Institut für Parallele und Verteilte Systeme

Universität Stuttgart
Universitätsstraße 38
D–70569 Stuttgart

Bachelorarbeit Nr. 279

Dokumentation und automatische Generierung von Service-Beschreibungen

Philipp Kraus

Studiengang:	Softwaretechnik
Prüfer/in:	Prof. Dr.-Ing. habil. Bernhard Mitschang
Betreuer/in:	Dipl.-Inf. Jan Königsberger
Beginn am:	18. November 2015
Beendet am:	11. Mai 2016
CR-Nummer:	D.2.11, H.2.7, H.3.5, I.2.4

Kurzfassung

Serviceorientierte Architekturen (SOA) gewinnen in unserer heutigen modernen Welt als Unternehmensarchitektur immer mehr an Relevanz. Deren Vorteile wie die Wiederverwendung von Funktionalitäten und die einfache Integration von neuen und alten Anwendungen in das Gesamtsystem, ermöglichen es Unternehmen schnell auf die sich stetig ändernden Anforderungen des globalisierten Markts zu reagieren. Um nach einer Umsetzung des Serviceorientierten Paradigmas auch weiterhin von dessen Vorteilen zu profitieren müssen einheitliche Richtlinien, Mechanismen, Standards und Normen definiert werden. Eine funktionierende SOA Governance hat dies zum Ziel und verhindert es den Überblick über die SOA zu verlieren. Das Erfassen von Diensten in einem zentralen Metadaten Repository ist beispielsweise einer der umzusetzenden Aspekte innerhalb einer SOA Governance. Wurden anfangs Serviceorientierte Architekturen meist mit Webservices und dem SOAP-Protokoll realisiert, so nimmt das REST-Protokoll und dessen Programmierparadigma eine immer wichtigere Rolle ein. Gerade wenn es darum geht leichtgewichtige Anwendungen zu entwickeln ist SOAP häufig ein zu umfangreiches Protokoll. Sollen die Schnittstellen dieser Webservices beschrieben werden, so ist für SOAP-Services, mit WSDL, schon ein allgemein gültiger Standard gefunden. Für REST-Services existiert dieser jedoch noch nicht.

Nach dem Finden, einer für REST-Schnittstellen passenden, Beschreibungssprache geht es in dieser Arbeit darum eine Möglichkeit zu schaffen, das Einpflegen von Diensten in ein zentrales Repository mit dem Import der Schnittstellenbeschreibungen, zu erleichtern. Ziel ist es anschließend diese Schnittstellenbeschreibungen in die gefundene Beschreibungssprache zu exportieren und somit Dienste unabhängig von der technischen Realisierung ihrer Schnittstelle anzubieten. Hierfür wird ein Konzept beschrieben und dessen Umsetzung prototypisch untersucht.

Inhaltsverzeichnis

Abbildungsverzeichnis

Tabellenverzeichnis

Verzeichnis der Listings

1 Einleitung

Immer häufiger müssen in großen Unternehmensarchitekturen sowohl Altsyteme als auch neue Anwendungen miteinander kommunizieren, in das Gesamtsystem integriert und für Nutzer verfügbar gemacht werden. Ein mögliches Architekturparadigma sind hierbei Serviceorientierte Architekturen (SOA). Integrierte Anwendungen kommunizieren lose gekoppelt über eine Mittelschicht miteinander und können vielseitig kombiniert und eingesetzt werden. Serviceorientierte Architekturen werden daher im heutigen Unternehmensumfeld immer allgegenwärtiger. Vorteile wie die hohe Wiederverwendbarkeit von Funktionalitäten, die Integration von Legacy Systemen und die Möglichkeit auf neue und sich stetig ändernde Anforderungen schnell zu reagieren, machen sie zu einem immer wichtiger werdenden Architekturstil. Bei der Umsetzung einer SOA reicht es jedoch nicht, sich nur auf die technische Realisierung zu konzentrieren. Häufig verlieren Beteiligte den Überblick über die integrierten Dienste und deren Abhängigkeiten und es entsteht eine Unternehmensarchitektur, welche keine der oben genannten Vorteile mehr mit sich bringt. Es muss also, über die technische Durchführung hinaus, ein Rahmenwerk geschaffen werden, welches die Umsetzung eines unternehmensweiten Verständnisses für das Paradigma der Serviceorientierten Architektur schafft – Eine SOA Governance. Ein wichtiger Aspekt hierbei ist die Einführung von Richtlinien, Steuerungsmechanismen und Prozessen zum Verwalten und Überwachen der in einer SOA beteiligten Dienste, Rollen und Akteure [KSM14]. Metadaten, Artefakte und Zusatzinformationen zu allen in einer SOA vorhandenen Objekte können z.B. in einem SOA Governance Repository gespeichert werden, welches diese Informationen zentral verwaltet und die Umsetzung der definierten Richtlinien und Mechanismen überwacht. Auch die formale Beschreibung von Schnittstellen spielt bei der Verwirklichung solch einer Governance eine wichtige Rolle. Formale Schnittstellenbeschreibungen existenter und neu entwickelter Services, wie beispielsweise mit der Webservice Description Language (WSDL) beschriebene Dokumente, ermöglichen es, sowohl eine für den Menschen als auch für Maschinen lesbare Repräsentation zu speichern.

1.1 Motivation

Waren anfangs die meisten Serviceorientierten Architekturen mit Hilfe von Webservices und SOAP realisiert, so verwenden heutige Unternehmensarchitekturen auch das Representational State Transfer kurz REST Paradigma zur Bereitstellung von Anwendungsschnittstellen. Durch die Umsetzung einer SOA auch in kleineren Betrieben und das Problem einer technisch schwergewichtigen Umsetzung wächst die Nachfrage nach einer leichtgewichtigen Umsetzung

für eine Serviceorientierte Architektur [Jon08]. Insbesondere für mobile Anwendungen, für Mikroservices oder für das Auslagern von Anwendungen in die Cloud, ist das schwergewichtige SOAP-Protokoll eher ungeeignet und wird von REST-APIs abgelöst. Existierende SOAP-Services müssen bei Bedarf in REST-Services umgewandelt werden, damit deren Funktionalität von einer konkreten Schnittstelle angeboten werden kann. In ein zentrales Repository aufgenommene Schnittstellenbeschreibungen und deren Umwandlung in neue Beschreibungen bieten hierfür eine mögliche Lösung.

Auch in der Softwareentwicklung nehmen Schnittstellenbeschreibungen eine immer größere Rolle ein. Existiert für SOAP-Services das gängige Format WSDL, so ist für REST-APIs noch kein etablierter Standard gefunden. Trotz der sich selbst beschreibenden REST-Services besteht gerade im Unternehmensumfeld die Notwendigkeit für eine eindeutige Schnittstellenbeschreibung. Schnell geht der Überblick über die bereitgestellten Ressourcen einer Schnittstelle verloren und das Prinzip REST-Schnittstellen als traversierbaren Graphen mit einem Startpunkt anzubieten (HATEOAS), nicht mehr eingehalten. Eine formale Beschreibung aller verfügbaren Ressourcen schafft wieder einen Überblick und ein Dokument zum gemeinsamen Austausch über die bereitgestellte Funktionalität. Auf dieser Grundlage können automatisch Quellcode generiert, Tests geschrieben oder ein Spezifikationsdokument erstellt werden. Gerade für kleinere DevOps- bzw. Entwicklerteams ist diese Art der Softwareentwicklung, API-First Approach genannt, eine schnelle Möglichkeit, Dienste zu entwickeln.

1.2 Zielsetzung

Ziel der Arbeit ist es, eine Möglichkeit zum Importieren von Schnittstellenbeschreibungen in ein zentrales Repository zu schaffen und daraus weitere Beschreibungen, insbesondere für REST-Services, zu generieren. Im Zuge dessen wird eine für REST-Services und in ein Unternehmensumfeld passende Beschreibungssprache gesucht. Wie bereits erwähnt, existieren schon einige Beschreibungssprachen für REST-Schnittstellen, jedoch unterscheiden sich diese in Umfang und Funktionalität. Daher muss eine Beschreibungssprache gefunden werden, welche die besten Chancen hat, sich als gängiger Standard zu etablieren. Ist eine passende Beschreibungssprache gefunden, soll auf dieser Grundlage ein Konzept zum Importieren existierender WSDL-Beschreibungen und dem anschließenden Umwandeln in die gefundene Beschreibungssprache entwickelt werden. Das Konzept gliedert sich in drei umzusetzende Aufgaben. Dem Speichern der im WSDL-Dokument befindlichen Informationen in einem passenden Datenmodell, dem Umwandeln möglicher SOAP-Services in REST-Schnittstellen und dem Exportieren in die neue Beschreibungssprache. Anschließend soll die Umsetzbarkeit des vorgestellten Konzepts geprüft werden. Dazu wird dieses prototypisch für ein existierendes SOA Governance Repository implementiert.

1.3 Aufbau der Arbeit

Nach **Kapitel 1** befasst sich **Kapitel 2** mit den notwendigen Grundlagen für diese Arbeit.

Kapitel 3 mit dem Finden, einer passenden Beschreibungssprache. Hierfür werden nach einer Marktanalyse in Frage kommende Sprachen anhand verschiedener Kriterien verglichen und bewertet.

Kapitel 4 stellt das Konzept zum Importieren existierender WSDL-Beschreibungen und dem anschließenden Umwandeln in die gefundene Beschreibungssprache vor.

Kapitel 5 prüft die beschriebene Umsetzbarkeit des Konzepts mittels einer prototypischen Implementierung.

Kapitel 6 gibt eine abschließende Zusammenfassung über die in der Arbeit behandelten Themen und einen Ausblick auf noch nicht angesprochene Aspekte.

2 Grundlagen

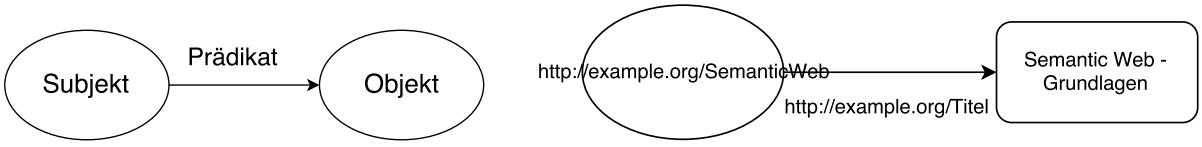
Dieses Kapitel beschreibt die für das Verständnis dieser Arbeit notwendigen Grundlagen. Insbesondere im Bereich der Entwicklung von Webservices und dem Themenfeld der SOA wird ein grundlegend breit gefächertes Wissen vorausgesetzt. Im Folgenden werden die für das Verständnis erforderlichen Grundbegriffe erklärt und in ihren Grundzügen umrissen.

2.1 Grundlegende Technologien

In diesem Abschnitt werden zwei für diese Arbeit grundlegenden Technologien vorgestellt und näher erläutert.

2.1.1 Ressource Description Framework (RDF)

Das Ressource Description Framework (RDF) [Api16] ist eine Technologie des Semantic Web[1]. Es beschreibt ein graphenbasiertes Datenmodell zum Darstellen von Informationen im Web [CWL14]. Jeder Datensatz besteht aus einer Menge von Triplen. Jedes Triple besteht aus zwei Knoten und einer verbindenden Kante. Der erste Knoten wird Subjekt, die Kante Prädikat und der zweite Knoten Objekt genannt. Abbildung 2.1a stellt ein simples RDF Triple graphisch dar.



(a) Simples RDF Triple **(b)** Simples RDF Literal mit URIs [Hit08]

Abbildung 2.1: Das RDF Graphenmodell

Durch die Graphenstruktur von RDF eignet es sich perfekt dazu, Beziehungen zwischen Ressourcen zu beschreiben. Ressourcen werden, wie im Web üblich, mit eindeutigen URIs identifiziert. Sollen konkrete Datenwerte abgespeichert werden, so geschieht dies über Literale.

[1]Semantic Web https://www.w3.org/standards/semanticweb/

Abbildung 2.1b zeigt ein Triple mit eindeutigen URIs und einem Literal als Objekt. Sollen RDF-Triples persistent gespeichert werden, kann dies in verschiedenen Dateirepräsentationen, wie zum Beispiel XML, aber auch in Datenbanksystemen, wie zum Beispiel einem RDF Triple Store, getan werden. Bekannte Triple Stores sind Sesame[2], Jena[3] und OpenLink Virtuoso[4] [GKO15]. Im Gegensatz zu Relationalen Datenbanken wird nicht mit der Structured Query Language (SQL) auf die Daten zugegriffen, sondern mit der SPARQL Protocol And RDF Query Language [PS13] über den Graphen traversiert.

Ontologien

Mit RDF können nur konkrete Beziehungen zwischen Ressourcen dargestellt werden. Sollen einem Graphen jedoch weitere Informationen anstatt lediglich konkreter Werte mitgeben werden stößt das Ressource Description Framework an seine Grenzen. So könnte es auch Knoten geben, welche Datentypen, Klassen und deren Instanzen repräsentieren. Wird beispielsweise eine Aussage darüber getroffen um was für Objekte es sich bei dem Graphen in Abbildung 2.1b handelt, so kommt ein Mensch schnell dahinter, dass es sich bei dem Literal *Semantic Web - Grundlagen* wohl um den Titel eines Buches handelt. Für eine Maschine ist dies jedoch, wie alle anderen möglichen Literale, nur eine Folge von Zeichen. Es muss also ein Vokabular über Bücher definiert werden, welches die konkreten Triple mit einem Buch-Datentypen verbindet. Dadurch weiß die Maschine, dass alle Instanzen des Buchvokabulars vom Typ Bücher und somit Bücher sind.

Ein solches Vokabular wird Ontologie genannt. Mit dieser können letztendlich Hintergrundinformationen zu den einzelnen RDF-Triplen vermerkt und Datenobjekte zu konkreten Instanzen verschiedener Ontologien gemacht werden. Eine Maschine kann dann Rückschlüsse über die gespeicherten Informationen und deren Beziehungen zueinander ziehen. Ontologien können entweder durch RDF-Schema (RDFS) [BG14] oder durch die Web Ontology Language (OWL) [MPP12] beschrieben werden. RDF ist hierbei sowohl eine echte Teilmenge aus RDFS, als auch aus OWL, was bedeutet, dass RDFS-und OWL-Beschreibungen in RDF-Triple überführt werden können.

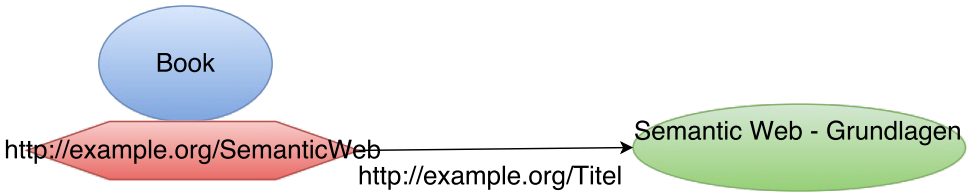


Abbildung 2.2: RDF Triple mit zusätzlichen Informationen zu deren Datentypen

[2]Sesame http://rdf4j.org/
[3]Jena https://jena.apache.org/
[4]http://virtuoso.openlinksw.com/

Die Ressource *http://example.org/SemanticWeb* könnte zum Beispiel eine konkrete Instanz der Klasse Buch sein. Abbildung 2.2 verdeutlicht dies. *Book* ist eine Klasse und *http://example.org/SemanticWeb* und dessen Literal eine konkrete Instanz. Bekannte Ontologien sind die *Friend of a Friend (FOAF)*-Ontologie [BM14] oder die DBpedia-Ontologie [5]. Die FOAF-Ontologie beschreibt Personen und deren Beziehungen zueinander, die DBpedia-Ontologie die in Wikipedia existierende Objekte und deren Typen.

2.1.2 XSL Transformation (XSLT)

XSL Transformation (XSLT) ist eine Spezifikation zum Definieren von Regeln, mit deren Hilfe ein bestehendes XML-Dokument in ein anderes XML-Dokument umgewandelt werden kann. In dieser Arbeit wird die Version 1.0 der XSLT-Spezifikation verwendet [Jam99]. Mit der Zunahme der Extensible Stylesheet Language (XSL) kann ein XML-Dokument, basierend auf einem bestimmten XSL-Schema, in ein anderes Schema überführt werden. Jedes XSLT-Dokument ist grundlegend gleich aufgebaut. Mit Hilfe des *xsl:template*-Elements werden Regeln definiert, welche alle von einem *xsl:stylesheet*-Element oder *xsl:transform*-Element umschlossen werden. In den Regeln können dann Elemente (*xsl:element*) Attribute (*xsl:attribute*) geschrieben und Texte kopiert (*value-of*) oder neu erstellt (*xsl:text*) werden. Weitere Regeloperationen sind aus der XSLT-Spezifikation zu entnehmen [Jam99]. Mit Hilfe des *match*-Attributs werden dann Bedingungen für das Auslösen der definierten Regeln festgelegt. Zum Traversieren durch den Baum der Quelldatei wird hierfür XPath [RCS15] verwendet, eine Ausdruckssprache durch welche sich Knoten in einem XML-Dokument auswerten lassen. Der Baum der Quelldatei wird somit in einen Ergebnisbaum transformiert.

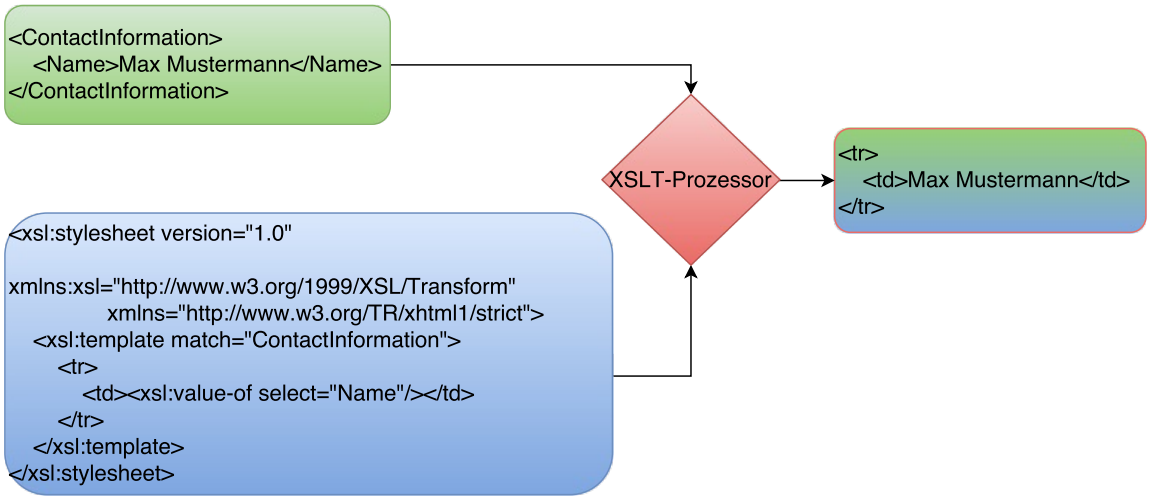


Abbildung 2.3: Umwandlung eines XML-Baums in einen HTML-Baum mit Hilfe einer XSLT-Regel und eines XSLT-Prozessors

[5] DBpedia Ontologie http://wiki.dbpedia.org/

Um eine Transformation praktisch durchzuführen benötigt es zudem einen XSLT-Prozessor. Dieser ist eine Software, welche die Quelldatei und die XSLT-Datei als Eingabe nimmt und daraus, mit den Regeln der XSLT-Beschreibung, einen neuen Ergebnisbaum generiert. Abbildung 2.3 zeigt eine simple Umwandlung eines XML-Baums mit Kontaktinformationen in eine HTML Tabellenrepräsentation.

2.2 Serviceorientierte Architektur (SOA)

Serviceorientierte Architektur (SOA) ist ein Architekturparadigma zum Verwalten und Zusammenführen von wiederverwendbaren Funktionalitäten in einer meist verteilten Umgebung [LL09]. Anders ausgedrückt bietet das Serviceorienterte Paradigma die Möglichkeit, eine Funktionalität in kleinere Bausteine, sogenannte Dienste/Services, zu kapseln. Diese können wiederum in neue Funktionalitäten zusammengesetzt werden [Erl05]. Melzer et al. definieren hierfür vier für die Architektur grundlegende Eigenschaften [Dos05]:

Verteiltheit Bedeutet, dass alle Dienste in einer verteilten Umgebung integriert sind.

Lose Kopplung Beurteilt die Abhängigkeiten zwischen den Diensten. Je mehr Abhängigkeiten, desto weniger lose sind die einzelnen Dienste miteinander gekoppelt.

Verzeichnisdienst Beschreibt die Auffindbarkeit eines Services durch einen Verzeichnisdienstes, in welchen alle im System integrierten Dienste und deren Funktionalitäten registriert sind.

Prozessorientiertheit Die Möglichkeit, die in der Architektur vorhandenen Services zu Prozessen zusammenzufügen. Hierbei soll es möglich sein schnell auf wechselnde Anforderungen reagieren zu können.

Das sogenannte SOA-Dreieck beschreibt die wichtigsten Bestandteile einer SOA. Es stellt die Rollen und Aktionen innerhalb der Architektur dar. So kommuniziert ein Dienstnutzer nie mit einem festen Dienstanbieter, sondern sucht erst im Verzeichnisdienst nach einem gerade verfügbaren Service mit der passenden Funktionalität und schickt dann an diesen seine Anfragen. Das Dienstverzeichnis hat die Aufgabe, integrierte Dienste zu registrieren und bei Anfrage einem Servicenutzer einen passenden Dienst zur Verfügung zu stellen.

Abbildung 2.4 stellt das SOA-Dreieck dar. Eine mögliche technische Realisierung dieses SOA-Dreiecks und damit auch einer SOA ist die Realisierung mit Hilfe von Webservices. Einzelne Services kommunizieren über das SOAP Protokoll miteinander (siehe Abschnitt 2.5) und registrieren sich in einem UDDI-Verzeichnisdienst mit ihrer WSDL-Datei (siehe Abschnitt 2.6.1). UDDI steht für „Universal Description, Discovery and Integration“ und bezeichnet einen standardisierten Verzeichnisdienst zum Speichern von Servicebeschreibungen. Diese Beschreibung dient dann als Grundlage zum Erfassen der einzelnen Funktionalitäten eines Services und dem Abrufen im Verzeichnisdienst.

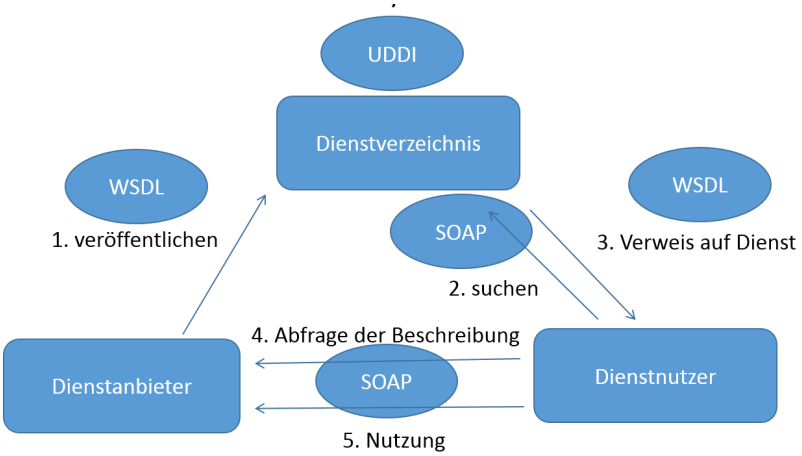


Abbildung 2.4: Das SOA-Dreieck in einer Realisierung mit Webservices [Dos05]

2.3 SOA Governance

Um eine Serviceorientierte Architektur zu realisieren, reicht die technische Umsetzung einer SOA alleine nicht aus. Das serviceorientierte Paradigma darf nicht nur Einzug auf technischer Ebene halten, sondern muss sich vielmehr als Grundverständnis im gesamten Unternehmen etablieren [Erl05]. SOA Governance versucht dies, indem die Kontrolle über technische, strategische, organisatorische Anforderung an eine SOA und deren Nutzer ermöglicht wird und hierfür nötige Richtlinien und Kontrollmechanismen etabliert werden [KSM14]. Eine fehlende SOA Governance ist somit meist der Hauptgrund, warum die Vorteile einer SOA nicht ersichtlich werden oder die Architektur ganz scheitert [GF07]. Königsberger et al. definieren dazu elf grundlegende Anforderungen an ein SOA Governance Framework [KSM14]:

- Service Life Cycle Management
- Consumer Management
- Meta Data Management
- Organizational Structure
- Portfolio Management
- Architectural Standards
- Governance Hierarchy
- Funding Model
- Service Monitoring
- Maturity Measurement
- Business Object Management

Vor allem beim Meta Data Management, Portfolio Management und Business Objekt Management spielen Servicebeschreibungen eine wichtige Rolle. Meta Data Management beschreibt das Zusammentragen von allen zu einem Service zugehörigen Artefakten. Hierzu gehören beschreibende Dokumente wie Design und Spezifikation. Auch Informationen über die Servicenutzer und deren formale Verträge zwischen Service und Nutzer, wie zum Beispiel Servicebeschreibungen, gehören dazu. Portfolio Management beschreibt das Verwalten der in einer Architektur vorhanden Services und deren Funktionalität. Dies verhindert, dass schon existierende Funktionalitäten noch einmal entwickelt werden. Auch hierfür bieten sich Servicebeschreibungen an, da diese eine formale Beschreibung der vorhandenen Funktionalitäten enthalten. Im Business Object Management werden die hinter einem Service liegenden Datenmodelle, wie beispielsweise die von einer Schnittstelle geforderten Datentypen, in sogenannte Businessobjekte abstrahiert. Somit werden Wiederverwendbarkeit, Integration und Datentransparenz innerhalb einer SOA gefördert.

2.3.1 SOA Governance Metamodel (SOA-GovMM)

Aufgrund der mangelhaften Umsetzung existierender SOA Governance Frameworks stellen Königsberger et al. ein eigenes, alle oben genannten Aspekte umfassendes Meta Modell für ein mögliches SOA Governance Repository vor. Dieses wird in Abbildung 2.5 gezeigt. Es ist in vier Bereiche strukturiert, dem Service Provider, dem Service Consumer der Organizational Structure und dem Business Object Bereich. Der Service Provider hält alle Metadaten rund um die in einer SOA integrierten Services sowie deren einzelne Versionen und Artefakte. Der Service Consumer beinhaltet alle Daten des Servicenutzers und verwaltet den Vertrag (Contract) zwischen Service Consumer und Service Provider. Der Business Object Bereich beinhaltet Objekte, welche abstrakte Datenmodelle der einzelnen Services darstellen sollen. Ein Businessobjekt ist somit auch ein mögliches Artefakt eines Services. Die Organizational Structure umfasst alle in der Unternehmensarchitektur beteiligten Nutzer und verwaltet ein feingranulares Rollen -und Rechtesystem. Einzelne Nutzer sind über ihre Rollen in der Architektur an Services, Serviceversionen, Consumer oder Businessobjekte geknüpft.

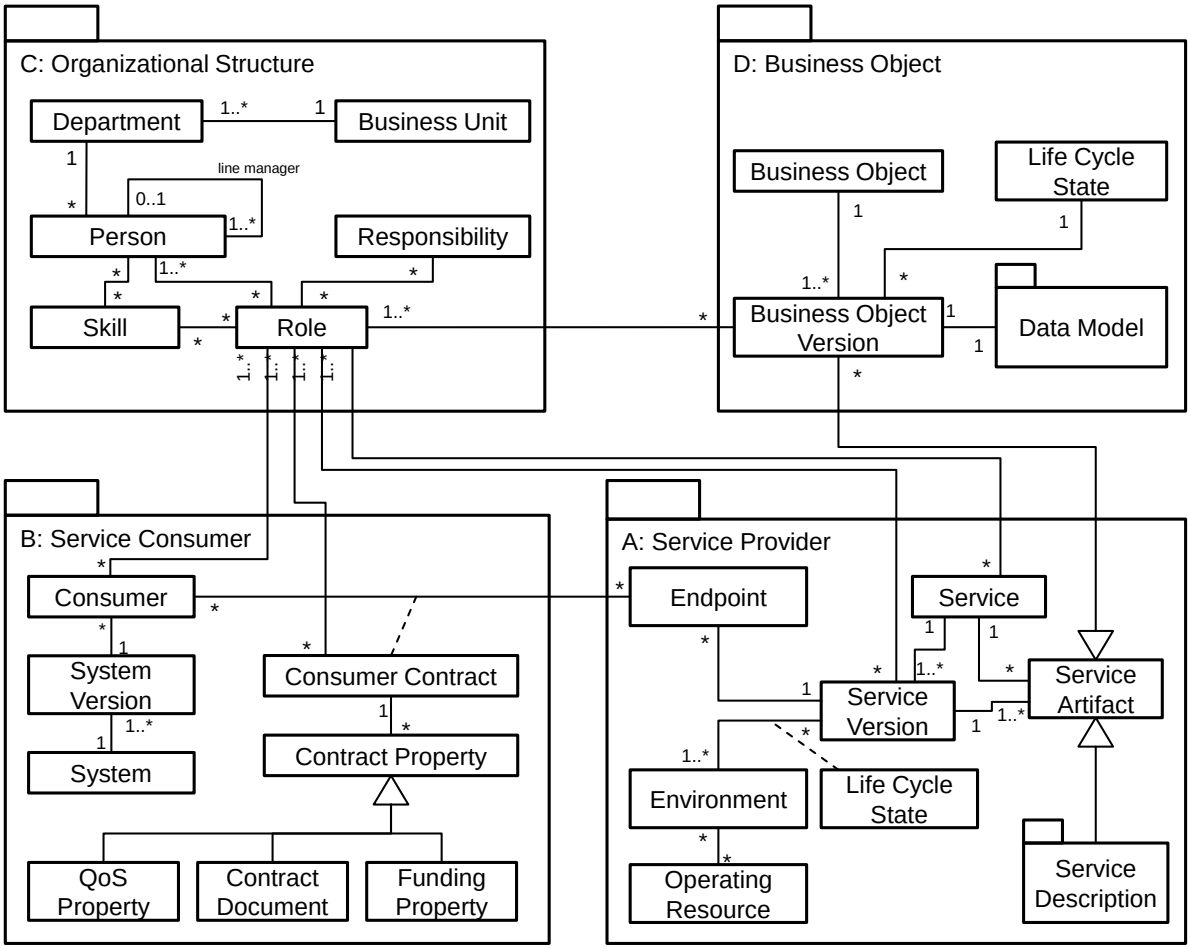


Abbildung 2.5: SOA Governance Meta Model [KSM14]

2.4 Governance Repository Prototyp

Im Folgenden wird eine mögliche Implementierung des SOA Governance Meta Models vorgestellt. Diese dient als Implementierungsgrundlage für das in Kapitel 4 vorgestellte und Kapitel 5 implementierte Konzept. Der Governance Repository Prototyp ist ein zentrales Repository für Metadaten in einer exemplarischen Serviceorientierten Architektur. Die im SOA-GovMM gespeicherten Daten werden vom implementierten Prototypen in einer Weboberfläche anschaulich dargestellt. Stakeholder innerhalb einer SOA können Metadaten von Nutzern, Services, deren Businessobjekte und Serviceconsumer verwalten und deren jeweilige Beziehungen untereinander einsehen und bearbeiten. Abbildung 2.6 zeigt einen Screenshot aus dem SOA Governance Repository für die Verwaltung von Services.

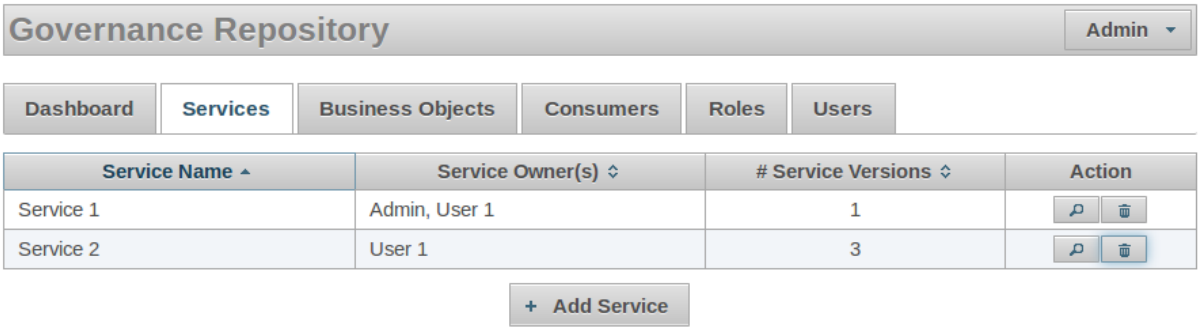


Abbildung 2.6: Screenshot der Serviceansicht des Governance Repository Prototypen

2.4.1 Grobarchitektur

Die Architektur des Governance Repositories basiert auf dem Model View View Model (MVVM) Pattern [Gar11] und besteht aus fünf Hauptkomponenten. Das MVVM ist eine Abänderung des Model View Controller (MVC) Patterns und dient dazu, Logik und GUI vollständig voneinander zu entkoppeln. Somit können GUI und Businesslogik getrennt voneinander entwickelt werden und ohne großen Aufwand ausgetauscht werden. Neben den MVVM-Komponenten spaltet sich das Model noch in eine Datenbank-Controller-Komponente. Diese dient dazu die Datenbankanbindung lose gekoppelt an die Model-Komponente zu entwickeln und beliebig auszutauschen. So wurde in der Arbeit von Victor Miyai [Miy15] die ursprüngliche relationale Anbindung durch eine RDF Triplestoreanbindung ersetzt. Abbildung 2.7 zeigt die erwähnten Komponenten. Um weitere Wartungsarbeiten so gering wie möglich zu halten, soll diese Kopplung auch bei der Implementierung neuer Funktionalitäten eingehalten werden.

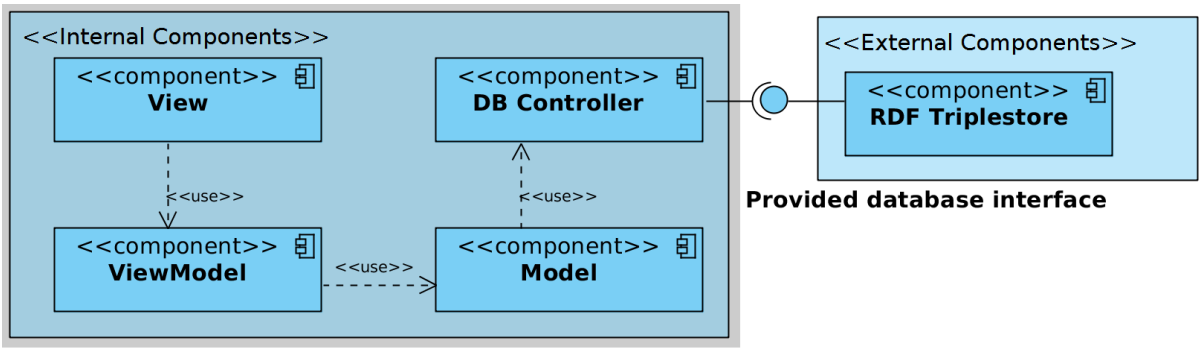


Abbildung 2.7: Komponenten des Governance Repositories

View Model und View sind mit Hilfe von Primefaces[6], einem auf Java Serverfaces und JavaScript basierenden Ajax Framework, implementiert. Die Datenbankanbindung ist mit Empire RDF[7], einer für RDF umgesetzte Java Persistence API (JPA)-Implementierung [Ora13], implementiert und spricht einen Sesame[8] Triplestore an.

2.4.2 RDF Modell

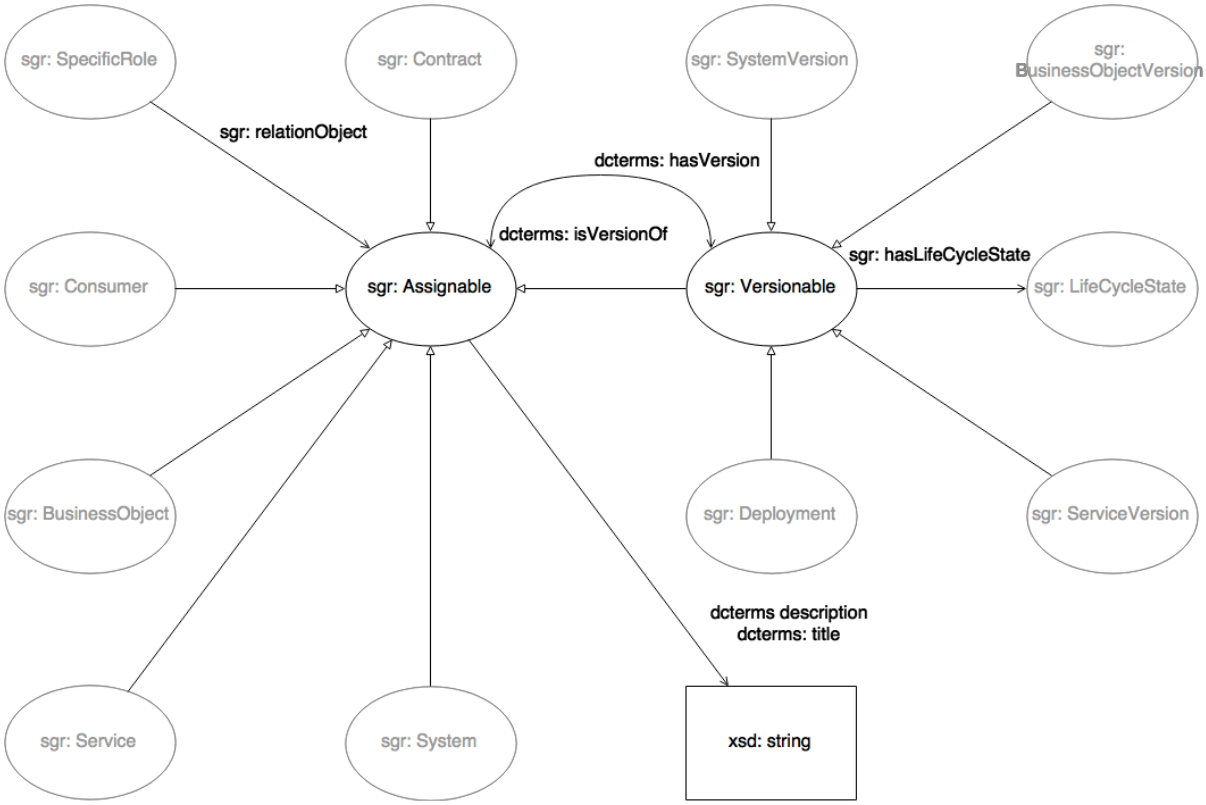


Abbildung 2.8: Basisentitäten des RDF Modells [Miy15]

Um die Daten des Governance Repositories in einer Datenbank zu persistieren, wurde das SOA Governance Meta Model in eine Ontologie überführt. Somit können einerseits die Daten des Governance Repositories verwaltet werden und andererseits beliebig neue Ontologien, bzw. Datenmodelle, an den Graphen des Repositories angehängt werden. Abbildung 2.8 veranschaulicht die Hauptentitäten der Ontologie. Die Basisentitäten des Modells sind *sgr:Assignable* und *sgr:Versionable*. Als „assignable“ wird jedes Objekt bezeichnet, welches mit einer Rolle (*sgr:SpecificRole*) in Verbindung gebracht werden kann. Hierzu gehören die Entitäten *sgr:Service*,

[6]Primefaces http://primefaces.org/
[7]Empire RDF https://github.com/mhgrove/Empire
[8]Sesame http://rdf4j.org/

sgr:BusinessObject, *sgr:Consumer* oder *sgr:Contract*. Diese sind wiederum Klassen für weitere Relationen wie Namen (*sgr:Name*) oder Beschreibung (*sgr:Description*). Jede Instanz der *sgr:Assignable*-Klasse kann mit beliebig vielen Instanzen von *sgr:Versionable* verknüpft werden, welche den einzelnen Versionen der Objekte, also beispielsweise Serviceversionen oder Businessobjektversionen, entsprechen. Eine detailiertere Beschreibung findet sich in der Arbeit von Victor Miyai [Miy15].

2.5 Webservices mit REST & SOAP

Bei der technologischen Umsetzung einer SOA gibt es verschiedene Möglichkeiten. Da Webservices die meisten Anforderungen an eine SOA, wie lose Kopplung oder Verteiltheit, umsetzen, bietet es sich an bei der Umsetzung einer Serviceorientierten Architektur Webservices zu verwenden [Erl05]. Kommunizierten Webservices anfangs nur über das schwergewichtige SOAP Protokoll, so kommunizieren in heutigen Serviceorientierten Architekturen immer mehr Services mit Hilfe des deutlich leichtgewichtigeren REST-Protokolls. Dieser Abschnitt gibt einen grundlegenden Einblick in das SOAP und REST Protokoll und zeigt deren grundlegende Unterschiede auf.

2.5.1 SOAP (Ursprünglich: Simple Object Access Protocoll)

SOAP ist ein vom World Wide Web Consortium (W3C) spezifiziertes Kommunikationsprotokoll mit welchem, die Kommunikation zwischen einzelnen Webservices, über RMI (Remote Procedure Call) oder primitiven Datenaustausch, ermöglicht werden soll. Die auszutauschenden Daten werden in einer XML Datei repräsentiert und über HTTP übertragen. Die aktuelle Version ist Version 1.2 [ML07]. Abbildung 2.9 veranschaulicht den grundsätzlichen Aufbau einer SOAP-Nachricht.

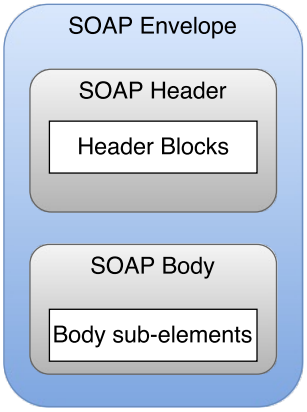


Abbildung 2.9: Struktur einer SOAP Nachricht nach [ML07]

Listing 2.1 Exemplarischer SOAP Request

```
<?xml version='1.0' ?>
<soap:Envelop xmlns:soap="http://schemas.xmlsoap.org/soap/envelope/">
      <soap:Header>
      ...
      </soap:Header>
      <soap:Body>
            <TradePriceRequest xmlns="http://example.com/stockquote.xsd">
                  <tickerSymbol>GOOG</tickerSymbol>
            </TradePriceRequest>
      </soap:Body>
</soap:Envelope>
```

Um alle Elemente in einer SOAP-Nachricht befindet sich das *envelope*-Element. Es ist das Wurzelelement der Datei. Im *envelope*-Element befinden sich das *header* -und das *body*-Element. Das header Element ist optional, in ihm können Informationen für die Infrastruktur wie zum Beispiel Routing-Hinweise oder Sicherheitsinformationen, enthalten sein. Im *Body*-Element befinden sich die eigentlichen Informationen, welche der Servicenutzer zurückgegeben bekommt, bzw. der Serviceanbieter fordert. Die innere Struktur des *Body*-Elements wird in der WSDL-Beschreibung des Serviceanbieters definiert. Listing 2.1 zeigt exemplarisch eine SOAP-Nachricht, welche das Aktien-Tickersymbol von Google (GOOG) als Eingabe an einen Service sendet.

2.5.2 Representational State Transfer (REST)

Representational State Transfer (REST) beschreibt ein Programmierparadigma für Webservices, welches in den meisten Fällen mit Hilfe des HTTP-Protokolls implementiert wird. L. Richardson et al. definieren für das von Roy T. Fielding eingeführte Paradigma [FT00] vier Grundkonzepte und Eigenschaften, welche von einem Service implementiert werden müssen um als RESTFul zu gelten [RR07]:

Ressourcen Bedeutet das Aufspalten der vom Service nach außen hin verfügbaren Objekte in Ressourcen. Eine Ressource ist, alles was als eigenständiges Objekt bezeichnet werden kann.

URIs Jede Ressource muss durch einen eindeutigen Bezeichner identifiziert werden können. Im Falle des HTTP-Protokolls bieten sich Uniform Resource Identifier kurz URIs an.

Ressourcen Repräsentationen Jede Ressource wird durch eine konkrete Datenrepräsentation oder einer Repräsentation des Zustands des dahinter liegenden Objekts repräsentiert

HATEOAS „Hypermedia as the engine of application state“ definiert die Vernetzung aller Ressourcen, beispielsweise mit Hilfe von Hypermedia. Die Idee ist, dass der Zustand des Services nicht in der Anwendung, sondern in der REST-Schnittstelle gehalten wird und jede Ressource, wie ein endlicher Automat, auf den nächst möglichen Zustand verlinkt.

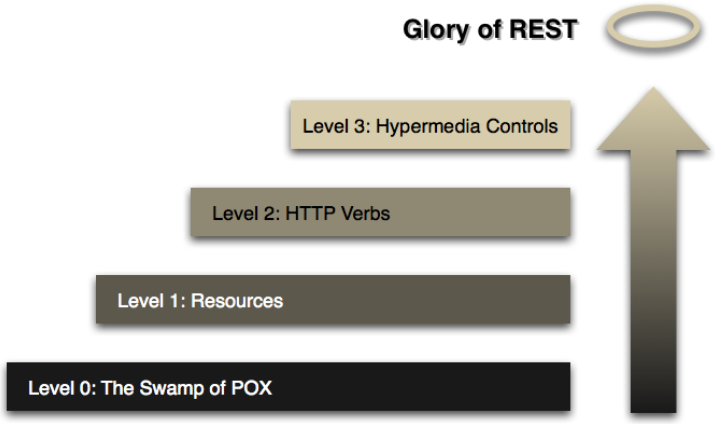


Abbildung 2.10: Level des Richardson Maturity Models [Fow10]

Des Weiteren soll auf folgende Eigenschaften Wert gelegt werden:

Adressierbarkeit Verlangt, dass die definierte URI bzw. die Schnittstelle des Services für den Nutzer ansprechbar gemacht wird. Der Nutzer muss also, beispielsweise bei der Umsetzung mit HTTP, einen eindeutigen Link bekommen, welcher ihn immer wieder zum Service weiterleitet.

Zustandslosigkeit Bedeutet, dass jeder Zugriff atomar ausgeführt wird und der Service nicht von weiteren Informationen, beispielsweise vorher getätigten Anfragen, abhängig ist, um die nächste Anfrage zu beantworten. Viel mehr resultieren identische Anfragen auch immer in einem identischen Zustand.

Zusammenhängigkeit Ist die Eigenschaft, welche sich durch die Umsetzung des HATEOAS-Konzepts ergibt. Alle Ressourcen und funktional ähnlichen Services sollen auf irgendeine Weise miteinander verknüpft sein.

Einheitliche Schnittstelle Jede Ressource soll eine einheitliche Schnittstelle mit fest definierten Operationen bereitstellen, welche für jede Ressource gleich sind. Im Falle von HTTP sind es die gängigen HTTP-Verben GET, PUT, POST, DELETE. Damit können Datenobjekte von Ressourcen gelesen, bearbeitet, hinzugefügt oder gelöscht werden.

Richardson Maturity Model

Das Richardson Maturity Model ist ein von Leonard Richardson definiertes [Ric08] und von Martin Fowler [Fow10] beschriebenes Reifegradmodell für RESTFul Webservices. Richardson klassifiziert hierbei REST-Services in vier verschiedene Kategorien. Die Kategorien sind absteigend geordnet und jede Kategorie bildet eine Teilmenge ihrer Oberkategorie, indem sie weitere Anforderungen an den untersuchten REST-Service stellt. Abbildung 2.10 stellt die einzelnen Ebenen dar. Die erste Kategorie *Level 0* beschreibt alle Services, welche zumindest das HTTP-Protokoll als Transportprotokoll nutzen, dies jedoch über einen Endpunkt und meist

REST	SOAP
Hauptunterschiede	
Ressourcen beschreiben sich selbst	Operationen sind in den WSDL Ports/Operations definiert
Jede Ressource hat einheitlich definierte Operationen	Operationen der Schnittstelle werden frei definiert
Komponenten sind lose gekoppelt	Komponenten sind eng gekoppelt
Einheitliche Adressierung der Ressourcen	Jede Operation besitzt eindeutige Adresse
Zustandslose Kommunikation	Zustandsändernde Kommunikation
Synchrone Kommunikation	Asynchrone Kommunikation
Beliebiges Nachrichtenformat	XML-Basiertes Nachrichtenformat
Vorteile	
Leichtgewichtige Nachrichten	Abstraktion von komplexen Operationen
Keine Routing Informationen nötig	Umwandeln von existierenden Schnittstellen einfach
Einfaches und schnell verständliches Paradigma	Sicherheitsmechanismen
Client seitige Entwicklung wesentlich einfacher	Unterstützung WS-Policy/Security/Reliablity
Nachteile	
Große Anzahl an Ressourcen	Client muss Operationen und deren Semantik kennen
Verwalten des URI-Namespaces kann umständlich werden	Schwergewichtige Nachrichten
(da) Contract zwischen Nutzer und Anbieter fehlt	Caching von Abfragen nicht möglich

Tabelle 2.1: Hauptunterschiede zwischen REST und SOAP (abgeändert aus [ZNS05])

nur für Remote Procedure Call (RPC). Fowler bezeichnet dieses Level als „Swamp of POX (Plain Old XML)“, also als Sumpf von Services, welche nur mit RPC und XML kommunizieren. *Level 1* beschreibt einen Services, welcher nicht mehr alle Anfragen an einem Endpunkt empfängt, sondern diese funktional auf verschiedene Ressourcen verteilt. Ist es in *Level 1* noch egal welches HTTP-Verb hierfür verwendet wird, so fordert *Level 2* das Verwenden von semantisch passenden HTTP-Verben. Zum Lesen von Daten soll zum Beispiel eine sichere Operation wie ein HTTP-GET verwendet werden. Zum Schreiben soll ein zustandsänderndes Verb wie zum Beispiel PUT oder POST verwendet werden. Die letzte Kategorie, *Level 3*, klassifiziert einen Service als RESTFul-Service, wenn er noch zusätzlich zu den anderen Kategorien das oben vorgestellte HATEOAS-Prinzip umsetzt. Folglich setzt nur ein Level 3 REST-Service alle Prinzipien des REST Paradigmas um und ist somit RESTFul.

2.5.3 Gegenüberstellung

Die SOAP und REST Paradigmen unterscheiden sich somit in vielerlei Hinsicht. Tabelle 2.1 listet deren Hauptunterschiede und Vor - und Nachteile auf. Je nach Anwendungsfall bietet es sich daher an, entweder eine SOAP-Schnittstelle oder eine REST-Schnittstelle zu verwenden. REST bietet sich perfekt für eine Anwendung mit limitierter Bandbreite und Ressourcen, in der eine Umsetzung von zustandslosen Operationen mit synchronen Aufrufen möglich ist und Caching umgesetzt wird. SOAP wiederum bietet sich für eine Anwendung an, in welcher asynchrone Aufrufe mit einem hohen Level an Sicherheit und Zuverlässigkeit umgesetzt werden müssen, formale Contracts/Verträge zwischen Nutzer und Anbieter nötig sind und auf Zustandslosigkeit verzichtet werden kann [Roz10].

2.6 Beschreibungssprachen

Wie bereits in Abschnitt 2.2 erwähnt, braucht es zum Speichern eines Services in einem Verzeichnisdienst eine Beschreibungssprache, welche die Schnittstelle des Services dokumentiert. Im besten Falle sollte diese Beschreibungssprache sowohl von einer Maschine, als auch von einem Menschen lesbar sein, in keinem Fall jedoch in unstrukturierter Form vorliegen. Nur so kann ein Servicenutzer nach einer bestimmten Funktionalität suchen und einen passenden Service vom Verzeichnisdienst zurückbekommen. Auch in der SOA Governance spielen Servicebeschreibungen, insbesondere beim Portfolio-Management und Metadata-Management, eine gewisse Rolle. Beim Metadata-Management helfen Servicebeschreibungen beim Einpflegen von neuen Services in ein Governance Repository. Beim Portfolio-Management helfen sie, um zum Beispiel zu überprüfen, ob eine Funktionalität schon einmal im Repository vorhanden ist [KSM14]. Dies kann unnötige Neuentwicklungen verhindern.

Um diesen Anforderungen gerecht zu werden, muss eine Servicebeschreibung zumindest drei Aspekte abdecken. *Was* für eine Funktionalität mit Hilfe der Beschreibung abgedeckt wird, *wie* auf den beschriebenen Dienst zugegriffen wird und *wo* die beschriebenen Funktionalitäten verfügbar sind [Wee05].

API-First Approach

Seitdem der API-First Approach als Designrichtung immer beliebter wird, nehmen Beschreibungssprachen eine teilweise völlig neue Rolle ein. Waren WSDL oder WADL eher dazu gedacht, Servicebeschreibungen aus schon fertigem Client -oder Servercode zu generieren, so werden beim API-First Approach schon während der Anforderungsanalyse Schnittstellen definiert und mit einer passenden Schnittstellenbeschreibungssprache beschrieben [Lan15]. Die Schnittstellenbeschreibung wird somit eine der wichtigsten Artefakte im Softwareentwicklungsprozess von Webservices. Aus ihr lassen sich die Grundgerüste für Client - und Servercode generieren, es lassen sich automatisch Tests schreiben und das entstehende Dokument dient sowohl als technische Spezifikation, als auch als Teil des Vertrags zwischen Servicenutzer und Serviceanbieter.

2.6.1 Webservice Description Language (WSDL)

Als exemplarische Beschreibungssprache und da deren Grundkonzepte im Verlauf der Arbeit relevant sind, wird an dieser Stelle WSDL als Beschreibungssprache für hauptsächlich auf SOAP basierende Webservices erläutert. WSDL ist eine auf XML basierende Spezifikation, zur Beschreibung von Webservices, basierend auf einem abstrakten Modell der Funktionalitäten, welche der Webservice bietet [Kop07b]. Spezifiziert wurde WSDL erstmals 2000 (WSDL 1.1) von der W3C mit dem Ziel RPC und Nachrichtenorientierte Kommunikation zu vereinen und

trotzdem verschiedenste Protokolle und Nachrichtenformate zu unterstützen [Wee05]. Die aktuelle Version ist WSDL 2.0.

Architekturkonzepte

WSDL baut auf sieben architektonischen Grundkonzepten auf [Wee05].

Erweiterbarkeit Das erste Konzept von WSDL ist es eine Möglichkeit zu schaffen, die Beschreibungssprache nach Belieben zu erweitern. Ein Beispiel hierfür ist die Erweiterung für Service Level Agreements.

Unterstützung verschiedener Typsysteme In WSDL können Typen mit Hilfe von XML Schemas definiert und in das WSDL Dokument eingebunden werden. Durch das Verwenden des W3C XML Schemas wird deutlich die Wahrscheinlichkeit erhöht, dass ein Service Consumer die geforderten Eingabe - und gelieferten Ausgabedaten versteht.

Abstraktion WSDL ist aufgeteilt in eine abstrakte Definition der Schnittstelle, der Definition des Kommunikationsprotokolls und in alle weiteren Information zum Zugriff auf den Webservice.

Einheitlicher Nachrichtenaustausch und RPC Hauptziel von WSDL ist es, sowohl Nachrichtenorientierte Systeme, als auch auf RPC basierende Systeme zu unterstützen. Dies wird durch die oben genannte Abstraktion gewährleistet.

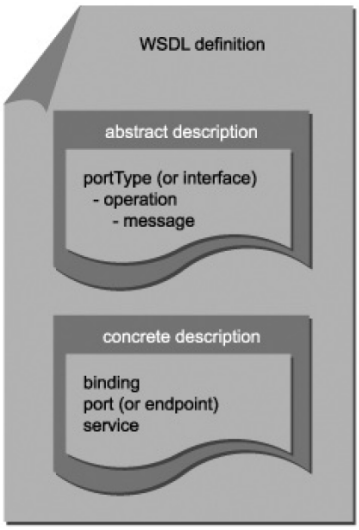
Unterstützung verschiedener Protokolle und Transport Layer Durch die Abstrahierung zwischen Schnittstelle und konkreter Realisierung ist WSDL nicht auf ein festes Protokoll oder eine feste Transportschicht festgelegt, sondern ermöglicht es, aufgrund abstrakter Schnittstellen, verschiedene Protokollbindungen zu verwenden.

Keine Darstellung von Workflows WSDL gibt keine feste Reihenfolge für das Ausführen der bereitgestellten Operationen eines Services vor. Somit werden auch keine Ablaufsequenzen, wie zum Beispiel die Reihenfolge von Operationen zum Login in ein System, in WSDL beschrieben.

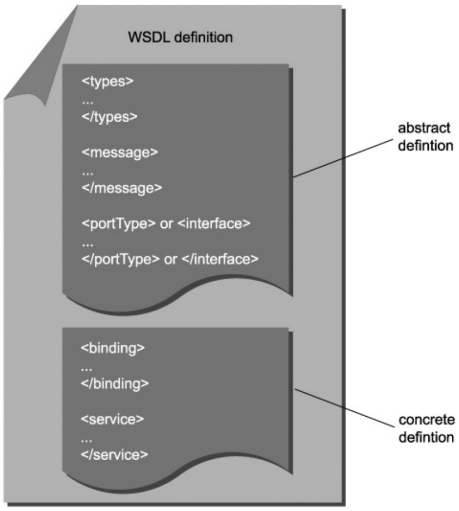
Keine Semantik WSDL bietet nur eine strukturelle Beschreibung des Services, nicht jedoch eine Möglichkeit zur semantischen Beschreibung.

Aufbau

In diesem Abschnitt wird nun näher auf die Struktur eines WSDL-Dokuments eingegangen. Durch die vorher genannte Abstraktion ist dieses in zwei Teile aufgeteilt. Die abstrakte Beschreibung des Webservices und die konkrete Beschreibung samt Definition des Kommunikationsprotokolls. Abbildung 2.11 zeigt den strukturellen Aufbau eines WSDL Dokuments. Im weiteren Verlauf des Abschnitts wird sich auf die Version 1.1 der WSDL Spezifikation bezogen.



(a) Grobe Struktur eines WSDL 1.1/2.0 Dokuments



(b) Wichtigste XML-Elemente eines WSDL 1.1/2.0 Dokuments

Abbildung 2.11: Struktur eines WSDL Dokuments [Erl05]

Um die beiden Beschreibungen der Schnittstelle befindet sich das *definitions*-Element, das alle weiteren Elemente umschließt und alle im Dokument genutzten Namensräume beschreibt. Im *types*-Element werden die für das Verständnis der Schnittstelle notwendigen Datentypen definiert. Dies kann, wie schon beschrieben, mit Hilfe eines XML-Schemas geschehen. Das darauf folgende *message*-Element beschreibt jede vom Service zu empfangende oder zu sendende Nachricht. Es enthält ein *part*-Element, welches abstrakt den Inhalt, also den Datentyp, der jeweiligen Nachricht beschreibt. Das letzte Element der abstrakten Beschreibung ist das *portType*-Element. In ihm werden die einzelnen, aufrufbaren Funktionen des Services beschrieben. Hierbei legen *input*-und *output*-Elemente fest, welche vorher definierten Nachrichten als Parameter dienen. Die Reihenfolge der einzeln beschriebenen Operationen hat keine Bedeutung. Mit dem *binding*-Element fängt die konkrete Beschreibung an. Hier wird ein konkretes Protokoll und beispielsweise bei einem Nachrichtenorientierten Dienst, das Eingabe - und Ausgabeformat definiert. Durch das Konzept der Erweiterbarkeit können hier beliebig definierte Bindings genutzt werden. Es ist lediglich wichtig, dass diese wiederum in einem XSD definiert wurden. Für eine Serviceorientierte Architektur mit Webservices wird meistens SOAP als Protokoll und somit das SOAP-Binding XML-Schema[9] verwendet. Im *service* Element wird dann der Endpunkt (nachrichtenorientiert) oder der Port des RPC Clients definiert.

[9]XSD Soap Binding: http://schemas.xmlsoap.org/soap/http/

3 Analyse verschiedener Beschreibungssprachen

Um im nachfolgenden Verlauf der Arbeit eine Import/Export-Möglichkeit für Schnittstellenbeschreibungen von REST/SOAP Webservices in das SOA Governance Repository zu realisieren, beschäftigt sich der erste Teil dieser Arbeit mit der Analyse infrage kommender Beschreibungssprachen.

Hierbei ist WSDL als Beschreibungssprache für SOAP Webservices ein anerkannter Standard. Für REST Webservices existiert jedoch, gerade im Unternehmensumfeld noch immer kein allgemein angewandter Standard. Grund dafür sind zum größten Teil nicht funktionale Anforderungen an die Beschreibungssprache, welche durch die sich eigentlich selbst beschreibende REST Schnittstelle nicht umgesetzt werden. Ein Beispiel hierfür wäre die Notwendigkeit, schon in frühen Phasen des Service Lifecycles, eine Möglichkeit zu haben Beschreibungssprachen als Kommunikationsgrundlage für die Entwicklung des Services zu nutzen. Somit gilt es bestenfalls eine Beschreibungssprache für REST Schnittstellen zu finden, welche zum einen die gleichen Funktionalitäten wie WSDL mit sich bringt, aber auch die Vorteile des REST Paradigmas, wie zum Beispiel Leichtgewichtigkeit, beibehält. Wichtig ist es, den Fokus auf die Eigenschaften einer etablierten Serviceorientierten Architektur in und außerhalb eines Unternehmens nicht zu verlieren.

3.1 Vorstellung der Beschreibungssprachen

Im Folgenden werden kurz die verbreitetsten Beschreibungssprachen, mit denen REST Schnittstellen beschrieben werden können, vorgestellt.

WSDL 2.0 Wie schon in Kapitel 2 beschrieben, handelt es sich bei WSDL um eine auf XML basierende Spezifikation zur Beschreibung von Webservices. WSDL 2.0 [CMRW07] ist eine Überarbeitung der WSDL 1.1 Spezifikation, wobei ein größeres Augenmerk auf Simplizität und Nutzerfreundlichkeit, als in der mächtigeren Version 1.1, gelegt wurde [Wee05].

WADL war eine der ersten von Maschinen lesbaren Beschreibungssprachen für auf HTTP basierende, Web Applikationen. Die Web Application Description Language (WADL) baut genauso wie WSDL auf XHTML auf und wurde 2009 vom World Wide Web Consortium

(W3C) spezifiziert [Had09]. Der größte Unterschied zu WSDL ist der Weg weg von einer Abstrakten Beschreibung, hin zu einer von Anfang an konkreten Beschreibung des Services.

Open API Spezifikation auch bekannt als Swagger[1], ist eine 2011 von SmartBear[2] entwickelte auf der JavaScript Object Notation (JSON) und YAML Markup Language (YAML) basierende Sprache für das Beschreiben und Dokumentieren von RESTful APIs [Ope14]. Ein großer Wert wird hierbei auf die Bereitstellung verschiedenster Werkzeuge zur Codegenerierung, API-Design oder der graphischen Visualisierung der API gelegt. Seit Januar 2016 ist Swagger in die Open API Initiative[3] übergegangen und wird unter dem Namen Open API Spezifikation weiterentwickelt.

RAML Bei RAML[4] handelt es sich um eine 2013 von der RAML Workgroup ins Leben gerufene Sprache zum Dokumentieren von auf HTTP basierenden APIs, welche fast ausschließlich oder zu großen Teilen die Prinzipien des REST Programmierparadigmas umsetzen. RAML baut auf der YAML 1.2 Spezifikation [BEN11] auf und ermöglicht es, mit Hilfe des YAML Syntax vorherige genannte Schnittstellen zu beschreiben.

API Blueprint API Blueprint[5] ist eine 2013 spezifizierte, auf dem Github Syntax, Markdown [GS04], basierende Beschreibungssprache. API Blueprint hat durch ihre hohe Lesbarkeit zum Ziel, der Fachlichkeit einen möglichst einfachen Zugang zur technischen Spezifikation der Schnittstelle zur ermöglichen. Schon am Anfang des Service Lifecycles soll ein Austausch über die Schnittstelle geschaffen werden, damit diese als Implementierungsgrundlage verwendet werden kann.

3.2 Vergleichskriterien

Um jede einzelne Beschreibungssprache allgemein und unabhängig voneinander vergleichen zu können, werden in diesem Abschnitt verschiedene Vergleichskriterien vorgestellt. Anhand eines exemplarischen Beispiel-SOAP-Services wird die Eignung der Beschreibungssprache zum Beschreiben dessen Schnittstelle als REST-Umsetzung untersucht. Die zu untersuchenden Kriterien sind in funktionale und nicht funktionale Kriterien unterteilt. Die umgesetzten Funktionalitäten der WSDL 1.1 Spezifikation dienen als Vergleichswert und es wird untersucht, inwiefern die anderen Beschreibungssprachen diese Funktionalitäten umsetzen. Ist in den Kriterien allgemein von WSDL die Rede, ist die beschriebene Funktionalität in beiden WSDL Spezifikationen (Version 1.1 und Version 2.0) vorhanden.

[1] Open API Spezifikation/Swagger http://swagger.io
[2] SmartBear https://smartbear.com/
[3] Open API Initiative: https://openapis.org/
[4] RAML http://raml.org/
[5] API Blueprint http://apiblueprint.org

3.2.1 Funktionale Kriterien

In diesen Kriterien sollen die gewählten Beschreibungssprachen hinsichtlich ihres Funktionsumfangs und ihrer Mächtigkeit untersucht werden. Es wird nicht auf die Unterstützung verschiedener Protokolle eingegangen, da es für eine Beschreibungssprache einer REST Schnittstelle ausreichend ist, wenn explizit nur REST mit HTTP als Protokoll unterstützt wird. Trotzdem ist zu erwähnen, dass WSDL verschiedene Arten von Protokollen unterstützt, wie zum Beispiel SOAP über HTTP oder RMI über TCP. Des Weiteren wird davon ausgegangen, dass bei der Vorauswahl einer Beschreibungssprache schon die Gewährleistung der Grundfunktionalitäten berücksichtigt wurde und somit nur noch untersucht wird, in welcher Weise diese im Vergleich zu den anderen Beschreibungssprachen umgesetzt wurden. Grundfunktionalitäten bedeutet, dass es prinzipiell möglich ist, mit der zu untersuchenden Beschreibungssprache einen REST Service zu beschreiben.

Hauptfunktionalitäten Die Hauptaufgabe einer Beschreibungssprache ist es, die zu beschreibende Schnittstelle oder den Webservice eindeutig zu definieren. Auf die in Abschnitt 2.5 beschriebenen Funktionalitäten des REST Paradigmas wird Bezug genommen. Hierbei müssen Fragen über die bereitgestellte Funktionalität des Services, die Art wie auf diese Funktionalität zugegriffen wird und der Ort also der Endpunkt des Services geklärt werden.

Eines der vorgestellten Prinzipien von WSDL ist es, diese drei Anforderungen voneinander zu entkoppeln und zu unterscheiden, *was* der Service tut (<interface>), *wie* mit dem Service interagiert wird (<binding>) und von *wo* der Service angeboten wird (<port>) [Wee05]. Der Vorteil besteht darin, dass für die gleiche Funktionalität eines Services verschiedene Interaktionsmöglichkeiten spezifiziert werden können.

Typsicherheit Ein weiteres Konzept, auf welchem WSDL aufbaut und auch jede weitere Beschreibungssprache aufbauen sollte, ist die Unterstützung von verschiedenen Typsystemen [Wee05]. Dies ist vor allem deswegen wichtig, da durch gegebene Typsicherheit eindeutig definiert werden kann, welchen Datentyp die zu beschreibende Schnittstelle als Parameter empfängt bzw. welchen sie auch wieder zurückgibt.

Versionierung Im Idealfall sollte eine einmal definierte Schnittstelle sich über einen längeren Zeitraum hinweg nicht ändern. Jedoch gehört das Spezifizieren der Schnittstelle eines Services zu den frühesten Designentscheidungen und es ist möglich, dass sich Anforderungen ändern. Kommt es dabei zu nicht rückwärts kompatiblen Änderungen an der Schnittstelle, so ist ein Konzept zur Versionierung notwendig. Auch in Bezug auf den Lebenszyklus eines Services oder während der Spezifikation im API-First-Approach, ist es nötig diesen in verschiedenen Versionen bereitzustellen. Für WSDL gibt es indessen keinen vordefinierten Standard. Jedoch ist es möglich, über das Nutzen eindeutig definierter „namespaces“ die WSDL Beschreibungen

verschiedenen Versionen zuzuordnen [BE04]. Hierbei ist die Namensgebung in der W3C XML Schema Definition einheitlich definiert [SCH12].

Semantische Beschreibung In der Arbeit von Bruder et al. [BHK13] werden Beschreibungssprachen auch auf ihre Funktionalität gegenüber der semantischen Beschreibung einer Schnittstelle hin verglichen. Nur wenige Beschreibungssprachen bieten eine Möglichkeit, semantischen Kontext mit zu beschreiben. WSDL grenzt sogar seine Spezifikation auf die rein strukturelle Beschreibung der Schnittstelle ab [Wee05]. Im Zuge der Arbeit soll eine Möglichkeit implementiert werden, die Daten einer Schnittstellenbeschreibung in das gegebene RDF Modell des Governance Repositories zu überführen. Durch eine Überführung der Beschreibungen verschiedenster Services in einen RDF Graphen bietet sich die Möglichkeit, Beziehungen zwischen Ressourcen oder Operationen zu definieren und somit der Schnittstellenbeschreibung eine semantische Bedeutung zu geben. Dieses Kriterium zielt somit darauf ab die Beschreibungssprache auf ihre Funktionalität zur semantischen Beschreibung einer Schnittstelle und mehr noch auf ihre Eignung zur Überführung in ein einheitliches RDF Modell zu untersuchen.

3.2.2 Nichtfunktionale Kriterien

Lesbarkeit Mit diesem Kriterium sollen die Beschreibungssprachen, welche anfänglich eher als maschinenlesbare formale Definitionen entworfen wurden, auf ihre Lesbarkeit durch einen Menschen untersucht werden. Gerade im Unternehmensumfeld kommen nicht immer nur technikaffine Nutzer mit funktionalen Beschreibungen von Services in Kontakt. Ist die an sich formale Beschreibungssprache auch ohne Probleme für einen Menschen lesbar, kann diese beispielsweise als Grundlage für die Interaktion mit dem Fachbereich verwendet werden, ohne dass weiterer dokumentarischer Aufwand betrieben werden muss. Gerade im Zuge der DevOps Bewegung wird eine Beschreibungssprache benötigt, welche in kurzer Zeit erlernbar und mit noch weniger Zeitaufwand lesbar ist. Zudem muss deren Inhalt leicht verständlich sein. Bruder et al. [BHK13] beschreiben die Klärung von zwei Fragen als Bewertungsgrundlage. Zum einen der Anteil des syntaktischen Overheads, welcher so gering wie möglich sein sollte und zum anderen wie intuitiv den einzelnen Konstrukten ihre Bedeutung angesehen werden kann. Ein weiterer zu untersuchender Punkt ist die Untersuchung hinsichtlich einer alternativen Darstellung, welche die Lesbarkeit der Sprache erhöht.

WSDL lässt sich als eine für den Menschen sehr schwer lesbare Sprache einstufen. Durch die Verwendung von XHTML und die Abstrahierung der Schnittstelle in verschiedene Bestandteile entsteht ein sehr hoher sowohl syntaktischer als auch semantischer Overhead, welcher es schwer macht auf den ersten Blick die Schnittstelle zu verstehen. Es kann sein trotz eines guten Verständnisses der WSDL Spezifikation die Beschreibung des Services nur zu Teilen oder gar nicht verstanden werden. Dies ist auf die hohe Erweiterbarkeit der Spezifikation mit Elementen, wie SLAs oder unbekannten Datenypen zurückzuführen.

Verständlichkeit Das Kriterium Lesbarkeit soll vor allem im Hinblick darauf untersucht werden, in wie fern ein Nutzer die angebotene Funktionalität der Schnittstelle nach außen hin mit Hilfe einer Beschreibungssprache versteht. In diesem Kriterium geht es um die Untersuchung wie die, hinter einem Service liegende, Funktionalität anhand der Beschreibung erkannt werden kann. Bekommt der Leser nur einen strukturellen Eindruck der Schnittstelle oder versteht er, welche Anwendungsfälle umgesetzt werden. Ist letzteres der Fall kann ohne eine existierende Implementierung oder das Einlesen in eine teils komplexe Spezifikation die Funktionalität des Services erkannt werden. Dies erleichtert zu großen Teilen die Arbeit eines Entwicklers und dessen Austausch mit anderen Stakeholdern im Unternehmen.

WSDL 1.1 bietet ein optionales Tag an mit dessen Hilfe für den Menschen lesbare Beschreibungen erstellt werden können. Ansonsten besteht lediglich die Möglichkeit, mit Hilfe von sprechenden Bezeichnern die Funktionalität der Schnittstelle zu beschreiben.

Erweiterbarkeit Eines der Grundkonzepte auf denen WSDL entwickelt wurde, ist das Konzept der Erweiterbarkeit der Beschreibungssprache [Wee05]. In diesem Kriterium soll die Beschreibungssprache hinsichtlich ihrer Erweiterbarkeit untersucht werden. Gerade beim Beschreiben von SOA spezifischen Anforderungen, wie zum Beispiel WS-Policies [VOH+07], spielt dieses Kriterium eine große Rolle.

WSDL ermöglicht es, mit Hilfe von sogenannten Extensibility-Elementen die Spezifikation an definierten Stellen beliebig zu erweitern. Mit WSDL ist es möglich, durch die Erweiterbarkeit der Sprache jedes beliebige Nachrichten- und Netzwerkprotokoll zu verwenden und somit die Sprache mit bis dato noch nicht etablierten Protokollen zu erweitern.

Dokumentation & Spezifikation Vor allem bei der Spezifikation einer formalen Beschreibungssprache, ist es besonders wichtig eine eindeutig interpretierbare Dokumentation vorzufinden, welche alle aufkommenden Fragen klärt. Nur dies ermöglicht es, die zu untersuchende Sprache als Standard anzuerkennen und ohne Absprache mit anderen Entwicklern eine unmissverständliche Schnittstellenbeschreibung bereit zu stellen. Sind bestimmte Punkte in der Dokumentation bzw. Spezifikation unklar definiert, so kann es zu fehlerhaften Schnittstellenbeschreibungen und schlimmer, zu fehlerhaften Contracts zwischen Serviceprovider und Consumer kommen. Auch Informationen und Materialien außerhalb der Spezifikation, welche zum besseren Verständnis der Beschreibungssprache beitragen sind wichtig.

Toolunterstützung Neben einer Spezifikation für die Beschreibungssprache bieten viele Anbieter auch Tools an, welche auf Basis der Spezifikation entwickelt wurden. Diese Werkzeuge ermöglichen zum Beispiel das durch Annotationen im Quellcode oder graphisch unterstützte Erstellen einer Schnittstellen. Auch Werkzeuge zum syntaktischen Parsen der Beschreibung werden oft angeboten. Solch eine Toolunterstützung erhöht die Attraktivität der Spezifikation und verbreitert die Reichweite der Beschreibungssprache.

Verbreitung der Sprache In diesem Kriterium wird die Beschreibungssprache auf ihren Einsatz in der Praxis untersucht. Es ist wichtig zu untersuchen, inwiefern die Sprache bei den verschiedensten Stakeholdern bekannt, verwendet oder auch nur berücksichtigt wird. Des Weiteren ist es von Vorteil eine Beschreibungssprache zu verwenden, welche schon Einzug in Unternehmen gefunden hat und diese schon Erfahrungen unter der Verwendung in einem großen Software-Ökosystem aufweisen können. Dies ist vor allem bei der Dokumentation von REST-Schnittstellen von Bedeutung, da es hier noch keinen einheitlichen Standard gibt und es von Vorteil ist sich auf eine Beschreibungssprache zu einigen, welche schon eine große Akzeptanz und Verbreitung gefunden hat.

3.3 Beispielservice - Stockquoteservice

Nachstehend wird exemplarisch der Beispielservice aus der WSDL 1.1 Spezifikation aufzeigt [CCMW01]. Er bietet eine Funktionalität zum Abrufen von Aktienpreisen für ein gegebenes Aktien-Tickersymbol.

3.3.1 WSDL1.1-Beschreibung des Beispielservices

Nachfolgend wird die WSDL-1.1-Beschreibung des Beispielservices dargestellt. Wie in Abschnitt2.6.1 erklärt, fängt die Beschreibung in WSDL mit dem alles umschließenden definitions-Operator an.

```
<?xml version="1.0"?>
<definitions name="StockQuote"
          targetNamespace="http://example.com/stockquote.wsdl"
          xmlns:tns="http://example.com/stockquote.wsdl"
          xmlns:xsd1="http://example.com/stockquote.xsd"
          xmlns:soap="http://schemas.xmlsoap.org/wsdl/soap/"
          xmlns:wsdl="http://schemas.xmlsoap.org/wsdl/">
```

Nun werden die verwendeten Datentypen definiert. Im Beispiel existieren zwei einfache Typen. Der *TradePriceRequest*, welcher aus einem String, dem Aktien-Tickersymbol besteht und der TradePrice, welcher aus einem float, dem price besteht. Hierbei handelt es sich um normale XSD-Schemas welche auch importiert werden können.

```
<wsdl:types>
  <xsd:schema targetNamespace="http://example.com/stockquote.xsd"
       xmlns:xsd="http://www.w3.org/2000/10/XMLSchema">
     <xsd:element name="TradePriceRequest">
       <xsd:complexType>
          <xsd:all>
             <xsd:element name="tickerSymbol" type="string"/>
          </xsd:all>
       </xsd:complexType>
     </xsd:element>
     <xsd:element name="TradePrice">
       <xsd:complexType>
          <xsd:all>
             <xsd:element name="price" type="float"/>
          </xsd:all>
       </xsd:complexType>
     </xsd:element>
  </xsd:schema>
</wsdl:types>
```

Anschließend werden die möglichen Nachrichten des Services beschrieben. Der Beispielservice hat eine *GetLastTradePriceInput*-Nachricht, welche aus *TradePriceRequests* besteht und eine *GetLastTradePriceOutput*-Nachricht, welche aus *TradePrices* besteht.

```
<wsdl:message name="GetLastTradePriceInput">
   <wsdl:part name="body" element="xsd1:TradePriceRequest"/>
</wsdl:message>
<wsdl:message name="GetLastTradePriceOutput">
   <wsdl:part name="body" element="xsd1:TradePrice"/>
</wsdl:message>
```

An dieser Stelle werden die vorher abstrakt definierten Nachrichten und Typen im portType an eine WSDL-Operation gebunden. *GetLastTradePriceInput* wird als eingehende, *GetLastTradePriceOutput* als ausgehende Nachricht an die Operation *GetLastTradePrice* gebunden. In Folge nimmt die Operation *GetLastTradePrice* ein Aktiensymbol als Eingabe und liefert den Preis als Ausgabe.

```
<wsdl:portType name="StockQuotePortType">
   <wsdl:operation name="GetLastTradePrice">
     <wsdl:input message="tns:GetLastTradePriceInput"/>
     <wsdl:output message="tns:GetLastTradePriceOutput"/>
   </wsdl:operation>
</wsdl:portType>
```

Jetzt kommt die konkrete Beschreibung des Services. Im binding-Operator wird die genannte Operation *GetLastTradePrice* an eine SOAP-Operation gebunden.

```
<wsdl:binding name="StockQuoteSoapBinding"
          type="tns:StockQuotePortType">
   <soap:binding style="document" transport="http://schemas.xmlsoap.org/soap/http"/>
   <wsdl:operation name="GetLastTradePrice">
     <soap:operation soapAction="http://example.com/GetLastTradePrice"/>
     <wsdl:input>
        <soap:body use="literal"/>
     </wsdl:input>
     <wsdl:output>
        <soap:body use="literal"/>
     </wsdl:output>
   </wsdl:operation>
</wsdl:binding>
```

Daraufhin wird der anzusprechende Endpunkt in der service-Operator beschrieben.

```
<wsdl:service name="StockQuoteService">
   <wsdl:documentation>Returns the TradePrice for a given Stockquote</documentation>
   <wsdl:port name="StockQuotePort"
           binding="tns:StockQuoteBinding">
     <soap:address location="http://example.com/stockquote"/>
   </wsdl:port>
</wsdl:service>
```

3.3.2 Überführung in REST-Service

Um den vorgestellten Service in einen REST-Service umzuwandeln, bieten sich verschiedenste Möglichkeiten. Am trivialsten wäre es, die beschriebene Operation einfach auf eine REST-Ressource zu mappen. Am besten ist jedoch das RESTful Paradigma umzusetzen, wenn ein zum Service passendes Ressourcen Modell entworfen wird. So könnte es eine Ressource */stockquotes* geben, welche mit Hilfe eines HTTP-GETs alle in der Datenbank befindlichen Aktien-Tickersymbole zurück gibt und eine weitere Ressource */stockquotes/{TickerSymbol}*, welche für das im Pfad mitgegebene Aktien-Ticker-Symbol den Preis zurückgibt. Die Host Url ist *http://example.com*, versioniert wird über die Host-URL bzw. über einen Pfad relativ zur Host-URL und der REST Ressource. Ein curl-Befehl auf die erstellte Schnittstelle mit dem Google Tickersymbol *GOOG* sähe wie folgt aus:

```
curl -X GET http://example.com/v1/stockquotes/GOOG
```

Auffallend ist, dass es schwer fällt sich ohne passende Beschreibungssprache über die entstandene Schnittstelle zu unterhalten. Ein Austausch über Elemente wie Datentypen, relative Pfade oder Methoden zur Versionierung kann somit ohne entsprechende Sprachkonstrukte zum Hindernis werden.

3.4 Untersuchung der Beschreibungssprachen

Nachfolgend werden die einzelnen Sprachen auf die vorgestellten Kriterien hin untersucht. Dazu wurde der, zu einem REST-Service umgewandelte, Beispielservice aus Abschnitt 3.3 in allen Beschreibungssprachen beschrieben.

3.4.1 WSDL 2.0

WSDL 2.0 ähnelt stark seinem Vorgänger WSDL 1.1. Die Spezifikation von WSDL 2.0 ist unter der W3C Document License [W3C15] veröffentlicht und für jeden nutzbar. Hauptunterschiede zu WSDL 1.1 sind [Dhe04]:

- Möglichkeit, semantische Beschreibungen zu ergänzen
- Entfernen der *message* Komponente
- Umbenennung des *portType* Operators in den *interface* Operator
- Unterstützung von Vererbung von Schnittstellen
- Überarbeitung des *service* Operators
- Unterstützung aller HTTP-Methoden

WSDL 2.0 versucht Simplizität in die Spezifikation zu bringen und die Sprache für das Beschreiben von REST-Service zu ermächtigen.

Funktionale Kriterien

WSDL Version 2.0 hat neben den möglichen semantischen Annotationen und den HTTP-Bindings, ungefähr den gleichen funktionalen Umfang wie Version 1.1. Was weg fällt ist die Möglichkeit Nachrichten im **message** Operator zu definieren. Stattdessen werden im **interface** Operator, dem Ersatz für den **portType** Operator, die vorher definierten **types** direkt als Parameter angegeben. Folgendes Codebeispiel zeigt dies für den StockquoteService. Die Fragen nach der Funktionalität des Services, der Art des Zugriffes und dem Zugriffsort werden genauso wie in WSDL 1.1 entkoppelt voneinander beantwortet.

```
<wsdl:interface name="StockQuoteInterface">
      <wsdl:operation name="getLastTradePrice" pattern="http://www.w3.org/ns/wsdl/in-out"
          style="http://www.w3.org/ns/wsdl/style/iri">
        <wsdl:input element="xsd1:TradePriceRequest"/>
        <wsdl:output element="xsd1:TradePrice"/>
      </wsdl:operation>
 </wsdl:interface>
```

Das nachfolgende Codebeispiel zeigt die Verwendung des HTTP-Bindings. Hier wird ein GET-Befehl auf das Interface **GetLastTradePrice** aufgerufen.

```
 <wsdl:binding name="StockQuoteHTTPBinding"
              type="http://www.w3.org/ns/wsdl/http" interface="tns:StockQuoteInterface">
      <wsdl:operation ref="tns:getLastTradePrice" http:method="GET" />
</wsdl:binding>
```

Nachfolgend wird das definierte binding im schon bekannten **service** Operator an einen Endpunkt gebunden. Großer Nachteil ist, dass path-Parameter an der Endpunkt-URL nicht definiert werden können. Auffallend ist der **documentation** Operator, welcher überall verwendet werden kann, um bestimmte Abschnitte mit dokumentarischen Inhalten zu versehen.

```
<wsdl:service name="StockQuoteService" interface="tns:StockQuoteInterface">
    <wsdl:documentation>Returns the TradePrice for a given Stockquote</wsdl:documentation>
    <wsdl:endpoint name="StockQuoteHTTPEndpoint"
               binding="tns:StockQuoteHTTPBinding"
                   address="http://example.com/stockquotes/getTradePrice">
    </wsdl:endpoint>
</wsdl:service>
```

Typsicherheit Wie auch in WSDL 1.1 können in WSDL 2.0, neben den Standard XML-Schema-Typen, feste Datentypen definiert werden. Die Verwendung von weiteren Schemasprachen ist möglich, sofern diese die Funktionalität unterstützen, in die WSDL-Beschreibung importiert oder direkt geschrieben zu werden. Die Spezifikation empfiehlt jedoch die Verwendung von XML-Schemas [CMRW07].

Versionierung Prinzipiell wird die Versionierung von Schnittstellen mit Hilfe von WSDL 2.0 nicht unterstützt. Wie schon im Kriterienkatalog beschrieben, existiert jedoch die Möglichkeit, genauso wie bei WSDL Version 1.1, die Schnittstelle durch das Definieren von eindeutigen Namespaces zu versionieren [BE04]. Jedoch ist dies nur ein Lösungsvorschlag, um das Problem der fehlenden Versionierung zu lösen und kein einheitlich spezifizierter und angewandter Standard.

Semantische Beschreibung Zum einen können ab WSDL 2.0 die einzelnen Elemente innerhalb der Beschreibung mit Annotationen versehen werden [FL07], zum anderen existiert eine Spezifikation des W3C zum Überführen eines WSDL2.0-Dokuments in ein RDF-Datenmodell [Kop07b]. Ersterer Ansatz ermöglicht das Erweitern der strukturellen Beschreibung mit Referenzen auf semantische Modelle, wie zum Beispiel Ontologien. Zweiterer Ansatz das Integrieren und Verlinken der existierenden Beschreibung in andere RDF-Datenmodelle.

Nichtfunktionale Kriterien

Lesbarkeit Wie bereits beschrieben, ist WSDL 1.1 eine für den Menschen relativ schwer lesbare Beschreibungssprache. WSDL 2.0 macht rein syntaktisch keinen Unterschied und versucht, durch die genannten Veränderungen im Aufbau des Dokuments Einfachheit in die Sprache zu bekommen. Dies fällt jedoch nur beim direkten Vergleich zwischen WSDL 1.1 und 2.0 auf. Ansonsten ist Version 2.0 immer noch sehr schwer lernbar und lesbar.

Verständlichkeit Genauso wie in WSDL 1.1 gibt es auch in WSDL 2.0 die **documentation** Komponente, mit welcher Hintergrundinformationen zu den einzelnen Elementen beschrieben werden können. Diese Komponente ist neben der Verwendung von sprechenden Bezeichnern, die einzige Möglichkeit, Informationen über die dahinter liegende Funktionalität des Services herauszufinden. Die unübersichtliche Darstellung von Datentypen und Operationen trägt des Weiteren negativ zum Verständnis der bereitgestellten Funktionalität des Service bei.

Erweiterbarkeit Wie schon im Kriterium erläutert, bietet WSDL Erweiterungsmöglichkeiten für Nachrichten- und Netzwerkprotokolle, WS-Policies und Funktionalitäten außerhalb der Spezifikation mittels WSDL-Extensions.

Dokumentation & Spezifikation Zusammenfassend lässt sich sagen, dass die WSDL 2.0 Spezifikation vollständig und unmissverständlich spezifiziert ist. Jedoch wird die Vorkenntnis verschiedenster anderer Spezifikationen vorausgesetzt. So muss der Leser nicht nur Vorwissen über XML, XML-Schemas sondern auch über konkrete Schema-Definitionen, wie die HTTP-Binding oder SOAP-Binding Extensions, mitbringen.

Toolunterstützung WSDL 2.0 hat verschiedene Werkzeuge zum Generieren von WSDL-Beschreibungen aus Sourcecode, bzw. andersherum. Die bekannteste Bibliothek hierfür ist Apache Axis2/Java[6]. Darüber hinaus gibt es verschiedene graphische Editoren, wie zum Beispiel der Altova WSDL Editor[7]. Mit ihm lassen sich nicht nur WSDL-Dokumente graphisch bearbeiten, sondern auch WSDL 1.1 in WSDL 2.0 Dokumente umwandeln. Trotz der relativ guten Toolunterstützung ist zu erwähnen, dass fast alle Werkzeuge nur Notgedrungen aus der Empfehlung durch die W3C entstanden sind und eigentlich für WSDL 1.1 gedacht waren.

[6] Apache Axis2/Java http://axis.apache.org/axis2/java/core/index.html
[7] http://www.altova.com/de/xmlspy/wsdl-editor.html

Verbreitung der Sprache Im Gegensatz zu WSDL 1.1 findet Version 2.0 nicht die notwendige Verbreitung welche sie zu einem etablierten Standard machen würde. Die Unterschiede zwischen WSDL 1.0 und WSDL 2.0 sind bei der Beschreibung von SOAP-Services nicht ausreichend um auf Version 2.0 umzusteigen, und bei der Beschreibung von REST-Services ist die Erweiterung des HTTP-Bindings eher ein Lösungsversuch als eine wirklich gute Umsetzung. Somit ergeben sich zu wenige Vorteile, welche, einen kostenintensiven Infrastrukturwechsel von einer Unterstützung der Version 1.1 auf eine Unterstützung der Version 2.0 rechtfertigen würden [Wee05]. Des Weiteren ist WSDL eher dafür ausgelegt spezifische Operationen, als generische Schnittstellen zu beschreiben [Til15]. Dies wiederum führt dazu, dass diese Beschreibungssprache keine Akzeptanz bei der Entwicklung von REST-APIs hervorruft.

3.4.2 WADL

Wie schon angedeutet, ist WADL [Had09] die erste richtige Antwort auf der Suche nach einer Beschreibungssprache für REST-Services. Großer Unterschied zu WSDL ist, dass WADL keine Mechanismen zum Abstrahieren zwischen Funktionalität und Kommunikationsart bietet, sondern von Grund auf für das HTTP Protokoll und somit für REST-Services ausgelegt ist. Die WADL-Spezifikation fällt genauso wie die WSDL-Spezifikationen unter die W3C Document License [W3C15] und ist somit auch für jeden frei nutzbar.

Funktionale Kriterien

Hauptfunktionalitäten Im Vergleich zu WSDL ist WADL weniger umfangreich, da explizit nur das REST Paradigma unterstützt wird. Die Art der Kommunikation erfolgt hierbei, wie bei den folgenden Beschreibungssprachen, über das HTTP-Protokoll. Die Fragen nach der Funktionalität und dem Ort, an dem diese bereitgestellt wird, sind nicht voneinander entkoppelt. Trotzdem wird im Gegensatz zu WSDL zwischen Ressourcen und der Host-URL unterschieden. Eine Host-URL wird allerdings nicht näher definiert.

WADL ist in zwei Bestandteile unterteilt. Der erste Teil besteht aus der **resources** Komponente, in welchem die Ressourcen und ihre dazugehörigen HTTP-Operationen beschrieben werden. Der zweite Teil besteht aus dem **reprensentation** Operator, mit welchem Formate für Zustände und Parameter der Ressourcen definiert werden können. Listing 3.1 zeigt den Beispielservice in WADL beschrieben.

Listing 3.1 In WADL beschriebener Beispielservice

```
<application xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
    xmlns:xsd="http://www.w3.org/2001/XMLSchema" xmlns="http://wadl.dev.java.net/2009/02">
 <doc title="StockQuoteService" xml:lang="en">Get several Stockquotes</doc>
 <resources base="http://example.com/v1">
  <resource id="StockquotesById" path="/Stockquotes/{TickerSymbol}">
   <method id="StockquotesByTickerSymbol" name="GET">
    <doc title="StockquotesByTickerSymbol" xml:lang="en">Returns the TradePrice for a
        given Stockquote TickerSymbol</doc>
    <request>
     <param name="TickerSymbol" type="xsd:string" required="true">
      <doc title="TickerSymbol" xml:lang="en">A TickerSymbol of a Stockquote.</doc>
     </param>
    </request>
    <response status="200">
     <representation mediaType="application/json" href="#TradePrice" />
    </response>
    <response status="500">
     <doc title="500" xml:lang="en">Unexpected error</doc>
           <representation mediaType="application/json" href="#Error" />
    </response>
   </method>
  </resource>
 </resources>

 <representation id="TradePrice" mediaType="application/json">
  <doc title="TradePrice" xml:lang="en"></doc>
  <param name="Price" type="xsd:float">
   <doc title="Price" xml:lang="en">The price of the given StockQuote TickerSymbol</doc>
  </param>
 </representation>
 <representation id="Error" mediaType="application/json">
  <doc title="Error" xml:lang="en"></doc>
  <param name="code" type="xsd:integer">
   <doc title="code" xml:lang="en"></doc>
  </param>
  <param name="message" type="xsd:string">
   <doc title="message" xml:lang="en"></doc>
  </param>
 </representation>
</application>
```

Typsicherheit WADL bietet ein anderes Konzept zur Typisierung von Eingabe und Ausgabeparametern bzw. den Zuständen der definierten Ressourcen als WSDL an. Mit dem **reprensentation** Element können alle möglichen Datenrepräsentationen zum Beschreiben der Parameter einer Operation auf einer Ressource und deren Zuständen definiert werden. Großer Vorteil zu WSDL ist, dass durch das *mediaType* Attribut nicht nur XML, sondern alle möglichen MIME-Typen definiert werden können.

Versionierung Die Versionierung der Schnittstelle in WADL erfolgt lediglich durch das Hinzufügen einer Versionsnummer zur Stamm-URL des Services oder, wie bei WSDL, durch das Setzen von eindeutigen Namespaces. Einen **version** Operator, welcher die Version vom Endpunkt weg abstrahieren würde, gibt es nicht.

Semantische Beschreibung Im Gegensatz zu WSDL existiert für WADL nicht die Option semantische Annotationen im Code zu hinterlegen. Des Weiteren besteht keine Spezifikation zum Überführen einer WADL-Beschreibung in einen RDF-Graphen.

Nichtfunktionale Kriterien

Lesbarkeit Da WADL genauso wie WSDL auf XML aufbaut, ist auch WADL relativ schwer zu lesen. Aufgrund der jedoch wesentlich geringeren Komplexität der Sprache und die eindeutig und sprechend gewählten Bezeichner bekommt der Leser trotzdem einen guten Überblick über die beschriebene Schnittstelle. Das Prinzip der Definition der Ein - und Ausgabeparameter lässt sich gut lesen und in den entsprechenden MIME-Typ überführen.

Verständlichkeit Wegen des immer noch vorhandenen syntaktischen Overheads ist es sehr schwer das hinter der Schnittstelle liegende Datenmodell und somit auch die Funktionalität des Services zu verstehen. Jedoch helfen genauso wie bei WSDL sprechende Bezeichner und der **documentation** Operator weitere Informationen abzugeben.

Erweiterbarkeit WADL kann, genauso wie WSDL, sehr gut mit weiteren Schemas erweitert werden. Jedoch wird in der Spezifikation ausdrücklich darauf hingewiesen, dass die meisten WADL-Prozessoren keinerlei Erweiterungen unterstützen [Had09].

Dokumentation & Spezifikation Die WADL Spezifikation ist vollständig und schlüssig geschrieben. Alle Funktionalitäten sind gut verständlich und eindeutig definiert. Trotzdem hätte in der Spezifikation mehr Priorität auf die dokumentarischen Inhalte der Beschreibungssprache gelegt werden müssen und das Verwenden des **documentation** Operators nicht nur möglich sein, sondern auch empfohlen werden sollen. Vorteil gegenüber WSDL ist, dass viel weniger Vorwissen notwendig ist, um alle Funktionen zu verstehen.

Toolunterstützung Auch für WADL gibt es einige Werkzeuge. Genauso wie bei WSDL 2.0 finden sich diese jedoch nicht sofort und sind meistens nur für Java implementiert. So gibt es im Jersey Framework[8] Möglichkeiten zum Generieren von WADL-Beschreibungen aus Java

[8] https://jersey.java.net/

Code und von GlassFish eine Bibliothek zum Generieren von Clientseitigem Java-Code aus einer WADL-Beschreibung[9]. Des Weiteren existiert von NetBeans ein relativ umfangreicher graphischer WADL Editor [10].

Verbreitung der Sprache Trotz ausführlicher Recherchen konnte keine wirkliche Verbreitung der Beschreibungssprache festgestellt werden. WADL ist immerhin eine erste wirkliche Antwort an die Notwendigkeit einer Beschreibungssprache, welche ausschließlich für REST-Services gedacht ist, trotzdem wird sie nur sehr selten verwendet.

3.4.3 Open API Spezifikation

Die Open API Spezifikation, auch bekannt als Swagger, ist neben RAML und API Blueprint eine der ersten Beschreibungssprachen, welche zum Ziel hat den API-First-Approach zu fördern. Neben den W3C-Standards ist sie eine der ersten Spezifikationen zum Beschreiben von REST-Schnittstellen. Baute sie anfangs noch ausschließlich auf JSON auf, ist es nun möglich Schnittstellen mit YAML, einer funktionalen Teilmenge von JSON, zu beschreiben. Genauso wie bei WADL handelt es sich bei Open API um eine Beschreibungssprache zum ausschließlichen Beschreiben von REST-Schnittstellen. Veröffentlicht ist die aktuelle Open API Spezifikation unter der Apache License 2.0 [Apa04].

Funktionale Kriterien

Hauptfunktionalitäten Open API bietet alle nötigen Funktionalitäten um REST-Services zu beschreiben. Wie schon erwähnt besitzt Open API eine JSON und eine YAML-Repräsentation. Die folgenden Codebeispiele beziehen sich auf die YAML-Repräsentation. Ein Open API Dokument besteht aus Objekten zum Speichern von Metadaten, einem Pfad-Objekt und dem definitions-Objekt. Durch das Definieren der einzelnen REST-Ressourcen, unabhängig des Hosts und der zum Host relativen Path-URI, werden die Fragen nach der bereitgestellten Funktionalität und deren Ort zumindest teilweise entkoppelt voneinander beantwortet.

Die ersten Objekte dienen dazu grundlegende Informationen über die Schnittstelle zu geben. So werden zum Beispiel der Titel, die Beschreibung, die Version der Schnittstelle und die Übertragungsprotokolle definiert. Im Pfad-Objekt werden dann die einzelnen Ressourcen und ihre Operationen definiert. Das definitions-Objekt ist dafür da, Datentypen für Operationen zu definieren. Listing 3.2 zeigt den YAML-Code des in Open API beschriebenen Beispielservices.

[9] Wadl2Java https://wadl.java.net/
[10] WdlVisualDesigner http://wiki.netbeans.org/WadlVisualDesigner

Listing 3.2 In Open API beschriebener Beispielservice

```
swagger: '2.0'
info:
  title: StockQuoteService
  description: Get several Stockquotes
  version: 1.0.0
host: example.com
schemes:
  - http
basePath: /v1
produces:
  - application/json
paths:
  /Stockquotes/{TickerSymbol}:
    get:
      summary: Stockquotes
      description: |
        Returns the TradePrice for a given Stockquote TickerSymbol
      parameters:
        - name: TickerSymbol
          in: path
          description: A TickerSymbol of a Stockquote.
          required: true
          type: string
      tags:
        - Stockquote
      responses:
        '200':
          description: The price of the given Stockquote
          schema:
            $ref: '#/definitions/TradePrice'
        default:
          description: Unexpected error
          schema:
            $ref: '#/definitions/Error'
definitions:
  TradePrice:
    type: object
    properties:
      Price:
        type: number
        description: The price of the given StockQuote TickerSymbol
  Error:
    type: object
    properties:
      code:
        type: integer
      message:
        type: string
```

Typsicherheit Die Open API Spezifikation sieht sowohl die Verwendung von primitiven Datentypen, als auch die Definition von komplexen Datentypen vor. Hierfür werden im Dokument sogenannte Models definiert, welche dann in ein JSON Schema überführt werden. JSON-Schema ist eine, aus dem Bedürfnis nach einem Schemaabgleich heraus entstandene, Spezifikation zum Validieren von JSON-Objekten auf ein bestimmtes Schema. Die neueste Version ist Draft v4 [Int13] und ermöglicht ein Definieren von Schemas ähnlich wie in XML. Dieses JSON-Schema kann wiederum in den jeweiligen MIME-Type überführt werden. Somit können auch mehrere MIME-Repräsentationen für dasselbe Model verwendet werden.

Versionierung Im Gegensatz zu den vorher beschriebenen Sprachen stellt Open API eine Möglichkeit zur Verfügung, die Beschreibung der Schnittstelle zu versionieren. Hierbei lässt sich einerseits jeder Beschreibung eine eindeutige Version geben, andererseits ein Pfadpräfix für die URIs der Ressourcen definieren. Versionsnummer und Pfadpräfix sind jedoch nicht voneinander abhängig.

Semantische Beschreibung Für Open API Beschreibungen existiert keine in die Spezifikation integrierte Möglichkeit eine Beschreibung mit Semantischen Informationen zu füllen. Auch ein Mapping in eine Ontologie ist nirgendwo spezifiziert.

Nichtfunktionale Kriterien

Lesbarkeit Da YAML einen viel geringeren syntaktischen Overhead erzeugt als XML, sind Beschreibungen nach der Open API Spezifikation deutlich einfacher zu lesen als bei den vorher untersuchten Beschreibungssprachen. Besonders die Wahl der Bezeichner und der Aufbau des Dokuments erhöhen die Lesbarkeit. Des Weiteren ist Open API darauf ausgelegt direkt in eine HTML-Repräsentation übersetzt zu werden, was bei allen gängigen Editoren umgesetzt wird.

Verständlichkeit Durch den hierarchischen Aufbau des Pfad-Objekts und die Möglichkeit, die Schnittstelle mit verschiedenen Elementen dokumentarisch zu beschreiben, bekommt der Leser ziemlich schnell ein Verständnis von der Funktionalität des Services über die Struktur der Schnittstelle hinaus. Darüber hinaus ist es möglich, mit Hilfe von Tags die Ressourcen und ihre Operationen funktional zu klassifizieren.

Erweiterbarkeit Die Open API Spezifikation bietet die Möglichkeit, mit Hilfe von extensions „x-“, Elemente über die Spezifikation hinaus zu definieren. Dies wurde gerade für anbieterspezifische Use-Cases ermöglicht, jedoch gibt es noch keine Spezifikation für SLAs. Seit dem

Wechsel zu Open API wollen die Entwickler häufig genutzte Erweiterungen in die Spezifikation integrieren[11].

Dokumentation & Spezifikation Die Open API Spezifikation ist vollständig und unmissverständlich definiert, jedoch relativ unübersichtlich. Was fehlt, ist eine grobe Übersicht über alle Funktionalitäten der Beschreibungssprache und deren Umsetzung. Teilweise werden Konstrukte, wie das Schema-Objekt, an mehreren Stellen verwendet und somit auch redundant erklärt. Außerhalb der Spezifikation existieren Tutorials, Live-Demos und das Github-Repository, welche beim Erlernen und Anwenden der Beschreibungssprache helfen.

Toolunterstützung Wie erwähnt, wird bei Open API viel Wert darauf gelegt, eine möglichst große Bandbreite an Tools bereitzustellen. Gerade zur Förderung des API-First Approach existieren viele Werkzeuge. So gibt es Werkzeuge zum Generieren von Codebausteinen aus einem Open API Dokument, Visualisierungswerkzeuge zum Veranschaulichen einer in Open API spezifizierten API, Editoren zum Schreiben von Open API Dokumenten oder Bibliotheken zum automatischen Generieren von SDKs für die Schnittstelle. Gerade der „Swagger Editor“ [12] gibt die Möglichkeit, Fehler in der Beschreibung sofort zu erkennen und übersetzt das Dokument direkt in eine HTML-Repräsentation.

Verbreitung der Sprache Die Open API Spezifikation ist unter den neueren Beschreibungssprachen die bekannteste. Verwendet wird sie vor allem von Firmen wie Reverb, 3Scale oder Apigee [Sto14]. Open API/Swagger hat im Vergleich zu RAML und API Blueprint die größte Community und ist dabei sich als Standard zu etablieren. Unterstützt wurde dies durch den Wechsel zur Open API Initiative. Die Open API Initiative ist ein Projekt der Linux Foundation und zählt Google, IBM, Microsoft oder PayPal zu seinen Mitgliedern. Somit ist es absehbar, dass die Open API Spezifikation wohl zu einem allgemeingültigen Standard heranwachsen wird.

3.4.4 RAML

Genauso wie Open API ist RAML eine Beschreibungssprache mit YAML-Repräsentation. Die Autoren von RAML wollen das Verteilen von API-Lösungen, Mustern und Ressourcen ermöglichen um dadurch die Redundanz bestimmter Muster innerhalb der Beschreibung einer Schnittstelle zu vermeiden [Til15]. RAML ist unter der Apache License Version 2.0 [Apa04] veröffentlicht. Wird die Spezifikation abgeändert, so darf das Nutzen der Beschreibungssprache nicht mehr mit der RAML-Spezifikation in Verbindung gebracht werden. Die Herausgeber

[11] Adding optional SLA definitions with the Spec https://github.com/OAI/OpenAPI-Specification/issues/541
[12] Swagger Editor: http://editor.swagger.io/

wollen hiermit die Eindeutigkeit der Spezifikation bewahren. Die aktuellste stabile Version ist 0.8 [RAM13]. Version 1.0 [RAM15] liegt derzeit als Release Candidate vor, wird teilweise jedoch bereits produktiv eingesetzt.

Funktionale Kriterien

Hauptfunktionalitäten Auch RAML bietet alle Funktionalitäten an, um eine REST-Schnittstelle zu beschreiben. Eine einheitlich umgesetzte Typsicherheit fehlt hingegen. Die Entkopplung zwischen Funktionalität und dem Zugriffsort findet hier genauso wie bei Open API statt.

RAML ist in einen Teil zur Beschreibung verschiedenster Metadaten und einen Teil zur Beschreibung der Ressourcen aufgeteilt. Die Beschreibung der Ressourcen unterscheidet sich vom Aufbau her nur leicht von der Open API Spezifikation. Anstelle von Beispielwerten als Parameter können JSON- oder XML-Schemas verwendet werden.

```
#%RAML 0.8
title: StockQuote
version: v1
#baseUri: http://example.com/{version}
mediaType: application/json
/Stockquotes/{tickerSymbol}:
 displayName: StockquoteTickerSymbol
 uriParameters:
  tickerSymbol:
   displayName: Ticker Symbol
   type: string
 get:
  description: Returns the TradePrice for a given Stockquote
  responses:
   200:
    description: The price of the given Stockquote
    body:
     example: |
      {
       "price" : 150.0
      }
   500:
     description: Unexpected error
     body:
      example: |
       {
        "code": 501,
        "message": "string"
       }
```

Eine Funktionalität, mit der sich RAML von Open API abhebt, ist die Möglichkeit sogenannte *Resource Types* und *Traits* zu definieren. Ressource Types sind eine Abstrahierung zwischen

der bereitgestellten Funktionalität einer Ressource und der konkreten Ressource an sich. Traits sind vordefinierte Eigenschaften, welche dann an Ressourcen gebunden werden können. Ein Beispiel für einen Trait wäre das Definieren eines Zugriffsschlüssels im HTTP-Header. Folgendes Codebeispiel zeigt einen Resource Type zum Beschreiben von Listenfunktionalitäten einer Schnittstelle [RAM13].

```
resourceTypes:
  - collection:
      usage: This resourceType should be used for any collection of items
      description: The collection of <<resourcePathName>>
      get:
        description: Get all <<resourcePathName>>, optionally filtered
      post:
        description: Create a new <<resourcePathName | !singularize>>
```

Typsicherheit Im Gegensatz zur Open API besitzt RAML 0.8 kein eigenes Konzept für komplexe Datentypen. Es können nur XML Schema Definitionen oder JSON Schemas definiert werden. Zum Anbieten verschiedener Schemas, beispielsweise zur Parameterübergabe, müssen diese immer explizit definiert werden. Dies kann eine inkonsistente Typisierung der Daten in der Schnittstelle zur Folge haben. Mit RAML Version 1.0 kommt jedoch ein einheitliches Typensystem hinzu.

Versionierung Genauso wie Open API bietet auch RAML ein Konzept zur Versionierung an. Im Vergleich zur Open API Spezifikation sind URI-Präfix und Versionsnummer aneinander gekoppelt.

Semantische Beschreibung Ebenso wie für die Open API Spezifikation existiert für RAML keine Möglichkeit semantische Informationen hinzuzufügen, bzw. eine Beschreibung in eine Ontologie zu mappen.

Nichtfunktionale Kriterien

Lesbarkeit Zwischen RAML und Open API gibt es bei der Lesbarkeit der Beschreibungssprache keine großen Unterschiede, da beide eine YAML-Repräsentation besitzen. Lediglich beim Verwenden von JSON- bzw. XML-Schemas in RAML wird das RAML Dokument etwas unübersichtlich, da gerade bei XML ein syntaktischer Overhead hinzukommt. RAML bietet hierfür jedoch die Möglichkeit an, Schemas zu importieren anstatt sie an der nötigen Stelle zu definieren.

Verständlichkeit RAML nutzt einen hierarchischen Aufbau der Ressourcen, was es einfach macht, die dahinter liegende Funktionalität der Schnittstelle zu verstehen. RAML fordert zwar nicht das Hinzufügen von dokumentarischen Inhalten, jedoch wird spätestens bei der Kompilierung in ein HTML-Dokument davon ausgegangen, dass dies gemacht wurde.

Erweiterbarkeit Es wird keinerlei Möglichkeit zur Erweiterung der Spezifikation geboten. Grund ist die Absicht, die Spezifikation so eindeutig und konsistent wie möglich zu halten.

Dokumentation & Spezifikation Die Dokumentation der RAML Spezifikation 0.8 ist eher unübersichtlich und verwirrend gegliedert. Dieses Manko wird mit der RAML Version 1.0 behoben. Für diese Version existiert eine übersichtliche Gliederung mit der Darstellung aller Funktionalitäten.

Toolunterstützung RAML bietet im Grunde dieselben Werkzeuge wie Open API an. Großer Pluspunkt ist der Editor, welcher eine Autovervollständigung bereitstellt und Services anhand von Beispielwerten simulieren kann. Die bekanntesten Werkzeuge werden von MuleSoft entwickelt und vertrieben.

Verbreitung der Sprache RAML findet weite Verbreitung. Firmen wie Sonic, Spotify, Cisco oder VMWare verwenden RAML zum Beschreiben ihrer APIs. Des Weiteren wird RAML von MuleSoft als Standard für die von MuleSoft angebotenen Produkte verwendet und aktiv weiterentwickelt. Durch die genannte Möglichkeit, ResourceTypes und Traits als Blueprints vordefinieren zu können, wird eine Option geschaffen Entwurfsmuster auszutauschen und sie der Community zur Verfügung zu stellen. Dies macht RAML im Vergleich zu den anderen Beschreibungssprachen attraktiv, da somit gute Lösungen konsistent in allen Schnittstellenbeschreibungen wiederverwendet werden können.

3.4.5 API Blueprint

Wie schon beschrieben, ist API Blueprint eine auf Markdown basierende Beschreibungssprache und unterstützt genauso wie RAML und Open API den API-First Approach. Durch die Verwendung von Markdown wird innerhalb der Beschreibung sehr viel Fließtext geschrieben und schon somit ein sehr gut lesbares Dokument generiert [Api16]. Lizensiert ist API Blueprint unter der MIT License [Ope88]. Sie ist somit genauso wie alle anderen Beschreibungssprachen frei verfügbar.

Funktionale Kriterien

Hauptfunktionalitäten API Blueprint erfüllt alle funktionalen Anforderungen um eine REST-Schnittstelle zu beschreiben. Im Gegensatz zu RAML wurde an ein einheitliches Typensystem gedacht.

API Blueprint gliedert sich genauso wie die Open API Spezifikation in drei Bestandteile. Dem Definieren von Metadaten, dem hierarchischen Beschreiben der Ressourcen und dem Definieren von Datentypen. Da die Host-URL gesondert von den Ressourcen angegeben werden kann, kann hier wieder von einer Abstraktion zwischen Endpunkt und beschriebener Funktionalität ausgegangen werden. Es kann jedoch nicht wie bei Open API oder RAML eine relative Pfad-URI angegeben werden, welche den Host weiter von der REST-Ressource entkoppeln würde.

```
FORMAT: 1A
HOST: http://example.com/v1

# StockQuoteService
Get several Stockquotes

# Group Stockquote

## Stockquotes By TickerSymbol [/Stockquotes/{TickerSymbol}]

+ Parameters
   + TickerSymbol: google (string, required)

      A TickerSymbol of a Stockquote.

### StockquotesByTickerSymbol [GET]
Returns the TradePrice for a given Stockquote TickerSymbol

+ Response 200 (application/json)

      The price of the given Stockquote

   + Attributes (TradePrice)

+ Response 500

      Unexpected error
```

Ein Nachteil, welcher beim Schreiben auffällt, ist der teilweise fließende Übergang zwischen Syntax und Semantik. Somit ist nicht immer klar, welche Textstücke Teil der Syntax und welche Stücke dokumentarische Inhalte sind. So ist im folgenden Beispiel, in welchem die verwendeten Datenstrukturen definiert werden, *Data Structures* ein syntaktisch gefordertes Element, *Properties* jedoch ein frei definierbarer Platzhalter, welcher die Bedeutung der folgenden Attribute des Datentyps näher bringen soll.

```
# Data Structures

## TradePrice (object)

### Properties
+ ‘Price‘: 50 (number, optional) - The price of the given StockQuote TickerSymbol

## Error (object)

### Properties
+ ‘code‘ (number, optional)
+ ‘message‘ (string, optional)
```

Typsicherheit API Blueprint gleicht den bei RAML genannten Nachteil der Spezifikation von Datentypen durch MSON [Api14] aus. MSON (Markdown Syntax for Object Notation) ist ein von der Apiary [13] spezifizierter Markdown Syntax zum Beschreiben von Datentypen, welche dann wiederum in JSON-Objekte oder JSON-Schemas umgewandelt werden können.

Versionierung API Blueprint bietet kein Konstrukt zum Versionieren der Schnittstelle an. Einzige Möglichkeit ist es also, den Endpunkt zu versionieren. Dies bedeutet jedoch, dass einlesende Tools zwei Versionen derselben Schnittstelle als zwei verschiedene APIs wahrnehmen. Somit ist der Unterschied zwischen einem in eine SOA integrierten Service und dessen Versionen nicht mehr implizit gegeben.

Semantische Beschreibung Es existiert keine Möglichkeit zur Annotation von Semantik. Der Übergang zwischen Syntax und Semantik ist fließend, daher ist eine Beschreibung in API-Blueprint sehr schwer in eine Ontologie überführbar.

Nichtfunktionale Kriterien

Lesbarkeit Durch die Verwendung von Markdown lässt sich eine gegebene Schnittstellenbeschreibung fast ohne Vorwissen über die verwendete Beschreibungssprache fließend lesen.

[13] Apiary https://apiary.io/

Die sinnvolle Aufgliederung der einzelnen Aspekte einer REST-Schnittstelle, wie zum Beispiel Bereiche für die Definition der Ressource an sich, die Definition der einzelnen Operationen oder den Bereich für die Definition von Datenstrukturen, hilft beim Lesen.

Verständlichkeit Im Vergleich mit Open API und RAML ist API Blueprint die verständlichste Beschreibungssprache. API Blueprint ermöglicht es nicht nur an jeder Stelle, die Beschreibung mit dokumentarischen Inhalten zu ergänzen, sondern fordert diese auch. So müssen wie schon erwähnt, Attribute von Datentypen klassifiziert oder die einzelnen Ressourcen funktional gruppiert werden.

Erweiterbarkeit API Blueprint bietet keine Möglichkeit eigene Anwendungsfälle, wie zum Beispiel Service Level Agreements, als Erweiterungen in die Spezifikation einfließen zu lassen.

Dokumentation & Spezifikation Die Spezifikation ist gut in ihre funktionalen Bestandteile aufgeteilt. Jedoch wird in der Spezifikation eher die Restriktion zum Markdown Syntax beschrieben, was ein ausgeprägtes Vorwissen über Markdown vorausgesetzt. Neben der Spezifikation werden verschiedenste Tutorials, Beispiele und weiterführende Ressourcen bereit gestellt.

Toolunterstützung API Blueprint bietet eine große Bandbreite an Tools an. Hierbei sticht das mächtige Werkzeug „Apiary“ heraus. Mit „Apiary“ lassen sich alle Aspekte der REST-API-Entwicklung abdecken. Innerhalb eines Werkzeugs lassen sich Schnittstellen beschreiben, Codefragmente und Tests generieren, die Schnittstelle gegen einen gemockten Endpunkt testen und eine Contineous Integration laufen lassen. War „Apiary“ anfangs nur für API Blueprint gedacht wird jetzt auch der Open API Standard unterstützt.

Verbreitung der Sprache API Blueprint wird von Apiary unterstützt und findet vor allem in der Github Community starke Verbreitung. Grund hierfür ist die Integration des API Blueprint Markdown in Github. In API Blueprint beschriebene Schnittstellen werden direkt in eine lesbare Repräsentation überführt. Schon über 600 Github Repositories nutzen API Blueprint. Trotz gut lesbarem Markdown ist API Blueprint jedoch beim Schreiben nicht intuitiv genug. Dies könnte dazu führen, dass diese Beschreibungssprache sich für den First-API Approach nicht gegen Open API oder RAML durchsetzen kann.

3.5 Abschließendes Fazit

In diesem Abschnitt werden nun die Resultate der einzelnen Analysen miteinander verglichen. Tabellen 3.1 und 3.2 zeigen die auffallenden Unterscheidungsmerkmale der verschiedenen Beschreibungssprachen in den jeweiligen Kriterien auf. Abschließend wird gezeigt, welche Beschreibungssprache welches Kriterium am besten umgesetzt hat.

	Hauptfunktionalitäten	Typsicherheit	Versionierung	Semantische Beschreibung
WSDL 2.0	Alle nötigen Funktionalitäten	XSD-Typsystem, andere Schemas möglich	Nicht direkt unterstützt	Semantische Annotationen, WSDL zu RDF Spezifikation
WADL	Alle nötigen Funktionalitäten	Representation-Operator, Jeder MIME-Type möglich	Nicht direkt unterstützt	Keine Semantischen Annotationen möglich
Open API	Alle nötigen Funktionalitäten	Einheitliches Typensystem, in verschiedene MIME-Types überführbar	Abstraktion zwischen Endpunkt, Version und Pfad-URI	Keine Semantischen Annotationen möglich
RAML	Zusätzlich Ressource Types und Traits	Kein einheitliches Typensystem	Versionsnummer an URI-Präfix gekoppelt	Keine Semantischen Annotationen
API Blueprint	Alle nötigen Funktionalitäten, jedoch fließender Übergang zwischen Syntax und Semantik	Einheitliches Typensystem nach MSON Spezifikation	Nur über Endpunkt oder URI-Präfix	Keine Annotationen möglich, sehr schwer in RDF überführbar
Am besten umgesetzt	RAML	Open API	Open API	WSDL 2.0

Tabelle 3.1: Vergleich der Funktionalen Kriterien

	Lesbarkeit	Verständlichkeit	Erweiterbarkeit	Doku & Spezi	Toolunterstützung	Verbreitung
WSDL 2.0	Schwer lesbar, hoher syntaktischer Overhead	Documentation-Operator und sprechende Bezeichner	Sehr gute Erweiterungsmöglichkeiten	Vollständig und Unmissverständlich, Vorkenntnisse nötig	Umfangreich, jedoch nur notgedrungen entwickelt	Wenig verbreitet
WADL	Übersichtlich, hoher syntaktischer Overhead	Documentation-Operator und Sprechende Bezeichner	Sehr gute Erweiterungsmöglichkeiten	Vollständig und schlüssig	Nur nötigste Werkzeuge	So gut wie gar nicht genutzt
Open API	YAML-Syntax gut lesbar, Generation von HTML Dokument	Dokumentarische Elemente, hierarchischer Aufbau der Ressourcen, Funktionale Klassifizierung	Extensions Element, Fokus auf spezifische Use-Cases	Vollständig aber unübersichtlich, Redundanzen	Große Bandbreite an Werkzeugen	Stark verbreitet, Open API Initiative
RAML	YAML-Syntax gut lesbar, Generation von HTML Dokument	Dokumentarische Elemente, hierarchischer Aufbau der Ressourcen	Keine Erweiterungsmöglichkeit	0.8 unübersichtlich, 1.0 gut gegliedert	Breite Bandbreite an Werkzeugen, Sehr guter Editor	Gute Verbreitung, Mulesoft
API Blueprint	Markdown Fließend Lesbar, Generation von HTML Dokument	Sehr gut verständlich, fordert dokumentarische Inhalte	Keine Erweiterungsmöglichkeit	Vorwissen zu Markdown notwendig, verwirrend	Breite Bandbreite an Werkzeugen, Apiary	Github Integration, Apiary
Am besten umgesetzt	API Blueprint	API Blueprint	WSDL 2.0/ WADL	WSDL 2.0	API Blueprint / Open API	Open API

Tabelle 3.2: Vergleich der Nicht Funktionalen Kriterien

Beim Untersuchen der Hauptfunktionalitäten hat RAML am besten abgeschnitten. RAML bietet nicht nur alle nötigen Funktionalitäten zum Beschreiben von REST-Services, sondern ermöglicht es zusätzlich, Blueprints für Ressourcen zu definieren, um diese innerhalb der Schnittstelle und über diese hinaus wiederzuverwenden. Zu erwähnen ist allerdings, dass die Open API Spezifikation die am intuitivsten zu schreibende Sprache definiert. Die Kriterien Typsicherheit und Versionierung sind von der Open API Spezifikation am besten umgesetzt. Open API bietet ein einheitliches Typensystem, welches als Metamodell beispielsweise für die Überführung in ein JSON-Schema oder XML-Schema dienen kann. Des Weiteren fehlt bei RAML 0.8 ein einheitliches Typsystem. Open API ist die einzige Beschreibungssprache, welche mit Hilfe des Version- und BasePath-Parameters die Abstraktion zwischen Endpunkt, Version und Pfad-URI ermöglicht. Mit der Möglichkeit von Semantischen Annotationen und der Überführung in ein RDF-Schema, hat WSDL 2.0 das Kriterium der Semantischen Beschreibung am besten umgesetzt. Beim Untersuchen der Lesbarkeit und der Verständlichkeit der Beschreibungssprache hat API Blueprint durch seine fließend lesbare Beschreibungssprache und den fließenden Übergang von Syntax und Semantik am besten abgeschnitten. Die Verwendung von Markdown als Metasprache hebt API Blueprint von allen anderen Beschreibungssprachen ab. Auch wenn es lange dauert, sich in die Spezifikation von WSDL 2.0 einzulesen, ist diese doch am eindeutigsten und am unmissverständlichsten beschrieben. Es wird deutlich, dass sehr viel Erfahrung und Wissen in das Dokument gesteckt wurden. Gerade beim Entwickeln von neuen Tools ist dies ein bedeutender Faktor. Die größte Auswahl und Bandbreite an Werkzeugen bieten API Blueprint und Open API. Größter Pluspunkt ist das Werkzeug Apiary, welches nur für API-Blueprint entwickelt wurde, nun aber auch eine Open API Unterstützung anbietet. Dies hebt den Werkzeugkatalog der Open API deutlich von dem von RAML ab. Ein Nachteil von Markdownsprachen wie YAML oder Markdwon ist, dass der Syntax doch nicht so frei ist wie angenommen. Überall muss auf die richtigen Einrückungen und vordefinierten Codeblöcke geachtet werden. Somit ist ein richtiger Editor entscheidend, was alle drei Sprachen, Open API, RAML und API Blueprint, mitbringen. In der Verbreitung schneidet Open API/Swagger mit der Übernahme zur Open API Initiative definitiv am besten ab. Swagger bekommt somit den größtmöglichen Rückhalt bei der Etablierung als Standard zum Beschreiben von REST-Schnittstellen. Da die Suche nach einem gemeinsamen Standard als essentiellen Bestandteil für eine herstellerübergreifende Dokumentation zu sehen ist spielt das Kriterium der Verbreitung einer Beschreibungssprache eine wichtige Rolle.

Weiterhin lassen sich die analysierten Beschreibungssprachen in ihre jeweiligen Entwicklungsprozesse einteilen. So bietet es sich an, mit Markup-Languages wie Open API, RAML oder API Blueprint über den API-First Approach, Outside-In zu entwickeln, während WADL und WSDL immer nur generiert werden sollten und sich somit nur für einen Inside-Out Entwicklungsprozess eignen. Da sich jedoch der API-First Approach wie schon angesprochen als der bessere Entwicklungsprozess für REST-Services herausgestellt hat, sind WSDL und WADL somit eher weniger zu empfehlende Standards zum Beschreiben von REST-Services. Vielmehr ist auch fraglich, ob WADL oder WSDL 2.0 sich jemals als Standard durchsetzen werden. Durch den Ansatz der Umsetzung des RMI/RPC-Paradigmas, bei dem für jede Anwendung eine eigene Schnittstelle erstellt wird, ist WSDL 2.0 schon an sich eher weniger für REST-Services

geeignet [Til15]. Auch WADL wird sich durch seine geringe Verbreitung wohl nie als Standard etablieren.

Beim Umsetzen des Vergleichs ist aufgefallen, dass keine der Beschreibungssprachen die Berücksichtigung des Hypermedia-Aspekts im RESTFul Paradigma unterstützt. Im Gegenteil, durch das Binden der Ressourcen an einen festen Endpunkt und die statische Dokumentation dessen, wird der dieser eher vernachlässigt. Andererseits wäre der Hypermedia-Aspekt auch nur dann vollends umgesetzt, wenn sich die Schnittstellen entweder selber beschreiben würden oder die Beschreibungen an ihrer jeweiligen Ressource angeheftet sind. Diese Art der Dokumentation reicht jedoch im Unternehmensumfeld nicht aus, welches von einem zentralen Metadatenmanagement ausgeht. Somit kann dieses Kriterium für das Finden einer Beschreibungssprache vernachlässigt werden, sollte jedoch, beim Entwickeln von RESTFul-Services, immer im Hinterkopf bleiben.

Abschließend ist zu sagen, dass die Open API Spezifikation in den meisten Kriterien am besten abgeschnitten hat, nirgendwo ein deutliches bzw. nicht zu vernachlässigendes Defizit aufweist und auch die am weitesten verbreitete Beschreibungssprache ist, welche sich gerade als gemeinsamer Standard etabliert. Somit geht Open API als die Beschreibungssprache aus dem Vergleich heraus, welche im weiteren Verlauf der Arbeit als Standard für REST-Schnittstellen im SOA Governance Repository und somit für die Import/ Exportmöglichkeit, verwendet wird.

4 Konzept der Implementierung

Im zweiten Teil der Arbeit soll eine Importmöglichkeit für Webservices in das SOA Governance Repository geschaffen und deren relevante Metadaten über die Benutzeroberfläche angezeigt werden. Nach dem Import soll es möglich sein, aus importierten Servicebeschreibungen neue Beschreibungen zu generieren. Bei den importierten Services handelt es sich zum größten Teil um mit WSDL beschriebene SOAP-Services, welche dann in eine Open API Beschreibung einer REST-Schnittstelle umgewandelt werden sollen. Eine wesentliche Herausforderung ist es in erster Linie, eine Möglichkeit der Konvertierung von importierten SOAP-Services in REST-Services zu schaffen. Diese können dann unabhängig von ihrer technischen Realisierung im Repository gespeichert und angeboten werden. Das folgende Kapitel beschreibt ein Konzept für die gegebene Aufgabenstellung, dessen Umsetzbarkeit in Kapitel 5 untersucht wird.

4.1 WSDL-Import

Die erste Aufgabe ist es, eine Funktionalität zum Import von Services zu entwerfen. Da in einer Serviceorientierten Architektur mit Webservices SOAP-Services vorliegen und diese in erster Linie mit WSDL beschrieben werden, wird der Import von Services auf das Einlesen von WSDL-Dokumenten beschränkt. In dieser Form kommen sie zum Beispiel im Unternehmensumfeld zum Einsatz. Hierbei liegen die meisten Servicebeschreibungen der bestehenden Services noch in Version 1.1 vor. Trotzdem wurde sich für die aktuellere Version 2.0 entschieden, zum einen, da nur WSDL 2.0 alle HTTP-Verben und somit REST-Services unterstützt und zum anderen, da eine für WSDL 2.0 existierende Spezifikation zum Mappen von WSDL 2.0-Beschreibungen in ein RDF-Datenmodell existiert.

Es existieren verschiedenste Werkzeuge zum Konvertieren von WSDL 1.1 -zu WSDL 2.0-Dokumenten. Die bekanntesten sind der vom W3C bereitgestellte „WSDL 1.1 to WSDL 2.0 Converter“ [1] und die Apache Woden Bibliothek [2].

[1] WSDL 1.1 to WSDL 2.0 Converter https://www.w3.org/2006/02/WSDLConvert.html
[2] Apache Woden http://ws.apache.org/woden/

Die Umsetzung des WSDL-Imports unterteilt sich in drei Schritte:

1. Dem Einlesen der WSDL-Datei und Überführen in ein RDF-Datenmodell
2. Dem Erweitern des Governance Repository Modells um Triples, welche den bestehenden Graphen mit dem neuen Graphen verbinden
3. Dem Erfassen von für den Nutzer relevanten Daten, welche im Frontend angezeigt werden sollen

4.1.1 WSDL zu RDF Mapping

Um die importierten Daten persistent, schnell und an das existierende Datenmodell geknüpft zugänglich zu machen, wird das zu importierende WSDL 2.0 Dokument in einen RDF-Graphen überführt. Ein weiterer Vorteil ist die Möglichkeit nach dem Überführen des Graphen, diesen mit anderen überführten Dokumenten zu verbinden. So könnte der Graph eines WSDL-Dokuments mit der Repräsentation eines Policy-Dokuments oder einem Graphen über den Autor des Dokuments zusammengebracht werden [Kop07b]. Dieser Abschnitt befasst sich mit dieser Überführung.

Wie in Kapitel 3 erwähnt existiert eine Spezifikation des W3C zum Mappen der WSDL 2.0-Komponenten in einen RDF-Graphen [Kop07b]. In dieser Spezifikation wird eine Ontologie beschrieben, welche ein Eins-zu-Eins-Mapping für WSDL 2.0 definiert. Jede Komponente im WSDL-Namespace wird einer OWL-Klasse zugewiesen. Instanzen dieser Klassen wiederum sind über Prädikate miteinander verbunden [Kop06b]. Die Klassen sind nach den einzelnen Komponenten benannt und deren Instanzen nach der Definition für eindeutige Bezeichner aus der WSDL 2.0 Spezifikation [CMRW07]. Diese definiert Internationalized Resource Identifier (IRI)-Referenzen für jedes Element der WSDL-Spezifikation.

Klasse	**Instanz**
Description	tns:description()
Interface	tns:interface(*interfaceName*)
InterfaceOperation	tns:interfaceOperation(*interfaceName*/*operationName*)
InterfaceMessageReference	tns:interfaceMessageReference(*interfaceName*/ *operationName*/*messageLabel*)
Binding	tns:binding(*bindingName*)
BindingOperation	tns:bindingOperation(*bindingName*/*bindingOperation*
Service	tns:service(*serviceName*)
Endpoint	tns:endpoint(*serviceName*/*endpointName*)

Tabelle 4.1: OWL-Klassen und deren Instanzen

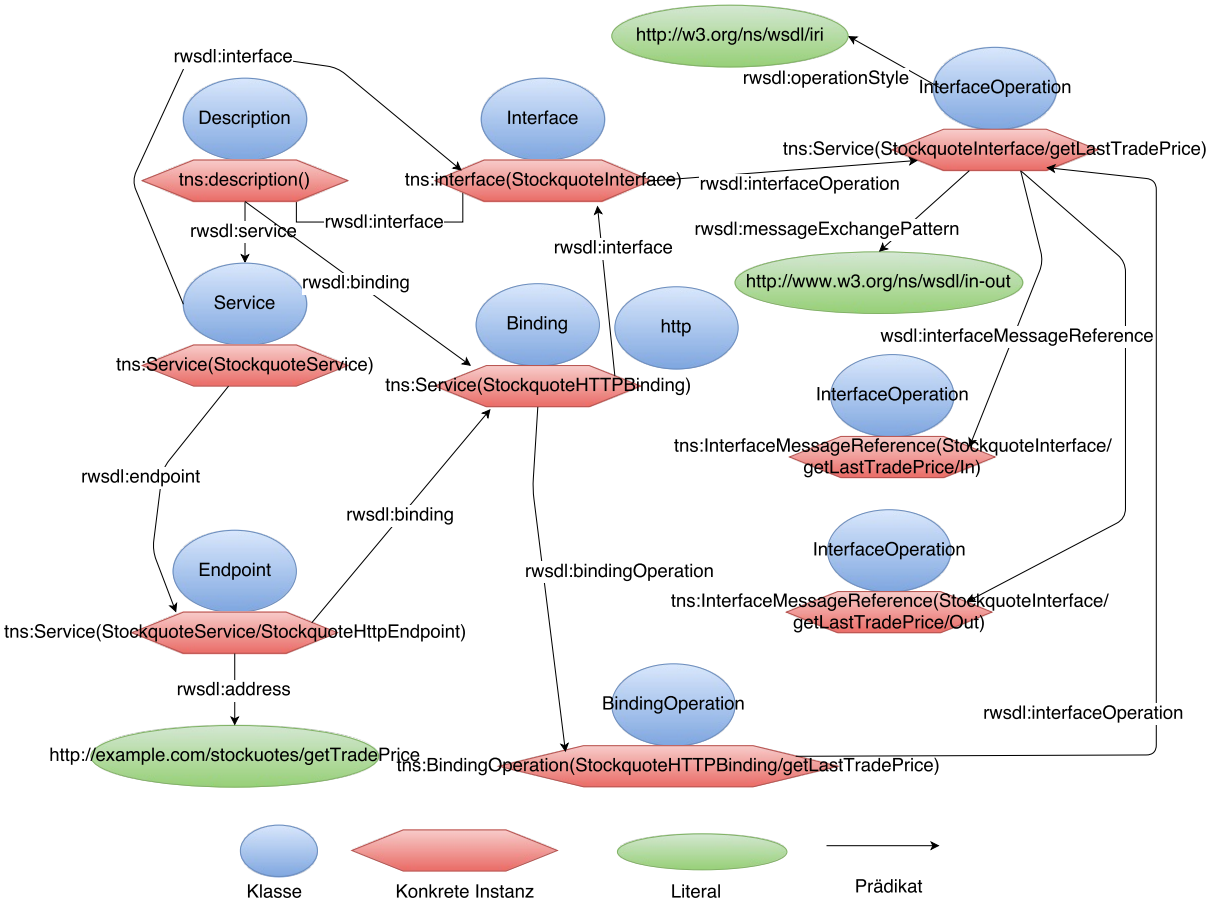


Abbildung 4.1: Ausschnitt des RDF Graphen für die WSDL Beschreibung des exemplarischen Aktienservices

Tabelle 4.1 listet die möglichen Klassen und deren Instanzen auf. *Tns* beschreibt den Targetnamespace des Dokuments, welcher für jedes WSDL-Dokument eindeutig sein muss. Des Weiteren werden die einzelnen Attribute, wie zum Beispiel das *MessageExchangePattern* einer WSDL-Operation oder der *BindingType* eines Bindings definiert. Wichtig zu erwähnen ist, dass das Mapping von Datentypen nicht vorgesehen ist. Auch WSDL-Extensions, wie HTTP-Bindings oder SOAP-Bindings, werden nicht implizit mit spezifiziert.

Abbildung 4.1 zeigt exemplarisch den RDF Graphen des in Kapitel 3 vorgestellten Aktienservices. Aus Übersichtsgründen wurde auf die *InterfaceMessagesReference* und einige Attribute verzichtet. Ein Überblick über den groben Aufbau eines WSDL-RDF-Graphen sollte ersichtlich werden. Tabelle 4.2 listet die in der Spezifikation, Abbildung und weiterhin im Dokument genutzten Namespaces auf.

Um einen besseren Überblick über die importierten WSDL-Dateien und die somit im Triple Store gespeicherten Graphen zu haben, empfiehlt es sich, die Graphen mit Hilfe von N-Quads [Car14] oder Named Graphs [CBHS05] um einen Kontext zu erweitern, welcher alle zum

Präfix	Namespace
tns	Targetnamespace/
rwsdl	http://www.w3.org/2005/10/wsdl-rdf
whttp	http://www.w3.org/ns/wsdl/http
sgr	http://sgr
dterms	http://purl.org/dc/terms/

Tabelle 4.2: Im Dokument genutzte Präfixe und Namespaces

Graphen gehörenden Triple eindeutig zuordnet. N-Quads definiert die Erweiterung eines RDF-Triples neben dem Subjekt, Prädikat und Objekt, um einen vierten Knoten, einem Label. Triples ohne Label werden trivialer Weise als N-Quads mit einem leeren Knoten definiert. Named Graphs können als Überarbeitung der „N-Quads-Idee“ gesehen werden, in welcher zwischen der Semantik und der Syntax des Graphkontextes stärker abstrahiert wird. Der Kontext wird nicht als vierter Knoten gespeichert, sondern eher als, über dem Graphen liegendes, Objekt definiert [CBHS05]. Als Kontext für die importierten WSDL-Beschreibungen bietet sich der Targetnamespace der Beschreibung an. Sowohl für Named Graphs als auch für N-Quads existieren SPARQL Unterstützungen. Somit können leicht Operationen, wie das Löschen oder Abrufen eines gesamten Kontextes, ausgeführt werden. Listing 4.1 zeigt exemplarisch einen N-Quad von der WSDL-Description zu einem WSDL-Interface für den Beispielservice.

Listing 4.1 N-Quad für das Interface Triple

```
<http://example.com/stockquote.wsdl#description()> rwsdl:interface
    <http://example.com/stockquote.wsdl#interface(StockquoteInterface)>
    <http://example.com/stockquote.wsdl> .
```

Erweiterung des Models um HTTP-Binding

Da mit Hilfe des HTTP-Bindings in WSDL schon REST-Services bzw. Webservices, welche alle HTTP-Verben nutzen, beschrieben werden können, bietet es sich an, dieses auch zu nutzen um Ressourcen für Services zu beschreiben, welche kein REST-Protokoll verwenden. Nach der Überführung der HTTP-Binding-Extension in das RDF-Model können somit die für einen REST-Service notwendigen Informationen gespeichert werden. Die WSDL 2.0-RDF-Spezifikation beschreibt hierfür das Mapping von WSDL-Extensions, insbesondere der HTTP-Binding-Extension [Kop07b]. Abbildung 4.2 veranschaulicht den entstehenden Graphen.

Auslesen neuer REST-Ressourcen

Ist in der WSDL-Beschreibung für die in der Service-Komponente definierten BindingOperations kein HTTP-Binding und somit auch kein HTTP-Endpoint vorhanden, so muss für jede

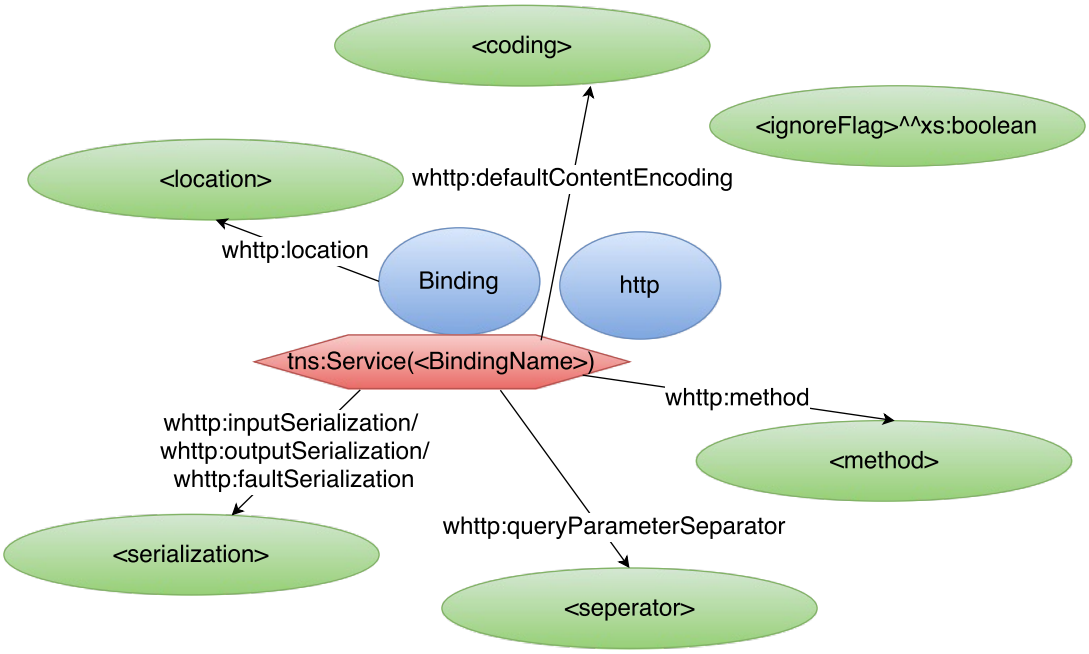


Abbildung 4.2: HTTP-Binding-Extension für das RDF Model

Operation aus den BindingOperations ein neues Binding und ein neuer Endpoint angelegt werden. Die hierfür benötigten Daten werden zu Teilen aus einer externen Quelle und zu Teilen aus dem WSDL-Dokument gewonnen. Die minimal benötigten Daten für jede Operation sind die Host-URL, über welche der Service erreichbar ist, die URI einer REST-Ressource und die semantisch passende HTTP-Methode. Folgende Aufzählung zeigt, wie die Daten erfasst werden können.

Host-URL Muss Extern übergeben werden, da einzelne Endpunkte beliebige URLs haben können und es nicht klar ist, welcher Teil der Endpunktadresse nur Präfix einer REST-Ressource ist.

REST-Ressource Muss extern übergeben werden. Nur so ist gewährleistet, dass ein semantisch korrektes Ressourcen-Modell aufgebaut wird.

HTTP-Methode Um einen Level 2-Service zu beschreiben, muss extern definiert werden, welche HTTP-Methode zur Operation gehört.

Eingabeparameter Kann aus dem Input-Element des Operation-Elements in der WSDL-Beschreibung gewonnen werden.

Ausgabeparameter Kann aus dem Output-Element des Operation-Elements in der WSDL-Beschreibung gewonnen werden.

Datentypen Die Input- und Output-Elemente referenzieren ihre jeweiligen Datentypen als XSD-Schema.

Aus diesen gewonnen Daten kann dann für die jeweilige Operation ein HTTP-Binding mit passendem Endpunkt angelegt werden. Abbildung 4.3 demonstriert das Erstellen eines HTTP-Bindings für die Aktienpreis-Operation.

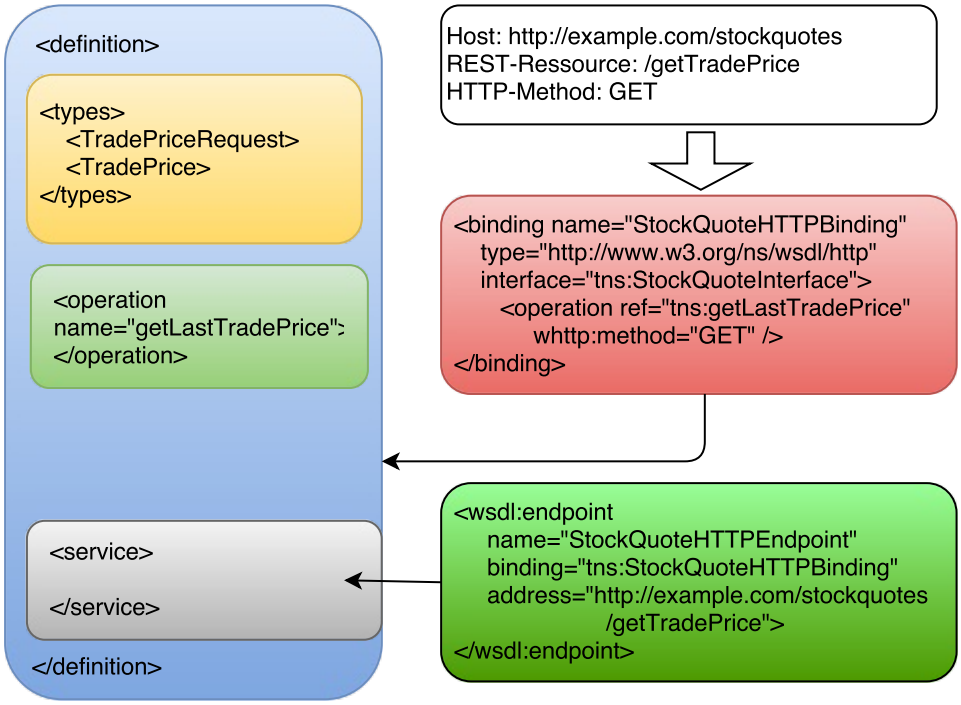


Abbildung 4.3: Erstellen eines HTTP-Bindings für die *getLastTradePrice*-Operation des Aktienservices.

4.1.2 Erweiterung des Governance Repository Modells

Damit über den Governance-Repository-Graphen auf die Daten des WSDL-RDF-Graphen zugegriffen werden kann, muss das Datenmodell des Repositories aus Abschnitt 2.4.2 entsprechend angepasst werden. Wie in der Arbeit von Victor Miyai [Miy15] erwähnt, muss eine Entität von der Serviceversion eines Services zur entsprechenden WSDL-Beschreibung (*rwsdl:Description*) erzeugt werden. Das Prädikat *sgr:serviceDescription* ermöglicht dies. Somit wird ein Triple zwischen der Instanz der Klasse *sgr:serviceVersion* und der Instanz der Klasse *rwsdl:Description* erstellt. Da in einer WSDL-Beschreibung mehrere Endpunkte definiert werden können und somit nicht immer nur ein SOAP-Endpunkt existiert, werden die Endpunkte nicht direkt über die Serviceversion verknüpft, sondern über die Entität zur WSDL-Beschreibung zugreifbar gemacht, siehe Abbildung 4.1. Des Weiteren wird in der Serviceversion noch ein Host Attribut hinzugefügt, welches später für den Export zu einer REST Schnittstelle relevant sein wird. Dessen Triple setzt sich aus der Instanz der *sgr:ServiceVersion* Klasse, dem Prädikat *sgr:Host* und einem Host-Literal zusammen. Abbildung 4.4 zeigt die erstellten Triple und die Traversierung

hin zu den Endpunkten für den Beispielservice, welcher an eine beliebige Serviceversion gehängt wird.

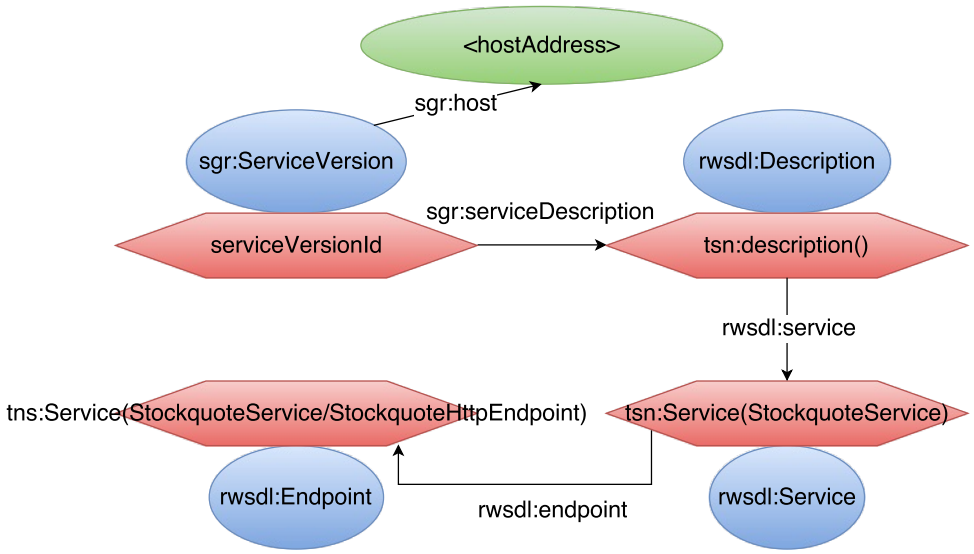


Abbildung 4.4: Ausschnitt des erweiterten Teil des Governance Repository Modells

4.1.3 Für den Nutzer relevante Daten

In erster Linie ist es das Ziel, mit Hilfe des Imports einer WSDL-Datei in das Governance Repository für verschiedene Stakeholder Informationen auszulesen und somit das Einpflegen neuer Services zu vereinfachen. Jede Komponente einer WSDL-Datei und ihre Unterkomponenten bieten dabei verschiedene relevante Informationen an:

Description Beinhaltet den *Targetnamespace* des Dokuments, welcher als Namen für die dazugehörige Serviceversion dient.

Types Können als Grundlage für die Beschreibung des Datenmodells eines Services dienen.

Interfaces Deren Namensgebung kann einen groben thematischen Überblick über die Funktionalität des Services geben.

InterfaceOperations Deren Namen und Ein- und Ausgabenachrichten geben einen konkreten Überblick über die vom Service bereitgestellte Funktionalität.

Bindings Geben einen Überblick über das genutzte Nachrichtenprotokoll des Services.

BindingOperations Zeigen die konkrete Realisierung des Nachrichtenprotokolls je Operation .

Services Geben einen Überblick über den Ort, an welchem die Schnittstelle zur Verfügung gestellt wird.

ServiceEndpoints Geben den konkreten Endpunkt für jede definierte Operation.

Documentations Können genutzt werden, um weitere Informationen zu den einzelnen Komponenten zu erlangen. Insbesondere das *Documentation*-Element als direktes Kind des *Description*-Elements kann als Beschreibung für die Serviceversion genutzt werden.

Diese Daten unterstützen den Nutzer beim Erfassen der Funktionalität der bereitgestellten Schnittstelle eines Services.

4.2 Open API-Export

Die zweite Aufgabe ist es, die in Abschnitt 4.1 importierten Servicebeschreibungen in eine Open API Beschreibung zu exportieren. Falls die importierte Beschreibung nur einen SOAP-Service beschreibt, muss dessen Schnittstelle zuvor in eine REST-API umgewandelt werden. Auch andere Ausgangsprotokolle wie zum Beispiel RMI sind möglich.

4.2.1 Anforderungen an importierte Servicebeschreibungen

Vor der Überführung möglicher SOAP-Protokolle in REST und der Konvertierung in Open API müssen noch einige Annahmen getroffen werden. Ziel ist es, so wenige Einschränkungen wie möglich an das Mapping und den SOAP zu REST-Export zu setzen. Ganz ohne Einschränkungen sind diese jedoch gar nicht oder nur schwer umsetzbar.

Verwendung von eindeutigen Targetnamespaces Wie schon im vorherigen Abschnitt 4.1.1 beschrieben, muss für jedes WSDL-Dokument ein eindeutiger Namespace definiert sein, da dieser als Kontext des importierten Graphen verwendet wird. Sind zwei verschiedene Beschreibungen mit gleichen Namespaces importiert, so kann im RDF-Datenmodell keine Unterscheidung mehr zwischen diesen getroffen werden. Die Triple des schon existierenden Graphen werden entweder überschrieben oder mit falschen Datensätzen erweitert. Dies kann wiederum zu einer fehlerhaften oder nicht eindeutigen Open API Beschreibung führen.

Types in XSD definiert Es ist nicht zwingend notwendig, dass alle verwendeten Typen in XSD definiert werden, jedoch wird ein konkretes Datenmodell benötigt, um dieses später auf Datentypen innerhalb der Open API Beschreibung zu übertragen. Da die WSDL 2.0 Spezifikation ohnehin die Empfehlung für XSD-Typen ausspricht, ist dies ohne weitere Einschränkungen annehmbar.

Existierendes HTTP-Binding beschreibt REST-Ressourcen Existieren in der gegebenen WSDL-Beschreibung schon HTTP-Bindings und Verknüpfungen zu einem Endpunkt, so wird angenommen, dass die Adressen Endpunkte nach dem URI-Modell für REST-Ressourcen definiert sind. Es existiert somit für alle Operationen ein gemeinsamer Host und für jede einzelne Operation eine Ressourcen-URI relativ zur Host-URL.

4.2.2 SOAP zu REST

Ein notwendiger Schritt hin zu einer Open API Beschreibung ist es, die beschriebene Schnittstelle, falls nicht schon vorhanden, in eine REST-Schnittstelle umzuwandeln. Es muss immer beachtet werden, dass REST und SOAP sich hierbei nicht nur als Protokoll, also auf technischer Ebene unterscheiden, sondern zumindest REST ein gewisses Paradigma vorgibt (siehe Grundlagen 2.5). Es wäre also falsch, schon allein wegen den völlig verschiedenen Konzepten beim Beschreiben einer Schnittstelle, beispielsweise den vorhandenen SOAP-Endpunkt und dessen Operationen trivialer Weise auf REST-Ressourcen zu übertragen. REST beschreibt seine Schnittstelle durch das Definieren eines Ressourcen Modells, SOAP durch das einfache Mappen der Schnittstelle auf SOAP Operationen. Das Ziel ist es somit, einen REST-Service zu generieren, welcher einerseits alle Funktionalitäten des vorhergehenden Services abdeckt und andererseits das RESTful-Paradigma mindestens nach Level 2 des Richardson Maturity Models [Fow10] implementiert (siehe Abschnitt 2.5.2). Nach dessen Definition implementiert ein REST-Service nach Level 2 die Verwendung verschiedener URIs mit verschiedenen Ressourcen und die Verwendung von mehr als einem HTTP-Verb, im besten Fall *GET*, *PUT*, *POST* und *DELETE*.

Datenerfassung durch den Nutzer

Wie schon in Abschnitt 4.1.1 beschrieben können bei einem fehlenden HTTP-Binding einige Daten nicht aus der WSDL-Beschreibung gewonnen werden. Somit müssen die fehlenden Daten durch den Nutzer hinzugefügt werden. Dies ist die tiefgreifendste Einschränkung bei der SOAP zu REST Konvertierung.

Alternativ können Ressourcen-URIs und HTTP-Verben auch automatisch generiert werden. So könnten als Ressourcen auch die Namen der einzelnen Operationen und als HTTP-Verb immer ein POST dienen. Dies würde jedoch einerseits die Entscheidungsfreiheit beim Bestimmen der HTTP-Endpunkte einschränken und andererseits den beschriebenen Service auf einen Level 1 Service im Richardson Maturity Model herabsetzen, da nur noch ein HTTP-Verb verwendet wird. Das Erfassen des entsprechenden HTTP-Verbs mit Hilfe des in der Operation spezifizierten MessageExchangePatterns würde auch nur eine Unterscheidung zwischen einem GET und einem POST ermöglichen und wäre meist semantisch nicht korrekt. Somit empfiehlt es sich, die fehlenden Daten durch den Nutzer bestimmen zu lassen, um einen semantisch korrekten Level 2 REST-Service zu erhalten.

Die Host-URL muss allerdings immer über den Nutzer übergeben werden. Existieren für mehrere Operationen beispielsweise nur SOAP-Bindings und ein dafür definierter Endpunkt so ist es nicht möglich, aus diesem Endpunkt ein für die Operationen passendes REST-Ressourcen-Modell aufzustellen.

4.2.3 WSDL RDF Model zu Open API Mapping

Nachdem alle Daten für eine REST-Schnittstelle erfasst wurden, kann der bestehende RDF-Graph der WSDL-Beschreibung in eine Open API Repräsentation überführt werden. Hierbei geht es wieder um ein rein funktionales Mapping, welches einen funktional äquivalenten, durch Open API beschriebenen REST-Service schaffen soll. Ein Eins-zu-Eins-Mapping zwischen WSDL und Open API wäre mit der aktuellen Open API Spezifikation nicht möglich, da diese keine Abstraktion zwischen Protokoll und funktionaler Beschreibung wie bei WSDL bietet. Somit gehen bei einem Mapping zu viele Informationen verloren, als dass dieses zurückgeführt werden könnte.

Erweiterung des Models um Businessobjekte

Wie schon erwähnt, sieht die WSDL zu RDF Spezifikation kein Mapping von Datentypen vor, sondern weist lediglich jedem Datentypen einen eindeutigen Bezeichner mit seinem Namespace und LocalName im XSD-Schema zu. Die Definition von Ein- und Ausgabeparametern ist jedoch in der Open API Spezifikation ein essentieller Bestandteil. Des Weiteren werden die von einem Service genutzten Datenmodelle im Governance Repository als Businessobjekte abgespeichert. Somit bieten sich die im WSDL-Dokument spezifizierten Datentypen an, um die Parameter der REST-Schnittstelle zu beschreiben und um ein Businessobjekt für den dazugehörigen Service anzulegen. Am einfachsten ist dies umzusetzen, indem eine XSD-Datei für die Typenbeschreibung erstellt wird und diese mit einem neuen Businessobjekt verlinkt wird. Anschließend wird jede InterfaceMessage und deren Datentyp mit dem neuen Businessobjekt verlinkt. Somit hält ein Businessobjekt die betreffende XSD-Datei als RDF-Ressource und eine Referenz zu jedem im Service verwendeten XSD-Datentypen. Zur Umsetzung kann der bestehende Graph des SOA Governance Repositories mit den Prädikaten *sgr:hasXSDType* und *sgr:isXSDTypeOf* erweitert werden. Diese schaffen die genannte Verknüpfung zwischen WSDL-Beschreibung und Businessobjekt. Abbildung 4.5 zeigt dies. Alle grau hinterlegten Knoten zeigen die schon bestehenden Knoten aus dem Graphen des Repositories.

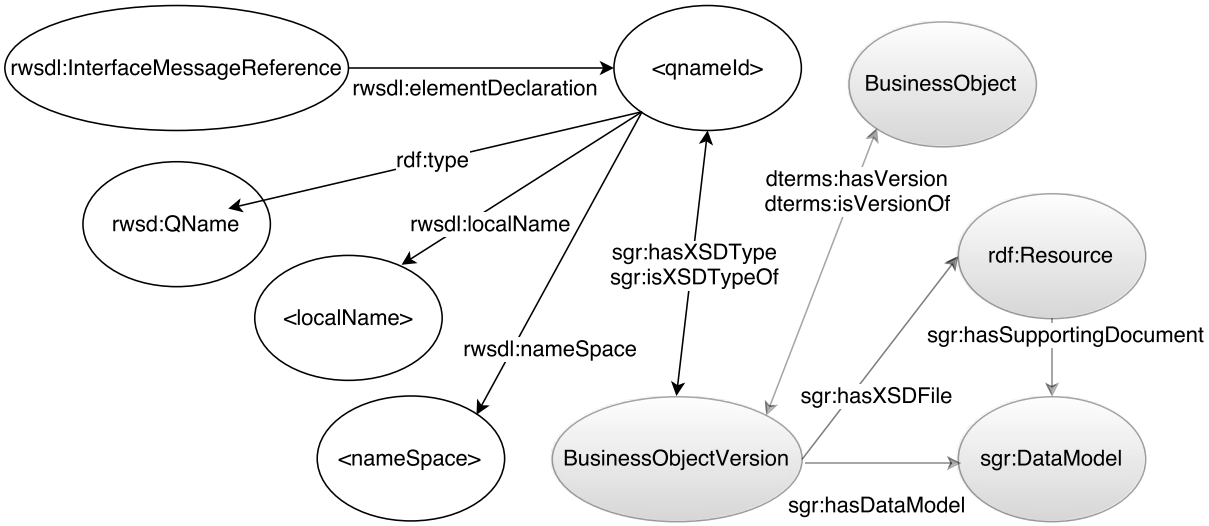


Abbildung 4.5: Verknüpfungen des Teilgraphen für Businessobjekte aus [Miy15] und der InterfaceMessageReference des WSDL-Graphen

Mapping

Tabellarisch wird die Konvertierung der WSDL-Beschreibung und der vorher erfassten Daten in eine Open API/(Bekannt als) Swagger-Beschreibung aufgezeigt. Ein Swagger-JSON-Objekt besteht aus mehreren JSON-Objekten. Es werden alle Objekte überführt, welche in der Spezifikation als vorausgesetzt gekennzeichnet sind (in Tabellen fett gedruckt), des Weiteren werden all die Objekte gemappt, zu denen genügend Informationen aus der WSDL-Beschreibung oder dem Governance Repository gelesen werden können.

Swagger Objekt	
ParentName	*None*
swagger	'2.0'
info	siehe Tabelle 4.3b
host	Host-URL
paths	siehe Tabelle 4.4b
basePath	Substring zwischen Host-URL und REST-Ressource
schemes	'http'/'https'-Präfix der Host-URL

(a) Mapping des Swagger Objekts

Info Objekt	
ParentName	*info*
description	Inhalt des documentation-Elements in der description-Komponente
version	Targetnamespace der WSDL-Datei
title	Servicename + Serviceversionname
contact	siehe Tabelle 4.4a

(b) Mapping des Info Objekts

Tabelle 4.3: Mapping der Swagger und Info Objekte

Das Wurzel-Element des JSON-Objekts ist das *Swagger*-Objekt. Es enthält alle weiteren Objekte, wie die Version der Spezifikation, dem *Info*-Objekt der *host*-URL, dem relativen *basePath*, dem Übertragungsprotokoll und dem *Path*-Objekt, welches die Pfade der einzelnen REST-Ressourcen enthält. Tabelle 4.3a macht das Mapping dieser Objekte deutlich. Das Info-Objekt,

siehe Tabelle 4.3b, besteht aus einer *description*, einem *version*-String, einem *Titel* für die Schnittstellenbeschreibung und einem *Contact*-Objekt. Hierfür können alle Daten bis auf den Titel aus der WSDL-Beschreibung genommen werden. Der Titel setzt sich aus den Namen der zugehörigen Serviceversion und ihrem Service zusammen.

Contact Objekt	
ParentName	*contact*
name	Name des ServiceOwners/ ServiceVersionOwners
email	Dessen Mailadresse

(a) Mapping des Contact Objekts

Paths Objekt	
ParentName	*paths*
(Für jede Operation) /ressourceName	siehe 4.5a

(b) Mapping des Path Objekts

Tabelle 4.4: Mapping der Contact und Path Objekte

Das *Contact*-Objekt, siehe Tabelle 4.4a, ist nicht zwingend erforderlich. Es bietet sich jedoch an, dieses in die Beschreibung mit hinein zunehmen, da im Governance Repository alle hierfür notwendigen Daten gespeichert sind. Tabelle 4.4b zeigt das *paths*-Objekt. Es besteht aus je einem *Ressource*-Objekt für die vorher definierten Ressourcennamen.

Ressource Objekt	
ParentName	*/ressourceName*
(Für jede Operation mit passender Ressource) Definiertes HTTP-Verb	siehe 4.5b

(a) Mapping des Ressource Objekts

Operation Objekt	
ParentName	*Definiertes HTTP-Verb*
summary	Name der WSDL-Operation
description	(Falls vorhanden) Inhalt des documentation-Elements in der operation-Komponente
tags	Name des zur Operation gehörenden Interfaces
(Wenn InputMessage definiert) parameters	siehe Tabelle 4.6a
responses	siehe Tabelle 4.6b

(b) Mapping des Operation-Objekts

Tabelle 4.5: Mapping der Ressource und Operation Objekte

Jedes *Ressource*-Objekt besteht wiederum aus einem oder mehreren *Operation*-Objekten, siehe Tabelle 4.5a. Diese sind nach dem HTTP-Verb benannt, welches vorher für jede WSDL-Operation definiert wurde. Somit fallen in ein *Ressource*-Objekt jeweils alle WSDL-Operationen mit gleicher REST-Ressource und deren HTTP-Verben. Es ist zu erwähnen, dass beim Erfassen der Daten für jede Ressource jedes HTTP-Verb maximal einmal belegt werden sollte. Tabelle 4.5b mappt für jedes *Operation*-Objekt eine Beschreibung, *tags*, mögliche Eingabeparameter, und Antworten. Das *Tags*-Objekt ist ein String, welcher sich aus dem Namen des WSDL-Interface der WSDL-Operation zusammensetzt. So kann die Beschreibung später nach den WSDL-Interfaces gefiltert werden.

Parameter Objekt	
ParentName	*parameters*
in	body
name	Name der InputMessage
schema	siehe Abschnitt 4.2.3

(a) Mapping des Parameter Objekts

Responses Objekt	
ParentName	*responses*
default	siehe Abbildung 4.6b
(Wenn OutputMessage definiert) 200	siehe Tabelle 4.6a

(b) Mapping des Response Objekts

Tabelle 4.6: Mapping der Bodyparameter und Responses

Die Eingabeparameter werden im *Parameter*-Objekt gemappt, die WSDL-InputMessage in den HTTP-Body. Dies ist nicht für jeden Fall optimal, da der Vorteil von Pfad-Parametern verloren geht. Diese müssten jedoch auch vom Nutzer explizit definiert und von einer normalen REST-Ressource unterschieden werden können. Da das Mapping jeglicher Eingabeparameter auf den HTTP-Body funktional keinen Nachteil mit sich bringt, fällt die Entscheidung auf diese einfachere und mit weniger Aufwand verbundene Variante. Neben dem Namen und dem *in*-Objekt definiert Tabelle 4.6a auch noch ein *schema*-Objekt, welches den Datentypen aus dem XSD-Schema repräsentieren soll. Tabelle 4.6b definiert die Ausgabeparameter der jeweiligen Operation. Hierbei muss für jede Antwort ein HTTP-Statuscode definiert werden. Gibt es eine WSDL-OutputMessage in der WSDL-Operation, so wird diese mit dem Statuscode 200 gemappt. Des Weiteren wird noch ein default-Statuscode definiert, welcher zur Fehlerbehandlung dient.

Abbildung 4.6b zeigt den YAML-Code des Default-Codes. Abbildung 4.6a stellt das Mapping der WSDL-OutputMessage dar.

200 Response Objekt	
ParentName	*responses*
description	Name der OutputMessage
schema	siehe Abschnitt 4.2.3

(a) Mapping des 200 Response Objekts

```
default:
 description: Unexpected error
  schema:
   type: object
   properties:
    code:
     type: integer
     format: int32
    message:
     type: string
```

(b) Default Response in YAML-Syntax

Abbildung 4.6: Mapping des 200 Response Objekts und Darstellung der Default Response

Schema-Objekt Die Open API Spezifikation sieht im *Schema*-Objekt eine Definition der verwendeten Datentypen nach einer Schnittmenge [Ope14] der JSON-Schema Spezifikation vor [Int13]. Somit können JSON-Schemas nur in die Beschreibung eingebunden werden, wenn sie

dieser Schnittmenge auch entsprechen. Sollen XSD-Datentypen in die Beschreibung integriert werden, so ist es notwendig, diese erst in JSON-Schema umzuwandeln und dann in die erwähnte Teilmenge zu überführen. Obwohl das JSON-Schema noch am Anfang seiner Entwicklung steht, existieren schon Werkzeuge und Literatur zum Konvertieren von XSD-Typen in JSON-Schema. Das mächtigste Werkzeug ist der *jsonix-schema-compiler* [3]. In der Arbeit von F. Nogatz [NF13] wird außerdem ein konzeptuelles Mapping von XSD zu JSON-Schema beschrieben. Es ist somit nicht trivial, die im WSDL-Dokument spezifizierten Datentypen in die Open API Beschreibung mit einzubinden. Die Definitionsvielfalt von XSD erschwert dies noch zunehmend. Andererseits könnten somit aber auch weniger komplexe XSD-Typen direkt in ein Schema-Object überführt werden. Abbildung 4.7 veranschaulicht dies an einem simplen Datentypen.

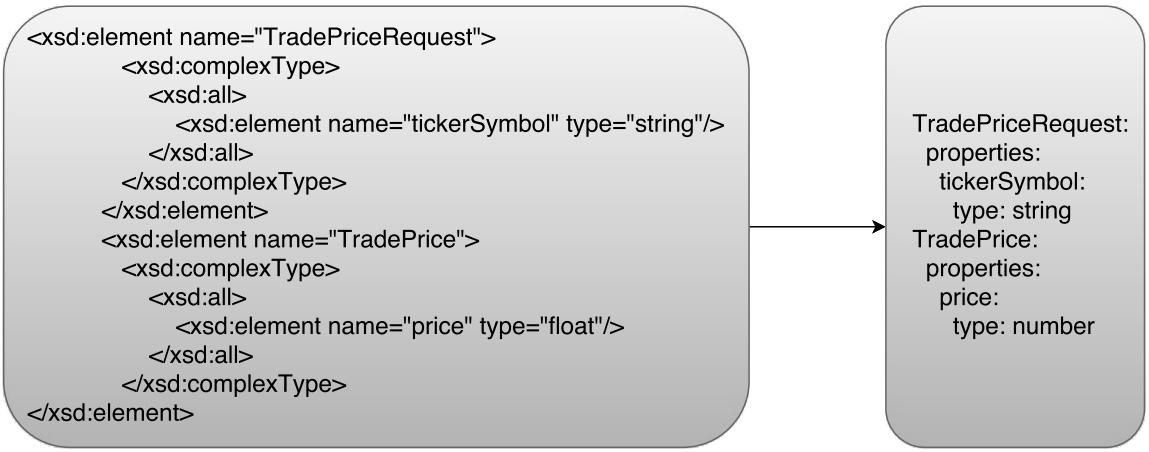


Abbildung 4.7: Mapping eines einfachen XSD-Types zu einem Open API Schema Objekt in YAML Syntax.

4.2.4 Nicht umgesetzte Komponenten

Beim Mapping von WSDL zu Open API wurden einige Komponenten der WSDL-Spezifikation nicht berücksichtigt. So kann beispielsweise die Möglichkeit der Fehlerbehandlung mit Hilfe von Fault-Interfaces- oder Bindings vorerst vernachlässigt werden. Auch die Vererbung von Interfaces wurde nicht berücksichtigt. Dennoch können mit dem aktuellen Mapping alle noch fehlenden Komponenten ergänzt werden.

Die entstehende Open API-Beschreibung ist valide und funktional vollständig, allerdings wurden keine Funktionen wie zum Beispiel HTTP-Security oder HTTP-Header gemappt. Hierfür ist beispielsweise ein vollständiges HTTP-Binding oder SOAP-Binding im WSDL-Dokument oder wieder eine ausführlichere Nutzereingabe erforderlich. Dies ist jedoch nicht immer der Fall bzw. erfordert zu großen Aufwand und technisches Vorwissen seitens des Nutzers.

[3] JSONix Schema Compiler: https://github.com/highsource/jsonix-schema-compiler

5 Implementierung

Folgend soll das in Kapitel 4 gezeigte Konzept mit Hilfe des Governance Repository Prototypen umgesetzt werden. Hierdurch wird die Umsetzbarkeit des theoretisch vorgestellten Imports bzw. Exports von Servicebeschreibungen und dem Mapping von REST zu SOAP vorgestellt.

Implementiert wurden insgesamt drei Komponenten, einmal der Wsdl2Rdf-Importer, die *ServiceDescription*-Komponente und die *UtilityImplementation*-Komponente, siehe Abbildung 5.1. Der *Wsdl2Rdf* -Importer dient dazu, eingehende WSDL-Dokumente in einen RDF-Graphen zu überführen, um diesen in einen beliebigen TripleStore importieren zu können. Die *ServiceDescription*-Komponente dient zur Überführung der WSDL-Beschreibung in ein Java-Objekt und verarbeitet dieses weiter, um es dann zum Beispiel im Frontend anzuzeigen oder in eine Open API Beschreibung zu exportieren. Die *UtilityImplementation* implementiert ein von der *ServiceDescription* bereitgestelltes Interfaces, um sich einerseits auf einen TripleStore zu verbinden und andererseits auf den *Wsdl2Rdf* -Importer zuzugreifen.

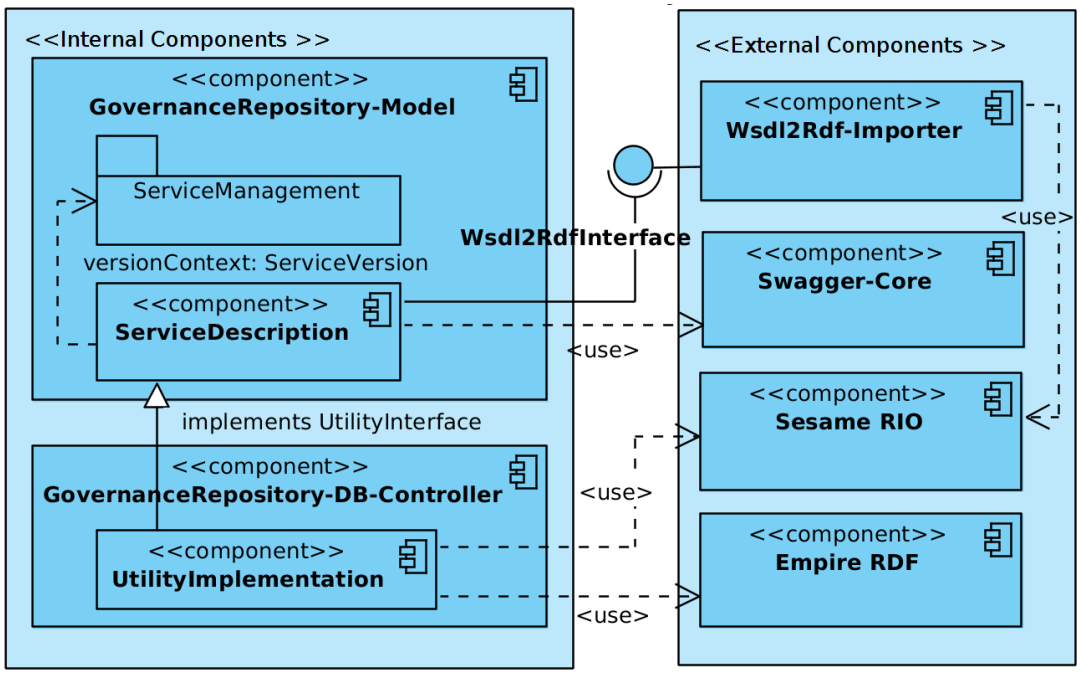


Abbildung 5.1: Komponentendiagramm der implementierten Anwendung

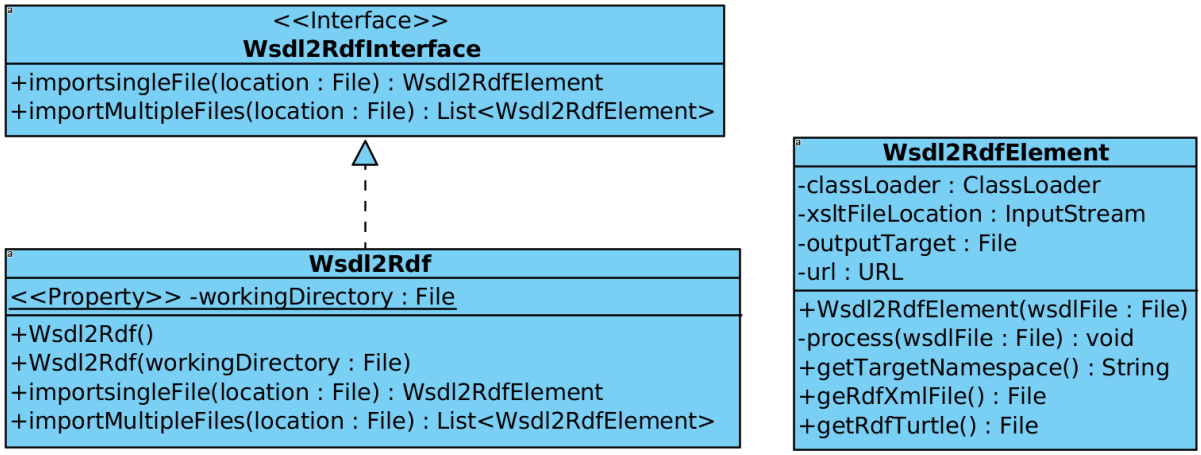


Abbildung 5.2: Klassendiagramm der erstellten Komponente zum Umwandeln eines WSDL-Dokument in einen RDF-Graphen

5.1 WSDL2RDF-Importer

Der *WSDL2RDF*-Importer dient dazu, die zu importierende WSDL-Datei in einen RDF-Graphen zu überführen. Hierfür wird die in Abschnitt 4.1.1 vorgestellte WSDL-zu-RDF Ontologie, des W3C [Kop07b], implementiert. Zwei Implementierungsansätze von Jacek Kopecky sind schon vorhanden. Eine nicht ganz vollständige XSLT-Datei [Kop06a] und eine Java-Implementierung [Kop07a]. Da die Java-Implementierung jedoch nicht fehlerfrei ist und deren Anpassung an fehlende Literale zeitaufwändiger wäre als mit der XSLT-Datei, wurde sich trotzdem für die Transformation der WSDL-Beschreibung in eine XML/RDF-Datei mit Hilfe der XSLT-Datei entschieden.

Zur Verarbeitung der gegebenen XSLT-Datei wurde der Xalan[1] XSLT Prozessor verwendet. Xalan ist ein von der Apache Foundation entwickelter Prozessor zum Verarbeiten von XSLT-Beschreibungen. Hauptmerkmal ist, dass der unter der Apache Lizenz veröffentlichte Prozessor auch XSLT Extensions implementiert. Diese werden in der gegebenen Datei verwendet.

Listing 5.1 zeigt einen Ausschnitt des Umwandelns der WSDL-Endpoint-Komponente in entsprechende RDF/XML-Triple. Um die bereitgestellte XSLT-Datei für den WSDL2RDF-Importer nutzen zu können, mussten noch einige Fehler und zusätzliche Triple hinzugefügt werden. Folgende Dinge wurden abgeändert oder hinzugefügt:

- Aktualisierung des WSDL2.0-Namespaces auf den allgemeineren und häufiger verwendeten Namespace *http://www.w3.org/ns/wsdl*
- Die XSLT-Datei verfügt über verschiedene Abbruchbedingungen, falls bestimmte Komponenten fehlen oder der Prozessor bestimmte XSLT-Funktionen nicht unterstützt.

[1]XALAN XSLT Prozessor https://xml.apache.org/xalan-j/

Diese waren jedoch fehlerhaft und mussten entfernt oder abgeändert werden. Eine Terminierung des Programms lässt sich beispielsweise mit dem Element *<xsl:message terminate="yes"/>* hervorrufen.

- Da in der vorhanden Transformation keine Literale hinzugefügt, diese aber zum Anzeigen in der Oberfläche des Governance Repositories benötigt werden, wurden Literale für die Namensattribute der einzelnen WSDL-Komponenten hinzugefügt. Dies geschieht mit dem *<rdfs:label rdf:parseType="Literal"/>* Element. Es erstellt ein Triple zwischen einer WSDL-Komponente und deren Namensattribut. Als Prädikat wird *rdfs:label* verwendet, welches die Definition eines für den Menschen lesbaren Namens bedeutet.
- Als letztes wurden Triple für das Einlesen des HTTP-Bindings, falls vorhanden, aus Abschnitt 4.1.1, hinzugefügt.

Abbildung 5.2 veranschaulicht die Klassen der *WSDL2RDF*-Komponente. Die besagte Komponente implementiert ein Interface, welches Funktionen zum Umwandeln von entweder einer oder mehreren WSDL-Beschreibungen bereitstellt. Zurück geliefert werden ein oder mehrere *Wsdl2RdfElement*-Objekte, welche es ermöglichen, die in der *process*-Methode erstellten Triples in verschiedene Ausgabeformate zu exportieren. Die *process*-Methode erstellt die Triples, indem sie den XALAN-Prozessor anstößt und diese in einen lokalen Sesame Triplestore lädt. Zusätzlich wird für jeden Graphen, in der *getTargetNameSpace*-Methode der Targetnamespace extrahiert und zwischengespeichert, um diesen einerseits wie in Abschnitt 4.1.1 erwähnt, als Kontext und andererseits zum Erstellen von Instanzen der Ontologie-Klassen zu nutzen. Um eine Weiterentwicklung der XSLT-Datei und des Quellcodes für beispielsweise weitere WSDL-Extensions zu unterstützen, wird das WSDL2RDF-Projekt mit Veröffentlichung der Arbeit auf Github bereitgestellt[2].

[2]WSDL2RDF-Importer https://github.com/TwinbrotherPro/Wsdl2Rdf-Importer

Listing 5.1 XSLT-Ausschnitt, der das Umwandeln der *endpoint*-Komponente in RDF Triple darstellt, abgeändert aus [Kop06a]

```
<xsl:template match="wsdl:endpoint">
      <xsl:variable name="wsdl-tns"
            select="ancestor-or-self::wsdl:description/@targetNamespace" />
      <xsl:variable name="wsdl-namespace"
            select="concat(str:split($wsdl-tns, '#')[1], '#')" />

      <rwsdl:Endpoint
            rdf:about="{concat($wsdl-namespace, 'wsdl.endpoint(', parent::*/@name,
                '/', @name, ')')}">
            <rdfs:label rdf:parseType="Literal">
                  <xsl:value-of select="@name" />
            </rdfs:label>
            <xsl:variable name="binding-ns">
                  <xsl:call-template name="qname-ns">
                        <xsl:with-param name="qname" select="@binding" />
                        <xsl:with-param name="node" select="." />
                  </xsl:call-template>
            </xsl:variable>
            <xsl:variable name="binding-name">
                  <xsl:call-template name="qname-local">
                        <xsl:with-param name="qname" select="@binding" />
                        <xsl:with-param name="node" select="." />
                  </xsl:call-template>
            </xsl:variable>
            <rwsdl:binding
                  rdf:resource="{concat(str:split($binding-ns, '#')[1],
                      '#wsdl.binding(', $binding-name, ')')}" />
            <xsl:if test="@address">
                  <rwsdl:address rdf:resource="{@address}" />
            </xsl:if>
            <xsl:apply-templates
                  select="*|@*[namespace-uri() != '' and namespace-uri() !=
                      'http://www.w3.org/2006/01/wsdl' and namespace-uri() !=
                      'http://www.w3.org/2001/XMLSchema-instance' and
                      namespace-uri() != 'http://www.w3.org/2001/XMLSchema']"
                  mode="linking" />
      </rwsdl:Endpoint>
</xsl:template>
```

5.2 ServiceDescription-Komponente

Die *ServiceDescription*-Komponente dient einerseits dazu, gegebene WSDL-Dateien, mit der Hilfe des *WSDL2RDF*-Importers, in den Triple-Store des Governance Repositories zu importieren und alle notwendigen Daten in ein Java-Objekt zu schreiben und andererseits dazu die importierte WSDL-Beschreibung in eine Open API Beschreibung zu exportieren. Abbildung 5.3 legt das Klassendiagramm dieser Komponente dar.

Um dies zu realisieren wird im Datenbank Controller des Repositories das *UtilityInterface* ausimplementiert, welches es ermöglicht einfache SPARQL Selects und Updates an den Sesame Triple-Store zu senden, einen WSDL-RDF-Graphen zu löschen oder auf die *WSDL2RDF*-Komponente zuzugreifen. Die *IriReferences*-Klasse dient dazu, bei einem gegeben Namespace IriReferenzen für die einzelnen WSDL-Komponenten zu generieren. Mit ihnen ist es möglich, durch den RDF-Graphen zu traversieren oder neue Triples zu erstellen. Soll eine WSDL-Beschreibung importiert werden, so geschieht dies durch das Erstellen eines neuen *WsdlBuilder*-Objekts, welches aus der importierten Datei und den *WSDL-Model-Classes* ein *WsdlDescription*-Objekt erstellt. Hierfür wird mit SPARQL, über die Implementierung des UtilityInterface, durch den Graphen iteriert und alle für das *WsdlDescription*-Objekt notwendigen Informationen ausgelesen.

Abbildung 5.4 zeigt das Java-Modell des *WsdlDescription*-Objekts. Die Hauptklasse ist die *WsdlDescription*-Klasse. Sie beinhaltet alle Basiskomponenten einer WSDL-Beschreibung, wie *WsdlInterface*, *WsdlBinding* und *WsdlService*. Deren jeweilige Unterkomponenten werden auf die gleiche Weise umgesetzt. Alle Komponenten sind von der WsdlComponentElement-Klasse abgeleitet. Die Attribute *ID*, *Name* und der optionale *Documentation*-Operator können in allen WSDL-Komponenten vorkommen und sind deswegen in dieser generalisierten Klasse zusammengefasst. Die *WsdlBindingOperation*-Klasse kann mit beliebigen Spezialisierungen erweitert werden. Eine umgesetzte Spezialisierung ist die *WsdlHttpOperation*, welche die Daten des HTTP-Bindings beinhaltet. Der RDF-Graph der importierten WSDL-Datei wurde in ein Java-Modell umgewandelt, da mit diesem schnell und flexibel auf alle Informationen des Graphen zugegriffen werden und die *ServiceDescription*-Komponente alle Daten des Graphen bereitstellen kann. WSDL-Fault-Komponenten wurden vorerst nicht umgesetzt, können aber schnell an das vorgestellte Modell ergänzt werden.

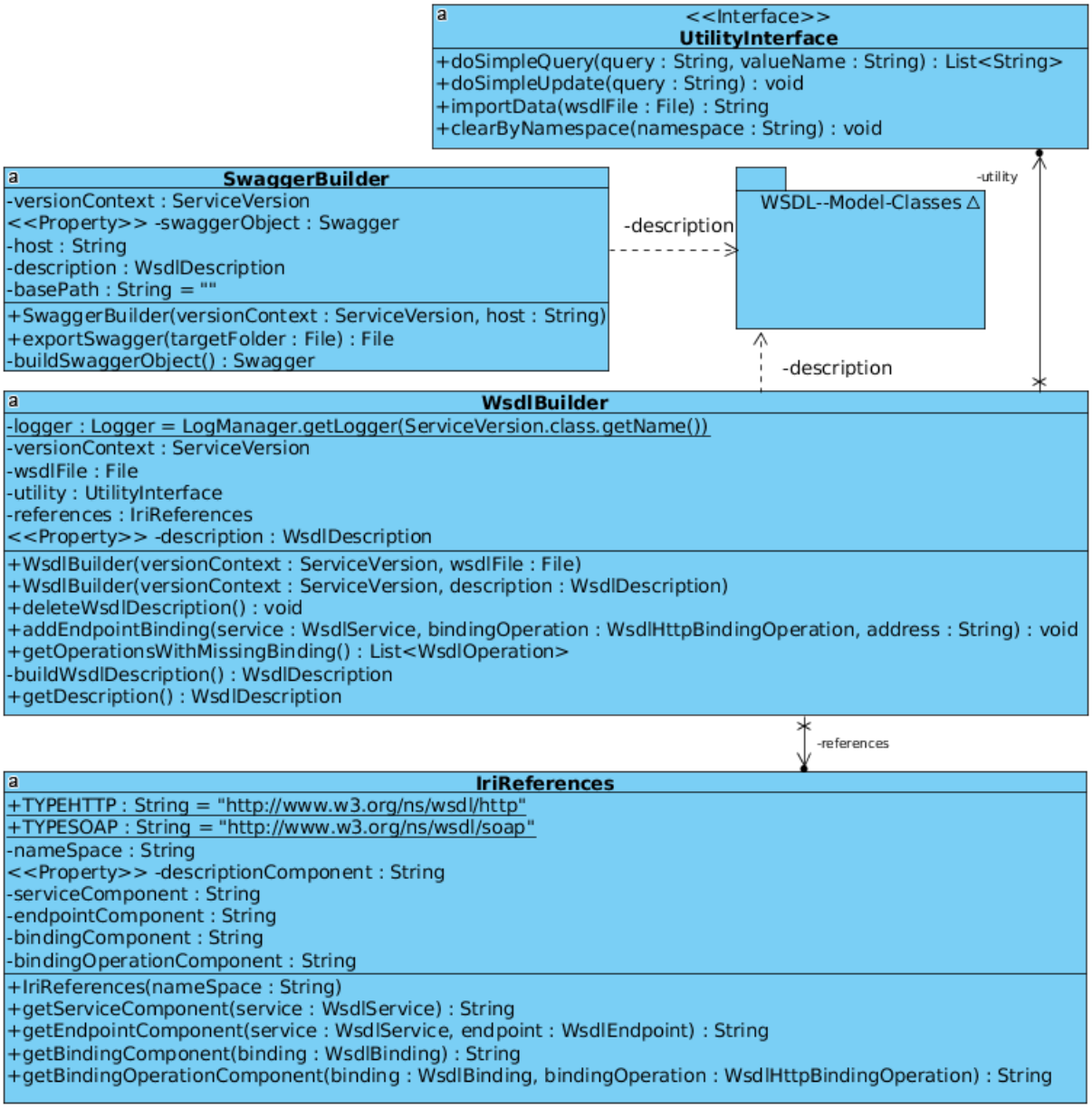


Abbildung 5.3: Klassendiagramm der erstellten Komponente zum Einlesen und Auslesen von Servicebeschreibungen

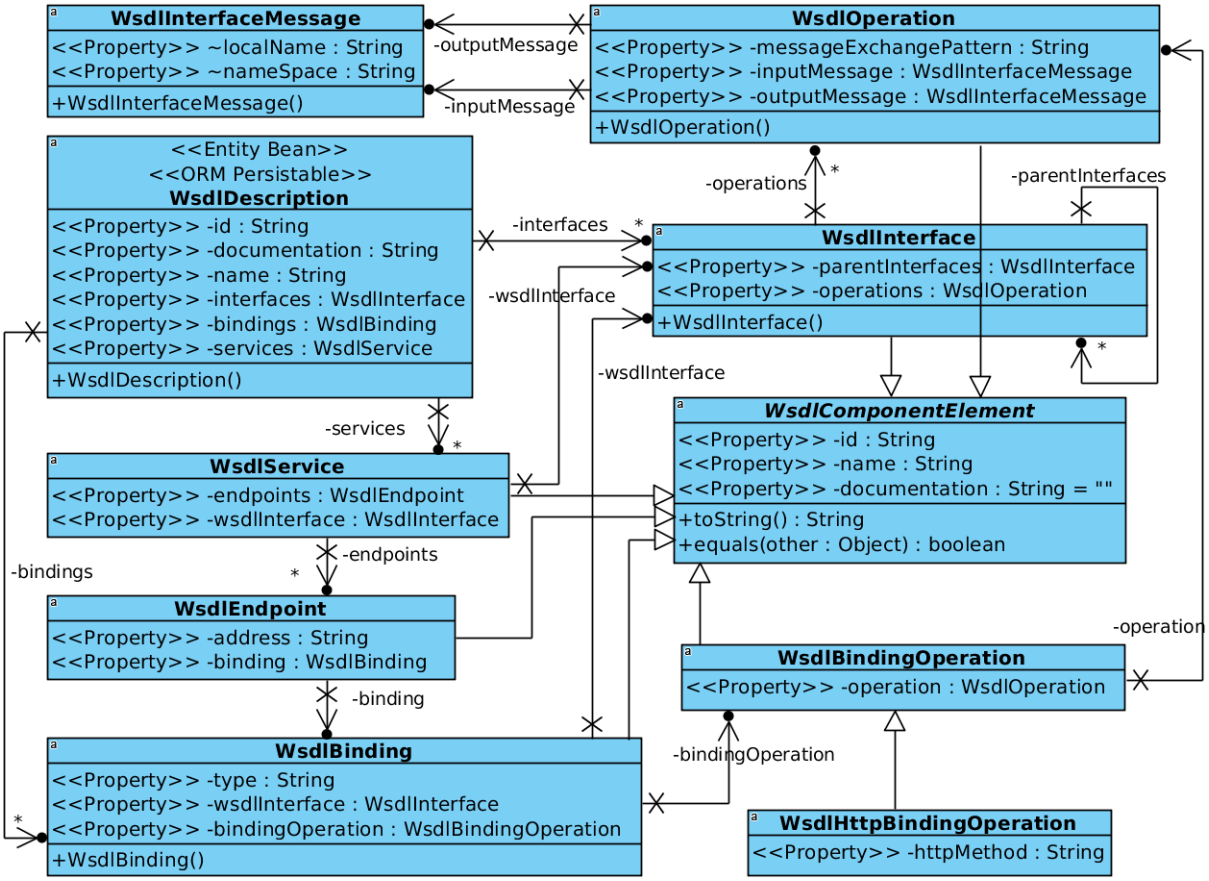


Abbildung 5.4: Klassendiagramm der WSDL-Model-Klassen

Ist das *WsdlDescription*-Objekt erzeugt so wird, wie in Abschnitt 4.1.2 beschrieben, mit Hilfe des *VersionContexts* im *WsdlBuilder*, ein Triple zwischen der zugehörigen Serviceversion und der WSDL-Beschreibung angelegt. Hierdurch kann von der Serviceversion auf die passende Beschreibung zugegriffen werden. Die *getWsdlDescription*-Methode liefert anschließend das erstellte *WsdlDescription*-Objekt. Das Löschen eines WSDL-RDF-Graphen erfolgt ebenso über die *WsdlBuilder*-Klasse mit der *deleteWsdlDescription*-Methode.

5.2.1 Anzeigen der WSDL-Datei

Die *ServiceDescription*-Komponente wird über die Serviceversion in das Modell integriert und es können die für den Nutzer relevanten Daten im Frontend angezeigt werden. Abbildung 5.5 zeigt eine exemplarische Serviceversion mit denen vom Beispielservice aus Abschnitt 3.3 stammenden Daten. Die aus dem WSDL-Dokument importierten Daten sind hier wie im Konzept erfasst der Name der Serviceversion, die Beschreibung und die zur Verfügung stehenden Endpunkte. Weitere Daten können einfach hinzugefügt werden, da ein komplettes *WsdlDescription*-Objekt vorliegt.

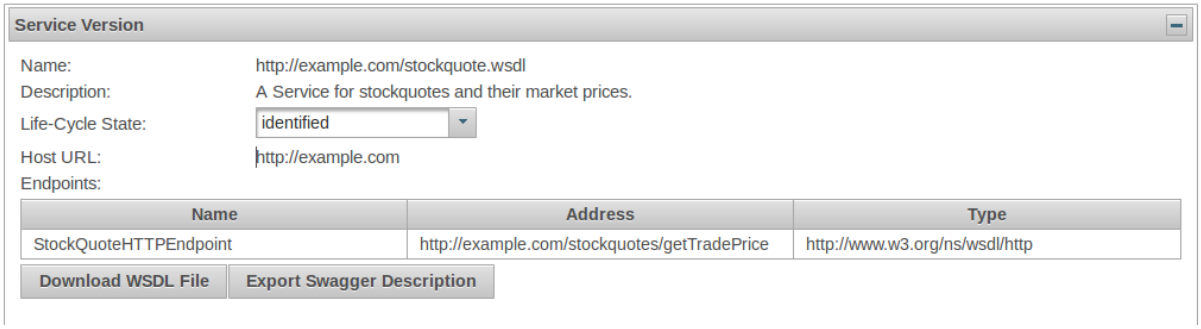


Abbildung 5.5: Screenshot aus dem Governance Repository für die importierte Beschreibung des Beispielservices.

5.2.2 Open API-Export

Wie in Abbildung 5.5 schon angedeutet, kann über den Button „Export Swagger Description“ die importierte WSDL-Datei in eine Open API/Swagger Beschreibung exportiert werden. Vor dem Export öffnet sich ein Dialog, in welchem die im Konzept in Abschnitt 4.1.1 zu erfassenden REST-Ressourcen und im Falle noch fehlend, eine Host-URL definiert werden können. Abbildung 5.6 zeigt den Dialog für den Stockquoteservice, für welchen ein SOAP-Binding definiert wurde. Die *WsdlBuilder*-Klasse bietet zwei Methoden an. Die *getOperationWithMissingBinding*-Methode liefert alle WSDL-Operationen zurück, für welche noch kein HTTP-Binding definiert wurde. Die *addEndpointBinding*-Methode ermöglicht es, fehlende HTTP-Bindings und ihre Endpunkte dem WSDL-RDF-Graphen hinzuzufügen. Im Dialog wird also die Operation *getLastTradePrice* angezeigt, da für diese kein HTTP-Binding existiert.

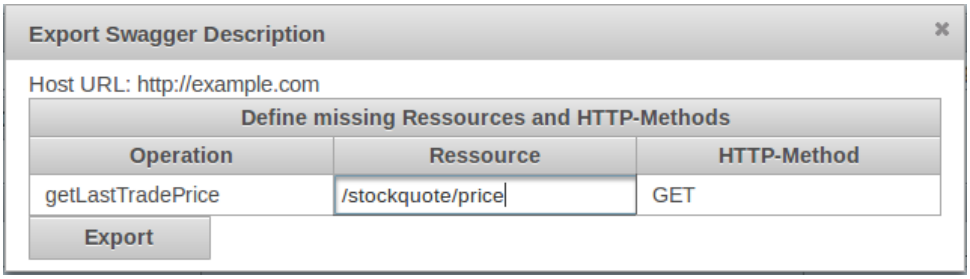


Abbildung 5.6: Screenshot des Dialogs zum Hinzufügen fehlender Daten.

Sind alle fehlenden Daten erfasst, so kann mit dem Button „Export“ die WSDL-Beschreibung und die hinzugefügten Daten in eine Open API Beschreibung exportiert werden. Dies wird mit Hilfe einer Instanz der *SwaggerBuilder*-Klasse realisiert. Der *SwaggerBuilder* erstellt mit Hilfe der *Swagger-Core*-Komponente[3] ein *Swagger*-Objekt, ähnlich wie im *WsdlBuilder*, welches alle Inhalte gemäß dem in Abschnitt 4.2.3 gezeigten Mapping enthält.

[3]Swagger-Core https://github.com/swagger-api/swagger-core

Schema Objekt	
ParentName	*XSD-Schemreferenz des Datentyps*
type	'string'

Tabelle 5.1: Mapping der XSD Schema Referenz

Beim Mapping der Datentypen musste bei der Implementierung jedoch eine Abweichung vorgenommen werden. Wie im Konzept beschrieben, sollten in WSDL beschriebene XSD-Datentypen an passende Businessobjekte gehängt und dann in ein JSON-Schema exportiert werden, welches dann wieder an die Open API Beschreibung angehängt wird. Dies ist aufgrund der verwendeten Schnittmenge des JSON-Schemas in der Open API Spezifikation jedoch nicht möglich. Ein eigenes Mapping von der JSON-Schema-Spezifikation hin zu Open API Schema-Objekten wurde aufgrund des Umfangs nicht umgesetzt. Um eine sichere Typisierung in ein valides Open API Dokument zu bringen, wird die Referenz des Datentyps auf das XSD-Dokument und den XSD-Typ in einem Open API Datentyp gemappt. Eingehende Nachrichten werden als String erkannt und können mit dem im Parameter angegebenen XSD-Schema über ein externes Tool abgeglichen werden. Tabelle 5.1 erklärt das Mapping des dadurch entstehenden Schema-Objekts. Für die nächste Version der Open API Spezifikation soll eine volle Unterstützung des JSON-Schemas umgesetzt werden.

Nach dem Erstellen des *Swagger*-Objekts lässt sich dieses und dessen Entitäten mit Hilfe des Jackson JSON Processors[4] in ein JSON-Objekt umwandeln, welches in der *exportSwagger*-Methode in eine Datei geschrieben und im Frontend zum Download bereitgestellt wird. Der Jackson JSON Processor ist eine unter der Apache License veröffentlichte Java Bibliothek, welche es unter anderem ermöglicht, ganz normale Java Objekte bzw. POJOs (Plain Old Java Objects), in ein JSON-Objekt zu überführen oder JSON-Objekte in diese zu überführen. Listing 5.2 zeigt eine Open API Beschreibung des aus diesem Mapping heraus entstandenen Stockquote-REST-Services. Zum besseren Verständnis wurde das JSON-Objekt in eine äquivalente YAML-Repräsentation umgewandelt.

[4]Jackson JSON Processor http://wiki.fasterxml.com/JacksonHome

Listing 5.2 Mit der ServiceDescription-Komponente exportierter Stockquote Service

```
swagger: '2.0'
info:
  description: A Service for stockquotes and their market prices.
  version: 'http://example.com/stockquoteSoap.wsdl'
  title: 'Service 2 - http://example.com/stockquoteSoap.wsdl'
host: example.com
schemes:
  - http
paths:
  /stockquote/price:
    get:
      tags:
        - StockQuoteInterface
      description: ''
      parameters:
        - in: body
          name: TradePriceRequest
          required: true
          schema:
            properties:
              'http://example.com/stockquote.xsd#TradePriceRequest':
                type: string
      responses:
        '200':
          description: TradePrice
          schema:
            type: object
            properties:
              'http://example.com/stockquote.xsd#TradePrice':
                type: string
        default:
          description: Unexpected error
          schema:
            $ref: '#/definitions/Error'
definitions:
  Error:
    properties:
      code:
        type: integer
        format: int32
      message:
        type: string
```

6 Zusammenfassung und Ausblick

Die nachfolgenden zwei Abschnitte befassen sich mit einer kurzen Zusammenfassung der Arbeit und analysieren das Konzept mit dessen Implementierung auf die Umsetzung der gegebenen Aufgabenstellung hin. Anschließend erörtert ein Ausblick mögliche, in einer zukünftigen Arbeit zu untersuchenden Aspekte.

6.1 Zusammenfassung und Analyse

Ziel dieser Arbeit war es, eine Möglichkeit zum Importieren von Servicebeschreibungen in ein SOA Governance Repository zu schaffen und diese anschließend in eine weitere Beschreibungssprache zu exportieren. Hierdurch sollte eine Abstraktion zwischen der technischen Realisierung der Schnittstelle und der Funktionalität des Services geschaffen werden. Beim Import handelte es sich um WSDL-Beschreibungen für sowohl SOAP als auch REST-Services. Für den Export wurde nach einer vorangegangenen Analyse eine ins Unternehmensumfeld passende Sprache zum Beschreiben von REST-Services ermittelt. Die importierten WSDL-Beschreibungen wurden dann, wenn notwendig, in REST-Schnittstellen umgewandelt und in die ermittelte Beschreibungssprache exportiert.

Als mögliche Beschreibungssprachen kamen *WSDL 2.0*, *WADL*, *Open API*, *RAML* und *API Blueprint* infrage. Alle fünf eignen sich, um einen REST-Service zu beschreiben. Open API ist jedoch die am weitesten verbreitete und die mit den meisten Werkzeugen unterstützte Beschreibungssprache. Seit dem Übergang zur Linux Foundation und der Gründung der Open API Initiative entwickelt sich die Open API Spezifikation immer mehr zum gängigen Standard für REST-API-Beschreibungen.

Nach der Analyse wurde ein Konzept zum Import von WSDL-Beschreibungen in ein SOA Governance Repository entwickelt. Zu importierende WSDL-Dokumente wurden dafür in ein RDF-Datenmodell überführt. Dies ermöglichte die einfache Integration der Daten in das bestehende Datenmodell des Repositories. Um nun die WSDL-Beschreibungen in Open API überführen zu können, wurde ein Konzept zum Konvertieren von SOAP-Services in REST-Services vorgestellt. Hierfür wurden die bestehenden Daten mit weiteren Informationen, wie zum Beispiel einem WSDL-HTTP-Binding erweitert. Somit konnten alle SOAP-Services in einen funktional äquivalenten und nach dem Richardson Maturity Model Level 2 REST-Service exportiert werden. Anschließend wurde ein Mapping vom WSDL-RDF-Graphen in eine Open

API Repräsentation konzipiert. Hierfür wurden auch Daten aus dem schon bestehenden Governance Repository Modells verwendet, um eine möglichst vollständige und valide Beschreibung zu erhalten.

Abschließend wurde das vorgestellte Konzept in einer prototypischen Implementierung des SOA Governance Meta Models umgesetzt. Hierfür wurden drei Komponenten entwickelt. Die *WSDL2RDF*-Komponente zum Überführen einer WSDL-Beschreibung in einen RDF-Graphen, die *ServiceDescription*-Komponente welche das WSDL-Dokument in ein Java-Objekt umwandelt, um dieses einerseits in der Nutzeroberfläche des Repositories anzuzeigen und andererseits in eine Open API Beschreibung zu exportieren. Als dritte Komponente wurde die *Utility*-Komponente, zur Verbindung zwischen *ServiceDescription*-Komponente und RDF-Modell, entwickelt. Um eine Weiterentwicklung an der Implementierung der WSDL-Ontologie zu fördern und diese der Community bereitzustellen, wird die *WSDL2RDF*-Komponente[1] in einem Github Repository veröffentlicht. Auch die *ServiceDescription*-Komponente wird entsprechend veröffentlicht.

Das Umwandeln eines Nachrichtenprotokolls in ein anderes, vom Paradigma völlig verschiedenen, Protokolls und das Exportieren einer Beschreibungssprache in eine Sprache mit vollkommen anderen Konzepten, brachte einige Herausforderungen mit sich. Schon das Suchen nach einer geeigneten Beschreibungssprache, für das relativ neue REST Paradigma, war mit Schwierigkeiten verbunden. Einerseits sind Standards noch immer nicht etabliert. Andererseits besteht ein Mangel an Erfahrung seitens der Community, um eine unmissverständliche und fehlerfreie Spezifikation zu veröffentlichen. Einschränkungen wie die fehlenden REST-Ressourcen vom Nutzer abzufragen und nicht automatisiert aus der WSDL-Beschreibung zu erlangen, mussten getroffen und auf einige Elemente beim Mapping verzichtet werden. Trotzdem wurde mit Open API eine passende Beschreibungssprache gefunden und erfolgreich eine Möglichkeit zum Importieren von Servicebeschreibungen und dem anschließenden Export in eine REST-Beschreibung umgesetzt. Darüber hinaus wurde eine Abstraktion von der technischen Realisierung eines Services und seinen Funktionalitäten geschaffen. Serviceanbieter können einfacher Services in das Governance Repository einpflegen und die exportierten Schnittstellenbeschreibungen als Grundlage für eine neue REST-Schnittstelle verwenden.

6.2 Ausblick

Zum behandelten Thema existieren einige Aspekte, welche in dieser Arbeit nicht bearbeitet wurden. Außerdem ergaben sich neue Problemstellungen, welche in folgendem Abschnitt aufgezeigt werden.

Das im Konzept vorgestellte Mapping wurde für die Richtung von WSDL zu Open API gezeigt und umgesetzt. Ein weiterer zu implementierender Anwendungsfall wäre es, das Szenario aus

[1] WSDL2RDF-Komponente https://github.com/TwinbrotherPro/Wsdl2Rdf-Importer

der Sicht einer bestehenden Open API Spezifikation zu betrachten, welche in WSDL und auch SOAP überführt werden soll. Ein Remapping mit dem vorgestellten Konzept ist nicht möglich, da beim Konvertieren einer WSDL-Beschreibung in Open API zu viele Daten, wie zum Beispiel die Abstrahierung zwischen funktionaler und technischer Beschreibung verloren gehen. Auch wenn im Unternehmensumfeld eher der vorgestellte Anwendungsfall nachgefragt wird, ist ein Remapping trotzdem interessant. Hierdurch kann einerseits eine konsistente Abstraktion zwischen der Schnittstelle und dem Service geschaffen werden. Andererseits können relativ einfach die Vorteile von SOAP für REST-Services genutzt werden.

Des Weiteren wurden, wie in Abschnitt 4.2.4 erwähnt, noch nicht alle Komponenten der WSDL-Spezifikation und Elemente der Open API Spezifikation umgesetzt. Da besonders Funktionalitäten wie WS-Security oder Sicherheitseigenschaften im HTTP-Header von Interesse sind, könnten diese in einer fortführenden bzw. tiefergehenden Arbeit untersucht werden.

Ein weiterer Schritt ist es, die exportierten Open API Beschreibungen zu nutzen, um aus ihnen einen *REST-zu-SOAP-Wrapper-Service* zu generieren. Einen Service, welcher mit seiner REST-Schnittstelle Anfragen annimmt und diese an eine existierende SOAP-Schnittstelle weiterleitet. Dadurch können existierende SOAP-Services schnell für REST-Anfragen bereitgestellt werden ohne die Schnittstelle neu implementieren zu müssen. Weitere Aspekte wie das Cachen von zustandslosen SOAP-Anfragen oder das Routing an verschiedene Instanzen des SOAP-Service über den REST-Wrapper, müssen in Betracht gezogen werden.

Bei genauerer Auseinandersetzung mit dem REST Paradigma fällt auf, dass dieses eine Selbstbeschreibung der Ressourcen, beispielsweise mit HATEOS, fordert. Die Entwicklergemeinschaft scheidet sich in der Frage inwieweit Schnittstellenbeschreibungen das REST Paradigma fördern oder es dadurch sogar einschränken. Um das Ressourcenorientierte Paradigma zu unterstützen, müssen andere Konzepte zum Beschreiben von Schnittstellen, als die triviale Dokumentation der Ressourcen und ihrer Parameter, gefunden werden. Ideen, wie das Definieren eines Zustandsautomaten für den RESTful-Service könnten Bestandteil einer passenderen Beschreibungssprache sein.

Literaturverzeichnis

[Apa04] Apache Software Foundation. *Apache License, Version 2.0*. 2004. URL: http://www.apache.org/licenses/LICENSE-2.0.html (Zitiert auf S. 43, 46).

[Api14] Apiary. *Markdown Syntax for Object Notation*. 2014. URL: https://github.com/apiaryio/mson (Zitiert auf S. 51).

[Api16] Apiary. *API Blueprint Specification*. 2.02.2016. URL: https://apiblueprint.org/documentation/specification.html (Zitiert auf S. 13, 49).

[BE04] K. Brown und M. Ellis. *Best practices for Web services versioning*. 30.01.2004. URL: https://www.ibm.com/developerworks/webservices/library/ws-version/#icomments (Zitiert auf S. 32, 38).

[BEN11] O. Ben-Kiki, C. Evans und I. Net. *YAML Ain't Markup Language (YAML™) Version 1.2*. 14.11.2011. URL: http://yaml.org/spec/1.2/spec.html (Zitiert auf S. 30).

[BG14] D. Brickley und Guha. *RDF Schema 1.1*. Hrsg. von W3C. 18.02.2014. URL: https://www.w3.org/TR/rdf-schema/ (Zitiert auf S. 14).

[BHK13] L. Bruder, F. Harth und N. Karaoguz. „Vergleich von Sprachen, Methoden und Tools zur Modellierung und Beschreibung von REST Schnittstellen". Fachstudie. Universität Stuttgart, 10.04.2013 (Zitiert auf S. 32).

[BM14] D. Brieckley und L. Miller. *FOAF Vocabulary Specification*. 14.01.2014. URL: http://xmlns.com/foaf/spec/ (Zitiert auf S. 15).

[Car14] G. Carothers. *RDF 1.1 N-Quads*. Hrsg. von W3C. 25.02.2014. URL: https://www.w3.org/TR/n-quads/ (Zitiert auf S. 59).

[CBHS05] J. J. Carroll, C. Bizer, P. Hayes und P. Stickler. „Named graphs, provenance and trust". In: *the 14th international conference*. Hrsg. von A. Ellis und T. Hagino. 2005, S. 613 (Zitiert auf S. 59, 60).

[CCMW01] E. Christensen, F. Curbera, G. Meredith und S. Weerawarana. *Web Service Definition Language (WSDL)*. Hrsg. von W3C. 14.03.2001. URL: https://www.w3.org/TR/wsdl (Zitiert auf S. 34).

[CMRW07] R. Chinnici, J.-J. Moreau, Ryman Arthur und S. Weerawarana. *Web Services Description Language (WSDL) Version 2.0 Part 1: Core Language*. Hrsg. von W3C. 19.06.2007. URL: https://www.w3.org/TR/wsdl20 (Zitiert auf S. 29, 38, 58).

[CWL14] R. Cyganiak, D. Wood und M. Lanthaler. *RDF 1.1 Concepts and Abstract Syntax*. Hrsg. von W3C. 18.02.2014. URL: https://www.w3.org/TR/rdf11-concepts/ (Zitiert auf S. 13).

[Dhe04] A. Dhesiaseelan. *XML.com: What's New in WSDL 2.0*. Hrsg. von I. O'Reilly Media. 20.05.2004. URL: http://www.xml.com/lpt/a/1414 (Zitiert auf S. 37).

[Dos05] W. Dostal. *Service-orientierte Architekturen mit Web Services: Konzepte - Standards - Praxis*. 1. Aufl. München: Elsevier Spektrum Akad. Verl., 2005 (Zitiert auf S. 16, 17).

[Erl05] T. Erl. *Service-Oriented Architecture: Concepts, Technology, and Design*. Pearson Education, 2005. URL: https://books.google.com/books?id=y2MALc9HOF8C (Zitiert auf S. 16, 17, 22, 28).

[FL07] J. Farrell und H. Lausen, Hrsg. *Semantic Annotations for WSDL and XML Schema*. 27.08.2007. URL: https://www.w3.org/TR/sawsdl/ (Zitiert auf S. 38).

[Fow10] M. Fowler. „Richardson Maturity Model: steps toward the glory of REST". In: *Online at http://martinfowler. com/articles/richardsonMaturityModel. html* (2010) (Zitiert auf S. 24, 65).

[FT00] R. T. Fielding und R. N. Taylor. „Principled design of the modern Web architecture". In: *the 22nd international conference*. Hrsg. von C. Ghezzi, M. Jazayeri und A. L. Wolf. 2000, S. 407–416 (Zitiert auf S. 23).

[Gar11] R. Garofalo. *Building Enterprise Applications with Windows® Presentation Foundation and the Model View ViewModel Pattern*. Sebastopol: Microsoft Press, 2011. URL: http://gbv.eblib.com/patron/FullRecord.aspx?p=680878 (Zitiert auf S. 20).

[GF07] L. Goasduff und C. Forsling. *Bad Technical Implementations and Lack of Governance Increase Risks of Failure in SOA Projects*. 2007 (Zitiert auf S. 17).

[GKO15] K. Gödecke, P. Kraus und S. Ogando. „Vergleich von Semantic Web Frameworks". Fachstudie. Universität Stuttgart, 9.11.2015 (Zitiert auf S. 14).

[GS04] J. Gruber und A. Swartz. *Daring Fireball: Markdown*. 2004. URL: http://daringfireball.net/projects/markdown/ (Zitiert auf S. 30).

[Had09] M. Hadley. *Web Application Description Language*. Hrsg. von W3C. 9.09.2009. URL: https://www.w3.org/Submission/wadl/#x3-180002.9 (Zitiert auf S. 30, 40, 42).

[Hit08] P. Hitzler. *Semantic Web: Grundlagen*. 1. Aufl. eXamen.press. Berlin: Springer, 2008 (Zitiert auf S. 13).

[Int13] Internet Engineering Task Force. *JSON Schema: core definitions and terminology*. 30.01.2013. URL: http://json-schema.org/latest/json-schema-core.html (Zitiert auf S. 45, 69).

[Jam99] C. James. *XSL Transformations (XSLT)*. Hrsg. von W3C. 16.11.1999. URL: https://www.w3.org/TR/xslt (Zitiert auf S. 15).

[Jon08] A. Jonge. *RESTful SOA using XML*. 12.02.2008. URL: http://www.ibm.com/developerworks/webservices/library/x-restfulsoa/ (Zitiert auf S. 10).

[Kop06a] J. Kopecký. *Partial XSLT stylesheet*. 2006. URL: https://lists.w3.org/Archives/Public/www-ws-desc/2006Feb/0064.html (Zitiert auf S. 7, 72, 74).

[Kop06b] J. Kopecký. *WSDL RDF Mapping: Developing Ontologies from Standardized XML Languages*. 2006. URL: http://link.springer.com/content/pdf/10.1007/11908883_37.pdf (Zitiert auf S. 58).

[Kop07a] J. Kopecký. *WSDL RDF mapping implementation*. 2007. URL: https://lists.w3.org/Archives/Public/www-ws-desc/2007May/0043.html (Zitiert auf S. 72).

[Kop07b] J. Kopecký. *Web Services Description Language (WSDL) Version 2.0: RDF Mapping*. Hrsg. von W3C. 25.06.2007. URL: https://www.w3.org/TR/wsdl20-rdf/#example (Zitiert auf S. 26, 38, 58, 60, 72).

[KSM14] J. Konigsberger, S. Silcher und B. Mitschang. „SOA-GovMM: A meta model for a comprehensive SOA governance repository". In: *Information Reuse and Integration (IRI), 2014 IEEE 15th International Conference on*. 2014, S. 187–194 (Zitiert auf S. 9, 17, 19, 26).

[Lan15] K. Lane. *API Industry Guide on API Design*. 2015. URL: http://pages.3scale.net/api-design-provider-guide-wb.html (Zitiert auf S. 26).

[LL09] K. B. Laskey und K. Laskey. „Service oriented architecture". In: *Wiley Interdisciplinary Reviews: Computational Statistics* 1.1 (2009), S. 101–105 (Zitiert auf S. 16).

[Miy15] V. Y. Miyai. „Rdf-Based Data Model for a SOA Governance Repository". In: (2015) (Zitiert auf S. 20–22, 62, 67).

[ML07] N. Mitra und Y. Lafon. *SOAP Version 1.2 Part 0: Primer (Second Edition)*. 27.04.2007. URL: https://www.w3.org/TR/soap12-part0/ (Zitiert auf S. 22).

[MPP12] B. Motik, P. Patel-Schneider und B. Parsia. *OWL 2 Web Ontology Language Structural Specification and Functional-Style Syntax (Second Edition)*. 11.12.2012. URL: https://www.w3.org/TR/owl2-syntax/ (Zitiert auf S. 14).

[NF13] F. Nogatz und T. Frühwirth. „From XML Schema to JSON Schema-Comparison and Translation with Constraint Handling Rules". Bachelor Thesis. Ulm: University of Ulm, 2013 (Zitiert auf S. 70).

[Ope14] OpenAPI Initiative. *OpenAPI Specification: (fka Swagger RESTful API Documentation Specification)*. 2014. URL: http://swagger.io/specification/ (Zitiert auf S. 30, 69).

[Ope88] Open Source Initiative, Hrsg. *The MIT License (MIT) | Open Source Initiative*. 1988. URL: https://opensource.org/licenses/MIT (Zitiert auf S. 49).

[Ora13] Oracle. *JSR-000338 Java Persistence 2.1 Final Release for Evaluation*. 21.05.2013. URL: http://download.oracle.com/otndocs/jcp/persistence-2_1-fr-eval-spec/index.html (Zitiert auf S. 21).

[PS13] E. Prudhommeaux und A. Seaborne. *SPARQL Query Language for RDF*. 26.03.2013. URL: https://www.w3.org/TR/rdf-sparql-query/ (Zitiert auf S. 14).

[RAM13] RAML Workgroup, Hrsg. *RAML 0.8 Spezifikation*. 2013. URL: https://github.com/raml-org/raml-spec/blob/master/raml-0.8.md (Zitiert auf S. 47, 48).

[RAM15] RAML Workgroup, Hrsg. *RAML 1.0 (RC)*. 2015. URL: http://docs.raml.org/specs/1.0/ (Zitiert auf S. 47).

[RCS15] J. Robie, M. Cyck und J. Spiegel. *XML Path Language (XPath) 3.1*. Hrsg. von W3C. 17.12.2015. URL: https://www.w3.org/TR/xpath-31/ (Zitiert auf S. 15).

[Ric08] L. Richardson. *The Maturity Heuristic*. 2008. URL: https://www.crummy.com/writing/speaking/2008-QCon/act3.html (Zitiert auf S. 24).

[Roz10] M. Rozlog. *REST and SOAP: When Should I Use Each (or Both)?* 1.04.2010. URL: http://www.infoq.com/articles/rest-soap-when-to-use-each (Zitiert auf S. 25).

[RR07] L. Richardson und S. Ruby. *RESTful web services*. Sebastopol, Calif.: O'Reilly, 2007 (Zitiert auf S. 23).

[SCH12] G. Shudi, C. M. Sperberg-McQueen und Henry S. Thompson. *W3C XML Schema Definition Language (XSD)*. Hrsg. von W3C. 4.04.2012. URL: https://www.w3.org/TR/xmlschema11-1/ (Zitiert auf S. 32).

[Sto14] M. Stowe. *RAML vs. Swagger vs. API Blueprint | MIKESTOWE.COM*. 7.07.2014. URL: http://www.mikestowe.com/2014/07/raml-vs-swagger-vs-api-blueprint.php (Zitiert auf S. 46).

[Til15] S. Tilkov. *REST und HTTP: Entwicklung und Integration nach dem Architekturstil des Web*. 3., aktualisierte und erw. Aufl. Heidelberg: dpunkt, 2015 (Zitiert auf S. 40, 46, 56).

[VOH+07] A. S. Vedamuthu, D. Orchard, F. Hirsch, M. Hondo, P. Yendluri, T. Boubez und Ü. Yalçinalp. *Web Services Policy 1.5 - Framework*. Hrsg. von W3C. 29.08.2007. URL: https://www.w3.org/TR/ws-policy/ (Zitiert auf S. 33).

[W3C15] W3C. *W3C Document License*. 14.05.2015. URL: https://www.w3.org/Consortium/Legal/2015/doc-license (Zitiert auf S. 37, 40).

[Wee05] S. Weerawarana. *Web services platform architecture: SOAP, WSDL, WS-Policy, WS-Addressing, WS-BPEL, WS-Reliable Messaging, and more*. Upper Saddle River, NJ: Prentice Hall PTR, 2005 (Zitiert auf S. 26, 27, 29, 31–33, 40).

[ZNS05] M. Zur Muehlen, J. V. Nickerson und K. D. Swenson. „Developing web services choreography standards—the case of REST vs. SOAP“. In: *Decision Support Systems* 40.1 (2005), S. 9–29 (Zitiert auf S. 25).

Alle URLs wurden zuletzt am 08. 05. 2016 geprüft.

Erklärung

Ich versichere, diese Arbeit selbstständig verfasst zu haben. Ich habe keine anderen als die angegebenen Quellen benutzt und alle wörtlich oder sinngemäß aus anderen Werken übernommene Aussagen als solche gekennzeichnet. Weder diese Arbeit noch wesentliche Teile daraus waren bisher Gegenstand eines anderen Prüfungsverfahrens. Ich habe diese Arbeit bisher weder teilweise noch vollständig veröffentlicht. Das elektronische Exemplar stimmt mit allen eingereichten Exemplaren überein.

Ort, Datum, Unterschrift