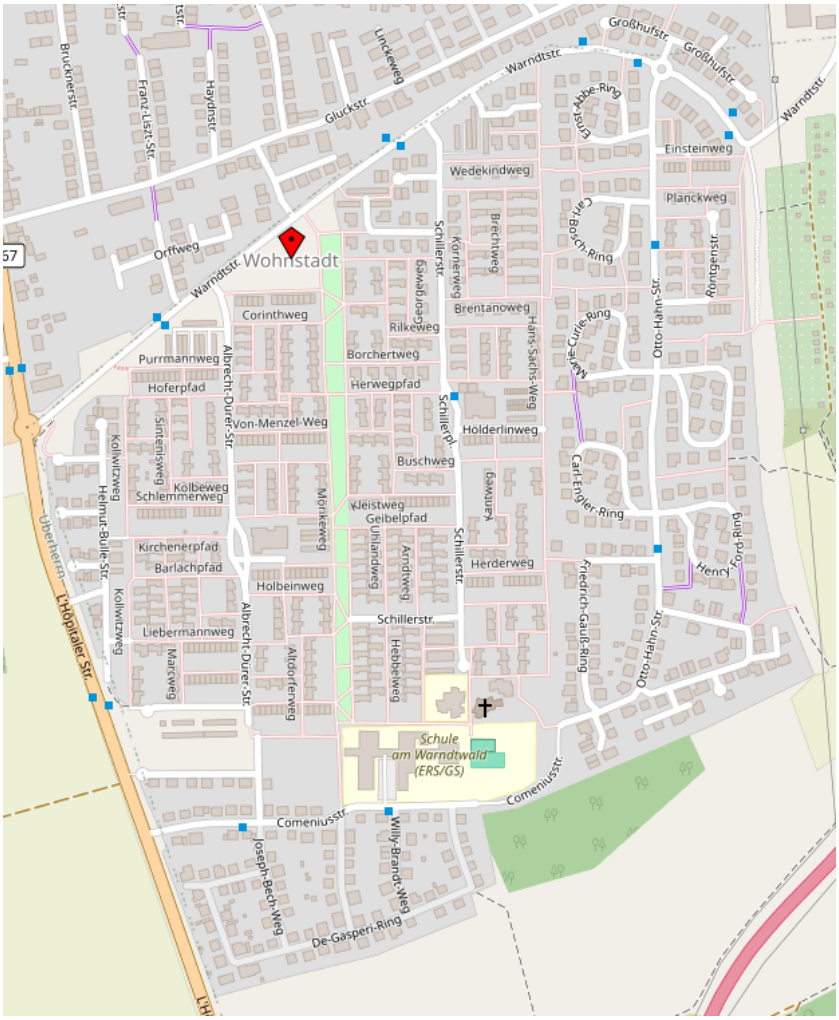

Abbildung 2.1: Kartenausschnitt Überherrn-Wohnstadt

Permanentlink zur Karte: `https://www.openstreetmap.org/query?map=20/49.23853/6.70488`

2.3 Formate und Standards

Die Formate, die im Rahmen der Arbeit mit OpenStreetMap Verwendung finden, sind ein XML[1]-Format mit einem speziellen OSM-Schema und GeoJSON. GeoJSON ist ein offenes Format, um geografische Daten nach der Simple-Feature-Access-Spezifikation zu repräsentieren. Dafür wird die JavaScript Object Notation (kurz: JSON) verwendet.

[1]Extensible Markup Language

2.3.1 OSM-Aufbau

Das in dieser Arbeit verwendete Schema ist unter `https://wiki.openstreetmap.org/wiki/API_v0.6/XSD` definiert und wird im Folgenden erläutert.

Mit dem Tag `<osm>` werden die Daten eingeleitet und gezeigt, dass es sich um Daten im OpenStreetMap-Format handelt.

Der Tag `<bounds>` (vgl. Listing 2.1) informiert mit seinen Attributen über den Umfang der enthaltenen Daten. Beim Exportieren der Daten wird demnach ein Bereich exportiert, der die Koordinaten 49.2304, 6.994 und 49.2408, 6.7125 umfasst.

```
<?xml version="1.0" encoding="UTF-8"?>
<osm version="0.6"
    generator="CGImap␣0.8.8␣(1864941␣spike-06.openstreetmap.org)"
    copyright="OpenStreetMap␣and␣contributors"
    attribution="http://www.openstreetmap.org/copyright"
    license="http://opendatacommons.org/licenses/odbl/1-0/">
 <bounds minlat="49.2304000" minlon="6.6994000"
 maxlat="49.2408000" maxlon="6.7125000"/>
...
</osm>
```

Listing 2.1: OSM: Wrapper exemplarisch

Der Tag `<node>`, Listing 2.2, beschreibt jeden im exportierten Bereich vorkommenden Knoten. Grundsätzlich ist über den Knoten selbst kein direkter Rückschluss über das Objekt nachvollziehbar, lediglich, dass er existiert. In manchen Fällen besteht über Zusatzattribute die Möglichkeit einer Interpretation. Jeder Knoten hat eine Id, eine Version, das Changeset, einen Zeitstempel, einen User der den Knoten angelegt hat und die Koordinaten in Längen- und Breitengrad.

```
 <node id="290835255" visible="true" version="12" changeset="117112169"
 timestamp="2022-02-07T13:02:39Z" user="Teddy73" uid="136321"
 lat="49.2416049" lon="6.7130046"/>
 ...
  <node id="846717377" visible="true" version="4" changeset="37187126"
    timestamp="2016-02-13T13:55:29Z" user="FahRadler" uid="344561"
    lat="49.2387303" lon="6.7011775">
  <tag k="amenity" v="doctors"/>
  <tag k="wheelchair" v="yes"/>
 </node>
 ...
  <node id="906550053" visible="true" version="4" changeset="58852208"
```

```
13      timestamp="2018-05-10T14:20:38Z" user="nw520" uid="6895624"
14      lat="49.2395006" lon="6.7054403">
15    <tag k="highway" v="bus_stop"/>
16    <tag k="name" v="Ueberherrn␣Schillerstraße"/>
17    <tag k="network" v="saarVV"/>
18    <tag k="public_transport" v="platform"/>
19    <tag k="stop_id" v="47208"/>
20   </node>
```

Listing 2.2: OSM: Node exemplarisch

Das <way>-Objekt wird grundlegend für zwei Typen verwandt. Jedes Objekt hat zusätzlich Tags mit Attributen <tag k="" v=""> untergeordnet, mit denen Straßen, Wege und Objekte beschrieben werden.
Straßen und Wege sind erkennbar durch eine Typdefinition, in dem für das Key-Attribut beispielsweise k="highway" und das Value-Attribut v="footway" oder ein anderer Straßentyp definiert ist.
Ebenfalls ist der Wegname angegeben. Beispiel: <tag k="name" v="Herderweg"/>.

Gebäude sind erkennbar durch Attribute die <tag k="addr:" v=""> enthalten.
Zumeist sind bei den Gebäudeobjekten die gesamten Adressdaten, als auch die Typisierung gepflegt. Öffentliche Gebäude haben weitere, zusätzliche Attribute, die unter anderem Sportstätten oder religöse Gebäude beschreiben.

Gemeinsamkeiten aller Way-Objekte sind, dass alle beschreibenden Nodes als Referenz untergeordnet sind. Beispiel: <nd ref="906610865"/>. Bei Wegen und Straßen umfassen diese alle Kreuzungspunkte und bei Gebäudeobjekten definieren diese Knoten alle Ecken eines Gebäudes. Eine Besonderheit bei Gebäuden ist außerdem, dass ausnahmslos genau ein Knoten doppelt vorkommt, der sowohl den Anfang, als auch das Ende einer umfassenden Fläche definiert (vgl. Listing 2.4).

```
1  <way id="76979394" visible="true" version="1" changeset="5747585"
2      timestamp="2010-09-11T08:20:15Z" user="Nevek" uid="170961">
3    <nd ref="906610865"/>
4    <nd ref="906610899"/>
5    <tag k="highway" v="footway"/>
6    <tag k="name" v="Herderweg"/>
7  </way>
```

Listing 2.3: OSM: Way (Straße) exemplarisch

```
1  <way id="329707205" visible="true" version="2" changeset="37187126"
2      timestamp="2016-02-13T13:54:03Z" user="FahRadler" uid="344561">
3    <nd ref="3366359078"/>
4    <nd ref="3366359079"/>
```

```
  <nd ref="3366359034"/>
  <nd ref="3366359031"/>
  <nd ref="3366359078"/>
  <tag k="addr:city" v="Überherrn"/>
  <tag k="addr:country" v="DE"/>
  <tag k="addr:housenumber" v="10"/>
  <tag k="addr:postcode" v="66802"/>
  <tag k="addr:street" v="Holbeinweg"/>
  <tag k="building" v="yes"/>
</way>
```

Listing 2.4: OSM: Way (Gebäude) exemplarisch

Objekte vom Tag `<relation>` werden in der vorliegenden Arbeit nicht verwendet. Diese Objekte ”verbinden” beispielsweise Routen des öffentlichen Nahverkehrs, beschreiben unter anderem Buslinien und die darin enthalten Nodes, die wiederum die Haltepunkte definieren. Zusätzliche Attribute beschreiben die Relation und typisieren sie.

```
<relation id="58589" visible="true" version="393" changeset="128790596"
 timestamp="2022-11-11T19:36:11Z" user="Raa39" uid="13852369">
  <member type="way" ref="1077517364" role=""/>
  ...
  <tag k="colour" v="#00ADEF"/>
  <tag k="from" v="Saarlouis␣ZOB␣Kleiner␣Markt"/>
  <tag k="name" v="Bus␣409:␣Saarlouis␣ZOB␣Kleiner␣Markt
␣␣␣␣=&gt;␣Überherrn␣Schillerplatz"/>
  <tag k="network" v="saarVV"/>
  <tag k="network:wikidata" v="Q2209244"/>
  <tag k="operator" v="Kreisverkehrsbetriebe␣Saarlouis␣GmbH"/>
  <tag k="public_transport:version" v="2"/>
  <tag k="ref" v="409"/>
  <tag k="route" v="bus"/>
  <tag k="to" v="Überherrn␣Schillerplatz"/>
  <tag k="type" v="route"/>
  <tag k="url" v="https://wiki.osm.org/saarVV"/>
</relation>
```

Listing 2.5: OSM: Relation exemplarisch

2.3.2 GeoJSON

Das GeoJSON-Format ist unter dem RFC7946-Standard spezifiziert und unter `https://datatracker.ietf.org/doc/html/rfc7946` beschrieben.

GeoJSON ist ein offenes Standardformat zur Darstellung von geografischen Daten in JSON und wird hauptsächlich in Webanwendungen und bei Geoinformationssystemen verwendet, um räumliche Daten und deren Geometrien zu speichern und auszutauschen. GeoJSON kann verschiedene Arten von geografischen Objekten darstellen und ist sowohl menschen- als auch maschinenlesbar.

Ein GeoJSON-Objekt besteht typischerweise aus den Bestandteilen `type`, `coordinates` und optional auch `properties`:

- **type**: Gibt den Typ des GeoJSON-Objekts an, z. B. Point, LineString, Polygon und andere.
- **coordinates**: Ein Array, das die Koordinaten der Geometrie beschreibt. Die Struktur dieses Arrays hängt von der Art des GeoJSON-Objekts ab.
- **properties (optional)**: Ein Objekt, das zusätzliche Informationen (Attribute) zum geografischen Objekt speichert.

Point
Ein Point repräsentiert einen einzelnen Punkt im Raum, dargestellt durch ein Koordinatenpaar (Längengrad und Breitengrad).

```
1 {
2     "type": "Point",
3     "coordinates": [30, 10]
4 }
```

Listing 2.6: GeoJSON: Darstellung eines Point

`coordinates` wird durch ein Array mit genau zwei Werten dargestellt: dem Längengrad und dem Breitengrad. Optionale Höheninformationen können als drittes Element hinzugefügt werden.

LineString
Ein LineString repräsentiert eine Linie, die durch eine Reihe von Punkten (Koordinatenpaaren) definiert wird. Beispielsweise eine Straße, ein Flusslauf oder eine Route.

```
1 {
2     "type": "LineString",
3     "coordinates": [
4         [30, 10], [10, 30], [40, 40]
5     ]
6 }
```

Listing 2.7: GeoJSON: Darstellung eines LineString

`coordinates` hält ein Array von mindestens zwei Koordinatenpaaren, die die Linie definieren. Jedes Paar enthält den Längen- und Breitengrad, und optional eine Höhe als drittes Element im jeweiligen Koordinatenpaar.

Polygons
Ein Polygon repräsentiert eine Fläche, die durch eine oder mehrere geschlossene Linienringe definiert wird. Polygone sind einfache Formen wie Dreiecke oder Rechtecke, oder auch komplexe Konturen mit mehreren Außen- und Innenkanten (wie beispielsweise Inseln).

```
1 {
2     "type": "Polygon",
3     "coordinates": [
4         [[30, 10], [40, 40], [20, 40], [10, 20], [30, 10]]
5     ]
6 }
```

Listing 2.8: GeoJSON: Darstellung eines Polygon

`coordinates` ist bei Polygonen ein multidimensionales Array bestehend aus Koordinatenpaaren. Das erste Array beschreibt den äußeren Rand des Polygons (den Außenring) mit seinen Koordinatenpaaren. Optionale zusätzliche Arrays definieren innere Ringe, die sogenannte Löcher im Polygon darstellen.

Die Möglichkeit Polygone zu verschachteln (Löcher im Polygon), werden in dieser Arbeit nicht verwendet.

2.4 Daten und Zusammenhänge

Hinweis: Der Übersicht wegen, werden in den kommenden Beispielen auf beschreibende Attribute in den XML-Objekten, wie dem Ersteller und dem Zeitstempel, verzichtet und nur auf für die Arbeit relevante Daten bezogen.

2.4.1 Knoten von Straßen

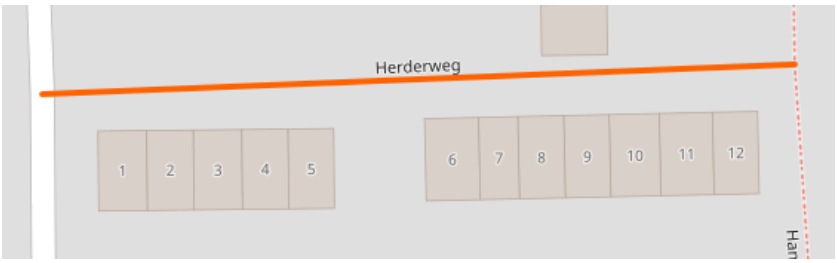

Abbildung 2.2: OSM-Kartenausschnitt Herderweg

Der Straßenzug des Herderwegs (`https://www.openstreetmap.org/way/76979394`) wird bei Openstreetmap folgendermaßen beschrieben:

```
<way id="76979394" ...>
    <nd ref="906610865"/>
    <nd ref="906610899"/>
    <tag k="highway" v="footway"/>
    <tag k="name" v="Herderweg"/>
</way>
```

Das Tag `<nd>` mit seinem Attribut `ref` (kurz: reference) beschreibt die Knoten der Straße. Der Zahlenwert in `ref` repräsentiert die Id der zugehörigen Node.

2.4.2 Knoten von Gebäude

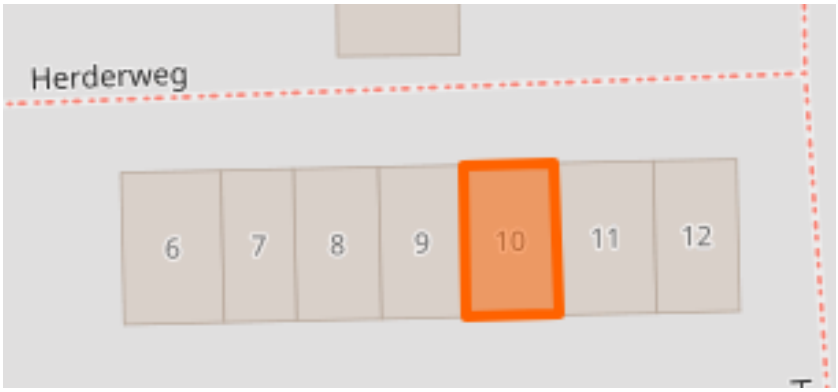

Abbildung 2.3: OSM-Kartenausschnitt Herderweg 10

Das Gebäude (`https://www.openstreetmap.org/way/329707214`) im Herderweg mit der Hausnummer 10, wird bei Openstreetmap folgendermaßen beschrieben:

```
<way id="329707214" ...>
    <nd ref="3366359046"/>
    <nd ref="3366359047"/>
    <nd ref="3366359002"/>
    <nd ref="3366359001"/>
    <nd ref="3366359046"/>
    <tag k="addr:city" v="Überherrn"/>
    <tag k="addr:country" v="DE"/>
    <tag k="addr:housenumber" v="10"/>
    <tag k="addr:postcode" v="66802"/>
    <tag k="addr:street" v="Herderweg"/>
```

```
    <tag k="building" v="yes"/>
</way>
```

Das Tag <nd> mit seinem Attribut `ref` beschreibt bei einem Way-Objekt, das ein Gebäude ist, die Knoten die das Gebäude umfassen. Der Wert in `ref` repräsentiert die Id der referenzierten Node.

2.4.3 Straßen und Gebäude

Zusammenhänge zwischen Straßen und den auf ihnen befindlichen Gebäude werden lediglich über den Straßennamen gebildet.

Die Nodes die in den Weg-Objekten hinterlegt sind, an dem das Gebäude angeschlossen ist, sind nicht in den Daten vorhanden (vgl. OSM-Objekt aus Abschnitt 2.4.1 und Abschnitt 2.4.2). Die einzige Gemeinsamkeit bildet der Name der Straße und die für das Gebäude hinterlegte Adresse. Demnach sagt eine Datenrelation folgendes aus:

Eine Straße mit einem Namen hat beliebig viele Gebäude, deren Tag-Attribut v, gleich dem Namen dieser übergeordneten Straße ist.

2.4.4 Nachbargebäude

Gemeinsame Node-Objekte bilden ebenfalls Zusammenhänge. Liegen Gebäude unmittelbar nebeneinander, so teilen sich diese Objekte manche Knoten.

Beispiel:
Das Gebäude-Objekt mit der Hausnummer 6 und das Objekt mit der Nummer 7 aus der Beispielstraße in Abbildung 2.2, liegen nebeneinander und teilen sich zwei Knoten:

```
<way id="329707196" ...>
    <nd ref="3366359042"/>
    <nd ref="3366358996"/>
    <nd ref="3366358998"/>
    <nd ref="3366359043"/>
    <nd ref="3366359042"/>
    <tag k="addr:city" v="Überherrn"/>
    <tag k="addr:country" v="DE"/>
    <tag k="addr:housenumber" v="6"/>
    <tag k="addr:postcode" v="66802"/>
    <tag k="addr:street" v="Herderweg"/>
    <tag k="building" v="yes"/>
</way>
```

```
<way id="329707187"...>
    <nd ref="3366359043/>
    <nd ref="3366359044/>
    <nd ref="3366358999/>
    <nd ref="3366358998/>
    <nd ref="3366359043/>
    <tag k="addr:city" v="Überherrn"/>
    <tag k="addr:country" v="DE"/>
    <tag k="addr:housenumber" v="7"/>
    <tag k="addr:postcode" v="66802"/>
    <tag k="addr:street" v="Herderweg"/>
    <tag k="building" v="yes"/>
</way>
```

Liegen Gebäude nebeneinander, sind aber durch nicht direktes Verbinden getrennt, so ist kein Zusammenhang durch gemeinsame Daten möglich. Als Beispiel dient das Haus mit der Nummer 5 aus der Beispielstraße Abbildung 2.2, Herderweg:

```
<way id="329707216" ...>
    <nd ref="3366359038"/>
    <nd ref="3366359039"/>
    <nd ref="3366358892"/>
    <nd ref="3366358891"/>
    <nd ref="3366359038"/>
    <tag k="addr:city" v="Überherrn"/>
    <tag k="addr:country" v="DE"/>
    <tag k="addr:housenumber" v="5"/>
    <tag k="addr:postcode" v="66802"/>
    <tag k="addr:street" v="Herderweg"/>
    <tag k="building" v="yes"/>
</way>
```

Ein direkter Vergleich mit dem Gebäude mit der Nummer 6, aus Abschnitt 2.4.4 zeigt, dass es keine gemeinsamen Knoten gibt. Dies ist nur durch die Bildung der übergeordneten Relation zu einer gemeinsamen Straße umsetzbar.

2.5 Export

Um Daten über die Openstreetmap-Schnittstelle `https://www.openstreetmap.org/api/0.6` zu bekommen, ist es möglich diese Schnittstelle für einzelne, wenige Daten über einen HTTP-Request anzufragen. Sollen größere Datenmengen für ganze Bereiche wie den oben beschriebenen Ortsteil exportiert werden, sind weitere Werkzeuge die einen Schnittstellenzugriff erlauben notwendig.

2.5.1 Export mit Overpass-Turbo

Mit Hilfe der Overpass-Turbo-API (`https://overpass-turbo.eu/`) können OSM-Daten durch eine Skriptsprache in verschiedene Formate exportiert werden.
Eine Hilfe und ein Verweis auf eine ausführliche Dokumentation ist auf der Webseite der Schnittstelle zu finden.

```
1 /* Überherrn - Wohnstadt */
2 nwr(49.2304,6.6994,49.2408,6.7225);
3 out;
4
5 /* Überherrn + OT Wohnstadt */
6 nw(49.22982,6.68977,49.25013,6.71775);
7 out;
```

Listing 2.9: Overpass-Query

In der zweiten Zeile des Listing 2.9 ist eine Abfrage mit dem Befehl `nwr` zu sehen, mit der für den angegebenen Bereich alle vorhandenen *Nodes*, *Ways* und *Relations* gesucht werden sollen. Der Ausdruck in der nächsten Zeile befiehlt die Ausgabe der Daten. Das Standardausgabeformat ist XML, sofern nicht anders angegeben.

Die Abfrage in Zeile 6 des Listings 2.9 mit dem Befehl `nw` umfasst einen größeren Bereich und beschränkt sich auf die Auswahl von den ausschließlich hinterlegten *Nodes* und *Ways*.

Die Übergabe der Breiten- und Längengrade in den Beispielen beschreiben eine sog. *Bounding-Box*. Die Parameter sind in der Reihenfolge: Süden, Westen, Norden und Osten.

Nach dem Ausführen werden die bekannten Daten direkt in der Karte angezeigt. Durch den Wechsel der Karte zu den Daten, werden diese im angegeben Format (hier: XML) angezeigt. Der tatsächliche Export der Daten erfolgt über den Klick auf den Export-Button und durch Auswahl des gewünschten Formats. Für die XML-Daten müssen dann ”OSM-Rohdaten” gewählt werden. Im nachfolgenden Screenshot wurden die Abfragen bereits ausgeführt und die gefundenen Nodes innerhalb des angegebenen Bereichs sind markiert.

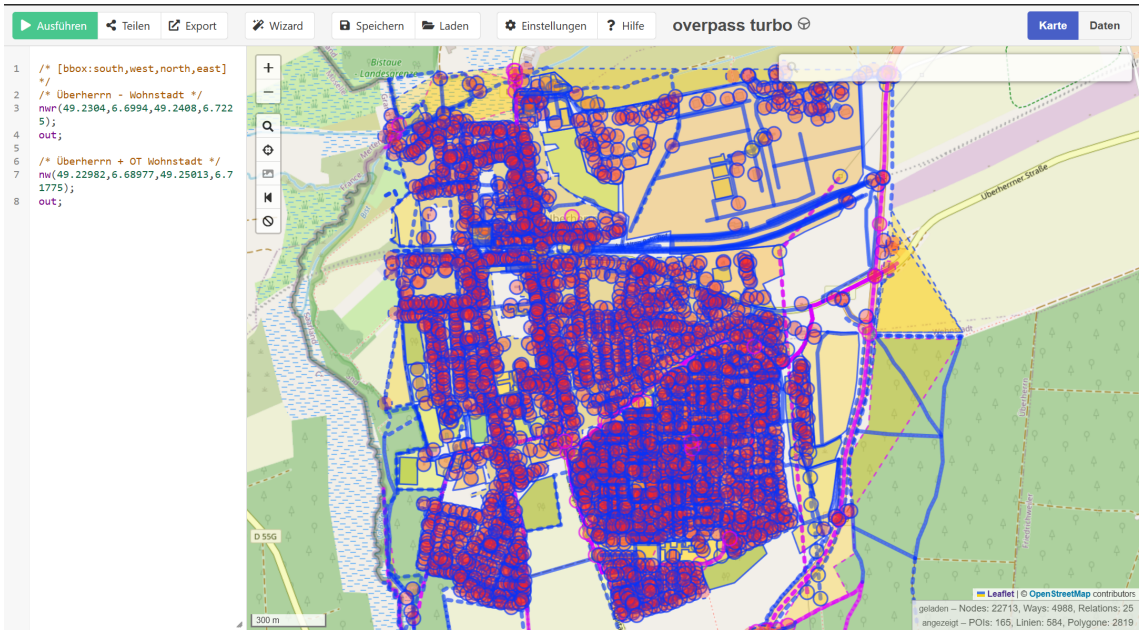

Abbildung 2.4: Overpass-Turbo

2.5.2 Export mit OSMOSIS

Osmosis ist eine Kommandozeilenbasierte Java-Anwendung zur Verarbeitung von OSM-Daten. Das Werkzeug besteht aus verschiedenen Komponenten die miteinander verbunden werden können. Osmosis wurde entwickelt, um möglichst leicht neue Funktionen hinzufügen zu können, ohne dass gängige Aufgaben wie die Datei- und Datenbankverarbeitung neu geschrieben werden müssen.

Eine ausführliche Beschreibung und eine Download-Quelle sind auf den offiziellen Seiten von Openstreetmap unter `https://wiki.openstreetmap.org/wiki/Osmosis` zu finden.

Das Werkzeug Osmosis hilft zum Beispiel beim überführen von pbf[2]-Dateien in ein osm-Format. Vorteile des pbf-Dateiformats sind, es benötigt die Hälfte des Speichers im Vergleich zu gzip[3]. Das Format kann fünf mal schneller als gzip gelesen und geschrieben werden und ist darauf ausgelegt, wahlfreien Zugriff auf Dateiblockebene zu unterstützen. Jeder Datei-Block ist unabhängig dekodierbar und enthält etwa achttausend OSM Einträge in der Standardkonfiguration. [11]

[2]Protocolbuffer Binary Format
[3]`https://de.wikipedia.org/w/index.php?title=Gzip&oldid=244425040`

```
## Execute OSMOSIS
PS C:\...\osmosis-0.48.3\bin> .\osmosis.bat

## Modify pbf to osm
.\osmosis --read-xml ./wohnstadt.osm --write-pbf wohnstadt.osm.pbf
```

Listing 2.10: Osmosis-Beispiel

Hinweis:
Zum Ausführen muss in der `osmosis.bat` ein lokales JDK eingetragen werden:

```
IF "%JAVACMD%"=="" set JAVACMD="\.jdks\corretto-1.8.0_352\bin\java.exe"
```

Listing 2.11: osmosis.bat - lokales JDK-Verzeichnis

Der Export einer pbf-Datei und das nachträgliche Wandeln in ein lesbares osm-Format gestaltet den Weg über OSMOSIS umständlicher. Die grafische Darstellung mit der intuitiven Bedienung, sowie der verhältnismäßig geringere Aufwand über die Overpass-Turbo-API machen die Exportmöglichkeiten von OpenStreetMap-Daten einfacher.

2.6 Eingesetzte Frameworks

2.6.1 Spring Boot

Spring Boot ist ein Framework, dass auf dem Java-Framework Spring aufbaut und ermöglicht das schnelle Erstellen von eigenständigen Spring-Anwendungen, die ”Out of the Box” laufen, da viele Prozesse und Konfigurationen automatisiert ablaufen. Spring Boot wird häufig in der Entwicklung von Microservices und REST-APIs verwendet, da es viele Features mitbringt, die die Entwicklung und den Betrieb dieser Architekturen beschleunigen.

Das Spring Framework wurde im Jahr 2002 von Rod Johnson als Lösung für komplexe Java-EE-Anwendungen entwickelt. Ziel war es, die Entwicklungszeit und Komplexität neuer Anwendungen zu reduzieren. Bis heute hat sich Spring Boot als führendes Framework für Microservices und moderne Cloud-Native Anwendungen etabliert und wird kontinuierlich weiterentwickelt.

Ein bedeutender Vorteil ist ein eingebetteter Server. Spring Boot-Anwendungen können mit diesem Server (Apache Tomcat) aus der IDE heraus gestartet werden. Das ermöglicht den Einsatz als eigenständige, ausführbare JAR-Datei und spart bei der Entwicklung aufwendige Serverinstallationen und -konfigurationen.

Mit der `@SpringBootApplication`-Annotation wird eine Vielzahl von Standard-Beans und

Konfigurationen automatisch bereitgestellt. Diese Konfiguration ist eine zentrale Funktion von Spring Boot und kann bei Bedarf durch benutzerdefinierte Einstellungen überschrieben bzw. erweitert werden.

Website: `https://spring.io/`

2.6.2 OpenAPI

OpenAPI ist ein Standardformat zur Beschreibung und Dokumentation von RESTful APIs. OpenAPI bietet eine einheitliche, maschinenlesbare Struktur, um Endpunkte, Datenstrukturen und Methoden einer Schnittstelle zu beschreiben. Typischerweise wird die Beschreibung in YAML- oder JSON-Format verfasst. Durch den Standard OpenAPI können Entwickler API-Spezifikationen generieren, validieren und interaktive Dokumentationen bereitstellen.

Mittels OpenAPI kann Code auch automatisch generiert werden. Tools wie Swagger Codegen nutzen die OpenAPI-Spezifikationen zur Generierung von Client-Bibliotheken und Server-Stub-Code in verschiedenen Programmiersprachen. Außerdem vereinfacht und ermöglicht OpenAPI Schnittstellentests und Validierungen durch eine entsprechende Spezifikation.

Das Projekt startete in 2010 ursprünglich als Swagger. Es wurde entwickelt, um die Spezifikationen von APIs standardisiert zu dokumentieren und zu kommunizieren. Heute ist OpenAPI ein weit verbreiteter Standard für API-Dokumentation und wird von großen Plattformen und Tools unterstützt.

2.6.3 Swagger-UI

Swagger-UI ist ein Open-Source-Tool zur Visualisierung und Interaktion mit API - Spezifikationen. Swagger generiert automatisch eine interaktive Web-Oberfläche. Swagger-UI wird in der API-Dokumentation eingesetzt, um eine leicht zugängliche und verständliche Darstellung von RESTful APIs bereitzustellen, sodass Entwickler die API-Aufrufe testen und verstehen können.

Ein großer Vorteil ist insbesondere, dass Benutzer- und Entwickler, die mit einer Schnittstelle nicht vertraut sind schnell nachvollziehen können, wie die API verwendet wird, ohne die zugrunde liegende Implementierung verstehen zu müssen.

Das Erstellen einer Swagger-UI mit Spring Boot ist mit einfachen Mitteln schnell zu bewerkstelligen, indem in die Abhängigkeiten (`pom.xml`), die Spring-Doc Bibliothek integriert wird. Per Standard ist die Swagger-UI unter `http://localhost:8080/swagger-ui/index.html` erreichbar. In den `application.properties` einer Spring Boot Anwendung können weitere Parameter angepasst werden.

Zur Dokumentation eines Endpunkts, können verschiedene Annotationen im Controller

benutzt werden:

```
1 @Operation(summary = "Compute␣way␣and␣respond␣with␣GeoJson-Object.")
2 @ApiResponse(responseCode = "404", description = "Way␣not␣computable.")
```

Listing 2.12: Swagger-Annotationen

`@Operation` legt eine Beschreibung für einen Methodenaufruf fest, während bei der `@ApiResponse`-Annotation die Beschreibung für einen Statuscode einer Antwort festgelegt wird.

2.6.4 Leaflet

Leaflet ist eine Open-Source-JavaScript-Bibliothek zur Erstellung von interaktiven Karten für Webanwendungen. Sie wird verwendet, um dynamische Karten auf Webseiten einzubinden und interaktiv zu gestalten. Leaflet ist bekannt für die Arbeit mit OpenStreetMap-Daten und bietet eine API für die Einbindung und Anpassung von Kartenfunktionen wie Markern, Popups, Linien und verschiedenen Layern.

Leaflet wurde im Jahr 2010 von Vladimir Agafonkin gegründet, um leichte, einfache und anpassbare Lösungen für die Erstellung interaktiver Karten zu schaffen. Ziel war es, eine leichtgewichtige Alternative zu Google Maps oder anderen Kartenlösungen anzubieten. Mittlerweile ist Leaflet eine der am weitesten verbreiteten JavaScript-Bibliothken für Karten und wird aktiv weiterentwickelt.

Leaflet bietet eine kompakte Basisfunktionalität, die über Plugins erweitert werden kann. Weiterhing ist Leaflet für mobile Geräte optimiert und unterstützt Touch-Gesten wie Zoomen und Schwenken. Karten sind automatisch responsiv und passen sich an unterschiedliche Bildschirmgrößen an.

Website: `https://leafletjs.com/`

2.6.5 Bootstrap

Bootstrap ist ein Frontend-Framework zur Erstellung von Webseiten, die für Mobilgeräte optimiert sind. Es bietet eine Sammlung von vorgefertigten HTML-, CSS- und JavaScript-Komponenten wie Layout-Rastern, Buttons, Formularen, Navigationselementen und weiteren Elementen. Bootstrap erleichtert das Design und die Entwicklung, indem es einheitliche Stile und eine anpassbare Struktur zur Verfügung stellt.

Bootstrap wurde 2010 bei Twitter von den Entwicklern Mark Otto und Jacob Thornton als interner Stil- und Design-Guide entwickelt. Die Idee war, eine einheitliche Designsprache innerhalb von Twitter zu etablieren. Ein Jahr später veröffentlichte Twitter Bootstrap als Open-Source-Projekt auf GitHub. Es wurde schnell populär und wurde das weltweit beliebteste CSS-Framework. Im Jahr 2020 erschien Bootstrap 5 und führte ein JavaScript-Modul

ohne Abhängigkeit von jQuery ein. Dies ermöglichte mehr Flexibilität und modernisierte das Framework.

Website: `https://getbootstrap.com/`

3 Algorithmus

In diesem Kapitel werden zwei spezifische Algorithmen zur Lösung des Problems der Bestimmung von kürzesten Wegen vorgestellt und miteinander verglichen. Anschließend wird die Modellierung und Implementierung eines der Algorithmen in einer Java-Applikation dargestellt, um dessen praktische Anwendbarkeit zu demonstrieren und die theoretischen Erkenntnisse in die Praxis zu übertragen. Dieses Kapitel legt damit die Basis für die technische Umsetzung und vertieft das Verständnis der algorithmischen Ansätze im Kontext der gewählten Problemstellung.

3.1 Dijkstra vs. A-Stern

Uninformierte Algorithmen, wie der Dijkstra-Algorithmus, haben keine zusätzliche Informationen über das Ziel und durchsuchen einen Graphen systematisch. Die Algorithmen arbeiten nach festen Regeln ohne Bestimmung einer Restdistanz.

Informierte Algorithmen hingegen nutzen Zusatzinformationen (wie beispielsweise Heuristiken, wie die geschätzte Entfernung zum Ziel), um effizientere Entscheidungen zu treffen und schneller zum Ziel zu gelangen.

3.1.1 Dijkstra-Algorithmus

Der Dijkstra-Algorithmus ist nach seinem Erfinder Edsger W. Dijkstra, einem niederländischen Informatiker, benannt und löst das Problem der kürzesten Pfade für einen gegebenen Startknoten. Er berechnet den kürzesten Pfad zwischen einem Startknoten und dem Zielknoten (oder allen übrigen Knoten) in einem kantengewichteten Graphen. Vorausgesetzt der Graph ist nicht negativ. [5]

Die Idee des Algorithmus ist es, immer der Kante zu folgen, die die kürzeste Entfernung vom Startknoten zu sein scheint. Andere Kanten werden erst dann betrachtet, wenn alle kürzeren Entfernungen berücksichtigt wurden. Diese Art und Weise stellt sicher, dass bei Erreichen eines Knotens kein kürzerer Pfad zu ihm existiert. Eine berechnete Distanz zwischen einem Startknoten und einem besuchten Knoten wird gespeichert. Die kumulierten Distanzen zu noch nicht betrachteten Knoten können sich im Laufe des Algorithmus verändern, nämlich verringern. Dieses Schema wird so lange fortgesetzt, bis die Distanz zum Zielknoten berechnet wurde oder die Distanzen aller Knoten zum Startknoten bekannt sind. [4]

3.1.2 A-Stern-Algorithmus

Der A*-Algorithmus ("A Stern") gehört zu den informierten Suchalgorithmen. Er berechnet einen kürzesten Pfad zwischen zwei Knoten in einem Graphen mit positiven Kantengewichten. Der Algorithmus gilt als Verallgemeinerung und Erweiterung des Dijkstra-Algorithmus, der in vielen Fällen aber auch umgekehrt auf den Dijkstra-Algorithmus reduziert werden kann.

Im Gegensatz zu uninformierten Suchalgorithmen verwendet der A*-Algorithmus eine Heuristik, um zielgerichtet zu suchen und damit die Laufzeit zu verringern. Der Algorithmus ist vollständig und optimal, was bedeutet, dass immer eine optimale Lösung gefunden wird, sofern eine existiert. [1]

Der A*-Algorithmus untersucht die Knoten zuerst, die wahrscheinlich schnell zum Ziel führen. Um den vielversprechendsten Knoten zu berechnen, wird allen Knoten x jeweils ein Wert $f(x)$ zugeordnet, der eine Abschätzung angibt, wie weit der Pfad vom Start zum Ziel unter Verwendung des betrachteten Knotens im optimalen Fall ist. Der Knoten mit den geringsten Kosten wird als nächster untersucht.

$$f(x) = g(x) + h(x)$$

Für einen Knoten x bezeichnet $g(x)$ die bisherigen Kosten vom Startknoten aus, um x erreichen zu können. $h(x)$ bezeichnet die geschätzten Kosten (Heuristik) von x bis zum Zielknoten. Die verwendete Heuristik darf die Kosten nie überschreiten. Für eine Wegsuche ist die euklidische Distanz eine geeignete Annahme: Die tatsächliche Strecke ist nie kürzer als die Luftlinie.

Der A*-Algorithmus ist vollständig: falls eine Lösung existiert, wird sie gefunden. Der Algorithmus ist optimal: es wird immer die optimale Lösung gefunden und wenn mehrere existieren, wird nur eine gefunden. Weiterhin ist der A*-Algorithmus optimal effizient, das heißt es gibt keinen anderen Algorithmus, der die Lösung unter Verwendung der gleichen Heuristik schneller findet. [3]

3.1.3 Implementierung

Durch die Betrachtung der gängigen Kürzeste-Wege-Algorithmen in der Studienarbeit "Kürzeste Wege auf allgemeinen Graphen" [14] ist der A-Stern-Algorithmus für eine Berechnung von Wegen mit Geokoordinaten der am Besten geeignete Algorithmus.

	negative Kanten	Zeitkomplexität (n: Knoten, m: Kanten)
Dijkstra Algorithmus	Nein	$O(n^2)$
A-Stern Algorithmus	Nein	$O(n \cdot \log n + m \cdot n)$ (mit PriorityQueue)
Bellman-Ford Algorithmus	Ja	$O(n^2)$
Floyd-Warshall Algorithmus	Ja	$O(n^3)$

Tabelle 3.1: Vergleich gängiger Algorithmen

Zur Bestimmung eines optimalen Weges zwischen zwei Zielen ist es mit den Daten die benutzt werden sollen nicht möglich, dass ein negatives Kantengewicht zum Tragen kommt. Weiterhin ist der A-Stern-Algorithmus nach der Tabelle 3.1 der schnellste Algorithmus bei Betrachtung der Zeitkomplexität.

3.2 Heuristik

Die in der Implementierung des A*-Algorithmus verwendete Heuristik ist eine Implementierung der Distanz mittels Haversine-Formel, da Berechnungen mit dieser Methode die Genauigkeit für sehr kleine Entfernungen am Höchsten ist. Das Diagramm zeigt den Großkreisabstand (rot gezeichnet) zwischen zwei Punkten auf einer Kugel, P und Q, darstellt.

Diese Heuristik misst die kürzeste direkte Entfernung zwischen zwei Punkten auf kugelförmigen Oberflächen.

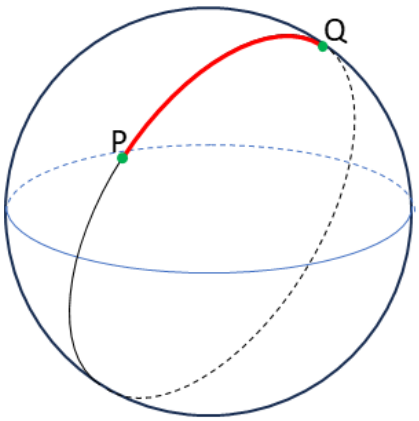

Abbildung 3.1: Distanz auf einer Kugel

Für zwei Punkte mit den geografischen Koordinaten Breitengrad ϕ_P, Längengrad λ_P und Breitengrad ϕ_Q, Längengrad λ_Q lautet die Haversine-Formel:

$$a = \sin^2\left(\frac{\Delta\phi}{2}\right) + \sin^2\left(\frac{\Delta\lambda}{2}\right) \cdot \cos\left(\phi_P\right) \cdot \cos\left(\phi_Q\right)$$

$$c = 2 \cdot \arcsin\left(\sqrt{a}\right)$$

$$d = R \cdot c$$

$\Delta\phi = \phi_Q - \phi_P$ ist die Differenz der Breitengrade (in Radianten)
$\Delta\lambda = \lambda_Q - \lambda_P$ ist die Differenz der Längengrade (in Radianten)

Der mittlere Erdradius[15] R beträgt etwa 6.378.137m und d steht für die berechnete Entfernung zwischen den beiden Punkten auf der Erdoberfläche.

Die Implementierung in JAVA ist folgendermaßen umgesetzt:

```
1  public class HaversineFormula implements IScorer<NodeDTO> {
2
3      @Override
4      public double computeDistance(NodeDTO start, NodeDTO target) {
5          double R = 6378.137;
6          double dLat = Math.toRadians(target.getLatitude()
7              - start.getLatitude());
8          double dLon = Math.toRadians(target.getLongitude()
9              - start.getLongitude());
10         double lat1 = Math.toRadians(start.getLatitude());
11         double lat2 = Math.toRadians(target.getLatitude());
12
13         double a = Math.pow(Math.sin(dLat / 2), 2)
14                 + Math.pow(Math.sin(dLon / 2), 2)
15                 * Math.cos(lat1) * Math.cos(lat2);
16         double c = 2 * Math.asin(Math.sqrt(a));
17         return R * c;
18     }
19 }
```

Listing 3.1: HaversineFormula.java

3.3 Berechnungen eines Weges

Alle Knoten werden in einer Tabelle erfasst. Die Vorgänger-Knoten bleiben zunächst leer und als Gesamtkosten wird für den Startknoten der Wert 0 initial eingetragen. Alle anderen Knoten werden mit ∞ gefüllt.

Minimale Restkosten werden ebenfalls mit den zuvor berechneten Restkosten zum Zielknoten befüllt. Die Summe aller Kosten ist derzeit nicht bekannt und werden mit ∞ vor befüllt. Ausgenommen der Startknoten, dessen Restkosten die errechneten Minimalkosten sind.

Gespeichert werden folgende Informationen zur Bestimmung eines kürzesten Pfades:

die Kosten, also das Kantengewicht von einem Knoten zu seinem Nachbarn, die Gesamtkosten, also die Summe aller Teilkosten vom Startknoten zu einem bestimmten Knoten über eventuelle Zwischenknoten und die Restkosten, Mindestkosten berechnet durch die euklidische Distanz.

Die Summe aller Kosten werden gebildet aus den Gesamtkosten und den minimalen Restkosten zum Ziel. In jedem Durchlauf, werden dann die jeweiligen Kosten zu den möglichen Vorgängern berechnet und der jeweils günstigste gespeichert.

Am Ende entsteht eine Tabelle, in der zu allen Knoten ein möglicher Vorgänger steht und die Summe aller Kosten zum Ziel. Über einen Backtrace wird dann der optimale Weg ermittelt. Im Falle, dass die minimalen Restkosten zum Ziel größer sind, als die Vorgängerkosten, wird der Knoten nicht weiter betrachtet. Der Weg kann dann nicht optimal sein.

3.4 Informelle Beschreibung

Vorbereitung

1. Tabelle erstellen mit allen Knoten und den Attributen: Vorgänger und Gesamtdistanz
2. Gesamtdistanz des Startknotens auf 0 setzen und für alle anderen auf unendlich

Abarbeitung
Solange die Tabelle noch Knoten enthält, werden alle Elemente mit jeweils der kleinsten Gesamtdistanz genommen und folgendes gemacht:

1. Speichere, das Knoten betrachtet wurde
2. Prüfung: Ist das betrachtete Element der Zielknoten? Wenn ja, Backtrace über Vorgänger bis zum Start
3. Betrachte Nachbarknoten die noch in Tabelle sind:
    a) Kalkuliere Gesamtdistanz aus Summe der Gesamtdistanz des betrachteten Knotens und Distanz zum Nachbarelement
    b) Prüfe: Errechnete Gesamtdistanz kleiner als aktuelle Gesamtdistanz, dann überschreibe

3.5 Modell

Das Datenmodell für die Anwendung wird durch DTOs[1] repräsentiert. Mit diesen DTOs werden Daten zwischen dem Webservice und dem User-Interface ausgetauscht.

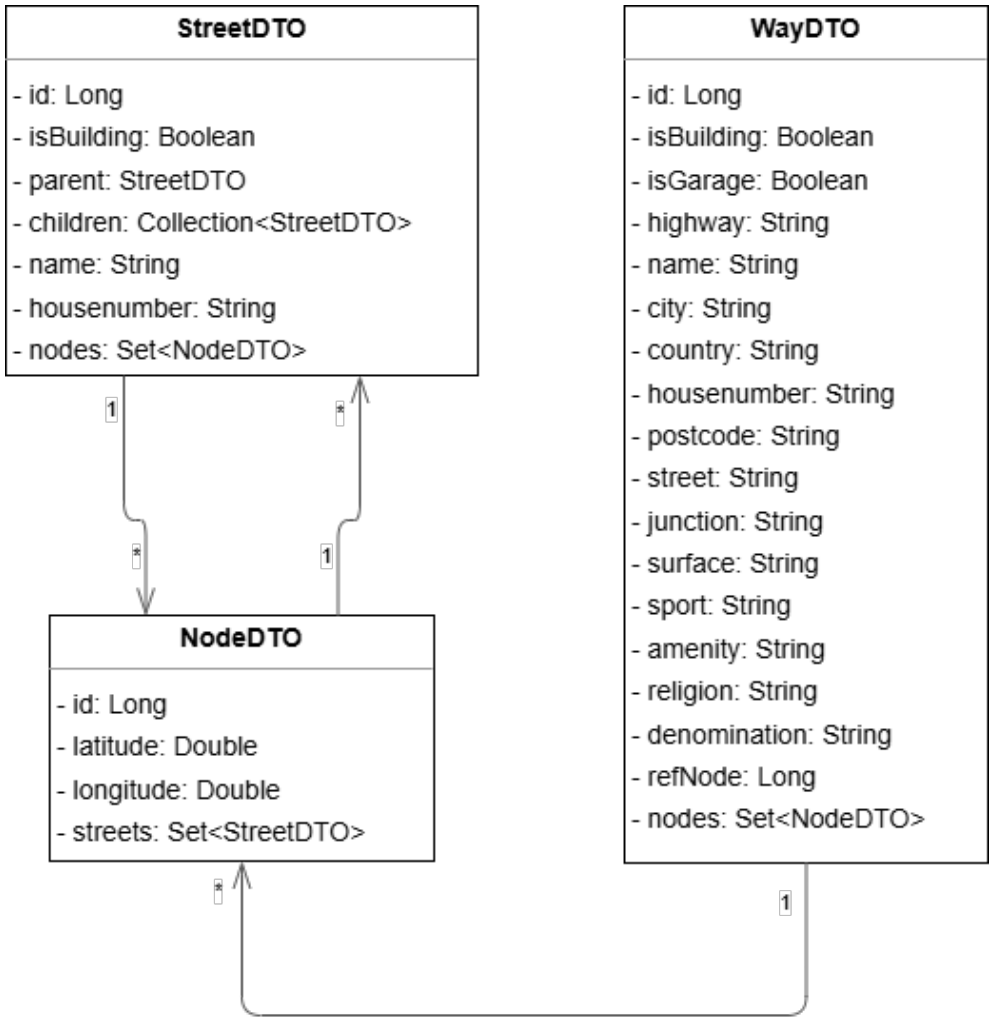

Abbildung 3.2: UML: DTO-Model

3.5.1 StreetDTO

Das `StreetDTO`-Objekt ist das zentrale Objekt im Kosmos der Applikation. Innerhalb des Objekts wird eine Eltern-Kind-Beziehung als One-To-Many-Relation hergestellt. Ist ein zu speicherndes Weg-Objekt als Straße und nicht als Gebäude identifiziert, wird damit ein Eltern-Objekt gespeichert. Die Identifikation erfolgt zum einen über das `isBuilding`-Flag und über den Inhalt von Daten. Ein Eltern-Objekt trägt eine Collection an Kind-Elementen und keine Hausnummer.

[1] Data Transfer Object

Alle StreetDTO-Objekte, die als Gebäude und nicht als Straße identifizierbar sind, werden demjenigen Elternobjekt als Kind hinzugefügt, dass den gleichen Straßennamen trägt. Das stellt innerhalb der Abhängigkeiten sicher, das ein Gebäude immer Teil einer Straße ist und erleichtert im späteren Berechnen die Lokalisierung der Ziele. Weiterhin hat jedes Kind-Element eine Hausnummer und das zugehörige Eltern-Element.

In jedem Fall sind in jedem StreetDTO-Objekt alle beschreibenden Knoten als Set mit NodeDTOs gespeichert.

3.5.2 NodeDTO

Die Klasse `NodeDTO` bildet die Knoten innerhalb der Applikation ab. Jeder Knoten hat zur Identifikation eine Id und jeweils die Koordinaten als Breiten- und Längengrad. Ein Knoten kann eine Referenz für ein `WayDTO` sein und/oder eine Referenz in einem `StreetDTO`. Die Beziehung ist in beiden Fällen jeweils eine Many-To-Many-Beziehung.

3.5.3 WayDTO

Weg-Objekte (Wege und Gebäude) werden zwar in der `WayDTO`-Klasse mitgespeichert, sind aber im Rahmen der Applikation von geringer Wichtigkeit.

Die WayDTO-Objekte werden verarbeitet, um `StreetDTO`-Objekte zu erstellen und den Referenzknoten zur Lokalisierung eines Gebäudeknotens festzustellen. Die daraus extrahierbaren Information werden eingangs erfasst und anschließend in der Datenbank gespeichert.

3.6 (Un)Marshalling

Marshalling ist das Umwandeln von strukturierten oder elementaren Daten in ein Format, das die Übermittlung an andere Prozesse ermöglicht. Auf Empfängerseite werden aus diesem Format die Daten in ihrer ursprünglichen Struktur wiederhergestellt, was als Unmarshalling bezeichnet wird. Marshalling ist ähnlich und abhängig vom Kontext ein Synonym für Serialisierung.

Da die Dateneingabe über ein String-XML-Objekt erfolgt, werden die Daten gelesen, validiert und in JAVA-Objekte gewandelt. Dazu wird mittels XML-Marshaller die Datei als zusammenhängender String eingelesen und versucht in ein XML-Model zu wandeln.

Das Model besteht aus einzelnen Klassen, die die in der OSM-Datei vorkommenden Tags mit ihren Attributen repräsentieren. Diese XML-Objekte werden wiederum auf die zuvor beschriebenen DTOs gemappt, um die Daten einerseits in der Datenbank zu speichern und andererseits während der Laufzeit der Applikation verwenden zu können.

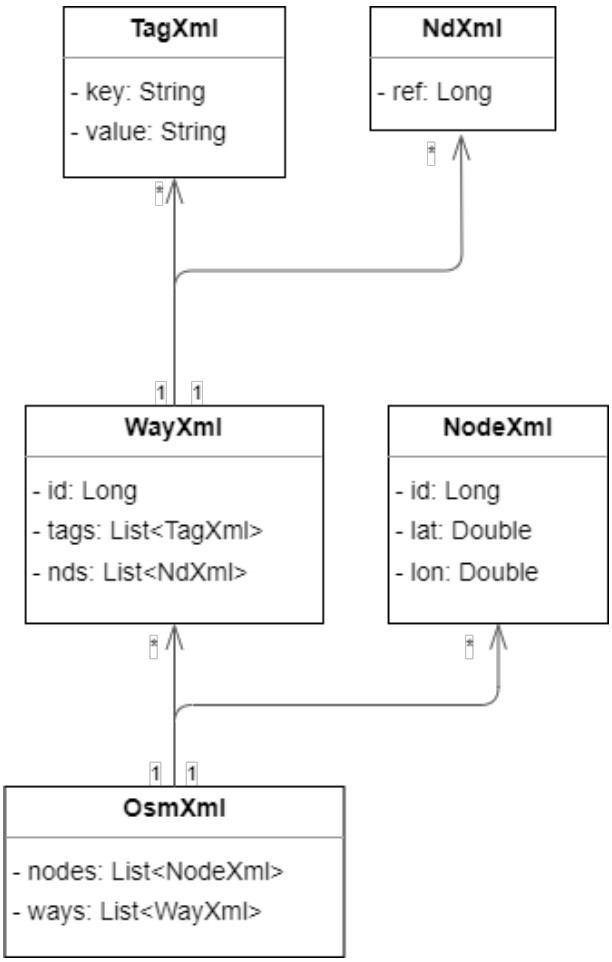

Abbildung 3.3: UML: XML-Model

3.7 Ausgabe

Die Ausgabe des WebService nach der erfolgreichen Berechnung eines Weges ist im GeoJSON- Format.

Die Applikation beinhaltet eine Abbildung der GeoJSON-Struktur als zusammenhängende Objekte. Damit kann nach einer Wegberechnung der gesamte Weg mit seinen Eigenschaften an das GeoJSON-Objekt überführt werden und entsprechend aufgebaut. Das Deserialisieren in JSON übernimmt der REST-Controller automatisch und das Frontend kann die entsprechenden Informationen entgegen nehmen und anzeigen lassen.

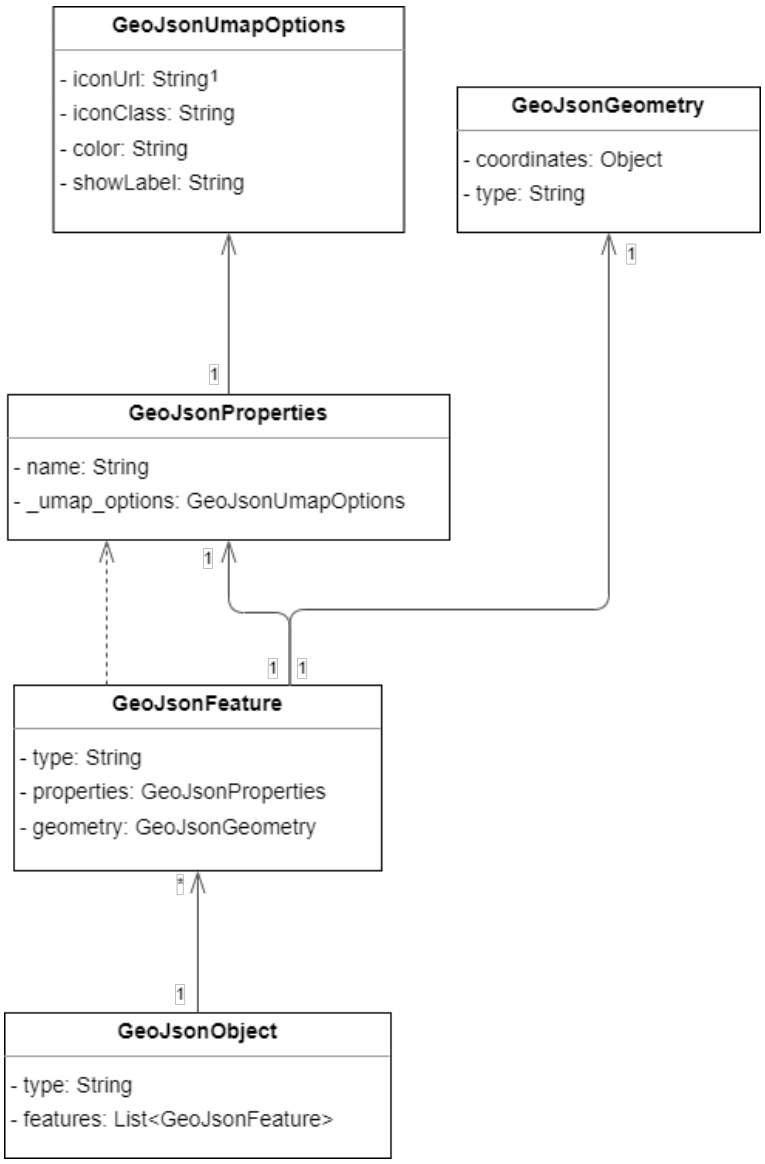

Abbildung 3.4: UML: GeoJSON-Model

Nachfolgend ein Beispiel einer GeoJSON-Ausgabe, jedoch reduziert. Auf zusätzliche Informationen, wie farbige Start- und Zielmarker wird verzichtet, als auch auf ausführliche Weginformationen.

```
1 {
2   "type": "FeatureCollection",
3   "features": [
4     {
5       "type": "Feature",
6       "properties": {
7         "_umap_options": {},
8         "name": "Start"
```

```
      },
      "geometry": {
        "type": "Point",
        "coordinates": [6.707776, 49.234935]
      }
    },
    {
      "type": "Feature",
      "properties": {
        "_umap_options": {},
        "name": "Ziel"
      },
      "geometry": {
        "type": "Point",
        "coordinates": [6.703935, 49.235881]
      }
    },
    {
      "type": "Feature",
      "properties": {},
      "geometry": {
        "type": "LineString",
        "coordinates": [
          [6.707762, 49.234920],
          [6.706711, 49.234899],
          [6.706695, 49.235384],
          [6.705992, 49.235391],
          [6.705496, 49.235382],
          [6.704935, 49.235380],
          [6.704672, 49.235386],
          [6.704667, 49.235508],
          [6.703940, 49.235515],
          [6.703921, 49.235876],
          [6.703927, 49.235918]
        ]
      }
    }
  ]
}
```

Listing 3.2: Exemplarisch: Berechneter Weg in GeoJSON

4 Architektur

Die gewählte Architektur ist für mittlere bis große Projekte, bei denen eine klare Trennung der Verantwortlichkeiten, Wartbarkeit und Skalierbarkeit eine hohe Priorität haben.

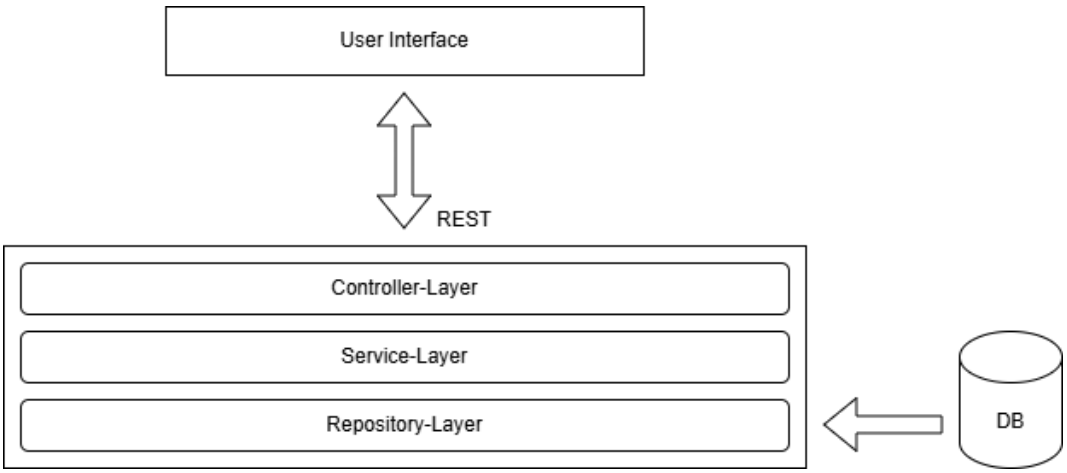

Abbildung 4.1: Architektur Übersicht

4.1 Datenbank

4.1.1 Container

Ein Container ist eine leichtgewichtige und eigenständige Einheit, die Software und alle ihre Abhängigkeiten enthält, um sicherzustellen, dass sie zuverlässig in jeder Umgebung ausgeführt werden kann. Container bieten eine isolierte Umgebung für Anwendungen, die vom Rest des Systems unabhängig sein sollen.

Container laufen also isoliert voneinander und vom Host-Betriebssystem. Sie haben eigene Prozesse, eigene Dateisysteme, Netzwerkschnittstellen und Ressourcen. Ein Container enthält alles, was eine Anwendung benötigt, einschließlich des Betriebssystems, Bibliotheken, Konfigurationen und andere Abhängigkeiten. Dadurch wird vermieden, dass es zu Konflikten mit anderen Anwendungen auf dem System kommt.

Aufgrund der Portabilität, ist es möglich, einen Docker-Container auf nahezu jedem System funktionsfähig auszuführen.

Mittels Skripten ist es möglich, innerhalb von wenigen Sekunden eine vollumfängliche Umgebung zu starten. Speziell für Entwicklungs- und Testumgebungen bringt das einen

enormen Zeitvorteil. Zusätzlich ist der gesamte Anwendungscode und seine Abhängigkeiten in einem Container gebündelt, sodass die identische Anwendung auf verschiedenen Systemen mit identischen Konfigurationen ausgeführt werden kann.

Zum Starten einer Datenbankinstanz mittels Docker, liegt ein `docker-compose.yml`-File im Dateiverzeichnis des Projekts. Das Skript hat folgenden Inhalt:

```
version: '3'
services:
  db:
    image: postgres:latest
    container_name: wegfinder-db
    environment:
      POSTGRES_USER: postgres
      POSTGRES_PASSWORD: postgres
      POSTGRES_DB: wegfinder
    volumes:
      - ./init:/docker-entrypoint-initdb.d
    ports:
      - "5432:5432"
```

Listing 4.1: docker-compose.yml

Gestartet werden kann der Container über eine Konsole:

```
docker-compose up -d
```

Listing 4.2: docker.cmd

Wichtig: Der Benutzername (POSTGRES_USER) und das Passwort (POSTGRES_PASSWORD) für die Datenbank sind für das Projekt einfach gehalten. Außerhalb einer Testumgebung sollte ein entsprechend sicheres Passwort gewählt werden. Weiterhin sollte der Upload in ein Repository oder auf ein öffentliches Speichermedium vermieden werden.

4.1.2 PostgreSQL

Der Einsatz einer PostgreSQL[1]-Datenbank als Datenbanklösung ergibt sich, weil PostgreSQL eine leistungsstarke, flexible und hochgradig erweiterbare relationale Open-Source-Datenbank ist. PostgreSQL ist vollständig Open-Source und wird unter der PostgreSQL-Lizenz bereitgestellt, die eine freie und uneingeschränkte Nutzung, Modifikation und Verteilung erlaubt, sowohl in privaten als auch kommerziellen Projekten. Das bedeutet, dass keine Lizenzgebühren oder versteckten Kosten anfallen.

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

4.1.3 JPA

JPA (Java Persistence API) ist eine Schnittstelle in Java, die es ermöglicht, auf einfache Weise mit Datenbanken zu arbeiten. Sie hilft dabei, Daten aus Java-Objekten in Datenbanktabellen zu speichern und umgekehrt.

Anstatt mit SQL-Befehlen, werden mit JPA Java-Objekte benutzt. JPA überträgt die Objekte in die Datenbank und die Ergebnisse aus der Datenbank werden in Java-Objekte gewandelt.

4.1.4 Code-First

Beim Code-First-Ansatz wird das Datenbankschema aus den Entitäten generiert, was den Entwicklungsprozess beschleunigt.

Die Entitäten können Validierungen und Constraints direkt auf den Feldern definieren, was erheblich zur Verbesserung der Datenintegrität beiträgt.

Berücksichtigt werden muss, dass Änderungen am Datenbankschema immer über den Code erfolgen müssen. Dies kann bei größeren Änderungen am Schema zu einem Overhead führen und bei komplexen oder stark optimierten Datenbankschemata kann der Code-First-Ansatz zu Konflikten in den generierten Tabellen führen.

Aufgrund der geringen Komplexität, ist der Code-First-Ansatz für dieses Projekt geeignet.

4.1.5 Node-Entity

Die Node-Entität bildet das Schema der Knoten in der Datenbank ab. Jede Node hat eine Id die aus den OSM-Daten übernommen wird, Längen- und Breitengrad und einen Verweis auf die Street-Tabelle zur Identifikation einer Referenz.

```
1  @Getter
2  @Setter
3  @Entity
4  @Table(name = "nodes")
5  public class Node implements Serializable {
6
7      @Id
8      @Column(name = "id", nullable = false)
9      private Long id;
10
11     @Column(name ="lon", nullable = false)
12     private Double longitude;
13
14     @Column(name = "lat", nullable = false)
15     private Double latitude;
```

```
16
17      @ManyToMany(fetch = FetchType.LAZY, mappedBy = "nodes")
18      private Set<Street> streets;
19  }
```

Listing 4.3: Node.java

4.1.6 Way-Entity

Mit der Way-Entität werden generell alle Objekte vom Typ `Way` gespeichert und verwaltet. Allerdings werden auf Basis in der Applikation integrierter Logik, wie der Prüfung des Vorhandensein von Adressdaten ein Gebäude-Flag gesetzt.

Neben einer Id, die aus dem OSM-Objekt übernommen wird, sind diverse mögliche Attribute abgebildet. Weiterhin ist in dem Objekt eine Many-To-Many Beziehung zu der Node-Tabelle implementiert, um die Knoten in den Straßen und die Knoten die die Gebäude definieren zu referenzieren.

4.1.7 Street-Entity

Die zentrale Entität des Street-Objekts ist im Grunde auf die nötigsten Informationen beschränkt und bildet die Beziehungen ab, die für eine effiziente Berechnung notwendig sind.

Neben der Identifikation über die Id, ist das Objekt als Gebäude oder Straße identifiziert und trägt die Straße und bei einem Gebäude auch eine Hausnummer.

Außerdem ist über die Kombination Straße und Nummer ein Index generiert, um die Suche etwas zu optimieren. Dieser ist nicht als ”unique” gekennzeichnet, da es bei einem größeren Datensatz durchaus möglich ist, dass in verschiedenen Orten die gleiche Straßen-Hausnummer-Kombination vorkommen kann.

Weiterhin hat jedes Street-Objekt eine Referenz auf die zugehörigen Knoten.

```java
@Getter
@Setter
@Entity
@Table(name = "streets", indexes = {
        @Index(
            name = "street_houseNumber_idx",
            columnList="street,houseNumber")
})
public class Street implements Serializable {

    @Id
    @Column(name = "id", nullable = false)
    private Long id;

    @Column(name = "isBuilding")
    private Boolean isBuilding;

    @Column(name = "street")
    private String street;

    @Column(name = "houseNumber")
    private String houseNumber;

    @ManyToMany(
        fetch = FetchType.EAGER,
        cascade = CascadeType.ALL)
    @JoinTable(
            name = "street_node_relation",
            joinColumns = @JoinColumn(
                name = "street_id",
                referencedColumnName = "id"),
            inverseJoinColumns = @JoinColumn(
                name = "node_id",
                referencedColumnName = "id"))
    private Set<Node> nodes;
}
```

Listing 4.4: Street.java

4.1.8 Entity-Relation

Im Grunde sind die Entitäten sehr ähnlich zu den DTOs aus Kapitel 3.5 implementiert. Der maßgebliche Unterschied bildet die Darstellung des Street-Modells in der Datenbank, in dem auf die Eltern-Kind-Beziehung verzichtet wird.

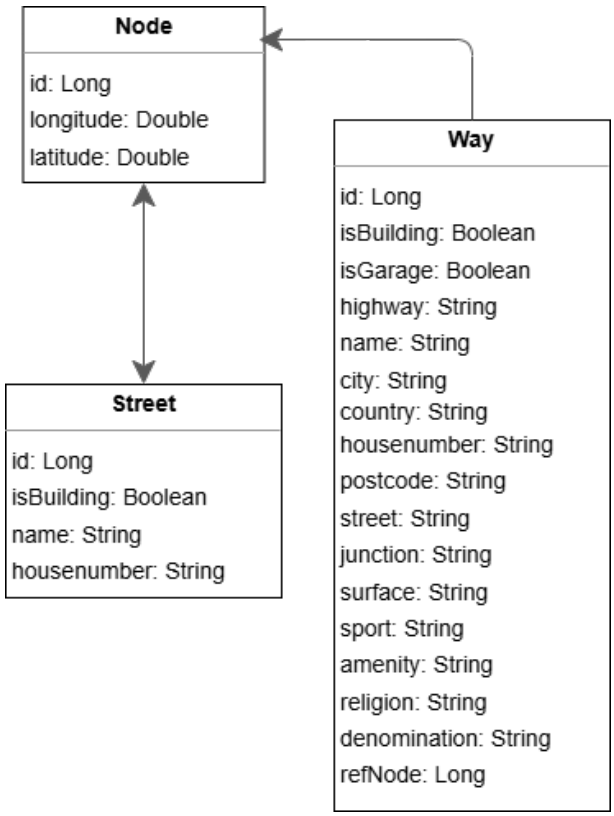

Abbildung 4.2: Entity Relation Diagramm

Der Grund für den Verzicht sind der Mehrwert an Performance beim Abruf der Daten. Die Datenbank muss so lediglich die reinen Daten halten und keine komplexen Beziehungen zwischen Objekten verwalten.

4.2 Backend

Die Architektur, die auf einem mehrschichtigen Spring-Webservice basiert, bietet eine klare Trennung der Verantwortlichkeiten und eine robuste Struktur für die Entwicklung von Webanwendungen.

Die Vor- und Nachteile dieser Architektur sind:

Vorteile:

1. **Trennung der Verantwortlichkeiten (Separation of Concerns):**
   Jede Schicht hat eine klar definierte Aufgabe (z. B. Controller verarbeitet Anfragen, Service führt Geschäftslogik aus, Repository interagiert mit der Datenbank). Dadurch wird der Code modular und einfacher zu warten.
2. **Wiederverwendbarkeit:**
   Durch die Schichtentrennung können Komponenten (z.B. der Service) unabhängig

wiederverwendet werden. Ein Service kann von verschiedenen Controllern oder Anwendungen genutzt werden.

3. **Testbarkeit:**
   Durch die Trennung in Schichten können einzelne Komponenten einfacher getestet werden. Unit-Tests lassen sich für Controller, Service und Repository-Schichten unabhängig voneinander erstellen.
4. **Skalierbarkeit:**
   Diese Architektur erleichtert das Hinzufügen neuer Funktionen. Neue Features können durch Hinzufügen neuer Services, Repositories oder Endpunkte integriert werden, ohne bestehende Komponenten zu verändern.
5. **Flexibilität:**
   Jede Schicht kann unabhängig weiterentwickelt und optimiert werden. Änderungen an der Datenbank (Repository-Schicht) oder am Service haben keinen direkten Einfluss auf andere Schichten.
6. **Wartbarkeit:**
   Die Struktur ist für Entwickler gut nachvollziehbar und erleichtert langfristig die Wartung und Erweiterung des Systems.

Nachteile:

1. **Komplexität:**
   Für einfache Anwendungen kann diese Architektur übertrieben sein. Der Aufwand für die Implementierung und Wartung von mehreren Schichten kann die Entwicklung von kleineren Projekten unnötig kompliziert machen.
2. **Leistungseinbußen:**
   Jede Schicht fügt zusätzlichen Overhead hinzu. Daten werden von einer Schicht zur anderen weitergegeben, was zu einer geringen Verzögerung führen kann, insbesondere bei komplexeren Anfragen.
3. **Erhöhter Entwicklungsaufwand:**
   Für jede Schicht muss zusätzlicher Code geschrieben werden, auch wenn die Anforderungen an das Projekt einfach sind. Das bedeutet mehr Boilerplate-Code und potenziell längere Entwicklungszeiten.

4.2.1 Repository-Layer

Die Repository-Schicht ist die Abstraktion von Datenbankoperationen. Sie bietet die Möglichkeit, einen Datenbankzugriff an einer zentralen Stelle zu verwalten, was mögliche Code-Duplikationen vermeidet.

Die Verwendung von `Spring Data JPA` vereinfacht die Implementierung, da die meisten Standardmethoden zum Finden, Speichern und Löschen bereits integriert sind. Dazu wird jedes Interface um das JpaRepository erweitert.

Der Zugriff auf die Operationen für das Street-Modell ist um eine Methode

`findByNameIgnoreCase()` erweitert. Das JPA-Repository erkennt über den Methodennamen den Zugriff auf eine in der Entität angegebene Variable `street` und erlaubt über diesen einfachen Weg, das direkte Suchen in dem entsprechenden Datenbankfeld. Eine weitere Besonderheit im Methodennamen, ist der Zusatz *IgnoreCase*. Dadurch wird automatisch nicht die Groß- und Kleinschreibung validiert.

```
1 @Repository
2 public interface IStreetRepository extends JpaRepository<Street, Long> {
3     List<Street> findByNameIgnoreCase(String name);
4 }
```

Listing 4.5: IStreetRepository.java

4.2.2 Controller-Layer

Der Controller bildet die Schnittstelle nach Außen. Er verarbeitet die HTTP-Anfragen und liefert Antworten. Die Schnittstelle ist klar definiert und für die Interaktion mit dem Webservice verantwortlich.

Der Controller enthält keine Geschäftslogik, sondern delegiert diese an einen entsprechenden Service. Dadurch bleibt der Controller schlank und fokussiert sich lediglich auf die Verarbeitung von Anfragen. Ansonsten könnte der Controller überladen und komplex werden.

Endpunkte können schnell und einfach hinzugefügt, bearbeitet und entfernt werden, allerdings kann ohne ordentliche Fehlerbehandlung, der Controller schwierig zu debuggen sein. Jede HTTP-Anfrage kann unter Umständen spezifische Fehlerbehandlungen erfordern.

Die Controllerintegration ist am Beispiel des `PathfindingController` beschrieben:

```
1  @Operation(summary = "Compute␣way␣and␣respond␣with␣GeoJson-Object.")
2  @ApiResponse(
3      responseCode = "200",
4      description = "GeoJson␣Object␣successfully␣created.")
5  @ApiResponse(
6      responseCode = "404",
7      description = "Way␣not␣computable.")
8  @GetMapping(produces = MediaType.APPLICATION_JSON_VALUE)
9  public ResponseEntity<GeoJsonObject> getComputedWay(
10         @RequestParam(name = "stStr") String startStreet,
11         @RequestParam(name = "stNo") String startNumber,
12         @RequestParam(name = "tgStr") String targetStreet,
13         @RequestParam(name = "tgNo") String targetNumber)
```

```
14          throws StreetNotFoundException,
15          NodeNotFoundException,
16          WayNotComputableException {
17
18      // streets and graph will be initialized and stored in cache
19      List<StreetDTO> streets = streetService.findAllStreets();
20
21      // look up for streets from interface
22      List<StreetDTO> startStreets =
23          streets
24          .stream()
25          .filter(st -> st.getStreet().equals(startStreet))
26          .toList();
27      List<StreetDTO> targetStreets =
28          streets
29          .stream()
30          .filter(st -> st.getStreet().equals(targetStreet))
31          .toList();
32
33      if (startStreets.isEmpty()) {
34          String errMsg =
35              String.format("No␣match␣for␣start␣street␣with␣name:␣%s.",
36                  startStreet);
37          log.error(errMsg);
38          throw new StreetNotFoundException(errMsg);
39      }
40      if (targetStreets.isEmpty()) {
41          String errMsg =
42              String.format("No␣match␣for␣target␣street␣with␣name:␣%s.",
43                  targetStreet);
44          log.error(errMsg);
45          throw new StreetNotFoundException(errMsg);
46      }
47
48      Graph<NodeDTO> graph = aStarAlgorithm.prepareGraph(streets);
49
50      // find main nodes
51      NodeDTO startNode =
52          streetService.getNodeOfStreet(startNumber, startStreets);
53      NodeDTO targetNode =
54          streetService.getNodeOfStreet(targetNumber, targetStreets);
55
56      List<NodeDTO> route = aStarAlgorithm.findRoute(
57          graph, startNode.getId(), targetNode.getId());
58
```

```
59      GeoJsonObject geoJsonObject =
60          geoJsonService.createGeoJsonObject(
61              startNode, targetNode, route);
62
63      return ResponseEntity.ok(geoJsonObject);
64  }
```

Listing 4.6: Auszug - PathfindingController.java

Der Endpunkt lautet `/pathfinding?stStr={}&stNo={}&tgStr={}&tgNo={}`.

Bei einer Anfrage gegen den REST-Endpunkt soll nach der Start- und Zielstraße gesucht und das mögliche Ergebnis in ein DTO konvertiert werden.
Anschließend soll jeweils die Referenz-Node gesucht werden, um zwischen diesen Punkten einen möglichen Weg zu finden und zu berechnen.

Zum Erstellen des Graphen werden dann für jeden Request alle Node-Objekte und Street-Objekte aus der Datenbank genommen.

Das hat sich als nicht performant bewiesen, da für jede Anfrage nach einem Weg, alle Streets, alle Nodes abgefragt werden und alle Streets für die Startadresse und alle Streets für die Zieladresse durchsucht werden. Das wird dahingehend angepasst, dass alle Node- und Street-Objekte beim Starten des WebService in den Zwischenspeicher (Cache) der Anwendung geladen werden, sodass das Suchen und Berechnen zur Laufzeit schneller ist und die Datenbankzugriffe nur zum Start stattfinden. Dementsprechend werden auch die Start- und Zieladresse aus den vorgeladenen Objekten die im Cache mitgeführt werden gesucht.

Lediglich zum Start der Applikation findet ein Datenbankzugriff zum Füllen des Zwischenspeichers statt. Ab dann werden die benötigten Daten zur Laufzeit aus dem Cache genommen.

Diese Vorgehensweise hat den Nachteil, dass bei Veränderungen der Quelldaten, der Zwischenspeicher entsprechend verändert werden muss.

4.2.3 Service-Layer

Der Service-Layer enthält für alle notwendigen Dienste die gesamte Geschäftslogik. Das macht es einfacher Änderungen in der Logik vorzunehmen, ohne einen Controller oder ein Repository zu verändern.

Die Service-Schicht kann von mehreren Controllern verwendet werden und trennt klar die Logik von der HTTP- und Datenbankschicht.

Ein wichtiges Merkmal ist, dass bei vielen Services die Gefahr von zirkulären Abhän-

gigkeiten entstehen kann, die schwer zu handhaben sein können.

```
@Slf4j
@Service
public class StreetService {

    @PersistenceContext
    private EntityManager entityManager;

    @Autowired
    private ModelMapper modelMapper;

    @Autowired
    private IStreetRepository streetRepository;

    @Transactional
    @Cacheable(value = "streets", sync = true)
    public void saveAllStreets(List<StreetDTO> streetsDTOs) {
        List<Street> streets = streetsDTOs
                .stream()
                .map(this::convertToStreetEntity)
                .toList();
        streetRepository.saveAll(streets);
    }

    @Cacheable(value = "streets", sync = true)
    public List<StreetDTO> findAllStreets() {
        List<Street> streets = streetRepository.findAll();
        List<StreetDTO> streetDTOs = streets
            .stream()
            .map(this::convertToStreetDto)
            .toList();

        return updateStreetRelations(streetDTOs);
    }

    @Cacheable(value = "streets", sync = true)
    public List<StreetDTO> findListOfAllStreets() {
        List<StreetDTO> streetDTOs = findAllStreets();
        return streetDTOs
            .stream()
            .filter(st -> !st.getIsBuilding())
            .filter(st -> !st.getChildren().isEmpty())
            .toList();
```

```
43      }
44
45      @Cacheable(value = "streets", sync = true)
46      public List<StreetDTO> findListOfAllObjects() {
47          List<StreetDTO> streetDTOs = findAllStreets();
48          return streetDTOs
49              .stream()
50              .filter(StreetDTO::getIsBuilding)
51              .toList();
52      }
53
54      @Cacheable(value = "streets", key = "#id", sync = true)
55      public Optional<StreetDTO> findStreetById(Long id)
56           throws StreetNotFoundException {
57          Optional<Street> street = streetRepository.findById(id);
58          if (street.isEmpty()) {
59              String errMsg =
60                  String.format("No␣match␣for␣street␣with␣id:␣%s.", id);
61              log.error(errMsg);
62              throw new StreetNotFoundException(errMsg);
63          }
64
65          return street.map(this::convertToStreetDto);
66      }
67
68      @Cacheable(value = "streets", key = "#name", sync = true)
69      public List<StreetDTO> findByStreet(String name)
70          throws StreetNotFoundException {
71          List<Street> streets = streetRepository.findByNameIgnoreCase(name);
72          if (streets.isEmpty()) {
73              String errMsg =
74                  String.format("No␣match␣for␣street:␣%s.", name);
75              log.error(errMsg);
76              throw new StreetNotFoundException(errMsg);
77          }
78
79          return streets
80              .stream()
81              .map(this::convertToStreetDto)
82              .toList();
83      }
84
85      private StreetDTO convertToStreetDto(Street street) {
86          return modelMapper.map(street, StreetDTO.class);
87      }
```

```
88
89      private Street convertToStreetEntity(StreetDTO streetDTO) {
90          return modelMapper.map(streetDTO, Street.class);
91      }
92
93      private List<StreetDTO> updateStreetRelations(List<StreetDTO> streets) {
94          List<StreetDTO> streetDTOs = new ArrayList<>();
95          for (StreetDTO streetDTO : streets) {
96              if (streetDTO.getIsBuilding()) {
97                  StreetDTO parent = streetDTOs
98                      .stream()
99                      .filter(st ->
100                         st.getName()
101                         .equals(streetDTO.getName()) &&
102                         !st.getIsBuilding())
103                     .findFirst()
104                     .get();
105
106                 streetDTO.setParent(parent);
107                 streetDTO.setChildren(null);
108                 streetDTO.getParent().getChildren().add(streetDTO);
109             } else {
110                 streetDTO.setParent(null);
111                 if (streetDTO.getChildren() == null) {
112                     streetDTO.setChildren(new ArrayList<>());
113                 }
114             }
115
116             streetDTOs.add(streetDTO);
117         }
118
119         return streetDTOs;
120     }
121 }
```

Listing 4.7: StreetService.java

4.2.4 Weitere Services

Zum Verarbeiten verschiedenster Funktionen ist weitere Geschäftslogik notwendig, welche im Service-Layer als eigene ”Services” implementiert sind. Nachfolgend die Aufrufstruktur:

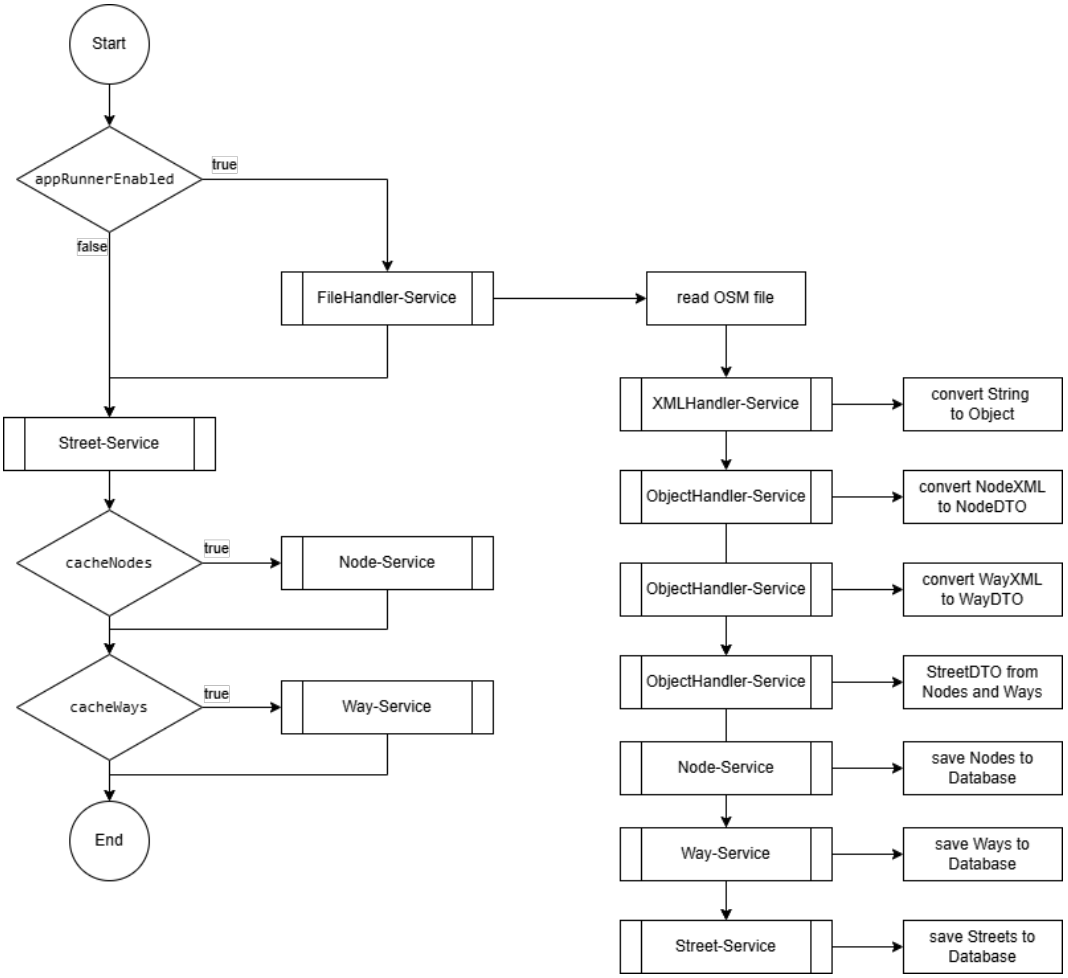

Abbildung 4.3: Service-Aufruf-Architektur

Das Deklarieren mit der Service-Annotation sorgt dafür, dass das Spring-Framework die Instanzen der Klassen selbst verwaltet und bei Bedarf erstellt oder verwirft. Das Erstellen einer eigenen Instanz ist somit nicht notwendig und wird mittels Dependency-Injection implementiert.

Im Grunde gibt es zwei Möglichkeiten abhängige Instanzen einzubringen: Über die `@Autowired`-Annotation, die ebenfalls von Spring Boot verwaltet wird, oder über einen Aufruf im Konstruktor. Wegen der Konsistenz der Verwendung von Beans und der Lesbarkeit sind im Projekt Abhängigkeiten über Annotationen eingebracht.

```
1  public class ExampleController {
2      \\Dependency Injection mittels Bean
3      @Autowired
4      private Klasse klasse;
5
6      \\Dependency-Injection mittels Konstruktor
7      private final Klasse klasse;
8      public ExampleController(Klasse klasse) {
9          this.klasse = klasse;
10     }
11 }
```

Listing 4.8: Dependency-Injection

4.2.4.1 FileHandler-Service

Der FileHandler-Service repräsentiert eine Hilfsklasse zur Verarbeitung der osm-Dateien. Die Klasse besteht im Grunde aus zwei Methoden, die den Inhalt einer osm-Datei einlesen und zu einem String verarbeiten und einer Methode, die einen String versucht in einen OSM-Kontext zu konvertieren und die Daten nach einem Mapping in eine angeschlossene Datenbank speichert.

4.2.4.2 GeoJson-Service

Der GeoJSON-Service ist eine einfache Klasse, die das Erstellen eines GeoJSON-Objekts kapselt, nach dem ein berechneter Weg übergeben wurde. Das Format des Objekts folgt dem Standard, dass in Abschnitt 2.3.2 beschrieben ist.

4.2.4.3 Objecthandler-Service

Der ObjectHandler-Service bildet eine Sammlung von Methoden die maßgeblich für die Aufbereitung der benutzten Objekte verantwortlich ist.

Die Methode `prepareStreetsFromWaysAndNodes()` bereitet die Street-Objekte aus den vorhanden Wegen, Gebäuden und Knoten vor. Durch Eingabe aller Way-Objekte werden diese getrennt in Gebäude und Straßen, ihren Eltern und Kind-Eigenschaften zugeführt und für die Speicherung vorbereitet. Den jeweiligen Straßen und Gebäuden werden dann zusätzlich ihre Knoten zugewiesen die in Relation zueinander stehen.

Neben einigen privaten Hilfsmethoden die lediglich zur besseren Lesbarkeit des Codes angelegt sind, existieren Konverter, die aus den XML-Objekten DTOs erstellen. Auch eine Methode die die doppelten Nodes eines Gebäudes findet, dass zur Identifikation bei der Suche verwendet werden kann.

Eine wichtige zusätzliche Methode ist die `wayAlreadyExists()`-Funktion und die

`nodeAlreadyExists`-Funktion. Es ist nicht auszuschließen, dass die Datenbasis identische Objekte mehrfach hält. Um doppelte Keys, also eine ”Duplicated constraint violation”, zu vermeiden, wird beim Vorbereiten bzw. dem Unterteilen der Objekte diese Vorabprüfung durchgeführt.

4.2.4.4 XmlHandler-Service

Der XMLHandler-Service kapselt das Marshalling eines Strings, der einen String im XML-Standard führt, in definierte XML Objekte.

Zu beachten gilt, dass die Struktur der XML-Datei in einer Objektstruktur mit all ihren Abhängigkeiten abgebildet werden muss.

4.2.5 Application Cache

Ein globaler Cache einer Anwendung bringt eine signifikante Leistungssteigerung, besonders wenn es sich um oft genutzte, unveränderliche Daten handelt. Die Vorteile wie schneller Zugriff und Entlastung von Datenquellen sind besonders in datenintensiven Anwendungen sinnvoll. Allerdings müssen dabei Speicherverbrauch, Cache-Invaliderung und mögliche Konsistenzprobleme beachtet werden, um die Nachteile zu minimieren.

Vorteile:

1. Wenn Daten im Cache vorgehalten werden, können Komponenten schneller auf sie zugreifen, anstelle bei jedem Request eine Datenbank- oder API-Abfrage auszuführen.
2. Durch Caching entfallen viele Abfragen auf externe Datenquellen wie Datenbanken, was die Last reduziert.
3. Ein zentraler Cache ist in der gesamten Anwendung und in allen Komponenten verfügbar, was die Wiederverwendung von Daten vereinfacht.
4. Die Daten werden beim Start oder beim ersten Aufruf eines Repository-Zugriffs vorgeladen, sind also direkt bei der Inbetriebnahme oder Erstausführung verfügbar, ohne auf Abrufe oder Aktualisierungen zu warten.

Nachteile:

1. Das Vorhalten von großen Datenmengen im Cache kann viel Speicher benötigen, was dann zu Speicherengpässen führt.
2. Daten im Cache könnten veralten. Es ist wichtig, Mechanismen zur Cache-Invaliderung zu implementieren, um sicherzustellen, dass veraltete Daten nicht verwendet werden, sofern notwendig.
3. Wenn sich die Daten ändern, muss der Cache entsprechend aktualisiert oder invalidiert werden, was zu zusätzlicher Komplexität führt.

4. Werden Daten beim Start vorgeladen, kann die Startzeit der Anwendung verlängert werden, insbesondere wenn viele oder komplexe Daten geladen werden müssen.

5. In verteilten Systemen ist es schwierig, die Konsistenz zwischen verschiedenen Knoten sicherzustellen, die jeweils ihren eigenen Cache haben.

Es gibt eine einfache Möglichkeit direkt aus dem Spring Boot-Framework eine Cacheverwaltung[2] mittels Beans zu implementieren und diese zu nutzen. Dazu muss lediglich `@EnableCache` annotiert werden und in der Cache-Konfiguration die zu verwendenden Speicher, sowie der verwendete CacheManager definiert werden.

Standardmäßig verwaltet die `Cachable`-Annotation den Cache auf Methodenebene. Um den Cache global zur Verfügung zu stellen, muss dies konfiguriert werden.

Als Entwickler muss sich nicht direkt um den Cache gekümmert werden, da Spring das Caching automatisch handhabt. Ein weiterer nennenswerter Vorteil ist, dass der Cache durch die Annotation von der Geschäftslogik getrennt ist, was den Code sauber und wartbar hält.

4.2.6 Data Transfer Objects

Die Trennung von DTO und Entitäten, also die verschiedenen Verantwortlichkeiten klar zu trennen, bringt sowohl Vor- als auch Nachteile mit sich.

DTOs werden verwendet, um Daten zwischen den Schichten (Controller, Service), als auch nach außen zu übertragen und stellt sicher, dass nur die relevanten Informationen an den Client gesendet werden. Allerdings ist der Implementierungsaufand höher. Nicht nur, dass die Objekte quasi doppelt angelegt werden müssen, weiterhin müssen Mapper integriert werden, die die Code-Komplexität erhöhen.

Entitäten repräsentieren die Datenbankstruktur und sind eng mit der Datenbank verbunden. Diese Kopplung bleibt verborgen, wenn nur DTOs an die oberen Schichten weitergegeben werden. Bei Änderung des Datenmodells ist der Pflegeaufwand dagegen zusätzlich hoch.

Durch die Verwendung von DTOs wird verhindert, dass sensible Felder oder intern genutzte Datenstrukturen (z.B. IDs, Passwörter oder technische Metadaten) direkt an den Client weitergegeben werden. DTOs helfen außerdem Angriffsflächen zu verringern, da exakt definiert wird, welche Daten nach außen sichtbar gemacht werden. Aber kontextspezifische Informationen dürfen nicht verloren gehen und sollten in der API richtig abgebildet werden.

Durch die klare Trennung sind Änderungen an der Datenbank oder am Datenbankschema

[2] `https://docs.spring.io/spring-boot/reference/io/caching.html`, abgerufen 11.10.2024

leicht durchführbar, ohne dass es direkte Auswirkungen auf die API hat. So lange die DTO-Schnittstelle unverändert bleibt, bleibt auch die API kompatibel.

DTOs bieten die Möglichkeit, separate Validierungsregeln für Daten zu erstellen, die der Client sendet, ohne die Entitätslogik zu beeinträchtigen. Die Validierungsregeln sollten Konsistent bleiben, sonst entstehen möglicherweise Bugs.

4.2.7 Fehlerbehandlung

In der Appliaktion ist der `GlobalControllerExceptionHandler` ein Mechanismus, um globale Fehlerbehandlung für alle Controller in der Spring-Anwendung bereitzustellen. Damit kann zentral festgelegt werden, wie bestimmte Arten von Fehlern oder Ausnahmen behandelt werden sollen, anstatt in jedem Controller jeden Fehler separat abzufangen.

Die Annotation `@RestControllerAdvice` macht die Klasse zu einem globalen Fehlerbehandler, der für alle Controller in der Anwendung gilt. Damit werden alle Controller überwacht und reagieren auf bestimmte Ausnahmen die in dem Handler definiert sind.

```
1  @RestControllerAdvice
2  public class GlobalControllerExceptionHandler {
3
4      @ExceptionHandler(WayNotComputableException.class)
5      public ResponseEntity<ApiError> handleNotComputable(
6          WayNotComputableException ex) {
7          ApiError apiError =
8              new ApiError(
9                  HttpStatus.NOT_FOUND,
10                 "Try␣other␣addresses." ,
11                 ex.getMessage());
12         return ResponseEntity.status(apiError.getStatus()).body(apiError);
13     }
14
15     @ExceptionHandler(WayNotFoundException.class)
16     public ResponseEntity<ApiError> handleWayNotFound(
17         WayNotFoundException ex) {
18         ApiError apiError =
19             new ApiError(
20                 HttpStatus.NOT_FOUND,
21                 "Use␣a␣known␣way." ,
22                 ex.getMessage());
23         return ResponseEntity.status(apiError.getStatus()).body(apiError);
24     }
25
26     @ExceptionHandler(NodeNotFoundException.class)
```

```
27      public ResponseEntity<ApiError> handleNodeNotFound(
28          NodeNotFoundException ex) {
29          ApiError apiError =
30              new ApiError(
31                  HttpStatus.NOT_FOUND,
32                  "Use a known node." ,
33                  ex.getMessage());
34          return ResponseEntity.status(apiError.getStatus()).body(apiError);
35      }
36
37      @ExceptionHandler(StreetNotFoundException.class)
38      public ResponseEntity<ApiError> handleStreetNotFound(
39          StreetNotFoundException ex) {
40          ApiError apiError =
41              new ApiError(
42                  HttpStatus.NOT_FOUND,
43                  "Use a known street." ,
44                  ex.getMessage());
45          return ResponseEntity.status(apiError.getStatus()).body(apiError);
46      }
47  }
```

Listing 4.9: GlobalControllerExceptionHandler.java

4.2.8 Logging

Das Logging hilft zu verstehen, an welchem Punkt sich die Applikation während des Prozesses befindet. Die konfigurierten Ausgaben sind eine Konsolen- und eine Dateiausgabe.

Die Struktur des Loggings ist:
INFO: Alle Informationen die für den ”normalen” Betrieb wichtig sind.
DEBUG: Erweiterte Informationen, die wichtig sind, wenn die Prozesse genau untersucht werden müssen.
TRACE: Wichtige Informationen, um den Datengehalt während des Ablaufs nach zu vollziehen.
ERROR: Fehler, die zum Abbruch der Applikation führen können, oder die Ausgabe eines Ergebnisses verhinden.

Vor jedem Aufruf einer Log-Nachricht, ist im Code eine if-Abfrage zur Prüfung, ob das benutzte Log-Level aktiviert ist. Damit wird vermieden, das unnötige Strings verarbeitet werden. Auch wenn das Log-Level nicht benutzt wird, wird das Objekt berechnet und aufbereitet.

Gerade bei komplexeren Berechnung oder einer Ausgabe von Informationen handelt es sich nicht um einfache Stringverarbeitungen, sondern beinhalten Erzeugungen von

komplexen Objekten.

```
1 if (log.isDebugEnabled()) {
2     log.debug("Open␣Set␣contains:␣{}",
3         openSet
4         .stream()
5         .map(RouteNode::getCurrent)
6         .collect(Collectors.toSet()));
7 }
```

Listing 4.10: Beispiel eines komplexen Logs

Damit wird nicht nur die Performance verbessert, indem übermäßig viele Log-Nachrichten erstellt werden, auch der Speicherverbrauch wird reduziert. Zusammengefasst: Der Ablauf wird effizienter.

4.3 Schnittstellen

4.3.1 Dateneingabe

Das Eingeben der Daten erfolgt über den FileHandler-Service.

Zum Aktivieren der Dateneingabe über die Datei zum Applikationsstart, muss in den `application.properties` die Eigenschaft `app.runner.enabled=false` auf true gestellt werden. Die PathfindingApplication-Klasse liest zum Start den Wert der Eigenschaft ein und führt den FileHandler bei einem ”wahren” Ergebnis aus.

Ist die Eigenschaft ”falsch”, wird das Einlesen einer Datei übersprungen. In beiden Fällen wird lesend auf die Datenbank zugegriffen und die Daten in den Applikationscache geschrieben.

4.3.2 REST-API

Alle Endpunkte können über die integrierte Swagger-UI grafisch getestet werden. Dazu muss lediglich im Browser `http://localhost:8081/swagger-ui.html` aufgerufen werden.

Methode	Endpunkt	Beschreibung
GET	/pathfinding?stStr=&stNo=&tgStr=&tgNo=	Berechnet einen Weg zwischen zwei Adressen
GET	/nodes/all	Gibt alle Nodes zurück
GET	/nodes/{id}	Gibt Node mit einer Id zurück
GET	/nodes/add	Fügt neue Node hinzu
GET	/streets/all	Gibt alle Streets zurück
GET	/streets/list	Gibt die verfügbaren Streets zurück
GET	/streets/{id}	Gibt Street mit einer Id zurück
GET	/streets/add	Fügt neue Street hinzu
GET	/streets?name=&number=	Sucht Straße nach Adresse number ist optional
GET	/ways/all	Gibt alle Ways zurück
GET	/ways/{id}	Gibt Way mit einer Id zurück
GET	/ways/add	Fügt neue Way hinzu

Tabelle 4.1: Endpunkt-Definition

Die CORS-Policy ist nur für den `pathfinding`-Endpunkt definiert.

4.4 User-Interface

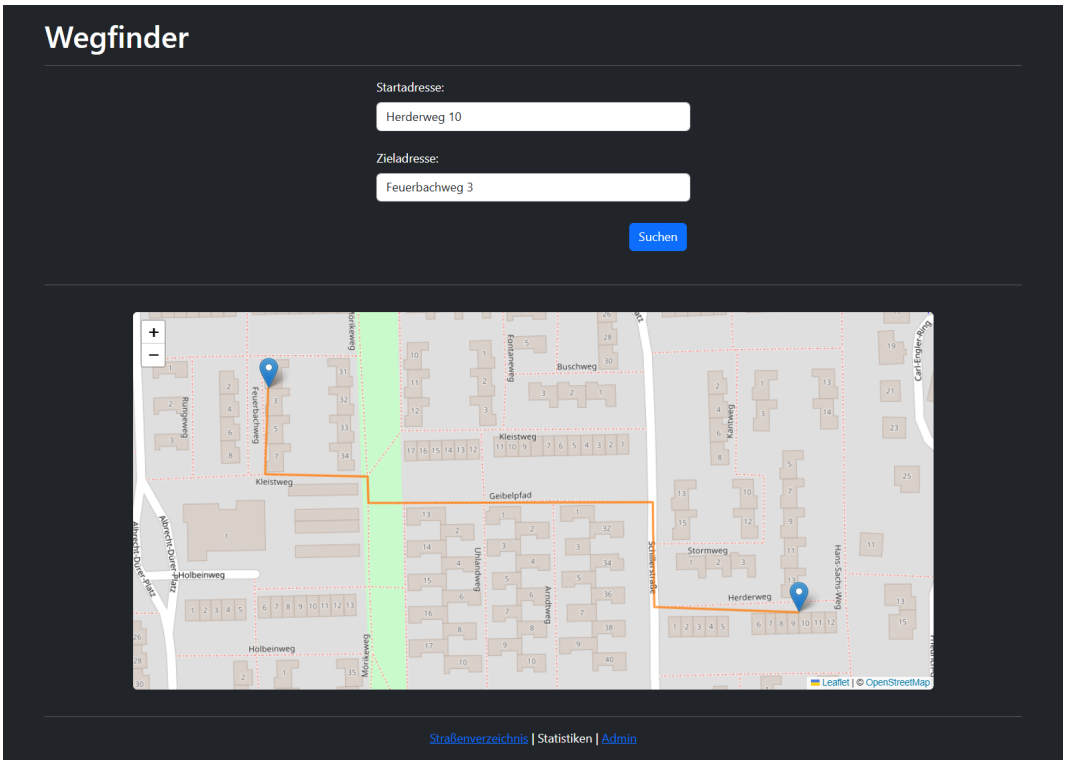

Abbildung 4.4: UI-Overview

Das User-Interface ist einfach gehalten. In den beiden oberen Eingabefeldern wird die Start- und die Zielstraße mit Hausnummer eingegeben und durch Drücken auf den Suchen-Button, wird eine Anfrage aufbereitet und an das Backend gesendet.

Werden beide Adressen gefunden, kommt über diese Schnittstelle ein berechneter Weg zurück, der von der dargestellten Karte angezeigt wird.

Im unteren Teil des User-Interface befindet sich ein Link zu einem Straßenverzeichnis, bei dem durch klicken eine Liste aller in der Datenbank befindlichen Straßen mit der Anzahl der ihnen untergeordneten Gebäude aufgelistet werden.

Durch drücken auf ”Admin” öffnet sich eine Seite, bei der ein nicht umgesetztes Feature integriert ist, mit dem eine osm-Datei hochgeladen werden könnte und eine Möglichkeit zur Anzeige aller in der Datenbank befindlichen Objekte abzurufen. Dies demonstriert vor allem den Performancegewinn durch das Caching.

4.4.1 API-Zugriff

Der Schnittstellenzugriff erfolgt über eine JavaScript-Methode die nach einer Adressvalidierung aufgerufen wird.

Im Grunde wird ein Request erstellt, der gegen den Pathfinding-Endpunkt anfragt. Allerdings wird beim Erstellen der Anfrage eine Bedingung implementiert, sodass das Auflösen des möglichen Ergebnisses nur dann zum Tragen kommt, wenn der Rückgabestatus OK ist.

Dabei wird das zurückgeführte GeoJSON-Objekt in einen JSON-Parser übergeben, der den Weg an die Karte im Frontend übergibt.

Im Falle eines Fehlers oder dem nicht Erreichen des Backends, wird in der Konsole des Browsers eine entsprechende Meldung ausgegeben.

```
async function fetchGeoJsonData(start, startNo, target, targetNo) {
    try {
        let url = `${globalAddress}/pathfinding?
        stStr=${start}
        &stNo=${startNo}
        &tgStr=${target}
        &tgNo=${targetNo}`;

        const response = await fetch(url);
        if (response.status === 0) {
            printErrorOnUI("API nicht erreichbar...");
        } else if (response.status === 404 || response.status === 500) {
            let err = await response.json();
            printErrorOnUI(err.error);
        }

        const geoJsonData = await response.json();
        processGeoJson(geoJsonData);

        const geoObj = geoJsonData.features;
        computeHeuristic(geoObj, start, startNo, target, targetNo);

    } catch (error) {
        console.error('Error on calling GeoJSON-data: ', error);
    }
}
```

Listing 4.11: API-Anfrage von UI

Hinweis: im Backend sind exemplarisch nur die Fehler 404 (Not Found) und 500 (Bad Request) behandelt. Daher wird im Frontend auch ausschließlich auf diese Fehler abgeprüft.

5 Lessons Learned

5.1 CORS-Policy

CORS (Cross-Origin Resource Sharing) ist eine Sicherheitsfunktion, die von Webbrowsern verwendet wird, um Webinhalte zu kontrollieren, die von anderen Domains als ihrer eigenen abrufen dürfen. Normalerweise dürfen Webseiten nur Daten von der gleichen Domain anfragen, von der sie geladen wurden. CORS erweitert diese Regel und erlaubt es, unter bestimmten Bedingungen Daten von einer anderen Domain abzurufen.

Im verwendeten Framework der Applikation wird ein Backend erstellt, das von einem Frontend angesprochen wird. Dadurch, das diese beiden Teile auf unterschiedlichen Domains oder Ports laufen, führt dies zu Problemen mit der CORS-Policy des Browsers. Damit das Frontend trotzdem auf die Daten über die Schnittstelle zugreifen darf, muss der Spring Boot-Server explizit erlauben, dass Anfragen zugelassen werden.

Die `@CrossOrigin`-Annotation in Spring Boot wird genau dafür verwendet, diese Anfragen zu konfigurieren. Die Konfiguration definiert im Backend, von welchen Domains Anfragen erlaubt sind und welche HTTP-Methoden genutzt werden dürfen.

Anstatt die `@CrossOrigin`-Annotation für jeden Controller einzeln hinzuzufügen, kann in Spring Boot auch eine globale Konfiguration erstellt werden, indem die `@CorsConfiguration` als Bean in der Konfigurationsklasse definiert wird. Diese globale Bean regelt dann die CORS-Einstellungen für die gesamte Anwendung. Da aber nur die Pfadberechnung für eine UI erreichbar sein soll, ist das genannte Bean lediglich im Path-Controller annotiert.

5.2 Datenbankzugriffe

Ein N+1-Abfrageproblem tritt auf, wenn JPA für eine Sammlung von Objekten (z.B. 2000 Objekte) nicht nur eine Abfrage für die Hauptdaten ausführt, sondern auch für jedes einzelne Objekt eine zusätzliche Abfrage für seine abhängigen Entitäten. Das kann die Anzahl der Datenbankabfragen drastisch erhöhen und die Performance erheblich beeinträchtigen.

Beispiel:
Wird eine Liste von 2000 Objekten vom Typ Street geladen, und jedes Objekt eine Beziehung zu einer Node oder einem Child oder Parent-Objekt hat, wird JPA eine Abfrage für das Street-Objekt und dann je eine Abfrage pro Node und pro Beziehungselement durchführen.

Im ersten Entwurf, hatte das Street-Entity-Model wenn es ein Parent-Objekt ist, eine

Anzahl an Child-Elementen (Gebäude) und die referenzierten Nodes. Eine Abfrage bezog sich daher immer zuerst gegen das Eltern-Element, dann die Kind-Elemente, die zugehörigen Nodes, für jedes Kind-Element und die zugehörigen Nodes und damit wieder gegen das Eltern-Element.

Die Verschachtelung ist so immens, dass bei einer Suche nach einem Straßenzug mit etwa 10 Gebäuden, die Response zwischen 15 und 20 Sekunden dauert. Das Auflösen des Modells bringt einen Gewinn auf etwa ein bis zwei Sekunden bei nicht gecachten Daten.

5.2.1 Probleme bei der Speicherstrategie

Das initiale Speichern aller Daten musste aufgrund des zuvor gewählten Modells in zwei Schritten erfolgen.
Alle Objekte werden beim mappen auf die Street-Entität zwar genauso aufgebaut, wie vom Modell vorgesehen, allerdings ist die Speicherreihenfolge unterschiedlich. Wird beispielsweise ein Kind-Objekt gespeichert, deren Elternobjekt noch nicht existiert, wird eben dieses Objekt automatisch angelegt und im Kontext verwaltet.

An einer Stelle im Ablauf wird der Entity-Manager versuchen dieses Elternobjekt zu speichern, weil es während der Iteration aufgerufen wird. Dann ist dieses Objekt bereits aus einer vorangegangenen Operation im Kontext. An dieser Stelle bemerkt das Framework ein bereits vorhandenes Objekt und verlässt die Transaktion mit einem Fehler.

Um diesen Effekt zu vermeiden, wird nach den Kind-Elementen (Gebäude) mittels Stream-API gefiltert und diese werden ohne Eltern-Beziehung gespeichert. Der Kontext wird nach jedem Element bereinigt.

Nachdem alle Gebäude in dem Kontext hinterlegt wurden, werden alle Eltern-Objekte gefiltert und für jedes Eltern-Element ein eigenes `merge()` aufgerufen. Dadurch das die Gebäude alle ”detached” sind, wird lediglich das Element angelegt und die Beziehung bidirektional aufgebaut.

Eine ”detached-Entity” ist eine Entität, die zuvor im Persistence Context verwaltet wurde, aber jetzt vom Persistence Context getrennt wurde. Das bedeutet, dass Hibernate oder JPA die Entität nicht mehr überwacht und Änderungen an der Entität nicht automatisch in die Datenbank schreibt. Sie bleiben also nur im Arbeitsspeicher vorhanden, aber Hibernate verfolgt sie nicht mehr. Durch die Methode `merge()` kehrt der Zustand in ”managed” zurück und wird mit der Datenbank synchronisiert. Ansonsten müsste die Entität erneut aus der Datenbank geladen werden.

Diese Vorgehensweise ist nur sinnvoll, weil die Entitäten direkt weiterverarbeitet werden.

5.3 Dateneingabe

Die Integration der Daten kann über einen Interpreter der OverpassAPI erfolgen. Dadurch können die infrage kommenden Daten direkt an der Schnittstelle angefragt werden. Die Dokumentation ist unter `https://wiki.openstreetmap.org/wiki/Overpass_API` zu finden.

Um die Daten und die Zusammenhänge zu verstehen und vor allem um unabhängig von einer fremden Schnittstelle arbeiten zu können, werden die Daten einmal über eine exportierte Datei in die angeschlossene Datenbank der Applikation gespeichert und zur Verfügung gestellt. Weiterhin wird die Unabhängigkeit der Online-Daten gewährleistet.

Nach der Dateneingabe in die Datenbank greift die Applikation immer zuerst auf die vorhandenen Daten zu und versucht den angefragten Weg zu berechnen.

5.4 Nachbarschaft und Sortierung

Betrachtung eines einfachen Weges wie den ”Herderweg” aus Abbildung 2.2 lässt darauf schließen, dass auch bei nicht zusammenhängenden Knoten über eine Sortierung der Nachbarschaften über die Hausnummer eine Wegberechnung möglich ist.

In dem beschriebenen Beispielfall ist theoretisch eine Wegberechnung bis zu einer Straßenkreuzung der Zielstraße möglich, ab der dann die Reihenfolge der Häuser angenommen wird, bis zum Erreichen des angegebenen Ziels.

Eine komplexere Gebäudeanordnung ist im nebenanliegenden ”Kantweg” erkennbar.

Durch die Verschiebung der Objekte und durch die unregelmäßige Anordnung, ist algorithmisch über die Attribute zwar generell eine Sortierung umsetzbar, aber dies kann nicht zur korrekten Berechnung einer Wegstrecke genutzt werden. Eine Annahme, dass ein Haus mit der Nummer 3 neben einem Haus mit Nummer 2 liegen muss, ist falsch.

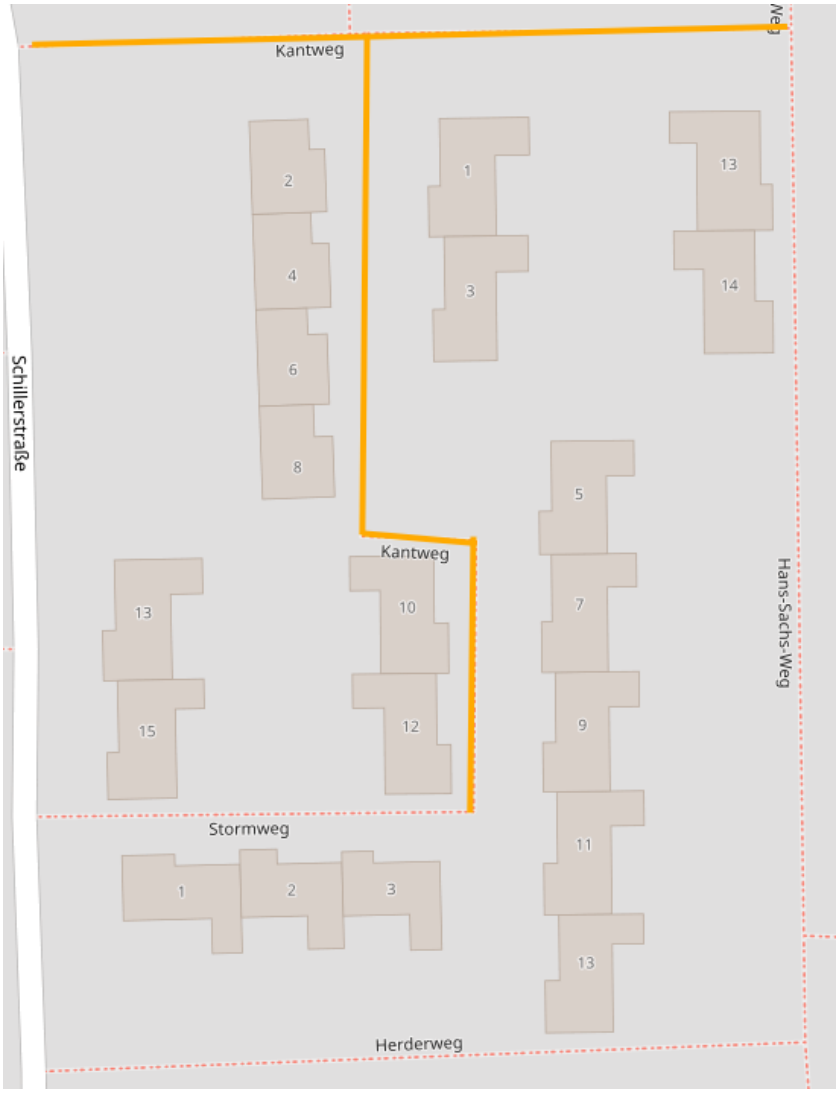

Abbildung 5.1: OSM-Kartenausschnitt Kantweg

Aus diesem Grund kann das Merkmal der doppelten Node an einem Gebäude angenommen werden. Durch Analyse des Gebäudes aus Kapitel 2.4.2 ist zu erkennen, dass die Node `<nd ref="3366359046"/>` doppelt vorhanden ist. Wird also bei jedem zu speichernden Gebäude der doppelte Knoten gefunden, kann dieser als Referenzknoten für ein Gebäude heranzgezogen werden.

5.5 Häuser als Ziel

Eine erste Betrachtung an exemplarischen Wegen legt nahe, die doppelt vorhandene Node als Referenz zu nehmen. Diese wird beim Anlegen der Daten mitgespeichert.

(a) (b)

Abbildung 5.2: OSM-Kartenausschnitt Feuerbachweg 4

Im aufgeführten Beispiel wird das Gebäude (`https://www.openstreetmap.org/way/329707243`) mit 8 Knoten beschrieben, wobei ein Knoten doppelt vorhanden ist. Der doppelte Knoten ist an der oberen, linken Ecke. Wird also ein Weg von der Straßenkreuzung auf dem direkten Weg zum Haus dargestellt, wird der Weg quer über die vorderen Häuser gerendert.

Zur Lösung des Problems einer besseren Anzeige, wird eine Methode implementiert, die für das Ziel die Node sucht, die am nächsten zur Straßenkreuzung ist. Damit wird immer eine vordere Ecke als Ziel angenommen.

5.6 Kantengewicht

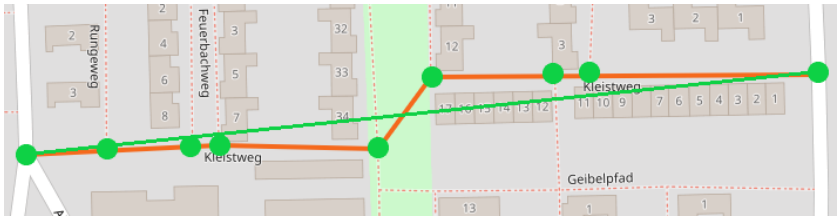

Abbildung 5.3: OSM-Kartenausschnitt Kleistweg

Am Beispiel des Kleistweg ist zu erkennen, dass dieser Weg diverse Wegkreuzungen, also

Nodes (grün), trägt. Wird ein Weg über den Kleistweg gesucht und es gibt kein Kantengewicht zwischen diesen Straßenkreuzungen, nimmt der Algorithmus folgerichtig eine kürzeste Verbindung der beiden Knoten äußerst links und äußerst rechts an.

Aus diesem Grund muss bei jeder Berechnung, die Entfernung zwischen allen in Frage kommenden Knoten berechnet werden. Im Grunde geht dies zu Lasten der Performance, aber vereinfacht das Datenmodell, da die Kantengewichte nicht für jedes Knotenpaar vorberechnet und gespeichert werden müssen.

5.7 Layer-Verwaltung

Ein Problem der Anwendung betrifft die Darstellung der berechneten Wege auf der Leaflet-Karte. Jeder neue Weg wird zusätzlich zu den bereits angezeigten Wegen gerendert, ohne dass vorherige Berechnungen entfernt werden. Dies führt dazu, dass die Karte nach mehreren Berechnungen überladen wirkt und die Benutzerfreundlichkeit eingeschränkt wird. Um nur den Weg anzuzeigen, der zuletzt angefragt wurde, muss beim Erstellen der Daten ein Layer gelöscht werden, sofern ein Layer besteht.

```
1  function processGeoJson(geoJsonData) {
2      if (geoJsonData.type === 'FeatureCollection') {
3          if (currentLayer) {
4              map.removeLayer(currentLayer);
5          }
6
7          currentLayer = L.geoJSON(geoJsonData.features, {
8              style: function (feature) {
9                  switch (feature.properties.name) {
10                     case 'Route':
11                         return {
12                             color: "#ff7800",
13                             weight: 3,
14                             opacity: 0.75
15                         };
16                 }
17             }
18         }).addTo(map);
19     } else {
20         console.error('Unerwartetes GeoJSON-Format');
21     }
22 }
```

Listing 5.1: api-call.js

Die Methode `processGeoJson()` nimmt ein GeoJson (sofern gültig) entgegen und fügt die

features, die die zu rendernden Objekte enthalten, einem Kartenobjekt hinzu. Das erstellte Objekt ist der aktuelle Layer und wird entsprechend gespeichert. Da initial kein Layer existiert, wird die Abfrage zum Entfernen eines bestehenden Layers nur dann ausgeführt, wenn es vorher bereits ein Rendering gab.

6 Fazit und Ausblick

In dieser Thesis wurde erfolgreich eine Anwendung zur Berechnung kürzester Wege entwickelt, die den A*-Algorithmus nutzt und eine moderne Softwarearchitektur implementiert. Das Ziel, mittels Implementierung eines A*-Algorithmus in JAVA als Web-Service mit passendem Datenmodell und einer Architektur mit dem freien Kartenmaterial von OpenStreetMap, kürzeste Wege innerhalb eines festgelegten Areals berechnen zu lassen, wurde durch die Kombination eines algorithmisch fundierten Ansatzes, einer performanten Datenbank und einer RESTful-API erfüllt.

Die Herausforderung Geokoordinaten als Datenbasis zu verwenden, um die Applikation unabhängig und mit generellen Daten aus OpenStreetMap lokal verwenden zu können, wurde erfolgreich durch die Integration eines speziell für diese Anwendung entworfenes Datenmodell realisiert.

Das Kernstück der Anwendung ist der A*-Algorithmus, da er eine optimale Balance zwischen Suchkomplexität und Effizienz bietet. Durch die Erweiterung des Dijsktra-Algorithmus mit einer Heuristik konnte sichergestellt werden, dass die Berechnungen nicht nur korrekte, sondern auch schnelle Ergebnisse liefern.

Die Integration einer Datenbank ermöglicht eine flexible und skalierbare Speicherung der zugrunde liegenden Geodaten. Hierbei wurde darauf geachtet, dass die Abfragen effizient gestaltet und optimal genutzt werden, um schnelle Zugriffsmöglichkeiten zu gewährleisten. Dies ist insbesondere für Anwendungen in der Praxis von Bedeutung, bei denen Geschwindigkeit und Datenmengen eine zentrale Rolle spielen.

Die Entwicklung der REST-API mittels Spring Boot bildet eine einfach und standardisierte Kommunikation zwischen der Anwendung und externen Systemen. Die Schnittstelle wurde so konzipiert, dass sie leicht erweiterbar und robust gegenüber zukünftigen Anforderungen bleibt. Insbesondere die Modularität der Spring-Boot-Architektur trägt dazu bei, eine klare Trennung der Komponenten und eine einfache Wartung sicherzustellen.

Die Anwendung ist als Prototyp zu verstehen und dient in erster Linie dazu, die praktische Umsetzbarkeit und die Leistungsfähigkeit des A*-Algorithmus in Kombination mit einer RESTful-API und einer Datenbank zu demonstrieren.

Blick in die Zukunft:
Während die grundlegende Funktionalität erfolgreich umgesetzt wurde, besteht noch Verbesserungspotenzial in Bezug auf Skalierbarkeit, Fehlerbehandlung und Performance-Optimierungen, um die Anwendung für den produktiven Einsatz weiterzuentwickeln. Die

Ergebnisse dieser Arbeit bieten eine solide Grundlage für weiterführende Forschung und Entwicklung. Potenziale zur Optimierung bestehen insbesondere in der Parallelisierung des Algorithmus und der Schnittstellen, der Nutzung von verteilten Datenbanksystemen sowie der Integration fortgeschrittener Heuristiken für spezifische Anwendungsbereiche. Ebenso könnte die Anwendung durch eine mobile Oberfläche ergänzt werden, um den Einsatz in der Praxis weiter zu erleichtern.

Das größte Potenzial liegt in einer Trennung der Services in getrennte Dienste, also einer Microservice-Architektur. Zum Beispiel mit einem eigenständigen NodeService. Wird eine Service-Discovery installiert, könnten alle Dienste mehrfach zur Verfügung gestellt werden und die Datenbank erweitert, ohne das die Dienste eingeschränkt sind.

Wird ein Dienst neugestartet, dann stehen während des Neustartens weiterhin die anderen Dienste zur Verfügung. Dadurch ist die Robustheit höher.

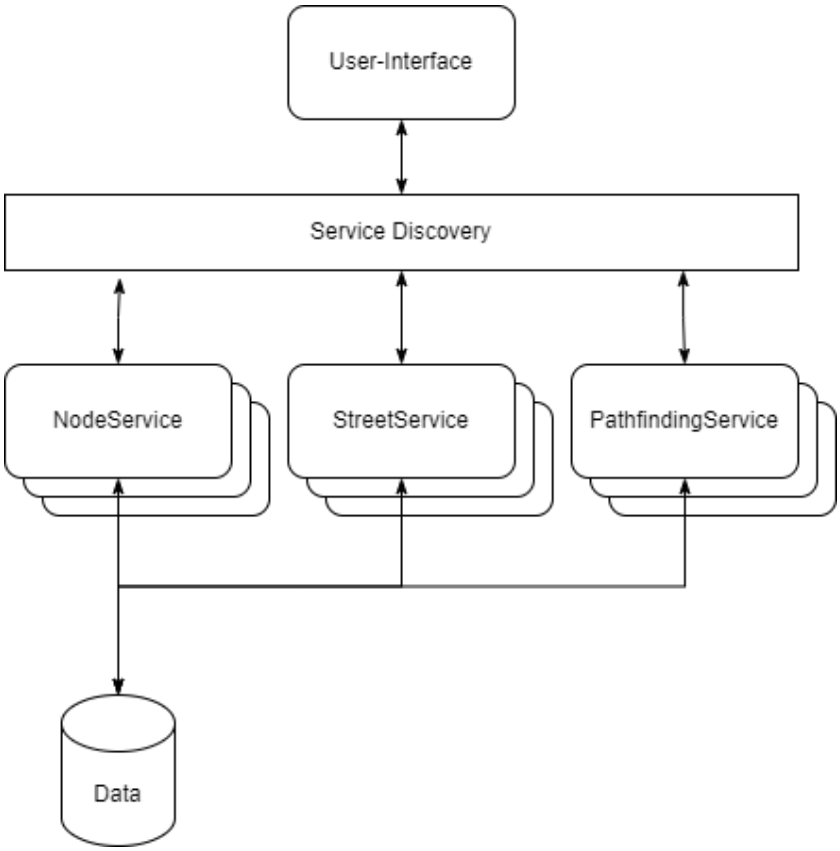

Abbildung 6.1: Service Discovery Architektur

Eine Schwierigkeit ergibt sich: der Cache muss mehrfach vorgehalten werden und es besteht Gefahr, dass der Zwischenspeicher über die Services nicht konsistent ist.

Der Cache ist im Rahmen dieser Arbeit lediglich für das Nutzererlebnis von Bedeutung.

Wird diese Anwendung weiter ausgerollt, muss neben der Konsistenz auch über die Lebenszeit des Caches nachgedacht werden. Das bedeutet konkret, dass Suchanfragen die eine bestimmte Zeit zurück liegen, verworfen werden sollten, um den Zwischenspeicher der Anwendung nicht zu belasten.

Eine Optimierung des Algorithmus ist ebenfalls über den Aufbau der Daten denkbar. Durch Betrachtung des beschriebenen Problems aus Kapitel 5.5 der nächsten Node, kann durch eine etwas komplexere Lösung ersetzt werden, in dem für die jeweils vorderen Nodes eine imaginäre Node auf die Straße gelegt wird. Werden diese imaginären Straßennodes in den Graph mit aufgenommen und Wege über diese Knoten berechnet, kann eine noch genauere Weganzeige vorgenommen werden.

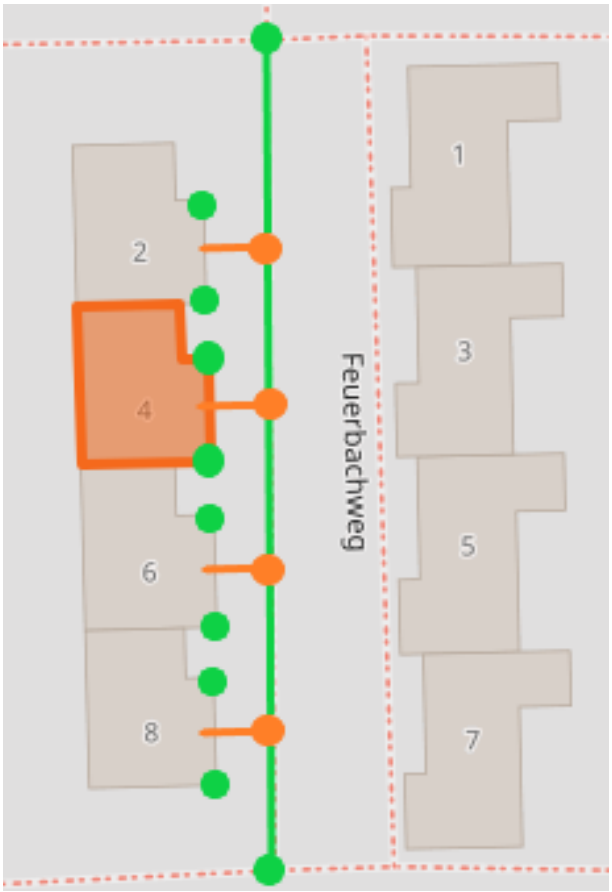

Abbildung 6.2: Virtuelle Nodes

Die grün dargestellten Knoten und der grün markierte Weg sind in der Datenbank vorhandene Daten. Die orangenen Knoten repräsentieren virtuelle Nodes, die jeweils aus der Mitte zweier vorhandener Nodes auf den Weg verschoben werden.

Weitere Möglichkeiten zur Steigerung der Berechnungseffizienz zur Bestimmung eines optimalen Weges, ist eine Wegtypisierung über ein Enum ein denkbares Feature, dass eine

Geschwindigkeit in ein Kantengewicht einfließen lassen kann.

Durch Vorwahl bestimmter Bewegungsarten in der UI, mittels Fahrzeug- oder Bewegungstypen (bspw. Fußweg, Fahrrad, Auto) eine Geschwindigkeit im Backend vorzuwählen, die bei der Berechnung eines möglichen Zeitbedarfs für einen kürzesten Weg eine Rolle spielen.

Das Enum repräsentiert diese möglichen Fahrzeuge bzw. Bewegungsarten und labelt diese. Ein Vorteil dieser Label ist, dass über die Methode `valueOfLabel()` der Fahrzeugtyp als Name übergeben wird und das ENUM zurück gegeben wird. Auf der anderen Seite kann bei Zugriff auf ein Enum das Label über die Schnittstelle an das Frontend übergeben werden. Weitere Verarbeitung im Interface ist demnach nicht notwendig.

```
1  public enum TypeEnum {
2      FOOT("foot"),
3      BIKE("bike"),
4      CAR("car");
5
6      public final String label;
7
8      private TypeEnum(String label) {
9          this.label = label;
10     }
11
12     public static TypeEnum valueOfLabel(String label) {
13         for (TypeEnum typeEnum : values()) {
14             if (typeEnum.label.equals(label)) {
15                 return typeEnum;
16             }
17         }
18         return null;
19     }
20
21     @Override
22     public String toString() {
23         return this.label;
24     }
25 }
```

Listing 6.1: TypeEnum.java

Ein Aspekt, der in der weiteren Entwicklung Beachtung finden sollte, ist die Integration automatisierter Tests. Diese gewährleisten nicht nur die Qualität und Zuverlässigkeit der Software, sondern erleichtern auch zukünftige Anpassungen und Erweiterungen.

Insbesondere Unit-Tests für die Logik und Integrationstests für die Schnittstellen könnten dazu beitragen, Fehler frühzeitig zu erkennen und die Stabilität des Systems zu erhöhen.

Abbildungsverzeichnis

Tabellenverzeichnis

Listings

Quellenverzeichnis

[1] *A*-Algorithmus*. Stand: 17.05.2024. URL: https://de.wikipedia.org/w/index.php?title=A*-Algorithmus&oldid=241111126.

[2] *Bundesamt für Kartographie und Geodäsie: Geodätische Referenzsysteme*. Stand: 03.11.2024. URL: https://www.bkg.bund.de/DE/Themen/Referenzsysteme/referenzsysteme.html.

[3] *Der A*-Algorithmus*. Stand: 17.05.2024. URL: https://algorithms.discrete.ma.tum.de/graph-algorithms/spp-a-star/index_de.html.

[4] *Der Dijkstra-Algorithmus*. Stand: 17.05.2024. URL: https://algorithms.discrete.ma.tum.de/graph-algorithms/spp-dijkstra/index_de.html.

[5] *Dijkstra-Algorithmus*. Stand: 17.05.2024. URL: https://de.wikipedia.org/w/index.php?title=Dijkstra-Algorithmus&oldid=243279594.

[6] *Greedy-Algorithmus*. Stand: 20.05.2024. URL: https://de.wikipedia.org/w/index.php?title=Greedy-Algorithmus&oldid=240097417.

[7] *GZIP*. Stand: 24.11.2024. URL: https://de.wikipedia.org/w/index.php?title=Gzip&oldid=244425040.

[8] *Haversine-Formula*. Stand: 02.11.2024. URL: https://en.wikipedia.org/w/index.php?title=Haversine_formula&oldid=1235273703.

[9] Krumke, Sven Oliver und Noltemeier, Hartmut. *Graphentheoretische Konzepte und Algorithmen*. Bd. 2. Vieweg-Teubner, 2009. ISBN: 978-3-8348-0629-1. DOI: 10.1007/978-3-8348-9592-9.

[10] *Offizielle Website GeoJSON*. Stand: 19.09.2024. URL: https://geojson.org/.

[11] *PBF-Format*. Stand: 24.09.2024. URL: https://wiki.openstreetmap.org/wiki/DE:PBF_Format.

[12] Prof. Dr.-Ing. Krischke, André und Dipl.Math. techn. Röpcke, Helge. *Graphen und Netzwerktheorie*. Carl Hanser Verlag, 2015. ISBN: 978-3-446-43229-1. URL: %7Bhttps://www.hanser-elibrary.com/isbn/9783446432291%7D.

[13] Teschl, Gerald und Teschl, Susanne. *Mathematik für Informatiker. Band 1: Diskrete Mathematik und Lineare Algebra*. Bd. 3. Springer-Verlag, 2008, S. 411–461. ISBN: 978-3-540-77432-7. DOI: 10.1007/978-3-540-77432-7.

[14] Wiltz, Alexander. *Kürzeste Wege auf allgemeinen Graphen. Studienarbeit*. Hochschule Kaiserslautern, 2024.

[15] *World Geodetic System 1984*. Stand: 02.11.2024. URL: https://de.wikipedia.org/w/index.php?title=World_Geodetic_System_1984&oldid=246550103.