Friedrich-Alexander-Universität Erlangen-Nürnberg
Technische Fakultät, Department Informatik

MATHIAS ZINNEN
BACHELOR THESIS

DESIGN UND IMPLEMENTIERUNG EINER RESTFUL API FÜR HETEROGENE DATEN

Eingereicht am 25. September 2018

Betreuer: Prof. Dr. Dirk Riehle, M.B.A.
Professur für Open-Source-Software
Department Informatik, Technische Fakultät
Friedrich-Alexander-Universität Erlangen-Nürnberg

Versicherung

Ich versichere, dass ich die Arbeit ohne fremde Hilfe und ohne Benutzung anderer als der angegebenen Quellen angefertigt habe und dass die Arbeit in gleicher oder ähnlicher Form noch keiner anderen Prüfungsbehörde vorgelegen hat und von dieser als Teil einer Prüfungsleistung angenommen wurde. Alle Ausführungen, die wörtlich oder sinngemäß übernommen wurden, sind als solche gekennzeichnet.

Erlangen, 25. September 2018

License

Erlangen, 25. September 2018

Abstract

Civic institutions produce huge amounts of publicly available data. Still, due to its poor quality and inconsistent structure, the potential of this data remains unused for the most part.
The Open-Data-Service was established to enhance the quality of this data and give it a standardized structure. It is supposed to make the potential of open data more accessible to the public by exposing it via a simple, comprehensible and scalable API.
The current most popular approach for the design of such high quality APIs is the architectural style called "REST".
The existing API of the Open-Data-Service has been designed as a RESTful API but does not meet the standards of the architectural style.
The present thesis aims at creating a new API that fulfills all requirements that are needed to be genuinely RESTful.
For this purpose, I carried out a closer analysis of the REST paradigma, as presented by it's inventor Roy Fielding. With this conceptual background, a truly RESTful API could be designed and implemented to help realizing the full potential of open data.

Zusammenfassung

Öffentliche Institutionen generieren jedes Jahr eine riesige Menge an allgemein zugänglichen Daten. Da die Datenqualität aber häufig zu wünschen übrig lässt und das Format der Daten sehr uneinheitlich ist, bleibt der Großteil des darin liegenden Potentials unausgeschöpft.
Der Open-Data-Service wurde ins Leben gerufen, um die Qualität dieser Daten zu verbessern und ihnen eine einheitliche Struktur zu geben. Indem er die verbesserten Daten der Öffentlichkeit über eine einfache, zugängliche und skalierbare API zur Verfügung stellt, soll er einen Beitrag dazu leisten, dieses Potential nutzbar zu machen.
Der Architekturstil "REST" ist der aktuell verbreiteste Ansatz zum Entwurf solcher qualitativ hochwertigen APIs.
Die bestehende API des Open-Data-Service wurde zwar als RESTful API entworfen, genügt aber nicht den Ansprüchen dieses Architekturstils.
Das Ziel der vorliegenden Arbeit besteht darin, eine neue API zu erstellen, die allen Anforderungen genügt, die erfüllt werden müssen, um wirklich RESTful genannt zu werden.
Zu diesem Zweck habe ich zuerst eine genauere Analyse des REST Paradigmas, wie es von dessen Erfinder Roy Fielding eingeführt wurde, angestellt. Vor diesem konzeptuellen Hintergrund konnte schließlich eine API entworfen und implementiert werden, die man mit allem Recht als RESTful bezeichnen kann – damit offene Daten ihr volles Potential entfalten können.

Inhaltsverzeichnis

1 Einleitung **2**

2 Theoretische Grundlagen **4**
2.1 REST und HTTP 4
2.2 Der Architekturstil REST und seine Herleitung 5
2.3 REST Constraints 6
2.3.1 Client-Server 6
2.3.2 Statelessness 6
2.3.3 Cache 6
2.3.4 Layered System 7
2.3.5 Code-on-Demand 7
2.3.6 Uniform Interface 8
2.3.7 Zusammenfassung der Constraints 11
2.4 REST im Vergleich zu anderen Paradigmen 12
2.4.1 Remote Procedure Call (RPC) 12
2.4.2 Graph Query Language (GraphQL) 16

3 Anforderungen **20**
3.1 Funktionale Anforderungen 20
3.1.1 Erfüllung der REST Constraints 20
3.1.2 Versionierung 20
3.1.3 Paginierung 20
3.1.4 Dokumentation 21
3.1.5 Verwendung von Tests 21
3.2 Nichtfunktionale Anforderungen 21
3.2.1 State of the Art Implementierung 21
3.2.2 Minimalinvasive Implementierung 22

4 Architektur des Open-Data-Service **23**
4.1 Bestehende Architektur 23
4.2 Bestehende API 27

5 Design und Implementierung einer RESTful API **32**
5.1 Technologie . . . 32
5.1.1 Bereits verwendete Technologien . . . 32
5.1.2 Neue Technologie . . . 34
5.2 Konzeption der neuen API . . . 41
5.2.1 Versionierung . . . 41
5.2.2 Endpunkte . . . 42
5.2.3 Dokumentation . . . 44
5.2.4 Datenaustausch . . . 44
5.3 Implementierungsdetails . . . 47
5.3.1 Response und Request Klassen . . . 47
5.3.2 Wrapper und Identifiable-Interface . . . 53
5.3.3 Umsetzung der Dokumentation . . . 53
5.3.4 Testcases . . . 55

6 Evaluation **57**
6.1 Funktionale Anforderungen . . . 57
6.1.1 Erfüllung der REST Constraints . . . 57
6.1.2 Versionierung . . . 59
6.1.3 Paginierung . . . 59
6.1.4 Dokumentation . . . 61
6.1.5 Verwendung von Tests . . . 62
6.2 Nichtfunktionale Anforderungen . . . 62
6.2.1 State of the Art Implementierung . . . 62
6.2.2 Minimalinvasive Implementierung . . . 63

7 Fazit **65**
7.1 Rückblick und Ergebnisse . . . 65
7.2 Ausblick . . . 67
7.2.1 Ausbesserungen . . . 67
7.2.2 Erweiterung der Funktionalität . . . 67
7.2.3 Konzeptuelle Weiterentwicklung . . . 69

8 Anhänge **70**
Anhang A Ausführliches JSON API Beispieldokument . . . 71
Anhang B EBNF Darstellung von (Teilen der) JSONAPI Syntax . . . 72
Anhang C Swagger-UI Benutzeroberfläche . . . 73
Anhang D Klassendiagramm der Response Klassen . . . 75
Anhang E Quellcode der Requestvalidierung und -konversion . . . 76
Anhang F Klassen zur Erzeugung der Swagger Schemata . . . 77

Literaturverzeichnis **78**

1 Einleitung

Von der öffentlichen Verwaltung werden Daten in enormen Umfang generiert. Die Konrad-Adenauer-Stiftung beziffert den potentiellen volkswirtschaftlichen Wert öffentlich zugänglicher Daten auf über 40 Mrd. Euro jährlich (Dapp et al., 2016). Dieses Potential liegt weitestgehend brach. Denn die Nutzung und Weiterverarbeitung öffentlicher Daten wird durch ihre uneinheitliche Darstellungsform extrem erschwert. Um die Daten überhaupt lesen und interpretieren zu können, muss häufig ein großer Aufwand betrieben werden. Gerade unter dem Blickwinkel eines *programmable web*, das eine Konsumption der Daten nicht nur durch Menschen, sondern auch durch Softwaresysteme ermöglichen soll,[1] entsprechen die Daten den Qualitätsansprüchen nicht im Geringsten.
Der Open-Data-Service (ODS) ist ein Open-Source Projekt, das das Problem des "Wildwuchses" öffentlicher Daten lösen soll.[2] Er wurde ins Leben gerufen, um öffentliche Daten unter Qualitätsgarantien in einheitlicher, aufbereiteter Form zur Verfügung zu stellen und das Potential von Open Data besser nutzen zu können. Die Aufbereitung durch den ODS ermöglicht es Anwendungsentwicklern, die auf öffentliche Daten angewiesen sind, sich allein auf die Funktionalität ihrer App zu konzentrieren, ohne sich um die Anbindung heterogener Daten über unterschiedliche Schnittstellen kümmern zu müssen.
Die Schnittstelle des ODS ist ein *Application Programming Interface* (API), die im Unterschied zu einem *Graphical User Interface* (GUI) vor allem auf die Konsumption der Daten durch Maschinen ausgelegt ist.
Die Nutzer des ODS sollen nur eine, einheitliche Schnittstelle kennen und bedienen müssen – weswegen das Design dieser Schnittstelle um so wichtiger ist. Um dem allgemeinen Chaos der verschiedenen Repräsentationen offener Daten nicht einfach eine weitere, schlecht zu bedienende Schnittstelle hinzuzufügen, muss die API des ODS einheitlich, intuitiv verständlich und performant sein.
Um diese Ansprüche zu erfüllen, eignet sich das Paradigma des REpresentational State Transfer (REST), das als konzeptueller Rahmen der Architektur des world

[1] Für eine detaillierte Diskussion des Begriffs (vgl. Richardson, 2007).

[2] Sein Quellcode ist, ebenso sowie Instruktionen zur Installation einer Instanz des ODS, unter der URL https://github.com/jvalue/open-data-service verfügbar.

wide web[3] einen nicht zu unterschätzenden Anteil an dessen kometenhaften Aufstieg hat. REST ist - trotz des aktuellen Aufkommen von GraphQL - weiterhin der mit Abstand erfolgreichste und populärste Ansatz des modernen API Designs.
Die bestehende API des Open-Data-Service wurde als RESTful[4] API konzipiert, kann aber leider nicht allen Prinzipien dieses Paradigmas gerecht werden.
Insbesondere erfüllt sie nicht die Anforderung der Hypertextorientierung, die für den REST-"Erfinder" Roy Fielding eine zentrale Bedeutung besitzt.

> *"if the engine of application state (and hence the API) is not being driven by hypertext, then it cannot be RESTful and cannot be a REST API."* (Fielding, 2008)

Die Vehemenz, mit der Fielding noch acht Jahre nach der Veröffentlichung seiner REST Prinzipien die Notwendigkeit der Hypertext-Orientierung betont, unterstreicht, wie wichtig diese für die Umsetzung von RESTful APIs ist.
Ziel der vorliegenden Arbeit ist es, eine API für den ODS zu implementieren und umfassend zu dokumentieren, die allen REST Prinzipien entspricht und zu Recht als RESTful bezeichnet werden kann.
Dafür werden zunächst in Kapitel 2 die theoretischen Grundlagen gelegt, indem der Frage nachgegangen wird, worin der Architekturstil REST eigentlich genau besteht. Daraufhin werden in Kapitel 3 die funktionalen und nichtfunktionalen Anforderungen spezifiziert, denen die Implementierung der RESTful API genügen soll. Für die funktionalen Anforderungen wird jeweils zusätzlich beschrieben, nach welchem Schema die Bewertung der Anforderung vorgenommen wird. Kapitel 4 widmet sich schließlich der Architektur des bestehenden Softwaresystems, sowie im Speziellen der bereits existierenden API. In Kapitel 5 erfolgt zunächst eine Bestandsaufnahme der bestehenden sowie der im Rahmen der Arbeit hinzugefügten Technologie, bevor die wichtigsten Aspekte der Konzeption und schließlich einige der interessantesten Implementierungsdetails vorgestellt werden. Darauf folgt in Kapitel 6 eine Evaluierung der implementierten API auf Basis der in Kapitel 3 spezifizierten Anforderungen. Schließlich werden in einem Fazit (7) die Ergebnisse der vorliegenden Arbeit dargestellt und ein Ausblick auf weitere Entwicklungsmöglichkeiten des ODS gegeben.

[3] Der Entwickler des REST Paradigmas Roy Fielding betont beispielsweise den Einfluss, den REST auf die für das Web grundlegenden Standards URI und HTTP hatte (vgl. Fielding, 2000, S. 36)

[4] Um APIs zu bezeichnen, die die Prinzipien des REST-Paradigmas umsetzen, hat sich der Begriff RESTful eingebürgert.

2 Theoretische Grundlagen

REST ist in aller Munde. Heute wird sich kaum eine Schnittstellendefinition finden, die nicht als "RESTful" bezeichnet wird – oder, wo dieser Hinweis fehlt, eine Rechtfertigung für das Abweichen von REST angegeben wird. REST ist ein de-facto Standard für den Entwurf moderner Webservices. Doch worin besteht REST eigentlich genau? Ist es eine Architektur, ein Architekturstil, ein Paradigma?
Das folgende Kapitel soll den Begriff von REST und dessen theoretische Grundlagen kurz erklären, damit die Anforderung, eine RESTful API zu implementieren, richtig verstanden und präziser gefasst werden kann.

2.1 REST und HTTP

Die erste Fassung von REST wurde von Roy Fielding bereits Mitte der 1990er Jahre entwickelt. Fielding war damals wesentlich an der Spezifikation des für das grundlegende Internetprotokoll HTTP beteiligt und definierte in diesem Rahmen ein einheitliches Konzept für die zustandslose Kommunikation im Web (Tilkov, 2015, S. 9). Dieses Konzept trug ursprünglich noch den Namen "HTTP object model", spätestens mit seiner Dissertation aus dem Jahr 2000 abstrahierte Fielding allerdings von HTTP und bezeichnete es protokollunabhängig als "Representational State Transfer" (Fielding, 2000, S. 127).
REST und HTTP sind damit auf unterschiedlichen Abstraktionsebenen anzusiedeln: HTTP ist eine, aber nicht die einzig mögliche Umsetzung der REST-Prinzipien. Theoretisch wären auch andere Protokolle zur Realisierung von REST denkbar, wenngleich in der Realität RESTful Services fast ausschließlich durch HTTP realisiert werden.
Im Sinne der umfassenden Anwendung von HTTP als Transportprotokoll lässt sich auch das Internet selbst als umfassendste Realisierung eines RESTful Services bezeichnen. Eine Erklärung für den Erfolg dieser „world's largest distributed application" (Fielding, 2000, S. 150) liegt in der Anwendung des Architekturstils REST. Ein moderner Browser kann jede beliebige Internetseite darstellen, oh-

ne dafür irgendein Vorwissen über deren Inhalt zu benötigen. Das hohe Maß an Entkopplung zwischen Server und Client, das dafür notwendig ist, wird durch die konsequente Einhaltung der REST-Prinzipien sichergestellt.

2.2 Der Architekturstil REST und seine Herleitung

Eine *Softwarearchitektur* bezeichnet laut Fielding die konkreten Komponenten eines Softwaresystems, sowie deren Zusammenspiel (Fielding, 2000, S. 7). Welche Rollen die Komponenten innerhalb eines Systems einnehmen, und in welchen Beziehungen diese Rollen stehen wird durch die Architektur vorgegeben.
Einen *Architekturstil* hingegen definiert Fielding als geordnete Menge architektonischer Bedingungen, denen die Komponenten eines Systems und deren Beziehungen untereinander unterliegen (Fielding, 2000, S. 13). Demnach legt ein Architekturstil die Komponenten und deren Beziehungen nicht abschließend fest, sondern definiert Einschränkungen (Constraints), denen ein diese gerecht werden müssen. Der Architekturstil bestimmt kein konkretes Softwaresystem, sondern definiert lediglich einen Raum möglicher Architekturen.
Architekturstile sind demnach eine Abstraktionsebene höher anzusiedeln als Architekturen: Wenn sich Architekturen durch Abstraktion von einer Menge konkreter Softwaresysteme herleiten lassen, stellt ein Architekturstil die Abstraktion einer Menge von Architekturen da - genau der Menge, deren Elemente die Einschränkungen des Architekturstils erfüllen.
REST versteht Fielding als Architekturstil für verteilte Hypermedia Systeme ("architectural style for distributed hypermedia systems", (Fielding, 2000, S. 76).
Fieldings Herleitung des REST-Architekturstils findet ihren Ausgangspunkt in der völlig unbeschränkten Menge aller möglichen, Architekturen, dem "null style" (Fielding, 2000, S.77), die er dann durch hinzufügen von Constraints immer weiter einschränkt, bis nur noch die Architekturen übrig bleiben, die dem REST-Architekturstil genügen.
Die Menge dieser Constraints ist der Architekturstil REST. Im nachfolgenden Kapitel werden die Constraints nacheinander aufgezählt und jeweils kurz erklärt.

2.3 REST Constraints

2.3.1 Client-Server

Die erste Einschränkung erscheint für Webanwendungen so selbstverständlich, dass sie sehr leicht übersehen wird: Benutzerschnittstelle und Datenhaltung sollen voneinander unabhängig sein und Server und Client so voneinander entkoppelt werden. Das ermöglicht es Server und Client, unabhängig voneinander weiterentwickelt zu werden und erhöht damit die Portabilität zwischen unterschiedliche Clientsystemen sowie die Skalierbarkeit der Anwendung (Fielding, 2000, S. 78).

2.3.2 Statelessness

Die Kommunikation zwischen Client und Server muss in einem REST System zustandslos geschehen. Das bedeutet, dass jede Anfrage, die der Client an den Server sendet, alle zum Verständnis der Anfragen nötige Information bereits enthalten muss und sich dabei nicht auf bei dem Server gespeicherte Zustandsinformationen verlassen darf.
Fielding gesteht zu, dass die zustandslose Kommunikation eine erhöhte Redundanz und damit eine Verringerung der Performanz beinhaltet, sieht diesen Nachteil aber durch die die dafür erkauften Vorteile aufgewogen.
Einerseits ermöglicht die eingebaute Redundanz eine höhere Zuverlässigkeit, da sich das System eher mit der erneut gesendeten Information eher von Teilausfällen erholen kann. Andererseits wird die Skalierbarkeit wesentlich erhöht, denn da der Client für die Speicherung des Sitzungszustands verantwortlich sind, wird der Server von einer großen Zahl an parallelen Anfragen sehr viel weniger beeinträchtigt (Fielding, 2000, S. 78f.).

2.3.3 Cache

Die Cache-Einschränkung erfordert, dass jede Antwort auf eine Anfrage die in ihr enthaltenen Daten (sofern sie überhaupt Daten enthält) implizit oder explizit als zwischenspeicherbar (cacheable) oder nicht zwischenspeicherbar (non-cacheable) kennzeichnet.
Wenn eine Antwort als cacheable gekennzeichnet wurde, kann der Client die mit ihr gelieferten Daten später für gleichartige Anfragen wieder verwenden.
Einerseits kann sich damit zwar die Zuverlässigkeit der Kommunikation verringern, da die Daten im Cache des Client nach eingier Zeit möglicherweise stark von denen des Servers abweichen, andererseits sind mit dem Caching enorme Steige-

rungen der Performanz möglich, da im besten Fall viele Anfragen gar nicht erst vom Client zum Server gesendet werden müssen.

2.3.4 Layered System

Der Layered System Constraint ermöglicht eine hierarchische Schichtenarchitektur.
Die Schichten einer REST Architektur dürfen dabei, abgesehen von ihren unmittelbaren Kommunikationsschnittstellen zu den benachbarten Schichten, keinerlei Kenntnis über das Gesamtsystem besitzen.
Dadurch wird einerseits die Gesamtkomplexität des Systems beschränkt, andererseits dessen Skalierbarkeit erhöht.
Eine Voraussetzung für diese Schichtenbildung sind selbstbeschreibende Nachrichten, die weiter unten in der Beschreibung des Uniform Interface Constraints noch thematisiert werden.

2.3.5 Code-on-Demand

REST ermöglicht es, die Funktionalität des Clients durch den Download und die Ausführung von Code in Form von Applets oder Skripts. Die Client Software kann so vereinfacht werden, da nicht alle Features clientseitig implementiert sein müssen.
Fielding führt Code-on-Demand als "optional constraint" ein, da die Möglichkeit zur Umsetzung von äußeren Faktoren abhängig sein kann. Durch die Optionalität kann dessen Umsetzung je nach Szenario eingesetzt werden oder ausbleiben. So wäre es beispielsweise denkbar, dass ein Server innerhalb eines firmeninternen Intranet Code-on-Demand unterstützt, außerhalb aber aus Sicherheitsgründen daraus verzichtet.
Auf den logischen Widerspruch, der in einer *optionalen Einschränkung* liegt, soll hier nicht weiter eingegangen werden.[1]
Für die Funktionalität des Open-Data-Service hat der Constraint keine Bedeutung und soll hier nur der Vollständigkeit halber erwähnt werden.

[1] Fielding selbst weist übrigens explizit auf diesen Widerspruch hin, wenn er bemerkt: "The notion of an optional constraint may seem like an oxymoron" (Fielding, 2000, S. 84)

2.3.6 Uniform Interface

Die Einschränkung des Uniform Interface bildet das Herzstück des Architekturstils REST. Das zeigt sich bereits daran, dass diese Einschränkung selbst wieder vier ihr untergeordnete Einschränkungen enthält, die ich im Folgenden auflisten und kurz erklären werde.
Zunächst müssen allerdings die Begriffe von Ressource und Repräsentation eingeführt werden, da Fieldings spezielles Verständnis dieser Begriffe für die Uniform Interface Constraints grundlegend ist.

Das Konzept von Ressourcen und Repräsentationen

Eine Ressource kann, so Fielding, zunächst jede benennbare Information sein "*Any information that can be named can be a resource*", (Fielding, 2000, S. 88). Dieser sehr unspezifischen Beschreibung fügt Fielding im Folgenden eine präzise Definition hinzu, die wegen ihrer Kürze und Prägnanz hier vollständig wiedergegeben wird:

> *"A resource R is a temporally varying membership function MR(t), which for time t maps to a set of entities, or values, which are equivalent. The values in the set may be resource representations and/or resource identifiers."* *(Fielding, 2000, S. 88)*

Eine Repräsentation hingegen bezeichnet Fielding als "*a sequence of bytes, plus representation metadata to describe those bytes*" (Fielding, 2000, S. 90). Kurz gefasst sind Repräsentationen also das, was nach Anforderung einer Ressource als Antwort über die Leitung geht.
Zentral für Fieldings Begriff einer Ressource ist die konzeptuelle Trennung der Repräsentationen eines Objektes von dessen Zuordnung in einen begrifflichen Rahmen (der Ressource).
Vereinfacht gesagt, stellen Fieldings Ressourcen die Zuordnung vom Konzept (oder einfach dem Namen) eines Objektes zu dessen Repräsentation her. Diese Zuordnung kann statisch oder dynamisch von statten gehen: So wird sich etwa die Repräsentation der Ressource "Der am 21.12.2018 auf der Konferenz Y veröffentlichte Aufsatz des Autors X" nicht mehr ändern, während die Referenz der Ressource "das heutige Wetter" sich je nach Aufenthaltsort und Zeitpunkt verändert. Ein Beispiel aus der Welt der Softwareentwicklung stellen etwa Programmversionen dar, die entweder als "latest version" (dynamische Ressource) oder als "version 1.2.7" (statische Ressource) identifiziert werden können.
Der Mehrwert der Trennung von Ressource (der Identifizierung) und Repräsentation (beispielsweise dem Quellcode des Programms) erschließt im Fall der statischen Ressource nicht unmittelbar, wird aber sofort deutlich wenn man die dynamische Ressource betrachtet: Erst dadurch, dass die Repräsentation des Programms von

ihrer Bezeichnung getrennt wird, lässt sich die Adressierung als “latest version” überhaupt sinnvoll einsetzen.
Dadurch wird es beispielsweise einem Anwender eines Programms möglich, stets dessen neueste Version zu beziehen, ohne die aktuelle Versionsnummer kennen zu müssen.
Wenn Ressourcen als Funktionen verstanden werden, die selbst stabil adressierbar bleiben, aber gleichzeitig dynamisch die Zuordnung zu variablen Repräsentation übernehmen, können Anwender sich einerseits darauf verlassen, dass sie an einem bestimmten Ort am Web stets die Information finden, nach der sie konzeptionell suchen. Andererseits ermöglicht es dem Anbieter der Information, die Repräsentation flexibel an die aktuellen Bedürfnisse anzupassen.

Identification of Resources

Der Architekturstil Rest macht sich die konzeptionelle Trennung zwischen Ressourcen und Repräsentationen zu nutze, indem er fordert, dass zur Identifizierung von Ressourcen ”resource identifier”verwendet werden.
Diese “resource identifier” werden bei HTTP durch *URIs* (Uniform Resource Identifier) realisiert. Eine URI lässt sich definieren als *kompakte Sequenz von Zeichen, die eine abstrakte oder physische Ressource* (im oben definierten Sinne) *eindeutig identifizieren* (vgl. Berners-Lee, Fielding et al., 2005, S. 1).
URIs sollen die beiden älteren Begriffe *URL* (Uniform Resource Locator, zur Lokalisierung einer Ressource) und *URN* (Uniform Ressource Name, zur Benennung einer Ressource) ersetzen und deren Bedeutung vereinigen, indem eine URI beide Aspekte in sich vereint (vgl. Berners-Lee, Fielding et al., 2005, S. 7).
Konkret bedeutet das, dass sich Referenzen nie auf bestimmte Repräsentationen eines Objektes, sondern stets auf dessen konzeptuelle Einordnung beziehen sollen (im einfachsten Falle besteht eine solche Einordnung einfach in einem Namen).
Die URI `"http://www.example.com/path/to/some-Resource.json"` verletzt beispielsweise die Einschränkung, da mit dem “.json” am Ende Bezug auf eine bestimmte Repräsentation der Ressource genommen wird. Wenn nun der bereitstellende Webservice die Repräsentation der Ressource ändern möchte, etwa indem die Ressource im yaml, statt im json-Format bereitgestellt wird, müssen alle bestehenden Referenzen auf die Ressource geändert werden. Diese unflexibilität wird, so Fielding, den Anforderungen nicht gerecht, die das Ausmaß und die Vielfältigkeit der Benutzer des Web an Webservices stellen.

Aus dieser Form der Identifizierung folgt direkt die nächste Einschränkung des Uniform Interface Constraint.

Manipulating of Resources through Representations

Da die Ressourcen unabhängig von ihren Repräsentationen existieren und unabhängig von deren Änderungen stabil bleiben sollen, muss die Manipulation der Ressourcen über ihre Repräsentation geschehen. Um dieser Einschränkung zu genügen darf sich, wenn an einer bestimmten Entität Änderungen vorgenommen werden sollen, nicht die Ressource verändern, sondern nur ihre Repräsentation.

Wenn beispielsweise in einem online Adressbuch die Telefonnummer eines bestimmten Kontakts geändert werden soll, muss die Referenz auf diesen Kontakt (die Ressource) stabil bleiben. Ändern darf sich lediglich der Eintrag zur Telefonnummer in dem Dokument, das zurückgeliefert wird (der Repräsentation), wenn jemand die Kontakt-Ressource anfordert.

Würde sich die Ressource selbst selbst ändern, müssten in der Folge alle anderen Links, die auf den Kontakt bestehen ebenfalls angepasst werden, was nicht mit den Anforderungen an die Skalierbarkeit einer Webanwendung vereinbar wäre.

Self descriptive messages

Die Anforderung selbstbeschreibender Nachrichten (self descriptive messages) soll in erster Linie die Zustandslosigkeit von REST-Services sicherstellen. Um eine einzelne Anfrage oder Antwort zu verstehen soll möglichst wenig Kontextinformation notwendig sein.
Daher muss jede Anfrage über Standardmethoden gestellt werden und durch einen Medientyp angeben, wie möglicherweise enthaltene Daten zu interpretieren sind. Zudem muss jede Antwort explizit die Cacheability (Zwischenspeicherbarkeit) der in ihr enthaltenen Daten angeben (Fielding, 2000, S. 98-99).
Im Fall von HTTP werden diese Einschränkungen unter anderem durch die Anfragemethode (GET, POST, DELETE ...), sowie ein Media-Type Feld im Request-Header und ein Cache-Control Felder im Response-Header realisiert.
Weiterhin muss die Information darüber, wann das Ende einer Nachricht erreicht ist, in der Nachricht selbst enthalten sein und darf nicht durch die darunterliegende Transportschicht signalisiert werden.
Bei HTTP geschieht das etwa durch die Content-Length Felder oder den Chunked Encoding Mechanismus, aus denen sich ablesen jeweils ablesen lässt, wie viele Bytes noch gelesen werden müssen.[2]

[2]Im Falle der Content-Length wird die Länge der gesendeten Nachricht in Bytes einmalig im Header der Response angegeben, bei Chunked Encoding wird das Ende der Nachricht durch ein Chunk der Länge null signalisiert

Hypermedia as the Engine of the State

Die Einschränkung mit dem unhandlichen Namen *hypermedia as the engine of the application state* (HATEOAS) soll ein Höchstmaß an Entkopplung zwischen Client und Server sicherstellen.
Der Nutzer eines REST-Services soll - abgesehen von der Addresse eines Einstiegspunkts - kein Vorwissen über die interne Struktur des Services benötigen, um sich orientieren zu können.
HATEOAS garantiert, dass vom Moment der Ansteuerung des Einstiegspunkts eines Webservices an, alle serverseitigen Zustandsübergänge dadurch gesteuert werden können müssen, dass der Nutzer zwischen serverseitig generierten Möglichkeiten auswählt (Fielding, 2008).
Konkret bedeutet das, dass jeder erreichbare Endpunkt eines Webservices über eine Abfolge von Links ansteuerbar sein muss, so dass der Nutzer des Service außer der Adresse des Startpunktes keine weitere Information über die Speicherungs- oder Adressierungsstruktur des Dienstes besitzen muss.
Durch diese Verlinkung der Endpunkte untereinander lässt sich die gesamte API als endlicher Zustandsautomat darstellen. Dabei entspricht jeder Endpunkt einem Zustand, während die HTTP Methoden als Kanten zwischen den Zuständen modelliert werden (vgl. Sobocinski, 2014).
Ob HATEOAS in der von Fielding geforderten Rigorosität für das Design von APIs geeignet ist, ist Gegenstand leidenschaftlicher Diskussionen.[3]
Für Fielding selbst ist die Einhaltung der HATEOAS Einschränkung in jedem Fall unverzichtbar. Das stellt er acht Jahre nach der Veröffentlichung seiner Dissertation noch einmal unmissverständlich klar:
"If the engine of application state (and hence the API) is not being driven by hypertext" , schreibt er dort, *"then it cannot be RESTful and cannot be a REST API"* (Fielding, 2008).

2.3.7 Zusammenfassung der Constraints

Eine Softwarearchitektur ist genau dann "RESTful", wenn sie alle oben beschriebenen Constraints erfüllt. Tabelle 2.1 fasst die Menge der REST-Constraints, jeweils mit einer kurzen Beschreibung, noch einmal zusammen.

[3](Reiser, 2018) plädiert beispielsweise dafür, HATEOAS nicht umzusetzen. Ein Vertreter der Gegenseite ist etwa (Korando, 2016)

#	Constraint	Beschreibung
1	Client-Server	Trennung der Zuständigkeit von Client und Server
2	Statelessness	Keine Speicherung von Informationen über Requests
3	Cache	Zwischenspeicherbarkeit explizit oder implizit markieren
4	Layered System	Schichten dürfen nur Kenntnis über ihre Nachbarn haben
5	Code on Demand	Optionaler Download von ausführbarem Code
6	Uniform Interface	
6.1	Identification of Ressources	Alle Referenzen beziehen sich auf "Resource Identifier"
6.2	Representations	Nur Repräsentationen dürfen manipuliert werden
6.3	Self Descriptive	Nachrichten müssen alle zu ihrem Verständnis nötigen Kontextinformationen enthalten
6.4	HATEOAS	Erreichbarkeit aller Endpunkte über Hyperlinks

Tabelle 2.1: Menge der REST-Constraints.

2.4 REST im Vergleich zu anderen Paradigmen

REST ist natürlich nicht das einzige Paradigma, an das man sich beim Design einer API halten kann.
Um zu legitimieren, warum für den Implementierung der API des Open-Data-Service gerade dieser Architekturstil gewählt wurde, werden im Folgenden zwei der prominentesten Alternativen kurz dargestellt und ihre Eignung für den ODS diskutiert.

2.4.1 Remote Procedure Call (RPC)

Bevor REST zum de-facto Standard des API Designs wurde, war das vorherrschende Paradigma der *remote procedure call* (RPC), realisiert vor allem durch das *Simple Object Access Protocol* (SOAP).
Im Folgenden werde ich, basierend auf einer sehr lesenswerten Gegenüberstellung von RPC und REST von Phil Sturgeon (Sturgeon, 2016), die Funktionsweise von RPC in ihren Grundzügen erklären und mit dem REST-Ansatz vergleichen.
REST basiert, wie oben beschrieben, im Wesentlichen auf dem Konzept abstrakter Ressourcen und der Manipulation ihrer Repräsentationen, ist somit *ressourcenorientiert*.
RPC orientiert sich hingegen an den Operationen, die auf dem Server ausgeführt werden sollen, was sich ja bereits am Namen "remote *procedure* call" ablesen lässt.
Damit geht einher, dass die Namen der Endpunkte einer RPC-API üblicherweise

nicht, wie bei REST, aus Substantiven, sondern aus Verben bestehen – sie bezeichnen keine Ressourcen, sondern Operationen.
Während bei RESTful APIs zumeist die Gesamtheit der zulässigen HTTP Verben verwendet werden um zu spezifizieren, welche Operation auf den Daten ausgeführt werden sollen, beschränkt sich RPC auf die Verwendung von GET und POST, wobei GET verwendet wird, um Informationen zu beziehen, und POST für alles andere.
Zwar ist REST das jüngere und weitaus verbreitetere Paradigma, doch es ist deswegen nicht grundsätzlich das "bessere" – vielmehr hängt die Eignung davon ab, zu welchem Zweck es eingesetzt werden soll.
REST kann seine Stärken am besten dann ausspielen, wenn wenige, einfache Operationen auf vielen Daten ausgeführt werden sollen. Für diese Operationen hat sich der Begriff CRUD (Create, Read, Update, Delete) eingebürgert.
Die Abstraktion von den Operationen hin zu den Ressourcen führt dann zu intuitiv verständlichen und eleganten Interfaces. Die Vorteile von REST für diese Anwendungsfälle lassen sich am besten an einem Beispiel verdeutlichen:[4]
Sucht man mit Google nach "SOAP example" bekommt man ein als Ergebnis[5] ein Beispiel von Google, das eine Methode namens *getAdUnitsByStatement* vorstellt. Die Anfrage ist nach den Anforderungen des SOAP Protokolls verfasst und sieht so aus:

```
<?xml version="1.0" encoding="UTF-8"?>
<soapenv:Envelope
       xmlns:soapenv="http://schemas.xmlsoap.org/soap/envelope/"
       xmlns:xsd="http://www.w3.org/2001/XMLSchema"
       xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance">
  <soapenv:Header>
    <ns1:RequestHeader
         soapenv:actor="http://schemas.xmlsoap.org/soap/actor/next"
         soapenv:mustUnderstand="0"
         xmlns:ns1="https://www.google.com/apis/ads/publisher/v201605">
      <ns1:networkCode>123456</ns1:networkCode>
      <ns1:applicationName>DfpApi-Java-2.1.0-dfp_test</ns1:applicationName>
    </ns1:RequestHeader>
  </soapenv:Header>
  <soapenv:Body>
    <getAdUnitsByStatement
        xmlns="https://www.google.com/apis/ads/publisher/v201605">
      <filterStatement>
        <query>WHERE parentId IS NULL LIMIT 500</query>
```

[4]Das Beispiel ist, ebenso wie alle weiteren in diesem Kapitel verwendeten Beispiele, von (Sturgeon, 2016) übernommen

[5]https://developers.google.com/ad-manager/api/reference/v201802/InventoryService, zuletzt abgerufen am: 09.09.2018, 20:25

```
      </filterStatement>
    </getAdUnitsByStatement>
  </soapenv:Body>
</soapenv:Envelope>
```

Der enorme Umfang der Anfrage hängt mit der Rigorosität der SOAP Spezifikation zusammen, auf die an dieser Stelle nicht detaillierter eingegangen werden soll.[6]
Doch selbst wenn man sich eine vereinfachte API, die nicht der SOAP Spezifikation folgt, vorstellt, erhält man eine Anfrage, die etwa folgendermaßen aussieht:

```
POST /getAdUnitsByStatement HTTP/1.1
HOST: api.example.com
Content-Type: application/json

{"filter": "WHERE parentId IS NULL LIMIT 500"}
```

Zwar lässt sich der Umfang der Anfrage so enorm reduzieren, doch dennoch brauchen wir für jede auch nur minimal anders geartete Operation *getAdUnitsBySomething* einen eigenen neuen Endpunkt. Der Nutzer der API braucht also ein ausgeprägtes Vorwissen über die möglichen Operationen und die damit verbundenen Endpunkte.
Die REST-Variante dieser Anfragen wirkt sehr viel schlanker, einfacher und eleganter: Mit Anfragen wie `GET /ads?statement={foo}`, `GET /ads?something={bar}` oder `GET /ads?statement={foo}&limit={500}` lassen sich beliebige Operationen auf dem Endpunkt intuitiv anfordern und einfach kombinieren.
Dass es dennoch Situationen gibt, in denen RPC möglicherweise die bessere Wahl für das API-Design ist, zeigt sich in Anwendungsfällen, in denen viele komplexe (nicht-CRUD) Operationen auf wenigen Daten ausgeführt werden. Sturgeon nennt als Beispiel für einen solchen Anwendungsfall das soziale Netzwerk "Slack": Slack bietet für einzelne User unter anderem die Operationen "kick", "ban" und "leave" an – jeweils sowohl in Bezug auf einen einzelnen Channel, als auch auf das gesamte Slack-Team.
Wie würde man diese Operationen in einer REST API modellieren?
Der naheliegende Ansatz, die HTTP Methode `DELETE` zu verwenden, ist dafür zu unspezifisch. Ein `DELETE users/xyz` kann vom Server als Aufforderung verstanden werden, den Account des Users aus der Datenbank zu löschen (kick oder leave) oder ihn zu sperren (ban). Zusätzlich ist unklar, wie sich damit etwa ein Kick aus einem einzelnen Channel modellieren ließe.
Ein anderer Ansatz wäre die Verwendung des HTTP Verbs `PATCH`:

```
PATCH /users/xyz HTTP/1.1
```

[6] Eine genauere Darstellung findet sich etwa bei (Snell, Tidwell & Kulchenko, 2001)

```
Host: api.example.com
Content-Type: application/json

{"status": "kicked", "kick_channel": "catgifs"}
```

Auf diese Weise ließen sich die Operationen und ihr Anwendungsbereich zwar unterscheiden, aber es müssten eigene Attribute für User eingeführt werden, deren einziger Zweck die Modellierung der Kick-Funktionalität ist.
Weitere Möglichkeiten sieht Sturgeon in der Manipulation von (als Attribute einer Ressource dargestellten) Beziehungen oder globalen "kick", "ban" und "leave" Collections, denen mit `POST`-Requests User hinzugefügt werden können.
Doch all diese Modellierungen scheinen eine Anwendung in ein Paradigma zu zwingen, das eigentlich nicht dazu passt.
Elegant und einfach erscheint hingegen die Modellierung über RPC, die bei Slack folgendermaßen aussieht:

```
POST /api/channels.kick HTTP/1.1
Host: slack.com
Content-Type: application/json

{
  "token": "xxxx-xxxxxxxxx-xxxx",
  "channel": "C1234567890",
  "user": "U1234567890"
}
```

Nun muss man Sturgeon in seiner Ablehnung der Modellierung der Operationen über Beziehungen nicht zwingend folgen. Beispielsweise bietet die Spezifikation "JSON API", die für die Implementierung der ODS API verwendet wird und in Abschnitt 5.1.2 detaillierter vorgestellt wird, elegante Möglichkeiten, solche Beziehungen abzubilden.
Zugestanden sei ihm aber, dass RPC für diesen Anwendungsfall tatsächlich mindestens ebenso gut geeignet erscheint wie REST.
Gleichzeitig bleibt ein grundlegender Nachteil aber bestehen: Der Grad an Entkopplung zwischen Server und Client, der mit REST erreicht wird, lässt sich mit RPC APIs nicht erreichen. Nutzer der API müssen die möglichen Operationen und zugehörigen Endpunkte kennen, während bei einer konsequenten Umsetzung der REST Prinzipien kein Vorwissen außer einem initialen Eingangspunkt nötig ist.
Für den Open-Data-Service ist REST in jedem Fall die bessere Wahl, da die Operationen, die auf den Daten ausgeführt werden sollen, sich im Wesentlichen auf CRUD beschränken.

2.4.2 Graph Query Language (GraphQL)

GraphQL ist eine Abfragesprache, die 2012 von Facebook entwickelt und zunächst intern verwendet wurde, um die Anforderungen von Datenmodellen für Client-Server Anwendungen zu beschreiben. 2015 wurde es von Facebook veröffentlicht und wird seitdem als offener Standard weiterentwickelt (vgl. "GraphQL Specification", 2018).
Im Folgenden werde ich die Funktionsweise von GraphQL kurz anhand eines Beispiels in ihren Grundzügen erläutern, wobei ich mich an einer Einführung von (Buna, 2016) orientieren werde.
Eine GraphQL Anfrage besteht aus einem einfachen String, der über einen GraphQL Endpunkt wie etwa `/graphql?query={...}` an den Server gesendet wird. Angenommen die JSON Repräsentation einer Klasse Person sieht folgendermaßen aus:

```
{
  "data": {
    "person": {
      "name": "Darth Vader",
      "birthYear": "41.9BBY",
      "planet": {
        "name": "Tatooine"
      },
      "films": [
        { "title": "A New Hope" },
        { "title": "The Empire Strikes Back" },
        { "title": "Return of the Jedi" },
        { "title": "Revenge of the Sith" }
      ]
    }
  }
}
```

Wie leicht zu erkennen ist, bietet der Server einen unverzichtbaren Star-Wars Informationsdienst an.
Interessiert man sich nun für einen den Namen eines bestimmten Star-Wars Protagonisten und die Titel der Filme, in denen dieser aufgetaucht ist, müsste man folgende Anfrage an den Server stellen:

```
{
  person(ID: ‘‘42’’) {
    name
    films {
      title
```

```
    }
  }
}
```

Die entsprechende Antwort würde so aussehen:

```
{
  "data": {
    "person": {
      "name": "Darth Vader",
      "films": [
        { "title": "A New Hope" },
        { "title": "The Empire Strikes Back" },
        { "title": "Return of the Jedi" },
        { "title": "Revenge of the Sith" }
      ]
    }
  }
}
```

Auffallend ist die strukturelle Ähnlichkeit zwischen Anfrage und Antwort: Betrachtet man die JSON-Repräsentation von Objekten als Key-Value Paare, entspricht die Anfrage der Antwort ohne den Value Anteil. Buna bezeichnet dieses Verhältnis als Frage-Antwort-Relation – "the question is the answer statement without the answer part" (Buna, 2016).
Mit einer einfachen Analogie zur natürlichen Sprache verdeutlicht er diesen Zusammenhang: Angenommen man erwartet eine Antwort der Form

"Der Planet, der der Sonne am nächsten ist, ist Merkur".

Dann kann man die zugehörige Frage formulieren, indem man den selben Satz verwendet und den Teil, der die Antwort enthält, weglässt.

"(Was ist) der Planet, der der Sonne am nächsten ist".

Analog zu dieser intuitiv verständlichen Form der Fragebildung funktionieren GraphQL Anfragen: "Take a JSON response, remove all the *answer* parts (which are the values), and you'll get a GraphQL query very suitable to represent a question for that JSON response" (Buna, 2016).
Allein – so intuitiv diese Funktionsweise zunächst erscheint – dem aufmerksamen Leser wird nicht entgangen sein, dass die Anfrage nur dann formuliert werden kann, wenn dem Client die ID des angefragten Protagonisten bekannt ist.
Hier zeigt sich im Vergleich zu REST ein großer Nachteil des GraphQL Ansatzes: Das Maß an Entkopplung, das REST bietet, wird durch GraphQL nicht erreicht. Um eine GraphQL API verwenden zu können, ist einiges an Vorwissen nötig.

Zwar lassen sich für die Auswahl des Abfrageobjektes auch global zugängliche Aliase, wie etwa “me” zur Bezeichnung des aktuell eingeloggten Users, verwenden, doch auch diese müssen dem Nutzer der API vor dem Zugriff bekannt sein. Gleichzeitig bietet GraphQL eine große Flexibilität in der Formulierung von Anfragen. Einerseits können Datenabfragen verschiedene Argumente übergeben werden, andererseits sind auch Manipulationen der Daten sowie Subscriptions zur kontinuierlichen Benachrichtigung über sich verändernde Daten möglich.
Diese Möglichkeiten hier detaillierter zu besprechen würde den Rahmen der vorliegenden Arbeit sprengen.[7]
Aus dem Bisherigen sollte aber bereits deutlich geworden sein, welchen großen Vorteil GraphQL bietet: In jeder Anfrage wird genau spezifiziert, welche Attribute eines Objekts benötigt werden. Diese – und *nur* diese – sind in der Antwort enthalten. Das Ausmaß an unnötig versendeten Daten lässt sich damit fast auf null reduzieren.
Der Effizienzunterschied wird besonders deutlich, wenn die Daten stark vernetzt sind. Bei REST sind in diesem Fall üblicherweise mehrere Anfragen nötig, um die Referenzen auf in Beziehung stehende Objekte aufzulösen, während sie bei GraphQL durch eine einfache Modifikation der Anfrage integriert werden können (vgl. die Integration der Filmtitel im Beispiel oben). [8] (Buna, 2016) gibt ein Beispiel, in dem eine einzelne GraphQL Anfrage sechs REST Anfragen ersetzt.
So überzeugend die Vorteile von GraphQL auch zunächst erscheinen mögen, gibt es doch drei gewichtige Gründe, warum eine REST API für den ODS besser geeignet ist:

1. Um sinnvolle Anfragen zu formulieren, muss die Struktur der Daten bekannt sein (welche Attribute existieren?, welche Namen haben diese?, welche ID?). Beim ODS mit seinen generischen Daten ist das im Allgemeinen nicht gegeben.

2. Die Daten des ODS weisen in der Regel keine besondere Vernetzung untereinander auf. Den Vorteilen von GraphQL kommt daher weniger Gewicht zu.

3. REST APIs sind, zumindest stand jetzt, sehr viel etablierter als GraphQL APIs. Die Exposition der API über einen einzelnen Endpunkt ist für Nutzer, die keine oder wenig GraphQL Kenntnisse besitzen sehr viel unübersichtlicher und weniger intuitiv.

Unter Berücksichtigung all dieser Gesichtspunkte scheint eine REST API für den ODS schlussendlich die bessere Wahl zu sein. Es kann aber ein lohnendes Projekt

[7] Dafür sei hier auf die ausführliche und gut lesbare Spezifikation verwiesen (“GraphQL Specification”, 2018).

[8] Hier ist allerdings anzumerken, dass die für den ODS verwendete JSON Spezifikation `JSON API` die Möglichkeit bietet, solche Relationspartner direkt zu integrieren.

für spätere Arbeiten sein, zusätzlich zur bestehenden REST API einen GraphQL Endpunkt zu integrieren.

3 Anforderungen

3.1 Funktionale Anforderungen

3.1.1 Erfüllung der REST Constraints

Das grundlegende Ziel der vorliegenden Arbeit war es, eine RESTful API für den ODS zu implementieren. Die neu implementierte API soll den von Fielding aufgestellten REST Constraints gerecht werden.
Evaluationsschema: Die Anforderung gilt als erfüllt, wenn alle in Tabelle 2.1 aufgelisteten Constraints eingehalten werden.

3.1.2 Versionierung

Um die Abwärtskompatibilität zuzusichern und die Funktionalität bereits bestehender ODS Nutzer wie etwa der Pegelalarm App (vgl. Referenz) nicht zu gefährden, soll neben der aktuellen API Version auch die ursprüngliche API weiterhin verwendbar bleiben. Zu diesem Zweck soll ein Versionierungskonzept verwendet werden.
Evaluationsschema: Die Anforderung gilt als erfüllt, wenn der Zugriff auf die ursprüngliche API weiterhin unverändert möglich ist.

3.1.3 Paginierung

Da die Datenquellen des ODS potentiell sehr große Datensätze liefern, sollen Nutzer die Möglichkeit haben, die Größe und Position des Datensatzes in der Anfrage zu spezifizieren. Zu diesem Zweck soll die API eine geeignete Paginierungslösung implementieren.
Evaluationsschema: Die Anforderung gilt als erfüllt, wenn mit jeder Datenanfrage die Größe und Position des Datensatzes spezifiziert werden können sowie

jede Antwort, die Daten enthält, gleichzeitig Informationen liefert, wie der nächste Datensatz angefordert werden kann.[1]

3.1.4 Dokumentation

Die neu implementierte API soll eine ausführliche Dokumentation all ihrer Endpunkte, der darauf anwendbaren HTTP-Methoden sowie der erwarteten Datenformate besitzen. Zur Erstellung dieser Dokumentation soll das Anwendungspaket Swagger verwendet werden.
Evaluationsschema: Die Anforderung gilt als erfüllt, wenn eine Dokumentation vorliegt, die alle Endpunkte, alle auf ihnen aufrufbaren Methoden inklusive der dafür nötigen Authentifikation, sowie alle gesendeten und empfangenen Datenschemata beschreibt.

3.1.5 Verwendung von Tests

Für die Erstellung qualitativ hochwertiger Software ist deren gründliche Testung bereits zur Entwicklungszeit unverzichtbar.
Die Implementierung der API soll der Bedeutung von Tests für die Softwarequalität Rechnung tragen und über eine möglichst hohe Testabdeckung verfügen.
Evaluationsschema: Die Anforderung gilt als erfüllt, wenn die neu implementierte API über eine Unit Test abdeckung von wenigstens 85% der Methoden und Zeilen verfügt und zusätzlich durch Integrationstests überprüft werden kann.

3.2 Nichtfunktionale Anforderungen

3.2.1 State of the Art Implementierung

Die neue Implementierung der ODS API soll aktuellste Technologien verwenden und aktuellen best practices folgen.

[1] Als Datenanfrage sind hier nur Anfragen nach den von den Datenquellen bereitgestellten Nutzdaten zu verstehen. Alle anderen Anfragen, zum Beispiel nach den Datenquellen, darauf definierten Views etc., sind zum jetzigen Zeitpunkt noch nicht so umfangreich, dass Paginierung notwendig wäre.

3.2.2 Minimalinvasive Implementierung

Um die Funktionalität bestehender Nutzung nicht zu gefährden (3.1.2), soll das Ausmaß der Eingriffe in bestehenden Quellcode so klein wie möglich gehalten werden. Optimalerweise sollte die API allein durch Erweiterung und ohne Manipulation der bestehenden Codebasis erstellt werden.

4 Architektur des Open-Data-Service

Die im Rahmen der vorliegenden Arbeit vorgenommene Implementierung einer RESTful API für den Open-Data-Service ist nicht aus dem Nichts enstanden. Vielmehr konnte auf die Grundlage einer gut funktionierenden Architektur und einer bereits bestehenden REST-like API aufgebaut werden.
In diesem Kapitel sollen zunächst die bestehende Architektur, sowie die bereits existierende API vorgestellt werden. Dabei werde ich immer wieder auf die Masterarbeit von Konstantin Tsysin (Tsysin, 2014) zurückgreifen, der 2014 den Open-Data-Service prototypisch implementiert und in diesem Zuge das zugrunde liegende Design beschrieben hat.

4.1 Bestehende Architektur

Der Open-Data-Service besteht im Wesentlichen aus drei Komponenten:

1. Einer **Importkomponente**, deren Aufgabe darin besteht, Daten aus verschiedenen externen Datenquellen anzubinden und der Datenbankkomponente zur Verfügung zu stellen.
2. Einer **Datenbankkomponente**, die die von der Importkomponente bereitgestellten Daten persistiert und über eine Schnittstelle der Serverkomponente zugänglich macht.
3. Einer **Serverkomponente**, die sich um die Bereitstellung der Daten über eine öffentliche API kümmert.

Das Zusammenspiel der drei Hauptkomponenten verdeutlicht Abbildung 4.1. Anzumerken ist allerdings, dass die strikte Trennung der Komponenten, die diese Abbildung suggeriert, aktuell nicht mehr ganz durchgehalten wird.
Beispielsweise besteht in der aktuellen Version des ODS inzwischen die Möglichkeit über die API direkt Datenquellen und zugehörige Adapter (vgl. dazu unten) zum ODS hinzuzufügen oder zu entfernen.

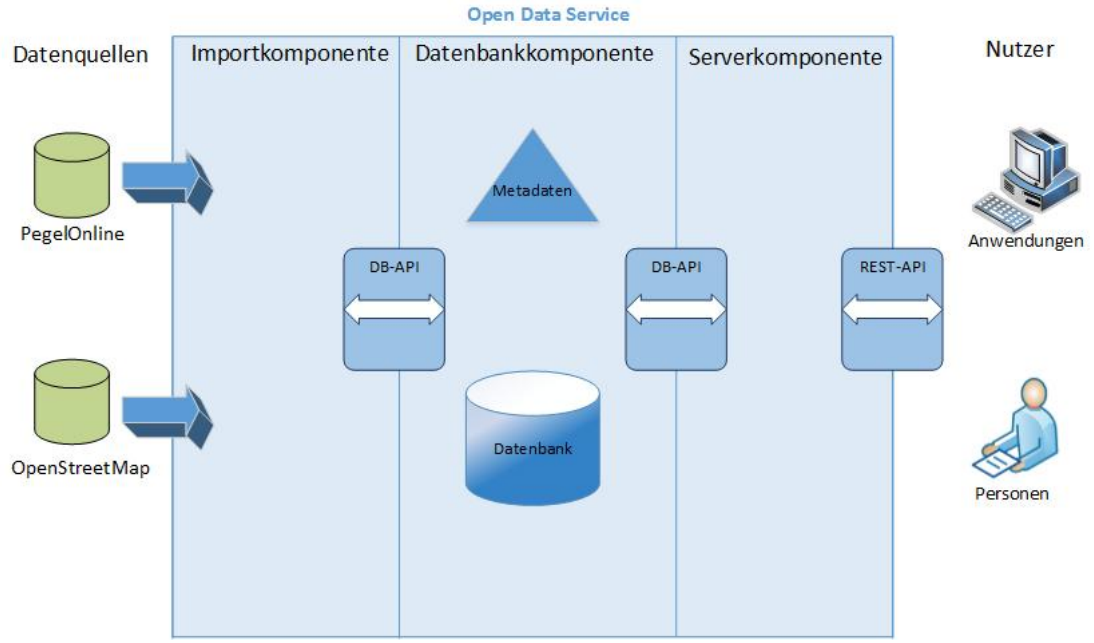


Abbildung 4.1: Hauptkomponenten des Open-Data-Service (Tsysin, 2014)

Abbildung 4.2 verdeutlicht, dass das Zusammenspiel der Komponenten in der aktuellen Version des Open-Data-Service inzwischen sehr viel komplexer geworden ist. Im Folgenden sollen die einzelnen Komponenten des ODS noch einmal genauer betrachtet werden

Importkomponente

Abbildung 4.3 zeigt die Funktionsweise der Importkomponente des ODS. Basierend auf dem "Pipes and Filter"-Architekturmuster[1] werden die eingehenden Daten durch mehrere Filterschritte weiterverarbeitet, bereinigt und gefiltert, bevor sie der Datenbankkomponente zur Verfügung gestellt werden (vgl. Tsysin, 2014, S. 18f.). Im Fall des ODS werden die von den Datenquellen stammenden Daten zunächst mit einem Adapter in eine für den ODS zugängliche Form gebracht, bevor sie durch eine beliebige Anzahl von Filtern (realisiert durch die Klasse FilterChain) weiterverarbeitet und bereinigt werden können.
Dabei können neue Datenquellen ebenso wie Adapter und Filter zur Laufzeit über die API hinzugefügt werden. Templates müssen allerdings hart codiert vorhanden sein (wie etwa im Beispiel der Abbildung 4.3 ein REST-Adapter oder in der realen Anwendung die Klasse JsonSourceAdapter).

[1] Eine detailliertere Erklärung dieses Architekturmusters findet sich etwa bei(Garlan & Shaw, 2014)

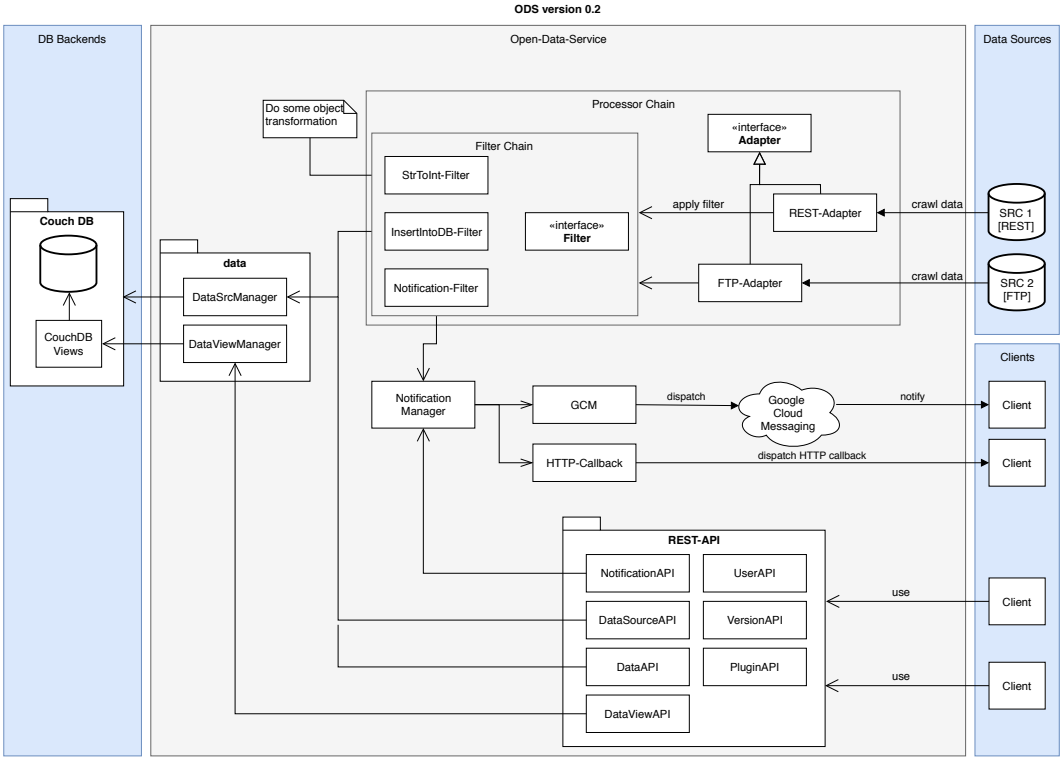


Abbildung 4.2: Architektur des Open-Data-Service

Damit eine Datenquelle Daten liefern kann, muss mindestens ein Adapter und ein InsertIntoDB-Filter hinzugefügt werden.

DB-Persistierung

Die von den Datenquellen bezogenen Daten sollen durch die Datenbankkomponente des ODS persistiert werden. Dadurch kann einerseits auch bei Datenquellen, deren APIs häufiger ausfallen, zuverlässig garantiert werden, dass Daten zur Verfügung gestellt werden. Andererseits können – sofern die Datenquellen diese Möglichkeit nicht anbieten – auch historische Daten abgespeichert werden.
Als Persistierungslösung wurde kein klassisches relationales Datenbankmanagementsystem (RDBMS), sondern eine nichtrelationale NoSQL-Datenbank gewählt, da RDBMS im Gegensatz zu NoSQL auf ein festes Schema der Daten angewiesen ist (vgl. Tsysin, 2014, S. 21).[2] Über ein solches Datenschema besteht beim ODS aber im Allgemeinen kein Vorwissen, da die bereitgestellten Daten aufgrund der potentiell unbeschränkten Anzahl an Datenquellen extrem generisch sind.

[2]Eine einführende Erklärung beider Ansätze inklusive deren Gegenüberstellung findet sich etwa bei (Sadalage, 2013)

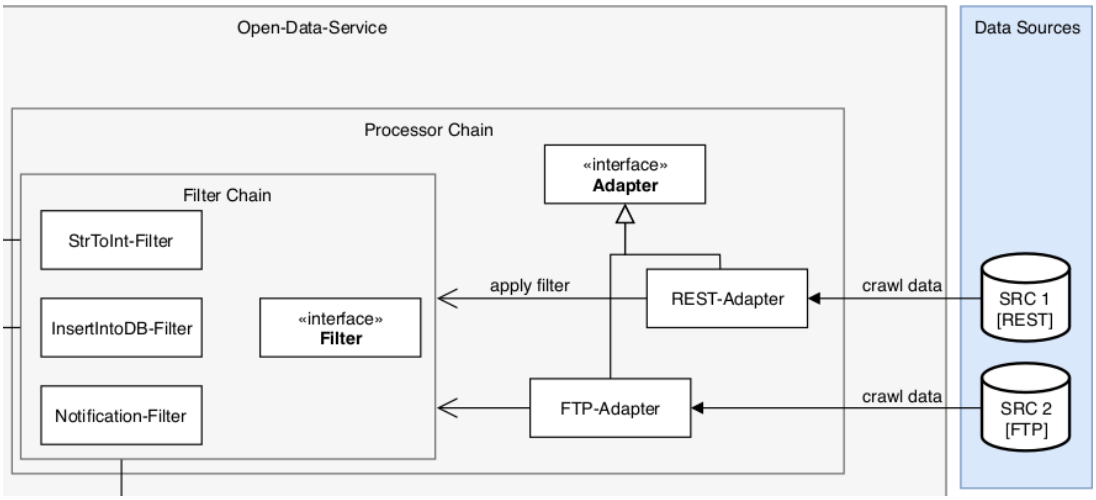


Abbildung 4.3: Architektur der Importkomponente des Open-Data-Service abbrufbar unter https://drive.google.com/drive/folders/1eqm6wXhCGOf4UhHXckxNp6bfUjyi82h0

Aktuell verwendet der ODS "Apache CouchDB" als Backend,[3] wobei die Anbindung der Datenbank so konzipiert ist, dass ein Austausch der Datenbank einfach und flexibel möglich ist.
In Abbildung 4.4 ist gut erkennbar, dass auf Seiten des ODS die wesentlichen an der Persistierung beteiligten Klassen der DataSourceManager sowie der DataViewManager sind. Diese beiden Klassen stellen die Schnittstelle zwischen der Datenbank und der Datenimport- bzw. Serverkomponente des ODS dar. Um die Datenbank des ODS auszutauschen, müssen also grundsätzlich nur diese beiden Klassen angepasst werden.[4]

Serverkomponente

Die Aufgabe der Serverkomponente ist einerseits die technische Bereitstellung der grundsätzlichen Serverfunktionalität. Dieser Anteil wird größtenteils von den beteiligten Frameworks übernommen (siehe 5.1) und wird hier daher nicht näher beschrieben.
Andererseits ist hier die REST-API verortet, über die Nutzer des ODS unter anderem Daten abfragen, Datenquellen, Filter und Adapter verwalten können. Eine graphische Übersicht über die bestehenden (Teil-)APIs findet sich in Abbildung 4.5.

[3] Zur Begründung dieser Wahl vgl. (Tsysin, 2014, S. 22).

[4] Tatsächlich greift allerdings auch die DataAPI direkt auf an CouchDB gekoppelte Klassen zu, ohne dem Umweg über einen der beiden Manager zu gehen – was ein Ansatzpunkt für späteres Refactoring sein könnte.

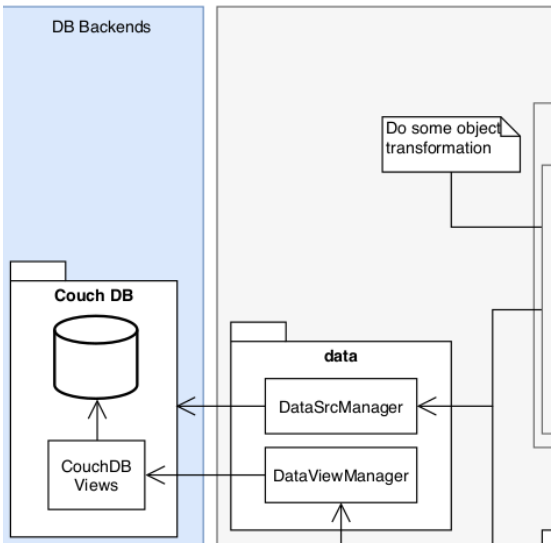


Abbildung 4.4: Architektur der Datenbankkomponente des Open-Data-Service

4.2 Bestehende API

Über die bestehende API (im Folgenden auch als API v1 bezeichnet) können Nutzer des ODS Daten von verschiedenen Datenquellen beziehen sowie Datenquellen, Filter und Views anzeigen, hinzufügen und entfernen. Weiterhin bietet die API die Möglichkeit, sich für kontinuierliche Benachrichtigungen über sich ändernde Daten zu registrieren oder Nutzer zu verwalten. Eine Übersicht über die auf den bestehenden Endpunkten ausführbahren Operationen geben Tabellen 4.1 und 4.2.
Die bestehende API ist als REST-API konzipiert worden(vgl. Tsysin, 2014, S. 26) und erfüllt damit tatsächlich fast alle der in Abschnitt 2.3 vorgestellten Constraints – sie lässt sich insofern als *REST-like* bezeichnen.
Leider wird sie in einem wesentlichen Aspekt den Anforderungen an einer RESTful API nicht gerecht: Sie ist nicht Hypertext-orientiert. Da es keine Möglichkeit gibt, sich durch Links innerhalb der API zu orientieren, muss bereits vor Verwendung der API deren innere Struktur bekannt sein – der HATEOAS-Constraint (vgl. 2.3.6) ist nicht erfüllt.
Um das zu verdeutlichen, zeigt Abbildung 4.6 beispielhaft eine Antwort auf die Anfrage "`GET /ods/api/v1/datasources`".
Es ist leicht erkennbar, dass einfach die Collection der angefragten Objekte zurück geliefert wird, ohne dass eine Verlinkung, etwa auf die einzelnen Elemente der Collection oder zugehörige Filterchains, in der Antwort enthalten wäre.
Das Folgende Kapitel wird sich dementsprechend mit dem Design und der Implementierung einer API beschäftigen, die *allen* REST-Constraints gerecht wird

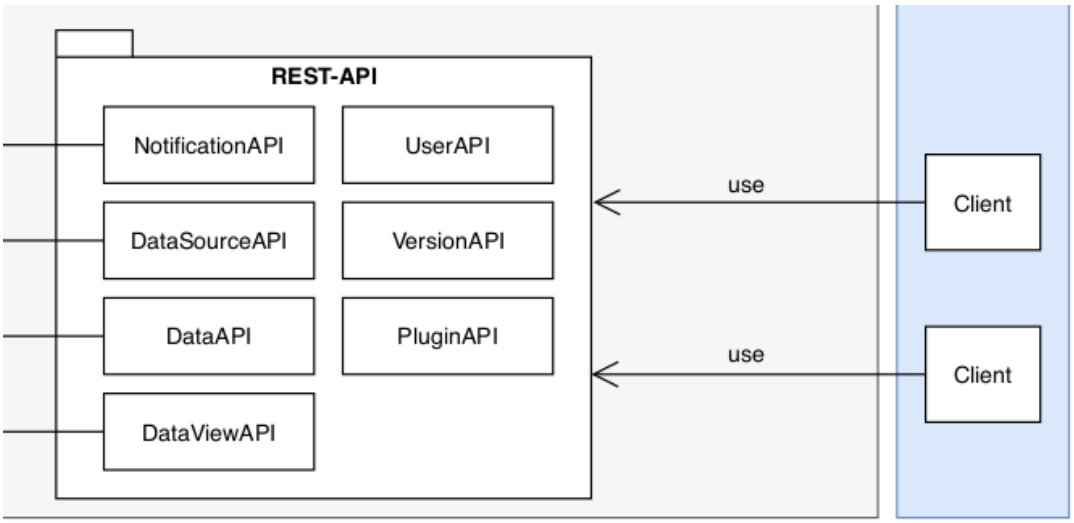


Abbildung 4.5: Architektur der Serverkomponente des Open-Data-Service

und somit mit Fug und Recht als *RESTful* bezeichnet werden kann.

```
[
    {
        "id": "osm",
        "domainIdKey": "/id",
        "schema": {},
        "metaData": {
            "name": "OSM",
            "title": "OSM",
            "author": "",
            "authorEmail": "",
            "notes": "",
            "url": "",
            "termsOfUse": ""
        }
    },
    {
        "id": "camper",
        "domainIdKey": "/id",
        "schema": {},
        "metaData": {
            "name": "Trashcans",
            "title": "Trashcans",
            "author": "",
            "authorEmail": "",
            "notes": "",
            "url": "",
            "termsOfUse": ""
        }
    },
]
```


Abbildung 4.6: Beispielantwort der ODS API v1

Data	Base-URL: /datasources/{sourceId}	
Verb	URL	Description
GET	/data	Get data objects from {sourceId}
GET	/pointer:/.*	Get certain attributes of a data object
DELETE	/data	Delete all data objects from {sourceId}
DataSources	Base-URL: /datasources/	
Verb	URL	Description
GET	/	Get all data sources
GET	/{id}	Get a data source by its {id}
GET	/{id}/schema	Get the data schema for a data source
PUT	/{id}	Add a data source with a specified {id}
DELETE	/{id}	Delete a datasource
DataView	Base-URL: /datasources/sourceId/views	
Verb	URL	Description
GET	/	Get all data views
GET	/{id}	Get a data view by its {id}
PUT	/{id}	Add a data view with a specified {id}
DELETE	/{id}	Delete a datasource
Notifications	Base-URL: /datasources/sourceId/notification	
Verb	URL	Description
GET	/	Get all clients registered for notifications
GET	/{id}	Get a client by its {id}
PUT	/{id}	Register a client with a specified {id}
DELETE	/{id}	Unregister a client
Plugins	Base-URL: /datasources/sourceId/plugins	
Verb	URL	Description
GET	/	Get all plugins
GET	/{id}	Get a plugin by its {id}
PUT	/{id}	Add a plugin with a specified {id}
DELETE	/{id}	Delete a plugin
ProcessorChain	Base-URL: /datasources/sourceId/filterChains	
Verb	URL	Description
GET	/	Get all processor chains
GET	/{id}	Get a processor chain by its {id}
PUT	/{id}	Add a processor chain with a specified {id}
DELETE	/{id}	Delete a processor chain

Tabelle 4.1: Liste der V1 Endpunkte I

ProcessorSpecification	Base-URL: /filterTypes	
Verb	URL	Description
GET	/	Get all specifications
Users	Base-URL: /users	
Verb	URL	Description
GET	/	Get all users
GET	/{id}	Get a user by its {id}
GET	/me	Get currently logged in user
PUT	/{id}	Create new user with a specified {id}
DELETE	/{id}	Delete a user
Version	Base-URL: /version	
Verb	URL	Description
GET	/	Get version information

Tabelle 4.2: Liste der V1 Endpunkte II

5 Design und Implementierung einer RESTful API

Ausgehend von der in Kapitel 3 spezifizierten Anforderungen, sowie der in Kapitel 4 vorgestellten bestehenden Konfiguration des Open-Data-Service, soll in diesem Kapitel die Konzeption und Implementierung der neuen RESTful API vorgestellt werden. Dazu werde ich zunächst eine Übersicht über die dafür verwendete Technologie – sowohl die bereits bestehende, als auch die für die neue API verwendete – voranstellen.

5.1 Technologie

5.1.1 Bereits verwendete Technologien

Docker

Docker[1] ist ein Programm, das es ermöglicht Software in isolierten virtuelle Umgebungen, so genannten Containern, zu betreiben.
Container kann man sich vereinfacht als sehr leichtgewichtige virtuelle Maschinen vorstellen: Sie ermöglichen einen ähnlichen Grad an Isolierung wie eine "echte" virtuelle Maschine, lassen aber eine flexiblere Zuteilung der Ressourcen zu, da unmittelbarer auf die Hardware des Host Systems zugegriffen werden kann (vgl. Negus, 2015).
Ein großer Vorteil bei der Verwendung von Docker ist, dass sich bei einer Software, die in einem Docker Container läuft, die gesamte Ausführungsumgebung kontrollieren lässt. Damit lässt sich eine weitestgehend systemunabhängige Softwareauslieferung erreichen, wodurch mögliche System- oder Hardwareabhängige Fehler minimiert werden können.

[1] https://www.docker.com/, eine Einführung in die Anwendung von Docker findet sich etwa bei (Khare, 2015).

Zusätzlich existiert eine Software Registry namens DockerHub, bei der Software zentral registriert werden und dann von jedem Ort heruntergeladen werden kann.
Der ODS verwendet Docker, um kontrollierte Ausführungsumgebungen für die Datenbank CouchDB und eine Instanz des ODS zu schaffen.
Diese kontrollierte Umgebung erleichtert einerseits die Verteilung des ODS, sowie andererseits das automatisierte Testen mit Jenkins.

Dropwizard

Dropwizard[2] ist ein open-source Java Framework zur Entwicklung von RESTful Web Services. Vom Aufsetzen eines Webservers über die Verarbeitung von JSON Dateien bis hin zur Exposition der API Endpunkte unterstützt Dropwizard den Großteil der für die Entwicklung nötigen Aufgaben.[3].
Zu diesem Zweck bündelt das Framework eine Vielzahl an gut etablierten Tools und Bibliotheken, weswegen man das Framework ebenso gut als Software Bibliothek bezeichnen könnte[4]. Entwickler, die Dropwizard verwenden, müssen sich dadurch weder um die Verwaltung der einzelnen Abhängigkeiten, noch um deren Kompatibilität untereinander kümmern und können sich auf die inhaltlichen Aspekte der Webanwendung konzentrieren. Die wichtigsten Komponenten des Frameworks werden im folgenden kurz vorgestellt:

- **Jetty**[5] ist eine HTTP Bibliothek, mit deren Hilfe sich ein schneller und zuverlässiger HTTP Server einfach in das Dropwizard Projekt integrieren lässt.
- **Jersey**[6] ist ein Framework, das sich um die Exposition der Daten über HTTP Endpunkte kümmert. Jersey ermöglicht es, mittels weniger, selbstbeschreibender Annotationen wie beispielsweise "@GET" oder "@PUT" komplette APIs zu implementieren.
- **Jackson**[7] ist eine Bibliothek von Funktionen zur Transformation zwischen (Java) Objekten und ihrer Repräsentation als JSON String. Jackson ermöglicht eine Vielzahl von Konfigurationsmöglichkeiten für diese Transformation allein durch den Einsatz von Annotationen.

[2] https://dropwizard.io
[3] Eine grundlegende Einführung zu dem Framework findet sich etwa bei (Dallas, 2014) oder direkt im ("Dropwizard Manual", 2018).
[4] Die Dropwizard Entwickler selbst merken beispielsweise an: "Dropwizard straddles the line between being a library and a framework"("Dropwizard Manual", 2018).
[5] https://www.eclipse.org/jetty/
[6] https://jersey.github.io/
[7] https://github.com/FasterXML/jackson

Darüber hinaus verwendet Dropwizard viele weitere externe Bibliotheken wie beispielsweise Guava, Logback, Hibernate Validator oder JDBI[8].
Nun gibt es neben Dropwizard eine Vielzahl an alternativen Frameworks zur Entwicklung von Web Services. Zu nennen wären hier insbesondere Spring Boot, das hervorragend mit Swagger (vgl. 5.1.2) harmoniert, Spark oder Play[9].
Wie kam also die Entscheidung zustande, weiterhin Dropwizard zu verwenden? Bezüglich der Qualität der Technik und des Supports ließen sich bei meinen Recherchen zwischen Dropwizard und den oben genannten Alternativen keine nennenswerten Unterschiede ausmachen. Den Ausschlag bei der Entscheidung für Dropwizard hat dann schließlich einfach die Tatsache gegeben, dass es bereits verwendet wird und bislang gut funktioniert hat.
Da Dropwizard auch weiterhin regelmäßige Updates erhält und ein ausführliches Manual existiert, gab es schlicht keinen Grund, der schwer genug gewogen hätte, den zusätzlichen Aufwand eines Umstiegs zu rechtfertigen.
Gerade in Bezug auf Spring Boot sollte die Möglichkeit einer späteren Umstellung aber doch nicht aus den Augen verloren werden. Denn einerseits ist Spring Boot im Vergleich zu Dropwizard das sehr viel verbreitetere und bekanntere Framework[10]. Andererseits könnte sich die bessere Integration der Swagger-Dokumentation als starkes Argument für einen zukünftigen Wechsel zu Spring Boot herausstellen. Dafür müsste allerdings zunächst eine genauere Abschätzung des Aufwandes für den Umstieg erfolgen, der im Rahmen dieser Arbeit nicht erfolgen kann.

5.1.2 Neue Technologie

JSON API

JSON API[11] ist eine Spezifikation für die HTTP Kommunikation zwischen Servern und Clients. Sie liegt fest, in welcher Form der Client Anfragen zum Datenabruf oder zur Datenmanipulation an den Server stellt und wie genau die Antwort des Servers auf diese Anfragen auszusehen hat. Eine formale Spezifikation des Datenaustauschsformats zu verwenden hat grundsätzlich den Vorteil, dass das Format der Daten außer Acht gelassen werden kann und die ODS Entwickler sich auf den inhaltlichen Teil konzentrieren können. Speziell JSON API ist genau auf

[8] Eine vollständige Liste der Fremdbibliotheken, inklusive jeweils einer kurzen Beschreibung ihrer Funktion, findet sich im "Getting-Started" Kapitel des ("Dropwizard Manual", 2018).

[9] Vergleichende Übersichten finden sich etwa unter https://www.gajotres.net/best-available-java-restful-micro-frameworks/ oder https://raygun.com/blog/popular-java-frameworks/, jeweils zuletzt aufgerufen am 17.09.2018.

[10] Eine mögliche Kennzahl dafür wäre beispielsweise die Anzahl der Github Forks (ca. 20000 Forks bei Spring Boot gegen ca. 3000 bei Dropwizard).

[11] http://www.jsonapi.org

```
Document :=
'{'
   data | error | meta,
   [jsonapi],
   [links],
   [included]
'}';
```

die Entwicklung von RESTful APIs inklusive einer konsequenten Verwendung des HATEOAS Prinzips ausgelegt (vgl. Abschnitt 2.3). Die Konzeption und Entwicklung der neuen RESTful API lässt sich durch die Verwendung von JSON API stark vereinfachen – etwa indem zur Realisierung der Verlinkung der Endpunkte auf die dort spezifizierten Link- oder Relationship Objekte zurückgegriffen wird. Da JSON API zudem ein etablierter Standard mit festgelegtem Media-Type ("application.vnd+api") ist, können Nutzer der ODS API auf eine bereits bestehende und gut gepflegte Dokumentation (Katz, Klabnik, Gebhardt, Kellen & Resnick, 2015) zurückgreifen.

Struktur eines JSON API Dokuments

Den Inhalt (HTTP Message Body) einer nach den Regeln der JSON API Spezifikation gebildeten Anfrage oder Antwort bezeichne ich im Folgenden als *JSON API Dokument*. Um die zulässige Struktur eines solchen Dokuments zu beschreiben, wird in diesem Abschnitt immer wieder eine (Pseudo-)[12] Erweiterte Backus-Naur-Form (EBNF) zum Einsatz kommen, deren Syntax und Semantik ich hier als bekannt voraussetze[13].
Eine EBNF Gesamtdarstellung aller in dieser Arbeit relevanten Elemente eines JSON API Dokuments findet sich in Anhang B. Ein Beispiel für eine (etwas komplexere) JSON API Response gibt Abbildung 5.1. Auf der höchsten Ebene besteht ein JSON API Dokument immer entweder aus einem data-, einem error- oder einem meta-Objekt[14]. Zusätzlich können jeweils ein jsonapi-, ein links- sowie ein included-Objekt enthalten sein.

[12] JSON Strings mit der EBNF zu beschreiben ist schwierig, da die Symbole "{", "}", "[","]" sowie Anführungszeichen sowohl in der EBNF- als auch in der JSON Semantik eine jeweils eigene Interpretation besitzen. Um die daraus resultierende Mehrdeutigkeit aufzulösen, werde ich sie, wo der Bezug nicht aus dem Kontext ersichtlich ist, mit einfachen Anführungszeichen markieren, wenn sie als im resultierenden JSON Dokument vorkommende Symbole zu interpretieren sind.

[13] Die Regeln zur Bildung und Interpretation der EBNF lassen sich zum Beispiel auf Wikipedia oder in jeder Einführung in die theoretische Informatik nachlesen (vgl. etwa Hedtstück, 2012).

[14] Streng genommen darf auch ein meta-Objekt mit einem der beiden anderen Top-Level Objekte koexistieren, aber dieser Spezialfall ist für unsere Bedürfnisse nicht relevant.

```
resourceObject :=
{
   '"type":' String,
   '"id":' String,
   [attributes],
   [relationships],
   [links].
   [meta];
}
```

Error-, meta- sowie jsonapi Objekte werden wir im Folgenden nicht weiter betrachten, da sie für die Implementierung der ODS API, zumindest auf ihrem jetzigen Stand, nicht verwendet werden.
An dieser Stelle interessieren uns das data-Objekt sowie die optionalen included- und links-Objekte.
Das links-Objekt ist essentiell für die Umsetzung des HATEOAS-Constraints. Hier können auf Dokumentebene bereits Verweise auf andere, verwandte Ressourcen mitgeliefert werden.

```
links :=
'"links":{'
   {'"name":' String}
'}';
```

Strukturell besteht es einfach aus einer Liste beliebig vieler Name-Link-Tupel. Um den Constraint selbstbeschreibender Nachrichten umzusetzen, wird diese Liste üblicherweise mindestens ein Tupel – einen Selflink – enthalten.
Die primären Nutzdaten hingegen finden sich vor allem im data-Objekt. Dieses besteht entweder aus einem einzelnen `resourceObject` oder einem Array von `resourceObjects`.

```
data :=
'"data":{'
   resourceObject | '[' {resourceObject} ']'
'}';
```

Ein resourceObject wiederum enthält mindestens ein `type` und ein `data` Feld, wobei optional weitere `attributes`, `links` sowie `meta` Felder hinzukommen können. Die Werte der Felder `type` und `id` ermöglichen im Zusammenspiel eine eindeutige Identifikation der Ressource – jede Kombination von Type und Id darf nur einmal vergeben werden. Enthält ein Attribute-Objekt über `type` und `id` hinaus keine weiteren Felder, so wird es als *resource identifier* bezeichnet.

```
included :=
'"included":{['
     { resourceObject }
']}';
```

Das Attributes-Objekt enthält die primären Nutzdaten der Nachricht.

```
attributes :=
'"attributes:{'
   {String':' String | Integer | Float}
'}';
```

Beispielsweise wäre im Fall einer einfachen `GET` Anfrage auf ein einzelnes Objekt hier die JSON Repräsentation des Objekts (mit Ausnahme potentieller Id oder Type Felder) zu finden. Interessant für die Umsetzung von HATEOAS sind die Felder `relationships` und `links`.
Das links-Objekt ist uns bereits auf Dokumentebene begegnet. Dass es hier – innerhalb der Nutzdaten – noch einmal auftaucht, zeigt welchen großen Stellenwert die Umsetzung des HATEOAS Constraints beim Entwurf der JSON API Spezifikation gespielt hat. Ebenso wie das links-Objekt ermöglicht auch das relationships-Objekt die Navigation zwischen verwandten Ressourcen.

```
relationships :=
'"relationships":{'
   links | resourceIdentifier | '[' {resourceIdentifier} ']'
'}';
```

Das relationshipObject kann entweder ein linkObject, einen resourceIdentifier oder ein Array von resourceIdentifiern enthalten.
Eine häufig sinnvolle Kombination beim Einsatz von resourceIdentifiern ist es, auf Dokumentenebene das included-Objekt (siehe oben) zu verwenden und die hier referenzierten Objekte in die Antwort zu integrieren. Das included-Objekt besteht immer aus einem Array von resourceObjects. Das Zusammenspiel all der hier besprochenen Elemente lässt sich noch einmal anhand von Beispiel 5.1 oder, noch ausführlicher, in Anhang A nachvollziehen.

```
{
   "links":{
      "self":"http://www.domain.com/link/to/self"
   }
   "data": {
      "type":"ObjectType",
      "id":"ObjectId",
      "attributes": {
         "veryImportantAttribute":"veryImportantValue",
         "importantNumber":42
      },
      "relationships":{
         "relatedObject":{
            "type":"relatedType",
            "id":"relatedId"
         }
      }
   }
   "included":{
      "relatedObject":{
         "links":{
            "self":"http://www.domain.com/link/to/related"
         }
         "type":"relatedType",
         "id":"relatedId",
         "attributes":{
            "pleaseIncludeThisAttribute":"ItIsImportant"
         }
      }
   }
}
```

Abbildung 5.1: Beispiel eines JSON API Dokuments

Alternativen

Neben JSON API existieren weitere Spezifikationen für den Nachrichtenaustausch über HTTP[15]. Die bekanntesten der Alternativen sind Collection+Json, HAL und JSON-LD.

- **Collection+Json**[16] ist wahrscheinlich die bekannteste der Hypermedia Spezifikationen. Der Fokus liegt hier allerdings weniger auf der Umsetzung von REST, als das bei JSON API der Fall ist.

- **HAL**[17] bietet ähnliche Möglichkeiten wie JSON API. So sind bei HAL beispielsweise "_links" und "_embedded" Attribute spezifiziert, die den Links beziehungsweise Included Attributen von JSON API entsprechen.

- **JSON-LD**[18] legt den Fokus auf das Verlinken von Daten ohne allzu stark in bestehende APIs einzugreifen. Dazu wird, im Unterschied zu JSON API, keine feste Struktur des Dokumentes vorgegeben, sondern zusätzliche Schlüsselworte (z.B. @id, @context, ...) eingeführt, mit denen bestehende API Nachrichten erweitert werden können.

Dass die Wahl schließlich auf JSON API gefallen ist, liegt einerseits an dessen starker REST-Orientierung (Features wie Möglichkeiten zur Verlinkung an verschiedenen Stellen, Relationships, Includes). Andererseits besitzt JSON API eine sehr gut geschriebene und ausführliche Dokumentation, wodurch es sich noch einmal deutlich von ähnlich REST-orientierten Spezifikationen wie HAL oder JSON-LD absetzt.

Swagger

Swagger[19] ist ein Paket von Anwendungen für den gesamten Entwicklungszyklus von Web APIs – von Design und Codegenerierung, über die Dokumentation bis hin zu Tests und Veröffentlichung (vgl. Rajput, 2018, S. 299).
Die Grundlage des Swagger Projekts bildet die zugehörige API Beschreibungssprache, die 2011 unter dem Namen "Swagger" veröffentlicht wurde. Als 2015 die Firma "SmartBear Software" das Swagger Projekt übernahm, wurde die Sprache

[15] Vergleichende Übersichten finden sich bei (Tilkov, 2015, S. 88-96), (Stowe, 2015) oder (Sookecheff, 2014).
[16] http://amundsen.com/media-types/collection/
[17] http://stateless.co/hal_specification.html
[18] https://json-ld.org/
[19] https://swagger.io/

in “OpenAPI Specification (OAS)” umbenannt und der Linux foundation gestiftet (vgl. Anonymous, o.D.). Unter der Schirmherrschaft der OpenAPI Initiative (OAI)[20] wird die OAS seitdem unter einer open-source Lizenz weiterentwickelt. Um die Spezifikation herum entstand nach und nach ein ganzes Ökosystem von Anwendungen zur API Entwicklung:

- **swagger-core** ist eine Java-Bibliothek zur Umsetzung der OpenAPI Spezifikation. Es bietet Möglichkeiten zur dynamischen Generierung der OpenAPI Spezifikation aus eigenen Swagger-Annotationen oder bereits vorhandenen Jax-RS (Jersey) Annotationen. Bei der Implementierung der neuen API für den ODS wurde swagger-core mit Swagger-Annotation verwendet, um die OpenAPI Spezifikation auf einem eigenen Endpunkt zu veröffentlichen.
- **Swagger Editor** ist ein Editor für die OpenAPI Spezifikation, mit dessen Hilfe sich APIs entwerfen lassen.
- **Swagger Codegen** generiert Serverstümpfe aus vorhandener OpenAPI Spezifikation und ermöglicht so die Umsetzung des API-First Design Ansatzes der API Entwicklung (vgl. Trieloff, 2017). Für den ODS ist das Tool nicht geeignet, da dessen Implementierung bereits vorhanden ist, weswegen die Entwicklung der neuen API auf einen Code-First Ansatz (vgl. Vasudevan, 2017) festgelegt ist.
- **Swagger-UI** visualisiert OpenAPI Spezifikationen in einer interaktiven Benutzeroberfläche. Swagger-UI kommt in der Implementierung zum Einsatz, um die von swagger-core veröffentlichte OAS auf einem weiteren Endpunkt zu visualisieren.
- **Swagger Inspector** generiert OpenAPI Spezifikationen, ohne auf den Code zurückgreifen zu müssen, indem die Endpunkte bestehender APIs angesteuert werden. Swagger Inspector wurde bei der Implementierung nicht verwendet, da die Generierung der Spezifikation dynamisch erfolgen soll und Veränderungen im Quellcode sich direkt auf die veröffentlichte Spezifikation auswirken sollen. Zudem steht die Anwendung unter einer proprietären Lizenz.
- **SwaggerHub** bündelt die Funktionen der oben aufgeführten Tools und ermöglicht (kostenpflichtig) das Design, Testen und Dokumentieren von APIs in Teams.

[20] https://www.openapis.org/

	Sponsor	Initial commit	Latest stable release	Stable release date	Software license[7]	Format	Open Source	Code generation (client)	Code generation (server)
RAML	MuleSoft	September, 2013	1.0	May 16, 2016	Apache 2.0	YAML	Yes	Yes	Yes
API Blueprint	Apiary	April, 2013			MIT	Markdown	Yes	limited	Yes
OpenAPI	Open API Initiative (OAI)	July, 2011	3.0	July 26, 2017	Apache 2.0	JSON or YAML	Yes	Yes	Yes
SERIN	UNIFOR	2011	2.0	December 2014	Creative Commons	RDF	yes	no	yes

Abbildung 5.2: Tabellarischer Vergleich der bekanntesten API-Beschreibungssprachen. Entnommen aus ("Overview of RESTful API Description Languages", 2017).

Alternativen

Neben Swagger beziehungsweise OAS existiert eine Vielzahl an alternativen domänenspezifischen Sprachen zur Beschreibung von REST APIs, deren bekannteste Vertreter *RAML*[21] und *API Blueprint*[22] sind[23]. Eine Übersicht über einige API Spezifikationssprachen zeigt Tabelle 5.2.
Da die Verwendung von Swagger zur Dokumentation der API von Beginn an Teil der Anforderungen dieser Arbeit war, stand allerdings keine der Alternativen jemals wirklich zur Debatte. In Anbetracht der Verbreitung und des umfangreichen Ökosystems hat sich diese Wahl auch nach Sichtung der Alternativen als die richtige ausgewiesen.

5.2 Konzeption der neuen API

5.2.1 Versionierung

Die Möglichkeit späterer Versionierung der API war bei der bestehenden API glücklicherweise bereits angedacht: Die URLs aller Endpunkte der v1 API sind unter der Basis-URL "`/ods/api/v1/`" erreichbar. Die Basis URL der neu implementierten API ergibt sich daraus logisch als "`ods/api/v2`". Dadurch kann parallel zur neu implementierten v2 API weiterhin die bestehende v1 API ge-

[21] https://raml.org/
[22] https://apiblueprint.org/
[23] Im englischsprachigen Wikipedia Artikel gibt es eine sehr ausführliche Auflistung dieser Sprachen ("Overview of RESTful API Description Languages", 2017). Eine Vergleich der bekanntesten Vertreter findet sich auch bei (Sandoval, 2016).

nutzt werden.
Ein mit der Versionierung zusammenhängendes Designziel bestand darin, so wenig wie möglich in den bestehenden Code einzugreifen, um das Risiko, bestehende Nutzungen einzuschränken, zu minimieren.
Diese Zielsetzung spiegelt sich in der Package-Struktur der neuen API Implementierung: Alle neu angelegten Klassen finden sich in einem eigenen Package, außerhalb dessen so wenige Änderungen wie möglich vorgenommen wurden.

5.2.2 Endpunkte

Das Design der Endpunkte in der bestehenden API wird den Anforderungen einer REST API und Best Practices wie beispielsweise (Haldar, 2017) bereits weitestgehend gerecht: Es werden Substantive in ihrer Pluralform verwendet und die Namen der Endpunkte sind von ihrer Repräsentation unabhängig insofern sie keine Dateiendungen enthalten. Da sich die Aufteilung in bisherige Endpunkte zudem weitestgehend logisch aus der Funktionalität des Open-Data-Service ergibt, mussten keine tiefgreifenden Änderungen vorgenommen werden.
Der Größte Unterschied ergibt sich aus der Anforderung, den HATEOAS Constraint des REST Architekturstils umzusetzen:
Um zu erreichen, dass jeder Endpunkt von jedem beliebigen anderen Endpunkt der API zu erreichen ist, wurde ein API Entrypoint hinzugefügt, von dem aus alle in API v1 verwendeten Basis-Endpunkte (vgl. die grau markierten Kategorien in den Tabellen 4.1 und 4.2) erreichbar sind. Umgekehrt besitzt nun jeder dieser Basis-Endpunkte eine Referenz auf den Entrypoint.
Durch die beschriebene (wenigstens mittelbare) Verlinkung aller Endpunkte untereinander lässt sich die API als endlicher Zustandsautomat modellieren, bei dem jeder Zustandsübergang dem Wechsel von einem Endpunkt zum nächsten entspricht.
Weggefallen ist der Version-Endpoint aus v1, der in der Neuimplementierung in den Entrypoint integriert wurde. Bei Betreten der API über den Entrypoint werden nun die Versionsinformationen mit den Links zu den Basis-Endpunkten gleich mitgeliefert. Ein weiterer neuer Endpunkt ist der “doc”-Endpunkt. Der alleinige Zweck dieses Endpunkts ist es, die (rohe) OPEN API Spezifikation der ODS API bereitzustellen, damit diese dann von Swagger-UI konsumiert und visualisiert werden kann.
Neben diesen Veränderungen der Endpunkte selbst wurde auch eine kleine Änderung der auf den Endpunkten ausführbaren Operationen vorgenommen: In API v1 wurden Anfragen zur Neuerstellung von Objekten (Datenquellen, Nutzer, Filterchains etc.) durch das HTTP Verb `PUT` realisiert. Die HTTP1.1 Spezifikation empfiehlt, dass der Aufruf der `PUT` auf einem bereits besthenden Objekt zu einer Modifikation dieses Objekts führt (Fielding et al., 1999, S. 54).
Da der ODS solche Modifikationen allerdings nicht erlaubt, wurden die `PUT`-

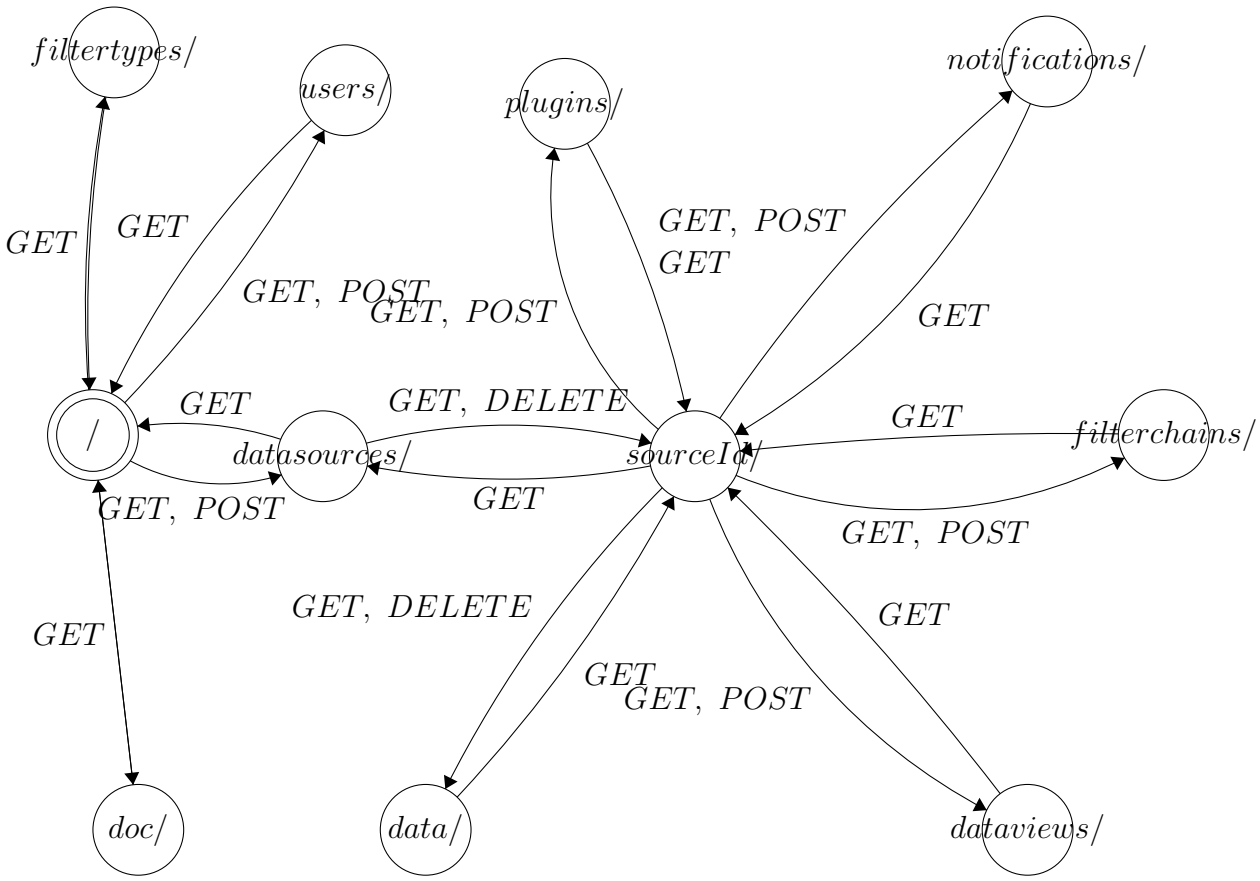


Abbildung 5.3: Darstellung der ODS API v2 als Zustandsautomat

Methoden in der API v2 durch `POST`-Methoden ersetzt, die der in der HTTP Spezifikation vorgegebenen Semantik besser entsprechen(Fielding et al., 1999, S. 53). Abbildung 5.3 zeigt die resultierende API als Zustandsautomat (vgl. Abschnitt **??**). Dabei entsprechen die Zustände den möglichen Endpunkten sowie die Kanten den möglichen Operationen.
Die Gesamtheit der resultierenden Endpunkte und der darauf ausführbaren Operationen lässt sich anhand der Tabellen 5.1 und 5.2 nachvollziehen.

5.2.3 Dokumentation

Eine Anforderung an die Implementierung der API v2 ist deren umfassende Dokumentation mit Swagger. Nutzer der API können sich dann einfach und schnell mittels der von Swagger bereitgestellten graphischen Benutzeroberfläche über die verwendeten Endpunkte und die Struktur der gesendeten Daten informieren.
Um zu verhindern, dass sich in der Implementierung der API Änderungen ergeben, ohne dass die Dokumentation entsprechend angepasst wird, bestand eine Designziel bei der Erstellung der Dokumentation darin, die generierung der Dokumentation so eng wie möglich an den Quellcode zu koppeln.
Wird die Dokumentation statisch aus einem gesondert gespeicherten Spezifikationsdokument generiert, divergiert die tatsächliche Umsetzung der API erfahrungsgemäß immer weiter von ihrer Dokumentation – das Anpassen des Spezifikationsdokuments wird bei kleinen Änderungen der Implementierung schlicht häufig vergessen.
Um eine möglichst enge Kopplung von Code und Dokumentation zu erreichen wurden im Quellcode mit Hilfe der Bibliothek *swagger-core* (vgl. 5.1.2) direkt an den für die Implementierung der API verwendeten Stellen Annotationen zur Generierung der Spezifikation hinzugefügt.
Die aus den Annotationen generierte Spezifikation wird an einem eigenen Endpunkt (dem oben beschriebenen "/doc"-Endpunkt) veröffentlicht. Bei Bedarf lässt sich in einem eigenständigen Docker-Container (vgl. 5.1.1) Swagger-UI (vgl. 5.1.2) starten. Swagger-UI greift dann auf den "/doc"-Endpunkt zu, um aus der Spezifikation eine menschenlesbare Dokumentation zu erstellen und unter einer frei wählbaren URL (momentan noch `localhost:8082`) zu veröffentlichen. Diese URL, mit einem beliebigen Webbrowser angesteuert, zeigt eine Visualisierung der API Spezifikation, die leicht zu erfassen und sogar testbar ist. Anhang C zeigt die für die API resultierende Oberfläche ausschnitssweise.

5.2.4 Datenaustausch

Die Möglichkeit zur Umsetzung der REST Constraints ist in großem Maße von dem Format der ausgetauschten Daten abhängig. Insbesondere die geforderte Ver-

Data	Base-URL: /datasources/{sourceId}	
Verb	URL	Description
GET	/data	Get data objects from {sourceId}
GET	/pointer:/.*	Get certain attributes of a data object
DELETE	/data	Delete all data objects from {sourceId}
DataSources	Base-URL: /datasources/	
Verb	URL	Description
GET	/	Get all data sources
GET	/{id}	Get a data source by its {id}
GET	/{id}/schema	Get the data schema for a data source
POST	/{id}	Add a data source with a specified {id}
DELETE	/{id}	Delete a datasource
DataView	Base-URL: /datasources/sourceId/views	
Verb	URL	Description
GET	/	Get all data views
GET	/{id}	Get a data view by its {id}
POST	/{id}	Add a data view with a specified {id}
DELETE	/{id}	Delete a datasource
Notifications	Base-URL: /datasources/sourceId/notification	
Verb	URL	Description
GET	/	Get all clients registered for notifications
GET	/{id}	Get a client by its {id}
POST	/{id}	Register a client with a specified {id}
DELETE	/{id}	Unregister a client
Plugins	Base-URL: /datasources/sourceId/plugins	
Verb	URL	Description
GET	/	Get all plugins
GET	/{id}	Get a plugin by its {id}
POST	/{id}	Add a plugin with a specified {id}
DELETE	/{id}	Delete a plugin
ProcessorChain	Base-URL: /datasources/sourceId/filterChains	
Verb	URL	Description
GET	/	Get all processor chains
GET	/{id}	Get a processor chain by its {id}
POST	/{id}	Add a processor chain with a specified {id}
DELETE	/{id}	Delete a processor chain

Tabelle 5.1: Liste der V2 Endpunkte I

ProcessorSpecification	Base-URL: /filterTypes	
Verb	URL	Description
GET	/	Get all specifications
Users	Base-URL: /users	
Verb	URL	Description
GET	/	Get all users
GET	/{id}	Get a user by its {id}
GET	/me	Get currently logged in user
POST	/{id}	Create new user with a specified {id}
DELETE	/{id}	Delete a user
Entrypoint	Base-URL: /	
Verb	URL	Description
GET	/	Get initial links and version information
OPEN API Spec	Base-URL: /doc	
Verb	URL	Description
GET	/	Get OPEN API Spec for the API

Tabelle 5.2: Liste der V2 Endpunkte II

linkung der Endpunkte untereinander muss durch den Inhalt der ausgetauschten HTTP Nachrichten realisiert werden. Zwar bietet das HTTP Protokoll theoretisch auch die Möglichkeit, Links in einem eigenen Header Feld der HTTP Nachricht zu versenden, doch gerade für komplexere Zusammenhänge eignet sich der Linktransfer über den Nachrichteninhalt besser. Beispielsweise können, wenn eine Nachricht mehrere Entitäten enthält, etwa wenn Collections oder Inkludierte Entitäten übertragen werden, die Links im Nachrichtenheader nicht mehr den einzelnen Entitäten zugeordnet werden.
Um die Verlinkung im Inhalt der HTTP Nachricht zu realisieren, mussten nur die von JSON API vorgegebenen Möglichkeiten ausgeschöpft werden – die Konzeption wurde in diesem Fall zum größten Teil von der ausgewählten Technologie vorgegeben.
Die Entscheidung, welche der von JSON API ermöglichten Form der Verlinkung für die Antworten des ODS verwendet wird, wurde dabei nach inhaltlich-logischen Kriterien getroffen (vgl. Abschnitt 5.1.2):

- Auf **Dokumentenebene** wurden Links angelegt, wenn sich die abzubildende Beziehung auf den Inhalt des gesamten JSON API Dokuments bezieht. Ein Beispiel für diese Situation ist etwa der Link von einer Datenquelle auf die Collection aller Datenquellen des ODS innerhalb einer Antwort, die als einzige Entität diese Datenquelle enthält.
- Auf **Ressourcenebene** wurden Links angelegt, wenn eine Nachricht mehrere Entitäten enthält und sich der Link nur auf eine einzelne dieser En-

titäten bezieht, wie beispielsweise bei den Selflinks der einzelnen Datenquellen in einer Collection von Datenquellen.

- Auf **Relationen** wurde zurückgegriffen, wenn zwischen zwei Entitäten eine Beziehung besteht, die sich aus der inneren Logik der Domäne ergibt, beispielsweise bei einer Verlinkung zwischen einer einzelnen Datenquelle und der Collection der darauf definierten DataViews.
- Zusätzlich wurden im **included**-Feld bei bestehenden Relationen die jeweiligen Relationspartner mit der Nachricht mitgeschickt.

Bei den Anfragen an den ODS hingegen, wurde auf die Möglichkeit, Links zwischen den Ressourcen zu erstellen vollständig verzichtet, da die logische Verknüpfung der Entitäten des ODS in den Aufgabenbereich des Servers fällt.
Neben diesen Überlegungen inhaltlicher Art musste entschieden werden, ob zur Umsetzung der JSON API Spezifikation ein Framework verwendet wird, oder ob die Daten "manuell" in die nötige Form gebracht werden. Mit *crnk*[24], *elide*[25] und *katharsis*[26] existieren gar drei Java-Frameworks die JSON API serverseitig implementieren. Leider haben alle diese Frameworks den Nachteil, dass sie sehr tief ansetzen: Um sie zu verwenden, muss die Struktur der Modellklassen, wenn nicht sogar der Datenbankpersistierung den Anforderungen von JSON API entsprechen. Da eine Vorgabe der Implementierung der ODS API darin bestand, möglichst wenig in das bestehende System einzugreifen, kam leider keines der drei Frameworks in Frage und die Umwandlung in eine JSON API-konforme Datenrepräsentation musste von Hand geschehen.

5.3 Implementierungsdetails

5.3.1 Response und Request Klassen

Um die Konvertierung zwischen den Modellklassen des ODS auf der einen sowie deren JSON API-konformer Repräsentation auf der anderen Seite zu bewerkstelligen, wurden JsonApiResponse beziehungsweise JsonApiRequest Klassen erstellt.

Requests

Wie ein JSON API Response von den API-Klassen aus konsumiert werden kann, zeigt Abbildung 5.4.

[24] http://www.crnk.io/
[25] http://elide.io/
[26] https://github.com/katharsis-project/katharsis-framework

```java
@PUT
@Path("/{viewId}")
public DataView addView(
        @RestrictedTo(Role.ADMIN) User user,
        @PathParam("sourceId") String sourceId,
        @PathParam("viewId") String viewId,
        @Valid DataViewDescription viewDescription) {

    DataSource source = sourceManager.findBySourceId(sourceId);
    if (viewManager.contains(source, viewId))
        throw RestUtils.createJsonFormattedException("data view with id " + viewId + " already exists", 409);

    DataView view = new DataView(viewId, viewDescription.getMapFunction(), viewDescription.getReduceFunction());
    viewManager.add(source, sourceManager.getDataRepository(source), view);
    return view;
}
```

DataViewApi v1

```java
@POST
public Response addView(
    @RestrictedTo(Role.ADMIN) @Parameter(hidden = true)
        User user,
    @PathParam("sourceId")
        String sourceId,
    @RequestBody(
        description = "Description of the DataView to be added.",
        required = true,
        content = @Content(schema = @Schema(implementation = JsonApiSchema.DataViewSchema.class)))
        JsonApiRequest viewDescriptionRequest) {

    DataViewDescription viewDescription = JsonMapper.convertValue(
        viewDescriptionRequest.getAttributes(),
        DataViewDescription.class
    );

    DataSource source = sourceManager.findBySourceId(sourceId);
    assertIsValidViewDescription(viewDescription, viewDescriptionRequest.getId(), source);

    DataView view = new DataView(viewDescriptionRequest.getId(),
        viewDescription.getMapFunction(),
        viewDescription.getReduceFunction());
    viewManager.add(source, sourceManager.getDataRepository(source), view);

    URI directoryURI = getSanitizedPath(uriInfo);

    return JsonApiResponse
        .createPostResponse(uriInfo)
        .data(DataViewWrapper.from(view))
        .addLink(JsonLinks.SELF, directoryURI.resolve(viewDescriptionRequest.getId()))
        .addLink(VIEWS, directoryURI)
        .build();
}
```

DataViewApi v2

Abbildung 5.4: Gegenüberstellung der Request Konsumption

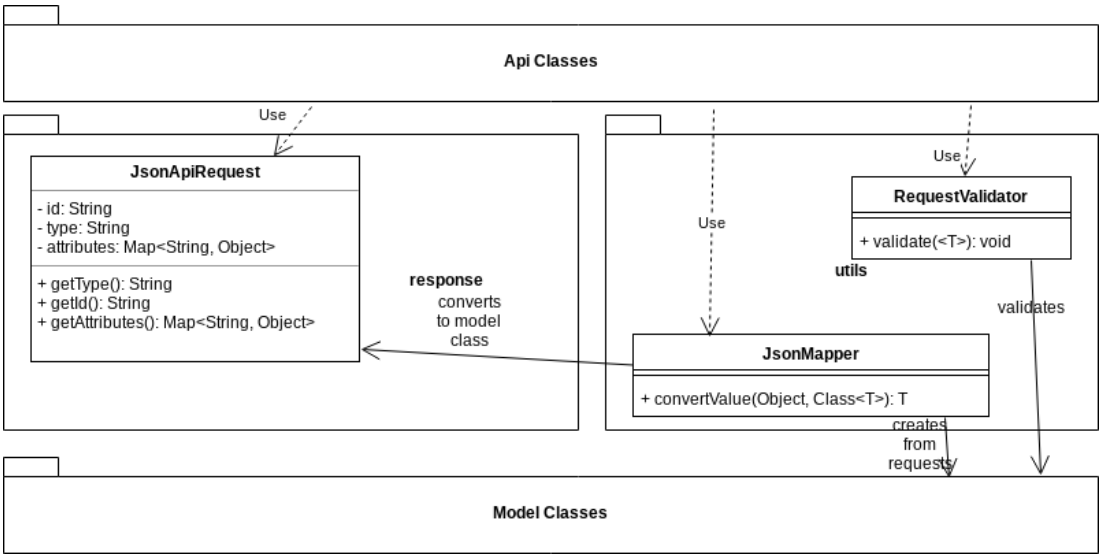


Abbildung 5.5: Klassendiagramm der Request Klassen

Die mit `@PUT`- und `@POST`-Annotationen versehenen Methoden werden von der Jersey Bibliothek aufgerufen, wenn an einem (hier auf Klassenebene) definierten Pfad die entsprechenden HTTP Methoden über die API aufgerufen werden.
Der Eingabeparameter, der dem Message Body des Requests entspricht, ist jeweils der letzte – die viewDescription in v1 beziehungsweise der viewDescription-Request in v2.
Anhand des Typs des Eingabeparameters lässt sich die unterschiedliche Funktionsweise der API v1 und v2 gut aufweisen: Während in v1 aus dem Java Objekt vom Typ einer Modellklasse (DataViewDescription) die erwartete json Repräsentation abgeleitet wurde, wird in v2 die generische JsonApiResponse Klasse verwendet. Wie in Abbildung 5.5 dargestellt, besitzt die JsonApiResponse Klasse id und type Felder sowie eine attributes Map. In Verbindung mit einigen Annotationen zur Konfiguration der JSON Deserialisierung konnte so erreicht werden, dass Jackson die Struktur beliebiger Message Bodies im JSON API Format erwartet und korrekt deserialisiert.
Da die so erhaltene Klasse aber nun noch den falschen Typ besitzt muss er nun zunächst vom JsonMapper in den erwarteten Modellklassentyp umgewandelt werden.
In API v1 wurden die empfangenen Daten bei der Deserialisierung durch die `@Valid` noch einmal auf ihre Konsistenz geprüft. Um diese Validierung in der API v2 nachzubilden, müssen die gerade in den Typ der Modellklassen umgewandelten Objekte von dem RequestValidator noch einmal validiert werden. Die für die Typkonversion und Validierung zuständigen Methoden sind jeweils sehr kurz und generisch, weswegen ihr Quellcode in Anhang E vollständig angefügt ist.

Responses

Abbildung 5.7 zeigt, wie die JsonApiResponse Klasse aufgerufen werden kann, um eine Antwort im nötigen Format zu erzeugen.
In der linken Spalte ist erkennbar, dass in der API v1 der Rückgabetyp der Methode einfach in einer Liste der Modelklasse besteht. Jersey und Jackson erstellen daraus automatisch eine HTTP Antwort, die eine JSON Repräsentation der entsprechenden Java Objekte enthält.
Die rechte Spalte der Abbildung zeigt, wie die JsonApiResponse Klasse aus der API aufgerufen wird. Dabei werden zwei Besonderheiten der Implementierung deutlich:

1. Der Erzeugung des Rückgabewerts der Methode geschieht durch eine Kette hintereinander ausgeführter Methodenaufrufe, die sich fast wie ein Satz in englischer Sprache lesen lassen. Diese Art von Methodenaufrufen wurde von Martin Fowler als *Fluent Interface* eingeführt und hat den primären Zweck flüssig lesbaren Code zu ermöglichen.[27]
   In der JsonApiResponse Klasse wird das Prinzip des Fluent Interface durch den Einsatz des Builder-Patterns umgesetzt. Der erste Methodenaufruf `JsonApiResponse.createGetResponse()` gibt einen Builder zurück, auf dem weitere Modifikationen der zu erstellenden JsonApiResponse durchgeführt werden können. Nachdem der Response alle benötigten Informationen mitgegeben wurden, kann durch den Aufruf der `build()` Methode eine fertig konfigurierte Response erhalten werden.
   Die Zwischenprodukte in diesem Erzeugungsprozess haben jeweils eigene statische Typen, die definieren, welche Operationen auf ihnen ausgeführt werden können, wodurch sich eine Reihenfolge der ausführbaren Operationen vorgeben lässt. Abbildung 5.6 zeigt, wie diese Interfaces im Code umgesetzt wurden.

2. Der Rückgabetyp der mit `@GET` annotierten Methode ist Response. Die Klasse Response wird von der Jersey Bibliothek standardmäßig verwendet, um Antworten auf HTTP Anfragen zu generieren.
   Um auf die von Jersey bereitgestellte Funktionalität aufbauen zu können, wurde JsonApiResponse als Unterklasse der Jersey-Response konzipiert.
   Die Funktionsweise der JSON API Response lässt sich als Wrapper um die Jersey-Response beschreiben. Während des oben beschriebenen Fluent Interface Konfigurationsprozesses, der von dem Builder übernommen wird, werden die jeweils zu konfigurierenden Daten in eine JSON API konforme Form gebracht und intern in einer JsonApiResponse Instanz gespeichert.

[27] Zur Einführung in das Thema empfiehlt sich (Fowler, 2005), wo Martin Fowler den Begriff zum ersten mal verwendet.

```
    public static class Builder implements RequiredEntity, WithRelationship, Buildable {...}

    /**
     * Interface for a Responsebuilder that needs an entity for further processing
     */
    public interface RequiredEntity {...}

    /**
     * Interface for a Responsebuilder that meets all requirements to build the response.
     */
    public interface Buildable {...}


    /**
     * Interface for a Responsebuilder which has at least one relationship added.
     * It is needed to ensure that included entities can only be added to responses that contain relationships.
     * Extends Buildable since it can only be returned if the Responsebuilder already contains an entity
     */
    public interface WithRelationship extends Buildable {
        /**...*/
        WithRelationship addIncluded(JsonApiIdentifiable included);
    }

}
```

Abbildung 5.6: Realisierung der Reihenfolgevorgaben für ein Fluent Interface

ProcessorSpecificationApi v1

```
@GET
public List<Specification> getAllSpecifications() {
    return new LinkedList<>(descriptionManager.getAll());
}
```

ProcessorSpecificationApi v2

```
@GET
public Response getAllSpecifications() {
    Set<Specification> specs = descriptionManager.getAll();

    return JsonApiResponse
        .createGetResponse(uriInfo)
        .data(SpecificationWrapper.fromCollection(specs))
        .addLink(ENTRYPOINT, getDirectoryURI(uriInfo))
        .build();
}
```

Abbildung 5.7: Gegenüberstellung der Response-Erstellung

Wenn schließlich die `build()` Methode aufgerufen wird, werden die nun JSON API konformen Daten an den Jersey Response Builder übergeben und damit eine entsprechende Jersey Response zurück gegeben.

Abbildung 5.8 visualisiert die oben beschriebenen Zusammenhänge. Zusätzlich ist erkennbar, wie sich die Struktur eines zulässigen JSON API Dokuments in der Klassenstruktur des Response Packages spiegelt.
Jede Response besitzt ein JsonApiDokument (entspricht der Dokumentebene der Spezifikation), dieses mindestens eine JsonApiResource (Ressourcenebene der Spezifikation), für die wiederum die Möglichkeit besteht, beliebig viele JsonApiRelationships hinzuzufügen.
Die Möglichkeit, Links anzulegen besteht laut Spezifikation auf Dokumenten- sowie Ressourcenebene und wird hier durch das Interface JsonLinks realisiert.
In JSON API-konformen Message Bodies enthaltene Entitäten werden durch die Klassen JsonApiResource oder JsonApiRelationship abgebildet (inkludierte Klassen sind ebenfalls JsonApiResourcen). Diese beiden Klassen implementieren das Interface JsonApiIdentifiable, das mindestens die Attribute `id` und `type` garantiert, und stellen damit die Schnittstelle zu den Modellklassen des ODS dar.

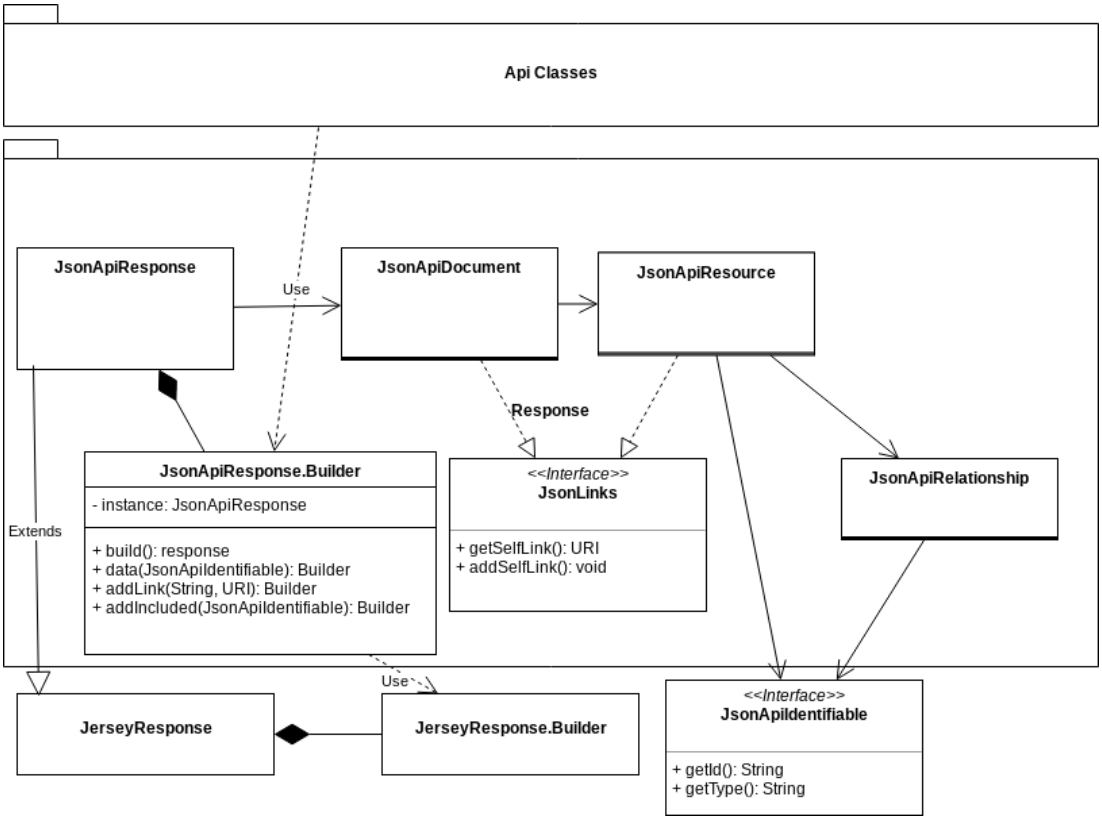


Abbildung 5.8: Klassendiagramm der Response Klassen

5.3.2 Wrapper und Identifiable-Interface

Um die von `GET`-Anfragen angeforderten Java Objekte in eine JSON API-konforme Repräsentation zu bringen, muss auf die Attribute `type` und `id` zugegriffen werden.
Dabei ergaben sich mehrere Probleme: Einerseits verfügen viele der Modellklassen gar nicht über beide Attribute, andererseits implementieren die Modellklassen kein gemeinsames Interface, über das der Zugriff auf diese Felder möglich wäre.
Da die Eingriffe in die bestehende Codebasis minimal gehalten werden sollten, war es auch keine überzeugende Option, ein solches Interface zu erstellen und die Modellklassen implementieren zu lassen.
Stattdessen wurde für jede Modellklasse eine Wrapperklasse entworfen, die eine Referenz auf die entsprechende Modellklasse besitzt und das JsonApiIdentifiable Interface implementiert.
Durch diese Konstruktion konnte den Response Klassen der Zugriff auf die type und id Felder der Modellklassen ermöglicht werden, ohne die Modellklassen verändern zu müssen. Gleichzeitig konnten so bei nicht vorhanden Attributen Defaultwerte definiert werden. So ergibt sich beispielsweise der Wert des type-Attributs aus dem jeweiligen Typnamen der entsprechenden Java Modellklasse.
Das Klassendiagramm in Abbildung 5.9 zeigt, wie diese Strategie auf Klassenebene umgesetzt wurde.

5.3.3 Umsetzung der Dokumentation

Für die Umsetzung der Swagger Dokumentation war neben der Konfiguration der Bibliothek und der Einarbeitung in die Funktionsweise der Annotationen eine enorme Menge an – teilweise monotoner und stupider – Handarbeit nötig.
Abbildung 5.10 zeigt beispielhaft, wie eine vollständig annotierte API-Methode aussieht.
Man erkennt, dass die Annotationen, aus denen die OAS generiert wird, zwar direkt über der Methode verortet sind, aber dennoch eine Trennung existiert. Es ist weiterhin möglich, die innere Logik der Methode zu verändern, ohne die Annotationen anzupassen, wenngleich die räumliche Nähe der beiden die Hemmschwelle, einen inkonsistenten Zustand zu hinterlassen hoffentlich erhöht. Eine Schwierigkeit, die bei der Erstellung der Dokumentation auftauchte, bestand darin, die Schemata für die erwarteten Daten korrekt zu erstellen. Wie in der Abbildung zu erkennen ist, bieten die Swagger Annotationen, die Möglichkeit für das Datenschema als Implementierung eine Java Klasse anzugeben. Auf Basis der Attribute dieser Java Klasse wird dann die Dokumentation der erwarteten JSON Repräsentation der Daten erstellt.
Da durch die Verwendung der JSON API Spezifikation die JSON Repräsentation der Objekte aber von der Struktur der entsprechenden Java Klassen abweicht,

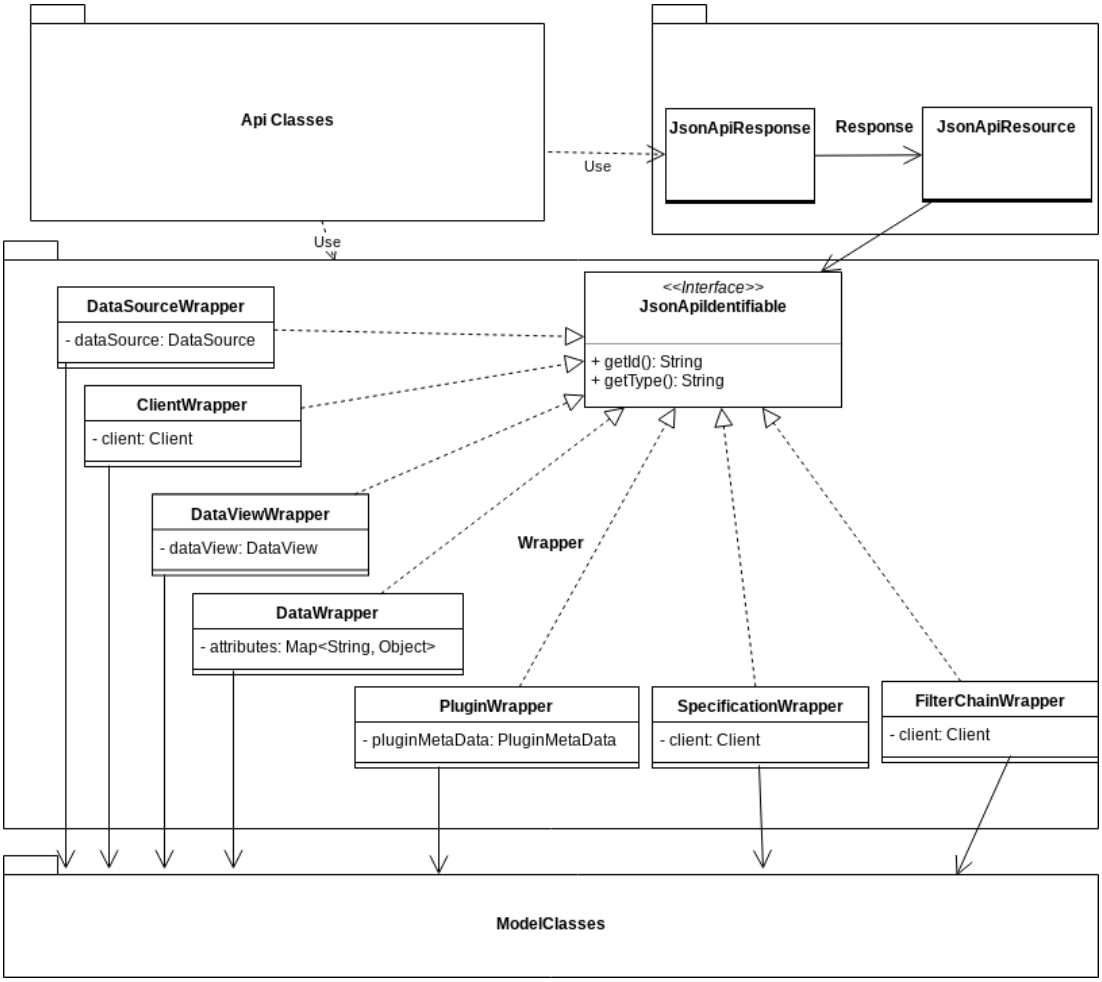


Abbildung 5.9: Klassendiagramm der Wrapper Klassen

```java
@Operation(
    tags = DATASOURCES,
    operationId = DATASOURCE,
    summary = "Get a datasource",
    description = "Get datasource with a specific sourceId")
@ApiResponse(
    responseCode = "200",
    description = "Ok",
    content = @Content(schema = @Schema(implementation = JsonApiSchema.DataSourceSchema.class)),
    links = {
        @Link(name = VIEWS, operationRef = VIEWS, description = "Get views for this datasource"),
        @Link(name = NOTIFICATIONS, operationRef = NOTIFICATIONS, description = "Get notification clients for this datasource"),
        @Link(name = PLUGINS, operationRef = PLUGINS, description = "Get plugins for this datasource")
    })
@ApiResponse(responseCode = "404", description = "Datasource not found")
@GET
@Path("/{sourceId}")
public Response getSource(
    @PathParam("sourceId") @Parameter(description = "The id of the datasource")
        String sourceId) {
    DataSource source = sourceManager.findBySourceId(sourceId);

    JsonApiResponse.Buildable response = JsonApiResponse
        .createGetResponse(uriInfo)
        .data(DataSourceWrapper.from(source))
        .addLink(DATA, getSanitizedPath(uriInfo).resolve(DATA))
        .addLink(DATASOURCES, getDirectoryURI(uriInfo));

    response = addDatasourceRelationships(response, source);

    return response.build();
}
```

Abbildung 5.10: Beispiel einer mit Swagger Annotationen versehenen API-Methode

musste auch hier wieder auf Wrapper Objekte zurückgegriffen werden (vgl. JsonApiSchema.DataSourceSchema.class in der Abbildung). Anhang F zeigt die Implementierung der hier verwendeten Wrapper Klassen.
Da die Swagger Annotationen zur Erzeugung des Schemas nur Methodendeklarationen betrachten und deren Implementierung ignorieren, konnte für jedes benötigte Schema einfach eine eigene abstrakte Klasse angelegt werden, die zentral in der JsonApiSchema Klasse gesammelt wurden.

5.3.4 Testcases

Um die Anforderung 3.1.5 umzusetzen, wurden für neu angelegte Klassen und Methoden konsequent Tests implementiert. Dafür wurden die für das Projekt bereits vorher angewendeten Test-Frameworks junit und jmockit verwendet. Wie in Abbildung 5.11 erkennbar, konnte dabei auf etablierte Konventionen zurückgegriffen werden, nach denen die Tests in einer Package Struktur geordnet werden, die die der zu testenden Klassen widerspiegelt. Abbildung 5.12 zeigt beispielhaft eine Verwendung der beiden Test-Frameworks.

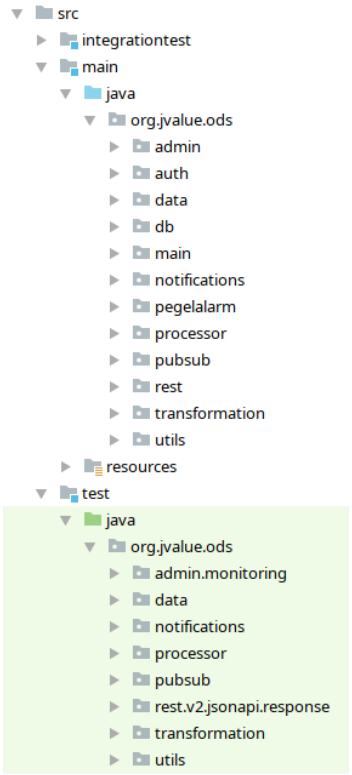


Abbildung 5.11: Package Struktur der Unit Tests

```
@Test
public void testPostResponse() {
    //Record
    JsonApiIdentifiable testEntity
        = createEntityWithAttributes();

    //Replay
    Response result = JsonApiResponse
            .createPostResponse(uriInfo)
            .data(testEntity)
            .build();

    //Verify
    Assert.assertEquals(
         expected: 201, result.getStatus());
    assertIsValidJsonApiDataResponse(
        result);
    assertHasValidData(
        extractJsonEntity(result));
}
```

Abbildung 5.12: Beispiel eines Unit-Tests mit Verwendung von jmockit

6 Evaluation

In diesem Kapitel werden die in Kapitel 3 aufgestellten Anforderungen noch einmal aufgegriffen und im einzelnen überprüft, inwieweit die erstellte Implementierung ihnen gerecht werden kann.

6.1 Funktionale Anforderungen

6.1.1 Erfüllung der REST Constraints

Die Anforderung 3.1.1 fordert die Einhaltung aller REST-Constraints, wie sie in Abschnitt 2.3 beschrieben und in Tabelle 2.1 aufgelistet wurden. Um zu überprüfen, ob die Anforderung erfüllt werden konnte, werden die REST-Constraints im Folgenden nacheinander aufgelistet und im einzelnen evaluiert.

1. **Client-Server:** Der Constraint ist erfüllt. Die Implementierung einer API definiert ja gerade die Kommunikation zwischen Client und Server und stellt die Trennung der Zuständigkeit zwischen den beiden dadurch zwangsweise her.

2. **Statelessness:** Auf Seiten des ODS werden keine Informationen über bereits eingegangene Requests gespeichert. Jeder Request wird isoliert abgearbeitet. Die Anforderung der Zustandslosigkeit ist damit erfüllt.

3. **Cache:** Formal erfüllt jede Kommunikation über HTTP diesen Constraint, denn HTTP definiert für nicht explizit markierte Dateien als Defaultwert deren volle Zwischenspeicherbarkeit (vgl. Fielding et al., 2014). Implizit ist damit die Zwischenspeicherbarkeit jedes Pakets markiert. Die Verwendung des Cache Control Header Felds beschränkt sich damit auf Fälle, in denen die Zwischenspeicherbarkeit eingeschränkt werden soll. In dem ODS wird aktuell kein Gebrauch von dieser Möglichkeit gemacht. Domänenabhängig kann es, je nach Datenquelle, unterschiedliche Gültigkeitszeiten der bereitgestellten Daten geben. Es könnte sich demnach als lohnend herausstellen,

zukünftig die gesendeten Daten mit einem spezifischen Ablaufdatum für den Client Cache zu versehen.

4. **Layered System:** Als Schichten im Sinne Fieldings kann man die Datenbank-, API/Server- sowie Importkomponenten bezeichnen, wenngleich Fielding wahrscheinlich eher Netzwerkkomponenten im Sinn gehabt haben dürfte (vgl. Fielding, 2000, S. 82-83). Konzeptuell sind diese Komponenten vollständig, in ihrer Umsetzung zum allergrößten Teil voneinander entkoppelt (vgl. Abschnitt 4). Der Layered-System Constraint kann damit als erfüllt gelten, wenngleich die Voraussetzungen dafür weniger in der API als in der ursprünglichen Konzeption des ODS gelegt wurden.

5. **Code-on-Demand:** Da die Erfüllung dieses Constraints optional ist, gibt es logisch gar keine Möglichkeit ihn *nicht* zu erfüllen. Die Möglichkeit zur Umsetzung von Code-on-Demand wurde im ODS nicht ausgeschöpft. Angesichts der primären Funktion des ODS, der Aufbereitung und Verfügbarmachung öffentlicher Daten, gibt es schlicht keinen sinnvollen Einsatzzweck für den Download und die Ausführung serverseitig generierten Codes beim Client.

6. **Uniform Interface:**

   - **Identification of Resources:** Die Einhaltung dieses Constraints lässt sich Anhand der Auflistung der Endpunkte in den Tabellen 5.1 und 5.2 nachvollziehen: Die Endpunkte werden durch URIs addressiert und bezeichnen abstrakte Ressourcen wie im in Abschnitt 2.3.6 beschriebenen Sinne.

   - **Manipulation of Resources through Representations:** Über die API des ODS gibt es nur zwei Möglichkeiten Ressourcen zu manipulieren:

     – **POST:** das Anlegen neuer Entitäten geschieht in der API durch die HTTP-Methode Post. Ein Post-Request muss als Nachrichteninhalt die JSON API Repräsentation der anzulegenden Entität beinhalten. Die Manipulation der Resource (die Erstellung) geschieht somit über ihre Repräsentation.

     – **DELETE:** Hier wird der Constraint streng genommen nicht vollständig eingehalten. Wenn `DELETE` über die API auf einem Endpunkt aufgerufen wird, wird die entsprechende Entität im ODS gelöscht. Bei folgenden Aufrufen der `GET`-Methode auf diesem Endpunkt wird der Fehlercode “404 - not found” zurück geliefert, was nach der HTTP Semantik als Fehlen der *Ressource* zu interpretieren ist. Um den Constraint vollständig einzuhalten, müsste eigentlich bei dem erneuten Anfordern der gelöschten Ressource ein “410 - gone” zurückgeliefert werden, was ein Fehlen einer *Repräsentation*

signalisiert – während die Ressource, verstanden als Zuordnungsfunktion zwischen URI und Repräsentation, weiterhin existiert. Ob dieses eher akademische Problem aber den Mehraufwand rechtfertigt, serverseitig alle jemals entfernten Ressourcen zu speichern, um nach deren erneuter Anforderung den richtigen Fehlercode zurückzuliefern, darf bezweifelt werden.

- **Self Descriptive:** Die Selbstbeschreibung der Nachrichten wird bereits durch die Verwendung des vom HTTP Protokoll definierten Content-Type Header Feld garantiert. Darüber hinaus besitzt jede Antwort des ODS, die Daten enthält, mindestens einen Selflink, wodurch sich die Aussagekraft der Selbstbeschreibung noch einmal erhöht, da anhand des Selflinks der zu der Antwort führende Request nachvollzogen werden kann.
- **HATEOAS:** Innerhalb der API ist es möglich von jedem Endpunkt aus jeden anderen Endpunkt mittels einer Kette von Querverweisen zu erreichen (vgl. dazu noch einmal das Zustandsdiagramm 5.3). Abbildung 6.1 zeigt zur Verdeutlichung die Antwort, die man beim betreten der API über ihren Entrypoint (`GET .../ods/api/v2`) erhält. Inhaltlich verwandte Ressourcen werden darüber hinaus in dem von JSON API bereitgestellten Relationship-Feld untereinander verknüpft. Der HATEOAS-Constraint ist damit erfüllt.

Die Anforderung 3.1.1 kann im Wesentlichen als erfüllt gelten. Formal wurde zwar der Constraint "Manipulation of Resources through Representations" nicht eingehalten. Die Auswirkungen dieser Abweichung fallen allerdings nicht merklich ins Gewicht. Der Aufwand, der für die strikte Einhaltung des Constraints nötig wäre, steht daher in keinem Verhältnis zu den daraus resultierenden Vorteilen.

6.1.2 Versionierung

Die neu implementierte API läuft, isoliert von API v1, unter einer eigenen Basispfad. API v1 ist weiterhin unverändert und uneingeschränkt nutzbar. Damit konnte die Anforderung 3.1.2 vollständig erfüllt werden.

6.1.3 Paginierung

Durch die Verwendung von Links auf Dokumentenebene (vgl. 5.1.2) konnte Anforderung 3.1.3 vollständig erfüllt werden.
Abbildung 6.2 zeigt die (gekürzte) Antwort des ODS auf die Anfrage
`.../v2/datasources/pegelonline/data?limit=2&offset=10`.

```
{
    "links": {
        "datasources": "http://localhost:8080/ods/api/v2/datasources",
        "self": "http://localhost:8080/ods/api/v2",
        "doc": "http://localhost:8080/ods/api/v2/doc",
        "filterTypes": "http://localhost:8080/ods/api/v2/filterTypes",
        "users": "http://localhost:8080/ods/api/v2/users"
    },
    "data": {
        "id": "jValue_OpenDataService",
        "type": "EntryPoint",
        "attributes": {
            "API_version": "2.0.0",
            "ODS_version": {
                "version": "0.2.2",
                "build":
                    "https://github.com/jvalue/open-data-service/commit/9ae79c0fb72a"
            }
        }
    }
}
```


Abbildung 6.1: Json Repräsentation des EntryPoints zur ODS API

```
{
    "links": {
        "next":
        "http://localhost:8080/ods/api/v2/datasources/pegelonline/
        data?limit=2&offset=12",
        "datasource":
        "http://localhost:8080/ods/api/v2/datasources/pegelonline/",
        "self":
        "http://localhost:8080/ods/api/v2/datasources/pegelonline/data?limit=2&offset=10",
        "first":
        "http://localhost:8080/ods/api/v2/datasources/pegelonline/data?limit=2"
    },
    "data": [
        ...
    ]
}
```

Abbildung 6.2: Auszug einer Antwort der ODS API auf einen Data-Request

An der Anfrage ist erkennbar, dass durch den Parameter `limit` die Größe, sowie durch den Parameter `offset` die Position des angefragten Datensatzes spezifizierbar ist. Aus der Abbildung geht zudem hervor, dass die Antwort des ODS Links zum nächsten, ersten sowie aktuellen Datensatz enthält.

6.1.4 Dokumentation

Die Anforderung 3.1.4 konnte nicht vollständig erfüllt werden. Eine Dokumentation liegt vor und ist über einen eigenen Endpunkt durch Swagger-UI exponiert. Sie enthält zwar Beschreibungen aller Endpunkte, aller ausführbaren Methoden sowie aller gesendeten und empfangenen Datenschemata, doch die für den Aufruf der Methoden nötige Authentifizierung konnte in der gegebenen Zeit nicht implementiert werden.
Insgesamt ergeben sich folgende Ansatzpunkte für eine spätere Verbesserung der Dokumentation, von denen allerdings nur der erste einen Verstoß gegen die oben spezifizierten Implementierungsanforderungen darstellt:

- Die **Authentifizierung**, die für die Verwendung einiger Methoden notwendig ist, wurde noch nicht in die Dokumentation aufgenommen. Das Wissen über die zur Nutzung der Methoden nötigen Authentifizierung ist für die Nutzer der API des ODS ist eine unverzichtbare Information, weshalb sie im Rahmen der weiteren Arbeit am ODS so bald wie möglich nachgetragen

werden sollte.

- **Links und Includes** tauchen in den Schemata der ausgetauschten Daten noch nicht auf.
  Ob es sich hierbei tatsächlich um einen Mangel handelt, ist allerdings fraglich, denn die Ansätze von HATEOAS und einer vollständigen Swagger Dokumentation sind komplementär zueinander. Konzeptionell zielt HATEOAS darauf, dass eine API ohne jedes Vorwissen betreten und verwendet werden kann – eine Dokumentation ist hier streng genommen also gar nicht mehr nötig. Umgekehrt werden bei einer vollständig dokumentierten API Querverweise zwischen den Endpunkten nicht mehr unbedingt benötigt, da die Nutzer das Wissen über Verwandte Ressourcen ja bereits aus der Dokumentation erhalten haben.
- einige **Schemadefinitionen** sind möglicherweise noch unvollständig oder fehlerhaft. Mir hat an manchen Stellen schlicht ein tiefes Wissen um die innere Logik der Prozesse des ODS gefehlt. Im Prozess der weiteren Verbesserung der Dokumentation sollten die Schemadefinitionen immer wieder angepasst werden – optimalerweise in Rücksprache mit anderen an der Entwicklung des ODS Beteiligten, die möglicherweise tiefere Kenntnisse über das nötige Datenformat besitzen.

6.1.5 Verwendung von Tests

Wie in Tabelle 6.1 erkennbar ist, wurde die in Anforderung 3.1.5 geforderte Abdeckung von 85% sogar übertroffen – bei den Methoden um 5% und bei den Zeilen sogar um 10%. Leider gelang es in der zur Verfügung stehenden Zeit nicht, automatische Integrationstest für die API v2 umzusetzen, denn einige der verwendeten Annotationen scheinen mit dem vorhandenen Integrationstest-Framework nicht kompatibel zu sein.
Die Implementierung funktionierender Integrationstest stellt sich damit als wichtiger Anknüpfungspunkt der weiteren Arbeit am ODS heraus. Anforderung 3.1.5 wurde folglich leider nur teilweise erfüllt.

6.2 Nichtfunktionale Anforderungen

6.2.1 State of the Art Implementierung

Alle verwendeten Frameworks und Tools besitzen eine aktive Community, werden aktiv gepflegt und sind auf dem aktuellen Stand der Technik.

Klasse	Testcases	Method, %	Line, %
JsonApiDocument	6	94%(16/17)	96%(62/64)
JsonApiRelationship	4	85%(6/7)	95%(19/20)
JsonApiRequest	4	33%(2/6)	73%(11/15)
JsonApiResource	6	93%(15/16)	97%(38/39)
JsonApiResourceIdentifier	2	83%(5/6)	93%(15/16)
JsonApiResponse	13	100%(16/16)	100%(51/51)
JsonLinks	n.a.	100%(2/2)	100%(3/3)
JsonUtils	2	100%(4/4)	100%(11/11)
HttpUtils	5	100%(4/4)	82%(19/23)
RequestValidator	2	100%(2/2)	100%(15/15)
Gesamt	44	90%(72/80)	95%(244/257)

Tabelle 6.1: Code Coverage der Unit Tests

- In besonderem Maße gilt das für **Swagger**, das mit seinem Ökosystem eine enorme Flexibilität bietet und aktuell einen Quasi-Standard (vgl. Kieselhorst, 2018) der API-Dokumentation darstellt.
- Auch **Dropwizard** wird weiterhin gut gepflegt und findet breite Anwendung. Hier muss allerdings einschränkend angemerkt werden, dass Spring Boot in Sachen Verbreitung und Kompatibilität mit Swagger leichte Vorteile besitzt. Da Dropwizard sich in der Entwicklung des ODS bislang aber sehr gut bewährt hat, wurde von einem Umstieg abgesehen.
- Im Fall von **JSON API** muss sich in den nächsten Jahren noch zeigen, ob es sich gegen seine Mitbewerber wie HAL oder JSON-LD behaupten kann. Aktuell gehört es auf jeden Fall zu den am weitesten verbreiteten JSON Spezifikationen. Da es unter den in Abschnitt 5.1.2 beschriebenen Alternativen die jüngste Spezifikation ist, ist zudem zu erwarten, dass es seine Popularität noch weiter steigern kann.

Ein Kritikpunkt bei der Umsetzung dieser Anforderung könnte es sein, dass für die Verwendung von JSON API zwar Frameworks existieren, auf diese aber nicht zurückgegriffen wurde. Dieses Vorgehen erklärt sich daraus, dass die Anwendung eines der beschriebenen Frameworks zwangsweise einen größeren Eingriff in die bestehende Codebasis nach sich gezogen hätte und damit in Konflikt mit Anforderung 3.2.2 steht.

6.2.2 Minimalinvasive Implementierung

Eingriffe in die bestehende Codebasis konnten nicht vollständig vermieden, aber doch minimal gehalten werden. Tatsächlich wurden nur zwei Eingriffe an den

Modellklassen mit minimalen Seiteneffekten vorgenommen:

- Bei einigen Modellklassen musste der `final`-Modifikator entfernt werden, um es zu ermöglichen, eigene Klassen von ihnen abzuleiten.
- Einige Modellklassen wurden um Swagger-Annotationen erweitert um Beispielwerte für die Dokumentation zu generieren.

7 Fazit

7.1 Rückblick und Ergebnisse

Der Open-Open-Data-Service wurde ins Leben gerufen, um für die Aufbereitung und Veröffentlichung öffentlich zugänglicher Daten zu sorgen, die zwar in großem Umfang zur Verfügung stehen, hinsichtlich ihrer Datenqualität und der Einheitlichkeit ihrer Formate aber oft große Mängel aufweisen.
Ziel der vorliegenden Arbeit war das Design und die Implementierung einer API, die den Qualitätsansprüchen des Architekturstils REST entspricht.
Als konzeptionelle Vorarbeit wurde zu diesem Zweck zunächst eine Analyse des Architekturstils REST, wie er im Jahr 2000 von Roy Fielding eingeführt wurde, angestellt (2.2).
Als Ergebnis hat diese Analyse insgesamt acht Bedingungen zu Tage gefördert, denen Softwarearchitekturen genügen müssen, um als RESTful im Sinne Fieldings bezeichnet zu werden (2.3.7). Das dabei entwickelte, genauere Verständnis des Architekturstils wurde daraufhin mit zwei weiteren verbreiteten Paradigmen des API Designs (RPC und GraphQL) kontrastiert (2.4).
Vor diesem theoretischen Hintergrund konnte die Arbeit an Entwurf und Implementierung der API beginnen.
Nach einer genauen Spezifikation der Anforderungen sowie deren Evaluationsschema (3), wurde eine grobe Bestandsaufnahme der bisherigen Architektur vorgenommen (4).
Die Konzeption der API beinhaltete zunächst die Auswahl der Verwendeten Technologie. Zu diesem Zweck wurden in einem ersten Schritt die wichtigsten bereits verwendeten Technologien vorgestellt (5.1.1). Im zweiten Schritt konnten dann die Technologien ausgewählt werden, die bei der Implementierung verwendet werden sollen. Das beinhaltete jeweils auch die Erstellung einer kurzen Übersicht über Alternativen sowie die Begründung der tatsächlich getroffenen Auswahl.
Zur Festlegung des Formats der über die API ausgetauschten Daten wurde die JSON API Spezifikation verwendet, da sie hervorragende Möglichkeiten zur Umsetzung der REST-Bedingungen, insbesondere HATEOAS, bietet (5.1.2). Bei der Erstellung der Spezifikation hat sich, auch nach Sichtung möglicher Alternati-

ven, die bereits in den Anforderungen getroffene Wahl, Swagger zu verwenden, als vorteilhaft heraus gestellt (5.1.2). Swagger ist aktuell mit Abstand die populärste Lösung in der Dokumentation von APIs und besitzt ein umfangreiches Ökosystem von Anwendungen und Werkzeugen.
Durch die ausgewählte Technologie waren viele die weitere Konzeption der zu implementierenden API betreffenden Entscheidungen bereits vorgezeichnet.
So konnte etwa beim Design der Datenrepräsentation (5.2.4) auf die Möglichkeiten zurückgegriffen werden, die JSON API bietet. Wesentliches Ergebnis der hier getroffenen Erwägungun war die Entscheidung, die von HATEOAS geforderte Verlinkung der API Endpunkte untereinander auf Ebene der HTTP Message-Bodies zu realisieren.
Die generische Erstellung und Exposition der Dokumentation (5.2.3) war zu großen Teilen von der Funktionsweise der Anwendung Swagger-UI und der Bibliothek swagger-core bestimmt. Weitere wesentliche Punkte des API Entwurfs waren das Design der Endpunkte (5.2.2) und die Versionierung (5.2.1).
Nach Abschluss des konzeptuellen Entwurfs konnte die tatsächliche Implementierung der API begonnen werden. Zentrale Herausforderungen dabei bestanden in der Implementierung von Response- und Request-Klassen, die die Einhaltung der JSON API Spezifikation sicherstellen (5.3.1 und 5.9), sowie in der Erstellung der Dokumentation aus dem Quellcode anhand von Swagger-Annotationen (5.3.3).
Der letzte Schritt in der Anfertigung der vorliegenden Arbeit bestand schließlich darin, die erstellte Software anhand der vorher definierten Anforderungen und deren Evaluationsschema zu bewerten (6).
Als Ergebnis dieser Evaluation ergab sich, dass von den sechs an die Implementierung gestellten Anforderungen vier erfüllt, und zwei nicht oder nur teilweise erfüllt werden konnten.
Insgesamt haben sich einige Verbesserungen des ODS ergeben:

- Durch die gelungene Implementierung der JSON API Spezifikation wurde ein einheitliches, extern dokumentiertes Datenformat erreicht. Nutzer des ODS müssen sich daher nicht speziell mit dem Datenformat des ODS befassen, sondern können sich auf die Einhaltung eines verbindlichen Vertrags verlassen und so die vom ODS gelieferten Daten sehr viel einfacher interpretieren. Zudem existieren einige Bibliotheken und Frameworks, mit deren Hilfe ein Client für Server, die sich an die JSON API Spezifikation halten, leicht implementiert werden kann[1].

- Die Dokumentation durch Swagger bietet neben ihrer graphisch ansprechenden Präsentation einen weiteren Vorteil: Da sich der Quellcode der API und die für die Generierung der Dokumentation zuständigen Annotationen an der gleichen Stelle befinden, wird es auch bei zukünftigen Veränderungen der API sehr viel einfacher sein, die Dokumentation aktuell zu halten.

[1] Eine ausführliche Liste ist abrufbar unter: http://jsonapi.org/implementations/.

- Die konsequente Umsetzung des HATEOAS Constraints ermöglicht die Navigation durch die API auch ohne Kenntnis der Dokumentation. Dadurch, dass jede Nachricht Querverweise auf verwandte Ressourcen enthält, lässt sich mit sehr geringem Aufwand eine graphische Web-Oberfläche erstellen, über die die Daten des ODS in menschenlesbarer Form präsentiert werden können.

Deutlich wurde aber auch, dass trotz der erreichten Verbesserungen noch ein großes Entwicklungspotential besteht.
Auch wenn nicht alle Anforderungen vollständig erfüllt wurden, konnte im Rahmen dieser Arbeit eine solide Basis dafür gelegt werden, den ODS weiter zu verbessern, damit er dazu beitragen kann, das volle Potential offener Daten auszuschöpfen.

7.2 Ausblick

7.2.1 Ausbesserungen

Den ersten Ansatzpunkt für die weitere Entwicklung des ODS stellt natürlich die Umsetzung der Anforderungen dar, die nicht vollständig realisiert werden konnten.

- Für die API v2 müssen **Integrationstests** implementiert werden oder das bestehende Integrationstest Framework muss so angepasst werden, dass die neue API ebenfalls getestet werden kann.
- Die **Dokumentation** muss angepasst und erweitert werden: Höchste Priorität hat dabei die Dokumentierung der Authentifizierung. Darüber hinaus sollten aber auch bestehende Datenschemata überprüft und gegebenenfalls angepasst werden und es sollte noch einmal untersucht werden, welchen Mehrwert eine Integration der Querverweise und inkludierten Objekte in die Schemata bringen kann.

7.2.2 Erweiterung der Funktionalität

GraphQL

In Abschnitt 2.4.2 wurde im Kontext des Vergleichs zwischen REST und GraphQL die Möglichkeit ins Auge gefasst, zu einem späteren Zeitpunkt zusätzlich zu der bestehenden REST-API einen Endpunkt zu integrieren, über den GraphQL

Anfragen möglich sind. Um zu eruieren, ob sich der Aufwand dieser Erweiterung auszahlt müssen im Wesentlichen drei Fragen beantwortet werden:

- Gibt es von Seiten der potentiellen Nutzer des ODS ein Interesse, auf die von dem ODS bereitgestellten Daten über eine GraphQL API zuzugreifen?
- Mit welchem Refactoring-Aufwand wäre die Einführung der GraphQL API verbunden? Wie gut lässt sich beispielsweise eine Graphdatenbank in die bestehende Struktur des ODS integrieren?
- Ist es absehnbar, dass der ODS Datenquellen verwendet, deren Daten eine so vernetzte Struktur haben, dass sich die Nutzung von GraphQL lohnen würde?

Optionale Inklusion verwandter Objekte

In Kapitel 5.1.2 wurden die von JSON API definierten Relationship-Objekte vorgestellt. Außerdem wurde gezeigt, dass die JSON API Spezifikation die Möglichkeit bietet, dass eine Antwort neben dem primären angefragten Entität weitere Entitäten, *inkludierte Objekte* enthalten kann. Die im Rahmen dieser Arbeit implementierte API nutzt diese Möglichkeit genau dann, wenn die Antwort eine Relationship-Objekt mit Verweis auf die betreffende Entität enthält.
Eine sinnvolle Erweiterung der API könnte darin bestehen, Clients selbst darüber entscheiden zu lassen, welche Objekte in der Antwort enthalten sein sollen, indem sie über Parameter eine Liste inkludierter Objekte übergeben. Die Anzahl der Anfragen könnte so möglicherweise stark reduziert werden.

Graphische Oberfläche für den ODS

Eine graphische Oberfläche könnte die Attraktivität des ODS, gerade für Nichtinformatiker, enorm erhöhen. Wer entscheiden muss, aus welcher Quelle die Daten für eine neue Anwendung bezogen werden sollen, selbst aber kein Programmierer ist, lässt sich mit einer graphischen Oberfläche sicherlich besser von den Qualitäten des ODS überzeugen, als von der auf Maschinenlesbarkeit ausgelegten JSON API Repräsentation. Die Umsetzung eines einfachen graphischen Webinterfaces sollte, wie oben beschrieben, aufgrund der konsequenten Umsetzung von HATEOAS mit sehr wenig Aufwand durchführbar sein.

7.2.3 Konzeptuelle Weiterentwicklung

Hinzufügen von Datenquellen

Der ODS erlaubt es Clients, über die API selbst Datenquellen hinzuzufügen sowie die Regeln für den Import und die Weiterverarbeitung der Daten zu definieren. Um diese Funktionalität zu nutzen sind allerdings ziemlich detaillierte Kenntnisse der internen Funktionsweise des ODS nötig. Zwar hat theoretisch jeder Nutzer die Möglichkeit, sich diese Kenntnisse durch das Studium des Quelltexts oder der Postman-Beispiele zu erwerben. Um eine große Auswahl an verfügbaren Datenquellen zu ermöglichen, wäre es aber sehr sinnvoll, eine detaillierte Anleitung zu erstellen, wie neue Datenquellen in den ODS eingebunden werden können.
Es bietet sich zudem an, vor der Erstellung dieser Anleitung das bestehende System noch einmal zu überprüfen und gegebenenfalls anzupassen, beziehungsweise zu vereinfachen. Beispielsweise müssen Clients nach der jetzigen Funktionsweise bei Erstellung einer Datenquelle den Namen der vom ODS intern verwendeten ID angeben. Die Übertragung dieser Aufgabe in den Zuständigkeitsbereich des ODS würde das Anlegen neuer Datenquellen bereits etwas vereinfachen.
Optimalerweise könnte so erreicht werden, dass dem ODS nur noch die URI übergeben werden muss, über die die Daten zu beziehen sind, während alle anderen Parameter – wie etwa der Name der Datenquelle, das Datenformat oder das Attribut, was die ID enthält – automatisch vom ODS generiert werden.

Value Object Pattern

Die ursprüngliche Entwurf des ODS entstand im Rahmen des *JValue*-Frameworks (vgl. Riehle, 2006). Das dort definierte *Value Object Pattern* bietet Möglichkeiten zur Konsistenzprüfung der von den Datenquellen stammenden Datensätze.
Aufgrund der generischen Natur der Datensätze wurde in der weiteren Entwicklung allerdings doch von der Verwendung des Patterns abgesehen (vgl. Reischl, 2014, S. 2).
Zu ermitteln, ob das Value Object Pattern dahingehend angepasst werden kann, dass es auch auf generische Datentypen angewendet werden kann, könnte ein lohnender Ausgangspunkt für weitergehende Forschungen sein.

8 Anhänge

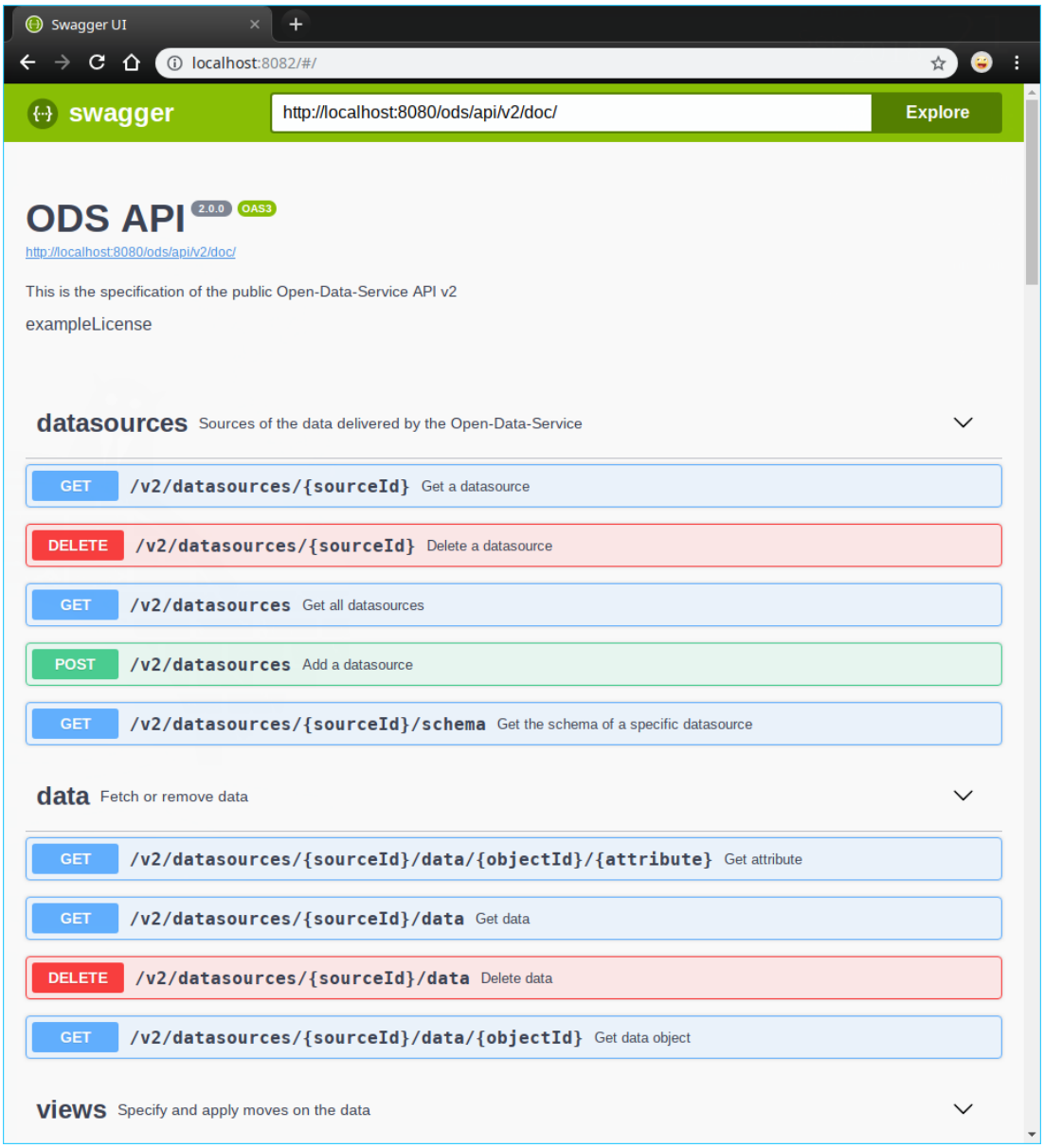
Anhang A Ausführliches JSON API Beispieldokument

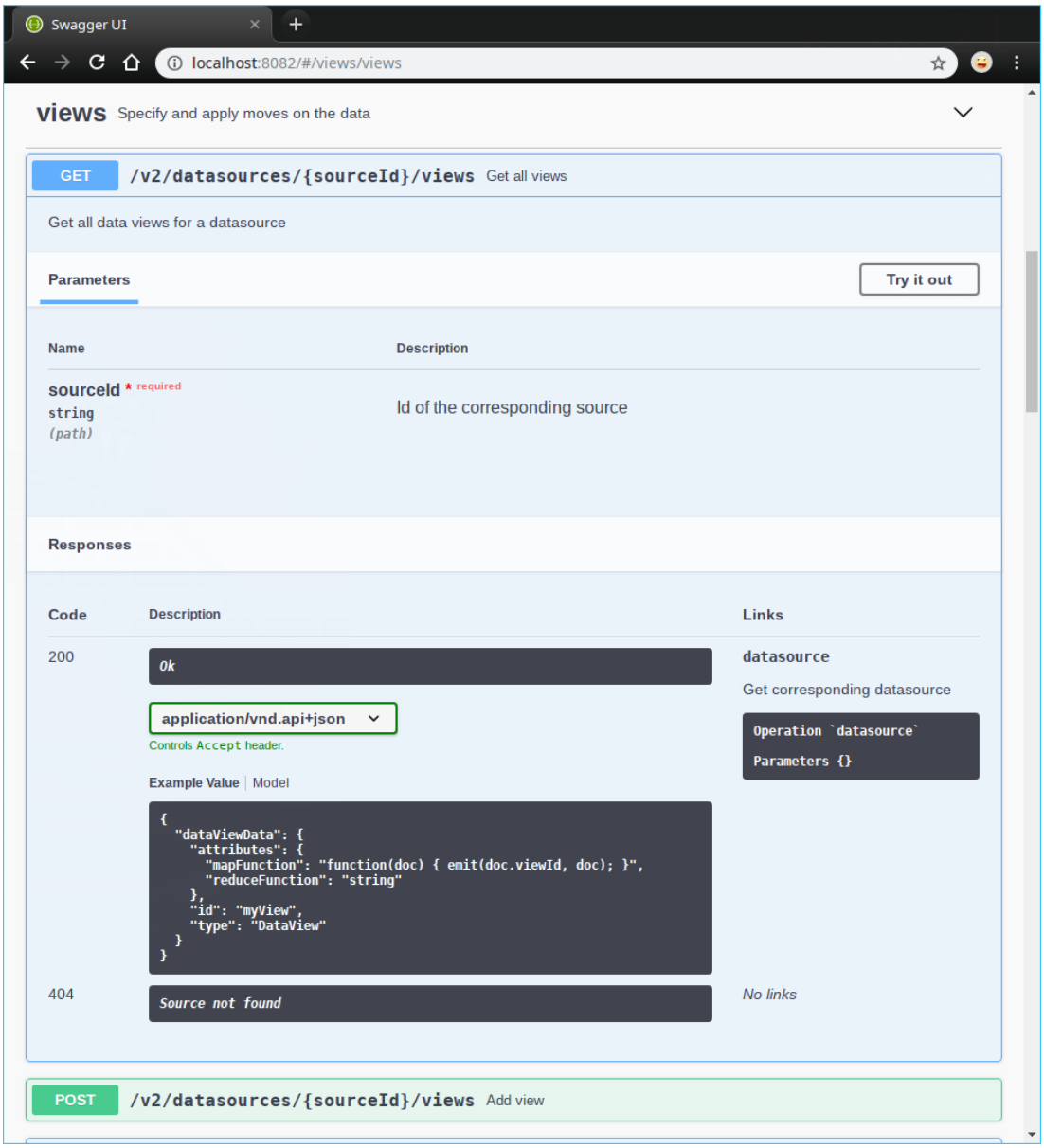
```
{
  "links": {
    "self": "http://example.com/articles",
    "next": "http://example.com/articles?page[offset]=2",
    "last": "http://example.com/articles?page[offset]=10"
  },
  "data": [{
    "type": "articles",
    "id": "1",
    "attributes": {
      "title": "JSON API paints my bikeshed!"
    },
    "relationships": {
      "author": {
        "links": {
          "self": "http://example.com/articles/1/relationships/author",
          "related": "http://example.com/articles/1/author"
        },
        "data": { "type": "people", "id": "9" }
      },
      "comments": {
        "links": {
          "self":
              "http://example.com/articles/1/relationships/comments",
          "related": "http://example.com/articles/1/comments"
        },
        "data": [
          { "type": "comments", "id": "5" },
        ]
      }
    },
    "links": {
      "self": "http://example.com/articles/1"
    }
  }],
  "included": [{
    "type": "people",
    "id": "9",
    "attributes": {
      "first-name": "Dan",
      "last-name": "Gebhardt",
      "twitter": "dgeb"
    },
    "links": {
      "self": "http://example.com/people/9"
    }
  }, {
    "type": "comments",
    "id": "5",
    "attributes": {
      "body": "First!"
    },
    "relationships": {
```

Anhang B EBNF Darstellung von (Teilen der) JSONAPI Syntax

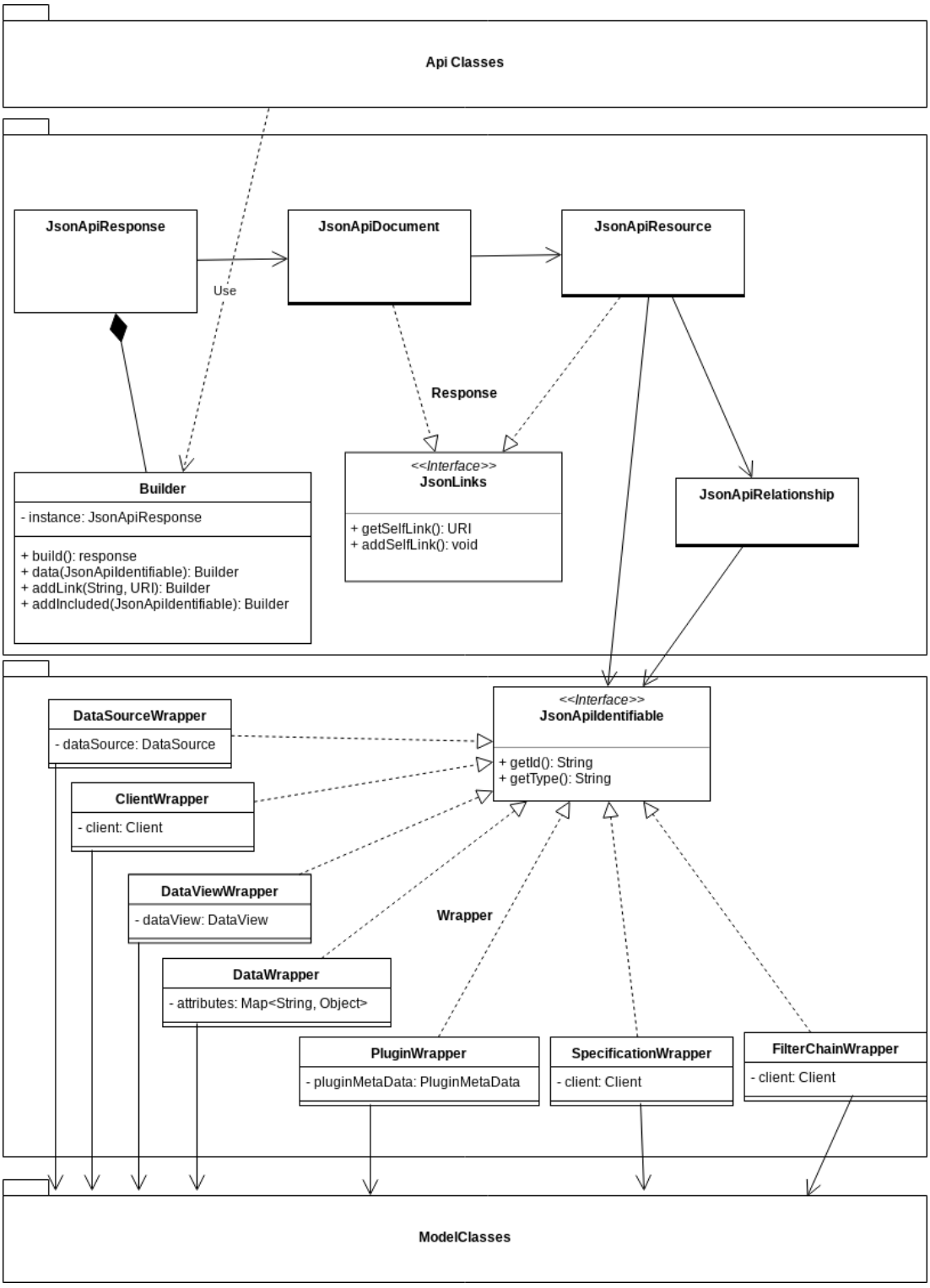
```
Document :=
'{'
  data | error | meta,
  [jsonapi],
  [links],
  [included]
'}';

links :=
'"links":{'
  {'"name":' String}
'}';

data :=
'"data":{'
  resourceObject | '[' {resourceObject} ']'
'}';

resourceObject :=
{
  '"type":' String,
  '"id":' String,
  [attributes],
  [relationships],
  [links].
  [meta];
}

attributes :=
'"attributes:{'
  {String':' String | Integer | Float}
'}';

included :=
'"included":{['
    { resourceObject }
']}';
```

Anhang C Swagger-UI Benutzeroberfläche




Swagger UI
localhost:8082/#/views/views
views
Specify and apply moves on the data
GET
/v2/datasources/{sourceId}/views
Get all views
Get all data views for a datasource
Parameters
Try it out
Name
Description
sourceId * required
string
(path)
Id of the corresponding source
Responses
Code
Description
Links
200
Ok
application/vnd.api+json
Controls Accept header.
Example Value | Model
{
"dataViewData": {
"attributes": {
"mapFunction": "function(doc) { emit(doc.viewId, doc); }",
"reduceFunction": "string"
},
"id": "myView",
"type": "DataView"
}
}
datasource
Get corresponding datasource
Operation `datasource`
Parameters {}
404
Source not found
No links
POST
/v2/datasources/{sourceId}/views
Add view

Anhang D Klassendiagramm der Response Klassen



Anhang E Quellcode der Requestvalidierung und -konversion

```
public static <T> T convertValue(Object fromValue, Class<T> toValueType) {
    T result;
    try {
        result = getInstance().convertValue(fromValue, toValueType);
    } catch (IllegalArgumentException e) {
        throw RestUtils.createJsonFormattedException("Malformed "
            + toValueType.getSimpleName()
            + ": "
            + e.getMessage(),
            400);
    }

    return result;
}
```

JsonMapper

```
public static <T> void validate(T obj) {
    Set<ConstraintViolation<T>> violations = getInstance().validate(obj);

    if (!violations.isEmpty()) {
        StringBuilder violationStringBuilder = new StringBuilder();
        for (ConstraintViolation<T> violation : violations) {
            violationStringBuilder
                .append(violation.getPropertyPath().toString())
                .append(" ")
                .append(violation.getMessage())
                .append(System.lineSeparator());
        }
        throw RestUtils.createJsonFormattedException("Malformed "
            + obj.getClass().getSimpleName()
            + ": "
            + violationStringBuilder.toString(),
            400);
    }
}
```

RequestValidator

Anhang F Klassen zur Erzeugung der Swagger Schemata

```java
@Schema(name = "JsonApiDocument")
public class JsonApiSchema {

    public abstract class ProcessorSpecificationSchema extends JsonApiSchema {
        @Schema(required = true)
        public abstract SpecificationWrapper getData();
    }

    public abstract class ClientSchema extends JsonApiSchema {
        @Schema(required = true)
        public abstract ClientWrapper getData();
    }

    public abstract class DataSourceSchema extends JsonApiSchema {
        @Schema(required = true)
        public abstract DataSourceWrapper getData();
    }

    public abstract class DataViewSchema extends JsonApiSchema {
        @Schema(required = true)
        public abstract DataViewWrapper getData();
    }

    public abstract class EntryPointSchema extends JsonApiSchema {
        @Schema(required = true)
        public abstract EntryPoint.EntryPointData getData();
    }

    public abstract class PluginMetaDataSchema extends JsonApiSchema {
        @Schema(required = true)
        public abstract PluginMetaDataWrapper getData();
    }

    public abstract class ProcessorReferenceChainSchema extends JsonApiSchema {
        @Schema(required = true)
        public abstract ProcessorReferenceChainWrapper getData();
    }

    public abstract class UserSchema extends JsonApiSchema {
        @Schema(required = true)
        public abstract UserWrapper getData();
    }

    public abstract class DataSchema extends JsonApiSchema {
        @Schema(required = true)
        public abstract DataWrapper getData();
    }

    public abstract class DataCollectionSchema extends JsonApiSchema {
        @ArraySchema(schema = @Schema(implementation = DataWrapper.class, required = true))
        public abstract DataWrapper[] getData();
    }
}
```

Literaturverzeichnis

Anonymous. (o.D.). about swagger. Zugriff unter https://swagger.io/about/. (Zuletzt abgerufen am 19.09.2018)

Berners-Lee, T., Fielding, R. et al. (2005). *RFC 3986: Uniform Resource Identifier (URI)*. Zugriff unter https://tools.ietf.org/html/rfc3986

Buna, S. (2016). Learning GraphQL and Relay: build data-driven React applications with ease using GraphQL and Relay. (Kap. RESTful APIs versus GraphQL APIs). Packt Publishing.

Dallas, A. (2014). *RESTful Web Services with Dropwizard*. Packt Publishing.

Dropwizard Manual. (2018). Dropwizard Team. Zugriff unter https://www.dropwizard.io/1.3.5/docs/manual/index.html. (zuletzt abgerufen am 12.09.2018)

Fielding, R. (2000). *Architectural Styles and the Design of Network-based Software Architectures* (Diss., University of California, Irvine).

Fielding, R. (2008). REST APIs must be hypertext-driven. Blog. Zugriff unter http://roy.gbiv.com/untangled/2008/rest-apis-must-be-hypertext-driven. (Zuletzt abgerufen am 19.07.2018)

Fielding, R. et al. (1999). *RFC 2616: HTTP/1.1*. Zugriff unter https://www.ietf.org/rfc/rfc2616.txt

Fielding, R. et al. (2014). *RFC 7234: HTTP/1.1: Caching*. Zugriff unter https://www.rfc-editor.org/rfc/rfc7234.txt

Fowler, M. (2005). FluentInterface. Blog. Zugriff unter https://martinfowler.com/bliki/FluentInterface.html. (Zuletzt abgerufen am 19.09.2018)

Garlan, D. & Shaw, M. (2014). *An Introduction to Software Architecture*. An introduction to software architecture. School of Computer Science, Carnegie Mellon University. Zugriff unter https://books.google.de/books?id=iBGvPgAACAAJ. (Zitiert nach [Tsysin, 2014])

GraphQL Specification. (2018). Facebook Inc. Zugriff unter http://facebook.github.io/graphql/June2018/. (zuletzt abgerufen am 12.09.2018)

Haldar, M. (2017). RESTful API Design Guidelines. Blog. Zugriff unter https://hackernoon.com/restful-api-designing-guidelines-the-best-practices-60e1d954e7c9. (Zuletzt abgerufen am 20.09.2018)

Hedtstück, U. (2012). Einführung in die theoretische Informatik: formale Sprachen und Automatentheorie. (Kap. Erweiterte Backus-Naur-Form, S. 39–41). Oldenbourg Verlag.

Katz, Y., Klabnik, S., Gebhardt, D., Kellen, T. & Resnick, E. (2015). *JSON API Specification*. Zugriff unter http://jsonapi.org/format/1.0. (Version 1.0)

Khare, N. (2015). *Docker Cookbook*. Packt Publishing.

Kieselhorst, M. (2018). Swagger: Mehr als nur Schnittstellenbeschreibung. Blog. Zugriff unter https://entwickler.de/online/web/openapi-swagger-579827368.html. (Zuletzt abgerufen am 21.09.2018)

Korando, B. (2016). Don't Hate The Hateoas. Blog. Zugriff unter https://keyholesoftware.com/2016/02/29/dont-hate-the-hateoas/. (Zuletzt abgerufen am 21.09.2018)

Negus, C. (2015). Docker Containers: Build and Deploy with Kubernetes, Flannel, Cockpit and Atomic. (Kap. Containerizing Applications with Docker). Prentice Hall.

Overview of RESTful API Description Languages. (2017). Zugriff unter https://en.wikipedia.org/wiki/Overview_of_RESTful_API_Description_Languages. (Zuletzt abgerufen am 20.09.2018)

Rajput, D. (2018). Mastering Spring Boot 2.0: Build modern, cloud-native, and distributed systems using Spring Boot. (Kap. API Management, S. 283–314). Packt Publishing.

Reischl, P. (2014). *Improving Data Quality using Domain-Specific Data Types* (Masterarbeit, Friedrich-Alexander-Universität Erlangen-Nürnberg). Zugriff unter https://osr.cs.fau.de/2014/10/13/final-thesis-improving-data-quality-using-domain-specific-data-types-in-german/. (Zuletzt abgerufen am 24.09.2018)

Reiser, A. (2018). Why HATEOAS is useless and what that means for REST. Blog. Zugriff unter https://medium.com/@%22andreasreiser94/why-hateoas-is-useless-and-what-that-means-for-rest-a65194471bc8. (Zuletzt abgerufen am 21.09.2018)

Richardson, L. (2007). RESTful web services. (Kap. The Programmable Web and Its Inhabitants). O'Reilly.

Riehle, D. (2006). Value object. *Proceedings of the 2006 conference on Pattern languages of programs*, 30. (Zitiert nach [Reischl, 2014])

Sadalage, P. J. (2013). NoSQL distilled: a brief guide to the emerging world of polyglot persistence. (Kap. Why NoSQL, S. 3–13). Addison-Weasley.

Sandoval, K. (2016). Top Specification Formats for REST APIs. Blog. Zugriff unter https://nordicapis.com/top-specification-formats-for-rest-apis/. (Zuletzt abgerufen am 20.09.2018)

Snell, J., Tidwell, D. & Kulchenko, P. (2001). *Programming Web Services with SOAP*. O'Reilly Media, inc.

Sobocinski, P. (2014). Hypermedia API: The Benefits of HATEOAS. Blog. Zugriff unter https://www.programmableweb.com/news/hypermedia-apis-benefits-hateoas/how-to/2014/02/27. (Zuletzt abgerufen am 20.09.2018)

Sookecheff, K. (2014). On choosing a hypermedia type for your API - HAL, JSON-LD, Collection+JSON, SIREN, Oh My! Blog. Zugriff unter https://sookocheff.com/post/api/on-choosing-a-hypermedia-format/. (Zuletzt abgerufen am 19.09.2018)

Stowe, M. (2015). API Best Practices: Hypermedia. Blog. Zugriff unter https://blogs.mulesoft.com/dev/api-dev/api-best-practices-hypermedia-part-3/. (Zuletzt abgerufen am 19.09.2018)

Sturgeon, P. (2016). Understanding RPC Vs REST for HTTP APIs. *Smashing Magazine*, (September 20). Zugriff unter https://www.smashingmagazine.com/2016/09/understanding-rest-and-rpc-for-http-apis/. (Zuletzt abgerufen am 09.09.2018)

Tilkov, S. (2015). *REST und HTTP: Entwicklung und Integration nach dem Architekturstil des Web*. dpunkt Verlag.

Trieloff, L. (2017). Three Principles of API First Design. Blog. Zugriff unter https://medium.com/adobetech/three-principles-of-api-first-design-fa6666d9f694. (Zuletzt abgerufen am 20.09.2018)

Tsysin, K. (2014). *Design of a Reflective REST-based Query API* (Masterarbeit, Friedrich-Alexander-Universität Erlangen-Nürnberg). Zugriff unter https://osr.cs.fau.de/2015/01/26/final-thesis-definition-of-a-domain-specific-language-using-qualitative-data-analysis-in-german/%20[sic!]

Vasudevan, K. (2017). Design First or Code First: What's the Best Approach to API Development? Blog. Zugriff unter https://swagger.io/blog/api-design/design-first-or-code-first-api-development/. (Zuletzt abgerufen am 20.09.2018)