Ontologiebasierter Transfer semistrukturierter Daten zu Linked Data

Bachelor-Thesis

zur Erlangung des akademischen Grades B.Sc.

Tim Fock

2270398

Hochschule für Angewandte Wissenschaften Hamburg
Fakultät Design, Medien und Information
Department Medientechnik

Erstprüfer: Prof. Dr.-Ing. Sabine Schumann

Zweitprüfer: Prof. Dr. Larissa Putzar

Hamburg, 23. 12. 2020

Inhaltsverzeichnis

1 Einleitung **5**

2 Grundlagen **6**
2.1 Semantic Web 6
2.2 Linked (Open) Data 8
2.3 Resource Description Framework (RDF) 10
2.4 Ontologien 15
2.4.1 RDF-Schema (RDFS) 18
2.4.2 Web Ontology Language (OWL) 20
2.5 SPARQL 23
2.6 Shapes Constraint Language (SHACL) 25

3 Konzept **28**
3.1 Beschreibung des Datensatzes 28
3.2 Idee 29
3.3 Anforderungen 30

4 Implementation **32**
4.1 Verwendete Technologien 32
4.1.1 Protégé 32
4.1.2 OnToology 32
4.1.3 RML & YARRRML 33
4.1.4 Ontotext GraphDB 33
4.2 Entwicklung der Ontologie 34
4.3 Erstellen der Mapping-Regeln 38
4.4 Mapping-Prozess & Abspeichern in der Datenbank 40

5 Validierung **42**

6 Fazit **45**

A Material **46**
A.1 JSON-Dokument einer Immobilie 46
A.2 YARRRML-Regeln 48
A.3 Auszug aus dem SHACL-Shape 52

Abbildungsverzeichnis **54**

Tabellenverzeichnis **55**

Literaturverzeichnis **56**

Abstract

The aim of this work is to develop a process for transferring semistructured data to Linked Data. Linked Data is a form of structured data that, by being interlinked with other data, enables machines to understand relationships and correctly interpret the data in its context. The approach uses a self-developed ontology as a data model and shows a way to bring this model to use via a mapping solution. Finally, a way to validate Linked Data with respect to self-selected expectations for the data is presented.

Zusammenfassung

Ziel dieser Arbeit ist die Entwicklung eines Prozesses für den Transfer semistrukturierter Daten zu Linked Data. Linked Data sind eine Form von strukturierten Daten, die durch Vernetzung mit anderen Daten Maschinen in die Lage versetzen Zusammenhänge zu verstehen und die Daten in ihrem Kontext richtig zu interpretieren. Der Ansatz nutzt eine selbst entwickelte Ontologie als Datenmodell und zeigt eine Möglichkeit, dieses Modell über eine Mapping-Lösung zur Anwendung zu bringen. Abschließend wird ein Weg präsentiert, Linked Data hinsichtlich selbst gewählter Erwartungen an die Daten zu validieren.

1 Einleitung

Daten bilden den Kern der digitalen Gesellschaft und sind die treibende Kraft des Internets. Tim Berners-Lee, der Erfinder des *World Wide Web*, hat die Vision eines *Semantic Web*, bei dem das Internet als eine globale Datenbank funktioniert. Alle Daten sollen miteinander verbunden und über einzigartige Adressen erreichbar sein. Linked Data greift zu diesem Zweck auf standardisierte Web-Technologien wie HTTP, RDF und URIs zurück. Intelligente Systeme werden so in die Lage versetzt, Daten aus vielen verschiedenen Quellen miteinander zu kombinieren und ermöglichen völlig neue Anwendungsfälle. Ein prominentes Beispiel dafür stellen Suchmaschinen dar.

Diese Arbeit zeigt anhand eines Datensatzes von Immobilien, wie sich bisher unverknüpfte Daten zu Linked Data transformieren lassen. Die Daten gehören zu einem PropTech-Unternehmen, das die Immobilien unter anderem über die eigene Plattform inseriert. Um die Suchmaschinen-Ergebnisse weiter zu verbessern, sollen auf den einzelnen Seiten der Inserate Linked Data eingebettet werden, die die gezeigten Immobilien maschinenlesbar repräsentieren.

Im zweiten Kapitel werden die Grundlagen beschrieben, die für diese Arbeit nötig sind. Es beginnt mit einem Überblick über das Semantic Web und dessen Prinzipien, sowie den Grundregeln für Linked Data. Als Nächstes wird das Sprachwerkzeug RDF (*Resource Description Framework*) präsentiert, welches genutzt wird, um Linked Data auszudrücken. Es folgt ein Abschnitt über Ontologien und deren Sprachen. Ontologien werden in der Informatik als eine Art der Wissensrepräsentation verwendet und stellen in RDF formulierte Meta-Modelle dar. Den Abschluss dieses Kapitels bilden die Abfragesprache SPARQL und die Validierungssprache SHACL.

Das dritte Kapitel geht näher auf den Aufbau des Ausgangsdatensatzes ein. Außerdem wird hier der gewählte Ansatz zum Transfer zu Linked Data genauer erklärt und Anforderungen formuliert, die der Ansatz erfüllen soll.

Das vierte Kapitel behandelt die konkreten Software-Werkzeuge, die zur Umsetzung verwendet wurden. Hier findet sich auch die Erklärung zum Entwicklungsprozess der Ontologie und dem anschließenden Mapping. Schließlich folgt das Abspeichern der Daten in einer geeigneten Datenbank.

Im fünften Kapitel geht es um die Validierung der generierten Daten mithilfe der Validierungssprache SHACL.

Den Abschluss bildet das sechste Kapitel mit einem Fazit und einem Ausblick auf weiteres Vorgehen.

2 Grundlagen

In diesem Kapitel werden die für diese Arbeit relevanten Grundlagen in Bezug auf das Semantic Web und die dazugehörigen Technologien erklärt. Ein Hauptaugenmerk wird hierbei auf Ontologien und Ontologiesprachen gelegt, um Werkzeuge zu präsentieren, die eine maschinenlesbare Wissensrepräsentation ermöglichen.

2.1 Semantic Web

Das Internet besteht aus Nutzersicht aus leicht verständlichen Informationen. Ein Anwender ist in der Lage eine Webseite aufzurufen, die enthaltenen Informationen zu erfassen und sie je nach Komplexität zu verstehen. Um den letzten Punkt noch weiter auszuführen: neu gewonnene Informationen können nicht nur verstanden, sondern gleichzeitig mit bereits vorhandenem Wissen und Erfahrungen in Verbindung gesetzt werden. So kann ein Mensch aus den Überschriften eines Artikels: „Traumtor in der Nachspielzeit“ und „Enttäuschte Hamburger fahren ohne Punkte nach Hause“ ableiten, dass eine Hamburger Fußballmannschaft ein Auswärtsspiel durch ein spätes Tor verloren hat. Dieses Wissen bleibt einer Maschine verborgen, weil es zwischen den Zeilen steht, also impliziert wird. Ein Problem ergibt sich, wenn ein Nutzer über eine Suchmaschine Informationen über genau dieses Fußballspiel finden möchte und die Begriffe „HSV“ und „Auswärtsspiel“ eingibt. Würden Suchmaschinen ausschließlich stichwortbasiert arbeiten, könnte der oben erwähnte Artikel nur gefunden werden, wenn genau die genannten Begriffe auch in dem Artikel vorkommen. Google und Co. leisten hier mittels statistischer Methoden schon sehr gute Arbeit, aber im Endeffekt basieren alle diese Verfahren auf der Analyse von Zeichenketten.

Ein weiteres Problem stellt die Heterogenität des Internets dar. In ihrer Gänze ist keine Webseite so aufgebaut wie eine andere. Daten liegen in den unterschiedlichsten Dateiformaten und mit verschiedensten Kodierungen vor. Außerdem gibt es keine einheitliche Sprache oder allgemein gültige Grammatik. Informationen aus unterschiedlichen Teilen des Internets miteinander zu vernetzen, also zu integrieren, ist nahezu unmöglich. Zum Beispiel ergibt die Information, dass New York in den USA und Hamburg in Deutschland liegt, zusammen mit der Information, dass zwischen den Lokalzeiten der Orte fünf Stunden Differenz herrscht, dass möglichst nicht vor 14 Uhr in New York angerufen werden sollte. Hierbei handelt es sich ebenfalls um implizites Wissen, in diesem Fall sogar als Produkt aus verschiedenen Quellen.

Die Lösung dieser Probleme strebt das Semantic Web an. Es basiert auf einem Vorschlag des Begründers des World Wide Web (WWW) Tim Berners-Lee, das Internet

auf die nächste Stufe zu heben und es zusätzlich zur Ausrichtung auf den Menschen auch maschinenlesbar und -interpretierbar zu machen. Es zeigt mittels einer Art globaler Datenbank einen Weg auf, Informationen auf dem Bedeutungslevel miteinander zu verknüpfen. Um dies erreichen zu können, bedarf es eines formal klar definierten Standards, der gleichzeitig flexibel und erweiterbar genug ist, um allen Anforderungen gerecht zu werden. Das World Wide Web Consortium (W3C) definiert solche Standards und hat im Bezug auf das Semantic Web grundlegende Spezifikationen für das Resource Description Framework (RDF) und die Web Ontology Language (OWL) erarbeitet.

Ein weiterer zentraler Punkt des Semantic Web sind Methoden aus vorhandenem, bekannten Wissen mittels Maschinen neue Schlussfolgerungen zu ziehen. In dieser Arbeit wird Maschinen zur Vereinfachung die menschliche Fähigkeit zugewiesen, die Bedeutung von Informationen zu verstehen. Übergeordnet geht es aber um das Finden von Wegen und Methoden *„Informationen so zu repräsentieren, dass Maschinen damit in einer Art und Weise umgehen können, die aus menschlicher Sicht nützlich und sinnvoll erscheint. Dabei mag das entstehende Verhalten von Systemen in den Augen mancher Nutzer durchaus als intelligent erscheinen.“* (Hitzler et al. 2008: 12)

Web 1.0 / 2.0 / 3.0 Summary

Crawl	Walk	Run
Web 1.0	Web 2.0	Web 3.0
Mostly Read-Only	Wildly Read-Write	Portable & Personal
Company Focus	Community Focus	Individual Focus
Home Pages	Blogs / Wikis	Lifestreams / Waves
Owning Content	Sharing Content	Consolidating Content
Web Forms	Web Applications	Smart Applications
Directories	Tagging	User Behavior
Page Views	Cost Per Click	User Engagement
Banner Advertising	Interactive Advertising	Behavioral Advertising
Britannica Online	Wikipedia	The Semantic Web
HTML / Portals	XML / RSS	RDF / RDFS / OWL

Abbildung 2.1: Vergleich der Webversionen (Alyona 2012)

Häufig wird in Verbindung mit dem Semantic Web auch vom Web 3.0, in Ab-

grenzung zu Web 1.0 und Web 2.0, gesprochen. Das Web 1.0 besteht aus statischen Seiten, die eine Interaktion der Nutzer auf ein Minimum reduzieren und im Prinzip reinen Lesezugriff geben. Im Vergleich dazu erlaubt das Web 2.0 einen deutlich höheren Grad der Interaktion und neben Lese- auch einen Schreibzugriff. Stellvertretend für das Web 2.0 stehen Dienste wie YouTube oder Wikipedia. Auch die sozialen Netzwerke wie Facebook oder Twitter können dem Web 2.0 zugeschrieben werden. Beim Web 3.0 stehen Dinge wie die Nutzerinteraktion eher im Hintergrund. Es geht um semantische, also bedeutungstragende, Verbindungen zwischen Inhalten und die daraus abzuleitenden neuen Informationen. Die Technik der Verarbeitung ist dabei Linked Data und die Sprache RDF. (Alyona 2012)

2.2 Linked (Open) Data

Wie im vorherigen Abschnitt bereits angedeutet, kann das Semantic Web nicht allein als ein Web von Daten verstanden werden. Verlinkungen zwischen diesen Daten sind der zentrale Aspekt von Linked Data. Wenn nur genug Daten vorhanden sind und diese Daten adäquat miteinander verknüpft wurden, besteht die Möglichkeit mit Hilfe weniger Daten auf viele zu schließen und ähnliche Daten zu entdecken. Der Begriff Linked Data geht ebenfalls auf Berners-Lee zurück. Ähnlich zu dem Web of Hypertext baut das Web of Data auf Dokumenten auf. Der Unterschied besteht in der Art der Verlinkungen. Beim Web of Hypertext werden die Beziehungen über die HTML eingegangen, beim Web of Data erfolgt die Verbindung über das RDF. Durch einen *Uniform Resource Identifier* (URI) kann eine Zeichenkette, die einem festgelegten Muster folgt, im Internet angesprochen werden. Diese URIs stehen stellvertretend für ein abstraktes oder physisches Objekt oder Konzept. Sie bestehen aus fünf Bestandteilen: *scheme*, *authority*, *path*, *query* und *fragment* (siehe Abbildung 2.2).

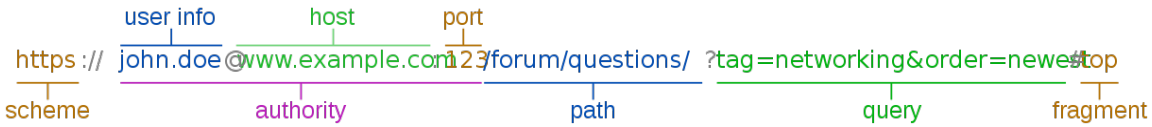

Abbildung 2.2: Bestandteile eines URI (Wikipedia 2020)

Ein URI kann somit jede Ressource einzigartig identifizieren. Darauf aufbauend hat Berners-Lee vier Regeln für Linked Data aufgestellt:

1. *„Use URIs as names for things.“*
2. *„Use HTTP URIs so that people can look up those names.“*
3. *„When someone looks up a URI, provide useful information, using the standards (RDF*, SPARQL)“*

4. *„Include links to other URIs. so that they can discover more things.“* (Berners-Lee 2009)

Diese Regeln sind nicht als Gesetze zu verstehen, eher als Richtlinie, die helfen soll die Qualität der Daten, die im Internet zu erreichen sind, zu erhöhen. Die erste Regel besagt, dass URIs benutzt werden sollen und zwar für jedes Datum, egal ob es ein generelles Konzept oder ein konkretes, physisches Objekt repräsentiert. Regel zwei besagt, dass für Daten, die hinter URIs zu finden sind, das HTTP-Protokoll verwendet werden sollte, da es der universelle Zugriffsmechanismus im Web ist und auch im klassischen WWW verwendet wird. Die dritte Regel pocht auf das Einhalten von Standards, in diesem Fall RDF und SPARQL. So ist beispielsweise die Einigung im klassischen Web auf HTML einer der wichtigsten Faktoren für dessen Erfolg. Regel vier ist der Kern von Linked Data und besagt, dass Daten sich zu einem Netz (*web*) verbinden sollten, um so neue Informationen zu entdecken. So hängt der Wert der eigenen Daten auch von den verlinkten Daten ab.

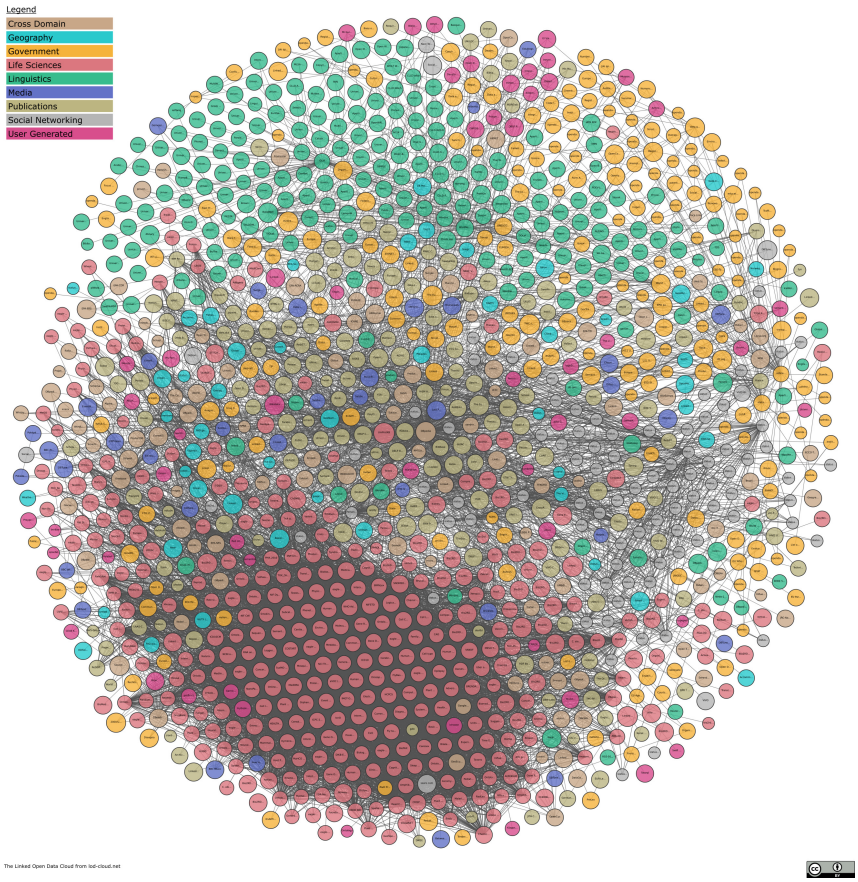

Abbildung 2.3: Visualisierung der LOD-Cloud (LODCloud 2020)

Eine Erweiterung von Linked Data stellt Linked Open Data (LOD) dar. Als Linked Open Data lassen sich Daten bezeichnen, die den Ansprüchen an Linked Data

genügen und zusätzlich unter einer offenen Lizenz frei im Netz verfügbar sind. In Abbildung 2.3 sind Datensätze und ihre Beziehungen untereinander visualisiert.

Für Open Data hat Berners-Lee ein Fünf-Sterne-Bewertungssystem vorgeschlagen, bei dem jeder neue Stern auch alle Anforderungen der vorherigen erfüllen muss. Null Sterne gibt es für Daten, die überhaupt nicht im Internet verfügbar sind. Einen Stern erhalten Datensätze, wenn sie unter einer freien Lizenz im Web zu finden sind, unabhängig vom Format. Dazu würden beispielsweise auch handschriftlich eingescannte Dokumente zählen. Zwei Sterne gibt es für Daten, die einer Struktur folgen, zum Beispiel Excel-Tabellen. Drei Sterne erhalten Daten, die zusätzlich auf proprietäre Formate verzichten und freie Formate wie CSV nutzen. Daten, die mittels URIs identifiziert werden, erhalten vier Sterne. Die maximale Anzahl an Sternen werden den Daten zugeschrieben, die zusätzlich auf andere Daten verlinken, um neue Kontexte und Informationen abzuleiten. Linked Open Data ist also per Definition Fünf-Sterne-Open Data. (Hausenblas 2015)

In dieser Arbeit werden im Prinzip Fünf-Sterne-Daten angestrebt, allerdings wird der Datensatz nicht öffentlich zugänglich sein, da er datenschutzrelevante Informationen wie Telefonnummern und Wohnadressen enthält. Damit handelt es sich nicht um Open Data, aber trotzdem um Linked Data.

2.3 Resource Description Framework (RDF)

Das Resource Description Framework (RDF) ist die Sprache des Semantic Web (vgl. Abschnitt 2.1) und bildet somit einen Grundbaustein des sogenannten Semantic Web Technology Stack (siehe Abbildung 2.4). Die Beispiele in diesem Abschnitt sind (Hitzler et al. 2008) entnommen bzw. an sie angelehnt.

Das RDF ermöglicht die formale Beschreibung von Ressourcen. Ressourcen können alles sein, über das Aussagen getroffen werden können. So sind sind beispielsweise Webseiten und Bilder im Web Ressourcen, aber auch Dinge der realen Welt wie Orte oder Gegenstände und abstrakte Konzepte wie Prozesse oder Zustände sind Ressourcen. Im RDF werden Ressourcen URIs zugewiesen, die diese einzigartig referenzieren und sie so identifizierbar machen. So wird ein Datenaustausch über das Web ermöglicht, ohne dass dabei die Bedeutungen der Ressourcen verloren gehen. Im Gegensatz zu Formaten wie XML und HTML steht beim RDF die Darstellung der Dokumente nicht im Vordergrund. Heutzutage existieren eine Vielzahl von Werkzeugen im Umgang mit dem RDF, wie beispielsweise Triple Stores, die Systeme zur Speicherung und Auswertung von RDF-Daten sind, oder RDF-Schnittstellen für jede gängige Programmiersprache. (Hitzler et al. 2008: 35)

RDF-Dokumente zeichnen gerichtete Graphen aus. Sie bestehen aus einer Menge von Knoten (*nodes*) und Kanten (*edges*). Eine Kante verbindet immer zwei Knoten miteinander. Abbildung 2.5 zeigt ein einfaches Beispiel mit zwei Knoten und einer Kante, das einer Wohnung eine Stadt zuweist.

Informationen im XML-Format werden anders als im RDF in einer Baumstruktur

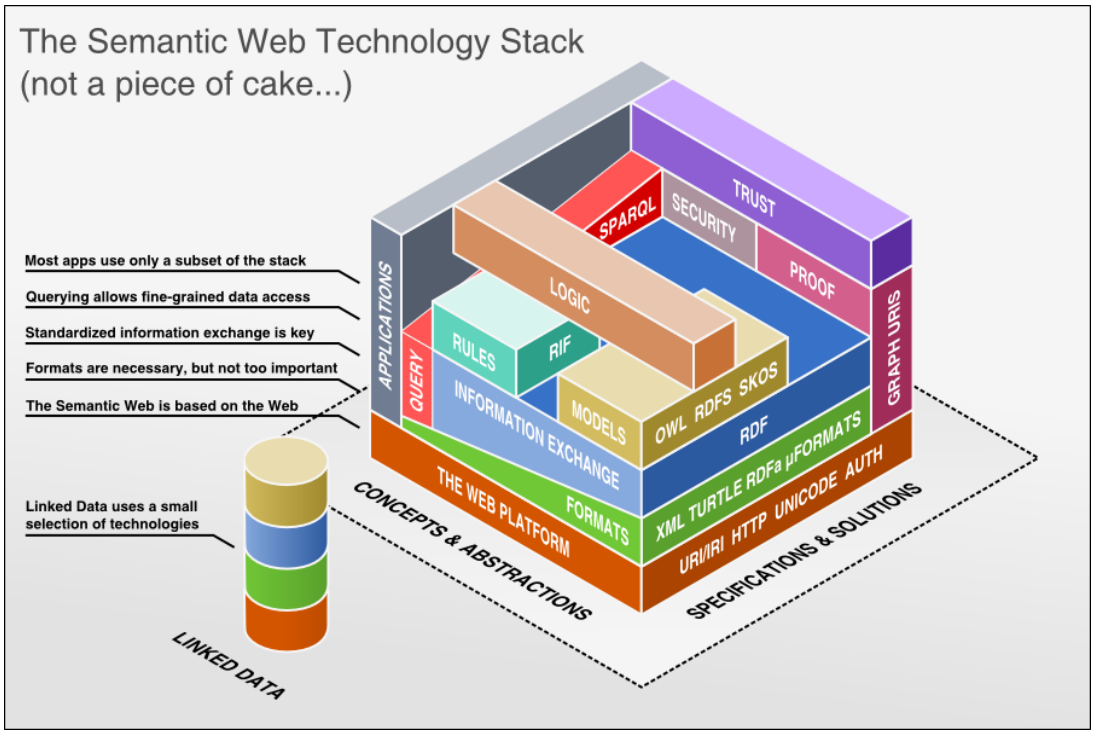

Abbildung 2.4: Semantic Web Technology Stack (Nowack 2009)

dargestellt. Der Grund hierfür sind die häufig streng hierarchischen Strukturen von Informationen in solchen elektronischen Dokumenten. Außerdem lassen sich Informationen über Baumstrukturen effizient finden und verarbeiten. RDF nutzt keine Bäume, da es nicht explizit für hierarchische Strukturen entwickelt wurde. Die Beziehung *„liegtIn“* aus Abbildung 2.5 lässt sich keiner der beiden Knoten wirklich unterordnen und daher als eine grundlegende Informationseinheit bezeichnen. Aus einer Ansammlung solcher Beziehungen entsteht zwangsläufig ein Graph, aber kein Baum. Ein weiterer Grund ist der dezentrale Aufbau der Informationslandschaft im Internet. Im RDF ist es leicht möglich Daten aus vielen Quellen miteinander zu kombinieren. Beim Versuch mehrere Baumstrukturen im XML-Format miteinander zu kombinieren wird schnell eine Grenze erreicht. So können Daten aus der einen Quelle, auf einer anderen Ebene des Baumes gespeichert sein, als Daten aus einer anderen Quelle. Im Gegensatz zu XML ist RDF demnach gut geeignet für die Komposition dezentraler Informationen. (Hitzler et al. 2008: 36-37)

Im RDF werden sowohl Knoten als auch Kanten immer eindeutige URIs zugewiesen (vgl. Abbildung 2.6). URIs sind eine Verallgemeinerung von URLs und beschreiben Adressen im Web. Dabei ist es nicht entscheidend, dass Ressourcen auch tatsächlich über diese URIs online abgerufen werden können. Wichtiger ist, über die einzigartigen Adressen Verwechslungen auszuschließen. Um weiterhin zu verhindern, dass URIs mehrfach vergeben werden, sollte auf einen ausreichend individuellen Namensraum

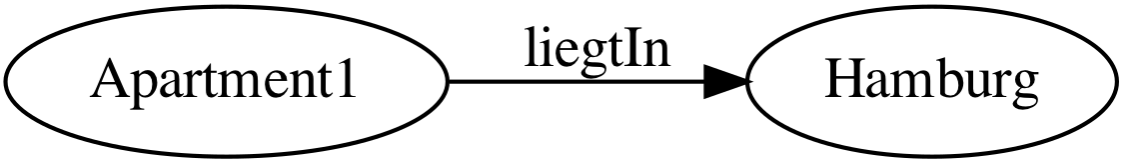

Abbildung 2.5: Beispiel eines Graphen

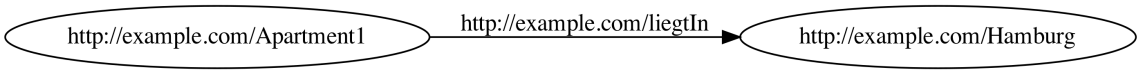

Abbildung 2.6: Beispiel eines Graphen mit URIs

(*namespace*) geachtet werden. Ein mehrmals verschachtelter Pfad (vgl. Abbildung 2.2) oder eine selbst kontrollierte Domain sollte dies immer gewährleisten können. (Hitzler et al. 2008: 38, 48-49)

Eine Sonderstellung im RDF-Kontext nehmen sogenannte Literale ein. Dies sind reservierte Bezeichner für Ressourcen eines bestimmten Datentyps, die keine URI besitzen. Jedes Literal wird durch eine Zeichenkette beschrieben. In der Abbildung 2.7 ist dies „1899160“. Ohne explizites Erwähnen des Datentyps gilt dieses Literal als ungetypt. Das RDF selbst stellt keine Datentypen zur Verfügung, empfiehlt aber das XML-Schema für diesen Zweck. In diesem Fall wäre es eine ganze Zahl (*integer*). Abbildung 2.8 zeigt wie ein getyptes Literal aussieht. In der grafischen Darstellung werden Literale mit einem Rechteck gezeichnet, um sie von Ressourcen mit URIs zu unterscheiden. Eine weitere Besonderheit der Literale ist, dass über sie keine Aussagen getroffen werden können. Sie dürfen also niemals Ausgangspunkt einer Kante sein. Eine Kante darf ebenfalls nicht durch ein Literal beschriftet werden, dies würde allerdings auch keinen sinnvollen Zweck erfüllen. (Hitzler et al. 2008: 39, 50ff.)

Bisher wurden in Verbindung mit RDF nur grafische Darstellungen gezeigt. Diese eignen sich allerdings weder zum Datenaustausch noch zur Verarbeitung durch Maschinen. Um diese Graphen maschinenlesbar zu machen, werden sie in Zeichenketten

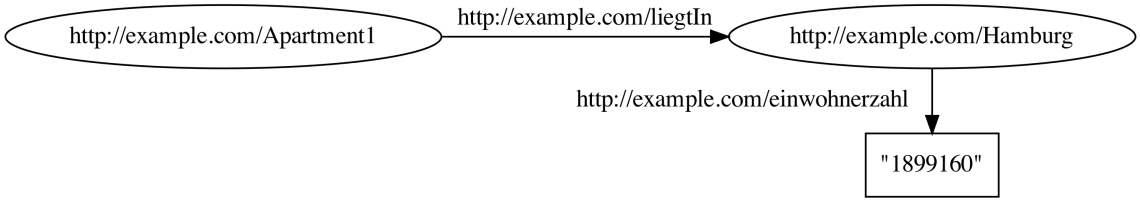

Abbildung 2.7: Beispiel eines Graphen mit ungetyptem Literal

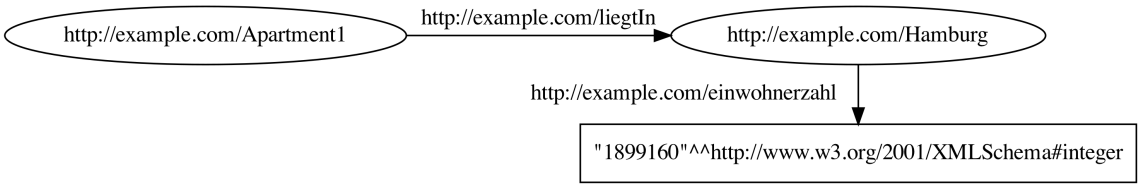

Abbildung 2.8: Beispiel eines Graphen mit getyptem Literal

überführt. Dies wird Serialisierung genannt. Bei diesem Prozess wird der Graph in kleine Teilgraphen zerlegt, die jeweils nur aus zwei Knoten und einer sie verbindenden Kante bestehen. Der Ausgangsknoten wird dabei *Subjekt*, der Zielknoten *Objekt* und die Kante *Prädikat* genannt. Diese drei Bestandteile bilden zusammen ein sogenanntes *Tripel*. Subjekte und Prädikate sind dabei immer bezeichnet durch URIs, lediglich Objekte können auch Literale beeinhalten. Es existieren verschiedene Serialisierungen, die sich in ihrer Syntax und dem Dateiformat unterscheiden. Die gängigsten Serialisierungen sind N-Triples, die darauf aufbauende Terse RDF Triple Language (Turtle) und RDF/XML. Turtle gilt allgemein hin als übersichtlichste Serialisierung und ist deshalb am besten von Menschen lesbar. (Hitzler et al. 2008: 40-42) Der Graph aus Abbildung 2.8 würde in Turtle so aussehen:

```
1 <http://example.com/Apartment1> <http://example.com/liegtIn>
  ↪ <http://example.com/Hamburg> .
2 <http://example.com/Hamburg> <http://example.com/einwohnerzahl>
  ↪ "1899160"^^<http://www.w3.org/2001/XMLSchema#integer> .
```

Subjekt, Prädikat und Objekt werden jeweils durch ein Leerzeichen getrennt. URIs werden in spitze Klammern gefasst und ein Tripel wird mit einem Punkt abgeschlossen. Diese Darstellung lässt sich noch weiter vereinfachen, indem verhindert wird, dass Teile eines URIs, die sich gleichen, jedes Mal voll ausgeschrieben werden müssen. Hierfür gibt es Präfixe:

```
1 @prefix ex: <http://example.com/> .
2 @prefix xsd: <http://www.w3.org/2001/XMLSchema#> .
3
4 ex:Apartment1 ex:liegtIn       ex:Hamburg .
5 ex:Hamburg    ex:einwohnerzahl "1899160"^^xsd:integer .
```

In Zeile 4 und 5 stehen die eigentlichen Tripel. Die Präfixe sind im Prinzip frei wählbar, allerdings geben viele Vokabulare und Ontologien vor, welche Präfixe verwendet werden sollten. Dies hat auf die Funktion keinerlei Einfluss und dient nur der Lesbarkeit. Um einmal den Kontrast zu einer von Menschen deutlich schlechter lesbaren Serialisierung zu verdeutlichen, so sehen die gleichen beiden Tripel in RDF/XML aus:

```
<?xml version="1.0" encoding="utf-8" ?>
<rdf:RDF xmlns:rdf="http://www.w3.org/1999/02/22-rdf-syntax-ns#"
         xmlns:ex="http://example.com/">
  <rdf:Description rdf:about="http://example.com/Apartment1">
    <ex:liegtIn>
      <rdf:Description rdf:about="http://example.com/Hamburg">
        <ex:einwohnerzahl
         ↪  rdf:datatype="http://www.w3.org/2001/XMLSchema#integer">
            1899160
        </ex:einwohnerzahl>
      </rdf:Description>
    </ex:liegtIn>
  </rdf:Description>
</rdf:RDF>
```

Trotz der deutlich schlechteren Lesbarkeit wird RDF/XML häufig verwendet, weil XML ein sehr weit verbreitetes Format ist und Daten in dieser Serialisierung von jedem XML-Parser verstanden werden. Mittlerweile existieren dutzende Tools um nahezu beliebig große Datenmengen von der einen, in die andere Serialisierung zu überführen. In dieser Arbeit wird zwecks Lesbarkeit ausschließlich die Turtle-Serialisierung zum Einsatz kommen.

Eine weitere Turtle-Kurzform kann verwendet werden, wenn ein Graph mehrere Tripel des selben Subjekts bzw. des selben Subjekts und Prädikats enthält:

```
@prefix ex: <http://example.com/> .

ex:Apartment1 ex:liegtIn    ex:Hamburg .
ex:Apartment1 ex:hatMerkmal ex:Duschwanne .
ex:Apartment1 ex:hatMerkmal ex:Einbauküche .
```

Bei der Kurzschreibweise trennt ein Semikolon Tripel mit gleichem Subjekt und Kommata trennen Tripel, die sich Subjekt und Prädikat teilen. Abgeschlossen werden diese Strukturen ebenfalls mit einem Punkt:

```
@prefix ex: <http://example.com/> .

ex:Apartment1 ex:liegtIn    ex:Hamburg ;
              ex:hatMerkmal ex:Duschwanne ,
                            ex:Einbauküche .
```

Ein Problem von RDF-Daten ist, dass ohne weiteres verschiedene URIs für die gleiche Ressource verwendet werden können und umgekehrt, lässt sich auch die gleiche URI für unterschiedliche Ressourcen benutzen. Um diesem Problem zu begegnen, wurde sich im Laufe der Jahre auf wohlformulierte Vokabulare verständigt. Darunter

wird eine Zusammenstellung von URIs mit klar definierter Bedeutung verstanden. Ein prominentes Beispiel ist die Ontologie *Friend Of A Friend* (FOAF), die geeignet ist, Personen und ihre Beziehungen untereinander zu beschreiben. So findet sich beispielsweise unter der URI `http://xmlns.com/foaf/spec/#term_Person` die Beschreibung: *„The Person class represents people. Something is a Person if it is a person. We don't nitpic about whether they're alive, dead, real, or imaginary. The Person class is a sub-class of the Agent class, since all people are considered 'agents' in FOAF.“* (Brickley & Miller 2014)

Obwohl durch URIs Konzepte eindeutig zugeordnet werden können, versetzen solche Vokabulare eine Maschine nicht in die Lage, Konzepte wie *Person* wirklich zu verstehen. Die inhaltliche Bedeutung der Daten kann nach wie vor nur von einem Menschen erfasst werden. Komplexe Zusammenhänge zwischen Ressourcen lassen sich aber maschinenlesbar ausdrücken, und zwar mit Hilfe von Ontologien und Ontologiesprachen wie RDF-Schema und OWL. Häufig existiert allerdings kein passendes Vokabular für die Daten, die beschrieben werden sollen. In solchen Fällen kann es nötig sein, eigene URIs einzuführen und sein eigenes Vokabular aufzustellen. (Hitzler et al. 2008: 47-48)

2.4 Ontologien

Der Ontologiebegriff ist sehr weit gefasst. Je nach Disziplin kann er stark abweichende Bedeutungen haben. Ursprünglich stammt er aus der Philosophie und *„bedeutet die „Lehre vom Sein“. Ontologie soll – und das gilt für alle Philosophen – die umfassendste aller Wissenschaften sein, insofern als sie alles Seiende behandelt.“* (Busse et al. 2012) In dieser Definition steckt schon die Uneinigkeit über den Begriff. Jeder Philosoph wird seine allumfassende Wissenschaft des Seins anders aufstellen, es existiert also keine einheitliche Bedeutung des Terms Ontologie.

In der Informatik ist der Begriff etwas deutlicher definiert, aber auch hier gibt es verschiedenste Formulierungen. So heißt es zum Beispiel bei Gruber: *„An ontology is a formal explicit specification of a shared conceptualization.“* (Gruber 1993)

Oder bei Uschold und Grüninger: *„An ontology is a shared understanding of some domain of interest.“* (Uschold & Grüninger 1996)

Es geht also um eine festgelegte Domäne bzw. eine festgelegte Konzeptualisierung und die Begriffe und Beziehungen, die diese beschreiben, zwecks eines gemeinsamen Verständnisses. Damit diese Begriffe und Beziehungen verstanden werden können, bedarf es einer formalen Definition. Formal bedeutet, dass sie in einer Sprache formuliert sind, die einer festgelegten Syntax und Semantik folgen und die von Maschinen verarbeitet werden kann. Ontologien sollen also die Kommunikation zwischen Entitäten ermöglichen, seien es Maschinen oder Menschen. (Dengel 2012: 65-66)

Der Ontologiebegriff ist eng mit dem Begriff der Wissensbasis (*knowledge base*) verknüpft. Eine Wissensbasis im Ontologiekontext besteht aus zwei Bestandteilen, zum einen aus einem Terminologischen Formalismus (*TBox*), der die Klassen von

Objekten einer Domäne und ihre möglichen Eigenschaften beinhaltet und zum anderen aus einem Assertionalen Formalismus (*ABox*), der die Instanzen oder Individuen von Klassen und deren gesetzten Eigenschaften und Beziehungen zueinander enthält. Ontologien werden eigentlich nur der TBox zugeordnet und die ABox enthält dann die Daten, die mittels der Ontologie ausgedrückt werden. In der Praxis verschwimmen die Grenzen zwischen TBox und ABox und aus Modellierungsgründen enthalten Ontologien häufig auch Instanzen, die dann wiederverwendet werden. (May 2006: 492-496)

So hat sich aus dem philosophischen Ansatz einer umfassenden Weltsicht, in der Informatik eine Art der konzeptuellen Datenmodellierung entwickelt. Die Aufgaben von Ontologien umfassen also den Datenaustausch zwischen Entitäten, Ausdruck von Semantik strukturierter und semistrukturierter Informationen und die Vereinheitlichung von Begriffen einer Domäne. (Blumauer & Pellegrini 2006: 12)

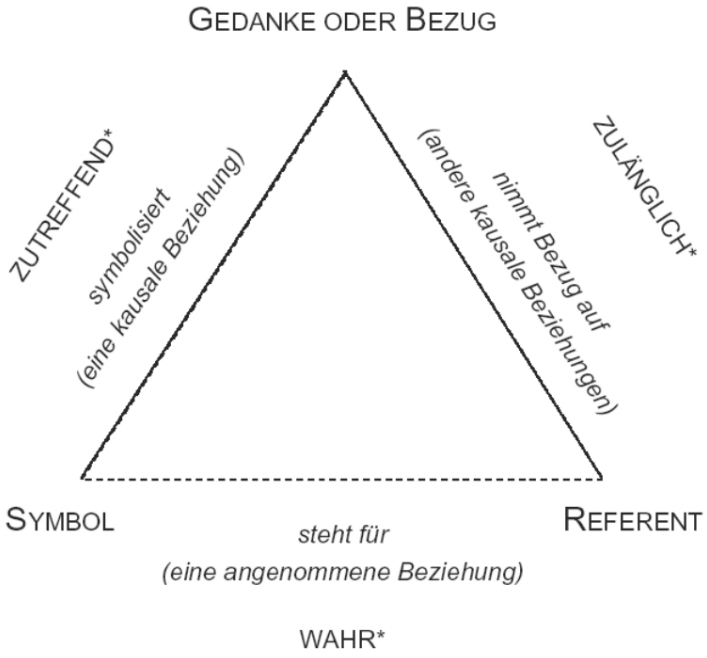

Abbildung 2.9: Das semiotische Dreieck (Schirra 2017)

Zusätzlich ermöglicht eine Ontologie eine Reduzierung von Abbildungen von Symbolen auf Referenten (siehe Abbildung 2.9). So kann beispielsweise das Symbol *Jaguar* ohne weiteren Kontext die Referenten *Automarke*, *Auto*, *Großkatze* zulassen. Eine biologische Ontologie könnte ein kontrolliertes Vokabular liefern, welches dem Symbol *Jaguar* über einen Bezug klar den Referenten *Großkatze* zuweist. Die Zahl der möglichen Abbildungen reduziert sich im Idealfall soweit, dass Verwechslungen ausgeschlossen sind. (Stuckenschmidt 2011: 5-8)

Es lassen sich zwei Arten von Ontologien anhand der Allgemeingültigkeit unter-

scheiden:

- *„Upper Ontologies beschreiben Begriffe von hoher Allgemeingültigkeit, die von vielen Anwendern in unterschiedlichen Szenarien sinnvoll eingesetzt werden können. Beispiele sind etwa Konzepte zur Beschreibung von Raum und Zeit, von physikalischen Eigenschaften oder auch von weitverbreiteten Informationselementen.“*
- *„Domain Ontologies beschreiben dann die spezielleren, für einzelne Anwendungsbereiche brauchbaren Konzepte. Diese enthalten typischerweise detailliertes und umfangreiches Wissen über den spezifischen Anwendungsbereich, sind aber anderweitig oft nicht sinnvoll einzusetzen.“* (Dengel 2012: 67)

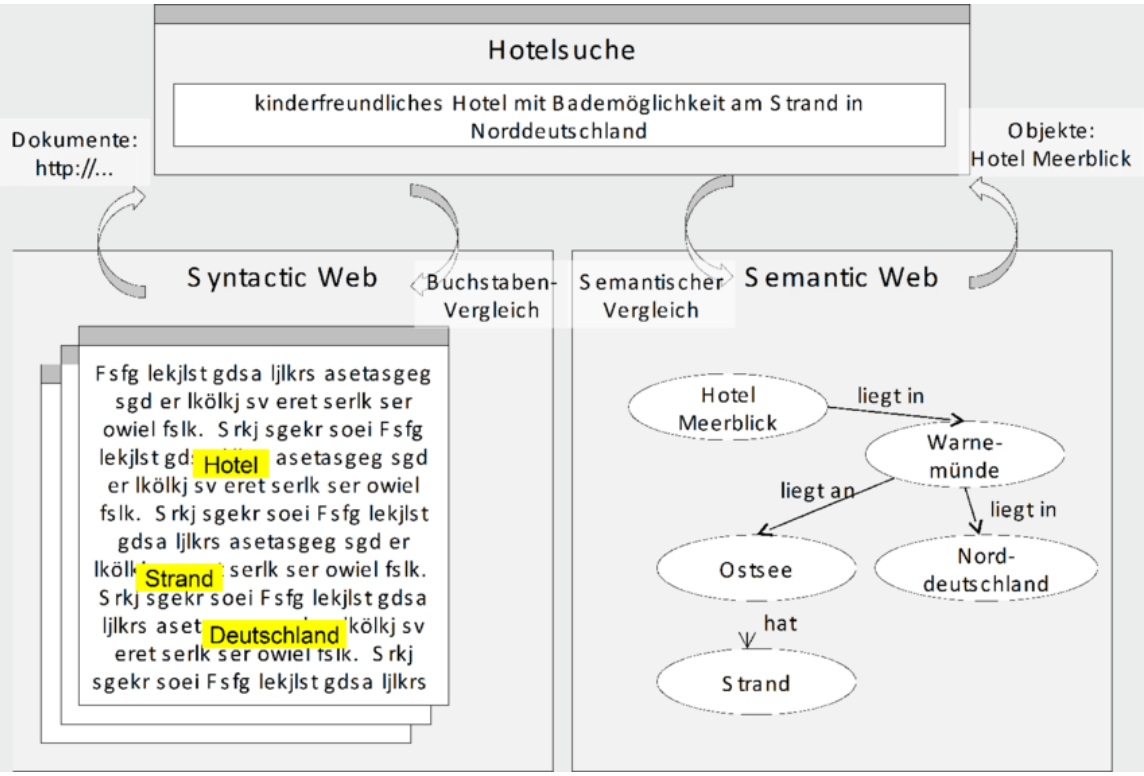

Abbildung 2.10: Semantische Hotelsuche (Busse et al. 2012)

Normalerweise gelten für Wissensbasen die sogenannte *Closed-World-Assumption*. Diese besagt, dass Dinge, die nicht in der Wissensbasis enthalten sind, auch nicht existieren bzw. unwahr sind. So kann bei einer Kundendatenbank davon ausgegangen werden, dass Personen, die nicht in der Datenbank auftauchen, keine Kunden sind. Diese Sicht vereinfacht den Umgang mit der Wissensbasis, führt aber auch starke Restriktionen ein. So wäre es theoretisch möglich, dass ein tatsächlicher Kunde aufgrund

eines Datenbankfehlers nicht auffindbar ist. An so einer Stelle scheitert die *Closed-World-Assumption*. Ontologien folgen der *Open-World-Assumption*, die besagt, dass wenn Fakten nicht vorhanden sind, nicht automatisch angenommen werden kann, sie seien falsch. Sie können falsch sein, sie können aber auch richtig sein. Es wird einfach keine Aussage darüber getroffen, solange es nicht eindeutig bewiesen wird. Stattdessen wird mit den Fakten gearbeitet, die vorhanden sind, ohne Unbekanntes auszuschließen. Dies basiert auf dem Ansatz, dass sich viel von unserem Wissen nicht vollständig formal repräsentieren lässt. (May 2006: 495-496, 500)(Busse et al. 2012)

So zeigen folgende Fragen die Schwierigkeit einer *Closed-World-Assumption* auf:

- Gehören Rollschuhe zum Konzept Fahrzeug?
- Haben alle Landfahrzeuge Räder?
- In welche Kategorie fallen Schlitten?

Zum Abschluss dieses Abschnitts soll noch ein kleines Anwendungsbeispiel einer Ontologie präsentiert werden (siehe Abbildung 2.10). Hier geht es um zwei Systeme, die Hotelvorschläge basierend auf formulierten Anforderungen liefern sollen. Im konkreten Fall sind die Anforderungen: *kinderfreundlich*, *Bademöglichkeit*, *Strand* und *Norddeutschland*. Auf der linken Seite ist der Prozess im Syntactic Web visualisiert (vgl. Abschnitt 2.1). Das rechts dargestellte ontologiebasierte System könnte so vorgehen:

1. Mit einer lexikalen Ontologie (z.B. *WordNet*[1]) sucht das System nach ähnlichen oder synonymen Worten zu den Anforderungsbegriffen: *kinderfreundlich* → *familienfreundlich*, *Strand* → *Meer* → *Bademöglichkeit*
2. Mit einer Ontologie werden den geografischen Anforderungen entsprechend Hotels über ihren Standort gesucht. (siehe Abbildung 2.11)
3. SPARQL-Query um, basierend auf der transitiven Folgerung, Hotels zu finden: `SELECT ?hotel WHERE {?hotel ex:liegtIn ex:Norddeutschland .}`
4. Die gefundenen Hotels mit den ähnlichen und synonymen Begriffen abgleichen. (Busse et al. 2012)

2.4.1 RDF-Schema (RDFS)

Das RDF stellt erst einmal nur ein Datenmodell zur Verfügung, welches verfeinert werden kann. Es lassen sich also semantische Anwendungen entwickeln, die eindeutig Wissen verknüpfen. Sobald mehr Wissen über diese Daten ausgedrückt werden soll, stößt reines RDF an seine Grenzen. Wird das Beispiel am Schluss von Abschnitt 2.3 noch einmal betrachtet, könnte die Idee aufkommen der Ressource *Apartment1* die

[1] https://wordnet.princeton.edu

```
ex:liegtIn rdf:type owl:ObjectProperty ,
                    owl:TransitiveProperty .

ex:HotelMeerblick          ex:liegtIn ex:Warnemünde .
ex:Warnemünde              ex:liegtIn ex:Mecklenburg-Vorpommern .
ex:Mecklenburg-Vorpommern ex:liegtIn ex:Norddeutschland .
# Transitive Folgerung:
ex:HotelMeerblick          ex:liegtIn ex:Norddeutschland .
```

Abbildung 2.11: Beispiel einer transiviten Folgerung

Klasse *Wohnung* und den beiden Eigenschaften von *Apartment1* die Klasse *Wohnungsmerkmal* zuzuweisen. Weiterhin ließe sich über die Relation *hatMerkmal* sagen, dass sie Wohnungen Merkmale zuordnet. Ein Mensch nimmt diese Generalisierungen intuitiv an, einer Maschine müssen solche Zusammenhänge irgendwie verständlich gemacht werden, damit sie intelligente Schlüsse daraus ziehen kann. Sonst sind die Daten nichts weiteres als menschengemachte Zeichenketten. (Hitzler et al. 2008: 66-67)

An dieser Stelle kommt das RDF-Schema (RDFS) ins Spiel. Es nutzt die Werkzeuge des RDF um Modellierungsmöglichkeiten einzuführen und so das RDF mit einer extern spezifizierten Semantik zu erweitern. Es erlaubt terminologisches Wissen (siehe Abschnitt 2.4) über die Begriffe in einem nutzerdefinierten Vokabular zu spezifizieren. RDF selbst kann Ressourcen nur über benannte Properties (Prädikate) oder Literale verbinden. Es gibt keine Möglichkeit, einer Ressource eine Klasse oder einen Typ zuzuweisen. Außerdem gibt es im RDF keinen Weg, Properties auf Klassen von Ressourcen einzuschränken. Diese Lücken werden von RDFS gefüllt.

Im Vergleich zu objektorientierten Sprachen ist RDFS *property-zentriert*. Beispielsweise hat in Java eine Klasse *Wohnung* das Attribut *Makler* vom Typ *String*. In RDFS hat eine Property *Makler* den Definitionsbereich (*domain*) *Wohnung* und den Wertebereich (*range*) *String*. Im Rückblick auf die Tripelstruktur von Subjekt, Prädikat und Objekt bedeutet dies, dass das Prädikat *Makler* als Subjekte nur Instanzen der Klasse *Wohnung* und als Objekte nur Literale vom Typ *String* erlaubt. Sollen die erlaubten Werte für *Makler* noch genauer spezifiziert werden, wäre das Einführen einer Klasse *Person* denkbar. Der Wertebereich von *Makler* könnte dann auf Instanzen dieser Klasse begrenzt sein.

Um diese Strukturen auch formal beschreiben zu können, führt RDFS vordefinierte Klassen und Properties ein. Die wichtigsten Klassen sind:

- rdfs:Resource
- rdfs:Class
- rdf:Property

rdfs:resource ist das Äquivalent zu *Thing* oder *Objekt* in objektorientierten Programmiersprachen. Alle Ressourcen sind von dieser Klasse abgeleitet. Mit *rdfs:Class* wird eine Ressource zu einer Klasse und entsprechend wird mit *rdfs:Property* eine Ressource zu einer Property. Die wichtigsten vordefinierten Properties sind:

- rdf:type
- rdfs:subClassOf
- rdfs:subPropertyOf
- rdfs:domain
- rdfs:range

rdf:type weist Ressourcen einer Klasse zu. *rdfs:subClassOf* erlaubt das Bilden von Unterklassen und entsprechend erlaubt *rdfs:subPropertyOf* das Bilden von spezifischeren Properties. *rdfs:domain* und *rdfs:range* wurden schon implizit erwähnt und sind Properties für Einschränkungen (*constraints*) von Definitons- und Wertebereich. (Dengel 2012: 124-125)(Allemang et al. 2020: 201ff.)

2.4.2 Web Ontology Language (OWL)

Es gibt Fälle in denen RDFS nicht ausreicht, um Ressourcen detailliert zu beschreiben:

- Domain und Range-Constraints funktionieren ausschließlich global: Beim Beispiel aus Abschnitt 2.3 lässt sich über das Property *hatMerkmal* nicht sagen, dass es bei Instanzen der Klasse *Wohnung* die Range *Wohnungsmerkmal* und bei Instanzen der Klasse *Mensch* die Range *MenschlichesMerkmal* haben soll. Dies schränkt die Wiederverwendbarkeit der Ontologie erheblich ein.
- Existenz- und Kardinalitäts-Constraints fehlen: Es lässt sich nicht sagen, dass alle Instanzen der Klasse *Wohnung* einen *Makler* haben. Auch die Aussage: *„Jede Wohnung, die exakt zwei Zimmer hat, ist eine Zweizimmerwohnung.“*, lässt sich nicht abbilden.
- Properties können nicht als transitiv, invers oder symmetrisch deklariert werden: Die Property *liegtIn* lässt sich nicht transitiv darstellen. Außerdem können die beiden Properties *liegtIn* und *beinhaltet* nicht zueinander invers markiert werden. Auch symmetrische Eigenschaften wie *hatKollege* lassen sich nicht deklarieren. (Dengel 2012: 127-128)

Möglichkeiten für die Modellierung dieser Ausdrücke bieten Beschreibungslogiken, eine Teilmenge der Prädikatenlogik erster Stufe. Die vom W3C im Jahr 2004 entwickelte Web Ontology Language (OWL) bildet diese Beschreibungslogiken in RDF ab. OWL ist die am weitesten verbreitete Ontologiesprache und wurde 2009 weiterentwickelt zu OWL 2. OWL 2 existiert in fünf Ausprägungen: OWL 2 DL, OWL

2 Full, OWL 2 EL, OWL 2 QL und OWL 2 RL. OWL 2 DL ist die gängigste Ausprägung. Das DL steht für *description logic* (Beschreibungslogik) und es gibt einige Restriktionen. So darf beispielsweise eine Klasse nicht die Instanz einer anderen Klasse sein. OWL 2 Full bildet alle Konzepte von RDFS ab und erweitert diese um OWL-Konzepte. Dies ist die umfassendste Ausprägung, die den Nachteil hat, unentscheidbar zu sein, d.h. es können keine Schlüsse gezogen werden. Die letzten drei Ausprägungen werden auch Profile genannt, da sie nur kleine Änderungen beinhalten, die Performance in Hinblick auf Schlussfolgern (*reasoning*) liefern sollen. OWL 2 EL ist optimiert für eine große Anzahl an Klassen und Properties in einer Ontologie. OWL 2 QL ist optimiert für Anwendungen, die großen Mengen an Instanzen von Klassen verarbeiten. OWL 2 RL simplifiziert den Schlussfolgerungs-Prozess und erlaubt regelbasierte Schlussfolgerungs-Algorithmen (*Reasoner*), büßt dafür aber an Expressivität ein. Folgend wird mit OWL immer OWL 2 DL gemeint sein, außer es wird explizit anders erwähnt. (W3C 2012)

OWL-Ontologien bestehen genau wie RDFS-Ontologien aus Klassen und Properties. Der Unterschied besteht in der Möglichkeit, diese in komplexe Beziehungen zueinander zu setzen. OWL erbt die meisten Konzepte von RDFS, überschreibt aber einige. Statt mit *rdfs:Class* werden Klassen durch *owl:Class* definiert. Statt *rdf:Property* werden die Subklassen *owl:ObjectProperty* für Beziehungen zwischen Instanzen, *owl:DatatypeProperty* für Beziehungen zwischen Instanzen und Werten (*String*, *Integer* o.Ä.) und *owl:AnnotationProperty* für Beschreibungen und Erklärungen, die auf den Reasoning-Prozess keinen Einfluss haben, eingeführt. (Uschold 2018: 44-48)

Um Domain- und Range-Constraints lokal, also nur für einzelne Klassen geltend, ausdrücken zu können, gibt es in OWL sogenannte Restriktionen (*owl:Restriction*), die einzelnen Klassen zugewiesen werden können. Das Zuweisen von Restriktionen ist auch bei Properties möglich. Diese gelten dann allerdings wieder global. Dem Beispiel vom Beginn des Abschnitts folgend, lassen sich für die beiden Klassen *Wohnung* und *Mensch* individuelle Range-Constraints für das Property *hatMerkmal* definieren.

```
1  @prefix ex: <http://example.com/> .
2
3  ex:Wohnung rdf:type          owl:Class ;
4             rdfs:subClassOf [ rdf:type       owl:Restriction ;
5                               owl:onProperty ex:hatMerkmal ;
6                               owl:onClass    ex:Wohnungsmerkmal ] .
7
8  ex:Mensch rdf:type          owl:Class ;
9            rdfs:subClassOf [ rdf:type       owl:Restriction ;
10                             owl:onProperty ex:hatMerkmal ;
11                             owl:onClass    ex:MenschlichesMerkmal ] .
```

Für Existenz- und Kardinalitäts-Constraints werden ebenfalls Restriktionen verwendet. Hier zuerst das Beispiel mit dem Makler und einem Existenz-Constraint.

Dies besagt, dass alle Wohnungen mindestens einen Makler haben, der eine Instanz der Klasse *Person* ist. Dies erlaubt auch mehreren Maklern einer Wohnung zugewiesen zu werden.

```
ex:Wohnung rdf:type          owl:Class ;
           rdfs:subClassOf [ rdf:type           owl:Restriction ;
                             owl:onProperty     ex:Makler ;
                             owl:someValuesFrom ex:Person ] .
```

Die Aussage: *„Jede Wohnung, die exakt zwei Zimmer hat, ist eine Zweizimmerwohnung.“*, lässt sich wie folgt darstellen:

```
ex:Zweizimmerwohnung rdf:type owl:Class ;
        owl:equivalentClass [ owl:intersectionOf(
                                 rdf:type ex:Wohnung;
                                [rdf:type owl:Restriction ;
                                    owl:onProperty ex:hatZimmer ;
                                    owl:hasValue "2"^^xsd:int
                                ]
                              )
                            ] .
```

Der Ausdruck *owl:equivalentClass* bedeutet, dass die folgende Bedingung eine abstrakte Klasse darstellt und alle Instanzen, die die Bedingung erfüllen, äquivalent zu Instanzen der Klasse *Zweizimmerwohnung* sind. Mit *owl:intersectionOf* wird die Schnittmenge aller Instanzen von Klassen innerhalb des Ausdrucks gebildet. In diesem Fall gehören alle Instanzen der Klasse *Wohnung* und alle Instanzen einer abstrakten Klasse, die auf dem Property *hatZimmer* genau den Wert *2* haben, zusammen.

Transitivität, Symmetrie und Inverse von Properties lassen sich wie folgt darstellen:

```
ex:liegtIn rdf:type owl:ObjectProperty ,
                    owl:TransitiveProperty .

ex:beinhaltet rdf:type      owl:ObjectProperty ;
              owl:inverseOf ex:liegtIn .

ex:beruehrt rdf:type owl:ObjectProperty ,
                     owl:SymmetricProperty .
```

Diese Bezeichnungen lassen Rückschlüsse auf Instanzen zu, die mit diesen Properties verbunden werden.

```
ex:Hamburg     ex:liegtIn  ex:Deutschland ;
               ex:beruehrt ex:Niedersachsen .
```

```
3
4
5 ex:Deutschland ex:liegtIn  ex:Europa .
```

Daraus kann ein Reasoner folgendes schließen:

```
1 ex:Hamburg       ex:liegtIn    ex:Europa ;  # Transitivität
2
3 ex:Deutschland   ex:beinhaltet ex:Hamburg ; # Inverse
4
5 ex:Niedersachsen ex:beruehrt   ex:Hamburg ; # Symmetrie
```

2.5 SPARQL

SPARQL ist eine vom W3C entwickelte Abfragesprache für RDF-Graphen und stellt den de facto Standard für das Semantic Web dar. Sie basiert auf dem sogenannten *graph pattern matching*, also dem Abgleich eines Abfrage-Graphen mit dem RDF-Graphen, der befragt wird. Die Syntax von SPARQL orientiert sich zum einen an einigen Schlüsselbegriffen, die der Abfragesprache für relationale Datenbanken SQL entlehnt sind. Zum anderen nutzt SPARQL die Turtle-Syntax, die in Abschnitt 2.3 bereits näher erläutert wurde.

Ein einfacher SPARQL SELECT-Query besteht aus zwei Teilen, einem oder mehreren Frageworten und einem Abfragemuster. Ein Query, der die Frage nach der Stadt, in der *Apartment1* liegt, beantwortet, könnte so aussehen:

```
1 SELECT ?stadt
2 WHERE
3 {
4     ex:Apartment1 ex:liegtIn ?stadt .
5 }
```

Frageworte werden in SPARQL mit einem vorgestellten Fragezeichen versehen. Das Abfragemuster, das mit dem Schlüsselwort WHERE beginnt, folgt dem bekannten Tripel-Muster *Subjekt*, *Prädikat* und *Objekt*. Resultate werden, ähnlich wie bei SQL, in Tabellen zurückgegeben. Dabei stellen die Frageworte die Spalten dar und die gefundenen Resultate die Zeilen. Frageworte lass sich auch wie Variablen verwenden, sogar ohne sie als Resultate auszugeben.

```
1 SELECT ?merkmal
2 WHERE
3 {
4     ex:Hamburg ex:beinhaltet ?wohnung .
5     ?wohnung   ex:hatMerkmal ?merkmal .
6 }
```

Hier wäre zu erwarten, sämtliche Wohnungsmerkmale aller Wohnungen in Hamburg zu erhalten. Wirklich sicher wäre dies aber nur, wenn *ex:beinhaltet* die *rdfs:range Wohnung* hat. Zur Verfeinerung des Queries wird eine weitere Bedingung eingeführt, außerdem wird dieses Mal auch das Fragewort *?wohnung* als Resultat zurückgegeben:

```
SELECT ?wohnung ?merkmal
WHERE
{
    ex:Hamburg ex:beinhaltet ?wohnung .
    ?wohnung   rdf:type      ex:Wohnung ;
               ex:hatMerkmal ?merkmal .
}
```

Das Resultat des Queries würde dann so aussehen:

?wohnung	?merkmal
ex:Apartment1	ex:Einbaukueche
ex:Apartment1	ex:Duschwanne
...	...

Tabelle 2.1: SPARQL-Resultat

Der Query liefert also alle Wohnungen, die in Hamburg liegen und die jeweils zugehörigen Wohnungsmerkmale. Die ersten beiden Tripel des WHERE-Blocks ließen sich auch austauschen. Welche Reihenfolge mehr Sinn, ergibt hängt davon ab, ob es mehr Wohnungen im RDF-Graphen gibt oder mehr Dinge die Hamburg *beinhaltet*.

Eine weitere Möglichkeit Resultate zu verfeinern, stellt das FILTER-Schlüsselwort dar. Mit diesem können Resultate, die eigentlich die WHERE-Bedingung erfüllen, ausgeschlossen und nur die Ergebnisse zurückgegeben werden, welche die zusätzliche FILTER-Bedingung erfüllen. FILTER führt dabei einen Boolean-Test durch, liefert also immer entweder *true* oder *false*. (Allemang et al. 2020: 119ff.)

Dieser Query liefert alle Wohnungen der Wissensbasis, die mehr als zwei Zimmer haben.

```
SELECT ?wohnung ?zimmerAnzahl
WHERE
{
    ?wohnung rdf:type      ex:Wohnung ;
             ex:hatZimmer ?zimmerAnzahl .
    FILTER ( ?zimmerAnzahl > "2"^^xsd:int )
}
```

2.6 Shapes Constraint Language (SHACL)

Alle bisherigen Semantic Web Technologien arbeiten mit der *Open-World-Assumption* (OWA) (siehe Abschnitt 2.4). Für viele Anforderungen ist dies jedoch nicht die passende Herangehensweise. Sollen beispielsweise die Eingaben eines Nutzers in einem Webformular überprüft oder ein Datensatz auf ein spezifiziertes Schema validiert werden, wird die *Closed-World-Assumption* (CWA) benötigt. Um Linked Data in Form von RDF-Graphen nicht extra in ein CWA-konformes Datenmodell übertragen zu müssen, hat das W3C 2017 die Shapes Constraint Language (SHACL) vorgestellt. Diese Sprache baut auf dem RDF auf und nutzt dessen Syntax. Ein RDF-Graph der SHACL nutzt wird auch *Shapes-Graph* genannt.

Konträr zu den meisten Anwendungen im Semantic Web sollen mit SHACL Daten nicht interpretiert oder neues Wissen abgeleitet werden, sondern Erwartungen an Daten gerichtet werden. Um das Beispiel mit der Nutzereingabe in einem Webformular noch genauer auszuführen wäre es denkbar, dass in einem Eingabefeld für eine deutsche Postleitzahl nur Zahlen erlaubt sein sollen, die genau fünf Ziffern haben. Dies ist mit SHACL und sogenannten *regular expressions* leicht umzusetzen, ohne das Datenmodell wechseln zu müssen. Auch Pflichtfelder eines Formulars lassen sich so leicht festlegen.

Die Erwartungen an einen Datensatz, die mittels eines Shapes-Graphen beschrieben werden, lassen sich auch als Spezifikation der Daten verstehen. Es wird ausgedrückt welche Form (*shape*) die Daten haben sollen und so eine Blaupause geliefert, die hilft zwischen mehreren Parteien Daten zu integrieren. Es ist sofort klar, welche Daten die Erwartungen erfüllen, also gültig sind und welche nicht. (Allemang et al. 2020: 193ff.)

Hier ein kleiner Shapes-Graph, der das Beispiel mit der Postleitzahl umsetzt und zusätzlich die Erwartungen an ein Feld für eine Stadt spezifiziert. Die Präfixe wurden hierbei weggelassen.

```
1  shape:Address rdf:type        sh:NodeShape ;
2                sh:targetClass  ex:Address ;
3                sh:property.    shape:zipCode ,
4                                shape:city .
5
6  shape:zipCode rdf:type     sh:PropertyShape ;
7                sh:path      ex:zipCode ;
8                sh:minCount  1 ;
9                sh:maxCount  1 ;
10               sh:datatype  xsd:int ;
11               sh:pattern   "^[0-9][0-9][0-9][0-9][0-9]$" .
12
13 shape:city rdf:type     sh:PropertyShape ;
14            sh:path      ex:city ;
```

```
15              sh:minCount 1 ;
16              sh:maxCount 1 ;
17              sh:datatype xsd:string .
```

Hier wird einer Klasse *ex:Address* mittels *sh:targetClass* eine *sh:NodeShape* zugewiesen. Für eine NodeShape lassen sich mit dem Prädikat *sh:property* beliebig viele Bedingungen auf explizite Properties ausdrücken. Die Bedingungen haben dabei die Form einer Property-Shape und werden mit *sh:path* dem gewünschten Property zugeordnet. Die Art der Bedingungen, die sich ausdrücken lassen, sind dabei sehr vielfältig und reichen von Kardinalitäten (*sh:minCount*, *sh:maxCount*) über den Datentyp (*sh:datatype*) bis zu regular expressions (*sh:pattern*).

Ein SHACL-Prozessor nimmt diesen Shapes-Graph zusammen mit einem Datengraph entgegen und prüft den Datengraph auf das Einhalten der Bedingungen. Anschließend liefert er einen *Validation Report* zurück, der ebenfalls in RDF gehalten ist. Dieser lässt sich also mit allen Semantic Web Technologien, wie beispielsweise SPARQL, maschinenlesbar weiter verarbeiten.

```
1  ex:Address1 rdf:type.  ex:Address ;
2              ex:zipCode "21129"^^xsd:int ;
3              ex:city.   "Hamburg" .
4
5  ex:Address2 rdf:type   ex:Address ;
6              ex:zipCode "121129"^^xsd:int .
7
8  ex:Address3 rdf:type   ex1:Address ;
9              ex:zipCode "Postleitzahl" .
```

Der Validation Report zu diesen Daten sieht so aus:

```
1  [
2          a sh:ValidationResult ;
3          sh:resultSeverity sh:Violation ;
4          sh:sourceConstraintComponent sh:PatternConstraintComponent ;
5          sh:sourceShape ex:zipCode ;
6          sh:focusNode ex:Address2 ;
7          sh:value "121129"^^xsd:int ;
8          sh:resultPath ex:zipCode ;
9          sh:resultMessage "Value does not match pattern
   ↪  "[0-9][0-9][0-9][0-9][0-9]"" ;
10 ] .
11 [
12         a sh:ValidationResult ;
13         sh:resultSeverity sh:Violation ;
```

```
14          sh:sourceConstraintComponent sh:MinCountConstraintComponent ;
15          sh:sourceShape ex:city ;
16          sh:focusNode ex:Address2 ;
17          sh:resultPath ex:city ;
18          sh:resultMessage "Less than 1 values" ;
19  ] .
```

Der SHACL-Prozessor hat bei den Eingabedaten insgesamt zwei Verstöße gegen die gewünschten Bedingungen festgestellt. Einmal fehlt bei der *Address2* die Stadt (*sh:minCount 1*) und die Postleitzahl folgt nicht dem gewünschten Muster von genau fünf Ziffern zwischen null und neun. Bei *Address3* liegt kein Verstoß vor, obwohl auch dort die Angabe zur Stadt fehlt und die Postleitzahl falsch gespeichert wurde. Zusätzlich stimmt nicht einmal der Datentyp überein. Der Grund hierfür liegt in der Klassenzuweisung der Bedingungen (*sh:targetClass*). *Address3* gehört zur Klasse *ex1:Address* und nicht zu *ex:Address*. Dann ist es auch irrelevant, ob Properties zum Einsatz kommen, für die Bedingungen im Shapes-Graphen vorliegen.

3 Konzept

In diesem Kapitel soll zunächst kurz geschildert werden, mit welchen Daten gearbeitet wird und welche Probleme dabei vorliegen. Anschließend wird erläutert, wie dieser Datensatz verändert werden soll, um neuen Zwecken gerecht zu werden. Schließlich werden die Anforderungen für eine Umsetzung des Konzeptes dargelegt.

3.1 Beschreibung des Datensatzes

Der Ausgangsdatensatz gehört einem PropTech-Unternehmen, das deutschlandweit agiert und Wohnungen und Häuser im Auftrag der Besitzer oder Hausverwaltungen vermietet oder verkauft. Der Datensatz enthält und beschreibt etwa 2000 Immobilien, die im Laufe der letzen beiden Jahre angelegt wurden. Das Schema, welches den Daten zugrunde liegt, ist lose an den OpenImmo-Standard[1] angelehnt, um den Aufwand für den Datenaustausch mit anderen Immobilien-Services zu minimieren. Bekannte Unternehmen wie die Immobilienscout GmbH, immowelt AG und eBay Kleinanzeigen orientieren sich ebenfalls an diesem Standard.

Der Datensatz stammt aus einer MongoDB-Umgebung. Dies ist ein dokumentbasiertes NoSQL-Datenbankmanagementsystem, welches Sammlungen (*collections*) von JSON-Dokumenten enthält. Diese Sammlungen sind vergleichbar mit einer Tabelle einer relationalen Datenbank. Der Unterschied besteht in dem Fehlen eines festen Schemas für eine Sammlung in MongoDB.

Das PropTech-Unternehmen besteht seit 2015 und hat seitdem nie das Datenbankmanagementsystem gewechselt. Dementsprechend sind auch im Laufe der Jahre einige Datenfelder hinzugekommen, während andere ab einem gewissen Zeitpunkt nicht mehr genutzt wurden. Der Aufbau von MongoDB lässt diese Dokumente mit abweichenden Schemata ohne Widerspruch in der selben Sammlung zu. Im Zusammenhang mit JSON-Dokumenten wird deshalb auch von semistrukturierten Daten gesprochen.

In einer Analyse des aktuellen Stands des Systems wurden die wichtigsten Felder identifiziert. Für diese Arbeit wurden die Immobiliendaten nur mit den relevanten Feldern aus der Datenbank exportiert. Der vorgestellte Ansatz zum Transfer semistrukturierter Daten benötigt nicht zwingend vorher bereinigte Daten. Dieser Schritt wurde nur vorgenommen, um bei der Arbeit mit dem Datensatz Übersichtlichkeit zu gewinnen.

[1] http://www.openimmo.de/

In den Daten befinden sich auch datenschutzrelevante Inhalte wie Name und Telefonnummer von Hausverwaltern oder Vormietern. Aus diesem Grund wird der Datensatz und das transferierte Abbild auch im Zuge dieser Arbeit nicht veröffentlicht. Bei gezeigten Auszügen wurden persönliche Daten abgeändert oder weggelassen.

Ein JSON-Dokument einer Immobilie wird zunächst über einen einzigartigen Identifikatoren (*publicId*) identifiziert. Danach folgt die Adresse des Objektes. Dann gibt es einige Felder, die die Immobilie genauer kategorisieren. Handelt es sich um ein Haus oder eine Wohnung? (*type*) Soll die Immobilie verkauft oder vermietet werden? (*disposalType*) Soll die Immobilie befristet oder unbefristet vermietet werden? (*usageType*) Um was für eine Art von Wohnung handelt es sich genau? (*apartmentType*) Für jedes dieser Felder gibt es vordefinierte Werte, aus denen ausgewählt werden kann, wenn eine Immobilie über die Website des PropTech-Unternehmens angelegt wird. Es folgen Angaben zu Austattung, Anzahl der Zimmer und Größe des Objektes (*allocation*). Dann werden die Kosten, seien es Miete und Nebenkosten oder Kaufpreis, angegeben (*costs*). Bei den meisten Objekten gibt es Angaben zum Energieausweis und der Heizung (*energyPass*). Auch hier gibt es vordefinierte Werte, welche die Felder füllen können. Bei vielen Immobilien gibt es eine Angabe zum allgemeinen Zustand des Objektes (*condition*), bei möblierten Objekten auch noch eine Angabe zum Zustand der Inneneinrichtung (*equipmentCondition*). Den Abschluss eines JSON-Dokuments bilden Angaben zu Mieterrichtlinien bezüglich Haustieren, Wohngemeinschaften und Ähnlichem (*tenantPreferences*). Bei etwas weniger als der Hälfte der Immobilien gibt es noch mehr oder weniger detaillierte Daten zum Ansprechpartner, sei es der Vormieter, ein Hausmeister oder eine Hausverwaltung (*caretakerContact*). Ein vollständiges Beispiel für ein typisches Objekt findet sich im Anhang dieser Arbeit (A.1).

Insgesamt besteht der Ausgangsdatensatz aus Daten zu 2025 Immobilien im JSON-Format. Die Daten liegen in einer einzelnen Datei vor, die ein JSON-Array enthält und rund 158.000 Zeilen umfasst.

3.2 Idee

Da sich das Unternehmen in einem gewissen Rahmen neu ausrichten möchte, ist ein Faktor deutlich wichtiger geworden, der bisher nicht im Fokus stand: *Search Engine Optimization* (SEO). Bisher wurden Miet- und Kaufinteressenten für die Objekte über die großen Immobilienplattformen aggregiert. Dafür werden die Objekte samt Beschreibungstexten über OpenImmo-Schnittstellen automatisiert auf viele verschiedene Plattformen gleichzeitig hochgeladen. Im Anschluss fließen die Interessenten ebenfalls automatisiert zurück auf die Plattform des PropTech-Unternehmens. Von dort aus werden dann Besichtigungen organisiert und der gesamte Vermietungsprozess begleitet und abgebildet. Der Vorteil dieser Vorgehensweise liegt in dem Bekanntheitsgrad der großen Immobilienplattformen. Interessenten suchen intuitiv auf diesen Webseiten und Ergebnisse von Suchmaschinen werden dementsprechend hoch

bewertet.

Das bisherige Vorgehen wird beibehalten, allerdings sollen die Immobilien der eigenen Plattform ebenfalls über Suchmaschinen leichter auffindbar sein, um Interessenten auch auf diesem Wege anzusprechen. Gängige SEO-Methoden werden aus diesem Grund bereits angewandt. Zur Unterstützung der Maßnahmen sollen die Daten selbst maschinenlesbar und mit einem etablierten Vokabular ergänzt werden. Google gibt Webseiten-Betreibern einige Hinweise, wie sie ihre Suchergebnisse verbessern können. In diesem Kontext wird von *strukturierten Daten* gesprochen, gemeint ist Linked Data. Die Begriffe können in diesem Fall synonym verwendet werden. Google empfiehlt die Verwendung der weit verbreiteten *schema.org*-Ontologie[2] als Vokabular und JSON-LD als Format für Linked Data auf Websiten. Die in dieser Arbeit verwendete Turtle-Serialisierung lässt sich automatisiert zur JSON-LD-Serialisierung konvertieren. Um JSON-LD-Daten in eine Webseite einzubinden, sodass Suchmaschinen sie lesen können, ist es ausreichend die Daten in ein *<script>*-Tag im Seitentext einzufügen. Dies kann auch dynamisch erfolgen und muss nicht statisch festgeschrieben werden. (Google 2020)

Für jede Immobilie aus dem Datensatz existiert eine Unterseite auf der Plattform des PropTech-Unternehmens, in die die JSON-LD-Daten eingebunden werden können. Das tatsächliche Einbetten in die Webseite liegt außerhalb des Rahmens dieser Arbeit. Es soll ein Weg gezeigt werden, die vorhandenen Daten in ein Linked Data-Format zu überführen.

Zu diesem Zweck soll, basierend auf den Ausgangsdaten, eine Domänen-Ontologie entwickelt werden, welche es erlaubt die Daten in einer Linked Data-Umgebung adäquat zu beschreiben. Gleichzeitig wird ein Schema geliefert, das den Daten zugrunde liegt. Dieses Schema soll dann mit einigen Abwandlungen zur Validierung der Daten verwendet werden. Basierend auf der Ontologie soll ein Prozess gezeigt werden, die Bestandsdaten der Immobilien mittels Datenmapping vom JSON-Format in RDF-Graphen in Turtle-Serialisierung zu transferieren.

3.3 Anforderungen

Der Ansatz in dieser Arbeit soll sich dafür eignen, erweitert zu werden, sei es um zusätzliche Begriffe aus einem neuen Vokabular oder um Änderungen in den Ausgangsdaten.

Die entwickelte Domänen-Ontologie, die als Datenmodell fungiert, muss die Ausgangsdaten ausreichend detailliert beschreiben, um mindestens die gleiche Datenqualität und Aussagekraft zu gewährleisten wie die Ausgangsdaten. Ziel ist eine höhere Aussagekraft durch die Verwendung etablierter Vokabulare und die damit einhergehende Vernetzung zu gleichen Konzepten.

Die Ontologie soll eine feste und stabile URI haben und *dereferenzierbar* sein. Das bedeutet, dass die URIs tatsächlich die Ressourcen zurückliefern, für die sie stehen.

[2] https://schema.org/

Auch wenn die Ressourcen auf weitere URIs verweisen, sollen diese auf die erwarteten Ressourcen zeigen. Weiterhin soll eine Anfrage an die Ontologie, basierend auf dem Anfragenden, Ressourcen im korrekten Format zurückgeben (*content negotiation*). So soll die Anfrage eines Programms beispielsweise Ontologieressourcen in RDF/XML-Serialisierung zurückgeben, während die Anfrage eines Webbrowsers an die gleiche URI eine HTML-Repäsentation der Ontologie in Form einer Dokumentation liefern soll.

Die verwendeten Technologien sollen einen gewissen Grad der Etablierung aufweisen, um im Zweifel Unterstützung von Fachkundigen in Anspruch nehmen zu können. Außerdem soll sich der präsentierte Lösungsvorschlag in Produktionsprozesse und -umgebungen integrieren lassen, ohne eigene Schnittstellen entwickeln zu müssen.

4 Implementation

Das Kapitel beschäftigt sich mit der Umsetzung des Konzepts aus dem vorherigen Kapitel unter Berücksichtigung der formulierten Anforderungen.

4.1 Verwendete Technologien

In diesem Abschnitt werden die eingesetzten Werkzeuge und Services beschrieben, die zur Realisierung des Konzepts verwendet wurden. Falls erforderlich, wird darauf eingegangen welche Funktionen genau genutzt wurden.

4.1.1 Protégé

Protégé ist ein Ontologie-Editor, der vom *Stanford Center for Biomedical Informatics Research* (BMIR) der Stanford Universität entwickelt wurde. Er steht unter einer Open-Source-Lizenz. Protégé gilt als das am weitesten verbreitete Werkzeug im Umgang mit und für das Erstellen von Ontologien. Es gibt einen vollumfänglichen Support für OWL 2, ein hochanpassbares User-Interface, eingebaute Reasoning-Engines und Plugin-Unterstützung. Protégé nutzt für die Arbeit mit Ontologien eine grafische Benutzeroberfläche und für Konstruktionen wie Klassenrestriktionen die Manchester OWL Syntax, die im nächsten Abschnitt noch genauer erklärt wird. (Protégé 2020)

4.1.2 OnToology

OnToology ist ein Webservice rund um den Ontologie-Entwicklungsprozess. Es vereint Werkzeuge zur Dokumentation von Ontologien (Widoco[1]), Diagramme von RDF-Graphen (AR2DTool[2]) und übernimmt die Publizierung von Ontologien unter permanenten w3id-URIs[3]. OnToology greift dabei auf Github Repositories zurück, die mit dem Service verbunden werden. Die generierte Dokumentation und die Dateien für die *content negotiation* werden dann in das entsprechende Nutzer-Repository geschrieben. Über Github Pages wird so eine statische Website zur Verfügung gestellt, die basierend auf der Anfrage die Ressourcen in der geforderten Serialisierung zurück liefert. OnToology überwacht außerdem die verbundenen Repositories und öffnet *Pull*

[1] https://github.com/dgarijo/Widoco
[2] https://github.com/idafensp/ar2dtool
[3] https://w3id.org/

Requests mit angepassten Dokumentationen und Diagrammen, wenn Änderungen an den Ontologien festgestellt werden. (OnToology 2020)

4.1.3 RML & YARRRML

Die RDF Mapping Language (RML) ist eine Verallgemeinerung der *RDB to RDF Mapping Language* (R2RML[4]) des W3C. R2RML ist eine Sprache, die auf RDF aufbaut und für das Mapping von Daten aus relationalen Datenbanken zu RDF-Graphen verwendet werden kann. RML erweitert das Konzept von R2RML auf verschiedenste Datenformate wie zum Beispiel CSV, XML und JSON. Für das Mapping werden deklarative Regeln aufgestellt, welche dann von einem geeigneten Prozessor in Kombination mit den Ausgangsdaten abgearbeitet werden. Das Resultat sind RDF-Graphen, die der Struktur der RML-Datei entsprechen. Da RML auf dem RDF aufbaut, sind RML-Dateien auch immer selbst RDF-Graphen.

Der Ausgangspunkt einer RML-Datei ist eine sogenannte *rml:source*, die entweder auf eine lokale Datei mit Ausgangsdaten oder auf eine Remote-Datenquelle wie beispielsweise eine relationale Datenbank zeigt. Zusätzlich wird ein *rml:iterator* formuliert, der angibt wie über die Eingangsdaten iteriert werden soll. In dieser Arbeit wird auf die JSON-Datei mittels JSONPath[5] zugegriffen. (RML 2020)

Mit einem RML-Prozessor wie dem RMLMapper[6] werden die deklarativen RML-Regeln umgesetzt.

YARRRML ist ein Repräsentationsdialekt von RML, der auf der YAML-Syntax aufbaut. Diese Syntax wurde entwickelt, da sie gut von Menschen lesbar ist. RML-Dateien werden schnell sehr unübersichtlich und YARRRML wirkt dem mit einer sehr kompakten und leicht verständlichen Ausdrucksweise entgegen. Um von YARRRML zu RML zu kommen, gibt es einen YARRRML-Parser[7], der die Regeln übersetzt. (YARRRML 2020)

Entwickelt wurden diese Sprachen und Werkzeuge von IDLab, einer Forschungsgruppe der Universitäten von Gent und Antwerpen in Zusammenarbeit mit dem Interuniversity Microelectronics Centre (IMEC).

4.1.4 Ontotext GraphDB

Ontotext GraphDB ist eine Graphdatenbank mit RDF- und SPARQL-Unterstützung, die auf dem RDF4J-Framework[8] aufbaut. GraphDB erlaubt das Definieren eigener Reasoning-Regelsätze und liefert gleichzeitig vorgefertigte Regelsätze verschiedener OWL-Ausprägungen und -Profile. Um importierte Daten zu validieren, existiert die

[4] https://www.w3.org/TR/r2rml/
[5] https://goessner.net/articles/JsonPath/
[6] https://github.com/RMLio/rmlmapper-java
[7] https://github.com/RMLio/yarrrml-parser
[8] https://rdf4j.org/

Möglichkeit für ein Repository SHACL-Shapes zu definieren. Gegen diese werden die Daten überprüft, wenn sie in die Datenbank geladen werden.

Die Datenbank lässt sich lokal betreiben und ist dann im internen Netzwerk erreichbar. Alternativ ist das Deployen auf einem Server möglich.

Die GraphDB Workbench ist eine grafische, webbasierte Administrationsoberfläche mit der sich die einzelnen Repositories der GraphDB-Instanz verwalten lassen. Über die Workbench lassen sich neue RDF-Daten importieren und bestehende Daten in sämtlichen gängigen RDF-Serialisierungen exportieren. Außerdem ermöglicht die Oberfläche das Ausführen von SPARQL-Queries und das Durchstöbern der Repositories. Auch einige Visualisierungsmöglichkeiten der gespeicherten Daten werden angeboten.

Die Software steht in drei Varianten zur Verfügung: Free, Standard und Enterprise. Die freie Variante bietet für lokale Projekte alle nötigen Funktionen und ist nicht auf einen Testzeitraum beschränkt. Sollte der Wunsch bestehen auf eine der anderen Versionen aufzurüsten, findet sich in der Dokumentation eine unkomplizierte Anleitung für den Umzug. (Ontotext 2020)

4.2 Entwicklung der Ontologie

Der Entwicklungsprozess für die Ontologie ist stark vom Ausgangsdatensatz und dem gewählten Mapping-Ansatz beeinflusst. Für alle Datenfelder, die festgelegte Werte aufweisen, wurden Klassen erstellt. Zusätzlich erhalten einige dieser Felder deskriptivere Namen. In Tabelle 4.1 sind die Felder und entsprechenden Klassen aufgelistet.

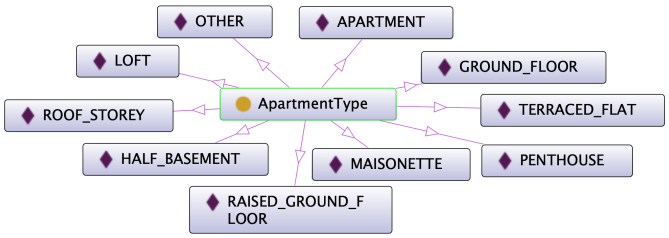

Abbildung 4.1: Instanzen der Klasse *ApartmentType*

Die festgelegten Werte sind als Instanzen dieser Klassen realisiert (siehe Abbildung 4.1). Ein Ausnahme bilden die Datenfelder unter *tenantPreferences*. Hier gibt es für die Felder *cats*, *dogs*, *petAllowed*, *smoking*, *livingCommunity* und *commercialUse* jeweils die gleichen festen Werte. Um Wiederholungen zu vermeiden, gibt es in der Ontologie für diese Felder nur eine Klasse namens *TenantGuidelines*, die als Instanzen die fixen Werte enthält.

Für die weitere Modellierung gibt es noch vier weitere Klassen. *Address* vereint Angaben zur Adresse einer Immobilie, wie Straße, Hausnummer und Postleitzahl. *EnergyPass* ist eine Sammelklasse für alle Daten bezüglich der Heizung, des Energieträgers

Datenfeld	Klasse
type	RealEstateType
usageType	UsageType
disposalType	MarketingType
apartmentType	ApartmentType
houseType	HouseType
caretakerContact.title	ContactPersonType
energyPass.energyPassType	EnergyPassType
energyPass.energyPresenType	EnergyPassPresenType
energyPass.energyEfficiencyCategory	EnergyEfficiencyClass
energyPass.energyCarrier	EnergyCarrier
energyPass.heatingType	Heating
conditions.condition	Condition
conditions.equipmentCondition	EquipmentCondition
allocation.carspaceType	CarSpace

Tabelle 4.1: Datenfelder mit festen Werten und den entsprechenden Ontologieklassen

und des Energieausweises. *ContactPerson* umfasst Angaben zum Ansprechpartner für eine Immobilie, wie Mailadresse und Telefonnummer. Die wichtigste Klasse ist *RealEstate*. Diese bildet den Grundbaustein für die Repräsentation einer Immobilie, auf ihr sind auch die meisten Klassenrestriktionen definiert.

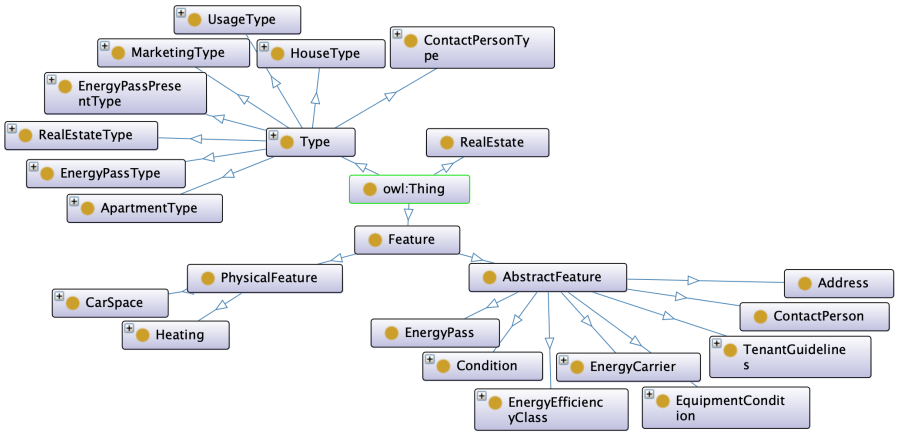

Abbildung 4.2: Klassenhierarchie der Ontologie

Die Klassenhierarchie unterteilt sich in die drei Superklassen *Feature*, *Type*, und *RealEstate* (siehe Abbildung 4.2). Mit *Feature* sind Eigenschaften im weitesten Sinne gemeint. Deshalb unterteilt sich diese Klasse noch in die beiden spezifischeren Klassen *PhysicalFeature*, für Eigenschaften, die einen physischen Körper haben, und *AbstractFeature*, für Eigenschaften, die nicht wirklich greifbar sind. *Type* kategorisiert

generellere Aussagen über die Immobilie. Alle Subklassen von *Type* haben ausschließlich festgelegte Werte in Form von Instanzen.

Die Properties teilen sich auf in Object Properties und Data Properties. Mit Object Properties sind alle Prädikate gemeint, die als Objekt Instanzen einer Klasse haben. Für jede Klasse mit fixen Werten gibt es ein entsprechendes Object Property. Zusätzlich noch drei für *Address*, *EnergyPass* und *ContactPerson*, um Instanzen dieser Sammelklassen mit Instanzen von *RealEstate* verbinden zu können.

Data Properties umfassen alle Prädikate, die als Objekt getypte oder ungetypte Literale haben. Mithilfe dieser werden alle Datenfelder abbgebildet, die keine festen Werte haben, zum Beispiel *numberOfRooms*.

Die Ontologie enthält keine Domain- oder Range-Constraints, damit das Vokabular auch von anderen Anwendungen genutzt werden kann, ohne auf die Klassen dieser Ontologie beschränkt zu sein. Stattdessen gibt es Klassenrestriktionen auf den Klassen *Address*, *EnergyPass*, *ContactPerson* und *RealEstate*. In der Abbildung 4.3 sind die Klassenrestriktionen von *ContactPerson* in Manchester OWL Syntax aufgeführt. Abbildung 4.4 zeigt die Übersetzung dieser Restriktionen in Turtle-Serialisierung.

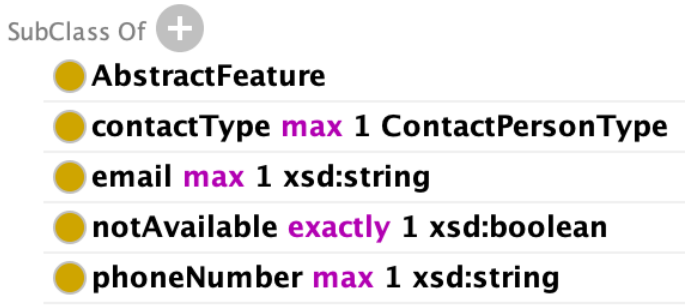

Abbildung 4.3: Klassenrestriktionen von *Address* in Manchester OWL Syntax

Einige Instanzen wurden exemplarisch über das Property *owl:sameAs* mit Instanzen einer verbreiteten Ontologie verbunden. *reo:GAS* zeigt beispielsweise über dieses Property auf *dbpedia:Gasheizung*. DBpedia[9] ist ein Projekt, welches Konzepte von Wikipedia in Linked Open Data transferiert und im Semantic Web großen Anklang findet. Anwendungen können Instanzen, die mit DBpedia-Konzepten verbunden sind, weiterverfolgen und so ähnliche und verwandte Konzepte entdecken, um die Bedeutung der ursprünglichen Instanzen zu verstehen.

Sämtliche Ressourcen, seien es Klassen, Properties oder Instanzen, haben via *rdfs:label* einen deutschen Bezeichner erhalten. In der generierten Dokumentation werden diese Bezeichner verwendet.

Die Ontologie wurde mittels OnToology unter der URI `https://w3id.org/def/reo` und mit dem Namen *Real Estate Ontology* publiziert.

[9] `https://wiki.dbpedia.org/`

```
reo:ContactPerson rdf:type owl:Class ;
                  rdfs:subClassOf reo:AbstractFeature ,
                                  [ rdf:type owl:Restriction ;
                                    owl:onProperty reo:contactType ;
                                    owl:maxQualifiedCardinality
    ↪  "1"^^xsd:nonNegativeInteger ;
                                    owl:onClass reo:ContactPersonType
                                  ] ,
                                  [ rdf:type owl:Restriction ;
                                    owl:onProperty reo:notAvailable ;
                                    owl:qualifiedCardinality
    ↪  "1"^^xsd:nonNegativeInteger ;
                                    owl:onDataRange xsd:boolean
                                  ] ,
                                  [ rdf:type owl:Restriction ;
                                    owl:onProperty reo:email ;
                                    owl:maxQualifiedCardinality
    ↪  "1"^^xsd:nonNegativeInteger ;
                                    owl:onDataRange xsd:string
                                  ] ,
                                  [ rdf:type owl:Restriction ;
                                    owl:onProperty reo:phoneNumber ;
                                    owl:maxQualifiedCardinality
    ↪  "1"^^xsd:nonNegativeInteger ;
                                    owl:onDataRange xsd:string
                                  ] .
```

Abbildung 4.4: Klassenrestriktionen von *Address* in Turtle-Serialisierung

4.3 Erstellen der Mapping-Regeln

In Abschnitt 4.1.3 wurde bereits erwähnt, dass für das Erstellen der Mapping-Regeln der Repräsentationsdialekt YARRRML zum Einsatz kommt. Für diesen Dialekt existiert eine Webanwendung namens *Matey*[10]. Diese Anwendung soll den Entwicklungsprozess der deklarativen Regeln unterstützen, indem sie einige Beispiel-Mappings zur Verfügung stellt, welche zu Testzwecken leicht abwandelbar sind. Außerdem lassen sich eigene Mappings und Testdaten einfügen. Per Knopfdruck werden die Mappings in RML übersetzt und zusätzlich die resultierenden RDF-Daten angezeigt. In Abbildung 4.5 ist die Oberfläche der Anwendung zu sehen.

Den Einstieg in ein YARRRML-Mapping bilden die Präfixe (siehe Material A.2). Im Fall dieser Arbeit umfassen diese *reo* (eigene Ontologie), *ex* (Platzhalternamensraum, der sich später leicht austauschen lässt), *xsd* (Datentypen) und *schema* (siehe Abschnitt 3.2).

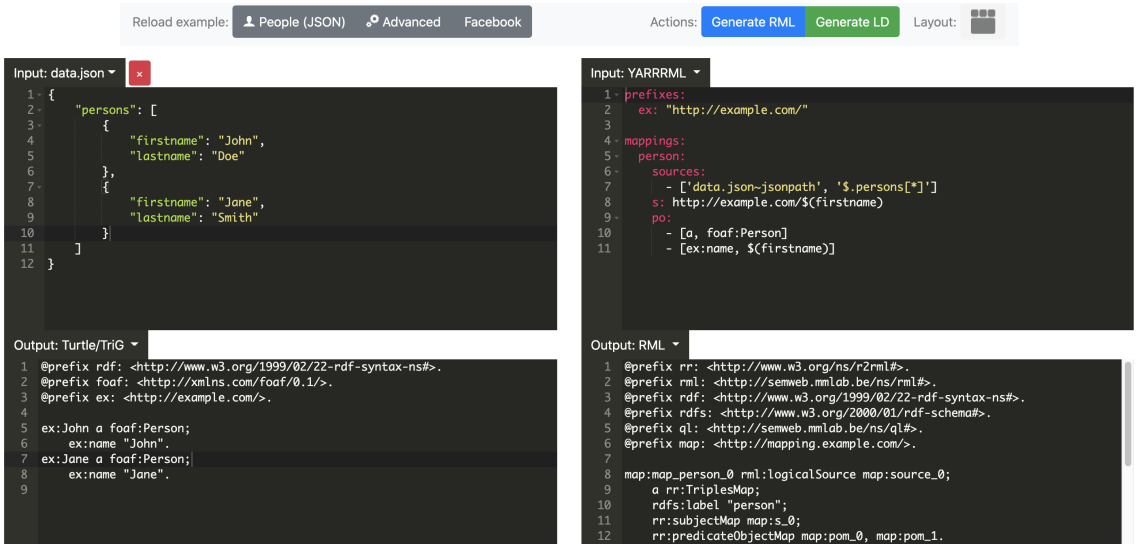

Abbildung 4.5: Weboberfläche von *Matey*

Im nächsten Teil folgen die Angaben zu den Eingangsdaten über das Schlüsselwort *sources*. An dieser Stelle lassen sich theoretisch mehrere Datenquellen vereinen. Jede Datenquelle bekommt einen Bezeichner, in diesem Fall *realEstateList*. Mit *access* wird der Dateinname und -pfad angegeben. Die Kombination aus *referenceFormulation* und *iterator* spezifiziert den Einstieg in die Datei und wie durch die Daten iteriert wird. Das Symbol *$* steht für die höchste Ebene eines JSON-Dokuments, dann wird mit *realEstate[*]* nach einem Array dieses Namens gesucht. Das *-Symbol bezeichnet eine sogenannte Wildcard und umfasst somit alle Elemente des Arrays. Das Resultat ist ein zeilenweises Iterieren über alle Elemente des Arrays *realEstate*.

[10] https://rml.io/yarrrml/matey/

Die eigentlichen Mapping-Regeln werden über das Schlüsselwort *mappings* eingeleitet. Ähnlich den verschiedenen Datenquellen erhält auch jedes Mapping einen Bezeichner, über den es innerhalb der YARRRML-Datei referenzierbar ist. Jedes Mapping besteht dabei aus drei Kernbestandteilen. Zuerst steht eine Referenz auf die verwendeten Datenquellen (*sources*). Als nächstes folgt über das Schlüsselwort *s* oder *subject* eine Angabe wie das jeweilige Subjekt heißen soll. Um Verwechslungen auszuschließen, sollte hier eine Formel gewählt werden, die einzigartige URIs garantiert. Für das erste Mapping *realEstate* bietet sich im Ausgangsdatensatz die einzigartige *publicId* an. Die URI resultiert dann aus dem Präfix *ex* und der Referenz auf die *publicId* via *$*. Zuletzt folgt der *po*-Block. Die alternative Schreibweise dieses Schlüsselworts lautet *predicateobject*. In diesem Block werden alle Kombinationen aus Prädikat und Objekt zum Subjekt gelistet. Die Auflistung beginnt mit der Typenzuweisung für das Subjekt. Die Bezeichnung *a* ist eine Kurzform des Prädikats *rdf:type*. Mit eckigen Klammern lassen sich sowohl Objekte, die das gleiche Prädikat haben, als auch Prädikate, die das gleiche Objekt haben, zusammenfassen. Jede Immobilie wird also zu einem *reo:RealEstate* und einer *schema:Accomodation*. So werden alle Properties der Real Estate Ontology (*reo*) abgearbeitet und teilweise durch *schema.org*-Prädikate ergänzt, wenn ein passendes Property von *schema:Accomodation*[11] existiert. Für Prädikate, die ein getyptes Literal fordern, wird noch eine extra Angabe in Form einer *xsd*-Typenzuweisung hinter dem Objekt ergänzt. Prädikate, welche das Subjekt mit einer Instanz verbinden, erhalten den Objektzusatz *~iri*. Dieser Zusatz markiert die Objekte als IRIs, eine internationalisierte Form der URIs.

Am Schluss des *po*-Blocks stehen einige Referenzen auf andere Mappings. Für diese Referenzen werden Prädikat und Objekt getrennt formuliert. Im Objekt-Block wird mit dem Schlüsselwort *mapping* über den Bezeichner auf das jeweilige Mapping gezeigt. Zusätzlich wird über *condition* eine Bedingung gestellt, die erfüllt werden muss, damit das Referenz-Mapping ausgeführt wird. Die Bedingung hat bei allen Referenzen die Form der eingebauten Funktion *equal*, die zwei String-Parameter annimmt, vergleicht und bei Gleichheit den Wert *true* zurückgibt und die Bedingung damit erfüllt. Der erste Parameter stammt aus dem Ausgangs-Mapping *realEstate*, der zweite Parameter aus dem jeweiligen Referenz-Mapping. Da es in diesem Fall nur eine Datenquelle gibt, sind alle Parameter identisch. Die Bedingungen wurden nur eingesetzt, da sonst pro Subjekt vom Mapping *realEstate* alle Adressen, Kontaktpersonen, Energieausweise etc. des gesamten Datensatzes verknüpft würden. Dies liegt darin begründet, dass alle Mappings die gleiche Datenquelle verwenden und jedes Mapping eine neue Iteration über den Datensatz startet.

Die Mappings *address*, *contactPerson* und *energyPass* setzen jeweils die auf sie definierten Klassenrestriktionen der Ontologie um. Die Mappings *schemaAddress* und *schemaRealEstateListing* werden genutzt um weitere passende *schema.org*-Ausdrücke zu verwenden, die auf *schema:Accomodation* nicht passen.

[11] https://schema.org/Accommodation

4.4 Mapping-Prozess & Abspeichern in der Datenbank

Das Umsetzen der erstellten Mapping-Regeln nimmt einen vergleichsweise kleinen Teil in Anspruch. Der erste Schritt besteht im Übersetzen von YARRRML zu RML. Hierfür kommt der bereits erwähnte YARRRML-Parser (siehe Abschnitt 4.1.3) zum Einsatz. Das CLI-Tool wird über *npm* installiert mit dem Befehl:

```
npm i -g @rmlio/yarrrml-parser
```

Die Übersetzung wird über folgenden Befehl angestoßen:

```
yarrrml-parser -i reo_yarrrml.yaml -o reo_rml.ttl
```

Über die *-i*-Flag wird der Pfad zur Eingabedatei in YARRRML und über die *-o*-Flag der Pfad zur Ausgabedatei in RML angegeben. Als nächstes werden die generierten RML-Regeln mit dem RMLMapper umgesetzt. Die Java-Anwendung lässt sich über https://github.com/RMLio/rmlmapper-java/releases/ herunterladen. Wird nun ein Terminal vom Speicherort des Java Archive (JAR) gestartet lässt sich der Mapper mit folgendem Befehl ausführen:

```
java -jar rmlmapper.jar -m reo_rml.ttl -o output.ttl -s turtle
```

Die *-m*-Flag zeigt auf die RML-Regeln, die *-o*-Flag auf die zu erstellende Ausgabedatei und über die *-s*-Flag kann die gewünschte Serialisierung eingestellt werden. Zur Verfügung stehen: *nquads*, *turtle*, *trig*, *trix*, *jsonld* und *hdt*. An dieser Stelle ist zu beachten, dass die Angabe zum Ausgangsdatensatz in den RML-Regeln steht. Es ist also sicherzustellen, dass sich die Datei auch an dem angegeben Ort befindet und den korrekten Dateinamen hat.

Der generierten RDF-Graph *output.ttl* lässt sich nun in eine GraphDB-Instanz abspeichern. Für diesen Zweck muss zuerst über die Workbench ein neues Repository angelegt werden (siehe Abbildung 4.6). Dem Repository wird ein Name und eine ID gegeben. Außerdem wird das Reasoning-Profil ausgewählt, in diesem Fall *No inference*, da für den Anwendungsfall dieser Arbeit kein Schlussfolgern notwendig ist. Nach dem Verbinden mit dem neu erstellten Repository kann über die Workbench der RDF-Import erfolgen. Zuerst wird die Ontologie eingebunden. Dafür steht sowohl der Dateiupload als auch der Import über die URI https://w3id.org/def/reo zur Verfügung. Als nächstes können die RDF-Daten über den Upload importiert werden. Anschließend lässt sich über den Reiter *Explore* → *Graphs overview* prüfen, ob alles korrekt im Repository angekommen ist.

Hier sei noch kurz erwähnt, dass GraphDB ebenfalls eine Lösung für das Mapping von semistrukturierten Daten bereitstellt. Der Ansatz ist eher grafischer Natur und spricht deshalb Anwender an, die sich nicht mit komplexeren Sprachen wie YARRRML und RML auseinander setzen wollen. Das Tool stellt viele Möglichkeiten zur Datenbereinigung von großen Mengen an Daten zur Verfügung. Dies war für

diesen Anwendungsfall nicht erforderlich. Ein großer Nachteil des Ansatzes besteht darin, dass pro Array-Element nur ein einziges Subjekt erzeugt werden kann. Damit kam diese Lösung nicht infrage.

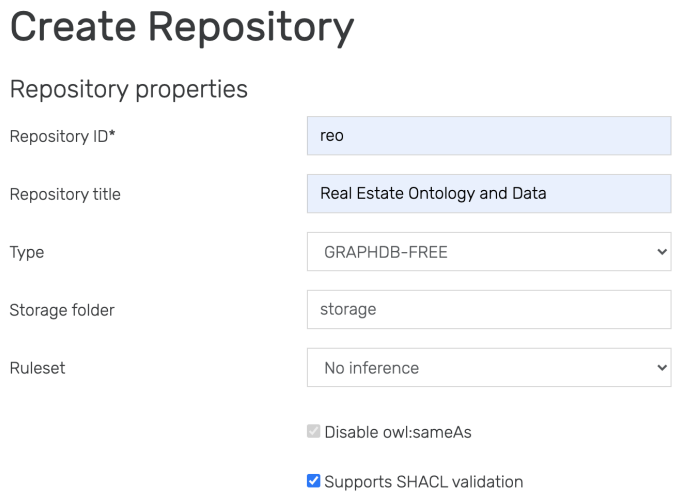

Abbildung 4.6: Erstellen eines GraphDB-Repository

5 Validierung

Zur Validierung des Datensatzes wurde ein SHACL-Shape erstellt, das auf den Klassenrestriktionen der Ontologie basiert. Diese Einschränkungen verbieten nicht das Hinzufügen von Daten, die den Restriktionen widersprechen, sondern liefern einer Reasoning-Engine Hinweise, um auf Klassenzugehörigkeit zu schließen. In dieser Arbeit fungiert die Ontologie allerdings als Datenmodell und ein Schlussfolgern ist nicht erforderlich. Stattdessen sollen die Daten unter einer *Closed-World-Assumption* betrachtet werden. Dies ist mit einer Ontologie nicht möglich und zu diesem Zweck wird ein SHACL-Shape verwendet.

Das in dieser Arbeit verwendete SHACL-Shape besteht aus vier *sh:NodeShape*, die über das Property *sh:targetClass* mit den vier Modellierungsklassen der Ontologie verbunden werden. Diese Klassen sind *RealEstate*, *ContactPerson*, *Address* und *EnergyPass*. Für jede Klassenrestriktion der Ontologie existiert eine *sh:PropertyShape*, welche über *sh:property* mit der entsprechenden *sh:NodeShape* verknüpft wird.

Es lassen sich zwei Arten von *sh:PropertyShape* unterscheiden, die zum Einsatz kommen. Die erste prüft Properties, die zwingend für Instanzen gesetzt sein müssen (Pflichtfelder). Pflichtfelder lassen sich mit SHACL über eine Kombination aus *sh:minCount 1* und *sh:maxCount 1* ausdrücken. Gleichzeitig sagt dies aus, dass ein Property mit diesen Restriktionen auch wirklich nur einmal gesetzt sein darf. Die zweite verwendete Art von Property-Shapes prüft Properties, die gesetzt sein können, aber nicht müssen. Für diesen Zweck reicht *sh:maxCount 1*, was ebenfalls ein mehrfaches Setzen abfängt.

Eine weitere Einschränkung wird mit *sh:nodeKind* auf die Property-Shapes festgelegt. Der Wert ist entweder *sh:IRI*, für Properties deren Objekt eine Instanz mit einer URI sein soll, oder *sh:Literal*, für Properties deren Objekt ein Literal sein soll. Für den zweiten Fall wurde via *sh:datatype* immer ein konkreter XSD-Datentyp mit angegeben.

Einen Sonderfall nimmt die Unterteilung von Immobilien in Verkaufs- und Vermietungsobjekte ein. Daraus resultiert, dass eine Immobilie entweder einen *reo:houseType* und einen *reo:purchasePrice* haben muss oder über einen *reo:apartmentType* in Kombination mit einer *reo:netRent* verfügen muss. Beide Fälle stellen je zwei verschiedene Pflichtfelder dar. Für diese Konstruktion wurde *sh:xone* verwendet. Über dieses Property lassen sich verschiedene Restriktionen spezifizieren von denen genau eine erfüllt sein darf (siehe Material A.3).

Über den Ausgangsdatensatz ist bekannt, dass er Testobjekte enthält, die für firmeninterne Zwecke angelegt wurden. Diese Objekte sind, neben der Erwähnung des Wortes *Test* im Titel, nicht besonders gekennzeichnet. Um diese Objekte herauszu-

filtern, wurde ein Property-Shape erstellt, das mit einer *regular expression* den Titel nach dem Wort *Test* durchsucht und nur Titel zulässt, die dieses Wort nicht enthalten. Über das Property *sh:flags* wurde zusätzlich angegeben, dass dabei Groß- und Kleinschreibung keine Rolle spielt.

```
val:title a sh:PropertyShape ;
        sh:nodeKind sh:Literal ;
        sh:minCount "1"^^xsd:nonNegativeInteger ;
        sh:maxCount "1"^^xsd:nonNegativeInteger ;
        sh:path     :title ;
        sh:pattern  "^((?!test).)*$" ;
        sh:flags    "i" ;
        sh:datatype xsd:string .
```

GraphDB bietet für Repositories eine Unterstützung für Datenvalidierung mit SHACL-Shapes. Es gilt zu beachten, dass ein Repository bei seiner Erstellung extra als SHACL-Repository markiert werden muss. Diese Einstellung ist nachträglich nicht mehr möglich.

Das Vorgehen bei der Validierung ähnelt dem normalen Import von RDF-Graphen. Zuerst wird auch hier die Ontologie eingebunden. Im nächsten Schritt wird über den Dateiupload das SHACL-Shape importiert, dabei wird der Zielgraph explizit benannt. Aus der Dokumentation[1] lässt sich entnehmen, dass der Zielgraph die URI `http://rdf4j.org/schema/rdf4j#SHACLShapeGraph` haben muss. Über diese URI erfolgt die Verbindung zum SHACL-Service des RDF4J-Framework namens *Shacl-Sail*.

Die eigentliche Validierung wird immer dann aktiv, wenn neue RDF-Graphen zum Repository hinzugefügt werden. Beim Import der Daten, die über die RML-Regeln erstellt wurden, liefert GraphDB einen Validation Report mit 155 Verstößen. Einen Auszug zeigt Abbildung 5.1. Die Verstöße lassen sich in drei Kategorien aufteilen:

- 99 Objekte enthielten das Wort *Test* in irgendeiner Form im Titel.
- Bei 45 Objekten fehlte die Angabe zum *reo:marketingType*.
- Elf Objekte nutzten eine veraltete Angabe zum *reo:energyPassType*.

Einige Objekte wiesen Verstöße in mehreren Kategorien auf. Die 99 identifizierten Testobjekte wurden aus dem Datensatz entfernt. Für Objekte ohne *reo:marketingType* wurde dieser entsprechend ergänzt. Die Angabe zum Energieausweistyp wurde bei den elf Objekten auf den aktuelleren Wert korrigiert.

Beim erneuten Import wurden keine weiteren Verstöße gegen das SHACL-Shape festgestellt.

[1] https://graphdb.ontotext.com/documentation/free/shacl-validation.html

```
:node1eq09njdmx41686 a sh:ValidationResult;
  sh:focusNode <http://example.com/MV-ZCD578>;
  sh:sourceConstraintComponent sh:MinCountConstraintComponent;
  sh:sourceShape <http://example.com/validation_shape#marketingType>;
  sh:resultPath <https://www.w3id.org/def/reo#marketingType> .

:node1eq09njdmx41689 a sh:ValidationResult;
  sh:focusNode <http://example.com/MV-AJM566>;
  sh:sourceConstraintComponent sh:PatternConstraintComponent;
  sh:sourceShape <http://example.com/validation_shape#title>;
  sh:resultPath <https://www.w3id.org/def/reo#title>;
  sh:value "Patrick Test - Onboarding Mail" .

:node1eq09njdmx41799 a sh:ValidationResult;
  sh:focusNode <http://example.com/MV-XDI600_energyPass>;
  sh:sourceConstraintComponent sh:ClassConstraintComponent;
  sh:sourceShape
  ↪  <http://example.com/validation_shape#energyPassType>;
  sh:resultPath <https://www.w3id.org/def/reo#energyPassType>;
  sh:value <https://www.w3id.org/def/reo#DEMAND_INDUSTRY> .
```

Abbildung 5.1: Auszug aus dem Validation Report

6 Fazit

Das Ziel dieser Arbeit war das Überführen eines Datensatzes zu Linked Data anhand einer Ontologie, die das zugrunde liegende Datenmodell adäquat abbildet. Gleichzeitig sollten die überführten Daten mit Begriffen eines verbreiteten Vokabulars angereichert werden, um Systemen, die die Daten verarbeiten, eine Möglichkeit zu geben, den Kontext der enthaltenen Informationen einordnen zu können.

Da die Ontologie direkt aus dem Datensatz erwachsen ist, lässt sich das resultierende Datenmodell als gleichwertig zum Ausgangsdatenmodell betrachten. Durch die stärkere Vernetzung mit Konzepten aus bekannten Quellen wie DBpedia ließen sich die Instanzen der feststehenden Begriffe noch präziser beschreiben.

Der vorgestellte Ansatz erlaubt, durch die Trennung von Mapping und Datenmodell, die Erweiterung mit zusätzlichen *schema.org*-Begriffen oder Begriffen aus gänzlich anderen Vokabularen wie zum Beispiel *GoodRelations*[1]. Auch auf Änderungen in den Ausgangsdaten kann so leicht reagiert werden. Zusätzlich lassen sich die verwendeten Werkzeuge in größere Prozesse einbinden, da sie nicht auf lokale Dateien, wie sie in dieser Arbeit verwendet wurden, angewiesen sind, sondern auch Datenströme verarbeiten können.

Mithilfe des entwickelten SHACL-Shape lassen sich große Mengen an Daten auf die formulierten Erwartungen testen. Weiterhin wäre es möglich, das Shape in Kombination mit der Ontologie als Basis für die Entwicklung eines dynamischen Webformulars zur Erstellung von Immobiliendaten zu verwenden. So könnten die Instanzen der Ontologie die Auswahlmöglichkeiten im Formular liefern und das Shape Nutzereingaben den Erwartungen entsprechend validieren. Außerdem ließen sich Beschreibungstexte zu den einzelnen Datenfeldern in die Ontologie integrieren.

Die Ontologie eignet sich nur begrenzt zur Verwendung in anderen Systemen, da sie stark auf den vorliegenden Anwendungsfall zugeschnitten ist. Der gezeigte Ansatz hingegen lässt sich auch auf andere Domänen anwenden und kann dazu beitragen, semistrukturierte Daten für Suchmaschinen im Netz zugänglicher zu machen.

[1] http://www.heppnetz.de/ontologies/goodrelations/v1.html

A Material

A.1 JSON-Dokument einer Immobilie


```
{
    "publicId": "MV-EHB956",
    "title": "Charmante 4-Zi.-Erdgeschosswohnung",
    "address": {
        "street": "Hagener Allee",
        "streetNumber": "76",
        "zipcode": "22926",
        "city": "Ahrensburg",
        "countryCode": "DE"
    },
    "type": "FLAT",
    "usageType": "LIVING",
    "disposalType": "RENT",
    "apartmentType": "GROUND_FLOOR",
    "allocation": {
        "livingSpace": 97.76,
        "rooms": 4,
        "floor": 0,
        "floorCount": 4,
        "bedroomCount": 3,
        "bathroomCount": 2,
        "balcony": true,
        "garden": true,
        "kitchen": true,
        "washingmachine": true,
        "cellar": true,
        "carspaceType": "UNDERGROUND"
    },
    "costs": {
        "rentNet": {
          "amount": 970,
          "currency": "EUR"
        },
```

```
        "additionalCost": {
          "amount": 120,
          "currency": "EUR"
        },
        "carspaceCosts": {
          "amount": 60,
          "currency": "EUR"
        },
        "bailCosts": {
          "amount": 2910,
          "currency": "EUR"
        },
        "heatingCost": {
          "amount": 90,
          "currency": "EUR"
        }
    },
    "stats": {
        "buildYear": 1995
    },
    "energyPass": {
        "energyPresentType": "PRESENT",
        "energyPassType": "CONSUMPTION",
        "energyCarrier": "GAS",
        "heatingType": "CENTRAL",
        "energyEfficiencyCategory": "B"
    },
    "conditions": {
        "condition": "NEAT",
        "equipmentCondition": "NORMAL"
    },
    "tenantPreferences": {
        "petAllowed": "AGREEMENT",
        "dogs": "AGREEMENT",
        "cats": "AGREEMENT",
        "smoking": "AGREEMENT",
        "livingCommunity": "AGREEMENT",
        "commercialUse": "AGREEMENT",
        "minIncome": {
          "amount": 2910,
          "currency": "EUR"
        },
        "maxHouseholdPersons": 3
```

```
    },
    "caretakerContact": {
        "title": "OLD_TENANT",
        "email": "max.mustermann@gmail.com",
        "notAvailable": false
    }
}
```

A.2 YARRRML-Regeln

```
prefixes:
  ex: http://example.com/
  reo: https://www.w3id.org/def/reo#
  xsd: http://www.w3.org/2001/XMLSchema#
  schema: http://schema.org/

sources:
  realEstateList:
    access: real_estate_data.json
    referenceFormulation: jsonpath
    iterator: "$.realEstate[*]"

mappings:
  realEstate:
    sources: realEstateList
    s: ex:$(publicId)
    po:
      - [a, [schema:Accomodation, reo:RealEstate]]
      - [[schema:name, reo:title], $(title)]
      - [reo:marketingType, reo:$(disposalType)~iri]
      - [reo:realEstateType, reo:$(type)~iri]
      - [reo:apartmentType, reo:$(apartmentType)~iri]
      - [reo:houseType, reo:$(houseType)~iri]
      - [reo:usageType, reo:$(usageType)~iri]
      - [[schema:floorSize, reo:livingSpace],
        ↪ $(allocation.livingSpace), xsd:float]
      - [[schema:numberOfRooms, reo:numberOfRooms],
        ↪ $(allocation.rooms), xsd:float]
      - [[schema:numberOfBedrooms, reo:numberOfBedrooms],
        ↪ $(allocation.bedroomCount), xsd:int]
      - [[schema:numberOfBathroomsTotal, reo:numberOfBathrooms],
        ↪ $(allocation.bathroomCount), xsd:int]
```

```
29        - [reo:floor, $(allocation.floor), xsd:int]
30        - [reo:floorCount, $(allocation.floorCount), xsd:int]
31        - [reo:hasGarden, $(allocation.garden), xsd:boolean]
32        - [reo:hasFittedKitchen, $(allocation.kitchen), xsd:boolean]
33        - [reo:hasBalcony, $(allocation.balcony), xsd:boolean]
34        - [reo:hasCellar, $(allocation.cellar), xsd:boolean]
35        - [reo:hasWashingMachineConnection,
          ↪  $(allocation.washingmachine), xsd:boolean]
36        - [reo:netRent, $(costs.rentNet.amount), xsd:float]
37        - [reo:bail, $(costs.bailCosts.amount), xsd:float]
38        - [reo:additionalCost, $(costs.additionalCost.amount),
          ↪  xsd:float]
39        - [reo:carspaceCost, $(costs.carspaceCosts.amount), xsd:float]
40        - [reo:purchasePrice, $(costs.buyCost.amount), xsd:float]
41        - [reo:heatingCost, $(costs.heatingCost.amount), xsd:float]
42        - [[schema:yearBuilt, reo:constructionYear],
          ↪  $(stats.buildYear), xsd:int]
43        - [reo:modernizationYear, $(stats.modernizationYear), xsd:int]
44        - [reo:condition, reo:$(conditions.condition)~iri]
45        - [reo:equipmentCondition,
          ↪  reo:$(conditions.equipmentCondition)~iri]
46        - [reo:cat, reo:$(tenantPreferences.cats)~iri]
47        - [reo:dog, reo:$(tenantPreferences.dogs)~iri]
48        - [reo:otherPet, reo:$(tenantPreferences.petAllowed)~iri]
49        - [reo:commercialUse,
          ↪  reo:$(tenantPreferences.commercialUse)~iri]
50        - [reo:livingCommunity,
          ↪  reo:$(tenantPreferences.livingCommunity)~iri]
51        - [reo:smoking, reo:$(tenantPreferences.smoking)~iri]
52        - [reo:maxHouseholdPersons,
          ↪  $(tenantPreferences.maxHouseholdPersons), xsd:int]
53        - [reo:minIncome, $(tenantPreferences.minIncome.amount),
          ↪  xsd:float]
54        - [schema:petsAllowed, $(tenantPreferences.petAllowed)]
55        - [schema:accommodationCategory, $(apartmentType)]
56        - [schema:accommodationCategory, $(houseType)]
57        - [schema:floorLevel, $(allocation.floor)]
58        - p: reo:address
59          o:
60            mapping: address
61            condition:
62              function: equal
63              parameters:
```

```
64                - [str1, $(publicId)]
65                - [str2, $(publicId)]
66          - p: reo:contact
67            o:
68              mapping: contactPerson
69              condition:
70                function: equal
71                parameters:
72                  - [str1, $(publicId)]
73                  - [str2, $(publicId)]
74          - p: reo:energyPass
75            o:
76              mapping: energyPass
77              condition:
78                function: equal
79                parameters:
80                  - [str1, $(publicId)]
81                  - [str2, $(publicId)]
82          - p: schema:address
83            o:
84              mapping: schemaAddress
85              condition:
86                function: equal
87                parameters:
88                  - [str1, $(publicId)]
89                  - [str2, $(publicId)]
90          - p: schema:additionalType
91            o:
92              mapping: schemaRealEstateListing
93              condition:
94                function: equal
95                parameters:
96                  - [str1, $(publicId)]
97                  - [str2, $(publicId)]
98
99      address:
100       sources: realEstateList
101       s: ex:$(publicId)_address
102       po:
103         - [a, reo:Address]
104         - [reo:street, $(address.street)]
105         - [reo:houseNumber, $(address.streetNumber)]
106         - [reo:zipCode, $(address.zipcode), xsd:int]
```

```
107         - [reo:city, $(address.city)]
108         - [reo:countryCode, $(address.countryCode)]
109
110     contactPerson:
111       sources: realEstateList
112       s: ex:$(publicId)_contactPerson
113       po:
114         - [a, [schema:RealEstateAgent,reo:ContactPerson]]
115         - [reo:contactType, reo:$(caretakerContact.title)~iri]
116         - [reo:notAvailable, $(caretakerContact.notAvailable),
            ↪ xsd:boolean]
117         - [[schema:email, reo:email], $(caretakerContact.email)]
118         - [[schema:telephone, reo:phoneNumber],
            ↪ $(caretakerContact.phoneNumber)]
119
120     energyPass:
121       sources: realEstateList
122       s: ex:$(publicId)_energyPass
123       po:
124         - [a, reo:EnergyPass]
125         - [reo:energyPassType, reo:$(energyPass.energyPassType)~iri]
126         - [reo:energyPassPresentType,
            ↪ reo:$(energyPass.energyPresentType)~iri]
127         - [reo:energyCarrier, reo:$(energyPass.energyCarrier)~iri]
128         - [reo:heatingType, reo:$(energyPass.heatingType)~iri]
129         - [reo:energyEfficiencyClass,
            ↪ reo:$(energyPass.energyEfficiencyCategory)~iri]
130
131     schemaAddress:
132       sources: realEstateList
133       s: ex:$(publicId)_schemaAddress
134       po:
135         - [a, schema:PostalAddress]
136         - [schema:streetAddress, $(address.street)
            ↪ $(address.streetNumber)]
137         - [schema:postalCode, $(address.zipcode)]
138         - [schema:addressLocality, $(address.city)]
139         - [schema:addressCountry, $(address.countryCode)]
140
141     schemaRealEstateListing:
142       sources: realEstateList
143       s: ex:$(publicId)_schemaRealEstateListing
144       po:
```

```
        - [a, [schema:RealEstateListing, schema:Offer]]
        - [schema:price, $(costs.rentNet.amount)]
        - [schema:price, $(costs.buyCost.amount)]
```

A.3 Auszug aus dem SHACL-Shape

```
@prefix :       <https://www.w3id.org/def/reo#> .
@prefix sh:     <http://www.w3.org/ns/shacl#> .
@prefix rdf:    <http://www.w3.org/1999/02/22-rdf-syntax-ns#> .
@prefix xsd:    <http://www.w3.org/2001/XMLSchema#> .
@prefix val:    <http://example.com/validation_shape#> .

val:RealEstate a sh:NodeShape ;
        sh:nodeKind sh:IRI ;
        sh:targetClass :RealEstate ;
        sh:xone (
        [
                sh:nodeKind sh:IRI ;
                sh:minCount         "1"^^xsd:nonNegativeInteger ;
                sh:maxCount         "1"^^xsd:nonNegativeInteger ;
                sh:path     :apartmentType ;
                sh:class            :ApartmentType  ;
        ]
        [
                sh:nodeKind sh:IRI ;
                sh:minCount         "1"^^xsd:nonNegativeInteger ;
                sh:maxCount         "1"^^xsd:nonNegativeInteger ;
                sh:path     :houseType ;
                sh:class            :HouseType  ;
        ]   ) ;
        sh:xone (
        [
                sh:nodeKind sh:Literal ;
                sh:minCount         "1"^^xsd:nonNegativeInteger ;
                sh:maxCount         "1"^^xsd:nonNegativeInteger ;
                sh:path     :netRent ;
                sh:datatype xsd:float ;
        ]
        [
                sh:nodeKind sh:Literal ;
                sh:minCount         "1"^^xsd:nonNegativeInteger ;
                sh:maxCount         "1"^^xsd:nonNegativeInteger ;
```

```
37                  sh:path     :purchasePrice ;
38                  sh:datatype xsd:float ;
39          ]   ) ;
40          sh:property val:address, val:contact, val:energyPass,
41                  val:marketingType, val:realEstateType, val:usageType,
42                  val:tenantGuidelines, val:hasBalcony, val:hasCellar,
43                  val:hasElevator, val:hasFittedKitchen, val:hasGarden,
44                  val:hasGuestToilet, val:hasWashingMachineConnection,
45                  val:bail, val:carspaceCost, val:additionalCost,
46                  val:heatingCost, val:minIncome,
    ↪ val:modernizationYear,
47                  val:constructionYear, val:floor, val:floorCount,
48                  val:livingSpace, val:maxHouseholdPersons, val:title,
49                  val:numberOfRooms, val:numberOfBathrooms,
50                  val:numberOfBedrooms, val:numberOfCarSpaces .
```

Abbildungsverzeichnis

2.1 Vergleich der Webversionen (Alyona 2012) . . . 7
2.2 Bestandteile eines URI (Wikipedia 2020) . . . 8
2.3 Visualisierung der LOD-Cloud (LODCloud 2020) . . . 9
2.4 Semantic Web Technology Stack (Nowack 2009) . . . 11
2.5 Beispiel eines Graphen . . . 12
2.6 Beispiel eines Graphen mit URIs . . . 12
2.7 Beispiel eines Graphen mit ungetyptem Literal . . . 12
2.8 Beispiel eines Graphen mit getyptem Literal . . . 13
2.9 Das semiotische Dreieck (Schirra 2017) . . . 16
2.10 Semantische Hotelsuche (Busse et al. 2012) . . . 17
2.11 Beispiel einer transiviten Folgerung . . . 19

4.1 Instanzen der Klasse *ApartmentType* . . . 34
4.2 Klassenhierarchie der Ontologie . . . 35
4.3 Klassenrestriktionen von *Address* in Manchester OWL Syntax . . . 36
4.4 Klassenrestriktionen von *Address* in Turtle-Serialisierung . . . 37
4.5 Weboberfläche von *Matey* . . . 38
4.6 Erstellen eines GraphDB-Repository . . . 41

5.1 Auszug aus dem Validation Report . . . 44

Tabellenverzeichnis

2.1 SPARQL-Resultat 24

4.1 Datenfelder mit festen Werten und den entsprechenden Ontologieklassen 35

Literaturverzeichnis

Allemang, Dean & Hendler, Jim & Gandon, Fabien: *Semantic Web for the Working Ontologist: Effective Modeling for Linked Data, RDFS, and OWL*, 3. Aufl., ACM Books 2020

Berners-Lee, Tim: *Semantic Web Road map*, https://www.w3.org/DesignIssues/Semantic, 1998, letzter Zugriff: 9. 12. 2020

Berners-Lee, Tim: *Linked Data*, https://www.w3.org/DesignIssues/LinkedData, 2009, letzter Zugriff: 9. 12. 2020

Blumauer, Andreas & Pellegrini, Tassilo: *Semantic Web und semantische Technologien: Zentrale Begriffe und Unterscheidungen*, In: Pellegrini T., Blumauer A. (eds) Semantic Web, Springer 2006

Brickley, Dan & Miller, Libby: *FOAF Vocabulary Specification 0.99: Namespace Document 14 January 2014 - Paddington Edition*, http://xmlns.com/foaf/spec/, 2014, letzter Zugriff: 10. 12. 2020

Busse, Johannes & Humm, Bernhard & Schlüter, Veronika & Tegtmeier, Erwin & Seiler, Bernhard: *Was bedeutet eigentlich Ontologie?*, Informatik-Spektrum, 37(4), pp. 286–297, Springer 2012

De Meester, Ben & Dimou, Anastasia & Heyvaert, Pieter: *YARRRML*, https://rml.io/yarrrml/spec/, Bearbeitungsstand: 27. 8. 2020, letzter Zugriff: 19. 12. 2020

Dengel, Andreas (Hrsg.): *Semantische Technologien: Grundlagen - Konzepte - Anwendungen*, Springer 2012

Dimou, Anastasia & Vander Sande, Miel & De Meester, Ben & Heyvaert, Pieter & Delva, Thomas: *RDF Mapping Language (RML)*, https://rml.io/specs/rml/, Bearbeitungsstand: 6. 10. 2020, letzter Zugriff: 19. 12. 2020

Google: *Understand how structured data works*, https://developers.google.com/search/docs/guides/intro-structured-data, Bearbeitungsstand: 3. 12. 2020, letzter Zugriff: 18. 12. 2020

Gruber, Thomas R.: *A translation approach to portable ontology specifications*, Knowledge Acquisition, 5(2), pp. 199-220, 1993

Hausenblas, Michael: *5 Star Open Data*, https://5stardata.info, 2015, letzter Zugriff: 9. 12. 2020

Heath, Tom & Bizer, Christian: *Linked Data: Evolving the Web into a Global Data Space. Synthesis Lectures on the Semantic Web: Theory and Technology*, 1. Aufl., Morgan & Claypool 2011

Hitzler, Pascal & Krötzsch, Markus & Rudolph, Sebastian & Sure-Vetter, York: *Semantic Web: Grundlagen*, Springer 2008

K., Alyona: *What's the difference between Web 1.0, Web 2.0 and Web 3.0*, https://textinart.wordpress.com/2012/09/24/whats-the-difference-between-web-1-0-web-2-0-and-web-3-0/, 2012, letzter Zugriff: 9. 12. 2020

LOD Cloud: *The Linked Open Data Cloud*, https://lod-cloud.net/, letztes Update: 16. 11. 2020, letzter Zugriff: 10. 12. 2020

May, Wolfgang: *Reasoning im und für das Semantic Web*, In: Pellegrini T., Blumauer A. (eds) Semantic Web, Springer 2006

Nowack, Benjamin: *The Semantic Web - Not a piece of cake*, http://bnode.org/blog/2009/07/08/the-semantic-web-not-a-piece-of-cake, 2009, letzter Zugriff: 10. 12. 2020

Ontology Engineering Group - Universidad Politécnica de Madrid: *OnToology*, http://ontoology.linkeddata.es/, letzter Zugriff: 16. 12. 2020

Ontotext: *GraphDB Documentation Free Version*, https://graphdb.ontotext.com/documentation/free/, Bearbeitungsstand: 7. 12. 2020, letzter Zugriff: 19. 12. 2020

Schirra, Jörg R. J.: *Zeichen, Zeichenträger, Zeichensystem*, http://www.gib.uni-tuebingen.de/netzwerk/glossar/index.php?title=Zeichen,_Zeichentr%C3%A4ger,_Zeichensystem, In: Schirra, J.R.J.; Halawa, M. & Liebsch, D. (Hg.): Glossar der Bildphilosophie. (2012-2020), 2017, letzter Zugriff: 11. 12. 2020

Stanford Center for Biomedical Informatics Research: *Protégé: A free, open-source ontology editor and framework for building intelligent systems*, https://protege.stanford.edu/, letzter Zugriff: 19. 12. 2020

Stuckenschmidt, Heiner: *Ontologien: Konzepte, Technologien und Anwendungen*, 2. Aufl., Springer 2011

Uschold, Michael: *Demystifying OWL for the Enterprise*, In: Synthesis Lectures on the Semantic Web: Theory and Technology, Morgan & Claypool 2018

Uschold, Michael & Grüninger, Michael: *Ontologies: Principles, Methods and Applications*, Knowledge Engineering Review, 11(2), pp. 93-136, 1996

Wikipedia: *Uniform Resource Identifier*, https://en.wikipedia.org/wiki/Uniform_Resource_Identifier, Bearbeitungsstand: 8. 12. 2020, letzter Zugriff: 11. 12. 2020

W3C OWL Working Group: *OWL 2 Web Ontology Language Document Overview (Second Edition)*, https://www.w3.org/TR/owl2-overview/, Bearbeitungsstand: 11. 12. 2012, letzter Zugriff: 14. 12. 2020

Ich versichere, die vorliegende Arbeit selbstständig ohne fremde Hilfe verfasst und keine anderen Quellen und Hilfsmittel als die angegebenen benutzt zu haben. Die aus anderen Werken wörtlich entnommenen Stellen oder dem Sinn nach entlehnten Passagen sind durch Quellenangaben eindeutig kenntlich gemacht.

Ort, Datum Tim Fock