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

Konstantin Tsysin

MASTERARBEIT

Design of a Reflective REST-based Query API

Eingereicht am 30.12.2014

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.

_____

Nürnberg, 30.12.2014

License

_____

Nürnberg, 30.12.2014

Abstract

In this thesis a prototypical implementation of the Open Data Service is performed in order to solve the problem of data and information overload. After a reflection and evaluation of related and existing solutions the requirements for the Open Data Service and the reflective REST API were gathered and a conceptual solution has been developed. Thereby quality attributes for a well-developed interface were elaborated and the benefits of metadata were illustrated on the example of the metaobject protocol. These two aspects have been incorporated into the design of the Open Data Service and the reflective API. The service makes use of publicly available data sources, processes them and makes them available through a uniform interface. This facilitates easy access to freely available and usable data from heterogeneous data sources. A major emphasis of this thesis has been put on the design and implementation of the reflective REST-API, taking into account the data structures and metadata.

Zusammenfassung

In dieser Arbeit wird die prototypische Umsetzung des Open Data Service vollzogen, um die Probleme der Daten- und Informationsflut zu lösen. Nach einer Betrachtung und Evaluation verwandter und bereits existierender Lösungen werden die Anforderungen an den Open Data Service und an die reflektierende REST-API erfasst sowie eine konzeptuelle Lösung ausgearbeitet. Dabei wurden die Qualitätsmerkmale für eine gute Schnittstelle ausgearbeitet und der Nutzen der Metadaten am Beispiel des Metaobjektprotokolls veranschaulicht. Diese beiden Themenaspekte flossen in die Konzeption des Open Data Service und der reflektierenden Schnittstelle mit ein. Der Dienst bedient sich dabei öffentlich zur Verfügung stehender Datenquellen, bereitet diese auf und stellt sie über eine einheitliche Schnittstelle zur Verfügung. Diese ermöglicht einen komfortablen Zugriff auf frei verfügbare sowie nutzbare Daten hete-rogener Datenquellen. Einen Schwerpunkt dieser Arbeit bildet die Konzeption und Realisie-rung der reflektierenden auf dem Paradigma REST basierenden Schnittstelle unter Berück-sichtigung der Datenstrukturen und der Metadaten.

Inhaltsverzeichnis

1 Einleitung

Das Wachstum des Internets und damit dessen Bedeutung hat in den letzten Jahren stetig zugenommen. In weiten Teilen Europas, Nordamerikas und Ostasiens ist das Internet mittlerweile ein bedeutender Wirtschaftsfaktor. Ebenso rasant steigt die über das Internet verfügbare Gesamtdatenmenge, die sich in weniger als zwei Jahren schätzungsweise verdoppelt. Die Digitalisierung der Daten bringt jedoch auch Probleme mit sich. Daten liegen oft in unstrukturierten, heterogenen Formaten vor und fehlende Metadaten erschweren zusätzlich die Extraktion von Informationen. So lassen sich derzeit nur fünf Prozent der weltweit verfügbaren Daten über ein Schlagwort suchen. Der Bedarf einer effektiven Datenabfrage steigt somit ebenso. Der Engpass verlagert sich von der reinen Verfügbarkeit der Daten hin zu deren Auswertbarkeit. [1]

Diese genannten Probleme kann man gut am Beispiel der Gewässerdaten in der Bundesrepublik Deutschland erkennen. Die Gewässerdaten werden dezentral in verschiedenen Institutionen der jeweiligen Bundesländer erhoben und liegen in verschiedenen Formaten vor. Obwohl die Daten frei zugänglich sind, gestaltet sich die Auswertung dieser schwer. Die Nutzer müssen die unterschiedlichen Datenquellen durchforsten und die Informationen aus den teilweise schwer analysierbaren Datenbeständen herausfiltern. Die Auswertung über Bundesländer hinweg und die maschinell automatisierte Verarbeitung der Daten gestaltet sich so aufwendig. Dabei birgt die Nutzung der freien Daten ein großes Innovationspotential.[2]

Aus wirtschaftlicher Sicht können freie Daten auf EU-Ebene einen monetären Nutzen von 200 Milliarden Euro ermöglichen. Allerdings ist die Zugänglichkeit eine wichtige Voraussetzung dafür. (Steria, 2012)

1.1 Ziele der Arbeit

In dieser Arbeit wird der „JValue Open Data Service“ (ODS) als Prototyp realisiert. Er versucht die Probleme der Daten- und Informationsflut zu lösen, indem er eine einfache Schnittstelle bereitstellt, welche es ermöglicht, komfortabel auf frei verfügbare sowie nutzbare Daten heterogener Datenquellen zuzugreifen.

Als Datenquellen werden in dieser Arbeit Kartenmaterial von „OpenStreetMap“, einer Datenbank für geographische Daten, wie auch Gewässerdaten von „PEGELONLINE“, einem

[1] http://germany.emc.com/leadership/digital-universe/2014iview/executive-summary.htm
[2] http://www.opendata.bayern.de/ueber.html

gewässerkundlichen Informationssystem der Wasser- und Schifffahrtsverwaltung des Bundes, gewählt.[1]

Um eine einheitliche Schnittstelle zu den PEGELONLINE- und OpenStreetMap-Daten zu ermöglichen, müssen die Quellen angebunden werden. Anschließend werden die unterschiedlich strukturierten Daten eingesammelt, extrahiert, gegebenenfalls in ein einheitliches Format transformiert und in einer Datenbank gespeichert. Schließlich werden diese über eine einfache Schnittstelle öffentlich bereitgestellt.

Die maschinell automatisierte Verarbeitung der Daten und die Möglichkeit der übergreifenden Auswertungen wird durch die Vereinheitlichung des Datenformats, Verbesserung der Datenqualität und dem Anbieten einer zentralen Schnittstelle gewährleistet. Der Nutzer kann so über die standardisierte wohldefinierte Schnittstelle alle aufgenommenen Daten der verschiedenen Datenquellen bequem abrufen. Statt sich mit den Quellen und den Formaten abzumühen, kann er die Informationen direkt aus dem ODS entnehmen, welcher die Daten in einem einheitlichen Format validiert und bereinigt zur Verfügung stellt.

Durch die Vielfalt der angebotenen Daten benötigt der Nutzer eine Möglichkeit Metadaten, einschließlich der Typinformationen über die angebotenen Daten, abfragen zu können. Außerdem bedarf es einer Möglichkeit der Suche oder Filterung für die großen Datenmengen. Deshalb ist ein Schwerpunkt dieser Arbeit die Konzeption und Realisierung der reflektierenden auf dem Paradigma REST basierenden Schnittstelle unter Berücksichtigung der Datenstruktur und der Metadaten.

[1] http://www.pegelonline.wsv.de/ http://www.openstreetmap.de/

2 Forschungsstand und Konzeption

In diesem Kapitel werden die Grundlagenaspekte behandelt und Forschungsansätze vorgestellt. Außerdem werden bereits existierende Lösungen erörtert und evaluiert. Mit Hilfe dieser Ergebnisse werden die Anforderungen an den ODS und an die reflektierende REST-API erfasst sowie eine konzeptuelle Lösung ausgearbeitet.

2.1 Grundlagen

In diesem Abschnitt werden zunächst die grundlegenden Definitionen und Grundbegriffe zum Thema erörtert. Zuerst wird der Begriff Open Data erklärt. Anschließend werden das REST-Paradigma und die JSON-Struktur erläutert.

2.1.1 Open Data

Der Open Data Service wurde konzipiert, um „Open Data“ zu aggregieren, vereinheitlichen und über eine einfache Schnittstelle dem Nutzer bereitzustellen. Unter Open Data versteht man die frei nutzbaren gesammelten Daten (Shadbolt, et al., 2012). Dazu gehören die Geo- und Umweltdaten sowie statistische Daten und weitere zur Verfügung gestellte Daten öffentlicher Behörden. Diese nennt man auch „Open Government Data“ (Maali, Cyganiak, & Peristeras, 2012). Die bereitgestellten Daten können von Nutzern weiterverarbeitet werden. Ausgenommen sind hierbei Daten, die dem Datenschutz oder anderen Einschränkungen unterliegen, wie z.B. Betriebsgeheimnisse oder sicherheitsrelevante Daten.
Eine allgemeine Definition gemäß der Open Knowledge Foundation gibt vor, dass offene Daten vollständig, nur zu ihren Reproduktionskosten für jeden frei verfügbar sein müssen und weiterverarbeitet werden dürfen. Das Modifizieren und die Anfertigung von Derivaten sind erlaubt. Dabei müssen offene Dateiformate verwendet werden. Der Einsatzzweck darf ebenfalls nicht eingeschränkt werden, solange niemand durch die Nutzung diskriminiert wird. (Lucke & Geiger, 2010) Frei zugängliche Daten weisen ein großes Innovationspotential auf. Sie können genutzt oder auf neuartige Weise kombiniert werden und somit zur Erkenntnisgewinnung führen. Sie fördern Potentiale zur Wieder- und Weiterverwendung bestehender Datenbestände und unterstützen die Partizipation, Transparenz und Kollaboration. Dabei können neue Dienste und Anwendungen entstehen.[1]

[1] http://www.opendata.bayern.de/ueber.html

2.1.2 REST

REST steht für Representational State Transfer und ist ein Architekturstil für netzwerkbasierte Anwendungen. REST ist kein Protokoll und weist keine streng definierten Regeln auf. Es definiert nur wie Web Services und Dienste funktionieren sollen. (Burbiel, 2007) Eine REST-Architektur beinhaltet Ressourcen. Jede Ressource hat eine eindeutige Id, welche die Ressource identifiziert. Die Darstellung dieser Ressource nennt man auch Repräsentation. Mit Verben werden die auf dieser Ressource ausführbaren Operationen beschrieben. Sie ermöglichen es beispielsweise, die Ressourceninformationen auszuliefern, zu verändern oder zu löschen.
Web-Technologien bieten alle Voraussetzungen zur erfolgreichen Umsetzung einer REST-Architektur. Folgende Tabelle zeigt die Umsetzung von REST mithilfe von Web-Technologien:

REST	Umsetzung mit Web-Technologie
Eindeutige Id	Uniform Ressource Identifier (URI)
Repräsentation	Medienformat: HTML, XML, JSON, ...
Verben	HTTP-Methoden: GET, PUT, POST, DELETE

Tabelle 1: Umsetzung von REST mit Web-Technologien

REST hat eine Client-Server Architektur. Die Datenhaltung am Server ist getrennt von den Belangen der Darstellung der Informationen. Durch die Standardisierung der Kommunikation über die REST-Schnittstelle wird eine Entkopplung der verwendeten Technologien des Clients und Servers erreicht. Die Kommunikation erfolgt auf Abruf des aktiven Clients vom passiven Server. Jeder Client kann selbst entscheiden wie er die Informationen des Servers nutzt. Die gleichen Daten können für verschiedene Anwendungen verwendet werden. Dabei wird empfohlen die etablierten Standards zu nutzen: Einheitliche Bezeichner (URIs) für die Adressierung und beliebig etablierte Datenformate wie XML, JSON, PNG etc. für die Repräsentationen der Ressourcen.

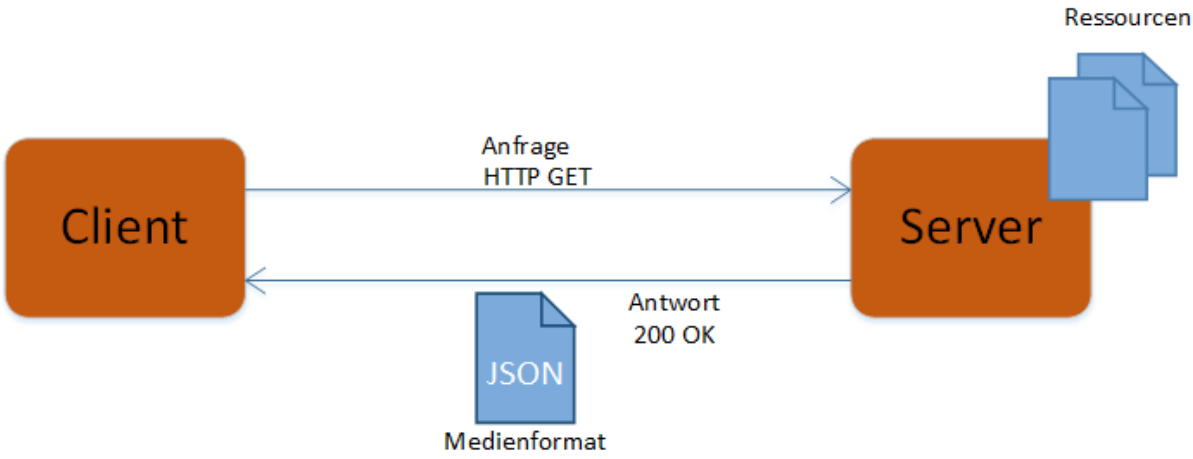

Abbildung 1: Client-Server Architektur

REST ist ein zustandsloses Protokoll. Jede Anfrage – auch desselben Auftraggebers – wird als voneinander unabhängige Transaktion behandelt ohne Bezug zu früheren Anfragen. Es wird keine Sitzungsinformation ausgetauscht oder vom Server verwaltet. In jeder Anfrage stecken somit alle Informationen, um einen Request auszuführen. Da alle Interaktionen zustandslos sind, kann man die Bearbeitung der Anfragen auf verschiedene Server verteilen. Die Zustandslosigkeit des Protokolls hat also positive Auswirkungen auf die Skalierbarkeit des Systems. (Fielding, Architectural Styles and the Design of Network-based Software Architectures, 2000)

Neben REST werden oft auch auf SOAP basierende Web Services benutzt. SOAP ist eine „Remote Procedure Call“ (RPC) Middleware, eine anwendungsneutrale Dienstschicht, die HTTP als Transportprotokoll und XML als Nachrichtenformat verwendet, um zwischen Anwendungen zu vermitteln. SOAP ist eine Art Protokollbaukasten, bei der jeder Entwickler sein eigenes Anwendungsprotokoll definiert. Dieses Protokoll beschreibt die Schnittstelle, also alle verfügbaren Methoden und den genauen Aufbau der Anfrage und Antwort. Im Gegensatz zu REST hat eine Änderung an dem Nachrichtenformat sowie das Hinzufügen oder Entfernen von Informationen zur Folge, dass die Schnittstelle angepasst werden muss. Somit fehlt in SOAP die Möglichkeit der Evolution des bestehenden Web Services. Ein weiterer Unterschied zu REST ist der Adressraum. Nur REST bietet mit dem Konzept der URIs einen globalen Adressraum, über den jede Ressource, auch über die Grenzen der Anwendung oder Organisation hinweg, adressiert werden kann. (Curbera, et al., 2002)

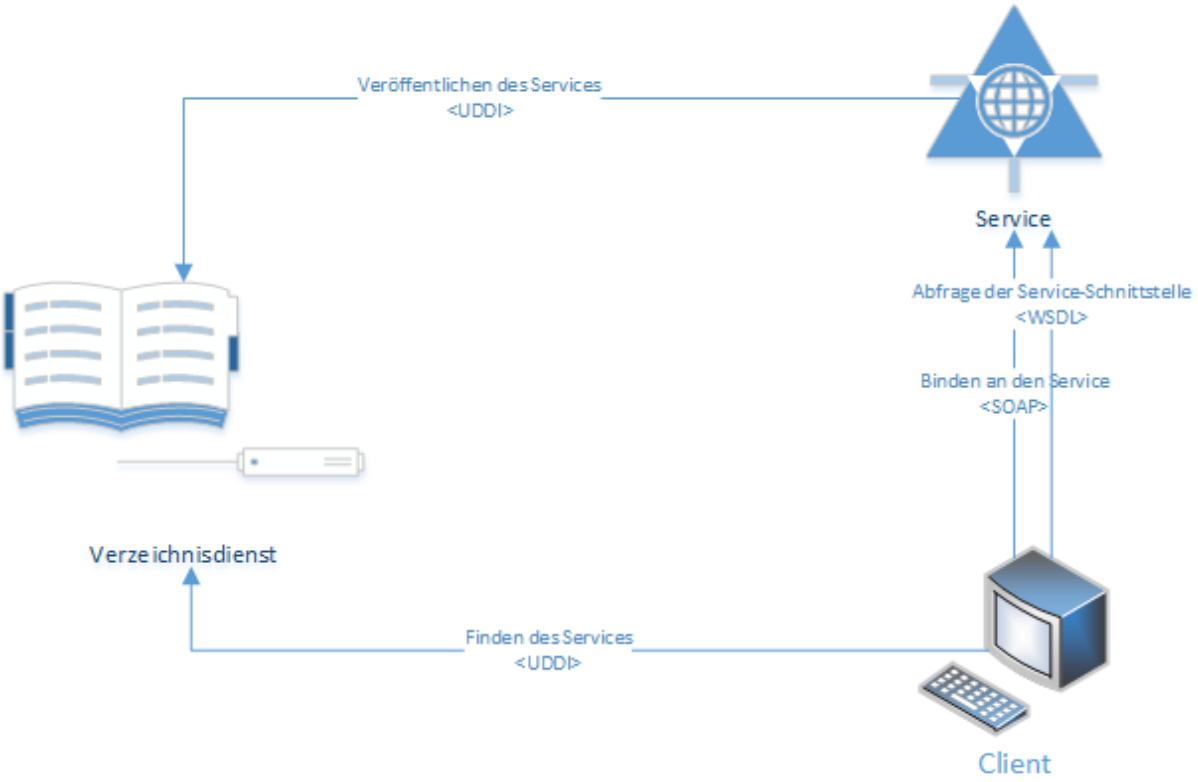

Abbildung 2: SOAP-Architektur

2.1.3 JSON

JSON ist die Abkürzung für **J**ava**S**cript **O**bject **N**otation und ist ein schlichtes, kompaktes, aber vollständiges Datenaustauschformat zwischen Anwendungen.[1]

JSON basiert auf zwei Strukturen:

- Einer ungeordneten Menge von Schlüssel-Wert-Paaren. In vielen Programmiersprachen ist dies realisiert als *object*, *record*, *struct*, *dictionary*, *hash table*, *keyed list* oder *associative array*.
- Einer geordneten Liste von Werten. In den meisten Sprachen ist dies realisiert als *array*, *vector*, *list*, oder *sequence*.

Dies sind universelle Datenstrukturen, die von praktisch allen modernen Programmiersprachen unterstützt werden. Deshalb basiert JSON, welches als Datenaustauschformat zwischen diesen Sprachen agiert, auf diesen Strukturen.[2] Die folgenden Abbildungen entstammen http://www.json.org und stellen den Aufbau der JSON-Datenstrukturen dar.

[1] http://wikis.gm.fh-koeln.de/wiki_db/Datenbanken/JSON
[2] http://www.json.org

Ein Objekt beinhaltet eine ungeordnete Menge an Key/Value Paaren und ist die Grundlage eines jeden JSON-Dokuments.

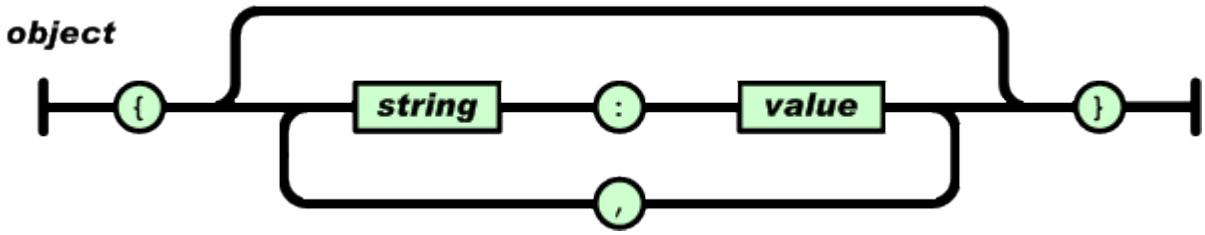

Abbildung 3: JSON Object

Ein Array ist eine geordnete Liste von Werten

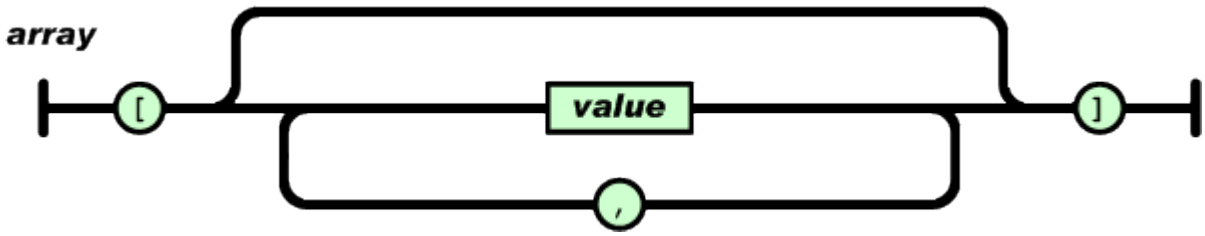

Abbildung 4: JSON Array

Ein Wert kann ein Objekt, ein Array oder ein primitiver Datentyp sein. Eine Verschachtelung ist ebenfalls möglich.

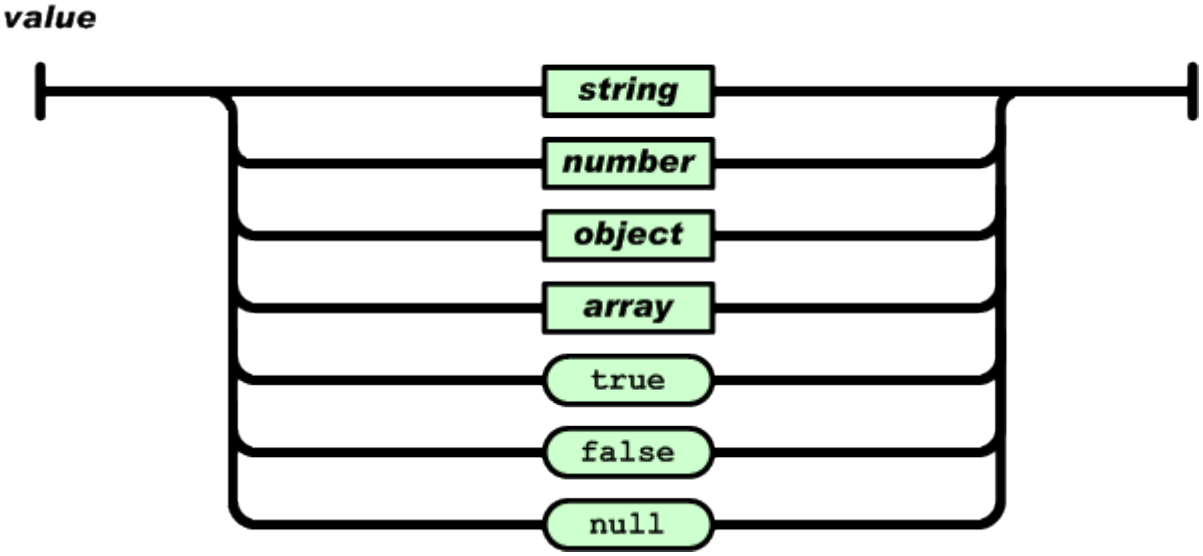

Abbildung 5: JSON Value

JSON ist ein schlankes textbasiertes Datenaustauschformat. Es gibt keinen Standard, so wie es bei einem XML-Schema der Fall ist. Die Syntax hat wenig Redundanz. Deshalb ist die zu übertragene Datenmenge geringer als bei XML. Metadaten müssen in JSON durch den Web Service in das Übertragungsobjekt als zusätzliches Objekt hinzugefügt werden. (Ihrig, 2013)

2.2 Verwandte Arbeiten

Im Folgenden werden verwandte Arbeiten zum Design einer reflektierenden REST-API erörtert. Zunächst werden die Qualitätsmerkmale einer Programmierschnittstelle erörtert und der Aspekt der Metadaten in Programmiersystemen untersucht. Schließlich wird die Open Source Software CKAN vorgestellt und evaluiert.

2.2.1 Application Programming Interface (API) – Design

Die im ODS konzipierte REST-API stellt das Bindeglied zwischen dem System und deren Nutzern dar. Die Nutzer können dabei Menschen oder auch andere Anwendungssysteme sein, welche die Daten weiterverwenden. Eine API, eine Schnittstelle zur Anwendungsprogrammierung, wird aus vielerlei Gründen erstellt und genutzt. Zum einen als Zeitersparnis für den Nutzer, indem es die Funktionalität bereitstellt, die der Nutzer sonst selbst entwickeln müsste. Zum anderen dient sie dem Prinzip der Datenkapselung. Informationen und Daten der internen Datenstruktur werden versteckt. So können intern Änderungen oder Optimierungen erfolgen, von denen der Nutzer nichts bemerkt, solange sich die Schnittstelle nicht ändert. Schließlich ermöglichen Schnittstellen den Zugriff auf Funktionalitäten, die andernfalls kaum möglich wären. Beispiele dazu stellen die Betriebssystem- und Treiberschnittstellen dar.

Das Design einer API hat direkte Auswirkungen auf wichtige Qualitätsmerkmale wie die Leistung, Aussagekraft, Erweiterbarkeit, Veränderlichkeit und Benutzerfreundlichkeit. In der wissenschaftlichen Publikation von (Stylos & Myers, 2007) wurden die existierenden Entwurfsempfehlungen zu Anwendungsprogrammierschnittstellen in Zusammenhang gesetzt und die Design-Entscheidungen und Qualitätsmerkmale ausgearbeitet. Dabei stellte sich heraus, dass der Nutzer selbst die Qualitätsmerkmale der API beeinflusst. Verschiedene Stakeholder haben unterschiedliche Anforderungen an eine Schnittstelle.

Die **Designer** haben die Ziele der möglichst hohen Akzeptanz der Schnittstelle, der Kostenreduzierung für die Betreuung und Entwicklung sowie der fristgerechten Fertigung der Schnittstelle.

Die **API-Nutzer**, welche die API verwenden, um eigene Anwendungen zu schreiben, wollen möglichst schnell fehlerfreie, effiziente Programme mit Hilfe einer robusten, sich bewährten Schnittstelle entwickeln. Dabei können sich die API-Nutzer in ihrer Expertise stark unterscheiden. Programmieranfänger haben andere Vorstellungen und Anforderungen an eine API als Experten.

Schließlich die **Kunden** oder Nutzer der Endprodukte, welche die Anwendungen basierend auf der API einsetzen. Sie erwarten einheitlich standardisierte Produkte mit gewünschter Funktionalität und ohne Fehler.

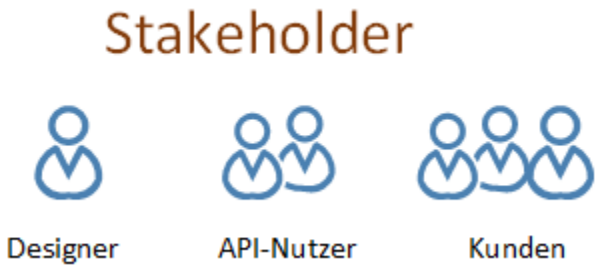

Abbildung 6: Stakeholder

Nach (Stylos & Myers, 2007) lassen sich die Qualitätsmerkmale in zwei Kategorien unterteilen, Benutzerfreundlichkeit (Usability) und Mächtigkeit (Power). Unter der Benutzerfreundlichkeit versteht man Qualitätsmerkmale, die die Nutzung der API betreffen. Die Qualitätsmerkmale der Kategorie Mächtigkeit beschreiben die Grenzen des Codes, der mit der API erstellt werden kann.

Qualitätsmerkmale der Kategorie **Benutzerfreundlichkeit** (Usability):

Erlernbarkeit (Learnability): Wie leicht ist die API zu erlernen oder zu verstehen?

Produktivität (Productivity): Wie produktiv wird die API genutzt?

Fehlervermeidung (Error-Prevention): Wie verhindert die API Fehler?

Einfachheit (Simplicity): Wie einfach ist die API?

Konsistenz (Consistency): Wie konsistent ist die API?

Übereinstimmung mit mentalem Modell (Matching Mental Models): Wie gut stimmt die API mit dem mentalen Modell des Nutzers überein?

Abbildung 7: Qualitätsmerkmale der Benutzerfreundlichkeit

Qualitätsmerkmale der Kategorie **Mächtigkeit** (Power):

Ausdrucksmächtigkeit (Expressiveness): Welche Programmarten können mit der API erzeugt werden?

Erweiterbarkeit (Extensibility): Wie erweiterbar ist die API, um benutzerspezifische Komponenten zu erzeugen?

Evolvierbarkeit (Evolvability): Wie evolvierbar ist die API? Wie gut lässt sie sich weiter entwickeln bzw. ausbauen?

Leistung (Performance): Wie viele Ressourcen verbraucht die API?

Robustheit (Robustness): Wie robust und fehlerfrei ist die API-Implementierung?

Abbildung 8: Qualitätsmerkmale der Mächtigkeit

Die Benutzerfreundlichkeit betrifft hauptsächlich die API-Nutzer, die Fehlervermeidung hingegen auch die Kunden. Die Qualitätsmerkmale der Kategorie Mächtigkeit betreffen bei der Ausdrucksmächtigkeit und der Erweiterbarkeit die API-Nutzer. Die Evolvierbarkeit ist für den Designer wichtig. Schließlich beeinflussen die Leistung und die Robustheit der API den Kunden. (Stylos & Myers, 2007)

2.2.2 Metadaten und das Metadata Object Protocol

Metadaten sind Daten über (Nutz-) Daten. Sie beinhalten Informationen über Merkmale anderer Daten. Sie helfen die Nutzdaten besser verständlich zu machen, indem sie die Daten beschreiben und ihre Struktur darstellen. Man unterscheidet zwischen entitätsbezogenen Metadaten und typbezogenen Metadaten (Schemata).

Metadaten werden unterschiedlich genutzt und haben verschiedene Ziele. Sie können Nutzern des Systems zum Beispiel helfen, die Nutzdaten besser einordnen zu können. In Dokumentenmanagementsystemen werden Metadaten genutzt, um Dokumente zu kategorisieren. Dies erleichtert später die Suche nach bestimmten Dokumenten. Außerdem können Metadaten genutzt werden, um Daten maschinell zu interpretieren, zu verarbeiten oder zu validieren.

Metadaten treten in vielen Bereichen auf, wie zum Beispiel bei Datenbanken, in der

Dokumentenverwaltung und in der Software-Modellierung. In objektorientierten Sprachen gibt es das Prinzip der Reflexion (reflection). Dabei nutzt man Metadaten, um Informationen über die Typstruktur von Objekten darzustellen.

Die Reflexion umfasst zwei Aktivitäten, die Fähigkeit eines Programms, das eigene Verhalten zu beobachten (Introspection) und der daraus resultierenden Änderung des Verhaltens (Intercession). Die folgende Abbildung zeigt den gegenseitigen Kausalzusammenhang einer reflektierenden Anwendung, abgebildet auf der Basisebene (base level), und deren Selbstrepräsentation, abgebildet auf der Metaebene (Metalevel). Änderungen an der einen Ebene werden auf die andere Ebene reflektiert. Das „Metaobject Protocol“ (MOP) ist eine Schnittstelle und ermöglicht die systematische „Introspection“ und „Intercession“ der Objekte der Basisebene. Dabei unterstürzt MOP wahlweise strukturbasierende oder verhaltensbasierende Reflexion. Die strukturbasierende Reflexion (structurial reflection) betrifft Belange im Zusammenhang mit Klassenhierarchie, der Objektbeziehungen und der Datentypen. So kann ein „Metalevel Object“ untersuchen, welche aufrufbaren Methoden ein Objekt der Basisebene besitzt. Im Gegensatz dazu konzentriert sich die verhaltensbasierende Reflexion (behavioral reflection) auf die Berechnungssemantik der Anwendung. Beispielsweise kann eine verteilte Anwendung die verhaltensbasierende Reflexion nutzen, um ein Kommunikationsprotokoll zu wählen und zu laden, welches die jeweiligen Anforderungen der Umgebung am besten erfüllt. (McKinley, Sadjadi, Kasten, & Cheng, 2004)

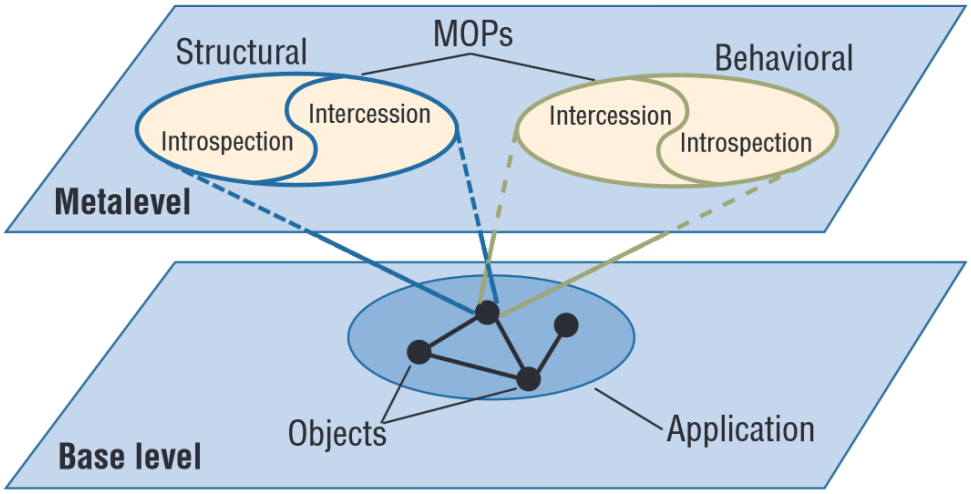

Abbildung 9: Metaobject Protocol (McKinley, Sadjadi, Kasten, & Cheng, 2004)

Ein Entwickler kann die eingebauten Mechanismen zur Reflexion einer Programmiersprache nutzen, wie bei „Common Lisp Object System“ (CLOS) und Python, ansonsten ist er auf die vom Anwendungssystem bereitgestellten Schnittstellen angewiesen. (Kiczales, Bobrow, &

Rivieres, 1991) (Kiczales, Ashley, Rodriguez, Vahdat, & Bobrow, 1993)

Ein System bezeichnet man als selbstreferentiell, wenn es Metadaten im gleichen Datenbankmodell wie die Nutzdaten repräsentiert. Der Vorteil dabei ist, dass man Standardschnittstellen und Standardmechanismen des Systems verwenden kann, um die Metadaten zumindest auszulesen. Veränderungen an den Metadaten dagegen sind nicht trivial.[1]

2.2.3 Comprehensive Knowledge Archive Network (CKAN)

CKAN ist eine Open Source Software (GNU GPLv3) und wird unter der Schirmherrschaft der Open Knowledge Foundation (OKF) weiterentwickelt. Das Ziel ist es, einen Softwarestandard für Open Data anzubieten. Das CKAN lässt sich portieren und überall installieren. Die OKF bietet auch eigene Hosting-Möglichkeiten an, welche einen Support und laufende Aktualisierung beinhalten. Das System wird bereits von einigen Nationen als technische Basis ihrer Open Government Lösungen genutzt und gehört zur Gruppe der Data Hubs.[2]

Als Data Hub bezeichnet man ein System, welches Daten von mehreren Quellen zusammenfasst, diese speichert und in leicht zugänglicher sowie standardisierter Form zur Weiterverwendung bereitstellt.

Wichtige Funktionalitäten eines Data Hub sind:

- Extraktion, Transformation und Laden von Daten aus beliebigen Quellen
- Speicherung der Datenmengen und Anbieten einer Abfragemöglichkeit auf die Daten
- Anbieten und Visualisieren der Daten
- Bereitstellung von Datenanalysen und Auswertungen
- Messen und Gewährleisten der Datenqualität
- Berechtigungs- und Rechteverwaltung[3]

Grundsätzlich ist das CKAN eine sehr stabile und reife Softwareplattform. Die Community rund um das Projekt ist sehr aktiv, die Dokumentation ist weitestgehend vollständig und aktuell.[4]

CKAN bietet einen auf Python basierenden Webdienst an, der über eine Web-Oberfläche und eine REST-Schnittstelle Datensätze bereitstellt. Dabei bietet die REST-Schnittstelle eine größere Funktionalität als die Web-Oberfläche. Der Datenaustausch erfolgt mittels Java Script Object Notation (JSON).

---

[1] http://pi.informatik.uni-siegen.de/kelter/lehre/14w/lm/lm_md_20081109_a5.pdf
[2] http://datahub.io/about
[3] http://blog.okfn.org/2012/03/09/from-cms-to-dms-c-is-for-content-d-is-for-data/
[4] http://www.w3.org/egov/wiki/images/f/f1/W3C_OpenSource_OpenData.pdf

Die Bereitstellung von Daten erfolgt in Form von Datensätzen (dataset), beispielsweise Pegelstände deutscher Flüsse oder städtische Temperaturreihen. Ein einzelner Datensatz besitzt Metadaten, zu denen Titel, Id, Beschreibung, Tags, Lizenz, Autor und die Lizenz gehören. Die Nutzdaten (resources) können in beliebiger Form vorliegen und sind in der Lage auf andere (Nutz-) Daten innerhalb oder außerhalb des CKAN zu verweisen. CKAN tauscht Metadaten im JSON-Format aus. Die Metadaten der Nutzdaten enthalten Titel, Beschreibung, Format und die Id. CKAN erlaubt es, die Standardmetadatenschemata benutzerspezifisch zu erweitern. Die Konzentration auf das Wesentliche zusammen mit der großen Flexibilität sind die Gründe für die Verbreitung dieses Metadatenmodells.[1]

Die CKAN-API basiert auf REST und ist sehr umfangreich. Sie unterstützt beispielsweise eine Freitext- und Tag-Basierte-Suche. REST ist wie in Kapitel 2.1.2 erwähnt kein abgesegneter Standard. Er gibt aber gewisse Konformitätsbedingungen vor. So soll der Zugriff auf die Ressourcen über eine eindeutige URL erfolgen. Es gibt inzwischen drei Versionen der CKAN-API. Die Nutzung der jeweiligen Version ist durch die URL bestimmt:

Version 1: …/api/1/… oder …/api/…

Version 2: …/api/2/…

Version 3: …/api/3/…

Zum Großteil folgt die API auch den empfohlenen Design-Entscheidungen von REST-APIs. Eine Ausnahme bildet die Fehlerbehandlung. Die API von CKAN gibt immer den HTTP-Statuscode „200 OK“ zurück, auch wenn bei der Anfrage ein Fehler passiert und beispielsweise die angeforderte Ressource nicht gefunden werden konnte. In der Antwort, transportiert über ein JSON-Dokument, wird der Fehler dann detailliert beschrieben. Der HTTP-Statuscode bezieht sich nur auf die HTTP-Übertragung, nicht auf den Inhalt. (Winn, 2013)

2.3 Konzeption

Für die Konzeption des JValue Open Data Service müssen zunächst die Anforderungen, also Bedingungen und Fähigkeiten, die er erfüllen muss, spezifiziert werden.

Der ODS wurde im Rahmen eines Projekts an der Forschungsgruppe für Open-Source-Software der Friedrich-Alexander-Universität Erlangen-Nürnberg entwickelt. Die Motivation war es, den Dienst für studentische Projekte zu nutzen und so den Studenten die Erstellung der auf freien Daten basierenden Android-Applikationen, zu erleichtern. Der Open Data Service unterstützt die Studenten, indem die oft mühsame und zeitintensive Anbindung der verschiedenen

[1] https://www.govdata.de/metadatenschema

Datenquellen vereinfacht wird.

Als konkretes Beispiel wurden OpenStreetMap, eine Datenbank für geographische Daten, und PEGELONLINE, das Gewässerkundliche Informationssystem der Wasser- und Schifffahrtsverwaltung des Bundes, als Datenquellen ausgewählt, unter der Berücksichtigung auch andere Datenquellen später anbinden zu können.

Das Kartenmaterial von OpenStreetMap liegt in großen Datenmengen vor. Die Datenbank besitzt weltweit 2,7 Milliarden Knoten (Nodes), 264 Millionen Wege (Ways) und 3 Millionen Relationen (Relations). Diese Gesamtdatenmenge ist im „OSM XML"-Format bereits ca. 500GB groß (Stand Dezember 2014). Das „OSM XML"-Format basiert auf der Extensible Markup Language (XML), einer Auszeichnungssprache zur Darstellung hierarchisch strukturierter Daten. Die Karteninformationen werden von der Online-Community gepflegt und sind von unterschiedlicher Qualität bezüglich Genauigkeit und Vollständigkeit. Die Anbindung an OpenStreetMap geschieht über eine REST-Schnittstelle. Diese bietet die grundlegenden Datenbankoperationen an, um die Informationen erstellen, lesen, verändern und löschen zu können. Außerdem wird Kartenmaterial für bestimmte einzelne Gebiete wie deutsche Bundesländer, Deutschland oder ganz Europa bereitgestellt. So können Nutzer gezielt die benötigten Kartenbereiche laden. Damit der Nutzer nicht bei jedem Update die Karten neu laden muss, werden Updates auch als Changesets veröffentlicht, welche die Unterschiede zwischen den Versionen beinhalten.[1]

PEGELONLINE publiziert tagesaktuelle Rohwerte unterschiedlicher gewässerkundlicher Parameter der Wasserstraßen des Bundes der letzten 30 Tage. Die Wasserstände werden dabei ungeprüft und ohne Gewähr auf deren Genauigkeit von den fachlich zuständigen Wasser- und Schifffahrtsämtern bzw. Direktionen zur Verfügung gestellt. Die bereitgestellten Daten enthalten unter anderem Informationen zu den Messstellen, Gewässern, Wasserständen, Luft- und Wassertemperaturen sowie Zeitreihen mit den aktuell gemessenen Werten. Je nach Messstation ist die Vielfalt der Informationen unterschiedlich ausgeprägt. Zum Abrufen der Daten stellt PEGELONLINE auf REST- und SOAP- basierende Schnittstellen bereit.[2]

[1] https://wiki.openstreetmap.org/

[2] http://www.pegelonline.wsv.de/

Abbildung 10: PEGELONLINE Messpunkte Stand Dez. 2014[1]

Die Anforderungen an den ODS lassen sich in funktionale und nichtfunktionale Anforderungen unterscheiden. Während die funktionalen Anforderungen die Arbeitsweise und das Verhalten beschreiben, kennzeichnen die nichtfunktionalen Anforderungen die qualitativen, quantitativen und systembezogenen Eigenschaften des Dienstes.

2.3.1 Funktionale Anforderungen

Anbindung unterschiedlicher Datenquellen

Eine der zentralen Funktionalitäten des ODS ist die Anbindung unterschiedlicher Datenquellen. Hierbei sind die Besonderheiten der Quellen zu berücksichtigen. Die Quellen haben unterschiedliche Schnittstellen, nutzen verschiedene Datenformate und Encodings. Außerdem können sie rechtlichen oder technischen Einschränkungen unterliegen. So nutzt PEGELONLINE für Sonderzeichen das UTF-8-Format.[2]

Vereinigung der Daten

Der Nutzer des ODS erwartet einheitlich formatierte Daten zu bekommen. Deshalb müssen die

[1] https://www.pegelonline.wsv.de/gast/karte/standard
[2] http://www.pegelonline.wsv.de/webservice/dokuRestapi

Daten der gegebenenfalls unterschiedlich strukturierten Datenquellen in einem einheitlichen Format vereinigt werden. Dazu werden die Daten aus den verschiedenen Quellen extrahiert und in das für die Datenbank und den Nutzer des ODS geeignete einheitliche Format überführt.

Qualitätsverbesserung der Daten

Die Daten besitzen unterschiedliche Datenqualität. So werden die Daten bei OpenStreetMap von der Online-Community erfasst und sind unterschiedlich genau oder vollständig. PEGELONLINE markiert ungültige Messwerte mit dem „xsd:nil“.[1] Der Nutzer des ODS will möglichst nur valide gereinigte Daten bekommen. Deshalb muss der ODS die Qualitätsverbesserung der Daten durchführen. Dazu gehört die Filterung und Validierung der Daten. Der Nutzer sollte jedoch die Möglichkeit haben auch die Originaldaten abrufen zu können.

Aktualisierung der Daten

Die in den ODS überführten Daten verlieren mit der Zeit an Aktualität. Der Nutzer interessiert sich jedoch für möglichst aktuelle Informationen. Deshalb muss der ODS die Daten der externen Quellen regelmäßig aktualisieren. Die Zeitperiode sollte für jede Quelle individuell bestimmbar sein.

Einfache einheitliche Schnittstelle

Die Nutzer des Dienstes sollen die Daten aus dem ODS über eine einfache einheitliche auf der auf dem Paradigma REST basierenden Schnittstelle abrufen können. Die ehemals heterogenen Daten werden über diese Schnittstelle in einem einheitlichen Format angeboten. Es sollen dabei einheitliche Zugriffs- und Anfragemethoden verwendet werden. Der Nutzer soll auch abrufen können, welche Anfragen unterstützt und welche Datenquellen über den ODS abgerufen werden können. Schließlich sollen über die Schnittstelle auch Metadaten, also Informationen über die Quellen und die verwendeten Datenstrukturen und Datentypen, abgefragt werden können.

2.3.2 Nichtfunktionale Anforderungen

Benutzbarkeit

Der ODS sollte verständlich, intuitiv und einfach zu bedienen sein. Um dies zu gewährleisten, muss die Schnittstelle sich an etablierte und bewährte Standards für REST-Schnittstellen

[1] http://www.pegelonline.wsv.de/webservice/dokuRestapi

orientieren und dabei die Qualitätsmerkmale aus Kapitel 2.2.1 berücksichtigen. Durch die Vereinheitlichung der angebotenen Daten und der Gestaltung der REST-API soll die Erlernbarkeit und Nutzung des Dienstes gesteigert werden.

Zuverlässigkeit

Eine hohe Zuverlässigkeit ist für den ODS wichtig. Der Dienst sollte stabil laufen. Einzelne Verarbeitungsfehler sollten nicht zum Ausfall des ganzen Dienstes führen. Die Verbindung zum Dienst sollte zuverlässig funktionieren und die Daten sollten vollständig und ohne Fehler übertragen werden.

Verfügbarkeit

Der ODS wird von vielen Nutzern genutzt und diese erwarten eine hohe Verfügbarkeit des Systems. Downtime oder Ausfälle sollten vermieden werden. Bei Wartungsarbeiten oder Updates sollte der Dienst weiterhin ansprechbar sein.

Performance

Die Anfragen an den ODS sollen schnell verarbeitet werden, um Nutzern möglichst kurze Antwortzeiten zu bieten. Dies ist auch wichtig für die Skalierung des Dienstes, denn das Wachstum der Daten und der Nutzerzahl soll bewältigt werden können.

Wartbarkeit

Der ODS soll für den Nutzer einfach zu warten sein. Gleiche Anfragen sollen gleiche Ergebnisse liefern. So soll man Fehler leicht reproduzieren können. Durch das Loggen von Aktivitäten wird die Fehleranalyse erleichtert. Die Erweiterung um neue Datenquellen soll möglichst einfach stattfinden.

2.4 Konzeption der Lösung

In diesem Abschnitt werden die gesammelten Anforderungen in die Konzeption der Lösung einbezogen. Zuerst wird die Gesamtarchitektur vorgestellt und die einzelnen Komponenten sowie ihr Zusammenspiel erläutert. Anschließend wird speziell auf die Konzeption der Query API detailliert eingegangen.

2.4.1 Architektur des Open Data Service

Der Open Data Service besteht aus drei durch Schnittstellen entkoppelten Komponenten, der Import-, Datenbank- und Serverkomponente. Die folgende Abbildung zeigt die grobe Architektur des ODS.

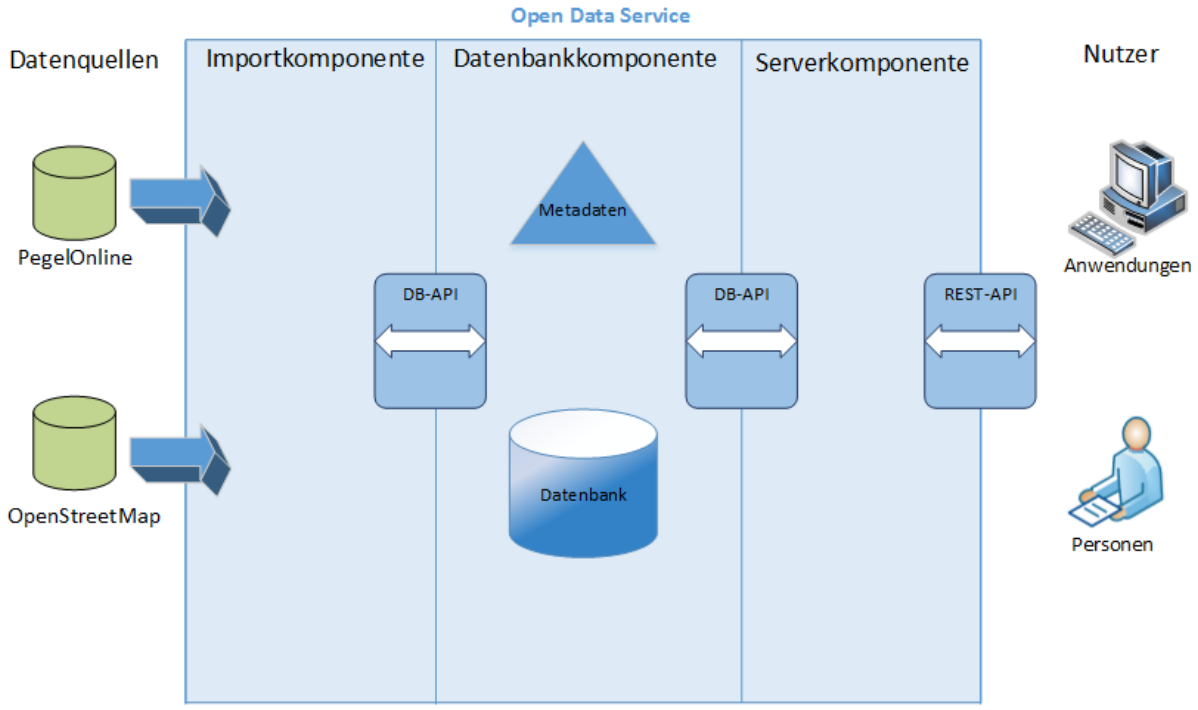

Abbildung 11: Grobe Architektur des Open Data Service

Die Importkomponente ist zuständig für die Anbindung der externen Datenquellen. Die extrahierten Daten werden dann zusammen mit den Metadaten in der Datenbankkomponente persistiert. Schließlich kann der Nutzer des Open Data Services die Daten aus der Datenbank über die REST-Schnittstelle der Serverkomponente abrufen und nutzen. Nun werden die einzelnen Komponenten detailliert erörtert.

2.4.2 Importkomponente

Die Importkomponente des ODS basiert auf dem „Pipes und Filter“-Architekturmuster. Dies ist ein Datenfluss-System-Muster, welches Systeme beschreibt, die Datenströme verarbeiten. Ein Filter stellt dabei einen Verarbeitungsschritt dar. Die eingehenden Daten werden verarbeitet, umgewandelt und schließlich ausgegeben. (Garlan & Shaw, 1994) Die Art der Umwandlung ist durch den Filter bestimmt. In der folgenden Abbildung ist die „Pipes und Filter“-Architektur schematisch dargestellt.

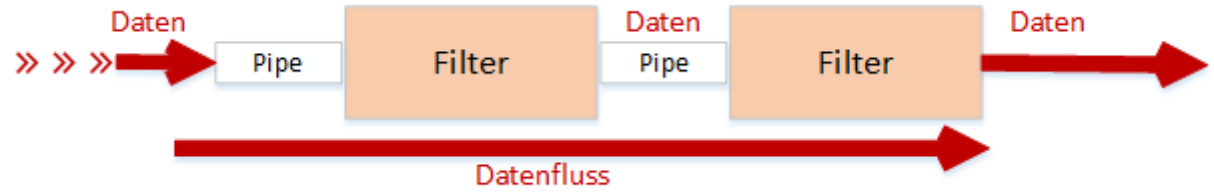

Abbildung 12: Pipes und Filter

Die links ankommenden Daten (Dateneingabe) werden in mehreren Schritten verarbeitet und das Ergebnis der Verarbeitung wird rechts wieder ausgegeben (Datenausgabe). Daten werden durch Kanäle (Pipes) von Filter zu Filter weitergegeben. Dabei ist der Ausgang eines Verarbeitungsschrittes gleichzeitig der Eingang des nachfolgenden Verarbeitungsschrittes. Das Ergebnis ist somit abhängig von der Eingabe. Der Vorteil dieses Musters ist die Möglichkeit von zukünftigen Systemerweiterungen durch Austausch oder Neuordnung der Verarbeitungskomponenten. Außerdem ist eine gleichzeitige Verarbeitung der einzelnen Phasen möglich. (Buschmann, Meunier, Rohnert, Sommerlad, & Stal, 1998)

Die Importkomponente stellt einen ETL-Prozess (Extraction, Transformation and Load) dar. Dieser besteht aus mehreren nacheinander gelagerten Verarbeitungsschritten, um Daten aus den unterschiedlich strukturierten Datenquellen in die ODS-Datenbank zu überführen. Die einzelnen Verarbeitungsschritte sind in der Abbildung 13 illustriert. Der ETL-Prozess wird vor allem bei Data-Warehouses genutzt. Hier müssen große Datenmengen aus mehreren operationalen Datenbanken in einem einheitlichen Format konsolidiert werden, um dann im Data-Warehouse gespeichert zu werden. (Bodendorf, 2003)

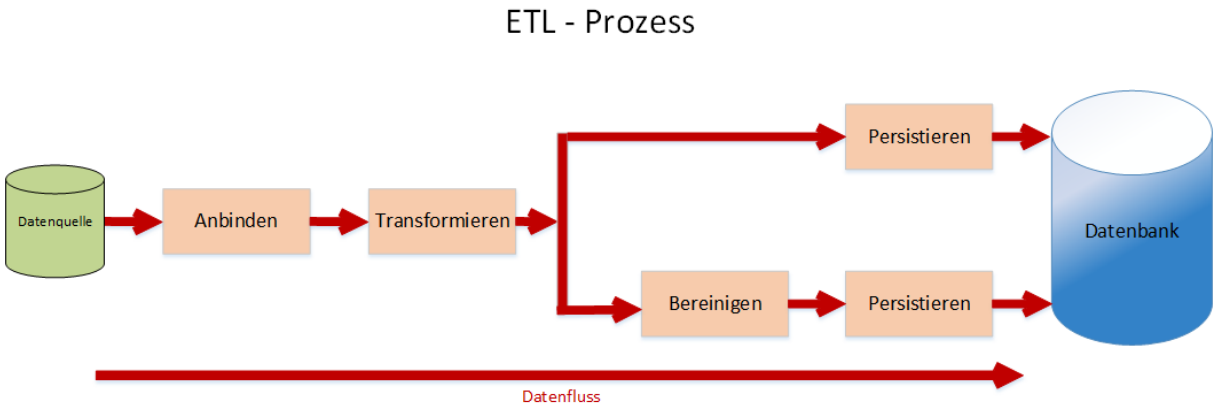

Abbildung 13: ETL-Prozess im Open Data Service

Anbinden

Bei der Anbindung der externen Quellen ist das Ziel, die von den Quellen bereitgestellten Daten zu sammeln. Da jede Quelle eigene Schnittstellen besitzt, benötigt der ODS für jede Quelle einen spezifischen Adapter. Dieser kümmert sich um den Zugriff und das Extrahieren der Daten.

Oft stellen die Schnittstellen Daten auf ähnliche Art und Weise im gleichen Format bereit. So lassen sich beispielsweise über das HTTP-Protokoll viele Datensätze im XML- oder JSON-Format abrufen. Für solche Fälle nutzt man spezielle generische „Grabber“. Ein „Grabber“ ist spezialisiert darauf, Daten eines bestimmten Formates von beliebigen Datenquellen zu extrahieren.

Transformieren

Bevor man die Daten in den ODS integriert, müssen diese erst in das für die Datenbank passende Format transformiert werden. Wegen der unterschiedlichen Datenmodelle bedarf dies zunächst der Überführung in ein einheitliches Datenmodell. Die Überführung wird im ODS von so genannten „Translatoren“ durchgeführt. Dabei ist ein „Translator“ zuständig für ein bestimmtes Datenformat. So überführt der „XML-Translator“ die XML-Daten in das für die Datenbank passende Datenmodell und passt die Daten gegebenenfalls dabei an.

Bereinigen

Datenfehler sind unvermeidlich. Daten können unvollständig oder fehlerhaft sein. Bei der Aufbereitung der Daten werden z.B. fehlende Werte durch entsprechende Default-Werte ersetzt. Falsch erkannte Werte werden als invalide markiert oder sogar entfernt. Diese qualitätsverbessernden Maßnahmen werden mit Hilfe der Qualitätsverbesserungs- und Datenergänzungsfilter umgesetzt.

Persistieren

Schließlich werden die nicht veränderten Roh- und die gesäuberten Daten über die Datenbankschnittstelle in die Datenbank geladen. Dabei sind Strategien für das Zusammenführen und Aktualisieren der Daten vorzusehen, da während der Aktualisierung die Problematik entsteht, dass die Daten in diesem Zeitfenster nicht zur Verfügung stehen oder inkonsistent vorliegen. Das Abrufen und die Speicherung der Daten erfolgt über die Datenbankschnittstelle.

2.4.3 Datenbankkomponente

Die Datenbankkomponente ist für die Speicherung und die Bereitstellung der gesammelten Daten zuständig. Sie besteht aus einer Datenbank und einer Datenbankschnittstelle. Die Schnittstelle kennt die konkrete Datenbank und delegiert die Anweisungen an diese weiter. Die Schnittstelle dient als Abstraktion der Datenbank und entkoppelt so die konkrete

Implementierung. Dies ermöglicht es, bei Bedarf die Datenbank zu wechseln.

Stellt man das klassische relationale Datenbankmanagementsystem (RDBMS) einer NoSQL-Datenbank gegenüber, so sieht man, dass das RDBMS im Gegensatz zur NoSQL-Datenbank von festen Schemata abhängig ist, auf deren Basis die Daten abgebildet und gespeichert werden. Dies bedeutet, dass im Vorfeld viel über die Daten bekannt sein muss und zukünftige Änderungen nicht stark vom bisherigen Schema abweichen dürfen. Die heterogenen schemalosen und unstrukturierten Daten machen heutzutage einen nennenswerten Teil der Gesamtinformationen aus und übersteigen das Leistungspotential der Abbildung durch relationaler Datenbanken. Die NoSQL-Datenbanken sind sehr heterogen. Sie lassen sich in Key-Value Stores, Spalten/Tabellen-Orientierte und dokumentenorientierte Systeme unterteilen. Während diese Unterscheidung sich auf die Datenmodelle bezieht, hat die Einordnung der Datenbank entlang des CAP-Dreiecks Auswirkung auf den Zugriff, die Konsistenz und Sicherheit der Daten.

Das CAP-Theorem besagt, dass es in einem verteilten System nicht möglich ist, zugleich die Konsistenz (Consistency), Verfügbarkeit (Availability) und Partitionstoleranz (Partition tolerance) zu garantieren. Dabei ist der Bezug auf verteilte Systeme als Ganzes, nicht dessen innere Datenstrukturen wichtig. Systeme, die konsistent arbeiten, sehen zur selben Zeit dieselben Daten. Bei einem System mit hoher Verfügbarkeit ist die Beantwortung einer Anfrage stets und in sehr kurzer Zeit möglich. Ein verteiltes System mit hoher Partitionstoleranz kann Verluste von Nachrichten und Teilen des Netzwerks verkraften. Allerdings können gemäß dem CAP-Theorem nur zwei der drei Eigenschaften gleichzeitig von einen verteilten System unterstützt werden. (Gilbert & Lynch, 2002)

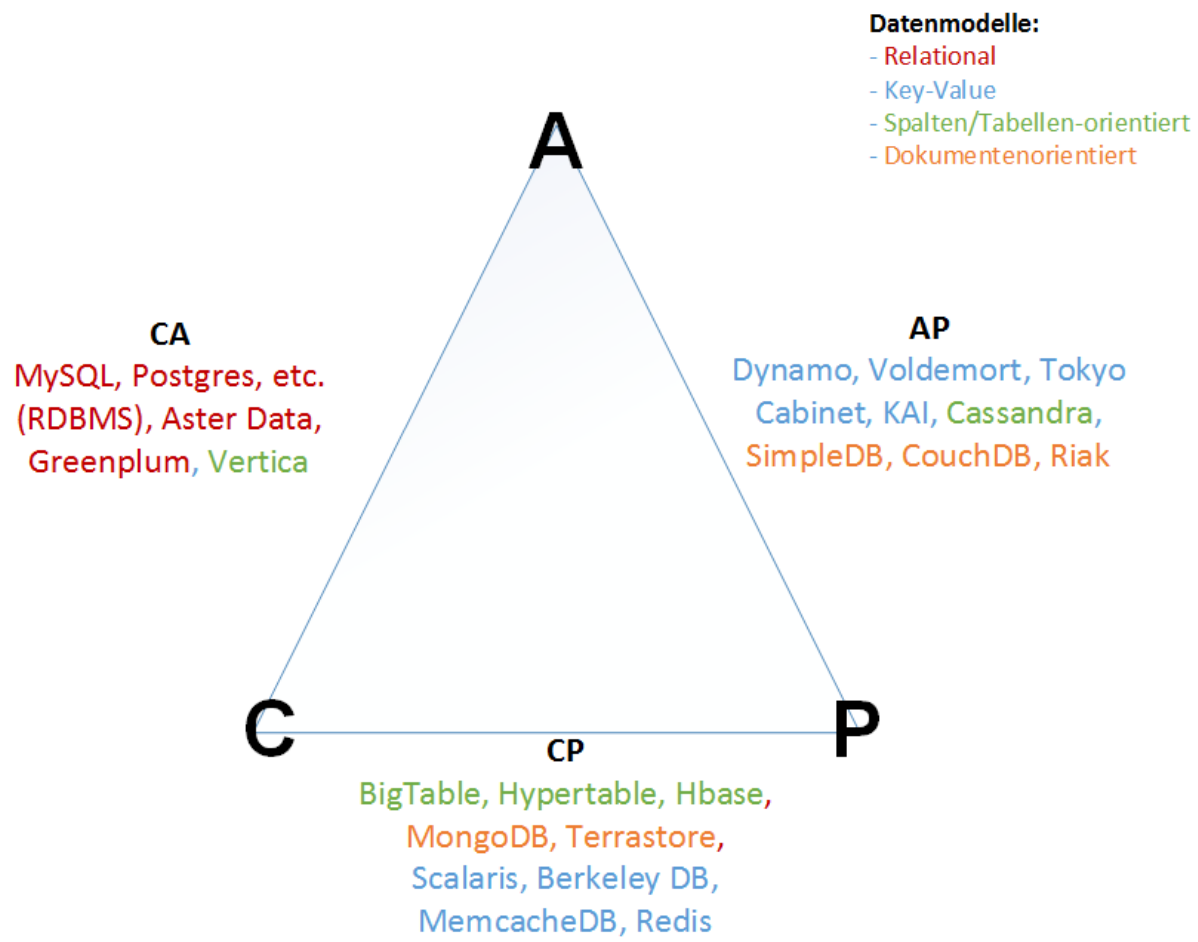

Abbildung 14: CAP-Theorem[1]

Im ODS wird zur Speicherung der Nutz- und Metadaten die dokumentenorientierte Datenbank „Apache CouchDB“ verwendet. CouchDB ist ein Akronym und steht für „**C**luster **o**f **u**nreliable **c**ommodity **h**ardware **D**ata **B**ase“. Die herauszustellenden Merkmale dieses Datenbanksystems sind die Nutzung der JSON-Dokumente als Datenspeicher und des HTTP-Protokolls (per REST) für Anfragen sowie die Gewährleistung der Zuverlässigkeit und Konsistenz der Datenspeicherung.[2]

Bei dokumentenorientierten Datenbanken bilden Dokumente die Grundeinheit zur Speicherung. Ein Dokument ist zu verstehen als eine strukturierte Zusammenstellung bestimmter Daten. Die Daten in einem Dokument werden in Form von Schlüssel-Wert-Paaren gespeichert. Bei CouchDB sind dies JSON-Objekte, die als Schlüssel einen String und als Wert einen primitiven oder wiederum geschachtelten Datentyp besitzen. Die JSON-Strukturen sind im Kapitel 2.1.3 visualisiert.

[1] Angelehnt an http://stelmach.biz/blog/2012/post2.html
[2] http://wikis.gm.fh-koeln.de/wiki_db/Datenbanken/CouchDB

Neben der Schemafreiheit , der weniger restriktiven Lizenz (Apache 2.0) und besserer Skalierung war auch die Tatsache, dass man zusammengehörige Daten auch in einem einzelnen Dokument speichern kann, ein entscheidender Faktor bei der Wahl der Datenbank. Der Nachteil dabei ist, dass Operationen mit den in Dokumenten gespeicherten Daten manuell zu programmieren sind, da CouchDB selbst über keine mächtige Abfragesprache wie SQL verfügt, in der entsprechende Operationen bereits enthalten sind.[1]

2.4.4 Serverkomponente

Die Serverkomponente ermöglicht den Datenzugriff des Nutzers auf den ODS. Die Serverkomponente besteht aus der REST-Schnittstelle, an die der Nutzer Anfragen stellen kann, und den Routingkomponenten, welche die Anfragen bearbeiten und ausführen. Die Anfragen werden schließlich an die DB-API delegiert und der Nutzer erhält die Antwort in Form von Nutz-, Metadaten oder einer Fehlernachricht. Als Datenformat wird dabei einheitlich JSON benutzt.

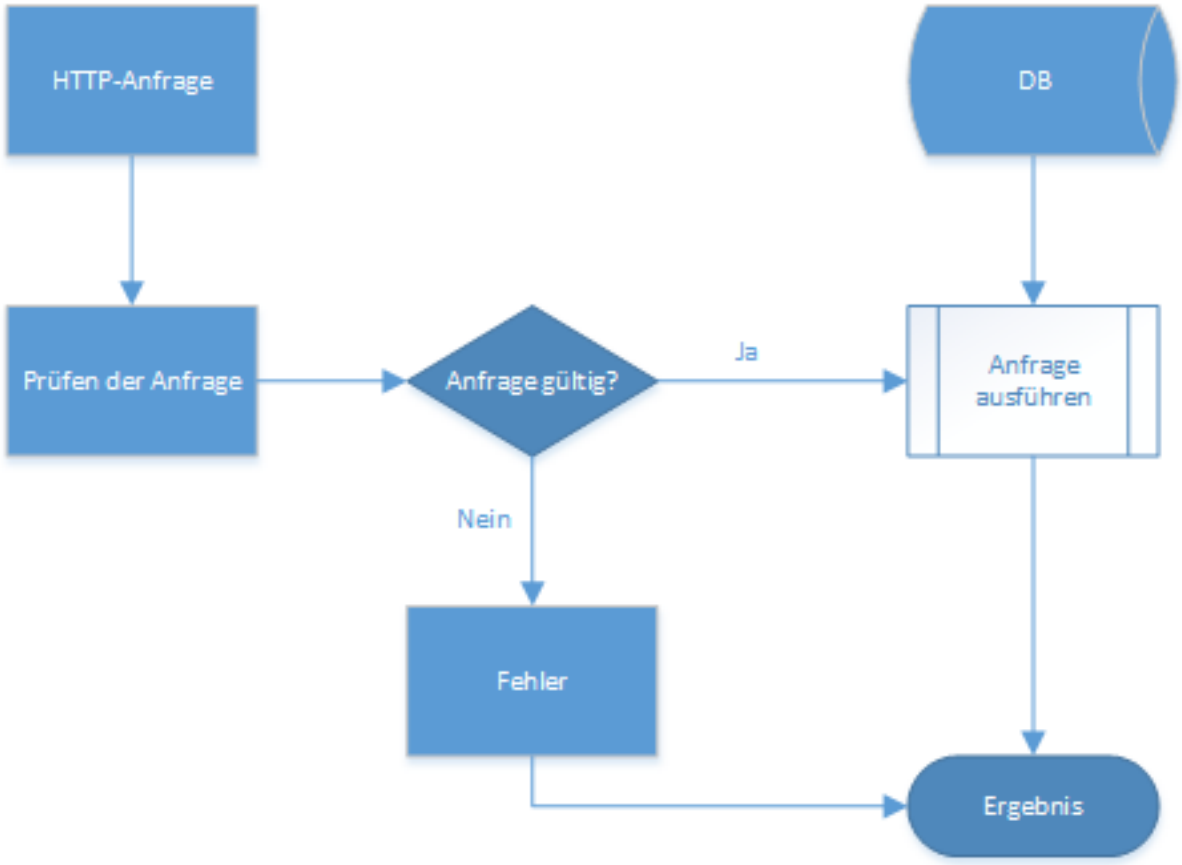

Abbildung 15: Ablaufdiagramm der Anfrageverarbeitung

[1] http://wikis.gm.fh-koeln.de/wiki_db/Datenbanken/DokumentenorientierteDatenbank

Bekannte Frameworks zur Entwicklung von Webdiensten legen ihren Schwerpunkt auf die Kommunikation zwischen Mensch und Computer. Der ODS soll in erster Linie ein Dienst für andere Applikationen sein. Dies wird erreicht, indem die Kommunikation über eine REST-basierte Schnittstelle umgesetzt wird. Das „Restlet“-Framework erleichtert dabei den Implementierungsaufwand erheblich und unterstützt zugleich die korrekte Umsetzung der Architektur. Restlet bietet Logging-Maßnahmen zum Speichern von Zugriffen, erleichtert die Darstellung und Umsetzung von Statuscodes und schützt Ressourcen vor unautorisiertem Zugriff. [1]
Wie der Name Restlet andeutet, beinhaltet das Restlet-Framework für REST optimierte Servlets. Dies sind Java-Klassen, deren Instanzen innerhalb von Webservern Anfragen von Clients entgegennehmen und dynamische Antworten liefern. Dadurch war es auch möglich den ODS auf einem Apache Tomcat-Webserver zu installieren.[2] Die folgende Grafik veranschaulicht die Serverkomponente des ODS.

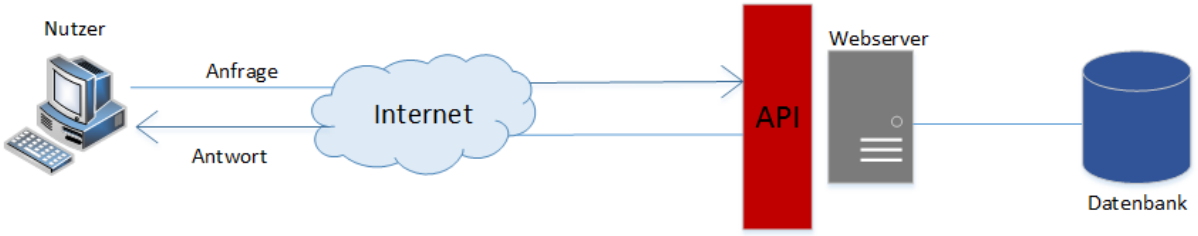

Abbildung 16: Übersicht Serverkomponente

2.4.5 Gesamtarchitektur des ODS

Nachdem nun alle Komponenten detailliert beschrieben worden sind, lässt sich Gesamtarchitektur des ODS vervollständigen. Man sieht die jeweiligen Funktionalitäten der einzelnen Komponenten und ihr Zusammenspiel. Die Importkomponente aggregiert die Daten, bereinigt sie und speichert diese zusammen mit den Originaldaten in der CouchDB-Datenbank. Von der anderen Seite kann der Nutzer über die REST-API der Serverkomponente auf die Nutz- und Metadaten zugreifen.

[1] http://restlet.com/
[2] http://tomcat.apache.org/

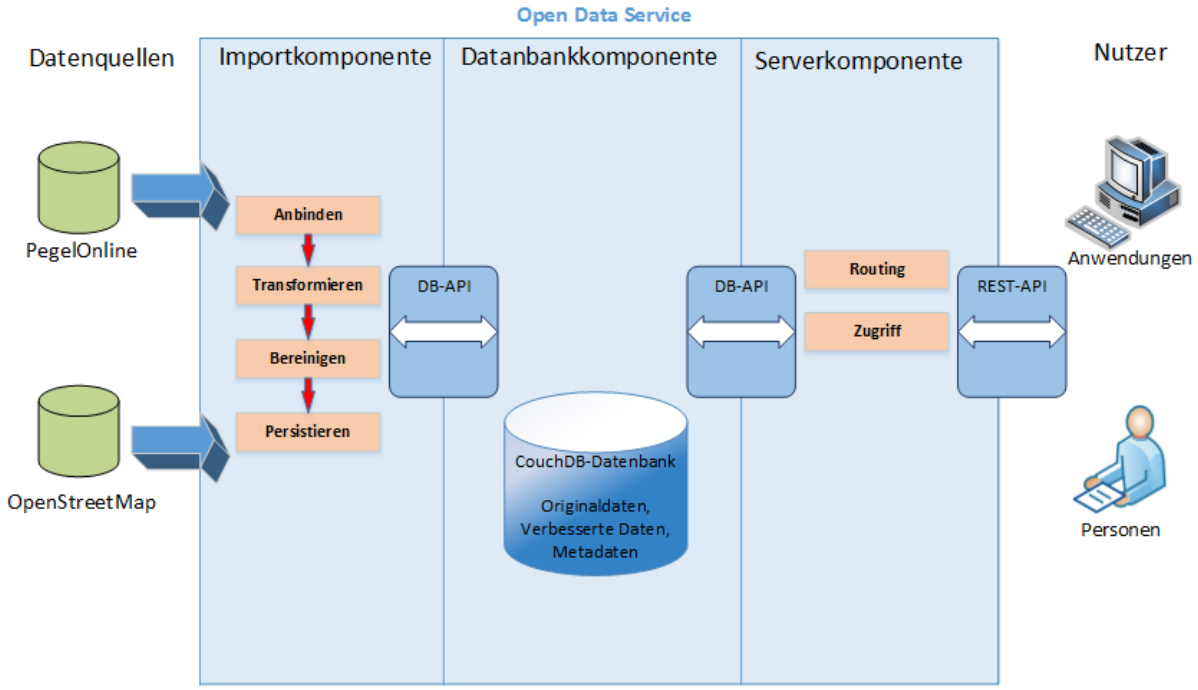

Abbildung 17: Gesamtarchitektur des Open Data Service

2.4.6 Reflektierende Abfrageschnittstelle (Query API)

Die Open Data Service Query API ermöglicht den Nutzern den lesenden Zugriff auf die Daten aus dem ODS. In diesem Abschnitt wird die auf REST basierende Query API ausgearbeitet. Die Hauptbestandteile bilden dabei der Namensraum, das Konzept der Abfragen sowie die Struktur der Daten und Metadaten.

Das Ziel der REST-API ist eine einheitliche Schnittstelle für den Zugriff der Clients auf alle relevanten Daten des ODS zu ermöglichen. Der Zugriff soll dabei schlicht und einfach gehalten sein und die Qualitätsmerkmale für Programmierschnittstellen berücksichtigen. Die REST-API entkoppelt den Client vom Server. Der Client kennt nur die API, Änderungen am Server, wie der Wechsel von Software oder Hardwarekomponenten, sind für den Client nicht sichtbar, solange die Schnittstelle sich nicht ändert. Die Kommunikation zwischen Client und der Serverkomponente erfolgt über das Hypertext Transfer Protocol (HTTP). Die Ressourcen lassen sich dann über HTTP-Methoden (GET, POST, PUT, DELETE) abrufen bzw. manipulieren.[1]

[1] http://www.vinaysahni.com/best-practices-for-a-pragmatic-restful-api

HTTP-Methode	**Auswirkung auf die Ressource**
GET	Rückgabe einer Ressource in einer bestimmten Repräsentation
PUT	Erstellen oder Ersetzen einer Ressource durch Mitgabe einer neuen Ressource
DELETE	Löschen eine Ressource
POST	Einer Ressource zusätzliche Informationen mitgeben

Tabelle 2: Übersicht der HTTP-Methoden

Die Kommunikationseinheiten in HTTP zwischen Client und Server werden als Nachrichten bezeichnet. Es gibt dabei zwei Arten: Die Anfrage (Request) vom Client an den Server und die Antwort (Response) als Reaktion darauf vom Server zum Client. Eine Nachricht besteht aus zwei Teilen, dem Nachrichtenkopf und dem Nachrichtenrumpf. Der Nachrichtenkopf enthält Informationen über den Rumpf, zum Beispiel die verwendete Kodierung, die Länge usw. Der Nachrichtenrumpf enthält schließlich die Nutzdaten. (Abts & Mülder, 1996)

Design und Abstraktion

Die REST-API folgt den REST-Prinzipien, die im Kapitel 2.1.2 über REST beschrieben sind. Das Hauptmerkmal dabei ist die Trennung der API in logische Ressourcen. Bei einer REST-API arbeitet man nicht mit Aktionen, sondern mit den Ressourcen.

Die vom ODS gesammelten Daten lassen sich in Ressourcen aufteilen. So bietet PegelOnline eine Liste von Stationen an. Jede Station enthält Attribute, wie Name, das Gewässer der Station, dessen Länge, den Zeitstempel usw. Wie granulös man diese Informationen aufteilt ist dem Entwickler überlassen. Allgemein lässt sich sagen, dass zusammengehörige Attribute einer Ressource zugewiesen sind. Dabei sollte die Ressource einem Nomen entsprechen. In unserem Beispiel wäre das die Station. (Masse, 2011) (Fielding & Taylor, 2002)

Die Ressourcen lassen sich über einheitliche Bezeichner für Ressourcen (URIs) identifizieren. Mit der URI kann die API die bestimmte Ressource identifizieren, die angefragt wurde. Dabei ist zu beachten, dass unterschiedliche URIs auf die gleiche Ressource zeigen können.

Für größere Applikationen wird empfohlen die API in eine eigene Subdomain zu legen. Dies wird bei ODS auch eingehalten. Der Einstiegspunkt für den ODS ist */api*. Die Datenquellen befinden sich alle unter */ods/<Datenquelle>*. Details sind unter dem Punkt Namensraum festgehalten. (Hunter, 2014)

Der Einstiegspunkt */api* enthält alle Informationen über die vorhandenen Zugriffspunkte. Die Beschreibung der Zugriffspunkte orientiert sich an der „URI Template“-Beschreibung (RFC6570).[1] Es handelt sich dabei um eine Vorlage für URIs und URLs, die eine gemeinsame Grundstruktur aufweisen. Im Punkt Metadaten wird diese Liste detailliert erläutert.

Anfrageergebnis

Die interne Datenstruktur der Ressource ist entkoppelt von ihrer Darstellung. Der ODS arbeitet mit unstrukturierten und meistens schemalosen Informationen. Durch das Datenformat JSON ist man dem Bedürfnis nach mehr Flexibilität nachgekommen. Deshalb wird beim ODS für die Darstellung der Ressourcen JSON benutzt. Dabei können die Ressourcen sowohl Nutz-, als auch Metadaten sein. Die Metadaten werden weiter unten differenziert betrachtet. Das Ergebnis einer Anfrage im ODS ist stets eine JSON-Struktur. (siehe Kapitel 2.1.3)

Beispiele:

- Ein einzelner Wert: *"Donau"*
  Dabei kann ein Wert auch ein Objekt sein

- Ein Objekt (Ungeordnete Menge von Schlüssel-Wert-Paaren):

```
{
    "Name": "Eitze",
    "Koordinate":
        {
            "Längengrad": 9.27676943537587,
            "Breitengrad": 52.90406541008721
        }
}
```

- Eine geordnete Liste von Werten: *["Eitze", "Rethem"]*

Anfrage

Die REST-API basiert auf HTTP und unterstützt die CRUD-Operationen. Diese stehen für Create, Read, Update und Delete. Die HTTP-Methoden GET, POST, PUT, DELETE repräsentieren diese Operationen. Beim ODS darf der Client nur lesend auf den Service

[1] https://tools.ietf.org/html/rfc6570

zugreifen. Es werden nur „HTTP GET“-Anfragen verarbeitet. Änderungen vom Client an den Ressourcen sind nicht möglich. Da REST zustandslos ist, müssen alle Informationen für die Bearbeitung der Anfrage in der Anfrage selbst enthalten sein.

Eine wichtige Eigenschaft für Anfragen in REST-APIs ist die Einheitlichkeit. Diese wird durch die Einhaltung der REST-Konventionen gefördert. Das erleichtert es Entwicklern, die API zu nutzen. So sollte eine Anfrage an *.../stations/* eine Liste aller Stationen liefern. Ein Aufruf *.../stations/Eitze* sollte ein Objekt, die Station Eitze selbst, zurückliefern. Dabei sollte das Objekt jeweils gleich ausschauen, unerheblich ob der Aufruf über alle Stationen genutzt worden ist oder über die konkrete eigene URL. Beide Repräsentationen sollten gleich sein, also die gleichen Attribute besitzen.[1]

Die API unterstützt neben dem Zugriff auf einzelne Ressourcen auch den Zugriff auf Attribute der Ressourcen. Der Zugriff sieht so aus:

/ods/<DATENQUELLE>/<RESSOURCE>/<ATTRIBUT>.

So kann man über */ods/de/pegelonline/stations/EITZE/timeseries* die Zeitreihen der gemessenen Wasserstände an der Station EITZE abrufen. Zu jeder Datenquelle gibt es auch die Möglichkeit das Schema und die Metadaten abzurufen. Über */ods/<DATENQUELLE>/<RESSOURCE>/$class* kann man das Schema abrufen, welches die Struktur der Ressource beschreibt. Über */ods/schema* lassen sich alle vorhandenen Schemata abrufen. Analog ruft man über */ods/<DATENQUELLE>/metadata* die Metadateninformationen zu der Quelle ab.

Die Nutzdaten einer Datenquelle haben stets einen Datentyp. Die API unterstützt eine Filterung nach Datentypen über Abfragen der Form */ods?dataType={dataType}*. Eine Abfrage über alle Stationen sieht beispielsweise so aus: */ods?dataType=Station*

Einer der funktionalen Anforderungen war, dass es eine Möglichkeit geben soll, auch auf die Rohdaten zugreifen zu können. Dies lässt sich über den Parameter *dataQualityStatus=raw* durchführen. So kann man von jeder Ressource die nicht verbesserte Version abfragen. Dies gilt auch für Schemata. Beispielsweise werden die Rohdaten zur Station RETHEM aufgerufen mit: */ods/de/pegelonline/stations/RETHEM?dataQualityStatus=raw*

Jede Ressource im ODS besitzt eine eindeutige globale Id als Attribut *_id*. So kann der Nutzer schnell nur die benötigten Daten einer Ressource abrufen. Dies spart Bandbreite und er muss sich die vollständige URI nicht merken. Der Abruf der Ressource über die Id erfolgt über */ods/$<ID>*. Auch auf Schemata und Metadaten lässt sich einheitlich über die Id zugreifen.

[1] http://dev.billysbilling.com/blog/How-to-make-your-API-better-than-the-REST

Metamodell

Im Gegensatz zu SOAP hat man bei REST über HTTP keine standardisierte Möglichkeit, die vom Web Service angebotenen Methoden zu erfragen. Es gibt also drei Unbekannte für den Client:

- Die verfügbaren Ressourcen und deren URIs zum Abrufen
- Auf die Ressourcen anwendbare HTTP-Methoden
- Für die Interpretation notwendige Struktur der Rückgabe-Objekte

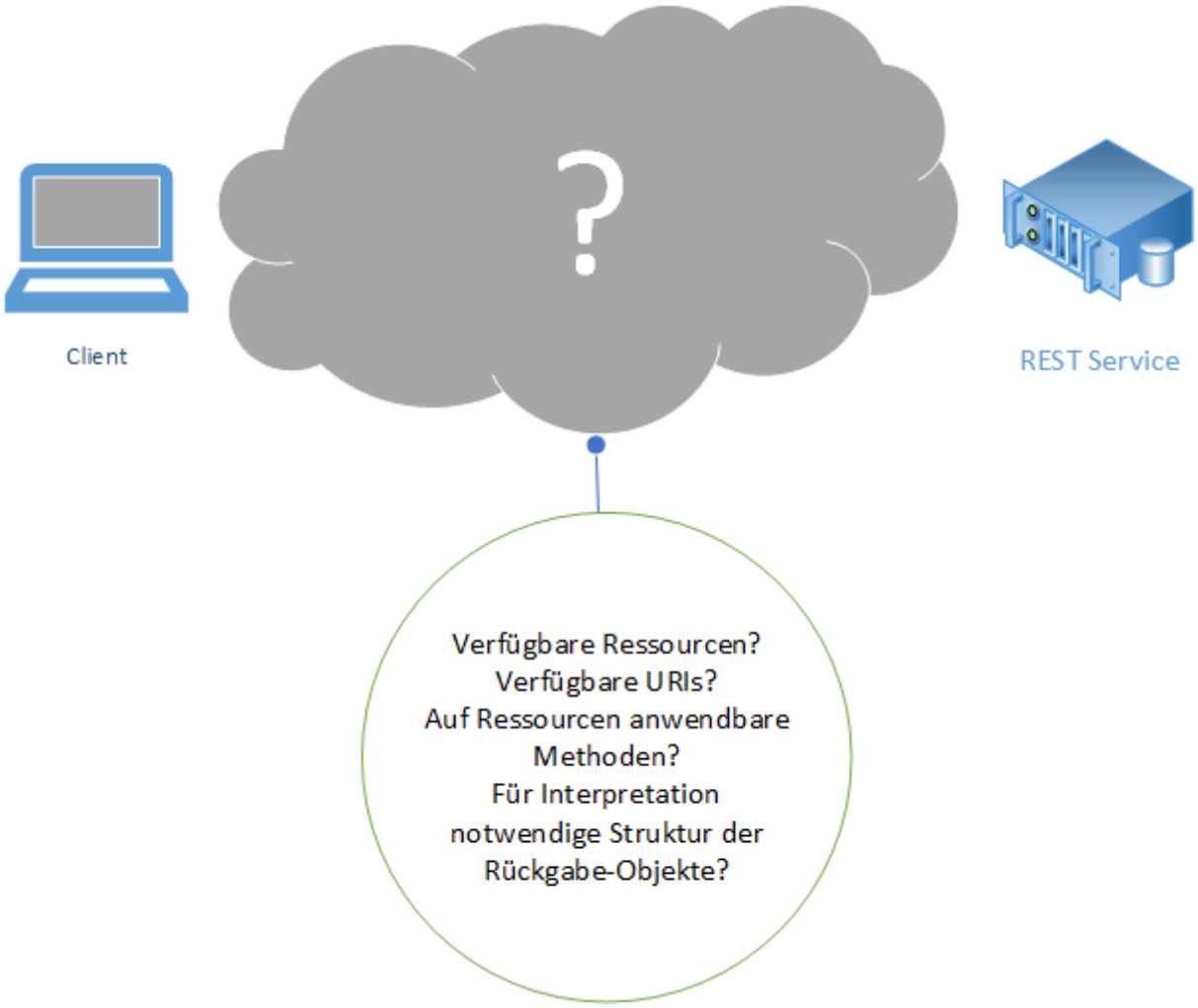

Abbildung 18: Die drei Unbekannten für den Client

Der Abruf der anwendbaren HTTP-Methoden wird beim ODS nicht benötigt. Wie im Punkt Anfrage beschrieben, sind derzeit nur lesende Abfragen erlaubt. Dafür reichen alleine GET-Methoden.

Eine Übersicht der verfügbaren Datenquellen und ihrer Ressourcen kann abgerufen werden über */api*. Für jede Datenquelle existieren Metainformationen, welche die Quelle beschreiben. Diese enthalten die Beschreibung der Quelle, den Autor, die Nutzungsbedingungen und die Webseite des Anbieters. Mit dieser Information kann der Nutzer Informationen über die Quelle

abrufen. Außerdem gibt es zu jeder abfragbaren Ressource der Quelle ein Schema. Dieses beschreibt die Struktur der Ressource. Der Nutzer kann mit diesem Schema die Rückgabe-Objekte interpretieren und beispielsweise validieren. Die Schemata sind im JSON-Schema-Format. Das JSON-Schema-Format basiert auf JSON und ist ein Standard zur Beschreibung von JSON-Dokumenten. Der Client kann so verbreitete Bibliotheken oder Programme nutzen, um die Daten und das Schema einfach zu editieren und zu validieren.[1]
Diese Metainformationen bilden die Fähigkeit des ODS, die eigenen Daten reflektiert zu beobachten (Introspection). Sowohl die Schemata, als auch die Metadaten sind als Ressourcen im ODS hinterlegt. Der Nutzer kann die Nutzdaten und die Metadaten abrufen. Er bekommt so selbstbeschreibende Datensätze. Das ODS-System repräsentiert die Metadaten im gleichen Datenbankmodell wie Nutzdaten. Solche Systeme nennt man, wie in Kapitel 2.2.2 beschrieben, selbstreferentiell.

Namensraum

Die einzelnen extrahierten Datenquellen werden in der API abgebildet. Die vollständige Form ist */ods/<DATENQUELLE>/<RESSOURCE>/<ATTRIBUT>*.
Allgemein verweisen alle URLs, die mit */ods/* anfangen auf Datenquellen. Dies trennt den Zugriff auf die Datenquellen von anderen Teilen des Systems. Zugriffe wie der Abruf aller Schemata über */ods/schema* oder die Filterung nach Daten bestimmten Typs über */ods?dataType={dataType}* sind globale Zugriffsmöglichkeiten auf alle Datenquellen. Auch der Zugriff auf eine bestimmte Ressource mit der global eindeutigen Id über */ods/<ID>* ist eine globale Zugriffsmöglichkeit. Zugriffe auf einzelne Ressourcen einer Datenquelle haben die Form */ods/<DATENQUELLE>/<RESSOURCE>*. Sie sind hierarchisch eine Stufe tiefer im Namensraum.
Die Benennung der Datenquelle ist orientiert an der Namensgebung von Paketen bei Java. Die Konventionen dabei sind, dass die Paketnamen alle klein geschrieben sind und ihren Domainnamen rückwärts schreiben. So wird aus *pegelonline.de* das der Paketname *de.pegelonline*. Da jedoch der Punkt in einer URL für die Trennung der Domain steht, wurde beim ODS der Slash (‚/') als Trennzeichen gewählt. So wird aus pegelonline.de die Datenquelle de/pegelonline. Die Übersicht der verfügbaren Datenquellen und ihrer Ressourcen kann über */api* abgerufen werden.

[1] http://json-schema.org

Statuscodes und Fehlerbehandlung

Im Erfolgsfall bekommt der Client den „Statuscode 200 OK“ und die Daten zurück. Im Fehlerfall wird ein anderer Statuscode zurückgegeben und wenn möglich wird in der Nachricht auch der Fehler beschrieben.

Anfrage-URL	**Fehler-Statuscode**	**Nachricht**
/ods/de/pegelonline/nichtexistierend	400 Invalid Request	{"message":"Invalid query, see /api for available queries","status":400}
/ods/de/pegelonline/stations/nichtexistierend	500 Internal Server Error	
ods/$ungueltigeId	404 Not Found	{"message":"No data found for id 'ungueltigeId'","status":404}

Tabelle 3: Beispiele für die Fehlerbehandlung

3 Umsetzung

In diesem Kapitel wird die prototypische Implementierung des Open Data Service vorgestellt. Der Open Data Service wurde in Java entwickelt und als Build-Management-Automatisierungstool kam Gradle zur Verwendung. Das Projekt wurde auf GitHub, einem Hosting-Dienst für Software-Entwicklungsprojekte veröffentlicht und kann unter folgender URL aufgerufen werden: https://github.com/jvalue/open-data-service

Wie in Kapitel 2.4.1 beschrieben, besteht der ODS aus drei Teilen, der Import-, Datenbank- und Serverkomponente. In diesem Abschnitt werden die einzelnen Komponenten erläutert und anschließend die Implementierung der REST-API aus Kapitel 2.3 präsentiert. Im Folgenden werden die wichtigsten Aspekte des Open Data Service beschrieben.

3.1 Importkomponente

Die Importkomponente stellt den ETL-Prozess dar (siehe Kapitel 2.4.2) und besteht aus den nacheinander gelagerten Schritten Anbinden, Transformieren, Bereinigen und Persistieren. Dies geschieht mit der Umsetzung des „Pipes und Filter“-Architekturmusters.

Zu jeder Quelle existiert eine Konfiguration. Sie werden durch Quellenkonfigurationsklassen realisiert. Diese Konfigurationsklassen beschreiben jeweils genau eine Quelle. In der Konfigurationsklasse werden die einzelnen Filter hintereinandergeschaltet, die für die Anbindung der Quelle nötig sind. Diese hintereinandergeschaltete Struktur der Filter wird als Filterkette bezeichnet.

Die Konfigurationsklassen implementieren das *Configuration*-Interface und implementieren damit die Methoden *getFilterChain()* und *getDataSource()*.

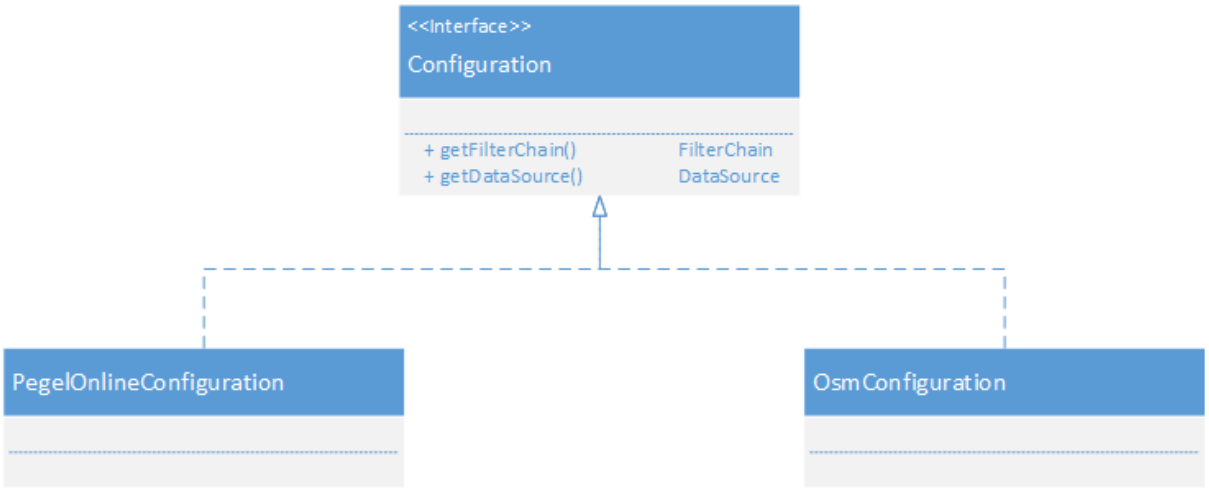

Abbildung 19: Klassendiagramm der Konfiguration

Wie der Name schon sagt gibt *getFilterChain()* die zuvor beschriebene Filterkette zurück. *getDataSource()* instanziiert die *DataSource*-Klasse und gibt sie zurück. *DataSource* ist eine Klasse, die alle Informationen zum Anbinden und Verwenden der Quelle beinhaltet. Sie enthält eine Id, die URL, über die die externen Daten der Datenquelle angebunden werden, Metadaten über die Quelle, drei Schemata und vorgefertigte Datenbankabfragen für CouchDB. Diese Datenbankabfragen werden Views genannt und bei der Serverkomponente im Kapitel 3.3 erklärt. Die drei Schemata unterteilen sich in ein externes, ein nicht verbessertes und ein verbessertes Schema. Das externe Schema beschreibt die Datenstruktur der zu importierenden Daten. Die nicht verbesserte und das verbesserte Schemata beschreiben das dem Client später angebotene Schema beim Zugriff über die REST-API.

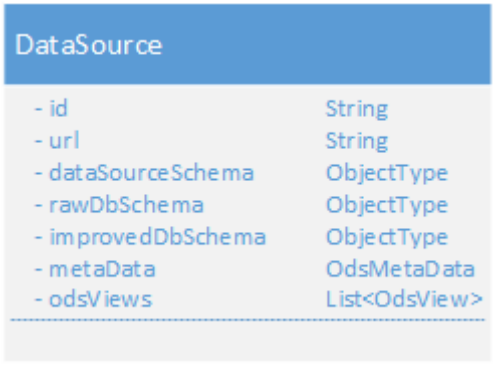

Abbildung 20: Klassendiagramm von DataSource

Um zu verstehen wie die Filterkette (FilterChain) funktioniert, ist es sinnvoll den Codeausschnitt der *getFilterChain()*-Methode zu betrachten.

```
@Override
public FilterChain<Void, ?> getFilterChain(DbAccessor<JsonNode> accessor) {
    DataSource source = getDataSource();

    FilterChain<Void, File> chain = FilterChain.instance(GrabberFactory.getResourceGrabber(source));
    chain.setNextFilter(TranslatorFactory.getOsmTranslator())
            .setNextFilter(new DataAdditionFilter(source))
            .setNextFilter(new DbInsertionFilter(accessor, source));
    return chain;
}
```

Abbildung 21: Codeausschnitt der FilterChain

Diese konkrete Methode entstammt der *OsmConfiguration*-Klasse. In der Methode wird das zuvor beschriebene quellspezifische *DataSource*-Objekt angefordert. Anschließend wird die Filterkette instanziiert und die Filter werden hintereinander geschaltet. Um die *OpenStreetMap*-Daten in den ODS zu importieren, wird zunächst der *ResourceGrabber*-Filter angewandt. Dieser liest die OSM-Daten aus einem lokalen Dateipfad ein. Anschließend überführt der *OsmTranslator*-Filter die eingelesenen Daten in das intern für die Datenbank passende JSON-Format. Diese JSON-Daten werden im DataAdditionFilter um einige Attribute ergänzt, die für

Datenbankabfragen nötig sind. Schließlich kommt der *DbInsertion*-Filter zur Anwendung, der die JSON-Daten und die Schemata aus dem *DataSource*-Objekt in die Datenbank schreibt. Die Methode bekommt als Parameter eine Referenz zur Datenbank-Schnittstelle vom Typ *DbAccessor*. Diese Referenz nutzt der *DbInsertionFilter*, um die Daten im letzten Filterschritt in die Datenbank zu schreiben.

Eine Qualitätsverbesserung findet also beim Importieren von OSM-Daten nicht statt. Beim Import von PEGELONLINE-Daten ist vor dem Schreiben in die Datenbank noch ein Qualitätsverbesserungs-Filter des Typs *PegelOnlineQualityAssurance* dazwischen geschaltet. Die Filterkettenstruktur ist dynamisch. Man kann die Reihenfolge der Filter beliebig setzen. Alle *Grabber*-, *Translator*-, und der *DbInsertion*-Filter implementieren das Filter-Interface. Eine mögliche Aneinanderreihung der Filterkette ist im folgenden Diagramm dargestellt.

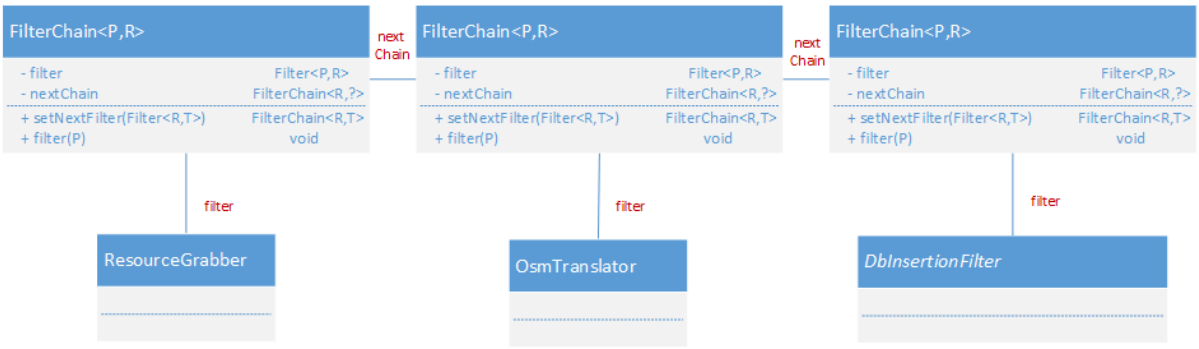

Abbildung 22: Klassendiagramm der Filterkette

Die *Grabber* und *Translatoren* sind abstrakte Klassen und entsprechen dem Anbinden und Transformieren in der Filterkette. Der Grabber, spezialisiert darauf Daten eines bestimmten Formates zu extrahieren, hat die spezifischen Unterklassen *JsonGrabber*, *XmlGrabber* und *ResourceGrabber*. Dazu hat der *Translator* die abgeleiteten Klassen *JsonTranslator*, *XmlTranslator* und *OsmTranslator*, die für die Überführung der JSON-, XML- und OSM-Formate in das für die Datenbank passende Format sorgen. Dies ist im folgenden Klassendiagramm veranschaulicht.

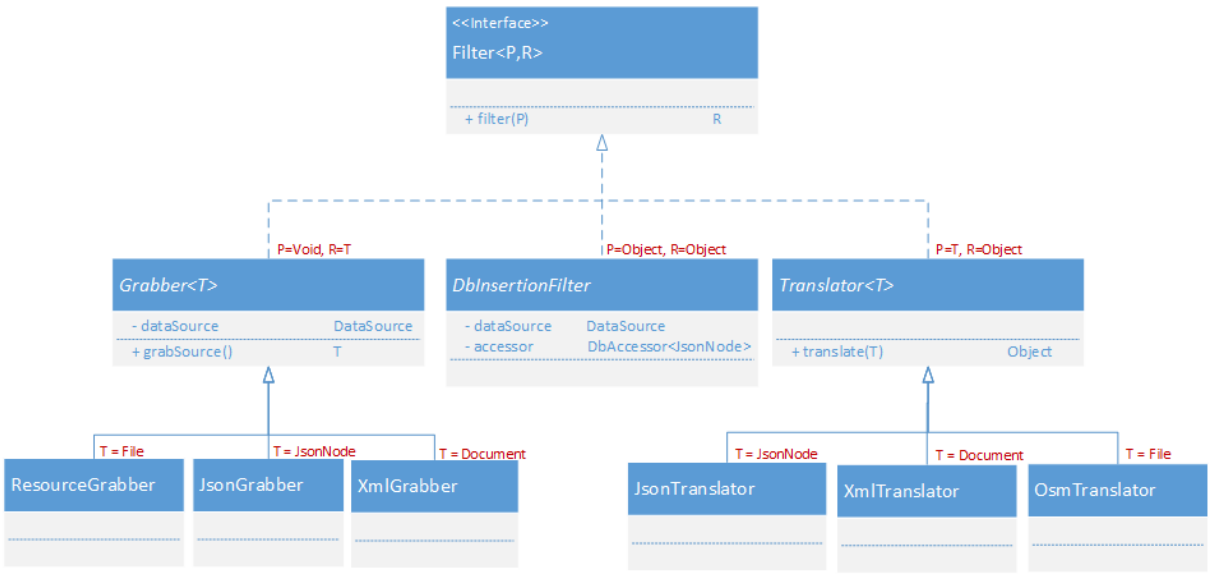

Abbildung 23: Klassendiagramm der Filter

Das Filter-Interface ist parametrisierbar. Dabei steht P für den eingehende Datentyp und R für den ausgehenden Datentyp. *Grabber*, die die Daten von der Quelle einlesen, haben keinen einkommenden Datenstrom, deshalb ist bei ihnen die Typ-Variable P mit *Void* belegt. Der Rückgabetyp der *Grabber* ist wie bereits beschrieben durch den jeweilig konkreten *Grabber* festgelegt. Analog dazu hat der *Translator* als Eingabedaten den vom konkreten *Translator* bestimmten Typ T und als Ausgabedaten das überführte und für die Datenbank passende *Object*. Im konkreten Fall wird für *Object* eine JSON-kompatible Datenstruktur benutzt. Schließlich bekommt der *DbInsertionFilter* als Eingabedaten das *Object* und schreibt dieses in die Datenbank. Der *DbInsertionFilter* gibt daraufhin das *Object* wieder zurück. Dies ist notwendig, da es den Fall gibt, dass nach dem Speichern der Rohdaten in die Datenbank die Daten durch einen qualitätsverbessernden Filter gereicht werden, bevor dann die verbesserten Daten in die Datenbank geschrieben werden.

Die Filterketten werden beim *FilterChainManager* registriert. Jede Filterkette entspricht dabei der ETL-Phase einer Datenquelle. Wenn im *FilterChainManager* die Methode *startFilterChains()* aufgerufen wird, ruft der *FilterChainManager* iterativ jede Filterkette auf. Dies führt dazu, dass alle Daten von den Datenquellen neu abgerufen bzw. aktualisiert werden. Im folgenden Sequenzdiagramm wird ein solches Update visualisiert.

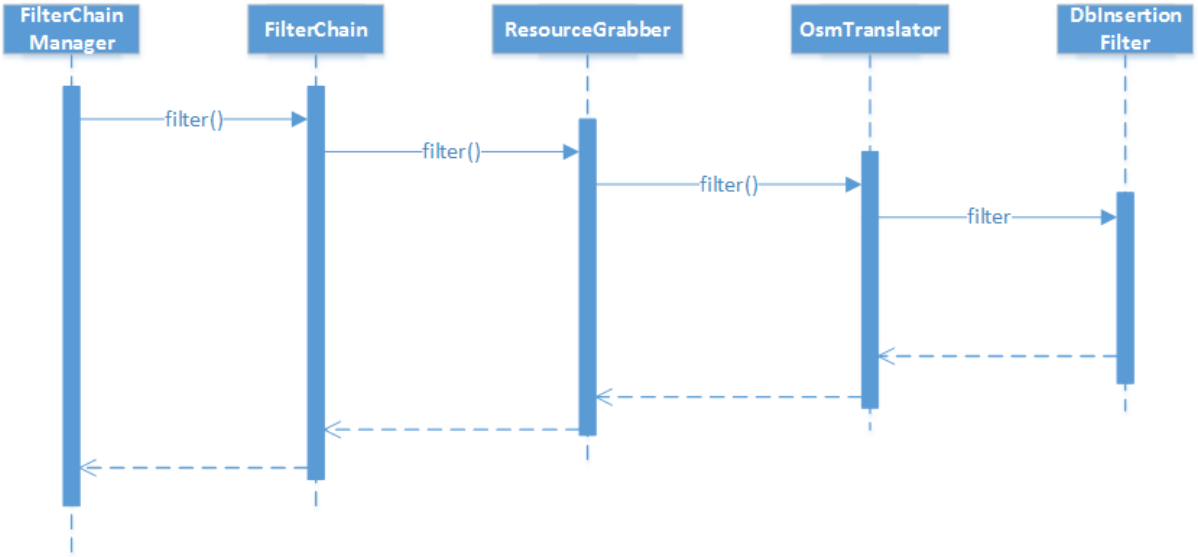

Abbildung 24: Sequenzdiagramm der Filterkette

3.2 Datenbankkomponente

Die Datenbankkomponente besteht aus der Datenbankschnittstelle und der die Schnittstelle implementierenden CouchDB-Zugriffs-Komponente. Die Datenbankkomponente ist für die Speicherung und die Bereitstellung der gesammelten Daten zuständig.

Die Schnittstelle stellt alle Methoden zur Verfügung, um Daten in die Datenbank zu schreiben und zu lesen. Sie abstrahiert von der konkreten Umsetzung, so dass es möglich ist die darunterliegende konkrete Datenbank zu wechseln. Derzeit implementieren zwei Klassen die Datenbankschnittstelle *DbAccessor*. Der *CouchDbAccessor* implementiert die Schnittstelle und ermöglicht den Zugriff auf die CouchDB-Datenbank. Außerdem existiert ein *MockDbAccessor*. Dieser simuliert eine echte Datenbank und wird bei den Unit-Tests benutzt, um Funktionalität testen zu können, ohne eine echte Datenbank einsetzen zu müssen.

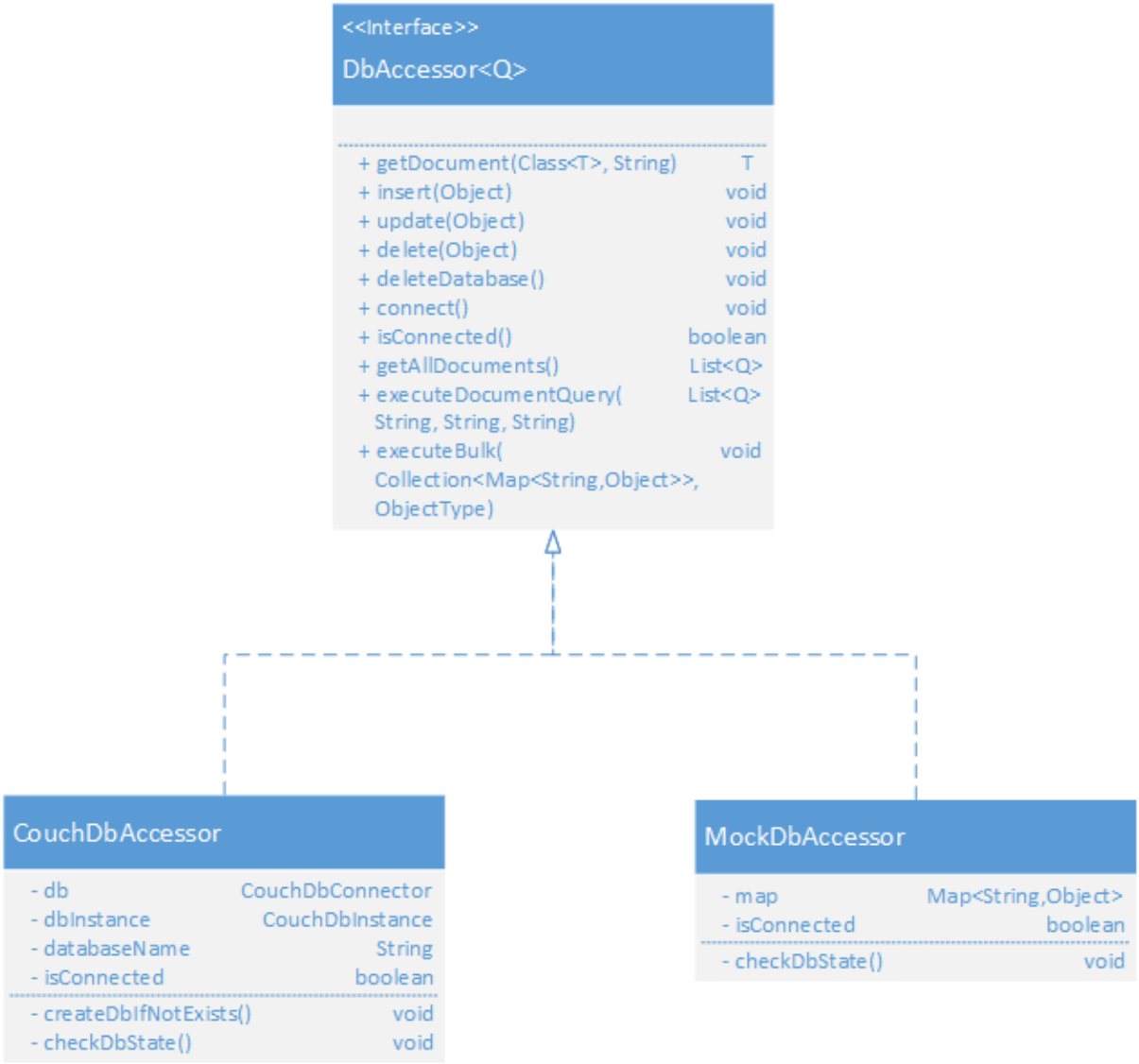

Abbildung 25: Klassendiagramm der DbAccessor

Mit der Methode *connect()* stellt ein *DbAccessor* die Verbindung zur Datenbank her. Über *isConnected()* lässt sich der derzeitige Status der Verbindung abfragen. Mit *getAllDocuments()* lassen sich alle Dokumente in der Datenbank abrufen und mit *getDocument()* kann genau ein Dokument mit einer bestimmten Id aus der Datenbank gelesen werden. Die Methoden *insert(), update()* und *delete()* sind für das Einfügen, Aktualisieren und Löschen von Dokumenten zuständig. Um eine Query auf der Datenbank auszuführen, benutzt man *executeDocumentQuery()*. Aus Performance-Gründen gibt es noch eine *executeBulk()*-Methode. Diese ermöglicht es, mehrere Dokumente auf einmal in die Datenbank zu schreiben. Schließlich existiert noch *deleteDatabase()*. Diese Methode löscht die Datenbank. Sie wird im Produktiveinsatz selten benutzt, dagegen aber oft bei Unit-Tests.

Die Erzeugung des konkreten *DbAccessors* wird durch ein Fabrik-Entwurfsmuster entkoppelt. Die *DbFactory*-Klasse bietet Methoden zur Bereitstellung der verschiedenen *DbAccessor*-

Implementierungen. So geben die Methoden *createDbAccessor()* und *createMockDbAccessor()* einen *DbAccessor* zurück, in unserem Fall ist es der *CouchDbAccessor* bzw. der *MockDbAccessor*. Die verschiedenen Klassen verwenden die *DbFactory*, wenn sie den *DbAccessor* für den Datenbankzugriff benötigen. Die konkrete Implementierung ist aber für die Nutzer der Komponente nicht sichtbar. (Gamma, Helm, Johnson, & Vlissides, 1994)

Der *CouchDbAccessor* implementiert das *DbAccessor*-Interface und stellt die Verbindung zur CouchDB-Datenbank her. Intern nutzt er für die Verbindung das Framework Ektorp. Ektorp stellt eine einfache und mächtige Schnittstelle bereit, die es ermöglicht Objekte in CouchDB zu persistieren. Es unterstützt die CRUD-Operationen und bietet Funktionalität zum Verwalten der Dokumente und Abfragen.[1]

3.3 Serverkomponente und REST-API

Die Serverkomponente und die REST-Schnittstelle sind eng gekoppelt. Sie sind für das Routing und den Zugriff von außen auf die Daten im ODS zuständig. In der ServerMain-Klasse wird der ODS-Web-Service auf dem Port 8182 gestartet. Dabei wird eine neue Instanz der ContainerRestletApplication erstellt und initialisiert. Außerdem wird asynchron ein Thread gestartet, um die Datenquellen zu aktualisieren. Die ContainerRestletApplication nutzt die Restlets, um die ankommenden Anfragen zu verarbeiten und an die richtigen Verarbeiter weiterzuleiten. Beim ODS gibt es für jede Quelle eine entsprechende Router-Klasse. So gibt es beispielsweise die *PegelOnlineRouter*-Klasse für PegelOnline-Anfragen und die *OsmRouter*-Klasse für Anfragen, die OpenStreetMap-Daten betreffen. Alle Router-Klassen implementieren das Router-Interface. Das Router-Interface gibt eine Methode *getRoutes()* vor, die die Abbildungen der Anfrage-URLs auf die Verarbeitungsklassen darstellt.

[1] http://ektorp.org/

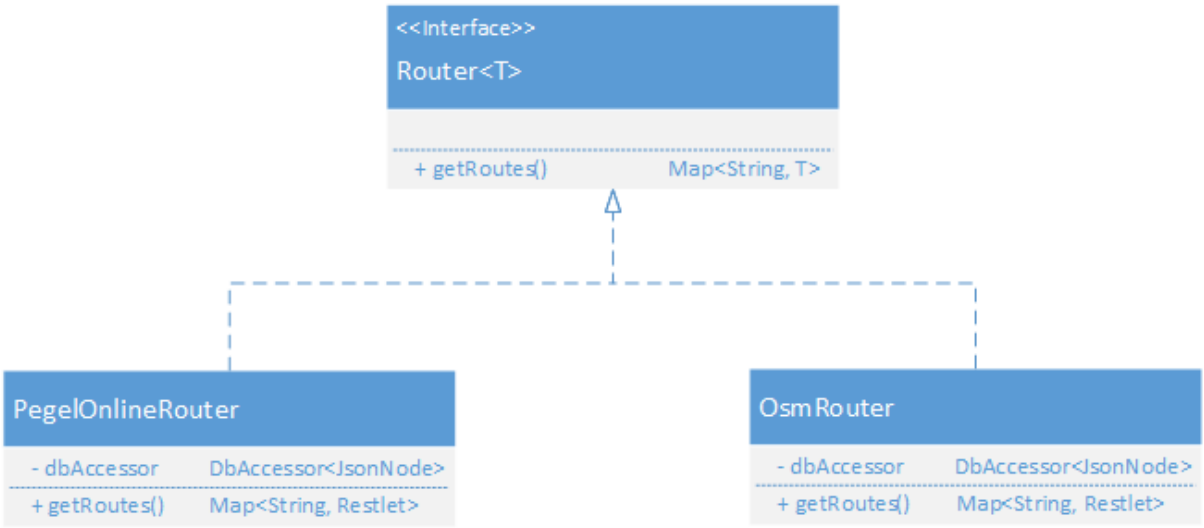

Abbildung 26: Klassendiagramm der Router

Die Router werden über die RouterFactory erzeugt. Die RouterFactory nutzt das Fabrik-Entwurfsmuster und bietet Methoden zur Bereitstellung der verschiedenen konkreten Router-Implementierungen. So geben die Methoden *createOsmRouter()* und *createPegelOnlineRouter()* einen *Router* zurück. Neben den quellspezifischen Routern existieren auch übergreifende Router, die übergreifende Zugriffe handhaben. Ein Beispiel dafür ist der ApiRouter, der alle unterstützten Query-URLs liefert oder der OdsRouter, der Zugriffe über die Id und die Auflistung aller Schemata erfüllt.

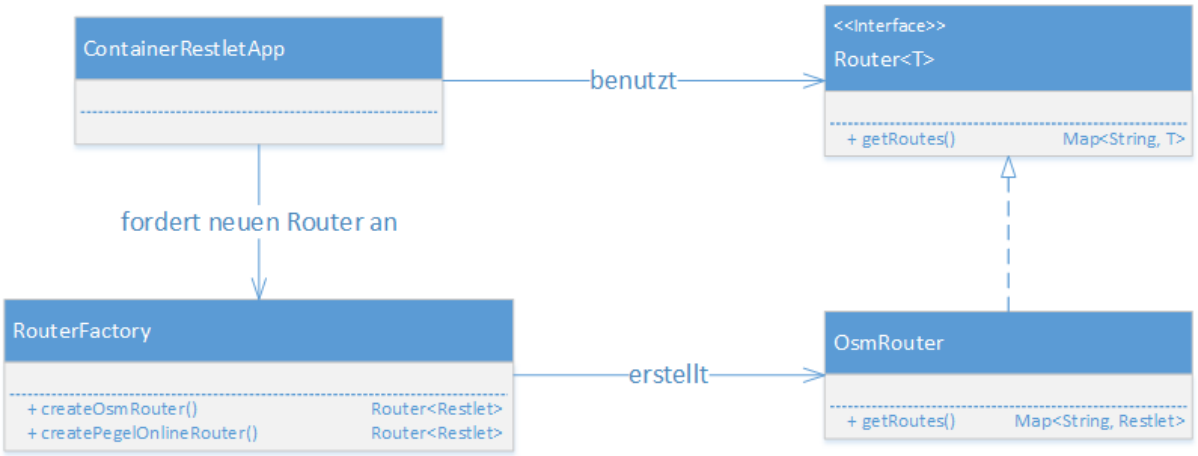

Abbildung 27: Fabri-Entwurfsmuster der Router

Im Folgenden eine Auflistung der Anfrage-URLs und der entsprechenden verantwortlichen Router:

Router	**Anfrage-URL**
OdsRouter	/ods/${id}
	/ods
	/ods/schema
ApiRouter	/api
OsmRouter	/ods/de/osm/nodes/{osm_id}
	/ods/de/osm/ways/{osm_id}
	/ods/de/osm/relations/{osm_id}
	/ods/de/osm/data/{osm_id}
	/ods/de/osm/keyword/{osm_keyword}
	/ods/de/osm/metadata
	/ods/de/osm
	/ods/de/osm/$class
	/ods/de/osm/$class_id
PegelOnlineRouter	/ods/de/pegelonline/stations
	/ods/de/pegelonline/stationsFlat
	/ods/de/pegelonline/stations/$class
	/ods/de/pegelonline/stations/$class_id
	/ods/de/pegelonline/stations/{station}
	/ods/de/pegelonline/metadata

Tabelle 4: Liste der URLs und verantwortlicher Router

In jeder dieser beschriebenen Routerklassen wird eine Anfrage-URL einem Restlet zugeordnet. Die Restlets sind die Verarbeiter der Anfragen. Sie bearbeiten Anfragen der Clients und liefern dynamische Antworten. Die Restlets haben alle die abstrakte Oberklasse *BaseRestlet*, die wiederum von Restlet erbt. In *BaseRestlet* wird die Methode *handle()* zum Abarbeiten einer Anfrage überschrieben. Diese stellt somit das Standardverhalten für die von *BaseRestlet* abgeleiteten Klassen dar. Das Verarbeiten wird in drei Teile geteilt. Zuerst werden in der Methode *handleRequest()* die Parameter extrahiert und geprüft ob diese valide sind. Bei ungültigen Parametern wird dem Client der Statuscode für „Bad Request“ zurückgegeben. Wenn die Parameter gültig sind, wird geprüft, um welche HTTP-Methode es sich handelt. Der

ODS erlaubt nur GET-Methoden. Wenn es eine GET-Methode ist, wird die Methode *doGet*() aufgerufen, welche die Anfrage weiterverarbeitet. Im *BaseRestlet* gibt *doGet*() eine Fehlermeldung zurück. Dies hat den Grund, dass die Unterklassen diese Methode überschreiben sollen, um eine richtige Verarbeitung der Anfrage zu gewährleisten.

```
@Override
public final void handle(Request request, Response response) {
    RestletResult result = handleRequest(request);
    response.setStatus(result.getStatus());
    if (result.getData() != null) {
        response.setEntity(result.getData().toString(), MediaType.APPLICATION_JSON);
    }
}


private final RestletResult handleRequest(Request request) {
    // validate params
    Set<String> paramNames = request.getResourceRef().getQueryAsForm().getNames();
    for (String param : mandatoryQueryParams) {
        if (!paramNames.remove(param)) {
            return onBadRequest("missing query param " + param);
        }
    }
    if (!allowAdditionalParams && paramNames.size() > 0) {
        return onBadRequest("found unknown query params");
    }

    // validate method type
    Method method = request.getMethod();
    if (method.equals(Method.GET)) return doGet(request);
    else return onInvalidMethod(method);
}
```

Abbildung 28: Codeausschnitt der Abarbeitung einer Anfrage

Die meisten Restlets interpretieren die Abfrage und nutzen die Datenbank-Schnittstelle, um die Daten und Schemata abzufragen. Dafür stehen auch in CouchDB vordefinierte CouchDB-Datenbankabfragen bereit. Diese vordefinierte Abfragen, sogenannte Views, werden in Design-Dokumenten gespeichert. Die Views sind JavaScript-Funktionen, die es ermöglichen MapReduce-Verfahren auf den Dokumenten anzuwenden. MapReduce ist ein besonderes Programmiermodell für nebenläufige Berechnungen über große Datenmengen. Die Views sind für jede Quelle spezifisch. Sie liegen in der quellspezifischen Konfiguration in der DataSource-Klasse vor und werden beim Anbinden der Quelle in die CouchDB-Datenbank geschrieben. Wenn der Nutzer eine Anfrage absetzt, die eine View aufruft, nimmt CouchDB den Java Script-Code und führt diesen für jedes Dokument in der Datenbank aus. CouchDB ist aus Gründen der Performance so aufgebaut, dass alle Dokumente nur dann durchlaufen werden, wenn die

View das erste Mal aufgerufen wird.[1]

```
getSingleStation
    map "function(doc) { if(doc.dataType == 'Station' && doc.dataQualityStatus == 'improved') emit(doc.longname, doc)}"
getAllStationsFlat
    map "function(doc) { if(doc.dataType == 'Station' && doc.dataQualityStatus == 'improved') emit (null, doc.longname) }"
getAllStations
    map "function(doc) { if(doc.dataType == 'Station' && doc.dataQualityStatus == 'improved')emit (null, doc) }"
getStationId
    map "function(doc) { if(doc.dataType == 'Station' && doc.dataQualityStatus == 'improved') emit (doc.longname, doc._id) }"
```

Abbildung 29: Darstellung der Views eines Design-Dokoments

Zum Abrufen einer vodefinierten CouchDB-Datenbankabfrage existiert der *ExecuteQueryRestlet*. Er führt die Datenbankabfrage aus, indem er die Datenbankschnittstelle (*DbAccessor*) und die erwähnte Methode *executeDocumentQuery()* nutzt. Die dafür benötigten Parameter bekommt der *ExecuteQueryRestlet* im Konstruktor übergeben. In der folgenden Abbildung wird das Zusammspiel visualisiert.

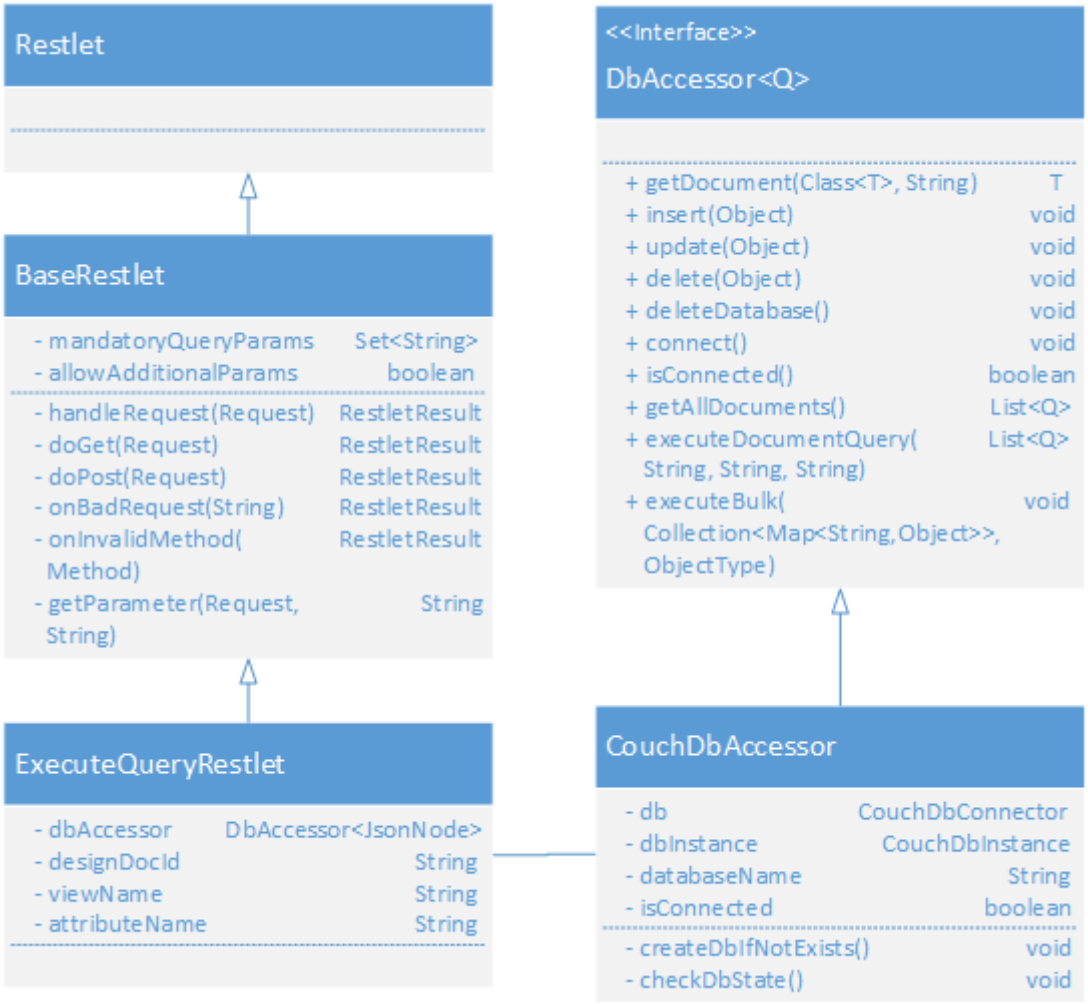

Abbildung 30: Zusammenspiel von ExecuteQueryRestlet und CouchDbAccessor

[1] http://guide.couchdb.org/editions/1/de/views.html

Bisher wurde beschrieben wie die Client-Anfragen verarbeitet werden. Neben den Nutzdaten spielen aber auch die Metadaten eine wichtige Rolle. Sie erlauben es dem Nutzer die verfügbaren Daten zu interpretieren. Der Abruf der Metadaten erfolgt identisch zu den Nutzdaten über die REST-API als Zugriff auf eine Ressource. Es gibt zwei Arten der Metadaten. Die Metadaten zu einer Quelle, welche die Quelle beschreiben und die Schemata, welche die Struktur der Nutzdaten beschreiben. Die Metadaten und die Schemata sind Teil der *DataSource*-Klasse. Diese enthält alle Informationen zum Anbinden der Quelle und wurde im im Kapitel 3.1 vorgestellt.

Die Metadaten zu einer Quelle werden durch die Klasse OdsMetaData modelliert. Sie enthält Informationen über Namen, Titel, Autor, E-Mail des Autors, die URL und die Nutzungsbedingungen. Die Metadatenstruktur orientiert sich dabei an den CKAN-Metadaten.[1]


```
{
  "_id" : "7276728344ac78c1b00d7766dda0841b",
  "_rev" : "1-c1bfe988c30fdddf734e68caa5e5c01b",
  "author" : "Wasser- und Schifffahrtsverwaltung des Bundes (WSV)",
  "author_email" : "https://www.pegelonline.wsv.de/adminmail",
  "date" : "2014-12-29 14:22:13.741",
  "name" : "de-pegelonline",
  "notes" : "PEGELONLINE stellt kostenfrei tagesaktuelle Rohwerte verschiedener gewässerkundlicher Parameter (z.B.
Wasserstand) der Binnen- und Küstenpegel der Wasserstraßen des Bundes bis maximal 30 Tage rückwirkend zur Ansicht und
zum Download bereit.",
  "terms_of_use" : "http://www.pegelonline.wsv.de/gast/nutzungsbedingungen",
  "title" : "pegelonline",
  "url" : "https://www.pegelonline.wsv.de"
}
```


Abbildung 31: Metadaten zu PegelOnline

Die Schemata beschreiben zum einen die Datenstruktur der zu importierenden Daten und zum anderen die Datenstruktur der für den Client angebotenen Daten. Die importierten Daten werden im ODS durch die Translator-Klassen in ein internes Datenmodell konvertiert, welches dem JSON-Format entspricht. Deshalb basieren die Schemata auf dem JSON-Schema-Format. So lassen sich die importierten Daten einfach validieren und der Client bekommt die gültigen Daten und das dazu passende Schema im verbreiteten JSON-Schema-Format.

Die auf JSON basierende Schemastruktur zur Validierung ist ähnlich der internen Datenmodellierungsstruktur. So beschreibt das Schema die Datentypen der JSON-Struktur und besteht aus *BoolType*-, *ListType*-, *MapType*-, *NullType*-, *NumberType*- und *StringType*-Klassen. Als Oberklasse all dieser Klassen dient die *Type*-Klasse.

Die Validierung der eingelesenen Daten mit dem Schemata wird durch den *SchemaManager* durchgeführt. Er stellt zwei Methoden bereit, *validateGenericValusFitsSchema()* zur

[1] https://github.com/fraunhoferfokus/ogd-metadata/blob/master/OGPD_JSON_Schema.json

Validierung der Daten mit einem Schema und *createJsonSchema()*, welche die Schema-Struktur in eine JSON-Schema-Zeichenkette wandelt. Letzteres wird im folgenden grafisch dargestellt.

```java
/**
 * Creates the combined schema.
 *
 * @return the map schema
 */
private static MapSchema createCombinedSchema() {
    Map<String, Schema> water = new HashMap<String, Schema>();
    water.put("shortname", new StringSchema());
    water.put("longname", new StringSchema());
    MapSchema waterSchema = new MapSchema(water);

    Map<String, Schema> currentMeasurement = new HashMap<String, Schema>();
    currentMeasurement.put("timestamp", new StringSchema());
    currentMeasurement.put("value", new NumberSchema());
    currentMeasurement.put("trend", new NumberSchema());
    currentMeasurement.put("stateMnwMhw", new StringSchema());
    currentMeasurement.put("stateNswHsw", new StringSchema());
    MapSchema currentMeasurementSchema = new MapSchema(currentMeasurement);

    Map<String, Schema> gaugeZero = new HashMap<String, Schema>();
    gaugeZero.put("unit", new StringSchema());
    gaugeZero.put("value", new NumberSchema());
    gaugeZero.put("validFrom", new StringSchema());
    MapSchema gaugeZeroSchema = new MapSchema(gaugeZero);
```

```json
"timeseries": [
    {
        "longname": {
            "type": "string"
        },
        "gaugeZero": {
            "unit": {
                "type": "string"
            },
            "validFrom": {
                "type": "string"
            },
            "value": {
                "type": "number"
            }
        },
        "unit": {
            "type": "string"
        },
```

Abbildung 32: Umwandlung in JSON-Schema

4 Ergebnisse

In diesem Kapitel werden die Ergebnisse der Arbeit zusammengefasst. Zunächst wird die implementierte Lösung untersucht und diskutiert. Anschließend werden mögliche Erweiterungen und zukünftige Entwicklungen für den ODS untersucht. Schließlich wird in einem Rückblick die Arbeit zusammengefasst.

4.1 Diskussion

Bei der Entwicklung des ODS ist man auf einige Probleme gestoßen. Diese sollen in diesem Abschnitt erörtert werden.

Performanceprobleme bei großen Datenmengen

Die aktuelle Implementierung des ODS hat Performanceprobleme bei der Verarbeitung großer Datenmengen, beispielsweise beim Importieren geografischer OSM-Daten. Trotz des Einsatzes einer spezialisierten Bibliothek, Osmosis, zum Auslesen der OSM-Daten im XML-Format, konnte die Performance nicht signifikant gesteigert werden. Das Einspielen großer Datenmengen in die CouchDB-Datenbank stellt dabei das eigentliche Problem dar. Für jeden Node, Way und Relation wird in CouchDB ein eigenes Dokument erstellt. Es bleibt zu untersuchen, ob eine andere Struktur der Speicherung möglich ist, oder eine andere Datenbanktechnologie diese Performanceprobleme lösen kann.

Update der Datenquellen

Beim Update von Daten kann es vorkommen, dass die Daten nicht eindeutig identifizierbar sind. Wenn Sie keine Id oder Zeitstempel besitzen, ist es derzeit nicht möglich die Daten richtig zu aktualisieren. Die einzige Lösung wäre die Datenbank komplett mit diesen Daten zu ersetzen. Dies hat jedoch zur Folge, dass die Daten eine neue Id bekommen würden. Dies wiederum verletzt die REST-Prinzipien, die besagen, dass jede Ressource eine eindeutige Id besitzt und über eine URI aufgerufen werden kann. Eine mögliche Lösung wäre zu versuchen für jede ankommenden Datei den Ursprung zu merken und die Identität über eine Art Hash zu gewährleisten.

Hinzufügen der Datenquellen

Neue Datenquellen und Schemata stellen die Architektur des ODS vor das Problem, dass hierfür neue Klassen erstellt oder bestehende angepasst werden müssen. Dazu muss der Dienst offline genommen werden und steht während der Wartung nicht zur Verfügung. Es ist geplant dies durch eine dynamische Konfigurierbarkeit des Dienstes zur Laufzeit zu beheben.

4.2 Ausblick

Neben der Unterstützung neuer Formate und der Anbindung neuer Quellen gibt es einige konkrete Funktionalitäten, die für die Zukunft eine wichtige Rolle spielen.

Update der Datenquellen

Derzeit werden die Datenquellen nacheinander aktualisiert. Dabei könnte eine Parallelisierung der Updates den Update-Vorgang deutlich beschleunigen. Dafür müsste jeder Update-Vorgang in Threads ausgelagert werden. Dabei ist die Auslastung des Servers zu berücksichtigen.

Konfigurierbarkeit der Quellenanbindung

Bisher werden Quellen mit Hilfe von quellspezifischen Konfigurations-Klassen angebunden. Eine dynamische Möglichkeit der Anbindung ist derzeit nicht möglich. Es ist geplant, die Konfigurierbarkeit der Anbindung neuer Quellen und die Wartbarkeit der bestehenden Quellen über ein Webinterface zu gewährleisten. So könnte man neue Quellen im laufenden Betrieb anbinden oder auf Schemaänderungen bestehender Quellen reagieren können.

Lizenzverwaltung

Die Lizenzverwaltung ist ein weiteres Feature, das bisher nicht implementiert wurde. Unterschiedliche Datenquellen veröffentlichen ihre Daten mit unterschiedlichen Nutzungsbedingungen und Lizenzen. Es gilt zu evaluieren, wie man Daten unterschiedlicher Quellen mit unterschiedlicher Restriktivität dem Client einheitlich zur Verfügung stellt. Dies stellt vor allem ein Problem dar, wenn man versucht diese Daten zu kombinieren.

Aktive Bereitstellung der Daten

Bisher stellen Quellen ihre Daten bereit und der ODS greift sie periodisch ab. Dieses passive Bereitstellen der Daten hat einige Nachteile. Der ODS prüft Daten nur in gewissen zeitlichen Abständen. Oft ist es nur schwer zu erkennen, ob sich die Daten verändert haben, und wenn, welche es waren. Dies ist mit viel Overhead verbunden. Eine Lösung für das Problem wäre eine aktive Bereitstellung der Daten, wie es CKAN bereits betreibt. Der ODS müsste dafür eine Schnittstelle anbieten, über die Quellen selbst die neuen Daten in den ODS übertragen. Der Vorteil wäre eine Art Push-Mechanismus. Es werden nur die neuen und veränderten Daten übertragen und sie sind im ODS stets aktuell.

4.3 Zusammenfassung

In dieser Arbeit wurde die prototypische Umsetzung des Open Data Service vollzogen. Dabei bildete der Schwerpunkt dieser Arbeit die Konzeption und Realisierung der reflektierenden auf dem Paradigma REST basierenden Schnittstelle.

Das Ziel der Arbeit war die Probleme der Daten- und Informationsflut zu lösen, indem ein Dienst konzipiert wurde, welcher es ermöglicht, komfortabel auf frei verfügbare, nutzbare Daten heterogener Datenquellen zuzugreifen. Dabei sollte der Zugriff über eine einfache Schnittstelle ermöglicht werden. Der Dienst bedient sich dabei öffentlich zur Verfügung stehender Datenquellen, bereitet diese auf und stellt sie über eine einheitliche Schnittstelle zur Verfügung.

Zunächst wurden im Forschungsteil die grundlegenden Definitionen und Grundbegriffe zum Thema erörtert. Danach wurde die Wichtigkeit von Open Data erläutert und anschließend wurden das REST-Paradigma sowie das für den Datenaustausch verbreitete JSON-Format vorgestellt. Im Anschluss wurden verwandte Arbeiten zu dieser Thesis erörtert. Dabei wurden die Qualitätsmerkmale für eine gute Schnittstelle ausgearbeitet und der Nutzen der Metadaten am Beispiel des Metaobjektprotokolls veranschaulicht. Diese beiden Themenaspekte flossen in die Konzeption des ODS und der reflektierenden API mit ein.

Um die Besonderheiten der Quellen zu berücksichtigen, die unterschiedliche Schnittstellen und Formate nutzen, wurden quellspezifische Adapter entworfen, mit denen es möglich ist verschiedene Datenquellen anzubinden. So macht es es der ODS möglich, Daten von PEGELONLINE

und OpenStreetMap abzufragen, um das Ziel, Gewässerdaten aufzubereiten und über eine zentral Schnittstelle bereitzustellen, zu erreichen. Außerdem aktualisiert der ODS diese Daten periodisch.

Da der Nutzer des ODS einheitlich formatierte Daten erwartet, werden die Daten im ODS in ein JSON-Format umgewandelt. Dabei werden sowohl Nutz- als auch Metadaten durch den ODS im gleichartigen JSON-Format zur Verfügung gestellt. Die Qualitätsanforderungen an die Daten werden mit Hilfe von qualitätsverbessernden Maßnahmen erreicht. So werden die integrierten Daten mit Hilfe der Schemata validiert und nur bei gegebener Gültigkeit angeboten. Der Nutzer hat jedoch immernoch die Möglichkeit, auch die Originaldaten abrufen zu können.

Bei der benutzerorientierten Konzeption der Schnittstelle wurde auf eine intuitive und einheitliche Gestaltung Wert gelegt, die dem REST-Paradigma folgt. Die ehemals heterogenen Daten werden über diese Schnittstelle über einheitliche Zugriffs- und Anfragemethoden angeboten. So kann der Client alle verfügbaren Datenquellen, die Nutz- und die Metadaten einheitlich abfragen. Zu den Metadaten zählen die Beschreibungsinformationen der Quellen sowie die verwendeten Datenstrukturen und Datentypen. Diese Metainformationen bilden die Fähigkeit des ODS, die eigenen Daten reflektiert zu beobachten (Introspection) und dem Nutzer als selbstbeschreibende Datensätze anzubieten. Das ODS-System repräsentiert die Metadaten im gleichen Datenbankmodell wie Nutzdaten. Es stellt damit ein selbstreferentielles System dar.

Für die Umsetzung des ODS wurde als technische Plattform ein Apache Tomcat Webserver verwendet. Implementiert wurde der ODS in Java, die Datenhaltung erfolgt in einer CouchDB-Datenbank. CouchDB zeichnet sich durch das dokumentenorientierte Paradigma aus und ermöglicht die Speicherung der Daten im JSON-Format. Die Handhabung der Daten wurde dadurch erheblich erleichtert.

Zum Schluss wurde die Arbeit bewertet. Die bei der Umsetzung aufgetretenen Probleme wurden diskutiert und ein möglicher Ausblick für die Weiterentwicklung und Gestaltung des Open Data Service gegeben.

Abbildungsverzeichnis

Tabellenverzeichnis

Literaturverzeichnis

Abts, & Mülder. (1996). *Grundkurs Wirtschaftsinformatik.* Springer.

Bodendorf. (2003). *Daten- und Wissensmanagement.* Springer.

Burbiel. (2007). *SOA & Webservices in der Praxis.* Franzis Verlag.

Buschmann, Meunier, Rohnert, Sommerlad, & Stal. (1998). *Pattern-orientierte Software-Architektur.* Addison Wesley.

Curbera, Duftler, Khalaf, Nagy, Mukhi, & Weerawarana. (2002). Unraveling the Web services web: an introduction to SOAP, WSDL, and UDDI. *IEEE Internet computing*, S. 86-93.

Fielding. (2000). *Architectural Styles and the Design of Network-based Software Architectures.*

Fielding, & Taylor. (2002). Principled design of the modern Web architecture. *ACM Transactions on Internet Technology*, S. 115-150.

Gamma, Helm, Johnson, & Vlissides. (1994). *Design patterns: elements of reusable object-oriented software.* Pearson Education.

Garlan, & Shaw. (1994). *An introduction to software architecture.*

Gilbert, & Lynch. (2002). Brewer's conjecture and the feasibility of consistent, available, partition-tolerant web services. *SIGACT News*, S. 51-69.

Hunter. (2014). *Consumer-Centric API Design.*

Ihrig. (2013). JavaScript Object Notation. In Ihrig, *Pro Node. js for Developers* (S. 263-270). Springer.

Kiczales, Ashley, Rodriguez, Vahdat, & Bobrow. (1993). Metaobject protocols: Why we want them and what else they can do. In *Object-Oriented Programming: The CLOS Perspective* (S. 101-118). MIT Press.

Kiczales, Bobrow, & Rivieres. (1991). *The art of the metaobject protocol.* MIT press.

Lucke, v., & Geiger. (2010). *Open Government Data Frei verfügbare Daten des öffentlichen Sektors.*

Maali, Cyganiak, & Peristeras. (2012). A publishing pipeline for linked government data. *ESWC'12 Proceedings of the 9th international conference on The Semantic Web: research and applications* , (S. 778-792).

Masse. (2011). *REST API design rulebook.* O'Reilly Media, Inc.

McKinley, Sadjadi, Kasten, & Cheng. (2004). Composing Adaptive Software. *IEEE Computer*, S. 56-64.

Shadbolt, O'Hara, Berners-Lee, Gibbins, Glaser, & Hall. (2012). Linked open government data: Lessons from data.gov.uk. *IEEE Intelligent Systems* , (S. 16-24).

Steria. (2012). *Beyond Efficiency.*

Stylos, & Myers. (2007). Mapping the Space of API Design Decisions. *2007 IEEE Symposium on Visual Languages and Human-Centric Computing.*

Winn. (2013). *Open Data and the Academy: An Evaluation of CKAN for Research Data Management.*