UNIVERSITÄT KOBLENZ · LANDAU
Fachbereich 4: Informatik

Entwicklung eines generischen Sesame-Sails für die Abbildung von SPARQL-Anfragen auf Webservices

Studienarbeit

im Studiengang Informatik

vorgelegt von

Dominik Brosius

Matrikel-Nr. 205210161

Betreuer: Prof. Dr. Steffen Staab
Institut für Web Science and Technologies, Fachbereich Informatik

Dipl.-Inf. Simon Schenk
Institut für Web Science and Technologies, Fachbereich Informatik

Koblenz, im Januar 2010

Erklärung

Ich versichere, dass ich die vorliegende Arbeit selbständig verfasst und keine anderen als die angegebenen Quellen und Hilfsmittel benutzt habe.

	Ja	Nein
Mit der Einstellung der Arbeit in die Bibliothek bin ich einverstanden.	☐	☐
Der Veröffentlichung dieser Arbeit im Internet stimme ich zu.	☐	☐

.....

(Ort, Datum) (Unterschrift)

Inhaltsverzeichnis

1. Motivation

1.1. Problemstellung

Die Idee des Semantic Web ist, Information im World Wide Web (WWW) „Maschinen-verständlich“ vorzuhalten und damit durch Computerunterstützung handhabbar und über Anwendungsgrenzen hinweg übertragbar zu machen. Obwohl das Semantic Web selbst auf z.T. vergleichsweise alten Ansätzen fußt [Bri01], entwickelte sich das WWW nach dessen Erfindung 1989 zunächst lange Zeit, ohne dass sich Techniken nach dieser Idee etabliert hätten. Stattdessen wurden (und werden) Daten vor allem Menschen-lesbar, zumindest eben nicht Maschinen-verständlich, im WWW bereitgestellt. Die überwältigende Mehrheit der Webanwendungen und deren Webservices repräsentieren ihre Daten nach wie vor auf eine Art und Weise, die eine semantische Maschinenverarbeitung ausschließt. Das betrifft auch (oder gerade) die konsumorientierten, damit aber i.d.R. auch besonders populären und großen Webanwendung wie YouTube, Flickr etc.. Das Semantic Web ist so nicht ohne weiteres erweiterbar auf diesen Teil des WWW. Es wäre daher besonders interessant eine Lösung für das Problem zu finden, solche Webservices dennoch für das Semantic Web zu erschließen.

1.2. Ziele dieser Studienarbeit

Auf Betreiben des World Wide Web Consortium (W3C) hin, ist seit Ende der 90er eine zunehmende Aktivität im Bereich Semantic Web zu verzeichnen. Ziel dieser Initiative ist es, das WWW um semantische Techniken zu ergänzen und so das Semantic Web zu realisieren - was sich bereits in Form von Standards niedergeschlagen hat. Wichtig ist hier RDF (siehe Abschnitt 2.1.3), mit Hilfe dessen semantisches Wissen ausgedrückt werden kann. Mit Sesame (siehe Kapitel 3) gibt es zudem seit 2002 ein Framework, das Anwendungen des Semantic Web den Zugriff auf solche RDF-Daten bereitstellt. Es ermöglicht die Verwaltung von verschiedenen RDF-Datenquellen und stellt die Möglichkeit zur Verfügung Abfragen über den Datenbestand auszuwerten. Solche Abfragen werden in einer eigenen Sprache namens SPARQL (siehe Abschnitt 2.1.4) formuliert, die von einem Sesame-Server angenommen, übersetzt und gegen den RDF-Datenbestand ausgewertet werden. Sesame hat hierfür Architekturschichten, sogenannte Sails (Storage and Inference Layers). Diese Sails bewerkstelligen eben dies für

die verschiedenen RDF-Quellen, abhängig von der zugrunde liegenden Speichertechnik.

In dieser Arbeit soll eine Lösung realisiert werden, die es ermöglicht, aufbauend auf dem Sesame Framework, Datenbestände von nicht-semantischen Web-Diensten im Sinne des Semantic Web auszuwerten. Konkret soll ein Sail (Webservice-Sail) entwickelt werden, dass einen solchen Web-Dienst wie eine RDF-Quelle abfragen kann, indem es SPARQL-Ausdrücke in Methodenaufrufe des Dienstes übersetzt und deren Ergebnisse entsprechend auswertet und zurückgibt. Um eine möglichst große Anzahl von Webservices abdecken zu können, muss die Lösung entsprechend generisch gehalten sein. Das bedeutet aber insbesondere auch, dass das Sail auf die Modalitäten konkreter Services eingestellt werden muss. Es muss also auch eine geeignete Konfigurationsrepräsentation gefunden werden, um eine möglichst gute Unterstützung eines zu verwendenden Webservices durch das Webservice-Sail zu gewährleisten. Die Entwicklung einer solchen Repräsentation ist damit auch Bestandteil dieser Studienarbeit.

1.3. Existierende Ansätze

Es existieren inzwischen recht viele Systeme, die jeweils *einzelne* Webservices für semantische Abfragen heranziehen können. Bei diesen handelt es sich zumeist um sogenannte SPARQL-Endpoints. Dies sind selbst Server, die SPARQL-Anfragen (i.d.R.) über das Netzwerk entgegennehmen und mit den ihnen zugänglichen Daten abgleichen und auswerten können[1]. Die Ergebnisse werden entsprechend an die anfragende Anwendung zurückgegeben. Ein Beispiel für einen solchen Endpoint, der den Datenbestand von Wikipedia abfragbar macht, ist *DBPedia*[2]. DBPedia transformiert Anfragen nicht in Suchaufrufe von Wikipedia, sondern wertet diese direkt am eigenen enstprechend aufbereiteten Bestand von Wikipedia-Daten aus.

Ein anderer Dienst, *flickr^{TM}wrappr*[3] genannt, stützt sich auf DBPedia-Daten, um zu einer angegebenen DBPedia-Ressource Fotos des Fotoportals Flickr aufzusuchen. Dabei finden auch tatsächlich Flickr-Suchaufrufe statt, die zu RDF-Ergebnissen umgewandelt werden.

Der Sinn von solchen SPARQL-Endpoints ist, dass sie den Vorgang für die Auswertung von Anfragen kapseln und über eine Schnittstelle nach außen hin bereitstellen. Dadurch lassen sich solche Systeme allerdings natürlich auch nicht direkt in das Sesame-System integrieren.

Eine Lösung, die sich, wie für diese Studienarbeit gefordert, hingegen direkt auf das Sesame-System stützt, wurde in der Arbeitsgruppe ISWeb der Universität Koblenz im

[1] Einen guten Überblick über aktuelle Entwicklungen bietet `http://esw.w3.org/topic/SparqlEndpoints`

[2] `http://dbpedia.org/About`

[3] `http://www4.wiwiss.fu-berlin.de/flickrwrappr/`

Rahmen des Projekts *Semaplorer*[4] entwickelt. Als Bestandteil dieses Projekts wurde ebenfalls ein Sesame-Sail spezialisiert, das im konkreten Fall den Zugriff auf Flickr ermöglicht. Mit diesem Sail lassen sich Daten aus dem Bestand von Flickr nach verschiedenen Suchkriterien abrufen und die Ergebnisse im Gesamtsystem weitervearbeiten.

Tatsächlich orientiert sich diese Studienarbeit an ebendiesem Flickr-Sail. Allerdings hat es (wie die anderen Lösungen auch) den Nachteil, dass es nicht universell ist, also nur auf einen konkreten Webservice zugeschnitten ist. Je nach Anwendung möchte man aber eventuell eine Vielzahl von Webservices einbinden, dabei aber natürlich den Arbeitsaufwand für die softwaretechnische Integration möglichst gering halten. Dem Autor ist keine bestehende Entwicklung bekannt, die einen generischen Ansatz verfolgt, wie er in dieser Studienarbeit verfolgt werden soll, der dies ermöglicht.

[4] `http://www.uni-koblenz-landau.de/koblenz/fb4/institute/IFI/AGStaab/Research/systeme/semap`

2. Grundlagen

2.1. Semantic Web

An das Semantic Web knüpfen sich Visionen, die in einem berühmten, vielzitierten Artikel [BLHL01] aus dem Jahre 2001 illustriert wurden, an dem u.a. Tim Berners-Lee beteiligt war. Ziel ist es demnach, dass Menschen beim Abruf von Information im Web von Maschinen assistiert werden. Maschinen sollen zukünftig in der Lage sein, Informationen aus dem Web automatisch zusammenzustellen, aufzubereiten und sogar logisch aus anderen Informationen herleiten zu können. Der Mensch muss sich dann nicht mehr um die eigentliche Informationssuche kümmern, sondern richtet nur noch seine Anfragen an einen Computer, der daraufhin den Rest erledigt, dabei aber hochrelevante Ergebnisse liefert. Dazu sollen Computer die Informationen, die sie aufsuchen, quasi verstehen und miteinander in Beziehung setzen können.

2.1.1. Herausforderungen und Ziele des Semantic Web

Das klassische World Wide Web, wie es in der Vergangenheit entstanden ist und wie es seither besteht, birgt eine Reihe von Problemen, die dieser Vision von rechnergestütztem Informationsabruf im Wege stehen (vgl. [HKRS08], S.10). Informationen im Web werden vor allem für Menschen bereitgestellt und sind auch genau so repräsentiert. Zwar können Menschen das eigentliche in Webressourcen dargebotene Wissen problemlos herauslesen. Für Maschinen stellt sich das aber als sehr schwierig (wenn nicht unmöglich) dar. Zunächst ist ein Dokument, wie etwa eine Webseite, im Allgemeinen zusätzlich mit Darstellungsinformationen angereichert, die es vom eigentlichen Text zu unterscheiden gilt. Aber auch den reinen Informationstext können Maschinen nicht wirklich gut verarbeiten, da sie nicht in der Lage sind die Bedeutung der Zeichenketten zu verstehen. Bisherige rechnergestützte Informationssuche im Internet erschöpft sich somit letztlich in rein syntaktischer Analyse der im Netz befindlichen Dokumente.

Das Vorgehen, Informationen aus Webressourcen, die eigentlich für Menschen gedacht sind, zu gewinnen, um sie anschließend Anwendungen zur Weiterverarbeitung zugänglich zu machen[1], ist eine häufige Praxis im WWW, aber ebenso problematisch

[1] Auch bekannt als *Screen Scraping*; sinnbildlich: „Informationen vom Bildschirm kratzen“

(vgl. [BLF99], S.258 ff.). Schließlich funktioniert dies im konkreten Fall nur solange, bis sich das vorliegende Repräsentationsformat ändert.

Dazu kommt, dass die Kodierungen und Repräsentationsformen auf den verschiedenen Transportebenen und in den unterschiedlichen Anwendungszusammenhängen im Web zum Teil stark variieren. In diesem Kontext spricht man auch von der *Heterogenität* des Web. Es gilt also Informationen so zu fassen, dass sie unabhängig von dem Anwendungskontext, aus dem sie stammen, von jeder Anwendung des Semantic Web gelesen und verarbeitet werden können. Das ist zudem eine Voraussetzung dafür, dass verteiltes Wissen aus unterschiedlichen Quellen kombiniert werden kann.

Es gibt grundsätzlich zwei unterschiedliche Herangehensweisen, um diese Probleme zu überwinden und die Idee hinter dem Semantic Web zu realisieren (vgl. [HKRS08], S.11 ff.). Zum einen kann man versuchen sich durch komplexere Algorithmen seitens der Webanwendung - des Clients, der auf Informationen im Web zugreift – diesem Ziel anzunähern. Man müsste hier bei der Anwendungsentwicklung im erheblichen Maße auf Methoden der Künstlichen Intelligenz (KI) zurückgreifen. Trotz der großen Forschungsanstrengungen in dem Bereich der KI ist die Befähigung einer Maschine zu derart komplexen Fähigkeiten, wie sie hierzu nötig wären, zumindest in absehbarer Zeit nicht umsetzbar. So müssten solche Anwendungen schließlich die intellektuelle Leistung, die ein Mensch bei der Recherche, dem Lesen, Verstehen und Herstellen von Zusammenhängen erbringt, zumindest nachbilden.

Der Ansatz, der durch die Initiative des Semantic Web verfolgt wird, ist ein anderer. Das Semantic Web soll sich zunehmend über das WWW hinweg ausbreiten, das klassische WWW also ergänzen. Im Semantic Web sollen darin enthaltene Informationen bereits so formuliert, strukturiert und organisiert sein, dass Maschinen diese mit vergleichsweise geringem Rechenaufwand leicht auffinden und verarbeiten können. Der entscheidende Aspekt dabei ist, dass einzelne Informationseinheiten zusätzlich um Wissen über deren Bedeutung, eben um semantische Informationen, ergänzt sind. Damit Maschinen bzw. semantische Anwendungen, diese gezielt verarbeiten können, muss offenbar Klarheit darüber herrschen, wie genau solche Informationen tatsächlich formuliert sind. Auch die Art und Weise, wie solche Anwendungen erhaltene Informationen verarbeiten sollen, muss dabei formal und einheitlich geklärt sein. Eine wichtige Konsequenz für diese Herangehensweise ist also, dass es globaler Standards bei der Umsetzung des Semantic Web bedarf.

2.1.2. Die Standards des Semantic Web

Die Standards, auf die sich die Realisierung des Semantic Web stützt, bilden gemeinsam den sogenannten Semantic Web Stack nach Vorschlag des W3C (vgl. [Bra07],[HKRS08]). Ressourcen im Semantic Web werden zunächst durch *Uniform Resource Identifier* (URI) global eindeutig identifiziert. Der formale Aufbau eines konkreten URI ist dabei genau

spezifiziert. Ein URI kann zugleich auch ein *Uniform Resource Name* (URN) oder ein *Uniform Resource Locator* (URL) sein. Im Falle eines URL kann dann zusätzlich auf die jeweilige Ressource, etwa über HTTP, zugegriffen werden (Dereferenzierung). Eine Ressource, die durch einen URI identifiziert ist, kann dabei auch ein beliebiges gedachtes Konzept sein. URIs sind also geeignet Ausschnitte der Realität und verschiedene Anwendungsdomänen konzeptuell abzubilden. Dies ist eine grundlegende Voraussetzung für das Semantic Web, in dem ja semantisches Wissen über beliebige Dinge ausgedrückt werden soll.

Für den Austausch von Informationen im Semantic Web dient die *Extensible Markup Language* (XML) als grundlegendes Format. XML ist aber noch von anderer Bedeutung, die weiter unten diskutiert wird. Es ist zu beachten, dass XML für den Formalismus des Semantic Web jedoch nicht von Bedeutung ist. XML ist bereits ein wichtiger Bestandteil der Architektur des WWW und wurde für das Semantic Web lediglich für die syntaktische Repräsentation von Information wieder aufgegriffen.

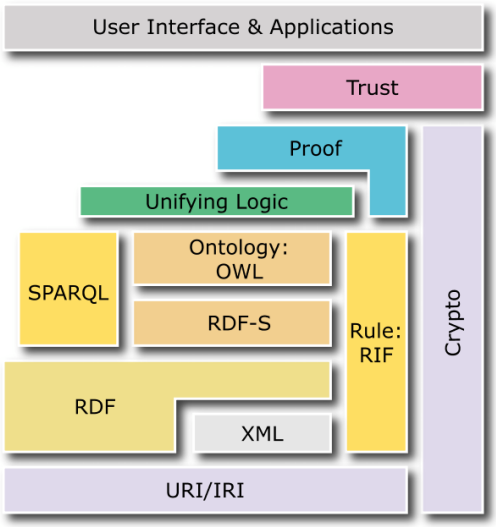

Abbildung 2.1.: Darstellung des Semantic Web Stack (Quelle: [Bra07]).

Zentral für das Semantic Web ist hingegen das *Resource Description Framework* (RDF). Dabei handelt es sich um ein formales Modell zur strukturierten Beschreibung von Wissen. Der Name leitet sich dabei von der ursprünglichen Zielsetzung ab, lediglich Metadaten, also Informationen über bestimmte (Web-)Ressourcen, zu formulieren. RDF lässt sich darüber hinaus jedoch ebenso gut verwenden, um allgemeines Wissen auszudrücken. Diese Tatsache spiegelt sich auch bei *RDF-Schema* (RDFS) wieder. Hierbei handelt es sich um eine Erweiterung von RDF, mit der schematisches Wissen

über Klassen und abstrakte Konzepte ausgedrückt werden kann. Auch die *Web Ontology Language* (OWL) ist eine Sprache, die wie RDFS der Beschreibung von solchem Schemawissen dient. Allerdings ist OWL hierbei deutlich ausdrucksstärker als RDF-Schema, und daher für diesen gesonderten Zweck die wichtigere der beiden Sprachen.

Natürlich möchte man Informationen, die in RDF spezifiziert sind auch abfragen können. Hierzu dient die *SPARQL Protocol and RDF Query Language* (kurz: SPARQL). Mit Hilfe von SPARQL lassen sich semantische Abfragen formulieren und die Repräsentation der zu liefernden Ergebnisse spezifizieren.

Für die logische Folgerung impliziten Wissens muss schließlich neben der Wissensrepräsentation noch die Beschreibung von Schlussregeln ermöglicht werden. Hierfür ist das *Rule Interchange Format* (RIF) vorgesehen, eine Sprache, mit der solche Regeln formuliert und ausgetauscht werden können.

OWL und RIF sind für die Studienarbeit jedoch nicht relevant, stattdessen werden im Folgenden die Sprachen RDF(S) und SPARQL näher betrachtet.

2.1.3. RDF und RDF-Schema

Das Resource Description Framework (RDF, vgl. [LS99], [MMM04]) ist ein wichtiger Standard für die Wissensrepräsentation im Semantic Web. Mit Hilfe von RDF lassen sich Informationen so spezifizieren, dass ein Computer diese semantisch Abfragen kann. Das Datenmodell von RDF basiert dabei auf Graphen. Ein solcher RDF-Graph ist ein (gerichteter) Graph, dessen Knoten *Ressourcen*, leere Knoten (sog. *Blank-Nodes*) oder Datenwerte (sog. *Literale*) sein können und dessen Kanten Beziehungen zwischen einzelnen Knoten bedeuten. Dabei sind Ressourcen-Knoten und Kanten jeweils durch URIs wie z.B. „http://example.org/photos/#42“ global eindeutig identifiziert. Blank-Nodes werden, wie Literale, zwar nicht durch URIs bezeichnet, werden aber als Knoten intern identifiziert. Der Sinn solcher leeren Knoten besteht vor allem in der Formulierung mehrwertiger Beziehungen.

Der Vorteil, den solche RDF-Graphen haben, liegt darin, dass man mehrere solcher Graphen leicht zu einem großen Graph kombinieren kann. Das bedeutet, dass sich mehrere RDF-Datenquellen einfach integrieren lassen, um einem entsprechenden Datenbanksystem als ein einziger Datenbestand für etwaige Abfragen zur Verfügung zu stehen (vgl. Kapitel 3).

Ein RDF-Dokument ist dabei die syntaktische Repräsentation eines solchen Graphen - man spricht auch von Serialisierung. Um einen Graphen effizient zu serialisieren, werden die einzelnen Kanten eines solchen Graphen separat betrachtet, allerdings jeweils mit Ausgangs- und End-Knoten. Die Beschriftungen dieser drei Teile bilden zusammen ein RDF-Tripel aus *Subjekt*, *Prädikat* und *Objekt* – man spricht in dem Zusammenhang auch von Aussagen bzw. *Statements*. Dabei entspricht das Subjekt dem URI einer Ressource, oder der ID einer Blank-Node, von der die zugehörige Kante ausgeht. Das Prädikat ist der URI, der die Kante selbst bezeichnet. Das Objekt ist hingegen entweder

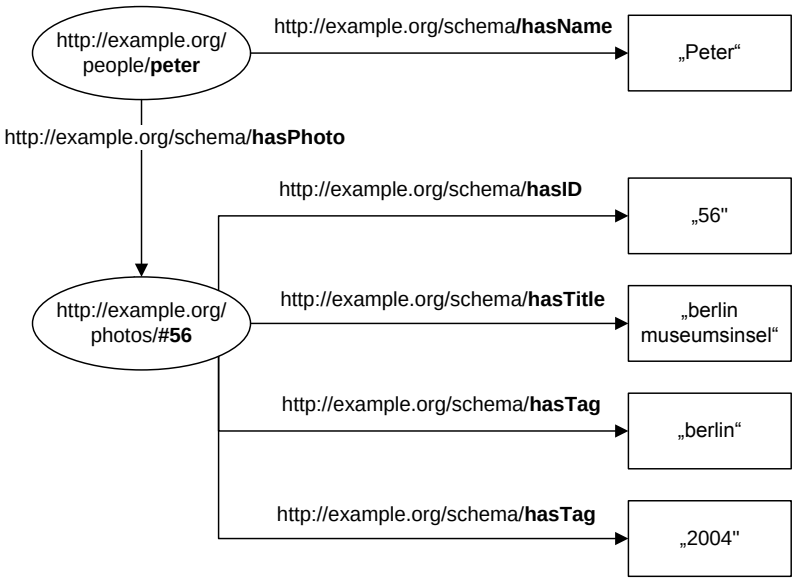

Abbildung 2.2.: Beispiel eines RDF-Graphen.

ein URI, eine Blank-Node-ID oder ein Literal.

Es existieren verschiedene Sprachen, mit denen solche RDF-Tripel aufgeschrieben werden können (und damit das RDF-Dokument). Eine Sprache, die besonders gut lesbar ist, ist die *Terse RDF Triple Language* (Turtle) (vgl. [BBL08]). Mit Turtle lassen sich einzelne RDF-Tripel leicht aufschreiben, indem man die URIs von Subjekt, Prädikat und evtl. Objekt in spitzen Klammern einfasst und in dieser Reihenfolge hintereinander aufschreibt. Blank-Nodes werden in Turtle einfach durch „[]“ bezeichnet. Ein jedes solches Tripel wird mit einem abschließenden Punkt versehen. Eventuelle Literale anstelle des Objekts werden einfach in Anführungszeichen gesetzt.

In Turtle lassen sich die URIs der Prädikate und Ressourcen abkürzend aufschreiben, indem der Basisteil des URI durch ein dokumentweit eindeutiges Präfix ersetzt wird. Hierfür muss das jeweilige Präfix im Kopf eines Turtle-Dokumentes einmalig mit diesem Basis-URI gleichgesetzt werden. Ein URI kann somit in der Form „Präfix:LokalName“ aufgeschrieben werden. Dabei repräsentiert das Präfix gleichzeitig einen Namensraum, in dem der lokale Name („LokalName“) eines konkreten URI bzw. der intendierten Ressource eindeutig ist.

In Turtle gibt es einige Möglichkeiten, Aussagen kompakter zu strukturieren und dadurch noch lesbarer aufzuschreiben. Diese im Detail zu erläutern, würde aber an dieser Stelle zu weit führen. Neben Turtle, die eigentlich eine Teilsprache von *Notation-3* ist, ist aber noch eine andere wichtige Sprache zu nennen. Diese Sprache heißt *RDF/XML*

```
<http://example.org/people/peter>
            <http://example.org/schema/hasName> "Peter".
<http://example.org/people/peter>
            <http://example.org/schema/hasPhoto>
                              <http://example.org/photos/#56>.

<http://example.org/photos/#56>
            <http://example.org/schema/hasTitle> "berlin museumsinsel".
<http://example.org/photos/#56>
            <http://example.org/schema/hasID> "56".
<http://example.org/photos/#56>
            <http://example.org/schema/hasTag> "berlin".
<http://example.org/photos/#56>
            <http://example.org/schema/hasTag> "2004".
```

Abbildung 2.3.: RDF-Serialisierung des Beispielgraphen 2.1.3 in Turtle.

(vgl. [BM04]) und basiert syntaktisch, wie durch den Namen angedeutet, auf XML. Subjekt, Prädikat und Objekt werden hier jeweils durch XML-Elemente repräsentiert. Dabei ist es möglich die Bestandteile mehrerer Tripel ineinander zu schachteln. Der Graph eines RDF-Dokuments wird hier also strukturell durch das Datenmodell von XML (also durch einen Baum) abgebildet.

RDF-Schema (RDFS, vgl. [BG04]) ist eine Erweiterung von RDF, mit der schematisches Wissen formuliert werden kann. Statt nur Aussagen über individuelle Ressourcen und deren Beziehungen zu treffen, lassen sich mit RDFS grundsätzliche Zusammenhänge ausdrücken, die in einer Anwendungsdomäne gelten. Dafür erweitert RDF-Schema RDF insoweit, dass man Aussagen über Klassen von Individuen und deren Beziehungen treffen kann. RDFS stellt dazu Ausdrucksmittel zur Verfügung, mit deren Hilfe man z.B. Klassen-Instanz-Beziehungen ausdrücken und Klassenhierarchien beschreiben kann. Auch lassen sich Untermengen-Beziehungen zwischen Prädikaten spezifizieren. Mit RDFS lassen sich also sogenannte Ontologien beschreiben[2], die bei der semantischen Suche die Interpretation von RDF-Daten erheblich präzisieren können.

2.1.4. Die RDF-Abfragesprache SPARQL

Man kann grundsätzlich drei Ebenen unterscheiden, auf denen man RDF-Daten Abfragen kann (vgl. [HBS08]). RDF-Daten lassen sich auf der Ebene ihrer syntaktischen Repräsentation, auf der Strukturebene der Tripel und auf ihrer semantischen (logischen) Ebene, der Ebene der RDF-Graphen, abfragen. Die ersten beiden Herangehensweisen

[2]Mit RDFS lassen sich nur vergleichsweise leichtgewichtige Ontologien beschreiben. Für komplexere Ontologien muss auf OWL zurückgegriffen werden.

sind problematisch, da sie sich jeweils eher an der zugrundeliegenden Struktur orientieren, statt am logischen Modell von RDF. Im konkreten Fall könnten so wichtige Zusammenhänge, wie beispielsweise schematische Informationen über eine Anwendungsdomäne, die im RDF-Graph eigentlich modelliert sind, verloren gehen.

Mit SPARQL [PS08] gibt es eine relativ neue Anfragesprache, die hingegen die Anfrage auf Graph-Ebene ermöglicht. Anfragen in SPARQL basieren auf Mustern in einem gegebenen Graphen. Eine einfache SPARQL- Anfrage hat dabei die Form wie in Beispiel 2.4.

```
PREFIX ex:  <http://example.org/schema/>
SELECT  ?person ?photo
WHERE  {
        ?person  ex:hasPhoto ?photo.
        ?photo   ex:hasTag "berlin". }
```

Abbildung 2.4.: Beispiel einer Anfrage in SPARQL.

Mit *PREFIX* wird der Namensraum für die verwendeten URIs deklariert. Dadurch lässt sich ein URI in der eigentlichen Anfrage kürzer schreiben. Mit *SELECT* wird das Ausgabeschema der Ergebnisse angegeben. Dabei stellen *?person* und *?photo* im Beispiel Variablen dar, die im *WHERE*-Block verwendet werden. In diesem Teil der Anfrage werden die eigentlichen Graph-Muster formuliert. Dabei handelt es sich ebenfalls um eine Reihe von RDF-Tripeln, die einen eigenen Graphen bilden. Jedoch können anstelle von Subjekt, Prädikat und Objekt hier aber die durch SELECT eingeführten Variablen treten. Gesucht wird dann ein Subgraph im angefragten Datenbestand, der auf dieses Graph-Muster passt. Dabei werden etwaige Variablen im Muster durch jede mögliche Lösung belegt. Dadurch entsteht eine Ergebnistabelle, die jede dieser Variablenbelegungen enthält, und deren Tabellenschema der SELECT-Angabe entspricht.

person	photo
http://example.org/people/peter	http://example.org/photos/#56
http://example.org/people/caroline	http://example.org/photos/#42
http://example.org/people/joan	http://example.org/photos/#116
...	

Tabelle 2.1.: Mögliche Ergebnistabelle, die von der Beispielanfrage geliefert wird.

Neben der Ausgabe in einem einfachen Tabellenschema, besteht in SPARQL auch die Möglichkeit alternative Repräsentationsformen für die Anfragenergebnisse anzugeben. So lassen sich aus den Ergebniswerten beispielsweise wiederum neue Graphen

erzeugen, die sich dann semantisch weiterverarbeiten lassen.

SPARQL kennt aber noch viele weitere Ausdrucksmittel, so etwa zur Beschränkung und Sortierung der Ergebnisse, auf die aber an dieser Stelle nicht weiter eingegangen werden kann.

2.2. XML

Mit Hilfe von XML (vgl. [BPSM+08],[HKRS08]) ist es möglich Dokumente zu erstellen, in denen Daten derart strukturiert angelegt sind, dass Computer diese besonders leicht verarbeiten können. Umgekehrt kann eine Anwendung von ihr angelegte Datenstrukturen mit XML einfach serialisieren (d.h. in Form von Zeichenketten niederschreiben). XML eignet sich damit also sehr gut als Format für den Austausch maschinenlesbarer Dokumente.

XML ist zunächst eine sogenannte Auszeichnungssprache. Das heißt mit ihr können einzelne Ausschnitte eines Textes ausgezeichnet, damit also vom Rest des Textes abgehoben und mit Zusatzinformationen annotiert werden.

Eine Auszeichnung in XML findet durch sogenannte *Tags* statt. Ein Tag ist dabei ein Bezeichner (*XML-Name*), der in spitzen Klammern („<“, „>“) eingefasst ist. Eine konkrete Auszeichnung wird dabei dadurch realisiert, dass der ausgezeichnete Textteil in XML von einem Start-Tag und einem korrespondierenden End-Tag (unterschieden durch ein vorangehendes „/“) umschlossen wird. In einem XML-Dokument stellt ein solches Tag-Paar ein sogenanntes *XML-Element* dar, dessen Inhalt der umschlossene Text ist.

Ein Beispiel:

```
This picture was taken on my vacation to <tag>berlin</tag>
in autumn of <year>2004</year>. It was awesome!
```

Die Grundstruktur eines XML-Dokuments ist dabei die eines Baums. Dies folgt aus der Art und Weise, wie die Tags eines Dokumentes aufeinander folgen dürfen bzw. müssen. Jedes Tag-Paar darf beliebig viele andere XML-Elemente umschließen, allerdings muss jedes dieser Kind-Elemente auch tatsächlich noch innerhalb des umgebenden Elements wieder durch ein End-Tag geschlossen sein.

```
<photos>
     ...
     <photo>berlin museumsinsel
           <tags>
                <tag>berlin</tag>
                <tag>2004</tag>
           </tags>
     </photo>
     <photo>eiffel tower</photo>
</photos>
```

Abbildung 2.5.: Eine Sammlung von Fotos repräsentiert in XML.

Damit hat jedes XML-Element eines Dokuments genau ein übergeordnetes Element

oder ist selbst das oberste Wurzel-Element. Ein Element kann aber beliebig viele untergeordnete Elemente besitzen. Insgesamt repräsentiert ein konkretes XML-Dokument damit allgemein einen Baum mit variablem Verzweigungsgrad, dessen Knoten die XML-Elemente selbst sind.

Neben Elementen gibt es in XML aber noch *Attribute*. Ein solches Attribut ist ein Bezeichner-Wert-Paar, das in eine Tag-Klammer eingetragen werden kann und damit das jeweilige Element näher beschreibt. Attribute werden im Baum-Modell dabei als assoziierte Knoten behandelt.

```
<photos>
     ...
     <photo id="56" title="berlin museumsinsel" ownerName="peter">
          <tags>
               <tag>berlin</tag>
               <tag>2004</tag>
          </tags>
    </photo>
    <photo id="12" title ="eiffel tower" ownerName="joan">
</photo>
</photos>
```

Abbildung 2.6.: XML-Dokument aus Beispiel 2.5 ergänzt um Attribute.

Für die Auswertung bzw. den Abruf einzelner Bestandteile eines XML-Dokuments existieren verschiedene Techniken, die in dieser Studienarbeit noch zum Tragen kommen. So gibt es verschiedene, unterschiedlich mächtige Anfragesprachen, die das Auffinden bzw. die Extraktion bestimmter Elemente und Attribute eines XML-Dokumentes ermöglichen. Relevant sind an dieser Stelle XPath [BBC+07b] und XQuery [BBC+07a]. Einfach gesagt, lassen sich mit XPath-Ausdrücken bestimmte Knoten im Baummodell eines XML-Dokumentes von der Wurzel aus ansteuern und auswählen. Auf dieser Basis lassen sich in XQuery ganze Mengen von Elementen liefern, die so in einem Dokument selektiert wurden, extrahieren und nach beliebiger Art und Weise als Abfragenergebnis zurückgegeben.

Die Bezeichner der Elemente und Attribute sind in XML frei wählbar. Auch in XML-Dokumenten lassen sich dabei mehrere Namensräume einbinden, in denen jeweils eine Reihe von XML-Namen gültig sind. Mit diesem Mechanismus lassen sich innerhalb eines Dokuments Namen aus verschiedenen Vokabularen verwenden, ohne dass es zu Namenskonflikten kommt.

Außerdem gibt es in XML die Möglichkeit mit Hilfe bestimmter schematischer Beschreibungsmittel, wie DTDs (Document Type Definitions) oder XML-Schema, den

```
for $entry in /photos/photo/
let $photo_id = $entry/@id, $title = $entry/@title
return
<solution photo='{data($photo_id)}' title='{data($title)}'/>
```

liefert (u.a.)

```
<solution photo='56' title='berlin museumsinsel'/>
<solution photo='12' title='eiffel tower'/>
...
```

Abbildung 2.7.: Beispielhafte XQuery-Anfrage an Beispiel 2.6 mit zugehöriger Ausgabe.

Aufbau konkreter Dokumente genau zu spezifizieren. Damit ist XML so universell, um als Metasprache Grundlage für andere spezifische (Auszeichnungs-)Sprachen fungieren zu können. So liegt XML im Semantic Web auch Serialisierungssprachen für RDF und OWL zugrunde.

2.3. Webservices

Unter Webservices versteht man grundsätzlich Softwaresysteme, die eine bestimmte Dienstleistung logisch abstrahieren und für die Interaktion mehrerer Systeme zur Verfügung stellen (vgl. [BHM+04]). Die Interaktion findet dabei über ein Netzwerk statt. Dazu besitzen Webservices eine öffentliche Schnittstelle, über die sie zugänglich sind. Die eigentliche Dienstleistung kann dadurch von entfernten Applikationen über Protokolle wie HTTP angefragt und das entsprechende Ergebnis empfangen werden. Der angerufene Dienst ist dabei eine Softwareanwendung, die in der lokalen Umgebung des zugehörigen Servers ausgeführt wird. Die Schnittstelle (API) eines Service besteht aus einer Menge von Methoden, die von außen aufgerufen werden können, um bestimmte Operationen auf Seite des Dienstes auszulösen.

Webservices gliedern sich in der Regel in übergeordnete Systeme, die sogenannten *Service Oriented Architectures* (SOA), ein. Als Teil einer solchen Architektur werden Webservices in aller Regel durch Sprachen wie der *Web Service Description Language* (WSDL) spezifiziert. Mit WSDL lässt sich die Syntax des jeweiligen Service, das heißt z.B. die verfügbaren Schnittstellen-Methoden und der erwartete Kommunikationsweg, formal beschreiben.

Sehr häufig sind solche Webservices anzutreffen, die über ein eigenes Protokoll namens *SOAP* kommunizieren. Es gibt aber auch eine Klasse von Diensten, deren Kommunikationsmodell leichtgewichtiger ist - sogenannte *RESTful Webservices*. Für die vorliegende Studienarbeit wurden diese beiden Arten von Services in Betracht gezogen. In den folgenden Abschnitten soll der Leser mit ihnen etwas vertrauter gemacht werden.

2.3.1. SOAP-basierte Webservices

SOAP ist eine W3C-Empfehlung für Nachrichten-basierte Kommunikation in einem Client/Server-System (vgl. [GHM+07],[BHM+04]). SOAP ist ein leichtgewichtiges Protokoll, das selbst auf bestehenden Anwendungsprotokollen wie z.B. HTTP und FTP aufsetzt.

Bei einem konkreten Webservice stellt der Server einen sogenannten SOAP-Endpoint, einen Kommunikationsendpunkt mit bekannter Adresse, bereit, an den die SOAP-Nachrichten dann gerichtet werden können. Dabei lassen sich mit Hilfe von SOAP neben der rein Nachrichten-orientierten Kommunikation auch sogenannte Remote Procedure Calls (RPC) realisieren. Bei RPC werden von einem Server implementierte Prozeduren von einem Client über das Netzwerk so aufgerufen und die Ergebnisse empfangen, wie man es von normalen Prozeduraufrufen gewohnt ist.

Nachrichten werden in SOAP in XML formuliert. Dabei wird in SOAP nur der Rahmen einer Nachricht vorgeschrieben. Die Repräsentation des eigentlichen Nachrichten-

inhalts und die Definition dessen Struktur obliegt dem Service-Provider. Eine SOAP-Nachricht besteht zunächst aus einem sog. *Envelope*. Dies ist ein XML-Element, dass gleichzeitig das Wurzelelement einer Nachricht ist. Ein Envelope kann ein *Header*-Element enthalten, in dem beispielsweise Service-spezifische Namensräume deklariert werden. Diese beziehen sich auf den eigentlichen Nachrichtentext, der im sogenannten *Body* der Nachricht steht, der wiederum Kind-Element des Envelope ist.

```
<?xml version="1.0" encoding="utf-8" ?>
<s:Envelope
     xmlns:s="http://www.w3.org/2003/05/soap-envelope"
     xmlns:xsi="http://www.w3.org/1999/XMLSchema-instance"
     xmlns:xsd="http://www.w3.org/1999/XMLSchema">
   <s:Body>
      <ex:ServiceRequest xmlns:ex="urn:exampleservice">
         <ex:method>get_photo</method>
           <ex:photo_id>56</photo_id>
      </ex:ServiceRequest>
   </s:Body>
</s:Envelope>
```

Abbildung 2.8.: Beispiel einer Request-Nachricht gemäß SOAP.

Um einen Serviceaufruf in SOAP zu starten, muss ein Client eine solche Nachricht (Request) an den SOAP-Endpoint des Service senden. Dazu fügt er all jene Informationen in den Body der Nachricht ein, die dazu vom Server gefordert sind. Auch die Strukturierung dieser Information wird durch den Service vorgeschrieben. War der Aufruf erfolgreich, so antwortet der Service seinerseits mit einer SOAP-Nachricht (Response), deren Body-Element das gelieferte Ergebnis enthält. Dieses ist in der Praxis aus nahe liegenden Gründen, ebenso wie der Nachrichtenrahmen, in XML ausgedrückt. Vorgaben hierzu gibt es durch SOAP, wie gesagt, jedoch nicht. Vielmehr werden solche Angaben, wenn nicht natürlichsprachlich an den Service-Anwender veröffentlicht, in WSDL-Dokumenten definiert. Das betrifft etwa sowohl den erwarteten Aufbau von Request- und Response-Nachrichten, als auch Informationen, die für die Realisierung von RPC mit SOAP notwendig sind.

2.3.2. RESTful Webservices

Viele neuere Dienste im WWW sind sogenannte RESTful Webservices. Dies sind solche Dienste, die sich in ihrer Implementation an einem Stil orientieren, dessen Prinzipien unter dem Schlagwort *REST* (Representational State Transfer) bekannt sind. REST wurde von Roy Fielding im Jahre 2000 im Rahmen seiner Dissertation (siehe [Fie00]) vorgeschlagen und ausgearbeitet und ist seither immer populärer geworden. REST ist kein Gegenentwurf zu SOA - ein SOA-System kann durchaus auch RESTful Webservices

```
<?xml version="1.0" encoding="utf-8" ?>
<s:Envelope
     xmlns:s="http://www.w3.org/2003/05/soap-envelope"
     xmlns:xsi="http://www.w3.org/1999/XMLSchema-instance"
     xmlns:xsd="http://www.w3.org/1999/XMLSchema">
   <s:Body>
      <ex:ServiceResponse xmlns:ex="urn:exampleservice">
         <photo id="56" title="berlin museumsinsel" ownerName="peter">
            <tags>
                 <tag>berlin</tag>
                 <tag>2004</tag>
              </tags>
         </photo>
      </ex:ServiceResponse>
   </s:Body>
</s:Envelope>
```

Abbildung 2.9.: Mögliche Response-Nachricht für Request 2.8.

beinhalten. Stattdessen ist REST genau genommen ein Stil einzelner Client/Server-Architekturen, bei denen der Zugriff auf (verteilte) Medien im Vordergrund steht.

Fielding entwickelte diesen Stil systematisch unter der Berücksichtigung von Erfordernissen des WWW, wie z.B. Skalierbarkeit, Uniformität der Schnittstellen, Vermeidung von Latenz. Dabei orientierte er sich auch an den Technologien, die sich bereits bei der Realisierung des Internet durchgesetzt haben. So stützt sich REST auf Webstandards wie HTTP und URI.

Ein wichtiger Aspekt einer REST-Architektur ist die zustandslose Kommunikation zwischen Clients und Server. Bei REST-Systemen muss der Server also nicht für eine Sitzungsverwaltung sorgen, in der er etwa vorherige Anfragen eines Clients für einen aktuellen Aufruf in Betracht ziehen muss. Stattdessen ist der Client selbst für die Zwischenspeicherung von Zuständen während der Kommunikation mit dem Server verantwortlich. Außerdem muss jeder einzelne Serviceaufruf des Clients an den Server selbst alle Informationen mitteilen, die nötig sind, damit der Server die erfragte Dienstleistung erbringen kann.

Informationen die von einem REST-Service vorgehalten und mit einem Client ausgetauscht werden, sind durch das Konzept der Ressource modelliert. Der Datenbestand eines RESTful-Webservice beinhaltet demnach einzelne Instanzen solcher Ressourcen. Einzelne Ressourcen werden in REST-Systemen durch URLs oder URNs global identifiziert. Ein Client kann dann über die URLs die Repräsentationen(!) der jeweiligen Ressourcen abrufen bzw. manipulieren. Ein solcher Aufruf bedient sich dafür einer der vier grundlegenden HTTP-Methoden GET, POST, PUT und DELETE (siehe [BLFF96],[FGM+99]). Zudem besteht durch HTTP die Möglichkeit dem API-Anwender Informationen, etwa zum Exception-Handling, über den HTTP-Statuscode eines Auf-

rufs mitzuteilen. Mit der direkten Integration von HTTP ins Kommunikationsmodell von REST wurde damit auf eine aufgesetzte Transportschicht, wie SOAP, verzichtet, was RESTful Webservices gegenüber „klassischen“ Diensten leichtgewichtiger sein lässt.

Ressource	**Methode**	**HTTP-Methode**	**URL**
user	Get User	GET	http://ex-service.org/users/*USER_ID*
	Search User	GET	.../users/search?name=*NAME*&...
	Delete User	DELETE	.../users/delete/*USER_ID*
photo	Search Photo	GET	.../photos/search?tag=*TAG*&...
	Upload Photo	PUT	.../users/*USER_ID*/photos/*PHOTO_ID*
	Delete Photo	DELETE	.../users/*USER_ID*/delete/*PHOTO_ID*

Tabelle 2.2.: Beispiel-Schnittstelle eines RESTful Webservice.

Die von einem konkreten Dienst angeforderten Ressourcen (bzw. deren aktuelle Zustände) werden durch Dokumente repräsentiert, die in der Praxis zumeist XML-basiert sind. In der Repräsentation einer Ressource können dabei andere Ressourcen des Webservice referenziert sein. Es gibt eine Reihe von Formaten, in denen solche Repräsentationen ausgedrückt sind. Hierzu sind neben reinem XML besonders das *Atom Syndication Format* (ASF) und *Really Simple Syndication* (RSS) zu zählen. Es gibt aber auch nicht-XML-basierte Dokumentformate wie *JavaScript Object Notation* (JSON). Außerdem können Ressourcen auch durch Bilddateien o.ä. repräsentiert werden.

3. SESAME 2

Wie im vorherigen Abschnitt angedeutet, können in RDF spezifizierte Daten zu großen Graphen zusammengefasst werden. Die RDF-Tripel, die einen solchen Graphen bilden, werden dafür in sogenannten RDF- oder *Triple-Stores* vorgehalten und verarbeitet [HBS08]. Dabei handelt es sich um Systeme, die in der Regel auf Datenbanksystemen aufsetzen, in denen diese Tripel persistent eingelagert werden. Die Aufgabenstellung dieser Studienarbeit bezieht sich im hohen Maße auf das RDF-Speicher- und Abfragesystem Sesame 2 (vgl. [BKvH02]). Es handelt sich hierbei um ein System für die Verwaltung von Triple-Stores, dessen Architektur jedoch unter der Maßgabe gestaltet wurde, eine ganze Reihe von verschiedenen Speicher-Mechanismen zu integrieren. In den so verfügbaren Speichern, sogenannten *Repositories*, können dann RDF-Daten (aber auch in RDF-Schema formulierte Ontologien) abgelegt werden. Im Folgenden soll die Architektur und Funktionsweise von Sesame vorgestellt und dabei der Grund für die Flexibilität des Sesame-Systems näher erläutert werden.

3.1. Architektur-Überblick

Als System, das semantischen Anwendungen (bzw. Clients) als Server den Zugriff auf RDF-Daten ermöglicht, muss Sesame 2 entsprechende Schnittstellen, sogenannte Application Programming Interfaces (API), bereitstellen. Im Wesentlichen ist in der Sesame-Architektur hierfür die *Repository-API* vorgesehen (vgl. [BKvH02],[Adub]). Dies ist eine Schnittstelle, die grundlegende Methoden für den Zugriff auf ebensolche Repositories, insbesondere für die Auswertung von Datenabfragen, bereitstellt. Die Repository-API dient dabei sowohl lokalen Anwendungen, die direkt auf ein konkretes Sesame-System zugreifen können, als auch Anwendungen, die in entfernten Laufzeitumgebungen ausgeführt werden. Für letzteren Fall ist in der Sesame Architektur ein dedizierter HTTP-Server vorhanden, der Anfragen für den Zugriff auf RDF-Repositories über das Netzwerk entgegennehmen kann.

Die Repository-API stellt grundlegende Methoden für den Zugriff auf RDF-Daten zu Verfügung. Die spezifischen Eigenheiten der letztlich für die Lagerung der RDF-Daten verwendeten Speicher-Mechanismen sind für Repository (und Anwendung) dabei transparent. Vielmehr greift ein angerufenes Repository-Objekt selbst auf Methoden des *Storage and Inference Layers* (Sail) zurück (vgl. [BKvH02],[Adub]). Diese Komponente des Sesame-Frameworks ist in der Schichtung der Architektur weiter unten

angesiedelt und damit stärker an der Funktionsweise der realen Datenspeicher orientiert.

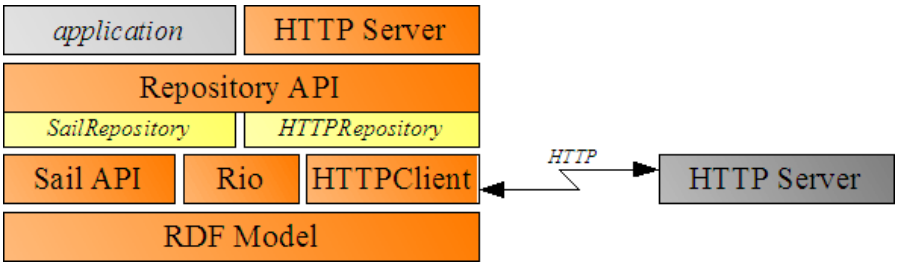

Abbildung 3.1.: Schichten der Sesame-Architektur (Quelle:[Adub]).

Für das Parsen von Datenabfragen, die von der Repository-Schnittstelle entgegengenommen werden, besitzt Sesame dafür zuständige Module. Diese *Query-Engines* wandeln einen Abfrageausdruck in eine Datenstruktur um, die diesen repräsentiert. Das Datenmodell für diese Struktur ist das *Query Model* von Sesame, das Abfragebestandteile wie Operatoren und Graph-Muster softwaretechnisch nachbildet. Sesame unterstützt neben SPARQL auch das Parsen einer eigene Implementation der *RDF Query Language* (SeRQL). Für die Studienarbeit wird allerdings nur die Verarbeitung von SPARQL-Ausdrücken in Betracht gezogen (siehe Abschnitt 3.3).

Für den Austausch von RDF-Daten mit anderen semantischen bzw. RDF-Anwendungen existiert *RDF I/O* (RIO). Diese funktionale Komponente von Sesame ermöglicht, wie vom Namen angedeutet, das Einlesen bzw. Parsen von RDF-Dokumenten, aber auch das Serialisieren von RDF-Daten, die sich in den Repositories eines Sesame-Systems befinden. Dabei lässt sich kontrollieren, ob sämtliche gerade geladene RDF-Daten ausgetauscht werden, oder nur der schematische bzw. nicht-schematische Anteil – je nach dem, für welche Art Anwendung der Datenexport gedacht ist. Sesame beherrscht in der aktuellen Version 2.2 durch RIO das Parsen und Serialisieren (u.a) in den Formaten RDF-XML, N3, N-Triples und Turtle.

Schließlich ist das *RDF Model* zu nennen. Dies ist die der gesamten Architektur zugrunde liegende Komponente des Sesame-Frameworks. Wie der Name bereits andeutet, handelt es sich hierbei um eine softwaretechnische Realisierung des formalen RDF Modells (vgl. Abschnitt 2.1.3). Die einzelnen Bestandteile dieses Modells, wie *Statement*, *Resource*, *Literal*, *URI*, *BlankNode*, sind darin als Schnittstellen und Klassen implementiert und kommen bei der internen Repräsentation von RDF-Daten, aber auch von SPARQL-Anfragen, zum Tragen. Der Anwendungsentwickler kann über die Methoden der einzelnen Klassen und Objekte dann auf geladene RDF-Daten zugreifen, etwa um den Wert eines Literals auszulesen oder das Subjekt eines RDF-Statements zu erfragen.

3.2. Repository- und Sail-API

Aus Sicht eines Anwendungsprogrammierers stellt die Repository-API die wichtigste Schnittstelle für den eigentlichen Datenzugriff dar (vgl. [BKvH02],[Adub]). Mit Hilfe dieser kann er erst Abfragen (beispielsweise in SPARQL) über den Datenbestand eines Repositories an das System stellen. Die API ermöglicht aber auch das Laden und die Manipulation einzelner RDF-Statements im RDF-Graphen. Auch ist es von hier aus möglich, weitere RDF-Dokumente bzw. die Graphen, die sie beschreiben, in das Repository aufzunehmen.

Da die Schnittstelle eines Repositories im Allgemeinen von mehreren Anwendungen zur gleichen Zeit verwendet werden kann, muss Sesame hier, wie bei klassischen Datenbanksystemen auch, für die Konsistenz der Daten sorgen. Folgen kritischer Manipulationen, die über die Repository-API eingeleitet werden, müssen hierfür in Transaktionen gruppiert werden. Dies geschieht automatisch, kann aber auch vom Programmierer manuell beeinflusst werden, z.B. um die Effizienz bei langen und/oder schnellen Zugriffsfolgen zu erhöhen.

Wie bereits angesprochen, werden Befehle, die an ein Sesame-Repository gerichtet werden, nicht direkt von oder in dem Repository-Objekt verarbeitet, sondern in der Schichtenarchitektur von Sesame nach unten weitergereicht. Es gibt in Sesame zwei wichtige Implementierungen der Repository-Klasse: zum einen das *HTTPRepository* und zum anderen das *SailRepository*. Ersteres ist ein Repository, das Zugriffsbefehle über HTTP an einen entfernten Sesame-Server richtet, der seinerseits den Zugriff auf RDF-Repositories in seiner Umgebung ermöglicht. Bei dem SailRepository handelt es sich um ein Repository, das auf ein Sail-Objekt zugreift, welches dann den tatsächlichen Zugriff auf einen lokalen RDF-Speicher realisiert. Dazu werden die von der Repository-Schnittstelle bereitgestellten Methoden in Aufrufe der Sail-API umgesetzt.

Die API des Sails ist genauer gesagt ein sogenanntes Service Provider Interface (SPI). Einzelne Sail-Implementierungen realisieren diese eine Schnittstelle und sind dadurch aus Sicht des aufrufenden Repositories austauschbar. Eine konkrete Implementierung der Sail-Klasse besitzt hierbei die Kenntnis über das Speichermedium bzw. dessen ganz eigene, spezifische Zugriffsmethodik, um die vorgegebenen Methoden der Sail-API auf diese abzubilden. Das hat zur Folge, dass sich die Speicher-spezifische Information im Hinblick auf die Gesamtarchitektur von Sesame nur in diesen konkreten Sail-Klassen konzentriert. Durch die Sails wird also für die anderen funktionalen Komponenten der Architektur von der physischen Speichertechnik und deren Eigenheiten abstrahiert, stattdessen ein grundlegender logischer Datenzugang auf Basis des RDF-Modells ermöglicht.

Ein Vorgang, der sich in Sesame eng an der Funktionsweise konkreter Speichermedien orientiert und daher in die Verantwortlichkeit der Sail-Implementierungen fällt,

ist die Inferenz, das heißt die Herleitung impliziter Informationen. Der Grund ist, dass sich dies wesentlich effizienter realisieren lässt, wenn dies beim direkten Datenabruf vom jeweiligen Medium – sozusagen on-the-fly – geschieht [BKvH02]. Aber auch die Transaktionsverwaltung ist ebenso Speichermedien-spezifisch und wird ebenfalls von den konkreten Sails realisiert.

Das Sesame-System besitzt standardmäßig (z.B.) bereits Sail-Implementierungen für die Verwendung des Hauptspeichers (*MemoryStore*), von Festplatten (*NativeStore*) und angeschlossener relationaler Datenbanken (*RDBMSStore*), auf die der Entwickler zurückgreifen kann.

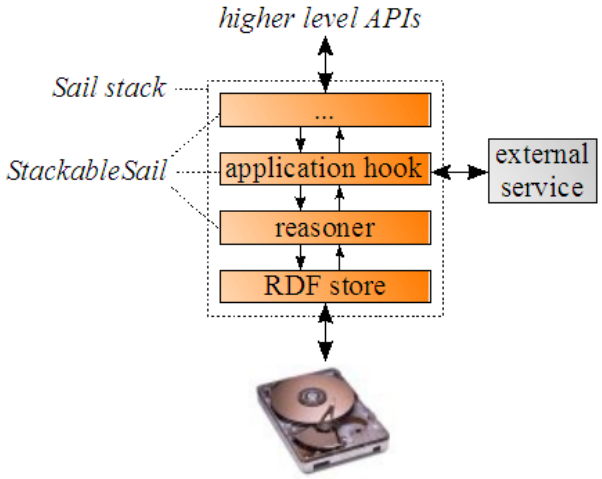

Abbildung 3.2.: Schaubild eines Sail-Stacks (Quelle:[Adua]).

Eine nennenswerte Besonderheit der Sesame-Sails ist schließlich, dass Objekte einer Unterklasse von Sail, die sogenannten *StackableSails*, gestapelt werden können (vgl. [BKvH02],[Adua]). Das heißt, dass das oberste Sail eines solchen Sail-Stacks einen Teil der Methodenaufrufe eines Repositories selbst verarbeitet, während es andere Aufrufe an das ihm unterstellte Sail weiterleiten kann. Dieses kann die von oben durchgereichten Aufrufe dann verarbeiten oder seinerseits weiter nach unten delegieren – usw.. Eine wichtige Anwendung dieses Mechanismus findet sich bei der Verarbeitung von ontologischen Daten in Sesame [BKvH02]. Dazu existiert eine spezielle Sail-Klasse, die schematische Daten in einem RDF-Store ausliest und in den Hauptspeicher lädt. Abfragen über Schemata, die von Repository-Objekten an dieses Sail gestellt werden, werden auch von diesem verarbeitet, alle anderen Anfragen werden an den unter ihm liegenden Sail-Stack übergeben.

3.3. Verarbeitung von Abfragen

Die wichtigste Art des Datenzugriffs in Sesame wird durch die Übergabe und Auswertung von Datenabfragen (Queries) realisiert (vgl. [BKvH02]). Dazu fordert eine entsprechende Awendung die Instanz einer Verbindung (*RepositoryConnection*) zu dem Repository an (siehe Beispiel 3.4). Die beispielsweise in SPARQL formulierten Ausdrücke werden anschließend einer Methode dieses Objekts übergeben. Der Anwendungsentwickler muss angeben, in welcher Sprache er seine Anfragen formuliert. Daraufhin wird eine an das Repository übergebene Anfrage von einer entsprechenden Query-Engine geparst und dabei in eine Datenstruktur übersetzt. Für diese Repräsentation von RDF-Anfragen ist in der Sesame-Architektur das angesprochene Query Model konzipiert.

Das Query Model von Sesame umfasst Klassen, die die verschiedenen üblichen Anfrageoperatoren, wie *Projection*, *Join*, *Union* usw., implementieren (vgl. [Adua]). Diese haben je nach Stelligkeit ein oder zwei Argumente. Die Argumente dieser Operatoren sind Implementionen von *TupleExpr*. Dies ist eine Schnittstelle, die auch von der *StatementPattern*-Klasse implementiert wird, die ebenfalls Teil des Query Model ist. StatementPattern-Instanzen repräsentieren dabei die einfachen Graph-Muster, wie man sie von SPARQL kennt (vgl. Abschnitt 2.1.4). Allerdings haben sie, wie vom Namen bereits angedeutet, nur einzelne RDF-Statements als Lösung– sie vertreten insofern genauer betrachtet nur Statement- bzw. Tripel-Muster. Graph-Muster (im eigentlichen Sinne) in zu parsenden Anfragen müssen somit von mehreren StatementPattern-Objekten im Query-Baum modelliert werden. Deren Ergebnisse, die für jedes der StatementPattern-Objekte einzeln zusammengetragen werden, werden schließlich gejoint.

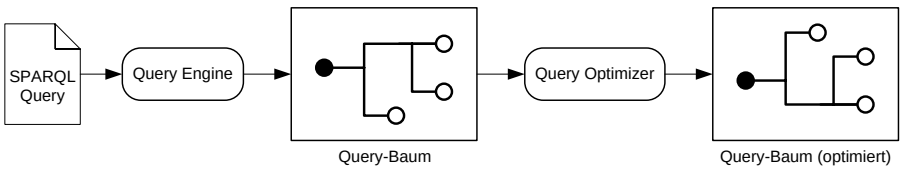

Abbildung 3.3.: Ablaufschema beim Parsen einer Query.

Durch das einheitliche TupleExpr-Interface lassen sich Operator- und StatementPattern-Objekte schachteln. Das heißt, Operator-Knoten können auf andere Operator- oder StatementPattern-Instanzen als Argumente zeigen, wodurch insgesamt ein Query-Baum gebildet werden kann. Ein solcher wird von den Query-Engines durch Parsen eines Anfrage-Ausdrucks geliefert.

Geparste Queries lassen sich nachträglich nach verschiedenen Heuristiken optimieren. Den dazu verwendeten *Query-Optimizern* wird ein konkreter Query-Baum übergeben. Die Optimizer transformieren den Baum rein strukturell in eine nach der jeweiligen Heuristik optimierte Form. Sesame stellt standardmäßig eine Reihe verschiedener dieser Query-Optimizer zur Verfügung, die vom Entwickler (optional) auf eine geparste Query angewendet werden können.

Die eigentliche Auswertung einer Anfrage findet bei Sail-Repositories, wie oben erwähnt, durch das verwendete Sail statt. Nachdem die dem Repository übergebene Anfrage geparst wurde, wird der gelieferte Query-Baum an das Sail-Objekt übergeben. Der Query-Baum wird daraufhin in einer Evaluierungsmethode des Sails traversiert, und zwar (grob gesagt) von oben nach unten und links nach rechts. Die einzelnen Baumknoten, d.h. Operator- und StatementPattern-Instanzen, werden dabei nacheinander abgearbeitet und jeweils ausgewertet. Die dabei entstehenden Teilergebnisse werden sukzessive zusammengetragen, bis sich nach Traversierung des gesamten Baums ein Endergebnis ergibt.

Bei Auswertung einzelner StatementPattern-Objekte findet jeweils ein Vergleich mit den Tripeln des RDF-Store statt, wobei das Sail auf das zugrunde liegende Speichermedium zugreift, um diese zu laden. Die Variablen eines Statement-Musters, das bei der Auswertung in Betracht gezogen wird, werden jeweils durch alle RDF-Tripel, die Lösungen dieses Musters sind, belegt. Dadurch entsteht eine Ergebnistabelle wie in Tabelle 2.1, die in einer bestimmten Datenstruktur abgelegt wird.

Aus Gründen der Speichereffizienz, insbesondere bei sehr großen Datenbeständen, wird der Zugriff auf diese Datenstruktur durch Iteratoren realisiert, mit denen die einzelnen Zeilen einer Ergebnistabelle nacheinander abgerufen werden können [Adua]. Joins mehrerer solcher Tabellen können mit Hilfe dieses Mechanismus ebenfalls einfach realisiert werden. Dies wird in Sesame durch sog. *JoinIterator*-Objekte gehandhabt, die sich selbst zweier Iteratoren (der Ergebnis-Iteratoren der beiden Join-Argumente) bedienen.

Zu Beginn der Join-Auswertung liegt immer nur das vollständige Ergebnis des linken Join-Arguments vor. Der Join-Iterator iteriert (falls angefordert) über dieses Ergebnis und fordert dabei für jede einzelne Lösung das von dieser abhängige Ergebnis des anderen Join-Arguments (Ergebnis-Iteration eines StatementPattern-Objekts) an. Die Lösungen jedes der dabei gelieferten Ergebnisse sind dann wiederum auch Lösungen des Joins. Bei Iteration über den Join-Iterator werden immer erst diese einzelnen Teilergebnisse des Joins erschöpft, bis entweder eine erneute Auswertung auf Seite des rechten Join-Arguments (über die nächste Lösung des linken Argumentes) angefordert werden muss oder der Join gänzlich zum Abschluss gekommen ist. Ein auf diese Weise hergestellter Join zweier Teilergebnisse kann selbst als (Teil-)Ergebnis eines Arguments eines Join-Operator-Knotens in einem Query-Baum weiterverwendet werden. Für die

```
...

//Anfordern einer Repository-Verbindung
RepositoryConnection connection =  repository.getConnection();

String query = "SELECT * WHERE { ... }";

//Parsen der Query
TupleQuery tupleQuery =
    connection.prepareTupleQuery(QueryLanguage.SPARQL, query);

//Anfordern der Ergebnisse (Iterator-Instanz)
TupleQueryResult result = tupleQuery.evaluate();

while (result.hasNext()) {
   //Lösung =
   BindingSet bs = result.next();

   ...
}
```

Abbildung 3.4.: Beispielhaftes Code-Fragment für eine Repository-Anfrage aus Sicht des Anwendungsentwicklers.

Auswertung von Query-Knoten anderer Operatoren existieren bei Sesame ebenfalls entsprechende Iterator-Klassen.

4. Lösungskonzept und Implementierung

4.1. Überblick

Im Rahmen dieser Studienarbeit soll ein Sesame-Sail, das Webservice-Sail, entwickelt werden, das verschiedene Webservices über deren Schnittstellen abrufen und die gelieferten Ergebnisse so transformieren kann, dass diese im Sesame-System weiterverarbeitet werden können. Das Webservice-Sail soll also die Abbildung einer SPARQL-Abfrage auf die Schnittstelle eines (nahezu) beliebigen Webservice realisieren.

Da unmöglich alle der vielen Ausprägungen von Webservices und deren Technologien hier abgedeckt werden können, verständigte man sich bei der Konzeption der Lösung auf bestimmte Einschränkungen. Die Webservices, die vom Webservice-Sail bedient werden sollen, müssen so folgenden Voraussetzungen genügen, die der softwaretechnischen Handhabung zu Gute kommen:

- Aufrufe der Schnittstellen-Methoden werden mit XML-Dokumenten beantwortet, die die Ergebnisse repräsentieren
- der jeweilige Webservice ist entweder RESTful oder kommuniziert über SOAP
- einzelne Methoden dienen entweder der Ressourcen-Suche nach mehreren Kriterien oder der direkten Identifikation einer Ressource im Datenbestand
- Aufrufe von Suchmethoden erfordern die Belegung (i.d.R.) mehrerer Parameter, die optional und (i.d.R.) unabhängig voneinander sind
- Aufrufe von Methoden zur Identifikation erfordern die Belegung genau eines Parameters (der eine eindeutige Kennung der jeweiligen Ressource aufnimmt)

Darüber hinaus sollen Methoden noch statisch zu belegende Parameter besitzen können, die dann für jeden Aufruf einer Methode den gleichen Wert enthalten. Die Belegung einzelner statischer Parameter soll bereits in den Methoden-Definitionen der jeweiligen Konfiguration vorgenommen werden können. Mehr dazu im folgenden Abschnitt.

Auf der Prozessebene muss sich das Webservice-Sail dabei an dem in Abschnitt 3.3 dargestellten Auswertungsvorgang orientieren. Das Sail würde dann wie folgt vorgehen (vgl. Abbildung 4.1):

1. Das Sail wird instanziiert und die zugehörige Konfiguration geladen
2. Eine Abfrage über den Service wird entgegengenommen
3. Die Abfrage wird geparst
4. Einzelne StatementPattern-Objekte des Baums werden zu Webservice-Aufrufen herangezogen
5. Die Aufrufe werden getätigt und die Antworten jeweils entgegengenommen
6. Die einzelnen Antworten werden ausgewertet und zu einer Ergebnistabelle umgewandelt

Dieser Prozess beinhaltet zwei grundlegende Teilvorgänge, nämlich die *Query-Übersetzung*, also das Parsen einer Query und die anschließende Abbildung auf einzelne Webservice-Calls, und das *Response-Mapping*, die Auswertung der zurückgegebenen Antwort-Dokumente.

Im konkreten Zusammenhang bleibt zu klären, was überhaupt gültige RDF-Anfragen an das Webservice-Sail sind. Anfragen über den Datenbestand eines Webservice beziehen sich nämlich auf ein Schema, das so explizit tatsächlich ersteinmal nicht existiert. Vielmehr muss der Anwender den Webservice, den er über das Sail abfragen möchte, zunächst selbst (zumindest informell) modellieren. Dieses Domänen-Modell kann dann als Grundlage der Sail-Konfiguration dienen, die dem Sail zur Verfügung gestellt werden muss und in der die notwendigen Informationen für die oben genannten Prozesse gespeichert werden. Es können dann nur solche Prädikate in einer Query verwendet werden, die auch in der Konfiguration angegeben sind. Näheres dazu aber im Folgenden.

4.2. Konfigurationssprache

Die wichtigste Aufgabe bei der Konzeption der Lösung, die in dieser Studienarbeit erarbeitet werden sollte, war die Ausarbeitung einer Konfigurationssprache. Diese sollte ausdrucksstark genug sein, um das zu entwickelnde Sail in die Lage zu versetzen, möglichst viele Webservices abrufen zu können. Dabei taten sich einige (z.T. entgegengesetzte) Erfordernissen auf, die jeweils berücksichtigt werden mussten.

So sollen Konfigurationen in der Sprache

- einfach und menschenlesbar aufzuschreiben sein,
- möglichst konzis zu formulieren sein,

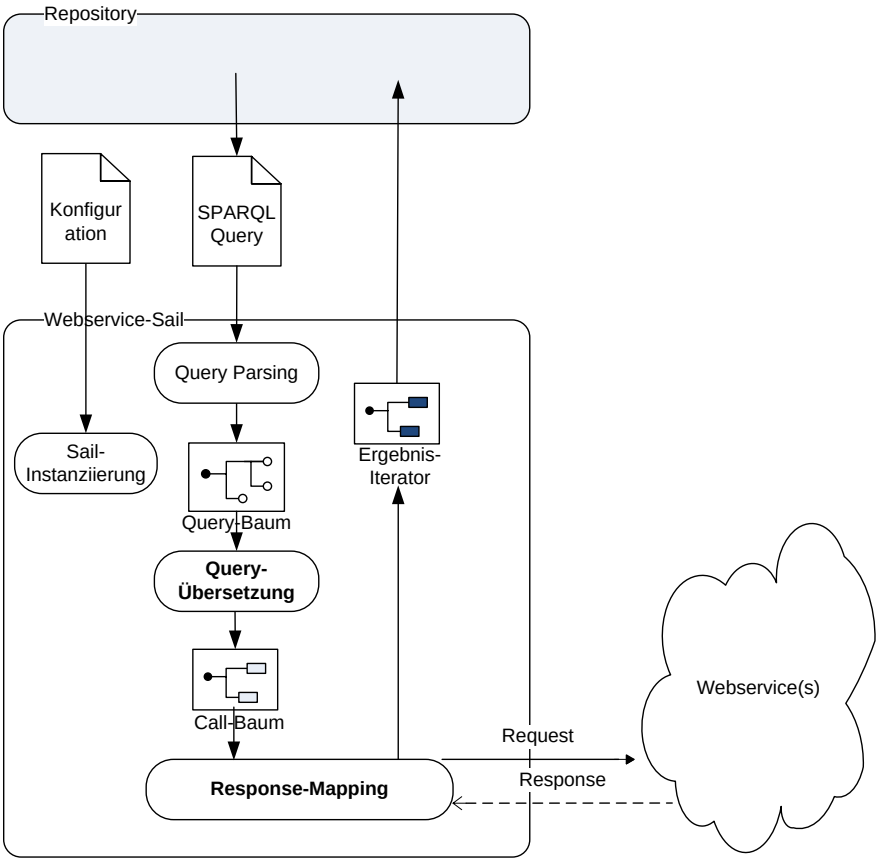

Abbildung 4.1.: Schematische Darstellung des erforderlichen Prozessablaufs im Webservice-Sail.

- möglichst viele Webservices abbilden können,
- einfach maschinell zu verarbeiten sein.

Die Konfigurationssprache[1] ist aus praktischen Gründen auf RDF basiert. Damit lässt sich eine konkrete Sail-Konfiguration mit Mitteln des RIO-Moduls des Sesame-Frameworks einlesen. Die Konfigurationssprache wird durch ein RDF-Vokabular gebildet, dessen Namensraum jedem Webservice-Sail bekannt ist. Eine Sail-Konfiguration ist dann ein Graph, bestehend aus zwei Hauptknoten, dem *Interface-* und dem *Schema-Knoten* (vgl. Abbildung 4.3).

Notiert werden Eigenschaften des Interface - aber auch des Schemas - jeweils durch Graph-Kanten. Die jeweiligen Prädikate dieser Kanten bilden insgesamt das Vokabular der Konfigurationssprache. Zusätzlich gehören aber auch bestimmte konstante Kno-

[1] Detailangaben zur Konfigurationssprache sind in Anhang A zu finden.

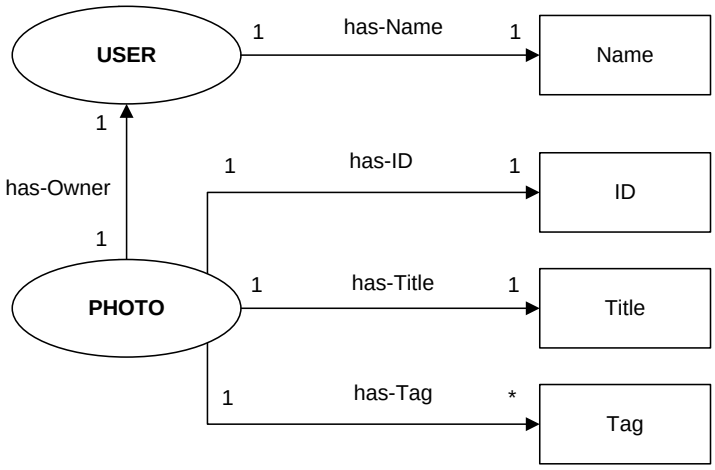

Abbildung 4.2.: Einfaches Domänen-Modell eines Beispiel-Webservice. In Ellipsen eingetragene Namen bezeichnen Ressourcen des Modells, nur diese sollen durch URIs repräsentiert werden. Die Kardinalitäten gelten jeweils nur für die Pfeilrichtungen.

tenwerte zu diesem Vokabular. Dazu gehören beispielsweise die Bezeichner der Kommunikationswege, aber auch die von Interface- und Schema-Knoten selbst.

Im Subgraph des Interface wird die Schnittstelle des Webservice beschrieben. Das beinhaltet die Beschreibung einzelner *Methoden*, die später durch das Sail aufgerufen werden sollen. Ebenso lassen sich Methoden-übergreifende Eigenschaften definieren, die direkt dem Interface-Knoten zugehörig sind.

Das Interface besitzt (u.a.) die folgenden Eigenschaften/Objekte:

- die einzelnen Methoden der Webservice-Schnittstelle
- die Art der Kommunikation des Webservice, also *REST* oder *SOAP*
- optional: Anzahl möglicher Neuverbindungen (*retryCount*)
- wenn nötig: *API-Nutzungskennung* für Schnittstellenaufrufe
- wenn nötig: zentrale Einstellungen bei SOAP
- wenn nötig: *XML-Namensräume* für das Response-Mapping

Die Methoden des Interface wiederum besitzen ihre eigenen lokalen Eigenschaften. Hierzu sind insbesondere *statische Parameter* und die (Basis-)*URL* bei REST zu zählen. Außerdem sind noch zwei Besonderheiten zu nennen. Bei vielen Webservices werden Suchergebnisse auf eine ganze Reihe von Dokumenten aufgeteilt, die jeweils einen

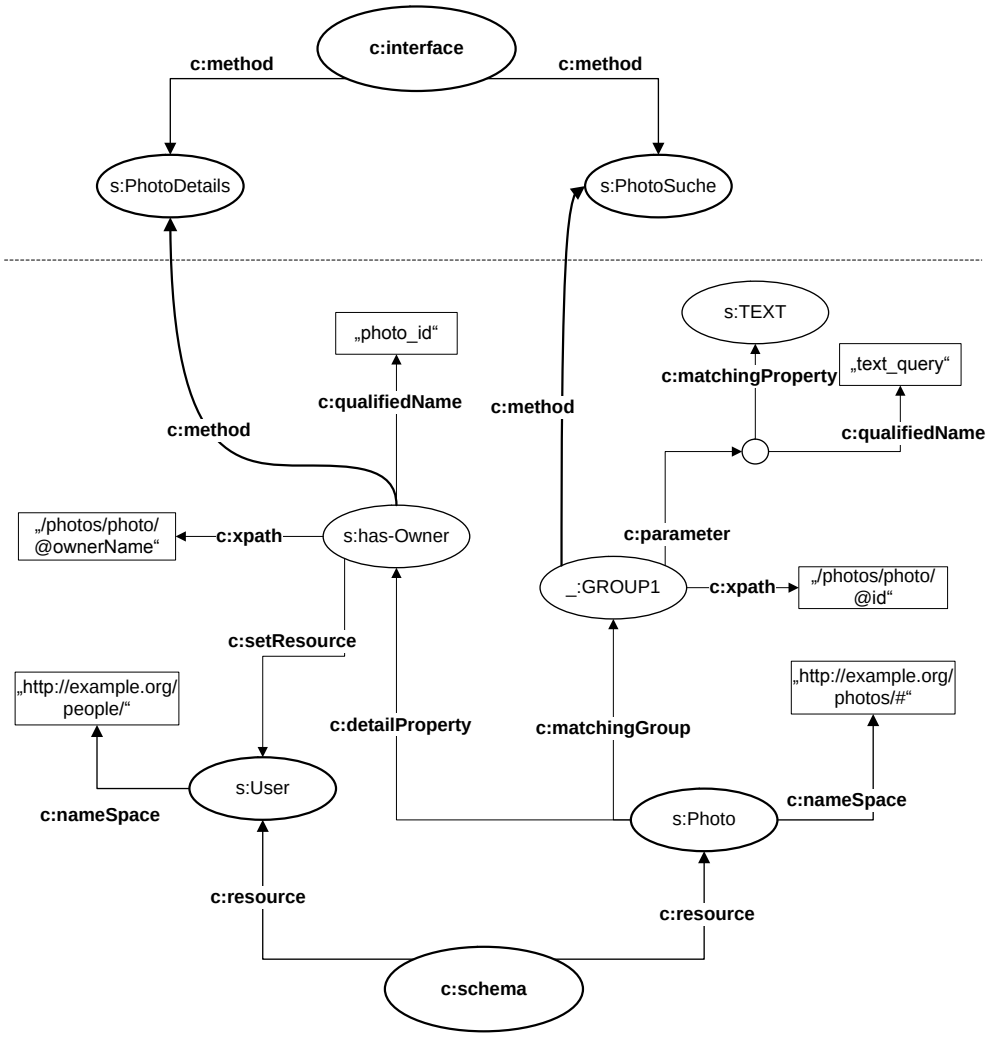

Abbildung 4.3.: Mögliche Konfiguration eines Webservice mit Modell aus 4.2.

eigenen Aufruf benötigen. Eine Methode in der Konfiguration kann hierzu *Paginations*-Einstellungen erhalten (siehe Anhang A), mit denen automatisch mehrere solcher Dokumente nebenläufig angefordert werden können.

Auch problematisch bei solchen Suchergebnissen ist, dass eben mehrere Webservice-Ressourcen in einem Dokument aufgelistet sind. Die Methode erhält hierfür einen *Basis-XPath*, mit dem diese Ressourcen vorab einzeln ausgewählt und dann weiterverarbeitet werden können (siehe Abschnitt 4.4). Für die spätere interne Identifikation der Methoden im Webservice-Sail werden die URIs der einzelnen Methoden-Knoten des Interface-Graphen herangezogen.

Im Schema-Subgraph einer Konfiguration werden nun die Objekte eines Webservice-Modells (wie in Beispiel 4.2) und deren Eigenschaftsbeziehungen auf die Bestandteile des Interface abgebildet. Der Schema-Knoten selbst zeigt auf eine Reihe von Modell-Objekten, die in der Konfiguration jeweils durch *Resources* abgebildet werden. Jede dieser Resources wiederum besitzt also Eigenschaften gemäß des Modells. Eine solche Eigenschaft kann auch eine Beziehung zwischen zwei solcher Resources sein. Die einzelnen Eigenschaftsbeziehungen werden in der Konfigurationssprache durch *Properties* modelliert, die in einer konkreten Konfiguration, ebenso wie die einzelnen Resources, durch URIs identifiziert werden. An dieser Stelle werden zwei Arten von Properties unterschieden. Die eigentlichen Eigenschaften einer Resource gemäß des Domänen-Modells des Webservice werden in der Konfigurationssprache durch *Detail-Properties* abgebildet.

Je einer dieser Detail-Properties wird nun im Allgemeinen ein Tripel zugeordnet, bestehend aus:

- genau einer Methode des Interface
- genau einem der *Parameter* der Methode
- und einem *XPath-Ausdruck*

Offenbar dienen die Detail-Properties der Extraktion von Bestandteilen aus einer Webservice-Response, die vom Aufruf einer Methode (Detail-Methode) geliefert wird. Dazu sind eben genau diese drei Angaben vonnöten. Allerdings müssen die Methodenaufrufe, auf die sich Detail-Properties beziehen, erst gebildet, die Methode bzw. deren Parameter dazu instanziiert werden. Da sich jede Detail-Property auf Instanzen einer Resource beziehen, müssen diese vor Aufruf der jeweiligen Detail-Methoden bereits vorliegen.

Dies gelingt durch die andere Art von Property, die *Matching-Property*. Properties dieser Art werden später in Queries prinzipiell wie Resource-Eigenschaften behandelt werden. Sie dienen aber nur dem technischen Zweck Aufrufe von Suchmethoden zu ermöglichen, die Instanzen für die zugehörige Resource liefern (matchen). Jede Resource kann eine (Matching-)Gruppe solcher Matching-Properties besitzen. Die Properties

einer gemeinsamen Matching-Gruppe dienen dazu, einzelne Parameter ein und derselben Suchmethode zu belegen. Ihnen werden also ebenfalls Parameter-Bezeichner für eine Methode zugeordnet. Für jede solche *Matching-Gruppe* muss zusätzlich noch ein XPath-Ausdruck angegeben werden. Mit diesem lassen sich dann Werte (die IDs der gesuchten Resource-Instanzen) aus den Antworten der Suchmethoden-Aufrufe extrahieren.

Vom Sprachvokabular sind die Bezeichner (URIs) der Methoden, Resources und Properties zu trennen. Diese sind Webservice-spezifisch und werden über den Namespace-Mechanismus von der Konfigurationssprache unterschieden. Die Bezeichner der einzelnen Properties können jeweils als Prädikate in einzelnen Statement-Mustern einer Query verwendet werden. In den folgenden Abschnitten wird besprochen, wie eine solche Query dann ausgewertet werden kann.

4.3. Query-Übersetzung

Die Query-Übersetzung bildet die Bestandteile, genauer die Statement-Muster, einer Query auf einzelne, möglicherweise voneinander abhängige Webservice-Aufrufe ab. Eine Anfrage wird dabei zunächst im Sesame-Framework geparst und in einen Query-Baum umgewandelt. Dieser wird anschließend zur Vorbereitung der Auswertung traversiert. Dabei muss das Webservice-Sail nun die einzelnen StatementPattern-Knoten verarbeiten. Für Queries an das Webservice-Sail wird davon ausgegangen, dass jedes Statement-Muster eine instanziierten Prädikat-Variable besitzt. Die Prädikate einzelner Statement-Muster werden dann dazu verwendet die besagten Properties in einer Konfiguration zu identifizieren. Somit handelt es sich bei dem Subjekt eines Statement-Musters automatisch um eine der angesprochenen Resources aus dem Schema der Konfiguration.

An die Struktur von Queries werden (auch aus Komplexitätsgründen) an dieser Stelle folgende Anforderungen gestellt:

- alle Statement-Muster besitzen eine unbelegte Subjekt-Variable, diese ist für alle Statement-Muster der Query dieselbe
- es müssen Statement-Muster vorhanden sein, deren Prädikate auf Matching-Properties verweisen
- Statement-Muster, die auf Matching-Properties verweisen, müssen Literale anstelle der jeweiligen Statement-Objekte aufweisen
- Statement-Muster, die auf Detail-Properties verweisen, müssen unbelegte Objekt-Variablen besitzen

- alle Matching-Properties, die durch Prädikate der Query identifiziert werden, gehören zur selben Matching-Gruppe (und zeigen damit auf dieselbe Methode)

Anders ausgedrückt heißt das, dass mit einer Query für die Subjekt-Variablen nur Lösungen eines einzigen Ressourcentyps aus der Konfiguration geliefert werden dürfen. Dazu wird durch die Matching-Properties (bzw. die zugehörigen Statement-Muster) genau ein Methoden-Aufruf zur Suche (Matching) von Instanzen dieser Ressource instanziiert. Durch die Detail-Properties ist es im Übrigen möglich, Instanzen weiterer Ressourcentypen zu liefern, die jedoch den jeweiligen Objekt-Variablen zugehörig sind. Dabei müssen sich auch diese Detail-Properties sämtlich auf ein und dieselbe Subjekt-Variable beziehen. Detail-Properties dienen also nur dem Zweck Lösungen für Objekt-Variablen herzustellen.

```
PREFIX s: <http://example.org/schema/>
SELECT  ?person ?photo
WHERE  {
        ?photo s:TEXT "2004";
               s:has-Owner ?person.
       }
```

Abbildung 4.4.: Beispielhafte Query, gestellt an ein Webservice-Sail mit der Konfiguration aus 4.3.

Um eine Webservice-Query wie in Abbildung 4.4 nun in einen Baum aufeinander aufbauender Methoden-Aufrufe (Calls) umzuwandeln, soll das Sail wie folgt vorgehen:

1. die Query wird wie gewohnt zu einem Query-Baum geparst
2. der Query-Baum wird traversiert, dabei werden die einzelnen Knoten (die StatementPattern-Objekte) besucht:
   - das Statement-Muster $sm = (sv, pv, ov)$ wird besucht[2]:
     a) die zu $val(pv)$ gehörige Property p in der Konfiguration wird geliefert
     b) die zu p gehörige Methode m wird geliefert
     c) der zu p gehörige XPath xp wird geliefert
     d) Fall 1: p ist eine Matching-Property:
        i. wurde durch vorherige Matching-Properties ein Aufruf von m vorbereitet und im Query-Baum eingetragen, so wird nun der fehlende Parameter dieses Aufrufs mit $val(ov)$ belegt – sm wird aus dem Query-Baum entfernt.

[2]Es ist zu beachten, dass es sich bei sv, pv, ov um (möglicherweise belegte) Variablen handelt, aus denen das Statement-Muster gebildet wird; der Wert einer Variablen v werde durch $val(v)$ symbolisiert

ii. existiert kein solcher Aufruf, so wird dieser hergestellt und mit $val(ov)$ teilinstanziiert. Dem Aufruf wird das Paar (sv, xp) zugeordnet, der Aufruf in einem speziellen Call-Knoten referenziert, der anstelle von sm in den Query-Baum eingetragen wird

e) Fall 2: p ist Detail-Property:

i. der zu p gehörige Parameter-Bezeichner $param$ wird geliefert

ii. existiert im Query-Baum ein Call-Knoten, der auf einen Aufruf von m zeigt, so wird diesem zusätzlich das Paar (ov, xp) zugeordnet – sm wird aus dem Query-Baum entfernt

iii. existiert noch kein solcher Call-Knoten, so wird ein neuer mit einem Aufruf von m anstelle von sm eingetragen. Dem Aufruf werden die Paare (ov, xp) und $(sv, param)$ zugeordnet

An die einzelnen Methodenaufrufe werden also neben Parameter-Werten noch andere Daten übergeben. Diese werden für die anschließende Auswertung im Response-Mapping benötigt. Durch die sukzessive Ersetzung der StatementPattern-Knoten im ursprünglichen Query-Baum, entsteht ein Query-Baum, der einzelne Methoden-Aufrufe referenziert, ein *Call-Baum*.

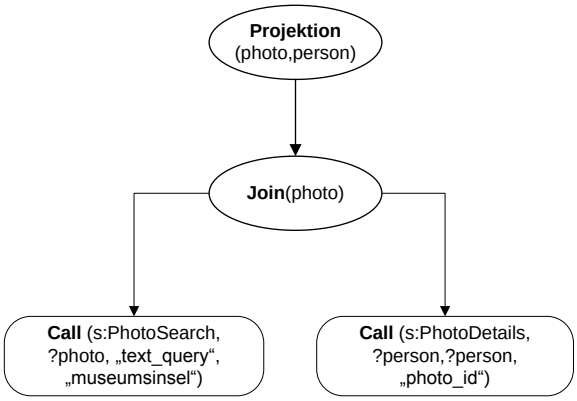

Abbildung 4.5.: Call-Baum nach Query-Auflösung der Beispiel-Query gemäß 4.3.

Eine besondere Rolle kommt den Joins zwischen den einzelnen StatementPattern-Knoten eines Query-Baums zu. Diese lassen sich, wie gewohnt, über das Query-Modell, also über Join-Iteratoren, mechanisch herstellen (vgl. Abschnitt 3.3).

Eine Alternative, die sich nun anbietet und die in obigem Algorithmus wenn möglich benutzt wird, ist die Bündelung von mehreren Statement-Mustern über die jeweils referenzierten Properties zu einem Methodenaufruf. Bei den Matching-Properties geschieht das ja ohnehin. Bei Detail-Properties ist dies jedoch nur dann möglich, wenn

sie sich auf dieselbe Methode wie die Matching-Properties oder mindestens eine andere bereits abgearbeitete Detail-Property beziehen. In Fällen, in denen das der Fall ist, wird der Join letztlich über die Auswertung desjenigen Dokuments realisiert, das als Ergebnis für den Aufruf der gemeinsamen Webservice-Methode zurückgegeben wird (vgl. Abschnitt 4.4).

In den Fällen in denen das nicht möglich ist, hängen die zu den betreffenden Detail-Properties gehörenden Methoden-Aufrufe von Teilen der bisherigen Lösungen ab. Mit den Voraussetzungen in 4.3 heißt das, dass (je genau ein) Parameter der Aufrufe mit den Lösungen für die Subjekt-Variable der Query belegt werden muss (Näheres dazu im Folgenden). Die Ergebnisse werden dann mit den Join-Iteratoren über diese Subjekt-Variable gejoint.

4.4. Response-Mapping

Im Response-Mapping sollen die Dokumente, die jeweils als Antwort (Response) auf einen Webservice-Aufruf empfangen werden, zu solchen Ergebnistabellen verarbeitet werden, die jeweils die Lösungen der zugehörigen Graph-Muster der Query enthalten (vgl. 2.1.4). Bestimmte Elemente einer Response müssen also extrahiert und entsprechend aufbereitet werden. Dabei werden diese Elemente jeweils als Lösungen bestimmten Query-Variablen zugeordnet.

Wie bereits angedeutet, werden für die Studienarbeit nur solche Webservices in Betracht gezogen, die ihre Ergebnisse in XML repräsentieren. Dadurch lässt sich die Extraktion von relevanten Bestandteilen aus einer Service-Response systematisch implementieren. Für die vorliegende Lösung wurde entschieden, diese Extraktion direkt über XQuery zu realisieren.

Es bieten sich einige Alternativen hierzu an. Dazu sind unter anderem XML-Datenbindung[3] und die Sprache XSPARQL[4] zu zählen. Bei ersterem handelt es sich um den Vorgang aus XML-Dokumente mit Hilfe der Schema-Definition Elemente zu extrahieren und direkt auf softwaretechnische Objekte abzubilden, die dann für die Weiterverarbeitung zur Verfügung stehen. XSPARQL hingegen ist eine Sprache, die SPARQL und XML so verknüpft, dass man SPARQL-Anfragen direkt auf XML-Dokumente anwenden kann.

Beide Ansätze erfordern jedoch selbst so viele Einstellungen, dass eine Sail-Konfiguration dadurch schnell unlesbar geraten würde. Zudem basieren zumindest XSPARQL-Implementationen ebenfalls auf XQuery. Die Annahme liegt daher nahe, dass das Response-Mapping effizienter, mit weniger Overhead direkt mit XQuery zu realisieren

[3] Hierzu stellt Java Architecture for XML Binding (JAXB) entsprechende Mittel zur Verfügung, siehe `https://jaxb.dev.java.net/`

[4] `http://xsparql.deri.org/spec/`

ist. Zu diesem Zweck dienen auch die XML-Einstellungen in der Sail-Konfiguration, die weiter oben bereits angesprochen wurden.

Eine konkrete Query, die erfolgreich geparst und in der Query-Übersetzung in einen Call-Baum überführt werden konnte, wird mit dem Response-Mapping nun wie folgt ausgewertet:

- das Webservice-Sail traversiert den Call-Baum, die einzelnen Bestandteile werden nacheinander abgearbeitet (vgl. 3.3):
  - der Call-Knoten $cnode$ mit Methodenaufruf c wird besucht
    1. falls c noch nicht voll instanziiert ist, also noch einen Parameter-Wert erwartet, so wird dieser Wert aus bisherigen Lösungen für die Subjekt-Variable der Query bezogen[5]
    2. c ist (nun) instanziiert und wird an den Webservice gerichtet
    3. das XML-Dokument der Repsonse von c wird entgegengenommen
    4. alle c zugeordneten Variablenbezeichner-XPath-Paare der Form (v_i, xp_i) werden verarbeitet:
       a) die Menge alle XPath-Ausdrücke wird zu einer XQuery-Anfrage über das Response-Dokument verarbeitet, in der ein Kreuzprodukt $\{xp_1\} \times \{xp_2\} \times \ldots \times \{xp_n\}$[6] über alle selektierten XML-Elemente hergestellt wird.
       b) Jedes Element-Tupel $(x_1, x_2, \ldots, x_n)$ des Kreuzproduktes wird zu einer Lösung der Form $(v_1 = x_1, v_2 = x_2, ..., v_n = x_n)$ umgewandelt
       c) Ein Iterator, der über alle diese Lösungen iteriert, wird dem auf $cnode$ zeigenden Join-Iterator für den Join mit den bisherigen Ergebnissen zurückgegeben

Methoden, die ganze Listen von einzelnen Ressourcen liefern, werden, wie gesagt, in der Sail-Konfiguration Basis-XPath-Ausdrücke zugewiesen. Das sind solche XPaths, die genau diese Ressourcen (repräsentiert durch XML-Elemente) im XML-Dokument der Response selektieren. Alle XPath-Angaben, die in der Query-Übersetzung zur Daten-Extraktion einem Methoden-Aufruf zugeordnet sind, beziehen sich in einem solchem

[5] Für diesen Zweck wird entsprechenden Methodenaufrufen in 4.3 der Bezeichner der Subjekt-Variable in Verbindung mit dem Bezeichner des betreffenden Methoden-Parameters zugeordnet. Es ist bei Formulierung der Query sicherzustellen, dass bei Auswertung solcher Call-Knoten bereits alle Lösungen für die Subjekt-Variable existieren.

[6] Der Ausdruck $\{xp\}$ steht hier für die Menge der durch xp selektierten XML-Elemente.

person	photo
http://example.org/people/peter	http://example.org/photos/#56
http://example.org/people/caroline	http://example.org/photos/#475
http://example.org/people/joan	http://example.org/photos/#12
http://example.org/people/joan	http://example.org/photos/#189
...	

Tabelle 4.1.: Mögliche Ausgabe nach Auswertung von Call-Baum 4.5.

Fall auf jeweils ein solches, vorab ausgewähltes XML-Element. Dadurch wird verhindert, dass sich das durch die XQuery-Anfrage hergestellte Kreuzprodukt über das gesamte Dokument erstreckt (also über die geforderten Elemente aller Ressourcen). Somit wird das Kreuzprodukt also lokal für jedes Ressourcen-Element separat hergestellt.

4.5. Implementierung – SPARQL2ws

Für die softwaretechnische Umsetzung der gerade erläuterten Vorgänge wurde ein Java-Projekt namens *SPARQL2ws* erstellt, in dem folgende Bestandteile entsprechend implementiert werden mussten:

- das Webservice-Sail als Spezialisierung des Sesame-Sails mit zugehöriger Auswertungs-Strategie
- die Query-Übersetzung
- das Response-Mapping auf Basis von XQuery
- ein Modul zum Laden der Webservice-Sail-Konfigurationen
- eine Repräsentation für Konfigurationen bzw. deren Bestandteile
- ein Modell für die Abbildung von Webservices bzw. deren Methoden

Das Webservice-Sail wurde in SPARQL2ws durch die Sail-Spezialisierung *WSSail* realisiert. Das WSSail sorgt dafür, dass nach Parsen einer Query durch die Mittel von Sesame automatisch der Algorithmus der Query-Übersetzung auf den entstandenen Query-Baum angewendet wird. Nach oder bei der Generierung eines WSSail-Objektes wird immer die Übergabe eines URL erwartet, der den Aufenthaltsort der zugehörigen Sail-Konfigurationsdatei lokalisiert.

Für das Laden angelegter WSSail-Konfigurationen existiert in SPARQL2ws die *WSSail-ConfigurationLoader*-Klasse. Diese sorgt für das Parsen der eingebundenen Konfiguration. Geparst werden können (z.Z.) in dieser Implementation nur Konfigurationsdateien, die in Turtle ausgedrückt sind. Die von einem konkreten WSSail verwendete

WSSailConfigurationLoader-Instanz legt für dieses WSSail dann Repräsentationen der Konfiguration an, auf die das WSSail in der Folge zugreifen kann.

Diese Repräsentation geschieht einerseits durch Klassen, die Bestandteile des Schema-Teils einer Konfiguration abbilden. In SPARQL2ws existiert dafür eine *Schema*-Klasse, deren einzige Instanz neben eigenen Informationen Referenzen zu Objekten der Klassen *MatchingProperty* und *DetailProperty* aufnehmen kann. Letztere verkörpern offenbar die verschiedenen Properties einer Sail-Konfiguration und werden bei der Query-Übersetzung vom WSSail-Objekt herangezogen.

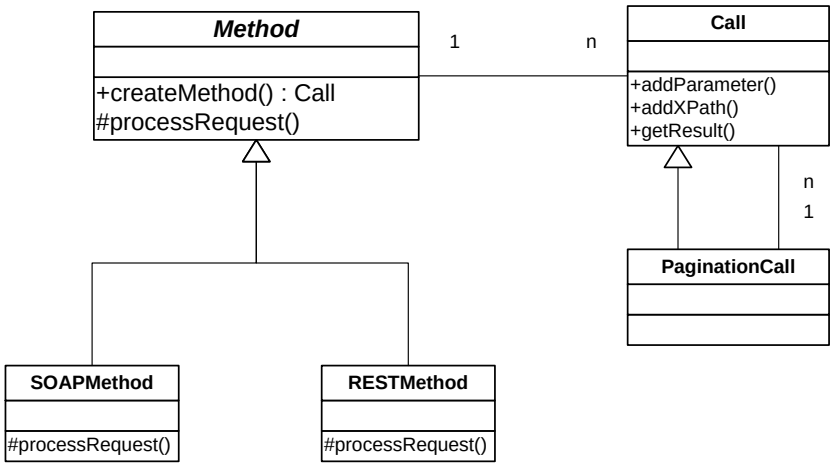

Abbildung 4.6.: Webservice-Modell von SPARQL2ws.

Für die Abbildung der in einer Konfiguration definierten Webservice-Schnittstelle andererseits besitzt SPARQL2ws ein Webservice-Modell (siehe 4.6), in dem entsprechende Klassen angelegt sind. Dazu gehören die abstrakte Klasse *Method* sowie die *Call*-Klasse. Method repräsentiert dabei eine Webservice-Methode, wie sie im Interface-Teil einer Sail-Konfiguration vorzufinden ist, bzw. deren Definition. Method sorgt auch für die Generierung ihr zugehöriger Aufrufe, die dabei durch *Call*-Instanzen repräsentiert werden. Spezialisierungen von Method sind transportspezifisch, besitzen somit das Wissen, wie eine Call-Instanz dieser Methode an den Webservice zu richten und die erwartete Antwort zu empfangen ist. Das Webservice-Paket von SPARQL2ws besitzt dabei zusätzlich die Klassen *RESTMethod* und *SOAPMethod*, die von Method abgeleitet sind und den Zugang zu RESTful bzw. SOAP-basierten Webservice-Schnittstellen ermöglichen. Im Falle von SOAP wird dabei auf Java-eigene Mittel zurückgegriffen, im Falle von RESTful Webservices wurde die JAX-RS-Implementation des Jersey-Projekts[7] verwendet.

[7] `https://jersey.dev.java.net/`

Das Response-Mapping gehört in den Zuständigkeitsbereich der Call-Klasse. Ein durch ein Call-Objekt repräsentierter Aufruf wird von einem Query-Baumknoten der Klasse *WSNode* referenziert. Instanzen von WSNode werden in der Query-Übersetzung in den entstehenden Call-Baum eingetragen. Bei Auswertung des Call-Baums werden dann die einzelnen Calls über die jeweiligen WSNode-Knoten ausgewertet. Dabei werden die zugehörigen Aufrufe getätigt und das Response-Mapping auf die entsprechend gelieferten Response-Dokumente angewendet. Wie in der Vorgangsbeschreibung in 4.3 gesehen, erhält ein Call-Objekt alle dazu nötigen Informationen. Für die Abarbeitung der entstehenden XQuery-Ausdrücke kommt in SPARQL2ws die XQuery-Engine von *Saxon-HE 9.2*[8] zum Einsatz. Diese arbeitet auf Repräsentationen von XML-Dokumenten, die im konkreten Fall von dem Java-eigenen *SAX-Parser* geliefert werden.

Schließlich ist mit *PaginationCall* noch diejenige Spezialisierung der Klasse Call zu nennen, die Aufrufe von solchen Methoden repräsentiert, die in den Konfigurationen entsprechende Paginationseinstellungen besitzen. Sie implementiert die gleichen Methoden wie Call, delegiert diese jedoch lediglich an eine angegebene Anzahl von „Sub-Calls“.

[8] Saxon Home Edition, die Open-Source Variante des proprietären XML-Prozessor-Pakets Saxon, ist zu finden unter `http://saxon.sourceforge.net/`.

5. Ergebnisse und Diskussion

5.1. Zielerfüllung und Effizienzbetrachtung

Für die Evaluierung von SPARQL2ws mussten natürlich existierende Webservices integriert werden und das Verhalten des Programms daran getestet werden. Dazu wurden vier verschiedene Konfigurationen angelegt, die jeweils einen solchen Dienst für SPARQL-Abfragen zugänglich machen[1]: Flickr(REST), Flickr(SOAP), Youtube und Twitter. Für Testzwecke wurde mit Hilfe des Webservice-Sails prinzipiell ein SPARQL-Endpoint realisiert. Damit war es möglich, die Datenbestände der jeweiligen Dienste über SPARQL-Ausdrücke den Konfigurationen gemäß anzufragen. Die Ergebnisse wurden dabei jeweils wie erwartet ausgegeben.

Im Kontext von Datenbanksystemen spielt die benötigte Zeit zur Auswertung von Queries, aber auch der Speicherverbrauch, eine sehr große Rolle, da man potenziell beliebig große Datenmengen verarbeiten möchte. Zwar sollten die Datenmengen in den für das WSSail relevanten Anwendungsfälle nicht allzu groß ausfallen. Dennoch ist die Performanz des WSSail (bzw. der implementierten Auswertungsverfahren) bei der Abfrage von Webservices selbstredend eines der vordergründigen Qualitätsmerkmale. Daher wurde diese in entsprechenden Tests ebenfalls untersucht. Dazu wurden vier verschiedene Muster von Call-Bäumen erzeugt, die jeweils für Flickr(SOAP) und Flickr(REST) ausgewertet wurden.

Die Ergebnisse dieser Tests lassen sich Tabelle 5.1 entnehmen. Das System, an dem die Messungen stattfanden, besitzt die folgenden Leistungsdaten:

- Intel Core 2 Duo CPU (E7300, 2x2.4 GHz)
- 3 GByte Hauptspeicher
- ca. 8 Mbit/s Netto-Datendurchsatz

[1] Die angesprochenen Konfigurationen liegen (zusammen mit dem Quellcode der Implementation) der Studienarbeit bei.

Call-Muster	Variante	Auswertungsdauer		RAM-Belegung
		Absolut(s)	Latenz(%)	(MByte)
1: Suche(500 Ressourcen)	REST	4.68	71.73	18.6
	SOAP	5.1	89.9	17.9
2: Suche(2500 R.)	REST	7.85	55.96	21.3
	SOAP	6.96	70.95	19.8
3: Suche(2500 R. +	REST	8.39	53.74	21.6
je 2 Details)	SOAP	7.1	68.34	21.4
4: Suche(20 R.+ je 2 D.)	REST	14.87	94.3	18.8
+Detail-Methode(2 D.)	SOAP	11.38	85.64	18.3

Tabelle 5.1.: Ergebnisse der Performanztests am WSSail.

Man beachte, dass in Call-Muster 2 und 3 des Tests zur Suche fünf einzelne Aufrufe (zu je 500 Ressourcen) stattfinden, die von einem PaginationCall-Objekt beauftragt werden. Es fällt dabei auf, dass dies (bzw. der parallele Ablauf der Sub-Calls) einen Einfluss auf den Anteil der Antwortzeit des Service (Latenzzeit) an der gesamten Abarbeitungszeit hat. Es scheint, dass das hier auftretende Multithreading dafür verantwortlich ist, dass die absolut (nahezu) konstanten Latenzzeiten weniger in Erscheinung treten.

Die Dauer der gesamten Abarbeitung eines Call-Baums nach Call-Muster 4 des Tests ist vergleichsweise hoch. Dies ist jedoch nicht überraschend und mit einem Verweis auf den in Abschnitt 3.3 skizzierten Auswertungsvorgang leicht einzusehen.

Der in der Query-Übersetzung hergestellte Matching- bzw. Suchmethoden-Aufruf muss immer zuerst ausgewertet werden, um die Subjekt-Variable, auf die sich sämtliche Detail-Properties beziehen, zu belegen. Aufrufe etwaiger sich anschließender (Detail-)-Methoden sind also vom Aufruf der Suchmethode abhängig und stehen in einer Join-Beziehung mit diesem. Als Konsequenz für die in 3.3 dargestellte Join-Auswertung wird aber nicht nur für jede Lösung der Subjektvariablen ein eigener Aufruf jeder Detail-Methode ausgewertet, sondern dies auch mit einem Aufruf zu einer Zeit. Bei Call-Muster 4 müssen demnach also insgesamt 21 Calls nacheinander abgearbeitet werden.

5.2. Mögliche Erweiterungen

Ausdrucksstärkere Abfragen

Bei der Abfrage eines Service ist der Anwender, wie gesehen, noch vergleichsweise eingeschränkt. Das folgt einerseits aus der angesprochenen Strukturvorgabe für Queries. Andererseits können bisher aber auch nur Informationen *aus* den Datenbeständen verschiedener Webservices erfragt werden, nicht etwa *über* diese Datenbestände. Es

fehlt also die Möglichkeit Schemas selbst abzufragen bzw. für die Abfrage zu verwenden. Da eine WSSail-Konfiguration nur für die Abbildung von Teilen eines Schemas sorgt, müsste die Konfigurationssprache für die Integration von Ontologien über einen Webservice die Einbindung von entsprechenden (externen) Definitionen unterstützen. In diesen könnten dann komplexere schematische Zusammenhänge spezifiziert werden. Die Einbindung könnte z.B. über die Bekanntmachung eines URL innerhalb einer Konfiguration geschehen. Die Abbildung auf die Webservice-Schnittstelle würde dann nach wie vor in der eigentlichen Konfigurationsdatei stattfinden, könnte sich dann aber explizit auf das eingebundene Schema beziehen. Das Schema selbst ließe sich dann, sofern entsprechend in RDF ausgedrückt, in den Datenbestand des Sails integrieren und daraufhin abfragen.

Eine direkte Anwendung dieser Idee wäre beispielsweise die Einführung von Typisierungs-Informationen - bisher werden sämtliche Daten eines Webservice als ungetypte Zeichenketten behandelt.

Nebenläufige Aufrufabwicklung

Bei der Evaluierung der entwickelten Software zeigten sich in Abhängigkeit von den jeweiligen Abfragen mitunter lange Ladezeiten. Diese sind in der sequentiellen Abarbeitung der einzelnen im Call-Baum notierten Aufrufe begründet. Jede Detail-Anfrage einer Query, die einen eigenen Webservice-Aufruf nach sich zieht, wird im aktuellen Haupt-Thread des Programms exklusiv abgearbeitet. Zudem wird sie für jede Lösung der bisher abgearbeiteten Teil-Query extra ausgewertet.

Die Lösung für dieses Problem liegt in einem nebenläufigen Auswertungsprozess. Solche Webservice-Calls, die unabhängig sind, also nicht notwendige Parameter-Werte aus den Ergebnissen anderer Calls beziehen, können parallel aufgerufen und ausgewertet werden. Die Idee ist, dass diese Calls bereits innerhalb der Query-Übersetzung an den Webservice gerichtet und die Call-Responses zu Ergebnistabellen abgearbeitet werden. Die Auswertung abhängiger Calls könnte dann im „Abhängigkeits-Baum“ entsprechend propagiert werden. Bei der Auswertung des Call-Baums, bzw. der Baum-Knoten (WSNodes), findet dann nur noch der Zugriff auf die im Hauptspeicher abgelegten Tabellen statt.

Authentifizierung - OAuth

Ein Webservice stellt in der Regel nicht nur öffentlich zugängliche Funktionen zur Verfügung. Einige Methoden können über Sicherheitseinrichtungen geschützt sein, die über die einfache Übergabe von API-Nutzungskennungen hinausgehen. Insofern kann es von Interesse sein, solche Einrichtungen auch im Webservice-Sail zu implementieren. In Anbetracht der Zielsetzung dieser Arbeit, stehen von diesen Methoden diejenigen im Fokus, die etwa den Zugriff auf private Datensammlungen ermöglichen. Bei

Flickr wären dies beispielsweise Suchmethoden für den Abruf von Fotosammlungen, die durch den Besitzer explizit der Betrachtung durch Fremde entzogen wurden. Bei Flickr ist nun die Mehrzahl der Fotos ohnehin öffentlich abrufbar. Von essentieller Bedeutung ist diese Funktionalität allerdings für Webservices, die nur solch geschützte Datenbestände verwalten – soziale Dienste sind hier an erster Stelle zu nennen, da sie Daten ihrer Nutzer natürlich entsprechend schützen müssen. Als Beispiel lässt sich an der Stelle OpenSocial[2] ansehen. Dabei handelt es sich nicht direkt um einen konkreten Webservice, sondern vielmehr um einen Schnittstellenstandard für viele soziale Netzwerke, bzw. deren Webservices. Wie OpenSocial-Plattformen verwenden viele andere Dienste Autorisierungsverfahren, die eine höhere Sicherheit bieten, als z.B. Basic-Authentication von HTTP [BLFF96]. Als vergleichsweise neues Protokoll etabliert sich zur Zeit OAuth[3], das dazu entsprechende APIs zur Verfügung stellt.

Von der Integration von Techniken wie OAuth würde das Webservice-Sail insofern profitieren, als dass es neben öffentlich zugänglichen Datenbeständen auch Benutzergeschützte Daten anfragen könnte. Dies ist ein wichtiger Aspekt, zumal gerade solche Webservices an Popularität gewinnen, die ihre Dienstleistung in einen sozialen (und damit Datenschutz-kritischen) Kontext stellen.

[2] http://www.opensocial.org/
[3] http://oauth.net/

6. Fazit

Der vorliegenden Studienarbeit wurde die Aufgabe zugetragen, zu untersuchen, wie sich mehrere Webservices über ein einziges, generisches Sesame-Sail semantisch abfragen lassen, und einen Ansatz dafür zu finden. Dazu sollte eine Softwarelösung entwickelt werden, die aufbauend auf dem Sesame 2 Framework, den Zugang zu solchen Webservices ermöglicht, die auf Basis von SOAP oder nach REST kommunizieren. Auch wurde eine dazu notwendige Konfigurationssprache ausgearbeitet, die die hierfür notwendigen Einstellungen abbilden kann.

All dies wurde mit SPARQL2ws in Java implementiert. Es konnte gezeigt werden, dass der vorgestellte Ansatz dem Ziel, Daten von Web-Diensten semantisch zu erfragen, genügt. Dabei handelt es sich um eine eher prototypische Implementierung, die vordergründig eine grundsätzliche Machbarkeit unter Beweis stellen soll.

Die ebenfalls aufgezeigten Leistungsdaten des Webservice-Sails bedeuten allerdings auch, dass die Art und Weise, wie einzelne Service-Aufrufe abgearbeitet werden, noch optimiert werden kann. Ein Vorschlag hierzu wurde ebenfalls erarbeitet und präsentiert. Zusammen mit den anderen an gleicher Stelle betrachteten Erweiterungsmöglichkeiten ergeben sich aber insgesamt gute Aussichten, die Lösung bzw. die einzelnen Bestandteile für den nutzbringenden Einsatz im Kontext des Semantic Web auszubauen.

A. Syntax der Konfigurationssprache

Die im Folgenden verwendete Notation der Graph-Muster ist an Turtle angelehnt. In „{„ „}“ eingefasste Ausdrücke werden durch nachfolgende „**(*)**“ („beliebig oft“) oder „**(?)**“ („optional“) quantifiziert. Alle anderen Muster werden exakt einmal an der jeweiligen Stelle erwartet. Der Namensraum der Konfigurationssprache wird dabei abgekürzt durch Präfix „**c**“.

Schnittstellen-Definition

Interface

Anlegen einer Interface-Schnittstelle:

Graph-Muster:

```
c:interface c:requestFormat Format .
{ c:interface c:method Method . }(*)
{ c:interface c:apiKey [ c:qualifiedName AKVar; c:value APIKey ]. }(?)
{ c:interface c:retryCount Retries . }(?)
```

Erläuterung:

- Kommunikationsweg ***Format*** (Mögliche Werte `c:REST`, `c:SOAP`)
- Methoden der Schnittstelle, wie hier ***Method*** (siehe A)
- mögliche APIKey-Zuweisung mit Variablen-Bezeichner ***AKVar*** und -Wert ***APIKey***
- mögliche maximale Anzahl evtl. Neuverbindungen, ***Retries***

Anlegen einer Interface-Schnittstelle(cont'd):

Graph-Muster:

```
{ c:interface c:soapSettings [
    c:endpoint Endpoint;
    c:requestName RequestName;
```

```
        c:requestPrefix RequestPrefix;
        c:requestNameSpace RequestNamespace ].
    }(?)
    { c:interface c:xmlNameSpace [ c:prefix Prefix; c:nameSpace XMLns ]. }(*)
    { c:interface c:httpStatusHandling [
        c:statusCode Code;
        c:action Action ].
    }(*)
```

Erläuterung:

- mögliche SOAP-Einstellungen mit Endpoint-Adresse ***Endpoint***, Request-(XML-)Name ***RequestName***, Request-(XML-)Namensraum ***RequestNamespace*** und zugehörigem Präfix ***RequestPrefix***
- mögliche Namensraum-Deklarationen (Präfix ***Prefix***, XML-Namenspace ***XMLns***) für das Response-Mapping über XQuery
- mögliche Zuordnung von HTTP-Status-Code ***Code*** zu einer Aktions-ID ***Action***. ***Action*** kann (z.Z.) nur `c:CALL_ESCAPE` für den vollständigen Kommunikationsabbruch sein.

Methoden

Anlegen einer Methoden-Defintion für *Method*:

Graph-Muster:

```
{ Method c:url [ c:template Url; c:style UrlStyle ].}(?)
{ Method c:methodName [ c:qualifiedName NameVar; c:value Name ]. }(?)
{ Method c:pagination [
            c:qualifiedName PageVar; c:pages Pages ].}(?)
{ Method c:staticParameter [
            c:qualifiedName ParamName; c:value ParamValue ].}(*)
{ Method c:baseXPath BaseXPath. }(?)
```

Erläuterung:

- mögliche ***Url*** URL im Falle einer REST-Methode. Diese kann einen ausgezeichneten Bestandteil enthalten, der durch Substitution bei einem Aufruf durch einen konkreten Wert ersetzt wird. In dem Fall muss der „URL-Stil“ ***UrlStyle*** mit `c:SUBSTITUTE` angegeben werden. Im anderen Falle werden Parameter-Belegungen im Query-Teil der URL angelegt (URL-Stil `c:QUERY`).

- mögliche Einstellungen zur gleichzeitigen Anforderung durchgezählter Dokumente (Pagination): Angabe des nötigen Seiten-Parameter-Bezeichners ***PageVar*** und der Anzahl anzufordernder Dokumente ***Pages***
- mögliche statische Parameter, die für jeden Aufruf von ***Method*** konstant sind: Angabe des Parameter-Bezeichners ***ParamName*** und des Wertes ***ParamValue***
- ggf. notwendiger Basis-XPath bei Suchmethoden (siehe 4.2), angegeben durch ***BaseXPath***

Schema-Abbildung

Abbildung von Ressourcen:

Graph-Muster:

```
c:schema c:resource Resource.
  Resource c:nameSpace WSNamespace.
 { Resource c:detailProperty DProperty. }(*)
 { Resource c:matchingGroup MGroup. }(*)
```

Erläuterung:

- Angabe des URI ***Resource***, der hinzuzufügenden Ressource
- ***Resource*** zugeordneter (Webservice-spezifischer) Namensraum ***WSNamespace***; wird im Response-Mapping gelieferten IDs der ***Resource***-Instanzen vorangestellt
- ggf. ***Resource*** zugeordnete Detail-Properties wie ***DProperty*** (siehe A)
- ggf. ***Resource*** zugeordnete Matching-Property-Gruppe(n) wie ***MGroup*** (siehe A)

Abbildung von Detail-Properties:

Graph-Muster:

```
DProperty c:method Method;
          c:xpath XPath.
{ DProperty c:qualifiedName ParamName. }(?)
{ DProperty c:setResource Resource]. }(?)
```

Erläuterung:

- ***DProperty*** zugeordnete Methode ***Method***, über die das Objekt von ***DProperty*** geliefert werden soll. Zu diesem Zweck die zwingende Angabe des XPath-Ausdrucks ***XPath***
- ggf. ***DProperty*** zugeordnete Bezeichner ***ParamName*** eines Parameters der Methode ***Method***
- ggf. ***DProperty*** zugeordnete Ressource ***Resource***, falls keine Literal-Werte sondern Instanzen von ***Resource*** geliefert werden

Abbildung von Matching-Properties:

Graph-Muster:

```
MGroup c:method Method;
       c:xpath XPath.
        { c:parameter [
                c:qualifiedName ParamName;
                c:matchingProperty MProperty; c:isURI B
         ]. }(*)
```

Erläuterung:

- Zuordnung der identifizierten Matching-Property-Gruppe ***MGroup*** zu Ressource ***Resource*** des Domänen-Modells eines Webservice
- Zuordnung einzelner Matching-Properties wie ***MProperty*** zu ***MGroup***. Verknüpfung von ***MProperty*** mit Bezeichner ***ParamName*** eines Parameters von ***Method***. Die Existenz von ***B*** alleine bedeutet schon, dass der für ***ParamName*** übergebene Wert ein URI-String ist. In dem Fall wird der lokale Name extrahiert, der Rest verworfen.
- notwendiger XPath-Ausdruck ***XPath*** zur Ansteuerung und Extraktion der ID der zu suchenden Ressource im XML-Dokument

Literaturverzeichnis

[Adua] Aduna B.V. System documentation for Sesame 2.0. http://www.openrdf.org/doc/sesame2/system/index.html. [Online; letzter Zugriff am 10-September-2009].

[Adub] Aduna B.V. User Guide for Sesame 2.2. http://www.openrdf.org/doc/sesame2/users/. [Online; letzter Zugriff am 11-September-2009].

[BBC+07a] Anders Berglund, Scott Boag, Don Chamberlin, Mary F. Fernández, Daniela Florescu, Jonathan Robie, and Jérôme Siméon. XQuery 1.0: An XML Query Language. http://www.w3.org/TR/xquery/, Januar 2007. [Online; letzter Zugriff am 12-Oktober-2009].

[BBC+07b] Anders Berglund, Scott Boag, Don Chamberlin, Mary F. Fernández, Michael Kay, Jonathan Robie, and Jérôme Siméon. XML Path Language (XPath) 2.0. http://www.w3.org/TR/xpath20/, Januar 2007. [Online; letzter Zugriff am 11-Oktober-2009].

[BBL08] David Beckett and Tim Berners-Lee. Turtle - Terse RDF Triple Language. http://www.w3.org/TeamSubmission/2008/SUBM-turtle-20080114/, 2008. [Online; letzter Zugriff am 25-August-2009].

[BG04] Dan Brickley and R.V. Guha. RDF Vocabulary Description Language 1.0: RDF Schema. http://www.w3.org/TR/rdf-schema/, 2004. [Online; letzter Zugriff am 21-August-2009].

[BHM+04] David Booth, Hugo Haas, Francis McCabe, Eric Newcomer, Michael Champion, Chris Ferris, and David Orchard. Web Services Architecture. http://www.w3.org/TR/ws-arch/, Februar 2004. [Online; letzter Zugriff am 3-September-2009].

[BKvH02] Jeen Broekstra, Arjohn Kampman, and Frank van Harmelen. Sesame: A Generic Architecture for Storing and Querying RDF and RDF Schema. http://www.openrdf.org/doc/papers/Sesame-ISWC2002.pdf, 2002. [Online; letzter Zugriff am 12-November-2009].

[BLF99] Tim Berners-Lee and Mark Fischetti. *Weaving the Web*. Harper San Francisco, 1999.

[BLFF96] Tim Berners-Lee, Roy Fielding, and Henrik Frystyk. Hypertext Transfer Protocol – HTTP/1.0. http://tools.ietf.org/html/rfc1945, Mai 1996. [Online; letzter Zugriff am 23-September-2009].

[BLHL01] Tim Berners-Lee, James Hendler, and Ora Lassila. The Semantic Web. *Scientific American*, pages 34–43, Mai 2001.

[BM04] Dave Beckett and Brian McBride. RDF/XML Syntax Specification (Revised). http://www.w3.org/TR/rdf-syntax-grammar/, 2004. [Online; letzter Zugriff am 23-August-2009].

[BPSM+08] Tim Bray, Jean Paoli, C. M. Sperberg-McQueen, Eve Maler, and François Yergeau. Extensible Markup Language (XML) 1.0 (Fifth Edition). http://www.w3.org/TR/2008/REC-xml-20081126/, 2008. [Online; letzter Zugriff am 12-Oktober-2009].

[Bra07] Steve Bratt. Semantic Web, and Other Technologies to Watch. http://www.w3.org/2007/Talks/0130-sb-W3CTechSemWeb/#%2824%29, 2007. [Online; letzter Zugriff am 7-Dezember-2009].

[Bri01] David Brickley. Semantic Web History: Nodes and Arcs 1989-1999 (the WWW Proposal as RDF). http://www.w3.org/1999/11/11-WWWProposal/, März 2001. [Online; letzter Zugriff am 1-September-2009].

[FGM+99] Roy Fielding, J. Gettys, J. Mogul, Henrik Frystyk, L. Masinter, P. Leach, and Tim Berners-Lee. Hypertext Transfer Protocol – HTTP/1.1. http://www.w3.org/Protocols/rfc2616/rfc2616-sec9.html#sec9.6, Juni 1999. [Online; letzter Zugriff am 23-September-2009].

[Fie00] Roy Thomas Fielding. REST: Architectural Styles and the Design of Network-based Software Architectures. http://www.ics.uci.edu/~fielding/pubs/dissertation/top.htm, 2000. [Online; letzter Zugriff am 18-September-2009].

[GHM+07] Martin Gudgin, Marc Hadley, Noah Mendelsohn, Jean-Jacques Moreau, Henrik Frystyk Nielsen, Anish Karmarkar, and Yves Lafon. SOAP Version 1.2 Part 1: Messaging Framework. http://www.w3.org/TR/soap12-part0/, April 2007. [Online; letzter Zugriff am 16-September-2009].

[HBS08] Alice Hertel, Jeen Broekstra, and Heiner Stuckenschmidt. RDF Storage and Retrieval Systems. http://ki.informatik.uni-mannheim.de/fileadmin/publication/Hertel08RDFStorage.pdf, 2008. [Online; letzter Zugriff am 09-September-2009].

[HKRS08] Pascal Hitzler, Markus Krötzsch, Sebastian Rudolph, and York ER Sure. *Semantic Web – Grundlagen*. Springer, Berlin [u.a.], 2008.

[LS99] Ora Lassila and Ralph Swick. Resource Description Framework (RDF) Model and Syntax Specification. `http://www.w3.org/TR/1999/REC-rdf-syntax-19990222/`, 1999. [Online; letzter Zugriff am 20-August-2009].

[MMM04] Frank Manola, Eric Miller, and Brian McBride. RDF Primer. `http://www.w3.org/TR/rdf-mt/`, 2004. [Online; letzter Zugriff am 18-August-2009].

[PS08] Eric Prud'hommeaux and Andy Seaborne. SPARQL Query Language for RDF. `http://www.w3.org/TR/rdf-sparql-query/`, 2008. [Online; letzter Zugriff am 27-August-2009].