UiO : Department of Informatics
University of Oslo

Hypermedia RDF for driving applications

How to enable smarter clients using true REST

Kristoffer Lie Braathen

Master's Thesis Autumn 2015

Hypermedia RDF for driving applications

Kristoffer Lie Braathen

17th August 2015

Acknowledgments

First of all, I would like to thank my supervisor, Kjetil Kjernsmo, for guiding me through this bumpy road. He has been essential for this thesis with his feedback when it comes to the development of the prototype and the writings.

I would also like to thank my co-supervisor, Ruben Verborgh, who is settled in Gent. He have helped me through the development given me input on how particular parts should be solved.

Writing the master thesis is not only an individual and lonely process. My fellow master students have helped me; the guys in 10th floor who have been keeping me busy playing chess, and especially the people in the ninth floor for all the fun conversations, the forever lasting coffee breaks and the card games.

At last, I would like to thank my family and friends who have believed in me, and supported me throughout this period, which made it possible for me to accomplish this.

Thank you all. This could not be done without you.

Abstract

In this thesis we will examine how we can make use of Semantic Web technologies, such as RDF, to create an application based on the Representational State Transfer architectural style, with a client that is both generic and autonomous. Further, we will look at how hypermedia enables self-descriptiveness, both for humans and machines.

The study is based on the development of the client. We have focused the work around the hypermedia messages, enabling the client to consume APIs, understand actions and relate those actions to its goals. To explore the self-descriptiveness, or the readability, we held a workshop with other developers.

Results shows us that it is feasible to develop a client that can be both generic and autonomous, through the use of true REST. With a relative new RDF serialization we could express semantic hypermedia which makes it possible to use a generic format, with the power of being specific. This again let the participants of the workshop relatively easy understand an API they had never seen before.

We hope that this thesis strengthens the understanding of REST, and that it encourage others to investigate the capabilities of clients interacting with hypermedia.

Contents

List of Figures

List of Tables

Listings

Abbreviations

API	Application Programming Interface
HTML	Hypertext Markup Language
HTTP	Hypertext Transfer Protocol
IRI	International Resource Identifier
JSON	JavaScript Object Notation
JSON-LD	JavaScript Object Notation for Linked Data
LD	Linked Data
LOD	Linked Open Data
N3	Notation3
OWL	Web Ontology Language
RDF	Resource Description Framework
RDFS	Resource Description Framework Schema
REST	Representational State Transfer
SPARQL	SPARQL Protocol and RDF Query Language
URI	Uniform Resource Identifier
URL	Uniform Resource Locator
WWW	World Wide Web
W3C	World Wide Web Consortium
XML	Extensible Markup Language

Chapter 1

Introduction

The Web today contains a tremendous amount of data, where most of it is human-readable documents. From Vannevar Bush's article about the Memex[14], Ted Nelson's Hypertext[40] and to Tim Berners-Lee articulation of the World Wide Web and Linked Data, the thoughts of linking documents together have existed. Bush had the idea of creating a machine where you could look up related articles and use it in science so other fields of studies could also access them. While Nelson first introduces us to the term *hypertext* that could be used for this purpose. What the Semantic Web community strives to achieve, next to make the data machine-readable, is to link all this data together, which was clearly stated on the first ever published Web page:

> World Wide Web is a wide-area hypermedia information retrieval initiative aiming to give universal access to a large universe of documents.[59]

Semantic Web can, in short, be seen as a layer outside today's Web. By some it is proposed as Web 3.0 since it takes the Web from one stage to another. The reason for this, is because it uses new technologies to give meaning to the already existing data that is available on the Web. By meaning, it is meant that computers can understand the content as well, not only human beings. This concept, together with the principles of how to link data togheter, through the Linked Data principles, are the core of the visions Tim Berners-Lee had for the Web.

Representational State Transfer, or the abbreviation REST, is an architectural style that is popular on the Web. The term is heavily used the last couple of years, and there are many APIs that claim to follow this style. In most cases, since it is frequently misunderstood, they are probably not. This means that there exists many APIs that do not really take full advantage of the style. The problem is that developers tend to forget the essence: applications that are RESTful must be driven by the messages that are passed in the system, between the server and the client, and not an API that is externally documented. Hypermedia is the term we use for the content that enables this.

When developing websites or other programs, a developer should always make the design so understandable that other humans can interact with it. This is something we would like to achieve for computers as well — let the documentation both be understandable for humans and computers. If we design the documentation right, we can achieve what we call *view source capability*. By that we mean that developers should be able to view the data that is sent, and easily get an understanding on how the API works. We believe that such a view source approach will win over time over APIs that have external documentation because of the ease of use.

A way of expressing data like that, is through the use of RDF or a serialization of it. RDF is one of the Semantic Web technologies that enables us to provide both the data and make it clear for the application what it can do with the message.

1.1 Research

In this thesis we will explore the capabilities that lays within REST and RDF, and create a client that is generic and autonomous. By this we mean that the client can work on different APIs and that it makes actions without human interaction. With this client we will investigate the following research questions:

- To what extent can a client be autonomous and consume an API it has never seen before?
- Can it reason about the consequences of actions, will it be able to relate it to its own goals?
- And if so, what role do hypermedia and RDF play therein?

To be able to test the client, the back-end must serve it. Through the thesis we will get an understanding of what decisions were made throughout the development of the application, both server and client side. The most essential part here is how the hypermedia messages were designed to contain valuable information for the client, so that it could drive the application. Hypermedia enables us further to explore the view source capabilities that lays within our chosen serialization of RDF.

1.2 Related work

The last couple of years there have been attempts on enabling generic and autonomous clients. Still, we have not seen much of the actual clients, only the technologies that are supposed to enable it. This calls for that it is a complex task.

Amundsen[1] shows us three different application that is driven by hypermedia. He has also been the author to several hypermedia formats. The drawback with his implementations is the lack of autonomy and

genericity. If one feature is present in one application, the other one is missing.

One of the driving forces within the hypermedia field is Lanthaler with his Hydra Console[30]. This is a generic tool that consumes all APIs that are based on a particular vocabulary, i.e. Hydra. This tool helps humans browse such APIs and was constructed to proof the generic nature of the vocabulary. The problem arises since it is not autonomous, which is a factor that we take into consideration as well.

Another attempt, from this year, by Bergwinkl[5] was to create a client that makes use of Hydra as well. This is a promising project which at the moment lacks the generic feature. So far the client is given all URIs that is needed.

This thesis will try to make a contribution of showing a proof of concept, that we are capable of making a client that is both generic and autonomous based on existing technologies.

1.3 Chapter overview

- **Chapter 2 - Background** will give us a deeper understanding of the different terms and technologies that where mentioned in the introduction and we dive into other relevant Semantic Web technologies.
- **Chapter 3 - Hypermedia** is where we will introduce REST and how to construct self-descriptive messages with hypermedia controls.
- **Chapter 4 - Prototype** will bring us through the steps of the implementation of the prototype; the generic and autonomous client. This is a way of exploring the capabilities of the technologies.
- **Chapter 5 - Exploring view source capabilities** takes us through the workshop that were held to validate the view source capabilities of the API created in the implementation.
- **Chapter 6 - Results** presents the hypermedia controls that were implemented and how the prototype did execute.
- **Chapter 7 - Discussion** is the chapter where we review our research questions, and discuss our findings, and recommend future work.
- **Chapter 8 - Conclusion** finalizes this thesis with concluding remarks and the contributions of this thesis.

Chapter 2

Background

The Semantic Web makes use of different technologies to function. Some of them are essential for the Web as well, after all, we have already claimed that it is an extension to the existing Web[9] and thus makes use of the same fundamental parts, and some are solely unique to the Semantic Web. In this chapter we are going to introduce the most important parts of the complete Semantic Web's technology stack, which can be seen in figure 2.1[1], and the architecture of the Web. This is essential to understand and be able to see the capabilities which lies within the Semantic Web, and therefor also our application and domain.

As a reminder, the structure of this chapter will be as following: first we will go through the basic ideas of the Semantic Web. We will then move on to the architectural properties of the Web which is the fundamentals of how the Web is built and how it works. This is followed with the important Semantic Web technologies; RDF as the knowledge format, RFDS and OWL to express more semantics and ontologies, URIs to avoid name conflicts, and SPARQL which enables querying of semantic data. In the end we will look into the principles of Linked Data that is an important part of the creation of the Semantic Web.

2.1 Semantic Web

The visions of the Semantic Web were expressed by Berners-Lee, Handler and Lassila as a "new form of Web content that is meaningful to computers"[9] and that establishes new possibilities compared to the existing Web.

If there existed an intelligent Semantic Web agent that could reason over data and information that human users provided it with, possible tasks it could carry out would maybe be something like the three following scenarios:

One scenario could be that based on your favorite genre of music, you would like a Semantic Web agent to produce a schedule for a festival, where it only schedules the bands that fits your taste of music, and it also

[1] Visit http://semanticweb.org/images/3/37/Semantic-web-stack.png for bigger version.

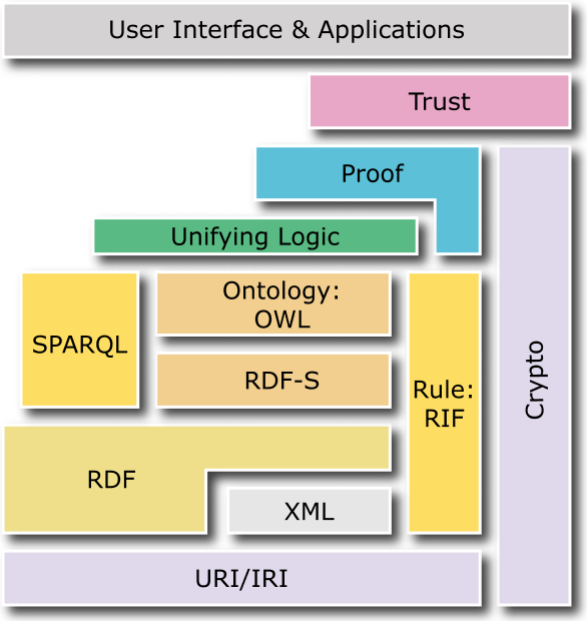

Figure 2.1: The Semantic Web technology stack.

considers time; two concerts, on two different stages, at the same time, are impossible for one person to reach. In this case, the input from the user would be the preferred kind of music and what festival you will attend. The job for the agent would then be to produce the schedule, and then provides it to the user through an interface, like a Web browser for instance.

Another scenario could be that you feel sick, you have a terrible headache and wants to go to the doctor. What you will then do, is to provide the Semantic Web agent your personal details and what you are struggling with. The agent will then find a doctor that is available, that have the right kind of knowledge to heal you and sets up and appointment.

A third scenario could be that you would like to go to a concert on a specific date. In this case, dependent on the implementation, you may have to provide the agent with information about your preferred genre of music, how many tickets you want, what date and what the price limit is. The agent will then crawl the Semantic Web for venues close to your location and book the tickets for you. If a venue hosts a concert that fits the provided data, the Semantic Web agent could book the tickets for the user.

In this thesis, our prototype, or implementation, will be consuming two APIs based on the two last scenarios, which it can relate to its own goal and carry out tasks to achieve it. To be able to develop such a Semantic Web agent, we need to establish the fundamental parts of what the Semantic Web consists of.

Expressing Meaning

Berners-Lee et al.[9] first articulates the need for a way of expressing meaning to already existing content on the Web, and future content. Today

is still most of the content on the Web not understandable for machines. This fact might change, especially since Google have added support for a way of giving content meaning[47], which is an important for the Semantic Web. There does not exist many applications today that combines data from several different sources. This is because most of the content is published in a way that is only human readable and therefor needs hard-coding to being used in applications. The Semantic Web tries do something about that.

The Semantic Web will make it possible for computers to browse through different web pages and understand the meaning of the content, it will also be able to see if the content of different pages can be used together, if it contains the same information and whatnot. In on way, it can be seen as an extension of the current Web. This enables agents to carry out complex tasks for users, like the ones we illustrated in the above scenarios. According to Berners-Lee et al.[9], will the Web evolve into the Semantic Web which makes it possible for computers to understand and process automatically the data and information that exists on the Web. At that stage, it is then we really can see how powerful hyperlinks can be. Agents will be able to gather structured data with meaning from different sources and carry out tasks with this data that gives meaning.

Knowledge Representation

To be able to let the Semantic Web function when it comes to expressing meaning for computers, there is a need for a structured way of expressing the information and the data. Berners-Lee et al. describes that there is also a need for sets of inference rules as well for enabling automated reasoning, which brings us to one of the challenges that is to provide a language "that expresses both data and rules for reasoning about the data that allows rules from any existing knowledge-representation system to be exported onto the Web."[9] The good thing is that there already exist technologies that provides us with these capabilities. These technologies are XML and RDF. XML lets everyone create their own tag which is helpful for structuring data. To express meaning and connect data to each other we do need RDF.

RDF is the technology this thesis will be focusing on and it is described in more details further down in the chapter. The most important element to remark is that RDF is built up by URIs, which stands for Universal Resource Identifier, which again works like any link on any web page; it helps us associate resources with links, and when a resource has a link, anyone on the Web can retrieve the representation of it and create new concepts and resources that might be related to it[8].

Ontologies

When a computer program tries to understand the meaning of the data that it is provided with from a web page, then it need something to relate it to. This is especially essential if it tries to process data meaningfully from two different sources. What the computer program needs is collections of

information called ontologies; in the world of Web-researchers ontologies are documents that defines relations between terms formally[9].

If we look at a word that gives meaning to human beings reading from a web page, for example the term *spouse*. Typically, this term will in a ontology be typically defined as inverted. This means that if we are stating that *Joan has the spouse Bob*, then we are implicitly stating that *Bob has the spouse Joan*. Berners-Lee et al.[9] states that this helps us create programs that are more advanced in the fashion of creating more accurate searches, the program can, based on inference rules, reason over the data and connect data in a sophisticated way. We will later go into further depth on how ontologies are built.

Agents

One time in the future, when there exist a significant amount of computer programs that enhances the technologies of the Semantic Web, then we really can see the power of it. The diverse range of programs will then, according to Berners-Lee et al.[9] gather data from different sources on the Web, process the content and exchange the results.

Put in another way, we could say that one program can create knowledge about a topic and share it on the Web while another program can do this about another topic. For a third computer program to use the information created by the two programs it needs a vocabulary, which contains the ontology as well, to understand the meaning of data and how the data is connected to each other through the inference rules of the ontology. Through the use of ontologies and and computer programs publishing meaningful content, the Semantic Web will grow and make it better and more valuable. If an ontology was to be built by using RDF, such as OWL, the ontology could refer to terms in other ontologies.[8] The ontologies becomes distributed and reusable which is perfect for the Semantic Web. And this brings us to the evolution of knowledge.

Evolution of Knowledge

The last thing of Berners-Lee et al.[9] visions of the Semantic Web, is the evolution of knowledge. We have already touched upon it; by naming every concept, or term, by a URI, it enables everyone to create new concepts. These concepts can again be described by ontologies which again makes it possible for programs to infer knowledge and reason over it. This will again create new links to other resources and concepts and the Web will evolve. Humans creating web content, whatever it is, will, by making use of the Semantic Web technologies, be able to be reached on the Web, not only by humans, but also by agents that are looking for data.

If we bear the above information in mind throughout this chapter, we will see that these visions are indeed the fundamentals of the Semantic Web.

Following up is the underlying architecture the Semantic Web relies on.

2.2 World Wide Web

The creation of the World Wide Web (WWW or the Web) was one of the first steps towards the Semantic Web, because the Semantic Web are functioning as a layer on top of the Web. As we will see in this section, while we are going through the architecture of the WWW, it uses technologies that achieves scalability, that means that it scales well based on how many users that are using it and connects to wast amount of different parts of it.

Because of its efficiency and utility as well, it have grown into the single, largest information space of linked information, in the world. This information is often referred to as resources and as we have seen in the section of Semantic Web, the way we identify resources is with Uniform Resource Identifiers (URIs). Further in this section we are looking into the architectural properties of the Web, defined by the World Wide Web Consortium (W3C)[11].

2.2.1 Identification

First of all, when you have a global network of information there definitely should exist a clever way of identify all the different resources of information in the network. And it does exist. There is a lot of different syntaxes of global identifiers for distinguish different artifacts from each other, like the ISBN for books. The way the Web have made use of unique identifiers to distinguish between resources in the global network, is, as mentioned above, the URIs. One example of how a URI could look like is http://www.example.com/pages/coffee. This URI could for instance identify a resource that tells us something about coffee.

The URI enables one of the goals of the Web that is to let any part "share information with any other part" within the global network and by giving your resource of information a URI then it can be identified and accessed by anyone.[11] The main benefit of the URI is that if two different sources of information are talking about the same thing, then the two sources can provide the URI to the referred source. These three different sources will of course have their own URIs. This is what we call *linking* and is made possible through the use of *hyperlinks*[40]. Other benefits of the URI that W3C[11] mentions are bookmarking, caching and that it is possible indexing it for search engines.

When we are referring to resources, we are referring to what ever that might be identified by a URI; a web page, an image, a word. To be a bit more precise, all information that have an URI and that can be transmitted in a message over the Web are "information resources"[11], or in short just resources.

So far, in the world of the Web, we do have *URIs* for identification of *resources* that essentially is information within a global network.

2.2.2 Interaction

The second thing we have to consider is how the interaction between the different agents should happen. What is common to consist in communication between the agents are URIs, messages and data. We already know that the URI identifies the resource, while the message include data and metadata about the identified resource, the data in the message and the message itself[11].

The protocols of the Web takes care of the exchange of the messages in different situations, and some of them that are used today are:

- **HTTP** HyperText Transfer Protocol
- **FTP** File Transfer Protocol
- **SMTP** Simple Mail Transfer Protocol
- **IMAP** Internet Message Access Protocol

HTTP is the most important protocol for the Web and is the most common one when it comes to interaction with regular web pages. This protocol is the one we will be using in the examples throughout the thesis.

HTTP is called a request/response protocol[20] because the client sends a request to the server, while the server responds to this request. The request contains different information such as the URI, what kind of method is used and protocol version, client information and a possible message-body. The server responds in a similar way where the most notably is the success or error code and body-content if that is what the request triggered.

For accessing a resource, the agent dereferences the URI, and with HTTP it is done with the method called GET. For other protocols, other methods are used. The HTTP GET will let the agent retrieve a representation of the accessed resource. There exist other method for adding or modifying the resource with POST or PUT, and you can delete the representation of the resource with DELETE.

2.2.3 Data Formats

The data that is sent in the message over the Web through HTTP is on a particular format. This format varies from application to application, it could even vary from different request within an application, and is specified in the message as metadata as well so that agents can state that they only accept messages that contains data on the format e.g. `application/json` or `text/rdf`. Data formats are there for structure the data so that agents can agree on how to interpret the data of the resource.[11] Because of the constant evolution of applications, the Web needs to be flexible and, hence the formats, the Web architecture does not constraint when it comes to which data format content providers could choose to use. That said, a preferred way is to use data formats that already exist. To create and implement new data formats is expensive,

so generation of new ones often builds upon or extends already existing ones[11], e.g. `application/ld+json` can be seen as an extension to `application/json`.

2.2.4 General Architectural Principles

Orthogonal Specification

This term, orthogonality, in the context of the World Wide Web, means that different features of the Web can be used in arbitrary combinations, but still end up with the same result[45, p. 228]. For the Web will this mean that the technologies behind the architectural properties of the Web, identification, interaction, and representations enables them to be orthogonal concepts and will therefor be able to evolve independently without any disorder in the functioning of the Web[11].

E.g. URIs are used for identifying resources, but you can still publish URIs on the Web without having any representations of the resource. This is confusing and breaks the principles of REST that we will introduce in the next chapter. For now, we can note that the orthogonal specification helps the Web stay robust and and flexible.

Extensibility

By extensibility, since the Web is evolving, we mean that the Web have the power of being extended; both by more information and the technologies that are used to represent and exchange that information.[11] Both the information and the technologies can change as well without the Web to break. E.g. one resource on the Web can either be change by an agent that extends it with more information, or it can be moved to another URI. With the technologies it can either change the way it handles the information, or that you extends the functionality of it, or create something new like a new media type. This helps us realize that the Web consist of orthogonal concepts and specifications.

Error handling

The Web is a global network of information and errors occur. When this happens the technologies and the different components of the network needs a way of handling those errors. The way the Web handles errors is in the way that agents takes care of it, not the users. If an error occurs when an agent tries to connect to a web page, then it receives an error code from the server[2], e.g. 500 Internal Server Error. W3C[11] states that based on the error message, the agent knows if it needs to go through *error correction* which means that the agent repairs what was broken and the system operates as the error never occurred. Another way of handling errors is *error recovery*, which means that the agent does not repair the error and keeps on processing in a state that it knows that the error happened.

[2]This is what happens when the communication goes over HTTP.

Protocol-based Interoperability

The principle of protocol-based interoperability is quite self-describing. The Web consist of many parts that vary in in how they work. To be able to function in a proper way together there needs to be structured standards for how to act in the global network. I.e. if there were no standards or protocols no part would be able to know how to send a message or how to interpret a received one.

After diving into the architectural principles of the Web, we have now a better understand of the underlying architecture that is part of enabling the Semantic Web. Now we need a way to express meaning and represent knowledge.

2.3 Resource Description Framework

As mentioned in the introduction, we do have a framework for describing data that enhance the principles of the Semantic Web — the Resource Description Framework, or RDF in short. This is a model that can be used for interchanging data on the Web. It extends the existing linking structures of the Web by using URIs as names for relationships and resources.[15] This way of naming and structuring the data is reflected through the triple pattern. Triple patterns are built with a subject, a predicate and an object. This means that all data on RDF triple pattern format[3] looks something like this:

Listing 2.1: RDF document in Turtle syntax.

```
@prefix foaf: <http://xmlns.com/foaf/0.1/> .
@prefix rdf: <http://www.w3.org/1999/02/22-rdf-syntax-ns#> .
@prefix dbpedia-owl: <http://dbpedia.org/owl/> .
@prefix dbpedia: <http://dbpedia.org/resource/> .

<dbpedia:Led_Zeppelin> rdf:type dbpedia-owl:Band ;
  foaf:name "Led Zeppelin" ;
  dbpedia-owl:recordLabel dbpedia:Atlantic_Records .
```

In the example above we state that this thing with the URI that we are going to describe is of the type band, that this band is called Led Zeppelin and that they are connected to the record label Atlantic Records. The URI, dbpedia.org/resource/Led_Zeppelin, is the subject in all of the three triples. Then the predicate, or what can be seen as the relation, comes before the object. So the predicate connects the subject and the object to each other. If we decide to draw these triples, we can easily see that this linking structure creates directed, labeled graphs.[15] And this graphs are what the Semantic Web consists of, thus it helps us express meaning and represent knowledge.

[3] These triples are written in Turtle[4] syntax which is a more readable syntax than native RDF where all URIs are written fully out.

The visions for RDF, in the Semantic Web, was to provide the existing Web with a way of expressing meaning and knowledge representations.[8] Speaking in such terms, we have to say that RDF have been successful by the means that RDF represents knowledge readable both for humans and for machines.

2.4 RDF Schema

RDF Schema also helps us express meaning and gives us the capability of representing knowledge, just as with RDF. This is because RDFS can be seen as an extension to RDF. More formally does this mean that RDFS provides RDF data with a data-modeling vocabulary.[13] What the basic RDF vocabulary gives us is e.g. the possibility of describing what resources are. This is done by the rdf:type predicate e.g.

```
<dbpedia:Led_Zeppelin> rdf:type dbpedia-owl:Band .
```

With the extension, RDFS, we can bring more semantic to to the data we are describing. We can describe "groups of related resources and relationship between these resources."[13] For example, if you have a schedule for a festival, and would like to state that a specific band is playing, then you can you state, helped by RDFS that

```
vocab:performer rdfs:range dbpedia-owl:Band .
```

The benefit of being able to implement semantics like this, is that if an automated agent wants to find all performers at a festival, then it can divide the performances into different groups, if some of the performers are not bands, like painters.

This approach of describing data, that RDF and RDFS does, is called property-centric[13] and helps us maintain the architectural principles of the Web, extensibility[11], by letting anyone extend the descriptions of resources without interrupting parts in the global network.

RDFS and OWL can be used together to reuse and create vocabularies. The co-existence of them gives us the third property of the Semantic Web; ontologies to help us create complex structures, meanings and inference rules to the data.

2.5 Web Ontology Language and Reasoning

As we now know, ontologies are one of the building blocks needed to fulfill the power of the Semantic Web. Ontologies are the first level above RDF where we "formally describe the meaning of terminology used in Web documents"[38]. We will in this section go into further depth on what OWL is, how it functions and what the capabilities of the technology is. The reason we present OWL here is because it is widely used and it helps us get an understanding on how ontologies and vocabularies are built.

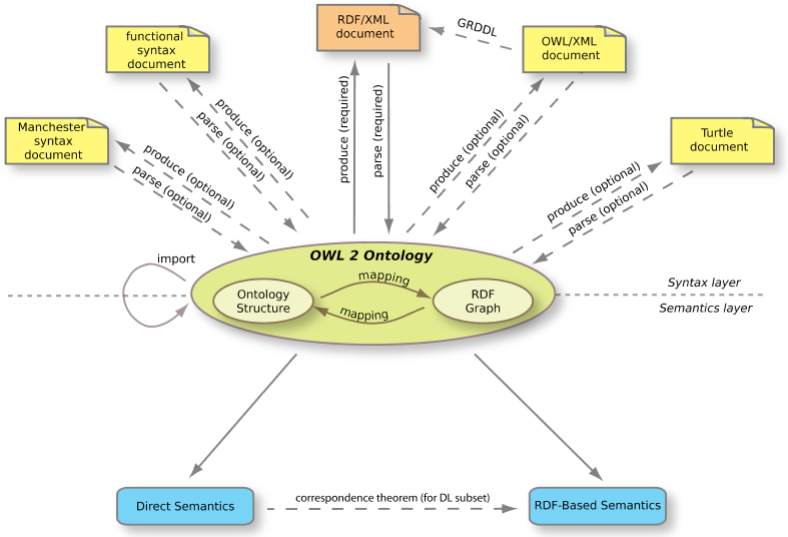

Figure 2.2: The structure of OWL Web Ontology Language.

The structure of OWL can be seen in figure 2.2[4]. The oval in the middle represents the ontology, or vocabulary, where the terms used in an application domain are stated. On top are different syntaxes, which usually there is only one of. At the bottom are the specifications of the semantics in the ontology.

The first thing we need to understand is what an ontology is. "Ontology is a term borrowed from philosophy that refers to the science of describing the kinds of entities in the world and how they are related."[46] Since ontologies has something to do about describing what exist in the world, OWL is therefor a language for describing what exist on the Web, by defining and instantiating Web ontologies. These ontologies consists of descriptions of classes and properties, and instances of them. Based on the OWL ontology, a computer program would be able to perform useful reasoning over the data that is given. This is because the semantics of OWL "specifies how to derive logical consequences"[46]; it exceeds the semantics of RDFS.

If an ontology states that Robert Plant is a band member of Led Zeppelin, and that Jimmy page is a band member in the same band as Robert Plant, then the computer program can infer that Jimmy Page is a band member of Led Zeppelin as well because of the semantics that will be stated in an ontology like that. If the above example where stated in N3 language[7], the reasoner called EYE[44] could have done the job to infer that Jimmy Page also was a band member of Led Zeppelin. We see that OWL helps us create applications that not only present information to humans through the Web browser, but gives us the capabilities of enabling applications that processes the content of information as well.

There are no definite way of distinguishing the difference between

[4]Visit http://www.w3.org/TR/owl2-overview/ for larger version.

a vocabulary and an ontology. We will in this thesis see vocabularies as smaller, with less complex structures and rules than ontologies. Still, what they share is that they define concepts and relations to describe and represent resources. This involves e.g. defining constraints on how to use a term, if data is gathered from different sources you can state that to terms have the same meaning, or you can state how a term is related to other terms.[57]

Since OWL is one of the building blocks of the Semantic Web, one of the aims of the creation of OWL is to enable automated reasoning and processing over Web content. It is too big of a task to re-structure all the content of the Web, and as we remember, one of the principles of the Web is extensibility. Smith, Welty and McGuinness[46] states that OWL must work in a way where you can add information to already existing resources as well. Another requirement is that OWL must work with distributed sources since the Semantic Web is a global distributed information system. Ontologies could thus be related. An example of this is if you create your own ontology, but uses terms from already existing ontologies like *The Music Ontology*[5].

As a result of this, that descriptions of resources does not only exist within one file, we can say that OWL, as well as the Semantic Web, "makes an *open world assumption*"[46]. To have an open world assumption means that everything that is not stated *can* be true. Or it can be false. In an *closed world assumption* it is opposite; everything that is not known is false. A class *MusicGroup* might be defined in The Music Ontology, the description of the class can be extended in other ontologies, e.g. you could define a sub-class of MusicGroup called *Band*. In both worlds it would be true that *Band* is a sub-class of *MusicGroup*. The difference shows if we want to know if *Pink Floyd* is a *Band*. We the limited knowledge we have in this case, the answer would be "no" in an closed world assumption, but it would be "I can not tell" in the other world.

One of the problems of this is that the new information that is added to a class can not retract the already existing information, and as a consequence the new information can create contradictions. You can never delete facts and entailments, only add it. Since it is a possibility for creation of contradictions, this is something the designer of the ontology has to keep in mind.[46]

2.6 Linked Data

Linked Data is a term that is important within the field of Semantic Web, and we understands that it has something to do with linking resources, as data, together through URIs or URLs. More formally, we can state it like Bizer, Heath and the founder of the Web, Bernes-Lee, that it "refers to a set of best practices for publishing and connecting structured data on the Web."[12] This have helped the evolution of the Web from being a global network of linked documents growing into a global network of linked

[5] Available via the link http://musicontology.com/.

documents and data. As we will learn more about below, *hypertext* has been crucial in this development. The hypertext links have enabled users to easily access documents via other documents through Web browsers.

> Linked Data refers to data published on the Web in such a way that it is machine-readable, its meaning is explicitly defined, it is linked to other external data sets, and can in turn be linked to from external data sets.[12]

To achieve this we need, again, turn to RDF. With Linked Data we are creating typed links, using RDF, between data that are hosted at different sources. Without RDF, a computer program would most probably not be able to understand data from two different sources, unless it was hard-coded to understand exactly those two sources. This breaks with the intention of enabling automated processing of data on the Web. Linked Data makes use of RDF to create typed statements about things in the world, thus link them together. The result is something that Bizer et al. call the "Web of Data"[12], which also can be seen as a web of things that is described by data on the Web. This helps us realize that one of the most important factors that are being enabled through Linked Data, is that it helps data from different sources on the Web to be used together, either by being tied to each other by a link or that a computer program can extract data from each source that occurs in its universe of discourse and process it in a meaningful way.

Linked Data makes it easier for computers to answer questions like *what band members did Led Zeppelin have*? The reason why this is possible, is if a music blog has written something about Led Zeppelin and linked to their DBPedia page, then it is really easy to look up what band members that took part of the band.

2.6.1 Linked Data principles

Further on, Berners-Lee[6] articulates four principles of Linked Data[6]:

- Use URIs as names for things.
- Use HTTP URIs so that people can look up those names.
- When someone looks up a URI, provide useful information, using the standards (RDF*, SPARQL).
- Include links to other URIs, so that they can discover more things.

As we will see in the next chapter, these four principles all fits well to Fielding's vision of how large-scale distributed hypermedia systems like the Web should be structured; by stating that you should name resources with URIs so that others can look it up. When looking up resources, useful information like links to related resources should be provided — this

[6]The principles have been known before, but not articulated in this particular way.

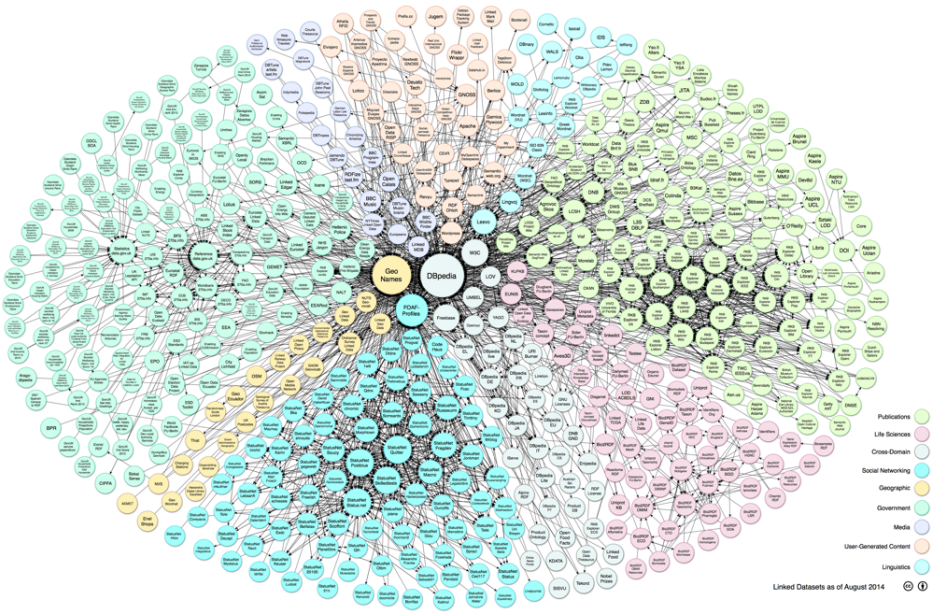

Figure 2.3: The Linked Open Data cloud.

encourages discovering of more information. Of course, these principles are not in-depth architectural constraints, but they serve as guidance for how to link data to each other. When it comes to how to consume Linked Data, the nature of it provides the necessary semantics for a computer to e.g. use the "follow your nose"[58] technique, which means that the it can follow links based on what it is looking for. This again is an enabler for autonomous clients.

2.6.2 Linked Open Data

To encourage data owners, especially governments, to publish what can be categorized as good Linked Data, Berners-Lee[6] have made a rating system for how well the open data is integrated with the the Linked Data principles. In relation to this, he introduces the term Linked Open Data, which is basically Linked Data that is publicly open. This rating system could also be used on regular Linked Data, only that to be open, everyone must be able to access them. For each star, the data must fulfill the requirements of the previous stars.

1. Available on the web (whatever format) but with an open license, to be Open Data
2. Available as machine-readable structured data (e.g. excel instead of image scan of a table)
3. As (2) plus non-proprietary format (e.g. CSV instead of excel)
4. All the above plus, Use open standards from W3C (RDF and SPARQL) to identify things, so that people can refer to your data

5. All the above, plus: Link your data to other peoples data to provide context

Please see figure 2.3[7] to get an image of how intricate the Linked Data world is with all its major Linked Open Data sources.

2.6.3 The Linked Data Technology Stack

Linked Data relies on two technologies that are part of the architectural principles of the Web; to identify resources it needs URIs and for interaction it needs HTTP. We are using HTTP to dereference URIs, this helps us retrieve descriptions of resources that can not be sent over the network itself[12] — such as a band, e.g. Led Zeppelin — but the description of the band can be sent. Those two technologies are again supported by RDF. Since Linked Data is using URIs as identifiers, HTTP for interaction and RDF for representing resources it directly builds upon the Webs architecture. Once again we can see that the Semantic Web, the Web of Data, can be seen as a layer on top of the existing Web we already know.

As we remember in the Semantic Web section, the idea of a Semantic Web started out with the desire to extend the capabilities of the Web by extending it to becoming a space where you can publish structured data. The goal of the Semantic Web is to create a global network, or a Web of data, that is machine-readable. To achieve this we need to fulfill the process of publishing machine-readable data on the Web, to create that Web of data. Linked Data is a way to reach that goal.[12]

2.7 SPARQL Query Language

As a last technology of the Semantic Web technology stack, SPARQL must be mentioned. SPARQL is a query language for RDF where you can express the queries across different data sources[24]. It contains support for aggregation, sub-queries, negation, value testing, value expressions and you could constrain a query to a given source. Through the built in CONSTRUCT you can build your own graphs of RDF; basically, if the data is expressed in RDF, you can build whatever graph wanted. As an example, if you want to create a query that retrieves all triples about Led Zeppelin on their dpbedia page, the query could look like the following listing:

Listing 2.2: A SPARQL query where we CONSTRUCT a graph with data about Led Zeppelin.

```
CONSTRUCT { ?s ?p ?o }
WHERE {
GRAPH
    <http://dbpedia.org/resource/led_zeppelin>
        { ?s ?p ?o } .
}
```

[7] Visit http://lod-cloud.net/versions/2014-08-30/lod-cloud.svg for larger version.

If you are building your own application where you use data from the Semantic Web, gathering it and structuring it, you usually store the triples from the SPARQL queries in triple stores like OWLIM-Lite[49]. Triple stores are basically graph data bases, because they store RDF data which again is graphs.

Summary

In this chapter we have gone through the most essential terms and technologies, their capabilities and what value they bring computers, that takes part of the Semantic Web world. As a starter, we encompassed the visions of the Semantic Web, followed up with different sections on how to achieve it. First up was an introduction to the architecture of the World Wide Web that made us understand how data can be interchange over the Web. Then we looked into the different technologies that helps machines uniformly understand the content on the Web. These technologies were RDF, as a common way of expressing knowledge, RDS as an extension to RDF, OWL as a further extension to express complex rules that again can be used to reason over. The Semantic Web is made up of Linked Data and makes it possible for machines to understand the content on the Web. As a way of gathering semantic data from different sources SPARQL can be used.

As a whole, this is what makes up the foundations of the Semantic Web, and have brought us a bit closer to understand how we can achieve Berners-Lee's ultimate goal; developing autonomous, generic client that takes part as a consumer of Web content, just as a human being.

Chapter 3

Hypermedia

We have now successfully been diving into the what the Semantic Web consist of. In this chapter we, will introduce an architecture for modern Web applications which is called REST. One of the essential constraints of the architectural principles of REST concerns around the term *hypermedia*. How REST is connected to the Semantic Web is through our belief that RDF is a great way of defining hypermedia messages. To fully understand this concept, we dedicated the whole chapter to this term.

To start with, we will go into details of the architectural principles of REST. We then move on to hypermedia itself, where different concepts and terms as *affordance* and *H-factor* will give us a better understanding of how we can apply hypermedia in applications. In the end we will look into what developers have to consider when implementing hypermedia in applications. This makes up the foundation of our prototype that will be presented in the next chapter.

3.1 Representational State Transfer

Why is REST suddenly such a hot topic, and where does hypermedia become relevant for this abbreviation? REST were first introduced by Roy Fielding in his PhD dissertation[17] and stands for **Representational State Transfer**. This term basically grasps around how clients should communicate with servers over e.g. HTTP.

REST is an architectural style that tells us how we should design large-scale distributed hypermedia systems. The Web is a system like that, and one of the most important requirements to that system is *scalability*[19]; it must be able to handle tremendous amount of requests and data over the network. REST takes this into account and defines five constraints[1] that must be followed if the application are to be called a RESTful Web application. We understand that REST makes use of the Web's architectural principles and adds some constraints to it. Following is what constraints Fielding defines in REST[17]:

[1] Also called Fielding's constraints.

3.1.1 Uniform interface constraints

This term is the most famous of Fielding's constraints and grasps around the following constraints:

Identification of resource

Every resource must be identified by a URI. On the Web, every URI is a URL that means that you are guarantied to get a representation of a resource. In RDF you are not obligated to use dereferenceable URIs to identify resources. This will break the constraint. If RDF is to be used in an application, make sure to follow the Linked Data principles[6] and use HTTP URIs, i.e. URIs that retrieves a resource.

Manipulation of resource through these representations

Through the representations of resource state the server sends to clients, clients will be able to manipulate resource state by sending representations to the server. The resource state can be manipulated by e.g. incrementing a counter with one.

Self-descriptive messages

The message that is sent between client and server must provide all necessary information to be able to understand the request or response. This information can either be contained in the message or be linked to.

Hypermedia as the engine of application state

The client can choose between different actions or options from the hypermedia controls that the server provides with the message, which means that the client is responsible for the application state. This state, the application state, can only be changed by the client doing a HTTP request. Based on the response from the server, the client would have to make a new action based on the hypermedia controls again and change the application state. We see that hypermedia is what makes the application change states. An often used abbreviation for **Hypermedia As The Engine Of Application State** is HATEOAS.

3.1.2 The rest of the Web's architectural constraints

Further is Fielding[17] highlighting some of the Web's constraints that enables REST.

Client-server

The communication on the Web is between two sides, one-to-one communication.

Caching

The client can save requests that have to go over the network by using responses from before that is stored in the cache.

Layered system

The system that is used, i.e. the Web, is layered which means that you can put intermediaries between the server and the client, like gateways.

Code on demand

The server can send executable code to the client. An example of this is in HTML where the server can send executable code through the `<script>` tag.

Statelessness

This is quite essential for REST, *state* is, as we know, part of the abbreviation. When talking about statelessness, we are talking about that the interaction between client and server must be stateless. Resource state and application state are two versions of states that can be recorded by the server. Verborgh[53] articulates that the difference between these two states is that *resource* state, that is the state of all the resources contained in an application. These states must be stored by the server to be able to give reasonable responses to requests. *Application* state on the other hand, is where the client is in the interaction with resources, what links it is following, etc, and this is not the server's responsibility. Storing all of this information would require tremendous amount of server space, and the messages sent over HTTP would contain too much data to be efficient. This would definitely not pass the goal of REST to focus on scalability because it would been impossible, or to costly, to maintain a service like that with a lot of users.

Here we have seen a list with explanations of how the Web's architectural constrains are. These constraints are part of fulfilling the Web's four architectural properties *low entry-barrier*, *extensibility*, *distributed hypermedia* and *Internet-scale*, also defined by Fielding[17]. In short we can summarize them with that the Web is easy to use, it is capable of handle extensions like adding resources to it, the presentation and control information can be stored at remote locations and, again, it must scale well. The architectural constraints takes care of realizing this as we have seen.

3.2 What is hypermedia?

Hypermedia is a term that might seem a bit ungraspable at a first glance, but it is closely related to the term *hypertext*. Just like hypertext connects

documents to each other through hyperlinks,

> Hypermedia connects resources[2] to each other, and describes their capabilities in machine-readable ways. Properly used, hypermedia can solve — or at least mitigate — the usability and stability problems found in today's web APIs.[43, p. 45]

Based on this quote, we can say that hypermedia is part of making it possible to describe an environment around the message that is sent. When the client receives a message from the server, the client knows the *capabilities* of the message. These descriptions are also called hypermedia controls, and according to Amundsen and Richardson[43, p. 52] do they have three jobs:

- It tells the client how to construct an HTTP request, what HTTP method to use, what URL to use, what HTTP headers and/or entity-body to send.
- They make promises about the HTTP response, suggesting the status code, the HTTP headers, and/or the data the servers is likely to send in response to a request.
- They suggest how the client should integrate the response in into its workflow.

This makes us understand that hypermedia is all about letting the message contain as much as possible of valuable information, e.g. links to the album if you are looking up the resource of one song on an album. It helps the guiding of the message, it is making promises about response and the added affordance[3] helps out with controlling the workflow.[43, pp. 52-55] It can be seen as a festival, hosted by the server, where clients have to choose between bands to see. The server knows what bands that the client can choose from, while the client decides what concerts it wants to retrieve a representation of.

3.3 Affordance

The self-descriptive nature of hypermedia is part of enabling smarter clients. In a way we can say that hypermedia controls helps the client to understand what the data in the response *affords*. But what is actual an affordance?

> An affordance is a relationship between the properties of an object and the capabilities of the agent that determine just how the object could possibly be used.[41, p. 11]

[2] A resource is in a hypermedia perspective something that has a representation from a HTTP GET request.

[3] This term will be presented in further depth below.

The term was used by Donald Norman[41], in the context of everyday design, wether it is good or bad. Generally, in good design, the affordance is visible, or easy to understand. As an example, affordance are all actions that can be performed on an object, or an resource on the Web[16]. When it comes to hypermedia, an affordance could invite you to POST some data to manipulate the state of both the application and the resource. Another example is that a cup's affordance is that you can hold it and have liquids in it. A typical example of bad design is when you do not understand how to open a door. In such cases are the affordance missing.

Roy Fielding puts the term in the context of hypertext, and by that also hypermedia, where we understand that affordance provides the message with actions and choices for users to explore:

> When I say Hypertext, I mean the simultaneous presentation of information and controls such that the information becomes the affordance through which the user obtains choices and selects actions.[18]

Hypermedia affordance have four common characteristics in applications that works in the context of a distributed network, such as the Web. These characteristics can work in any combination to produce different Hypermedia Factors[4]:[3]

- **Mutability** If the affordance supports *mutability*, the client knows that by using that affordance, something will change, e.g. POSTing data. It can also be *immutable*.
- **Presentation** The presentation characteristic of the affordance gives the client information about how the related information is presented, either *transclusional* like a picture or *navigational* like links.
- **Idempotence** This affordance represent an action, which again means a method if the application works over HTTP. GET is a typical *idempotence* action, because you can do one action multiple times without changing the outcome, while POST is *non-idempotence*.
- **Safety** The affordance can either be a *safe* or *unsafe* action. Once again, if the communication goes over HTTP, it comes to methods; GET is a safe action, while DELETE is an unsafe action. This is because deletion can cause side-effects the client can not take account for.

Amundsen[3] makes a point out of that the client is responsible for the presentation of the affordance, which means mutability and the presentation affordance. While the server takes care of the two other, in the way that it handles the requests based on the action. The server needs to be responsible of the idempotent and the safety affordance, to serve a reliable service. Of course this is dependent on the implementation of the server, but if it is implemented right, GET operations are implemented as

[4]See next section.

Links	LE	Embedded Links
	LO	Outobund Links
	LT	Templated Links
	LI	Idempotent Links
	LN	Non-Idempotent Links
Control Data	CR	Read Controls
	CU	Update Controls
	CM	Method Controls
	CL	Link Annotation Controls

Table 3.1: H-Factor table taken from from [1, p. 14]

safe. If a GET operation is implemented as an unsafe operation, where the operation actually adds resources, this will end up in a huge mess where the server can break down in the end. This is also why the client can not be responsible of defining if a POST request should be safe — which it in no cases should be because it can affect the server.

3.4 Hypermedia Factors

Mike Amundsen[1] defined nine different affordances, that in the context of hypermedia is called the Hypermedia Factors — or H-Factors. See them in Table 3.1. When constructing hypermedia messages, which is a design process, you have to concider how the requested data should be represented. This includes the metadata about the data and metadata about the application as well. This metadata could state, e.g. what format the data should be on when sent to the server for storage. The way this metadata is represented varies from what format and media type the server responds with. The H-factors are an abstraction of these options and can be applied to all formats and media types.

The Hypermedia Factors are divided into to two groups. Link factors and control factors. The link factors are defining the different kinds of linking interactions between server and client, while the control factors provides us with the support of being able to customize the metadata details. A link factor could be invoked by clicking an image which is provided with a link to another page. A control factor could be to define the HTTP Headers, such as defining what media types to accept.

As we have seen in this section, hypermedia helps computers to understand the message that is sent because it makes it self-descriptive. This means that if the client gets a representation of a person, the representation also contains information about relations to other people as well, like friends or spouse. For a client that gets this representation it will be easy to maneuver to these other relations that also has a representation. The client will, just by getting a representation that contains hypermedia, have a great understanding of the context of the message and become smarter because it is in possession of more knowledge than just one representation that does not lead any way.

The minimum requirement when it comes to hypermedia-oriented approaches, is that the client knows a valid URI where it receive a representation and that the client and the server shares the same knowledge of the hypermedia affordance[3].

3.5 Hypermedia Design Elements

For a client to be autonomous it needs to have a server to interact with. With hypermedia we are a bit further on the road because the client now gets an understand of the environment based on the self-descriptive messages. In this section we are going to look into what design elements that needs to be addressed when constructing the hypermedia messages. Since our application are going to work on the Web, we also look at how to apply semantics, hence Semantic Web.

3.5.1 Base format

All data needs a structured format so it can be processed and interpreted by computers. If data was not leveraged in a structured way, it would have been nearly impossible for machines to understand how to interpret the content. In such a situation every client that would have received a response with data not structured, would interpret the data in their own way. The most common formats to structure data in over HTTP is JSON, XML and HTML. In the Semantic Web it is RDF, or serializations[5] of it.

XML

Extensible Markup Language is a mature format that many technologies supports. This must be one of the main positive factors to consider when choosing a format. Knowing that your chosen media type is supported. How to interpret the format is standardized so using it with application clients will give consistent interpretations of the data.

XML has no native hypermedia controls, so this needs to be added if we are going to use it in the prototype.

JSON

Today, JavaScript Object Notation, or more commonly JSON, is widely used in Web applications. This is because it is supported by JavaScript which again is highly rated by Web developers. The programming language is default used in a wide range of applications both front-end, back-end and databases; Express.js[50], Node.js[22] and MongoDB[39] are some examples.

JSON is built up by objects with key-value pairs. Even though it does not have native hypermedia control, we will see in the next chapter how it can be integrated with RDF to express these controls. It must also be

[5]Serialization format means the chosen format the RDF is translated to.

consider as an option in our prototype because we are building a Web applicaiton.

HTML

HTML is not widely common when it comes to being used as base format in applications. We are more used to see it as the holders of the data that is presented in the Web browser.

The reason why HTML should be considered is because it supports a number of hypermedia controls[1, p. 23]. If used in the context where there is expected output to users through the Web browser, it would be easy to render the results because HTML is what Web pages are built up by. With the other base formats, this rendering is a bigger effort because there must be some kind of mapping between the other structures, in the way of what will be displayed in the different elements.

RDF

We have already explained what RDF is and why it is essential to the Semantic Web. Amundsen[1, p. 23] argues that it is neither a format or a media type, but a standard for interchanging data, based on a triple pattern using URIs. When serializing it to a format we will see in the next chapter that it can work well in the context of a generic, autonomous client.

3.5.2 State Transfer

Another aspect of our hypermedia application is how the client will change the state of the server. This is one of the key aspect of REST, as we have explained earlier, and is the aspect of REST that most do not fully understand when it comes to actually creating RESTful applications[18]. Amundsen[1], once again, guides us through three different terms that describes approaches to client-initiated state transfers.

Read-only

Read-only applications is exactly what you believe it is. A client can only retrieve representations of the data through, e.g., HTTP GET requests. Services on the Web that often have a design like this, are for example online news papers, where a client can only follow links to different articles while no data from the users are gathered by the client and sent to the server of the news paper.

This does not mean that the hypermedia affordance is abandoned. The hypermedia controls can still be present in the message sent to the client as links to related articles, the front page, or even the next page if the article is distributed in such a manner.

However, this is not a suitable variant for our application since we want it to interact in smarter way; it should be able to POST data as well and do actions based on that response again.

Predefined

Another way of solving the problem is that a client might have to initiate valid state transfers by the use of predefined message bodies. These bodies for state transfer must be understood by the client before initiating requests.

A way of solving this, except from hard-coding the transfer bodies the client are supposed to use, is to let the server provide the client with documentation with the set of valid message bodies. The client can then pre-load the documentation so it knows how to construct a valid body for a certain operations. This makes the coding of the client easier.

The documentation sent from the server need to consist of details about how clients can recognize state transfers, how clients should construct valid state transfer requests, with what protocol method to use and how the server should respond.[1, p. 25]

In our application, the server would then needed to provide the client with documentation on how the client can know where the client can initiate a state transfer. E.g. how to create a valid body for initiating the state transfer of booking tickets to one of the concerts. The documentation also need to provide the information on which method that is used and what kind of response the client should expect. As a response, the documentation could say that if there was a successful, e.g. PUT, the client can expect a `200 - Ok`.

Ad-Hoc

In ad-hoc state transfer designs, details about the what a valid state transfer must contain is sent with the hypermedia message.

Clients need to know how to recognize the hypermedia controls, the H-factors, since a message can contain several of them, as well as understanding how to interpret the rules for constructing valid state transfers as the messages appear. Ad-hoc means that the client need to be ready to respond to whatever type of state transfer rules it receives from the server in a response.

Main advantage is flexibility[1, p. 26] since their does not exist a strict rule on how many elements should be contained in a transfer body. In one request their might be needed one element, while in another there might be needed ten. This calls for a complex client because it needs to recognize what elements need what input. For a client that is initiated by humans, this is not a problem because a human can ,e.g., easily divide a name into given and surname, if one action calls for that. An autonomous client would trouble more with that if it only knows the name as one whole part. This problem could to some extent be solved with more semantics, e.g. if the person had a FOAF[6] profile.

Bottom line is that in an ad-hoc design for hypermedia, client applications needs to know how to create messages that correspond with the request body expected by the server. In our case that would mean that, let

[6]FOAF is a project for structuring data about peoples relations.

us say that you needed to log in with credentials to get to the entry point. When trying to GET /parkteateret, then the client would get a response containing information about how to compose a valid request. The request would most likely need to contain a password and a user name or an email address. When receiving the the response containing the entry point there might be additional information about how to POST the preferred genre of music which would take another form than the request body of the credentials.

3.5.3 Defining Domain

At one point we need to define what the domain of the server should be. Domain is in this setting the "application domain, or the problem space"[1, p. 26]. We understand this as what kind of data is the server going to retrieve to clients sending requests to it. E.g. is it a commercial site where you can click links to other merchandisers, or will it solely focus on aiding clients with data about items for dog owners.

In our case[7], one of the domains, is a venue providing data about different concerts in that location. If we look at an example from a Norwegian venue, we could say that Parkteateret[8] provides a list of concerts on their Web page. If you click on the link to a concert, you will get more information about that particular concert, like who is playing, what kind of music the band is playing, what a ticket costs and the exact date and time. This is what we try to simulate in our implementation. Modeling the domain with the right attributes and elements of the problem space makes it easier to understand and make use of the expressed information in applications.

Amundsen[1] presents three categories of domain styles when it comes to how a hypermedia format is directly related to the domain it describes:

Specific

This term is very descriptive for what it means. Domain-specific design concentrate the work of incorporate existing names and patterns of a problem space. This means that if we would have used a domain-specific format for describing a concert, it might have used terms like in listing 3.1.

Listing 3.1: Example of domain-specific design.

```
<concert>
  <concertPerformer>...</concertPerformer>
  <concertStart>...</concertStart>
  <concertEnd>...</concertEnd>
  <concertPrice>...</concertPrice>
</concert>
```

This is a good approach if your application and domain is not supposed to live long, or if the domain space is well-established. In our scenario, a

[7]The prototype will be thoroughly explain in the next chapter.
[8]Their actual web page is http://www.parkteateret.no/.

venue presenting information about concerts, must be seen as a problem space that is well established. This means that we have to consider using a domain-specific style when storing our data on the server.

What comes to mind when looking at the above example, is that there is no kind of semantic expressions of the data. This will definitely abolish the thoughts of using it in a Web application where we in the Semantic Web strive to publish semantic data.

If we give it another look we will see that it have a XML-like suit, which again looks similar to pure RDF[9]. This means that there are possibilities to e.g. add URIs in the element-tags to refer to resources, or add a separate document describes the semantic of the element. Since this comes to light it means that we again can consider using the domain-specific style to represent our data.

But how is it about hypermedia affordance in this design? Our client still need self-descriptive messages to be able to initiate state transfers to the server. "In domain-specific designs, the hypermedia controls have names that relate directly to application-domain information"[1, p. 150] as we can see in listing 3.2.

Listing 3.2: Example of domain-specific hypermedia design.

```
<create href="http://example.com/concerts">
  <concertPerformer>...</concertPerformer>
  ...
  <concertPrice>...</concertPrice>
</create>

<concert1 href="http://example.com/concerts/1">

<concerts href="http://example.com/concerts/">

<delete href="http://example.com/concerts/1">
```

If a client receives a response that looks like listing 3.2, it will understand that it can follow different links which makes the message more self-descriptive, since it knows that `href` means that the following URI can be dereferenced. E.g. if it follows the concerts-element it will most likely receive a list or a collection of concerts and it can create a concert if following the URI provided in the element. We can say that this data provides affordance to the client. Still, it can not be completely sure what lays behind the link at this stage, because there is still little semantic added.

General

As a step away from the specific, we there exist a style that is called domain-general, or generic. This style makes use of more general element terms, which can be used in more settings, and then give the elements attributes that are more domain-specific. This design can look something like listing 3.3.

[9]RDF is built up by triples as we already know. What we refer to here is the elements embraced by < and >.

Listing 3.3: Example of a domain-general format.

```
{
  "concert": {
    "id": "...",
    "performer": {"type": "concertPerformer",
        "performerName": "..."},
    "startDate": {"type": "concertStart", "date": "..."},
    "endDate": {"type": "concertEnd", "date":"..."},
    "price": {"type": "concertPrice", "value": "..."},
  }
}
```

A design like listing 3.3 can be used by more applications, because of the more general terms used in the elements that builds up the format. A performer is not necessarily just a musician playing a concert, it could also be an actress on a stage playing a drama written by Henrik Ibsen as well. The same aspect counts for startDate, endDate and price. With startDate and endDate we can see that both have the attribute date which means that we can re-use attributes in different elements. The more domain-specific attributes is applied in the "type"-indicator.

Even though this approach is more flexible than the pure domain-specific style, it needs more processing, thus more code, to traverse both the domain-general element and the domain-specific type attribute. And we still need to hard-code software to actually make use of it since there is no semantics added to it. Semantics in a format and style like the above example could be given semantic meaning by adding the semantics in a separate document. The above document would then look like a JSON-LD document, which is a format we will present in depth below.

Still, this approach will be a good fit for our application if the domain is not likely to change and has a core set of elements that can be reused.

So, how about the hypermedia affordance in this domain-general design? In this approach we need to define the hypermedia controls with actions they support on a protocol-level[1, p. 151] . This means that if data are supposed to be passed over HTTP, the hypermedia controls should be identified with the actions that HTTP supports.

Listing 3.4: Example of domain-general hypermedia format.

```
{
  "link": {
    "action": "read",
    "href": "http://example.com/concerts/",
    "rel": "concerts"
  }
}
{
  "link": {
    "action": "remove",
    "href": "http://example.com/concerts/1",
    "rel": "edit"
  }
}
```

In listing 3.4 we can see that the response contains two links. One where you can read some data from the URI contained in the `href` property. Following the other link will remove the resource identified by the URI provided. The actions here must be seen in association with the HTTP methods. We can also see that the hypermedia controls in a domain-general design must be integrated in such a way that the controls can be associated with the information which is domain-specific. This principle can be seen in the `rel` property.[1, p. 151]

Agnostic

The last approach for domain styles is domain-agnostic. This is the most flexible variant because of its use of completely generic terms on the elements, e.g. form. If we look at an example in HTML (listing 3.5) it will make more sense.

Listing 3.5: Example of agnostic design (HTML) with hypermedia affordance.

```
<div class="concert">
  <div class="id">
    ...
  </div>
  <div class="performer">
    <div class="performerName">
      ...
    </div>
  </div>
</div>
```

In this example, the only element that is used is the HTML's `div`. In another data format it might have had the name `data`. To make any sense of these elements, we need to use context-setting values[1, p. 28] like concert and performer, and we understand that these values are often more domain-specific.

The positives of this approach is that a design like this does not need a huge set of elements to express a domain. To paraphrase Amundsen[1, p. 29]: The trick is to employ a rich set of decorators to the elements. HTML is already mentioned, and the id and class attributes can be added to almost all the elements in the language. This means that many applications can adopt the format.

The biggest problem with the domain-agnostic design approach is that it might end up becoming too agnostic, so that the different elements does not provide either humans nor computers with sufficiently clear information. This can happen either if the element has too few attributes as decorators, or even too many.

We know that HTML can be given semantic meaning by adding RDFa[25] as decorators. This will lift[10] the applications readability for computers.

[10]When speaking of lifting, we speak of semantic lifting which means that we are adding meaning to the content. The more semantics we apply, the higher we lift it.

If your domain will evolve and the design will be used on different domains then this approach will suit you, because of the flexibility in the agnostic elements that can be used in a variety of ways.

When it comes to HTML and its hypermedia support, the language has native support for hypermedia affordance, that we will see in a later section.

3.5.4 Application Flow

Adding hypermedia to your application can help your messages contain application flow options for the client to choose between. With application flow elements, the client will be able to advance the application, or put in another way, skip states, when wanted. This require more than just links in the hypermedia message, it also needs identifiers for those different state transfers options.[1, p. 29]

Earlier, in the read-only state transfer scenario, we used an online news paper as an example. To extend that and describe the application flow in further depth, we have to bring back the different options we introduced: related articles, next page, last page, home page. The application flow options here can bring the application from one state to another state with the client only following one link. A natural step-wise procedure would bring the application from one page to the next in the article, but here the application state could be changed from page 1, to the last page, in one request. We can now really start to understand why we say hypermedia is all about self-descriptive messages (referanse).

In our prototype a typical application flow option could be to *book* a ticket. Below we will go through Amundsen's[1, p. 30] three different ways of hypermedia to handling application flow.

None

There is not every implementations that need hypermedia that takes care of distributing options to the client about application flow. In those cases we say, logical enough that there exist *none* application flow options.

Amundsen[1, p. 31] gives us an understanding of that this type of design can be used in automated systems. This means that we really have to consider using this approach in our prototype. Application flow options might be seen as "noise" for automated machines trying to reach their goal. Another consideration we have to take in account is if the system also should provide affordance for human interaction, i.e. that the application not only are made for machines, but humans as well. This is something we already touched upon and which is one of the main goal for the semantic web[9].

Intrinsic

On the other hand, if the application could benefit from having hypermedia that supports application flow, a design possibility is to use a format that

have this feature integrated. By that we mean that there exist identifiers for such application flow in the media type itself. If you design your own media type Amundsen[1, p. 31] states that this can be done by either identifying specific elements that represents application flow options in your design, like `<book>...</book>`, which can resemble a XML-like format; it can be achieved by identifying specific attributes in the same way, and this could look something like `<enter type="book" >...</enter>`; or at last, it can be reached by using decorators, as in HTML, that you identify to represent an application flow and apply to elements and attributes. This approach could look like `<div rel="book".../>`. Using application flow embedded in the format like this is called *intrinsic*.

An implementation with an intrinsic application flow can work well when you want your application to avoid relying on external definitions or specifications, and if the application flow can be expressed by a limited amount of elements and attributes[1, p. 32]. The reason for this is that if you apply a lot of different elements or attributes to identify application flow, it might accelerate the level of how difficult it is to read and understand for humans. With the limited set of elements and attributes, the intrinsic variant should be used in general use cases where the application flow does not change often.

Applied

Applied hypermedia design should be considered in situations where application flow can evolve over time, or can be used in various scenarios. This means that the server can, in various situations, respond with the same data, but provide a client with different kinds of external documentation depending on the use case. In the external documentation, there should be provided information about the application flow, of course, and how it should be interpreted as well. This document should be provided with e.g. a link in a hypermedia response in the the link header if we talk about HTTP requests.[1, p. 32]

The positives with the applied design for application flow is that it is flexible because of unlimited amount possibilities, independence and can be applied to any base format. It also enables evolvability to the media format because of because it is completely independent from the server.

Downsides are that clients need to control not only the main media type received, but also a second document it have to interpret. This calls for more complexity to the clients. With rapid changes in the external document, or that there is changes on the server itself, it may cause many invalid requests from clients.

In HTML, the relation from the main HTML document to the styling in the CSS file, is an example of how applied hypermedia documents can work. To state an example from our own prototype we could have supplied the code with a `meta` tag, if our chosen media type where HTML, the applied document could be related in the response in a manner that looks something like this: `<meta name="profile" content="http://example.com/concerts.html" >`. The important take

away is that the applied document with the application flow must be reachable in an intuitive way for the client.

Summary

In this chapter we have been introduced to REST, and how to actual be able to create truly RESTful service. To achieve this we need a complete understanding of the HATEOAS constraint, or hypermedia as the engine of application state. We got that by examine the term hypermedia, where we also introduced affordance and hypermedia factors. All over, we got the understanding that hypermedia helps the client to understand what the data in the response from the server affords. This again is done through different hypermedia controls, or Hypermedia Factors. In the last section we dived into what needs to be considered when designing these hypermedia messages; what format is suitable, how should state transfers be expressed, in what way should the domain of the application be structured and how could the server provide the client with an understanding of the aplication state.

Chapter 4

Prototype

We have so far been going through what the Semantic Web are and what technologies enables the visions of it and its capabilities. Throughout this chapter we will show and explain how to use these technologies to develop a prototype. This will help us getting a better understanding of what we are dealing with.

The aim of this chapter is to bridge the understanding of the technologies we took a look at in chapter 2, and to show how they can be used in a practical example. The implementation of the client-server application will lead us to the research of how autonomous a client can be, based on the message that is sent from the server. We will also explore if the client can be generic by let it consume two different APIs. This is essential to see that hypermedia and REST enables reuse of code.

This chapter can be seen as four different parts. At first we will briefly get a general overview of the application that has been developed. Second we will see how the hypermedia design elements have been implemented. Third we look at the client, how it was developed and how it functions. Lastly, we look at other considerations and reflections on the implementation of the prototype.

4.1 General overview

We are going to develop a client that can work with multiple servers, consume their APIs and relate it to its own goal. More precisely are we going to develop a client/server application where the client works autonomously. To be able to work autonomously, the client needs a goal, to achieve the goal it must vary from server to server. In this prototype we have developed two different servers with completely different domains and the generic, autonomously client.

4.1.1 Concert server

The first server is the concert server. This server hosts data for a concert venue. A concert's information contains what band are playing, what it costs, how many tickets are left and what genre it is. In this scenario the

client will connect to the server, and based on some predefined input, it will try find a suited concert and obtain a ticket.

4.1.2 Physician domain

The other server hosts data for a physicians office. Here a client can register a patients medical condition, i.e. what kind of disease or injury that is needed treatment, and look at what physicians are connected to the office. In this scenario will the client try to book an appointment with a physician based on some predefined input. Here must the physician be available and have the right medical specialty for the appointment to be booked.

4.1.3 Intelligence

What is worth noting is that all *intelligence* is possessed by the client. For both scenarios. It will be the client's job to understand if a physician is not available or if it does not have the required medical specialty. The only smartness the server possesses is to check if the request is valid.

This is all according to Verborgh's[52] visions for enabling smarter clients. We are going to try to make the environment enable the client to act smart. Such an approach fits well with REST and the Web's scalability since the server is offloaded work.

4.1.4 Programming language

The programming language that were used throughout the whole prototype was JavaScript. We have been arguing that in Web applications, JavaScript have become popular through that a lot of external projects is based on the language.

4.1.5 Server platform

As the core of the server we are using Node.js. "Node.js is a platform built on Chrome's JavaScript runtime for easily building fast, scalable network applications."[22] Together with Express.js it enables us to easily set up a server that hosts the service.

4.1.6 Web framework

On the server side, on top of the Node.js distribution we are using Express.js which is a "Fast, unopinionated, minimalist web framework for Node.js"[50]. It helps us with easy routing and handling of the requests from the client.

4.1.7 Code

The prototype has been continuously developed throughout the last twelve months, and the code has been stored in a GitHub repository. It has undergone major changes from the first commit, which this chapter will

reflect to some extent. See appendix A for further details on were to find the code in its entirety.

4.2 Prerequisites

As we got to know in chapter 3 about Hypermedia, what we definitely need in our hypermedia-oriented approach, is that the client we are developing knows a valid URI which leads it to what we will refer to as the *entry point*. The entry point is basically a representation of the underlying data on the first page you see when requesting a Web page. This could typically be the index-page if we put in the perspective of regular commercial sites.

The second thing we need, is the shared knowledge between the client and the server of the hypermedia affordance.

Lanthaler[33] introduces three steps of how it is common to make an API.

1. **Define URL structures**
2. **Expose the objects as JSON**
3. **Write API documentation**

However, he does not believe that this i a RESTful way of making an API. Therefor, in the companionship of Hydra and JSON-LD he introduces us for these new steps:

1. Do not care about defining URL structures. This is a matter for providers, not consumers.
2. Expose you objects as JSON-LD. When serializing your RDF into JSON-LD we get the advantage that also machines can understand the content. We know that the LD in JSON-LD stands for Linked Data, and therefor it is relevant to look at the guidelines for it. Lanthaler means that we should use URLs instead of URIs so it can be dereferenced. With the JSON-LD we also introduces self-descriptive messages with the `@context` tag.
3. Instead of writing external API documentation, we are going to use Hydra, the lightweight vocabulary for describing APIs.

We will see through this chapter that this is indeed the the way we have implemented our API.

4.3 Base format

Our chosen base format for data that is to be sent from the server to represent different resources, is JSON. It provides an well-known structure for most Web application developers, and since we are using various JavaScript libraries and technologies it makes sense to extend that line with

JSON as it works well with the programming language. The question is if we are capable of extending the JSON with the necessary semantics and hypermedia controls so that a client can consume, understand and interact with it autonomously.

4.3.1 JSON-LD

Our selected hypermedia type for the prototype were JSON-LD[35, 48]. As we have understood, does it exist several other hypermedia types that builds upon plain JSON.

One of the reasons we chose JSON-LD was because it is a serialization of RDF[35], which means that if we can express hypermedia controls with this media type, we are then using hypermedia RDF in the prototype. The integration of LD in the type also emphasizes the relation to RDF and thus is perfect for the Semantic Web since we are linking resources together and expressing meaning. How this is done will follow. Other serializations of RDF could also been used, but we will see that JSON-LD and Hydra is a good combination for expressing hypermedia controls. Another benefir of JSON-LD is that it enables self-descriptiveness both for humans and machines[35].

The rest of the thesis will then give us an answer to our research question, simplified; if a client can be both generic and autonomously, and solve its goals based on the JSON-LD responses the server provide it with.

So how does JSON-LD function, how is it structured and how can we express semantics and hypermedia controls with it?

4.3.2 From JSON to JSON-LD

A regular JSON object could look something like the listing 4.1.

Listing 4.1: A plain JSON representation.

```
{
  "name": "Led Zeppelin",
  "startDate": "2015-06-06T20:00",
  "genre": "Rock",
  "price": "249.00",
}
```

For a human, it is possible to understand what this is a representation of. Especially since it is a well-known band that performs; it is a representation of a concert will many resemble. For a computer is this ambiguous in best case. It can only understand that it is a JSON object. To apply semantics to the JSON documents, JSON-LD have introduced the `@context` keyword.

Listing 4.2: A beginning of a JSON-LD representation.

```
{
  "@context": "http://schema.org/"
  "name": "Led Zeppelin",
  "startDate": "2015-06-06T20:00",
```

```
  "genre": "Rock",
  "price": "249.00",
}
```

The context property is given a URL which is a documentation of the meaning of the property when applied to the other properties[35]. In this case is the URL in the context functioning as the base, which means that it is applied to the other properties. The ambiguous property *name*, becomes the unambiguous property http://schema.org/name.

Usually, when speaking of RDF and what it is describing, their is a class defined. In the example in listing 4.2, a computer can not be sure what the representation is about. This is because a http://schema.org/name can be a various amount of things.

If we look at the documentation, we see that the described property is used on the type *Thing*. We know that Led Zeppelin is a band, while a computer with the above descriptions does not know. Another downside is that there is no information in the representation on where to access it. This means that we need to be more specific if we want to apply this response in a application for autonomous clients.

We need to first introduce the way of identify the given resource. That is done by having a working URI as the value of the `@id` property. The way of expressing what kind of class is done with the keyword `@type`.

Listing 4.3: A full JSON-LD representation of a concert.

```
{
  "@context": "http://schema.org/",
  "@id": "/rockefeller/concerts/1",
  "@type": "MusicEvent",
  "name": "Led Zeppelin",
  "performer": {
    "@type": "MusicGroup",
        "name": "Led Zeppelin"
  },
  "startDate": "2015-06-06T20:00",
  "genre": "Rock",
  "offers": {
        "@type": "AggregateOffer",
        "offerCount" : "15",
        "price": "249.00",
        "priceCurrency": "NOK"
  }
}
```

Listing 4.3 is a representation that can be retrieved from the URI /rockefeller/concerts/1, and is a representation of a http://schema.org/MusicEvent. The performer of the concert is of the type *MusicGroup*, and the name of the group is Led Zeppelin. By adding these property keywords that is related to dereferenceable URIs with documentation, we get the semantics both for humans and machines incorporated in one document. Mark that the URI is *relative URI*, which can be used if the base URI is

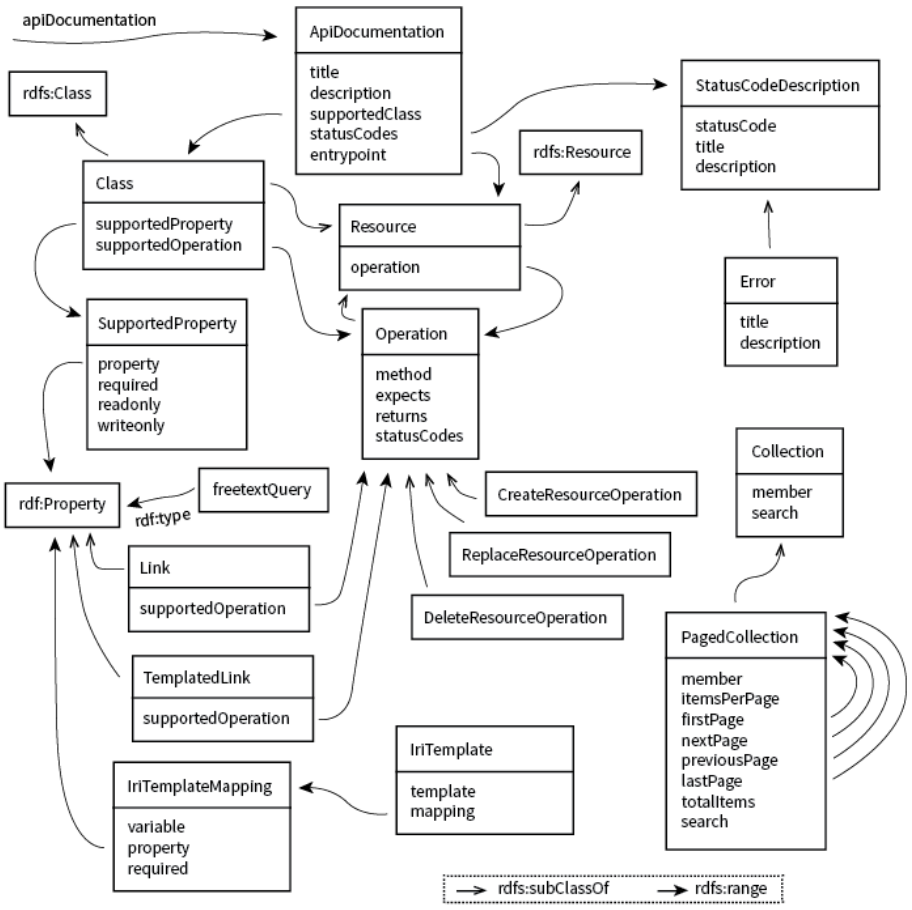

Figure 4.1: The Hydra vocabulary.

known[10]. In this implementation is the base http://localhost:3000 which the relative URI is applied to when dereferencing the URI.

4.3.3 Hydra

We now have the semantics, but were are the hypermedia controls? In listing 4.3 we can see that it affords us to follow the URI that identifies the resource, or the URIs in the `@context`. Hydra[34, 31] is a lightweight vocabulary that we will use together with JSON-LD to be able to afford more hypermedia controls so that the client can understand how to create HTTP requests to modify the server's state. Hydra will work as the shared knowledge between the server and client where the client can look up what different URIs means. This knowledge is discovered at runtime.

An overview of the core elements of the vocabulary can be seen in figure 4.1. The `ApiDocumentation` represent the possibility it gives us to document our Web API[34]. It enables us to define a main entry point[1], which in our

[1] All URIs should be an entry point. By this we mean that all information that is needed should be provided in each response, whatever URI you follow. In this setting will the main entry point be similar to a defined starting point.

case is /rockefeller for the concert domain and /physician for the physician domain.

Further, it enables us to document what classes, properties, and operation the API supports. The two different API documentations, or from now on Hydra APIs, can be retrieved when running the projects from /rockefeller/vocab and /physician/vocab. The reason why the API is called a vocab, or vocabulary, is because that was the common trend at the point of the implementation. API could have been used as well. Vocabulary is not wrong to use since the Hydra API in its essence is a vocabulary, or an ontology, were the terms used in the application is described.

Listing 4.4: Hydra description of a class.

```
{
    "@id": "schema:Physician",
    "@type": "hydra:Class",
    "subClassOf": null,
    "label": "Physician",
    "description": "A physician",
    "supportedOperation": [
      {
        "@id": "_:retrieve_physician",
        "@type": "hydra:Operation",
        "method": "GET",
        "label": "Retrieves a Physician entity",
        "description": null,
        "expects": null,
        "returns": "schema:Physician",
        "statusCodes": [ ]
      }
    ],
    "supportedProperty": [
      {
        "property": "schema:name",
        "hydra:title": "name",
        "hydra:description": "Name of the physician",
        "required": true,
        "readonly": false,
        "writeonly": false
      },
      {
        "property": "schema:medicalSpeciality",
        "hydra:title": "medicalSpeciality",
        "hydra:description": "The physicians medical
                             speciality",
        "required": true,
        "readonly": false,
        "writeonly": false
      }
    ]
}
```

Listing 4.4 shows us a description of a class using Hydra. It describes

a http://schema.org/Physician, were you can do a GET operation[2] were you can expect a http://schema.org/Physician in return which maybe does not make very much sense right now, but we will see how this is mapped to the resources below. Further, we see that a http://schema.org/Physician have two properties; a http://schema.org/name which is required and a http://schema.org/medicalSpeciality which also is required.

The API mapping

How to apply Hydra to our existing JSON-LD can vary from each use case. In some scenarios it can make sense to apply it in the JSON-LD representation itself. Our choice were to have an external document were all the hypermedia controls are defined.

Each resource in our application have a `@context` property according to the specification of JSON-LD[48]. The value of the property is in our case a dereferenceable URI, which leads us to another resource were the context is defined. This can be seen in listing 4.5.

Listing 4.5: Snippet of concert representation with external context.

```
{
  "@context": "/physician/contexts/Phyisican.jsonld",
  "@id": "/physician/physicians/1",
  "@type": "Physician",
  "name": "Marc van Overmars",
  ...
}
```

By dereferencing the context URI, we will meet the resource that defines the semantics of the properties in the received response. What we see in the listing 4.6 is that here is it a property called `vocab`, which value is a dereferencable URI as well.

Listing 4.6: Context example.

```
{
  "@context": {
    "hydra": "http://www.w3.org/ns/hydra/core#",
    "vocab": "http://localhost:3000/physician/vocab#",
    "Physician": "http://schema.org/Physician",
    "name": "http://schema.org/name",
    "medicalSpeciality":
                  "http://schema.org/medicalSpeciality"
  }
}
```

It is here we see how the documentation is mapped with the resource retrieved from the server. The `@type` we saw in listing 4.5, is defined in the context in listing 4.6 which gives it the semantic meaning, and further it is defined in the API, or the vocabulary in listing 4.4 were we see the hypermedia controls. We are now approaching what Verborgh[53] defines as *semantic hypermedia* which is hypermedia following the LD principles.

[2]The operations specified in Hydra corresponds to the HTTP methods.

4.4 State transfer

We have so far in the implementation seen what format our data is in, and how we have applied hypermedia controls to it. As we now know is one key aspect of hypermedia applications how the state is changed by the client.

With the help of Hydra, the prototype uses a predefined way of stating valid state transfers. This is because Hydra enables the server to share a vocabulary for what is valid state transfers to the client[34]. With this predefined documentation the client then can construct valid HTTP requests and drive the application state to achieve its desired goal.

As we did see, this documentation is accessible in both the prototypes vocabulary. When looking at the listing 4.4, we see that it describes a state transfer with the GET request. In that request there is not needed anything in the request body because it states that the request `expects: null`.

A huge benefit with Hydra and the predefined, shared knowledge between the server and the client is that it enables decoupling[34]. This is because the documentation on how to construct the valid state transfers is designed in a machine-interpretable way. The computer can then consume the information at runtime and not being hard-coded to work with the documentation at design-time. The client, server and the documentation can *evolve* independently.

4.5 Domain

We have now reached the step where we need to decide how the domain should be represented in the responses from the server. Since our application, does not have to be agnostic, we are choosing the a design style that resembles the domain-general one. The reason why we do not need to be agnostic is because that we have a defined problem domain, the concerts, and we will use attributes that are associated with an event like that. The same for the physician domain.

The reason why we ended up with the domain-general is because it gives us the flexibility to be both general and specific. We can be general when defining e.g. collections of concerts, and then be more specific when we add semantics to the information contained. Our goal is to be as generic as possible, and that is one other reason that we chose JSON-LD as format. JSON-LD is perfect for us in this situation; we can use JSON-LD as generic media-type[35] with the use of general terms like collections, and it provides us with the possibility to be specific by defining the semantics in the context. The generic nature of JSON-LD means that it can be applied to any domain.

JSON-LD, with its generic appearance, enables us to focus on defining the set of legal interactions; the domain application protocol[35]. In our prototype is this done in the Hydra API, which means that we have built up our own universe of discourse, or problem space, with a vocabulary. This approach is exactly how the Semantic Web are functioning, and as

such are we expanding it by publishing data on the Web in a semantic way, and we can further make use of other Semantic Web tools if that is wanted.

4.6 Application Flow

With the Hydra API, we understand that the prototype is developed with an applied application flow. Within each response from the server, we provide the client with a link to the Hydra API in the HTTP link header. This means that the client for each response it gets, it can look up in the API and understand what it can expect from each link that is present in the representation.

4.7 The autonomous, generic client

So far, we have seen all that enables a client to be smarter. But how do we implement the actual client to interact with the hypermedia affordance? In this section we will describe the different parts of the client and how they makes use of the affordance to achieve its goal.

In both implementations are the client related code found in the folder `public/javascripts` and contains the following elements:

1. a goal
2. the generic client code
3. domain-specific code
4. other helping modules like jsonld.js

4.7.1 The goal and other predefined input

First of all, the client can not be completely autonomous because we have not implemented artificial intelligence, and the client can therefor not come up with its own goal or know what information to feed the server with.

The goal

The goal on the concert server is to obtain a ticket based on preferred taste of music amount of tickets and the maximum price you are willing to pay. While on the physician server we would like to obtain an appointment with a physician based on the relevant specialty to the medical condition of the patient and preferred date.

Both goals are defined as a JSON object, and can be reviewed in listing 4.7 and 4.8.

Listing 4.7: The goal for the client working on the physician server.

```
{
  "@type":
    'http://localhost:3000/physician/vocab#Appointment',
```

```
  "startDate": "2015-02-08T10:15",
}
```

Listing 4.8: The goal for the client working on the concert server.

```
{
  '@type': "http://localhost:3000/rockefeller/vocab#Ticket",
  'genre': "Rock",
  'offerCount': 2,
  'price': 500,
}
```

Predefined input

In the concert implementation there is no need for other predefined input that the goal, but for the physician example, a registration is needed; by that we mean a patient that needs treatment. Such a registration, in this implementation needs a name, a cause for the incident and relevant specialty which will be checked to see if there is a doctor available to see the patient. All of those terms are based on http://schema.org/.

4.7.2 The generic code

Based on the provided goal, and a given main entry point, the client code start doing its job. The generic client is found under `public/javascripts/generic-client.js` in both projects.

This prototype is developed as a Web application, so to start the project, you need to start the server, and direct your Web browser to localhost:3000 and then the client starts running. The Web browser then works as one of the components in a REST application, and is the agent for the client code[19].

If the client works on the physician server, the provided main entry point is localhost:3000/physician, while for the concert server it is localhost:3000/rockefeller.

The work flow for the client is then as following:

Discovering the API

As recommended in the Hydra specification[31], for each response from the server there is a defined a link header with a relation to the documentation that looks something like the following:

```
Link: <http://localhost:3000/physician/vocab/>;
rel="http://www.w3.org/ns/hydra/core#apiDocumentation"
```

What the link header tells the client is that by following that link, it will retrieve a Hydra API documentation, and as a Hydra driven client it understands this term. More precise, it needs the Hydra documentation to function.

Relate to its own goal

The client have now obtained the documentation of the API. As a helper for making use of the API, we have borrowed[3] a module for structuring the documentation making it easily accessible for the client. In our prototype is this code found under `public/javascripts/hydra-core.js`.

The Hydra API is basically a large JSON-LD object, and what the the hydra-core module is used to, is structure that object and provide it with some functions to easily look up wanted information. An example is that it provides us with the possibility to find a wanted class without our main code needed to loop through.

Another essential part is that it maps the resources identifiers that is present in the application to where it is described in the API. E.g. when the hydra-core module receives the URI from the main entry point, http://localhost:3000/rockefeller, it finds the API documentation through the HTTP Link Header in the response and starts the structuring and mapping. In the API is it specified a `hydra:Link` from the entry point, which got the identifier (IRI) http://localhost:3000/rockefeller/vocab#EntryPoint/concerts. This is the IRI to the description of the resource that is identified by this URI: http://localhost:3000/rockefeller/concerts/.

So, when the Hydra API is mapped with the identifiers in the application, we then finally can start reaching the clients goal. The is defined as a JSON object with a `@type`. For the client to know if it is possible to reach such a goal — to obtain the right class — we search the mapped API for the wanted class. If it is not present, the client will stop the execution.

On the other hand, if it is present the client, the client will look to the API an see what is needed in a request for obtaining the wanted class. In the concert example we see that it is needed a related concert of the `@type` *MusicEvent*, a total price with the range of `@type` *Number* and the number of wanted tickets with the range of `@type` *Integer*[4].

The next step for the client, is to start explore the rest of the API, gather the right information and obtain that ticket.

Start exploring

When the client is looping through the API to try to solve its goal, it always conferences with the mapped API.

In its essence, what the client code does, is that it for each targeted value, that is the values needed to create the request to reach its goal, starts at the main entry point, and explores it. By exploring, we mean that it follows all of the possible properties that is a dereferenceable URI. E.g. on the concert server there is only one property to explore, while there is three in the physician example.

[3]The project the code is borrowed from is found at https://github.com/bergos/hydra-core.

[4]Bear with us that the way this is solved might not be the most logical or appropriate way. It is meant as an example to show that the client can work autonomously.

For each of those properties, the client checks what operations it can do, and further, what it can expect as a response. We are here using IRIs as identifiers, because this identifier leads us to the description of the resource in the API. The `@id` is the identifier which is a dereferencable URI and identifies the actual resource.

If the returned value is what the client is targeting, we execute the operation[5]. If the returned value is a `Collection`, it needs special treatment[6]. And finally, if the returned value is something we do not look for, go to the next property and check its operations. Ideally, the client could have gone deeper, followed the links and seen what laid behind this first step, but in this prototype we know that there is a maximum depth of three[7] and we know that to reach depth level three, you need to retrieve a collection first. Anyway, we see a clear pattern here that the client is following its nose, but in a bit more sophisticated manner, since it now can make up its mind and make itself known with the environment to the received resource based on the Hydra API.

Do operations

So, if the operation we are about to execute returns the value we are targeting, e.g. a *MusicEvent*, the client is ready to execute the operation on the associated URI.

The client then first need to check what operation is to be performed. If it is a GET or a DELETE it can just go on and perform it. If it is a PUT or POST, then it needs a request body. This is checked in the provided domain-specific code. After finding the right request body, then the operation can be executed.

When the client receives the response, it performs a compaction with the given API documentation of the response according to the JSON-LD compaction algorithm[37]. This gives the client a response with full URI identifying the properties so that the client can use this in a unambiguous way when looking for its goal. When the client receives a response, it returns to check if the final request can be executed. If not, it starts looking for the next needed element to the request.

Handle collections

If the target of the client is not a `collection`, the we need to look into what it contains. This is one problem with the `public/javascripts/hydra-core.js` code, that we do not see the `members` without the actual resource.

Therefor, when an operation returns a `Collection`, we need to follow the URI, and see what `@type` the `members` have. If they are what the client

[5] See section Do operations.

[6] See section Handle collections.

[7] We define the depth by each resource layer you follow. The main entry point is one, if you follow a URI from here you are on depth level to, and if you follow one URI from the depth level two, you are on level three.

is targeting, then we GET each and one of them, compact them and checks with the domain-specific code if there are any necessary processing needed to be done. When the process is done, the client checks if the final request can be done, if not, find the next target.

4.7.3 The domain-specific code

The domain-specific code is were we handle everything that is specific for the application domain. The code is found under `public/javascripts/domain-specifics.js` in both projects.

For example, if we look at the physician server; for a generic client, it is possible to understand that it needs a *registration*, a *date* and an *physician*. The problem arises when it comes to understanding that the physician have to be available at the wanted date, and need the relevant medical specialty. This must either be stated in an external document with rules, that again needs to be processed by external code, or let the server provide the client with domain-specific code.

After conferencing with Ruben Verborgh, the author's co-supervisor, we decided that in this case we could let the server provide the client with domain-specific modules. Ideally, the client could have discovered the code through, e.g a HTTP Link Header when the code was needed, but in this prototype we have taken a shortcut and the whole code is provided at runtime.

The main benefits of using an external rule set, written in e.g. the RDF serialization N3, then you can reuse components, like the EYE reasoner[44]. Still, some glue-code would most probably been used to get it work with the client, but as for our case, where we want to show the capabilities of hypermedia, the modules provided by the server works.

What the domain-specific code does is that it takes care of the special cases when it comes to what the client is looking for. When wanting to obtain a ticket for a concert, a generic client can not know that it needs to check if the wanted *genre* fits. so, when the client receives a response with a representation of a concert, it sends the response to the domain-specific code which checks if the concert fits the user input. If it does not, it keeps on looking, if it fits, it tells the client that the concert is good and that the final request can be made.

4.7.4 Other helping modules

Other helping modules that were used, were

- `public/javascripts/jsonld.js` which is an implementation of the JSON-LD specification[37, 48]. This file is gotten from `node-modules/jsonld/js`.
- `public/javascripts/es6-promise.js` which helps us handle asynchronous JavaScript calls. It helps us state that *this operation is not done yet, but we expect it to be in the future*. It is the file from the `node-modules/es6-promise/dist`.

- `public/javascripts/jquery.min.js` jQuery helps the application more easily work with browsers, i.e. manipulate the HTML[21]. This is mostly used when we see the output from the client.

These helpers are usually loaded through the browser and stored in the memory as long as you are on a particular Web page, but we decided to store it on the client side because some of the work were done with limited Internet access.

4.8 Issues

When developing the prototype we faced some issues. Some of minor effect on the development, others with bigger impact.

4.8.1 Rapid spec changes

In the beginning, it was the rapid changes of the Hydra specification. This made it a bit hard to decide how to use the specifications. At one point we had to move on and stick to what we got. In the end it looks like it is not that big of a difference in our use of terms and what the specification looks like today.

4.8.2 Where to use domain-specific knowledge

The fact that the author struggled understanding how the client could be 100% generic was a huge part of the process, so it is worth mentioning again. The way it was solved have already been discussed, so that is not going to be brought up again.

4.8.3 Asynchronous calls in JavaScript

When the client were handling the collections, it needed to GET each of the members. With the asynchronous nature of JavaScript, this caused us some troubles; the client finished all it checks in the search of constructing a valid HTTP request to obtain the wanted goal, but since the client uses `jQuery.ajax` to perform the HTTP request, the response from these request came too late so the client were done executing and ended up not being able to solve its goal.

To solve this we used both promises[8] and we forced the execution of the code to be sequential. The properties, the operations and the members of the collections were structured in arrays. So for each element in the arrays, the client had a promise that the operation would finish, and it waited until the operation were finished before it moved on.

[8]See the `public/javascripts/es6-promise.js`.

4.9 Other considerations

Throughout the development several files were started coded, but not everything ended up being in the final prototype. One of those were the integration with a triple store. Something that on the other hand still is in the prototype is some of the attempts to create a fully generic client. As already discussed, when understanding that the client needed some help by domain-specific knowledge, the `public/javascripts/generic-client.js` was created. This is the actual working code, while the other clients just illustrates the different attempts.

4.9.1 Storage

So why is there no storage in the prototype? First of all, the prototype is just a way of showing the theories behind hypermedia and how the client can work on its own and handle different APIs. And that is the essence of the prototype, how the client handles the hypermedia affordance. This means that the server side handling of the data is not that important. The server responds to requests with plain, predefined JSON-LD files.

Early in the development there were some experiments with triple stores and SPARQL integration. But as we decided to go for a implementation with JSON-LD, a MongoDB[39] integration were halfway developed. A MongoDB makes sense to use when handling data on JSON format. Another positive is that SPARQL endpoints' availability is rather limited with its downtime on 1.5 days per month[56].

These parts are now deleted from the project because this was not the essence of the final prototype.

4.9.2 RESTdesc

Early in the development there was an intention to create two different clients to compare them with each other, which used a different way of describing the API. The intention were to use RESTdesc[9], but when we realized that the task were to big to complete both implementations, we decided to go with Hydra based on the reasons already given.

Summary

In this chapter have we seen how to develop a Web application that we will argue is truly RESTful. We have gone through the steps on how to turn regular JSON into a format that makes use of the Linked Data principles — JSON-LD. This format together with Hydra makes us capable of the expressing self-descriptive messages so that the client can work autonomously without human interaction. How this client were developed and functions were reviewed as well.

[9]For more information see next chapter. There we will use these descriptions in comparison to Hydra.

Chapter 5

Exploring view source capabilities

In the previous chapters we have illuminated and demonstrated how we can enable machines to take a meaningful part of the Web, just as Berners-Lee visioned.

One aspect we have not highlighted yet, and that is often neglected when working with the development of specifications, is the readability of the specifications and the technology. Other developers are to read the specification, understand it and use it in their own projects. If this aspect is neglected, bright and revolutionary ideas and creations might not be used, because of its fall when it comes to being understood. To let developers easily make use of the technologies that enable clients to work autonomously, the technologies must be accessible and understandable.

In this chapter we are going to present a study of the view source capabilities of Hydra compared to RESTdesc. The Hydra vocabulary aims to enable the creation of APIs that contains both human-readable and machine-readable information. By having all information in the API, a developer is supposed to not need to look up external documentation. RESTdesc were chosen as comparison because it is also a quite new technology and it also describes APIs.

To validate these capabilities we were having a workshop to gain useful data for this matter.

We will first introduce the preparations for the study, before we take a look at the participants. After that we are presenting the material that were provided to the participants and how the study was carried out. Before we look at the what flaws the study had, we will see the results.

5.1 Preparations

The first thing that needed to be settled before executing the study was to choose in what form the study should be performed in. In our case it made most sense to have a workshop. The reasons for this was that in a workshop, we could provide the participants with tasks and afterwards have a discussion around how they found working with the different APIs.

To prepare the workshop, a workshop guide, similar to an interview guide, was created. You can see the steps of it in the section The workshop and methodology. A workshop guide contains the different steps of the workshop that is going to be carried out. An example could be *step 1) introduce yourself, your study and tell the participants why they are here*.

Another aspect that we needed to keen in mind before executing the study, was to choose what kind of data we wanted to gather. In our case, in the context of a workshop, it made sense to gather qualitative data. The goal of the study was to understand if the self-descriptive nature of Hydra and RESTdesc was satisfying for the participants. In such cases, numeric values from *1* to *5* can give meaning, but we feel that the discussion were the participants can tell us what parts were *good* or *bad*, or what was *easy* to understand were more valuable in this case.

Before holding the actual workshop, a pilot workshop was carried out with one person. This was done to make sure that everything that was presented during the workshop would make sense, and to prepare the workshop leader, i.e. the author of this thesis. The feedback from the person in the pilot workshop is also reviewed as one of the workshop participants.

5.2 Participants

All of the seven participants of the study were master students at the University of Oslo, writing their thesis at the Department of Informatics. Six of them finished their degree in 2015, while one is scheduled to deliver in 2016.

The participants were all volunteers assembled from the workplace for the master students. They varied in background within informatics, were three had a bachelor within interaction design — all of them continued on that field in their thesis. The rest, four of the participants, came from a programming directed bachelor. Two carried on with that on their master, while the two others entered their research within global infrastructures.

Two of the participants had little or no experience with JSON. One had some experience working with triple based formats, and one other had some general knowledge about semantic structures.

All of the participants have had two or more courses in programming throughout their studies.

28.6% of the participants were females and the average age were 26.6.

Since they have different knowledge and varied experience with both programming and in the owrk with the formats they were going to explore, it is interesting to compare either how different their experience are with the APIs, or how similar.

5.3 Material

The material that was used in the workshop and handed to the participants were:

- **Hydra API** What this API describes is the domain were an appointment with a physician can be booked. The complete API can be found in appendix B.
- **RESTdesc API** This API describes how a concert ticket can be bought. The complete API can be found in appendix C.

Both of the APIs carried out other information as well, like how to retrieve a representation of a concert or how to register a medical condition. Since we in the prototype had goals for booking a concert ticket and schedule a physician appointment, this is what we have highlighted here.

To get a grasp around what the participants had to deal with, listing 5.1 gives an example on how the operation of retrieving the entrypoint is described in the Hydra API[1].

Listing 5.1: Example of how a GET operation is described in the Hydra API.

```
"supportedOperation": [
  {
     "@id": "_:retrieve_collection_appointments",
     "@type": "hydra:Operation",
     "method": "GET",
     "label": "Retrieve appointments",
     "description": null,
     "expects": "null",
     "returns":"http://www.w3.org/ns/hydra/core#Collection",
     "statusCodes": [ ]
  }
]
```

To be clear, listing 5.1 tells us that we have something of the type `hydra:Operation`. By now, we know that it can be translated into that it is a HTTP operation, more precise a GET method. When we read the label and what the operation returns, we understand that we are GETing a collection of appointments. By looking at this example, we do not know anything more, like what kinds of appointments. If the context of this particular operation were to be given, as we can see in listing 5.2, we would hopefully understand more. When given this extra information, it would be interesting to see if the participants understands that the operation invites to dereference a URL. There are specifics that might be looked up in the Hydra specification, but *everyone* understands what a *link* is; on the Web it is something that brings you to another Web page.

Listing 5.2: Example of how the context of an operation is described in the Hydra API.

```
"property": {
  "@id": "vocab:EntryPoint/appointment",
  "@type": "hydra:Link",
  "label": "appointment",
  "description": "appointments",
```

[1] A small remark, the semantics in the listings are removed because of space issues.

```
  "domain": "vocab:EntryPoint",
  "range": "http://www.w3.org/ns/hydra/core#Collection",
  "supportedOperation": [ ... ],
},
"hydra:title": "appoinment",
"hydra:description": "Collection of registered medical
                      conditions.",
"required": null,
"readonly": true,
"writeonly": false
```

To finish the description in listing 5.2. The `domain` property tells us that we are at the main entry point, and we will retrieve a collection. The `hydra:description` here might be useful to understand the context, by stating that *here you can get the collection of the medical conditions that are registered in our database*.

To move over to a completely different syntax, listing 5.3 describes to us what the participants were to meet, and gives us an example on how a similar operation is described with RESTdesc.

Listing 5.3: Example of how a GET operation is described in RESTdesc.

```
{
  ?concert :concertId ?concertId .
}
=>
{
  _:request http:methodName "GET";
    tmpl:requestURI ("/rockefeller/concerts/" ?concertId);
    http:resp [ tmpl:represents ?concert ].
}.
```

What comes to mind first is that the example is much smaller, it is more compact. In short, RESTdesc[54, 55] is a combination of the RDF superset N3[7] and a vocabulary for expressing HTTP requests[28]. The way we can read the example in listing 5.3 is like this:

- **if** we have an ID to a concert.
- **then** we can do a GET request to the specified URI, were we will get a response with a representation of the concert.

Together with the two APIs that were handed out, we wanted to find out how well the participants could read the APIs and a set of tasks were created before the workshop took place.

For the Hydra API, we defined these tasks as for a starter[2]:

- **Retrieve entrypoint** — mark where the operation is described.
- **Book an appointment** — mark where the operation is described, and since it is a POST request, we wanted the participants to find out what elements are needed in such a request.

[2]More throughly tasks and descriptions of the participants thoughts was brought up during the discussion.

- **Register a medical condition** — mark where the operation is described and understand what is needed in such a request.
- **Retrieve physicians** mark were the operation is described.

While for the RESTdesc descriptions we defined these ones:

- **Retrieve concerts** — mark were the operation is described.
- **Retrieve a concert** — mark were the operation is described.
- **Buy a ticket** — mark were the operation is described and what is needed for a valid request.

These tasks were either given so that they could solve them on their own, and then tell the other participants afterwards, or the task was given after the individual part so it was part of the discussion.

5.4 The workshop and methodology

A group workshop with five participants and two single workshops were carried out, including the pilot workshop.

One of the positives with doing it in this way was to see if the results were the same, if the group were pointing at the same things as the ones that completed the workshop alone.

The workshop had six stages, four of them were with the participants being active. To keep the workshop semi-structured, we established a workshop guide that was tested in the pilot workshop. Why we call it semi-structured is because based on the feedback from the participants during the four steps they were active, we wanted to keep an open mind in case they had any point of views or interesting feedback that were worth following. In the end, the workshop guide looked like this:

1. Introduction: The workshop started out with an introduction of the topic, what the author's study is all about and why the participants were involved.
2. Explanation: After having the pilot workshop, there came to light that a small explanation was appropriate. Not everyone are familiar with the RESTdesc syntax, and neither working with contexts in JSON.
3. Reviewing the APIs: Then the two APIs was handed out to the participants for them to review and get a feeling of what it describes.
4. Tasks: After some minutes of reviewing the APIs, there were given tasks to solve. Either individually or as a duo. The tasks differed from each workshop. All of the tasks had something to do with how to handle an operation, or what a operation did.

5. Questions: The questions depended on how the tasks were solved. There were always a question where the participants were to solve a task together.

6. Discussion: In the end there were time to discussion and reflections for the participants, giving the author valuable data on the readability of the to different ways of describing an API.

Since the workshop was held in two different ways; two single workshops and one group workshop, did the completion of the session have minor variations.

5.4.1 Individual workshop

When holding the workshop with one person we encourage the person to "think loud", according to the think-aloud protocol[26]. This was especially important when the participant were reviewing the APIs and solving tasks. The think-aloud protocol involves the process of letting the participants state their thoughts out loud. By doing this the participant explicitly states whatever that comes into their mind.

The feedback from the think-aloud process is valuable in the sense of that we get an understanding of how users read an API, what were difficult to understand and what were easy.

Another aspect of the one-person-workshop was that the discussion part was more like a semi-structured interview. The questions and the tasks that were to be carried out were the same, but as in the nature of a semi-structured interview, the interviewer follows interesting thoughts of the participant[42, p. 299].

A positive side of having an interview with one person is that it allows us to go deeper. With one person present, you can really allow that person to give detailed thoughts and feedback about their experience with the APIs.

5.4.2 Group workshop

In the group workshop, one or two persons reviewed the APIs, together or alone, taking notes on what they were thinking about the readability, how easy it was to understand and how it was to see the whole picture.

When the participants got the tasks to solve, once again they worked either individually or in par of two, discussing and taking notes.

The question part and the discussion part of the workshop, had a blurry line and melted into each other and we can draw a lot of similarities to focus groups as described by Lazar, Feng and Hochheiser[36]. A focus group is basically an interview with more people present where it is room for discussion. This method for gathering data is good because you can gather more data in one session than with an interview. By more data, other than that there are more people present, we mean that the discussion often highlights a broader range of viewpoints. A reason for that is when

two persons have different points of view, it is easier to make them explain why they have that point of view and then compare the two statements.

5.5 Results

The main aim of this workshop were to explore the readability of two different ways of describing APIs. Therefor, at first, we thought of giving no information prior the hand outs, other than the introduction to the two first steps in the workshop guide. The reason for this was that we wanted to see how easily developers could understand and make use of an unknown API, maybe also on an unfamiliar format.

During the pilot workshop we understood that some information was needed. For example, the triple pattern does not make very much sense for developers that have had nothing to do with it before, hence RESTdec. Another remark, now for Hydra, was that JSON users are not familiar with the `@context`-key.

As for the matter of the workshop and the results, we understood that it would be realistic to let the participants first review the APIs without any information about the format and syntax, and see what the feedback was. If no one were to understand anything, we would give a brief introduction to the formats and syntax.

Such an approach would give us valuable data on how easy it is to understand these new ways of describing APIs.

5.5.1 Individual workshop

As for a starter, we look into what feedback got out from the individual workshops other that the above descriptions. This were where we encouraged the participants to carry out their thoughts verbally according to the think-aloud protocol. One of the individual workshop participants did see the RESTdesc API first, while the other one started with the Hydra API.

RESTdesc

The RESTdesc API the participants understood fast that it had something to do with concerts.

One of the participants that took part of the individual workshop was not familiar with triple patterns from before, therefor he stated that it was difficult for him to understand anything else than that it had something to with concerts. The other one had minor knowledge about triple patterns and understood quite fast what the descriptions stated. He said: “This is not very different from how you make calls in JavaScript.”

For the other participant, after a small introduction to triple patterns and the N3 format, he understood the descriptions. But for him, it was not intuitive, and as for the self-descriptiveness we are looking for, and readability of the descriptions, he thought it was not well documented.

Another aspect for him that he remarked, when you are not familiar with the different HTTP methods either, it is difficult to understand what actually happens when e.g. following a link. If he was to use RESTdesc, further documentation was needed.

Hydra

What we fast understood was that the participants thought the Hydra API was big, a bit overwhelming at a first glance. It contained several pages with JSON-LD. Almost the whole first page contained the context, which the participants used notable time to explore, before they started look at the rest of the API. The fact was that they did not understand what is was for.

When looking at the actual descriptions of what the API contains, they did not understand what the `@id`-key was for or were they could find the URLs to follow.

One of the participants had a hard time understanding anything at all before the brief introduction. After that he realized that the API contained descriptions of all actions and properties. When he realized that, he could solve all the task that were given. A for reason for his incapabilities of reading the API, he stated explicitly that it was overwhelming getting a specification on seven pages.

For the other participant, he discovered the descriptions straight away. He was one of the programming students with background that had given him experience with JSON, and he summed up his experience with Hydra in this manner:

> The `hydra:description` is very good. If developers of these APIs use that property for what it is worth, it is just to read the value of that property and you understand what it is doing. [...] I think it is really nice that everything is self-documented.

As we recall, this looks like what the aim of Hydra is, to be self-descriptive.

5.5.2 Group workshop

In the group workshop, five participants were present. This were where we got our main feedback through the method that resembles a focus group with a discussion. One noticeable aspect to have in mind is that none of the participants had any knowledge about triple formats and two had none or minor knowledge about JSON.

RESTdesc

Once again, the participants could understand that it had something to do with concerts. One of the participants that were familiar with JavaScript programming said:

> This looks like something I could have coded. I mean, the structure of it; you have something, following a path and based on the method you'll get something back. [...] I guess it depends what you are used to work with.

Since he was familiar working with JavaScript and JSON, the above enthusiasm were maybe guessed to be pointed at Hydra since that is JSON based. Another thing he marked as good was how the semantics were specified with prefixes in the top of the document. This could have something to do with that the document were fairly small.

The two that had minor JSON knowledge did not enjoy reading the RESTdesc documentation at all; they meant it was to little information. This statement might have been biased by that they looked at the Hydra API first. Anyway, it gives an understanding of that at a first glance you must be familiar with this kind of syntax to interpret it well.

Even after a brief introduction to the triple format, three of the participant were not comfortable with the API. One participant stated that it was a difficult syntax to read, "it misses the readability for humans." He admitted that it had something to do with the triples, which he were not comfortable working with. The two others, read it well.

Hydra

When it comes to the Hydra API, we once again must admit that the participants got a bit overwhelmed at first, but after some time looking at it, two of the participants quite easily understood where to find the operation for register a medical condition and what a request for an appointment needed.

The characteristics that were mentioned for this way of describing an API was that it was a big plus with the `hydra:description`, one said that it was "like reading source code" meant in a positive way. Other characteristics were that it was not intuitive and once again the question about where the request URL were came up.

5.5.3 Final remarks

Out of the seven that took part of the workshop, five had a strong opinion that they preferred to have the API described with Hydra, while the last two did not have a clear meaning. They found the readability of the RESTdesc descriptions better, but had to admit that one of the reasons for that might have been the size of the documents.

Else, it seems like the characteristics of the two APIs were quite divided in the sense that both had negative and positive sides. If we would dare to conclude, it must be that the efforts of Hydra to become a vocabulary that helps describing Hypermedia APIs in a accessible and readable way have succeeded to some extent.

Based on the quote from the participant in the individual workshop we see that he is pinpointing the core of what Hydra is trying to achieve.

A link for a computer is only something it can dereference. With some machine-readable documentation, it can know more about what will be retrieved if following that link. As well, a human user also need readable documentation within a document to be able to know how to interact with the code, or how to create a valid request following a link. We see that Hydra enables self-descriptiveness in messages both for humans and for computers.

5.6 Possible drawbacks with the study

When holding the workshop, some aspects came to light when it comes to how the participants chose to answer.

Size

First of all, there is a considerable size difference of the to different descriptions of the APIs. While the provided description of the concert API in RESTdesc is only on 49 lines of code, the Hydra API is over eight times bigger with 397 lines. This made it noticeably easier to get an overall understanding of the smaller API.

The size differences were also a matter of the nature of the two different specifications. Hydra is a much richer language when it comes to hypermedia affordance than RESTdesc.

Another aspect that is worth mentioning is that the Hydra API described the more complex physician scenario. This means that the API necessarily will be bigger, because it needs to cover more information.

Hand outs

The participants got one copy each of the two APIs on paper. The RESTdesc API covered only one A4 page, while the Hydra API covered almost seven full A4 pages. What the participants noticed here as a setback was that the print was on both sides of the pages. This made it even more difficult to get the whole picture of the Hydra API. Once again, as we see, was this the considerable difference in size a factor.

The prototype

If the prototype had been in a better shape when the workshop was conducted, it would have been really interesting to see the participants actually work with the different APIs and actually write some code to interact with it.

In such a case we could really have seen the view source capabilities, because when looking up the documentation on a computer, working with the code would been more authentic to an actual developing situation.

Unfortunately, such a workshop was not possible to conduct at that time.

The number of participants

Lazar et al.[36] discusses what is an appropriate amount of participants in a study. This discussion comes up under the topic of user-based testing, which might not be too far from what we did in the testing. They state that user-based testing is "a group of representative users attempting a set of representative tasks."[36, p. 260] We will argue that this is what we do in the workshop. The representative users are the participants, which one day might work with different APIs. The representative tasks are then understanding the API and finding information in it in various ways.

The point is that there are no definite number of participants in a study that is ideal. One might say that seven participants are too few, while some might think it is an okay amount.

As a concluding remark, we will state that the more participants, the more data you got, and again that means that you have a broader specter to make an evaluation of.

Reflections

These drawbacks might have been corrected if there were held a deeper pilot study in front of the workshop. If the workshop were to be more structured, also when holding the pilot study, corrections would most probably have been easier to make. This is of course because you are stricter to stick to the workshop guide.

Corrections that could have been made here was that we could have provided the participants with the same APIs, only with opposite way of describing them. By that we mean that the larger API, the physician domain, could have been described with RESTdesc, to eliminate some hypermedia affordance, and to have the concert domain described with Hydra.

That said, having a too structured workshop makes it more difficult to following interesting patterns and topics the participants takes.

Summary

We have in this chapter gone through the process of a minor study were we wanted to explore the view source capabilities of Hydra and RESTdesc. Both are technologies for describing APIs. The study were carried out through a workshop gathering data based on feedback from the participants. The participants had to read through two different APIs, solve some tasks and answer some questions which took more form as a discussion.

Chapter 6

Results

The results of the prototype is not easy to define, after all we are evaluating some qualitative measures like *the client could too some extent operate autonomously*. We are anyway going to try to put some words on how the client manage, but before that are we going to look into what hypermedia factors we were able to express in our application. Secondly, we are going to use Richardson Maturity Model too see if the application is truly RESTful. The results will make up the foundation together with the research question in the discussion.

6.1 What hypermedia controls are present in the Hydra API?

When extending the capabilities of JSON with JSON-LD and Hydra we have gone from using a format that does not have any native hypermedia controls to use a hypermedia rich format. What of the hypermedia factors that Amundsen[1] defined are presented to the client when it receives a response?

Mark that all of the hypermedia link factors can be present in a Hydra API, but we have limited the scope to provide the client with minimal, but sufficient with information. When it comes to the control factors, not all of them can be represented in a Hydra API.

6.1.1 Link Factors

The Link Factors is the links that enables the client to change the state of the application.

Embedded Links

The LE affordance is not present. This factor indicates for the client that the URI can be dereferenced by using the applications read operation. In this case it is the HTTP GET method. The reason why this is not present in this API is that the EL factor is transclusional and that means that the resource "on the other side of the link" is embedded in the source resource.

Outbound Links

On the other hand is LO supported. The LO factor represents navigational links and that is basically what the whole Hydra API is built upon. It indicates to the client that it can dereference the URI by using the HTTP GET method and the response replaces the source resource in the browser.

Templated Links

LT factors are not present in the application. This factor is also a read-only operation provided with HTTP GET, but in this case the client provides the request with a body. Read operations with a message body is not supported. It can be used when wanting to retrieve limited information like in a search, but in this application it is not needed.

Idempotent Links

Idempotent Links, or the LI factor is supported. It is a way to define support for HTTP PUT and HTTP DELETE operations. The concert server supports the PUT operation. It is used when booking a ticket because the resource is updated; the counter for how many tickets that are left. In the physician server there are possibilities to DELETE a registration and an appointment.

Non-Idempotent Links

The LN factor is supported in the physician server because it provides the possibilities to perform a HTTP POST method. The POST method is non-idempotent because doing the same operation several times will not give the same result within the application. Doing several POST request will most probably end up creating several similar resources with different identifiers.

6.1.2 Control Factors

The control factors are used to apply more metadata to the link operations we have been looking at. They can vary from protocol to protocol, but we are sticking to the use of HTTP.

Read Controls

If the CR factor is supported in the media-type it can help control the data in read operations. This is done by manipulating the HTTP Header. JSON-LD and Hydra does not have any support for this, so this is static through the application flow. In the header one can define that it only accepts requests on a particular format.

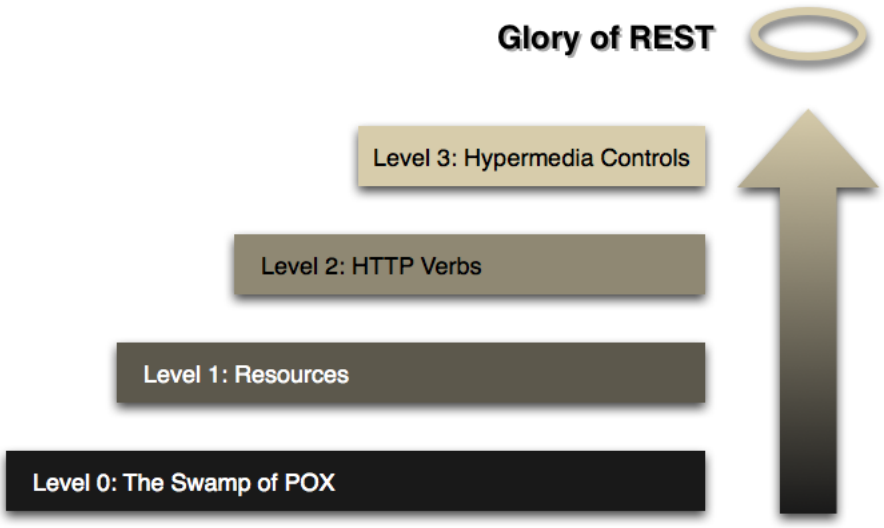

Figure 6.1: Richardson Maturity Model; the steps toward true REST.

Update Controls

The CU factor is not supported by Hydra or JSON-LD either. This control can set the content type of the message in the header. In the application this is done on the server side for each request.

Method Controls

On the other hand, method control is supported. For some of the links are there several methods, or operations, that can be performed, and Hydra helps us present this to the client. This can be controlled by the media-type and therefor is it supported in the application.

Link Annotation Controls

In the end we have the CL factor. This is a way of defining metadata about the links itself. We must say that this hypermedia factor definitely is present. By providing the Hydra API, we are describing the links so that the client can understand them and make use of them.

6.2 Is the prototype truly RESTful?

We have now confirmed that the application delivered a hypermedia API to the client. To truly validate the RESTfulness of the prototype we developed, we will now match the affordance provided through the hypermedia messages with the REST maturity model.

Richardson Maturity Model[23] in figure 6.1. shows us four levels that needs to be present in an application to claim the title as RESTful.

Level 0: ***The Swamp of POX***

The title here is just a euphemism to state that HTTP is used as tunnel for the communication between server and client. This level can be seen as a prerequisite that needs to be established before moving on to the other levels. And for our application, we have only used HTTP as the transfer protocol so we can move on to level 1.

Level 1: Resources

Instead of having the client only interact with one main entry point, only retrieving data from that spot, it interacts with several entry points. Let's say the client would only retrieve data from `http://localhost:3000/rockefeller`, that would not be RESTful. In such an architecture, the server would most probably have done the main job, finding what data that to respond with. This would make the server scale badly, which breaks one of the design goals of REST.

In REST, and in our application, the client interacts with several resources, each with its unique identifier.

Level 2: HTTP Verbs

This level encourages usage of the different HTTP Verbs. In our application we have seen the use of GET, POST and PUT, while DELETE also is a possibility in the prototype. The reason why it is encouraged to use different verbs, or methods, is that it removes unnecessary variations in the usage when handling similar situations in the same way.

Another aspect is the terms that we saw in the chapter Hypermedia; the *safe* and *unsafe* operations. E.g. GET is a safe, operation that does not affect any other parts other than possibly itself; any other resource will not be affected by the retrieval of another.

If using an unsafe operation like PUT, it might affect other resources. When updating one resource, if it in one way is connected to another, but this link breaks after the request, it has done some harm. This is why different HTTP Verbs should be used in different situations so that the client and developers can be aware of unsafe operations.

Level 3: Hypermedia Controls

The last level to achieve the RESTful status is to have hypermedia controls present. Without further ado, we can state that the application is RESTful since we have already gone through the hypermedia controls present in the application. The presence of the hypermedia controls enables discoverability and makes the application more self-descriptive.

6.3 Can the client work with the affordance?

To validate if the client can understand its environment based on the self-descriptive messages, did a minor set of test on the two servers. The tests were carried out using the same client to validate the genericity.

6.3.1 Concert server

Test 1

On the concert server, there were run a test where the clients goal were reachable. By reachable we mean that there existed a concert that had the same genre, the price were right and it was tickets left. In the scenario, it was only three concerts available for the client, still, it needed 15 requests to obtain the final goal. This is the worst case scenario, were the last concert the client checks is the fitting one.

We have to admit that this could have been further optimized because now it loops through the document in the search for the number of tickets wanted, while this is data the client already possesses. Still, the client needs to check the context for each resource, which means that for each request, there is one additional one.

This test were mainly run to see if it could obtain a ticket, and it was able to do that.

Test 2

When the client were given input that did not match any of the concerts, it was unable to find a matching concert. Still, since it uses the follow you nose principle, it needed to check each of the concerts, since it found that the API contained a matching type of what it wanted.

The main reason for this test were to see if the client were able to understand that the goal was unreachable, and it passed.

Test 3

This test were to see if it could abort the execution if the wanted type of the goal were not present, and it did pass the test.

6.3.2 Physician server

To be short, the same tests were run on the physician server, and we got the same results. This means that we have proven the client can be both generic and autonomous.

Summary

In this chapter we have looked at the results from the development of the prototype:

1. The prototype contains several hypermedia factors that is provided to the client which enables the messages sent to be self-descriptive.
2. Based on the hypermedia factors, and the rest of the architecture of the prototype we can state that it is a truly RESTful application.
3. We can state that due to the given hypermedia controls, the client can be both generic and autonomous.

Chapter 7

Discussion

We will in this chapter discuss the results from the workshop and the previous chapter. These results will be viewed in the light of the research questions so that we can get a broader insight on how truly RESTful APIs enables clients to be generic and autonomous, and how well humans can interact with it as well. This discussion will also work as a review of the research questions to see if we at all were able to answer them.

7.1 Autonomous consumption of APIs

Throughout this thesis we have mentioned the autonomous client several times. After all, it is Berners-Lee's[9] ultimate vision for the Semantic Web. For a machine to work autonomously it needs something to interact with, an interface, and this leads us to the first research question that is reviewed:

To what extent can a client be autonomous and consume an API it has never seen before?

One approach to this question was to develop a client that would connect to a server and retrieve the API. We also wanted to see if we, based on the technologies, could make the client consume several APIs to prove that it could be generic as well, and therefor encourage reuse of code. If we were capable of making a generic client then we are one step closer to making clients that can be used in several use cases, instead of making many clients that only can be used in limited use cases.

The API were provided to the client at runtime, and was found in the HTTP Link Header. Our client needs the API to be discovered there, or else it will fail with the consumption. This is a part of the consumption of the API — how to discover it. As we have already mentioned, the discovering of the API happens at runtime when the client connects to a server. With the first response, it looks in the link header to see if there is a link relation to the http://www.w3.org/ns/hydra/core#apiDocumentation which defines that by following the related link, we have obtained the API for the application.

The result of this were that we found out that we were able to develop a client that were both generic and that could discover and consume at least

two different APIs. In one way we could say that the first step for a client to actually be a functioning, autonomous part of the Web, it must be able to consume an API. If it can not consume it, it does not have anything to interact with.

The positives are that we now have successfully proved that the first step for the client to take part of the Semantic Web is achievable. This as long as the server provides the API in the desired way that the client want it to.

7.2 Relation to its own goal

The second step for an autonomous client, after consuming an API, is to start interacting with it. To prove that it can be generic as well, we need it to interact with two different APIs. A client with no artificial intelligence, must have gotten some kind of goal or task from a user to perform. How it is possible for it to carry out that task, leads us to the next research question:

Can the client reason about the consequences of actions, will it be able to relate it to its own goals?

If a client can relate its goal to an API, it needs to know some terms. The generic client that were developed knows its goal which in this scenario is stated as a JSON object where the `@type` entity is essential. In the provided goal, which a user most probably would have given the client, the `@type` is a URI. If this URI can not be found in the shared knowledge base, the API, the client will not execute any operations on that given API.

On the other hand, if it finds the particular type matching its own goal, it figures out how it can obtain its goal, usually by creating a valid HTTP request, and performs the operation on the server.

The reasoning, when it comes to implementing inference rules, is excluded from the prototype. Ideally, each term in the API could have had a rule set expressed in e.g. N3 so that a reasoner could have used the triples to give the client valuable instructions on how to obtain a particular resource or to express domain-specific knowledge. Instead we have implemented the server such that it provides the client with domain-specific modules. This is not reasoning by any means, but we have in a way simulated the reasoning part; for each URI that is dereferenceable, the client can check what it can expect as response. This means that by the help of explicitly stated triples, the client can make a choice, based on its goal, if it needs to follow the URI or not.

If a rule set were to be used instead, the client would need to be provided a reasoner from the server as well. The reasoner, e.g. EYE, could have given the client inferred statements, or even maybe instructions on how to solve its goal. This could mean that the client can be even more generic, but have not been explored in this thesis. In our prototype we end up with a similar result by providing the domain-specific code where the

client checks each resource it receives from the server to use the domain-specific knowledge.

Still, without pure Semantic Web reasoning, we are using the principles of Web APIs that makes us create generic clients. Thanks to Hydra and JSON-LD we can share machine-processable documentation between the client and the server at runtime[29] which makes it possible for both to evolve independently and the client does not need any prior knowledge about the server and the API.

7.3 The role of hypermedia and RDF

Now that we know that a client can be both generic and autonomous. How was this achieved? The last research question leads us to the fundamentals of how it is possible, what kind of technologies that were used:

What role do hypermedia and RDF play therein?

Through the chapters in this thesis we have dived into the world of hypermedia and how to express affordance with JSON-LD and Hydra. The result chapter showed us that the H-Factors OL, LI, LN, CM and CL were present in the application through the use of Hydra, while CU and CR were supported through the servers implementation.

The H-Factors is the essential part of enabling a client to be autonomous. When a message sent from the server affords a menu of link choices to the client, it enables the client to understand the environment and explore the content in a more sophisticated way than just following every link just to reveal what lays on the other side.

When hypermedia is present in an application, we finally starting to understand what REST truly is about. The server should only care about storing the resource state, e.g. that there is only 16 tickets left for the concert with the `@id` `rockefeller/concerts/3`. If someone wants to obtain a ticket, and let the client do it for them, the client will then have to make use of the hypermedia controls to understand what links brings the application to the concert resource, e.g. by GETing resources and thereby changing the application state, and in the end PUTing a request changing the resource state. This automation of a task was enabled by hypermedia and we understand that it has a rather important role, but what about RDF?

JSON-LD is a serialization of RDF. There exist other formats that have hypermedia features such as Siren[51], HAL[27] and Collection+JSON[2], but that does not contain the semantic bit, the integration with Linked Data. These three formats, as well as JSON-LD extends the capabilities of JSON to let it express hypermedia controls. But why do we think that JSON-LD is superior the other formats?

By using the RDF serialization, JSON-LD, we enables a higher autonomy of the clients, according to Verborgh[53]. This is because we are here using a semantic media type where the content can easily be understood by the client if the LD principles are applied. We are here at a

point where we can use the term *semantic hypermedia*. The reasons why we achieve a higher autonomy is the following:

1. First of all can the client now ignore irrelevant information. Since everything is identified by dereferenecable URIs, the client can follow information that it want.

2. Secondly, if applied, the server can afford vocabularies with inference rules.

3. Lastly, as we have discussed already, will the client be able to perform tasks that needs domain-specific knowledge because this if provided by the server, either as rule sets or executable modules.

Further, semantic hypermedia enables the client to be more generic as well, since the client is less and less tied to the server implementation. In the end it performs operations based on triples which is what the Semantic Web consists of.

7.4 Readability and prior knowledge

When working with Hydra we understood that one feature about it were that it enables us to write documentation that is both readable for humans and processable for machines. In the chapter about exploring view source capabilities we tried to validate its effort by holding a workshop.

In the workshop where we compared it to RESTdesc, which also is an effort to describe APIs and is a quite new technology, we first let the participants review the two APIs to try to understand them. By understand them we meant that they could tell us what a developer needed to do to obtain a specific entity, e.g. a concert or a registration for an appointment. These tasks and questions developed into a discussion of what the participants found as good features about the APIs and what they did find as less intuitive.

When talking about self-descriptive messages, and documentation that is both human readable and machine-processable the way the information is expressed must be balanced. For the ones that did not have much prior knowledge about JSON they found the `hydra:description` really helpful, as well as the more experienced ones. Looking into an API you have never seen before can be challenging, especially if you do not have a computer to try out different operations. What we found were that Hydra indeed are both machine-processable, hence the prototype, and human readable, hence that the participants understood it.

One of the participants gave us the impression that knowing what the different HTTP methods did, were hard in RESTdesc, while it in Hydra could easily be understood, again with the `hydra:description` property. Of course, developing application needs prior knowledge about many technologies, but what Hydra and JSON-LD have achieved is maybe to have lowered the bar for making use of semantic technologies. Lanthaler

and Gütl[32] talks about *semaphobia* as the fear of average Web developers to make use of these technologies. All developers have a different kind of skill set. Someone knows PHP, some JavaScript while others uses only Phyton. Unfortunately we did not get the time to see if the participants in the workshop were actually able to do some coding and interaction with the API, but we showed that it is possible for them to understand Hydra. Understanding the documentation is a major step to make use of it.

7.5 Existence of the client

A question that came up during the workshop were: "in what space is this client supposed to live?" and the participant followed up with that he did not understand what the use of a autonomous, generic client were. This grasps around the existence of of this research and takes us back to Berners-Lee visions:

A client of this nature will easily be able to crawl the Web, first to discover Hydra powered APIs, then browse the API. By crawling the Web in such a manner enables us to use data from several different sources within one application. When using a client that consumes Hydra API, the client further can convert the data in any wanted RDF serialization. This brings us to one point that needs further research; develop an application of the nature that we have in this thesis, only with rule sets that enable the application to discover and create new knowledge through reasoning. If we let the client do the browsing for us, solving tasks on the Web for us, we lower the *entry barrier*[17] even more.

For our client, we can say that it justifies its existence since it is generic. The `public/javascripts/generic-client.js` can be implemented in other projects to explore Hydra APIs autonomous, if it is given domain-specific knowledge through modules provided by the server, according to the REST principles.

To further justify the existence of such a client and application, Lanthaler and Gütl[35] points out some of the benefits of a REST application based on JSON-LD such as ours:

Loose coupling

Because of the hypermedia controls in the resource representation, the client of the application do not need any prior knowledge about the domain. If the client were more domain-specific, changes on the server would break the client, thus domain-specific implementations are tight coupled.

Evolvability

Because of the loose coupling, we then enable the evolvability of each component individually. Without changing the server of our application, we could go on and make the client more robust without breaking application as a whole. Another aspect is that we could let the resources

evolve as well. This is common in applications, either by that clients change the resource state, or that publishers changes the content. Still, if this happens, the client will know, based on the hypermedia controls where to obtain the resources.

Scalability

Since our client is the smart part of the application, we have enabled the server to scale good because it is there just to respond to requests, and provide the client with choices, the hypermedia controls. This is an essential design goal of REST[17] because we are here talking about something that is going to exist in a distributed network and the server should be able to handle many requests.

Self-descriptiveness

We have already touched upon this benefit in chapter 5 and discussed its positive sides in section 7.4.

Maintainability

The maintainability of the different components of application is heavily strengthened because of the loose coupling. Now, maintaining one part of the application does not mean that other parts necessarily also must be looked at. For instance, if the server breaks, we only need to maintain that component, not look at all of the parts of the application.

7.6 Constraints and limitations

So far we have seen all the good parts of REST and hypermedia. But what are the limitations and constraints in such an application, i.e. our prototype?

Complexity

First of all, the JSON-LD can get quite complex as we did see in the chapter 5. When the JSON becomes complex, and the semantics in the `@context` becomes various, there is needed processing of the data for the client to interpret the properties in the right way.

Latency and efficiency

This brings us to another downside, that is efficiency and latency of an application. If an application is mission critical, hence a pacemaker that need to process data fast to make the right decisions, our prototype of REST is not to be considered of course. This is because, the processing of the properties, e.g. JSON-LD expansion[37] takes time. Worse efficiency

is followed by increased latency which is not acceptable in mission critical services.

Processing

The need for possible extensible processing of the data retrieved from the server also makes this type of architecture possibly not suitable for devices with poor specifications, i.e. minimal of RAM. In a world where *Internet of things* will emerge, REST might be asking to much of the clients. This might perhaps call for another architecture. Let us say that you want to control different things in the environment at home, e.g. temperature, humidity and the amount of day light. This data is gathered from small sensors that sends this to a micro-controller. This micro-controller might not have sufficient with processing power to know what to do with the data, then it needs to send it to a central server that processes it. One can argue that this example is not part of a large distributed network, but at least it shows us that REST is not suitable in all use cases.

To take the above thoughts further: one of the architectural properties of REST that have been mentioned throughout this thesis is scalability. When developing the prototype, a question often came in mind: Can not the server take responsibility of solving this? By this the author meant operations that needed domain-specific knowledge. Let us look at an example:

If the physician server were to be implemented with more server intelligence, the work flow could have looked something like the following. The client will still book an appointment with a physician and still have some predefined user input. In this case the client would have posted its user input for the registration of the medical condition and the preferred date. With a smart server, the server would then do the remaining steps we remember from the chapter about the prototype. The server would then look up all the physicians and found a fitting one.

In this scenario, we would still be needing hypermedia, because the client needs to know where it should POST the registration. This sounds like REST, since the HATEOAS constraint is present. Where the REST is breaking, is when the server is starting to look up different resources, i.e. changing the state of the application. Such an approach does not scale well since the server needs to the the work and is not appropriate in a distributed information network such as the Web.

At this stage we have an understanding of that REST works in Web applications while there might be use cases it does not fit that well to.

Bandwidth

Lastly, with the possibly complex JSON-LD, much metadata in the HTTP Header and extra HTTP requests for each resource retrieved to understand the semantics in the `@context`, the application will use extensible more bandwidth.

7.7 Future research

As our research was focused on the proof of the concept of the client we have developed, there are obvious limitations in the research that we were unable to test. These parts are we encouraging others to research further. The parts are also closely related to the limitations with such an approach to develop an application.

For example, in what situations are REST not a good fit. In this research we have only reveled that a generic client code can be reused. Further research could have tested the implementation on a device with poor specification. Will it be able to process the data? In situations were a lot of data must be processed, will it be able to do that or does it need help from other devices to do some of the processing? All of these questions are related to the Internet-scale property of the Web were we want the server to do as little as possible.

Another part of future research could be to look at how the hypermedia extensions in the response will affect the network. If the hypermedia controls are present in the representation of the resource, will it affect the size of the message sent; will this ever be a problem that the message is too large? One more aspect related to this issue is when the hypermedia controls are present in a separate document, were the client for each resource must do at least one more HTTP request to obtain the hypermedia controls. To test if this will cause any troubles for either the server or the network, large messages must be transmitted by many requests from different clients.

A part that could have been done, were to see the differences in the client using rule set to explore domain-specific knowledge and what we did, let the server provide the client with modules. Will the rule sets, using more Semantic Web technologies, enable even more generic clients?

As a last point, what needs to be done, not necessarily as research, is to realize the opportunities that lies within true REST; there must be consensus on standards and practices that enhances all aspects of REST, even the bit about hypermedia. At the moment it is too much confusion on what REST actually is, and hopefully have this thesis helped getting a better understanding of it.

Summary

In this chapter we have reviewed and discussed the research question for this thesis. Through the discussion we highlighted the strengths of this research and how we were able to answer the questions. The arguments that were given were mainly based on the prototype that were developed throughout the process, and showed us that a client can consume two different APIs, relate to its own goals and perform actions on it. This were achieved by the use of semantic hypermedia, a format with hypermedia controls that applies Linked Data principles, i.e. JSON-LD and Hydra. As such were RDF a part of the creation and enabling of the prototype to be

generic and autonomous. We then discussed the readability and what prior knowledge developers need when using Hydra, and we looked at where such a client has its space, before looking at limitations and what can be done in future research.

Chapter 8

Conclusion

With the demand of better application and services, data must be created, found and used. The Web today is a tremendous web of data where you can find whatever you are looking for. Some information is easy to find while other calls for more processing. With the Web, and its low-entry barrier and easy access the possibilities with it is endless. Research on these capabilities are continuous and have brought us to the creation of Semantic Web and further how machines can be part of this Web, being autonomous, creating its own data and make it accessible for others.

In this thesis we have investigated the possibilities of enabling Berners-Lee's visions for the future Web with computers working autonomously on several APIs based on hypermedia messages provided by the server to the client. As a product of the nature of hypermedia, we also investigated the readability of a particular format, to get a better understanding of what do developers need of prior knowledge when working with such a REST API.

In order to achieve this, we developed a client that consumes two different APIs, and based on its goal tries to reach it solely through the interaction with the server's hypermedia messages. Since there have not been developed such a client before, we see this as a proof of concept. Thus can the development of the prototype and exploring possibilities be seen as our research, were the execution of it being the tests. Another aspect is that since REST is misunderstood, we have checked if our prototype is a truly RESTful application based on the hypermedia.

Our results tells us that it indeed is possible to let a client be both generic and autonomous. This means that in the future, such a software can be implemented in other applications. It can then be used to crawl the Semantic Web for wanted data based on the goal that the user provides it with. If the client has some knowledge, discovers new one and combines this, new knowledge can have been created and since it works with the Linked Data principles, this new knowledge will be accessible for everyone.

Truly RESTful clients is recommended to use when the scope is an application that are working on the Web. Since our research is limited, we do not have any data on the efficiency, but with genericity, more processing is needed to convert data to the proper format. In smaller scale projects

within companies, at home, or in a field where you need high performance, REST might not be the best fit either because the components actually should be tight coupled.

Based on our research, we conclude that:

- A client can be autonomous and consume APIs it have never seen before.
- It can then perform actions, understand what an action does and relate the action to its goal.
- In this scenario hypermedia were essential to let the client understand the actions and its consequences. The RDF helped us extend the application with semantics that were important for the relation to the goal.
- Semantic hypermedia formats enables us to lower the bar for the understanding of an API that is discovered through the HTTP message.

Bibliography

[1] Mike Amundsen. *Building Hypermedia APIs with HTML5 and Node*. O'Reilly, 2012.

[2] Mike Amundsen. *Collection+JSON - Document Format*. 2013. URL: http://amundsen.com/media-types/collection/format/ (visited on 01/08/2015).

[3] Mike Amundsen. 'Hypermedia-Oriented Design'. In: *W3C Workshop on Data and Services Integration*. Oct. 2011. URL: http://www.w3.org/2011/10/integration-workshop/p/hypermedia-oriented-design.pdf.

[4] David Beckett et al. *RDF 1.1 Turtle. Terse RDF Triple Language*. W3C Recommendation. 2014. URL: http://www.w3.org/TR/turtle/ (visited on 03/05/2015).

[5] Thomas Bergwinkl. *Hydra Core*. URL: https://github.com/bergos/hydra-core (visited on 01/06/2015).

[6] Tim Berners-Lee. *Linked Data*. 2009. URL: http://www.w3.org/DesignIssues/LinkedData.html (visited on 07/01/2015).

[7] Tim Berners-Lee and Dan Connolly. *Notation3 (N3): A readable RDF syntax*. W3C Team Submission. 28th Mar. 2011. URL: http://www.w3.org/TeamSubmission/n3/ (visited on 27/06/2015).

[8] Tim Berners-Lee, Wendy Hall and Nigel Shadbolt. 'The Semantic Web Revisited'. In: *IEEE Computer Society* (2006).

[9] Tim Berners-Lee, James Hendler and Ora Lassila. 'The Semantic Web'. In: *Scientific American* 284 (5 May 2001), pp. 34–43.

[10] Tim Berners-Lee, Larry Masinter and Mark McCahill. *Uniform Resource Locators (URL)*. 1994. URL: http://tools.ietf.org/html/rfc1738 (visited on 25/07/2015).

[11] Tim Berners-Lee et al. *Architecture of the World Wide Web, Volume One*. W3C Recommendation. 2004. URL: http://www.w3.org/TR/webarch/ (visited on 01/07/2015).

[12] Christian Biezer, Tom Heath and Tim Berners-Lee. 'Linked Data – The Story So Far'. In: *International Journal on Semantic Web and Information Systems* 3 (5 Mar. 2009), pp. 1–22. URL: http://tomheath.com/papers/bizer-heath-berners-lee-ijswis-linked-data.pdf.

[13] Dan Brickley and R. V. Guha. *RDF Schema 1.1*. W3C Recommendation. 2014. URL: http://www.w3.org/TR/rdf-schema/ (visited on 20/06/2015).

[14] Vannevar Bush. 'As we may think'. In: *The Atlantic Monthly* 176 (1 July 1945), pp. 101–108. URL: http://www.theatlantic.com/magazine/archive/1945/07/as-we-may-think/303881/.

[15] Richard Cyaniak, David Wood and Markus Lanthaler. *RDF 1.1 Concepts and Abstract Syntax*. W3C Recommendation. 2014. URL: www.w3.org/TR/2014/REC-rdf11-concepts-20140225/ (visited on 20/06/2015).

[16] Eric Elliott. 'Building RESTful APIs'. In: *Programming JavaScript Applications*. O'Reilly, 2014. Chap. 8.

[17] Roy Thomas Fielding. 'Architectural Styles and Desing of Network-based Software Architectures'. PhD thesis. University of California, Irvine, 2000.

[18] Roy Thomas Fielding. *REST APIs must be hypertext-driven*. Oct. 2008. URL: http://roy.gbiv.com/untangled/2008/rest-apis-must-be-hypertext-driven (visited on 31/02/2015).

[19] Roy Thomas Fielding and Richard N. Taylor. 'Principled Design of the Modern Web Architecture'. In: *Transactions of Internet Technology* 2 (2 May 2002), pp. 115–150. URL: http://www.ics.uci.edu/~fielding/pubs/webarch_icse2000.pdf.

[20] Roy Thomas Fielding et al. *Hypertext Transfer Protocol – HTTP/1.1*. 1999. URL: http://tools.ietf.org/html/rfc2616 (visited on 25/07/2015).

[21] The jQuery Foundation. *jQUERY*. 2015. URL: https://jquery.com/ (visited on 04/08/2015).

[22] Node.js Fountdation. *Node.js*. 2015. URL: https://nodejs.org/ (visited on 26/07/2015).

[23] Robert Fowler. *Richardson Maturity Model*. 2010. URL: http://martinfowler.com/articles/richardsonMaturityModel.html (visited on 28/07/2015).

[24] Steve Harris and Andy Seaborne. *SPARQL 1.1 Query Language*. W3C Recommendation. 2013. URL: http://www.w3.org/TR/sparql11-query/ (visited on 11/05/2015).

[25] Ivan Herman et al. *RDFa 1.1 Primer - Third Edition. Rich Structured Data Markup for Web Documents*. W3C Working Group Note. 2015. URL: http://www.w3.org/TR/xhtml-rdfa-primer/ (visited on 20/07/2015).

[26] Riitta Jääskeläinen. 'Think-aloud protocol'. In: *Handbook of Translation Studies*. Volume 1. John Benjamins Publishing Company, 2010, pp. 371–373.

[27] Mike Kelly. *HAL — Hypertext Markup Language*. 2013. URL: http://stateless.co/hal_specification.html (visited on 01/08/2015).

[28] Johannes Koch, Carlos A. Velasco and Philip Ackermann. *HTTP Vocabulary in RDF 1.0*. W3C Working Draft. 2011. URL: http://www.w3.org/TR/HTTP-in-RDF10/ (visited on 30/07/2015).

[29] Markus Lanthaler. 'Creating 3rd Generation Web APIs with Hydra'. In: *Proceedings of the 22nd international conference on World Wide Web — Companion*. May 2013. URL: http://www.markus-lanthaler.com/research/creating-3rd-generation-web-apis-with-hydra.pdf.

[30] Markus Lanthaler. *Hydra Console*. URL: http://www.markus-lanthaler.com/hydra/console/ (visited on 10/08/2015).

[31] Markus Lanthaler. *Hydra Core Vocabulary. A Vocabulary for Hypermedia-Driven Web APIs*. Unofficial Draft. 2015. URL: http://www.hydra-cg.com/spec/latest/core/ (visited on 09/07/2015).

[32] Markus Lanthaler and Christian Gütl. 'A semantic description language for RESTful Data Services to combat Semaphobia'. In: *Proceedings of the 2011 5th IEEE International Conference on Digital Ecosystems and Technologies Conference (DEST)*. 2011.

[33] Markus Lanthaler and Christian Gütl. 'Model Your Application Domain, Not Your JSON Structures'. In: *Proceedings of the 22nd International World Wide Web Conference — Companion*. May 2013.

[34] Markus Lanthaler and Christian Christian Gütl. 'Hydra: A Vocabulary for Hypermedia-Driven Web APIs'. In: *Proceedings of the 6th Workshop on Linked Data on the Web*. May 2013. URL: http://ceur-ws.org/Vol-996/papers/ldow2013-paper-03.pdf.

[35] Markus Lanthaler and Christian Christian Gütl. 'On Usin JSON-LD to Crete Evolvable RESTful Services'. In: *Proceedings of the Third International Workshop on RESTful Design*. Apr. 2012. URL: http://www.markus-lanthaler.com/research/on-using-json-ld-to-create-evolvable-restful-services.pdf.

[36] Jonathan Lazar, Jinjuan Heidi Feng and Harry Hochheiser. *Research Methods. In Human-Computer Interaction*. Wiley, 2010.

[37] Dave Longley et al. *JSON-LD 1.0 Processing Algorithms and API*. W3C Recommendation. 2014. URL: http://www.w3.org/TR/json-ld-api/ (visited on 08/06/2015).

[38] Deborah L. McGuinness and Frank van Harmelen. *OWL Web Ontology Language. Overview*. W3C Recommendation. 2014. URL: http://www.w3.org/TR/owl-features/ (visited on 20/06/2015).

[39] Inc. MongoDB. *MongoDB*. 2015. URL: https://www.mongodb.org/ (visited on 26/07/2015).

[40] Ted Nelson. 'Complex information processing: a file structure for the complex, the changing and the indeterminate'. In: *Proceedings of the ACM 20th National Conference*. 1965, pp. 84–100.

[41] Donald Norman. *Design of Everyday Things*. Basic Books, 2013.

[42] Jenny Preece, Yvonne Rogers and Helene Sharp. *Interaction Design. beyond human-computer interaction*. 2nd. John Wiley & Sons, Ltd, 2007.

[43] Leonard Richardson and Mike Amundsen. *RESTful Web APIs*. O'Reilly, 2013.

[44] Jos De Roo. *Euler Yet another another proof Engine*. URL: http://eulersharp.sourceforge.net/ (visited on 23/07/2015).

[45] Michael L. Scott. *Programming Language Pragmatics*. 3rd ed. Morgan Kaufmann, 2009.

[46] Michael K. Smith, Chris Welty and Deborah L. McGuinness. *OWL Web Ontology Language. Guide*. W3C Recommendation. 2004. URL: http://www.w3.org/TR/owl-guide/ (visited on 21/06/2015).

[47] Manu Sporny. *Google adds JSON-LD support to Search and Google Now*. 2013. URL: http://manu.sporny.org/2013/json-ld-google-search/ (visited on 20/07/2015).

[48] Manu Sporny et al. *JSON-LD 1.0. A JSON-based Serialization for Linked Data*. W3C Recommendation. Hydra W3C Community Group, 2014. URL: http://www.w3.org/TR/json-ld/ (visited on 07/06/2015).

[49] Dobri Stoilov. *OWLIM-Lite*. 2011. URL: http://owlim.ontotext.com/display/OWLIMv40/OWLIM-Lite (visited on 23/07/2015).

[50] Inc. StrongLoop. *Express.js*. 2015. URL: http://expressjs.com/ (visited on 26/07/2015).

[51] Kevin Swiber. *Siren: a hypermedia specification for representing entities*. URL: https://github.com/kevinswiber/siren (visited on 01/08/2015).

[52] Ruben Verborgh. *Fostering intelligence by enabling it. Intelligent agents require an environment that allows them to act smart*. 2015. URL: http://ruben.verborgh.org/blog/2015/02/25/fostering-intelligence-by-enabling-it/ (visited on 04/08/2015).

[53] Ruben Verborgh. 'Serendipitous Web Applications through Semantic Hypermedia'. PhD thesis. Ghent University, Belgium, 2014.

[54] Ruben Verborgh et al. 'Capturing the functionality of Web services with functional descriptions'. In: *Multimedia Tools and Applications* 64 (2 May 2013), pp. 365–387.

[55] Ruben Verborgh et al. 'Functional Descriptions as the Bridge between Hypermedia APIs and the Semantic Web'. In: *Proceedings of the Third International Workshop on RESTful Design*. Apr. 2012, pp. 33–40.

[56] Ruben Verborgh et al. 'Querying Datasets on the Web with High Availability'. In: *Proceedings of the 7th Workshop on Linked Data on the Web*. Oct. 2014. URL: http://linkeddatafragments.org/publications/iswc2014.pdf.

[57] W3C. *Vocabularies*. 2015. URL: http://www.w3.org/standards/semanticweb/ontology (visited on 20/07/2015).

[58] David Wood, Marsha Zaidman and Luke Ruth. *Linked Data. Structured data on the Web*. Manning, 2014.

[59] *World Wide Web*. URL: http://info.cern.ch/hypertext/WWW/TheProject.html (visited on 11/03/2015).

Appendices

Appendix A

The code base

The code base was too big to display in the thesis, so all the code can be found in two projects:

- **Concert server** https://github.com/kurtwood/concert_server
- **Physician server** https://github.com/kurtwood/master_hydra_example

Appendix B

The Hydra API

The Hydra descriptiosn of the physician API the participants of the workshop were provided with can be viewed at https://github.com/kurtwood/master_hydra_example/blob/master/data/vocab.json.

Appendix C

The RESTdesc descriptions

The RESTdesc descriptions of the concert API the participants of the workshop were provided with can be viewed at https://github.com/kurtwood/concert_server/blob/master/descriptions/concerts.n3.