FAKULTÄT FÜR !NFORMATIK
Faculty of Informatics

Open Data REST-Services

DIPLOMARBEIT

zur Erlangung des akademischen Grades

Diplom-Ingenieur

im Rahmen des Studiums

Business Informatics

eingereicht von

Aleksandar Kamenica, BSc
Matrikelnummer 1126294

an der Fakultät für Informatik
der Technischen Universität Wien
Betreuung: Ao.Univ.Prof. Mag. Dr. Horst Eidenberger

Wien, 14. Dezember 2016

Aleksandar Kamenica	Horst Eidenberger

FAKULTÄT
FÜR !NFORMATIK
Faculty of Informatics

Open Data REST-Services

DIPLOMA THESIS

submitted in partial fulfillment of the requirements for the degree of

Diplom-Ingenieur

in

Business Informatics

by

Aleksandar Kamenica, BSc
Registration Number 1126294

to the Faculty of Informatics

at the TU Wien

Advisor: Ao.Univ.Prof. Mag. Dr. Horst Eidenberger

Vienna, 14th December, 2016

Aleksandar Kamenica

Horst Eidenberger

Erklärung zur Verfassung der Arbeit

Aleksandar Kamenica, BSc
Korneuburgerstraße 23/3/13, 2100 Leobendorf

Hiermit erkläre ich, dass ich diese Arbeit selbständig verfasst habe, dass ich die verwendeten Quellen und Hilfsmittel vollständig angegeben habe und dass ich die Stellen der Arbeit – einschließlich Tabellen, Karten und Abbildungen –, die anderen Werken oder dem Internet im Wortlaut oder dem Sinn nach entnommen sind, auf jeden Fall unter Angabe der Quelle als Entlehnung kenntlich gemacht habe.

Wien, 14. Dezember 2016

Aleksandar Kamenica

Acknowledgements

There are some people, which I would like to thank for their support during the writing of this thesis.

In the first place, I would like to express my gratitude to Prof. Horst Eidenberger. His clear guidelines on how to write the thesis and his extraordinary responsiveness enabled me to proceed smoothly in writing this thesis.

Furthermore, I would like to thank my parents Živorad and Cvijeta as well as my brother Stefan Kamenica for their continuous support and for doing everything that is possible so I can work undisturbed on the thesis at home.

Another person that I would like to thank, is my girlfriend Tatjana for understanding the days and weeks I had to spend on writing the thesis instead of hanging with her and also, in her role as English Language and Linguistics student, for proof-reading the whole thesis.

Kurzfassung

Diese Arbeit befasst sich mit einer möglichen Lösung für jene Nachteile, welche sich durch die übliche Vorgehensweise beim Angebot von Open Data Datensätzen ergeben, bei welcher die Datensätze auf einer Webseite als Dateien zum Herunterladen angeboten werden. Stattdessen könnten die Erzeuger von solche Datensätzen diese über REST Services anbieten, welche OData Version 4, einem aufkommenden Standard für REST Services, entsprechen. Auf diese Art und Weise könnte man eine generische Client-Applikation bauen, welche dazu in der Lage wäre, solche Daten zu durchsuchen und deren zugrunde liegendes Datenmodell anzuzeigen, ohne eine Datei herunterladen zu müssen oder das Datemodell selbst ableiten zu müssen. Auch andere Nachteile könnten durch die Verwendung von REST Services eliminiert werden.

In dieser Arbeit wird sowohl ein Prototyp eines OData Version 4 konformen REST Services als auch ein Prototyp einer generischen Client-Applikation implementiert um zu zeigen, dass solch eine Konstellation realisierbar ist. Die generische Client-Applikation wurde zudem auch mit anderen OData Version 4 konformen REST Services erfolgreich getestet.

Zusätzlich wurde ein Fragebogen über OData von Fachexperten ausgefüllt. Diese Fachexperten haben bestätigt, dass OData das Bauen von generischen Client-Applikationen zum Abrufen und Durchsuchen von Daten, die über entsprechende Services veröffentlicht wurden, ermöglicht.

Abstract

This thesis deals with a possible solution for the shortcomings that result from the common approach of offering Open Data, where the corresponding data sets are offered as downloadable files on a website. Instead, Open Data producers could offer its data via RESTful web services which conform with OData Version 4, an emerging international standard for REST services. This way, one generic client application could be built, which could enable querying and viewing the exposed data and its data model, without needing to download a file or discovering the data model on your own. By using REST services, also other shortcomings of the mentioned approach could be eliminated.

In this thesis, prototypes of both an OData Version 4 compliant RESTful web service and a generic client application were implemented in order to show, that such a constellation is feasible. The generic client application has also been tested successfully against other OData Version 4 compliant REST services.

Additionally, a questionnaire on OData has been conducted with domain experts. The domain experts confirmed, that OData supports/enables the building of generic client applications for accessing and querying data published via such interfaces.

Contents

CHAPTER 1

Introduction

1.1 Motivation and problem statement

At the beginning of his first term of US-presidency in 2009, Barack Obama announced his plans for a transparency strategy of his government. The aim of this strategy was the openness of the data, which would ultimately result in the strengthening of democracy and better perspective for the US citizens over the actions of their government [HVdB11]. This was the time when "Open Data" has become a topic in the United States. In the following years (starting at the end of 2009), several other governments in Europe as well as the government in Australia started Open Data initiatives with the same goals that were stated by Barack Obama previously.

The results of the proclaimed Open Data initiatives were shaped similarly in each of these countries: There was a website created, which served for the presentation of the data sets. These data sets were published at a certain point in time in at least one format. Visitors of the website were now able to search for the data sets by defining search criteria, to view certain metadata of the same and finally to download the data sets in the format(s) which were offered.

Although this approach has been widely adopted, there are several shortcomings of the same:

1. First, the data model of the published data is not described or at least not in a standardized way. On the contrary, the data is just published in a certain format, which as a consequence made each person who wanted to use the data set discover the underlying datamodel himself/herself.

2. Second, querying the data set is only possible in a limited way. The first requirement is to download the data set in a certain format and then, further, to have an

appropriate software installed on the client desktop which would make querying of the data possible.

3. Next, the applications which shall be based on Open Data sets, would first need to download the data set and to deserialize it. The deserialization itself might be a problem if the underlying data model is not well defined.

4. Eventually, as the data sets are published at a certain point in time, they may not contain up-to-date data. This fact requires the publisher to publish the data set periodically in order to keep it up-to-date. This could lead to a delay in time between the point in time when the data set was created and the point in time when the data set was published.

On the other hand, there are web services (APIs), which are a common approach to offer access to certain functionality via machine-to-machine communication. The World Wide Web Consortium defines web services as follows [HB04]:

> *"A Web service is a software system designed to support interoperable machine-to-machine interaction over a network."*

In the category of web services, REST services (REST-APIs) are gaining popularity. One of the problems of REST services is that, until now, there is no common approach to how such a service should be constructed and offered.

However, there is an emerging standard for REST services: It is called OData, which is the short of Open Data Protocol. Version 4 of OData may become the international standard for REST APIs [Ben15]. By implementing REST services conforming with OData, it may be possible to implement generic client applications, which could consume any OData Version 4 conformant API and perform queries on the exposed data. The details on OData and how the previously mentioned shortcomings could be outweighed will be discussed in the following chapters. Standardisation in general would be beneficial both for Open Data producers (knowing in which way they should offer their data) and Open Data consumers (knowing they can rely on a standardised API).

All of these mentioned characteristics lead to the research questions which should be answered in this diploma thesis:

- To which extent and in which way(s) can OData REST interfaces which offer Open Data enable building generic clients for accessing and querying the same?
- How does an OData REST service differ in complexity of accessing/modifying the data compared to REST-APIs which do not implement it?
- How could the offer of Open Data in Austria benefit from a shift of data producers to offering Open Data via OData REST interfaces?

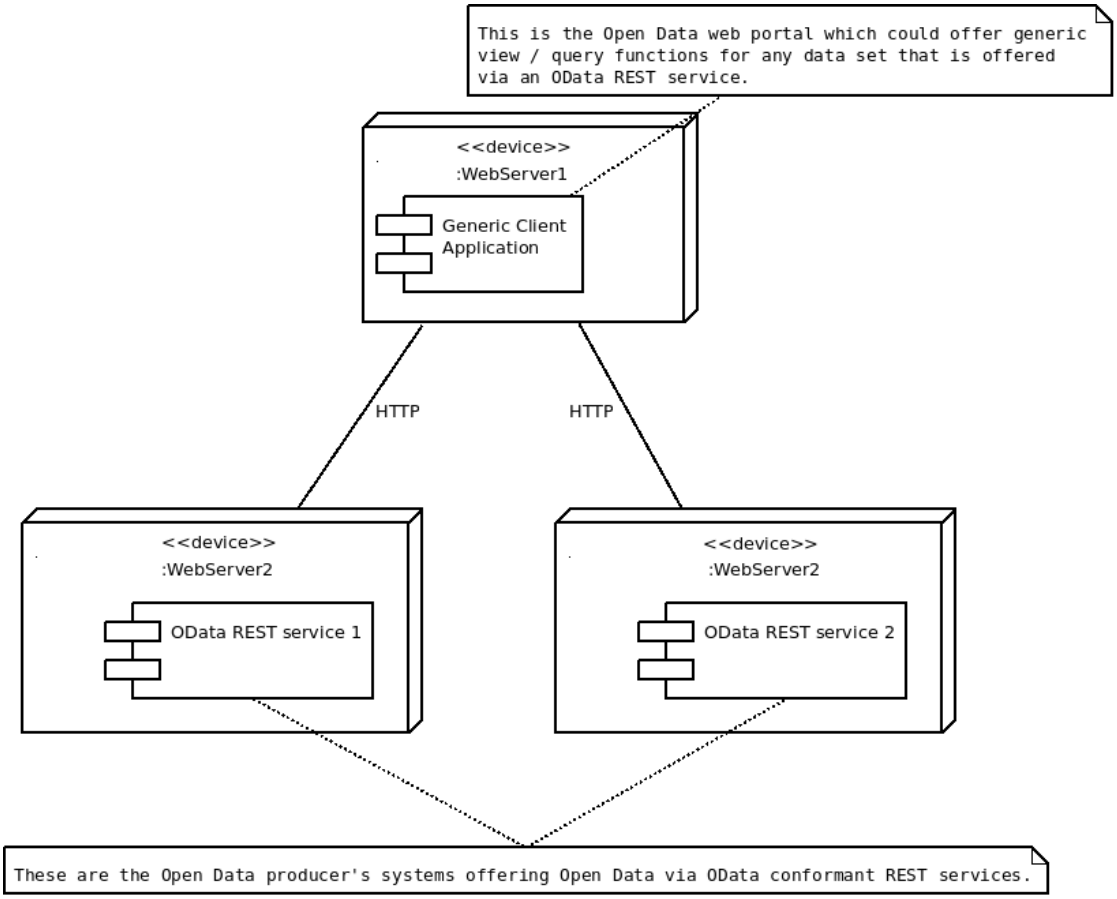

Figure 1.1: A simple example showing the target architecture with one generic Open Data client and two Open Data producers.

1.2 Expected Results

After defining the main problems with which this thesis is dealing, the expected results will be described. In order to be able to show prototypically whether the previously mentioned constellation is feasible, two prototypes will be implemented. Figure 1.1 shows the target architecture with one generic client application (playing the role of an Open Data portal) and two OData REST services which are the Open Data producers interested in offering their data.

The first prototype will be an OData REST service conforming with *OData JSON Format Version 4.0* [HPB16]. JSON Format is mentioned explicitly because OData also offers the possibility to use Atom format for request and response representations. But as the JSON Format Version 4 has been submitted to become an international standard, it will also be used for this prototype. The requirements for the REST API will be defined in the context of a real world example. Based on these requirements, a data model will be

designed and the corresponding REST API prototype will be implemented.

The second prototype will be a generic client application (a website). It will be implemented in order to be able to make a point on whether and to which extent building a generic client based on various OData REST services as backends is possible and what the complexity of connecting to a new OData REST API is from the client's point of view.

The implementation is followed by an evaluation, in which domain experts should answer certain questions enlisted in a form of a questionnaire. In the context of this thesis, a person is considered to be a domain expert if he/she has at least three years of experience in developing REST services. The questions asked in the questionnaire will be highly related to OData itself, but based on the prototypes that have been developed.

1.3 Methodological approach

The methodological approach in this thesis consists of four stages:

1. **Literature research:** First, a literature research needs to be conducted. Since the main problem of this thesis originates in the Open Data sphere, the first focus of the literature research is Open Data. Requirements/characteristics of Open Data, as well as the state-of-the-art of offering Open Data in Austria compared to other countries will be discussed. Furthermore, used technologies and further development strategies will be depicted. The next topic of the literature research will be REST services, as REST services might be the solution for the shortcomings of the Open Data offering approach. Thus, the foundations of REST need to be discussed in detail and the shortcomings of REST need to be elaborated explicitly. At the end of the literature research *OData JSON Format Version 4.0* will be elaborated. All aspects of the OData specification will be discussed and compared to "pure" REST. Apart from these topics, Software Engineering methods/technologies/patterns will be researched, as they are especially required for the implementation of the prototypes.

2. **Requirements specification / analysis:** The requirements for both prototypes need to be specified. On the one hand, there is the OData REST service. Its requirements will be defined in cooperation with the stakeholders that have been identified. On the other hand, there are the generic client application requirements, which will be elaborated based on the knowledge gained from the literature research on Open Data.

3. **Implementation of the prototypes:** First, the OData REST service will be implemented as specified. Afterwards, the generic client application will be implemented, which can then be tested against the previously developed OData REST service.

4. **Evaluation:** The evaluation of OData (and the implemented REST service) will be performed based on the experiences made during the implementation and the questionnaire that is conducted with the domain experts at this stage. The focus of the questionnaire, as already mentioned, will be the technology itself as well as its applicability and the complexity of usage (also in comparison to current state-of-the-art REST-services).

1.4 Structure of the work

Based on the methodological approach, the rest of the thesis will be structured as follows:

In Chapter 2, the outcomes of the literature research will be described. There are three main sections, as mentioned in the methodological approach, which will be Open Data (Section 2.1), REST (Section 2.2) and OData (Section 2.3). In the Open Data section, an introduction to Open Data will be given first. It will be followed by the approach of offering Open Data in Austria and in other countries. There will also be a comparison between Austria and other countries. The REST section will start with an introduction to REST, followed by the explicit architectural constraints of REST and the Resource Oriented Architecture. Eventually, the problems/shortcomings of REST will be discussed. In the Open Data Protocol section, followed by the obligatory introduction, the details of the specification will be elaborated. They need to be understood in order to give us a better perspective of OData and its scope. Also, as mentioned already in the methodological approach, a comparison to REST will be done.

Chapter 3 will focus on the description of the ideas behind both prototypes. Based on the concept, the requirements for both prototypes will be defined and so the scope of the projects will be set. Afterwards, in Chapter 4, the implementation will be illustrated.The functions/backlog, the explicit design as well as the results of both implementations will be depicted.

In the evaluation Chapter, on the one hand the testing of the generic OData client against different OData services is shown. On the other hand, the questionnaire that was filled out by the domain experts and their answers will be analyzed. In Chapter 6 the main conclusions will be drawn. They are based on the insights that were gained during the evaluation phase combined with the theoretical background from the literature research. Additionally, possible future work will be presented and an outlook to the future of OData will be suggested.

CHAPTER 2

Background

2.1 Open Data

2.1.1 Introduction

Before moving to the definition of Open Data itself and the discussion of its relevance, a short historical overview will be given first.

Historical overview

From a conceptual point of view, the idea of freely accessible research results originates in 1942, where Robert King Merton stated that researchers shall give up intellectual property rights on their research results in order to bring knowledge forward [Chi13]. According to Yu and Robinson, the term "Open Data" itself occurs for the first time in the 1970s, in a policy context [YR12]: Back then, operative international agreements between NASA and its international partners required NASA's partners to adopt an *"open-data policy comparable to that of NASA and other U.S. agencies participating in the program, particularly with respect to the public availability of data."* According to the same source, the National Academy of Sciences discussed and elaborated the idea of sharing environmental and geophysical data in 1995. The *Human Genome Project*, a publicly funded project on genetics data, is an early contributor to open data, as its results and related data have been made available to the public.

In 2005, *Open Defintion* was created as a project of *Open Knowledge* [Knob]. It provided the first definition of Open Data and today, it is the main international standard for Open Data and Open Data Licenses. The exact definition of Open Data by Open Definition will be discussed in detail after the historical overview.

In the last quarter of 2007 a meeting was held in Sebastopol (California). The participants discussed the government's possibilities of opening up electronically-stored government

data to the public [Won10]. They also developed a set of principles [BK11] reasoning why publishing Open Governmental Data (Open Data from the public sector) is essential for democracy. Barack Obama's transparency strategy, as mentioned in Section 1.1, followed in the beginning of 2009. This event worked as a trigger for several other countries, which started similar initiatives in the following years. These initiatives made the term Open Data become a common term in the public of the participating countries.

This short historical overview leads us to the definition of Open Data.

Definition

The different sources and drivers of Open Data could, to a certain extent, already be identified in the historical overview: Different scientific branches as well as the public sector are the main actors. There is a definition of Open Data, which is not prone to changes when it appears in different contexts. This is a short summary of the definition by *Open Definition*, which was mentioned in the historical overview and is the main international standard for Open Data [Defb]:

> *"Open data and content can be **freely used, modified, and shared** by **anyone** for **any purpose**".*

More precisely, there are, according to *Open Definition*, the following requirements for an open **work** ("work" denotes the item or piece of knowledge being trasferred) to be satisfied [Defa]:

1. *"The **work** must be in the **public domain** or provided under an **open license**. [...] The term **public domain** denotes the abscence of copyright and similar restrictions, whether by default or waiver of all such conditions."* An **open license** is also defined by Open Definition, but this definition will not be discussed in particular, because it is not relevant for the research done for this thesis.

2. *"The **work** must be provided as a whole and at no more than a reasonable one-time reproduction cost, and should be downloadable via the Internet without charge."*

3. *"The **work** must be provided in a form readily processable by a computer and where the individual elements of the work can be easily accessed and modified."*

4. *"The **work** must be provided in an open format. An open format is one which places no restrictions, monetary or otherwise, upon its use and can be fully processed with at least one free/libre/open-source software tool."*

The possibility of providing a work under an open license makes this definition of Open Data applicable without any context to the public domain. This means that, for example, private organizations, companies and other entities might publish Open Data

by providing their work under an open license (given that they satisfy also the other defined requirements).

Of course, there are other definitions of Open Data as well, which are mainly used in the context of the certain publication in which they have been published. The following definition, for example, has been used by Janssen and Charalabidis and Zuiderwijk [JCZ12]:

> *"In this research we define open data as non-privacy-restricted and nonconfidential data which is produced with public money and is made available without any restrictions on its usage or distribution. [...] Data can be provided by public and private organizations, as the essence is that the data is funded by public money."*

The requirement that the data needs to be funded by public money has not been mentioned in the definition by *Open Definition*. The other requirements that need to be satisfied by Open Data overlap the requirements defined by Open Definition.

As there is a common understanding of what Open Data is, the next question arises: What is the relevance of Open Data?

Relevance of Open Data

The relevance of Open Data is a crucial topic that needs to be discussed. If there was no relevance of Open Data, then the effort that has been put into the publication of the same would be questionable. The expected results of Barack Obama's transparency strategy were already mentioned in Section 1.1. The relevance and the benenfits of Open Data also depend on the source and category of the data sets as well as on the subject that wanted to make use of the data. For scientists, the benefits and relevance of Open Data in the scientific domain are different than those for citizens. On the other hand, citizens might be more interested in those data sets that are published by the government in order to gain more insight into the actions set by the government in different areas.

The authors of "The Data Harvest: How sharing research data can yield knowledge, jobs and growth" present the vision of making *"the whole world a single, living lab"* [GHL+14]. Furthermore, they identified benefits of Open Data for following exemplary personas:

- **The Citizen:** Products and services that are developed upon Open Data will be beneficial for all people, either directly or indirectly. They will furthermore be empowered by having all informations that are necessary for making decisions in different spheres of life. Businesses and the government will become *"more accountable, efficient and effective"*.
- **The Entrepreneur:** As no organization has the necessary resources (human resources, capital resources) to extract the full value from its data, innovation might be fostered by opening up the data. Entrepreneurs can develop new services and products, which would create new jobs and thus also be a benefit for the citizens.

- **The Scientist:** Reusing and sharing data would boost the research productivity and the creativity of the researchers. Scientists might collaborate more often internationally, their mindset would be changed by becoming more open and by sharing early findings.

In a paper published by Capgemini Consulting, the author suggests that there are several economic benefits of Open Data for both the government and the private sector [Tin13]:

- Both sectors could increase revenue through multiple areas. The government would benefit from an expanded economic activity by earning increased tax revenues. The private sector benefits from new business opportunities that arise upon the data.
- Costs could be reduced on governmental side by reducing the transactional costs. On the private sector side the costs of *"not having to invest in conversion of raw government data"* (aggregation is done on the side that publishes data) could be saved.
- Efficiency could be increased. The private sector could improve its decision making by basing it on more accurate information. The government's service efficiency could be increased through linked data.
- The government could create new jobs and stimulate entrepreneurship, while the private sector could *"gain skilled workforce"*.

The authors of "Benefits, adoption barriers and myths of open data and open government" also derived a list of political, social, economic, operational and technical benefits of Open Data [JCZ12]. But, they also conclude that open data should not be treated as a homogenous topic:

> *"The diverse nature of open data means that different types of results from open data have different benefits and are confronted with different barriers."*

Summing up, there is significant research on the potential and benefits of Open Data. The relevance of Open Data is implicitly confirmed by its numerous advantages for many different sides (groups). Next, the overall approach to Open Data in Austria will be discussed.

2.1.2 Approach in Austria

Open Governmental Data

In April 2012, Austria's national Open Governmental Data portal[1] for data sets originating from the public sector has been launched [Fut13]. The aim of this portal is to be the

[1] `https://www.data.gv.at`. Visited: October 16, 2016

central data catalogue of the metadata of the decentral data sets, which are published by the different public administration entities [Bunc]. Additionally, the national portal is the "single point of contact" to the European data portal.[2]

There are also decentral portals of the single public administration entities, which itself preprocess and present their published data sets. The authors of [EHL+13] have defined an URL-convention for all data portals which offer Open Governmental Data. The URL should be constructed in the following way: `data.organisation.gv.at`, where "organisation" needs to be replaced with an identifier of the organisation. E.g.: The data portal for the Open Government Data of Vienna (in german: "Wien") can be accessed via the URL `https://data.wien.gv.at`, for Graz the URL is `http://data.graz.gv.at/`. Additionally, Austria has made the regions co-owners of the national portal, which made the regions responsible for including their data on the national data portal [CNV16].

In order to publish a data set on the national Open Data portal, a form on the portal needs to be filled out and submitted including at least the representative's name, email address and name of the organisation the representative belongs to [Buna]. On the same page, it is stated that a responsible person would contact the submitting person to advise him/her how to publish the specific data set. In the report *Open Data Maturity in Europe 2016*, it is stated that the regional portals upload their data to the national portal through the portal's API (the API will be discussed next) [CNV16]. This makes Austria the country with the most machine-to-machine traffic that is generated via the portal API in the European Union [CNV16].

Open Governmental Data - Technology (CKAN)

In this section, the technology that is being used by the national Open Governmental Data portal will be described. The national portal is built using CKAN, which is an open source data portal software for creating open data websites [Ckac]. Data that are being published there is published as a "dataset". The dataset consists of the **metadata** (e.g. the title, the publisher, ...) and the **resources**, which hold the data itself. Several data formats are supported as resources and one dataset might have many resources of the data (there might be, for example, a XML file and a PDF document for one dataset, containing the same data). The following main use cases for Users are covered by CKAN [Ckac]:

- *Registration and logging in:* This is mainly needed for the publishing and personalization features of CKAN.
- *Adding a dataset:* When adding a dataset, a form needs to be filled out. Metadata to the data needs to be entered as well as the resource(s) where the actual data can be retrieved from. A resource might either be a link to a file, a link to an API or a file that is uploaded directly in the form.

[2] `https://www.europeandataportal.eu/`. Visited: October 16, 2016

- *Editing/deleting a dataset or resources of the dataset.*
- *Creating and managing an organization:* Each dataset is owned by an organization and each organization can have multiple users, who are able to manage the datasets of the organization they belong to. Managing permissions of the users of the organization is also included.
- *Finding data:* Data can be found by defining search words and search filters. It is also possible to restrict the results to the data of a certain organization. The search is solely based on the metadata of the dataset.
- *Exploring datasets:* After the selection of the dataset, all public metadata of the dataset is shown on one page. Additionally all resources of the dataset are shown including brief descriptions of and links to each of the resources. On the dedicated page of a resource the download of the resource is possible. There is also a preview offered for the structured types of resources (such as .CSV and .XLS).
- *Personalization:* For authenticated users a personal news feed as well as the user profile can be managed.

There is also a RPC-style API which enables API clients to access CKAN's features [Ckaa]. In case the "DataStore extension" is active, which provides the previously mentioned preview functionality, then it is also possible to search, filter and update qualified data sets by using the DataStore API provided by CKAN [Ckab].

Open Governmental Data - Linked Data

There is also an approach to Linked Data. For this purpose there is "LOD Pilot"[3], which is a pilot project for building up a Linked Data infrastructure in the public sector (Open Governmental Data). There is a SPARQL interface where visitors of the website can query available data using SPARQL. There are currently 18 data sets (as of October 16, 2016), which have been taken from the national or the Vienna Open Governmental Data portal, converted to RDF, linked to each other and to other sources and, finally, have been published as Linked Open Data [Bunb].

Non-governmental Open Data

For non-governmental Open Data there is a separate portal called "Open Data Portal Österreich"[4]. Data sets, which do not originate from the public sector, might be published and explored there. This portal is operated by the Austrian branch of Wikimedia [Ösb]. In July 2014, the portal has been launched in the Beta-phase [Ösa].

According to the web page, the process of publishing a data set includes the following steps [Ösc]:

[3] `https://www.lodpilot.at`. Visited: October 16, 2016

[4] `https://www.opendataportal.at/`. Visited: October 16, 2016

1. Identify a data set that is suitable to be published.
2. Put the data into a suitable format.
3. Describe the data by entering its metadata.
4. Upload the data or provide a link to it.
5. Keep your data up to date.

In order to be able to publish a data set, a user must be registered. Compared to the publishing process for governmental data, this process is simpler. One can register on the portal at any time and publish his/her data in just a few steps.

The underlying technology of this portal is the same software that has been used to create the governmental data portal (CKAN). Thus, these two portals have the same functional capabilities. In the next section, the Austrian approach to offering Open Data will be compared to the approaches in other countries.

2.1.3 Approaches in other countries and comparison to Austria

In this section, the comparison will be limited to Open Governmental Data portals. First, the portal(s) and the used technologies of a selected number of countries will be described; afterwards, some comparative parameters and facts will be presented. The countries, which will be used for comparison are:

- The USA as the driving force behind the Open Governmental Data movement.
- The UK as one of the leading countries in Europe concering Open Governmental Data. Facts that confirm this statement will be presented in the comparison section.

USA

The USA have been the driving force behind the Open Governmental Data movement. Drawing an analogy to the diffusion of innovations, the USA have clearly the role of the *innovator* in this field.

In the USA there is, like in Austria, a federal data portal. The federal Open Governmental Data portal[5] of the USA was launched in May 2009 [KA16]. Its goal is, as in Austria, to be the central catalogue of the decentrally published data sets. Another similarity to Austria is the fact that the open source application CKAN (besides WordPress) has been used to build the portal [UGSATa]. Consequently, there is the same functionality available on the portal (depending on the implementation of course) and there is also the same API that can be used by developers to access the functionality of the portal programatically.

[5] `https://www.data.gov`. Visited: October 17, 2016

The process of publishing data sets on the portal is similar to the Austrian approach: A form, which requires information about the data set as well as about the submitting person, needs to be submitted [UGSATb]. A responsible person on *Data.gov* side contacts the submitting person for further clarification about the data publishing process.

United Kingdom

In the UK, there is also a national Open Governmental Data portal[6] [Eur16]. In January 2010, the website was launched publicly labeled as "beta version" [Govb]. Again, the portal plays the same role as in Austria and the USA — it is a central catalogue of the decentrally published data sets. And again, CKAN has been used (besides Drupal) to build the portal [Gova], which leads to the same consequences as mentioned already for the USA in comparison to Austria.

One difference in comparison to Austria and the USA is the publishing process. There is a web page which describes in detail how to publish data on the portal.[7] To summarize shortly, a person needs to register in order to become an 'editor' or 'administrator' for the organisation he/she is working for. Once the account has been confirmed, the person who owns the account can publish data on the portal.

Quantitative comparison

In order to be able to quantitatively compare these countries in the context of their progress in their Open Data program ("Open Data Maturity"), suitable benchmarks need to be found or defined first. In the paper *Benchmarks for Evaluating the Progress of Open Data Adoption Usage, Limitations, and Lessons Learned*, the authors have compared five Open Data Benchmarks [SZJG14]. One part was dedicated to the comaprison of methodologies of these benchmarks. Two of these benchmarks ("World Bank ODRA" and "Capgemini OD Economy") are not created periodically, and the "ePSI Scoreboard" covers only the EU member states. The following two benchmarks have been selected for this thesis to make it possible to compare Austria, the USA and the United Kingdom:

- **Open Data Barometer** by the *Open Data Institute and the World Wide Web Foundation*: The components of this benchmark are *"readiness for Open Data initiatives, implementation of Open Data programmes and impact that Open Data has on business, politics and civil society"* [Fouc]. On each of these components 0-100 points are awarded to each of the examined countries and the arithmetic mean of these three scores is taken as the final score. The purpose of this benchmark is to identify challenges to Open Data in the countries [SZJG14]. The third edition of the Open Data Barometer (most recent one at the time of writing) is based upon the following types of data [Foud]:
  - A peer reviewed expert survey

[6] `https://data.gov.uk`. Visited: October 17, 2016
[7] `http://guidance.data.gov.uk`. Visited: October 17, 2016

  - government self assessment (survey)
  - secondary data selected from the World Economic Forum, World Bank, United Nations e-Government Survey and Freedom House.

- **Open Data Index** by the *Open Knowledge Foundation*: This index compares the countries upon "key data sets" from 13 different areas. One of the assumptions of the research is that *"the national government has a responsibility to ensure the open publication of such data even if it is held and managed by a third-party"* [Knoa]. For each of these key data sets, nine questions about the openness of the data sets (based upon the definition of Open Data by Open Definition, which has been described in Section 2.1.1) are being asked. 0 - 100 points are allotted for each key data set. Based on these scores an index percentage is calculated and the resulting ranking of the countries is based upon this index percentage. According to Susha et al., the index *"seeks to encourage advocacy and push governments to improve"* [SZJG14].

Table 2.1 presents the Open Data Barometer ranking of the countries for the year 2015. The United Kingdom is the "winner" of this ranking with the maximum overall score of 100, followed by the USA with a significantly lower score of 81.89. The USA is, according to this ranking, ready for Open Data initiatives. But there seems to be a lack in the implementation and impact of Open Data. Austria is placed 13th in the Open Data Barometer ranking. The implementation seems to be the biggest drawback of Open Data in Austria, while the readiness is rated better than the readiness of Denmark, which is on the fifth place of the ranking.

Country	Rank	Score	Readiness	Implementation	Impact
United Kingdom	1	100	100	100	100
United States of America	2	81.89	97	76	76
France	2	81.65	97	76	74
Canada	4	80.35	89	84	67
Denmark	5	76.62	77	77	78
...	...	...	...	...	...
Austria	13	64.18	81	49	70

Table 2.1: Ranking according to the Open Data Barometer for the year 2015 [Foub]

The Open Data Index ranking is depicted in Table 2.2. In this ranking Taiwan, which was not ranked by the Open Data Barometer, came out as the winner with a score of 78 % before the United Kingdom, which was placed as the runner up (76 %). The United States of America are ranked eighth, rather lower in comparison to the second place in the previous ranking, while Austria is even on the 23rd place with a score of 50 %.

In order to explain the ranking, the countries need to be compared on the level of the 13 key data sets:

Country	Rank	Score
Taiwan	1	78 %
United Kingdom	2	76 %
Denmark	3	70 %
Colombia	4	68 %
Finland	5	67 %
...	...	...
United States of America	8	64 %
...	...	...
Austria	23	50 %

Table 2.2: Ranking according to the Global Open Data Index for the year 2015 [Knoc]

- United Kingdom: The data set *Election Results* scored 0 points, which is the worst score among all data sets of the UK. Also the data sets *Water Quality* and *Land Ownership* were awarded less than 50 % of the possible points. In contrast to these three data sets, there are seven data sets with a maximum score.

- United States of America: Data set *Land Ownership* scored 0 points, while there are three other data sets with a score of less than 50 % of the possible points (*Election Results, Water Quality* and *Government Spending*. The status of the data set *Company Register* was rated as "unclear" for most of the relevant criteria. It is only clear that the data exists in a digital form. In comparison to the UK, there are six data sets with a maximum score.

- Austria: No data set has been rated with 0 points, but there are points of criticism for 10 of the 13 data sets. *Government Spending* scored the least points among all data sets in Austria, followed by *Land Ownership, Location datasets, Company Register* and *Weather forecast* (all of them scored less than 50 % of the possible points).

None of these rankings explicitly included the API of the Open Data portals of the ranked countries as criterion. On the other hand, in "Open Data Maturity in Europe 2016," the presence of an API was one of the indicators used to determine the Open Data readiness of the European countries [CNV16]. However, as the technology behind all three portals is CKAN, the countries provide the same API for interaction with the corresponding portal. The CKAN API is an RPC-style API. However, for purpose of this thesis a REST API will be created using OData. Thus the next section will deal with REST and Web Services (APIs) that are based on REST.

2.2 REST

2.2.1 Introduction

REST is an acronym that stands for *Representational State Transfer*. REST is an architectural style, which has been defined by Roy Thomas Fielding in his dissertation [Fie00].[8] It has been derived from the architecture of the Web as part of the efforts to standardize the Web architecture and protocols in the midst of the expansion of the Web that has started in the 1990s [HFdTC]. The main goal was to establish an *"architectural style for distributed hypermedia systems."* Six architectural constraints, which will be explained in Section 2.2.2, have been selected and applied sequentially by Fielding in order to "form" REST:

1. Client-Server
2. Stateless
3. Cache
4. Uniform interface
5. Layered system
6. Code-on-demand

Beside the architectural constraints, there are three different classes of archtitectural elements: Data elements, connectors and components.

- **Data Elements:** The *state* of the data elements is a central point of REST. At the communication of components, *representations* of *resources* are being transferred.
  - *Resource:* A *resource* is the key abstraction in REST, denoting any information that can be named. It is a *"conceptual mapping to a set of entities,"* where the semantics of this mapping is constant over time while the single entities of the resource are alterable as time passes.

    One example from software engineering: A source code file X in a version control system might be released in a version 2.0. The resource "version 2.0 of file X" will always point to the same entity. In contrast to that the resource "latest revision of X" points at some point in time to the same entity as in version 2.0, but as development goes on and the file X is being changed, the resource will point to a different entity.

    A particular resource is identified in REST by a *resource identifier*.

[8] Fielding's dissertation is the main source that is used for this introductory section, unless there is a different citation.

  - *Representation:* Actions on resources are performed in REST by using a *representation*. Components in REST transfer representations of the current or intended state of resources between each other. A representation is composed of a *sequence of bytes* (the data), *representation metadata* which describes the data and occasionally, *metadata which describes the metadata* (e.g. hash sums for verifying message integrity). A response might include additionally to the representation metadata also *resource metadata*, which is information about the resource that is not tied to the representation.

    Control data might also be supplied in the communication between the components. Control data can define the purpose of a message. Alternatively, it might be used to parametrize requests (e.g. influence the cache behavior by supplying control data in the request/response).

    Each representation is supplied in a certain data format. A representation's data format is called *media type*. The media type might be intended for a user to be viewed or for machine processing (some media types are capable of both). There is no restriction upon the data format as long as it is processable by all components involved.

- **Connectors:** Connectors are architectural elements which are in charge of managing the communication for a component. There are five different connector types:

  - *Client:* A client initiates the communication between components by sending a request.
  - *Server:* A server listens for connections, processes incoming requests and responds to the same.
  - *Cache:* A cache *"can be located on the interface to a client or server connector."* It stores recent cacheable responses of interaction, in order to reuse them for future requests.
  - *Resolver:* A resolver translates a resource identifier into a network address, which is required to establish the communication between components.
  - *Tunnel:* A tunnel relays communication. REST components are capable of switching dynamically from active to tunnel behavior.

- **Components:** There are four different components in REST. The classification is based on the roles of the single components inside an application:

  - *User agent:* The user agent initiates a request (by using a client connector) and is the final recipient of the server's response.
  - *Origin server:* The origin server is the final recipient of each request that is sent with the intention to modify values of resources. The request is received by using a server connector. It is also *"the definitive source for representations of its resources."*

Figure 2.1: Client-Server constraint [Fie00].

– *Intermediary components: Proxy* and *Gateway* are the two different intermediary components. Such components act as a client as well as a server in order to translate and forward requests and responses. The difference between a proxy and a gateway is that a client can explicitly choose to communicate with a proxy, but not with a gateway.

It is important to emphasize that REST does not place any restrictions on the used communication protocols and implementations of the single components. Only the interface of the single components is a part of the definition of REST. In the next section, the architectural constraints, which have already been mentioned in this section, will be discussed in detail in order to get a comprehensive overview of REST.

2.2.2 Architectural Constraints

The order of the architectural constraints is the order that has already been used by Fielding [Fie00] when he was defining the architectural boundaries of REST.

Client-Server

The first constraint that has been added to the set of constraints is the client-server constraint, which is pictured in Figure 2.1. It depicts the separation of concerns, of which the main advantage in the context of the Web and REST is that both can grow independently of each other.

Stateless

After the client-server constraint, the constraint that the communication must be stateless was added. This means that each client request must contain all information that is necessary for the server to understand and process it. The server remains stateless causing the session-handling to be located on the client's side. The advantages and drawbacks associated to this constraint are presented in Table 2.3.

Cache

The main reason for adding cache as an architectural constraint was to improve the network performance. The cache allows the client (and the server) to store and reuse a

Advantages	Drawbacks
• Visibility: The request contains all information needed to understand the same, which allows for example a monitoring system to analyze it without the need to inspect other requests. • Reliability: Recovering from partial failures becomes easier. • Scalability: As the state between two requests does not need to be stored, the resources can be freed on the server side after processing the request. Furthermore no resource management between two requests is necessary, which reduces the complexity.	• The network performance may be impacted by this constraint. The reason is that this constraint might make it necessary to repeatedly include the same information in several requests to the server, because the server needs the information to understand and process each of the requests. • Server looses the control over consistent application behavior to some extent, since there can be multiple clients and client versions which implement the semantics.

Table 2.3: Advantages and disadvantages of the stateless-communication constraint

certain response for potentially occurring later requests, under the assumption that the response is cacheable. By using a cache, it is possible to partially or completely eliminate some interactions, which leads to an improvement of efficiency at the cost of reliability. The reliability is reduced by the possibility of having stale data inside the cache, which differs from the server-state of the data [Fie00].

Uniform Interface

A unique characteristic of REST among network-based architectural styles is the focus on the presence of a uniform interface between components. Additionally to the advantage that the decoupling of the implementation from the interface allows the server as well as the client to grow independently, there are two other advantages:

- The overall architecture is simplified.
- The visibility of interactions is improved.

In Figure 2.2, the uniform interface constraint is shown together with the previously described architectural constraints and some of the introduced architectural elements.

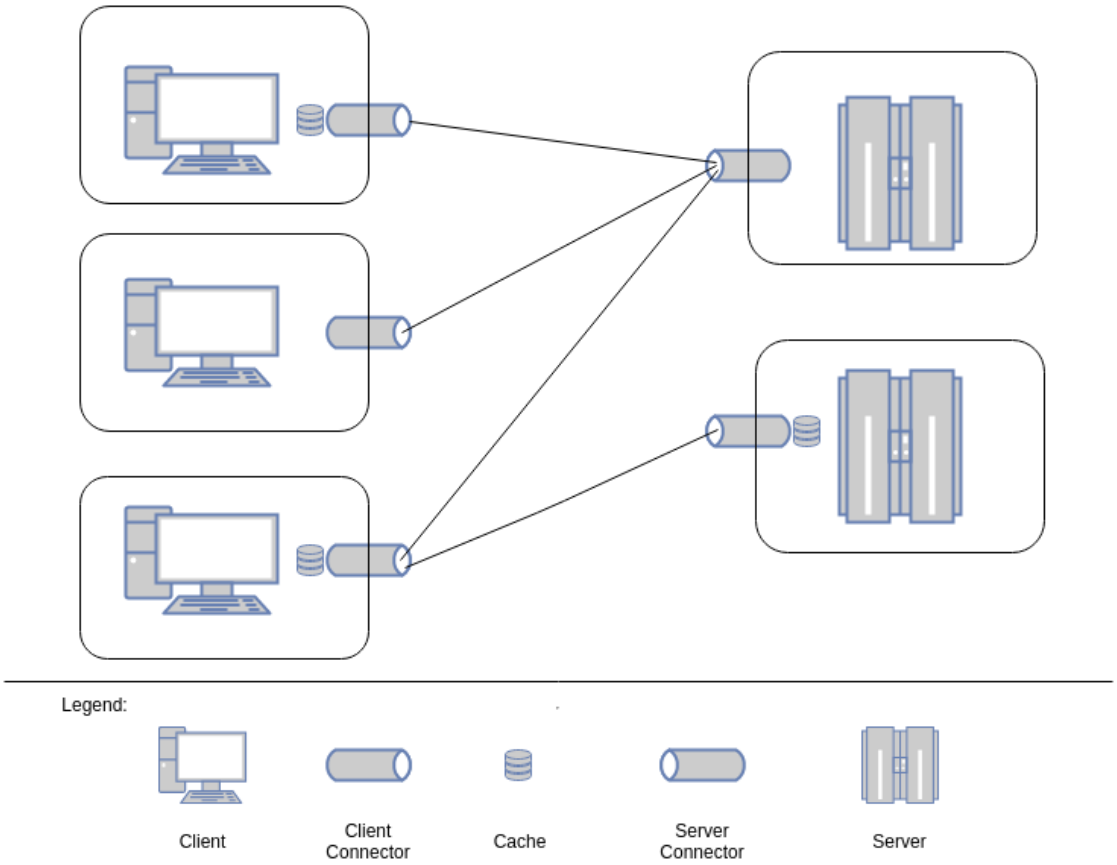

Figure 2.2: Uniform interface with a stateless server and caches on both server and client side [Fie00].

The drawback of applying this constraint is that efficiency is degraded, as there is no tailoring to an application's specific needs. The data is offered and transferred in a standardized way, which might not be optimal in every case [Fie00].

The uniform interface is additionally enforced by the presence of other constraints and architectural elements that have mostly been already introduced:

- Resources as the key abstraction, which map to a set of entities.
- Representations of resources as another architectural element, which are used to transfer data between components and to manipulate resources.
- Self-descriptive messages induced by the statelessness of the server.
- Hypermedia as the engine of application state.

Layered System

The overall complexity of the system is further reduced by adding the layered system constraint. This constraint allows multiple layers to be added to the overall architecture (consisting of architectural elements that have been discussed in Section 2.2.1). Each component only knows about the immediate layer with which it is interacting, without gaining any insight into possibly existing deeper layers. A layered system's main disadvantage, which might especially stand out in a network-based system, is the possible reduction of the performance. Still, caching on the client side or at intermediary components might compensate such a performance loss.

Code-On-Demand

The final and at the same time only optional constraint in REST is code-on-demand. Code-on-demand allows the extension of client functionality by downloading and executing code as an applet or script. Although there are benefits of applying this constraint as, for example, the improved system extensibility, it reduces the visibility and for this reason it was marked as optional.

REST is, as already mentioned, an architectural style. But REST does not define any concrete technologies and protocols which shall be used and is beyond that no concrete architecture itself. Therefore, the next section will cover a concrete architecture which is conformant with REST - the Resource Oriented Architecture.

2.2.3 Resource Oriented Architecture

The resource orientation as a concept has its roots in Fielding's dissertation [Fie00]. It emerged upon the constraints that have been defined for the service orientation: The server-client constriant, a globally unique reference (resource identification) and a stateless message exchange [Ove07]. The most important additional constraint of a resource oriented architecture is the stress on having a uniform interface, whose benefits have already been discussed in Section 2.2.2. Overdick comments in his paper the discussion on the topic "service orientation vs. resource orientation" as follows [Ove07]:

> *"[...] Thus, all discussion of services vs. resources are moot, as all resources are services by definition."*

The *Resource Oriented Architecture (ROA)*, which has been defined by Richardson and Ruby in their book [RR08a], is one of the earliest appearances of the term and one of the most referenced ones when it comes to RESTful Web Services and Resource Oriented Architecture. Hence, the Resource Oriented Architecture, as defined by Richardson and Ruby, will be presented in this section.

In the context of *web* services they tie their Resource Oriented Architecture to the technologies of the Web. Therefore, they use HTTP and URIs to make up a concrete

architecture. Their architecture makes use of *"resources, their names, their representations and the links between them"* [RR08b]. Resources as an architectural element have already been discussed in Section 2.2.1. In this concrete architecture, by making use of the technologies of the Web, resources are identified by an URI (Universal Resource Identifier). The URI is name and identifier of a resource at the same time. Although not part of the draft standard for URIs [BLFM], still the URI should be structured and predictable according to Richardson and Ruby.

The data is returned by the server as a series of bytes, which might represent the resource in a certain file format in a certain language. One resource can have one or multiple representations. A client can make use of HTTP headers to let the server know, which representation is preferred by the client. This process is called *Content Negotiation*. The `Accept` header lets the client send a list of preferred file formats of representations to the server. The client could, for example, prefer JSON to XML as format and thus set the corresponding `Accept` header in his request. The preferred language is another parameter that can be sent by the client. In order to notify the server about the preferred language, the `Accept-Language` header might be used. Information, which is sent in the `Accept` and the `Accept-Language` header, might also be sent as part of the URI according to Richardson and Ruby, without violating any REST concept. On the other hand, when the client intends to modify an existing resource, he needs to send a representation of the intended state of the resource in the request body. In order to tell the server the format of the representation of the resource, the `Content-Type` header needs to be set by the client.

The following four central properties are propagated by the authors:

- Addressability: The resources of a service are exposed via URIs and addressable via HTTP.
- Statelessness: The client includes all necessary information in his request (by using the request headers and possibly the request body) and after each request has been served, the application returns to its initial state. By using HTTP, which is a stateless protocol, the statelessness is given by default.
- Connectedness: *"Resources should link to each other in their representations"* according to Richardson and Ruby. The presence of this link meets the "Hypermedia as the engine of application state"-constraint, that has been mentioned in Section 2.2.2 at the description of the "uniform interface" constraint. For example: When retrieving the web page `https://www.google.com/search?q=cars`, the "next page" link is the information that is needed for the client to change its state (by changing the page).
- Uniform interface: HTTP provides a set of methods that are used to represent operations on resources. The following four most common operations (HTTP verbs) are mentioned explicitly by Richardson and Ruby:

- GET: GET is used to retrieve a resource in the requested representation.
- DELETE: A DELETE request is used to delete an existing resource.
- POST: A POST request is usually used to *"create subordinate resources."* Subordinate resources are resources that have been created in relation to some parent resource. Such a request needs to include the resource within the request body using a representation that is accepted by the server.
- PUT: A PUT request might be used to update or to create objects. Again, the request needs to include the resource within the body. The main difference between POST and PUT in the creation use case is that the PUT request's URI must point to the URI that the new resource should have, while the POST request is usually applied to a "collection" URI and the server decides about the URI of the resource which should be created. Here is an example to illustrate the difference: Given the URI for fetching cars from a system is `/cars`, then a PUT request, which wants to create a car, needs to point already to the final URI of the new car (e.g. `/cars/123`), while the POST request would point to `/cars` and the server would be in charge of deciding upon the URI of the newly created car.

The Uniform interface is encouraged by two additional properties of HTTP methods: safety and idempotence. Requests that are considered safe should not have any side effects on the server side. Of the presented operations, only HTTP GET is considered to be safe. Of course, there might be for example logging of each request on the server side to a log-file, which would be a side-effect. But these side-effects have not been asked by the client's request, thus he is not responsible for them. Idempotent requests are such requests that have the same effect, whether applied once or more than once. PUT and DELETE are, besides GET, the idempotent operations out of the set of the most common operations.

This was the overview over the Resource Oriented Architecture like it is defined by Richardson and Ruby, which is tied to the technologies of the Web and can be used to create RESTful web services. But REST is not the one and only architectural style, as the Resource Oriented Architecture is not the one and only architecture. There are also shortcomings and problems that raise together with this approach, which will be discussed in the next section in detail.

2.2.4 Shortcomings / Problems of REST

The disadvantages of REST as architectural style and ROA as RESTful architecture originate in the constraints that are proposed by REST. In Section 2.2.2, already some disadvantages that come along when applying certain constraints to the architecture, have been mentioned.

In [PZL08], the authors compare RESTful web services to WS-* web services. Thereby they have identified the following weaknesses of RESTful web services:

- HTTP-verbs: In the XHTML-form, the `method` attribute only supports POST and GET as HTTP methods. Additionally, firewalls and proxies may not always allow HTTP connections that use other verbs than GET and POST. These issues lead to several non-standard workarounds, which might not be understood by all web servers and thus require additional implementation and testing effort.
- The interoperability of a RESTful web service might be hindered by the flexibility of the format a representation of a resource might be provided in. For example: A client which expects data in JSON format might not be able to parse an XML response.
- The flexibility that is induced with the capability of serving a resource in mulitple representation formats also implicitly requires extra maintenance effort.
- The service description of RESTful web services is rather informal and textual, if it exists at all. There is no official interface description language, which would facilitate for example the generation of client-stub code automatically.
- Beyond HTTPS (point-to-point SSL security), there are no other security mechanisms proposed in conjunction with REST.
- There is no common service discovery mechanism for RESTful web services.
- HTTP is a synchronous protocol, hence the communcation is limited to synchronous communication.

All of these weaknesses are related to the architectural style and the used technologies. This means that a software architect, choosing to implement a RESTful web service with the given technology stack, should be aware of the weaknesses of the architecture he has chosen. In [GINT14], the authors criticize a missing service registry and service contract, which are essential *"in order to enable the ad hoc usage of services during service consumer runtime"*. This is relevant in the context of SOA. Beyond the presented weaknesses, which are mostly related to the used technologies and the architectural style, there is another problem that has been addressed in [GINT14] and is the source of other problems in the context of REST: There is a lack of standardisation in the REST domain.

In order to prove this lack of standardisation, the authors conducted an empirical study with six distinct REST frameworks. They have developed the same application with each of the selected frameworks. Furthermore a test case set was defined beforehand, which was executed against each of the applications. The key insights of their study are:

- The missing standardisation lead to diverse implementations, which causes incompatibilities (especially between components deployed on heterogeneous platforms).
- The main divergences were identified in the usage of HTTP headers as well as the HTTP status code of the response. For identical requests, different HTTP status codes were returned to the client.

- Sometimes, although the response was sent with the same HTTP status codes, the responses varied in respect to other aspects. The authors mentioned the example of the creation of a resource in ASP.NET. The response code 201 (= "created") is returned, but additionally to the "Location" header the generated resource is returned in the body of the response.

The authors suggest that a compulsive specification and mapping of REST to HTTP is needed in order to improve the out-of-the-box compatibility between REST implementations. Automatic code generation as well as a *"loose-coupling of REST services"* could be enabled by such a specification. Also the security aspects needs to be covered by such a specification according to the authors. This lack of standardisation as well as some other identified issues are tackled by the *Open Data Protocol (OData)*, which will consequently be discussed in the next section in detail.

2.3 OData - Open Data Protocol

2.3.1 Introduction

The Open Data Protocol (OData) is an open protocol *"that defines a set of best practices for building and consuming RESTful APIs"* [ODac]. It has been published by Microsoft under the Open Specification Promise [Cor]. The first revision of the protocol has been published on Feburary 27, 2009 [Cor]. OData Version 4.0 and OData Version 4.0 JSON Format have been approved as an OASIS[9] standard in February 2014 [OASc]. OData JSON Format defines the resource representations for the OData requests and responses using JSON [HPB16], while the protocol itself also gives the possiblity to use the Atom format [PHZ16a]. However, the Atom Format specification has not yet become an OASIS standard [OASc].

As already mentioned in Section 1.1, the most recent OData version (4.0) may become an international standard for REST APIs. It has been submitted to the ISO/IEC Joint Technical Committee (JTC) 1 to be approved as an International Standard [Ben15]. On the ISO web page, the two documents that have been submitted for becoming an International Standard ([ISOb] and [ISOc]) appear to be in stage 40.99, which means *"Enquiry stage - Full report circulated: DIS*[10] *approved for registration as FDIS*[11] *"*. As there is a stage 60.60, which means *"Publication stage - International Standard published"*, the conclusion can be drawn that OData has still not become an International Standard. Nevertheless, there is only one stage to go for OData (the *"Approval stage"*) to become an International Standard.

[9] *"OASIS is a non-profit consortium that drives the development, convergence and adoption of open standards for the global information society"* [OASa]

[10] Draft International Standard [ISOa]

[11] Final Draft International Standard [ISOa]

Since the genesis of OData has already been discussed, the content of OData will be elaborated next. The OData protocol specification covers a variety of topics related to RESTful web services [PHZ16a]:

- Data Model: The underlying data model of an OData service is described in a so-called *Entity Data Model (EDM)* and exposed via a so-called *metadata document*. In Section 2.3.2, this model will be discussed in detail.

- URL conventions: The URL construction inside an OData service should follow the rules that are proposed by OData. The URL-conventions, as well as so-called *system query options* and other aspects covered by the URL conventions proposed by OData will be discussed in Section 2.3.3.

- Service Model: Besides the data model, there is also the service model. In the *service document*, all entity sets, functions and singletons are listed, that can be retrieved by a client from an OData service. The service document is available at the root URL of the OData RESTful web service. Only those resources, which can be retrieved based on the root URL of the service, are listed in the service document. Following the URL conventions combined with the information gained from the service and the metadata document, URLs can be built to address any resource exposed by an OData service, given the RESTful web service conforms with OData. Figure 2.3 shows a sample service document. The difference between the service document and the metadata document is that a service document can be used to navigate the model, while the metadata document can be used to understand how to interact with entities in the service.

- Versioning: On the one hand, there is the protocol versioning, which states that clients shall include the `OData-MaxVersion` header in their requests, in which they articulate the maximum acceptable response version. On the other hand, the model versioning defines that in case of breaking changes of the data model, a new service version needs to be provided at a different service root URL. Also, a definition of "safe changes" is given, which can be made on a model without requiring a new service version.

- Headers and formats: Semantics for a set of general request and response headers (for example `Content-Type`) are defined in detail, as well as the semantics for OData-specific headers (for example `OData-MaxVersion`). It is defined whether these headers should/must be included in a request/response. Also a server's response in certain error cases is defined in the OData specification. One example: By providing the `Accept` header, a client may request a particular response format. In case the server does not support the requested format, it must reply with `HTTP 406 Not Acceptable`. Concerning formats, it is defined that at least JSON or Atom as representation formats must be supported by any OData service.

- Response status codes: The common response status codes are summarized and their meaning is conveyed in the specification. But, introducing the definition, it is explicitly stated that [PHZ16a]:

  > *"An OData service MAY respond to any request using any valid HTTP status code appropriate for the request. A service SHOULD be as specific as possible in its choice of HTTP status codes."*

- Data Requests: Several types of data requests, their connection to input/output headers and response status codes, the expected behavior of server and client side and other issues are specified. Beside basic read and write requests, the specification also covers media entities, operations, asynchronous requests and batch requests.

- Security: The specification is rather permissive on the security aspect. It only provides starting points for further investigation on security considerations. If authentication is required, it is specified that basic authentication over HTTPS should be supported to raise the interoperability with generic clients to the highest level. Additionally, other authentication methods may be supported according to the specification.

- Also, other topics like "Extensibility" (of query options, of the payload, etc.) and a "Context-URL", which should be present in each response payload as attribute, are covered by the specification, but will not be elaborated in this thesis in detail.

As OData is a rather comprehensive specification, three levels of conformance for an OData service were defined by the authors:

- OData Minimal Conformance Level
- OData Intermediate Conformance Level
- OData Advanced Conformance Level

Furthermore, it is defined that interoperable OData clients can expect to work with OData services that comply at least with the Minimal Conformance Level and implement the JSON format. The authors of the specification are aware of the fact that not all of these conventions will be followed by all OData services. They state therefore [PHZ16a]:

> *"Not all services will support all of the conventions defined in the protocol; services choose those conventions defined in OData as the representation to expose that functionality appropriate for their scenarios. [...] Services are encouraged to support as much additional functionality beyond their level of conformance as is appropriate for their intended scenario."*

```
{
  "@odata.context": "http://services.odata.org/TripPinRESTierService/$metadata",
  "value": [
    {
      "name": "People",
      "kind": "EntitySet",
      "url": "People"
    },
    {
      "name": "Airlines",
      "kind": "EntitySet",
      "url": "Airlines"
    },
    {
      "name": "Airports",
      "kind": "EntitySet",
      "url": "Airports"
    },
    {
      "name": "NewComePeople",
      "kind": "EntitySet",
      "url": "NewComePeople"
    },
    {
      "name": "Me",
      "kind": "Singleton",
      "url": "Me"
    }
  ]
}
```


Figure 2.3: The service document of the TripPinRESTierService, retrieved by calling GET `http://services.odata.org/TripPinRESTierService`.

This section gave an overview on the content of the OData specification. In the next two sections, first the Entity Data Model and then the URL-conventions will be discussed in detail. A deep understanding of these two topics is essential for understanding and elaborating the design of the generic OData client application prototype.

2.3.2 Entity Data Model

OData services are described in terms of an Entity Data Model (EDM). According to the specification of the protocol [PHZ16a], the provided way of describing the data and the data model in a uniform way is the unique feature of the OData protocol. The concepts listed below are central in the EDM (sources of the following list are [PHZ16a] and [PHZ16c]):

- *Entity:* Any uniquely identifiable record is an entity. Entities are instances of *Entity Types*.
- *Entity Type:* An entity type is a nominal (named) structured type with a key. An

entity type defines properties and relationships of an Entity. Single inheritance is enabled in the EDM, thus an entity type may be derived from another one. The key of an entity type is defined as a subset of the primitive properties that are defined for the entity type (for example `CustomerId` given an entity type called `Customer`).

- *Complex Type:* Complex types are nominal structured types without a key, which means that complex types are always referenced in the context of an entity type. A complex type consists of a set of properties.
- *Property:* There are two kinds of properties that can be defined:
  1. *Structural Property:* A structural property is a property of either a *Primitive Type* (for example Boolean, Byte etc.; OData defines a set of primitive types), a *Complex Type*, an *Enumeration Type* (such types represent a series of related values) or a collection of one of the mentioned types.
  2. *Navigation Property:* A navigation property represents a relationship between entity types. The multiplicity of the relationship, composition relationships as well as referential constraints and on-delete behavior can be modelled as attributes of the relationship.
- *Entity Set:* Entity sets are named collections of entities (`Orders` is for example an entity set containing `Order` entities). There can be multiple entity sets that use the same entity type.
- *Operation:* There are two types of operations defined by OData. Both types have in common that they can either be bound to a type or unbound (in that case they are called "static operations"):
  - *Function:* Functions are Operations that do not allow side effects. Functions may be further composed with filter operations, functions or an action.
  - *Action:* In constrast, an action allows side effects. Also, actions may not be further composed with filter operations, functions or another action.
- *Vocabulary and Annotation:* Vocabularies and annotations allow to annotate metadata as well as instance data. While metadata annotations can be used to define characteristics of a metadata element (for example the characteristics of a property or an entity type), instance annotations can provide additional information associated with an element (for example, in case a property of a certain entity is read-only). A vocabulary contains a set of terms, while an annotation applies such a term to a model element.

There is exactly one entity model that is exposed by an OData service. The Common Schema Definition Language (CSDL) defines an XML representation of the entity data model exposed by an OData service. The entity model is exposed by a single CSDL

```
<?xml version="1.0" encoding="utf-8"?>
<edmx:Edmx Version="4.0" xmlns:edmx="http://docs.oasis-open.org/odata/ns/edmx">
  <edmx:DataServices>
    <Schema Namespace="Microsoft.OData.Service.Sample.TrippinInMemory.Models" xmlns="http://docs.oasis-open.org/odata/ns/edm">
      <EntityType Name="Person">
       <Key>
           <PropertyRef Name="UserName" />
       </Key>
       <Property Name="UserName" Type="Edm.String" Nullable="false" />
       <Property Name="FirstName" Type="Edm.String" Nullable="false" />
       <Property Name="LastName" Type="Edm.String" />
       <Property Name="MiddleName" Type="Edm.String" />
       <Property Name="Gender" Type="Microsoft.OData.Service.Sample.TrippinInMemory.Models.PersonGender" Nullable="false" />
       <Property Name="Age" Type="Edm.Int64" />
       <Property Name="Emails" Type="Collection(Edm.String)" />
       <Property Name="AddressInfo" Type="Collection(Microsoft.OData.Service.Sample.TrippinInMemory.Models.Location)" />
       <Property Name="HomeAddress" Type="Microsoft.OData.Service.Sample.TrippinInMemory.Models.Location" />
       <Property Name="FavoriteFeature" Type="Microsoft.OData.Service.Sample.TrippinInMemory.Models.Feature" Nullable="false" />
       <Property Name="Features" Type="Collection(Microsoft.OData.Service.Sample.TrippinInMemory.Models.Feature)" Nullable="false" />
       <NavigationProperty Name="Friends" Type="Collection(Microsoft.OData.Service.Sample.TrippinInMemory.Models.Person)" />
       <NavigationProperty Name="BestFriend" Type="Microsoft.OData.Service.Sample.TrippinInMemory.Models.Person" />
       <NavigationProperty Name="Trips" Type="Collection(Microsoft.OData.Service.Sample.TrippinInMemory.Models.Trip)" />
       </EntityType>
       <EntityType Name="Airline">
           <Key>
               <PropertyRef Name="AirlineCode" />
           </Key>
           <Property Name="AirlineCode" Type="Edm.String" Nullable="false" />
           <Property Name="Name" Type="Edm.String" />
       </EntityType>
       <EntityType Name="Airport">
           <Key>
               <PropertyRef Name="IcaoCode" />
           </Key>
           <Property Name="Name" Type="Edm.String" />
           <Property Name="IcaoCode" Type="Edm.String" Nullable="false" />
           <Property Name="IataCode" Type="Edm.String" />
           <Property Name="Location" Type="Microsoft.OData.Service.Sample.TrippinInMemory.Models.AirportLocation" />
       </EntityType>
       <ComplexType Name="Location">
           <Property Name="Address" Type="Edm.String" />
           <Property Name="City" Type="Microsoft.OData.Service.Sample.TrippinInMemory.Models.City" />
       </ComplexType>
```

Figure 2.4: An excerpt of the metadata document of the TripPinRESTierService, retrieved by calling GET `http://services.odata.org/TripPinRESTierService/$metadata`.

document, which is also called the metadata document. It is machine-readable, due to its format (XML), which is especially important when building a generic client application. The reason is that such an application does not know the entity model of the service it will invoke in advance and thus needs to resolve it. The metadata document is available at the metadata URL, which is formed by appending `$metadata` to the service root URL. Figure 2.4 shows a part of a sample metadata document. In the metadata document, beside the entity model itself, there is exactly one *entity container* that defines the resources exposed by this services (entity sets and their navigation properties, singletons [12], function and action imports [13]). The difference between the service document and the metadata document, which might be raised here, has already been explained in Section 2.3.1.

The OData specification, of course, goes into more detail at the description of each of the presented concepts. However, the presented aspects should provide a sufficiently detailed overview of the Entity Data Model. In later sections, terms that have been introduced in this section will be used to explain the design and implementation of the prototypes.

[12] A singleton is a single entity that can be addressed directly without having to know its key.

[13] Function and action imports are used to expose a function or action, which is defined in an entity model, as a top level resource in the service.

Figure 2.5: OData URL and its components [PHZ16b]

From the perspective of a generic client application, the next section will cover another important aspect of the OData specification: the URL conventions.

2.3.3 URL Conventions

The URL conventions are covered by a separate document of the OData specification [PHZ16b], which is the source of information used for this section. In addition to the specification, there is also an ABNF which defines the rules for the construction of an URL.

In the context of OData, a URL consists of at most three parts: the *service root URL*, the *resource path* and *query options* (as shown in Figure 2.5).

Service root URL

The service root URL identifies the root of the OData service. As already stated in Section 2.3.1, the service document can be retrieving with a GET request to the service root URL, which makes it an ideal entry point for exploring the resources of the OData service.

Resource Path

In the specification, it is stated that the definitions for the resource path are optional. However, OData services should follow the proposed path construction for consistency and transparency reasons.

First, the URL for submitting batch requests to a service is defined by the resource path `$batch`.

There are several possibilities to address entities. The ABNF, which has been mentioned at the beginning of this section, defines the syntax rule for the resource path. Here are some examples to illustrate the most relevant parts of the syntax rule for this thesis:

- A collection of entities can be addressed via an entity set:
  `http://host/service/Orders`
- A single entity can be addressed by using the entity key. The entity key is provided in brackets. Here is an example, given the entity key consists of one numeric attribute: `http://host/service/Orders(2)`

If the entity key is a string attribute, the key needs to be provided inside single quotation marks:
`http://host/service/Orders('id1')`

- If there is a relationship between two entites (e.g. one order relates to a collection of products), the URL for retrieving the list of products of a certain order would be:
  `http://host/service/Orders(2)/Products`
  If there is a one-to-one relationship between two entities, an example for a URL would then be:
  `http://host/service/Products(2)/Supplier`
- The reference between entities itself (i.e. the relationship) can be addressed by appending `/$ref` to the constructed resource path:
  `http://host/service/Orders(2)/Products/$ref`
  A single entity reference between two entities is addressed by appending the query parameter `$id` followed by the absolute or relative (to the request URL) entity-id:
  `http://host/service/Orders(2)/Products/$ref?$id=../../Products(0)`
  This is especially relevant when trying to unreference two entities by sending a DELETE request to such a URL.
- The media stream of a media entity can be accessed by appending `$value` to the resource path of the media entity:
  `http://host/service/Pictures(123)/$value`

Specificly at the navigation from one entity to another, the content of the metadata document needs to be taken into consideration. In the metadata document, the URL paths of navigation properties are also returned (in the entity container) and shall be used at the construction of the resource path of the URL.

Query options

Three types of query options are distinguished in OData: Besides *custom query options*, whose only limitation is that they must not begin with a $ or @ character, and *parameter aliases*, which must start with an @ character and may be used to reference primitive/-complex/collection values inside the query options section of the URL, there are *system query options*. System query options are query string parameters, which start with a $ character. They control the amount and order of data returned for the identified resource. Not each system query option is applicable to each URL: Depending on whether a single entity or a collection is addressed by the URL, certain system query options might not be qualified for application. The ABNF also covers the grammar and syntax for system query options. There are the following system query options (in Table 2.4 an overview on the available system query options is given):

- `$filter`: Allows filtering of collection resources. The client needs to send a filter-expression, which is evaluated for each collection item. Only those collection items where the filter expression evaluates to `true` are included in the response. There is a set of logical operators (for example *equals*, *greater than*, etc.) as well as a set of arithmetic operators (for example *addition*, *subtraction*, etc.), diverse string functions (for example *contains*, *endswith*, etc.) and other functions and operators which might be used to create such a filter expression.

- `$expand`: The client can state related resources that he wants to be included in the response inline together with the retrieved resources.

- `$select`: A client can request only a specific set of properties for each entity in the result set.

- `$orderby`: By using this system query option, the client can define a particular order in which he expects the items to be returned.

- `$top` and `$skip`: System query option `$top` allows the client to limit the number of entities returned in the collection, while `$skip` defines the number of items that should be skipped (starting with the first item of the collection) and thus, not be included in the result. In combination, these two parameters can be used as pagination mechanism.

- `$count`: The count option allows the client to tell the server to include the count of items within the returned collection. The count of items is returned additionally to the result set, not instead of it.

- `$search`: By using search, a client can request only those entities matching the search expression. The OData specification for this query option is rather permissive, as the semantics of what is considered a match solely depends upon the service and its implementation.

- `$format`: This system query option is meant for clients which do not have access to request headers. They can use `$format` for content-type negotiation.

It is, of course, possible to combine system query options. In such cases (as already shown in the example of `$top` and `$skip` in Table 2.4) the system query options need to be concatenated by a '&' character. The relevance of the system query options for this thesis will become visible at the implementation design of the generic client application. The next section will deal with the comparison of OData to REST. As the goal is to offer Open Data via RESTful web services, it needs to be confirmed that OData conforms with the architectural style REST.

System query parameter	Applicability	Example(s)[a]

`$filter`	collection	Return all products whose name is "Milk" and whose price is lower than 2.55:
		`http://host/service/Products?$filter=Name eq 'Milk' and Price lt 2.55`
		
		Return all products where the name does not end with "ilk":
		`http://host/service/Products?$filter=not endswith(Name,'ilk')`

`$expand`	collection / single entity	Retrieve all products and the category of each product (product and category are separate entity types, but there is a navigation property from products to their categories): `http://host/service/Products?$expand=Category`
`$select`	collection / single entity	Select only the name and the price of the products: `http://host/service/Products?$select=Name,Price`
`$orderby`	collection	Order the products by name ascending and as second criterion by price descending: `http://host/service/Products?$orderby=Name asc,Price desc`
`$top` and `$skip`	collection	Return 12 products while skipping the first two entries: `http://host/service/Products?$top=12&$skip=2`
`$count`	collection	Return all products and the number of products contained in the set: `http://host/service/Products?$count=true`
`$search`	collection	Return all products that are red: `http://host/service/Products?$search=red`
`$format`	collection / single entity	Return the list of products in format `application/json`: `http://host/service/Products?$format=application/json`

Table 2.4: Summary of the available system query options

[a]Spaces and other characters have not been URL encoded in order to improve the readability of the examples.

2.3.4 Comparison OData vs REST

REST is an architectural style in constrast to OData, which defines a set of best practices for building and consuming RESTful web services. OData can more appropriately be compared to the Resource Oriented Architecture, because both ROA (as a RESTful architecture) and OData tie REST services to designated technologies.

First, the commonalities shared by ROA and OData will be discussed. Both ROA and OData use HTTP and URIs as foundation of their specifications. OData fully uses resources and representations as concepts in its specification, as it is used by the ROA. OData contains all central properties propagated by the Resource Oriented Architecture. Addressability is implied by the use of URIs and HTTP; statelessness is also given by using HTTP. Connectedness, although not required by ROA but rather formulated as a suggestion, is reached in several ways: Most obviously, connectedness is achieved by the context-URL, which has been mentioned shortly in Section 2.3.1. The context-URL always links to the relevant portion of the metadata document, where the content of the currently received response is explained. The uniform interface is also a property induced by the correct usage of HTTP and its methods. OData also makes use of the process of content negotiation, which is described by the ROA.

But OData goes beyond the loose specifications of ROA and REST and also tackles a lot of the shortcomings that have been identified in Section 2.2.4:

- The usage of HTTP verbs is strictly & in detail defined by the OData protocol specification [PHZ16a] for several cases (for example creating, reading, updating and deleting resources).
- On the one hand, the flexibility of the format of a representation is still given by OData. But, on the other hand, the specification states that interoperable clients expect an OData service to implement the JSON format, which tackles the issue of lacking interoperability due to the flexibility of the format of a representation.
- There is a machine-readable service description together with a machine-readable data model description, exposed using a formally defined representation (CSDL). This could enable the automated generation of client-stub code.
- The service-description is also a common service discovery mechanism for all OData services.
- Requests and responses as well as the usage of HTTP headers and HTTP status codes are prescribed by OData, both on a general level as well as in relation to cases that are covered by the specification.
- The lack of standardisation in the REST domain and the missing mapping of REST to HTTP (discussed in Section 2.2.4), which have been identified as the root of other problems in the context of RESTful web services, is compensated by the OData specification to a certain extent. Security aspects are only slightly covered

by the OData specification, but beside this issue, many other aspects are covered by OData. The compatibility among OData RESTful web services rises with the conformance level the services comply with.

OData covers additional issues that have not been identified as problems previously. It defines for example the way how an OData service can offer a resource for receiving batch requests. Asynchronous requests are also part of the specification. There is a clear statement on the versioning of an OData REST service with respect to the changes that should be applied.

To summarize, OData is fully compliant with the Resource Oriented Architecture, its properties and the applied technologies. OData solves some of the shortcomings of REST, which mostly originate in missing standarisation in the REST domain. Also, topics which have not been addressed by neither the shortcomings nor the ROA, are part of the specification. In the last section connected to OData, applications which make use of OData and libraries which support the development of OData services & clients will be presented.

2.3.5 Applications and Libraries

On the web page of OData [ODab], there is a list of references to libraries that can be used by developers to implement either an OData client or an OData server. Some libraries are explicitly designated as client / server libraries, while others contain both client and server modules. Libraries exist, according to [ODab], for: .NET, Java, JavaScript, C++, Python, Tcl/Tk and Objective-C. The most libraries are available for .NET, which might be related to the fact that both OData and .NET were created by Microsoft and thus most probably also used in applications developed by Microsoft in the first place.

The web page of OData also lists OData producers and OData consumers [ODaa]. OData producers are services that expose their data conforming with the OData protocol, while OData consumers denote applications that consume data exposed by services that use the OData protocol. There is a variety of available applications on both producer and consumer side. Based on the list, one of the conclusions that can be drawn is that several companies have been founded based on a self-developed application that uses the OData protocol in some way. But, there are also three “big players”, which utilize the OData protocol in some of their products: IBM, Microsoft and SAP. [14] The majority of the presented applications were created by Microsoft.

An interesting product in the context of Open Data, that is based on OData, is the OGDI DataLab. According to its GitHub page [Ope], it is written using C# and the .NET Framework and uses the Windows Azure Platform. It consists of three components:

[14] IBM and Microsoft are foundational sponsors of OASIS, while SAP is also a sponsor of OASIS [OASb]. There could be a connection between the fact that these three companies use OData and the fact that they are sponsors of OASIS, which has declared OData as one of its open standards.

- The Data Service, which is an OData conformant RESTful web service and exposes the data.
- The Data Browser, which is a web application that offers the possiblity to browse the data exposed by the data service and visualize the same in different ways.
- The Data Loader, which is a tool that takes CSV formatted data and publishes it into OGDI.

The Italian Ministry of Health Open Data Portal and the Portuguese Government Open Data Portal are examples for Open Data portals that are built using OGDI DataLab [ODaa].

After defining Open Data, REST and the OData protocol as the main theoretical concepts relevant for my empirical research, I will, in the next chapter, move to the project design.

CHAPTER 3

Project design

3.1 Idea

The idea behind this project has already been roughly discussed in Section 1.2. One of its main goals is to judge whether the target architecture (which is depicted in Figure 1.1) is feasible and to which extent. For this purpose, the Open Data producer as well as the generic Open Data consumer side need to be implemented prototypically. Once both applications are implemented, their interaction can be tested and rated in a real world environment.

The target architecture already induces certain requirements on both prototypes. The Open Data producer must expose its data via an OData Version 4.0 and OData JSON Format Version 4.0 conformant RESTful web service. The Open Data consumer should be able to connect to any such Open Data producer with as little configuration effort as possible. Once it is connected, the Open Data consumer can make use of the OData specification and offer displaying and querying functions on the exposed data sets.

In order to be able to imagine the full functional stack of both prototypes, the use cases will be presented in Section 3.2. In Chapter 4, the use cases will be broken down into specific functions, which need to be realized in order to be able to cover the use cases. Furthermore, the concrete technical design will be depicted in the same chapter.

3.2 Use cases

The UML use case diagram has been employed to depict the use cases that need to be covered by both prototypes. In Figure 3.1, the use cases for the client application are illustrated. Also a non-functional requirement is expressed in this diagram, which states that the configured OData service must expose its data conforming with at least the Minimal Conformance Level of OData Version 4.0 and OData JSON Format Version 4.0.

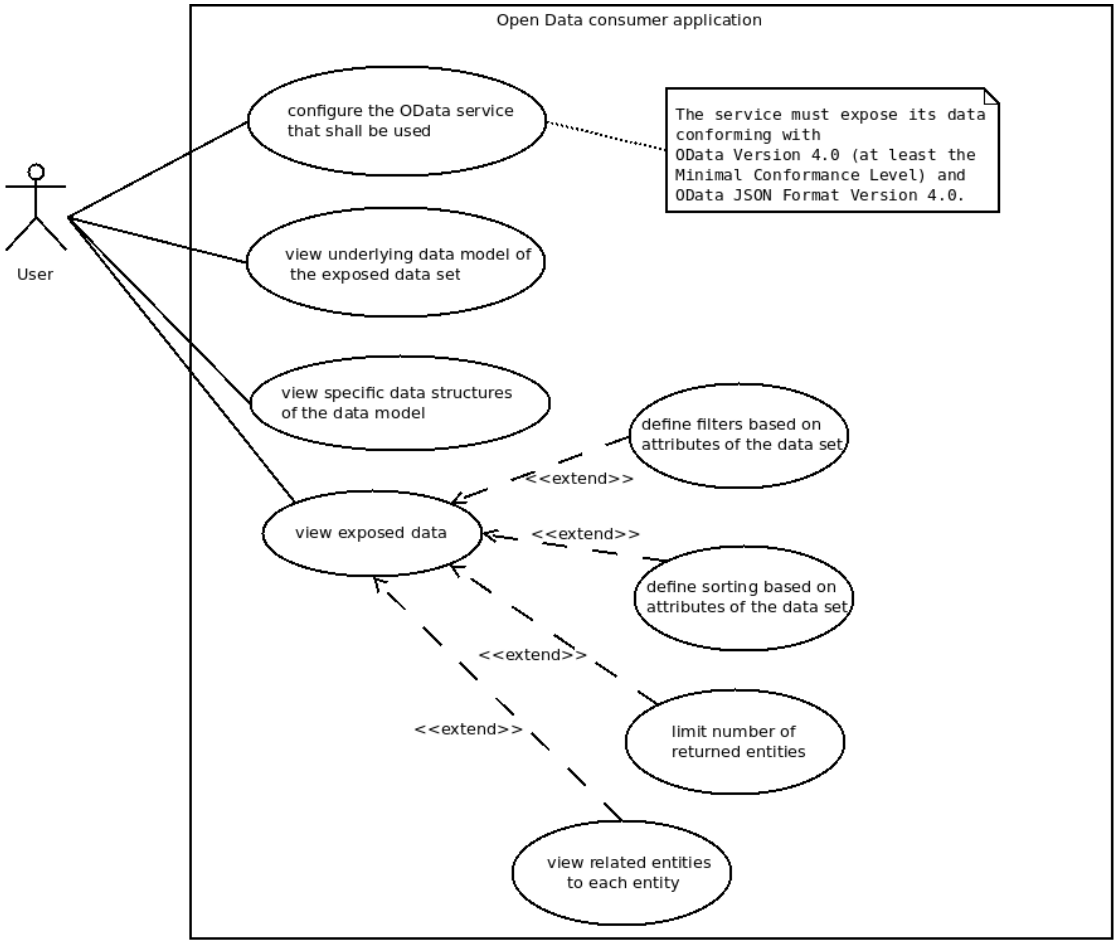

Figure 3.1: Use cases that need to be covered by the generic client application.

This is exactly what interoperable clients can expect from the OData services they are interacting with (this has already been discussed in Section 2.3.1). As it is not directly expressed in the diagram, it needs to be highlighted that the successful execution of use case "configure the OData service that shall be used" is the precondition for all other use cases.

On the other hand, in Figure 3.2 those use cases are shown, that need to be covered by the OData REST service. This diagram contains the symmetric non-functional requirement, that the service must expose its data conforming with at least the Minimal Conformance Level of OData Version 4.0 and OData JSON Format Version 4.0 in order to meet the expectations of interoperable clients. The underlying data model was defined based on requirements of the stake holders. In the next section, more details on the stake holders and on the concrete project will be given.

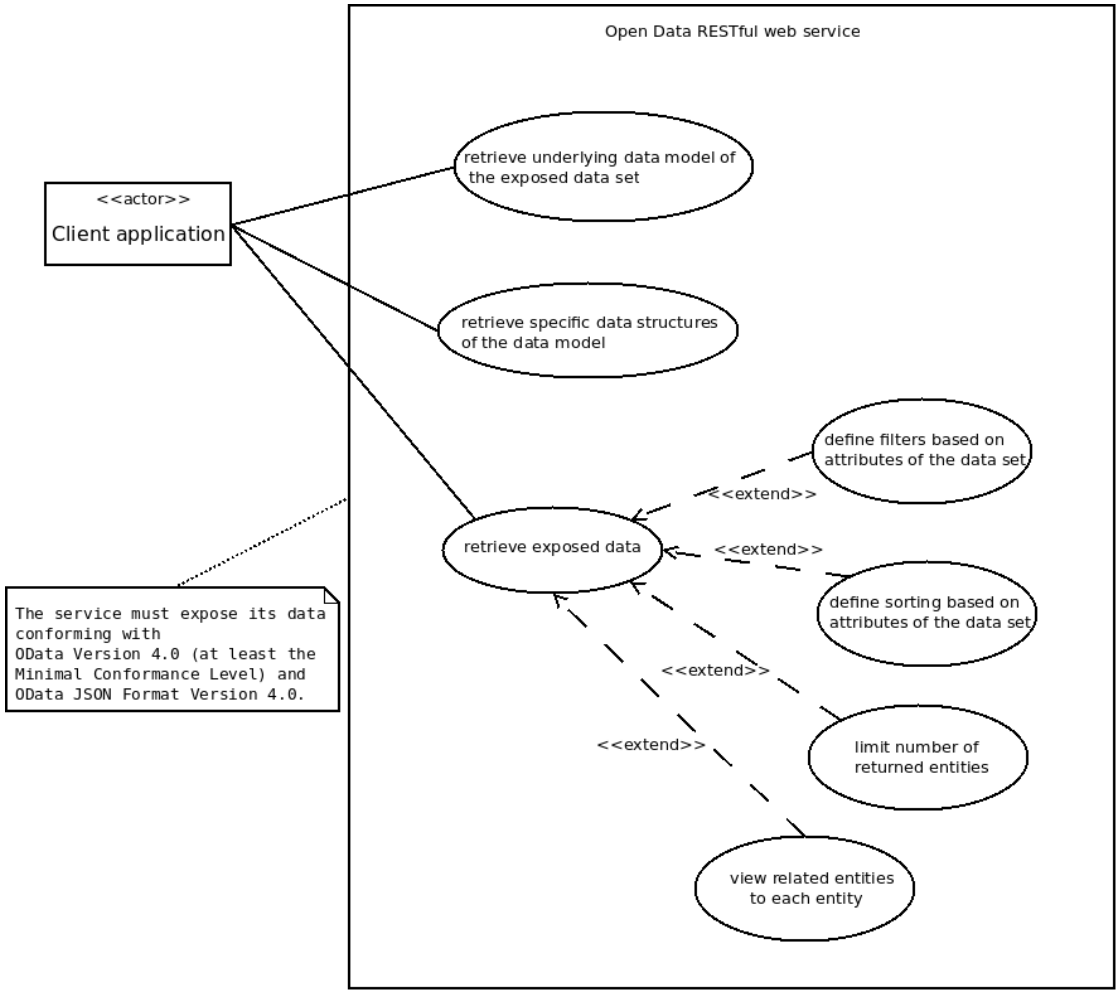

Figure 3.2: Use cases that need to be covered by the Open Data RESTful web service.

3.3 Additional proof-of-concept application

The data model that is exposed by the OData RESTful service, has been designed based on requirements of a project of the Technical University of Vienna together with the Vienna Chamber of Commerce. Its goal was to provide a platform, which exposes the research infrastructure of the university, that might be used by third parties (e.g. small businesses). But the platform should go beyond the displaying functionality of such data – it should also allow single departments/entities to record their infrastructure and to edit it. Supervisors can either approve the entered infrastructure, which makes the same publicly available, or deny it. In case it has been denied, the department needs to supply more information about the same. The use cases are illustrated in Figure 3.3.

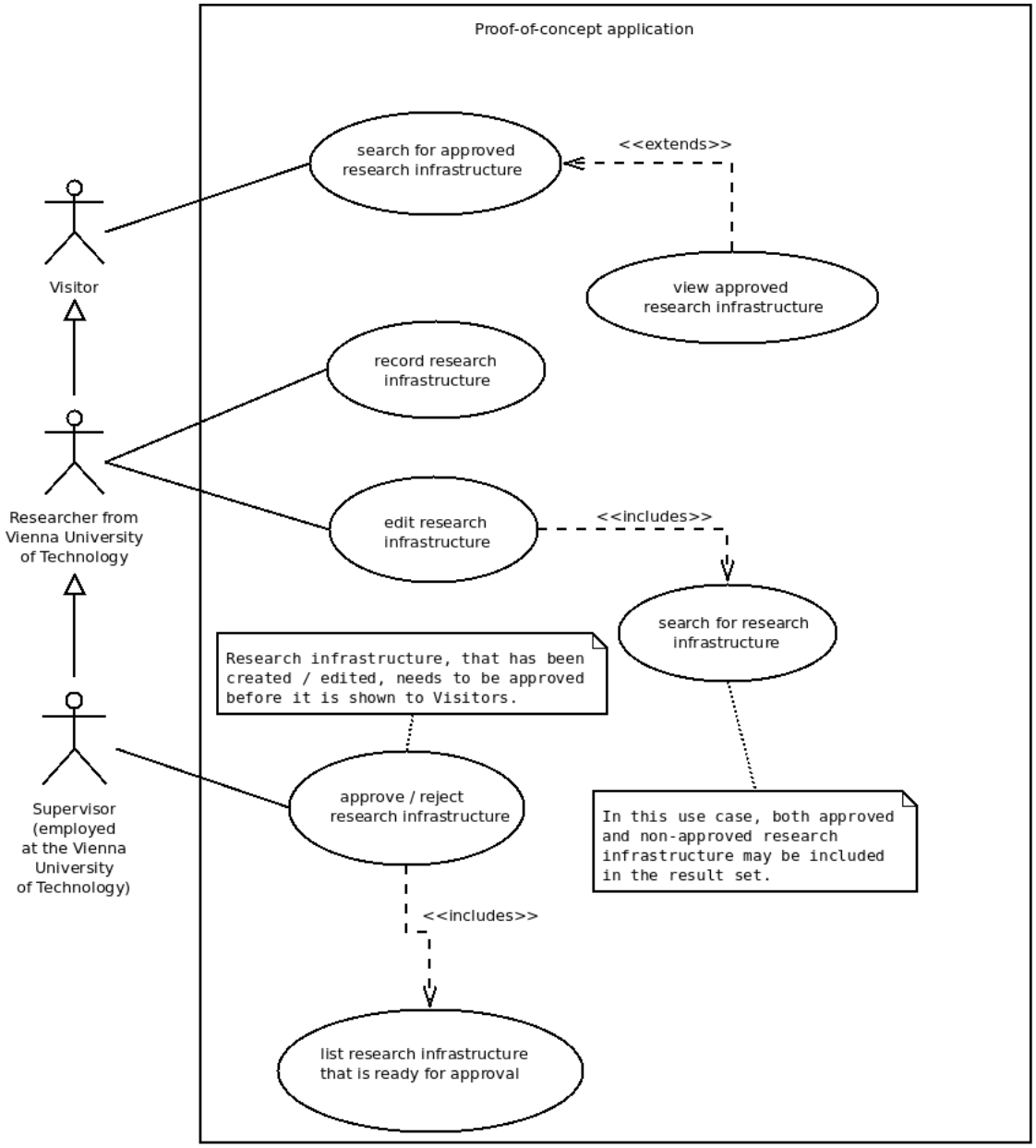

Figure 3.3: Use cases that need to be covered by the proof-of-concept application.

As there is already a portal which displays such infrastructure on national level[1], one requirement was to make the data model compatible with the data model of the already existing portal on national level. This has been achieved by collaborating with the creators of the portal, which opened up the most relevant parts of the data model and thus made it possible to design a data model, which is an extension of their data model.

Furthermore, the writing functionality of the portal made it necessary to implement write functionality on the server side. Therefore, the OData RESTful web service has been enhanced by adding write requests, media entity requests (for transferring images of the infrastructures) and other functionality according to the OData specification. This application is not directly relevant for the topic of this thesis, as the focus lies on the generic usage of read functionality of OData RESTful web services by a client application. Nevertheless, the implementation of this particular application enabled me to gain more insight into the implementation of writing and other functionality according to OData. This experience allows me to add input in the conclusion of my thesis with respect to the second research question of my thesis, which goes beyond the theoretical knowledge gained about the OData protocol.

[1] `https://forschungsinfrastruktur.bmwfw.gv.at/`. Visited: November 13, 2016

CHAPTER 4

Implementation

4.1 Implementation of OData REST service

4.1.1 Functions / Backlog

In the previous section, the use cases of the client application, as well as the use cases of the OData RESTful web service application were identified. Now, the implementation starts with the OData REST service. The reason for starting with the implementation of the OData REST service is the fact that the client application will need an available OData REST service to which it shall connect.

The use cases identified in Section 3.2 will be divided into single functions, which are required to enable the use cases. These functions need to be implemented:

- *Retrieve underlying data model of the exposed data set*: This use case can be enabled by implementing the metadata document of OData. By exposing the metadata document, the data model is exposed in a machine-readable way. It must be machine-readable, because the actor, which is triggering the use case, is itself a machine (the client application).
- *Retrieve specific data structures of the data model*: Data structures, that are used in the data model (e.g. complex types), are also described in the metadata document. Thus, the requirements for this use case are met once the metadata document is implemented.
- *Retrieve exposed data*: The concrete data needs to be exposed by the REST service. Thus, the resources and the corresponding representations need to be defined, which can be retrieved by the client application first. It must be possible to address collections as well as single entities. The URLs of the resources as well as the

payload and the need to conform with OData Version 4.0 and OData JSON Format Version 4.0.

- *Define filters based on attributes of the data set*: Filters may be applied only to collections of entities. The OData-conformant way of defining filters is by using the system query option `$filter`. Thus, this system query option needs to be implemented in order to cover this use case.
- *Define sorting based on attributes of the data set*: An order of the entities returned in a collection can be defined by making use of the system query option `$orderby`.
- *Limit number of returned entities*: Limiting the number of returned entities in a collection is done, when conforming with OData, through system query option `$top`.
- *View related entities to each entity*: In case there is a relation between entities, this relation needs to be described properly in the metadata document. The service must be capable of resolving URLs that navigate from one entity to another or to a set of entities (depending on the type of relation). These URLs are expected to be constructed according to the URL conventions of OData.

There is also a general premise, which is valid for all mentioned functions: The RESTful web service must conform at least with the Minimal Conformance Level of OData Version 4.0. This means that all requirements, that are imposed by the Minimal Conformance Level of OData Version 4.0, need to be met by the OData REST service, even if they are not covered by the already mentioned ones.

4.1.2 Design

First the database schema will be discussed, at it constitutes the foundation of the service. The database schema, which is used by this application, is shown in Figure 4.1. In general, there are two types of tables in this database schema: tables that contain static data and tables that contain dynamic data.

By the term static data I mean that the values in the database are inserted once and rarely changed. The following tables belong to this class of tables:

- `Finanzierungsart` (stores different types of financing)
- `Nutzungsart` (stores different types of use)
- `Oefos` (stores the Austrian fields of science)
- `InfrastrukturKategorie` (stores different types of infrastructure categories)
- `Institutionstyp` (stores different types of institutions)

- `Finanzierungsart_Institutionstyp` (stores the types of financing for each type of institution)
- `Infrastrukturart` (stores different infrastructure types)

The values from these tables are used to describe or classify other entities. A research infrastructure, for example, can belong to multiple infrastructure categories. The initial values for these static data tables have been provided by the creators of the Austrian research infrastructure portal.

Institutions (stored in table `Institution`), users (stored in table `User`), research infrastructure (stored in tables `Fi` and `FiVersion`), contact persons for the infrastructure (stored in table `Kontakt`), images of the infrastructure (stored in table `Bild`), services that are related to the research infrastructure (stored in table `Dienstleistung`), costs of acquisition (stored in table `FiInternAnschaffungskosten`), reinvestments (stored in table `FiInternReinvestition`) and various key data (stored in table `FiInternJahr`) belong to the class of tables that contain dynamic data. The remaining tables are junction tables, which store the data belonging to many-to-many relationships between the already mentioned tables.

As already discussed in Section 3.3, the creators of the Austrian research infrastructure portal have opened up the most relevant parts of their data model for this implementation. They have provided parts of their database schema as `.mwb`[1] file, which has been used as the initial schema and extended according to additional requirements.

The whole prototype has been implemented using Java 7. At the time of implementation, there was only one "featured" OData 4.0 library available for Java 7: Apache Olingo. Thus, it has been used as OData library for the service. Beside Apache Olingo, various Spring libraries (e.g. Spring JPA, Spring Boot) have been used to realize the prototype. A MySQL database stored the data in the previously described database schema. Build and dependency management is covered by Apache Maven. Apache Tomcat 7 is the Servlet container, which is used to run the service.

In the next section, the results of the implementation will be presented.

4.1.3 Result

The result of the implementation is a RESTful web service, which fulfills all requirements that have been defined in Section 4.1.1. First, the service had to be capable of receiving requests and responding to the same. When using solely Apache Olingo, one would implement the method `void service(ServletRequest req, final ServletResponse resp)` of Java's `HttpServlet` class. In this method, the request and response would be delegated to method `void process(HttpServletRequest request, HttpServletResponse response)` of Apache Olingo's class

[1] MySQL workbench

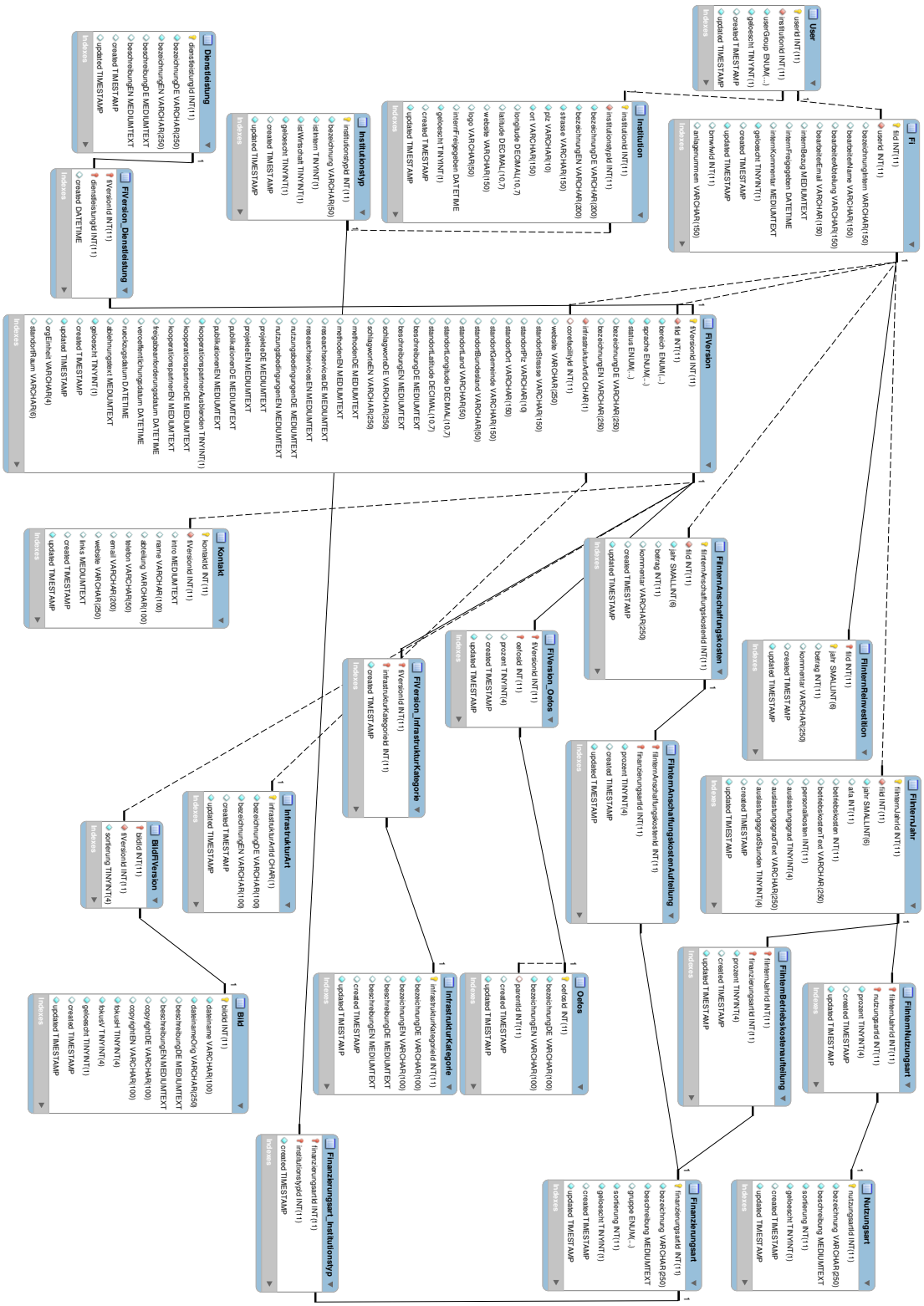

Figure 4.1: The database schema of the service.

`ODataHttpHandler`. The `process(...)` method of this class processes such a request as an OData request and performs following actions [Foua]:

- parsing the URI
- doing the content negotiation
- dispatching the request to a specific `Processor` implementation for handling the request (the `Processor` interface will be discussed later in this section)
- creating the serialized content for the response object

Because I wanted to make use of various Spring libraries (starting with Spring Boot) together with Apache Olingo, I manually needed to make Spring compatible with Apache Olingo. I had to create a Controller class, which is annotated as `RestController` (this is a Spring annotation). All incoming requests would be processed by the method `ResponseEntity<String> process(HttpServletRequest req)` of the same class. Due to the missing `HttpServletResponse` at this point, I could not simply call the above mentioned `process` method of the `ODataHttpHandler`. Instead, I had to use the method `ODataResponse process(final ODataRequest request)`, offered by the interface `ODataHandler`, which itself is a superclass of `ODataHttpHandler`. So I was required to manually convert a `HttpServletRequest` to an `ODataRequest` in order to be able to forward the request to Apache Olingo. Then, the received `ODataResponse` had to be converted to a `ResponseEntity<String>` to finish the request-response cycle.

Due to the fact that Apache Olingo's `ODataHttpHandler` performs the above-mentioned tasks, it needs to know the EDM of the service. For this purpose, Apache Olingo's server library requires the application to implement the `CsdlAbstractEdmProvider` interface. By implementing this interface, the whole Entity Data Model is exposed to Apache Olingo. Beside the quoted tasks of the HTTP handler class, also the metadata document and the service document are exposed by Olingo out-of-the-box once the `CsdlAbstractEdmProvider` interface has been implemented. Thus, based on the database schema, the Entity Data Model of the service has been defined. Entity types, their properties and entity sets have been defined first. The relationships between entities have been realized as navigation properties, which point from one entity type to another. Enumeration types have been used for database fields of type ENUM. Additionally, I made use of annotations (explained in Section 2.3.2). I have added a short description to each structural property of an entity type by using an annotation called *description*. In Figure 4.2, an excerpt of the resulting metadata document is depicted.

In the next step, I wanted to expose data from the database. For this purpose, Apache Olingo offers several `Processor` interfaces. The HTTP handler class of Olingo dispatches the request to the corresponding processor. For the purpose of this thesis, the `EntityCollectionProcessor` (which exposes collections of entities) and the `readEntity` method of `EntityProcessor` have been implemented.

```
<EntityType Name="infrastrukturart">
    <Key>
        <PropertyRef Name="id"/>
    </Key>
    <Property Name="id" Type="Edm.String">
        <Annotation Qualifier="description">
            <String>Die Id der Entität</String>
        </Annotation>
    </Property>
    <Property Name="created" Type="Edm.DateTimeOffset" Precision="3">
        <Annotation Qualifier="description">
            <String>Das Erzeugungsdatum des Datensatzes.</String>
        </Annotation>
    </Property>
    <Property Name="updated" Type="Edm.DateTimeOffset" Precision="3">
        <Annotation Qualifier="description">
            <String>Das letzte Änderungsdatum des Datensatzes</String>
        </Annotation>
    </Property>
    <Property Name="bezeichnungDE" Type="Edm.String">
        <Annotation Qualifier="description">
            <String>Die Bezeichnung der Entität auf Deutsch.</String>
        </Annotation>
    </Property>
    <Property Name="bezeichnungEN" Type="Edm.String">
        <Annotation Qualifier="description">
            <String>Die Bezeichnung der Entität auf Englisch.</String>
        </Annotation>
    </Property>
    <NavigationProperty Name="forschungsinfrastrukturen"
         Type="Collection(at.tuwien.infrastruktur.forschungsinfrastruktur)"
         Partner="infrastrukturart"/>
    <NavigationProperty Name="aktuellsteforschungsinfrastrukturen"
         Type="Collection(at.tuwien.infrastruktur.aktuellsteforschungsinfrastruktur)"
         Partner="infrastrukturart"/>
</EntityType>
```

Figure 4.2: An excerpt of the metadata document of the implemented OData RESTful web service.

Both methods receive as argument all information that is necessary to process the request properly and completely. These classes are also the entry point to the major part of the implementation. In these processor classes, a corresponding method of a custom `IDataProviderService` is called. It forwards the request to a class implementing the abstract class `AbstractDataService<A extends AbstractDao>`. `AbstractDao` is the abstract class for all `Dao`-classes, which are used together with Spring JPA repository classes to access the database. The concrete data services are responsible for fetching the corresponding data from the database.

In order to enable the `$filter` system query option, I have implemented Apache Olingo's interface `ExpressionVisitor<Object>`. Depending on the operator type(s) and the affected model elements of the filter expression), certain methods of the class are invoked. It is a rather generic interface, allowing arbitrary return types at each method. I have decided to implement the `ExpressionVisitor` in a way that ultimately

returns a Spring JPA `Specification<? extends AbstractDao>`. In this way I am able to pass the resulting `Specification` class to the JPA `Repository` classes, which makes the database return the appropriate entities. Especially for larger data sets, this might result in a performance gain compared to a filtering that would be done programmatically. Sorting and limiting the number of entities in the result set (i.e. pagination) are also completely handled by the database by passing a class that implements interface `Pageable`, which is also an interface originating in Spring JPA.

The system query option `$expand` has been implemented beside all system query options that have been stated in Section 4.1.1. Furthermore, navigation from one entity to another entity or another entity set has been implemented. But there is a limitation: Navigation has only been implemented up to one level. Here is an example, which should illustrate this limitation: Assuming that an entity *account* is related to one or multiple *customer* entities, then we could theoretically return the list of customers for one account by calling `GET http://<root of the service>/accounts(<account id>)/customers`. If we further assume, that one customer entity is related to one or more car entities, then it would be possible to get the list of cars by calling `GET http://<root of the service>/customers(<customer id>)/cars`, but it would not be possible to get the same list via `GET http://<root of the service>/accounts(<account id>)/customers(<customer id>)/cars`. In such a case, the service would respond with HTTP 501. It must be highlighted, that such a behavior is compliant with the OData specification.

Moreover, not all operators, which are defined by the OData specification and thus might be used in the `$filter` expression, have been implemented. The arithmetic operators, for example, have not been implemented, as they have not been considered relevant for defining filters. However, the service again responds with HTTP 501, which is the OData conformant answer, in case certain requested functionality has not been implemented on server side [PHZ16a]. In general, all criteria that are stated in the OData Minimal Conformance Level definition (except those, that are defined for *Updateable OData Services*, because the implemented service is not an updateable service), are fulfilled by the service. On top of this some criteria of the intermediate and even the advanced level are fulfilled (e.g. supporting system query option `$filter` is a criterion of the intermediate level). After discussing the results of the OData RESTful web service, I will move on to the implementation of the generic OData client application.

4.2 Implementation of a generic OData client (web application)

4.2.1 Functions / Backlog

Analogous to the OData RESTful web service, the identified use cases, that shall be covered by the client application (Open Data consumer application), were divided into single functions, which shall be implemented:

- The use case *configure the OData service that shall be used* requires a page, where the user can provide the information that needs to be known to the application in order to be able to communicate with the OData service.
- Once the OData service is configured, the use cases *view underlying data model of the exposed data set* and *view specific data structures of the data model* can be covered by reading the metadata document of the underlying OData service.
- The precondition for realizing the base usecase *view exposed data* is, that the OData service has been resolved successfully. The client application needs to know those entity sets that are available from the root of the service. After resolving the OData service, the user must be able to choose the data he wants to see.
- *Define filters based on attributes of the data set* can be realized by providing a user interface, which lets the user define filters based on the attributes of the entities, which shall be retrived. The filter that is defined by the user on the user interface, needs to be translated into a filter expression that can be passed to the REST service via the system query option `$filter`.
- *Define sorting based on attributes of the data set* also requires a user interface, where the user can optionally define a sorting based on attributes of the entities. The entered values should be passed to the OData service by using the system query option `$orderby`.
- The user must have the possibility to enter a maximum number of entities, that shall be displayed on the frontend. This value needs to be passed to the OData service as the value of the system query option `$top` in order to realize the use case *limit number of returned entities*.
- The precondition for executing the use case *view related entities to each entity* is, that the user must previously search for entities. For each entity in the result set of his search, the user must be able to navigate to possibly existing related entities. By following the URL conventions of OData, a corresponding URL can be built for fetching the related entities.

4.2.2 Design

The crucial part of the design is the way the OData service can be resolved by the client application. Once the OData service has been resolved, the client application is able to interact with it and to request information based on the user's input. In Figure 4.3, the process of resolving the OData RESTful web service is visualized.

In general, the client application fetches the service document and the metadata document. The entity sets, which are listed in the service document, are stored first. These entity sets are the entry point to the service, because they are available from the root of the REST service. The entity sets, which are listed in the metadata document, are stored

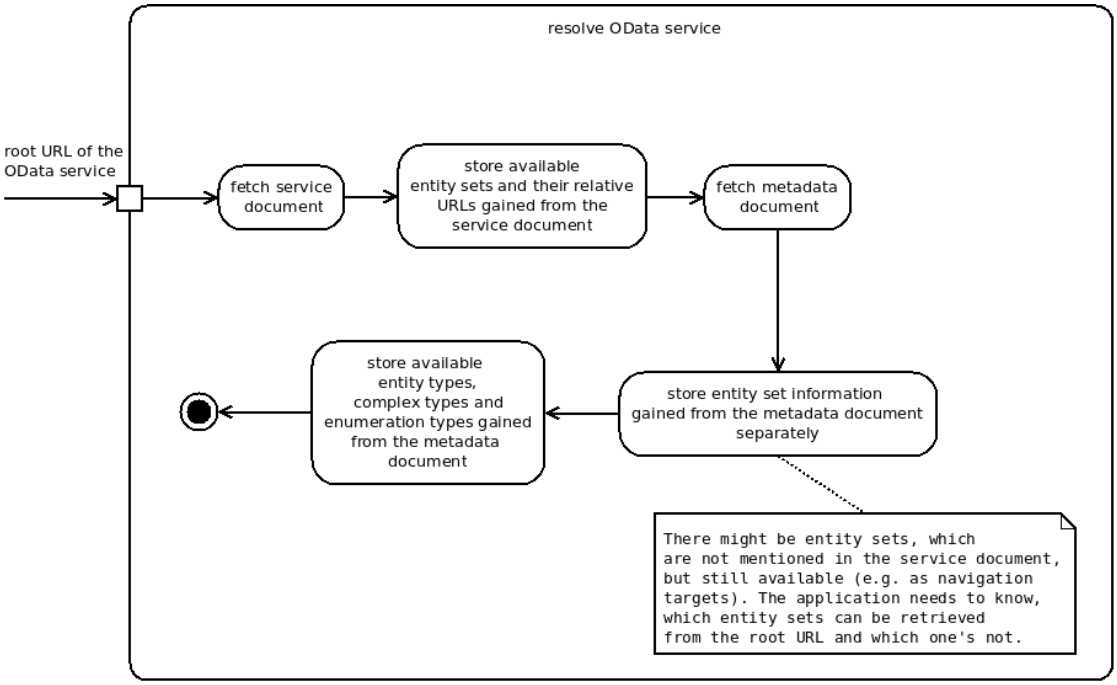

Figure 4.3: A UML activity diagram showing how the client application resolves an OData service.

```
<EntitySet Name="institutionstypen" EntityType="at.tuwien.infrastruktur.institutionstyp">
    <NavigationPropertyBinding Path="finanzierungsarten" Target="finanzierungsarten"/>
    <NavigationPropertyBinding Path="institutionen" Target="institutionen"/>
</EntitySet>
```

Figure 4.4: An entity set element that is returned in the metadata document of the implemented OData RESTful web service.

separately. The reason is that there might be entity sets, which are not available directly from the root, but only as navigation target. The client application needs to distinguish between the entity sets that can be fetched from the root and the ones that cannot be fetched from the root. The entity sets listed in the service document are a subset of the entity sets listed in the metadata document. Furthermore, the metadata document carries the following information about each entity set (in Figure 4.4, a sample entity set element is shown):

- The underlying entity type of the entity set.
- A `NavigationPropertyBinding` element, which itself has two attributes: The `Path` (it denotes the path from the entity type to the navigation property) and the `Target` (the entity set, to which the client can navigate).

Moreover, the entity types, the complex types and the enumeration types, that are returned in the metadata document, are stored in the application. This information is relevant for displaying the data model of the exposed data set. The data model shall be displayed in one or more tabs.

In order to view the exposed data, a separate tab needs to be added. It should contain a form, which enables the definition of filter parameters, sorting parameters and the maximum number of returned entities by the service. For each entity of the result set, it shall be possible to view possibly related entities by clicking on a button / icon. With this design, it is possible to handle all use cases, that are related to viewing the exposed data in a single tab.

The technology stack has mostly been prescribed by the supervisor of the thesis. Java Server Faces as framework for developing web-applications with Apache MyFaces 2.1.7 as its implementation have been used primarily. Version 4.2.0 of Apache Olingo's client library has been employed for accessing the OData services. For build and dependency management, Apache Maven has been used. In order to run the service, the .war file has been deployed to an Apache Tomcat 7. Furthermore, the style sheet and other static resources that influence the general appearance of the web application, have been provided by the Campus Software Development team from the Vienna University of Technology.

4.2.3 Result

The result of this implementation is a web application which is capable of resolving and exploring OData services compliant with version 4.0. The web application itself will be presented first, followed by the limitations of its implementation.

The web application consists of two pages. The first page that is presented to a user of the application is `servicediscovery.xhtml` (see Figure 4.5). On this page, the user is expected to provide the root URL of the OData RESTful web service that he/she wants to browse. The user is not able to perform any other action until he/she enters a valid URL of an OData service and submits the same by clicking on button *Service suchen*.

When clicking on the button, the OData service is being resolved as described in Section 4.2.2. For this purpose, the client library of Apache Olingo is used. The library provides a client factory class (`ODataClientFactory`), which allows the developer to get instances of certain client classes (e.g. `ODataClient`). These client classes themselves can be used to create various OData request classes (e.g. `ODataServiceDocumentRequest` for requesting the service document). There are also classes which encapsulate the retrieved information (e.g. `ClientServiceDocument` for the service document) of a request. There are, of course, several other classes of Apache Olingo, which have been used in the source code, but will not be mentioned now explicitly. It is important to highlight that Apache Olingo provides all necessary infrastructure for interacting with OData services.

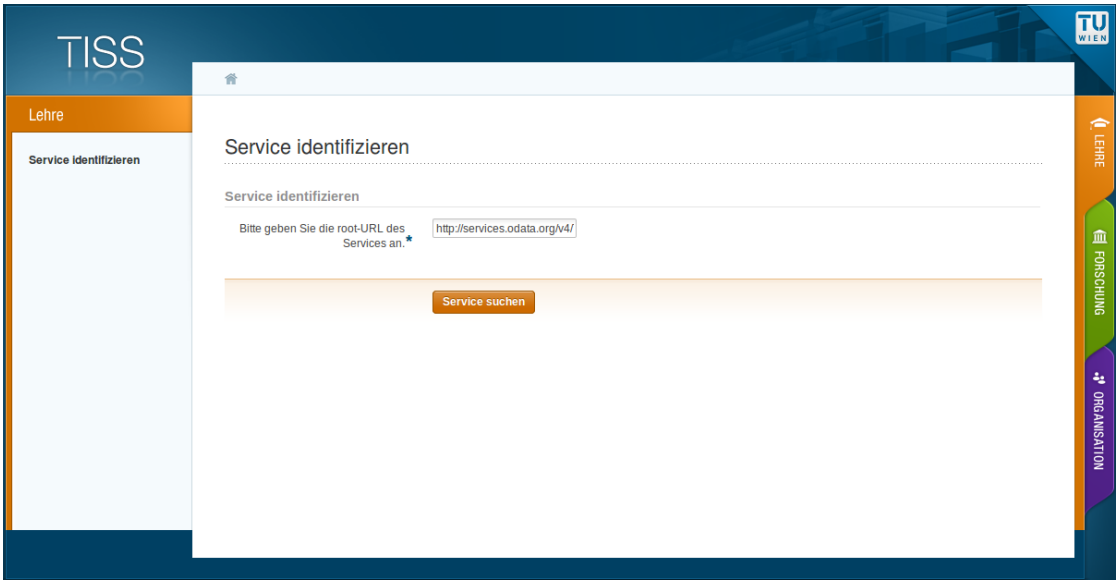

Figure 4.5: The page `servicediscovery.xhtml` of the web application.

If a valid root URL of an OData service has been provided, the service is being resolved and the user is forwarded to the second page of the application `serviceLoaded.xhtml`. This page consists of four tabs, as defined in Section 4.2.2. In the first tab (Figure 4.6), the entity types are described comprehensively. If the entity type is a sub-type of another entity type, then the parent entity type is displayed below the name of the entity type. For each entity type, all attributes are displayed in a paginated table together with the following information about each attribute:

- *Attribut Name*: The name of the attribute.
- *Verpflichtendes Attribut*: An indicator, whether the attribute has been declared as mandatory or not.
- *Attribut Typ*: The type of the attribute.
- *einzelnes Attribut / Collection*: Describes, whether the attribute consists of one single value or whether a collection of values might be returned.
- *Standard-Wert*: The standard value of the attribute, if such a value has been defined.
- *Maximale Länge*: The maximum length of the attribute, if it has been defined.
- *Zusätzliche Informationen*: If there are any instance annotations, an icon is displayed, where the annotation name as well as the annotation value are shown as mouseover text.

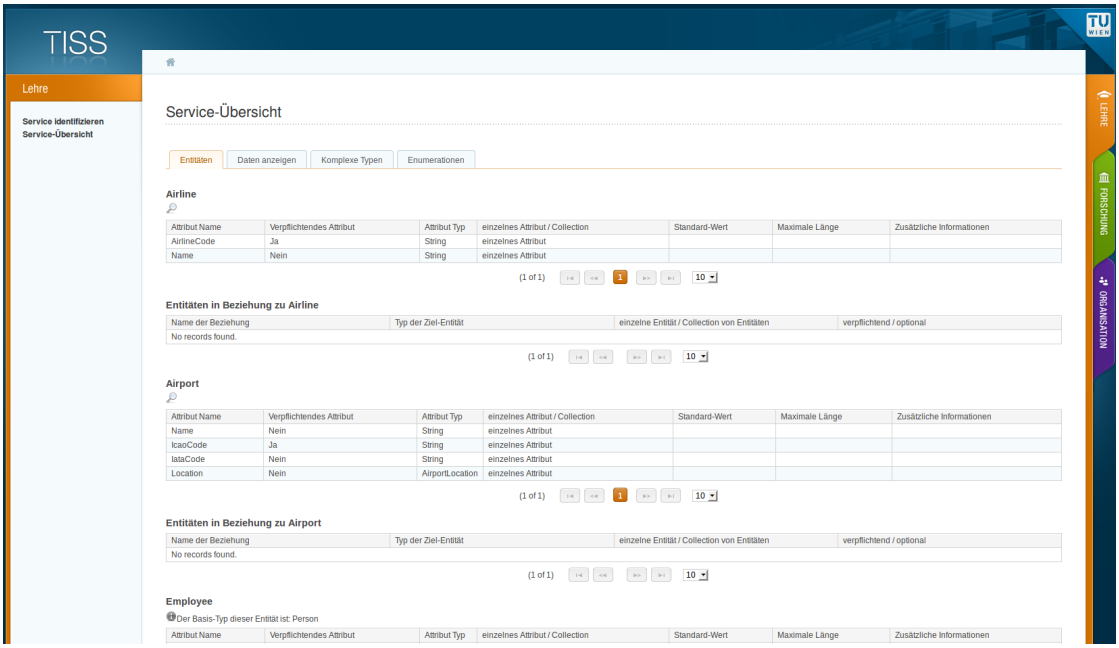

Figure 4.6: The first tab of page `serviceLoaded.xhtml` of the web application. Here the entity types and their relations to other entity types are described.

Below the attribute table, there is another table, where relations of an entity type are described:

- *Name der Beziehung*: The name of the relation.
- *Typ der Ziel-Entität*: The target entity type of the relation.
- *einzelne Entität / Collection von Entitäten*: Indicates, whether the target of the relation is a single entity or a collection of entitites.
- *verpflichtend / optional*: Indicates, whether this relation is mandatory or optional.

In the second tab (Figure 4.8), it is possible to explore the data that is exposed by the defined OData service. In the dropdown field *Entitäten*, those entity sets are offered, which can be accessed from the root URL of the service. After the user has made a selection in this field, the attributes of the underlying entity type are listed below twice. On the left side, the user can define filters per attribute of the entity type. In order to define a valid filter, a user needs to accomplish two tasks:

1. He/she must select a function from the dropdown field, which is located beneath the attribute name. Depending on the type of attribute, different functions are offered. In Table 4.1, the available functions are defined per attribute type. It needs

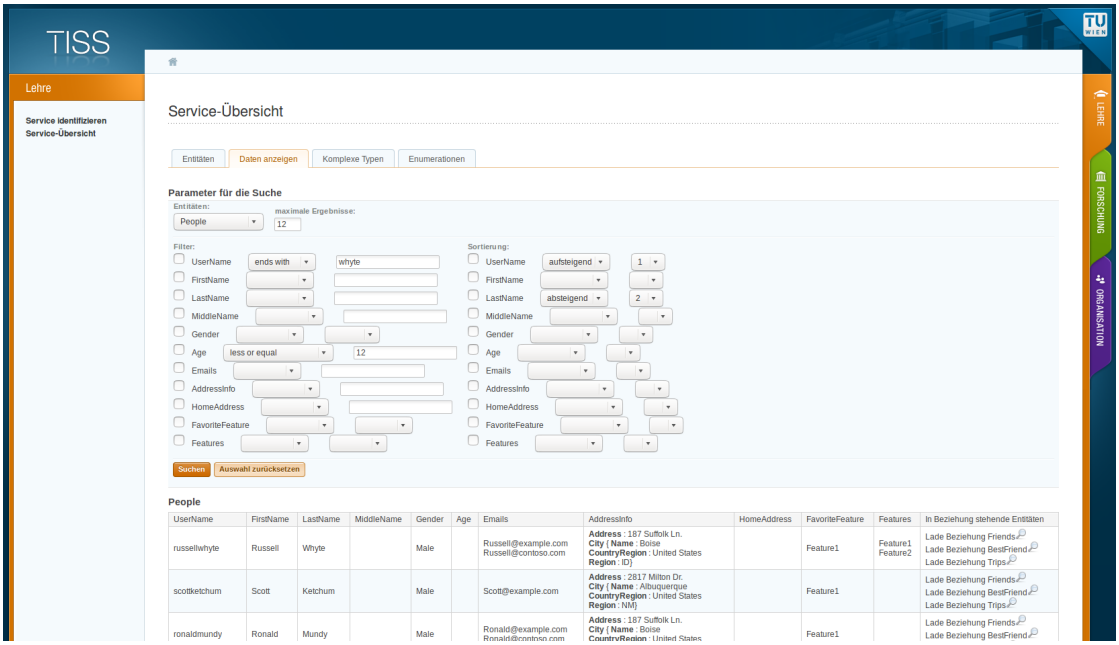

Figure 4.7: The second tab of page `serviceLoaded.xhtml` of the web application. In this tab it is possible to search for data that is exposed by the underlying OData service.

to be emphasized that the attribute types, as defined in the referenced table, are generalized. In the CSDL, several primitive types are defined (as already discussed in Section 2.3.2). There are, for example, five different primitive types, which denote an integer. Nevertheless, all of these integer types have been grouped in the mentioned table into one integer type.

2. He/she must enter a value, which shall be used to evaluate the function on the attribute. In case of enumeration values, instead of a textfield, a dropdown field is offered, where the user can select one of the available enumeration values.

If the user defines filters for multiple attributes, the logical *and* operator is used to connect the conditions. This means that all defined filters need to match for one entity in order to be included in the result set of the query. Technically, these filters are translated into an expression that is passed to the OData service via the `$filter` system query option. Apache Olingo provides appropriate classes, which allow to define the filters programmatically without needing to take care about the way the filter expression is serialized. The class `FilterArgFactory` allows to create arguments of the filter expression, while the class `FilterFactory` allows to create the filter expression itself (class `URIFilter`) by passing filter arguments to the available operations. The class `URIFilter` can be passed to the `URIBuilder` as argument.

The sorting of the result set can be defined by using the list of attributes next to the list of attributes that is used for defining the filters. In order to define a sorting, the

Attribute type	Available functions
Enumeration	`equal to`, `not equal to`
Integer	`equal to`, `not equal to`, `less or equal`, `less than`, `greater than or equal to`, `greater than`
DateTime	`equal to`, `not equal to`, `less or equal`, `less than`, `greater than or equal to`, `greater than`
Default (applied in case none of the above listed attribute types match)	`equal to`, `not equal to`, `ends with`, `starts with`, `contains`

Table 4.1: Available functions for defining a filter, depending on the attribute type.

user must select in the first dropdown field the order (ascending, descending) and in the second dropdown field the priority of the order of this attribute. This is analogous to an `ORDER BY` statement in SQL, where one can order the results by the values of multiple columns. In the raw OData request, the system query option `$orderby` is used to pass the desired ordering to the OData service. Apache Olingo allows the developer to pass the `$orderby` string to the `URIBuilder`, but it needs to be provided as a string. This means that I had to serialize the selected sorting options by writing custom code.

If the user wants to set a maximum number of entities that should be returned in the result set, he/she may enter a value in the text field *maximale Ergebnisse*. It is then passed as integer to the `URIBuilder`.

Once the request has been submitted, the returned entities are shown in a table below the section, where the search parameters are defined. Each attribute is shown in an own column in the table. In the last table of the column, the text `Lade Beziehung <name of relation>` (meaning *Load relation <name of relation>*) and a magnifier icon next to the text is shown. For each distinct relation of the underlying entity type, one such row is rendered. If a user clicks on such a magnifier button, a panel component, which overlays all other elements, is displayed. A table appears, which contains all attributes of all related entities (see Figure 4.8). In the source code, the URL of the selected entity is passed to the `URIBuilder` first. Afterwards, the name of the navigation property is passed, also as a string, to the `URIBuilder`, which then constructs a valid URL for requesting the related entities.

In the third tab, the complex types, that have been returned in the metadata document, are displayed. For each complex type, one table is shown, where all attributes of the complex type are listed. The table is equal to the table that is used for displaying the attributes of the entity types in the first tab.

In the fourth tab, the user can view all enumerations that the metadata document comprises (see Figure 4.9). For each enumeration, a table with all possible values (the

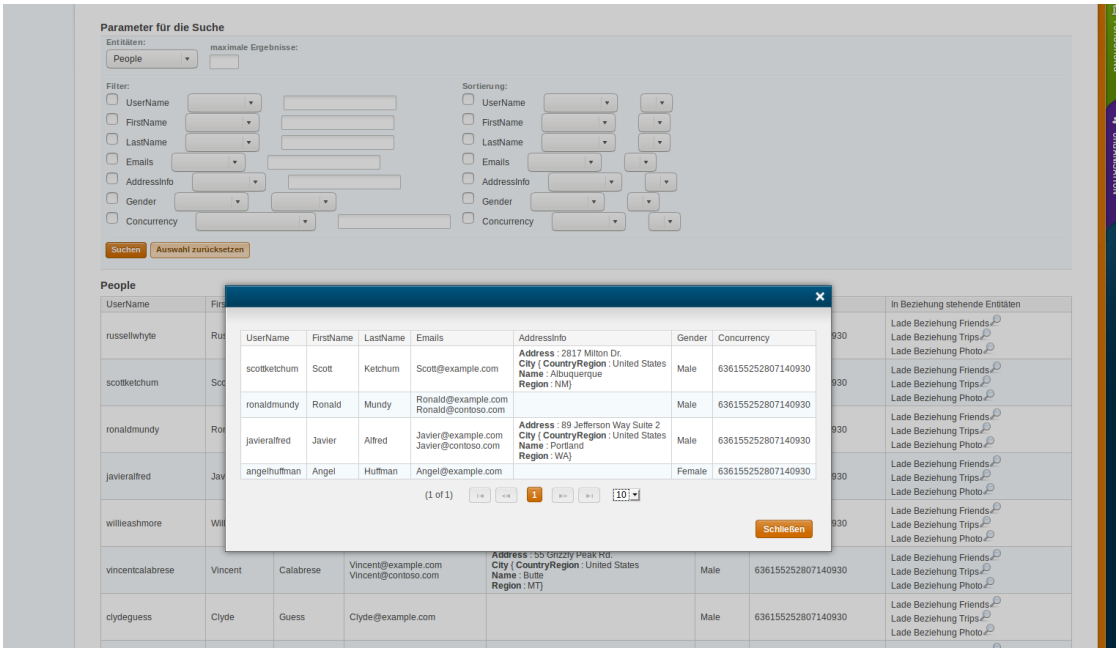

Figure 4.8: An example of the way related entities are displayed. In this particular case, all related *Friends* of the person with UserName "russellwhyte" are displayed.

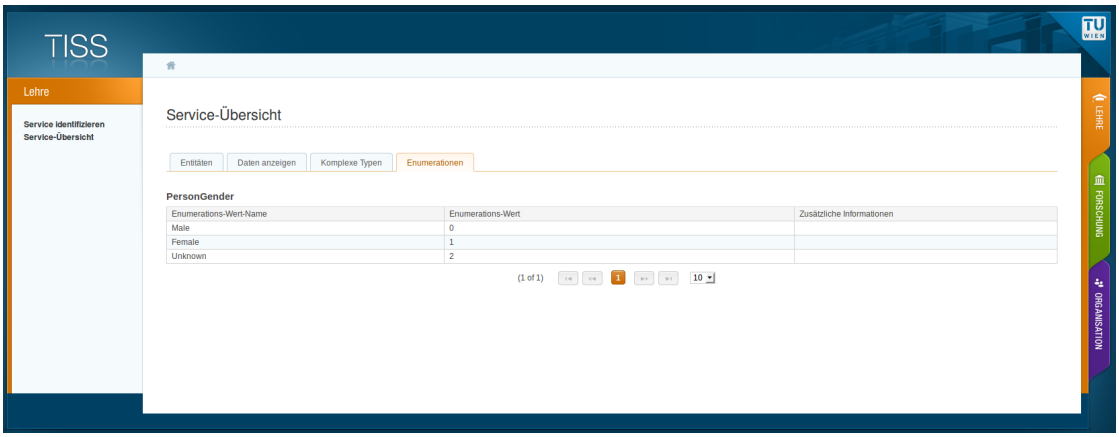

Figure 4.9: All enumerations are listed in the fourth tab of page `serviceLoaded.xhtml`.

name of the value and the real value as integer) and possibly existing annotations of the enumeration values are shown.

There are also some limitations of the web application, which need to be mentioned:

- Only entity sets, that are present in the service document, are taken into considera-

tion and thus can be explored. Singletons and functions, that might be offered by the service, are ignored when resolving the OData service.

- By definition, the metadata document might contain multiple schemas. The prototype always takes the first schema, that is returned in the metadata document and does not take into consideration any additionally provided schemas.
- The web application expects the OData service to expose a metadata document despite the fact that the metadata document is not mandatory at the Minimal Conformance Level of OData version 4. It becomes mandatory at the Advanced Conformance Level.
- The prototype does not allow to define filters or sortings for attributes of complex types. Complex types are displayed the same way as all primitive type attributes in the section, where the search parameters can be defined.

After discussing the implementation of both prototypes extensively, I will shortly also describe the implementation of the proof-of-concept application before moving to the evaluation chapter.

4.3 Implementation of the proof-of-concept application

As already mentioned in Section 3.3, the development of this application is not directly relevant for the topic of this thesis. Thus, the implementation will not be described in detail as it is done for the other applications. Instead, the results will be summarized in the next paragraphs.

The search for research infrastructure has been realized in a similar way as it has been done for the generic OData client application. By using `$filter`, it is possible to search for research infrastrcuture. A visitor can enter a term he wants to search for in a textfield. The application constructs a filter expression by employing the function `contains` on a set of attributes of the research infrastructure. All `contains` statements are connected with the logical *or* operator.

In order to view details of an infrastructure, also images of it shall be retrieved from the backend. This made it necessary to extend the read functionality of the OData REST service by adding so-called media entity requests, which allows the application to load the images by using the OData RESTful web service. Due to the fact that the record and edit infrastructure use cases will need to be capable of storing images for a certain research infrastructure, also the writing media entity requests needed to be implemented.

The proof-of-concept application takes advantage of the implemented system query option `$expand` at the retrieval of infrastructures to improve the efficiency. In order to fetch all information about a research infrastructure that should be displayed, it would be necessary to fetch the infrastructure itself and all related entities separately. This would result in one backend call for the infrastructure itself plus one call per related entity type.

Instead, the application just submits one request and uses the mentioned parameter to request the related entities together with the infrastructure itself. The images themselves needed to be fetched separately, because images have a binary representation which cannot be represented inline.

The record and edit infrastructure use cases required to enable writing functionality for the entities in the REST service, which are stored in the dynamic data tables (they have been discussed in Section 4.1.2). As there is a lot of data to be entered at the record and edit research infrastructure use cases, they are realized on the frontend by using wizards, which lead the user step-by-step through the creation process. On the backend side, the implementation of the create research infrastructure use case has been done in a way that it is not possible to create an infrastructure that is automatically approved. Furthermore, another OData specific feature called *deep insert* has been employed at the creation of research infrastructure. It uses the concept of representing related entities inline together with the entity that is addressed by the URL. The proof-of-concept application sends the infrastructure entity together with all other related entities in one request. The OData service creates the infrastructure and all related entities at once (plus one request per image that should be uploaded).

Listing the infrastructure that is ready for approval is realized by using system query option `$filter` to create the appropriate filter. Approving / rejecting research infrastructures was realized by editing the attribute `status` of it. The application itself needed to display all of the information, that has been entered for an infrastructure, on one page. Thus, all the information has been summarized on one page consisting of multiple tabs. The supervisor can either approve or reject the displayed infrastructure by clicking on the corresponding button, which triggers the submission of the request to the backend.

Lastly, a rudimentary login function has been implemented. This function has been implemented in order to show what the application looks like from the perspective of each of the three actor types. For the actors *Researcher* and *Supervisor*, the credentials of two users have been hard-coded into the proof-of-concept application. A *Visitor* can log in and, depending on his new role (*Researcher* or *Supervisor*), see additional menu items and conduct additional use cases.

CHAPTER 5

Evaluation

The evaluation of this thesis consists of two steps:

1. The implementation of the generic OData client application is evaluated by testing it against OData services.
2. A questionnaire about OData is given to domain experts to be filled out.

These settings of the evaluation steps and their results will be presented in the next two sections.

5.1 Testing against OData services

5.1.1 Setting

In order to test the prototype, it was necessary to define first the OData services, which it should be tested against. Beside the implemented OData RESTful web service prototype, there are other freely available OData services which have been used for testing the prototype:

- `http://services.odata.org/v4/TripPinServiceRW/`
- `http://services.odata.org/V4/OData/OData.svc/`

In order to be able to execute the test cases, first the developed OData RESTful web service and the generic client application have been deployed to a Tomcat 7 on a cloud server. For each of the three OData services, which have been selected for testing the prototype, the same tests have been conducted. The test cases are described in the Tables 5.1 and 5.2. These test cases have been performed manually and sequentially.

ID	Preconditions	Instructions	Expected results
TC1	–	Open the page `servicediscovery.xhtml`. Enter the root URL of the OData service that should be used into the corresponding text field on this page. Click button *Service suchen*.	The application shows the first tab *Entitäten* of the page `serviceLoaded.xhtml`. At least one entity type is shown in the content of this tab.
TC2	TC1 has been executed successfully.	Load the metadata document of the OData service (e.g. in the browser). Compare the entity types listed in the metadata document and those that are shown in the corresponding tab of the application.	The information that is shown in the tab about the entity types, the attributes of the entity types and the relationships of the entity types is the same in the metadata document and in the application. There is no entity type that is listed in the metadata document, but not shown in the application and vice-versa.
TC3	TC1 has been executed successfully.	Load the metadata document of the OData service (e.g. in the browser). Open the tab *Komplexe Typen* of the page `serviceLoaded.xhtml`. Compare the complex types listed in the metadata document and those that are shown in the corresponding tab of the application.	The information that is shown in the tab about the complex types and the attributes of the complex types is the same in the metadata document and in the application. There is no complex type that is listed in the metadata document, but not shown in the application and vice-versa.
TC4	TC1 has been executed successfully.	Load the metadata document of the OData service (e.g. in the browser). Open the tab *Enumerationen* of the page `serviceLoaded.xhtml`. Compare the enumeration types listed in the metadata document and those that are shown in the corresponding tab of the application.	The information that is shown in the tab about the enumeration types is the same in the metadata document and in the application. There is no enumeration type that is listed in the metadata document, but not shown in the application and vice-versa.
TC5	TC1 has been executed successfully.	Load the service document of the OData service (e.g. in the browser). Open the tab *Daten anzeigen* of the page `serviceLoaded.xhtml`. Compare the entity sets offered in the dropdown field *Entitäten* with the entity sets listed in the service document.	The entity sets offered in the dropdown field are equal to the entity sets listed in the service document.

Table 5.1: The description of the test cases that have been conducted (part 1).

ID	Preconditions	Instructions	Expected results
TC6	TC1 has been executed successfully.	Open the tab *Daten anzeigen* of the page `serviceLoaded.xhtml`. Select any of the offered entity sets.	The attributes of the underlying entity type are shown below twice (for defining a filter as well as a sorting).
TC7	TC6 has been executed successfully.	Click button *Suchen*.	In the table below, all retrieved entities are shown.
TC8	TC6 has been executed successfully.	Define a sorting. Click button *Suchen*.	In the table below, the same entities as in TC7 are shown, but in the defined order.
TC9	TC6 has been executed successfully.	Define a filter. Click button *Suchen*. Repeat this test case for attributes of the mentioned attribute types in Table 4.1.	In the table below, those entities that pass the defined filter are shown.
TC10	TC6 has been executed successfully.	Define a filter and a sorting. Click button *Suchen*.	In the table below, those entities that pass the defined filter are shown in the defined order.
TC11	TC6 has been executed successfully.	Define a maximum number of entities to be returned in the field *maximale Ergebnisse*, that is lower than the number of returned entities when executing TC6. Click button *Suchen*.	In the table below, the number of entities does not exceed the maximum number of entities that has been defined.
TC12	TC6 has been executed successfully. An entity set, where the underlying entity type has at least one relationship, is selected.	Click button *Suchen*.	In the table below, all retrieved entities are shown. For each entity, in the column *In Beziehung stehende Entitäten* it is offered to load all related entites. It is offered to load each distinct relation of the underlying entity type.
TC13	TC12 has been executed successfully.	Click the magnifier button in column *In Beziehung stehende Entitäten* for an offered relation of an entity.	All related entities of the entity are shown in a table.

Table 5.2: The description of the test cases that have been conducted (part 2).

5.1.2 Results

It was possible to execute all test cases successfully by using each of the mentioned OData services. It must be highlighted, that the results of those test cases, which include retrieving concrete entities from the configured OData service (TC7 - TC13), have been additionally verified by executing the same raw request via a REST client (Postman). This way it is ensured that all displayed data are completely equal to the data that are returned by the service itself.

5.2 Questionnaire with experts

5.2.1 Setting

In the first step, the questionnaire itself has been designed. The questionnaire starts with demographic questions, which also include the question about the experience in developing REST services in years. This is rather important in order to ensure that the participants meet the requirements of a domain expert, which have been mentioned in Section 1.2. Then, the experience of the participants with OData is enquired, followed by questions about the advantages and shortcoming of OData that they can identify. Furthermore, it is asked, whether the OData standard supports/enables building generic clients for accessing and querying data published via such interfaces. Next, the participants are asked to name differences between the way they have developed services up to now and the way OData proposes to implement them. Lastly, their opinion on whether the OData protocol will be widely adapted and whether they would consider building a REST-API conforming with OData is asked. The questions are a mixture of closed questions and open questions. For example, when asking for advantages of the OData protocol, first a closed question with multiple possible answers is given. This closed question is followed by an open question, which asks for additional advantages, that have not been mentioned previously.

As the focus of the questionnaire lies on OData, it was necessary to provide an introduction to OData in order to make the participants familiar with OData. For this purpose an *OData fact sheet* has been written, where the relevant data have been summarized.

The potential participants have been selected beforehand and contacted via email. The email contained information on the topic the thesis is dealing with and instructions on how the participants should proceed (read the OData fact sheet, test the generic client application, answer the questionnaire). The links to the developed prototypes have been included as well as the link to the questionnaire and the link to the OData fact sheet, which was stored in my Dropbox. The first page of the questionnaire also contained introductory information and the instructions, that the participants should follow before answering the questionnaire. Moreover, the recipients of the email are asked to forward the email to other software developers, which might be interested in the topic and in participating in the evaluation.

After presenting the setting of the questionnaire, the results will be presented in the following subsection.

5.2.2 Results

First, the demographic information, that has been collected, will be presented. This information will be followed by concrete questions related to OData that were asked and the corresponding answers of the participants. The interpretation of the results will be done in the conclusion section.

In sum, there were eight participants, that have started to answer the questionnaire. Only five of them have completed the questionnaire, which is the reason why only their answers will be taken into consideration in the results of this survey. The participants are all male and between 26 and 43 years old with an average age of 31 years. Their experience in software development lies between one and 20 years, while the range of their experience in the development of REST services is from one to five years. One of the participants has only one year of experience in developing REST services, while another has two years. This means, that they are no domain experts according to the definition of a domain expert in Section 1.2. Nevertheless, their answers will still be considered as a compromise between scientific demand and the availability of suitable participants which are willing to participate.

Only one of the five participants answered on the question *"Did you hear about OData before you were contacted to take part in this master thesis evaluation?"* with *yes*. The answers on the question, what advantages of OData the participants can identify, are depicted in Figure 5.1. Obviously, the service document and the metadata document are the features of OData that are perceived as the biggest advantages. This question was followed by an open question, where the participants were asked to name any additional advantages, that were not offered as answers in the closed question. One of the participants wrote *machine readable* as answer on this question. Probably, he referred to the service document and/or to the metadata document.

There were symmetric questions about the shortcomings of OData. The results of the closed question on the disadvantages of OData are illustrated in Figure 5.2. There were no answers on the corresponding open question on this topic as well as on the subsequent open question *"What are in your opinion possible improvements of OData?"*

40 percent of the participants answered with *yes* on the question, whether the OData standard supports/enables building generic clients for accessing and querying data published via such interfaces in their opinion. 60 percent of them ticked the answer *Only to some extent*. The participants were also asked to explain their answer on this question. One participant, who answered previously with *yes*, wrote the following explanation:

> *"For something like Python, JavaScript or even Java (with the use of ReflectionAPI) you can build up the necessary data holder (eg. classes or*

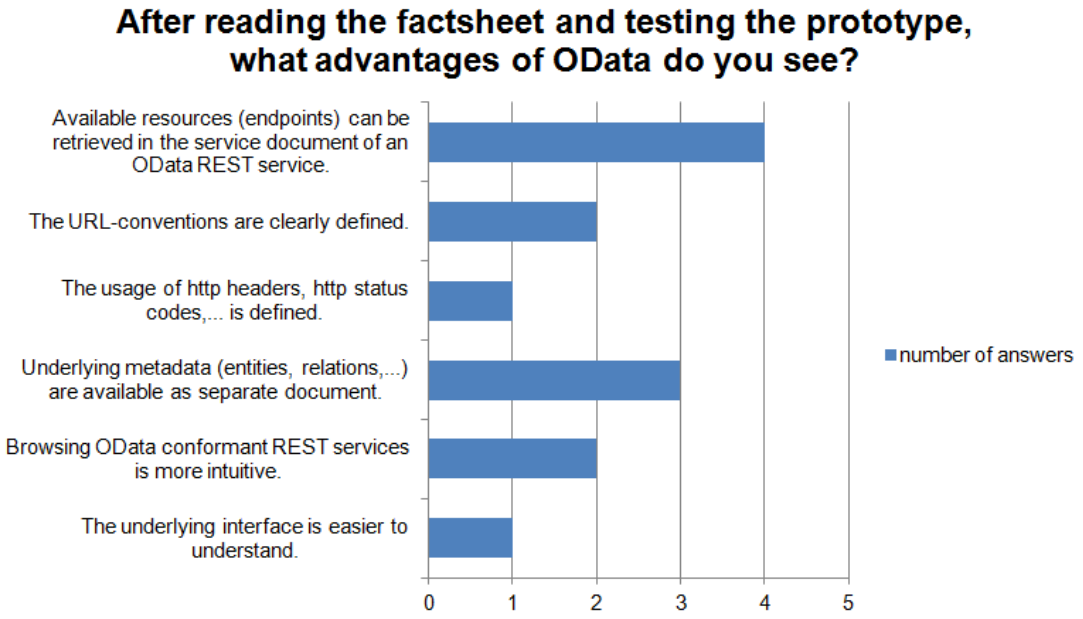

Figure 5.1: These are the results of the closed question which is related to the advantages of OData.

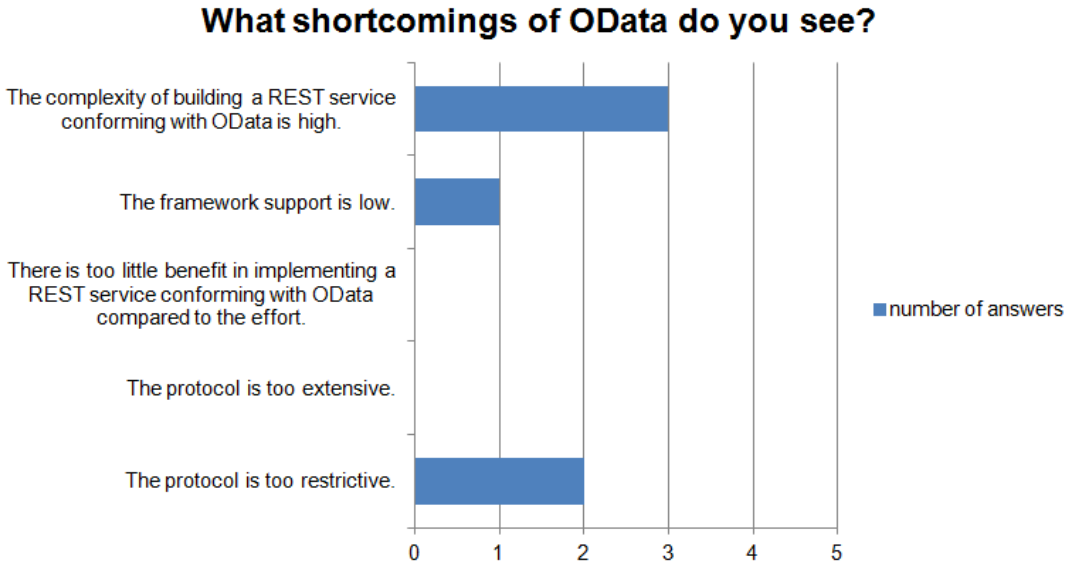

Figure 5.2: These are the results of the closed question which is related to the shortcomings of OData.

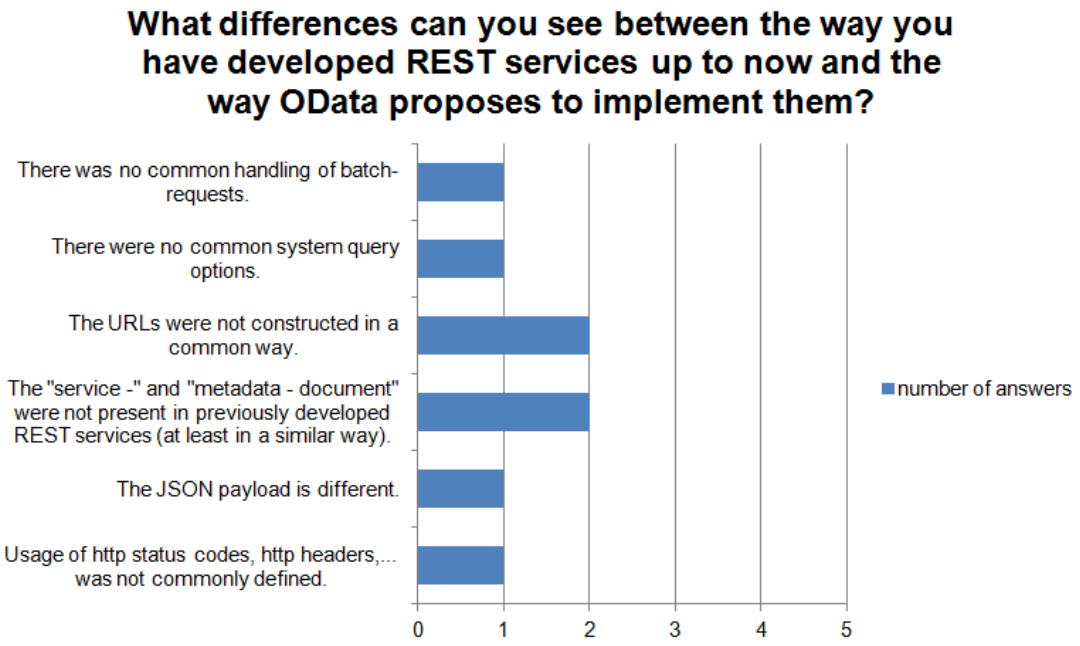

Figure 5.3: These are the results of the closed question which is related to the differences between the way the participants have developed REST services up to now and the way OData proposes to implement them.

> *dictionaries) from the queried metadata and use them. For something like C++ you one could still rely on HashMaps or similar."*

The results of the closed question dealing with the differences of the way the participants have developed REST services up to now and the way OData proposes to implement them are presented in Figure 5.3. The corresponding open question has been answered one time: *"The implementations were by far shorter and much less boilerplated then the examples of the various frameworks I found in a brief search on the internet."*

Four out of five participants stated that the OData protocol will not be widely adapted in their opinion. Three of them explained the reason why they answered with *no*:

> *"Once you have it up and running I think it is great to have such an OData service. Yet I think it is to much effort, especially for bigger REST interfaces delivering more then just a handful entities. In my projects providing REST interfaces the ability to fetch entities in such detail (e.g. with SQL-like queries) has not been an issue - there just wasn't any need for it. I didn't need the flexibility offered by OData - a much simpler, well defined REST interface did the trick for me. The underlying languages (i.e. Python and Java) provide nice frameworks for defining a REST interface with a bunch of annotations on*

the DataAccessObjects or the BusinessLogic-Services respectively - no further boilerplate needed. The approach relying on already present model information (like JPA) seems nice (less boilerplate = less code = less effort to create and maintain = great), yet allowing full CRUD on a JPA model is definitely something you don't want. So - in the - you end up defining the interface by hand, so no gain. So, IMHO the OData protocol is great for DataStores providing just Data for someone else to use it (e.g. OpenGovernment). But if you want to provide a specific service or you plan on writing some WebApp with a REST API I think the extra effort is just not worth the while. The Metadata stuff is nice, but some simpler approaches like WADL do that trick as well with close to no extra effort, since they can be autogenerated."

"I think it will stay niche tech - good only for some cases."

"It looks as if the protocol was too slightly too restricted. Developers got really fond with REST due to the fact that you could do virtually anything with it, there are some practices that are commonly adopted and are considered good, but technically there aren't many (if any) restrictions. At a first glance it seems that it's not really the case for OData. It might be successful in companies that do not have much experience with building REST services, but I personally do not see any particular reason to drop my current technology stack in favor of OData."

In contrast to the previous answers, 80 percent of the participants said that they would consider building a RESTful-API that conforms with OData depending on the use case/requirements, while the remaining 20 percent would definitely use it. Four of the participants also explained their answers:

"Im curious to see it in action in order to decide if it could be a smart solution to ease development of REST services. It seems to be worth a try."

"If and only if I am just a data provider (i.e. I do NOT consume the data I provide myself) it might be an option, since it gives my DataConsumers more flexibility. If I am provider and consumer at the same time - regardless of whether I am the only consumer - OData just seems overcomplex."

"I don't see a reason to use this in general case where REST is enough."

"Wouldn't bother if there were no restrictions and logic was quite simple, could consider to use it with more tricky services."

These were the results of the questionnaire that has been filled out. In the next section, my implementation experience will be illustrated before coming to the conclusions and the outlook on future work.

5.3 Implementation experience

First, I will share my implementation experience of the server side. After getting to know the OData library that I have decided to use, I started to implement the first read requests. As already described in Section 4.1.3, I have experienced some difficulties at the integration of certain libraries. On the other hand, Apache Olingo handles many tasks on its own once it has been integrated correctly. Nevertheless, extra effort had to be put into the implementation of the various system query options as well as into the navigation between entities. The `$filter` operation was the toughest feature to implement. In contrast, the writing functionality, which has been required by the proof-of-concept application, was rather straightforward to implement.

The client library of Apache Olingo was also easy to use. It provides the facilities for building OData requests without needing to take care about how the URL will look like in the end. This was especially helpful at the definition of filter expressions on the client side. The *deep insert* functionality of OData simplified the creation process of an infrastructure from the client's perspective, as all related entities could be sent in one request.

The main difference that I have encountered at the retrieval of entities via a REST client, was the fact that the identifier of an entity is provided in brackets instead of being a sub-resource of the entity (e.g. `GET /accounts(1)/` instead of `GET /accounts/1/`). Another difference was related to the writing functionality of OData. Another OData feature, that was different to the approaches that I have seen before, was the feature of relating two entities to each other. In order to relate an existing entity to another, you need to send a creation or modification request for one entity and provide the identifier(s) of the already existing entity(ies) in the body of the request. They must be provided in an attribute, whose name is built by concatenating the name of the relation (as specified in the metadata document) and `odata.bind`. This fact is, of course, hidden from a developer who just uses the libraries without trying to submit the raw request himself/herself.

In general, there are some differences between the OData services and those REST services, that I have seen before and that were not OData conformant. I personally do not think that it is more difficult to work with OData services, especially once you get to know them. One just needs to define the scope of his application and thus the OData features that are required in the context.

CHAPTER 6

Conclusions and future work

6.1 Conclusions

The implementation of the OData generic client application prototype was one step to show the way OData REST interfaces enable building generic clients for accessing and querying them. The successful implementation of the protoype and its successful evaluation by testing it against OData services (also against services that have not been developed as part of this thesis) showed that it is possible to build such a generic client, which is capable of:

- presenting the data model of the OData service
- retrieving concrete data from the OData service
- submitting requests to the OData service, which include filtering and sorting criteria the result set has to meet.

Furthermore, there was no domain expert who has answered with *no* on the corresponding question in the questionnaire, which confirms the result of the tests of the generic client against other OData services.

Judging the complexity of accessing and modifying data that is offered by an OData-conformant REST service, the conclusion can be drawn that the complexity is not higher compared to RESTful web services which are not conforming with OData. The uniform interface is, as already discussed, mostly induced by the correct usage of HTTP and its methods. The OData specification explicitly defines the usage of HTTP methods, headers and other components of HTTP. Possibly, accessing OData-conformant REST services is even easier than accessing other REST APIs, as other APIs might not use the HTTP capabilities correctly. Moreover, each API might interpret and therefore use

them differently, resulting in many REST services which are not compatible with each other. The lack of standardisation in the REST domain, which has been discussed in the theoretical part of this thesis and identified as one of the shortcomings of REST, does not contribute to a uniform interface when looking across different REST services.

The complexity of implementing an OData-conformant REST service, though, might be higher compared to non-OData-conformant REST services. The most frequently selected answer to the question which asks for the shortcomings of OData, is the high complexity of building a REST service conforming with OData. One of the experts wrote that it would be too much effort to implement such a service, especially for bigger REST interfaces. Still, none of the participants of the questionnaire stated, that he/she would not consider/suggest building a REST API conforming with OData. Most of them would tie the decision, whether to use OData, to the use cases/requirements the service must cover/meet. Hence the question arises, whether certain requirements might be easier met by implementing an OData-conformant REST service than by implementing it in another (custom) way. Especially the use cases that have been covered by the generic client application (treating the OData services as data stores and discovering its data), might be such a case. One expert also suggested, that it might be beneficial to use OData in such a case. It must be highlighted, that the overall implementation complexity can be reduced by the fact that there are three conformance levels of OData and that it is suggested to implement only those features of OData, which are needed in the certain case.

The implementation experience of the OData service leads to the following conclusions, which are related to the complexity of building an OData-conformant REST service:

- A lot of tasks are handled by using an OData library at implementation. On the one hand, this lowers the complexity of implementing such a service significantly in terms of making the service OData-compliant. On the other hand, the usage of an OData library might cause other difficulties. The usage of Spring libraries together with Apache Olingo lead to additional effort that needed to be spent on making them compatible with each other. In sum, the benefits of using an OData library outweigh the additional effort, that has been caused by using it.
- The complexity of implementation heavily depends on the features that need to be supported by the OData service.

The offer of Open Data in Austria could benefit from a shift of data producers to offering Open Data via OData REST interfaces, as the presence of an OData RESTful web service would solve all of the mentioned problems at the beginning of the thesis:

- The data model would be published in a standardized way.
- Querying the data would be possible according to the capabilities of OData services.

- There would be no need for downloading the Open Data sets in order to use them inside an application, as they would be available and accessible all the time via the Internet in a machine-readable way.
- If the OData service exposes data from the direct data source, its exposed data would always be up-to-date, eliminating the delay in time between the creation of the data sets and its publication.

By integrating the OData generic client application with the national Open Data portal, it would be possible to make the data exposed by OData REST interfaces discoverable even for those visitors of the Open Data portal who are not familiar with REST and OData and thus are not able to communicate directly with the OData service and to interpret the returned data.

The usage of the implemented OData service by the proof-of-concept application is additional evidence for the statement that data sets, which are exposed in such a way, can be used by applications without the need to download the data set in advance. Furthermore, the application will always receive the latest data this way.

Finally, the following conclusions can be drawn:

- OData enables the building of generic client applications.
- The Open Data offer could benefit from many Open Data producers offering their data via OData-conformant REST services. In order to gain all the benefits, the Open Data portal would need to be capable of the same features that the generic OData client application is capable of.
- The complexity of implementing such a service, as judged by the domain experts, is higher compared to non-OData-conformant REST services. Thus, they would mostly consider to use it only under certain circumstances, although there are several verified advantages of such services. It can be confirmed by the implementation experience that, without using an OData framework/library in the implementation, the complexity of implementing it would definitely be higher.

6.2 Future work

The future work that could be done based on this thesis, can be divided into two categories. The first category comprises tasks, which are oriented towards eliminating the limitations of the generic OData client application. These limitations have been listed in Section 4.2.3. The second category deals with further enhancements of the generic OData client application. These are some examples for such enhancements:

- Present the data model in different ways. The data model itself could, for example, additionally be presented as a diagram.

- Enable the communication with services, which require some sort of authentication. For this purpose, it would be necessary to configure the authentication mode and its credentials additionally to the root URL of the OData service. As the OData specification does not propose any concrete authentication mode, it would be necessary to select authentication modes, which shall be supported by the client application (e.g. Basic Authentication).

Apart from these two categories, effort could be spent on trying to integrate the generic client application into CKAN. As OData might become an international standard, there might also be willingness on CKAN side to enable additional features for OData services on their platform. Based on the gained knowledge about CKAN, I see two general steps required for integrating the application into CKAN:

1. CKAN needs to allow adding an OData service as a separate resource type, where the data can be retrieved from.

2. In case there is an OData resource defined for an Open Data set, show the generic OData client application (or the integrated part of it) and let the visitor explore the underlying data.

6.3 Outlook

This section will provide a short outlook on the perspectives of OData and the approach of offering Open Data via OData version 4 conformant REST services.

The answer to the question, whether OData will be widely used, heavily depends on the effects of the publication of OData as an ISO international standard. There might be legal entities or public sector entities, whose IT systems need to comply with ISO standards to a certain extent. This has not been researched in this thesis, thus no verified statement can be given on this topic. But if this is the case, then these entities might consider making use of OData in the future development of their systems. The results of the evaluation with domain experts showed that OData seems not to be widely known among developers in the RESTful web service domain. Also, the participants of the evaluation mostly stated, that they would use OData only in case there are certain requirements that can be met easier by using OData. The fact that building an OData service is perceived as a highly complex undertaking, leads to the conclusion that it will not be considered as a first choice by a RESTful web service developer. Thus, it seems that OData will not be widely used by developers that are not required to comply with ISO standards.

The perspective of the approach, where Open Data producers offer their data via OData version 4 conformant REST services, depends also on the extent to which the public sector entites will make use of OData. If these entities would make use of OData for more reasons than just exposing their Open Data (as, for example, the proof-of-concept

application, which makes use of both reading and writing capabilities of OData), then there is a higher chance that they would register their OData service as a resource on the national Open Data portal. On the other hand, the integration of the generic client application into CKAN might cause such institutions to think about using OData. There might be a mutual relationship between these two issues, where the realization of one would push the realization of the other.

List of Figures

List of Tables

Bibliography

[Ben15] Brian Benz. Oasis has submitted OData v4 and OData JSON Format v4 to ISO/IEC JTC 1 for approval as an International Standard. https://msopentech.com/blog/2015/04/21/oasis-has-submitted-odata-v4-and-odata-json-format-v4-to-isoiec-jtc-1-for-approval-as-an-international-standard/, April 21 2015. Visited: October 7, 2016.

[BK11] Florian Bauer and Martin Kaltenböck. Linked open data: The essentials. *Edition mono/monochrom, Vienna*, 2011.

[BLFM] Tim Berners-Lee, Roy Fielding, and Larry Masinter. Rfc 2396: Uniform resource identifiers (uri): Generic syntax, august 1998. *Status: Draft Standard*.

[Buna] Magistrat der Stadt Wien Bundeskanzleramt. Daten/Dokumente hinzufügen. https://www.data.gv.at/suche/daten-hinzufuegen/. Visited: October 16, 2016.

[Bunb] Magistrat der Stadt Wien Bundeskanzleramt. Linked data. https://www.data.gv.at/linked-data/. Visited: October 16, 2016.

[Bunc] Magistrat der Stadt Wien Bundeskanzleramt. Zielsetzung data.gv.at. https://www.data.gv.at/infos/zielsetzung-data-gv-at/. Visited: October 16, 2016.

[Chi13] Simon Chignard. A brief history of Open Data. *Paris Tech Review*, 29, 2013.

[Ckaa] Ckan. Api guide. http://docs.ckan.org/en/latest/api/index.html. Visited: October 16, 2016.

[Ckab] Ckan. DataStore extension. http://docs.ckan.org/en/latest/maintaining/datastore.html. Visited: October 16, 2016.

[Ckac] Ckan. User guide. http://docs.ckan.org/en/latest/user-guide.html. Visited: October 16, 2016.

[CNV16] Wendy Carrara, Margriet Nieuwenhuis, and Heleen Vollers. Open Data Maturity in Europe 2016. Technical report, European Commission, Directorate-General of Communications Networks, Content & Technology, 2016.

[Cor] Microsoft Corporation. [MS-ODATA]: Open Data Protocol (OData). https://msdn.microsoft.com/en-us/library/dd541188.aspx. Visited: October 30, 2016.

[Defa] Open Definition. Open Definition 2.1. http://opendefinition.org/od/2.1/en/. Visited: October 14, 2016.

[Defb] Open Definition. The Open Definition. http://opendefinition.org/. Visited: October 14, 2016.

[EHL+13] Gregor Eibl, Johann Höchtl, Brigitte Lutz, Peter Parycek, Stefan Pawel, and Harald Pirker. Rahmenbedingungen für Open Government Data Plattformen. Technical report, 2013.

[Eur16] European Data Portal. *Factsheet United Kingdom*, September 13 2016.

[Fie00] Roy Thomas Fielding. *Architectural styles and the design of network-based software architectures*. PhD thesis, University of California, Irvine, 2000.

[Foua] The Apache Software Foundation. Interface ODataHttpHander. https://olingo.apache.org/javadoc/odata4/org/apache/olingo/server/api/ODataHttpHandler.html. Visited: November 19, 2016.

[Foub] World Wide Web Foundation. The Open Data Barometer. http://opendatabarometer.org/barometer/. Visited: October 23, 2016.

[Fouc] World Wide Web Foundation. Ranking. http://opendatabarometer.org/data-explorer/?_year=2015&indicator=ODB&lang=en. Visited: October 23, 2016.

[Foud] World Wide Web Foundation. Research Method. http://opendatabarometer.org/3rdEdition/methodology/. Visited: October 23, 2016.

[Fut13] Futurezone. Data.gv.at mit neuen Funktionen. https://futurezone.at/netzpolitik/data-gv-at-mit-neuen-funktionen/24.591.074, January 16 2013. Visited: October 16, 2016.

[GHL+14] Francoise Genova, Hilary Hanahoe, Leif Laaksonen, Carlos Morais-Pires, Peter Wittenburg, and John Wood. The data harvest: How sharing research data can yield knowledge, jobs and growth. *RDA Europe*, 2014.

[GINT14] Peter Leo Gorski, Luigi Lo Iacono, Hoai Viet Nguyen, and Daniel Behnam Torkian. SOA-Readiness of REST. In *European Conference on Service-Oriented and Cloud Computing*, pages 81–92. Springer, 2014.

[Gova] UK Government. About. `https://data.gov.uk/about`. Visited: October 17, 2016.

[Govb] UK Government. FAQ. `https://data.gov.uk/faq`. Visited: October 17, 2016.

[HB04] Hugo Haas and Allen Brown. Web Services Glossary - W3C Working Group Note. Technical report, World Wide Web Consortium - W3C, February 11 2004.

[HFdTC] Wilson A. Higashino, M. Beatriz Felgar de Toledo, and Miriam A. M. Capretz. REST and Resource-Oriented Architecture. In *Proc. of International Symposium on Services Science (ISSS 2009)*.

[HPB16] Ralf Handl, Michael Pizzo, and Mark Biamonte. Odata json format version 4.0 plus errata 03. `http://docs.oasis-open.org/odata/odata-json-format/v4.0/errata03/os/odata-json-format-v4.0-errata03-os-complete.html`, June 02 2016. Visited: October 30, 2016.

[HVdB11] Noor Huijboom and Tijs Van den Broek. Open data: an international comparison of strategies. *European journal of ePractice*, 12(1):4–16, 2011.

[ISOa] ISO. ISO glossary of terms. `http://www.iso.org/iso/home/faqs/faqs_abbreviations.htm`. Visited: November 05, 2016.

[ISOb] ISO. ISO/IEC DIS 20802-1. `http://www.iso.org/iso/home/store/catalogue_tc/catalogue_detail.htm?csnumber=69208`. Visited: November 05, 2016.

[ISOc] ISO. ISO/IEC DIS 20802-2. `http://www.iso.org/iso/home/store/catalogue_tc/catalogue_detail.htm?csnumber=69209`. Visited: November 05, 2016.

[JCZ12] Marijn Janssen, Yannis Charalabidis, and Anneke Zuiderwijk. Benefits, Adoption Barriers and Myths of Open Data and Open Government. *Information Systems Management*, 29(4):258–268, 2012.

[KA16] Rashmi Krishnamurthy and Yukika Awazu. Liberating data for public value: The case of data.gov. *International Journal of Information Management*, 36(4):668 – 672, 2016.

[Knoa] Open Knowledge. Global Open Data Index - Methodology. `http://index.okfn.org/methodology/`. Visited: October 23, 2016.

[Knob] Open Knowledge. Open Knowledge: The Open Definition. https://okfn.org/about/our-impact/opendefinition/. Visited: October 14, 2016.

[Knoc] Open Knowledge. Place overview. http://index.okfn.org/place/. Visited: October 23, 2016.

[OASa] OASIS. About us. https://www.oasis-open.org/org. Visited: October 30, 2016.

[OASb] OASIS. Members. https://www.oasis-open.org/member-roster. Visited: November 1, 2016.

[OASc] OASIS. Standards. https://www.oasis-open.org/standards. Visited: October 30, 2016.

[ODaa] OData. Ecosystem. http://www.odata.org/ecosystem/. Visited: November 1, 2016.

[ODab] OData. Libraries. http://www.odata.org/libraries/. Visited: November 1, 2016.

[ODac] OData. OData - the best way to REST. http://www.odata.org/. Visited: October 30, 2016.

[Ope] OpenLab. Open Government Data Initiative v6. https://github.com/openlab/OGDI-DataLab. Visited: November 1, 2016.

[Ove07] Hagen Overdick. The Resource-Oriented Architecture. In *Services, 2007 IEEE Congress on*, pages 340–347. IEEE, 2007.

[PHZ16a] Michael Pizzo, Ralf Handl, and Martin Zurmuehl. OData Version 4.0. Part 1: Protocol Plus Errata 03. http://docs.oasis-open.org/odata/odata/v4.0/errata03/os/complete/part1-protocol/odata-v4.0-errata03-os-part1-protocol-complete.html, June 02 2016. Visited: October 30, 2016.

[PHZ16b] Michael Pizzo, Ralf Handl, and Martin Zurmuehl. OData Version 4.0. Part 2: URL Conventions Plus Errata 03. http://docs.oasis-open.org/odata/odata/v4.0/errata03/os/complete/part2-url-conventions/odata-v4.0-errata03-os-part2-url-conventions-complete.html, June 02 2016. Visited: October 31, 2016.

[PHZ16c] Michael Pizzo, Ralf Handl, and Martin Zurmuehl. OData Version 4.0. Part 3: Common Schema Definition Language (CSDL) Plus Errata 03. http://docs.oasis-open.org/odata/odata/v4.0/errata03/os/complete/part3-csdl/odata-v4.0-errata03-os-part3-csdl-complete.html, June 02 2016. Visited: October 31, 2016.

[PZL08] Cesare Pautasso, Olaf Zimmermann, and Frank Leymann. RESTful Web Services vs. "Big" Web Services: Making the Right Architectural Decision. In *Proceedings of the 17th international conference on World Wide Web*, pages 805–814. ACM, 2008.

[RR08a] Leonard Richardson and Sam Ruby. *RESTful Web Services*. O'Reilly Media, Inc., 2008.

[RR08b] Leonard Richardson and Sam Ruby. *RESTful Web Services*, chapter The Resource-Oriented Architecture, page 79. O'Reilly Media, Inc., 2008.

[SZJG14] Iryna Susha, Anneke Zuiderwijk, Marijn Janssen, and Åke Grönlund. Benchmarks for Evaluating the Progress of Open Data Adoption: Usage, Limitations, and Lessons Learned. *Social Science Computer Review*, 2014.

[Tin13] Dinand Tinholt. The Open Data Economy: Unlocking Economic Value by Opening Government and Public Data. *Capgemini Consulting Analysis*, 2013.

[UGSATa] Office of Citizen Services U.S. General Services Administration and Innovative Technologies. About data.gov. `https://www.data.gov/about`. Visited: October 17, 2016.

[UGSATb] Office of Citizen Services U.S. General Services Administration and Innovative Technologies. About data.gov. `https://www.data.gov/data-request/`. Visited: October 17, 2016.

[Won10] John Wonderlich. Ten Principles for Opening Up Government Information. *Washington, DC: Sunlight Foundation. August*, 11:2010, 2010.

[YR12] Harlan Yu and David G. Robinson. The New Ambiguity of 'Open Government'. Technical report, 59 UCLA L. Rev. Disc. 178 (2012), February 28 2012.

[Ösa] Wikimedia Österreich. FAQs – Häufig gestellte Fragen. `https://www.opendataportal.at/faqs/`. Visited: October 16, 2016.

[Ösb] Wikimedia Österreich. Impressum. `https://www.opendataportal.at/impressum/`. Visited: October 16, 2016.

[Ösc] Wikimedia Österreich. Open Data Portal Österreich. `https://www.opendataportal.at/`. Visited: October 16, 2016.