

Hochschule Darmstadt

Fachbereich Mathematik und Informatik

EINBINDUNG EINER VOLLTEXTSUCHE MIT ELASTICSEARCH IN DAS „WHEELSCOUT“-PROJEKT MIT POI-DATEN VON ACCESSIBILITY.CLOUD UND GEODATEN VON OPENSTREETMAP

Abschlussarbeit zur Erlangung des akademischen Grades

Master of Science (M.Sc.)

im Studiengang Data Science

vorgelegt von

Minh Duc Luong

Matrikelnummer: 758217

02.11.2020

Referentin: Frau Prof. Dr. Harriehausen-Mühlbauer

Koreferentin: Frau Prof. Dr. Groos

Erklärung der Urheberschaft

Ich versichere hiermit, dass ich die vorliegende Arbeit selbständig verfasst und keine anderen als die im Literaturverzeichnis angegebenen Quellen benutzt habe. Alle Stellen, die wörtlich oder sinngemäß aus veröffentlichten oder noch nicht veröffentlichten Quellen entnommen sind, sind als solche kenntlich gemacht. Die Zeichnungen oder Abbildungen in dieser Arbeit sind von mir selbst erstellt worden oder mit einem entsprechenden Quellennachweis versehen. Diese Arbeit ist in gleicher oder ähnlicher Form noch bei keiner anderen Prüfungsbehörde eingereicht worden.

Darmstadt, den 02.11.2020



Unterschrift

Vorwort

Ich möchte mich bei Christoph Boßmann und Jakob Höfner für die Unterstützung bei der Einbindung der Suchmaschine Elasticsearch in das „WheelScout“-Projekt herzlich bedanken. Zusätzlich möchte ich mich auch bei SOZIALHELDEN e.V. für die Bereitstellung der Points of Interest Daten mit Barriereninformationen bedanken. Zum Schluss möchte ich mich vor allem bei meiner Referentin Frau Prof. Dr. Bettina Harriehausen-Mühlbauer bedanken, die es mir ermöglicht hat, meine Masterarbeit in diesem Themenbereich zu verfassen.

Abstract

In society, physically handicapped people have no easy way to get from A to B in everyday life without encountering obstacles due to the non-existent barrier-free infrastructure. For this reason, the WheelScout project was initiated at Darmstadt University of Applied Sciences. WheelScout is a navigation app that is able to bypass barriers such as stairs when routing from A to B. With the app, everyday life can be better planned and organized. The WheelScout app does not yet have any information about barriers in buildings, and the WheelScout App does not yet offer a high-performance full text search or radius search to find the desired destination as efficiently as possible. In my master's thesis, I deal with the integration of Elasticsearch and the storage of points of interest data with barrier information and address data in order to enable optimal full-text and radius searches in the geodata environment. Complex search queries are to be implemented in the geodata, such as displaying selected categories such as toilets during a routed route or searching for specific terms within a location, such as „McDonald's in Darmstadt“. By integrating Elasticsearch on the WheelScout server, the is now able to carry out a full text and radius search in the geofield.

Keywords: Elasticsearch, NoSQL database, Document-oriented database, search engine, Fulltextsearch, Points of Interest

Kurzfassung

Körperlich eingeschränkte Menschen haben es in der Gesellschaft aufgrund von nicht vorhandener barrierefreier Infrastruktur nicht leicht im Alltag von A nach B zu gelangen, ohne auf Hindernisse zu stoßen. Aus diesem Grund wurde das Projekt WheelScout an der Hochschule Darmstadt initiiert. WheelScout ist eine Navigations-App, die in der Lage ist, beim Routen von A nach B Barrieren zu umgehen, wie beispielsweise Treppen. Durch die App kann der Alltag besser geplant und organisiert werden. Jedoch verfügt die „WheelScout“-App derzeit über keine Informationen über Barrieren in Gebäuden. Auch bietet die „WheelScout“-App noch keine performante Volltextsuche oder Radiusuche an, um gewünschte Ziele möglichst effizient zu finden. In dieser Masterarbeit beschäftige ich mich mit der Einbindung von Elasticsearch und der Speicherung von Points of Interest Daten, Barriereninformationen und Adressdaten, um eine optimale Volltextsuche und Radiusuche im Geodatenumfeld zu ermöglichen. Dabei sollen komplexe Suchanfragen für Geodaten implementiert werden. Dadurch können selektierte Kategorien, wie beispielsweise Toiletten, während einer gerouteten Strecke angezeigt oder die Suche nach bestimmten Begriffen innerhalb eines Ortes, wie zum Beispiel „McDonald’s in Darmstadt“, ermöglicht werden. Durch die Einbindung von Elasticsearch in das „WheelScout“-Projekt dieses nun in der Lage, eine Volltext- und Radiusuche im Geofeld durchzuführen.

Schlagwörter: Elasticsearch, NoSQL-Datenbank, Dokumentenorientierte Datenbank, Suchanwendung, Suchmaschine, Volltextsuche, Radiusuche, Points-of-Interest, Geodaten

Abkürzungen

e.V.	eingetragener Verein
HTTP	Hypertext Transfer Protocol
RAM	Random Access Memory
REST	Representational State Transfer
API	Application Programming Interface
ASCII	American Standard Code for Information Interchange
POI	Point of Interest
DSL	Domain Specific Language
DML	Data Modifying Language
SQL	Structured Query Language
JSON	JavaScript Object Notation
MUR	Minimal umgebendes Rechteck
URL	Uniform Resource Locator
GUI	Graphical User Interface
CSV	Comma-separated values
NoSQL	Not only SQL

Abbildungsverzeichnis

3.1	Beispiel: Chord-System	9
3.2	CAP-Theorem	11
4.1	Beispiel: Analyzer-Prozess	16
4.2	Beispiel: Invertierte Indexierung	19
4.3	Beispiel: Die Ergebnisse für die Anfrage: „Menschenrecht Wohnen“	21
5.1	Geometrienklasse im „Simple Feature Access“-Standard	27
6.1	Auswahl anhand der Wheelmap-Kategorien	30
7.1	Infrastruktur	36
7.2	Use-Case-Suchfunktion	37
7.3	Elasticsearch-Java-Client und POIServer: UML-Klassendiagramm	45
8.1	POIs in Deutschland	48
8.2	Duetschland: Minimal umgebendes Rechteck (MUR)	51
8.3	Suche nach dem Begriff „McDonald’s“	57
8.4	Grafische Darstellung der Koordinaten und Länge der gerouteten Strecke	63
8.5	Radiussuche während des Routings	66
9.1	Kommunikation zwischen REST API und „WheelScout“-App (Smartphone)	68
9.2	Erstellung von Icons für die „WheelScout“-App	72
9.3	Frontend „WheelScout“-App	73

Tabellenverzeichnis

4.1	Konzeptvergleich zwischen SQL und Elasticsearch	14
6.1	Vergleich der Geotools	31
6.2	Reverse Geocoding in Deutschland in Prozent	32
8.1	Datenvollständigkeit der POI-Daten in Deutschland in Prozent	56
8.2	Nach der Verarbeitung der POI-Daten in Prozent	57
8.3	Aggregationstabelle	58
9.1	Contabo-Server	67

Listings

2.1	Abbildung eines Gebäudes im GeoJSON-Format	4
4.1	Beispiel: Setting und Mapping in Elasticsearch	16
4.2	Beispiel: „Standard“-Analyzer	17
4.3	Beispiel: Ergebnis „Standard“-Analyzer	17
4.4	Beispiel: Mapping in Elasticsearch	18
4.5	Beispiel: Search-Template in Elasticsearch	24
7.1	Indexeinstellungen in Elasticsearch	38
7.2	Char-Filter, Tokenizer, Token-Filter	39
7.3	Analyzer für Points of Interest	41
7.4	Mapping der Points of Interest	43
8.1	Bildung von MUR für Deutschland und Extraktion der Koordinaten von Eckpunkten, um eine Umwandlung in XYZ-Werte zu ermöglichen	49
8.2	verschachtelte For-Schleife, um durch alle Kacheln zu iterieren	51
8.3	Beispiel: Rastersuchanfrage an accessibility.cloud	52
8.4	Beispiel: Response von accessibility.cloud	52
8.5	Kategorienanfrage an accessibility.cloud	53
8.6	Response zur Kategorienanfrage an accessibility.cloud	53
8.7	1. SQL Query, um 2 Attribute zu einem Attribut („accessibleWith“) zusammenzuführen	54
8.8	SQL Query, um für jede Straße nur eine Koordinate zu wählen	55
8.9	1. SQL Query, um Adressdaten in den POI-Daten zu ergänzen	56
8.10	2. SQL Query, um Adressdaten in den POI-Daten zu ergänzen	56
8.11	Verfahren zur Normalisierung der Begriffe von POIs	59
8.12	Volltextsuche nach POIs	60
8.13	Volltextsuche nach Straßendaten	61
8.14	Radiussuche für Points of Interest	62
8.15	GraphHopper API-Anfrage, um Koordinaten und Distanzen einer gerouteten Strecke zu erhalten	63
8.16	GraphHopper API-Anfrage an den „WheelScout“-Server, um Koordinaten und Distanzen einer gerouteten Strecke zu erhalten	64
8.17	Methode zur Extraktion von Koordinaten einer gerouteten Strecke	64
8.18	Aggregation von Kategorien	66
9.1	Aufruf des Searchtemplates „radiussearchAutocomplete“	69
9.2	Aufruf des Searchtemplates „POIFulltextsearchAutocomplete“	69
9.3	Aufruf des Searchtemplates „AddressFulltextsearchAutocomplete“	70
9.4	Aufruf des Searchtemplates „groupby_category“	70

9.5 Aufruf der Anfrage „POIs während des Routings anzeigen“	71
---	----

Inhaltsverzeichnis

Erklärung der Urheberschaft	I
Vorwort	II
Abstract	III
Kurzfassung	IV
Abkürzungen	V
Abbildungsverzeichnis	VI
Tabellenverzeichnis	VII
Listing	IX
1 Einleitung	1
1.1 Motivation	1
1.2 Ziel der Arbeit	2
1.3 Gliederung	2
2 Bisheriger Forschungsstand	3
2.1 „WheelScout“-App	3
2.2 Wheelmap und accessibility.cloud	4
2.3 Suchmaschinen im Geoumfeld	5
2.3.1 Google Maps	5
2.3.2 Nominatim	5
2.3.3 Bing Maps	5
2.3.4 Photon	6
3 Grundlagen NoSQL-Datenbanken	7
3.1 Hauptmerkmale von NoSQL-Datenbanken	7
3.2 Skalierbarkeit durch DistrubedHashTable	8
3.3 CAP-Theorem	11
3.3.1 ACID und BASE	12
4 Einführung in Elasticsearch	14
4.1 Bezug zu relationalen Datenbanken	14
4.2 Einstellungen (Settings) und Mapping	15
4.2.1 Shards und Replicas	15
4.2.2 Analyzer in Elasticsearch	15
4.2.3 Mapping	17
4.2.4 Überblick über Datentypen	18

4.3	Indexierung in Elasticsearch	18
4.3.1	Volltextsuche	19
4.3.2	Relevanz der Dokumente (Score)	20
4.4	Anfragen an den Elasticsearch-Server	22
4.4.1	Elasticsearch-Client	22
4.4.2	REST API	23
4.4.3	Search-Templates	23
5	Verwendete Technologien und Komponenten	25
5.1	Verwendete Programmiersprachen	25
5.1.1	Python	25
5.1.2	Java	25
5.2	Kibana	25
5.3	Docker	26
5.4	PostgreSQL mit PostGIS	26
5.5	Cerebro	27
5.6	elasticsearch-dump	28
5.7	Dejavu: Elasticsearch Web UI	28
5.8	Nutzung von APIs	28
5.8.1	GraphHopper-API	28
6	Methodik	29
6.1	Beschaffung der Geodaten	29
6.1.1	Beschaffung der Point of Interest Daten	29
6.1.2	Auswahl der Geotools-API	31
6.1.3	Datenqualität der Geotools	31
6.1.4	Geocoding der Geotools	32
6.1.5	Offline-Geocoding	32
6.1.6	Kosten und Lizenzierung der API-Nutzung	32
6.2	Zusammenführen der Daten und Implementierung der Anfragen in Elasticsearch	33
6.3	Einbindung von Elasticsearch in das „WheelScout“-Projekt	33
6.3.1	„WheelScout“-Backend	33
6.3.2	„WheelScout“-Frontend	34
7	Design	35
7.1	IT-Infrastruktur	35
7.2	Use-Case-Diagramm	36
7.3	Schema der Geodaten in Elasticsearch	38
7.3.1	Einstellungen	38
7.3.2	Analyzer	39
7.3.3	Mapping bzw. Datenstruktur der Geodaten in Elasticsearch	42
7.4	UML-Klassendiagramm vom Elasticsearch Java Client	44
7.4.1	„POIs während des Routings anzeigen“-Anfrage	45

8	Implementierung und Realisierung	47
8.1	1. Skript: Beschaffung der Geodaten	47
8.2	2. Skript: Datentransformation und -aufbereitung	54
8.2.1	Ergänzung der Adressdaten von POIs	55
8.3	3. Skript: Datentransformation und -aufbereitung	57
8.3.1	Standardisierung der Begriffe/Bezeichnung von POIs	57
8.4	4. Skript: Implementierung der DSL-Anfragen	59
8.4.1	Volltextsuche für Points of Interest und Straßendaten	60
8.4.2	Radiussuche für Points of Interest	61
8.4.3	Points of Interest während des Routings anzeigen	63
8.4.4	Aggregation von Feldern	66
9	Einbindung von Elasticsearch in das „WheelScout“-Projekt	67
9.1	Integration von Elasticsearch auf dem „WheelScout“-Backend-Server	67
9.1.1	Kommunikation zwischen „WheelScout“-Server und der „WheelScout“-App	68
9.1.2	Ausführung der Anfragen in der „WheelScout“-App	68
9.2	Frontend der „WeelScout“-App	71
9.2.1	Entwicklung eines Frontend für die Anzeige von POIs in der „WeelScout“-App (Prototyp)	71
9.2.2	Ergebnis des Frontend für die Anzeige von POIs in der „WheelScout“-App (Prototyp)	73
10	Fazit und Ausblick	74
10.1	Fazit	74
10.2	Ausblick	74
10.2.1	„WheelScout“-Backend	75
10.2.2	„WheelScout“-Client (Frontend)	76
	Inhalt der CD	XIII
	Literatur	XX

1 Einleitung

1.1 Motivation

„Niemand darf wegen seiner Behinderung benachteiligt werden“ (Artikel 3 Absatz 3 Satz 2 Grundgesetz). Doch müssen sich körperlich eingeschränkte Menschen bereits im alltäglichen Leben wie beispielsweise bei der Wohnungssuche, mit verschiedenen Hindernissen konfrontiert sehen, da nur zwei Prozent aller Wohnungen in Deutschland die Anforderungen einer Barrierefreiheit nach DIN-Norm 18040 [1] erfüllen. Zudem stellt das Wohnen ein elementares Grundbedürfnis für den Menschen dar und zählt unter anderem zu den Menschenrechten. Jeder Mensch hat das Recht auf Selbstbestimmung und darauf, sein Leben frei zu gestalten. Doch eingeschränkte Menschen haben nicht nur bei der Wohnungssuche deutliche Schwierigkeiten, sondern auch im Alltag. Beim Einkaufen, beim Treffen mit Freunden im Restaurant oder auch der Weg zur Arbeit können eine große Herausforderung darstellen. Die Infrastruktur in Deutschland weist hinsichtlich der Barrierefreiheit noch deutliche Defizite auf und das, obwohl in Deutschland 7.9 Millionen Menschen im Besitz eines Behindertenausweises sind. Zudem sind davon 4.6 Millionen Menschen körperlich stark eingeschränkt [2]. Durch die Folge der demografischen Entwicklungen in Deutschland nimmt die Zahl der altersbedingten Einschränkungen von Menschen in der nahen Zukunft deutlich zu und eine barrierefreie Infrastruktur sollte ein angestrebtes Ziel sein. Eine Untersuchung von ADAC [3] zeigt, dass eine barrierefreie Infrastruktur für 10 Prozent notwendig und für 30 Prozent der Bevölkerung hilfreich wäre. Denn auch Schwangere, Mütter mit Kinderwagen, ältere sowie übergewichtige Menschen oder auch Menschen mit temporären Einschränkungen profitieren von einer barrierefreien Infrastruktur. „Denn wo Orte, Räume oder Kommunikationsmittel nicht barrierefrei sind, bleibt Teilhabe am kulturellen und politischen Leben an der Arbeitswelt und in der Freizeit verwehrt“ [4]. Aufgrund der mangelnden barrierefreien Infrastruktur in Deutschland wurde das Projekt WheelScout an der Hochschule Darmstadt von Frau Prof. Dr. Harriehausen-Mühlbauer initiiert. Das „WheelScout“-Projekt verfolgt das Ziel, ein Navigationssystem für körperlich beschränkte Menschen oder eingeschränkte Personen aufgrund von anderen Umständen, zur Verfügung zu stellen, um den Alltag selbstständig und selbstbestimmt zu meistern. Das Projekt befindet sich aktuell noch in der Entwicklungsphase und wird stetig verbessert. Eine Routenführung von A nach B bietet die „WheelScout“-App bereits an. Jedoch beinhaltet die App derzeit kaum Barriereinformationen von Gebäuden (POIs). Auch bietet die „WheelScout“-App noch keine Volltext- oder Radiussuche mit unterschiedlichen Filtermöglichkeiten an. Die Suchanwendung bei einer Navigations-App stellt eine der wichtigsten Funktionen dar. Ohne eine ausgereifte Suchfunktion lässt sich der gesuchte Ort nur unter erschwerten Umständen oder auch gar nicht auffinden.

1.2 Ziel der Arbeit

Das Projekt Wheelmap (accessibility.cloud) des SOZIALHELDEN e.V. stellt die hierzu notwendigen Informationen zur Barrierefreiheit dieser sogenannten Points of Interest zur Verfügung. Es soll untersucht werden, inwiefern diese Informationen in das „WheelScout“-Projekt eingebunden werden können. Die accessibility.cloud-Daten sind anhand von Ampelfarben entsprechend ihrer Barrierefreiheit eingeordnet. Da WheelScout jedem Nutzer die Möglichkeit bietet, individuell anzugeben, welche Barrieren er überwinden kann, können für jeden Nutzer individuell die Gebäude ausgegeben werden, welche für diesen zugänglich sind. Des Weiteren bietet accessibility.cloud eine Einteilung der Gebäude in Kategorien. Aus diesen kann der Nutzer wählen, nach welcher Art von Points of Interest er suchen möchte. Die Aufgabe lässt sich in vier Teilaufgaben aufgliedern. Bei der ersten Aufgabe sollen alle Points of Interest, die sich in Deutschland befinden, sowie alle Straßendaten in Deutschland von accessibility.cloud oder von anderen Quellen bezogen werden. Dabei muss untersucht werden, wie die Points of Interest von accessibility.cloud bezogen und mit welchem Geotool die Straßendaten beschaffen werden können. Bei der zweiten Aufgabe soll ein Datenstruktur für Elasticsearch erstellt werden, um die Daten zusammenzuführen; Daten bereinigt werden gegebenenfalls angepasst werden, um ein Import der Daten in Elasticsearch zu ermöglichen. Die dritte Teilaufgabe beinhaltet die Implementierung von komplexen Suchanfragen in Elasticsearch, um eine Volltext- und Radiusuche mit Filtereinstellungen im Geoumfeld zu ermöglichen. Die vierte Teilaufgabe beschäftigt sich mit der Einbindung von Elasticsearch in die „WheelScout“-App, um eine performante skalierbare Suchmaschine im Geofeld für die „WheelScout“-App bereitzustellen.

1.3 Gliederung

In Kapitel 2 wird der aktuelle Stand des „WheelScout“-Projekts und das Wheelmap-Projekts näher beschrieben und weitere bekannte Suchmaschinen im Geoumfeld vorgestellt. In Kapitel 3 folgt eine kurze Einführung in die NoSQL-Datenbank. Im nächsten Kapitel werden grundlegende Konzepte in Elasticsearch vermittelt. In Kapitel 5 werden wegwendenden Programmiersprachen und Technologien vorgestellt. In Kapitel Methodik wird die Vorgehensweise bzw. die gewählte Methodik detailliert beschrieben. In Kapitel Design folgt ein grafischer Überblick über die gesamte IT-Infrastruktur und eine Darstellung der Dokumentenstruktur in Elasticsearch sowie Use-Cases, die in Elasticsearch abgebildet werden sollen. Im selben Kapitel wird noch ein Klassendiagramm des Elasticsearch-Java-Clients beschrieben. In Kapitel 8 wird der gesamte Prozess der Implementierung und Realisierung mit Ausschnitten aus dem Programmcode beschrieben. In Kapitel 9 erfolgt die Einbindung von Elasticsearch in das „WheelScout“-Projekt. Im letzten Kapitel erfolgt ein Fazit und ein Ausblick auf die weiteren Schritte.

2 Bisheriger Forschungsstand

In diesem Kapitel wird der aktuelle Stand des Projekts WheelScout näher erläutert. Daraufhin folgt eine detaillierte Beschreibung der beiden Projekte Wheelmap und accessibility.cloud. Am Ende des Kapitels werden bekannte Suchmaschinen im Geodatenumfeld vorgestellt.

2.1 „WheelScout“-App

Das „WheelScout“-Projekt wurde von Frau Prof. Dr. Harriehausen-Mühlbauer an der Hochschule Darmstadt initialisiert. Im Rahmen des „WheelScout“-Projekts arbeitet Frau Prof. Dr. Harriehausen-Mühlbauer in gemeinsamer Kooperation mit dem Verband der Zivilinvaliden LAPIC/UVZ aus Südtirol zusammen, der die Weiterentwicklung der „WheelScout“-App finanziert. Ziel des Projekts ist es, eine Navigations-App für Menschen mit körperlichen Einschränkungen zu entwickeln [5]. Dabei wurde Kartenmaterial von OpenStreetMap verwendet. Das Kartenmaterial bei OpenStreetMap sind lizenzfrei und können kostenlos für die Weiterverarbeitung genutzt werden [6]. Die „WheelScout“-App nutzt hierfür etablierte Technologien. Für die Routenplanung wird die GraphHopper Routing Engine verwendet und für die Speicherung von Barrieredaten wurde eine relationale Datenbank (MariaDB) gewählt. Zum aktuellen Zeitpunkt enthält die relationale Datenbank Barriereinformationen über „Treppen“ mit folgenden Attributen:

- geografische Koordinaten
- Breite
- Stufenhöhe
- Stufentiefe
- Geländer
- Rampe
- Anzahl Stufen

Dabei kann jeder Nutzer über die Profileinstellung - die „Barrieren“ eigenständig anpassen und definieren. Kann der Anwender beispielsweise ausschließlich Treppenstufen mit maximal 7 cm Höhe überwinden, kann er diese Information in der „WheelScout“-App hinterlegen und als Barriere deklarieren. Beim Routing der Strecke wird diese Information in der App genutzt, um eben solche Treppenstufen zu umgehen und der Anwender erhält dadurch die kürzeste überwindbare Route. Weitere Barrieren, wie Steigung oder Bodenbeschaffenheit, folgen demnächst. WheelScout bietet dem Anwender zudem die Möglichkeit noch nicht markierte Barrieren (Treppen) eigenständig in die „WheelScout“-App einzutragen, um somit den Datenbestand von Barrieren zu erweitern. Die „WheelScout“-App ist zurzeit nur auf Android Geräten nutzbar und soll Ende 2021 als Google Playstore App veröffentlicht werden.

2.2 Wheelmap und accessibility.cloud

Beide Projekte, Wheelmap und accessibility.cloud wurden von SOZIALHELDEN e.V. initialisiert. Ziel der SOZIALHELDEN e.V. ist es, gesellschaftliche Probleme anzusprechen, um Menschen zu sensibilisieren und zum Umdenken zu bewegen [7]. Im Projekt accessibility.cloud sollen Informationen von Gebäuden zur Barrierefreiheit in einem standardisierten Format geteilt werden. Im Frontend von accessibility.Cloud befindet sich die Wheelmap. Die Wheelmap ist eine Online-Karte, welche Points of Interest mit Barriereinformationen anzeigt und als Webseite oder als App zur Verfügung steht. Die Koordinaten sowie die POI-Daten stammen größtenteils von der kostenlosen Karte OpenStreetMap, wo diese auch gehostet werden. Informationen zu Barrieren von Points of Interest sind in den OpenStreetMap-Daten nur begrenzt vorhanden, weshalb die Wheelmap-App den Nutzern die Möglichkeit anbietet, Barriereinformationen von Points of Interest selbstständig in die Wheelmap-App einzupflegen. Eingepflegte Daten von Nutzern der Wheelmap und Daten weiterer Partner mit unterschiedlichen Lizenzen werden in accessibility.cloud zusammengeführt. Im Backend von accessibility.cloud befindet sich eine dokumentenorientierte NoSQL Datenbank (MongoDB Atlas). Die Datenbank speichert POI-Daten als binäre JSON (BSON) Dokumente ab. MongoDB verwendet bei der Indexierung der Dokumente eine inverse Indexierung, weshalb eine Volltextsuche in MongoDB möglich ist. Jedoch werden nur grundlegende Volltextsuchfunktionen angeboten [8, S. 144 - 152]. Eine Volltextsuche in Wheelmap wird nicht angeboten. accessibility.cloud bietet eine API, über welche per HTTP-Request, Daten im GeoJSON-Format abgefragt werden können. Das GeoJSON-Format verwendet JSON und ist ein offenes Format, um geografische Daten nach dem Internetstandard RFC 7946 abzubilden. Dieser Standard wurde im August 2016 von der Vereinigung Internet Engineering Task Force (IETF) veröffentlicht. Das GeoJSON-Format (siehe Listing 2.1) besteht aus („geometry“) und („properties“). Die („geometry“) beinhaltet ein Feld („type“) und ein Feld („coordinates“). Das Feld („type“) definiert dabei, um welche Geometrie es sich handelt. Dabei wird zwischen einer und mehrfacher Geometrie(n) unterschieden. Eine Geometrie kann ein geografischer Koordinatenpunkt, ein Streckenzug oder ein Polygon sein. Mehrfache Geometrien bestehen aus mehreren Koordinatenpunkten, Streckenzügen oder Polygonen. Das Feld („coordinates“) im Listing 2.1 beinhaltet eine geografische Koordinate, die den Standort eines Gebäudes abbilden soll. Im Feld „properties“ werden alle weiteren Informationen abgebildet [9].

```

1 {
2   "geometry": {"coordinates": [8.6382018, 49.8673135], "type": "Point"},
3   "properties": {
4     "name": "Hochhaus der Hochschule Darmstadt (Gebäude C10)",
5     "accessibility": {
6       "accessibleWith": {"wheelchair": true, "wheelchairLocalized": "Rollstuhl"},
7       "accessibleWithLocalized": "Zugänglich mit"
8     },
9     ...
10  }
11 }
```

Listing 2.1: Abbildung eines Gebäudes im GeoJSON-Format

Wheelmap verfolgt grundsätzlich das Ziel, weltweit genutzt werden zu können. Allerdings ist eine gute Abdeckung noch nicht überall gegeben. Nach Absprache mit SOZIALHELDEN e.V. ist die Nutzung der API in der Entwicklungsphase maximal für 12 Monate kostenlos. Erhält das Projekt eine Förderung oder wird einem externen Nutzerkreis zur Verfügung gestellt, würde eine Nutzungsgebühr in Höhe von 80,00 EUR pro Monat anfallen.

2.3 Suchmaschinen im Geoumfeld

2.3.1 Google Maps

Google Maps ist weltweit der bekannteste Anbieter für Routenführung. Google Maps wird als Webseite sowie als Smartphone-App angeboten. Mehr als 1 Milliarde aktive User kann die NavigationsApp monatlich vorweisen. Google Maps bietet Online-Karten in mehr als 220 Länder an und Verkehrsupdates erfolgen in 171 Ländern [10]. Weiterhin verfügt Google Maps über 150 Million Daten an Points of Interest und es werden 25 Million Aktualisierungen pro Tag ausgeführt [11]. Die Navigations-App bietet bei der Routenführung verschiedene Profile für den Anwender an. Ob zu Fuß, mit dem Auto oder mit dem Fahrrad - die App berechnet ganz individuell Routen und Zeiten, damit die User ihren gewünschten Zielort erreichen können. Google Maps bietet auch Routenführung mit der Einbindung von öffentlichen Verkehrsmitteln oder Flugzeugen an. Eine Volltextsuche für Points of Interest und Straßendaten bietet Google Maps an, jedoch sind bei den Gebäuden (POIs) keine Informationen zur Barrierefreiheit hinterlegt.

2.3.2 Nominatim

Nominatim ist ein Open Source Geocoding und Freitextsuche Tool im Geoumfeld und wurde 2009 von Brian Quinion implementiert. Im Backend von Nominatim befindet sich eine relationale PostgreSQL-Datenbank, die Geodaten mit PostGIS (Erweiterung für PostgreSQL-Datenbanken) als Objekte speichert [12]. Nominatim verwendet Daten von OpenStreetMap, wo jeweils ein minütliches Update erfolgt. Im Jahr 2013 wies die Datenbank mehr als 120 Mio. gespeicherte Objekte auf. Das sind mehr als 1 TB an Daten, die gespeichert werden müssen [13]. Die Nominatim API bietet auch eine Reverse-Geocoding an, d.h. Adressdaten lassen sich auch als geografische Koordinaten ausgeben. Die Software Nominatim wird auf GitHub kostenlos bereitgestellt [14].

2.3.3 Bing Maps

Bing Maps ist eine Online-Karte von Microsoft, die im Dezember 2005 unter der Bezeichnung „Live Search Maps“ erschienen ist. Bing Maps stellt eine Konkurrenz zu Google Maps dar.

„Bing Maps“-Suchanwendung verfügt über eine Volltextsuche und bietet auch eine API an, um Geocoding zu ermöglichen. Die Datenbank verfügt über 165 Terabyte an Satellitenbildern, die aus der Vogelperspektive in Bing Maps betrachtbar sind [15].

2.3.4 Photon

Photon ist ein Open Source Geocoding Tool, welches von Kosmoot entwickelt wurde. Das Tool basiert auf Elasticsearch. Photon bezieht seine Daten von OpenStreetMap, die wiederum über eine Nominatim-Datenbank bezogen werden. Die API Response wird im GeoJSON-Format ausgegeben. Photon bietet zurzeit eine Volltextsuche für Straßendaten in folgenden vier Sprachen an: Englisch, Deutsch, Französisch und Italienisch [16].

Im nächsten Kapitel folgt eine Einführung in NoSQL-Datenbanken.

3 Grundlagen NoSQL-Datenbanken

In diesem Kapitel werden die grundlegenden Konzepte von NoSQL-Datenbanken vorgestellt. Weiterhin werden Eigenschaften und Beschränkungen von NoSQL-Datenbanken thematisiert. Abschließend erfolgt ein Vergleich der NoSQL-Datenbank mit der relationalen Datenbank sowie eine Erläuterung der Vorteile von NoSQL-Datenbanken.

3.1 Hauptmerkmale von NoSQL-Datenbanken

Elasticsearch ist eine NoSQL-Datenbank. Im Vergleich zu relationalen Datenbanken verfolgen NoSQL-Datenbanken andere Ansätze. NoSQL-Datenbanken wurden so konzipiert, dass sie horizontal sehr gut skalierbar sind, d.h. NoSQL-Datenbanken lassen sich auf mehrere Knoten verteilen. Hierdurch können Massenanfragen im Milliardenbereich pro Tag effizienter beantwortet werden. Relationale Datenbanken hingegen sind nur begrenzt verteilbar und gelangen schnell an ihre Grenzen [8, S. 2]. Google gab 2019 bekannt, dass sie im Durchschnitt pro Tag 3.5 Milliarden Suchanfragen erhalten [17]. Ohne eine Verteilung der Datenbank auf mehrere Knoten, wäre es nicht möglich, Massenanfragen in diesem Ausmaß performant zu beantworten. Es gibt dabei verschiedene Arten von NoSQL-Datenbanken:

- **Key Value Store** speichert Daten als Schlüssel-/Wert-Paare ab. Der Key (Schlüssel) ist dabei unique (eindeutig), wie beim Primärschlüssel in einer SQL-Datenbank. Der Value (Wert) hat den Datentyp String und wird über den Schlüssel gefunden. Eine Schlüssel-Werte-Datenbank kann keine hierarchische Datenstruktur abbilden [18, S. 198].
- **Document Store** kann als Erweiterung zu Schlüssel-Werte-Datenbank betrachtet werden; mit dem Unterschied, dass Hierarchien oder komplexe Datenstrukturen in Form von Dokumenten abbildbar sind. Ein weiterer Vorteil ist, dass Werte mit komplexeren Datentypen umgehen kann [18, S. 198].
- **Column Store** ähnelt der relationalen Datenbank am stärksten. In Spaltenorientierte Datenbank werden Daten in Tabellen abgespeichert mit dem Unterschied, dass eine Verschachtelung von Tabellen möglich ist [18, S. 198].
- In **Graph Store** werden Daten als Knoten abgebildet und Kanten bildet die Beziehungen zu den jeweiligen Knoten ab [18, S. 198].

Alle diese Kategorien von Datenbanken weisen eine Gemeinsamkeit auf. Denn alle diese Datenbanken sind horizontal skalierbar und schemafrei, obwohl der Begriff Schemafreiheit nicht ganz treffend ist [19, S. 27]. NoSQL-Datenbanken werden oft als „schemafrei“ bezeichnet, weil im Vorfeld keine Struktur für die Daten festgelegt wird. Dieser Ansatz wird in der Fachwelt

als kritisch angesehen, denn ein Datenmodell hat enormen Einfluss auf die Performance beim Lesen und Schreiben der Daten und beeinflusst zudem die Speichergröße der Datenbank [20]. Man spricht eher von Schemaflexibilität bei NoSQL-Datenbanken [21]. Bei einer NoSQL-Datenbank ist es möglich während des laufenden Betriebs strukturelle Veränderungen an der Datenbank vorzunehmen. Beispielsweise können neue Attribute hinzugefügt oder verändert werden. Daher eignen sich NoSQL Datenbanken sehr gut für die agile Entwicklung, wo sich das Schema häufig durch neue Anforderungen verändert. Bekannte NoSQL-Datenbanken sind Google Bigtable (Column Store), Amazon DynamoDB (Key Value Store), Neo4j (Graph Store), MongoDB (Document Store) oder CouchDB (Document Store). Weitere NoSQL-Datenbanken können auf der Webseite¹ eingesehen werden.

3.2 Skalierbarkeit durch Distributed Hash Table

Die horizontale Skalierbarkeit wird im Wesentlichen durch verteilte Hashtabellen ermöglicht. Bei der Verteilung von Daten muss sichergestellt werden, dass die Daten permanent gespeichert sind und dass das Auffinden der Daten möglichst performant abläuft bzw. ein Information Retrieval der Datenbank muss möglich sein. Das ist durch Hashing der Daten möglich, denn beim Hashing wird kein Zentralrechner benötigt, der die Daten verwaltet, denn das würde dazu führen, dass der zentrale Knoten als „Bottleneck“ (Engpass) gilt, da fast alle Anfragen an den zentrale Knoten geschickt werden. Ein weiteres Problem beim Ansatz eines zentralen Knotens ist, dass der Knoten als „Single Point of Failure“ gilt, da wenn der Masterknoten ausfällt, keine weiteren Anfragen mehr bearbeitet werden. Beim Hashing hingegen wird nur die Hashfunktion benötigt, um die Daten aufzufinden. Jeder Knoten erhält die gleiche Hashfunktion. Ein Problem beim Hashing ist, wenn eine Hashfunktion gewählt wird, dass die Daten nicht gleichmäßig verteilt werden. Hierdurch kann es zu Kollisionen kommen, d.h. dass ein Knoten überfüllt ist bzw. ein Überlauf entsteht und keine weiteren Daten mehr gespeichert werden können. Das Problem lässt sich mit einer guten Hashfunktion, die die Daten gleichmäßig verteilt, lösen. Bei einer Hashfunktion wird häufig eine Modulo Funktion gewählt [22]. Wenn ein Start-up Unternehmen am Anfang einen kleinen Modulo-Wert für die Hashfunktion wählt, kann es dazu kommen, dass die Anzahl an Buckets bzw. Knoten nicht mehr ausreicht, um die Anfragen performant zu bearbeiten. Dies ist der Fall, wenn das Start-up Unternehmen hohe Wachstum generiert und immer mehr Anfragen bearbeiten muss, wie das beispielsweise bei Google, Facebook oder Amazon der Fall war. Das Problem führt dazu, dass mehrere Rechner benötigt werden, um die Anfragelast zu bewältigen. Die vorher verwendete Hashfunktion würde nicht mehr funktionieren, da nun neue Rechner hinzugefügt wurden. Es müsste eine neue Hashfunktion verwendet werden, d.h. alle Daten, die verteilt vorliegen, müssten neu indexiert werden und das bedeutet, dass keine weitere Anfrage mehr bearbeitet werden kann, solange die Indexierung nicht abgeschlossen ist.

Dieses Problem lässt sich durch eine Hashfunktion mit großem Modulo Wert lösen. Dadurch sind viele Buckets bzw. Knoten vorhanden. Es werden nur diejenigen Knoten mit einem

¹<https://hostingdata.co.uk/nosql-database/>

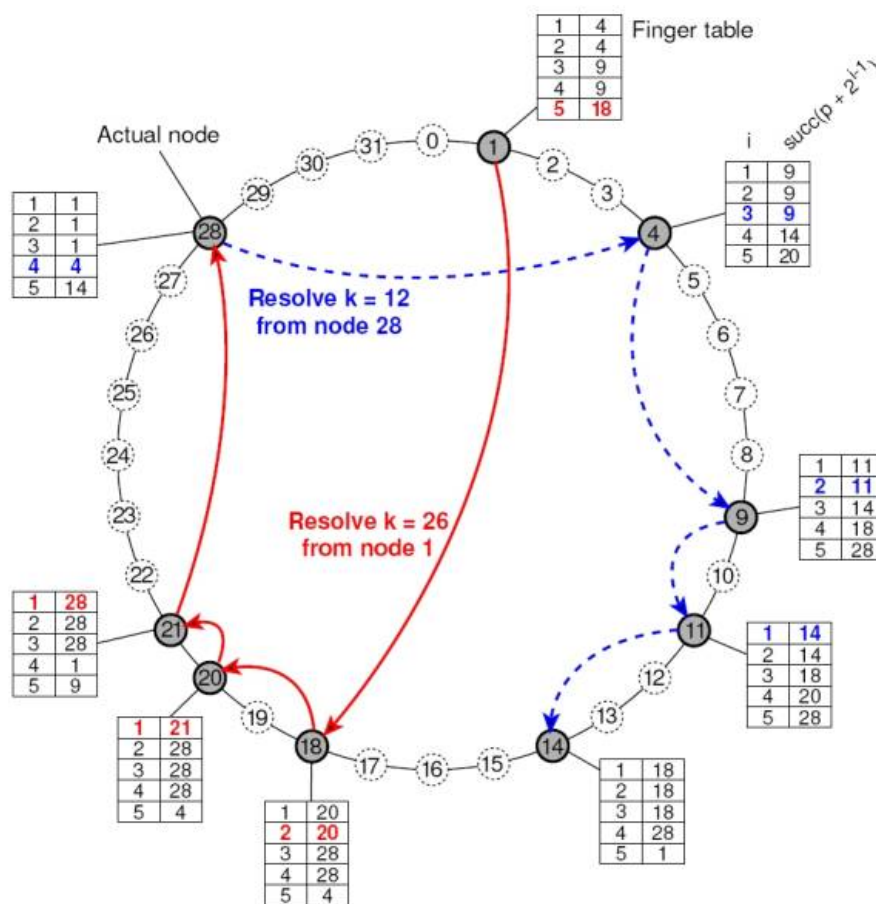


Abbildung 3.1: Beispiel: Chord-System

(Quelle: csis.pace.edu, 2007)

Beim Chord-Algorithmus bekommt jeder Rechner im Netzwerk einen Bezeichner aus einem n -Bit-Bezeichnerraum. Die Server werden durch einen Bezeichner mit einer Hashfunktion abgebildet. Die Buckets werden als Ring dargestellt. Jeder Knoten mit einem Rechner hält eine sogenannte Fingertabelle. In der 1. Spalte der Fingertabelle befindet sich eine

Nummerierung der Knoten. Es gibt so viele Nummerierung wie 2^n Knoten. Im Beispiel sind 32 Knoten also $2^n = 32$:

$$2^n = 32 / * \log() \iff \log(2^n) = \log(32) \iff n = \log(32)/\log(2) \iff n = 5 \quad (3.1a)$$

Die Nummerierung geht somit bis 5 und wird als i bezeichnet. In der 2. Spalte befindet sich jeweils der Bezeichner der Nachbarn, bei denen ein Rechner vorliegt. Der Abstand der Nachbarschaft wird bei jeder Iteration von i um 1 verdoppelt und lässt sich folgendermaßen berechnen:

$$Nachfolger(p + 2^{i-1}) \quad (3.2)$$

Dabei ist p die aktuelle Position des Bezeichners (Knoten). Es folgt ein Beispiel anhand des 1. Bezeichners:

$$i = 1$$

$$Nachfolger(1 + 2^{1-1}) = Nachfolger(2) = 4$$

$$i = 2$$

$$Nachfolger(1 + 2^{2-1}) = Nachfolger(3) = 4$$

$$i = 3$$

$$Nachfolger(1 + 2^{3-1}) = Nachfolger(5) = 9$$

$$i = 4$$

$$Nachfolger(1 + 2^{4-1}) = Nachfolger(9) = 9$$

$$i = 5$$

$$Nachfolger(1 + 2^{5-1}) = Nachfolger(17) = 18$$

Der Abstand wird bei jeder Iteration verdoppelt. Nun erfolgt eine Anfrage vom Bezeichner 28 (gestrichelte blaue Linie) an den Bezeichner 12. Der Wert 12 wird ermittelt, indem die gesuchten Daten in eine Zahl umgewandelt werden, durch beispielsweise Umwandlung der Zeichen in ASCII-Werte und davon die Summe bildet oder durch ein anderes Verfahren. Diese Zahl wird dann in eine Hashfunktion eingesetzt und ausgerechnet. Zuerst schaut der Knoten 28 in seine eigene Fingertabelle und vergleicht die Werte mit den Werten der 2. Spalte. 12 ist kleiner als 14 aber größer als 4, deshalb wird zum 4. Knoten gesprungen. Beim 4. Knoten wird der gesuchte Wert wieder mit der Fingertabelle verglichen und der gesuchte Wert ist kleiner als 14 aber größer als 9, deshalb wird auf den 9. Knoten gesprungen. Im 9. Knoten ist 12 größer als 11 aber kleiner als 14, deshalb wird zum 11. Knoten gesprungen und nochmals verglichen: 12 ist kleiner als 14. Nun stellt sich heraus, dass der Knoten 12 nicht existiert, dann wird der Nachfolger von 12 gesucht und das ist dann der Knoten 14 und die Anfrage wird auf den Knoten 14 ausgeführt. Wenn ein neuer Rechner hinzugefügt wird, müssen die Fingertabellen aktualisiert werden. Es werden maximal n Sprünge benötigt, um einen Knoten zu finden, bei dem auch ein Rechner vorliegt. Aufgrund, dass die maximalen Sprünge auf n begrenzt sind, ist der Algorithmus sehr performant [20]. Hier wird lediglich eine Vereinfachung des Chord-Algorithmus dargestellt. Der Algorithmus ist in der tatsächlichen Anwendung deutlich komplexer.

3.3 CAP-Theorem

Das CAP-Theorem ist ein Paradigma, welches besagt, dass in einem verteilten System nur maximal zwei der drei CAP-Eigenschaften erfüllt werden können [18, S. 196] (siehe Abb. 3.2).

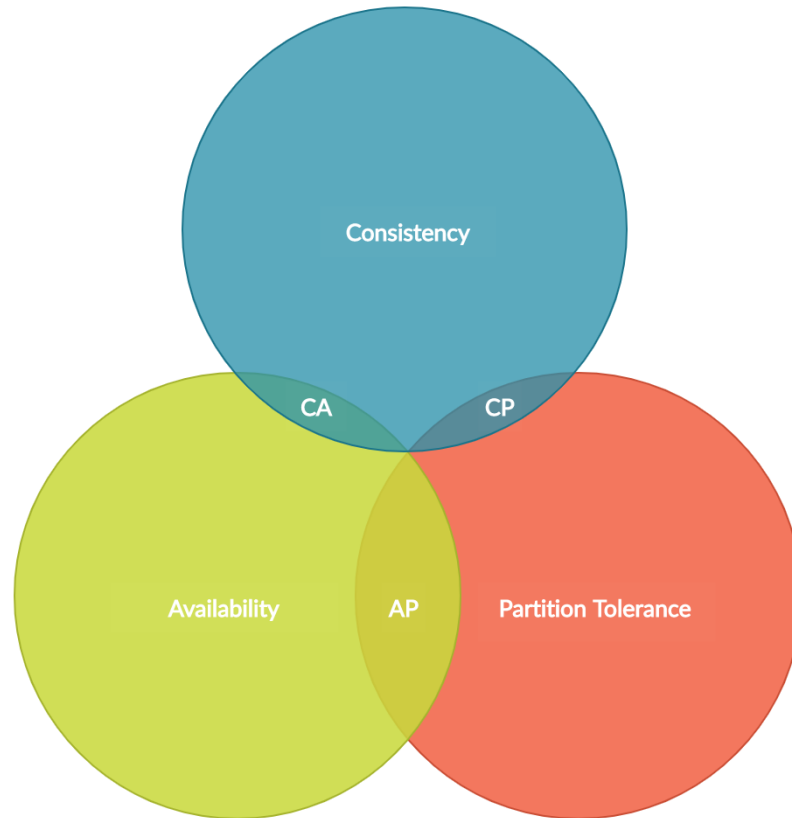


Abbildung 3.2: CAP-Theorem

(Quelle: Angelehnt an Brewer, 2000 - Towards robust distributed systems)

- **Konsistenz** (Consistency) bedeutet, dass nach einem Commit die Daten in einen korrekten Zustand überführt werden.
- **Verfügbarkeit** (Availability) bedeutet, dass die Knoten jederzeit erreichbar und die Daten jederzeit abrufbar sind.
- **Partitionstoleranz** (Partition Tolerance) bedeutet, dass das Gesamtsystem immer noch funktioniert, sogar wenn einige Server ausfallen.

Zuordnung von Datenbanken nach dem CAP-Eigenschaften in einer verteilten Umgebung:

- Konsistenz und Verfügbarkeit (CA): relationale Datenbanken wie beispielsweise Oracle (ACID)

- Verfügbarkeit und Partitionstoleranz (AP): NoSQL-Datenbanken wie beispielsweise Elasticsearch (BASE)
- Konsistenz und Partitionstoleranz (CP): Datenbanken mit hohen Konsistenzanforderungen wie beispielsweise Blockchain

3.3.1 ACID und BASE

Relationale Datenbanken verfolgen das ACID Prinzip:

- **Atomicity** (Atomarität) bedeutet, dass eine Anfrage komplett ausgeführt werden muss. Falls nicht, erfolgt ein Rollback, um auf den vorherigen Zustand der Daten zu kommen.
- **Consistency** (Konsistenz) bedeutet, dass jede erfolgreich ausgeführte Transaktion die Daten in einen korrekten Zustand überführt.
- **Isolation** (Abgrenzung) bedeutet, dass wenn mehrere Transaktion parallel ausgeführt werden, es so betrachtet wird, dass die Transaktionen nacheinander ausgeführt werden.
- **Durability** (Dauerhaftigkeit) bedeutet, dass die Daten langfristig gespeichert werden, so dass die Daten nicht verloren gehen.

Zur Veranschaulichung des ACID-Prinzips soll das folgende Beispiel dienen: Im Online Shop ist nur noch ein Artikel verfügbar. Wenn jetzt mehrere Personen den Artikel gleichzeitig kaufen möchten, muss sichergestellt werden, dass, wenn eine die Bestellung abschließt, kein weiterer Kunde den Artikel kaufen kann.

Wenn die ACID-Prinzipien eingehalten werden, entstehen viele Vorteile bei der Arbeit mit Datenbanken. Der Entwickler oder der Anwender kann davon ausgehen, dass die Daten in einem konsistenten Zustand sind. Sogar bei fehlgeschlagenen Bearbeitungsanfragen an Daten, kann der Anwender davon ausgehen, dass die Datenbank ein Rollback macht und dadurch die Informationen nicht in einem fehlerhaften oder inkonsistenten Zustand vorliegen. Das führt dazu, dass keine manuelle Bereinigung von Fehlern getätigt werden muss [23].

Bei NoSQL-Datenbanken gilt das Base-Prinzip:

- **BA**: *Basically Available* bedeutet, dass das System so gut wie immer verfügbar ist, auch im Falle von Teilsystemausfällen.
- **S**: *Soft state* ist ein Zustand, wenn Replika (Duplikate von Daten) sich auf einem Knoten befinden, auf dem eine vordefinierte Frist existiert. Erfolgt kein Update der Daten innerhalb der Frist, sind die Daten in einem „Soft State“ bzw. veralteten Zustand. Jedoch können trotzdem Anfragen bearbeitet werden und die Eigenschaft

„Verfügbarkeit“ ist somit gegeben. Nach Aktualisierung der Daten befinden sich die Daten in einem „Hard State“ [24].

- **E:** *Eventual Consistency* bedeutet, dass die Daten für eine bestimmte Zeit inkonsistent sind und die Daten erst nach Synchronisierung in verteilte Umgebungen wieder in einen konsistenten Zustand überführt werden [8, S. 7].

BASE soll den Gegensatz zu ACID darstellen. Bei BASE wird die absolute Konsistenz der Daten aufgegeben, um die Eigenschaft „Verfügbarkeit“ zu erfüllen. Beim BASE-Prinzip ist die Verfügbarkeit wichtiger als die Konsistenz der Daten, sodass das System bei einem etwaigen Ausfall von Teilsystemen weiterhin Anfragen bearbeiten kann [25].

Im nächsten Kapitel werden Grundlagen in Elasticsearch vermittelt, um in späteren Kapiteln die Implementierung nachvollziehen zu können.

4 Einführung in Elasticsearch

Elasticsearch ist eine performante Open Source verteilte RESTful Suchmaschine, die 2010 erschienen ist und von der Firma Elastic NV entwickelt wurde. Elasticsearch basiert auf einer Java-basierten Programmbibliothek, welche als Lucene bezeichnet wird und von der Apache Software Foundation entwickelt wurde, mit dem Ziel eine Volltextsuchfunktion anzubieten. Die Suchperformance von Lucene lässt sich sehr gut am Beispiel von Twitter zeigen. So konnte Twitter durch Integration der Lucene-Bibliothek seine Antwortzeit um das Dreifache verkürzen [26]. Die Volltextsuchmaschine Elasticsearch ist bei vielen Unternehmen sehr beliebt. Es nutzen aktuell mehr als 3.000 Unternehmen [27] die Suchmaschine und die Suchmaschine wurde seit der Veröffentlichung mehr als 350 Million Mal heruntergeladen [28]. Dazu zählen bekannte Unternehmen wie beispielsweise Wikipedia, Netflix, Uber, Stack Overflow, VW, Audi oder eBay [27]. Elasticsearch befindet auf der Webseite db-engines.com unter der Rubrik Volltextsuche seit Januar 2016 auf Platz 1 [29]. Die Bewertungskriterien des Rankings werden in [30] ausführlich beschrieben.

4.1 Bezug zu relationalen Datenbanken

Aufgrund des hohen Bekanntheitsgrads relationaler Datenbanken wird ein Konzeptvergleich zur Elasticsearch vorgenommen (siehe Abb. 8.1). In einem Cluster können sich ein oder mehrere Indizes befinden. Ein Index in Elasticsearch ist ähnlich wie eine Tabelle in einer relationalen Datenbank. In einem Index befinden sich Dokumente mit einem ähnlichen Schema. Es handelt sich hierbei um keinen klassischen Index, wie es in einer relationalen Datenbank bekannt ist, um SQL-Anfragen performanter auszuführen. In relationalen Datenbanken werden Attribute in Spalten eingetragen und in Elasticsearch werden Attribute in Felder eingetragen. Ein Datensatz (Zeile) in relationalen Datenbanken wird in Elasticsearch als ein Dokument abgebildet [31].

Tabelle 4.1: Konzeptvergleich zwischen SQL und Elasticsearch

SQL-Datenbank	Elasticsearch
Datenbank	Cluster
Tabelle	Indizes
Spalte	Felder
Zeile	Dokumente

4.2 Einstellungen (Settings) und Mapping

In den Settings werden wichtige Einstellungen für einen Index im Cluster vorgenommen und im Mapping wird das Schema festgelegt.

4.2.1 Shards und Replicas

In den Settings wird die Anzahl an Shards und Replicas festgelegt. Wenn in einem Index zu viele Daten vorliegen, kann Sharding genutzt werden, um einen Index in mehrere Shards aufzuspalten. Beim Anlegen eines Elasticsearch-Index werden 5 Primary-Shards als default angelegt und jeder Shard verfügt über einen eigenen Lucene-Index (Java Library: Lucene). Nach dem Anlegen eines Index mit der vordefinierten Anzahl an Shards, kann später keine Änderung mehr vorgenommen werden. Deshalb ist es wichtig, schon bei der Planung die Speichergröße der Daten in naher Zukunft einzuschätzen. Denn je mehr Shards ein Index hat, desto besser lässt er sich auf mehrere Knoten verteilen. Durch Verteilung von Primary-Shards auf mehrere Knoten kann die Indizierungsgeschwindigkeit erhöht werden, da insgesamt mehr Ressourcen (in Form von Rechenleistung) vorliegen. Es macht jedoch wenig Sinn bei wenigen Daten einen Index mit mehreren Shards anzulegen, da dadurch Vorteile bei der Performance und bei der Exaktheit der Suche verloren gehen können [19, S. 103 -105]. In den Settings werden auch die Anzahl an Replicas festgelegt. Replicas sind im Wesentlichen nichts anders als Kopien von Primary-Shards, die auch nach dem Anlegen eines Index konfigurierbar bzw. veränderbar sind. Das Primärziel von Replicas ist die CAP-Eigenschaft „Verfügbarkeit“ zu garantieren bzw. die Ausfallsicherheit zu erhöhen, um Suchanfragen jederzeit beantworten zu können, auch wenn einige Server ausgefallen sind. Lesezugriffsanfragen können von Primär-Shards und Replicas bearbeitet werden, aber Schreibzugriffsanfragen können nur von Primary-Shards bearbeitet werden. Nachdem die Daten im Primary-Shard bearbeitet wurden, werden die Daten an die Replikas weitergeleitet. Falls Suchanfragen erfolgen, bevor die Daten an die Replikas weitergeleitet werden, sind Daten nicht mehr aktuell, d.h. es liegt eine schwache Konsistenz vor und die Daten sind erst nach einer bestimmten Zeit wieder konsistent [19, S. 106],

4.2.2 Analyzer in Elasticsearch

In den Einstellungen wird die Wahl der Analyzer für einen Index bestimmt, die für Zerlegung von Textdaten in Token zuständig sind. Elasticsearch bietet Analyzer in 35 verschiedenen Sprachen an, um Wörter in Grundformen (wird auch als Lemmatisierung bezeichnet) zu bilden oder Wörter auf ihrem eigenen Wortstamm (wird auch als Stemming bezeichnet) zu reduzieren [32]. Zur Veranschaulichung folgen nun Beispiele:

- Beispiel für Lemmatisierung: Gab es vorher die Wortformen „traf“, „treffe“, „trifft“ und „treffen“ jeweils einmal im Korpus, so gibt es nach der Lemmatisierung nur noch das Lemma „treffen“.
- Beispiel für Stemming: Das Wort „geschlafen“ wird nach Stemming in das Wort „schlafen“ transformiert.

Beim Analyzer in Elasticsearch lässt sich der Prozess in die folgenden drei Stufen aufteilen:

- Der **Char-Filter** ist für die Entfernung von Metadaten in Textdaten zuständig.
- Der **Tokenizer** ist für die Zerlegung von Texten in Wörter zuständig. Elasticsearch bietet 3 Arten von Tokenizer an. Der Word-oriented-Tokenizer zerlegt den Text in Wörter. Der Partial-Word-Tokenizer zerlegt Wörter in Buchstaben, um Autocomplete bei einer Suchanfrage zu ermöglichen. Der Structured-text-Tokenizer zerlegt Texte in Wörter nach bestimmten Muster.
- Der **Token-Filter** in Elasticsearch bietet viele Verarbeitungsmöglichkeiten für die einzelnen Wörter wie beispielsweise Synonyme, Stoppwörter oder Umwandlung der Wörter in Kleinbuchstaben etc. [33].

In der Abbildung 4.1 wird der Prozess anhand eines Beispiels veranschaulicht. In der Abbildung ist zu erkennen, dass im Char-Filter der HTTP-Code entfernt wird. Beim Tokenizer wird der Text in einzelne Wörter zerlegt und im Token-Filter werden Stoppwörter und Sonderzeichen entfernt und Wörter in Kleinbuchstaben umgewandelt.

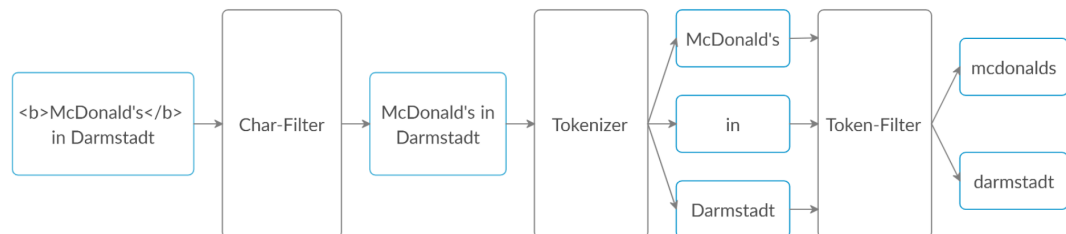


Abbildung 4.1: Beispiel: Analyzer-Prozess

(Quelle: In Anlehnung an Hopf, 2016, S. 38)

Im Listing 4.1 werden das Setting und Mapping in JSON-Format dargestellt. Wie zu erkennen ist, wird die Anzahl an Shards und Replikas festgelegt. Daraufhin folgt die („analysis“). Diese beinhaltet alle verwendete Token-Filter für einen Index. Beim („analyzer“) wird daraufhin der Tokenizer festgelegt und zudem, welche Token-Filter der Analyzer nutzen soll.

```

1 PUT myIndex {
2   "settings": {
3     "index": {"number_of_shards":3, "number_of_replicas":2},
4     "analysis": {
5       "filter": {
6         "german_stop": {"type":"stop", "stopwords": "_german_"},
7         "german_keywords": {"type":"keyword_marker", "keywords":["Beispiel"]},

```

```

8     "german_stemmer": {"type": "stemmer", "language": "light_german"}
9   },
10  "analyzer": {
11    "rebuilt_german": {
12      "tokenizer": "standard",
13      "filter": [
14        "lowercase",
15        "german_stop",
16        "german_keywords",
17        "german_normalization",
18        "german_stemmer"]
19    }
20  }
21 }
22 }

```

Listing 4.1: Beispiel: Setting und Mapping in Elasticsearch

4.2.3 Mapping

Im Mapping werden die Dokumentenstruktur und die Datentypen sowie die Wahl der Analyzer für die einzelnen Felder (Attribute) in einem Index festgelegt. Als Default wird für die Felder ein „Standard“-Analyzer verwendet. Der „Standard“-Analyzer basiert auf dem Unicode-Segmentierungsalgorithmus, der für viele Sprachen gut funktioniert [34]. Der „Standard“-Analyzer wird anhand eines Beispiels (siehe Lis. 4.2) erklärt. Mit „POST _analyze“ lassen sich Analyzer testen ohne, dass ein Elasticsearch-Index angelegt werden muss.

```

1 POST _analyze
2 {
3   "analyzer": "standard",
4   "text": "Hochschule Darmstadt, Schöfferstraße 3, 64295 Darmstadt"
5 }

```

Listing 4.2: Beispiel: „Standard“-Analyzer

Nach Anwendung des „Standard“-Analyzers wird der Text in Token zerlegt. Wie zu erkennen wurden Kommas entfernt und Buchstaben, die großgeschrieben waren, wurden in Kleinbuchstaben umgewandelt (Lis. 4.3).

```

1 [hochschule, darmstadt, schöfferstraße, 3, 64295, darmstadt]

```

Listing 4.3: Beispiel: Ergebnis „Standard“-Analyzer

Wenn kein Mapping beim Importieren von Daten in Elasticsearch übergeben wurde, so wird automatisch ein Mapping erstellt, welches Elasticsearch als dynamisches Mapping bezeichnet. Es wird nicht empfohlen ein dynamisches Mapping zu verwenden, da ohne eine Anpassung der Datentypen und Analyzer keine guten Suchergebnisse erzielt werden können. Denn ein Attribut kann mehrere Felder mit unterschiedlichen Datentypen und Analyzern belegen, um verschiedene Anfragen zu bedienen. Beispielsweise muss bei Aggregationsanfragen von

Kategorien der Datentyp „keywords“ oder „boolean“ genutzt werden, ansonsten würde die Aggregation nicht funktionieren. Bei einer Freitextsuchanfrage sind beide Datentypen ungeeignet. Für die Freitextsuche sollte der Datentyp „text“ verwendet werden. Wenn ein Feld im Mapping angelegt wird, kann es nicht mehr verändert werden und muss bei Veränderung gelöscht und wieder neu erstellt werden [19, S. 40]. Nähere Informationen über die Datentypen werden im nächsten Abschnitt vermittelt. Die Wahl der Analyzer hat große Auswirkung auf die Qualität der Suchergebnisse. Die passenden Analyzer für die jeweiligen Felder zu finden, erfordert viel Zeit, da viele Experimente vorgenommen und Suchergebnisse überprüft werden müssen [19, S. 38]. Analyzer können auch bei einer Suchanfrage übergeben werden. Dies wird auch als „search-analyzer“ bezeichnet. Dies dient dazu, dass die Daten in den Feldern auch mit den Daten der Abfrage übereinstimmen [35].

Zu Veranschaulichung wird ein Beispiel für das Mapping im JSON-Format dargestellt (siehe Listing 4.4). Das Feld („name“) wird einmal als Datentyp „text“ dargestellt und ein „stop_name_analyzer“ verwendet. Mit der Einstellung („fielddata“: true) können, wie oben erwähnt mehrere Datentypen für ein Attribut erstellt werden. In diesem Beispiel wurde ein zusätzlicher Feld mit dem Datentyp „keyword“ deklariert, um eine Aggregationsanfrage durchführen zu können. Dabei wird ein Token mit mehr als 256 Zeichen nicht indexiert, d.h. bei der Aggregation wird der jeweilige Text/Begriff nicht berücksichtigt.

```
1  "mappings" : {  
2    "properties" : {  
3      "name" : {"type" : "text", "analyzer" : "stop_name_analyzer",  
4        "fielddata": true,  
5        "fields" : {"keyword" : {"type" : "keyword", "ignore_above" : 256}}}  
6    }  
7  }
```

Listing 4.4: Beispiel: Mapping in Elasticsearch

4.2.4 Überblick über Datentypen

Elasticsearch bietet alle gängigen Datentypen an, wie „binary“, „boolean“, „keywords“, „numbers“, „Dates“ und für die Textverarbeitung gibt es Datentypen wie „text“, um unstrukturierte Texte zu verarbeiten, „completion“ für die Vervollständigung von Wörtern usw. Auch für die Bearbeitung von Geodaten gibt es Datentypen wie „geo_point“ die geografische Koordinaten abbilden und „geo_shape“ wird verwendet, um Flächen wie Polygon abzubilden. Es werden noch weitere komplexe Datentypen von Elasticsearch bereitgestellt [36].

4.3 Indexierung in Elasticsearch

In diesen Abschnitt wird die Indexierung in Elasticsearch behandelt. Nach der Zerlegung der Sätze in Token verwendet Elasticsearch eine invertierte Indexierung (siehe Abb. 4.2). Bei der invertierten Indextabelle liegt eine alphabetische Sortierung vor. Für jeden Term, der in

einem Dokument vorkommt und nicht vom Analyzer ausgeschlossen wurde, wird indexiert, indem jeder Term eine ID zugeordnet bekommt. In der Spalte wird (Dokument;Wort) jeweils die Position des Dokuments und die Position des Wortes im Dokument dargestellt. Beispiel: „Menschen“ (1;4) befindet sich im 1. Dokument an der 4. Stelle des Satzes. Eine invertierte Indexierung ist (zeitlich) deutlich aufwendiger als eine Indexierung mit einem B-Baum in einer relationalen Datenbank und benötigt auch viel mehr Speicherplatz. Dafür ist eine invertierte Indexierung sehr gut geeignet für eine Volltextsuche.

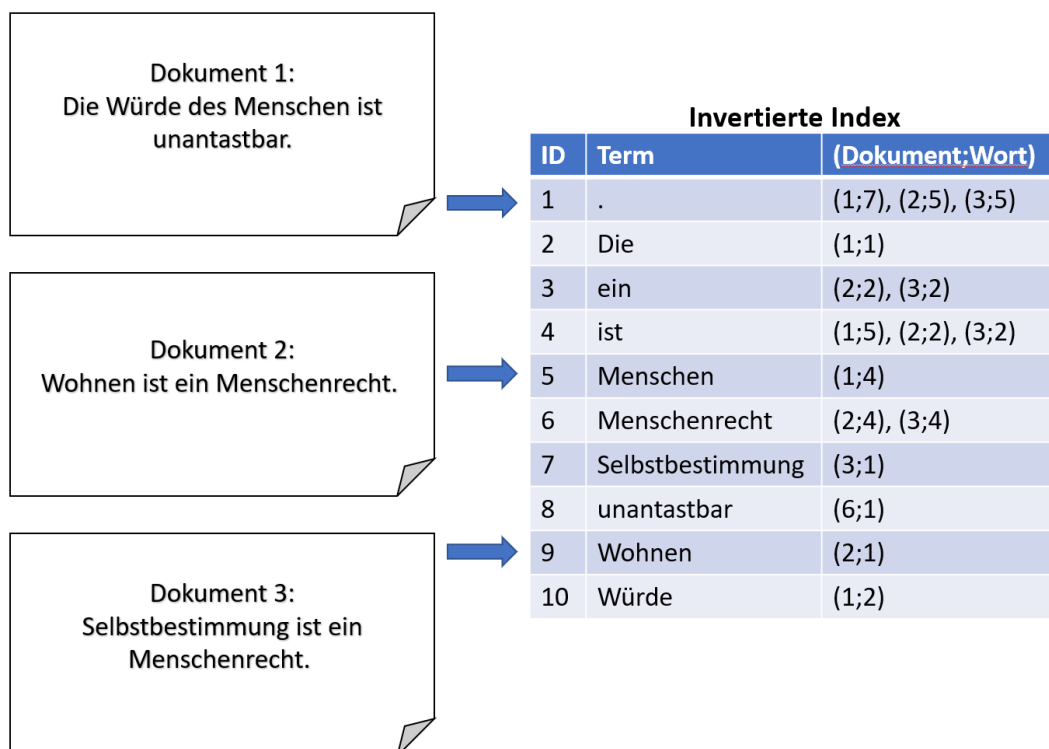


Abbildung 4.2: Beispiel: Invertierte Indexierung

4.3.1 Volltextsuche

Defintion: Bei einer Volltextsuche wird ein Text in eine Suchmaschine eingegeben, um eine Trefferliste zu erstellen. Eine Volltextsuche ist in der Lage, mit falschen Schreibweisen, Synonymen sowie unterschiedlichen grammatikalischen Formen (z.B. Singular-Plural-Abweichungen) umzugehen und ist in der Lage, Begriffe in Texten zu finden [37]. Der Vorteil einer Freitextsuche, im Vergleich zu einer vorgegebenen Auswahlmöglichkeit, ist, dass der Nutzer anhand einer ungenauen Beschreibung ein geeignetes Suchergebnis erhält, welches es dem Nutzer ermöglicht, den gesuchten Ort schnellstmöglich und mit wenig Aufwand zu finden. Außerdem kann auf diese Weise beispielsweise ein spezifisches Restaurant im Vergleich zu irgendeinem rollstuhlgeeigneten Restaurant einfacher gesucht werden. Der Nutzer hat somit bei einer Freitextsuche grundsätzlich jegliche Freiheiten bei der Eingabe, jedoch kann nicht garantiert werden, dass auch ein gewünschtes Ergebnis geliefert wird.

4.3.2 Relevanz der Dokumente (Score)

Wenn keine Sortierung oder Filterung der Attribute vorliegt, werden die Suchergebnisse nach dem Score (Relevanz der Dokumente) sortiert. Der Score ist ein Messwert, der aussagt wie gut eine Texteingabe mit den Inhalt eines gesuchten Dokuments übereinstimmt [38]. Zur Berechnung der Relevanz der Dokumente wird das Vector-Space-Modell verwendet. Der Inhalt eines Feldes wird als Vektor in einem n-dimensionalen Raum betrachtet. Die Ähnlichkeit eines Dokuments und einer Anfrage wird durch Distanz der beiden Vektoren bestimmt. In Elasticsearch wird das Tf-idf-Maß verwendet [19, S. 66]. Die Grundidee des Tf-idf-Maßes ist es, wenn ein Suchbegriff häufiger in einem Dokument auftaucht und im der Gesamtheit der Dokumente relativ selten vorkommt, dann wird das Dokument ein als Treffer gelten. Der TF-idf lässt sich mit folgender Formel berechnen:

$$w_{i,j} = tf_{i,j} * \log\left(\frac{\#d}{\sum_{j=1}^{\#d} \#t_i}\right) \quad (4.1a)$$

i : ist die ID bzw. die Position eines Terms (Tokens)

j : ist die Position eines Dokuments

w : kombiniertes Gewicht eines Wortes (Terms, Token) in einem Dokument

tf : absolute Häufigkeit eines Wortes (Terms) in einem Dokument

$\#d$: Anzahl der Dokumente im Index

$\#t_i$: Anzahl eines Tokens in einem Dokument

$idf = \log\left(\frac{\#d}{\sum_{i=1}^{\#d} \#t_i}\right)$: inverse Häufigkeit von von Anzahl Dokumente in einem Index geteilt durch

Anzahl eines Tokens in den gesamten Dokumenten (Index)

Die Formel wird mit der Anfrage: „Menschenrecht Wohnen“ in den drei vorigen Dokumente verdeutlicht (siehe Abbildung 4.3 - Ergebnisse, wurden auf vier Stellen nach dem Komma gerundet):

i	(Dokument;Wort)	M	r	idf	m	d ₁	d ₂	d ₃
1	(1;7), (2;5), (3;5)	0	1	0	0	0	0	0
2	(1;1)	0	3	0,4771	0	0,4771	0	0
3	(2;2), (3;2)	0	1,5	0,1761	0	0	0,1761	0,1761
4	(1;5), (2;2), (3;2)	0	1	0	0	0	0	0
5	(1;4)	0	3	0,4771	0	0,4771	0	0
6	(2;4), (3;4)	1	1,5	0,1761	0,1761	0	0,1761	0,1761
7	(3;1)	0	3	0,4771	0	0	0	0,4771
8	(1;6)	0	3	0,4771	0	0,4771	0	0
9	(2;3)	1	3	0,4771	0,4771	0	0,4771	0
10	(2;1)	0	3	0,4771	0	0	0,4771	0

Abbildung 4.3: Beispiel: Die Ergebnisse für die Anfrage: „Menschenrecht Wohnen“

M_i : gibt die Häufigkeit eines Treffers (match) an

$r_i = \frac{\#d}{\sum_{i=1}^{\#d} \#t_i}$: Anzahl alle Dokumente geteilt durch die Anzahl eines Tokens, die in den

Dokumenten (gesamter Index) vorkommen

$idf_i = \log(r_i)$

$m_i = \log(M_i) * M_i$

$d_{i,j} = \#d_j * idf_i$: Anzahl eines Tokens, die in einem Dokument vorkommen multipliziert mit dem idf

Nun werden die Vektorlänge für jedes Dokument und für M berechnet (Multiplikation mit 0 werden ausgelassen):

$$|d_1| = \sqrt{0.4771^2 + 0.4771^2 + 0.4771^2} = 0.8264$$

$$|d_2| = \sqrt{0.1761^2 + 0.1761^2 + 0.4771^2 + 0.4771^2} = 0.5382$$

$$|d_3| = \sqrt{0.1761^2 + 0.1761^2} = 0.2490$$

$$|m| = \sqrt{0.1761^2 + 0.4771^2} = 0.5086$$

Nun werden die Skalarprodukt für jedes Dokument und für M berechnet:

$$\langle m, d_1 \rangle = 0 * 0.4771 + 0 * 0.4771 + 0 * 0.4771 = 0$$

$$\langle m, d_2 \rangle = 0 * 0.1761 + 0.1761 * 0.1761 + 0.4771 * 0.4771 = 0.2586$$

$$\langle m, d_3 \rangle = 0 * 0.1761 + 0.1761 * 0.1761 + 0 * 0.1761 = 0.0310$$

Für das Ähnlichkeitsmaß wird das Kosinusmaß [39] gewählt und für jedes Dokument berechnet:

$$\begin{aligned} \cos(d_1) &= \frac{\langle m, d_1 \rangle}{|m| * |d_1|} = \frac{0}{0.5086 * 0.8264} = 0 \\ \cos(d_2) &= \frac{\langle m, d_2 \rangle}{|m| * |d_2|} = \frac{0.2586}{0.5086 * 0.5382} = 0.9447 \\ \cos(d_3) &= \frac{\langle m, d_3 \rangle}{|m| * |d_3|} = \frac{0.0310}{0.5086 * 0.2490} = 0.2448 \end{aligned}$$

Die Ergebnisse der Scores von den Dokumenten werden nun sortiert:

1. d_2 : 0.9447
2. d_3 : 0.2448
3. d_1 : 0

Es wurde nach „Menschenrecht Wohnen“ gesucht und in Dokument 2 sind beide Wörter vorhanden. Deshalb befindet sie sich auf Rang 1. Im Dokument 3 ist von den beiden gesuchten Wörtern das Wort „Menschenrecht“ im Dokument vorhanden und somit auf Platz 2. Im Dokument sind die beiden gesuchten Wörter nicht vorhanden und somit auf Platz 3.

In Elasticsearch wird ein normalisiertes TF*IDF-Maß verwendet, um die Scores untereinander besser vergleichen zu können. Der Faktor Feldlänge (Vorkommen in kürzeren Feldern werden als wichtiger betrachtet) wird noch beim Score berücksichtigt und durch Boosting ist es möglich, bestimmte Terme (Token) stärker zu gewichten und somit den Score zu manipulieren [19, S. 67].

4.4 Anfragen an den Elasticsearch-Server

Anfragen (requests) erfolgen über einen Elasticsearch-Client oder über eine REST API an den Elasticsearch-Server.

4.4.1 Elasticsearch-Client

Die Firma Elastic NV bietet Elasticsearch Client in verschiedenen Programmiersprachen an, wie beispielsweise Java, JavaScript, Python usw. Zusätzlich stellt die Elasticsearch Community weitere Elasticsearch-Clients in anderen Programmiersprachen zur Verfügung, die von Elastic NV nicht angeboten werden [40]. Beim Client ist es wichtig, dass die Entwicklung des Clients unabhängig vom Server erfolgt, da beispielsweise eine Änderung der Datenstruktur bei einer mobilen Anwendung, ohne dass die Datenstruktur auf Serverseite verändert werden muss, erfolgt, sodass jede Anwendung unabhängig von anderen wachsen kann [41].

4.4.2 REST API

Definition: Das REST API Design wurde 2000 von Roy Fielding in seiner Doktorarbeit definiert. REST API ist eine Programmierschnittstelle, die eine Kommunikation via HTTP (über Internet) von „Machine to Machine“ ermöglicht. Die REST API kann als Format JSON, XML oder in anderen Datenformate genutzt werden und ist auf kein Datenauschformat beschränkt. REST APIs sind zustandlos und synchron, d.h. alle Aufrufe erfolgen voneinander unabhängig und jeder Aufruf enthält alle Daten die angefragt werden. Eine REST API verlässt sich nicht darauf, dass die Daten auf dem Server oder auf einer Sitzung gespeichert sind, um so zu bestimmen, was mit einem Anruf zu tun ist, sondern verlässt sich auf die Daten, die im Aufruf bereitgestellt werden. Identifikationen erfolgen bei Anfragen über API-Schlüssel, Zugriffstoken, Benutzer-ID etc. Um „eine Anwendung so skalierbar wie möglich zu halten, erfordert eine RESTful-API, dass jeder Zustand auf dem Client gespeichert wird - nicht auf dem Server“ [41].

Elasticsearch nutzt für REST API als Datenauschformat JSON und unterstützt folgende HTTP-Methoden in Elasticsearch:

- **GET:** Abrufen von Dokumenten, Searchtemplates oder Testen von Analyzern etc. in Elasticsearch.
- **POST:** Update oder Hinzufügen von Searchtemplates, Indizes (mit Settings und Mapping) oder Update einzelner Felder in einem Dokument etc. in Elasticsearch.
- **PUT:** Erstellung oder Konfiguration des Elasticsearch-Index (Beispiel: Bestimmung Anzahl an Replikation) oder Update eines kompletten Dokument.
- **DELETE:** Löscht Dokumente, Indizes oder Searchtemplates etc. in Elasticsearch.

4.4.3 Search-Templates

Bei der Suche von Dokumenten in Elasticsearch erfolgt jeweils eine Suchanfrage. Die Suchanfrage kann gegebenenfalls auch sehr komplex sein. Um nicht jedes Mal bei der Suche die Suchanfrage neu aufzubauen, können die Suchanfragen in Elasticsearch als Search-Templates abgespeichert werden. Dazu wird die Template-Sprache Mustache genutzt, die in der Lage ist, Suchanfragen und Variablen voneinander zu trennen. Bei einer Suchanfrage muss der Searchtemplate-Name mit den zugehörigen Variablen, die sich im „params“-Block befinden, übergeben werden, um eine Suchanfrage auszuführen (siehe Lis. 4.5) [19, S. 33 - 34]. Search-Templates verhalten sich ähnlich wie ein Funktion- oder Methodenaufruf in einer anderen Programmiersprache.

```
1 POST _scripts/searchtemplate1
2 {
3   "script": {
4     "lang": "mustache",
5     "source": {
6       "query": {"match": {"{{name_field}}": "{{my_value}}"}}
7     }
8   }
9 }
10 }
11
12 GET _search/template
13 {
14   "id": "searchtemplate1",
15   "params": {
16     "name_field": "name",
17     "my_value": "Hochschule Darmstadt",
18   }
19 }
```

Listing 4.5: Beispiel: Search-Template in Elasticsearch

Es wurden grundlegende Konzepte in Elasticsearch vermittelt. Im nächsten Kapitel werden die verwendeten Komponenten beschrieben.

5 Verwendete Technologien und Komponenten

In diesem Kapitel wird auf die genutzten Technologien näher eingegangen; zudem werden gewählte Programmiersprachen für die Realisierung und Umsetzung des Projektes erläutert.

5.1 Verwendete Programmiersprachen

5.1.1 Python

Python ist nach dem PopularitY of Programming Language Index seit Juni 2018 die beliebteste Programmiersprache [42]. Python wurde 1991 von Guido van Rossum veröffentlicht und ist eine interpretierte höhere Programmiersprache [43]. Die Programmiersprache kombiniert Elemente aus der imperativen, objektorientierten und funktionalen Programmierung. Das Ziel war es, mit wenigen Zeichen eine kompakte und flexible Programmiersprache zu entwickeln, um eine hohe Lesbarkeit zu erzielen. Python ist im Vergleich zu der populären Programmiersprache Java für dieselbe Codierung/Programmierung eines um das 3 - 5 fache geringeren Zeichenumfanges [44, S. 2]. Des Weiteren verfügt Python über viele nützliche Bibliotheken, wie zum Beispiel Pandas, um tabellarische Daten zu verarbeiten.

5.1.2 Java

Java ist eine objektbasierte Programmiersprache und wurde von dem Computerhersteller Sun Microsystems entwickelt und im Januar 1996 veröffentlicht. Java ist nach dem PopularitY of Programming Language Index seit Juni 2018 die zweit beliebteste Programmiersprache [42]. Java kann Algorithmen performanter als Python ausführen, da Programmcode in Java Virtual Machine interpretiert wird [45].

5.2 Kibana

Kibana ist ein Tool, um Daten in Elasticsearch zu visualisieren. Weiterhin verfügt Kibana über eine integrierte Entwicklungsumgebung für Query-DSL Anfragen, die einen Debugger und Autovervollständigung anbieten [46]. Dies wiederum hilft bei der Implementierung von komplexen Suchanfragen und ermöglicht es, auf einfache Weise mittels Tests die Qualität der Suchanfragen zu überprüfen. Denn um optimale Suchergebnisse zu erzielen, müssen der Analyzer und der Datentyp optimal auf die Suchanfrage angepasst werden.

5.3 Docker

Docker ist eine Virtualisierungssoftware, die 2013 von der Firma Docker INC entwickelt wurde. Durch Docker-Container ist es möglich, unterschiedliche Softwarekomponenten wie Webserver, Programmiersprachen oder Datenbanken, ohne Overhead, den eine virtuelle Maschine mit sich bringt auszuführen [47, S. 9]. Denn bei einer virtuellen Maschine wird die Hardware emuliert und ein komplettes Betriebssystem aufgespielt und bei Docker-Container wird auf das Betriebssystem verzichtet und stattdessen das Host-Betriebssystem im Kontext genutzt [48]. Das hat zur Folge, dass weniger Ressourcen benötigt werden. Ein weiterer Vorteil von Docker-Container liegt in der Lastenverteilung durch eine horizontale Skalierung. Über Docker-Hosts lassen sich Dienste über mehrere Rechner verteilen, um beispielsweise Massenrequests performant zu beantworten. Dabei müssen die genutzten Softwarekomponenten auch verteilbar sein, wie es bei der NoSQL-Datenbank der Fall ist. Bei einer Anfragelastabnahme kann ein Container bzw. ein Rechner ausgeschaltet werden, um Kosten zu sparen.

5.4 PostgreSQL mit PostGIS

PostgreSQL ist eine relationale Open Source SQL-Datenbank. Mit der Erweiterung von PostGIS kann eine PostgreSQL Datenbank geografische Objekte speichern und verarbeiten. Die aktuelle PostGIS-Version bietet mehr als 1.000 Geooperationen, wie beispielsweise Schnittmengen von Geoobjekten oder Entfernungen von Geoobjekten, an [49]. Der „Simple Feature Access“-Standard wurde komplett in PostGIS implementiert und der Standard SQL/MM Spatial wurde zum Teil in PostGIS eingebunden. In der Abbildung 5.1 handelt es sich um eine vereinfachte Darstellung der Geometrienklasse nach dem „Simple Feature Access“-Standard. In der Abbildung ist zu erkennen, dass jede Subklasse (Point, LineString etc.) von der Klasse Geometry erbt, d.h. alle Subklassen können die Funktion die die Geometry-Klasse bereitstellt, anwenden. Es kann ein Point, LineString oder ein Polygon als Objekt dargestellt werden. In der Klasse der GeometryCollection befinden sich weitere Subklassen, die jeweils eine Ansammlung von Geopunkte, Polygone oder MultiLineString als Objekt darstellen können.

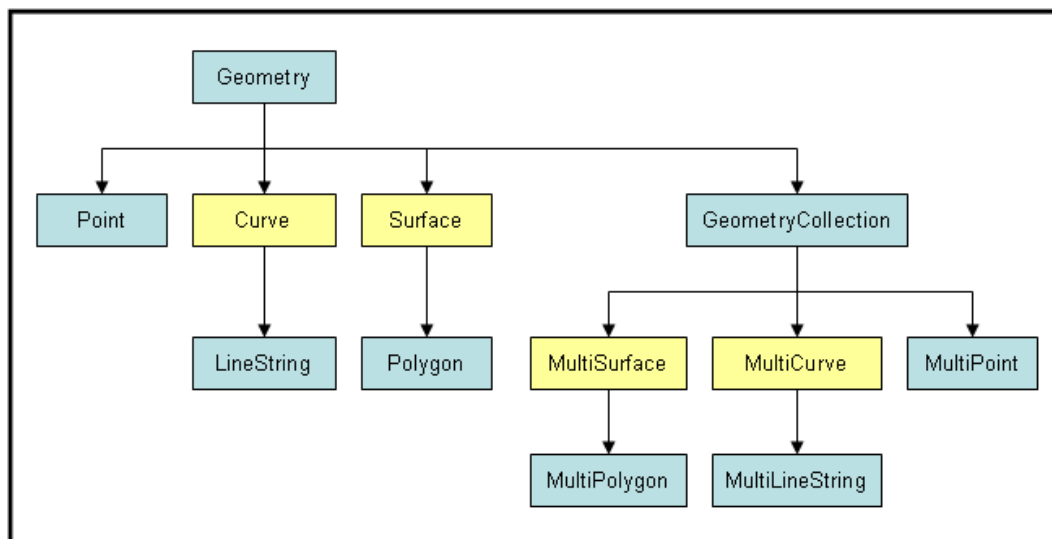


Abbildung 5.1: Geometrienklasse im „Simple Feature Access“-Standard

(Quelle: gsp.humboldt.edu, 2012 - 2015)

Beispiele für die unterschiedlichen Geometrienklassen:

POIs können als Point-Objekt oder als Polygon-Objekt dargestellt werden; in Nominatim werden POIs als Polygone dargestellt und in Photon werden Adressen als Koordinaten dargestellt. Ein LineString kann eine geroutete Strecke abbilden, wie es bei GraphHopper der Fall ist. Deutschland wird als MultiPolygon abgebildet, da Deutschland aus mehreren Polygonen besteht. Verkehrsnetze können als MultiLineString abgebildet werden. Bei einer großen Messe können die Stände als MultiPoint abgebildet werden.

5.5 Cerebro

Cerebro ist ein Management Tool, das Auskunft gibt über die Zustände von Clustern in Elasticsearch. Elasticsearch unterteilt den Zustand in drei Farben [19, S. 112]:

- **Green:** Der Cluster funktioniert einwandfrei.
- **Yellow:** Alle Primary Shards sind auf den Knoten verteilt, aber es sind nicht alle Replicas verteilt. Die Gefahr ist, wenn ein Knoten ausfällt, kann es zu Datenverlust kommen. Schreib- und Leseoperationen funktionieren weiterhin möglich.
- **Red:** Primary Shards konnten nicht auf anderen Knoten erfolgreich verteilt werden. Es kann zu Datenverlusten kommen. Schreibzugriffe werden nicht mehr entgegengenommen und Anfragen können teilweise unvollständige Ergebnisse zurückliefern.

Das Ziel von Cerebro ist die Verfügbarkeit des Servers zu erhöhen, denn wenn zu viele Daten im Cluster vorliegen, können die Daten im Primary Shards nicht verteilt werden und es kann zu einem Ausfall des Servers oder zu Datenverlust kommen.

5.6 elasticsearchdump

Elasticdump ist ein Add-on für Elasticsearch, womit eine Sicherheitskopie von Indizes und Mapping in Elasticsearch als CSV-Dateien erstellt werden kann, um Datenverluste zu vermeiden. Das Tool ist vor allem sinnvoll, wenn die Nutzung von Elasticsearch auf einen Server beschränkt ist.

5.7 Dejavu: Elasticsearch Web UI

Dejavu ist ein GUI für Elasticsearch und kann im Webbrowser genutzt werden. Dejavu gibt einen grafischen Überblick darüber, wie eine Benutzeroberfläche einer Suchmaschine aussehen kann und hilft bei der Analyse bzw. gibt einen guten Ausblick über die Datensätze, die sich in Elasticsearch befinden, da eine implementierte Such- und Aggregationsfunktion schon vorliegen, so kann nach Daten gesucht werden, ohne eine DSL-Anfrage in Elasticsearch zu implementieren.

5.8 Nutzung von APIs

5.8.1 GraphHopper-API

GraphHopper ist ein Open-Source-Routing-Server, bei der die Routenplanung individuell angepasst werden kann. Die GraphHopper Routing Engine wurde in der Programmiersprache Java implementiert und ist ein speichereffizienter Router, der als Server läuft und Offline Routing auf Android und iOS unterstützt [50]. Als Default wird Kartenmaterial von OpenStreetMap genutzt; es kann jedoch eine andere Datenquelle verwendet werden [51].

Weitere APIs, die im Projekt verwendet werden, sind die accessibility.cloud-API, Photon-API und Nominatim-API.

Im nächsten Kapitel wird auf die Methodik des Projekts eingegangen.

6 Methodik

Die Points of Interest von accessibility.cloud und die Straßendaten von OpenStreetMap sollen in die „WheelScout“-App mit Elasticsearch integriert werden, um eine Routenführung über gewünschte Points of Interest, unter Berücksichtigung der individuellen Barrieren, zu ermöglichen.

6.1 Beschaffung der Geodaten

6.1.1 Beschaffung der Point of Interest Daten

Zuallererst stellt sich die Frage, wie die POI-Daten von accessibility.cloud bezogen werden sollen. accessibility.cloud bietet eine JSON-API an, die zwei unterschiedliche geografische Bereichsanfragen bereitstellt:

Radiussuche: Bei der Radiussuche müssen bei der API-Nutzung der Token für die Authentifizierung, die geografische Koordinate sowie der Radius in Metern übergeben werden. Optional können zur Filterung noch weitere Parameter, wie Kategorie oder ein Limit, wie viele Ergebnisse zurückgegeben werden sollen, übergeben werden [52].

Rastersuche: Die Idee bei der Rastersuche ist es, die Karte in Kacheln zu unterteilen. Bei der Rastersuche (auch als XYZ-Suche bezeichnet) werden die Anzahl, Größe sowie Position der Kacheln von den XYZ-Werten bestimmt. Eine geografische Koordinate besteht aus der Latitude (Breitengrad) und der Longitude (Längengrad). Die Latitude muss auf X und die Longitude auf Y-Werte projiziert werden. Der Z-Wert muss vorher festgelegt werden, er bestimmt den Zoom bzw. die Größe und die Anzahl der Kacheln.

Zusätzlich hat SOZIALHELDEN e.V. einen **Dump** aller offenen POI-Daten aus accessibility.cloud als CSV-Datei (accessibility-cloud-places.csv) bereitgestellt. Der Dump stammt aus dem November 2019 und IDs und geografische Koordinaten mit identischem POI sind ab und zu in accessibility.cloud unterschiedlich gelistet.

Die Barrieren der Point of Interest werden in folgende Kriterien unterteilt:

- **Grün** - komplett barrierefrei - bedeutet, der Eingang ist stufenlos und alle Räume sind stufenlos erreichbar
- **Gelb** - eingeschränkt barrierefrei - bedeutet, der Eingang hat eine Stufe, die nicht höher als 7 cm ist und die meisten Räume können stufenlos erreicht werden
- **Rot** - nicht rollstuhlgeeignet - bedeutet, die obigen Kriterien werden nicht erfüllt

- **Grau** - nicht bewertet - bedeutet, das Gebäude wurde noch nicht auf Barrierefreiheit untersucht

accessibility.cloud vereint 138 Datenquellen und es befinden sich zur Zeit 2.782.055 Points of Interest in accessibility.cloud weltweit [53]. Davon befinden sich 550.248 Points of Interest in Deutschland. Barriereinformationen: 47.48 % der POIs in Deutschland sind laut accessibility.cloud barrierefrei, 34.78 % sind teilweise barrierefrei, 17.72 % der Gebäude sind nicht barrierefrei und nur 0.02 % werden als unbekannt deklariert (Diese Statistik wurde mit einer SQL-Query ermittelt).

Zudem werden einzelne Orte in Wheelmap in 12 verschiedene Oberkategorien eingeteilt (siehe Abb. 6.1 (a)) und jede Oberkategorie wird in weitere Unterkategorien unterteilt (siehe Abb. 6.1 (b)).

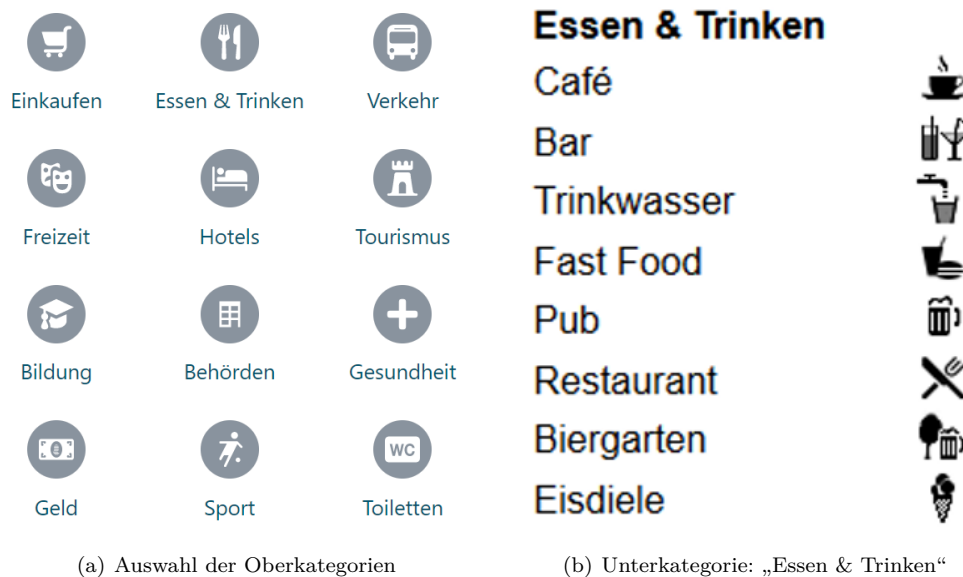


Abbildung 6.1: Auswahl anhand der Wheelmap-Kategorien

Toiletten können gesondert als rollstuhlgeeignet bewertet werden, wenn sie sich in einem barrierefreien Gebäude befinden und zusätzlich folgende Kriterien erfüllen:

- Türbreite von mindestens 90 cm
- Freie Bodenfläche von mindestens 150 cm * 150 cm
- Toilettensitz in Rollstuhllhöhe
- Aus dem Rollstuhl erreichbares Waschbecken

Die POI-Daten von accessibility.cloud beinhalten keine Adressdaten und Oberkategorien. Dies macht eine Volltextsuche in Elasticsearch nicht möglich.

6.1.2 Auswahl der Geotools-API

Für das Hinzufügen von Adressdaten zu den jeweiligen Points of Interest stellt sich die Frage, welches Geotool für das Geocoding am besten geeignet ist. Eine weitere Frage, die sich stellt, ist, welches Geotool für die Beschaffung der Straßendaten gewählt werden soll. Denn um alle Straßendaten in Deutschland zu erhalten, muss das ausgewählte Tool eine Bereichsanfrage anbieten, um möglichst effizient an die Geodaten zu kommen. In der Abbildung 6.1 wurde eine Gegenüberstellung von bekannten Geotools dargestellt.

Tabelle 6.1: Vergleich der Geotools

	Google Maps	Bing Maps	Photon	Nominatim
Datenqualität	★★★★★	★★★★☆	★★★★☆☆	★★★★☆☆
Geocoding	✓	✓	✓	✓
Bereichsanfrage	✗	✓ ¹	✓	✗
Offline Geocoding	✗	o	✓	✓
API Nutzung	kostenpflichtig ²	kostenpflichtig ³	kostenlos	kostenlos

¹<https://docs.microsoft.com/en-us/bingmaps/spatial-data-services/geocode-and-data-source-limits>

²<https://cloud.google.com/maps-platform/pricing?hl=de>

³<https://www.microsoft.com/en-us/maps/licensing>

6.1.3 Datenqualität der Geotools

Für die Bewertung der Datenqualität wird die Vollständigkeit sowie die Korrektheit der Daten betrachtet. Die Datenqualität bei Google Maps ist am besten. Google Maps umfasst 150 Millionen Orte (POIs) und deckt 99 % der Straßen und POIs auf der Welt ab. Es erfolgen pro Tag 25 Millionen Aktualisierungen. Die beiden Geotools Nominatim und Photon nutzen Kartenmaterial von OpenStreetMap. „Der Detailgrad der OSM-Daten ist regional sehr unterschiedlich. OpenStreetMaps behauptet, in vielen Städten schon besser als die meisten proprietären Karten zu sein“ [6]. Leider gibt es für Bing Maps keine Statistiken über die Daten, deshalb wurde ein Test in Deutschland durchgeführt. Wie in Tabelle 6.2 zu erkennen ist, sind die Adressdaten von Bing Maps bis auf die Hausnummer für Deutschland fast vollständig enthalten. Bei OpenStreetMap sind viele Adressdaten für Deutschland momentan lückenhaft. Das Tool Photon hat im Gegensatz zu Nominatim die Daten soweit wie möglich noch selber ergänzt. Bei Nominatim handelt es sich um Rohdaten von OpenStreetMap. Die Straßendaten von OpenStreetMap liegen als PBF-Format vor und lassen sich ohne Geotools nicht verarbeiten [54].

Tabelle 6.2: Reverse Geocoding in Deutschland in Prozent

Geotool	Adressdaten	Land	Bundesland	Stadt	Stadtteil	PLZ	Straße	Hsnr.
Bing Maps	555.375	100	100	99.99	0	99.99	99.99	96.00
Nominatim	550.248	100	91.72	43.71	32.13	99.83	98.86	68.05
Photon	7.187.037	100	99.99	96.77	0	99.09	41.90	18.97

6.1.4 Geocoding der Geotools

Alle der genannten Tools bieten Geocoding an. Bing Maps und Photon bieten beide Reverse-Geocoding-Bereichsanfragen an. Somit lassen sich mehrere Adressdaten pro Anfrage generieren.

6.1.5 Offline-Geocoding

Laut Webseite bietet Bing Maps Offline-Geocoding an [55], jedoch war die Webseite, von die Daten heruntergeladen werden sollen, offline. Deshalb kann es sein, dass das Offline-Geocoding doch nicht funktioniert. Nominatim und Photon lassen sich lokal verwenden bzw. Offline-Geocoding ist möglich.

Bei der Installation von Nominatim wird sehr viel Speicherkapazität benötigt; etwa 1 TB Festplattenspeicher und etwa 128 GB Arbeitsspeicher werden empfohlen, um den kompletten Kartenmaterial (weltweit) für das Geocoding zu verwenden [12]. Der Grund dafür ist, dass die Daten als Objekte bzw. Geometrien in der PostgreSQL-Datenbank gespeichert sind. Es handelt sich hierbei um komplexe Objekte, die viel Speicherplatz benötigen, wie beispielsweise MultiPolygon, Polygon, MultiLineString oder LineString usw. Zusätzlich ist die Installation von Nominatim auf einem Rechner extrem komplex [56].

Bei der Installation von Photon hingegen wird viel weniger Speicherkapazität benötigt, etwa 53 GB Speicherkapazität für die Daten [16] und zusätzlicher Speicherplatz für die Anwendung. Der Grund hierfür ist, dass Photon POIs und Straßen als geografische Koordinaten in Elasticsearch abspeichert; Mit dem Nachteil, dass Hausnummern von Straßen in Photon kaum erfasst werden (siehe Tab. 6.2).

6.1.6 Kosten und Lizenzierung der API-Nutzung

OpenStreetMap-Kartenmaterial wurde am 12. September 2012 unter der Lizenz „Open Database Licence (ODbL) 1.0“ verteilt. Die Nutzung von OpenStreetMap-Kartenmaterial ist kostenlos und steht für jede Person zur freien Verfügung. Die Daten dürfen weiterverarbeitet und kostenlos für alle Anwendungen genutzt werden und das auch gewerblich, solange die Lizenzbedingungen eingehalten werden [6].

Die Nutzung von Google Maps ist sehr teuer. Google Maps bietet jeden Monat 200 \$ an Budget an, wird das Budget nicht komplett aufgebraucht, wird im nächsten Monat das Budget wieder auf 200 \$ gesetzt (eine Kumulierung des Monatsbudgets ist nicht möglich). Pro 1.000 Geocoding-Anfragen verlangt Google Maps 5,00 \$ und die Daten dürfen nicht weiter verarbeitet, im Cache gehalten oder persistent gespeichert werden, d.h. es müssen immer neue Anfragen ausgeführt werden [57].

Auch Bing Maps ist kostenpflichtig. Die Preise sind nicht transparent und werden nur auf Anfrage beantwortet [58]. Die Daten bei Bing Maps dürfen ebenfalls nicht verarbeitet oder persistent gespeichert werden [55].

6.2 Zusammenführen der Daten und Implementierung der Anfragen in Elasticsearch

Wenn alle Daten vorliegen, muss überlegt werden, wie die Inhaltslogik der Points of Interest abgebildet werden soll. Wie sollen die Daten in Elasticsearch abgebildet werden? Denn ein Datenmodell hat enormen Einfluss auf die Performance beim Lesen und Schreiben der Daten und beeinflusst zudem die Speichergröße der Datenbank [20]. Wie sollen die Points of Interest geografisch abgebildet werden? Reichen Koordinatenpunkte aus, um die POIs zu repräsentieren oder müssen Polygone verwendet werden? Zusätzlich muss noch überlegt werden, wie die Bereichsanfrage in Elasticsearch aussehen soll. Ist eine Volltextsuche ausreichend oder wird auch eine Radiussuche benötigt, um Points of Interest in der „WheelScout“-App anzuzeigen? Welche Möglichkeiten der Filterung sollen bei der Anfrage angeboten werden? Alle diese Fragen müssen beantwortet werden, um eine Suchanwendung in die „WheelScout“-App zu integrieren.

6.3 Einbindung von Elasticsearch in das „WheelScout“-Projekt

Die Gebäudeinformationen aus der Elasticsearch sollen nun in die „WheelScout“-App mit einbezogen werden, um eine Routenführung über gewünschte Points of Interest, unter Berücksichtigung der individuellen Barrieren, zu ermöglichen.

6.3.1 „WheelScout“-Backend

Beim Backend stellt sich die Frage, wie läuft die Kommunikation zwischen dem Elasticsearch-Server und dem WheelScout-Backend-Server ab? Wie soll das Sicherheitskonzept aussehen, um vor Hacker-Angriffen geschützt zu sein?

6.3.2 „WheelScout“-Frontend

Bei der Entwicklung von Frontend stellt sich die Frage, wie die Points of Interest in der „WheelScout“-App dargestellt werden sollen. Sollen die Points of Interest als Marker dargestellt werden oder als eine Liste, wobei, einzelne POIs ausgewählt werden können? Wie soll die Barrierefreiheit von POIs in „WheelScout“-App abgebildet werden? Sollen dafür Symbole verwendet werden oder reichen unterschiedliche Farben der Marker aus? Und welche POIs sollten in der „WheelScout“-App als Default angezeigt werden? Alle diese Fragen werden in den nächsten Kapiteln behandelt und bereits im nächsten Kapitel „Design“ werden einige dieser Fragen beantwortet.

7 Design

Im Kapitel „Design“ wird die IT-Infrastruktur für die Einbindung von Elasticsearch in das „WheelScout“-Projekt dargestellt. Die erwünschte Suchfunktion für die „WheelScout“-App werden im Use-Case beschrieben. Zum Schluss wird das UML-Klassendiagramm für den Elasticsearch-Java-Client sowie den POIServer beschrieben.

7.1 IT-Infrastruktur

Die IT-Infrastruktur in Abbildung 7.1 gibt einen gesamten, groben Überblick über die Prozesse, die durchgeführt wurden, um eine Suchmaschine im Geodatenumfeld für die „WheelScout“-App bereitzustellen. Für die Implementierung der ETL-Skripts wurde die Programmiersprache Python gewählt, da Python viele Bibliotheken für die Datenverarbeitung anbietet. Zusätzlich stellen Nominatim und Photon Bibliotheken in Python zur Verfügung, die das Geocoding bzw. die Beschaffung von Adressdaten erleichtern. Ein weiterer Vorteil ist, dass die Anfragesprache in Elasticsearch (DSL), die in Kibana entwickelt wurde, 1 zu 1 in Python übernommen werden kann. Bei der Programmiersprache Java muss der Programmcode für die Suchanfragen in Elasticsearch komplett umgeschrieben werden. Points of Interest mit Barriereinformation in Deutschland werden von [accessibility.cloud](#) bezogen und als JSON-Datei abgespeichert. Für jeden Point of Interest wird die zugehörige Adresse mit der Nominatim-API ermittelt. Photon ist auch ein Geocoding Tool wie Nominatim, jedoch wurde Nominatim gewählt, weil Nominatim Point of Interest als Polygone in der Datenbank abbildet und Photon als geografische Koordinate. Dies führt dazu, dass Nominatim Adressdaten zu den einzelnen Points of Interest genauer zuordnen kann. Ein weiterer Grund für Nominatim ist, dass Nominatim viel mehr Adressdaten zur Verfügung stellt als Photon. Photon verwendet den gleichen Datensatz von OpenStreetMap wie Nominatim, jedoch filtert Photon nach Straßendaten; Hausnummern werden in Photon zum Großteil nicht beachtet. Für die Beschaffung der Straßendaten wurde die Photon-API verwendet, da die Straßendaten (siehe Statistik Kapitel 8: Implementierung) weniger Datenlücken aufweisen wie in Nominatim. Obwohl beide Kartenmaterialien von OpenStreetMap stammen, wurden bei Photon die Straßendaten teilweise vervollständigt und bei der Beschaffung der Straßendaten ist es mit Photon deutlich einfacher, die kompletten Daten zu erhalten als bei Nominatim, da Photon Bereichsabfragen anbietet. In der PostgreSQL-Datenbank werden die Adressdaten von Nominatim mit Adressdaten von Photon teilweise mit PostGIS-Funktionen vervollständigt und nochmals überarbeitet, damit ein Import der Daten in Elasticsearch möglich ist. Daraufhin werden die Volltext- und Radiusanfragen in Kibana implementiert und als Searchtemplate in Elasticsearch abgelegt. Der ELK-Stack (Elasticsearch, Logstash und Kibana) befindet sich auf einem Linux Docker Container und lässt sich aufgrund der Vorteile von Docker leichter horizontal skalieren, um Massenanfragen entgegen zu wirken. Mit dem `elasticdump`-Tool werden Sicherheitskopien von Elasticsearch-Indizes erstellt und

als CSV-Datei auf einem anderen Computer abgespeichert, um Datenverlust zu vermeiden. Mit dem Cerebro-Tool kann der Zustand von Elasticsearch betrachtet werden. Mit dem Elasticsearch-Java-Client oder über ein Elasticsearch REST API können Anfragen an den Elasticsearch-Server gesendet werden.

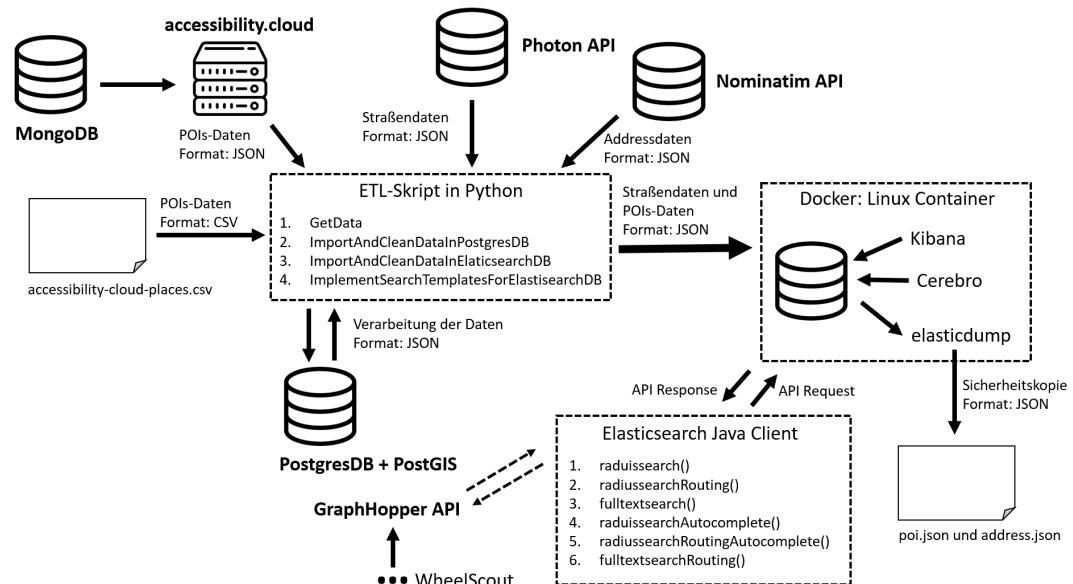


Abbildung 7.1: Infrastruktur

7.2 Use-Case-Diagramm

Das folgende Use-Case-Diagramm (7.2) visualisiert, welche Funktionalität in Elasticsearch abgebildet werden soll. Beim 1. Use-Case soll der „WheelScout“-Anwender einzelne Point of Interest oder Straßendaten in Elasticsearch überarbeiten, ergänzen oder hinzufügen. Die bearbeiteten Daten werden in einer CSV-Datei oder in einer relationalen Datenbank abgespeichert. Ein Prüfmechanismus soll die Daten auf Richtigkeit überprüfen. Die Idee ist, durch eine Abstimmung der Anwender der „WheelScout“-App die Richtigkeit der Daten zu evaluieren. Nach der Evaluierung der Daten, werden die Daten durch ein HTTP-Request von dem Elasticsearch-Python-Client an den Elasticsearch-Server gesendet, der die Daten updatet oder hinzufügt.

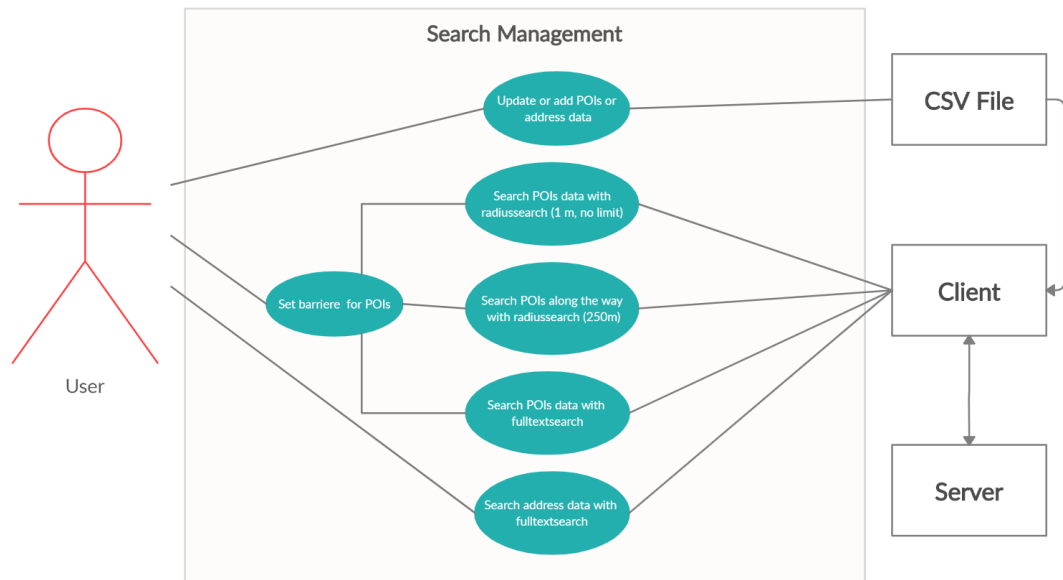


Abbildung 7.2: Use-Case-Suchfunktion

Bei der Suchanfrage von Points of Interest kann der Nutzer zunächst seine individuellen Barriere Einschränkungen angeben. Als Default werden alle Points of Interest angezeigt und eine Filterung nach Barrieren findet nicht statt. Es können folgende Filtereinstellungen vorgenommen werden:

- fully-accessible-by-wheelchair (1)
- at-least-partially-accessible-by-wheelchair (2)
- not-accessible-by-wheelchair (3)
- unknown-wheelchair-accessibility (4)

Die Auswahlmöglichkeit „unknown-wheelchair-accessibility“ wurde hier ebenfalls gelistet; zum einen, um Unterstützern des „WheelScout“-Projektes unbekannte Gebäude aufzuzeigen, um die Gebäude dann bewerten zu können, zum anderen, um dem Nutzer die Möglichkeit zu geben, selbst zu entscheiden, ob er dieses nicht gekennzeichnete Gebäude auswählen möchte.

Es sollen drei verschiedene Suchanfragen in Elasticsearch implementiert werden:

- Bei der **Radiussuche** werden im Umfeld einer festgelegten Radiusgröße in Meter, alle Points of Interest ausgegeben. Eine Filterung nach Ober- oder Unterkategorien sowie eine Autovervollständigung sollte auch vorhanden sein.
- Eine „**POIs während des Routings anzeigen**“-Anfrage ermöglicht es dem Nutzer, während des Routings der Strecke, alle erwünschten Gebäuden in näheren Umgebung anzuzeigen. Möchte der Nutzer beispielsweise auf die Toilette, werden entlang der Strecke alle vorhandene Toiletten gelistet.

- Bei der **Volltextsuche** soll nach konkreten Begriffen bzw. nach Bezeichnungen der POI, Ober- oder Unterkategorien gesucht werden können, wie beispielsweise „Hotel in Berlin“ oder „McDonald’s 64409“. Eine Volltextsuche ist für die Planung oder Organisation, wie z.B. ein Urlaubsreise in Berlin, sehr gut geeignet. Eine Volltextsuche nach Straßendaten soll auch gegeben sein, um es zu erleichtern, beispielsweise Freunde zu besuchen. Bei der Volltextsuche soll bei der Sucheingabe mit Singular-Plural-Abweichungen sowie Synonymen umgehen können. Wie die Volltextsuche im Detail funktioniert, wird bei der Implementierung erklärt.

7.3 Schema der Geodaten in Elasticsearch

In diesem Abschnitt werden das Schema bzw. das Mapping und die Einstellungen und die Wahl der Analyzer für die Points of Interest in der Suchmaschine Elasticsearch vorgestellt. Aus dem Grund, dass Straßendaten das gleiche Schema, wie bei den Adressen von Points of Interest verwendet, wird nur die Struktur der Points of Interest gelistet, um Redundanz zu vermeiden.

Zunächst eine kurze Wiederholung: In den Settings werden die Einstellungen für den Cluster vorgenommen (Anzahl an Shards und Replikas) und die Wahl der Analyzer wird bestimmt. Analyzer sind dafür zuständig, wie die Daten in Elasticsearch verarbeitet werden. Dabei lässt der Analyzer sich in 3 Stufen aufteilen: Char-Filter, Tokenizer und Tokenfilter.

7.3.1 Einstellungen

In der Listing 7.1 befinden sich die Einstellungen für den „POI“-Index. Aufgrund dessen, dass zurzeit nicht viele Daten vorliegen und die Anfragelast am Anfang nicht so drastisch ist, sollten 5 Shards für den Anfang in Elasticsearch ausreichen. Wenn die Anfragelast steigt, kann der Index neu indexiert werden, da sich zurzeit nicht viele Daten im Index befinden. Die Elasticsearch-Datenbank befindet sich zurzeit auf einem Server. Deshalb werden noch keine Replikas benötigt. Bei Verteilung der Datenbank können Replikas noch hinzugefügt werden. Der Parameter „max_ngram_diff“ ist für die Autovervollständigung notwendig. Dies wird im späteren Abschnitt noch genau erklärt.

```
1 {
2   "settings": {
3     "index": {
4       "number_of_shards": 5,
5       "number_of_replicas": 0,
6       "max_ngram_diff": 10
7     }
8   }
```

Listing 7.1: Indexeinstellungen in Elasticsearch

7.3.2 Analyzer

Der Analyzer ist für die Zerlegung von Texten in Token zuständig und durchläuft drei Prozesse: Char-Filter, Tokenizer und Token-Filter. In Lis. 7.2 befindet sich die Filter für den „POI“-Index in Elasticsearch. Beim **Char-Filter** wird mit dem („my_mappings_char_filter“), das Sonderzeichen „“ bei den weiteren Prozessen ignoriert. Beim **Tokenizer** gibt es zwei Varianten für die Autovervollständigung, den („ngram_tokenizer“) und den („edge_ngram_tokenizer“). Beim („edge_ngram_tokenizer“) ist der Cursor fixiert und bei („ngram_tokenizer“) ist der Cursor dynamisch. Nun erfolgt ein Beispiel für die Anfrage „Lidl Darmstadt“, um die beiden Prozesse besser nachvollziehen zu können:

edge(min:1, max:10): [L, Li, Lid, Lidl, D, Da, Dar, Darm, Darms, Darmst, ...]
ngram(min:1, max:3): [L Li Lid i id idl, d, dl, l, D, Da, Dar, ...]

Die Werte („min_gram“) und („max_gram“) wurden durch mehrere Experimente ermittelt, welches die besten Suchergebnisse liefert. Im („token_chars“) wird definiert, welche Zeichen im nächsten Prozess nicht ignoriert werden sollen. Es werden Buchstaben, Zahlen und das Sonderzeichen „&“ beachtet, andere Zeichen werden im weiteren Prozess ignoriert, jedoch werden beim Suchergebnis alle Zeichen angezeigt.

Nun wird der **Token-Filter** näher betrachtet:

- **„de_cities_stop_words_filter“:** Ist für die Nichtberücksichtigung der Städte in den Begriffen bzw. Bezeichnung der Points of Interest zuständig, wie beispielsweise „Arbeitsamt Fürth“
- **„name_abbreviations_words_filter“:** Abkürzung von Begriffen, wie Hauptbahnhof zu HBF und *straße zu *str
- **„german_stop_words_filter“:** Entfernung vom Stoppwort „in“ (zu viele Stoppwörter verschlechtern die Suchergebnisse, bei der Volltextsuche)
- **„de_categories_synonym_plural_words_filter“:** Ergänzung von Synonym und Pluralform für die Unterkategorien

```

1  "char_filter": {
2    "my_mappings_char_filter": {
3      "type": "mapping",
4      "mappings": [ " " => " " ]
5    }
6  },
7  "tokenizer": {
8    "ngram_tokenizer": {
9      "type": "ngram",
10     "min_gram": 1,
11     "max_gram": 3,
12     "custom_token_chars": [ "&" ],
13     "token_chars": [ "letter", "digit", "custom" ]
14   },

```

```

15     "edge_ngram_tokenizer": {
16         "type": "edge_ngram",
17         "min_gram": 1,
18         "max_gram": 10,
19         "custom_token_chars": ["&"],
20         "token_chars": ["letter", "digit", "custom"]
21     }
22 },
23 "filter": {
24     "de_cities_stop_words_filter": {
25         "type": "stop",
26         "stopwords_path": "stop.txt"
27     },
28     "name_abbreviations_words_filter": {
29         "type": "synonym",
30         "lenient": true,
31         "synonyms": ["Hauptbahnhof", "HBF", "stra ß e => str"]
32     },
33     "german_stop_words_filter": {
34         "type": "stop",
35         "stopwords": ["in"]
36     },
37     "de_categories_synonym_plural_words_filter": {
38         "type": "synonym",
39         "lenient": true,
40         "synonyms_path": "de_categories_synonym.txt"
41     }
42 }

```

Listing 7.2: Char-Filter, Tokenizer, Token-Filter

Aus den Bauteilen von Char-Filter, Tokenizer und Token-Filter lassen sich unterschiedliche Analyzer zusammenstellen. Aufgrund dessen, dass der Teil des Analyzers zu lange ist, werden nur Analyzer mit Autovervollständigung im Listing 7.1 dargestellt. Weitere Analyzer ohne Autovervollständigung werden nicht gelistet - der einzige Unterschied ist, dass bei Analyzern ohne Autovervollständigung der „Standard“-Tokenizer verwendet wird. Ein Analyzer kann nur einen Tokenizer besitzen, aber es lassen sich mehrere Token- und Char-Filter einem Analyzer zuordnen. Der „ngram“-Tokenizer wird speziell nur für Volltextsuchanfragen von Straßendaten oder POI-Daten benötigt und der „edge-ngram“-Tokenizer wird für die Autovervollständigung bei Radiussuche verwendet. Alle Analyzer verfügen über den gleichen Char-Filter und alle Wörter werden mit dem Token-Filter „lowercase“ in Kleinbuchstaben umgewandelt. Die Analyzer werden bei den Feldern im Mapping (Schema) und bei der Suchanfrage eingesetzt, um die gleiche Verarbeitung von Textdaten in Tokens zu gewährleisten. Der Einsatz der Analyzer ist in folgenden Anfragen oder Feldern erforderlich:

- „**stop_name_autocomplete_ngram_analyzer**“ wird für das Feld („name autocomplete_ngram“) benötigt, um Städtenamen in den Bezeichnung von POIs zu ignorieren, damit bei der Volltextsuchanfrage keine Redundanz entsteht.
- „**german_categories_synonym_plural_autocomplete_ngram_analyzer**“: Der Analyzer wird für das Feld („de_category autocomplete_ngram“) benötigt, um mit Synonyme oder Pluralformen von Kategorien in der Volltextsuche umzugehen.

- Der „**special_multimatch_autocomplete_ngram_analyzer**“ wird benötigt, um alle notwendigen Token-Filter für die Volltextsucheanfrage von POIs zu kombinieren.
- Der „**street_autocomplete_ngram_analyzer**“ wird dem Feld („street autocomplete“) zugewiesen, um Abkürzungen bei der Volltextsuche zu ermöglichen.
- „**standard_autocomplete_ngram_analyzer**“: Alle restlichen Felder werden mit diesem Analyzer belegt.
- Der „**radius_autocomplete_edge_ngram_analyzer**“ wird bei der Sucheingabe von POIs-Bezeichnungen benötigt, um eine Autovervollständigung für die Radiussuche zu ermöglichen.

```

1  "analyzer": {
2    "stop_name_autocomplete_ngram_analyzer": {
3      "type": "custom",
4      "tokenizer": "ngram_tokenizer",
5      "char_filter": ["my_mappings_char_filter"],
6      "filter": [
7        "name_abbreviations_words_filter",
8        "de_cities_stop_words_filter",
9        "lowercase"
10     ]
11   },
12   "german_categories_synonym_plural_autocomplete_ngram_analyzer": {
13     "type": "custom",
14     "tokenizer": "ngram_tokenizer",
15     "char_filter": ["my_mappings_char_filter"],
16     "filter": [
17       "lowercase",
18       "de_categories_synonym_words_filter"
19     ]
20   },
21   "special_multimatch_autocomplete_ngram_analyzer": {
22     "type": "custom",
23     "tokenizer": "ngram_tokenizer",
24     "char_filter": ["my_mappings_char_filter"],
25     "filter": [
26       "name_plural_words_filter",
27       "lowercase",
28       "german_stop_words_filter",
29       "de_categories_synonym_words_filter"
30     ]
31   },
32   "street_autocomplete_ngram_analyzer": {
33     "type": "custom",
34     "tokenizer": "ngram_tokenizer",
35     "char_filter": ["my_mappings_char_filter"],
36     "filter": [
37       "name_abbreviations_words_filter",
38       "lowercase"
39     ]
40   },
41   "standard_autocomplete_ngram_analyzer": {
42     "type": "custom",
43     "tokenizer": "ngram_tokenizer",
44     "char_filter": ["my_mappings_char_filter"],
45     "filter": ["lowercase"]
46   },

```

```
47     "radius_autocomplete_edge_ngram_analyzer": {  
48       "type": "custom",  
49       "tokenizer": "edge_ngram_tokenizer",  
50       "char_filter": ["my_mappings_char_filter"],  
51       "filter": ["lowercase"]  
52     }  
53 }
```

Listing 7.3: Analyzer für Points of Interest

7.3.3 Mapping bzw. Datenstruktur der Geodaten in Elasticsearch

Die Idee: Jeder Point of Interest bzw. jedes Gebäude soll in Elasticsearch als ein Dokument abgebildet werden.

Das Mapping für Points of Interest bzw. die Dokumentstruktur wurde im Lis. 7.4 verkürzt dargestellt. POI-Daten werden im Index „POI“ als Objekte eingebettet, d.h. es gibt keine Hierarchien bei Speicherung der Dokumente im Lucence-Index [19, S. 146]. Jedoch kann im Mapping eine Dokumentstruktur mit Hierarchien abgebildet werden. Wie zu erkennen ist, befindet sich Adressdaten im Feld („address“) und Geodaten im Feld („geometry“). Die Idee war, die Dokumentstruktur möglich nahe an dem GeoJSON-Format zu strukturieren. Im Mapping ist zu erkennen, dass jedes einzelne Feld durch Datentypen und Analyzer belegt ist.

Die Einstellung („dynamic“: false) bedeutet, dass dynamisches Mapping ausgeschaltet ist, d.h. wenn ein neues Attribut hinzugefügt werden soll, muss zuerst im Mapping ein neues Feld manuell erzeugt werden. Diese Einstellung soll sicherstellen, dass nicht der „Standard“-Analyzer für das neue Feld verwendet wird, da, wenn unterschiedliche Analyzer in Anfragen oder im Feld vorliegen, keine oder nur schlechte Suchergebnisse ausgegeben, da die Verarbeitung von Textdaten in Termen (Tokens) unterschiedlich differieren würde. Um zu verdeutlichen wie wichtig das Zusammenspiel von Datentypen, Analyzern und Suchanfrage ist, wird es anhand des Attributes („name“) dargestellt. Nur für das Attribut („name“) bzw. für die Bezeichnung der POIs existieren vier unterschiedliche Felder im Mapping. Im 1. Feld („name“) wird ein „Standard“-Tokenizer für eine Suchanfrage ohne Autovervollständigung verwendet. Der Vorteil bei der Nutzung von „Standard“-Tokenizern ist, dass die Anfragen sehr performant sind. Wenn eine hohe Anfragelast vorliegt, kann, anstatt einen 2. Server zu nutzen, was mit hohen Kosten verbunden ist, eine normale Suchanfrage verwendet werden, um gegen einer hohen Anfragelast entgegen zu wirken. Beim 2. Feld („name.keyword“) wird das Feld für eine Aggregationsanfrage benötigt. Dies wird später genutzt, um Bezeichnung von POIs zu standardisieren. Mit der Einstellung („ignore_above: 256“) werden alle Daten, die sich im Feld befinden und mehr als 256 Zeichen besitzen, nicht indexiert, d.h. bei einer Suchanfrage wird der Inhalt des Feldes im Suchergebnis nicht gelistet. Das 3. Feld („name.autocomplete_ngram“) wird für die Volltextsuche von POIs benötigt, um bei einer unvollständigen Sucheingabe das passende Suchergebnis zu erhalten. Das 4. Feld („name.autocomplete_edge_ngram“) wird bei der Radiussuchanfrage von POIs oder für die „Anzeige von POIs während des Routings“-Anfrage für die Autovervollständigung benötigt.

Toiletten („restrooms“) werden gesondert als eigenes Feld in der Datenstruktur dargestellt und nicht Teil als („de_parentCategory“), da sich („restrooms“) normalerweise in einem POI befindet. Deshalb macht es keinen Sinn, Toiletten als Points of Interest zu erfassen.

```

1  "mappings": {
2    "dynamic": false,
3    "properties": {
4      "name": {
5        "type": "text", "analyzer": "stop_name_analyzer",
6        "fields": {
7          "keyword": {"type": "keyword", "ignore_above": 256}}},
8          "autocomplete_ngram": {
9            "type": "text", "analyzer": "stop_name_autocomplete_ngram_analyzer"},
10         "autocomplete_edge_ngram": {
11           "type": "text", "analyzer": "
12             stop_name_autocomplete_edge_ngram_analyzer"},
13         "description": {"type": "text"},
14         "placeWebsiteUrl": {"type": "text"},
15         "toilet": {"type": "keyword"},
16         "accessibleWith": {"type": "keyword"},
17         "en_parentCategory": {"type": "text", "analyzer": "standard_analyzer",
18           "fields": {
19             "autocomplete": {
20               "type": "text", "analyzer": "standard_autocomplete_ngram_analyzer"},
21               "keyword": {"type": "keyword", "ignore_above": 256}}},
22         "de_parentCategory": {
23           "type": "text", "analyzer": "standard_analyzer",
24           "fields": {
25             "autocomplete": {
26               "type": "text", "analyzer": "standard_autocomplete_ngram_analyzer"},
27               "keyword": {"type": "keyword", "ignore_above": 256}}},
28         "en_category": {
29           "type": "text", "analyzer": "standard_analyzer",
30           "fields": {
31             "autocomplete": {
32               "type": "text", "analyzer": "standard_autocomplete_ngram_analyzer"},
33               "keyword": {"type": "keyword", "ignore_above": 256}}},
34         "de_category": {
35           "type": "text", "analyzer": "german_categories_synonym_plural_analyzer",
36           "fields": {
37             "autocomplete": {
38               "type": "text",
39               "analyzer": "
40                 german_categories_synonym_plural_autocomplete_ngram_analyzer"},
41               "keyword": {"type": "keyword", "ignore_above": 256}}},
42         "geometry": {
43           "properties": {"coordinates": {"type": "geo_point"},
44             "type": {"type": "text"}}},
45         "address": {
46           "properties": {
47             "street": {"type": "text", "analyzer": "standard_analyzer",
48               "fields": {
49                 "autocomplete": {
50                   "type": "text",
51                   "analyzer": "street_autocomplete_ngram_analyzer"}}},
52             "houseNumber": {"type": "text", "analyzer": "standard_analyzer"},
53             "postcode": {"type": "text", "analyzer": "standard_analyzer",
54               "fields": {
55                 "autocomplete": {
56                   "type": "text",
57                   "analyzer": "standard_autocomplete_ngram_analyzer"

```

```

57     "keyword": {"type": "keyword", "ignore_above": 256}},
58     "city": {
59       "type": "text", "analyzer": "standard_analyzer",
60       "fields": {
61         "autocomplete": {
62           "type": "text", "analyzer": "
             standard_autocomplete_ngram_analyzer"},
63         "keyword": {"type": "keyword", "ignore_above": 256}},
64     "city_district": {
65       "type": "text", "analyzer": "standard_analyzer",
66       "fields": {
67         "autocomplete": {
68           "type": "text", "analyzer": "
             standard_autocomplete_ngram_analyzer"},
69         "keyword": {"type": "keyword", "ignore_above": 256}},
70     "state": {
71       "type": "text", "analyzer": "standard_analyzer",
72       "fields": {
73         "autocomplete": {
74           "type": "text", "analyzer": "
             standard_autocomplete_ngram_analyzer"},
75         "keyword": {"type": "keyword", "ignore_above": 256}},
76     "country": {
77       "type": "text", "analyzer": "standard_analyzer",
78       "fields": {
79         "autocomplete": {
80           "type": "text", "analyzer": "
             standard_autocomplete_ngram_analyzer"},
81         "keyword": {"type": "keyword", "ignore_above": 256}
82       }
83     }
84   }
85 },
86 "infoPageUrl": {"type": "text"},
87 "originalId": {"type": "text"},
88 "sourceId": {"type": "text"},
89 "sourceImportId": {"type": "text"}
90 }
91 }

```

Listing 7.4: Mapping der Points of Interest

7.4 UML-Klassendiagramm vom Elasticsearch Java Client

Requests an Elasticsearch erfolgen durch eine Elasticsearch REST API oder über einen Elasticsearch-Java-Client. In diesem Fall wird die REST API sowie ein Elasticsearch-Java-Client verwendet. Die Entscheidung fiel auf einen Java-Client, da die „WheelScout“-App und die Suchmaschine Elasticsearch in der Programmiersprache Java geschrieben wurden und Java eine performantere Programmiersprache im Gegenzug zu Python ist. In der Abbildung 7.3 befindet sich das UML-Klassendiagramm zum Elasticsearch-Java-Client und weitere Klassen. Es handelt sich hierbei um ein Maven Java Projekt, um Bibliotheken einfacher in das Projekt zu integrieren. Der Elasticsearch-Java-Client wird benötigt, da die Anfrage „POIs während des Routings anzeigen“ nicht in Elasticsearch abgebildet werden kann. Im Folgenden werden die Klassen und Methoden beschrieben: Die **main**-Klasse ist für die Ausführung der Anfragen

an den Elasticsearch-Server zuständig und wird nur aus Testzwecken benötigt. In der **main**-Klasse wird ein Objekt der Klasse **Request** erzeugt. In der Klasse **Request** befindet sich Methoden von Suchanfragen für den Elasticsearch Server. In den implementierten Methoden in der Klasse **Request** wird die Wahl der Searchtemplates in Elasticsearch anhand der ID bestimmt bzw. welche DSL-Anfrage in Elasticsearch ausgeführt werden soll (Volltextsuche, Radiussuche etc.). Durch die Übergabe an Suchparameter, wie Suchbegriff, Kategorien oder Toiletten, lässt sich die Anfrage ausführen.

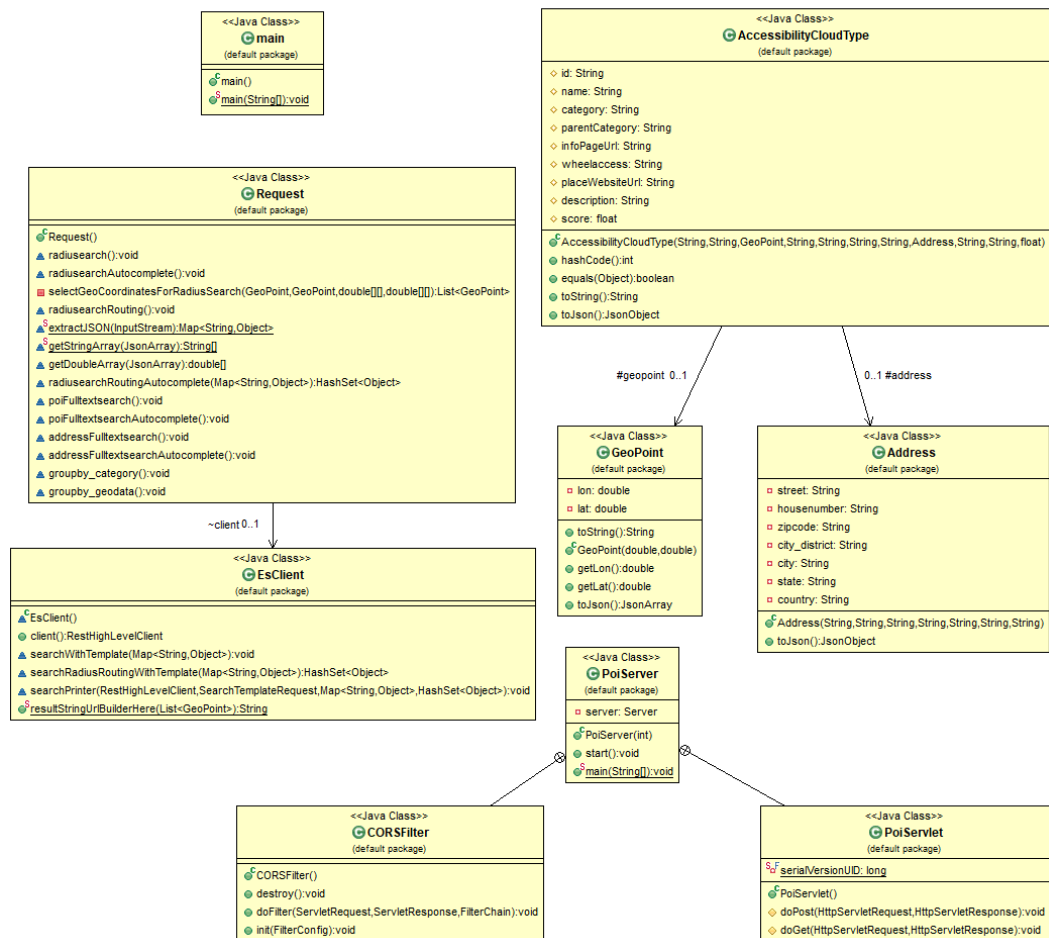


Abbildung 7.3: Elasticsearch-Java-Client und POIServer: UML-Klassendiagramm

7.4.1 „POIs während des Routings anzeigen“-Anfrage

Die Klasse **POIServer** ist ein Server, der HTTP-Methoden wie GET, POST, PUT oder DELETE ausführen kann. Der Server wird für die Anfrage „POIs während des Routings anzeigen“ benötigt. Wenn ein Nutzer der „WheelScout“-App diese Anfrage ausführt, wird zuerst die Anfrage an den „WheelScout“-Server gesendet, der für das Routing zuständig ist und GraphHopper nutzt. Die Routing Engine GraphHopper wird daraufhin, die gesuchte

Strecke unter Berücksichtigung der Barrieren, die sich in der Maria DB befinden berechnen. Daraufhin werden die Suchparameter des Anwenders sowie Daten wie Koordinaten von den Nodes und deren Abstände an den POIServer gesendet. Dieser führt die Methode „radiussearchRoutingAutocomplete()“, die sich auch in der Klasse **Request** befindet, aus. Die Methode „radiussearchRoutingAutocomplete()“ ruft zuallererst die Methode „selectGeoCoordinatesForRadiusSearch()“ auf. Diese Methode wählt bestimmte Koordinaten aus (das wird in Kapitel 8 detaillierte beschrieben). Auf diese ausgewählten Koordinaten wird jeweils eine Radiussuche in Elasticsearch ausgeführt, um alle Points of Interest in der Umgebung der Strecke zu erhalten. Dies wird mit der Methode „client.searchWithTemplate(params)“ bewerkstelligt, denn diese Methode übergibt alle Suchparameter an die Klasse **EsClient** und führt die Methode „client()“ aus, um eine Verbindung zum Elasticsearch-Server aufzubauen und die Radienanfragen auf dem Server auszuführen. Der Elasticsearch-Server beantwortet die Anfragen und gibt die Suchergebnisse als JSON-Format zurück. Aus der JSON-Response werden alle relevanten Daten extrahiert und es wird für jedes Dokument ein Objekt von der Klasse **AccessibilityCloudType** erzeugt, um die gefilterten Daten den Objektattributen (siehe Abbildung 7.3) zuzuordnen. Die **AccessibilityCloud** hat weitere Unterklassen wie **Adressen** und **Geopoint**, die Geodaten abbilden. Bei der Ausführung der Methode „radiussearchRoutingAutocomplete()“ entsteht Redundanz, da sich die Radien überschneidet (dies so gewollt). Um Redundanz der Daten zu entfernen, werden die AccessibilityCloudType-Objekte in ein HashSet abgelegt, welches mit der Methode „equals()“ in der Klasse **AccessibilityCloudType** dann ausgeführt wird, um zu überprüfen, ob ein identischer Hashwert der Objekte vorliegt. Falls ja, wird das Objekt von HashSet entfernt. Daraufhin wird die Methode „toJSON()“ die Ergebnisse wieder in das Format JSON überführen und die Daten werden an das Backend von der „WheelScout“-App weitergeleitet. Im Backend der WheelScout werden die Daten bzw. die Ergebnisse an die „WheelScout“-App versendet. Mit der Methode „resultStringUrlBuilderHere(List<GeoPoint> listGeo)“ in der Klasse **EsClient** wird eine URL erzeugt, die die Radien auf einer gerouteten Strecke grafisch dargestellt, um zu überprüfen, ob die gesamte Strecke mit Radien auch abgedeckt ist.

Im nächsten Kapitel wird die Implementierung der ETL-Skripts detailliert beschrieben.

8 Implementierung und Realisierung

ETL-Skripts in Python: Bei allen Skripten wurden Ausführung und Anwendungslogik voneinander getrennt dargestellt. Es gibt insgesamt 4 Skripte, die in Python implementiert wurden:

- Das **1. Skript** ist für die Beschaffung von POI-Daten und Geodaten aus verschiedenen Quellen und für die Zusammenführung der Daten zuständig.
- Im **2. Skript** werden die Daten in die PostgreSQL-Datenbank importiert und verarbeitet.
- Das **3. Skript** ist für den Import der Daten in die Elasticsearch-Datenbank und für die Normalisierung der Bezeichnung der POIs zuständig.
- Im **4. Skript** befindet sich die Implementierung der Elasticsearch Suchanfragen, die als Searchtemplate auf dem Elasticsearch-Server gespeichert werden.

Es folgt nun eine detaillierte Beschreibung der implementierten Funktion der ETL-Skripts.

8.1 1. Skript: Beschaffung der Geodaten

Die Beschaffung der Geodaten gehört zu den aufwändigsten und zeitintensivsten Prozessen. Hierbei wurden drei unterschiedliche Varianten implementiert, um POI-Daten von accessibility.cloud in Deutschland zu erhalten. Es wird zurzeit nur eine Variante der drei Varianten genutzt, jedoch werden bei der Beschaffung der POI-Daten im Ausland auch andere Varianten benötigt (siehe letztes Kapitel: Fazit und Ausblick).

Variante 1: SOZIALHELDEN e.V. hat einen Dump aller offenen Points of Interest Daten aus accessibility.cloud als CSV-Datei (accessibility-cloud-places.csv) bereitgestellt. Der Dump ist aus November 2019. Die Idee war, die Points of Interest der CSV-Datei mit den Daten aus accessibility.cloud zu kombinieren, da mehr aktuelle POI-Daten in accessibility.cloud vorliegen und auch teilweise bereinigt wurden. Jedoch sind die IDs sowie geografischen Koordinaten im Dump bei identischer POI, ab und zu unterschiedlich im Vergleich zu den Daten, die sich in accessibility.cloud befinden. Dieses Problem führt beim Import der Daten in Elasticsearch dazu, dass identische POI als unterschiedliche POIs betrachtet werden und dadurch zweimal in der Elasticsearch-Datenbank gelistet wird. Durch komplexe SQL-Queries mit PostGIS-Funktionen konnte dieses Problem in der PostgreSQL-Datenbank gelöst werden. Ein weiteres Problem war die Transformation von Tabellen in CSV in ein kompatibles Elasticsearch JSON-Format. Dieses Problem hat viel Zeit in Anspruch genommen, um es zu beheben, da viele Bereinigungs-schritte benötigt wurden.

Vorgehensweise: Aus Wikipedia wurden die 100 größten Städte in Deutschland entnommen. In der Abbildung 8.1 ist zu erkennen, dass sich viele POIs vor allem in Großstädten wie beispielsweise Berlin, Stuttgart, Frankfurt oder Düsseldorf befinden. Die Idee war es POIs, die sich in den Großstädten befinden von accessibility.cloud zu beziehen, da die Datenqualität besser ist. Durch umgekehrtes Geocodierung mit der Nominatim-API wurden die Städte in geografischen Koordinaten umgewandelt. Mit den Koordinaten wurden jeweils eine Radiussuche in accessibility.cloud durchgeführt. Die Radiuslänge wurde auf 2.500 m festgelegt, da die Ressourcen in accessibility.cloud begrenzt sind und eine größere Radiuslänge zu einer zu großen Belastung geführt hätte. In accessibility.cloud wurde ein Time-out integriert. Kann eine Anfrage nicht innerhalb einer bestimmten Frist bearbeitet werden, wird eine HTTP-Fehlermeldung (503: first byte timeout) auftreten. Dieses Problem konnte mit einer While-Schleife und einem Exception Handling gelöst werden. Das Exception Handling fängt den Fehler auf und die While-Schleife sorgt dafür, dass der Vorgang wiederholt wird.

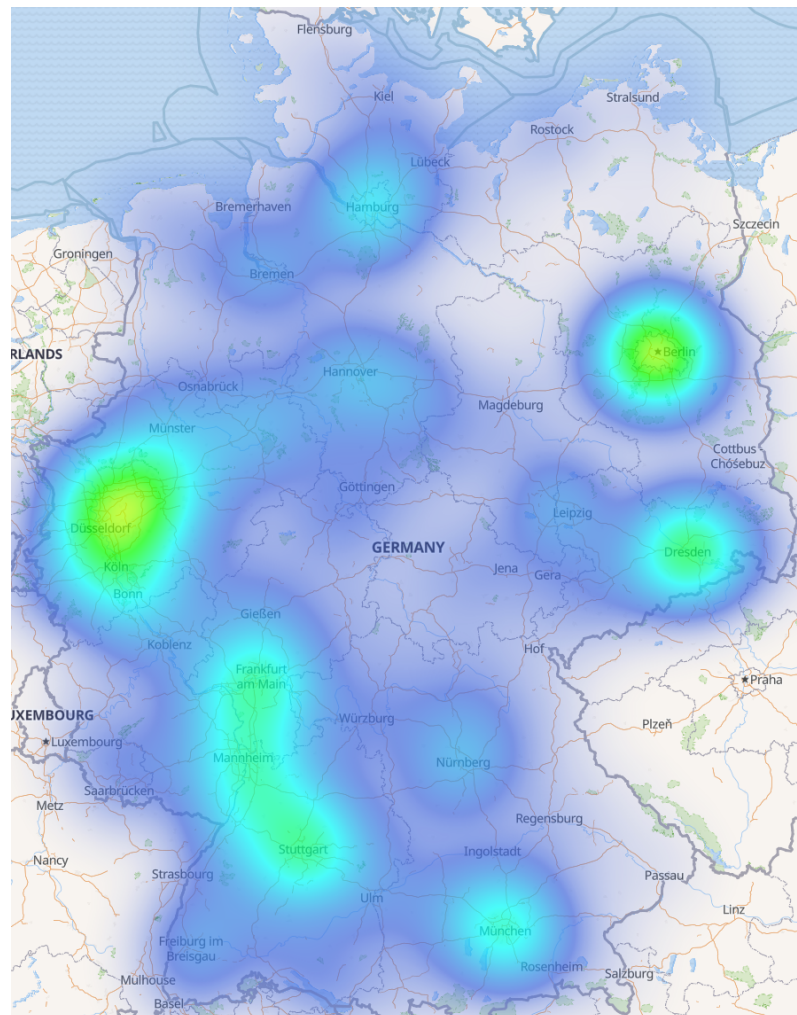


Abbildung 8.1: POIs in Deutschland

Bei **Variante 2** wird eine große Anzahl (500.000) an zufälligen geografischen Koordinaten in Deutschland in der PostgreSQL-Datenbank erzeugt. Mit einem k-Means Algorithmus

wurden 20.000 Zentroide gebildet. K-Means ist ein Algorithmus, der Gruppierungen von Objekten erzeugt [59]. Durch die gebildeten Zentroide soll der Abstand der Koordinaten zueinander möglichst gleichmäßig verteilt werden, sodass die Anzahl der Radiananfragen an `accessibility.cloud` möglichst gering gehalten wird und um Duplikate von POIs zu minimieren (Überschneidung der Radian). Beim Import der Daten in Elasticsearch wird das Problem der Redundanz behoben. Dies geschieht dadurch, dass die („_id“) in Elasticsearch eindeutig sein müssen und bei identischen („_id“) werden die Points of Interest überschrieben. Das liegt daran, dass Elasticsearch mit Versionierung arbeitet [60].

Variante 3: `accessibility.cloud` bietet eine Kachelsuche (wird auch als XYZ-Suche bezeichnet; s. Kapitel 6: Methodik) via API an. Bei Variante 3 wird die XYZ-Suche verwendet, um die Daten von `accessibility.cloud` zu beschaffen.

Vorgehensweise: In der PostgreSQL-Datenbank wird auf ein Multipolygon, das Deutschland abbildet, ein minimales umgebenes Rechteck (MUR) gebildet. Daraufhin werden die Koordinaten der Eckdaten vom MUR entnommen und mit einem Z-Wert von 14 in X- und Y-Werte transformiert. Durch die Erhöhung oder Reduzierung der X- oder Y-Werte um 1, wird jeweils die Position der Kachel nach links, rechts oder nach oben bzw. nach unten geschiftet. Dadurch kann keine Redundanz der POI-Daten entstehen.

Da bei der Variante 3 keine Redundanz der Daten bei der Beschaffung von POI-Daten in `accessibility.cloud` vorliegt, wird Variante 3 verwendet. Nun erfolgt eine detaillierte Beschreibung der Funktionen mit Ausschnitten des Programmcodes, um ein besseres Verständnis für die Implementierung zu erhalten:

Zu Anfang wurden Multipolygone, die die Länder abbilden, als GeoJSON-Dateien (`countries.geojson`) von `Opendatasoft` heruntergeladen. Mit der Funktion „`multipolygon_to_mbr_with_postgis()`“ wurden die Daten in Python eingelesen und das Multipolygon, das Deutschland abbildet, in die PostgreSQL-Datenbank importiert. Daraufhin wurde ein minimales umgebenes Rechteck (MUR) mit der SQL-Query 8.1 auf das Multipolygon gebildet (siehe Abbildung 8.2). Die SQL-Query wurde in Python ausgeführt und es wurden PostGIS-Funktion in der SQL-Query verwendet, um dies zu bewerkstelligen. Die Funktion „`st_envelope()`“ bildet aus dem MultiPolygon (Deutschland) das minimale umgebene Rechteck (MUR).

```
c.execute('create table ' + mbr_xyz +
        ' as SELECT lon2tile(ST_X((ST_DumpPoints(st_envelope(geom))).geom), '
        + str(z) + ') x, ' +
        'lat2tile(ST_Y((ST_DumpPoints(st_envelope(geom))).geom), ' + str(z) +
        ') y, ' + str(z) + ' as z FROM '
        + tbl_name_completed + ';'')
```

Listing 8.1: Bildung von MUR für Deutschland und Extraktion der Koordinaten von Eckpunkten, um eine Umwandlung in XYZ-Werte zu ermöglichen

Mit der „`ST_DumpPoints()`“-Funktion werden die Koordinaten, der Eckpunkte vom MUR extrahiert. Die Funktionen „`ST_X()`“ und „`ST_Y()`“ extrahieren die geografischen Koordinatendaten in Breitengrad (latitude) und Längengrad (longitude). Mit den Formeln in (8.1a)

bzw. mit der Funktion „lon2tile()“ (wurde von der Webseite¹ bezogen), in SQL lassen sich jeweils die X- und Y-Werte anhand der Breitengrad (latitude) und Längengrad (longitude) mit einem Z-Wert von 14 berechnen.

Der Z-Wert wurde für die Werte von 0 - 18 in der Tabelle auf der Webseite² definiert. Bei 0 beträgt 1 cm 5.547 km. Für jeden Anstieg des Z-Wertes um 1, halbiert sich die Reichweite. Bei einem Z-Wert von 1 beträgt die Reichweite $5.547 \text{ km} : 2 = 2.773,5 \text{ km}$ [61] (in der Tabelle wird die Nachkommastelle abgeschnitten, also 2773 km). Laut API von accessibility.cloud kann ein Z-wert erst ab 5 genutzt werden. Beim Testen des Zoomlevels hat sich gezeigt, dass die API erst ab einem Zoomlevel von 11 oder höher genutzt werden kann. Bei Z-Werten unter 11, benötigt die accessibility.cloud für die Bearbeitung der Anfrage zu lange und es kommt zur einer HTTP-Fehlermeldung („Error 503 first byte timeout“).

Der eingesetzte Z-Wert wurde durch mehrere Tests ermittelt, sodass jede Anfrage an accessibility.cloud maximal 1.000 POIs-Datenergebnisse liefert, da die API von accessibility.cloud nicht dafür ausgelegt ist, Massendaten zu exportieren. Denn bei Suchanfragen an accessibility.cloud wird als Default maximal 1.000 Points of Interest ausgegeben, sogar dann, wenn die Anfrage mehr als 1.000 Ergebnisse liefern sollte. Durch den Parameter („limit“) kann die Anzahl der Ergebnisse erhöht werden. Es wird nicht empfohlen, mehr als 1.000 Points of Interest pro Anfrage anzugeben, da die Daten nicht im Cache gehalten werden und ein performanter Export der Daten nicht möglich ist. Zusätzlich ist die Wahrscheinlichkeit hoch, dass eine „timeout“ HTTP-Fehlermeldung auftritt.

$$x = \left[\frac{lon + 180}{360} * 2^z \right] \quad y = \left[\left(1 - \frac{\ln\left(\tan\left(lat * \frac{\pi}{180}\right) + \frac{1}{\cos\left(lat * \frac{\pi}{180}\right)}\right)}{\pi} \right) * 2^{z-1} \right] \quad (8.1a)$$

Nachdem die X- und Y-Werte in der PostgreSQL-Datenbank berechnet wurden, werden die Daten in einer CSV-Datei (00_Germany_mbr_xyz.csv) abgespeichert.

¹https://wiki.openstreetmap.org/wiki/Slippy_map_tilenames#ECMAScript_.28JavaScript.2C.2FActionScript.2C.etc..29.

²https://wiki.openstreetmap.org/wiki/Slippy_map_tilenames#ECMAScript_.28JavaScript.2C.2FActionScript.2C.etc..29.

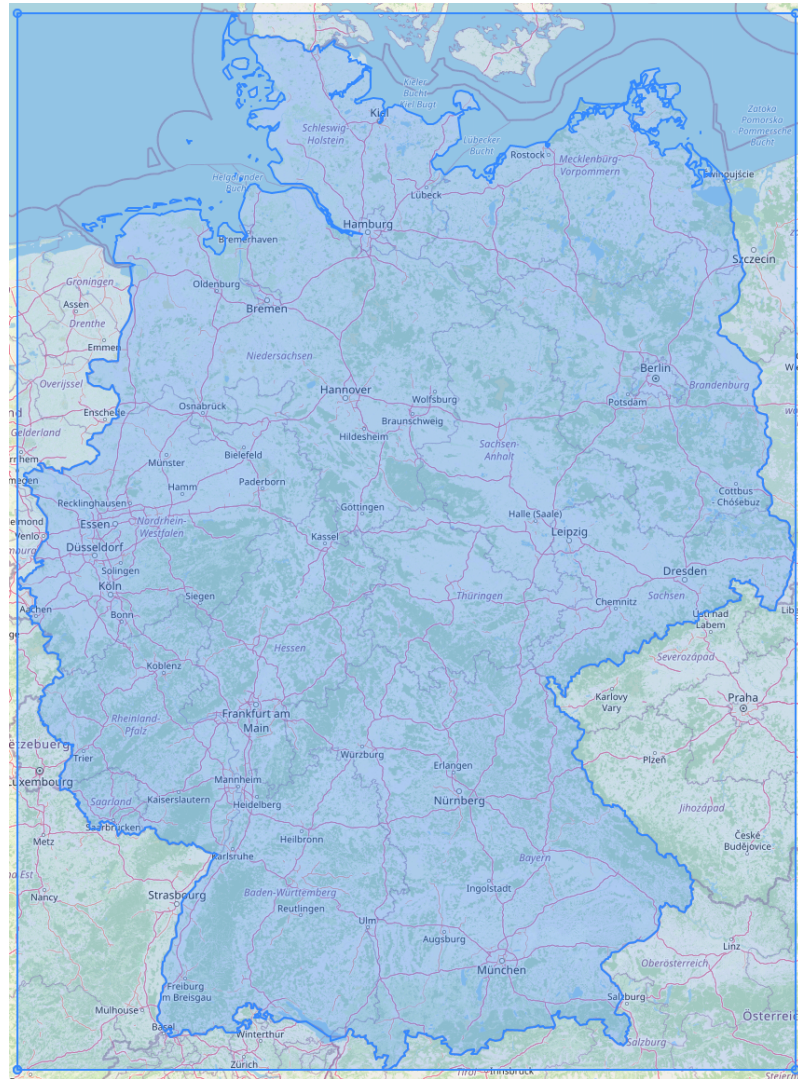


Abbildung 8.2: Duetschland: Minimal umgebendes Rechteck (MUR)

Mit der Funktion „get_poi_data-from_cloud_xyz()“ und den X- und Y-Werten der Eckdaten von Minimal umgebenes Rechteck (MUR), die aus der CSV-Datei (00_Germany_mbr_-xyz.csv) stammen, werden mit einer verschachtelte For-Schleife (siehe 8.2) über jede Kachel iteriert und eine URL (siehe Lis. 8.3) erzeugt und ausgeführt. Dabei führt eine Erhöhung des X-Wertes um 1 zu einem vertikalen Anstieg und bei einer Erhöhung des Y-Wertes um 1 erfolgt eine Kachelverschiebung nach rechts.

```
for xx in range(min(df['x']), max(df['x'])):
    for yy in range(min(df['y']), max(df['y'])):
        [...]
```

Listing 8.2: verschachtelte For-Schleife, um durch alle Kacheln zu iterieren

Der Response von accessibility.cloud wird als GeoJSON-Format dargestellt (siehe Listing 8.4). Alle Responses werden in einer JSON-Datei (01_accessibility_cloud_germany_pois_data.json) abgespeichert.

```
1 | accessibility-cloud.freetls.fastly.net/place-infos.json?appToken=[TOKEN]&x=[X]
   &y=[Y]&z=[Z]
```

Listing 8.3: Beispiel: Rastersuchanfrage an accessibility.cloud

```
1 | {"type": "FeatureCollection", "featureCount": 2, "totalFeatureCount": 2, [...]}
2 | "features": [
3 |   {"type": "Feature",
4 |     "geometry": {"type": "Point", "coordinates": [6.6416187, 53.5934008]},
5 |     "properties": {
6 |       "originalId": "471099238",
7 |       "infoPageUrl": "https://wheelmap.org/nodes/471099238",
8 |       "category": "attraction",
9 |       "name": "Seehundsbank",
10 |      "accessibility":
11 |        {"accessibleWith":
12 |          {"wheelchair": false, "wheelchairLocalized": "Rollstuhl"},
13 |          "partiallyAccessibleWith": {"wheelchair": true,
14 |            "wheelchairLocalized": "Rollstuhl"},
15 |          "accessibleWithLocalized": "Zugänglich mit",
16 |          "partiallyAccessibleWithLocalized": "teilweise zugänglich mit"},
17 |        "sourceId": "LiBTS67TjmBcXdEmX",
18 |        "sourceImportId": "rp34Lwek8Fhtmkuqf",
19 |        "_id": "5xrAP44GNzdpjem7q",
20 |        "localizedCategory": "Attraktion"}}], [...]}
21 | ]}
```

Listing 8.4: Beispiel: Response von accessibility.cloud

Ab und zu kommt es vor, dass Anfragen an accessibility.cloud fehlschlagen. Mit einer Error Exception Handling wird der Fehler abgefangen und die Anfrage wiederholt. Daraufhin werden mit der Funktion „extract_data_from_cloud()“: alle Metadaten entfernt und in einer JSON-Datei (02_accessibility_cloud_germany_pois_data_extract.json) abgespeichert. Wie in der Listing 8.4, erhalten die Response keine Adressdaten und eine Volltextsuche ist dadurch nicht möglich. Deshalb werden mit der Funktion „get_address_data_from_nominatim()“ Adressdaten zu jedem einzelnen Point of Interest mit der Nominatim-API ergänzt. Laut der Richtlinie von Nominatim [62] ist es nur erlaubt eine Anfrage pro Sekunde zu senden, ansonsten wird die IP-Adresse für eine bestimmte Zeit gesperrt. In der JSON-Datei befinden sich ca. 500.000 Points of Interest und es hat mehrere Tage in Anspruch genommen, um die Daten zu beschaffen. Da sehr viele Anfragen durchgeführt werden mussten, wurden die Anfragen aufgesplittet und später in einer JSON-Datei (03_nominatim_germany_address_data.json) zusammengeführt. Wenn eine Anfrage fehlschlug, wurde der Fehler aufgefangen und die Anfrage wiederholt, wie es bei der Beschaffung der POI-Daten von accessibility.cloud der Fall war.

Mit der Funktion „get_categories_from_cloud()“ wurden die POI-Daten mit Oberkategorien ergänzt, indem die API im Listing 8.5 die Anfrage an accessibility.cloud ausführt und als Response erhält (siehe Listing: 8.6). Die Daten werden ebenfalls als JSON-Datei (cloud_

category.json) abgespeichert. Für jede Oberkategorie (parentIds) wurden die jeweiligen Unterkategorien (icon) herausgefiltert und in einer CSV-Datei (cloud_category.csv) abgelegt.

```
1 | accessibility-cloud.freetls.fastly.net/categories.json?appToken=[TOKEN]
```

Listing 8.5: Kategorienanfrage an accessibility.cloud

```
1 | {
2 |   "count":177,
3 |   "totalCount":177,
4 |   "related":{},
5 |   "results":[{
6 |     "_id":"2nd_hand",
7 |     "icon":"2nd_hand",
8 |     "parentIds":["shopping"],
9 |     "translations":{"_id":{"de":"Second-Hand","en":"Vintage store"}},
10 |    "synonyms":["shop=second_hand","Thrift / Vintage Store",
11 |               "second_hand","Second-hand Shop","600-6900-0251"]
12 |   }, [...]
```

Listing 8.6: Response zur Kategorienanfrage an accessibility.cloud

Die Funktion „merge_clean_and_translate_data()“ ist für das Zusammenführen der Adressdaten und Kategorien zu den jeweiligen POI-Daten im JSON-Format zuständig, die kompatibel mit dem Elasticsearch-Mapping sind und als JSON-Datei (04_germany_pac.json) abgespeichert wird. Zuerst wurden alle Oberkategorien ins Deutsche übersetzt, damit eine Volltextsuche auf Deutsch möglich ist. Das Feld („restrooms“) bzw. Toilette von accessibility.cloud wurde angepasst bzw. vereinfacht (vorher bestand das Feld aus verschachtelten JSON-Arrays). Das Feld („restrooms“) kann nun folgende Begriffe annehmen:

- **true:** Die Toilette ist barrierefrei.
- **false:** Eine Toilette ist im Gebäude vorhanden, aber die Toilette ist nicht barrierefrei.
- **unknown:** Eine Toilette ist vorhanden, aber es ist nicht bekannt, ob die Toilette barrierefrei ist.
- **no:** In diesem Gebäude befinden sich keine Toiletten.

Zusätzlich erfolgten noch einige Bereinigungsschritte, wie beispielsweise das Anpassen von NULL-Werten. In der Funktion wird die JSON-Struktur für die Dokumente festgelegt (siehe Kapitel Design: Datenstruktur), sodass ein Import der Daten in Elasticsearch möglich ist.

Im 1. Skript werden noch zusätzlich alle Straßendaten (Adressdaten) in Deutschland durch Anfragen an die Photon-API bezogen. Die Photon-API bietet dafür Bereichsanfragen an. So ist es möglich mehrere Adressdaten (maximal 6.666) in einer Anfrage zu erhalten. Die Funktion „get_address_data_from_photon()“ beschafft sich alle Straßendaten aus Deutschland durch die Photon-API. Zuerst wurden alle PLZ in Deutschland als TXT-Datei von der Webseite www.suche-postleitzahl.org heruntergeladen. Dann wurde jeweils für jede PLZ eine URL-Anfrage erzeugt (PLZ können mehrfach vorkommen, eine Response einer HTTP-Anfrage

kann Straßendaten von unterschiedlichen Städten ausgeben). Mit der „extract_address_data_from_photon()“ wurden alle Metadaten entfernt und als JSON-Datei abgespeichert.

8.2 2. Skript: Datentransformation und -aufbereitung

POI-Daten und Straßendaten liegen zurzeit als JSON-Datei vor. Die Funktion „transform_data_and_import_to_postgres()“ transformiert beide Dateien in eine tabellarische Darstellung, sodass ein Import der beiden Tabellen in die PostgreSQL-Datenbank möglich ist. Bei den Straßendaten von Photon musste eine Umlaut-Konvertierung durchgeführt werden. Die Funktion „execute_sql_script()“ ermöglicht die Ausführung von SQL-Dateien in der PostgreSQL-Datenbank sowie den Export von Tabellen aus der PostgreSQL-Datenbank in CSV-Dateien. Mit der SQL-Query in Lis. 8.7 wurden die Attribute („wheelchair“) und („partiallyAccessibleWith“) zum Attribut („accessibleWith“) zusammengefasst und mit folgenden Werten klassifiziert:

- 1: POI ist vollständig barrierefrei
- 2: POI ist teilweise barrierefrei
- 3: POI ist nicht barrierefrei
- 4: Barrierestatus von POI ist unbekannt

Durch die Nummerierung können Aggregationsanfragen in Elasticsearch leichter angewendet werden. Zusätzlich wurden mit der SQL-Query noch alle Parkplätze entfernt, da die Bezeichnung von POIs für Parkplätze für die Volltextsuche irreführend ist, z.B. wird ein Parkplatz als „Aldi Parkplatz“ gelistet, wenn bei einer Volltextsuche „Aldi“ eingegeben wird.

```

1 SELECT _id, _index, "address.city", "address.city_district", "address.country"
   , "address.housenumber", "address.state", "address.street", "address.
   postcode", de_category, "de_parentCategory", description, en_category, "
   en_parentCategory", "infoPageUrl", "geometry.type", cast("geometry.
   coordinates" AS real[]) AS "geometry.coordinates", CASE WHEN name like '
   THEN 'Unbekannt' ELSE name end AS name, "originalId", "placeWebsiteUrl",
   restrooms, "sourceId", "sourceImportId", CASE WHEN "accessibleWith" LIKE '
   true' THEN '1' ELSE CASE WHEN "partiallyAccessibleWith" LIKE 'true' THEN '
   2' ELSE CASE WHEN "accessibleWith" LIKE 'false' OR "accessibleWith" LIKE '
   false' THEN '3' ELSE '4' END END END AS "accessibleWith"
2 INTO germany_poi_clean
3 FROM germany_poi
4 WHERE "address.country"
5 LIKE 'Deutschland' AND de_category NOT LIKE 'Parkplatz/Parkhaus';

```

Listing 8.7: 1. SQL Query, um 2 Attribute zu einem Attribut („accessibleWith“) zusammenzuführen

Weitere SQL-Queries wurden ausgeführt, um NULL-Werte in „Unbekannt“ in beiden Tabellen umzubenennen, da Elasticsearch NULL Werte nicht indexiert und dadurch eine Suche bzw. Aggregation nicht möglich ist [63]. Weiterhin war ein Export in Elasticsearch mit den vorigen NULL-Werten nicht möglich. Da Straßendaten in Photon kaum Hausnummern vorliegen,

wird nur eine Hausnummer für jede Straße mit der SQL-Query 8.8 gefiltert. Durch ein Ranking mit der Gruppierung nach Postleitzahl und Straßen, mit einer Sortierung nach „osm_type“. Für das „osm_type“ gibt es drei verschiedene Elemente:

- **Way (W)**: Ein Weg ist eine sortierte Liste, die 2 bis 2000 Knoten umfasst und eine Polylinie abbilden soll, wie Flüsse und Straßen.
- **Node (N)**: Ein Knoten repräsentiert einen genauen Punkt auf der Erdoberfläche durch eine geographische Koordinate, wie beispielsweise Gebäude.
- **Relation (R)**: Eine Beziehung ist eine Datenstruktur, die die Beziehung zwischen zwei oder mehreren Datenelementen, wie beispielsweise Knoten und Wegen abbildet.

Jeder Datensatz in Photon wird im Attributfeld „geometry.coordinates“ als eine geografische Koordinate abgebildet. Koordinaten von Wegen befindet sich häufig auf der Straße oder in der näheren Umgebungen von Straßen. Nodes repräsentieren häufig Gebäude und die Koordinaten von Relationen sind am häufigsten von den abgebildeten Straßen entfernt. Deshalb werden bei identischen Straßen zuerst Koordinaten von Wegen dann Knoten und zum Schluss Relation gewählt. Der Adressdatensatz von Photon betrug 7.187.037 Datensätze vorher und nach der Filterung waren es nur noch 1.490.174 Datensätze, d.h. im Durchschnitt hat eine identische Straße jeweils 5 Koordinaten. Deshalb ist sinnvoll, für jede Straße nur eine Koordinate zu wählen. Die Suchmaschine Photon ignoriert dabei auch Hausnummern und es können auch nur nach Straßennamen gesucht werden.

```

1 With cte_address AS (
2 SELECT *, row_number() over(PARTITION BY "address.city", "address.postcode", "
   address.street" ORDER BY CASE "address.osm_type" WHEN 'W' THEN 1 WHEN 'N'
   THEN 2 ELSE 3 END) AS rank
3 FROM germany_address2)
4 SELECT "_id", "_index", "geometry.coordinates", "geometry.type", "address.city",
   "address.country", "address.extent", "address.housenumber", "address.
   name", "address.osm_value", "address.osm_key", "address.osm_type", "address
   .postcode", "address.state", "address.street"
5 INTO germany_address3
6 FROM cte_address
7 WHERE rank = 1;

```

Listing 8.8: SQL Query, um für jede Straße nur eine Koordinate zu wählen

8.2.1 Ergänzung der Adressdaten von POIs

Mit einer SQL-Query wurde überprüft, wie vollständig die Adressdaten von Nominatim und Photon sind (siehe Tab. 8.1). In der Tabelle von Nominatim befinden sich 477.772 Adressdaten und in der Tabelle von Photon liegen 7.187.037 Datensätze vor. In der Tabelle 8.1 ist zu erkennen, dass bei Nominatim die Daten beim Attribut PLZ nur zu 43.71 Prozent vollständig sind. Die Daten von Photon hingegen sind bei den Attributen Bundesland und Stadt fast vollständig.

Tabelle 8.1: Datenvollständigkeit der POI-Daten in Deutschland in Prozent

Geotool	Land	Bundesland	Stadt	Stadtteil	PLZ	Straße	Hsnr.
Nominatim	100	91.72	43.71	32.13	99.83	98.86	68.05
Photon	100	99.99	96.77	0	99.09	41.90	18.97

Mit den beiden SQL-Queries (Listing: 8.9, 8.10) wird versucht, die Daten für die Attribute Bundesland und Stadt in Nominatim zu vervollständigen. Bei der SQL-Query 8.9 werden mit einem Join der beiden Tabellen „germany_poi_clean“ (Adressdaten von Nominatim) und „germany_address2“ (Adressdaten von Photon) und mehreren PostGIS-Funktionen, die Daten von Nominatim durch die Adressdaten ergänzt. Zusätzlich werden noch für jede Adresse von Nominatim innerhalb von 100 m alle Adressdaten mit der gleichen Postleitzahl von Photon ausgegeben. Da die Query sehr lange benötigt, wurde eine kleine Radiussuche von 100 m gewählt, dadurch benötigt die Query ca. 15 Minuten. Bei der 2. SQL-Query (siehe Lis. 8.10) wird für jede POI-Adresse die nächste Adresse von Photon selektiert und mit den Daten werden die Adressdaten von Nominatim bzw. die Adresse von den POI ergänzt. Das gleiche Verfahren wird nochmals mit einer Reichweite von 5000 m wiederholt. Es werden dabei nur Adressdaten von Nominatim mit noch nicht ausgefüllten Attributen, bei denen in der SQL-Query (8.9) keine Zuordnung einer Adresse stattfand, selektiert.

```

1 CREATE TABLE germany_poi_add_geodata AS
2 SELECT A.*, B."address.city" addresscity, B."address.state" addressstate,
   st_distance(st_setsrid(st_makepoint(A."geometry.coordinates"[1], A."
   geometry.coordinates"[2]),4326)::geography, st_setsrid(st_makepoint(B."
   geometry.coordinates"[1],B."geometry.coordinates"[2]),4326)::geography) AS
   distance
3 FROM germany_poi_clean A
4 LEFT JOIN germany_address2 B
5 ON A."address.postcode" = B."address.postcode"
6 WHERE st_distance(st_setsrid(st_makepoint(A."geometry.coordinates"[1], A."
   geometry.coordinates"[2]), 4326)::geography, st_setsrid(st_makepoint(B."
   geometry.coordinates"[1],B."geometry.coordinates"[2]),4326)::geography) <
   100;

```

Listing 8.9: 1. SQL Query, um Adressdaten in den POI-Daten zu ergänzen

```

1 CREATE TABLE germany_poi_add_geodata_incomplete AS
2 WITH cte AS(
3 SELECT *, row_number() over(PARTITION BY _id ORDER BY distance asc) AS rank
4 FROM germany_poi_add_geodata)
5 SELECT _id, _index, CASE WHEN "address.city" LIKE 'unknown' AND addresscity IS
   NOT NULL THEN addresscity ELSE "address.city" END AS "address.city", "
   address.city_district", "address.country", "address.housenumber", CASE
   WHEN "address.state" LIKE 'unknown' AND addressstate IS NOT NULL THEN
   addressstate ELSE "address.state" END "address.state", "address.street", "
   address.postcode", "de_category", "de_parentCategory", "description",
   "en_category", "en_parentCategory", "infoPageUrl", "geometry.type", "
   geometry.coordinates", "name", "originalId", "placeWebsiteUrl", "restrooms", "
   sourceId", "sourceImportId", "accessibleWith"
6 FROM cte

```

```
7 | WHERE rank = 1;
```

Listing 8.10: 2. SQL Query, um Adressdaten in den POI-Daten zu ergänzen

Nach der Ergänzung der Daten wurde nochmals mit einer SQL-Query überprüft, wie vollständig die Adressdaten der Points of Interest sind (siehe Tab. 8.2). Es ist zu erkennen, dass die Attribute für Land, Bundesland, Stadt, PLZ und Straße fast vollständig sind. Bei Hausnummer und bei Stadtteil besteht Nachholbedarf. Durch das Ergänzen der Daten konnte die Volltextsuche in Elasticsearch um ein Wesentliches verbessert werden. Beide Tabellen werden als CSV-Dateien (05_germany_poi_complete.csv und germany_address_complete_unique.csv) abgespeichert.

Tabelle 8.2: Nach der Verarbeitung der POI-Daten in Prozent

Geotool	Land	Bundesland	Stadt	Stadtteil	PLZ	Straße	Hsnr.
Nominatim	100	99.99	99.77	32.13	99.83	98.86	68.05

8.3 3. Skript: Datentransformation und -aufbereitung

Zu Beginn werden die Daten mit der Funktion „transform_data_to_json()“ wieder in das JSON-Format transformiert. Mit der Funktion „import_to_elasticsearch()“ werden die POI-Daten von accessibility.cloud und die Straßendaten mit dem vordefinierten Schema (siehe Kapitel Design) in die Elasticsearch-Datenbank importiert.

8.3.1 Standardisierung der Begriffe/Bezeichnung von POIs

Da die Points of Interest von Nutzern der Wheelmap befüllt wurden, ist die Datenqualität bei der Bezeichnung der POIs nicht ausreichend. Aus diesem Grund wurde ein Verfahren entwickelt, um die Bezeichnung zu normalisieren. Um das Problem zu verdeutlichen, wurde mit dem Tool Dejavu nach dem Begriff McDonald's gesucht (siehe 8.3). Es ist in der Abbildung zu erkennen, dass McDonald's in unterschiedlicher Schreibweise vorliegt.



Abbildung 8.3: Suche nach dem Begriff „McDonald's“

Die Funktion „normalize_name_from_poi_in_elasticsearch_1()“ führt zuerst eine Aggregationsanfrage in Elasticsearch mit den folgenden Feldern aus: („name.keyword“), („de_category.keyword“), („de_parentCategory“). Die Daten von der Response von Elasticsearch werden extrahiert und in eine tabellarische Darstellung gebracht (siehe Tab. 8.3). In der Tabelle ist zu erkennen, dass zum Beispiel Lidl unterschiedlich kategorisiert ist (Supermarkt, Spätkauf) und dass Lidl großgeschrieben wird, was nicht so sein sollte. Es wird nach „beforeCount“ sortiert, „beforeCount“ gibt an, wie häufig der Begriff bzw. die Bezeichnung des POI („name“) im Index vorliegt. Diese Tabelle wird als CSV-Datei (category_list_germany_before.csv) abgespeichert.

Tabelle 8.3: Aggregationstabelle

name	parentCategory	beforeCount	category1	category2	categoryCount1	categoryCount2
Netto	Einkaufen	2184	Supermarkt	Spätkauf	2171	6
Lidl	Einkaufen	2139	Supermarkt	Spätkauf	2125	6
...	...	...	...	...	...	...
LIDL	Einkaufen	334	Supermarkt	Spätkauf	332	1

Die Normalisierung der Begriffe wird nur auf die folgenden Oberkategorien angewendet: Einkaufen und Essen. Bei anderen Oberkategorien ist es nicht sinnvoll eine Normalisierung der Begriffe nach diesem Verfahren durchzuführen, da dadurch Information verloren gehen können, wie beispielsweise bei „Sparkasse Geldautomat in Darmstadt“. Durch dieses Verfahren wird der POI nur noch als „Sparkasse“ bezeichnet, was nicht erwünscht ist. Bei Bezeichnungen in denen die Begriffe wie Döner, Bäckerei, Imbiss oder Eiscafe vorkommen, werden ebenfalls von der aggregierten Tabelle entfernt, um Informationsverluste zu vermeiden. Mit der Funktion „normalize_name_from_poi_in_elasticsearch_2()“ erfolgt eine Umbenennung der Begriffe bzw. Vereinheitlichung der Bezeichnung der POIs. Es werden die 150 häufigsten Begriffe, die sich in der Tabelle befinden, umbenannt. Die 150 Begriffe wurden nach Sichtung der Tabelle bewusst so gewählt, da nur häufige Begriffe vereinheitlicht werden sollten. Zuerst werden die vorherigen Begriffe der POIs, die sich in Elasticsearch befinden, mit der Ausführung der Anfrage „searchquery“ (siehe Listing 8.11) für jeden Begriff als CSV-Datei abgespeichert, damit später, wenn die Daten normalisiert sind, die vorigen Begriffe mit den upgedateten Daten verglichen werden können.

Die Iteration der While-Schleife beginnt bei 150. Die Annahme ist, dass die Bezeichnung der POIs, die „falsch“ sind, seltener auftreten, als die, die „korrekt“ geschrieben sind, wie es in der Tabelle 8.3 bei Lidl der Fall ist. Es gibt insgesamt 2.134 Lidl, die korrekt geschrieben sind und nur 334, die falsch geschrieben wurden. Elasticsearch bietet mit der Funktion „update_by_query“ eine komplexe Suche- und Ersetzfunktion. Mit der Funktion ist es möglich die Bezeichnungen der POIs zu normalisieren. Es wurde eine „multi_match“-Anfrage bei der „update_by_query“ verwendet. Die Volltextanfrage bietet eine „minimum_should_match“-Funktion an. Bei der „minimum_should_match“-Funktion ist eine pessimistische Filterung möglich, d.h. wenn der Begriff nicht vollständig in der Bezeichnung der POIs vorliegt, wird das Dokument im Suchergebnis ausgeschlossen. Der „Standard“-Analyzer wird als Default

gewählt, wenn kein Analyzer vorgegeben wird. Die Funktion „fuzziness“ verwendet den Levenshtein-Distanz-Algorithmus. Dies ermöglicht eine unscharfe Suche. Ein Fuzziness-Wert von 1 bedeutet, dass 1 Zeichen bei der Suche nicht identisch sein muss, damit es in den Suchergebnissen ausgegeben wird. Wenn z.B. nach dem Begriff „McDonalds“ gesucht wird, dann wird auch der Begriff „McDonald’s“ ausgegeben. Der Fuzziness-Wert ist dynamisch, wenn der Begriff bzw. die Texteingabe weniger als 5 Zeichen hat oder ein Leerzeichen vorliegt, wird ein Fuzziness-Wert von 0 gewählt, ansonsten wird ein Fuzziness-Wert von 2 gewählt. Diese Werte wurden durch viele Tests ermittelt. Bei „most_fields“, die bei der „update_by_query“ verwendet wird, werden die Suchbegriffe mit jedem einzelnen Feld verglichen und der Score-Wert addiert und durch die Anzahl der Felder geteilt [64]. Da aber nur ein Feld („name“) verwendet wird, ist der Score identisch wie bei einer einfachen „query“-Anfrage.

```

while (0 <= i):
    searchquery = {
        "size": 10000,
        "_source": "name",
5       "query": {"bool": {"must": {"multi_match": {
            "query": count_category_df3['name'][i],
            "analyzer": "standard",
            "fuzziness": fuzziness(count_category_df3, i),
            "fields": ["name"],
10          "minimum_should_match": "100%",
            "type": "most_fields"}}},
            "filter": [{"term": {"de_parentCategory.keyword":
                                count_category_df3['parentCategory'][i]
                                }}}]},
        "sort": [{"_score": {"order": "asc"}}]
15    }
    [...]
    update_by_query = {
        "script": {"source":
            "ctx._source.name = \"\" + count_category_df3['name'][i] + \"\""},
20    "query": {"bool": {"must": {"multi_match": {
        "query": count_category_df3['name'][i],
        "analyzer": "standard",
        "fuzziness": fuzziness(count_category_df3, i),
        "fields": ["name"],
25      "minimum_should_match": "100%",
        "type": "most_fields"}}},
        "filter": [{"term": {"de_parentCategory.keyword":
                            count_category_df3['parentCategory'][i]}}}]}
30    }
    [...]

```

Listing 8.11: Verfahren zur Normalisierung der Begriffe von POIs

8.4 4. Skript: Implementierung der DSL-Anfragen

Im 4. Skript werden alle benötigten Anfragen für Elasticsearch implementiert und als Searchtemplate in der Elasticsearch-Datenbank abgespeichert.

8.4.1 Volltextsuche für Points of Interest und Straßendaten

Bei der Volltextsuche für die Points of Interest (siehe Lis. 8.12) wurde eine „multi_match“-Anfrage mit der Type „cross_field“ verwendet, um nach Begriffen, die sich in mehreren Feldern befinden, zu suchen. Diese Suche eignet sich besonders gut für strukturierte Dokumente. Die Type „cross_field“ zerlegt die Sucheingabe in einzelne Terme/Wörter und sucht in allen Feldern, die in „fields“ vorliegen, als wäre es ein großes Feld [65]. Mit dem Operator „and“ müssen die Wörter mindestens in einem dieser „fields“ vorkommen, sonst wird kein Suchergebnis ausgegeben. Mit dem „special_multimatch_autocomplete_ngram_analyzer“ ist es möglich, mit unvollständigen Wörtern, bei der Sucheingabe, gute Suchergebnisse zu erzielen. Beispielsweise reicht „McD Darmst“ aus, um alle „McDonald’s in Darmstadt“ im Suchergebnis zu listen. Eine Autovervollständigung ist durch das Vorliegen von mehreren Feldern nur beschränkt möglich. Bei „search“ erfolgt die Sucheingabe, die beim Aufruf des Searchtemplates übergeben wird. Da sehr viele Daten vorliegen, wurden Filteroptionen in die Suchanfrage integriert. Die Eingabe im Parameterfeld „access.values“ ist verpflichtend einzutragen. Das Feld ist für die Filterung von Barriereeinschränkungen von Gebäuden zuständig. Eine Filterung nach Toiletten ist optional möglich.

```

1 POST [HOST:PORT]/_scripts/POIFulltextsearchAutocomplete
2 {
3   "script": {
4     "lang": "mustache",
5     "source": "" {
6       "query": {
7         "bool": {
8           "must": [
9             {{#search}}
10            {
11              "multi_match": {
12                "query": "{{search}}",
13                "fields": ["name.autocomplete_ngram", "de_category.autocomplete",
14                  "de_parentCategory.autocomplete", "address.street", "address.
15                    city_district.autocomplete", "address.postcode.autocomplete",
16                    "address.city.autocomplete", "address.state.autocomplete", "
17                    address.country.autocomplete"],
18                "analyzer": "special_multimatch_autocomplete_ngram_analyzer",
19                "type": "cross_fields",
20                "operator": "and",
21                "minimum_should_match": "95%"
22              }
23            }
24            {{/search}}
25          ],
26          "filter": [
27            {{#access}}
28            {{#terms": {"accessibleWith": {{#toJson}}access.values{{/toJson}}}}
29            {{/access}}
30            {{#toilet}}
31            ,{{"terms": {"restrooms": {{#toJson}}toilet{{/toJson}}}}
32            {{/toilet}}
33          ]
34        }
35      }
36    }
37  }

```

```

34 }
35 }

```

Listing 8.12: Volltextsuche nach POIs

Eine Volltextsuche nach Straßendaten kann mit dem Searchtemplate 8.13 ausgeführt werden. Wie in „fields“ zu erkennen ist, ist eine Anfrage nach Hausnummer nicht möglich, da Hausnummer nicht im „fields“-Array gelistet ist. Bei der Entwicklung der Query wurde mit Hausnummer getestet. Falls die gesuchte Hausnummer nicht vorhanden ist, wird kein Suchergebnis ausgegeben, obwohl der Name der Straße im Datensatz vorliegt. Das liegt daran, dass eine „cross_fields“-Funktion mit einem „and“ Operator verwendet wird. Dieses Problem lässt sich mit ein wenig Aufwand lösen. Da jede Straße nur einmal abgebildet wird, ist es jedoch nicht sinnvoll nach Hausnummern zu suchen.

```

1 POST [HOST:PORT]/_scripts/AddressFulltextsearchAutocomplete
2 {
3   "script": {
4     "lang": "mustache",
5     "source": {
6       "query": {
7         "multi_match": {
8           "query": "{{search}}",
9           "fields": ["address.street.autocomplete", "address.
10                    city_district.autocomplete",
11                    "address.postcode.autocomplete", "address.city.
12                    autocomplete",
13                    "address.state.autocomplete", "address.country.
14                    autocomplete"],
15           "analyzer": "special_multimatch_autocomplete_ngram_analyzer",
16           "type": "cross_fields",
17           "operator": "and",
18           "minimum_should_match": "90%"
19         }
20       }
21     }
22   }
23 }

```

Listing 8.13: Volltextsuche nach Straßendaten

8.4.2 Radiussuche für Points of Interest

Bei der Radiussuche für die POI-Daten sind verpflichtend eine geografische Koordinate und eine Radiuslänge in Metern an die Query zu übergeben. Es soll die Koordinate vom Standort des Anwenders übergeben werden. Die Radiussuche bietet mit dem Analyzer „standard_autocomplete_ngram_analyzer“ eine Autovervollständigung an, die nach Eingabe eines Zeichens im Suchfeld Vorschläge macht. Es ist nur möglich nach Bezeichnung des Points of Interest zu suchen, wie beispielsweise „Aldi“ oder „McDonald’s“. Jedoch muss im Suchfeld keine Sucheingabe erfolgen, dafür ist der Befehl zuständig [#search ... /search]. Wird keine Sucheingabe übergeben, wird das Feld, indem der Befehl steht ignoriert. Diese

Art der Verschachtelung wird auch bei den folgenden Filterparameter: „access“, „toilet“, „de_category“ und „de_parentCategory“ angewendet. Bei der Radiussuche ist es auch möglich anhand von Filterungen nach Ober- und Unterkategorien gezielte Points of Interest auszugeben. Die Query sortiert nach der Entfernung der Points of Interest vom aktuellen Standort des Anwenders (Entfernung erfolgt über Luftlinie).

```

1 POST [HOST:PORT]/_scripts/radiussearchAutocomplete
2 {"script": { "lang": "mustache",
3 "source": ""
4 {
5   "query": {
6     "bool": {
7       "must": [
8         {{#search}}
9         {"match": {
10           "name.autocomplete_edge_ngram": {
11             "query": "{{search}}",
12             "analyzer": "radius_autocomplete_edge_ngram_analyzer"
13           }
14         }}
15       {{/search}}
16     ],
17     "filter": [
18       {"geo_distance" : {"distance" : "{{distance}}",
19       "geometry.coordinates" : { "lat" : "{{lat}}", "lon": "{{lon}}"} }}
20       {{#access}}
21       ,{
22         "terms": {
23           "accessibleWith": {{#toJson}}access{{/toJson}}
24         }
25       }
26       {{/access}}
27       {{#toilet}}
28       ,{
29         "terms": {
30           "restrooms": {{#toJson}}toilet{{/toJson}}
31         }
32       }
33       {{/toilet}}
34       {{#de_category}}
35       ,{
36         "terms": {
37           "de_category.keyword": {{#toJson}}de_category{{/toJson}}
38         }
39       }
40       {{/de_category}}
41       {{#de_parentCategory}}
42       ,{
43         "terms": {
44           "de_parentCategory.keyword":
45             {{#toJson}}de_parentCategory{{/toJson}}
46         }
47       }
48       {{/de_parentCategory}}
49     ]
50   }
51 },
52 "sort" : {
53   "_script" : {
54     "type" : "number",

```



```
1 | accessibility-cloud.freetls.fastly.net/place-infos.json?appToken=[TOKEN]&x=[X]
   | &y=[Y]&z=[Z]
```

Listing 8.15: GraphHopper API-Anfrage, um Koordinaten und Distanzen einer gerouteten Strecke zu erhalten

```
1 | http://wheelscout.org:5424/barrierserver/route?vehicletype=wheelchair&fromlat
   | =49.87618&fromlon=8.65730&tolat=49.8728&tolon=8.6510&number-of-steps*LT*1&
   | apikey=[key]
```

Listing 8.16: GraphHopper API-Anfrage an den „WheelScout“-Server, um Koordinaten und Distanzen einer gerouteten Strecke zu erhalten

Die Koordinaten sowie die Distanzen zweier Eckpunkte lassen sich vom Response der GraphHopper API-Abfrage extrahieren. Die Daten befinden sich in einem JSONArray und müssen in einer multidimensionale Double-Array in Java gecastet werden (siehe die Variablen „coordinates“, „distance“ und „startEndCoordinates“ im Listing 8.17), was mit viel Aufwand verbunden war. Es gibt bei der gerouteten Strecke von Darmstadt nach Luisenplatz insgesamt 91 Koordinatenpunkte bzw. Eckpunkte und 75 Distanzwerte. Das liegt daran, dass Koordinatenpunkte zusammengefasst werden (siehe Distanzwert (25, 28, 41.266) in der Variable „distance“). Die Länge des Radius wurde auf 250 m festgelegt, da die Strecke von 250 m etwa die Entfernung ist, die ein Nutzer gut überwinden kann, um zu seinem gewünschten Point of Interest zu gelangen. 250 m sind somit eine sinnvolle Wahl für den Radius, kann jedoch jederzeit angepasst werden. Die Methode „selectGeoCoordinatesForRadiusSearch()“ ist für die Selektion der Koordinaten, die sich auf einer Strecke befinden zuständig, es soll mit möglichst wenig Radian die komplette Strecke abgedeckt werden, da jede Radiusanfrage an Elasticsearch-Server Zeit in Anspruch nimmt. Die Funktion summiert die Distanzen im „distance“-Array, bis die Distanz größer als die „radiusDistance“ ist. Dann wird die Position der Koordinate ermittelt, die Koordinate in die Liste hinzugefügt und die Variable „sumDistance“ wird wieder auf 0 gesetzt und der Vorgang wiederholt sich wieder bis die For-Schleife ausläuft. Die „radiusDistance“ lässt sich wie folgt berechnen:

$$\text{Radiusdistanz} = \text{Radiuslänge} : \text{Faktor} \quad (8.2a)$$

Der Faktor beeinflusst die Anzahl der Radian in einer gerouteten Strecke. Falls die Strecke nicht komplett mit Radian abgedeckt ist, muss der Faktor erhöht werden. Der Faktor wurde auf 1 gesetzt.

```
5 | List<GeoPoint> selectGeoCoordinatesForRadiusSearch() {
   |
   | double[][] coordinates = {{ 8.628214, 49.872541 }, ...};
   | double[][] distance = {{ 0, 1, 36.79}, {1, 2, 32.57}, ... {25, 28, 41.277},
   |   ...};
   |
   | double[][] startEndCoordinates = {{8.628840, 49.872574}, {8.651180,
   |   49.872768}};
```

```

List<GeoPoint> listGeo = new ArrayList<GeoPoint>();
double sumDistances = 0;
10 double radiusSize = 250;
double diff = 0;
int position = 0;
double faktor = 1;
double radiusDistance = radiusSize / faktor;

15 GeoPoint geopointStart = new GeoPoint(startEndCoordinates[0][0],
    startEndCoordinates[0][1]);
GeoPoint geopointEnd = new GeoPoint(startEndCoordinates[1][0],
    startEndCoordinates[1][1]);

listGeo.add(geopointStart);
20 listGeo.add(geopointEnd);

StringBuilder sb = new StringBuilder();
sb.append("https://www.mapdevelopers.com/draw-circle-tool.php?circles=%5B%5
    B250%2C"
        + String.valueOf(geopointStart.getLat()) + "%2C" + String.valueOf(
            geopointStart.getLon())
25 + "%2C%22%23AAAAAA%22%2C%22%23000000%22%2C0.4%5D%2C%5B250%2C");

for (int i = 0; i < distance.length; i++) {
    sumDistances = sumDistances + distance[i][2];

30 diff = distance[i][1] - distance[i][0];
    position = (int) (position + diff);

    if (sumDistances > radiusDistance) {
        GeoPoint geopoint = new GeoPoint(coordinates[position][0], coordinates[
            position][1]);
35 listGeo.add(geopoint);
        sb.append(String.valueOf(geopoint.getLat()) + "%2C" + String.valueOf(
            geopoint.getLon())
            + "%2C%22%23AAAAAA%22%2C%22%23000000%22%2C0.4%5D%2C%5B250%2C");
        sumDistances = 0;
    }
40 }

sb.append(String.valueOf(geopointEnd.getLat()) + "%2C" + String.valueOf(
    geopointEnd.getLon())
    + "%2C%22%23AAAAAA%22%2C%22%23000000%22%2C0.4%5D%5D");

45 return listGeo;
}

```

Listing 8.17: Methode zur Extraktion von Koordinaten einer gerouteten Strecke

Zur Überprüfung, ob die geroutete Strecke mit den Radien komplett abgedeckt wurde, wird eine URL erzeugt, welche die Radien sowie die geroutete Strecke grafisch darstellt (siehe Abb. 8.5). Wie zu erkennen ist, wurde die komplette Strecke mit Radien abgedeckt. Es ist ersichtlich, dass sich die Radien schneiden. Dass ist auch so gewollt, ansonsten würden einige Points of Interest wegfallen. Dadurch, dass sich Radien schneiden, werden identische POI mehrfach im Response von Elasticsearch-Server vorkommen. Durch eine HashSet kann das Problem mit den redundanten POI-Daten gelöst werden. Die HashSet vergleicht, ob der Hashwert der Point of Interest schon in der HashSet vorliegt, falls ja wird er nicht in die Liste aufgenommen.

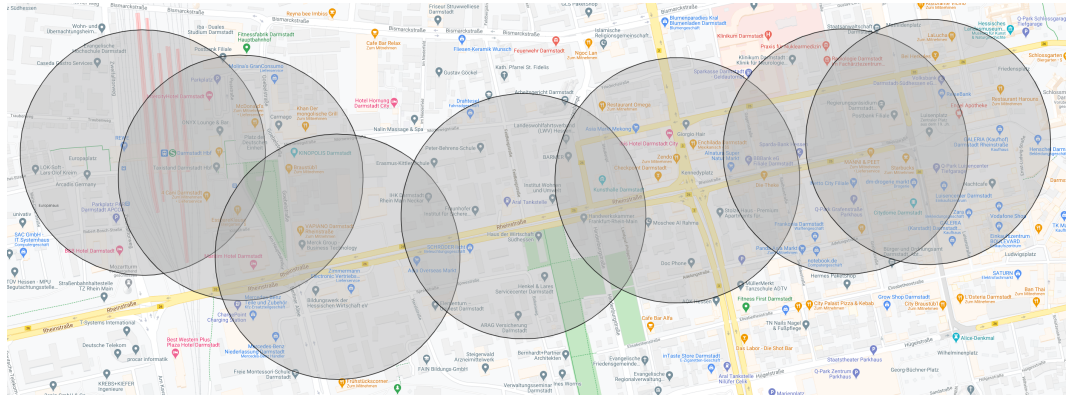


Abbildung 8.5: Radiussuche während des Routings

8.4.4 Aggregation von Feldern

Aggregationsanfragen von den Ober- und Unterkategorien (Lis. 8.18) werden benötigt, um eine Kategorienliste in die „WheelScout“-App zu integrieren. Eine Aggregationsanfrage für Geodaten, wie Land, Bundesland, Stadt, wurde auch erstellt. „size“ gibt die Anzahl an Dokumenten aus. „size“ in terms gibt die Anzahl an Ober- und Unterkategorien, die ausgegeben werden sollen, an.

```

1 POST [HOST:PORT]/_scripts/groupby_category
2 {
3   "script": {
4     "lang": "mustache",
5     "source":
6       {
7         "size": 0,
8         "aggs": {
9           "parentCategory": {
10            "terms": {"field": "de_parentCategory.keyword", "size": 100},
11            "aggs": {
12              "category": {
13                "terms": {"field": "de_category.keyword", "size": 100}
14              }
15            }
16          }
17        }
18      }
19    }

```

Listing 8.18: Aggregation von Kategorien

Im nächsten Kapitel erfolgt die Einbindung von Elasticsearch in das „WheelScout“-Projekt.

9 Einbindung von Elasticsearch in das „WheelScout“-Projekt

9.1 Integration von Elasticsearch auf dem „WheelScout“-Backend-Server

Auf dem „WheelScout“-Backend-Server werden die Karten für die „WheelScout“-App gerendert. Der Server wurde vom Anbieter Contabo gemietet und befindet sich in Nürnberg oder in München. Der Server weist folgende Spezifikationen auf (siehe Tab. 9.1):

CPU:	Intel(R) Xeon(R) CPU E5-2620 v3 @ 2.40GHz
OS:	Ubuntu 18.04 LTS
Bandbreite:	1 GB/s
Anzahl CPU:	6
Anzahl Threads:	12
Arbeitsspeicher:	128 GB
Festplatte:	256 GB SSD
Grafikkarte:	ASPEED Technology, Inc. ASPEED Graphics Family (rev 10)

Tabelle 9.1: Contabo-Server

Die Firma Elastic N.V. empfiehlt für eine performante Nutzung von Elasticsearch auf einem Server mit 128 GB Arbeitsspeicher, dass Elasticsearch auf zwei Knoten auf dem Server verteilt werden soll mit jeweils 32 GB an Arbeitsspeicher. Die restlichen 64 GB Arbeitsspeicher sollen für Lucence, d.i. eine Java Library zur Volltextsuche, zur Verfügung gestellt werden [66]. Bei einer verteilten Datenbank ist eine schnelle Bandbreite sehr wichtig. Es wird empfohlen eine Bandbreite von 1 GB/s oder 10 GB/s zu nutzen [67].

Das Aufsetzen von Elasticsearch in Docker wird via PuTTY auf dem Contabo-Server realisiert. Die Vorgehensweise war wie folgt: Zuerst wurden mit dem Tool WinSCP die Suchmaschine Elasticsearch, die Python-Skripte, der Elasticsearch-Java-Client, der POIServer und die Daten auf den Server hochgeladen [68]. Dann wurde mit PuTTY, ein Client (Tool), das eine Verbindung zu einem Server via Secure Shell (SSH) aufbaut [69], via Terminal die Suchmaschine Elasticsearch in Docker installiert und der Server gestartet. Daraufhin wurden alle benötigten Bibliotheken in Python, für die ETL-Skripte installiert. Nach der Installation der Bibliotheken wurden die ETL-Skripte in Python ausgeführt, um die Daten und das Searchtemplate in Elasticsearch einzubinden. Anschließend wurde der POIServer gestartet, um „POIs während des Routings anzeigen“-Anfragen entgegen zu nehmen.

9.1.1 Kommunikation zwischen „WheelScout“-Server und der „WheelScout“-App

Die Kommunikation zwischen dem Backend-Server, auf dem sich die Suchmaschine Elasticsearch befindet und der „WheelScout“-App erfolgt via Elasticsearch REST API und über den POIServer in Java (9.1), ebenfalls via REST API.

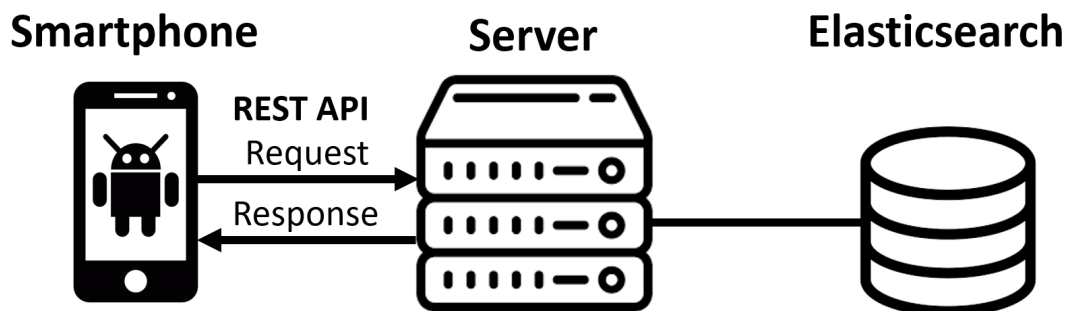


Abbildung 9.1: Kommunikation zwischen REST API und „WheelScout“-App (Smartphone)

9.1.2 Ausführung der Anfragen in der „WheelScout“-App

Zunächst eine kurze Wiederholung: Für die Ausführung von Anfragen werden Searchtemplates verwendet, damit Anfragen nicht jedes Mal neu aufgebaut werden müssen. Die ausgeführte Anfrage im Searchtemplate wird via („id“) gewählt. Nach der Auswahl des Searchtemplates müssen gegebenenfalls Suchparameter übergeben werden, um die Anfrage auszuführen.

In diesem Abschnitt soll es primär um das Verständnis von Suchparametern gehen, die bei der Übergabe von API-Anfragen erfolgen.

Die Suchparameter („toilet“) und („accessibleWith“) können folgende Begriffe annehmen:

(„toilet“):

- **true:** Toiletten sind barrierefrei.
- **false:** Toiletten sind im Gebäude vorhanden, aber nicht barrierefrei.
- **unknown:** Toiletten sind vorhanden, aber es ist nicht bekannt, ob Toiletten barrierefrei sind.
- **no:** In diesem Gebäude befindet sich keine Toilette.

(„accessibleWith“):

- **1:** POI ist vollständig barrierefrei (grün)
- **2:** POI ist teilweise barrierefrei (gelb)
- **3:** POI ist nicht barrierefrei (rot)
- **4:** Barrierestatus vom POI ist unbekannt (grau)

Anfragen ohne Autovervollständigung wurden hier nicht gelistet, da die Eingabe der Suchparameter identisch ist, nur bei der „id“ müsste der Begriff „Autocomplete“ entfernt werden, um die Anfragen ohne Autovervollständigung zu nutzen. Die Anfragen in Elasticsearch werden wie folgt ausgeführt:

Aufruf der Radiusanfrage

Bei Ausführung der Suchanfrage 9.1 sind folgende Suchparameter optional: („search“), („access“), („toilet“), („de_category“), („de_parentCategory“). Wird die Anfrage ohne diese Parameter ausgeführt, werden alle POIs innerhalb der Radiuslänge ausgegeben.

```
1 POST [HOST:PORT]/poi/_search/template
2 {
3   "id": "radiussearchAutocomplete",
4   "params": {
5     "search": "mcd",
6     "distance": "5000m",
7     "lon": 8.627746,
8     "lat": 49.872541,
9     "access": ["1", "2", "3", "4"],
10    "toilet": ["true", "unknown", "false", "no"],
11    "de_category": ["Fast Food"],
12    "de_parentCategory": ["Essen"]
13  }
14 }
```

Listing 9.1: Aufruf des Searchtemplates „radiussearchAutocomplete“

Aufruf der Volltextsuche

Bei der Volltextsuche von POIs erfolgt die Sucheingabe über das Feld „search“. Die Felder „access“ und „toilet“ sind bei der Ausführung der Anfrage verpflichtend anzugeben. Wenn keine Filterung erfolgen soll, müssen alle Keywords übergeben werden, wie z.B. muss bei „access“ der Array [„1“, „2“, „3“, „4“] übergeben werden, um Filterung von Barrierefreiheit auszuschließen. Die Volltextsuche funktioniert für folgende Felder: („name“), („de_category“), („de_parentCategory“), („street“), („postcode“), („city“), („state“) und („country“). Beispiel: „McDonalds in 64409“ oder „Kneipe in Darmstadt“ etc. Die Volltextsuche kann mit Synonymen und Pluralformen von Unterkategorien („de_category“) umgehen und durch eine Autocomplete-Funktion lassen sich auch bei einer unvollständigen Sucheingabe, wie sie in Listing 9.2 vorliegt, gute Suchergebnisse erzielen.

```
1 POST [HOST:PORT]/poi/_search/template
2 {
3   "id": "POIFulltextsearchAutocomplete",
4   "params": {
5     "search": "mcdon frankf",
6     "access": {"values": ["2", "3"]},
7     "toilet": ["unknown"]
8   }
9 }
```

```
8 } }
```

Listing 9.2: Aufruf des Searchtemplates „POIFulltextsearchAutocomplete“

Die Volltextsuche funktioniert auch für die Suche nach Adressen und wird wie folgt ausgeführt:

```
1 POST [HOST:PORT]/address/_search/template
2 {
3   "id": "AddressFulltextsearchAutocomplete",
4   "params": {
5     "search": "Riedes Darmst"
6   }
7 }
```

Listing 9.3: Aufruf des Searchtemplates „AddressFulltextsearchAutocomplete“

Aufruf von Aggregationsanfragen

Aufgrund dessen, dass POIs in insgesamt 177 Unterkategorien unterteilt sind, wird noch zusätzlich eine Aggregationsanfrage als Searchtemplate angeboten, um eine detaillierte Filterung von Unterkategorien (siehe Lis. 9.4) zu ermöglichen. Die Filterung kann bei der Radiussuche oder bei der Anfrage „POIs während des Routings anzeigen“ für das Feld „de_category“ übergeben werden, um die Ausgabe der POIs einzuschränken. Es sollen z.B. nur Bäckereien ausgegeben werden. Durch die Aggregationsanfrage sollen dem Nutzer, die Unterkategorien als Liste bereitgestellt werden, um eine Selektion der Unterkategorie „Bäckerei“ zu ermöglichen.

```
1 POST [HOST:PORT]/poi/_search/template
2 {
3   "id": "groupby_category"
4 }
```

Listing 9.4: Aufruf des Searchtemplates „groupby_category“

Spezielle Anfrage: „POIs während des Routings anzeigen“

Bei der Anfrage „POIs während des Routings anzeigen“ handelt es sich um eine spezielle Anfrage, die dem Anwender während einer geroutete Strecke, bestimmte POIs auf der Strecke anzeigen lässt. Die Anfrage „POIs während des Routings anzeigen“, die von einem Nutzer der „WheelScout“-App abgesetzt wird, wird zuerst an den „WheelScout“-Backend-Server gesendet. Im „WheelScout“-Backend-Server wird die kürzeste geroutete Strecke mit der Routing Engine GraphHopper berechnet, unter Berücksichtigung von vordefinierten Barrieren. Im Response von GraphHopper werden die Kanten der gerouteten Strecke, die sich in einem Array „coordinates“ befinden und Distanzen der Abstände, die sich ebenfalls in einem Array befinden, entnommen und mit weiteren Parametern, die sich im Lis. 9.5 befinden, an den POIServer via REST API übergeben und dort wird die Anfrage ebenfalls weiterverarbeitet,

bis der Elasticsearch-Java-Client mehrere Radiananfragen an den Elasticsearch-Server absetzt. Im Elasticsearch-Server werden die Radiananfragen verarbeitet und der Response von Elasticsearch-Server wird in JSON-Format auf den gleichen Weg an die „WheelScout“-App zurücksendet.

```

1 POST [HOST:PORT]/poi
2 {
3   "poisearch":{
4     "search":"mcdonalds",
5     "distance":"250m",
6     "toilet":["true", "false", "unknown"],
7     "de_parentCategory":["Essen"],
8     "access":["1", "2", "3", "4"],
9     "startlon":8.578,
10    "startlat":49.324,
11    "endlon":9.324,
12    "endlat":50.324,
13    "coordinates":[[8.628828, 49.872573, 126.86],
14                   [8.628832, 49.872542, 126.94],
15                   [8.628858, 49.872516, 126.98], ... ],
16    "coordinatesdistances":[
17      [0,1, 3.460376168091009], [1, 2, 3.513], [2, 3, 6.318], ... ]
18  }
19 }
```

Listing 9.5: Aufruf der Anfrage „POIs während des Routings anzeigen“

9.2 Frontend der „WeelScout“-App

Das Frontend bzw. der Client der „WeelScout“-App wurde mit dem Open Source Webframework Ionic entwickelt. Ionic ist im Kern ein ausgereiftes JavaScript Framework und stellt u.a. Komponenten für Layouts, Buttons, Statusleisten, Listen usw. für die mobile App-Entwicklung zur Verfügung. Ionic kann flexibel eingesetzt werden, wie beispielsweise in hybriden Apps oder Progressiven Web Apps (als Webseite). „Hybride Apps basieren auf den Web Technologien wie HTML5, CSS, JavaScript, laufen aber in einem Webview Container, sind als native Apps verpackt und können somit auf native APIs und Funktionen vom Betriebssystem zugreifen“ [70]. Eine besondere Herausforderung ist dabei stets die Gestaltung des User Interfaces. Die typischen Elemente und die meist auf Touch-Interaktion basierende Bedienung, unterscheiden sich erheblich von den Möglichkeiten einer klassischen Webanwendung, die etwa in einem Browser auf einem PC bzw. Notebook ausgeführt wird [71].

9.2.1 Entwicklung eines Frontend für die Anzeige von POIs in der „WeelScout“-App (Prototyp)

Für die Entwicklung einer Benutzeroberfläche werden zuerst Icons benötigt, die die Oberkategorien sowie die Barrierefreiheit der POIs repräsentieren sollen. Die Icons bzw. die

Kategorien wurden im PNG-Format (512 x 512) von der Webseite¹ heruntergeladen. Die Icons dürfen kostenlos verwendet werden, solange die Quellen bei Veröffentlichung der App im Google Store hinterlegt sind. Die Icons sollen als Marker dargestellt werden, um die POIs auf der Karte darzustellen. Mit dem Bildbearbeitungstool GIMP wurden die Icons in einem Marker integriert. Dabei muss der Hintergrund der Marker transparent sein, damit nicht Teile der Karte abgedeckt werden. In Abbildung 9.2 befinden sich die erstellten Marker. Durch unterschiedliche Farben der Marker wird die Barrierefreiheit abgebildet (grün: barrierefrei, gelb: teilweise barrierefrei, rot: nicht barrierefrei und grau: unbekannt). Es gibt insgesamt 12 unterschiedliche Kategorien zur Auswahl. Points of Interest bzw. Gebäude, die keinen Kategorien zugeordnet sind, werden mit „Unbekannt“ deklariert und können später vom Nutzer der WheelScout ergänzt werden.

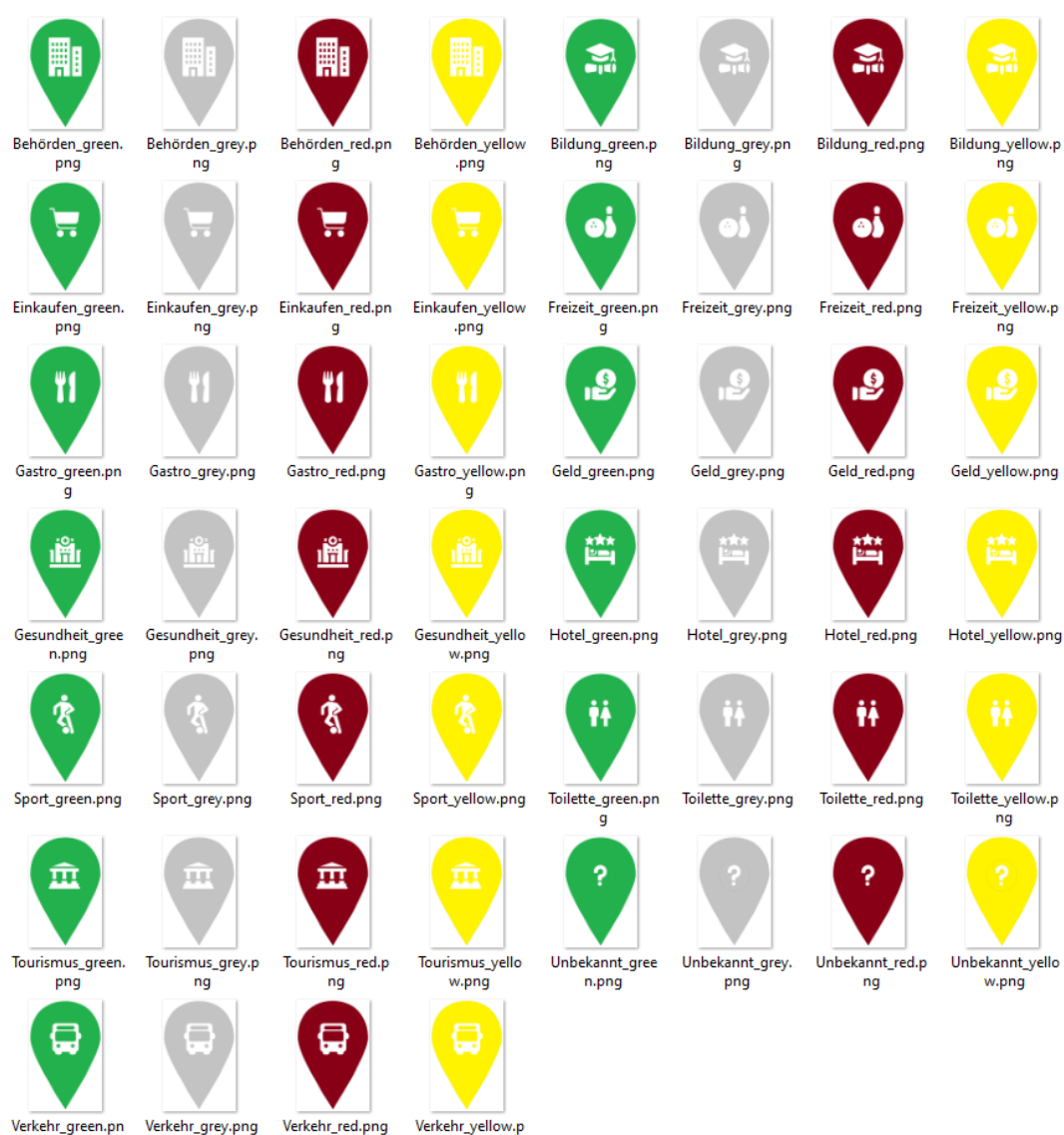


Abbildung 9.2: Erstellung von Icons für die „WheelScout“-App

¹<https://www.flaticon.com/de>

9.2.2 Ergebnis des Frontend für die Anzeige von POIs in der „WheelScout“-App (Prototyp)

Das Anzeigen der POIs erfolgt über eine Radiusanfrage via REST API an Elasticsearch. Es werden aktuell maximal 500 POIs angezeigt (maximal kann Elasticsearch 10.000 POIs ausgeben). In der Abbildung 9.2 ist das Frontend der „WheelScout“-App abgebildet. Bevor die POIs angezeigt werden, muss der Nutzer auf Einstellungen gehen und auf „POIs anzeigen“ klicken. Daraufhin werden alle POIs als Marker ausgegeben. Kategorien, die der Nutzer ausblenden möchte, kann der Nutzer im obigen Balken durch Anklicken der Symbole, die Marker, die sich auf der Karte befinden, ausblenden. Es handelt sich hierbei noch um einen Prototypen; Toiletten können noch nicht angezeigt werden, da („restrooms“) als eigenes Feld deklariert wurde. Weiterhin kann das Frontend noch nicht die vollständige Komplexität der Suchanfragen von Elasticsearch abbilden. Im nächsten Kapitel erfolgt eine Zusammenfassung der Masterarbeit und weitere Schritte werden angeführt, wie das „WheelScout“-Projekt weiterentwickelt werden kann.

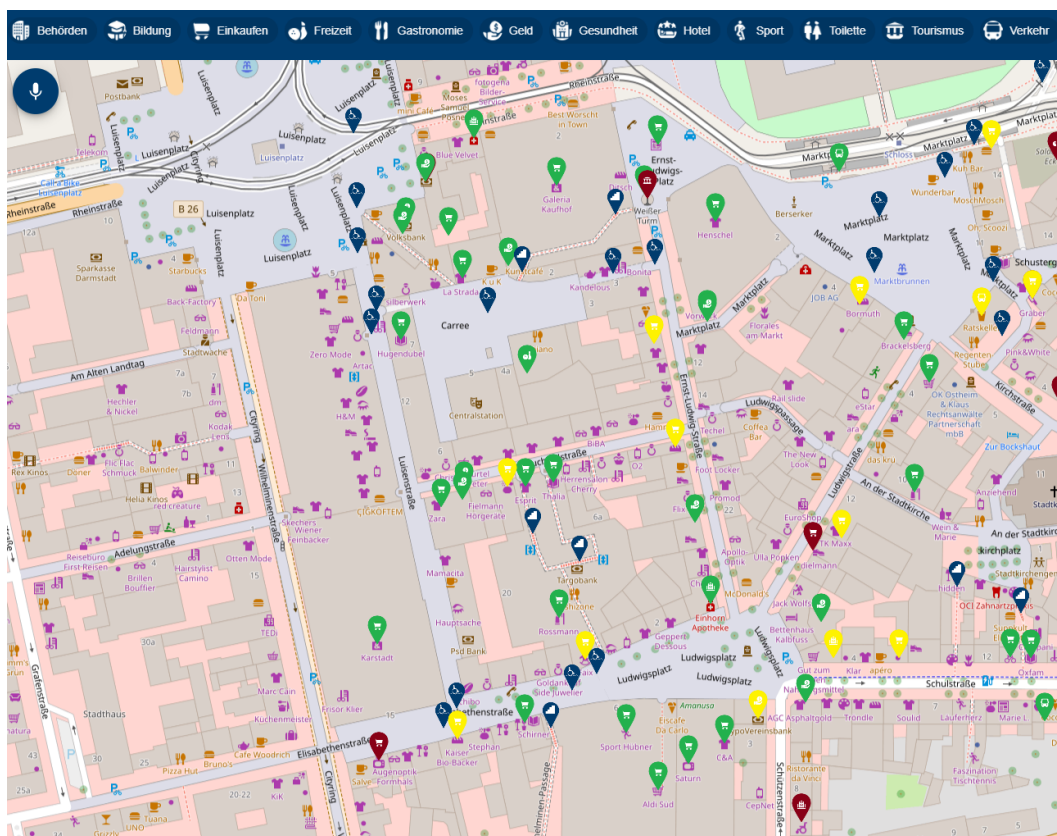


Abbildung 9.3: Frontend „WheelScout“-App

10 Fazit und Ausblick

10.1 Fazit

Die Daten von accessibility.cloud konnten erfolgreich durch eine Gridsearch-Anfrage entnommen werden und alle Straßendaten in Deutschland konnten mit der Photon API durch eine Bereichsanfrage beschafft werden. Durch das Geocoding Tool Nominatim konnten geographische Koordinaten von POIs mit Adressdaten ergänzt werden. Points of Interest konnte mit Oberkategorien ergänzt werden und von der englischen in die deutsche Sprache übersetzt werden. Lückenhafte Adressdaten der Points of Interest konnten durch Straßendaten von Photon in eine PostgreSQL-Datenbank mit PostGIS-Funktionen teilweise vervollständigt werden. Weiterhin konnten Daten in PostgreSQL bereinigt und bearbeitet werden, so dass ein Import der Daten in Elasticsearch in den weiteren Bearbeitungsschritten möglich ist. Es konnte eine effiziente Datenstruktur für die Daten erstellt werden, um die Daten in Elasticsearch zu importieren. Die Normalisierung der Bezeichnung von den Points of Interest für die Oberkategorien „Einkaufen“ und „Essen“ konnte mit dem entwickelten Verfahren erfolgreich angewendet werden. Die Implementierung von komplexen Suchanfragen, wie Radiusuche, Volltextsuche und „POIs während des Routings anzeigen“, konnte mit viel Aufwand und Tests in Kibana entwickelt werden. Die Dokumentenstruktur, Analyzer und Suchanfragen in Elasticsearch konnten aufeinander abgestimmt werden. Dadurch sind optimale Suchergebnisse für die Points of Interest und Straßendaten möglich. Die Suchmaschine Elasticsearch konnte im Docker auf dem Linux-Server erfolgreich installiert werden. Die POIs sowie Straßendaten konnten auf den Server hochgeladen und in die Suchmaschine Elasticsearch importiert werden. Die Kommunikation bzw. die Ausführung von Anfragen in dem „WheelScout“-Client („WheelScout“-App) und dem Elasticsearch-Server erfolgt via REST API. Die Kommunikation zwischen „WheelScout“-Backend-Server und dem Elasticsearch-Server konnte via REST API erfolgreich hergestellt werden, um die Anfrage „POIs während des Routings anzeigen“ in das „WheelScout“-Projekt zu integrieren. Ein Prototyp für das Frontend konnte für die „WheelScout“-App erstellt werden. Es konnte eine horizontale skalierbare Suchmaschine in das „WheelScout“-Projekt integriert werden, um dem Anwender eine performante Suchanwendung von POIs zu ermöglichen. Es können nun Radius- und Volltextsuche und die spezielle Anfrage „POIs während des Routings anzeigen“ für Points of Interest via REST API im „WheelScout“-Projekt genutzt werden.

10.2 Ausblick

Die Funktionalität der „Wheelscout“-App ist bereits zum jetzigen Zeitpunkt gegeben, dennoch können noch weitere Verbesserungen durchgeführt werden:

10.2.1 „WheelScout“-Backend

- Die POI-Daten für Deutschland wurden von [accessibility.cloud](#) bezogen. Jedoch wäre es zu umständlich, dieses Verfahren auf alle restlichen Ländern anzuwenden. Deshalb sollte das Verfahren nur für eine Auswahl an Ländern durchgeführt werden. Bei den restlichen Ländern sollten mit den Daten vom Dump, die als CSV-Datei vorliegen und von Sozialhelden e.V. bereitgestellt wurden in Elasticsearch ergänzt werden.
- Für jeden neu hinzugefügten Point of Interest muss wieder reverse Geocoding verwendet werden, um Points of Interest mit Adressdaten zu ergänzen. Für das Suchen von Straßen verwendet die „WheelScout“-App zurzeit die Nominatim-API. Jedoch ist die API beschränkt (aufgrund der Nominatim-Richtlinie) und es kann nur eine Anfrage pro Sekunde ausgeführt werden. Aus diesen Gründen wäre es sinnvoll, Nominatim auf einen lokalen Server zu installieren. Die Installation von Nominatim auf einen Server ist sehr aufwendig, nimmt viel Zeit in Anspruch und ist zusätzlich aufgrund hoher Hardwareanforderungen mit hohen Kosten verbunden. Jedoch hätte die „WheelScout“-App eine performante Suchmaschine für Straßendaten, würde das Geocodieren für POIs nicht mehr viel Zeit in Anspruch nehmen. Eine andere Variante wäre die Installation von dem Geotool Photon, das leichter lokal zu installieren ist und dessen Hardwareanforderung im Gegensatz zu Nominatim viel geringer ausfallen. Dies ist mit dem Nachteil verbunden, dass POIs und Straßen als Point-Koordinate abgebildet sind und Hausnummern für Straßen kaum vorliegen. Dafür bietet Photon eine Autovervollständigung für Straßen an.
- Mit dem Geotool Overpass API lassen sich weitere nützliche Information von den POIs beschaffen. Overpass API ist ein Tool, das es ermöglicht, durch Anfragen gezielte Daten von OpenStreetMap-Daten zu extrahieren. Die API ist sehr performant und es ist möglich in einigen Minuten mehrere Million Daten zu extrahieren [72]. Die Response erfolgt als JSON-Format. OpenStreetMap bietet zudem Information zu Öffnungszeiten von Läden an. Die Extraktion sowie Einbindung der Daten in Elasticsearch ist extrem aufwendig, da die Strukturen angepasst werden müssen und die Zuordnung der Öffnungszeiten auf die POIs als sehr schwierig betrachtet wird.
- [accessibility.cloud](#) wird demnächst Daten zu Aufzügen für POIs anbieten [53], diese Daten sollten dann auch in Elasticsearch integriert werden, um dies in der „WheelScout“-App anzeigen zu können.
- Die Volltextsuche in Elasticsearch funktioniert zurzeit nur für die deutsche Sprache. Weitere Sprachen, wie Englisch und Italienisch, sollen noch für die POIs in Elasticsearch integriert werden, dafür müssen neue Felder im Index „POI“ angelegt werden und eine Übersetzung der Kategorien, Städte etc. muss erfolgen. Gegebenenfalls müssen Analyzer an die Sprachen angepasst werden.

- Das Sicherheitskonzept, das zurzeit im „WheelScout“-Projekt für die REST API genutzt wird, muss noch für die REST API von Elasticsearch integriert werden.
- Eine performante horizontale Skalierung von verteilten Elasticsearch-Datenbanken ist ein sehr komplexes Thema, da viel Erfahrung benötigt und ein tiefes technisches Verständnis beim Routing bzw. der Verteilung der Daten von Elasticsearch vorausgesetzt wird. Alternativ dazu bietet, die Firma Elastic NV für Elasticsearch Cloud-Dienste an [73]. Die Dienste sind kostenpflichtig, aber es werden viele Services angeboten, um die Elasticsearch-Datenbank zu pflegen und zu überwachen. Eine horizontale Skalierung wird erst benötigt, wenn Massenanfragen in mehrfacher Millionenhöhe von den „WheelScout“-Nutzern vorliegen und die Anfragebearbeitung nicht mehr performant (bzw. in Millisekunden) bearbeitet werden kann.

10.2.2 „WheelScout“-Client (Frontend)

- Das Userinterface der „WheelScout“-App muss weiterentwickelt werden, um die Komplexität der Anfragen in Elasticsearch auch vollständig abzubilden. Dabei muss die Bedienung möglichst einfach und verständlich gestaltet werden. Die Volltextsuche sowie die Anfrage „POIs während des Routings anzeigen“ müssen im Frontend noch realisiert werden. Für die Realisierung des Frontend werden keine Fachkenntnisse in Elasticsearch benötigt.
- Integration des Elasticsearch-JavaScript-Clients im „WheelScout“-Client, um eine standardisierte Anfrage an den Elasticsearch-Server zu ermöglichen.
- Die Integration einer App-Steuerung durch Spracheingabe und Tonausgabe in die „WheelScout“-App wäre eine gute Funktion, um körperlich stark eingeschränkten Menschen, die Bedienung der „WheelScout“-App zu erleichtern.

Inhalt der CD

\	
_ 00_Masterarbeit	
_ Masterarbeit.pdf	Masterarbeit in PDF-Format
_ Code	
_ 01_ETL_Skripts	
_ main/	
_ A_GetData	
_ data.....	Speicherung der Daten in Dateien (CSV, JSON, TXT)
_ core.py.....	Ausführung der Funktionen
_ helpers.py.....	Implementierung von Funktionen zu Beschaffung und Zusammenführung der Daten
_ ...	
_ B_ImportAndCleanDataInPostgreDB	
_ data.....	Speicherung der Daten in Dateien (CSV, JSON, TXT)
_ sql.....	SQL-Dateien
_ core.py.....	Ausführung der Funktionen
_ helpers.py.....	Implementierung von Funktionen zum Import der Daten in die PostgreSQL-Datenbank und Ausführung von SQL-Dateien
_ ...	
_ C_ImportAndCleanDataInElasticsearchDB	
_ data.....	Speicherung der Daten in Dateien (CSV, JSON, TXT)
_ core.py.....	Ausführung der Funktionen
_ helpers.py.....	Implementierung von Funktionen zum Import der Daten in Elasticsearch sowie Standardisierung der POIs Bezeichnungen
_ ...	
_ D_ImplementSearchTempaltesForElasticsearchDB	
_ core.py.....	Ausführung der Funktionen
_ helpers.py.....	Implementierung und Speicherung der Searchtemplates in Elasticsearch
_ ...	
_ 02_Elasticsearch_Server	
_ elasticsearch/config/	
_ de_categories_synonym.txt	Synonyme für Unterkategorien
_ stop.txt	Stoppwörter für POIs Bezeichnungen
_ elasticsearch.yml	Konfiguration von Elasticsearch
_ 03_Elasticsearch_Client_and_POIServer	
_ src/test/java/	
_ Request.java.....	Aufbau von Searchtemplates durch Übergabe von Suchparametern
_ EsClient.java...	Aufbau einer Schnittstelle zur Kommunikation mit dem Elasticsearch-Server

• `PoiServer.java` Anfragen von WheelScout-Server (bzw. Client)
entgegennehmen und beantworten

• ...

Literatur

- [1] Statistisches Bundesamt. *Zahl der Woche Nr. 50 vom 10. Dezember 2019*. 2019. URL: https://www.destatis.de/DE/Presse/Pressemitteilungen/Zahl-der-Woche/2019/PD19_50_p002.html (zuletzt aufgerufen am: 30. März. 2020).
- [2] Statistisches Bundesamt. *Schwerbehinderte Menschen am Jahresende*. 2019. URL: <https://www.destatis.de/DE/Themen/Gesellschaft-Umwelt/Gesundheit/Behinderte-Menschen/Tabellen/geschlecht-behinderung.html> (zuletzt aufgerufen am: 30. April. 2020).
- [3] Ulrich Klaus Becker. *Barrierefreie Verkehrsinfrastruktur*. 2018. URL: <https://www.adac.de/-/media/pdf/vek/fachinformationen/infrastruktur/barrierefreie-verkehrsinfrastruktur-adac-bro.pdf> (zuletzt aufgerufen am: 30. April. 2020).
- [4] AktionMensch. *Barrierefreiheit – was heißt das?* URL: https://www.aktion-mensch.de/dafuer-stehen-wir/was-ist-inklusion/barrierefreiheit-bedeutung.html?gclid=CjOKCQjw7ZL6BRCmARIsAH6XFDJYhp0Bon0dU0ThlZUxy4g0H0ExCfLAUGXugyojvfaPmB3sjrRLTagaAnkAEALw_wcB (zuletzt aufgerufen am: 30. April. 2020).
- [5] EML European Media Laboratory GmbH. 12. Mai 2016. URL: https://www.eml.org/deutsch/medien/presse.php?we_objectID=198 (zuletzt aufgerufen am: 16. Oktober. 2020).
- [6] openstreetmap. *Was ist OpenStreetMap?* 2020. URL: <https://www.openstreetmap.de/faq.html> (zuletzt aufgerufen am: 30. März. 2020).
- [7] SOZIALHELDEN e.V. *Wir - SOZIALHELDEN e.V.* 25. Juni 2018. URL: <https://sozialhelden.de/wir/> (zuletzt aufgerufen am: 16. Oktober. 2020).
- [8] Tobias Trelle. *MongoDB: Der praktische Einstieg*. Heidelberg: dpunkt.verlag, 2014. ISBN: 978-3-86490-153-9. URL: <http://gbv.ebib.com/patron/FullRecord.aspx?p=1734323>.
- [9] GeoJSON. 7. Mai 2019. URL: <https://geojson.org/> (zuletzt aufgerufen am: 08. Oktober. 2020).
- [10] Google Maps marks 15-year milestone with new features. 7. Februar 2020. URL: <https://www.japantimes.co.jp/news/2020/02/07/business/tech/google-maps-marks-15-year-milestone-new-features/> (zuletzt aufgerufen am: 14. Oktober. 2020).
- [11] Google Cloud. *Places & Google Maps Platform & Google Cloud*. 10. Juli 2020. URL: <https://cloud.google.com/maps-platform/places?hl=de> (zuletzt aufgerufen am: 14. Oktober. 2020).
- [12] *Basic Installation - Nominatim 3.5.2*. 11. Oktober 2020. URL: <https://nominatim.org/release-docs/latest/admin/Installation/> (zuletzt aufgerufen am: 14. Oktober. 2020).
- [13] *Der OpenStreetMap-Geocoder Nominatim*. URL: <https://av.tib.eu/media/15845> (zuletzt aufgerufen am: 14. Oktober. 2020).

- [14] *Nominatim Usage Policy*. URL: <https://operations.osmfoundation.org/policies/nominatim/> (zuletzt aufgerufen am: 13. Oktober. 2020).
- [15] *Bing Maps - 165 Terabyte neue HighRes-Satellitenbilder*. 27. Juni 2012. URL: <https://blogs.bing.com/germany/2012/06/27/bing-maps-165-terabyte-neue-highres-satellitenbilder> (zuletzt aufgerufen am: 16. Oktober. 2020).
- [16] GitHub. *komoot/photon*. URL: <https://github.com/komoot/photon> (zuletzt aufgerufen am: 19. Oktober. 2020).
- [17] Christian Kunz. *Google liefert offizielle Zahlen zum täglichen Suchevolumen - SEO Südwest*. 2019. URL: <https://www.seo-suedwest.de/5431-google-liefert-offizielle-zahlen-zum-taeglichen-suchevolumen.html> (zuletzt aufgerufen am: 25. September. 2020).
- [18] Jonas Freiknecht und Stefan Papp. *Big Data in der Praxis: Lösungen mit Hadoop, Spark, HBase und Hive. Daten speichern, aufbereiten, visualisieren*. 2., erweiterte Auflage. München: Hanser, 2018. ISBN: 978-3-446-45396-8. DOI: 10.3139/9783446456013.
- [19] Florian Hopf. *Elasticsearch: Ein praktischer Einstieg*. 1. Auflage. Heidelberg: dpunkt.verlag, 2016. ISBN: 978-3-86490-289-5.
- [20] Jan Steemann. *Datenmodellierung in nicht relationalen Datenbanken*. 27. Dezember 2013. URL: <https://entwickler.de/online/datenbanken/datenmodellierung-in-nicht-relationalen-datenbanken-137872.html> (zuletzt aufgerufen am: 16. Oktober. 2020).
- [21] Amazon Web Services, Inc. *Was ist NoSQL? – Amazon Web Services (AWS)*. URL: <https://aws.amazon.com/de/nosql/> (zuletzt aufgerufen am: 16. Oktober. 2020).
- [22] Volker Bentz. *Was ist Hashing?* 17. Februar 2019. URL: <https://www.brandmauer.de/blog/it-security/was-ist-hashing> (zuletzt aufgerufen am: 19. Oktober. 2020).
- [23] Nico Litzel. *Was ist ACID?* 15. November 2018. URL: <https://www.bigdata-insider.de/was-ist-acid-a-776182/> (zuletzt aufgerufen am: 19. Oktober. 2020).
- [24] *What does soft state mean?* URL: <https://www.definitions.net/definition/soft+state> (zuletzt aufgerufen am: 19. Oktober. 2020).
- [25] Jaroslaw Kijanowski. *Eric Brewer on The CAP Theorem — Episode 227 on se-radio.net — tl;db series*. 5. Februar 2020. URL: <https://blog.softwaremill.com/eric-brewer-on-the-cap-theorem-tl-dr-series-83f058945e> (zuletzt aufgerufen am: 19. Oktober. 2020).
- [26] Krishna Gade. *Twitter Search is Now 3x Faster*. 2011. URL: https://blog.twitter.com/engineering/en_us/a/2011/twitter-search-is-now-3x-faster.html (zuletzt aufgerufen am: 30. März. 2020).
- [27] stackshare. *Who uses Elasticsearch?* 2020. URL: <https://stackshare.io/elasticsearch> (zuletzt aufgerufen am: 30. März. 2020).
- [28] Stefan Waldhauser. *Warum Open Source?* 2020. URL: <https://www.elastic.co/de/about/why-open-source> (zuletzt aufgerufen am: 30. März. 2020).

- [29] solid IT GmbH. *DB-Engines Ranking of Search Engines*. 2020. URL: <https://db-engines.com/en/ranking/search+engine> (zuletzt aufgerufen am: 30. März. 2020).
- [30] solid IT GmbH. *Berechnungsmethode der Wertungen im DB-Engines Ranking*. 2020. URL: https://db-engines.com/de/ranking_definition (zuletzt aufgerufen am: 30. März. 2020).
- [31] *Mapping concepts across SQL and Elasticsearch | Elasticsearch Reference [7.9] | Elastic*. 18. August 2020. URL: https://www.elastic.co/guide/en/elasticsearch/reference/current/_mapping_concepts_across_sql_and_elasticsearch.html#_mapping_concepts_across_sql_and_elasticsearch (zuletzt aufgerufen am: 24. Juni. 2020).
- [32] *Language analyzers | Elasticsearch Reference [7.9] | Elastic*. URL: <https://www.elastic.co/guide/en/elasticsearch/reference/current/analysis-lang-analyzer.html> (zuletzt aufgerufen am: 04. Oktober. 2020).
- [33] *Tokenizer reference | Elasticsearch Reference [7.9] | Elastic*. 18. August 2018. URL: https://www.elastic.co/guide/en/elasticsearch/reference/current/analysis-tokenizers.html#_structured_text_tokenizers (zuletzt aufgerufen am: 04. Oktober. 2020).
- [34] *Standard analyzer | Elasticsearch Reference [7.9] | Elastic*. URL: <https://www.elastic.co/guide/en/elasticsearch/reference/7.9/analysis-standard-analyzer.html> (zuletzt aufgerufen am: 19. Oktober. 2020).
- [35] *search_analyzer | Elasticsearch Reference [7.9] | Elastic*. 19. August 2020. URL: <https://www.elastic.co/guide/en/elasticsearch/reference/current/search-analyzer.html> (zuletzt aufgerufen am: 04. Oktober. 2020).
- [36] *Field data types | Elasticsearch Reference [7.9] | Elastic*. 4. September 2020. URL: <https://www.elastic.co/guide/en/elasticsearch/reference/current/mapping-types.html> (zuletzt aufgerufen am: 19. Oktober. 2020).
- [37] *Freitextsuche (oder Volltextsuche)*. URL: <https://www.excentos.com/de/glossar/freitextsuche> (zuletzt aufgerufen am: 19. Oktober. 2020).
- [38] *Query and filter context | Elasticsearch Reference [7.9] | Elastic*. 15. Oktober 2020. URL: <https://www.elastic.co/guide/en/elasticsearch/reference/current/query-filter-context.html#relevance-scores> (zuletzt aufgerufen am: 19. Oktober. 2020).
- [39] *Ähnlichkeitsmaße*. URL: <https://www.witi.cs.uni-magdeburg.de/mmdb/probe.pdf> (zuletzt aufgerufen am: 21. Oktober. 2020).
- [40] *Elasticsearch Clients | Elastic*. 21. September 2020. URL: <https://www.elastic.co/guide/en/elasticsearch/client/index.html> (zuletzt aufgerufen am: 19. Oktober. 2020).
- [41] IT-Talents. *Was ist REST (Representational State Transfer)?* 22. Dezember 2018. URL: <https://www.it-talents.de/blog/it-talents/was-ist-rest> (zuletzt aufgerufen am: 19. Oktober. 2020).

- [42] *PYPL PopularitY of Programming Language index*. 1. Oktober 2020. URL: <http://pypl.github.io/PYPL.html> (zuletzt aufgerufen am: 19. Oktober. 2020).
- [43] *History and License — Python 3.9.0 documentation*. URL: <https://docs.python.org/3/license.html> (zuletzt aufgerufen am: 19. Oktober. 2020).
- [44] Marcus Hofmann. *Python 3: Programmieren für Einsteiger - Vom Anfänger zum Python-Profi innerhalb 30 Tage inkl. exklusivem Begleitmaterial (Deutsch) Taschenbuch*. Heidelberg, 2019. ISBN: 9798634847030.
- [45] Sayantini. *Java vs Python : Comparison between the Best Programming Languages*. 17. November 2018. URL: <https://www.edureka.co/blog/java-vs-python/> (zuletzt aufgerufen am: 19. Oktober. 2020).
- [46] Elastic. *Was ist Kibana? | Elastic*. URL: <https://www.elastic.co/de/what-is/kibana> (zuletzt aufgerufen am: 19. Oktober. 2020).
- [47] Bernd Öggl und Michael Kofler. *Docker: Das Praxisbuch für Entwickler und DevOps-Teams. Für Windows, macOS und Linux*. 2. Auflage. 2020. ISBN: 9783836272261.
- [48] Stephan Augsten. *Was sind Docker-Container?* 25. April 2017. URL: <https://www.dev-insider.de/was-sind-docker-container-a-597762/> (zuletzt aufgerufen am: 30. Oktober. 2020).
- [49] PostGIS Developers. *PostGIS — Spatial and Geographic Objects for PostgreSQL*. URL: <https://postgis.net/> (zuletzt aufgerufen am: 19. Oktober. 2020).
- [50] GraphHopper Routing Engine. *GraphHopper Routing Engine*. 2020. URL: <https://github.com/graphhopper/graphhopper> (zuletzt aufgerufen am: 30. März. 2020).
- [51] GraphHopper Directions API. *Products - GraphHopper Directions API*. 14. September 2020. URL: <https://www.graphhopper.com/products/> (zuletzt aufgerufen am: 19. Oktober. 2020).
- [52] *sozialhelden/accessibility-cloud*. 8. Oktober 2020. URL: <https://github.com/sozialhelden/accessibility-cloud/blob/master/app/docs/json-api.md> (zuletzt aufgerufen am: 08. Oktober. 2020).
- [53] accessibility.cloud. *Dashboard*. URL: <https://www.accessibility.cloud/> (zuletzt aufgerufen am: 26. Oktober. 2020).
- [54] *GEOFABRIK // Downloads*. 16. Oktober 2020. URL: <https://www.geofabrik.de/de/data/download.html> (zuletzt aufgerufen am: 25. Oktober. 2020).
- [55] dev admin. *Frequently Asked Questions - Bing Maps*. 12. Februar 2020. URL: <https://www.microsoft.com/en-us/maps/faq> (zuletzt aufgerufen am: 26. Oktober. 2020).
- [56] GitHub. *osm-search/Nominatim*. URL: <https://github.com/osm-search/Nominatim> (zuletzt aufgerufen am: 26. Oktober. 2020).
- [57] Google Cloud. *Google Maps Platform Terms of Service & Google Cloud*. URL: https://cloud.google.com/maps-platform/terms/?_ga=2.107834571.834556403.1603610585-143490468.1596758206&_gac=1.187239258.1603224963.CjwKCAjwlbr8BRA0EiwAnt4MTvA9eLxjORNUdE5ow-PUvZs1ZY0dyLG0Jssyo5LEfJnsHsWz0X3n5RoCG0QQAvD_BwE#3.-license. (zuletzt aufgerufen am: 25. Oktober. 2020).

- [58] dev admin. *Bing Maps Licensing - Bing Maps*. 12. Februar 2020. URL: <https://www.microsoft.com/en-us/maps/licensing> (zuletzt aufgerufen am: 26. Oktober. 2020).
- [59] Nico Litzel. *Was ist der k-Means-Algorithmus?* 18. Juli 2018. URL: <https://www.bigdata-insider.de/was-ist-der-k-means-algorithmus-a-734637/> (zuletzt aufgerufen am: 20. Oktober. 2020).
- [60] *Elasticsearch Versioning Support*. 24. Juni 2013. URL: <https://www.elastic.co/de/blog/elasticsearch-versioning-support> (zuletzt aufgerufen am: 31. Oktober. 2020).
- [61] *Slippy map tilenames - OpenStreetMap Wiki*. 6. Oktober 2020. URL: https://wiki.openstreetmap.org/wiki/Slippy_map_tilenames#ECMAScript_.28JavaScript.2FActionScript.2C_etc..29 (zuletzt aufgerufen am: 08. Oktober. 2020).
- [62] *Nominatim Usage Policy*. 7. Oktober 2020. URL: <https://operations.osmfoundation.org/policies/nominatim/> (zuletzt aufgerufen am: 13. Oktober. 2020).
- [63] *null_value | Elasticsearch Reference [7.9] | Elastic*. URL: <https://www.elastic.co/guide/en/elasticsearch/reference/current/null-value.html> (zuletzt aufgerufen am: 20. Oktober. 2020).
- [64] *Multi-match query | Elasticsearch Reference [7.9] | Elastic*. URL: <https://www.elastic.co/guide/en/elasticsearch/reference/current/query-dsl-multi-match-query.html#type-most-fields> (zuletzt aufgerufen am: 23. Oktober. 2020).
- [65] *Multi-match query | Elasticsearch Reference [7.9] | Elastic*. URL: <https://www.elastic.co/guide/en/elasticsearch/reference/current/query-dsl-multi-match-query.html#type-cross-fields> (zuletzt aufgerufen am: 23. Oktober. 2020).
- [66] *Heap: Sizing and Swapping | Elasticsearch: The Definitive Guide [2.x] | Elastic*. URL: <https://www.elastic.co/guide/en/elasticsearch/guide/current/heap-sizing.html> (zuletzt aufgerufen am: 23. Oktober. 2020).
- [67] *Hardware | Elasticsearch: The Definitive Guide [2.x] | Elastic*. URL: <https://www.elastic.co/guide/en/elasticsearch/guide/current/hardware.html> (zuletzt aufgerufen am: 23. Oktober. 2020).
- [68] WinSCP - Free SFTP and FTP client. *WinSCP*. URL: <https://winscp.net/eng/index.php> (zuletzt aufgerufen am: 23. Oktober. 2020).
- [69] *Download PuTTY - a free SSH and telnet client for Windows*. 18. April 2018. URL: <https://www.putty.org/> (zuletzt aufgerufen am: 23. Oktober. 2020).
- [70] Davit. *Native Apps vs. Web Apps vs. Hybride Apps*. 3. September 2018. URL: <https://app3null.com/native-hybride-web-apps/> (zuletzt aufgerufen am: 25. Oktober. 2020).
- [71] Veikko Krypczyk. *Web-App Tutorial: Ionic - ein Framework für das User Interface mobiler Apps*. 29. Juli 2019. URL: <https://entwickler.de/online/web/web-app-tutorial-ionic-579898855.html> (zuletzt aufgerufen am: 25. Oktober. 2020).
- [72] Olbricht und Roland. *OpenStreetMap in GIScience*. 2015. URL: https://wiki.openstreetmap.org/wiki/Overpass_API (zuletzt aufgerufen am: 26. Oktober. 2020).

- [73] Elastic. *Elastic Cloud: Gehostetes Elasticsearch, Gehostete Suche / Elastic*. URL: <https://www.elastic.co/de/cloud/> (zuletzt aufgerufen am: 26. Oktober. 2020).