

Graz University of Technology

---

DI Markus Staud, BSc  
1031556

# Enabling Web Service Mash-Ups through Semantic Technologies in an automotive Context

Master's Thesis

---

In cooperation with the BMW Group.



---

Institute for Technical Informatics

**Supervisor - TU Graz:** Univ.-Prof. Marcel Carsten Baunach  
**Supervisor - BMW Group:** DI Daniel Wilms

---

Munich, November 2016

This document is set in Palatino, compiled with pdfL<sup>A</sup>T<sub>E</sub>X2<sub>ε</sub> and Biber.

The L<sup>A</sup>T<sub>E</sub>X template from Karl Voit is based on KOMA script and can be found online:  
<https://github.com/novoid/LaTeX-KOMA-template>

## Acknowledgements

First and foremost I want to express my deep sense of gratitude to my supervisor from BMW research and development, Daniel Wilms, for his perpetual support and belief in my work. Not only did he inspire and guide me in a vocational way, but he also helped to develop myself personally.

My appreciation also extends to our team of LT-3. They created an invaluable experience for me, both professionally and personally and I am happy to say that I have not only found competent colleagues but genuine friends. In particular, I want to thank Mr Adnan Bekan for his calming words in distress and his invaluable advice and expertise in vocational matters.

Moreover, I would like to express my appreciation for my supervisor from Graz University of Technology, Prof. Marcel Baunach, for enabling me to write this thesis and revising its content. In terms of revising, I also want to highlight the efforts of Siddarth Sreeram who revised and corrected this paper as proficient English speaker. Finally, I want to thank my family and friends, for without them I would never have come this far. Your endless support empowers me to thrive for new achievements every day and I cannot emphasize enough how important your company and trust are to me.

Vielen Dank!

Thank you very much!

Hvala!

A handwritten signature in blue ink, appearing to be 'MS' or similar initials, written in a cursive style.

Markus Staud



## Statutory Declaration

I declare that I have authored this thesis independently, that I have not used other than the declared sources/resources, and that I have explicitly marked all material which has been quoted either literally or by content from the used sources.

Graz, \_\_\_\_\_  
Date Signature

## Eidesstattliche Erklärung<sup>1</sup>

Ich erkläre an Eides statt, dass ich die vorliegende Arbeit selbstständig verfasst, andere als die angegebenen Quellen/Hilfsmittel nicht benutzt, und die den benutzten Quellen wörtlich und inhaltlich entnommenen Stellen als solche kenntlich gemacht habe.

Graz, am \_\_\_\_\_  
Datum Unterschrift

<sup>1</sup>Beschluss der Curricula-Kommission für Bachelor-, Master- und Diplomstudien vom 10.11.2008; Genehmigung des Senates am 1.12.2008



## Abstract

Development cycles in software engineering are progressively getting shorter, new apps gain popularity overnight and formerly unexploited data sources are tapped by the minute. The automotive industry faces a fierce battle against the rest of the consumer goods industry, causing automotive manufacturer's customers to demand the same rapid adaptation to innovation.

While the traditional approach in developing cars is to specify requirements three to five years ahead of start of production, this approach hardly works in software development. Thus, in order to enable cars to quickly integrate new services and consequently add value to the whole product, a sustainable software architecture needs to be found.

Therefore, this thesis assesses various approaches of describing web services and how to create value-adding mash-ups. This includes a close examination of the current state of Semantic Web technologies and how to utilize them for linking up services in a car manufacturer's backend infrastructure. An exemplary implementation then demonstrates where the key-issues are currently at, therefore following a bottom-up approach using existing real-world services.

The existing gap between high-level abstractions envisioning machine-agents reasoning upon meaningful data and proprietary non-interoperable low-level implementations can only be bridged in semantically controlled and, thus, unambiguous environments. The proposed workflow scheme introduced in the course of this thesis tries to cover this middle ground by exposing only semantically annotated data and manually mapping strings to semantically unambiguous things.

The results show, that existing web service implementations are still far from habituating a standardized environment. To enable Semantic Web technologies, data providers would still have to reach consensus on a multiplicity of low-level implementation details.

A final discussion concludes from the lessons learned and leads to an outlook that provides further implications for future research in this field.





# Contents

<b>Statutory Declaration</b>	<b>iv</b>
<b>Abstract</b>	<b>v</b>
<b>1 Introduction</b>	<b>1</b>
<b>2 Related Work</b>	<b>3</b>
2.1 The Semantic Web	3
2.1.1 The Web Platform	6
2.1.2 Data Model: RDF Graph	13
2.1.3 Formats	17
2.1.4 Ontologies	23
2.1.5 Reasoning	25
2.1.6 Linked Data	26
2.2 Web Service Architectures	28
2.2.1 Classical Service Oriented Architectures	28
2.2.2 REST - Representational State Transfer	32
2.3 Static Web Service Description	36
2.3.1 WADL - Web Application Description Language	37
2.3.2 Swagger, RAML and API blueprint	39
2.4 Dynamic Web Service Description	41
2.4.1 HAL - Hypertext Application Language	42
2.5 Semantic Web Service Description	44
2.5.1 Hydra	44
2.5.2 RESTdesc	46
2.6 Access Control	49
2.6.1 OAuth	49
<b>3 Methodology</b>	<b>53</b>
3.1 Concept of Generic Workflows	53
3.2 Proposed Architecture	55
3.2.1 Basic components	56
3.2.2 Workflow description	62
3.2.3 Workflow item description	70
3.2.4 Service mapping	75
3.3 Used APIs	85
3.3.1 Discovery services	85
3.3.2 Vehicle services	87

## Contents

3.3.3	Payment services . . . . .	87
3.3.4	Ordering services . . . . .	88
3.4	Limitations . . . . .	89
<b>4</b>	<b>Results</b>	<b>91</b>
4.1	Review of existing Web Service Description approaches . . . . .	91
4.2	Discovery Workflow Implementation . . . . .	95
4.3	Lessons learned . . . . .	101
<b>5</b>	<b>Conclusions</b>	<b>105</b>
5.1	Discussion . . . . .	105
5.1.1	Semantic Web Technologies . . . . .	107
5.1.2	Classical SOA vs. REST . . . . .	107
5.1.3	Workflow Concept Implementation . . . . .	108
5.2	Implications . . . . .	110
<b>Appendix</b>		<b>111</b>
Additional Listings . . . . .		112
List of Abbreviations . . . . .		116
List of Figures . . . . .		117
List of Listings . . . . .		120
<b>Bibliography</b>		<b>121</b>

# 1 Introduction

Since the creation of the world wide web, the web itself is constantly transforming. The current state of the art is that human beings invoke web services by browsing the internet with dedicated web-browsers, like Google Chrome, Mozilla Firefox, Microsoft Edge or others. Therefore, information is exchanged in a human-readable way, usually as plain text enriched with tags, providing additional information for its representation within a browser. The common standard here is the Hypertext Markup Language or HTML. However, as machines themselves are being connected to the web, the need for machine-*understandable* information exchange is arising. Thus, a multitude of proposals for standards and best-practices is emerging, trying to embed devices into the fabric of a so-called Web of Things (WoT).

Application Programming Interfaces (APIs) for web services are designed to offer a non-HTML way of exposing and consuming resources for servers and clients, respectively. As modern vehicles are equipped with an enormous variety of sensors and actuators, they represent the ultimate WoT-device. Hence, embedding cars as things into the WoT will be a critical endeavor for automotive manufacturers. Being one of them, BMW is well-aware of this demand. On the occasion of their 100<sup>th</sup> anniversary, they published their new strategy "NUMBER ONE >Next" in a press release[1], stating:

*"In the coming years, the Group will focus on broadening its technological expertise, expanding the scope of digital connectivity between people, vehicles and services and actively strengthening sustainable mobility."*[1, p. 1-2]

as well as

*"A clear focus will be placed on high definition digital maps, sensor technology, cloud technology and artificial intelligence, the decisive areas for success in this segment."*[1, p. 4]

In order to make these visions come true, infrastructure has to be created, allowing consumers to integrate their vehicles into the WoT and enabling them to integrate third-party services to ultimately add further value to their product. These value-adding services may occur in the form of data offered by physical devices, or enterprises exposing their internal business processes online. However, there still prevails the challenge of linking these individual services together to attain more complex targets. Current best-practice usually requires software developers drawing static links between services at creation or human reasoning for composition at runtime. Here, semantic web technologies might provide a remedy.

## 1 Introduction

In the course of this thesis, the following research questions should provide guidance for the narrative:

### **How can web services be described?**

After conducting extensive research, proposals for a generic description of web services are to be examined and compared among each other. Afterwards, one of these description methods is chosen for implementation in a proof-of-concept (PoC) solution. This PoC has to be based on real world use-cases, to demonstrate practical relevance.

### **How can various web services be composed to realize a specific use case?**

Based on this formalized description of web services, concepts for web service composition are to be examined. Yet again, existing proposals are to be compared and discussed. For prototypical implementation, one of them is then to be included in the PoC.

### **Which web service description and composition styles are most promising?**

Based on an evaluation of the actual implementation of the PoC, the implemented service composition style is verified for its capability in a real-world environment.

With all these questions answered, a further outlook and implications for improvements and future work can be made.

The thesis is divided into five chapters. The second chapter elaborates previous work on web service description and composition. It also intends to provide an overview of the required fundamentals as well as references for further reading. Based on this theoretical backing the third chapter elucidates the chosen methodology and explains how the results of this work were obtained. In chapter four, the results gained from the PoC proposed in the methodology chapter are presented and interpreted. The last chapter links these results to results from past work and distinguishes them from previous approaches. Finally, these conclusions are used for further implications on this topic.

## 2 Related Work

This chapter offers a theoretical background for the thesis, laying the foundation for all chapters to follow. Its goal is to facilitate a basic understanding of the topic, rather than provide an exhaustive clarification of all the theory used in the course of this thesis. For in-depth investigation, further reading following the given references is highly encouraged.

The first section starts with the basics of web technologies and the Semantic Web will be discussed. Specifications that are used in further considerations will be examined in more detail. Subsequently, an analysis of two major architectural approaches is conducted, dissecting them into their fundamental principles. Based on technical specifications and architectural rationale, various web service description techniques will be analyzed in regards to their conformance to architectural principles, maturity and application. In the last section, orthogonal components as authentication and authorization will be considered and the popular OAuth standard presented.

### 2.1 The Semantic Web

Since its beginnings in 1989, the World Wide Web (WWW) has been subject to unmatched progress and evolution, both in technologies enabling it and its impact on modern society. The idea for a world-wide network arose when Tim Berners-Lee and his colleagues tried to find a way to cut a Gordian knot of incompatible networks, disk formats, data formats and character encoding schemes that had emerged at CERN in the past decades. Therefore, they intended to create a shared information space, where people as well as machines could communicate with each other. To guide their endeavors, they formulated six criteria the web should be able to accomplish:[2]

1. An information system must be able to record random associations between any arbitrary objects, unlike most database systems;
2. If two sets of users started to use the system independently, to make a link from one system to another should be an incremental effort, not requiring unscalable operations such as the merging of link databases.
3. Any attempt to constrain users as a whole to the use of particular languages or operating systems was always doomed to failure;
4. Information must be available on all platforms, including future ones;
5. Any attempt to constrain the mental model users have of data into a given pattern was always doomed to failure;

## 2 Related Work

6. If information within an organization is to be accurately represented in the system, entering or correcting it must be trivial for the person directly knowledgeable.

The basic architectural principles to accomplish these goals comprised a few already well-established practices and protocols of software design.

For a better understanding, the original WWW architecture diagram is illustrated in figure 2.1 (reillustrated after the original by Tim Berners-Lee.):

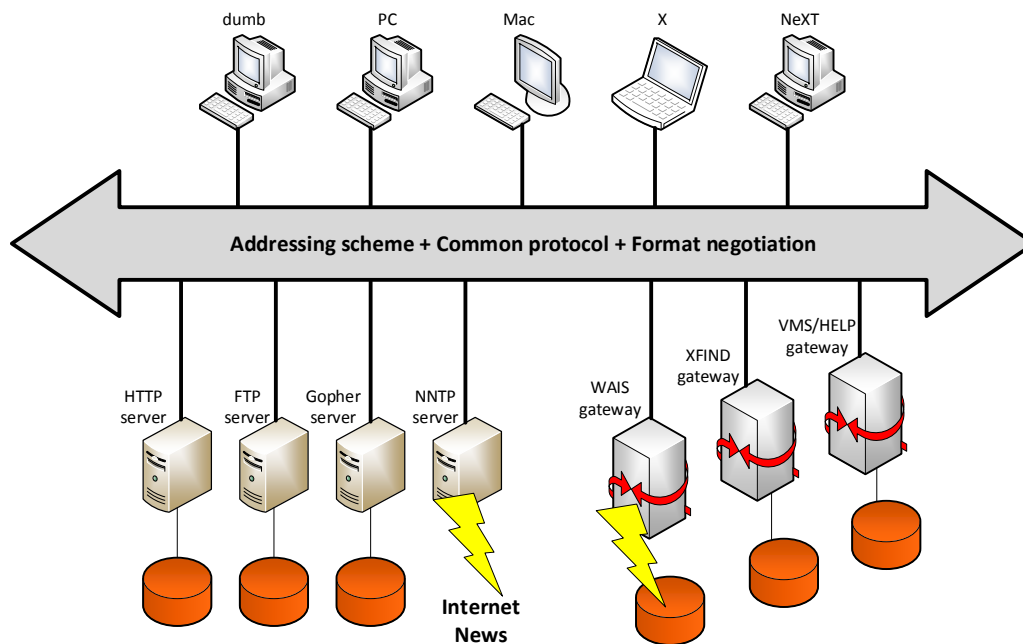


Figure 2.1: WWW architecture diagram from 1990.

A key point in designing the web architecture was that specifications should be independent while still being interoperable. This would allow parts of the web to evolve, while older specifications would still remain operable. Thus, various different protocols emerged, quickly becoming standards in what today is the World Wide Web. Subsection 2.1.1 will further elaborate on these standards, namely Universal Resource Identifiers (URIs), the Hypertext Transfer Protocol (HTTP) and the Hypertext Markup Language (HTML).

To navigate or *browse* the web, client interfaces had to be implemented, today usually referred to as "web browsers". The first popular web browser came in the form of Mosaic in 1993, comprising a graphical user interface and able to offer simple traversal of web content for the average user. Later on, large companies like Microsoft with their Internet Explorer, as well as open source initiatives like Mozilla Firefox (in fact a distant relative of Mosaic) further contributed to making the WWW accessible.

## 2.1 The Semantic Web

Web browsers seemed to offer a good way for human users to interact with the web. However, while most of the data offered through browsers was machine-readable, hardly any data was machine-understandable. Envisioning a web, where the meaning of data is not only comprehended by human agents, but also machine agents, Tim Berners-Lee proposed the concept of the Semantic Web. [3] There, machine agents should not be able to reason upon data through complex artificial intelligence, but by semantic information encoded in web pages themselves.

This section elaborates the idea of the Semantic Web, starting by introducing the semantic web technology stack. The following subsections will then further elaborate the main building blocks of semantic web technologies. The technologies that are used to realize the vision of the semantic web are usually depicted in a layered stack, the so-called Semantic Web Technology Stack and stems from the current world wide web standards, Hypertext Transfer Protocol (HTTP), Unified Resource Identifiers (URI) and Hypertext Mark-up Language (HTML).

Figure 2.2 illustrates the Semantic Web Stack in three dimensional graphics (courtesy of Benjamin Nowack).

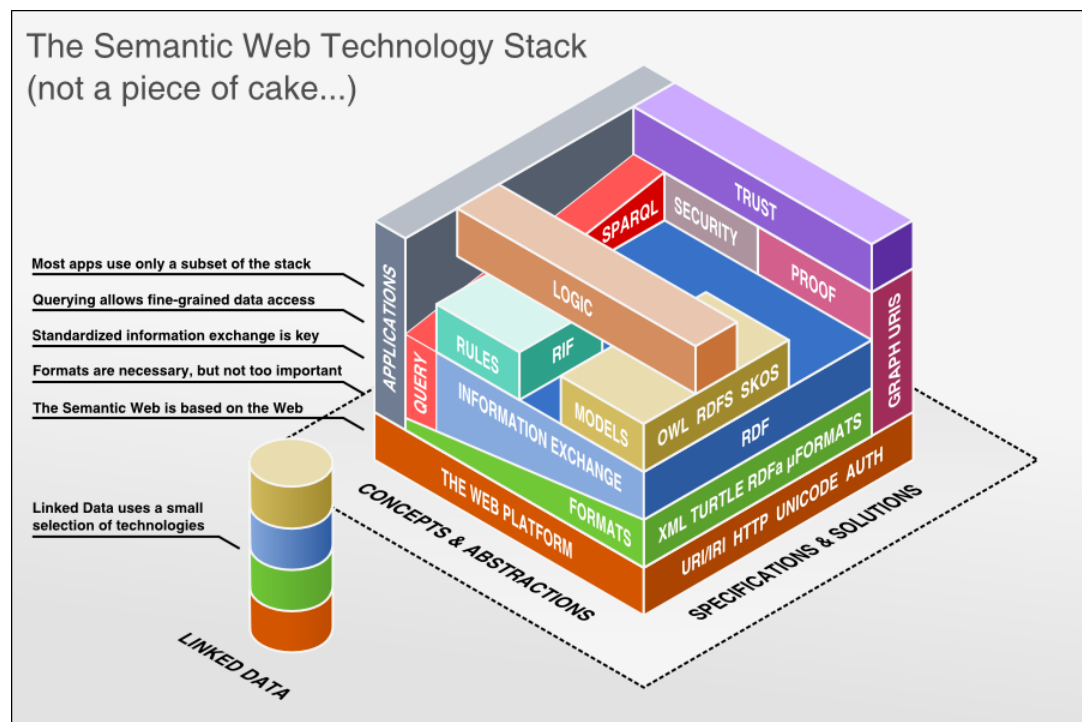


Figure 2.2: The Semantic Web Technology Stack.

This particular figure differs from other illustrations, as the creator tried to emphasize that the Semantic Web Technology Stack is "not a piece of cake", hinting at others that depict a strictly layered two-dimensional stack. This representation is more appropriate as some technologies are orthogonal to others and influence several layers in parallel. On top of the current world wide web standards, the Resource

## 2 Related Work

Description Framework (RDF) is serialized into one of many different formats, like XML, JSON-LD, RDFa and many others. Within RDF, a knowledge base may be defined by using typing and classification provided by an ontology. These are usually created in OWL and use a controlled set of terms. In parallel, SPARQL offers the possibility to query RDF graphs. On top of an existing knowledge-base, logical rules may be defined, to enable reasoners to create new knowledge from existing annotated knowledge. Orthogonal components involve security and trust issues and require authentication and authorization standards.

### 2.1.1 The Web Platform

The foundation of the Semantic Web is the World Wide Web, as has already been explained briefly in the introduction part of this section. This subsection is intended to point out key aspects of the URI, HTTP and HTML standards, which will be used for further considerations in this chapter and those to follow. To get a deeper understanding of the capabilities of these technologies, the corresponding official W3C recommendation documents provide detailed specifications and guidance for implementations.

#### URI/IRI - Uniform/Internationalized Resource Identifiers

In 1994, Tim Berners-Lee introduced the concept of Unified Resource Identifiers (URIs) as a syntax to unambiguously describe an abstract or physical resource of the web.[4] Initially, he also introduced subsets of URIs, namely Unified Resource Locators (URL) and Unified Resource Names (URNs), both then specified in separate documents[5][6]. However, the current definition of URIs [7] deems these distinctions obsolete and the definition of the term "resource" is given a much more generic meaning. According to it, resources can be locations, namespaces, electronic documents, services and many other abstract or even physical things (URIs do not necessarily have to identify web-resources only). To transcribe a URI, only a very limited set of characters is permitted. This set consists of the basic Latin alphabet, digits and a few more special characters, encoded in US-ASCII. However, as globalization is ever-progressing, the need for a more diverse character set is arising. URIs are often compiled in a way that allows them to be remembered easily, hence using words from natural language instead of strings compiled of random characters is a common practice. By following this practice, URIs are further enhanced by semantics, as they are then carrying human-interpretable information. However, for many people that are non-native in English, Latin characters are as meaningless as randomly compiled strings. This issue is often tried to be solved using existing transcriptions, like pinyin for Chinese characters (Hanzi). Herein lies the problem that this adds to further ambiguity in URIs (in pinyin there are only about 400, respectively 1500 unambiguous syllables, depending on differentiating tones or not<sup>1</sup>).

---

<sup>1</sup><http://chinese.stackexchange.com/questions/14596/how-many-syllables-does-chinese-have>



As a solution, Duerst and Suignard from W3C proposed Internationalized Resource Identifiers (IRIs), extending the available set of characters for URIs from US-ASCII only to UTF-8 encoded[8]. As the W3C has shown, to this date UTF-8 is the most used character encoding, making up 87,1% of the web<sup>1</sup>. Thus, it appears to be an obvious decision to incorporate UTF-8 in the syntax to describe resources too. Regarding applicability, the W3C lists three conditions to enable practical use of IRIs[8, p.4]:

- Protocols or format elements have to be explicitly designated to carry IRIs.
- Protocols or formats carrying IRIs have to be able to represent the extensive range of characters.
- A URI corresponding to a certain IRI has to encode original characters into so-called octets using UTF-8.

This means, for example, that according to its current specification HTTP does **not** natively support IRIs as request targets. However, there is a way of mapping an IRI onto a URI, replacing original characters by the hexadecimal notation of their octet value. Duerst and Suignard offer a step-by-step instruction to do so [8, p.10-11]. In section 7, they also offer informative guidelines on how to handle URIs/IRIs. To maximize interoperability they recommend that systems which generate resource identifiers should expose these in a URI using the aforementioned hexadecimal notation for non-ASCII original characters.

Regarding the syntax of URIs/IRIs, RFC 3986[7, p.16] offers the specification for URI composition.

$$\underbrace{\text{http}}_{\text{scheme}} : // \underbrace{\text{www.example.org} : 8080}_{\text{authority}} / \underbrace{\text{over/there}}_{\text{path}} ? \underbrace{\text{service} = 4sq}_{\text{query}} \# \underbrace{\text{venue}}_{\text{fragment}}$$

Only the scheme and path components are required, although they might be empty. All other components are optional.

**scheme** Each URI begins with a scheme name that refers to a specification for assigning identifiers within that scheme. While most schemes are directly named after a specific protocol they are associated with (e.g. http), it is a common mistake to refer to schemes as protocols. Schemes can exist outside of protocols, as can be observed with (e.g. the file-scheme). Within a scheme, the syntax and semantics of a URI can be restricted even further. The registration of a new scheme can be done at the Internet Assigned Numbers Authority (IANA). A detailed description of the registration process can be found in a best current practice (BCP) document, released by the IETF in 2015.[9]

**authority** Most URIs include the identifier of a naming authority that administrates the hierarchical structure and semantics of the rest of that URI (path, query and fragment). The authority name itself is also specified by a generic syntax:

<sup>1</sup>[https://w3techs.com/technologies/overview/character\\_encoding/all](https://w3techs.com/technologies/overview/character_encoding/all)

## 2 Related Work

$$\underbrace{user : unreserved}_{user\ information} @ \underbrace{www.example.org}_{host} : \underbrace{8080}_{port}$$

- User information may contain a user name and other optional scheme-specific information. The use of the format `user:password`, however, is highly discouraged as it is considered deprecated due to security considerations. The host is either an internet protocol (IP) address or a domain name system (DNS) registered name string. The port subcomponent indicates an optional port as used in various transport protocols (TCP, UDP etc.). The type of transport protocol used is defined by the scheme. Furthermore, schemes might also define default port numbers (e.g. 21 for FTP with TCP, 20 for FTP with UDP).
- path** The path component allows to further distinguish hierarchically structured resources within the scope of a URI's scheme and naming authority. Every subpath of the root path in the hierarchy is separated by the "/" character.
- query** The query component allows to further distinguish non-hierarchically structured resources within the scope of the URI's scheme and naming authority. Beyond that, its syntax is not defined. However, in most implementations it carries additional information in the form of `key=value` pairs.
- fragment** The fragment component identifies a subresource of a primary resource by providing additional information. This subresource may be a subset of the primary resource or a part or portion of it. Its interpretation is dependent on the media-type of the representation.

In Semantic Web and Linked Data technologies, IRIs are used to unambiguously identify things and their relations. However, as the generic syntax for IRIs allows for rather long and verbose denotations, scoped compaction mechanisms would allow to reduce verbosity and improve readability for human users. Based on these considerations the compact URI scheme (CURIE) was specified. [10] CURIEs offer a mechanism of compacting IRIs in a local scope (for example within a JSON or XML document) and a mechanism of extending existing CURIEs to denote new IRIs. A CURIE consists of two components, a `prefix` and a `reference`.

$$\underbrace{schema}_{prefix} : \underbrace{Place}_{reference}$$

The `prefix` component must therefore be defined as a valid IRI before, while the `reference` component must be able to be resolved to a valid IRI. In this example, instead of spelling out `http://schema.org/Place`, `prefix` is defined to be `http://schema.org/` while `reference` is set to be `Place`, which then allows to use the syntax `schema:Place` instead. This mechanism is also a key aspect of the JSON-LD specification as will be shown in section 2.1.3.

## HTTP - Hypertext Transfer Protocol

The Hypertext Transfer Protocol (HTTP) has been part of the world wide web from its earliest days on. When Tim Berners-Lee proposed the architecture for the WWW, HTTP was only capable of retrieving HTML documents. The first documented version of HTTP was version 0.9, while the first official RFC at the IETF was released with HTTP version 1.0.[11] In this section, the core features of HTTP are presented, as they will be extensively used for the implementation introduced in course of this thesis. This overview will be based on the specifications of HTTP 1.1.[12][13][14][15][16][17]

In its current version, HTTP/2 [18] these core features are still supported, however, HTTP/2 aims to increase the efficiency of the protocol.

According to its specifications, HTTP is defined as *an application-level protocol for distributed, collaborative, hypermedia information systems*. In addition, HTTPS provides a secure extension for end-to-end secured connections.

Communication via HTTP always involves a **connection** of two programs. Every program may be given the role of a **client**, establishing connections and sending HTTP **requests**, a **server**, accepting connections and serve HTTP **responses** for given requests, or both. Requests and responses are both HTTP **messages**. An HTTP request message begins with a request-line that includes a method, target URI, and protocol version (e.g. http/1.1). Then, header fields containing request modifiers, client information, and representation metadata follow. Finally, after an empty line denoting the end of the header, a message body containing a payload may follow.

An HTTP response begins with a status line that states the protocol version and a success or error code, paired with its textual reason phrase. Then, zero or more headers may follow, containing server information, resource metadata, and representation metadata. The header section is delimited by an empty line again and may be followed by an optional body, containing the payload.

In between a client and a server several intermediaries may exist, essentially categorized into three types: proxies, gateways and tunnels.

**Proxies** are forwarding messages and might also translate them in the process. This translation may even be from another application protocol to HTTP.

**Gateways** are also referred to as "reverse proxies". For outbound connections they act as if they were the origin of messages, internally, however, they may reroute a connection to another server.

**Tunnels** do not change a message's content between two endpoints, but rather deploys a virtually direct connection. A tunnel might be initiated by an HTTP request and ceases to exist when the connection is closed on both endpoints.

A very important feature, especially in regards to the REST architectural approach introduced later, is that HTTP is defined to be stateless, meaning that each message's semantics are understood on its own. Violating this requirement may result in security and interoperability issues.

HTTP is strongly linked to URIs: they are used to target requests, indicate redirects,

## 2 Related Work

and define relationships. HTTP does not support the IRI format, however, as mentioned in the previous section, there is a defined algorithm to map IRIs to URIs.[8, p.10-11]

Moreover, to retrieve and manipulate resource representations, HTTP provides dedicated methods with strict semantic definitions.

For the fundamental functionality of creating, reading, updating and deleting (CRUD) a resource, HTTP offers the following methods(as defined in [13]):

**POST** Perform resource-specific processing on the request payload (typically creating a resource).

**GET** Transfer a current representation of the target resource.

**PUT** Replace all current representations of the target resource with the request payload.

**DELETE** Remove a the target resource from the server.

Additionally, HTTP defines the semantics of the methods HEAD, CONNECT, TRACE and OPTIONS. OPTIONS provides a description of the communication options of a resource, which is useful for the discovery of resources, especially those representing web services. Each resource must support GET and HEAD, all other methods are optional and might therefore not be available for every resource. A method is considered as safe, when its semantics defines it to be read-only. Thus, GET, HEAD, OPTIONS and TRACE are all considered safe. The word "safe" implies that a *method is not expected to cause any harm, loss of property, or unusual burden on the origin server*. Another consideration regarding methods is whether they are idempotent or not. Idempotency is given when a single request has the same effect as multiple requests of the same kind. Of all the methods, every safe method plus PUT and DELETE are considered to be idempotent.

Header fields have several purposes in HTTP. In a request, a header should *provide more information about the request context, make the request conditional based on the target resource state, suggest preferred formats for the response, supply authentication credentials, or modify the expected request processing*. One functionality of headers is to perform content negotiation. Therefore, the header field names `Accept`, `Accept-Charset`, `Accept-Encoding` and `Accept-Language` are specified. They convey the expectations of a client, regarding media-type, character set, encoding and language in the response. Clients may use so-called quality-values to express their preference, e.g. the header `Accept-Language: de, en-gb;q=0.8, en;q=0.7` would indicate that a client prefers German, but would also accept British English and other types of English. Other applications for headers are authentication, controls and conditionals. Response headers provide *information about the server, about further access to the target resource, or about related resources*. To denote the content type of the body of the HTTP message, the `Content-Type` response header may be used. The response header `Location` is sometimes used to hint at a location in relation to a request. For example, when a resource is invoked using the POST method, the `Location` header denotes the URI of the created resource. The `Vary` response header tells a client which content negotiation parameters might provide different representations of a resource.

The body of an HTTP message is hardly specified, depending on the method, a

body may be required to be empty. For HTTP requests the body content type is usually `application/x-www-form-urlencoded`, but there may also be other content types used. In `application/x-www-form-urlencoded` parameters are denoted as key-value pairs, separated by the character `&` and escaped by `+`. HTTP response bodies may be serialized according to an Internet media type, or Multipurpose Internet Mail Extensions (MIME) type. Internet media types may be registered at the Internet Assigned Numbers Authority (IANA) and are denoted in the following pattern:

$$\underbrace{\text{text}}_{\text{type}} \backslash \underbrace{\text{html}}_{\text{subtype}} ; \underbrace{\text{charset = UTF - 8}}_{\text{parameters}}$$

Another feature to convey meaningful responses with defined semantics are HTTP status codes. Each status code consists of a three digit integer with the first digit allowing to classify them as follows:

- 1XX** denotes an informational status. The request was received, processing is continued.
- 2XX** denotes a successful status. The request was received, accepted and processed successfully.
- 3XX** denotes a redirection status. The client is usually redirected to another resource which requires further action to complete the original request.
- 4XX** denotes a client error. The request used bad syntax or cannot be fulfilled in general.
- 5XX** denotes a server error. The request was valid, but the server fails for some reason.

The HTTP specification defines an extensive set of terms and their semantics. However, applications may even extend this scheme by introducing their own status codes.

In addition to the definition of HTTP itself, there exist several proposed standards based on it. **URI templates** are defined as *compact sequences of characters for describing a range of Uniform Resource Identifiers through variable expansion*.<sup>[19]</sup> A URI template is delimited by `{` and `}` and may not be nested. Reserved operator characters from the HTTP specification keep their semantics, variables within a template are comma , separated. A common use of templates is to describe the parameters of a URI with query parameters.

$$\text{http} : // \text{www.example.org} : 8080 / \text{over} / \text{there} \underbrace{\{?latitude, longitude\}}_{\text{parameters}}$$

Another concept based on HTTP is the use of a `Link` header in the response. Proposed by the IETF<sup>[20]</sup>, this header should provide links between resources, independent of their serialization format. As per definition, each `Link` header may contain a context IRI, a link relation type, a target IRI, and optionally, target attributes. These target

## 2 Related Work

attributes comprise key/value pairs that further describe a link. A single resource may provide several links to other resources, and its retrieved representation may therefore carry more than one `Link` header.

### HTML - Hypertext Markup Language

Besides the concepts for how to address resources and how to interact with them, there need to be standards for how to represent them. The first and perhaps most popular one was the hypertext markup language (HTML). Originally, HTML allowed to annotate plain text with metadata, to denote the visual and audible representation of web pages. The current version of HTML is HTML5. While HTML is a markup language specifically representing web pages, there are also data formatting markup languages that are programming language-agnostic and allow to structure arbitrary data. Perhaps the most popular ones in this regard are the Extensible Markup Language (XML)[21] and the JavaScript Object Notation (JSON)[22]. Both allow hierarchical structuring of data.

The basic characteristics of XML may be described as follows:

The most essential definition of XML are `elements`. They may contain data of every kind and may be multiply nested. The name of each element can be arbitrarily chosen. Elements can be denoted in two ways. The first is to define key-value pairs, using a start and an end **tag**:

---

```
<person>
  <name> Markus Staud <\name>
<\person>
```

---

Listing 2.1: Basic XML element annotation

The other way of annotating data is to define it as an attribute of an element:

---

```
<person name=Markus Staud \>
```

---

Listing 2.2: XML element attributes

XML also supports comments:

---

```
<!-- This is a comment. -->
```

---

Listing 2.3: XML comment

Each XML document starts with a `root` or `document` element. This element contains information about what mark-up language and which version of it are following. The official Internet media type of xml is `application/xml`.

JSON's core features can be summed even briefer:

JSON is essentially made of key-value pairs, denoted as follows:

---

```
"person":{  
  "name":"Markus Staud"  
}
```

---

Listing 2.4: Basic JSON key-value pair annotation

Each key may reference one of six datatypes: a plain string, a number, a boolean, an array, an object or simply `null`. Listing 2.5 provides a minimal example for the annotation of each datatype. Each member of an object is separated by a comma. Objects may be multiply nested and are delimited in braces, while arrays are delimited in brackets. JSON defines that members of an objects are unsorted, while members of an array are sorted. This is important to notice, as it is the reason most parsers do not preserve the order among object members.

---

```
{  
  "name":"Markus Staud",  
  "age":24,  
  "student":true,  
  "birthplace":{  
    "city":"Graz",  
    "country":"Austria"  
  },  
  "education":[  
    "VS Peter-Rosegger",  
    "BRG Oeversee",  
    "HTBLuVA BULME Graz",  
    "TU Graz"  
  ],  
  "brother":null  
}
```

---

Listing 2.5: JSON datatype syntax

Other than XML, JSON does not support comments. The official Internet media type of JSON is `application/json`.

JSON is usually considered as more readable for human users. Also, its syntax is more terse than XML's in most cases. Therefore, less data has to be transmitted to convey the same information. Considering tooling, elaborate parsers for both standards are available in every respectable programming language.

### 2.1.2 Data Model: RDF Graph

Before considering formats for the Resource Description Framework (RDF), a basic understanding of RDF is required. This section is mostly based on the official W3C RDF Working Group technical reports. A good overview is provided by the RDF 1.1 primer[23] and normative specifications are found in the official W3C

## 2 Related Work

RDF 1.1. recommendation[24]. RDF provides a simple but powerful framework for representing information. In its core, RDF expresses the relationship of two entities using statements. These RDF statements consist of three fundamental elements: a subject, a predicate and an object. Because of the ternary nature of an RDF statement, they are also called **triples**. Figure 2.3 illustrates a triple in a common way, a node-arc-node link:

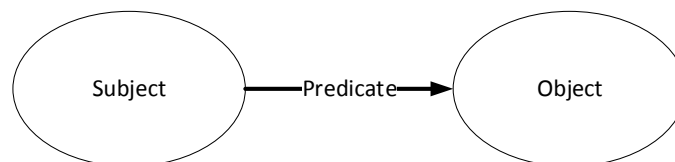


Figure 2.3: Fundamental data model of RDF.

This node-directed arc-node relation can be further extended to a whole set of triples, represented by a graph. The nodes of this graph can be of one of three types: IRIs, literals and blank nodes. Arcs, on the other hand, can only be IRIs as per definition of RDF.

Denoting triples in a code-style, textual format is called serialization. For RDF there exist many different serialization formats, as will be discussed in section 2.1.3.

In RDF every "entity" or "thing" is called a **resource**. These resources are unambiguously identified or denoted by IRIs.

Predicates may only be denoted as IRIs, in RDF called **properties**, and unambiguously state a directed binary relationship between two nodes.

Listing 2.6 provides an example of denoting the relation of subject Daniel knowing object Markus in Turtle syntax, while using the semantics of the schema.org vocabulary.

---

```
http://example.de/Daniel http://schema.org/knows http://example.at/Markus .
```

---

Listing 2.6: Formulation of an RDF statement in Turtle syntax

The owner of an IRI governs its inherent semantics. If third parties are reusing an IRI to describe a relation, the semantics should be abide by them, as there would be a loss in interoperability otherwise. In the example from listing 2.6 the semantics of a publicly open vocabulary are used to foster interoperability. A more detailed definition and delimitation of what vocabularies exactly are in Semantic Web technologies will be discussed in section 2.1.4.

As already mentioned, RDF incorporates three basic types: IRIs, literals and blank nodes.

**IRIs** syntax and specification of IRIs have already been described in detail in section 2.1.1. A specific resource denoted by an IRI is called the IRI's referent. Relative IRIs are allowed, if they can be resolved against a defined base IRI.



**Literals** are concrete values like numbers, strings, booleans or dates. A resource denoted by a literal is the literal's value and therefore called **literal value**. Only objects can represent literal values. Literals consist of two to three components: a lexical form, a datatype IRI and in case the datatype IRI is `http://www.w3.org/1999/02/22-rdf-syntax-ns#langString`, a language tag. The literal value of a literal is thus either:

- a pair of the lexical form and its language tag or
- the lexical form if it can be resolved within the lexical space of the datatype

Datatype IRIs are called **recognized** if they refer to datatypes that can be handled by the RDF processor of an implementation.

**Blank nodes** In the RDF specification, blank nodes are not well-defined, but merely as being *disjoint from IRIs and literals*. Blank nodes are used in RDF implementations to locally identify resource. As such, they have to be unique within their local boundaries to not create ambiguities. They can be useful when the resource itself is either not known, or deemed not to be important. It can also be used to model abstract concepts.

A final example should illustrate most of the concepts in RDF. The example includes several different concepts specified by RDF. It models that the city of Liverpool is home of two football stadiums, Anfield and Goodison Park. While Anfield is of type `schema:Place`, Goodison Park is only referenced by its name, using a blank node. The concept of blank nodes is also used to map different components of the address of the stadium to the IRI identifying Anfield. Here it would just make little sense to identify the address node binding the address components to the stadium with a dedicated IRI. When examining the literals in the graph, the different possibilities of annotation can be observed. The country, the stadium is in, is denoted in two different languages. Here, the datatype is explicitly defined as `xsd:langString`, but most parsers can infer this type automatically, whenever there is a language tag defined. Similarly, if there is only a string to be found as the lexical value of a literal, parsers usually default to the `xsd:String` datatype. Thus, this datatype is omitted in the graph.

Figure 2.4 shows the example in a node-arc-node graph

This very example will be used in the next section to demonstrate the characteristics of each serialization format considered in this thesis.

## 2 Related Work

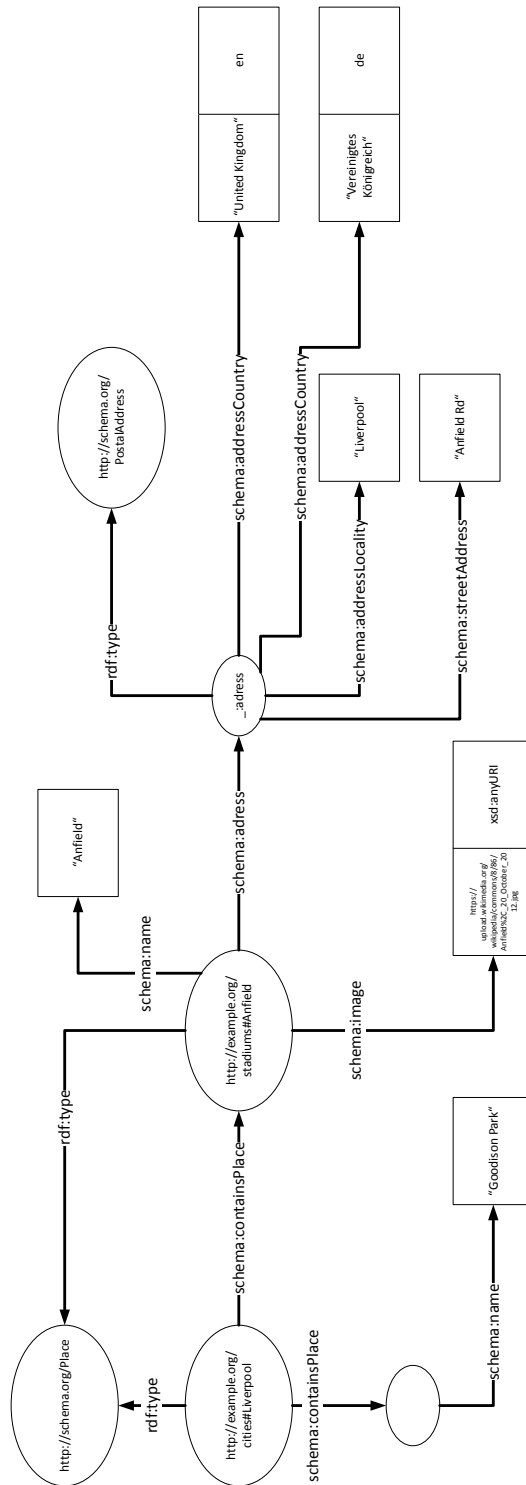


Figure 2.4: Example of an RDF graph.

### 2.1.3 Formats

Following the introduction of triple-based data mapping using RDF in subsection 2.1.2, this subsection provides a short overview of the most popular serialization formats.

As the web in its current form is already using a variety of formats to serialize meta-data, most of them have been only adapted or extended to ready them for semantic web technologies. This is particularly the case with RDF/XML and JSON-LD which are based on already well-known and widely-established serialization standards.

In the following, detailed overviews for the serialization formats of the Turtle family as well as JSON-LD are provided, while RDF/XML and RDFa are only briefly discussed for the sake of completeness. All of the serialization formats can be converted into another. Stolz et al. [25] proposed a useful online conversion tool that provides multi-format conversion for multiple input serialization formats.<sup>1</sup>

RDF/XML was the first syntax developed to serialize RDF graphs, but is not as widespread as all the others, mostly due to its verbosity. One unfortunate circumstance is, that CURIEs may only be used for XML element and attribute names and not in attribute values. Same as for JSON-LD, a big advantage of RDF/XML is that there are numerous standard XML parsers for every established programming language. This enables developers to use already existing components to speed up their implementation cycles. The RDF/XML specification provides further details about its technical aspects.[26] An RDF/XML serialization example for the graph presented in section 2.1.2 can be found in listing 1 in the appendix.

Other than RDF/XML, RDFa creates an additional semantic layer on top of HTML, rather than being a serialization format on its own. As HTML is a well-known standard in the web developers community, its uptake has therefore been much higher. The W3C provides a primer for RDFa[27] introducing its different subsets, RDFa Lite[28] and RDFa Core[29]. RDFa Lite is a rather lightweight specification, only using five simple attributes: `vocab`, `typeof`, `property`, `resource`, and `prefix`. Their semantics offer basic functionality to serialize RDF graphs. As HTML is still the dominant format used to communicate across the WWW, minting HTML documents with semantic annotations has high leverage. RDFa tries to accomplish just that. It provides a format able to convey both human- and machine-processable data. However, it might not always be efficient to provide data in both forms within one resource representation, especially because an average human user cannot process machine-understandable data and a machine cannot process human-understandable data. Thus, it might be useful to separate those representations and obtain the desired one through content negotiation. Same as for RDF/XML, an example for the RDFa serialization of the graph presented in section 2.1.2 may be found in listing 2 in the appendix.

---

<sup>1</sup><https://rdf-translator.appspot.com/>

## 2 Related Work

### Turtle

Turtle [30] was originally derived from another RDF serialization format, N-Triples. For both, extensions to support multiple graphs within a single RDF dataset are available: TriG for Turtle and N-Quads for N-Triples. Turtle uses the same basic syntax as N-Triples, but adds some syntactic sugar to increase readability for human users. Therefore, and for the sake of brevity, only a Turtle-serialized graph will be examined to compare its syntax to the other RDF serialization formats. Considering the example presented in section 2.1.2, the following listing demonstrates its Turtle serialization:

---

```
@base <http://example.org/> .
@prefix xsd: <http://www.w3.org/2001/XMLSchema#> .
@prefix schema: <http://schema.org/> .
<stadiums#Anfield>
  a schema:Place ;
  schema:name "Anfield";
  schema:image "https://upload.wikimedia.org/wikipedia/
commons/8/86/Anfield%2C_20_October_2012.jpg"^^xsd:anyURI ;
  schema:address _:address .
<cities#Liverpool>
  a schema:Place ;
  schema:containsPlace <stadiums#anfield>,
    [ schema:name "Goodison Park" ].
_:address
  a schema:PostalAddress;
  schema:addressCountry "United Kingdom"@en,
    "Vereinigtes Koenigreich"@de;
  schema:addressLocality "Liverpool";
  schema:streetAddress "Anfield Rd" .
```

---

Listing 2.7: Formulation of an RDF graph in Turtle

The keyword `@base` specifies an absolute base IRI which all relative IRIs within a graph definition relate to.

As may also be observed, Turtle enables developers to use CURIEs (see sec. 2.1.1) to allow for a terser serialization of RDF graphs. This results in a better readability, but leads to an increase in complexity for parser implementations. In Turtle, CURIEs are defined by placing the keyword `@prefix` in front of them.

In general, both absolute and relative IRIs are denoted by putting them in chevrons `<>`. CURIEs on the other hand must not be denoted by chevrons.

Every triple is denoted as `S P O` and has to be ended with a dot `.`. Furthermore, to increase terseness even more, there are two shortcuts to express similar triples:

When two or more triples refer to the same subject, using a semicolon `;` to end a triple denotes that the following property and object refer to the same subject. When two or more objects are linked to the same subject, by the same property a comma `,` may be used to end the first triple.

The syntax for denoting literals also allows for some terse annotations: adding an `@`-symbol followed by a qualified language acronym denotes the language tag of a string. If there is no language tag, adding `^^` followed by a datatype IRI denotes the datatype of the literal.

As it is considered to be good practice to type all resources, Turtle offers a special token to abbreviate the `http://www.w3.org/1999/02/22-rdf-syntax-ns#type` property. By simply using `a` instead of the property, the `rdf:type` of a resource is defined.

Blank nodes are denoted by placing `_:` in front of them. Blank node labels have to be unique within a single Turtle document, as they must refer to the same node.

Another possibility to implicitly generate blank nodes is to use nesting, expressed with `[]`. This way, Turtle creates unlabeled blank nodes, which can be useful to quickly denote property lists.

### JSON-LD

JSON-LD is a serialization format for linked data based on the popular data serialization standard JSON. Similar to RDFa it is therefore leveraging an existing, ubiquitous standard and adds a semantic layer on top of it. Its emergence, however, was quite troublesome, as there were different groups trying to achieve the same goals, according to Lanthaler and Gütl.[31]

Originally developed at Digital Bazaar in 2010 and therefore usually accredited to its CEO Manu Sporny, the RDF Working Group at W3C started to work on a JSON-based RDF serialization format named RDF/JSON (similar to RDF/XML).[32] In another place, Lanthaler and Gütl worked on a similar standard with SEREDASj[33]. While Lanthaler and Gütl would soon join forces with the JSON-LD community, there prevailed general confusion and a variety of different views on how JSON-LD and RDF/JSON should differ or complement each other. Consequently, JSON-LD was agreed upon to represent not only linked data, but also to be able to serialize RDF graphs. While this cumbersome process took a lot of discussion in all communities involved, it produced high leverage for JSON-LD in the end. Therefore, the most influential groups agreed upon making JSON-LD a standard, capable of not only expressing linked data but also of annotating semantics in JSON documents. The work on RDF/JSON was discontinued and explicitly not recommended to be used any further.

Currently, the W3C features the JSON-LD specification as an official recommendation [34] and is also providing an official recommendation for processing algorithms and implementation framework.[35]

In the specification of JSON-LD six design goals are stated:

- Simplicity, as no extra processor besides JSON and knowledge about the `@id` and `@context` keyword are needed for basic annotation
- Compatibility, as a JSON-LD document is always a valid JSON document

## 2 Related Work

- Expressiveness, as the data model is a directed graph, which can express almost every real-world concept
- Terseness, as the syntax is terse but still human-readable
- Zero edits, most of the time allowing JSON-LD to be embedded in an existing JSON-based system
- Usable as RDF serialization format

Same as key-value pairs in JSON, JSON-LD documents are compiled of key-value pairs as well. Due to its strong syntactical coupling to JSON, JSON-LD allows values in a key-value pair to be: objects, arrays, strings, numbers, booleans and null. As already mentioned, the data model of JSON-LD is that of a labeled, directed graph, which is exactly the one introduced in subsection 2.1.2 as RDF-graph. On top of JSON, JSON-LD defines a set of reserved keywords, marked by the prefix @ that is an integral part of the standard. Most notable are the @context and the @id keywords:

**@context** is used to define and abbreviate the semantic meaning of keys within a JSON-LD document. Thus, adding context to an existing JSON document enables developers to semantically annotate a plain JSON document without actually editing its original content.

**@id** is used to unambiguously identify a node within a graph.

**@type** defines the data type of either a node or a typed value (that is, a literal in RDF).

**@value** assigns concrete data to a typed value.

**@base** defines a base URI that is used by relative URIs in a JSON-LD document to refer to.

**@graph** denotes a graph in JSON-LD, containing several resources.

Furthermore, there are the keywords @language, @container, @list, @set, @reverse, @index and @vocab, which should not be elaborated here for the sake of brevity.

The key features of JSON-LD involve several concepts:

As has already been mentioned, the context of a JSON-LD document is used to expand the keys in a JSON-LD document, in the JSON-LD specification referred to as "terms", to IRIs. Terms may be arbitrary strings that are not defined as keywords and should, for further extensions, not be prefixed with the keyword-denoting @ character. Using the context, a JSON-LD parser is able to expand a term into its unambiguous IRI. Context may be defined locally within a JSON-LD document or at a remote location. Furthermore, the JSON-LD specification allows referencing of remote context in combination with (re)definition of local context.

The concept of context therefore allows several representation forms of a JSON-LD document: expanded, compacted and flattened. In the expanded form, the context is removed from the document as all terms and vocabulary prefixes are expanded into their full IRIs. Therefore, the expanded form is the most verbose form, with the advantage of having all available context information explicitly denoted within a document.

The compacted form is the opposite of the expanded form. By applying a given context to a JSON-LD file, verbosity is reduced and human readability increased.

Full IRIs are shortened to terms or prefixed terms, supposing appropriate context is provided.

Flattening takes expansion one step further and additionally collects all properties of a node in a single JSON object. Moreover, blank nodes are automatically labeled with blank node identifiers. For certain applications this can simplify processing the JSON-LD document enormously. When passing a context to the flattening algorithm, it returns a flattened and compacted JSON-LD document.

Another highly useful concept is type-coercion. Hence, data type IRIs may be mapped to terms, allowing for a sleeker representation and a better reusability of data type definitions. Of course, data type-coercion only affects the representation of JSON-LD documents in the compacted form. More advanced concepts can be found in the specification of the standard.

To provide a demonstration of JSON-LD, its syntax and how it may be used to serialize an RDF graph, the example presented in section 2.1.2 is serialized into JSON-LD and presented in listing 2.8.

## 2 Related Work

---

```
{
  "@context": [
    {
      "@base": "http://example.org/",
      "xsd": "http://www.w3.org/2001/XMLSchema#",
      "schema": "http://schema.org/"
    }
  ],
  "@graph": [
    {
      "@id": "stadiums#Anfield",
      "@type": "schema:Place",
      "schema:name": "Anfield",
      "schema:image": {
        "@value": "https://de.wikipedia.org/wiki/Datei:Anfield,_20_October_2012.jpg",
        "@type": "xsd:anyURI"
      },
    },
    "schema:address": {
      "@id": "_:address",
      "@type": "schema:PostalAddress",
      "schema:addressCountry": [
        {
          "@value": "United Kingdom",
          "@language": "en"
        },
        {
          "@value": "Vereinigtes Koenigreich",
          "@language": "de"
        }
      ],
      "schema:addressLocality": "Liverpool",
      "schema:streetAddress": "Anfield Rd"
    }
  ],
  {
    "@id": "cities#Liverpool",
    "@type": "schema:Place",
    "schema:containsPlace": [
      {
        "@id": "stadiums#anfield"
      },
      {
        "schema:name": "Goodison Park"
      }
    ]
  }
]
```

---

Listing 2.8: Formulation of an RDF graph in JSON-LD



### 2.1.4 Ontologies

The concept of ontologies originally derives from a branch of philosophy, metaphysics, and is defined as *the philosophical study of being in general, or of what applies neutrally to everything that is real*.<sup>[36]</sup> In computer sciences, ontology refers to a formal, standardized knowledge representation. In practice, developers often struggle to distinguish between **vocabularies**, **taxonomies**, **thesauri** and **ontologies**. A famous overview from Woody Pidcock tries to carve out distinctive features for each specimen.<sup>[37]</sup>

Pidcock describes these terms as follows:

**Controlled vocabulary** refers to a list of terms, controlled by one vocabulary registration authority. Each and every term should be unambiguous in its meaning and carry non-redundant semantics in comparison with the other terms. Pidcock proposes two rules to be adhered to:

1. If the same term is commonly used to mean different concepts in different contexts, then its name is explicitly qualified to resolve this ambiguity.
2. If multiple terms are used to mean the same thing, one of the terms is identified as the preferred term in the controlled vocabulary and the other terms are listed as synonyms or aliases.

**taxonomy** refers to a controlled vocabulary extended by establishing a hierarchy among the terms. Pidcock deems it good practice to limit all relations of multiple children to a single parent to be the same. A multi-hierarchical structure may also be allowed, by relating multiple parents to single child terms.

**thesaurus** refers to a taxonomy with additional associating relationships.

**ontology** may refer to each of the above, but expressed in an ontology representation language. This language defines a grammar using formal constraints to specify how relations can be established among the terms of an ontology.

The W<sub>3</sub>C created a standard to describe and serialize ontologies, based on RDF (see section 2.1.2). The Web Ontology Language (OWL) provides a defined syntax to annotate ontologies, and specifications on how to serialize them in different formats. According to the specifications of its current version (OWL 2) OWL is *designed to facilitate ontology development and sharing via the Web, with the ultimate goal of making Web content more accessible to machines*.<sup>[38]</sup> The illustration in figure 2.5 provides an overview of the main building blocks and structure of OWL.

## 2 Related Work

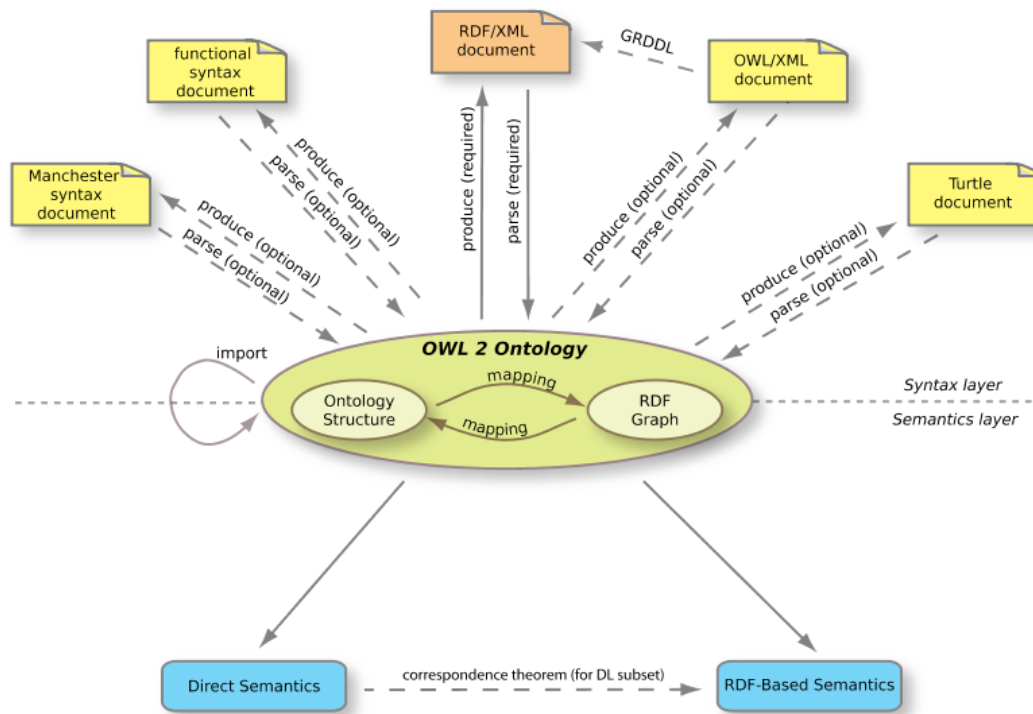


Figure 2.5: The structure of OWL 2.

The center of the graph illustrates the ontology itself, either in its abstract notion, or already mapped onto an RDF graph structure. In the upper "syntax layer", different serialization formats are presented. These are typically the same as for every RDF graph, including JSON-LD. The bottom "semantics layer" should depict the semantics inherent to and expressed by the ontology and its terms.

Ontologies are sometimes categorized into lightweight or heavyweight. This usually refers to how strictly an ontology forces rules on a certain domain. Lightweight ontologies may not aim to completely specify every relation in a certain domain, while heavyweight ontologies may be extensively verbose in their representation of knowledge. Therefore, establishing an ontology is usually a trade-off between how explicitly knowledge should be expressed and how easy applications based on an ontology are to implement.

In practice, there are a few already well-established ontologies/vocabularies. Probably the most famous and verbose one is [dbpedia](http://wiki.dbpedia.org/)<sup>1</sup>. [dbpedia](http://wiki.dbpedia.org/) is essentially an RDF representation of the popular [wikipedia](http://en.wikipedia.org/) online encyclopedia. The problem with [dbpedia](http://wiki.dbpedia.org/) is, that it is too verbose for most applications to be completely recognized. Therefore, a consortium of leading search engine providers, Google, Microsoft, Yahoo and Yandex, created [schema.org](http://schema.org/)<sup>2</sup>. [schema.org](http://schema.org/) defines a set of terms, describing

<sup>1</sup><http://wiki.dbpedia.org/>

<sup>2</sup><http://schema.org/>

basic concepts for things from everyday life. Other popular ontologies are Dublin Core<sup>1</sup> and GoodRelations<sup>2</sup>. Ontologies usually describe terms within encapsulated domains. One of these domain-specific ontologies is Hydra<sup>3</sup>. Hydra tries to provide terms to describes the structure of web resources and how to interact with them. Section 2.5.1 will provide a more detailed discussion of Hydra and how to use it.

### 2.1.5 Reasoning

Reasoning is sometimes referred to as “the engine of semantic web technologies”. Reasoners enable the inference of implicit knowledge from explicit knowledge bases and defined rules or assertions. Therefore, reasoning relies heavily on formal logics and their definition (first-order logic, description logic, etc.).

A simple example would be the following:

A sonOf B.  
B brotherOf C.

Now two rules are defined:

$$\begin{aligned} X \text{ brotherOf } Z \wedge Z \text{ gender "male"} &\rightarrow Z \text{ brotherOf } X \\ X \text{ uncleOf } Y &\iff X \text{ brotherOf } Z \wedge Y \text{ sonOf } Z \end{aligned}$$

Reasoning upon these two triples and the two rules, a reasoner may infer that:

C uncleOf A.

Reasoning upon data may be able to create new relations among resources of a knowledge base, however, depending on the complexity of the formal logic fundamental to the knowledge base, it can also be rather expensive. Apart from that, not every set of logics is decidable, meaning, that not every problem statement can be resolved to a boolean decision within finite time. For a more detailed discussion, see Krötzsch et al. [39].

As no knowledge base can ever be exhaustive in describing real-world relations, assumptions have to be made, whenever a certain relation is not known, neither explicitly nor implicitly upon reasoning. To escape this problem, assumptions about the world outside a knowledge base have to be made. An intuitive way of approaching this problem is the Open World Assumption (OWA). The OWA states, that if something is not explicitly or implicitly known, it is “unknown” or “undefined”. If, for example, a human being does not know whether Vienna is the capital of Austria, it will simply respond with “I don’t know, could be.”. On the other hand, the Closed

<sup>1</sup><http://purl.org/dc/terms/>

<sup>2</sup><http://purl.org/goodrelations/v1/>

<sup>3</sup><http://www.hydra-cg.com/>

## 2 Related Work

World Assumption (CWA) states, that if something is not explicitly or implicitly known, it is false. In this case, if asked whether Vienna was the capital of Austria, a human being would respond with a definite "No."

Obviously, the CWA would lead to a false negative here. However, in programming most data-models are based on the CWA, as unknown results or states can usually not be processed any further by a machine. In Linked Data (sec. 2.1.6) the creators embrace the fact that knowledge is incomplete and are thus assuming an open world (which does **not** mean that every application using LD does or is able to, respectively).

### 2.1.6 Linked Data

While academic workers and standardization groups are busy creating standards for the Semantic Web to enable a global web of semantic data, web developers and enterprises are still somewhat skeptical to use it. Some experts have even deemed it to fail (and still would) as web expert Clay Shirky<sup>1</sup> and various critics in a survey of 895 experts by the Pew Research Center.[40] Lanthaler and Gütl refer to this phenomenon as *Semaphobia*. [33] To bridge the gap between the envisioned Semantic Web and the current state of a web of encapsulated proprietary data silos, merely connected to the infrastructure, the idea of linked data emerged. Introduced by Tim Berners-Lee himself, the renowned creator of the web has actually declared linked data as an integral part of the Semantic Web. However, as can be seen in figure 2.2, modern literature usually refers to linked data as a reduced subset compiled from different layers of the Semantic Web technology stack. To classify whether data is "linked data" Berners-Lee first introduced a basic set of four rules:

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

Therefore, data is only classified as to be "linked data", when it satisfies **all** of these rules.

In 2010 this rating scheme was further extended to a classification scheme for Linked *Open Data*. This was meant to encourage data providers to make their data accessible under an open license. Here, the idea of granting "stars" is pursued, with five stars being the maximum rating. Accordingly, stars are to be "awarded" to data fulfilling the following requirements:[41]

---

<sup>1</sup>[http://www.shirky.com/writings/herecomeseverybody/semantic\\_syllogism.html](http://www.shirky.com/writings/herecomeseverybody/semantic_syllogism.html)

## 2.1 The Semantic Web

- \* Available on the web (whatever format) but with an open license, to be Open Data
- \*\* Available as machine-readable structured data (e.g. excel instead of image scan of a table)
- \*\*\* as (2) plus non-proprietary format (e.g. CSV instead of excel)
- \*\*\*\* All the above plus, use open standards from W3C (RDF and SPARQL) to identify things, so that people can point at your stuff
- \*\*\*\*\* All the above, plus: Link your data to other people's data to provide context

In terms of interlinking data in triples or quads, JSON-LD is highly capable of serializing relations between entities of data. However, accessing this data via defined web interfaces is a completely different challenge, especially when one of the basic principles of the web, to decentralize data, is taken into consideration.

### 2.2 Web Service Architectures

To expose web services to a consumer, a service provider has to make its resources accessible. In general, to enable two software components to communicate with each other, an application programming interface (API) has to be specified. The W3C defines an API as "a software interface that exposes access to some internal functionality of a piece of software for use by programmers, to get access to specific information, to trigger special behavior, or perform some other action." [42] Web APIs are therefore software interfaces that allow programmers to access the functionality of another software explicitly via the world wide web. For simplification reasons, the expression web API abbreviates to API within this thesis. The challenges in providing a service over the web prevails in countless facets: discovery of the service, trust issues, communicating implementation changes, state of transaction, efficient transfer of coarse-grained hypermedia and many more. To guide the provider of a service in the design of web services, architectural patterns apply a defined set of constraints to overcome some of these challenges. However, every constraint is of course a limitation of variance and thus often leads to a trade-off, improving one characteristic while reducing another. In this section, a brief overview of architectural approaches is given.

#### 2.2.1 Classical Service Oriented Architectures

Classical service oriented architectures (SOA) comprise all so-called WS\* standards. The prefix WS here indicates that the standard is part of a web services framework, based on the very first building blocks for web service exposure: SOAP, WSDL and UDDI.

#### SOAP - Simple Object Access Protocol

The W3C currently features SOAP in version 1.2 as a recommendation, hence a detailed specification document is provided. [Mitra.2007] SOAP (originally the acronym for "Simple Object Access Protocol", which was deprecated since version 1.2) is a protocol introduced to be a transport-agnostic way of exchanging information. Therefore, a standardized format was specified: the SOAP envelope. The SOAP envelope is nothing more than a well-defined XML template, containing a header and a body. Per definition, the SOAP message framework consists of four parts:

1. The SOAP processing model
2. The SOAP extensibility model
3. The SOAP underlying binding framework
4. The SOAP message construct

The processing model of SOAP was designed for distributed messaging, involving an initial sender, zero or more intermediaries and an ultimate receiver. Each of those participants is also called a node in SOAP. Via "roles", defined in the envelope's header, each node involved in the transmission of a SOAP message is given permissions to act upon. If permitted, the node then processes all mandatory header blocks and, if he is the ultimate receiver, the body. If a node does not understand a header block, he must return a fault message. An intermediate receiver of a SOAP message can either forward and process it as specified in its header blocks or "actively" forward it by taking actions *not* specified in the header blocks.

The extensibility model of SOAP allows to extend the SOAP messaging framework by SOAP features, messaging exchange patterns and modules. Features can be expressed either as header blocks within the SOAP processing model or within binding specifications in the SOAP binding framework. Message exchange patterns are patterns for message exchange between nodes. Therefore, they are a particular type of feature. Modules specify syntax and semantics of one or more header blocks and thus realize zero or more features.

The underlying binding framework enables to bind the SOAP protocol to underlying protocols. Popular bindings include HTTP, SMTP and FTP. Therefore, SOAP is transport-agnostic, that is, independent of the underlying protocol. The SOAP binding framework provides a specification to realize such a binding to an underlying protocol. This specification allows the "on the wire" representation of the SOAP envelope to be of a different format than XML, despite XML being the defined serialization format of SOAP.

The SOAP message construct defines the structure and syntax of a SOAP message serialized as XML 1.0.

This brief summary should convey a basic understanding of the principles that form the foundation of SOAP, as well as highlighting the key characteristics for further discussion.

### WSDL - Web Service Description Language

The Web Service Description Language (WSDL) is a formal approach to model the behavior of a web service using the XML format. As such it would have also fit perfectly into section 2.3 of this thesis. However, as WSDL, in combination with SOAP and UDDI, is part of the classic framework of SOA, it is described here.

WSDL separates the description model into two stages: an abstract and a concrete stage. At the abstract level, WSDL defines how messages are transmitted and received, independent from underlying protocols. This definition is usually done in XML schema or similar `type` definitions. `Operations` cluster one or more messages by defining message exchange patterns, orchestrating a sequence of incoming and outgoing messages. These operations are then grouped in `interfaces`.

At the concrete level, `bindings` specify how WSDL is bound to underlying protocols like SOAP or HTTP. `Endpoints` then associate network addresses with the previously defined bindings. Finally, `services` group together endpoints that implement the

## 2 Related Work

same interface. The highlighted expressions in this paragraph all denote components of WSDL. These components collectively describe a web service as can be seen in figure 2.6.

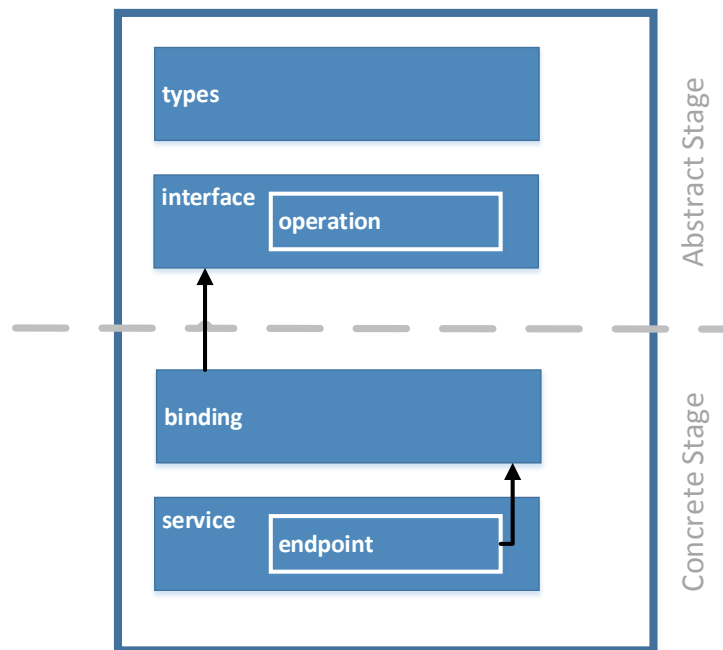


Figure 2.6: The WSDL 2.0 description framework.

However, according to the WSDL specification, only the type definition component is required in a description. Each WSDL document may then include one or more description components. Therein, each WSDL interface represents a set of operations which may or may not be actually invoked during an interaction.

With the emergence of semantic web technologies, there have also been approaches to enable WSDL for semantic annotations. Accordingly, the W3C defined the Semantic Annotations for WSDL (SAWSDL) standard in 2007.[43] Essentially, SAWSDL defines two mechanisms: the `xml-attribute modelReference` creates an association between a WSDL (or XMLS) component and a semantic resource. The `xml-attributes liftingSchemaMapping` and `loweringSchemaMapping` map datatypes in WSDL from plain XML to the semantics of defined IRIs. This way, a conversion between a service's input and output parameters and semantically meaningful terms of an ontology can be created.

Contrary to a common misconception, WSDL may actually be used to describe RESTful web services. However, it was originally intended to describe remote procedure calls (RPCs) to enable clients to correctly invoke a server's functions. Therefore, using it to describe RESTful APIs is usually a lot more complex than with newer description formats, dedicated to REST. For further explanation see section 2.3 ff.



### UDDI - Universal Description, Discovery and Integration

Complementary to SOAP as an access protocol and WSDL as a standardized description of a web service's capabilities and how to access them, UDDI is a registry system to publicly offer web services. UDDI was designed to provide access to registered WSDL descriptions of web services, using SOAP messages. It was intended to provide a central discovery of web services. As both SOAP and WSDL are XML-based, UDDI also relies on the XML standard.

In its current version 3.0 the UDDI specifications define:[44]

- SOAP APIs that applications use to query and to publish information to a UDDI registry
- XML Schema schemata of the registry data model and SOAP message formats
- WSDL definitions of the SOAP APIs
- UDDI registry definitions (technical models - `tModels`) of various identifier and category systems that may be used to identify and categorize UDDI registrations

Thus, UDDI allows service providers and business owners to register services in a registry. Each registration involves three sub-registries, emulating the concept of a telephone directory:

**White Pages** identify each service provider. They include basic information about a company, its data and the services it offers.

**Yellow Pages** offer a classification of businesses and the services they provide by using standardized taxonomies.

**Green Pages** provide technical descriptions of a web service. If multiple bindings are available in the WSDL description of a service, each binding would be listed in a separate Green Pages entry.

However, in practice UDDI did not gain the popularity the creators wished for. Although renowned software companies like Microsoft or IBM provided infrastructure and tooling for UDDI, they shut down their UDDI endpoints after they experienced dire straits.<sup>12</sup> Therefore, at present, UDDI is not used in public services anymore but only internally at a few companies.

---

<sup>1</sup><http://www.infoworld.com/article/2673442/application-development/microsoft--ibm--sap-discontinue-uddi-registry-effort.html>

<sup>2</sup><https://www.innoq.com/blog/st/2010/03/uddi-r.i.p./>

## 2 Related Work

### 2.2.2 REST - Representational State Transfer

A remarkable milestone in API architecture was set by Roy Fielding in his doctoral thesis in 2000.[45, p.76-107] There, Fielding introduced the concept of Representational State Transfer, typically abbreviated to REST. Based on extensive research and a combination of best practice efforts in previous literature, he derived this architectural style with a focus on distributed hypermedia systems. Starting from the "Null Style", an architectural style with no constraints at all, Fielding developed a set of constraints specifying the REST architectural style.

Essentially, REST consists of a set of five mandatory and one optional constraint.

**Client-Server** is the first and basic constraint. The idea is to separate the user interface concern (client) from the data storage concern (server). This allows both components to evolve independently from another, enhancing portability for the user interface and scalability as server components can be simplified.

**Stateless** requires that a server in a client-server communication does not store the state of a session. Therefore, a client must always send complete requests including all the information needed for the server to process them. This principle improves visibility, as no past requests have to be considered in monitoring systems, reliability, as partial failures are detected and fixed easier, and scalability, as session states don't have to be stored and managed by the server. The downside of this constraint is, however, that data has to be submitted to the server repetitively, as shared context cannot be stored on the server. Moreover, a server cannot monitor whether the application is consistently executed on the client-side, as it has no information about its state.

**Cache** allows clients to reuse response data from past requests whenever it is explicitly declared as cacheable. Consequently, some interaction can be avoided completely, meaning that the server can tend to other requests. Improvement in efficiency, scalability and user-perceived performance can be achieved. However, reliability could decrease due to inconsistency if the cached data is not up-to-date.

**Uniform Interface** represents the most important constraint of REST. Component interfaces are generalized and thus offer improved visibility of interactions and a simplified system architecture. The application on the server is decoupled from the interface itself, which, again, enables client and server implementations to evolve independently. A drawback using this principle is a reduction in efficiency, as information is communicated in a standardized format, often including overhead information not needed in a specific application. To implement this generality of interfaces, Fielding introduces four key principles:

1. Identification of resources
2. Manipulation of resources through representations
3. Self-descriptive messages

4. Hypermedia as the engine of application state (commonly abbreviated to HATEOAS)

As these principles have to be adhered to later in this work, a detailed examination will be provided later in this section.

**Layered systems** allow an architecture to comprise a hierarchy of layers. Each component has only components in its scope which it interacts with. This way the system knowledge is restricted to a single layer and complexity is reduced. By introducing intermediaries in higher layers, scalability can be improved by load balancing and shared caching. Intermediaries also enable additional security measures, such as firewalls. The only negative aspect in using layered systems may be increased latency, reducing user-perceived performance. REST also allows intermediaries to actively modify the content of messages, given that they are self-descriptive and their semantics are visible to intermediaries.

**Code on-demand** is an optional constraint. It allows additional functionality in clients by downloadable code segments. Therefore, clients need an engine able to execute these code snippets, for example a browser that supports JavaScript. If clients do not possess said engine, they should still be supported in a way that they would not break, as this constraint is considered optional.

APIs that comply to at least all the mandatory constraints are referred to as "RESTful". Finally, but probably most importantly, Fielding's goal was *"to create an architectural model for how the Web should work"* itself, trying to preserve *"the core constraints that make the Web successful"*.

### The four interface constraints of REST

As a central feature of REST, uniform interfaces between components are required. Thus, the following four key-principles have to be adhered to when designing a RESTful API:

**Identification of resources** is done by a naming authority, which has to make sure that the semantic validity of the assigned resource identifier remains intact over time. A resource does not have to necessarily exist yet to be addressed, templates for example may address only the concept of a resource. In HTTP, the concept of URIs is used to identify resources.

**Manipulation of resources through representations** a representation consists of data, metadata describing the data, and metadata to describe the metadata. A representation of a resource captures its current state. However, a representation can also be used to express the intended state of resource. Thus, a uniform interface may manipulate a resource according to the representation that was applied to it. In HTTP a resource's representation is retrieved via invocation of HTTP

## 2 Related Work

GET. This representation then may be altered by a client and transferred back to the server using HTTP PUT. The server then applies the alterations to the resource itself, resulting in a manipulation of the resource via representation.

**Self-descriptive messages** are enforcing stateless interaction between requests. Standard methods (in HTTP: GET, POST, PUT, etc.) and well-defined media types (ideally publicly registered at the IANA) are used to convey semantics and information. No information from previous requests must be used, the entire state handling is moved into the domain of the client.

**HATEOAS** defines that a server has to interact with a client by exposing dynamically created hypermedia descriptions of a resource. Depending on the state of a resource, a server may provide different responses. Moreover, in addition to a current representation of the value(s) of a resource, the server provides ways to interact with the resource, in a self-descriptive way. Therefore, interaction is driven by exposing state transition options in an hypermedia format. Examples and a more practical description will be provided in section 2.4.

### The Richardson Maturity Model

In their book Webber et al.[46, p.19] introduce an interesting concept for classifying a web services' maturity level. They refer to it as the Richardson Maturity Model (RMM), named after its creator, Leonard Richardson. Richardson defined four stages of "maturity" of an API, classifying to which extent an API adheres to REST principles. Figure 2.7 provides an illustration of the model (illustrated by Martin Fowler).

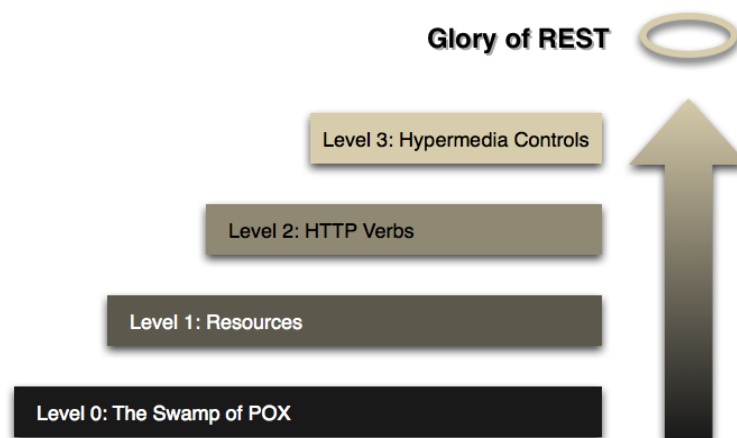


Figure 2.7: The Richardson Maturity Model.

As depicted in figure 2.7, the basic stage is level 0 or “the swamp of POX” (plain old XML). To classify for level 0, an API only has to support HTTP as communication protocol, but not use any of its inherent semantics. The exchanged HTTP messages are essentially XML (or any other mark-up language, like JSON) forms that have to be known by both client and server a priori, both in syntax and semantics. The server is contacted via a single address and there are no other URI-identified resources that the server handles. HTTP is therefore only used to transmit the data, but useful mechanisms as the resource data model, content negotiation or method and status code semantics are not used.

Level 1 of the Richardson Maturity Model is achieved by introducing resources. Instead of calling all procedures and retrieve every object from a single URI, resources provide a more structured data model. Instead of always invoking the same endpoint with different parameters, the functionality of the endpoint is distributed among resources, which embody more specific functionality.

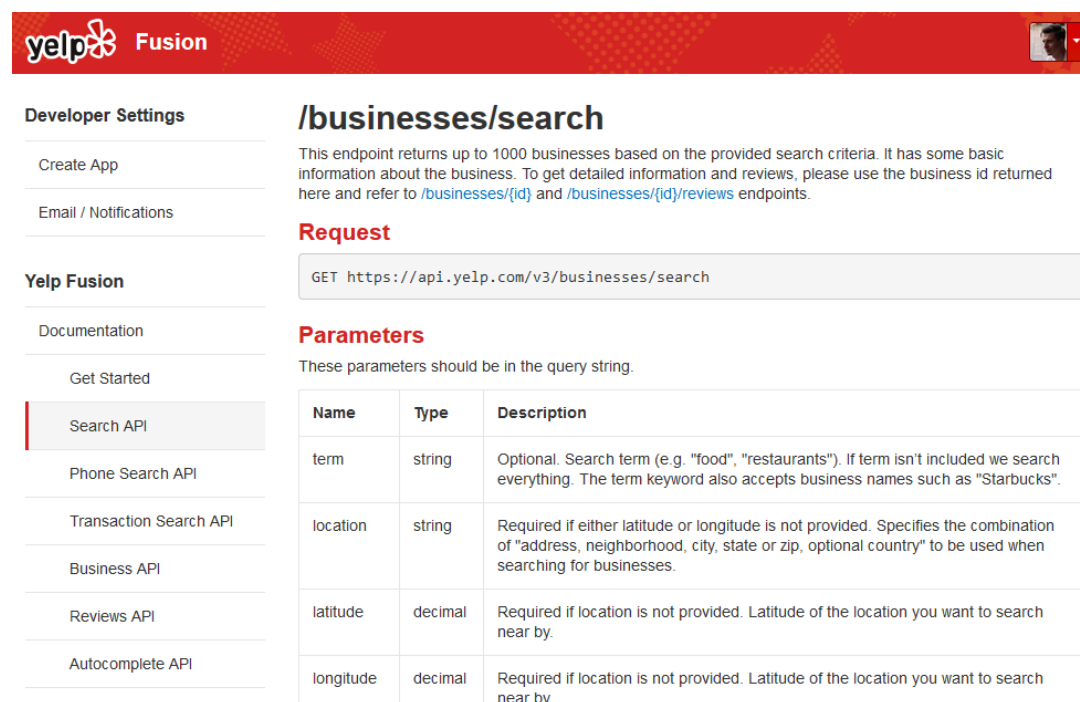
The next step towards the glory of REST lifts an API to level 2 of the Richardson Maturity Model. Instead of invoking each resource with the same method, the semantics of an operation may be conveyed terser by using the methods defined according to their HTTP specification. To retrieve a resource, HTTP GET should be used, to create a new resource a POST or a PUT may be disposed (see section 2.1.1 for details).

The final step to reach the glory of REST is achieved when an API is using HATEOAS. This means that a client only needs the entrypoint of an API and may then explore it by “following its nose”, meaning that each resource provides a description of itself, operations that it supports and links to access related resources in a meaningful way.

## 2.3 Static Web Service Description

Currently, the most popular style of providing documentation for an API is in plain textual form. This could, for example, be in the form of an HTML-document or a PDF-file. This way, developers are (ideally) supplied with a verbose description of how to invoke and interact with an API. They process the semantics of the content and create client applications according to their understanding. As this kind of a description does not involve any specified syntax, a machine cannot process its content.

Figure 2.8 provides an example, taken from the documentation of the current version of the Yelp Search API.



The screenshot shows the Yelp Fusion API documentation for the `/businesses/search` endpoint. The page is divided into a sidebar and a main content area. The sidebar contains sections for 'Developer Settings' (with links for 'Create App' and 'Email / Notifications'), 'Yelp Fusion' (with links for 'Documentation', 'Get Started', 'Search API', 'Phone Search API', 'Transaction Search API', 'Business API', 'Reviews API', and 'Autocomplete API'), and the 'Search API' link is highlighted. The main content area features the endpoint title `/businesses/search`, a description stating it returns up to 1000 businesses, a 'Request' section showing the GET request `https://api.yelp.com/v3/businesses/search`, and a 'Parameters' section with a table of query string parameters.

Name	Type	Description
term	string	Optional. Search term (e.g. "food", "restaurants"). If term isn't included we search everything. The term keyword also accepts business names such as "Starbucks".
location	string	Required if either latitude or longitude is not provided. Specifies the combination of "address, neighborhood, city, state or zip, optional country" to be used when searching for businesses.
latitude	decimal	Required if location is not provided. Latitude of the location you want to search near by.
longitude	decimal	Required if location is not provided. Latitude of the location you want to search near by.

Figure 2.8: Exemplary web service description in plain text.

To describe web services in a machine-readable syntax, a broad spectrum of proposals have emerged, based on both SOAP and REST principles.

WSDL (see section 2.2.1) was the first one introduced and widely used in combination with SOAP. [47] Metadata descriptions of web services provide a full documentation of the capabilities and accessible resources of an API. The approach of metadata descriptions for RESTful APIs has already been mentioned when examining WSDL. These descriptions can be downloaded by API designers at design time which then design a client application upon them. However, these so-called "out-of-bound" descriptions (as they have to be obtained outside of runtime interaction with an API) are widely considered to violate REST, where interfaces are required to be

self-descriptive at runtime. Thus, even if these API design frameworks claim to be RESTful, they may only conform to the second level of the Richardson Maturity Model, exploiting HTTP verbs and URI identified resources.

As the main motivation of these frameworks is to specify a machine-readable syntax, they must be able to describe the following aspects:

- A structured representation of the resources of an API.
- Links between resources.
- HTTP methods that can be applied to each resource and their expected inputs and outputs.
- Supported content/media types and the data schemas that are accepted and exploited by an API.

A formal description would then enable the automatic generation of client and server stubs, provide a portable format to configure clients and servers and enable visualization tools to provide structured graphical representations of an API.

WADL, RAML and Swagger all provide frameworks for creating resource paths and defining resources and their CRUD actions. In this section, the main characteristics of WADL, as the first of its kind, and Swagger, as the currently most popular, are discussed.

### 2.3.1 WADL - Web Application Description Language

The Web Application Description Language (WADL) was created to describe RESTful APIs, based on a resource-centred view. Originally, it was proposed as the REST version of the more with classic SOA associated WSDL (see section 2.2.1). The standard proposal has been officially submitted to the W3C in 2009, however, it has never been awarded the status of an official W3C standard.[48]

WADL uses XML as its serialization format. Each WADL description exhibits an `application` element as its root element. WADL supports the use of CURIEs to provide better readability for human readers. Dataformats that will be exchanged while interacting with the API may be declared using the `grammars` container element. The `resources` element contains all the resources provided by the API and is attributed with the `base` URI of the API. A single `resource` may be attributed by an `id`, a relative `path`, a `type` and a `queryType`. A `resource` may contain `param` elements that can be further specified as `header`, `query`, or `template` parameters. A `resource` may also specify `method` elements that describe the HTTP methods that can be applied to the resource. Moreover, `resource` elements may contain other `resource` elements, which are then hierarchical subresources of a resource.

Listing 2.9 illustrates the use of the WADL syntax by exemplifying the Yahoo News Search API.

## 2 Related Work

---

```
<?xml version="1.0"?>
<application xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
  xsi:schemaLocation="http://wadl.dev.java.net/2009/02 wadl.xsd"
  xmlns:tns="urn:yahoo:yn"
  xmlns:xsd="http://www.w3.org/2001/XMLSchema"
  xmlns:yn="urn:yahoo:yn"
  xmlns:ya="urn:yahoo:api"
  xmlns="http://wadl.dev.java.net/2009/02">
  <grammars>
    <include
      href="NewsSearchResponse.xsd"/>
    <include
      href="Error.xsd"/>
  </grammars>
  <resources base="http://api.search.yahoo.com/NewsSearchService/V1/">
    <resource path="newsSearch">
      <method name="GET" id="search">
        <request>
          <param name="appid" type="xsd:string"
            style="query" required="true"/>
          <param name="query" type="xsd:string"
            style="query" required="true"/>
          <param name="type" style="query" default="all">
            <option value="all"/>
            <option value="any"/>
            <option value="phrase"/>
          </param>
          <param name="results" style="query" type="xsd:int" default="10"/>
          <param name="start" style="query" type="xsd:int" default="1"/>
          <param name="sort" style="query" default="rank">
            <option value="rank"/>
            <option value="date"/>
          </param>
          <param name="language" style="query" type="xsd:string"/>
        </request>
        <response status="200">
          <representation mediaType="application/xml"
            element="yn:ResultSet"/>
        </response>
        <response status="400">
          <representation mediaType="application/xml"
            element="ya:Error"/>
        </response>
      </method>
    </resource>
  </resources>
</application>
```

---

Listing 2.9: Example of a WADL service description



The intent of the creators of WADL was to complete four goals:

1. Support for development of resource modeling tools for resource relationship and choreography analysis and manipulation.
2. Automated generation of stub and skeleton code and code for manipulation of resource representations.
3. Configuration of client and server using a portable format.
4. A common foundation for individual applications and protocols to re-use and perhaps extend rather than each inventing a new description format.

### 2.3.2 Swagger, RAML and API blueprint

While the uptake of WADL in real-world implementations has been rather low, other formats evolved from it. Swagger, the RESTful API modeling language (RAML) and API blueprint all deviate from WADL.

Swagger was originally created by **Wordnik** an online dictionary for English words and a nonprofit organization as such. In 2015 its specification was donated to the Open API Initiative making it open source and thus open for everyone to contribute. According to their official website, their goal is to *"define a standard, language-agnostic interface to REST APIs which allows both humans and computers to discover and understand the capabilities of the service without access to source code, documentation, or through network traffic inspection."*[49]

To achieve this goal, Swagger provides extensive tooling and supports two different design approaches:

1. The top-down approach involves the Swagger Editor to create a Swagger description of the API and the Swagger Codegen tool which can then be used to automatically generate server and client stubs.
2. The bottom-up approach uses an existing REST API and creates a Swagger definition from it, again using the Swagger editor. This can be done either manually or automatically if the API was implemented using a supported framework.

While Swagger offers open source tools for editing description files and code generation, there are also numerous commercial tools available.

As mentioned in the beginning of this section, Swagger originally deviated from WADL. Other than WADL, however, Swagger specifies JSON and YAML (YAML Ain't Mark-up Language), a superset of JSON but emphasizing indents, to be used as its serialization formats.

The basic skeleton code of a Swagger description is illustrated in listing 2.10

---

```
swagger: '2.0'  
info: <version, author, license info>  
host: api.example.org  
basePath: /v1  
schemes: <http, https, ws, wss>
```

## 2 Related Work

```
consumes:  
produces:  
  - application/json  
paths:  
  /payments/payment  
    get:  
      parameters:  
      responses:  
definitions:  
parameters:  
responses:  
securityDefinitions:  
security:  
tags:  
externalDocs:
```

---

Listing 2.10: Swagger skeleton code

Every Swagger document starts by providing a `swagger` key, stating the Swagger version it complies to. The most current version of Swagger, at present, is 2.0. An `info` key provides metadata about the API, like API version, author and license info. The `host` key contains the authority URI of the API, `basePath` a URI relative to the host representing the base path of the API every relative path URI defined in the description refers to. The protocols supported by the API are stated in `schemes`, whereas the Internet media-types consumed and exposed by the API are defined in `consumes` and `produces`, respectively. Resource identifiers and the operations that they support are contained in the `paths` object. Swagger supports the definition of complex types by defining so-called schemas in the `definitions` object. Data types in Schema are based on primitive types supported by JSON-Schema.<sup>[50]</sup> JSON Schema is a declarative format for describing data (therefore a meta-metadata format). Reusability is enforced by defining global `parameters` and `responses` that may be referenced in the parameter and response keys of a single operation in the `paths` objects.

Similar to Swagger, RAML provides a specification for resource-oriented API description, focusing on complying to the DRY("Don't repeat yourself!")-principle. Thus, more features enabling developers to reuse code are included in the specification. Other than Swagger, in its current version (1.0) RAML only supports the top-down approach of specifying and developing REST APIs. Moreover, RAML is specified to use only JSON. A package for the **Atom** editor supports syntax checking and, same as for Swagger, libraries exist to generate server and client stubs for a variety of programming languages.

As with Swagger and RAML, API blueprint provides similar extended functionality of WADL. However, other than the others, API blueprint is defined as to be using the Markdown Syntax for Object Notation (MSON).

## 2.4 Dynamic Web Service Description

A more recent approach to describe APIs is the concept of so-called hypermedia APIs. Instead of providing one exhaustive metadata description in a single document a priori, resources, and methods to interact with them, are described dynamically in the process of the client interacting with the resources. This is exactly what Roy Fielding stipulated with the HATEOAS constraint. A client may only possess knowledge about an API's endpoint and may then interact with the API by only acting upon the description provided in a response.

To get a practical understanding of the HATEOAS constraint, a fictional banking API might be considered. A user might retrieve a representation of his or her bank account by performing an HTTP GET. In the response the user gets his account balance, as expected. Without any further information conveyed to the client a priori, it does not know how to progress. What if the user wants to transfer money to his account? Or from his account? How would the client know if and how this is possible? The solution is, to not only respond with the account balance, but also provide additional information on which next steps might be followed. Depending on the account balance, if it is positive, a client might be provided with options to transfer money both to and from the account. However, in case the account balance is negative, the server may only provide the option to transfer money to the account. Assuming a client knows how to interpret the instructions for transferring money, a server implementation might be updated to then also support the retrieval of a detailed collection of transaction from the last 30 days. Consequently, after retrieving the account balance, the server provides the options to transfer money and, in addition, get a list of past transactions. The client, however, was not updated to semantically comprehend and act upon the new option. Still, it does not break (meaning that all other interaction would also fail) as the server only provided the new options in addition to the existing ones.

To serialize these capabilities of the HATEOAS principle, there have been several proposals for formats. One of the most lightweight and popular ones is the Hypertext Application Language (HAL). Besides HAL, there have also been other, more expressive approaches that are now being used in applications. The most notable ones in this regard are [SIREN](#), [Collection+JSON](#) and [MASON](#), all of which are registered as vendor-specific Internet media-types at the IANA.

## 2 Related Work

### 2.4.1 HAL - Hypertext Application Language

One proposed standard to implement the HATEOAS principle is the Hypertext Application Language (HAL).[51] In one sentence, its creator describes HAL as *a set of conventions for expressing hyperlinks in either JSON or XML*.

The conventions of HAL are designed to be simple, generic and working across different domains. HAL tries to implement HATEOAS by providing a uniform connector interface, decoupling servers and clients. Its dedicated media type `application/vnd.hal+json` defines a document to be HAL compliant and, by now, JSON is the only supported serialization format.

In its current version, HAL defines resource representations as Resource Objects with two different properties:

**\_links** contains single Link Objects or Link Object arrays, IRIs linking to other resources

**\_embedded** contains Resource Objects embedded within the current object

A resource object in HAL is not required to exhibit linked and embedded properties, however. Therefore, empty JSON objects are conforming to HAL as well.

Figure 2.9 illustrates the data model for each resource object:

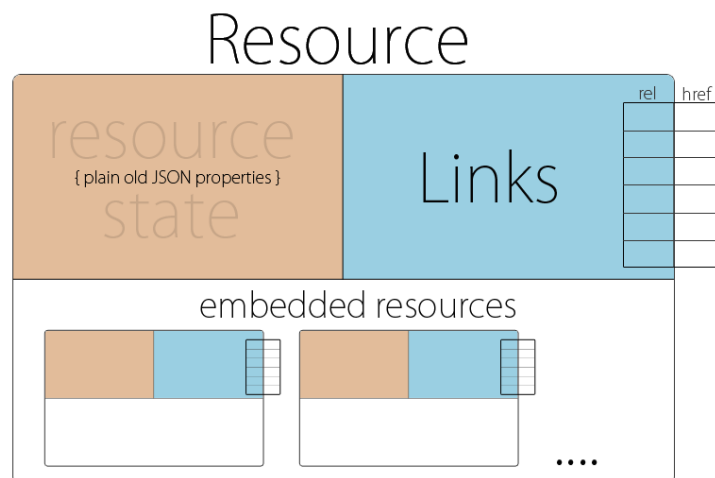


Figure 2.9: The fundamental data model of HAL.

As depicted in figure 2.9, each HAL resource object may consist of three entities: links exhibiting references to related resources, embedded resources, and properties of the resource itself (in plain JSON). To get a practical understanding of how to apply HAL to a real-world scenario, a similar example as the one introduced in the beginning of this section may be considered.

---

```
GET /accounts/9813497 HTTP/1.1
Host: bank.org
Accept: application/hal+json

HTTP/1.1 200 OK
Content-Type: application/hal+json

{
  "_links": {
    "self": { "href": "/accounts/9813497" },
    "next": { "href": "/accounts/9813497?page=2" },
    "paymentOrders": { "href": "/accounts/9813497/paymentOrders" },
    "movement": { "href": "/accounts/9813497/movements{?id}",
      "templated": true }
  },
  "_embedded": {
    "movements": [
      {
        "id": "m82374",
        "amount": -87.00,
        "currency": "EUR"
      }
      {
        "id": "m82374",
        "amount": 7.00,
        "currency": "EUR"
      }
    ]
  }
  "balance": 1712.90
  "currency": "EUR"
}
```

---

Listing 2.11: Exemplary account balance HAL resource

The example illustrates the response retrieved by a user checking his or her bank account. As a human user might guess from the link referencing a `next` resource, a single resource here embeds a subcollection of account movement resources. Moreover, a client may create payment orders by following the `paymentOrders` link. To obtain a more detailed view of a single account movement, instructions of how to create the associated URI template are provided in the link `movement`.

In conclusion, HAL creates an additional layer of defined semantics on top of JSON. A client still has to understand the semantics each link, embedded resource and plain JSON property provides. This rules out the use of automated machine clients, as plain strings do not inherent an unambiguous meaning for them. In theory, HAL does not prohibit the use of IRIs in JSON keys or values. It does, however, not provide any guidelines for their application either, unlike JSON-LD. This leaves HAL rather inept for semantic web technologies and the RDF data model.

### 2.5 Semantic Web Service Description

In their fundamental publication coining the term “Semantic Web” [3] Berners-Lee et al. present the idea of locating and interlinking services (or as they call them, “*programs that perform a specific function*”) by using Semantic Web technologies. Moreover, they suggest depositing some sort of service description in directories analogous to the Yellow Pages (or similar to the approach of UDDI).

In the previous sections, API description formats as Swagger and RAML, as well as dynamic hypermedia formats like HAL have been discussed. In this section, various proposals taking into account the Semantic Web approach for describing web services should be examined. While conventional API description languages (WADL, Swagger, etc.) offer a machine-*readable* way of documenting an API’s capabilities, they do not carry any machine-*understandable* semantics. This does not enable automatic service discovery and composition. When interacting with a web resource, using the HTTP protocol, the constraints of REST govern that only methods defined within the HTTP standard are to be used (using self-defined methods would imply the use of “out-of-band” information). The semantics of these methods are well-defined and can thus be understood by both server and client. Some of the methods may carry different headers, query parameters and/or input parameters. In an automated service composition scenario, these parameters may be subject to requirements that need to be met, which means they also have to be described semantically. Moreover, when a method is applied to a resource, its effect is often depending on its input parameters. This means that the resulting state after applying a method to a certain resource has also to be described semantically depending on the input parameters of the method. To differentiate between the meaning of dynamic and semantic web descriptions in this thesis, the following may be considered: a dynamic web-service description may not necessarily use RDF and IRIs to describe itself. As with HAL or SIREN, they may only specify a set of reserved keyword strings with a defined meaning. A semantic web-service description, on the other hand, is incorporating the HATEOAS approach paired with ontologies to disambiguate the vocabulary used to describe an API. Relevant examples discussed in this section are Hydra and RESTdesc.

#### 2.5.1 Hydra

Hydra is essentially a controlled vocabulary intended to offer a lightweight set of terms to enable semantically unambiguous descriptions of HATEOAS-supporting APIs.[52]

As a vocabulary, Hydra is not bound to any Semantic Web serialization format, although it is often associated with JSON-LD. Hydra is yet to be announced as a standard, therefore there is still a lot of discussion within the community, regarding how to represent API descriptions. Hence, a number of the concepts, ideas and proposals in the current Hydra specification might change, depending on which

ideas are backed by the most consensus. In this thesis, the unofficial draft of the Hydra specifications from June 2016 is the reference for all discussion.

The basic concept of Hydra is to provide unambiguous terms to enable HATEOAS-conforming interaction of a client with a resource exposed by a server. A client recognizing the terms of the Hydra vocabulary may then be able to interact with a server by generating the relevant HTTP requests.

Figure 2.10 tries to provide a graphical overview of the classes and properties defined in Hydra. Per definition, Hydra complies with the Open World Assumption, therefore not forcing its users to use class definitions as they are. However, implementation of the OWA in clients is not feasible and it is strongly recommended to use the data-model provided by Hydra as-is.

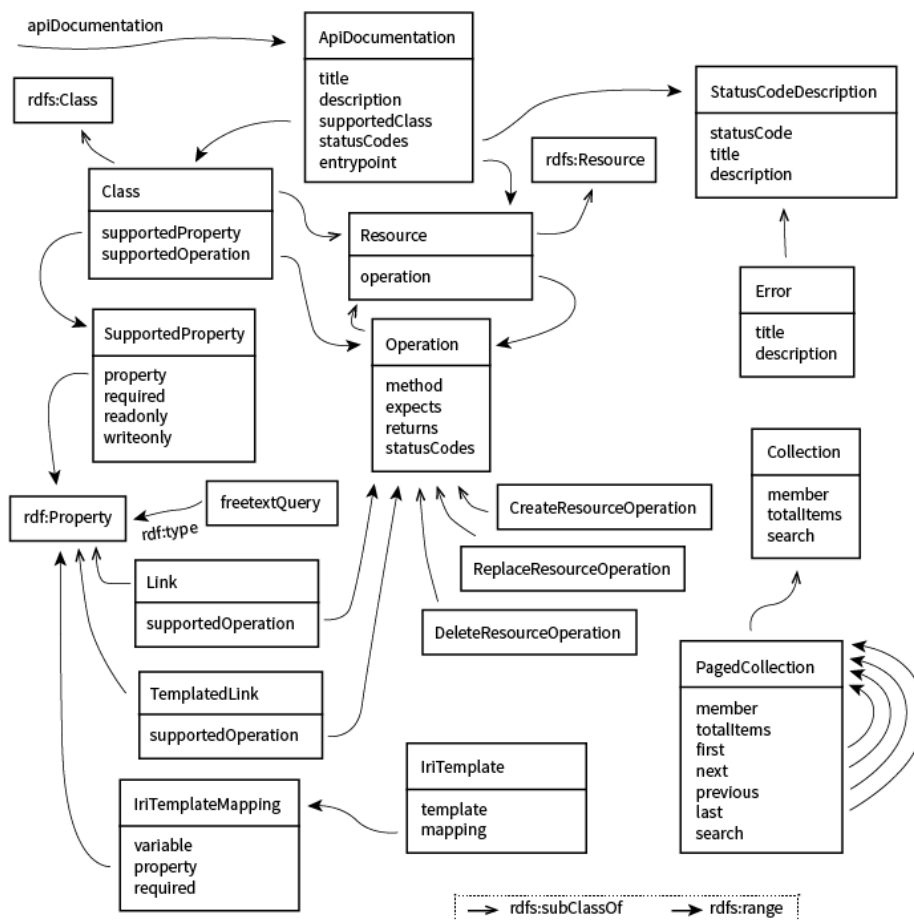


Figure 2.10: Graph illustrating the Hydra core vocabulary in its current form.

One key information missing in this illustration is the range definition for each property. The range of a property defines which types of objects are expected to be referenced by a property. The extensive ontology definition of the Hydra core vocabulary is obtained by retrieving Hydra's official JSON-LD remote con-

## 2 Related Work

text<sup>1</sup>. In the following description of the semantics of Hydra, Hydra's base URI <http://www.w3.org/ns/hydra/core#> is compacted by the CURIE `hydra`.

For the documentation of an API, Hydra offers the `hydra:ApiDocumentation` class. Every object in Hydra may be annotated with a human-readable `hydra:title` and `hydra:description`. Objects typed `hydra:ApiDocumentation` exhibit a property `hydra:entrypoint` which refers to the actual resource described. The referred resource itself may then be of type `hydra:Resource` or a subclass of it.

A resource may be invoked using different HTTP methods. The invocation and parameterization of these methods may be defined in an object of type `hydra:Operation` and related to the resource by the property `hydra:operation`.

`hydra:Class` definitions allow to describe the type of resources that are exposed by the server. A class describes what properties its instantiated objects exhibit and which operations they support. The properties supported may then be annotated according to whether they are required, read-only or write-only.

In its current version, Hydra also defines a concept for IRI templates in `hydra:IriTemplate`. As templates incorporate variables as strings, those have to be mapped to properties using `hydra:IriTemplateMapping`.

Resources often provide a collection of other resources. In Hydra the `hydra:Collection` and `hydra:PagedCollection` types annotate a collection's members, respectively a delimited view of members of a collection.

To annotate HTTP status codes, hydra offers little more semantics than the HTTP description itself. A statuscode may be describe only by its integer code, and a textual title and description.

Hydra also enables discovery of web services by a mechanism using the HTTP `Link` header field. An API might therefore be discovered by providing the link to its documentation <http://www.w3.org/ns/hydra/core#apiDocumentation> in the link header field. From there, a client may traverse the resources of an API.

As Hydra is still incomplete, important aspects of an API like authentication and authorization are both not yet covered in the current version. Nevertheless, Hydra proposes an intriguing approach to integrate API descriptions in linked data technologies.

### 2.5.2 RESTdesc

RESTdesc was created to provide a defined standard for both denoting a service's description and enabling its discovery.[53] Herein, the creators of RESTdesc explicitly state, that its focus is on the *functionality* of a web service, rather than the technical process of invoking a service. RESTdesc is serialized in Notation3 (N3), a superset of Turtle, thus very similar but with extended functionality support (and as such even superseding the RDF specification). N3 also offers support for most reasoners, thus allowing inference of new services compositions. These could potentially offer new functionality, conjuring a vision of automatic service composition.

---

<sup>1</sup><http://www.w3.org/ns/hydra/context.jsonld>



## 2.5 Semantic Web Service Description

On a technical level, RESTdesc describes the semantic of a web-service by so-called pre- and postconditions in N<sub>3</sub>. It also offers inherent support for the HTTP `Link` header, the HTTP `OPTIONS` method and URI templates (see section 2.1.1 for short references). In their article, Verbourgh et al. demonstrate the capabilities of RESTdec with a simple example. Subject of the example is a web service that exposes photographs by performing a request on the resource `/photos/`. It starts with an informal expression to describe the functionality of a service:

*I can retrieve a photo by going to `/photos/` and appending its identifier.*

Mapping this expression in RDF and applying the logical principles of universal and existential quantification, the formal N<sub>3</sub>-serialization of this expression can be denoted as in listing 2.12.

---

```
f?photo :photoId ?id.g
=>
f :request :uri ("/photos/" ?id);
:response [ :represents ?photo ].g.
```

---

Listing 2.12: Derivation of a RESTdesc service description

Using the basic example from listing 2.12, the full extent of RESTdescs capabilities may be demonstrated by applying concrete concepts from HTTP, like URI templates and HTTP methods. The result is demonstrated in listing 2.13.

---

```
@prefix : <http://restdesc.no.de/ontology#>.
@prefix http: <http://www.w3.org/2006/http#>.
@prefix tmpl: <http://purl.org/restdesc/http-template#>.
{
?photo :photoId ?id.
}
=>
{
_:request http:methodName "GET";
tmpl:requestURI ("/photos/" ?photoId);
http:resp [ tmpl:represents ?photo ].
}.
```

---

Listing 2.13: Example of a RESTdesc service description

This snippet essentially denotes that, given there exist a photograph and an identifier to it (precondition), a user may perform an HTTP GET request to retrieve it. The photograph's id has to be appended to the request URI and the response represents the requested photograph.

In a final example the concepts of pre- and postconditions may be discussed. While the service denoted in listing 2.13 retrieves a photograph by its URI, the example in listing 2.14 describes the upload of a photograph. The precondition here is that the client has a resource `foaf:Image`. For the request itself, the HTTP POST method shall

## 2 Related Work

be used. The request's body is specified to contain the photograph. The response of the service is specified to return the identifier of the uploaded photograph. The last line in the listing is the postcondition. In this case the postcondition hints at a possible retrieval of the uploaded picture, by using the photographs identifier that has just been obtained. On close examination, this postcondition turns out to be the precondition of the photograph retrieval service presented in listing 2.13.

---

```
@prefix foaf: <http://xmlns.com/foaf/>.
{
?photo a foaf:Image.
}
=>
{
_:request http:methodName "POST";
http:requestURI "/photos";
http:body [ tmpl:formData ("photo=" ?photo) ];
http:resp [ tmpl:location ("/photos/" ?photoId) ].

?photo :photoId ?photoId.
}.
```

---

Listing 2.14: RESTdesc service description with pre- and postcondition

Based on these findings, context-based service discovery is enabled. Assuming a client possesses a picture:

```
<http://example.org/photo.jpg>a foaf:Image.
```

it can recognize that this satisfies the precondition in listing 2.14. Furthermore, if the client follows the postcondition, it may also realize how to retrieve this very photo again. This mechanism can be extended even further by adding a goal to it. A client may then exploit various paths to achieve this goal.

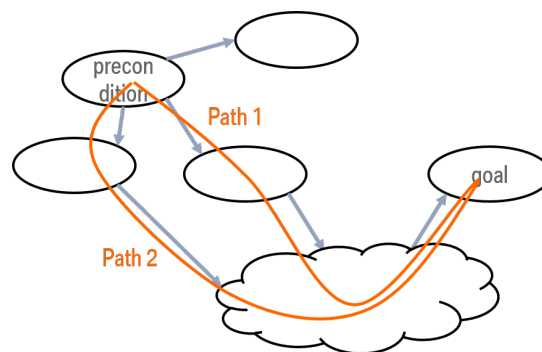


Figure 2.11: RESTdesc automatic service composition.

## 2.6 Access Control

An important horizontal factor in any communication stack is security. In terms of access to protected resources, security considerations usually try to solve two problems: the problem of authentication and the problem of authorization. While authentication is the process of verifying the identity of the entity trying to access a resource, authorization is the process of permitting such an access. In order to authorize an entity, this entity has to authenticate first, to make sure it really is who it claims to be. For authentication purposes, there exist two specification extensions for HTTP, the Digest and the Basic authentication mechanism.[54][55][56]

Besides, there is also an initiative supported by a group of major software companies to create an interoperable authentication mechanism, OpenID.[57] For authorization, OAuth emerged as a widely adopted concept and shall therefore be described in more detail in this section. As in the process of authorization authentication is a key part, HTTP Basic will also be discussed in this section.

### 2.6.1 OAuth

OAuth is an open standard for authorization and as such enables third party applications to obtain scoped access to a resource. That being said, OAuth is **not** an authentication protocol<sup>1</sup>. It is currently available at version 2.0[58] which replaced version 1.0[59] and is not backwards compatible. In fact, OAuth 1.0 and 2.0 share few implementation details and OAuth 1.0 is considered deprecated and unsafe<sup>2</sup>. Furthermore, it is important to note that OAuth was explicitly designed for HTTP only. OAuth addresses several of the inherent issues of authorization by separating the role of client and owner of a resource. This is done by issuing different credentials to a client requesting access to a resource than to the owner of a resource. The client obtains a so-called "token", specified as *a string denoting a specific scope, lifetime, and other access attributes*. Moreover, OAuth allows for tokens not only to be obtained via the server hosting the resource, but a dedicated separate authorization server. This way, resource providers can shift their security focus towards this dedicated server. OAuth 2.0 specifies the roles of four entities: resource owner, resource server, client and authorization server. The following figure (fig. 2.12) should convey a deeper understanding of the defined protocol flow.

<sup>1</sup><https://oauth.net/articles/authentication/>

<sup>2</sup><http://www.ceilers-news.de/serendipity/803-Verfahren-der-Kryptographie,-Teil-15-MD4,-MD5,-SHA-und-SHA-1-alle-unsicher!.html>(in German)

## 2 Related Work

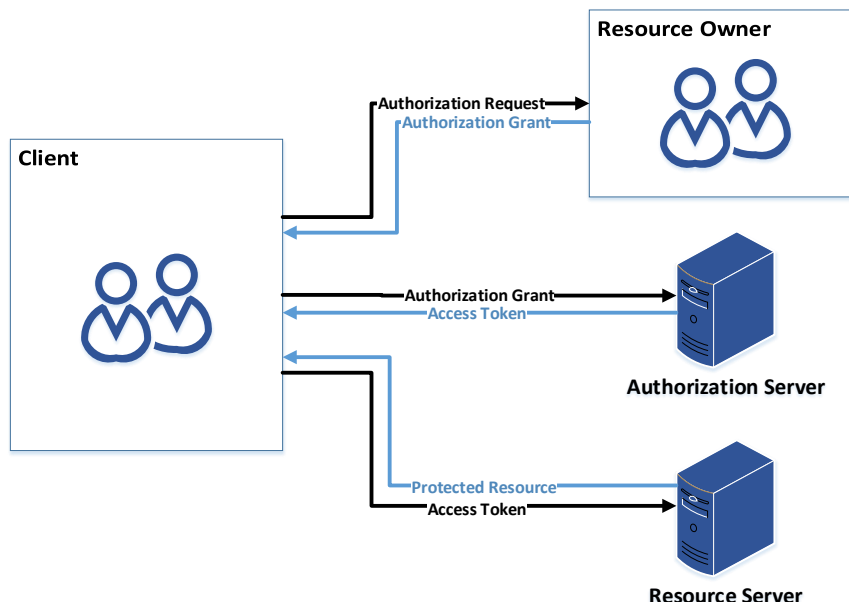


Figure 2.12: Abstract OAuth 2.0 Protocol Flow.

1. The client authenticates at the resource owner and requests authorization.
2. The resource owner replies with an authorization grant, which can be of five types.
3. The client requests an access token at the authorization server, authenticating via its authorization grant.
4. The authorization server validates the authorization grant and, upon success, issues an access token.
5. Using this token, the client can now request the protected resource from the resource server by presenting the obtained token.
6. The token is validated at the resource server and, upon success, the request is served.

As mentioned in step two, there are several types of authorization grant specified:

**Authorization Code** Here an authorization server works as an intermediary between a client of a resource and its owner. The client sends its request to the authorization server which then authenticates the resource owner and obtains authorization. Afterwards, the resource owner is directed to the client issuing an authorization code. The client may then request access tokens from the authorization server using this code.

**Implicit** Here no intermediate authorization code is issued, but the client directly obtains an access token from the resource owner. Compared to the Authorization Code Flow this Flow is simpler but also more vulnerable, as the access token is transmitted back to the client in the form of a URI fragment, exposing it to possible unauthorized parties.

**Resource Owner Password Credentials** Here the resource owner credentials, password and username, are used to directly obtain an access token. Therefore the resource owner credentials have to be accessed directly by the client. They are then exchanged for an access token, therefore not requiring the client to store the resource owner credentials but only use them once. To increase security, the resource owner's credentials may not be stored in the client but exchanged with a long-lived access token or a refresh token.

**Client Credentials** Here, an access token is issued upon authentication credentials provided by a permanently authorized client.

**Extension** OAuth 2.0 also allows for so-called extension grant-types, creating a loophole for custom authorization process definitions.

The OAuth 2.0 specification denotes a number of suggestions for the implementation of its authorization flow. For clients, two different types are defined, namely confidential and public. As already implied by the used terms, confidential clients assure to keep their credentials confidential, while public clients do not. To authenticate with an authorization server, two methods are proposed within the specification: The first method uses the HTTP Basic authentication scheme.[56] Therefore, the HTTP Basic `username` is set to the OAuth `client_id` and the HTTP Basic `password` is set to the OAuth `client_secret`. Conforming to the HTTP Basic scheme, the credentials are then encoded in a Base64-String and inserted into the `Authorization` header as follows:

```
Authorization: Basic TWFya3VzOIRoaXMgaXMgQW5maWVsZA==
```

The HTTP request itself is proposed to be encoded in the `application/x-www-form-urlencoded` encoding type.

The second method suggests to put `client_id` and `client_secret` into the body of the HTTP request, as a key-value pair. However, this method is not recommended. Transmission of the credentials as query-parameters appended to the request URI is explicitly forbidden according to the OAuth 2.0 specification.

As far as HTTP endpoints are concerned, the protocol defines three different endpoints that are utilized within a protocol flow. Server-side there are the authorization endpoint and the token endpoint. Client-side there is the redirection endpoint. The authorization endpoint handles the authorization of clients by resource-owners. The token endpoint is then used by the client to exchange its authorization grant or refresh token for an access token. To do so, clients have to authenticate at the token endpoint. The redirection endpoint is used by the authorization endpoint to return the authorization grant credentials to the client. The specification of OAuth 2.0 requires authorization endpoints to support HTTP GET and permit the implementation of HTTP POST for the authorization code retrieval. From a semantic point of view the use of HTTP POST is debatable, as an authorization code is retrieved. However, the request may also be interpreted as triggering an authorization code generation process, which would conform to the semantics of HTTP POST. A request

## 2 Related Work

to the authorization endpoint must contain a `response_type` parameter which can be of either `code` (for the authorization code protocol flow) or `token` (for the implicit protocol flow). Both public and confidential clients utilizing the implicit grant type are required to register their redirection endpoint URI at the authorization server. If multiple or no redirection URIs are registered, the client has to include the parameter `redirection_uri` in its request parameters.

To obtain an access token from a token endpoint, a client must use the HTTP POST method. To protect users from exposing their authorization grant credentials or access token, the authorization server must require the use of TLS (transport layer security).

Another concept defined in the OAuth specifications is that of the scope of an access token. Clients may request a certain scope for an access token. If a different scope is granted, there must be a `scope` parameter included in the server response. If a client does not include a `scope` parameter within its request parameters, the server may either throw an error or use a default scope.

In case the resource owner denies access, or any request parameter other than the redirection uri is missing or invalid, the authorization server must include an `error` parameter in its response. Optionally, the server may also include an `error_description` and an `error_uri` parameter in the response.

To provide a standardized way of transferring tokens in a request, the OAuth 2 specification is extended by a separate specification defining the use of so-called "Bearer Tokens".<sup>[60]</sup> A Bearer token may be included in a request by either putting it in a header field, a URI request parameter or the request body. When in the header, the `Authentication` header field must be used and the format of the value must use the syntax `Bearer {Base64Token}`. When using the body of a request to insert a token, the `Content-Type` header field must be set to `application/x-www-form-urlencoded`. The key then has to be named `access_token` and the value has to be the token itself. If an HTTP GET request is sent, the body insertion method is explicitly prohibited according to the specification. Putting a token into the request parameters of the target URI of a request is **not** recommended. However, if it has to be implemented, the token must be associated with the defined query key `access_token`.

As has already been mentioned before, the OAuth 2.0 specification leaves a few details on implementation intentionally undefined. This may encourage developers to define their own optimized software solutions, but complicates the definition of an exhaustive vocabulary or model to describe OAuth processes.

## 3 Methodology

After the previous chapter has deployed a firm theoretical background and pointed out the open ends of state-of-the-art research, this chapter proposes a concept to bridge the gap between high-level abstraction and low-level implementation by introducing a workflow-based architecture. The first section will provide an abstract concept introducing generic workflows. Starting from this conceptual high-level view, each section will go further into detail, guiding the considerations from the top-level abstraction to the bottom-level implementation. Following the concept proposal, an architecture linking up necessary components is presented. After an abstract consideration of these components, exemplary description files to implement them are provided. Next, various existing real-world services are presented to deploy an understanding of how web services are invoked and what they provide. Finally, the limitations delimiting the scope of this thesis are stated.

At the end of this chapter a conclusive understanding about the working principles of the proposed concept should be acquired, before the results of their implementation are dissected in the next chapter.

### 3.1 Concept of Generic Workflows

The fundamental concept proposed in this thesis stems in the abstract vision of considering web-services as single, encapsulated items, embedded within a defined workflow. Each item abstracts a specified functionality, without binding it to any specific web-service. Thus, items embody generic interfaces, similar to the concept of separating abstract and concrete stages in WSDL (see section 2.2.1). A generic item is basically defined by its inputs and outputs. A workflow may then be created manually by linking several generic workflow items to each other. This creates a vast number of possibilities of combining workflow items. Interlinking workflow items is done by mapping inputs and outputs, guiding the way of parameters through a workflow. In this first version, workflows do not support loops. However, a workflow item may be instantiated several times if a service is required to be called more than once. Branches may be created by simply mapping one workflow input or workflow item output to multiple workflow item inputs. However, inputs cannot be assigned to multiple outputs, resulting in a compulsory 1:n-relation of workflow item outputs (workflow inputs) to workflow item inputs (workflow outputs). The idea behind this manual workflow creation is that developers can create and deploy workflows in the backend without having to update software in any vehicle in the field. The

### 3 Methodology

services provided by a vehicle are in fact included in a workflow, again represented by workflow items. Users on the other hand, may then only execute a workflow in a single call. Internal execution and structure of a workflow are opaque to its caller.

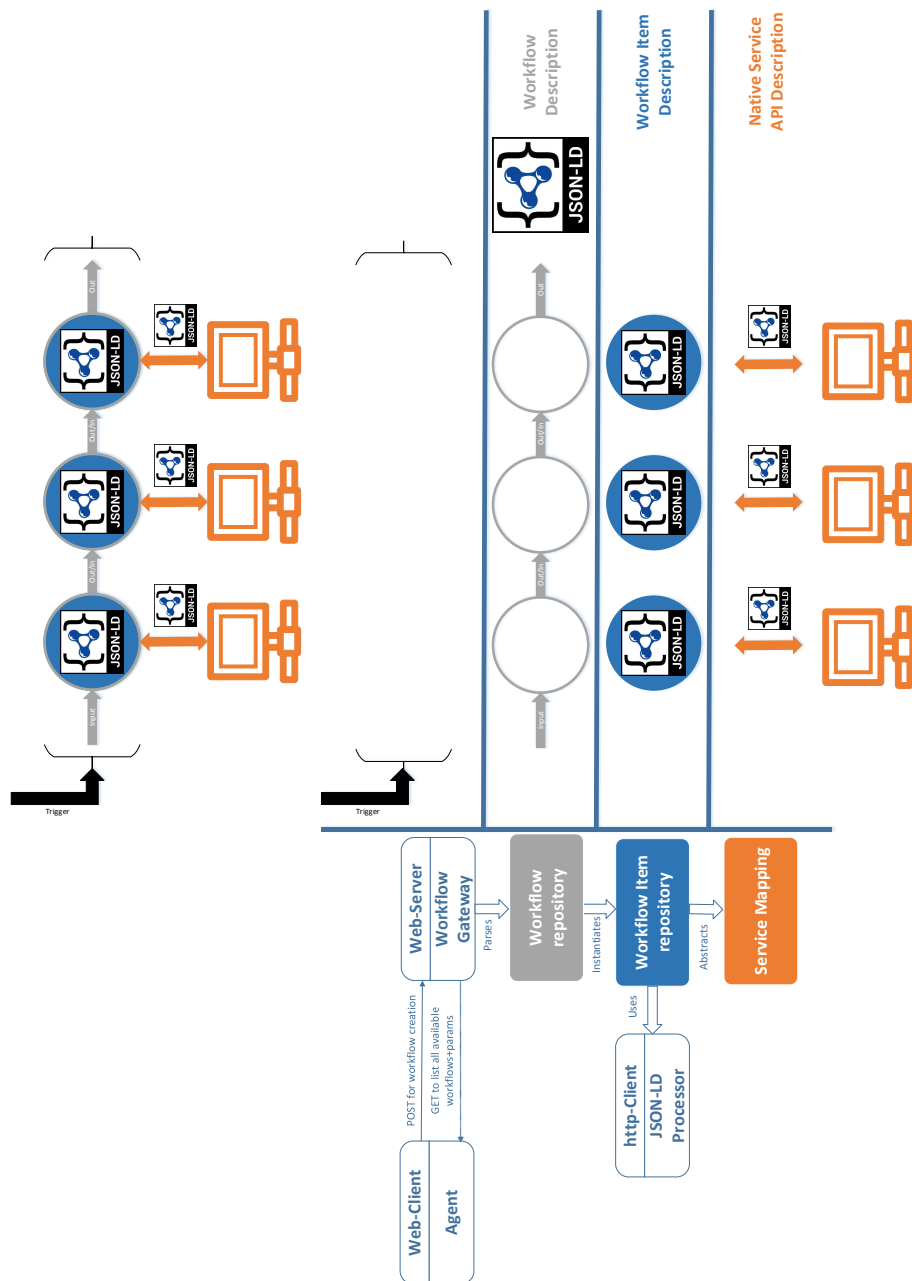


Figure 3.1: General workflows concept.



## 3.2 Proposed Architecture

The proposed architecture to handle and serve workflows consists of several components. These components are intended to mediate interaction between three fundamental entities: users (or agents acting on their behalf), developers and third party services.

**User** The entity invoking a workflow. The scope of this thesis does not specify whether a workflow is invoked directly by a user (for example directly via HMI device in a vehicle) or by a machine agent, acting on behalf of a user (for example an integrated personal assistant in a vehicle, or from the backend as a result of natural language processing).

**Developer** The entity creating new workflows, workflow items and, if necessary, service mappings and service-specific context descriptions.

**Third party service** The web APIs providing a service. This service must be accessible over the web and may typically involve acquisition of specific data, but also physical services like preparing food or beverages.

The mediating components themselves collectively form the desired **system**. In conclusion, several interfaces for this system have to be considered and defined. The system interface for users has to expose workflow descriptions and eventually workflow results. To allow machine-agents to interact with this interface, data exposed by this interface must be semantically annotated and compliant to well-known, defined LOD vocabularies. In the further course of this thesis, this interface will be referred to as the "User Interface". In addition to this user interface, users may also configure information stored in their account, like credit card data or credentials for third party applications. Thus, a "User Configuration Interface" may also be proposed. Another system interface has to accept new workflow, workflow item and service mapping descriptions, as well as service-specific context files. This interface will from here on be referred to as "Developer Interface", as it may only be accessible for developers of new descriptions. The third system interface to be defined, mediates between the system and third party service entities. Therefore, this interface will be referred to as "Third-Party Interface" in the following.

Figure 3.2 illustrates all the fundamental entities and their interfaces with the workflow handling system.

### 3 Methodology

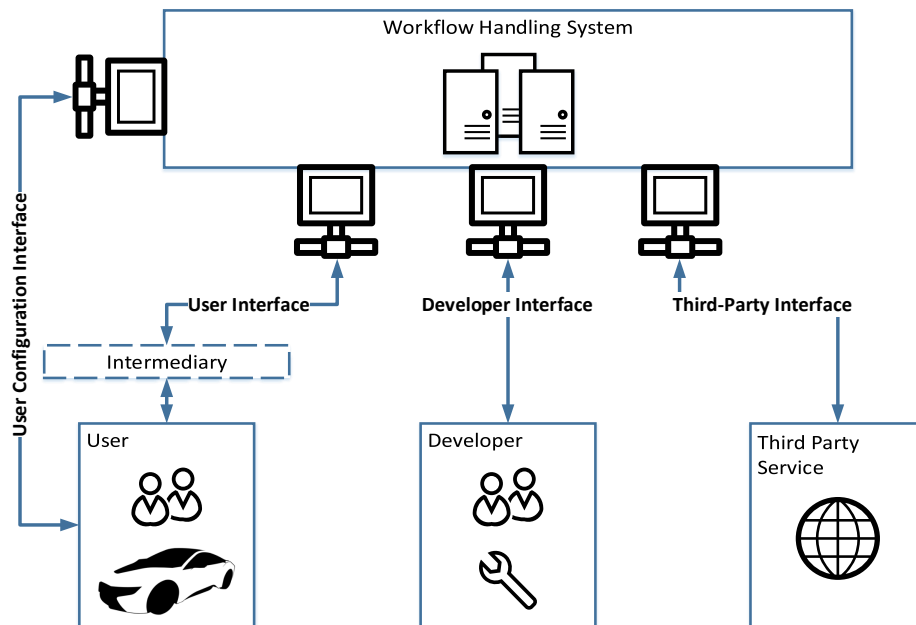


Figure 3.2: Overall architecture abstraction.

Based on these abstractions, the internal architectural structure of the system is designed.

#### 3.2.1 Basic components

After the desired functionality has been defined in a rather abstract high-level examination, the basic system components performing this functionality have to be designed. Therefore, each interface is examined on its own to identify its basic components.

##### User Interface

The user interface exposes workflow descriptions and allows users to invoke a workflow and receive its results. Additionally, all data should be semantically annotated using pre-defined, well-known vocabularies. To satisfy these requirements, the architectural approach of REST (see section 2.2.2) provides an excellent architectural rationale to guide design considerations. Furthermore, the Richardson Maturity Model provides a good reference guideline to verify whether the user interface is truly RESTful.

REST is not bound to any protocol, however, as Fielding himself was co-authoring the HTTP standard, it has been widely adopted in RESTful applications. The Richardson

## 3.2 Proposed Architecture

Maturity Model therefore requires HTTP as transport system, to qualify for level o. By using HTTP, three of Fielding's mandatory constraints are already satisfied: HTTP is based on client-server relations and natively supports caching and layered systems.

The next level of the Richardson Maturity Model is reached by using a URI-based representation structure server-side. This satisfies the identification of resource constraint, established by the principle of uniform interfaces in REST. For the system, this means that everything the user can access, has to be organized by IRI-identified resources. Instead of always querying the same endpoint, these resources can now be queried directly. From the requirements specified in the user interface definition, there are four fundamental resources interacting with a user: the endpoint of the API, a collection of available workflows, workflows themselves and the results of a processed workflow instance. These can be organized hierarchically within a controlled server host authority domain.

Level 2 in the Richardson Maturity Model is achieved by adhering to the semantics of the HTTP method verbs. An exhaustive list and short descriptions of the most important ones has already been provided in section 2.1.1. To some extent this satisfies the self-descriptive messages constraint. Applied to the proposed architecture, this means that the endpoint may be accessed by HTTP GET. The available collection of workflows, as well as the description for each workflow may be also retrieved by applying HTTP GET to the workflows resource. To create a new worker instance of a workflow, an HTTP POST carrying further request parameters is transmitted. The created workflow result can then be requested by HTTP GET again. The last level to climb in the Richardson Maturity Model is to enable the interface for HATEOAS. In section 2.4 different approaches for hypermedia-enabled serialization are described. However, few of them incorporate semantic web technologies and utilize only enclosed proprietary vocabularies, comprising string represented keywords. Hydra, on the other hand, provides an additional semantic layer on top of JSON-LD and exists under the open world assumption. This predestines Hydra for linked data applications, as it provides defined semantics for describing resources and their interaction. By enabling the interface to be self-descriptive and HATEOAS, all of the uniform interface constraints are satisfied. The only constraint left is the "stateless" constraint, demanding that each client request to the server must include all the relevant information without the server using context from previous interactions. Thus, the proposed architecture has to ensure that this constraint is also satisfied, by ensuring that each interaction with a resource has to be independent from previous or future interactions.

Taking all these constraints into account, the following components have been found:

- From the user's point of view, the system behaves like an HTTP server. Moreover, as this server acts as an intermediary, providing resources obtained from third-party servers, it is even more concise to denote the system as gateway (as defined in the HTTP terminology, see section 2.1.1), considering the overall context. This server will also be referred to as the "workflow server" in the

### 3 Methodology

following.

- Within the authority of the workflow server, there are several resources for users to interact with. When requesting the root of the server, the `hydra:ApiDocumentation` of the user interface is provided. It describes how to invoke the `/workflows` collection, to obtain a list of available workflows. By performing HTTP GET on the `/workflows` collection, an extensive list of available workflows is provided. Choosing one of them and performing HTTP GET yields a Hydra conforming description of the workflow. Performing an HTTP POST containing all required parameters should create a corresponding workflow instance. This instance contains the workflow results and can be retrieved by an HTTP GET.

Figure 3.3 depicts all the components and their orchestration.

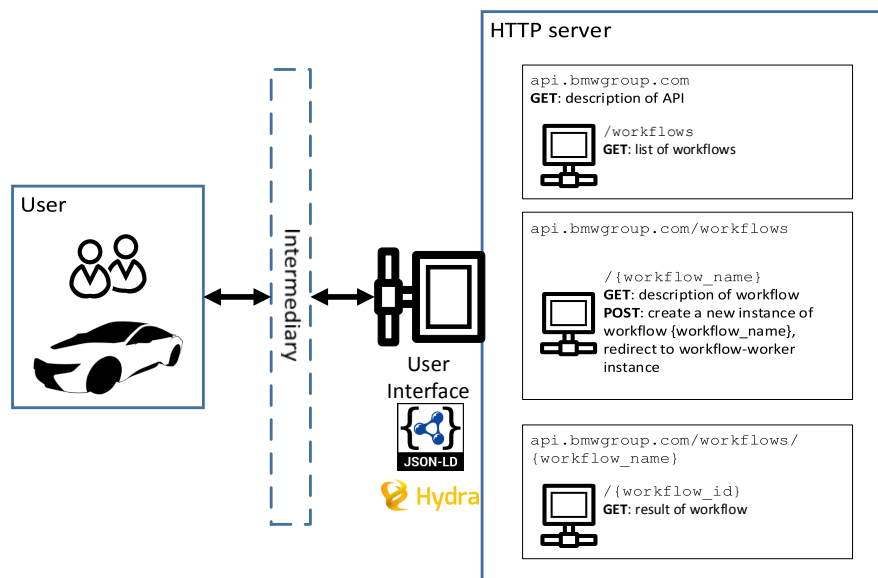


Figure 3.3: Basic components of the User Interface.

### User Configuration Interface

While the user interface enables users to invoke workflows and retrieve their results, a configuration interface may allow them to securely store data that may be used by workflows, to warrant a smooth and seamless workflow process. Third party APIs usually need authentication and authorization, a problem that can only be solved using an orthogonal component. This component provides a user, authenticated with the workflow server, to enable the workflow server to act on his or her behalf. The implementation of this component could also be in the form of a triplestore, thus enabling the data to be linked easily when needed by a workflow. If a user is not yet registered with a service, or needs to provide credentials for a service yet, a mechanism may be implemented, linking up a users phone with the configuration

## 3.2 Proposed Architecture

interface, thus enabling him or her to add his or her credentials. Other than in desktop or other handheld environments, the distraction of a driver from the essential driving task has to be taken into consideration too.

### Developer Interface

The developer interface has to be capable of adding and parsing new workflow descriptions, workflow item descriptions, service mappings and service-specific context. Therefore, it would be beneficial, to provide developers with tooling to allow an easy creation of descriptions and reduce error-proneness by validating and verifying these descriptions in a sandbox environment, before deploying them. However, these considerations are out of scope of this thesis and will be presented in the outlook section. Assuming a developer has created valid descriptions, they may be deployed on the server. For this thesis, a rather easy solution was found: developers upload their description files into the file system of the server, assuming they have the according rights and roles to do so. The workflow server's integrated parser will then periodically check these directories for changes and update its internal repositories.

Therefore, the following components have to be implemented:

- The file system and a directory hierarchy associated to the different types of description.
- A parser, recognizing the structure of each description file and parsing them for the server implementation.

The process of a developer creating a description is outlined in figure 3.4.

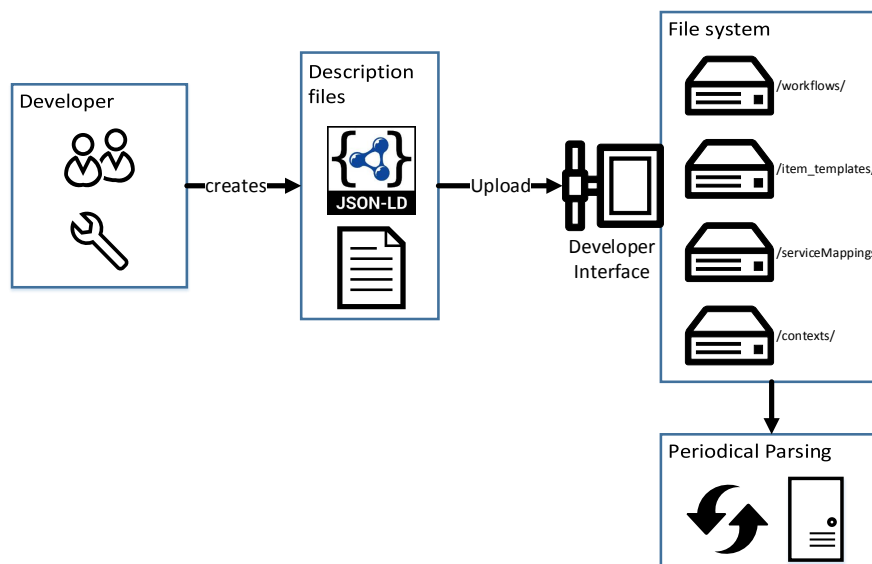


Figure 3.4: Basic components of the Developer Interface.

## 3 Methodology

### Third-Party Interface

The third-party interface has to support different kinds of service-implementations and might be the toughest interface to implement. Several assumptions have to be made to constrain the range of services supported. Few public APIs are actually truly RESTful, usually because they lack support of the HATEOAS constraint or do not provide self-descriptive responses. A multiplicity of standards for architectural aspects as serialization format, authorization method, authentication method, invocation style and many more exist and reaching consensus on how to design an API is a field of science on its own. Thus, one of the objectives in this thesis is to identify and integrate the most common practices and contemplate the most promising efforts in implementing an API. As already described in section 2.1.1 and 2.1.3 JSON has become a very commonly used media-type in real-world APIs. JSON-LD even adds a layer on top of it, enabling implementation of RDF graph structures. To semantically interpret these graphs, the workflow server-internal HTTP client has to recognize a defined set of IRIs, ideally even entire vocabularies. For the internal HTTP client to **understand** how to invoke a third-party service, the service has to provide specific instructions. In an ideal world, the third-party service would natively expose a description of its capabilities, using a vocabulary both parties understand. One vocabulary trying to establish this mutual understanding is Hydra. Hence, if a third-party service exposes instructions of how to interact with it in Hydra, the internal HTTP client would get a natively instructed and need no further input from developers trying to integrate a service. The world as we know it, however, is far from ideal. That said, a component has to be found, able to instruct the internal HTTP client whenever there is no Hydra description provided by a third-party provider itself. The proposed component intended to offer salvation is a server-internal repository, storing manually compiled service descriptions. Developers may then either bind this internal `hydra:ApiDocumentation` resource to a workflow item, or, in the best case it is provided externally. Once the client knows how to correctly invoke a third-party service, it may still not comprehend the response of the service. As already discussed in section 2.1.1 and 2.1.3, there are numerous serialization formats and media-types on the web. Therefore, the scope of this thesis is limited to JSON and its extension, JSON-LD. While a JSON-LD response conforming to recognized vocabularies may be natively processed by the workflow server, plain JSON responses cannot be integrated easily. Thus, there has to be some workaround, enhancing a plain JSON response with additional semantics, to enable a seamless integration into the internal RDF graph structure. This may be resolved by injecting service-specific JSON-LD context. A more detailed description, however, will be provided in section 3.2.4 when discussing the implementation of service-specific context.

To recap: given a third-party service provider is using Hydra and JSON-LD as serialization format, APIs may be integrated into a workflow description seamlessly, as the workflow server-internal client may then be capable of natively comprehending the service provider's semantics. In case the service provider is responding in plain

### 3.2 Proposed Architecture

JSON, a developer can create a mapping context, similar to a lifting scheme in SAWSDL. The illustration in figure 3.5 depicts the least optimal (and, unfortunately, to date most common) case of a non-Hydra descriptive, non-JSON-LD capable third-party API integrated into the workflow server environment (JSON icon designed by Madebyoliver from Flaticon).

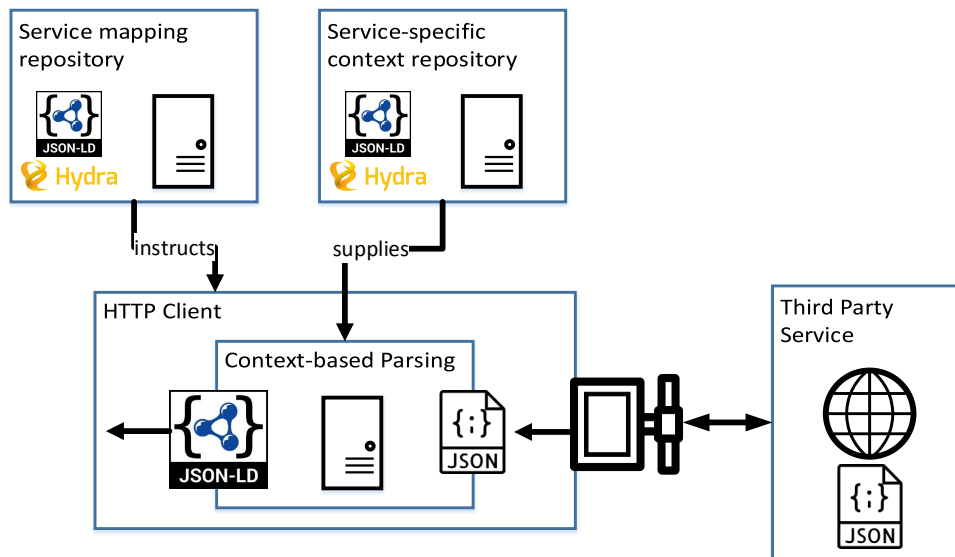


Figure 3.5: Basic components of the Third-Party Service Interface.

## 3 Methodology

### 3.2.2 Workflow description

A workflow is compiled of various different workflow items. Each of these items has an interface for input and output parameters. To describe a workflow in an RDF conforming structure that can later be parsed by the workflow server implementation, a recognized vocabulary has to be defined. Linked Open Data recommendations suggest to use publicly defined and accessible vocabularies to enable client developers to implement their semantics. However, finding vocabularies that natively provide all the semantics needed in a new system, is usually a quite cumbersome task. As already discussed in section 2.1.4, there are numerous publicly available vocabularies, some already extensively used. One of them was created to describe APIs in particular and already introduced in section 2.5.1: the Hydra vocabulary. Considering the external behavior of a workflow, Hydra appears to be capable of describing inputs, outputs and invocation method of a workflow, using the `hydra:operation` property. Regarding a description of the internal structure of a workflow, that is workflow items and their mapping, Hydra provides no support whatsoever. However, as previously defined, workflows are opaque for their invoking entity. Consequently, the internal description of a workflow is invisible for its invoking entity. This very fact allows for the conclusion that the internal structure of a workflow does not qualify as open as in Linked Open Data. Hence, the only entity required to parse the internal structure of a workflow is the workflow server. In conclusion, the definition of a proprietary vocabulary to describe the internal workflow structure is feasible. Each workflow contains a title, a human-readable description, operations to be applied to it, items and a mapping. Every workflow description is of the proprietary type `vocab:Workflow`. As only title, description and operation are exposed by the workflow server, they are tied to the Hydra IRIs `hydra:title`, `hydra:description` and `hydra:operation`. For the internal properties of a workflow, the proprietary vocabulary terms `vocab:items` and `vocab:mapping` are defined. An exemplary JSON-LD workflow description template is listed below in listing 3.1.

---

```
{
"@id": "host:workflows/transaction",
"@type": "vocab:Workflow",
"hydra:title": "Transaction Workflow",
"hydra:description": "Basic customer - provider transaction from within a
  vehicle.",
"hydra:operation": [
... ],
"vocab:items": [
... ],
"vocab:mapping": [
... ]
}
```

---

Listing 3.1: Basic structure of the workflow description concept



## 3.2 Proposed Architecture

In this example, the CURIE `host:` refers to the hosting authority of the workflow server, as for example `http://example.org/`. In a node-arc-node graph, a workflow description entity would look like demonstrated in figure 3.6.

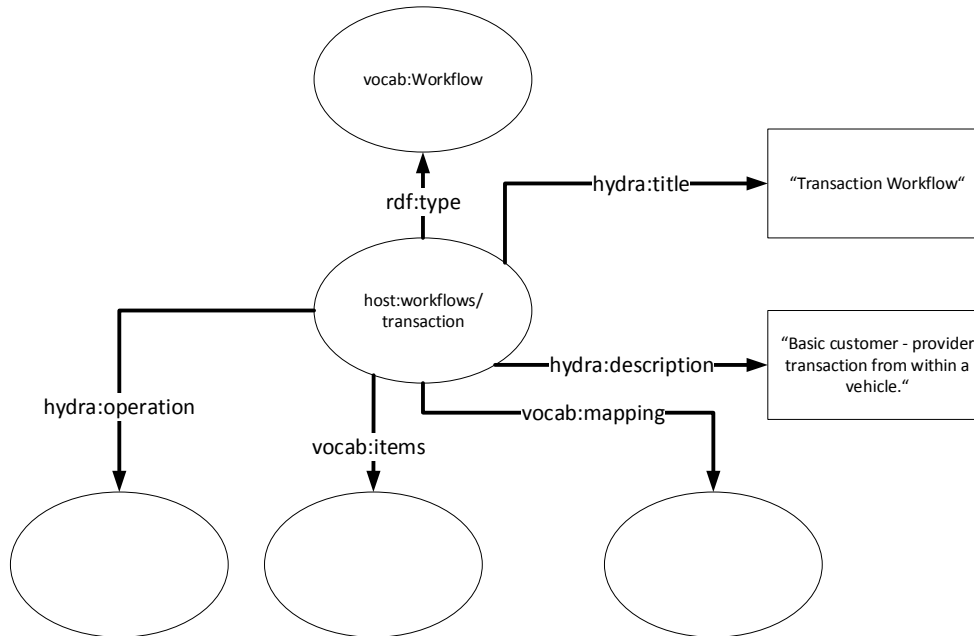


Figure 3.6: Exemplary workflow description resource.

The `hydra:operation` property defines how to invoke a workflow, what parameters need to be passed and what return parameters are to be expected. The concept is mostly based on the concept of `hydra:Resource` and `hydra:Operation` specifying how to invoke a resource of an API. Each operation has the properties `hydra:method`, `hydra:expects`, `hydra:returns` and `hydra:statusCodes`. Additionally, in its current version, Hydra defines three subclasses of `hydra:Operation`: `CreateResourceOperation`, `ReplaceResourceOperation` and `DeleteResourceOperation`. The `hydra:operation` property therefore has a range of `hydra:Operation` and all its subclasses. If the creator of a workflow wants to not only describe which property to expect as an input or output, but also add descriptive properties about a property, Hydra offers the property `hydra:supportedProperty`. At first glance this may sound very convoluted and confusing, hence a short example might provide some clarity: if a workflow expects an input `dc:identifier` at runtime, but its creator wants to convey that the parameter is required, this relation has to be described somehow. By introducing a resource of type `hydra:SupportedProperty` that describes a property `hydra:property`, for example with a property `hydra:required`, this relation can be established. Another reason to introduce this intermediary `hydra:SupportedProperty` node is to disambiguate mappings within a workflow. These are done by the ID of a `hydra:SupportedProperty` rather than the property's

### 3 Methodology

ID itself. A detailed discussion about why this is necessary will be provided in the explanation of the mapping mechanism of a workflow. The example in listing 3.9 demonstrates how this part of a description may look like.

---

```
...
"hydra:operation":[
  {
    "hydra:method":"POST",
    "hydra:expects":[
      {
        "hydra:supportedProperty":[
          {
            "@id":"#input:id",
            "property":"dc:identifier",
            "hydra:required":true
          }
        ]
      }
    ],
    "hydra:returns":[
      {
        "supportedProperty":[
          {
            "@id":"#output:order",
            "property":"schema:order",
            "hydra:required":true
          }
        ]
      }
    ]
  }
],
...
```

---

Listing 3.2: Exemplary hydra:Operation class object

## 3.2 Proposed Architecture

Demonstrated in the form of a node-arc-node graph node, a `hydra:Operation` class object (or one of its subclasses) would look like in figure 3-7.

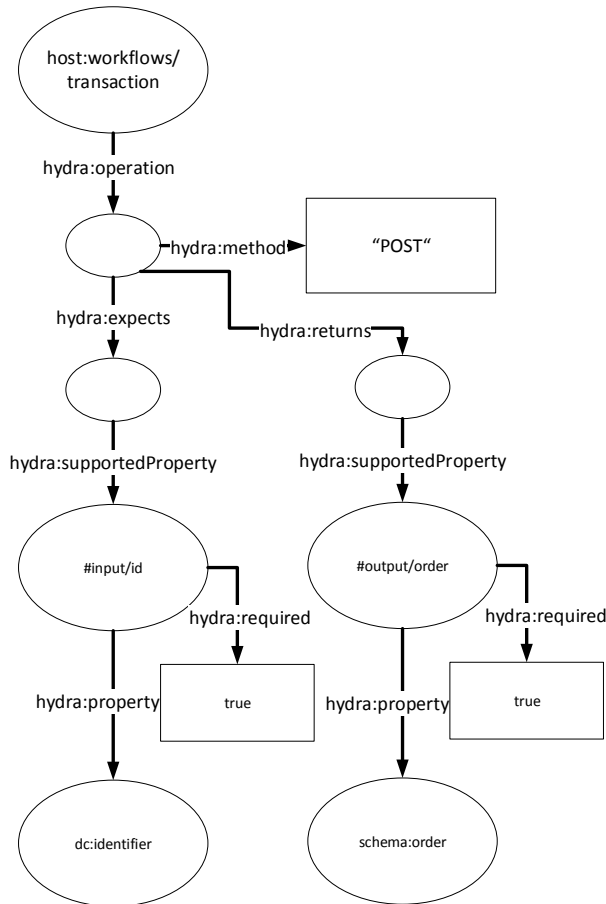


Figure 3.7: Exemplary `hydra:Operation` class object.

The proprietarily defined property `vocab:items` describes the instantiation of workflow items within the described workflow. This mechanism allows to detach definition and instances of a workflow item, similar to the concept of classes and objects in object-oriented programming. While every workflow item has the same functionality, different instances can be invoked with different parameters, altering the behavior of a workflow item. Instantiation is performed by defining JSON-LD `@id` and `@type` of an instance.

### 3 Methodology

Listing 3.3 demonstrates the instantiation of two workflow items, `discovery` and `hmiPickOne`.

```
...  
"vocab:items": [  
  {  
    "@id": "instances:discovery1",  
    "@type": "defs:/discovery"  
  },  
  {  
    "@id": "instances:hmi1",  
    "@type": "defs:/hmiPickOne"  
  }  
],  
...
```

Listing 3.3: Exemplary instantiation of workflow items

Figure 3.8 illustrates the item instantiation in a graph. As there is no concept of arrays in standard RDF, multiple identical property arcs have to be used within a graph.

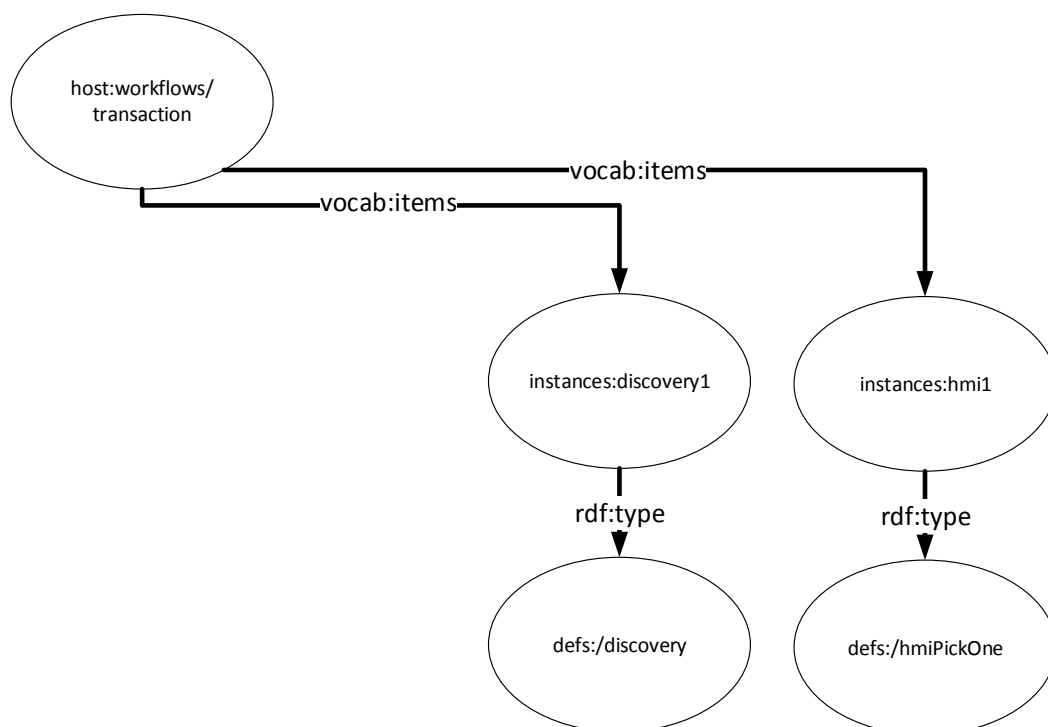


Figure 3.8: Exemplary instantiation of workflow items.

The second proprietarily defined property in workflow descriptions is `vocab:mapping`. It maps the inputs and outputs of the workflow and each workflow item it contains. Not to be confused with `hydra:mapping`, it inheres the meaning of mapping arbitrary **workflow** and **workflow item** inputs and outputs rather than URI template variables and properties, as in Hydra. Using `hydra:mapping` here would corrupt its semantics and may be interpreted falsely by a machine client. Thus, a context-specific term has to be introduced here. While the inputs and outputs of the workflow are defined within the same document, the workflow item inputs and outputs are defined in their separate description document (see next section). A single mapping is represented by a blank node, containing one of each proprietarily defined properties `vocab:from` and `vocab:to`. It is important to note that not the properties itself are mapped, but their descriptions in the corresponding descriptive `hydra:SupportedProperty` instance. As properties may be unambiguous within the definition of a workflow item, it is still possible to instantiate multiple workflow items from the same definition. Thus, to disambiguate the input and output properties of each instance, the information of which instance is associated with which mapped property must not get lost.

Listing 3.4 shows the continuous mapping of a workflow consisting of two workflow items.

---

```

...
"vocab:mapping": [
  {
    "vocab:from": "#inputs/id",
    "vocab:to": "instances:discovery1#input/id"
  },
  {
    "vocab:from": "instances:discovery1#output/location",
    "vocab:to": "instances:hmi1#input/collection"
  },
  {
    "vocab:from": "instances:hmi1#output/collection",
    "vocab:to": "#output/order"
  },
],
...

```

---

Listing 3.4: Example of input/output mapping of workflow items

### 3 Methodology

In figure 3.9 the corresponding RDF graph representation is illustrated.

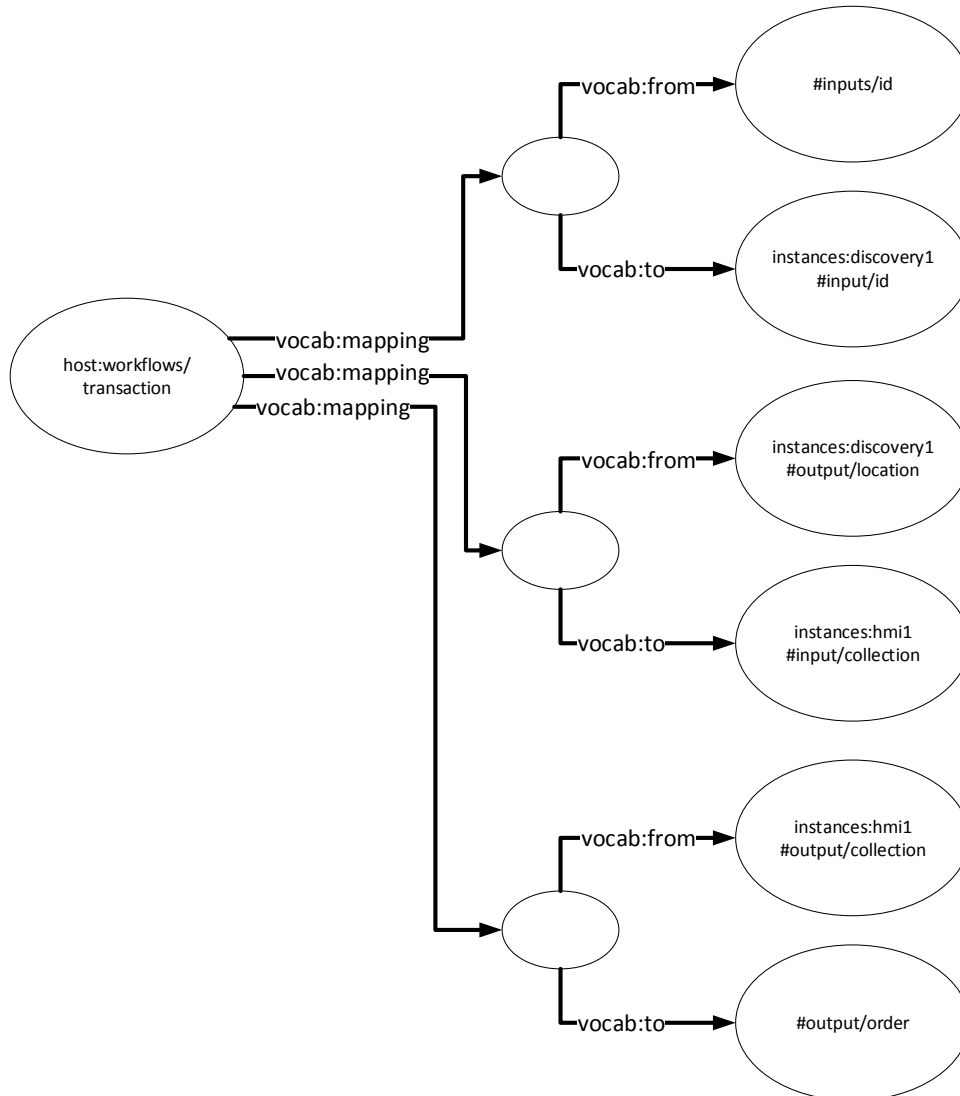


Figure 3.9: Exemplary mapping of workflow item instances.

### 3.2 Proposed Architecture

After discussing all the components of a workflow description in detail, the resulting graph shows how all the resources are related to each other. Compiled from all the graph examples introduced in this section, the overall graph can be assembled as presented in figure 3.10.

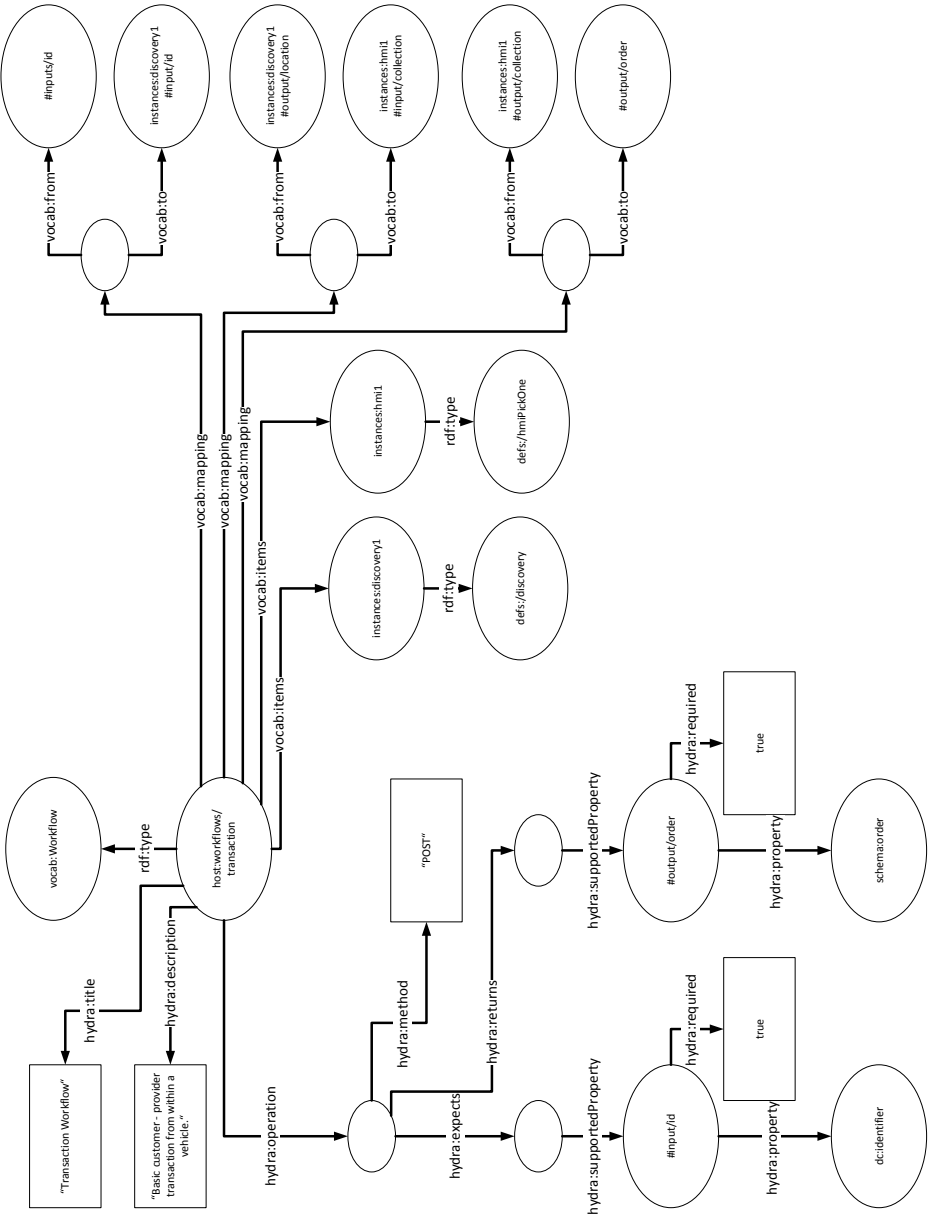


Figure 3.10: Overall graph of an exemplary workflow description.

## 3 Methodology

### 3.2.3 Workflow item description

While already mentioned briefly in section 3.2.2, this section goes further into detail about how to describe workflow items. In the proposal of the concept of workflow items, several ideas and approaches are tried to be taken into account. Similar to the concept of the abstract stage in WSDL, a workflow item description should be as generic as possible and not coercively bind to a specific third-party web API. It should enable the support of various different service providers for similar services, without the need of creating provider-specific workflow items or even entire workflows for each of them. The specific binding of a workflow item template then happens via the `vocab:serviceMapping` property and will be elaborated in detail in the following. Workflow item definitions are thus involving a trade-off: to enable them to support various providers of a service, only their common functionality can be used. However, if a developer desires to utilize specific, extensive functionality of a single provider, he or she would have to create a unique workflow item, binding it to only one service. This way, the freedom of choosing between generics and specifics is still left with the developer of a workflow.

As previously mentioned, workflow description files act similar to class definitions in object-oriented programming languages. Within a workflow, developers can create multiple instances of the same workflow item definition. As each instance of a workflow still has the same input and output properties, their identifying IRIs have to be disambiguated to preserve a well-defined, unambiguous mapping relation within a workflow. To foster a better understanding of the structure of a generic workflow item description, a minimal working example is compiled step-by-step in the course of this section.

The basic workflow item resource structure consists of the properties: `hydra:title`, `hydra:description`, `vocab:serviceMapping`, `vocab:input` and `vocab:output`. As all of these properties are for internal use (meaning they are not exposed by the user interface on the workflow server), proprietary vocabulary may be used. Moreover, each workflow item resource is defined to be of type `vocab:WorkflowItemTemplate`. `hydra:title` and `hydra:description` have already been discussed before, they merely offer human readable information about a resource. As has already been mentioned briefly in this section, `vocab:serviceMapping` inherits the information of a service binding. It can be assigned in two ways: If a service is bound at runtime, an input property can be indirectly assigned to the service mapping by using the `vocab:from` property. If the binding is static, the URI of the mapped service can be assigned directly to the `vocab:serviceMapping` property as a string literal. This mechanism again emphasizes the freedom of defining generic or specific workflow items. `vocab:input` and `vocab:output` define the input and output properties of a workflow item. They will be discussed in more detail later in this section.

In its basic structure, a description example then looks like in listing 3.19.



```

"@id":"http://example.org/definitions/discovery",
"@type":"vocab:WorkflowItemTemplate",
"hydra:title":"Discovery Service Item",
"hydra:description":"Discovery of locations close to a certain location.",
"vocab:serviceMapping":{
  "vocab:from":"#input/serviceUrl"
},
"vocab:input": [
  ...
],
"vocab:output": [
  ...
]

```

Listing 3.5: Basic structure of a workflow item description

Yet again, this example can be illustrated in form of an RDF graph (figure 3.11).

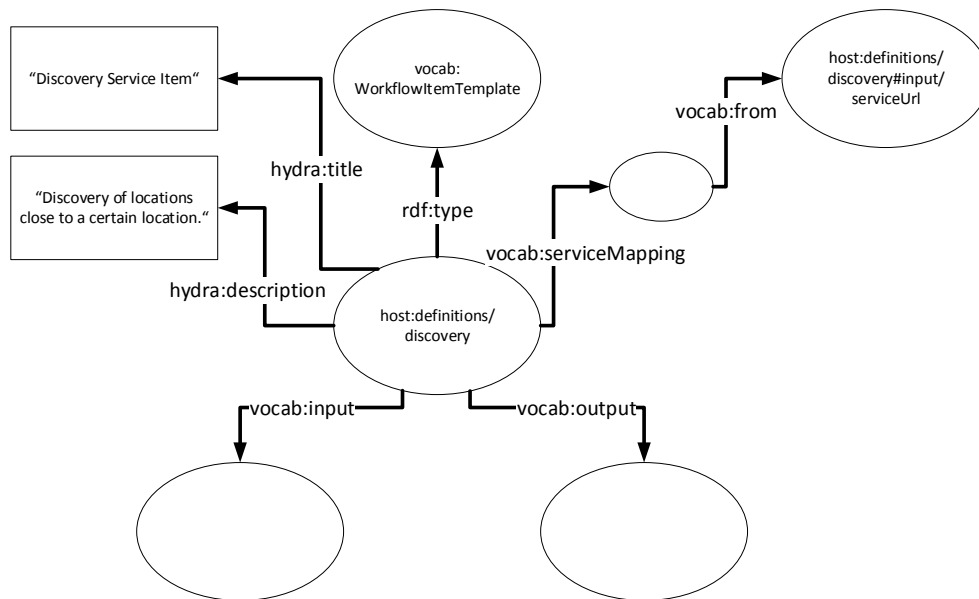


Figure 3.11: Exemplary workflow item description resource.

As already mentioned before, `vocab:input` and `vocab:output` define the input, respectively output interface of a workflow item. Similar to `hydra:expects` and `hydra:returns` they use the `hydra:supportedProperty` mechanism to define the incoming and outgoing properties of a workflow item at runtime. This mechanism is not needed for the mapping of concrete services to workflow items, as there services would use unambiguous properties themselves. Listing 3.6 shows a minimal example of how input and output of a workflow item could be defined. Figures 3.12 and 3.13 then provide the associated RDF graph form.

### 3 Methodology

---

```
...
: [
  {
    "@id": "#input",
    "supportedProperty": [
      {
        "@id": "#input/lat",
        "@type": "hydra:SupportedProperty",
        "property": "schema:latitude",
        "required": true
      },
      {
        "@id": "#input/lng",
        "@type": "hydra:SupportedProperty",
        "property": "schema:longitude",
        "required": true
      },
      ...
    ]
  }
],
"output": [
  {
    "@id": "#output",
    "supportedProperty": [
      {
        "@id": "#output/locationCollection",
        "property": "schema:collection",
        "required": true
      },
      ...
    ]
  }
]
...
```

---

Listing 3.6: Input and output property description of a workflow item

### 3.2 Proposed Architecture

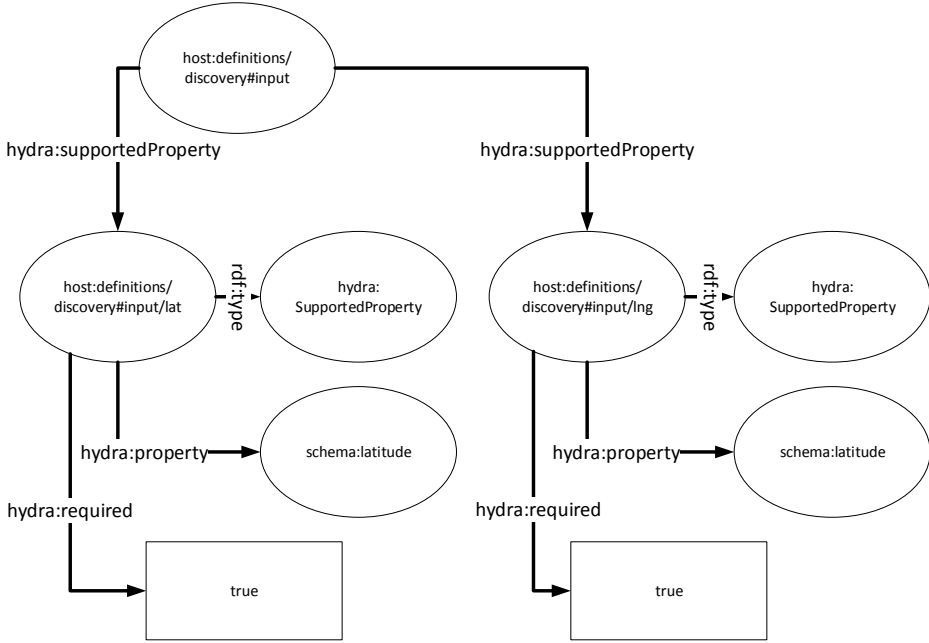


Figure 3.12: Exemplary workflow item input description resource.

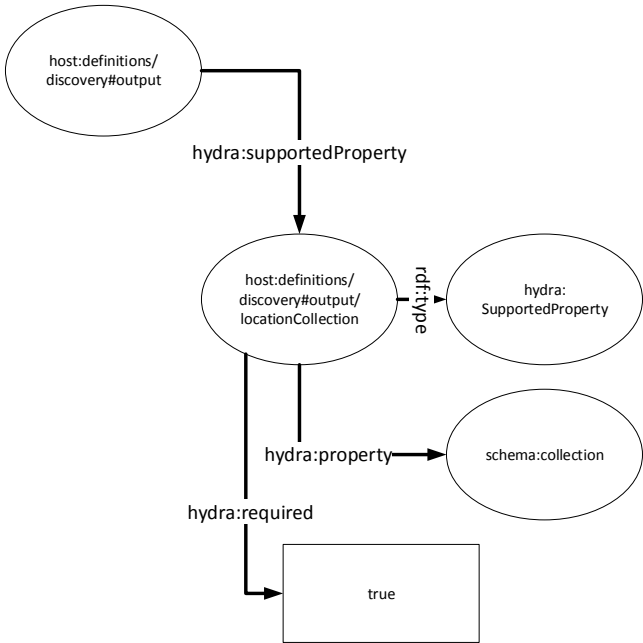


Figure 3.13: Exemplary workflow item output description resource.

### 3 Methodology

The overall minimal working example for a workflow item description can be illustrated as a whole in an RDF graph. This graph may easily be in the graph describing a workflow, as it represents the type definition of a workflow item instance. Therefore, the `rdf:type` property connects both graphs, while the mapping in the workflow interconnects inputs and outputs of the instantiated workflow items.

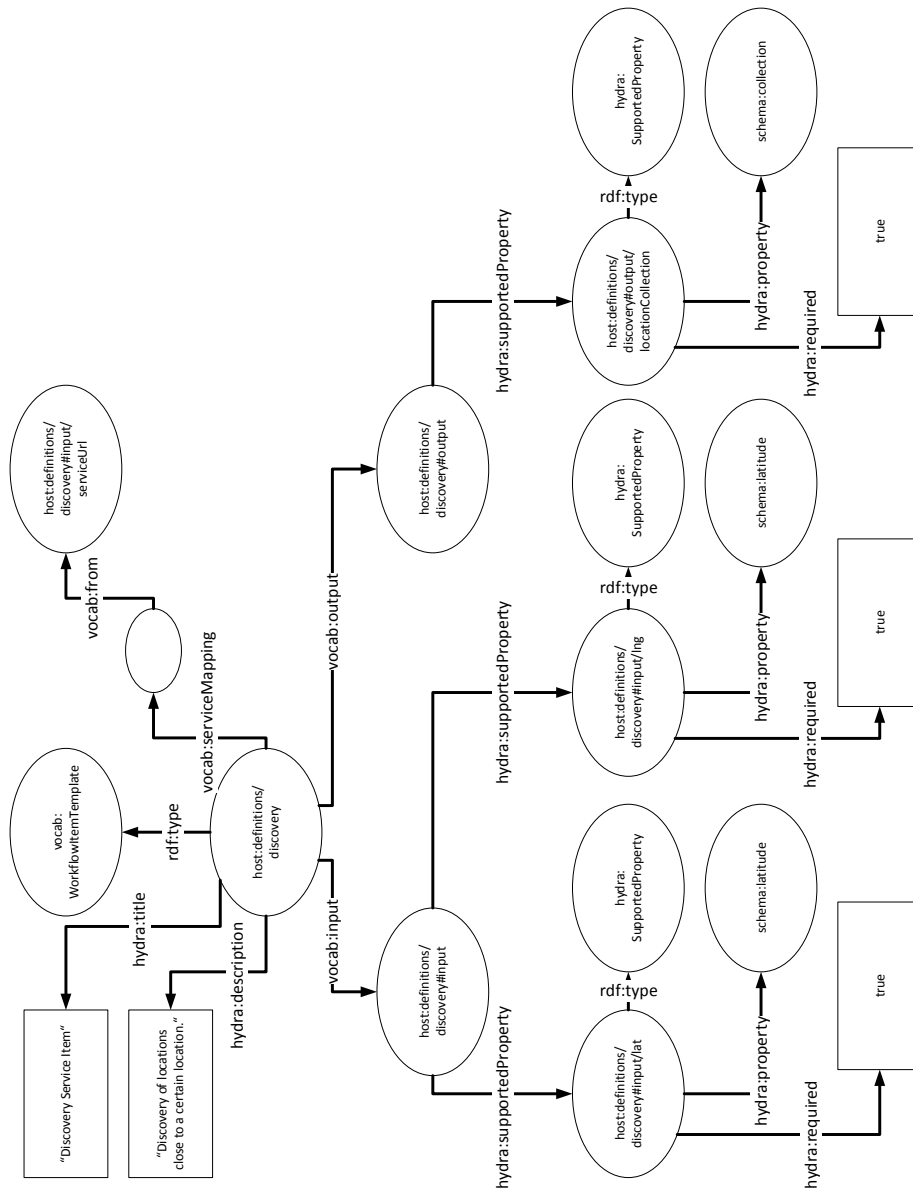


Figure 3.14: Overall graph of an exemplary workflow item description.

### 3.2.4 Service mapping

The last missing piece of the proposed workflows framework is the concept of service mappings. As has already been mentioned in the previous section, the service mapping implements the concrete binding of a workflow item to a third-party service API. In section 3.2.1, the user interface was specified to support not only self-created, internally stored service descriptions, but also the possibility to parse and *natively* understand service descriptions provided by an external service provider himself. Thus, Hydra has been identified as a promising approach to enable services to provide an RDF conforming solution. To access the description for the behavior of an API, Hydra offers the `ApiDocumentation` concept. In its domain, publicly defined properties are indicating API-specific classes, possible status codes and an endpoint. The workflow parser is currently comprehending objects from two different types in the range of `hydra:entrypoint:hydra:IriTemplate` and `hydra:Resource`. Both will be discussed in the course of this section.

In addition to the existing Hydra vocabulary, an extension to support the semantic annotation of OAuth authorization is proposed.

While the semantic description of a service enables the internal client to invoke the service API correctly, it does not warrant that the internal client understands the response. Most state-of-the-art APIs support only plain JSON and only few will provide JSON-LD. Thus, the JSON-to-JSON-LD parsing mechanism outlined in section 3.2.1 has to be implemented as well.

Discussing one issue after another, listing 3.7 starts by presenting an example for the basic structure of a descriptive `hydra:ApiDocumentation` resource:

---

```

"@id": "http://localhost:8080/definitions#YelpLocations",
"@type": "hydra:ApiDocumentation",
"hydra:title": "Yelp business search API",
"hydra:description": "Discovery of venues close to a certain location.",
"hydra:statusCodes": [
  ...
],
"hydra:entrypoint": {
  ...
},
"hydra_ext:authorization": {
  ...
}

```

---

Listing 3.7: Basic structure of a `hydra:ApiDocumentation` class object

The semantics of both `hydra:title` and `hydra:description` has already been presented. `hydra:statusCodes` contains semantically annotated descriptions of possible response status codes, in particular their meaning and possible cause. The most important property, however, is `hydra:entrypoint`, as it points directly to a description of how to interact with the API. As already mentioned, the proposed

### 3 Methodology

implementation recognizes two possible types for an endpoint. The first one is `hydra:IriTemplate`. An IRI template indicates a queryable resource, thus supporting HTTP GET only. `hydra:template` represents the IRI template, annotated as defined in IETF RFC 6570 (see section ??). A template contains several variables, placeholders for values. These variables are mapped via `hydra:IriTemplateMapping` objects, assigned to a `hydra:IriTemplate` by the `hydra:mapping` property. Each `hydra:IriTemplateMapping` maps a `hydra:variable` to a `hydra:property`. The property `hydra:variableRepresentation` indicates whether mapped IRIs should be interpreted as plain strings or extended by language and/or datatype information. Listing 3.8 provides an example for the description of an IRI template resource.

```
...
"hydra:entrypoint":{
  "@id":"https://api.yelp.com/v3/businesses/search/",
  "@type":"hydra:IriTemplate",
  "hydra:template":"https://api.yelp.com/v3/businesses/search/{?latitude,longitude}",
  "hydra:variableRepresentation": "BasicRepresentation",
  "hydra:mapping": [
    {
      "@type": "IriTemplateMapping",
      "hydra:variable": "latitude",
      "hydra:property":{
        "@id":"schema:latitude"
      }
    },
    {
      "@type": "IriTemplateMapping",
      "hydra:variable": "longitude",
      "hydra:property":{
        "@id":"schema:longitude"
      }
    }
  ]
}
...
```

Listing 3.8: Hydra description of an API in the form of a templated URI query

The second recognized type in the range of `hydra:entrypoint` is `hydra:Resource`. Each `hydra:Resource` supports the property `hydra:operation`, describing the `hydra:Operation` to interact with the resource. The semantics operation descriptions have already been examined when introducing workflow descriptions. `hydra:returns` may be omitted here, especially when a service-specific context has to be used to parse a JSON response into JSON-LD.

---

```

...
"hydra:entrypoint":{
  "@id":"https://api.yelp.com/v3/businesses/search/",
  "@type":"hydra:Resource",
  "hydra:operation":{
    "@type":"hydra:Operation",
    "hydra:method":"POST",
    "hydra:expects":[
      {
        "hydra:supportedProperty":[
          {
            "hydra:property":"schema:latitude",
            "hydra:required":true
          }
        ]
      }
    ],
    "hydra:returns":[
      {
        "hydra:supportedProperty":[
          {
            "hydra:property":"schema:collection",
            "hydra:required":true
          }
        ]
      }
    ]
  }
},
...

```

---

Listing 3.9: Exemplary `hydra:entrypoint`

With the internal HTTP client now able to request resources from third-party APIs, the next step would be to enable the client to parse responses in a meaningful way. Thus, a mechanism has to be implemented, parsing plain JSON responses into semantically annotated JSON-LD objects. In the proposed approach, some characteristics of the JSON-LD processing algorithm are tried to be exploited. By injecting a service-specific JSON-LD `@context` into the plain JSON response, strings can be retrospectively resolved to IRIs by JSON-LD term expansion. Moreover, a term definition cannot only be used to map a term to an IRI, but also to map a term to a keyword. The latter is then referred to as a keyword alias. To demonstrate this functionality, a short example may be considered. Listing 3.10 shows an example for a plain JSON response. In this example, a client may have obtained a list of Premier League fixtures.

### 3 Methodology

---

```
[
  {
    "id": "4f6e2f42e4b0e4284308fd7a",
    "name": "Liverpool FC vs. Everton FC",
    "date": "01.04.2017",
    "stadium": "Anfield Road"
  }
]
```

---

Listing 3.10: Example for a plain JSON reponse

As one can clearly see, all the keys in this response are plain strings rather than IRIs and could therefore not be processed by semantically enabled clients. However, by exploiting the term expansion mechanism of a JSON-LD processor, a service-specific context may be injected into the response before the JSON-LD expansion algorithm is applied. The template in listing 3.11 provides exactly this functionality. The response is wrapped by the `@graph` keyword, this way the context applies to all JSON objects in the response.

---

```
{
  "@context": [
    *** remote context ***
    {
      *** local context ***
    }
  ],
  "@graph": [
    *** plain JSON ***
  ]
}
```

---

Listing 3.11: Context injection template

For demonstration purposes, the example from listing 3.10 is now injected with specific context, using the template presented in listing 3.11. This context should transform every plain key string `name` with the IRI key string `http://schema.org/name`. In addition, it is assumed that a bound workflow item requires to extract the `id` of the element, thus the context maps it to the IRI `http://purl.org/dc/terms/identifier`. After injecting appropriate context, the result looks like in listing 3.12.



---

```

{
  "@context": [
    {
      "id": "http://purl.org/dc/terms/identifier",
      "name": "http://schema.org/name"
    }
  ],
  "@graph": [
    [
      {
        "id": "4f6e2f42e4b0e4284308fd7a",
        "name": "Liverpool FC vs. Everton FC",
        "date": "01.04.2017",
        "stadium": "Anfield Road"
      }
    ]
  ]
}

```

---

Listing 3.12: Exemplary context-injection

This representation now conforms to the compacted form of a JSON-LD document. As such, a JSON-LD processor can now apply the expansion or flattening algorithms. An extensive list of JSON-LD processor implementations may be found at the official webpage of JSON-LD<sup>1</sup>. For quick validation of contexts and whether they lead to the desired results, the JSON-LD playground provides excellent tooling support<sup>2</sup>. Applying the expansion algorithm to the example in listing 3.12, leads to the JSON-LD converted response in listing 3.13.

---

```

[
  {
    "http://purl.org/dc/terms/identifier": [
      {
        "@value": "4f6e2f42e4b0e4284308fd7a"
      }
    ],
    "http://schema.org/name": [
      {
        "@value": "Liverpool FC vs. Everton FC"
      }
    ]
  }
]

```

---

Listing 3.13: JSON-LD-converted JSON response after context-injection

---

<sup>1</sup><http://json-ld.org/#developers>

<sup>2</sup><http://json-ld.org/playground/>

### 3 Methodology

A client that recognizes the IRIs can now comprehend its meaning. However, there are still issues when using this mechanism. As JSON responses may be nested in multiple layers, they may consequently contain ambiguous JSON key names within a single response. Thus, a mechanism enabling hierarchical unwrapping has to be considered. To illustrate this issue, listing 3.14 provides a relevant example.

---

```
{
  "@context": [
    {
      "response": "_:response",
      "event": "_:event",
      "venue": "_:venue",
      "id": "http://purl.org/dc/terms/identifier",
      "name": "http://schema.org/name"
    }
  ],
  "@graph": [
    {
      "response": {
        "event": [
          {
            "id": "4f6e2f42e4b0e4284308fd7a",
            "name": "Liverpool FC vs. Everton FC ",
            "venue": {
              "name": "Anfield Road"
            }
          }
        ]
      }
    }
  ]
}
```

---

Listing 3.14: Example for a nested class definition

To resolve or *drill-down* the base object, the approach here is to assign blank nodes to the keys of the parenting instances. Applying the expansion algorithm, the resulting JSON-LD file from listing 3.15 is obtained.

---

```

[
  {
    "_:response": [
      {
        "_:event": [
          {
            "http://purl.org/dc/terms/identifier": [
              {
                "@value": "4f6e2f42e4b0e4284308fd7a"
              }
            ],
            "http://schema.org/name": [
              {
                "@value": "Liverpool FC vs. Everton FC "
              }
            ],
            "_:venue": [
              {
                "http://schema.org/name": [
                  {
                    "@value": "Anfield Road"
                  }
                ]
              }
            ]
          }
        ]
      }
    ]
  }
]

```

---

Listing 3.15: Context-injection converted nested class definition

Ignoring and unwrapping all the blank node keys in the evaluation of the response would then still create a feasible result, if it was not for the problem of ambiguity. As can be observed in listing 3.14, the JSON object now contains two keys called `name`, which is perfectly compliant to the JSON specification as there are no ambiguously named keys within the same nesting level of an object. To extract the `name` of the venue of the event, blank nodes for `response`, `event` and `venue` have to be defined. As both the `event` object and the `venue` object contain a `name` key, the processor expands them both as `http://schema.org/name`. To avoid this behavior the nesting structure of the response has to be included in the mapping for JSON-LD. Thus, the workflow framework simply adds a logical layer on top of JSON-LD, as shown in listing 3.16.

### 3 Methodology

---

```
{
  "@context": [
    *** remote context ***
    {
      *** local context ***
    }
  ],
  "vocab:disambiguateJson": {
    *** disambiguating mappings ***
  }
  "@graph": [
    *** plain JSON ***
  ]
}
```

---

Listing 3.16: Context injection template

The logical layer is implemented by adding the proprietary defined property `vocab:disambiguateJson` to the service-specific context file. By using the syntax `Parent.Child.Property.{IRI-Subproperty}(...).{IRI}` instead of only the name of the key, an internal parser (**not** the JSON-LD processor) can disambiguate wrongly applied context. Listing 3.17 therefore presents the same example extended by the `vocab:disambiguateJson` property, containing disambiguating key-value pairs.

---

```

{
  "@context": [
    {
      "response": "_:response",
      "event": "_:event",
      "venue": "_:venue",
      "id": "http://purl.org/dc/terms/identifier",
      "name": "http://schema.org/name"
    }
  ],
  "http://localhost:8080/vocab#disambiguate": {
    "http://schema.org/location": "response.event.venue.{http://schema.org/name}",
    "http://schema.org/name": "response.event.{http://schema.org/name}"
  },
  "@graph": [
    {
      "response": {
        "event": [
          {
            "id": "4f6e2f42e4b0e4284308fd7a",
            "name": "Liverpool FC vs. Everton FC ",
            "venue": {
              "name": "Anfield Road"
            }
          }
        ]
      }
    }
  ]
}

```

---

Listing 3.17: Nested objects response with ambiguities

The internal parser will then apply the disambiguation algorithm and return a response according to listing 3.18.

---

```

{
  "http://schema.org/location": [
    {
      "@value": "Anfield Road"
    }
  ],
  "http://schema.org/name": [
    {
      "@value": "Liverpool FC vs. Everton FC"
    }
  ]
}

```

---

Listing 3.18: Disambiguated JSON-LD converted response

### 3 Methodology

#### Implementation of API Authorization

As has already been elaborated in section 2.6.1, authorization is necessary with most APIs and must therefore be handled by the proposed workflow framework. OAuth offers a specified protocol flow and can thus be perfectly described by a standard vocabulary. However, Hydra is yet to address orthogonal aspects such as authentication and authorization. Thus, in this thesis, an extension to Hydra is proposed, enabling Hydra to convey descriptions of authorization protocol flows in JSON-LD. Basic properties of the OAuth flow are: grant type, token endpoint and authorization endpoint. Additionally, the introduction of a property `hydra_ext:requiredHeaders` may be considered, as some APIs require specific, but usually static, header parameters.

---

```
...
"hydra_ext:authorization":{
  "@type":"OAuth2",
  "hydra_ext:grantType":"password",
  "hydra_ext:tokenEndpoint":"https://api.yelp.com/v3/businesses/search/",
  "hydra_ext:requiredHeaders":[
    {
      "hydra_ext:headerKey":"Content-Type",
      "hydra_ext:headerValue":"application/json"
    }
  ]
}
...
```

---

Listing 3.19: Proposed Hydra OAuth extension

### 3.3 Used APIs

One key aspect in the target definition of this thesis was the utilization of real APIs. In the curb side pick-up use case mentioned in chapter 1 there are several services involved: one service discovering nearby coffee vendors, one service to pay for the coffee and various services within the vehicle (positioning, user interaction, etc.). To integrate these services into the workflow framework, their structure has to be examined in detail.

#### 3.3.1 Discovery services

The first class of services to be examined are discovery services. These services should provide a set of locations within a defined vicinity of a defined point of interest. Additionally, there may be further filters applied to constrain results. After an online survey, three services were found to offer these capabilities: the Foursquare Search Venues API<sup>1</sup>, the Yelp Search API<sup>2</sup> and the OpenTabs Locations API<sup>3</sup>. Based on a review for common capabilities and general style of invocation for these services, a generic discovery template may be designed. This design starts with a review of the request parameters of each service. The chosen approach in this thesis was to compare the semantics of all parameters and check for similarities. Following this review, table 3.1 has been assembled:

Generic	Foursquare	Yelp	OpenTabs
lat	ll.lat	cll.lat	latitude
lng	ll.long	cll.long	longitude
radius	radius	radius_filter	radius
query	query	term	filter
limit	limit	limit	limit

Table 3.1: Mapping of request parameters for discovery services.

In this table, a more or less arbitrary generic string is chosen to map the different string representations. However, as discussed on numerous occasions already, things should be defined through unambiguous IRIs, rather than plain strings. Thus, it should be tried to map IRI definitions from well-known vocabularies, rather than creating arbitrary strings. Table 3.2 defines the semantics of each of those parameters in a human-understandable way and exemplifies how a mapping of well-known property names could look.

<sup>1</sup><https://developer.foursquare.com/docs/venues/search>

<sup>2</sup>[https://www.yelp.com/developers/documentation/v3/business\\_search](https://www.yelp.com/developers/documentation/v3/business_search)

<sup>3</sup><http://www.opentabs.de/>

### 3 Methodology

Generic	Schema Property	Description
lat	schema.org/latitude	Latitude coordinate of the point of interest.
lng	schema.org/longitude	Longitudinal coordinate of the point of interest.
radius	schema.org/geoRadius	Radius from point of interest to be considered.
query	schema.org/query	String to filter the returned results.
limit	schema.org/numberOfItems	Parameter limiting the returned results.

Table 3.2: Request parameter description.

After this generic interface for HTTP requests has been found, an output interface generalizing possible responses has to be considered. Again, a review of each API provider's response specifications has been conducted. The results are presented in table 3.3.

Generic	Foursquare	Yelp	OpenTabs
id	id	id	ident
name	name	name	name
lat	location.lat	location.coordinate.latitude	gps_lat
lng	location.lng	location.coordinate.longitude	gps_lng
country	location.cc	location.country_code	country_iso
city	location.city	location.city	city
address	location.address	location.address	address1

Table 3.3: Mapping of response parameters for discovery services.

Same as for the request parameters, rather than using ambiguous strings as keys for the mapped response values, it is highly recommended to use unambiguous IRIs from well-known public vocabularies. Therefore, instead of using a generic string definition, suitable terms in public vocabularies should be used. In this concrete example, there are two concepts which can be used to map the outputs: for the id of the location `dc:identifier` provides suitable semantics, while all the other output parameters fit perfectly into `schema:Place`. A full list of the generic output interface for a discovery service can be found in table 3.4.

Generic	Schema.org/Place
name	name
lat	geo.latitude
lng	geo.longitude
contry	Country
city	addressLocality
address	streetAddress

Table 3.4: Mapping of response location parameters to schema.org/Place.



### 3.3.2 Vehicle services

As the vehicle will play a central role in processing a workflow, it will have to provide several services. Each vehicle represents a surrogate of its driver, implemented by its interior HMI capability. What used to be pressing buttons and turning knobs, has also evolved with remarkable speed during the last decade. Vehicles like the new BMW 7 series offer gesture control, speech assistance and multiple touch interfaces within the vehicle. Therefore, to interact with a user, each vehicle has to provide some sort of HMI service.

For the PoC in this thesis, only rudimentary services were mocked, to incorporate human interaction in a workflow. One interaction service is the `HmiPickOne` service. A client may HTTP POST a `hydra:Collection` to it, the vehicle's HMI device(s) presents it to the user and the user may pick one item of the collection. For creating an order, the implementation also mocks an `HmiPickMultiple` service, with the same basic function as `HmiPickOne`, but allowing a user to pick multiple items from a list. Besides interacting with the user, vehicle offer much more data that may be exploited in a workflow. Thus, car manufacturers are working on implementations for vehicle APIs exposing relevant parameters like position, speed, fuel consumption and many more, into their cloud. Services in the backend, like the workflow processor, may then use this data to add value to a user's experience. As there is no real-world implementation now, vehicle data are also only simulated in the PoC.

### 3.3.3 Payment services

One of the most delicate web service applications is web payment. Web payment and online banking are already widely implemented but the hype for the so-called "fintech" (abbreviated from financial technologies) business is still growing.<sup>1</sup> The W3C even set up a dedicated **Web Payments Working Group** to standardize web payment solutions.

Credit card providers like Visa with its CyberSource API<sup>2</sup> and MasterCard with its Payment Gateway<sup>3</sup> also provide APIs to enable online payment transactions. Interestingly, the latter therefore provides its own WADL description<sup>4</sup>.

Probably the most popular online payment service and one of the earliest players in this market, is **PayPal**. PayPal offers one the most RESTful APIs, incorporating HATEOAS by using the HAL (see section 2.4.1) standard. To create a payment, PayPal provides its Payment API. The PayPal Payment API allows direct credit card payments, stored credit card payments, or PayPal account payments. A payment

<sup>1</sup><https://www.bloomberg.com/gadfly/articles/2016-08-19/the-bubble-in-fintech-doesn-t-look-much-like-one>

<sup>2</sup><https://developer.visa.com/products/cybersource>

<sup>3</sup>[https://eu-gateway.mastercard.com/api/documentation/apiDocumentation/index.html?locale=en\\_US](https://eu-gateway.mastercard.com/api/documentation/apiDocumentation/index.html?locale=en_US)

<sup>4</sup>[https://eu-gateway.mastercard.com/api/documentation/apiDocumentation/reference/wadl20090202.xsd?locale=en\\_US](https://eu-gateway.mastercard.com/api/documentation/apiDocumentation/reference/wadl20090202.xsd?locale=en_US)

### 3 Methodology

request must contain a specified intent, the payer and the transaction. However, when using the PayPal payment, the user **must** be redirected to an approval URI, provided by PayPal, which contains an HTML login. This payment flow is already well-established in the desktop- and handheld-environments, but rather inconvenient when a user is busy driving a vehicle. Thus, the scope of this thesis excludes this option and concentrates on credit card payments. As discussed before, there are two ways to pay by credit card: via submitting the complete credit card object, or via using the PayPal Vault, where credit cards can be stored. This payment method is also called "tokenized" credit card.

Every Payment API interaction needs to be authorized via OAuth 2. PayPal utilizes the client credentials flow, providing tokens in exchange for dedicated client credentials. Therefore, in an implementation it would make sense to assign client credentials to each vehicle/user account. For each invocation of the Payment API, the token is then included in the `Authentication` header field according to the Bearer Token specification.

To create a payment, PayPal requires several parameters:

**intent** describes the intended payment process, for immediate payment, intent has to be set to `sale`.

**payer** describes the identity of the user and includes credit card data.

**transaction** describes transaction details, as item list, payment sum and payment recipient.

For the workflow handling system to act as an intermediary for credit card payments via PayPal, it must know the email and merchant ID of the provider of a service.

#### 3.3.4 Ordering services

Currently, there seem to be only a few services providing open APIs for third party applications. Most providers offering this kind of service typically act only in certain (geographic) areas. Thus, these services are a good example for the necessity of hyperlocal service support. OpenTabs is a start-up from Munich providing a mobile app for ordering food and beverages at a specified location. Similar providers are **YQ** in New Zealand, **beat the q** in Australia and **Tapingo** in the US. What they all have in common is that they provide services that allow users to order at partnered venues offering food and beverage. After ordering, a user may either pick up the order or, depending on the provider, has it delivered to a specified location. Thus, there is no queuing and the order is already processed upon arrival.

The process of ordering food is separated into several APIs: as has already been mentioned in section 3.3.1, OpenTabs provides its own location discovery API. After the OpenTabs ID of a location is obtained, OpenTabs provides a Menus API to request the available menu items at a defined location. From this menu, a user might then assemble an order and use the Orders API to post an order to the previously chosen location.

For authorization, OpenTabs utilizes the OAuth2 resource owner protocol flow.

However, there are some out of specification peculiarities in the authorization flow. While OAuth2 specifies the endpoints for requesting access and refresh tokens to be the same, OpenTabs defines two different endpoints. Moreover, in its current version, OpenTabs requires clients to add the header `"x-opentabs-api-key: ios_3.0"` to all of their calls, including authorization calls.

### 3.4 Limitations

The components proposed in this thesis have the potential to solve complex mashups and execute powerful workflows. However, to enable powerful performance, no less powerful and thus intricate implementations are required. Therefore, as time and resources were limited within the scope of this thesis, certain limitations had to be defined.

As already discussed in section 3.2, the scope of this thesis excludes the actual trigger of a workflow. The trigger may not be explicitly set by a human user, but may be set by some assistance function in the backend or after some artificial intelligence reasoning upon a user's behavioral patterns. For example, driver fatigue sensors may ask a tired driver for a coffee break and suggest nearby coffee shops.

For sake of simplicity there have also been numerous constraints in the implementation of the workflow server, as will be discussed in chapter 4 again. The implementation of user authentication may be solved using OpenID or similar standards, or within proprietary specifications. User credentials may also be stored on a hardware token, like a vehicle's keys. The PoC implementation in the course of this thesis assumes that each user acts in a user-specific environment and is securely authenticated with the orthogonal user store component. Thus no authentication with the user interface is implemented.

As has already been briefly discussed in section 3.3, there are countless different implementations of service authorization. Therefore, the scope of which methods to support has to be defined in advance. As only OpenTabs and Paypal will be used in the PoC implementation, OAuth 2 resource owner and client credential grant types are the only ones being implemented. Deprecated authorization protocols like OAuth 1.X and proprietary authorization methods are also deemed out of scope.

The user configuration interface will also not be implemented, but simulated by hardcoded parameters, again for the sake of brevity and demonstration purposes.

In its current state, the workflow processor will also not be able to perform federated queries within one workflow item. In particular, this refers to requests that have to be split, as different data from different APIs have to be obtained. An example would be a request for the friends of a friend from some social network API, e.g. Facebook (note: this is a hypothetical example and does not imply whether or not this is implemented in Facebook). One request may have to get a friends ID, while the other retrieves the people linked with that ID. In the current implementation, a workflow item for each request would have to be created.



## 4 Results

Following the implementation of the concepts proposed in the previous chapter, this chapter presents and discusses the findings and results obtained. First, a comparison and review of the different description approaches examined in chapter 2 is conducted. Next, the use-case driven implementation carried out in the course of this thesis is presented. The final section of this chapter will then provide an overview of the lessons learned during the implementation work.

### 4.1 Review of existing Web Service Description approaches

Several publications have investigated existing approaches to describe web services. Sheng et al.[61] provide an extensive overview of web service description and composition approaches. They also define and provide a set of assessment criteria for web services. Verborgh et al. have published a similar survey of existing semantic web service descriptions.[47] In their article, the authors differ between lightweight semantic descriptions, SPARQL-based descriptions, logic-based descriptions and JSON-based descriptions.

For the review conducted in the course of this thesis, an evaluation schema had to be defined. Therefore, several criteria were rated, based on the scale in table 4.1:

++	The criterion is satisfied to its full extent in every dimension.
+	The criterion is satisfied to its full extent in several dimensions.
o	The criterion is supported in all dimensions, but insufficiently satisfied.
-	The criterion is insufficiently supported and only met in a few dimensions.
--	The criterion is not met in any dimension.

Table 4.1: Definition of rating scale.

## 4 Results

The criteria used in this thesis have been defined as follows:

**Readability** is defined to assess the capability of a description format to be read and consequently understood without the need of a profound understanding of any specified standards. In general, the more verbose and human-readable a description style is, the better the rating.

**Flexibility** is defined as the ability of a description to adapt to changes in the capabilities of a web service during runtime. These changes may be due to restructuring or extending an API. For example, if clients will be caused to break without an out-of-bound update themselves, a rather negative rating will be awarded.

**Syntax** is defined to express the capabilities and rate the inherent semantic power of the syntactical allowance of a description standard. The semantic power is essentially rated based on the terseness of a syntax when expressing a relation.

**Semantics** is defined as the capability of a description standard to incorporate semantic web technologies as the RDF data model, IRIs, ontologies and reasoning.

Based on the evaluation schema and the criteria that have been established, each web-service description is examined and discussed accordingly.

**Plain text description** is according to the work of Verborgh et al.[47] the most popular form of web service description. Naturally, textual description allows for verbosity and the use of any vocabulary. Therefore, a description might provide concise and elaborate terms to provide the semantics for developers wishing to integrate a service. Assuming a diligent textual description is provided, this format offers great readability as it provides documentation in natural language, which is even amenable for non-experts. Thus, plain textual description offers great readability (++). As soon as an API is changed however, clients have no chance of adapting to the changes, as they have no in-band access to this information. Developers would have to start their whole development process again and manually adapt their clients. Thus, plain textual description offers no flexibility whatsoever (- -). Regarding the syntax of a textual description there are usually no specifications either. There may exist some conventions, as textual descriptions are often offered via HTML-document and are written in English, but the representation of the text itself is under no further syntax definition, enforcing terseness. A negative rating was deemed to be appropriate due to these syntactical insufficiencies(-). The unconstrained nature of textual descriptions denies them any capability for semantic web technologies (- -).

**Static description formats** provide structured documentation and tooling to automatically generate human-readable representations of the code-like textual description documents. A short phase of adaption is necessary to enable a

## 4.1 Review of existing Web Service Description approaches

human to comprehend the syntax and get used to the tooling, however, once experienced with the pattern of a standard, the rather rigorous structure enables a fast comprehension. Thus, readability was rated predominantly positively (+). A change in the structure or extension of an API is not registered by deployed clients in the field and thus out of band. However, developers may use automatic code generation tools to quick start implementations and may experience reduced development cycles. A client still has to be manually updated, but developers are saved effort, therefore flexibility is rated negatively (-). In terms of syntax, standards like Swagger offer a well-proven terminology that concisely describes the data model imposed by the standard. The syntax is capable of efficiently describing the structure used in most existing HTTP conforming APIs (++). Semantic web technologies are not supported by static description languages. Keywords are plain strings and given defined semantics within the standard, but are not using unambiguous IRIs. In conclusion the worst rating is applicable (- -).

**Dynamic description formats** typically involve lightweight terminology. Due to their distributed nature, a human would still need to explore APIs step-by-step and might assume certain functionality by interpreting the semantics of used key names. Moreover, for some standards an understanding of the semantics imposed by the HTTP specifications is required. Due to their shortcomings in these dimensions, dynamic description formats are rated only positively (+). In terms of flexibility, due to their dynamic nature, existing client implementations do not break upon server-side changes. While clients may not comprehend changes without an update, they are still capable of utilizing existing functionality. Therefore, the flexibility rating is also positive (+). The syntax used by most dynamic description standards is lightweight and easy to comprehend itself. However, due to their lightweight character, space for individual interpretations of the documentations and their underlying data-models is remaining. Thus, the syntax rating is given a neutral rating (o). As most dynamic description standards are based on some other serialization standard, they would in theory support the use of IRIs as keys and values. However, none of the standards of HAL, SIREN and Collection+JSON unambiguously use IRIs as keywords in their syntax. Their data models are usually similar to RDF, but do not use standards from the semantic web technology stack. Therefore, the semantics rating for them is only predominantly positive (+).

**Semantic description formats** are denoted in RDF-enabled serialization standards like JSON-LD, Turtle and RDF/XML. A basic understanding of the RDF model is therefore required to interpret a response. IRIs may be unambiguous but still require definition of their unambiguous meaning in some way. In vocabularies like schema.org this is done in a machine- and human-readable form, but other vocabularies might not be as extensively documented. Due to these reasons, human-readability may be achieved, but requires some effort by a human reader. Therefore, a neutral rating (o) is awarded in this dimension.

## 4 Results

As semantic description standards base on dynamic description standards, they also embrace the HATEOAS approach. Beyond that, clients that recognize vocabularies may even comprehend server-side extensions and changes as long as they adhere to the recognized vocabularies. Thus, semantic description standards are awarded the best flexibility rating (++). Complexity and capability of the syntax depend on the used serialization format. In general, the RDF data model allows to describe almost every kind of relationship in the existing world. However, this capability comes at the expense of verbosity and convoluted annotations sometimes. The syntax rating of semantic description formats is therefore neutral (o). As already implied by the classification category's name, semantic description formats provide elaborate functionality to incorporate semantic web technologies and evolve with the research efforts put into the Semantic Web. Unambiguous IRIs are an integral part of the standards and the RDF model is even extended in some formats. Therefore semantic standards are awarded the best possible rating (++).

Table 4.2 sums up the discussion in a more clearly arranged format.

	Readability	Flexibility	Syntax	Semantics
Plain text description	++	--	-	--
Static description formats	+	-	++	--
Dynamic description formats	+	+	o	+
Semantic description formats	o	++	o	++

Table 4.2: Evaluation of the different web service description approaches.



## 4.2 Discovery Workflow Implementation

For the validation of the proposed workflow concept, a minimal working example was implemented. The workflow handling system was deployed using the Java Jetty server framework.

To add workflow, workflow item, service mapping and service-specific context descriptions, the server periodically checks its filesystem for new files, parses them, and, on success, adds them to its internal repository.

Starting from the entry point of the user interface, a user client is provided with HATEOAS conforming responses, using JSON+LD as the default content type and Hydra, schema.org and dublin-core as defined recognized vocabularies. This enables clients that recognize all these vocabularies to interact with the user interface.

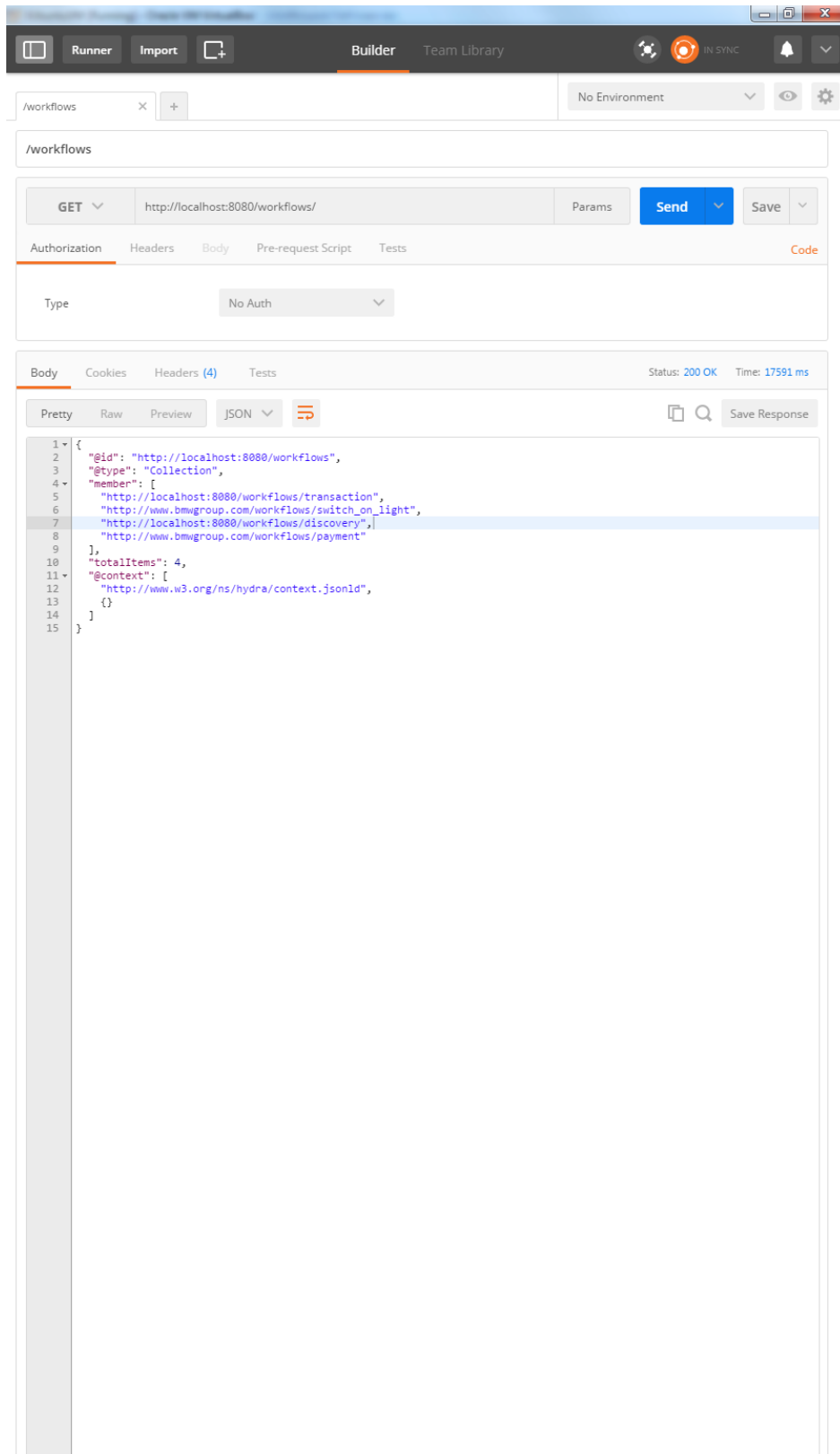
Before the workflow handling system is able to make use of a service, a client has to authorize it. As has already been discussed, a popular standard to authorize clients to act on someone's behalf is OAuth. Depending on the protocol flow used by an API, different scenarios to authorize the workflow handling system with the API are possible: the workflow handling system may redirect the authentication step to a user's phone, where the user may easily confirm his identity and authorize the client. The token endpoint would then be redirected to the workflow handling service system, which would then extract the token and refresh it automatically, everytime it has expired. Another possibility would be that the user has to use the user configuration interface to manually enter their credentials and step through the authorization flow.

To demonstrate the interaction of a client with the workflow handling system, a workflow with only a single generic workflow item was described. The workflow may be considered as a discovery workflow, expecting a location's latitude and longitude, a search radius around the location, the endpoint of a service that should be used and a limit to the locations that are returned. The whole workflow then returns an array of locations that have been discovered.

As the client implementation is out of scope for the considerations within this thesis, the client is simulated by using **Postman**, a Google Chrome plugin, that supports excellent tooling for producing and analyzing HTTP interaction.

The endpoint for a client wishing to interact with the workflow handling system was the resource at `/workflows/`. The base URI is depending on the actual authority providing this functionality in a productive system. For this PoC the server was set up locally and thus running on `localhost:8080`. A client may therefore retrieve a collection of the available workflows by performing an HTTP GET on the endpoint resource `http://localhost:8080/workflows/`. Figure 4.1 illustrates the response obtained using Postman.

## 4 Results



The screenshot displays a REST client interface with the following details:

- Request:** Method: GET, URL: `http://localhost:8080/workflows/`, Authorization: No Auth.
- Response:** Status: 200 OK, Time: 17591 ms.
- Response Body (JSON):**

```
1 {
2   "id": "http://localhost:8080/workflows",
3   "type": "Collection",
4   "members": [
5     "http://localhost:8080/workflows/transaction",
6     "http://www.bmigroup.com/workflows/switch_on_light",
7     "http://localhost:8080/workflows/discovery",
8     "http://www.bmigroup.com/workflows/payment"
9   ],
10  "totalItems": 4,
11  "@context": [
12    "http://www.w3.org/ns/hydra/context.jsonld",
13    {}
14  ]
15 }
```

Figure 4.1: Response of the endpoint resource.

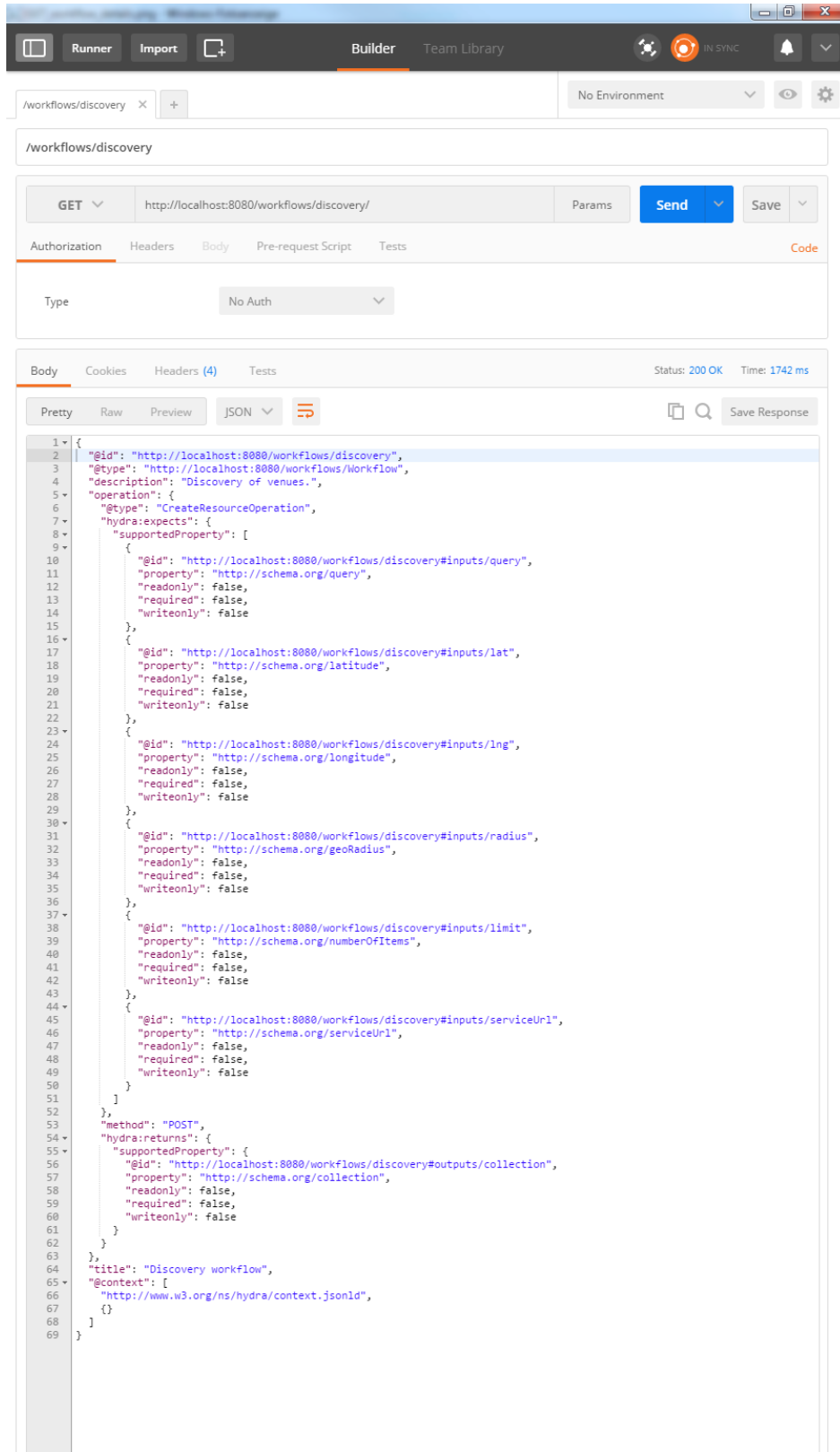
## 4.2 Discovery Workflow Implementation

The retrieved resource representation proves to be of type `hydra:Collection` and lists all workflows that are currently available in the repository. The user agent may discover the expected inputs and outputs of a workflow by performing an HTTP GET on a single workflow member, thus retrieving a detailed description of the workflow. Based on the output description, a user agent may then be able to decide whether the workflow satisfies the agent's objectives and whether the agent wants to invoke it. Figure 4.2 presents the response for an HTTP GET request to retrieve a detailed description of the discovery workflow.

If examined closely, it may be observed that the `vocab:mapping` and `vocab:items` properties are not included in the representation. This is due to the definition that workflows may be opaque for the user. Moreover, the user interface was defined to only expose Hydra, schema.org and dublin-core terms. Exposing proprietary defined terms would violate this constraint. Assuming a client is content with the output generated by a workflow, it will consequently try to invoke it. Upon recognizing the property `hydra:operation` in the retrieved workflow description depicted in figure 4.2, a client comprehends the instructions provided to perform the operation and acts accordingly. As a result, the client performs an HTTP POST to the `http://localhost:8080/workflows/discovery/` resource, which creates an instance of the discovery workflow and executes it. Upon successful execution, the server returns the status code `201: Created` and appends the `Location` header containing the ID of the resource representing the workflow result. This practice is recommended in the official HTTP 1.1 specifications and part of the semantics of POST. Therefore, some standard client programs, like Postman, automatically follow the link provided in the `Location` header by immediately performing an HTTP GET to the provided URI. Therefore, the response in figure 4.3 indicates that the server responds right away with the workflow results.

The results of the workflow are as-well serialized in JSON-LD and conforming to the vocabularies defined to be recognized. The result of a workflow may be stored for future applications and retrieved by an HTTP GET request as shown in figure 4.4.

## 4 Results



The screenshot displays a REST client interface with the following details:

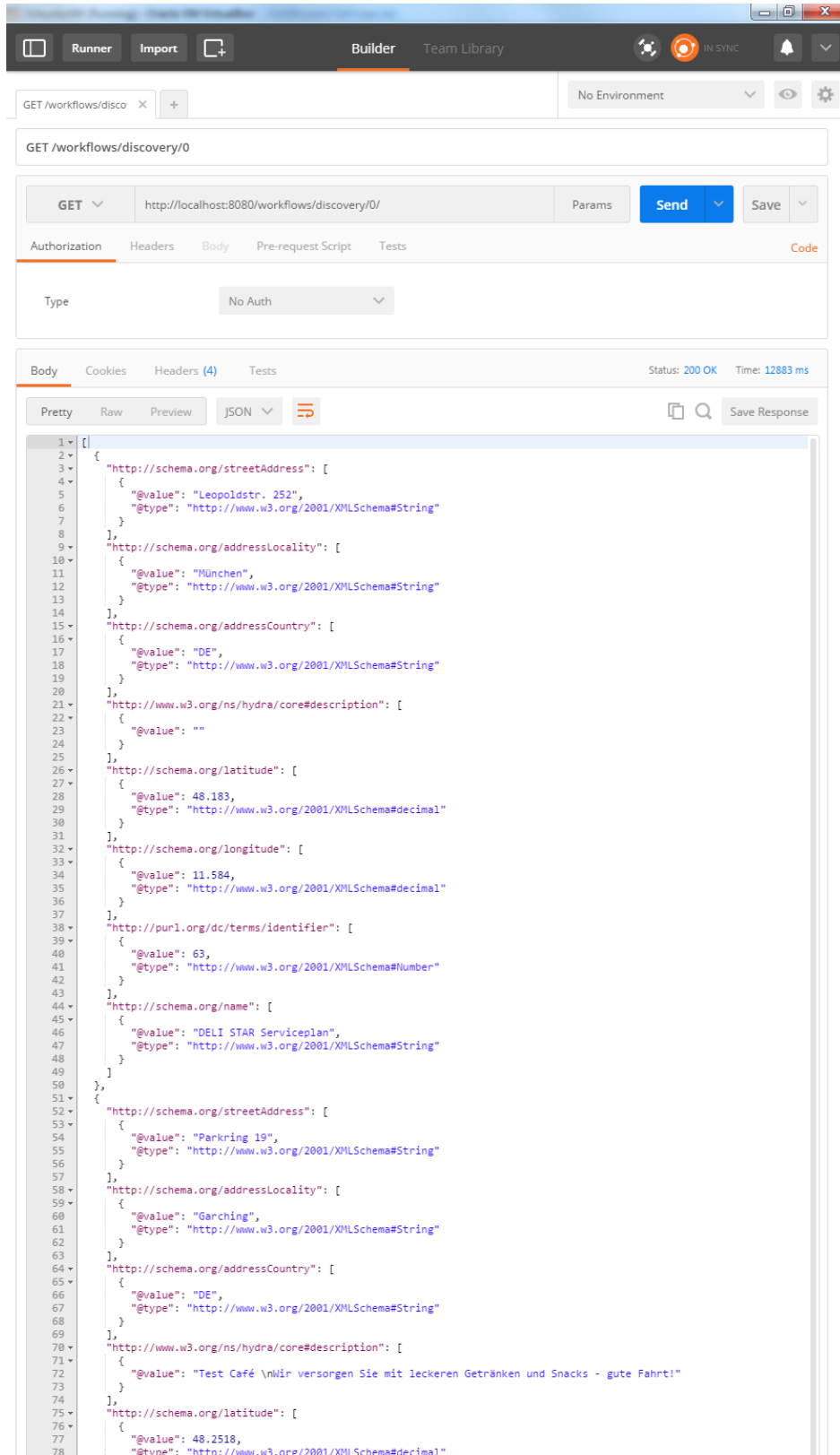
- Runner:** Builder, Team Library
- Environment:** No Environment
- Request:** GET http://localhost:8080/workflows/discovery/
- Authorization:** No Auth
- Status:** 200 OK, Time: 1742 ms
- Response Body (JSON):**

```
1 {
2   "@id": "http://localhost:8080/workflows/discovery",
3   "@type": "http://localhost:8080/workflows/Workflow",
4   "description": "Discovery of venues.",
5   "operation": {
6     "@type": "CreateResourceOperation",
7     "hydra:expects": {
8       "supportedProperty": [
9         {
10          "@id": "http://localhost:8080/workflows/discovery#inputs/query",
11          "property": "http://schema.org/query",
12          "readonly": false,
13          "required": false,
14          "writeonly": false
15        },
16        {
17          "@id": "http://localhost:8080/workflows/discovery#inputs/lat",
18          "property": "http://schema.org/latitude",
19          "readonly": false,
20          "required": false,
21          "writeonly": false
22        },
23        {
24          "@id": "http://localhost:8080/workflows/discovery#inputs/lng",
25          "property": "http://schema.org/longitude",
26          "readonly": false,
27          "required": false,
28          "writeonly": false
29        },
30        {
31          "@id": "http://localhost:8080/workflows/discovery#inputs/radius",
32          "property": "http://schema.org/geoRadius",
33          "readonly": false,
34          "required": false,
35          "writeonly": false
36        },
37        {
38          "@id": "http://localhost:8080/workflows/discovery#inputs/limit",
39          "property": "http://schema.org/numberOfItems",
40          "readonly": false,
41          "required": false,
42          "writeonly": false
43        },
44        {
45          "@id": "http://localhost:8080/workflows/discovery#inputs/serviceUrl",
46          "property": "http://schema.org/serviceUrl",
47          "readonly": false,
48          "required": false,
49          "writeonly": false
50        }
51      ]
52    },
53   "method": "POST",
54   "hydra:returns": {
55     "supportedProperty": {
56       "@id": "http://localhost:8080/workflows/discovery#outputs/collection",
57       "property": "http://schema.org/collection",
58       "readonly": false,
59       "required": false,
60       "writeonly": false
61     }
62   }
63 },
64 "title": "Discovery workflow",
65 "@context": {
66   "http://www.w3.org/ns/hydra/context.jsonld",
67   {}
68 }
69 }
```

Figure 4.2: Detailed description of a workflow resource.



## 4 Results



The screenshot shows the Swagger UI interface for a REST API. The URL is `http://localhost:8080/workflows/discovery/0`. The response status is `200 OK` and the time taken is `12883 ms`. The response body is displayed in JSON format, showing two workflow instances.

```
1  [
2  {
3    "http://schema.org/streetAddress": [
4      {
5        "@value": "Leopoldstr. 252",
6        "@type": "http://www.w3.org/2001/XMLSchema#String"
7      }
8    ],
9    "http://schema.org/addressLocality": [
10   {
11     "@value": "München",
12     "@type": "http://www.w3.org/2001/XMLSchema#String"
13   }
14 ],
15 "http://schema.org/addressCountry": [
16 {
17   "@value": "DE",
18   "@type": "http://www.w3.org/2001/XMLSchema#String"
19 }
20 ],
21 "http://www.w3.org/ns/hydra/core#description": [
22 {
23   "@value": ""
24 }
25 ],
26 "http://schema.org/latitude": [
27 {
28   "@value": 48.183,
29   "@type": "http://www.w3.org/2001/XMLSchema#decimal"
30 }
31 ],
32 "http://schema.org/longitude": [
33 {
34   "@value": 11.584,
35   "@type": "http://www.w3.org/2001/XMLSchema#decimal"
36 }
37 ],
38 "http://purl.org/dc/terms/identifier": [
39 {
40   "@value": 63,
41   "@type": "http://www.w3.org/2001/XMLSchema#Number"
42 }
43 ],
44 "http://schema.org/name": [
45 {
46   "@value": "DELI STAR Serviceplan",
47   "@type": "http://www.w3.org/2001/XMLSchema#String"
48 }
49 ],
50 },
51 {
52 "http://schema.org/streetAddress": [
53 {
54   "@value": "Parkring 19",
55   "@type": "http://www.w3.org/2001/XMLSchema#String"
56 }
57 ],
58 "http://schema.org/addressLocality": [
59 {
60   "@value": "Garching",
61   "@type": "http://www.w3.org/2001/XMLSchema#String"
62 }
63 ],
64 "http://schema.org/addressCountry": [
65 {
66   "@value": "DE",
67   "@type": "http://www.w3.org/2001/XMLSchema#String"
68 }
69 ],
70 "http://www.w3.org/ns/hydra/core#description": [
71 {
72   "@value": "Test Café \nWir versorgen Sie mit leckeren Getränken und Snacks - gute Fahrt!"
73 }
74 ],
75 "http://schema.org/latitude": [
76 {
77   "@value": 48.2518,
78   "@type": "http://www.w3.org/2001/XMLSchema#decimal"
79 }
```

Figure 4.4: Retrieve an existing workflow instance.

## 4.3 Lessons learned

Following the implementation of the workflow handling system concept, this section discusses issues that have emerged, proposes strategies to solve them and draws the lessons learned during the implementation of the concept.

A major challenge was to generalize web services to enable workflow items to bind to more than one service. From an economic point of view, in an open unregulated market, every competitor tries to outperform the others by providing more value. Thus, even when service providers offer similar essential functionality (Foursquare and Yelp are both offering venue discovery, rating, etc.) they will try to distinguish from one another in offering what they perceive as the best technical solution to provide a service. Thus, there is a fine line between using standards and creating individual solutions. Generic service abstractions make sense when, for example, quality and coverage of data vary depending on locality. While YQ may offer great service coverage in New Zealand, Tapingo may provide a similar service quality in California. This aspect is one of the main reasons to try and enforce a generic approach. However, most service providers typically develop their solutions independent from one another. Moreover, they may use different standards or interpret a standard's specification differently. Innovative service implementations may even have to propose new approaches that are not standardized at all. These circumstances lead to a host of implementation challenges.

When requesting a representation or a collection of representations, most providers adhere to the semantics of HTTP verbs and use GET to retrieve a representation from a server. However, some providers violate the semantics of HTTP verbs. In some client as well as server implementations, HTTP GET is only supported to a certain length of requests (2-8kB in modern browsers for example). Thus, some providers abuse HTTP's semantics and use HTTP POST instead, as there is no length limit for requests. One example would be the menu discovery service provider [locu](#) that changed its invocation style from using HTTP GET in its v1 API to HTTP POST in its v2 API. Exceptions like this are very hard to map in a service mapping description. Nevertheless, most providers adhere to the semantics imposed by HTTP and use HTTP GET.

A related issue arose when reviewing the possibilities of including parameters in a request. Parameters may be appended directly to a URI, using the embedded query mechanism. Another option would be to transfer key-value pairs as HTTP headers. A third possibility is to insert key-value pairs into the body of a request. This could even prove to be more complex, as data in the body may be formatted using an arbitrary Internet media-type, or even worse, vendor specific, unregistered Internet media-type. According to the HTTP specifications, a payload in an HTTP GET request body does not have defined semantics. However, best practices usually stipulate that parameters that are used in querying a collection resource are put into the query part of a URI. Hydra provides an appropriate mechanism with the `hydra:IriTemplate` and `hydra:IriTemplateMapping` classes that allow to map a property to a variable in a templated query. In its current version, however, Hydra

## 4 Results

does not provide any terms to annotate, whether specific headers have to be set and in which way they have to be set. Headers play a big part in content negotiation, therefore they are often required by a server. While hydra offers a way to express which parameter an operation expects to be performed, it does not provide semantics to annotate whether they are to be included in a header or in the body part. Moreover there is no term definition of what content-type the body should use.

Aside from the problem of where to actually put parameters, there are also some more complex issues in their representation. Considering two similar services that both require a radius in their request parameters. Both may be mapped and thus disambiguated to some vocabulary property, for example `schema:geoRadius`. In the semantic definition of `schema:geoRadius` the human-readable description states: *Indicates the approximate radius of a GeoCircle (metres unless indicated otherwise via Distance notation)*. When examining the definition of both radius request input parameters definitions, one turns out to be annotated in meters, the other in kilometers. Moreover, the first is defined to be an integer, while the second is specified as floating point number. These issues would require further conversions of both datatype and unit. Hydra provides no functionality whatsoever to tackle these issues. Therefore, the workflow handling system would not be able to represent these services in a single generic workflow item.

As API design is still in transformation, API providers come up with individual concepts and ideas of how to publish their data. The Foursquare venue search API defines something like a conditional required parameter. While this relation is easy to understand for human developers, it may be hard to comprehend by a machine and would probably require a rather verbose description. Conditional requirements are not supported in Hydra and are not considered in the scope of this thesis.

Another peculiarity worth noting is that some workflow items may be dependent on others, as in some would need to be used in combination with specific others. For example, if a discovery workflow item provides an id that needs to be used to construct the templated URI of another service (from the same provider) providing the menu of that venue, there exists an evident dependence. In that case, the menu providing service is bound to the discovery service. It is in general not possible to use the Foursquare-internal ID of a venue to invoke a Yelp service providing details of the same venue. For some domains there might be provider independent IDs that can be used, or even unambiguous URIs that identify a physical location.

While the idea of using unambiguous IRIs from publicly available and internally recognized vocabularies is both promising and intriguing, it can prove to be very cumbersome to be implemented and sometimes even impossible. On the one hand defined semantics of a term should not be abused, as otherwise reliability would be diminished. On the other hand there is no vocabulary able to express every concept in the real world in one term. Thus, vocabularies usually offer rather vague concepts to be used instead of a concise term. For client implementations this is a problem as they would then lack an unambiguous understanding semantics and will need additional (out-of-band) information.

As has already been discussed in chapter 3, more than one instance of the same workflow item may be instantiated within a workflow. However, each workflow



item expects the same inputs and output properties during execution. To enable the unambiguous mapping of workflow inputs and outputs and workflow item inputs and outputs, not the properties themselves are mapped, but the properties describing them in each instance. As each instance is disambiguated during instantiation by requiring a unique name within a workflow, input and output properties of each workflow item instance are also unambiguous. Within a service mapping, properties then have to be unambiguous again (which has to be the case as a service would naturally need unambiguous input parameters).

For the orthogonal authorization component, the proposed extension of Hydra was implemented by setting up a token store. Before each service invocation, the system would check if a valid token exists. Therefore, a token store stores the token details in a hash map with the URI of the service to be authorized as key. The token details include the access token, its expiration time and the refresh token (if available). Depending on the used OAuth protocol flow a token may be refreshed if it has expired or retrieved again.



## 5 Conclusions

This final chapter is intended to provide a short wrap-up of all previous chapters, draw conclusions and highlight the most essential findings. The research questions stated in chapter 1 shall be answered and further implications may provide an outlook and points of reference for future research and implementation work.

### 5.1 Discussion

The most important conclusions that were to be drawn from this thesis regard the answers to the research questions postulated in the very first chapter. They have been stated as follows:

#### **How can web services be described?**

Several approaches for web service description have been examined and compared among each other according to a self-defined rating scheme (see section 4.1). In a nutshell, four fundamental approaches were distinguished: plain textual description, static web service description, dynamic web service description and semantic web service description. Description may essentially be **provided** in two ways: either a priori of interaction or during interaction at runtime. Industrial standards like WADL, Swagger and RAML provide frameworks to describe web services a priori, but fail to satisfy the HATEOAS constraint of REST. HATEOAS conforming standards like HAL, SIREN and Collection+JSON provide a defined generic syntax to enable self-descriptive APIs at runtime, while semantic description standards like JSON-LD+Hydra go one step further and incorporate the RDF data model and IRI defined resources in their approach. Eventually, RESTdesc even extends the RDF data model by including logics and quantification.

#### **How can various web services be composed to realize a specific use case?**

In their work, Sheng et al.[61] offer an elaborate survey of existing methods of web service composition. They distinguish between manual and automatic composition and static and dynamic composition. Manual composition requires a human developer to manually link web services and map their inputs and outputs. Automatic composition tries to incorporate and leverage semantic web technologies and artificial intelligence techniques to attain a defined objective. One such example presented in this thesis was RESTdesc which supports the

## 5 Conclusions

exploration of different paths from a given set of preconditions to a final set of postconditions. However, there are no existing publicly available real-world implementations supporting RESTdesc or similar automatic composition services, therefore the implementation presented in this thesis relies on manual service description. Considering whether the implementation is classified as static or dynamic, it is found that the implementation offers hybrid capabilities: workflow and workflow item descriptions have to be created at design time, but the implementation allows entities invoking a specific workflow to choose a concrete service mapping at runtime. Moreover, newly created workflows may be added instantly to the workflow server, due to the runtime capabilities of the workflow parser. In summary, workflows are static in their functionality but dynamic with respect to service providers.

The results show that the implementation is indeed capable of realizing a stipulated use-case. However, automatic service composition may gain more traction in the future if standards like RESTdesc experience a wider acceptance among web service developers.

### **Which web service description and composition styles are most promising?**

RESTdesc provides an interesting approach for web service description and composition. Rather than focusing on describing technical details of invocation, the creators shift the focus to describing the functionality of a service. However, for this approach to work in practice, the underlying technical layer is required to strictly adhere to specified implementation details. In the world wide web developers tend to apply best practice approaches and not everyone is an expert in RDF graphs. The progress of a proposed architecture is critically dependent on its uptake and adoption in real-life implementations. At the time this thesis was compiled, there could be no popular public web-service found that allowed interaction in a way that RESTdesc could be applied. In conclusion, as one requirement of this thesis was to provide a solution that integrates existing web services, RESTdesc was deemed to be an inappropriate choice, albeit offering promising glimpses for future implementations. A more detailed examination was provided for JSON-LD and the Hydra vocabulary. JSON-LD allows the reuse of existing JSON parser implementations to utilize fundamental semantic web technologies and its popularity is ever increasing. Hydra is still a draft but is already providing useful terms to describe interaction with a resource. Defined semantics for authentication and authorization purposes are lacking in particular and need to be addressed in future extensions. To enable automatic service composition for JSON-LD and Hydra, however, extensions would have to be proposed.

### 5.1.1 Semantic Web Technologies

A fundamental consideration regarding the semantic web is whether the semantic web technology stack may ever deliver what it is promising. For example, the very statement *An important facet of agents' functioning will be the exchange of "proofs" written in the Semantic Web's unifying language* contradicts some of the fundamental design principles of the web as originally stated by Tim Berners-Lee (ironically, thus contradicting himself). Trying to standardize a unifying language for the Semantic Web would indeed constrain users to use a particular language (third principle) and constraining web users to represent their data in an RDF graph triple relation model most certainly constrains the mental model of data into a given pattern. Moreover, if ontologies have to be matched, e.g. by using the `sameAs` property, linking from one system to another is hardly scalable anymore (violating the second principle). Thus, the Semantic Web might never be the next level of the World Wide Web, but a constraint subset of it. Until this state is achieved, linked (open) data may generate enough leverage to demonstrate that the RDF data model may actually create added value in some domains.

### 5.1.2 Classical SOA vs. REST

The decision of which architecture to choose can be tricky and should best be use-case driven, as also remarked by Fielding.[45] While REST has always been defined as an architectural style for hypermedia driven network-based applications, there is no such explicit definition for classical SOA. Numerous discussions go under the title "REST vs. SOAP" although these two can hardly be compared to each other as one is an architectural style, while the other is merely a protocol. In fact, SOAP could theoretically be implemented in a RESTful architecture. However, in classical SOAs, SOAP is complying with WSDL, which per-definition establishes a contract between client and server. SOAP/WSDL APIs usually make extensive use of RPCs which leads to a tight coupling of client and server. This violates the REST principle of uniform interfaces as clients need a priori knowledge. Although possible in theory, describing RESTful APIs in WSDL is rather cumbersome, as WSDL was primarily designed to describe RPCs. In REST, other proposals to describe RESTful APIs have emerged, one of them being WADL, a WSDL-inspired description framework for RESTful API description.

While SOAP is *specified* to be serialized in XML only, there are derivatives of it using different serialization formats like SOAPjr, a hybrid of SOAP and JSON-RPC (thus the suffix jr). SOAPjr allows to still use a SOAP-like specification but on a lower overhead cost than using the more verbose XML format.

In theory, the concept of using SOAP, WSDL and UDDI as a technology stack for web service technologies seems rather intriguing. However, as reality has shown, few public service providers have made use of it, arguably because of the extensive amount of specifications web developers would have to comply with. Imposing this framework on everyone that wants to provide web services would also violate

## 5 Conclusions

several basic principles of the web. Most of the larger public UDDI nodes have been shut down in recent past, seemingly deeming this framework to only exist as a side note in future web service technologies.

As noted before, REST offers principles more in the form of an architectural rationale. A protocol natively occupying support for the REST architecture, and thus often mistakenly confused with REST itself, is HTTP. Whereas HTTP is only used as a transfer means in SOAP, it incorporates many mechanisms that make it comply with the requirements of REST.

Classical SOA offers a wide repository of standards, originally entirely based on XML. Besides SOAP, WSDL and UDDI there is a host of complimentary standards, known as the WS\* standards. These standards are still widely used in the business world of the likes of IBM and SAP, but have high entry barriers for web developers both on the server and the client side. In public APIs a veritable hype has propelled REST to become a buzzword for HTTP APIs in general. Understanding REST and designing truly RESTful APIs is perhaps no less difficult than comprehending SOA standards, thus the popular conception of REST being easier to implement is misleading. Especially the self-descriptive and HATEOAS principles are frequently violated or neglected, as pointed out by Fielding himself<sup>1</sup>. However, simple HTTP APIs that adhere to some of the principles imposed by REST have gained huge popularity and most providers rather abusively name their APIs RESTful to get their share of the hype. Hence, the Richardson maturity model was created to provide a practical framework to validate the "RESTfulness" of an HTTP API.

Hypermedia is arguably great for high-bandwidth, non-time-critical applications. However, for time-critical, low-bandwidth applications like single sensors in the WoT, neither REST nor SOA might not offer the right architectural rationale.

### 5.1.3 Workflow Concept Implementation

The concept proposed in this thesis tries to model complex mash-ups of web services by reducing each service to its functionality and design functional abstract workflow items accordingly. These items are then bound to one or more concrete web service APIs. Wrapping similar web services in generic templates leads to an obvious increase in interoperability, as the abstraction of APIs reduces them to their core functionality. The binding of different service providers to the same generic workflow item enables a single service-mashup to use the data stock of several providers. Consequently, this allows for a combination of locally constraint datasets and detaches third parties from relying on single data providers offering insufficient local data, eventually resulting in hyperlocal service-mashups. While generic workflow item classes increase interoperability, they severely constrain the individual services they are wrapping. For a certain domain some functionality might be considered fundamental as there are recurring concepts in each application, trying to solve domain-specific problems. However, individual characteristics of an API can also be

---

<sup>1</sup><http://roy.gbiv.com/untangled/2008/rest-apis-must-be-hypertext-driven>

crucial unique selling points for its provider. Here, a dual approach was proposed, allowing developers to create generic workflow items offering basic functionality for mash-up purposes, as well as specific workflow items with extensive functionality for more sophisticated applications.

The main benefit in using a standardized workflow server infrastructure lies in a reduced effort to deploy and validate every single service-mash-up. Workflow item descriptions may be reused and thus not validated again. The validation process of each workflow is reduced and may only involve tests for functionality, as the surrounding infrastructure assures other orthogonal aspects.

## 5 Conclusions

### 5.2 Implications

One of the existing challenges in expanding the idea of the semantic web is to tear down its sometimes rather high barriers to entry. Providing a triple or quad store in combination with a highly available and secure SPARQL endpoint is rather costly and demands knowledgeable experts in implementing and maintaining them. Also, the idea of The Semantic Web originally intended to encourage decentralized data retention rather than treasuring up data within encapsulated silos.

The initial vision of a single catholic ontology has been overtaken by reality, as the real world is too versatile, too individual to be mapped to unambiguous and atomic definitions. Domain-specific vocabularies, delimiting functionality but enabling interoperability, have shown promising results. Semantics may only be defined by reaching a consensus in a community of users. This process is cumbersome and could potentially lead to numerous debates. The main challenge of semantic web technologies lies in providing developers an infrastructure generic and well-defined enough to be interoperable, yet still open enough to allow them to set themselves apart from competition. In the recent past, controlled vocabularies like schema.org have increasingly gained leverage as they have been adopted by heavyweights of the software industry. In conclusion, different and more lightweight linked data approaches like JSON-LD could bridge the gap between semantic and existing conventional web technologies in the future.

Standardized service description will further evolve to cope with the growing increase of ubiquitous data. As human beings will be joined in the world wide web by an increasing number of machines, browsing the web has to be made accessible for them too. Thus, describing web services in plain text in an HTML document only will be deprecated soon. Content negotiation is already allowing clients to customize a retrieved representation depending on their nature.

Service composition will still have to be carried out manually. Unless service descriptions are widely adapting technologies like RESTdesc, the challenge of automatic service composition persists.

Security and trust consideration will continue to be major focus points of future research. However, by establishing standards in these fields, a seamless integration with semantic or linked data technologies seems possible. OAuth and OpenID are just two examples containing the threats imposed by potential attackers.

For consumer goods providers such as BMW, wrapping third-party services in semantically meaningful representation may provide short-time relief from the growing demands of their consumer base. In the rapidly transforming world wide web, however, consensus finding will prove to be the key to enable and facilitate interoperability.



# Appendix

## 5 Conclusions

### Additional Listings

---

```
<?xml version="1.0" encoding="UTF-8"?>
<rdf:RDF
  xmlns:rdf="http://www.w3.org/1999/02/22-rdf-syntax-ns#"
  xmlns:schema="http://schema.org/"
>
  <rdf:Description rdf:about="http://example.org/stadiums#Anfield">
    <schema:image rdf:datatype="http://www.w3.org/2001/XMLSchema#anyURI">
      https://upload.wikimedia.org/wikipedia/commons/8/86/Anfield%2C_20_October_2012.jpg
    </schema:image>
    <rdf:type rdf:resource="http://schema.org/Place"/>
    <schema:address rdf:nodeID="f1d8ef1e30572430d89e346c9e4490257b1"/>
    <schema:name>Anfield</schema:name>
  </rdf:Description>
  <rdf:Description rdf:nodeID="f1d8ef1e30572430d89e346c9e4490257b1">
    <schema:addressLocality>Liverpool</schema:addressLocality>
    <schema:addressCountry xml:lang="en">United
      Kingdom</schema:addressCountry>
    <schema:streetAddress>Anfield Rd</schema:streetAddress>
    <schema:addressCountry xml:lang="de">Vereinigtes
      Koenigreich</schema:addressCountry>
    <rdf:type rdf:resource="http://schema.org/PostalAddress"/>
  </rdf:Description>
  <rdf:Description rdf:about="http://example.org/cities#Liverpool">
    <rdf:type rdf:resource="http://schema.org/Place"/>
    <schema:containsPlace
      rdf:resource="http://example.org/stadiums#anfield"/>
    <schema:containsPlace rdf:nodeID="ub30bL13C15"/>
  </rdf:Description>
  <rdf:Description rdf:nodeID="ub30bL13C15">
    <schema:name>Goodison Park</schema:name>
  </rdf:Description>
</rdf:RDF>
```

---

Listing 1: Formulation of an RDF graph in RDF/XML

---

```

<div xmlns="http://www.w3.org/1999/xhtml"
  prefix="
    schema: http://schema.org/
    rdf: http://www.w3.org/1999/02/22-rdf-syntax-ns#
    xsd: http://www.w3.org/2001/XMLSchema#
    rdfs: http://www.w3.org/2000/01/rdf-schema#"
  >
  <div typeof="schema:Place" about="http://example.org/cities#Liverpool">
    <div rel="schema:containsPlace"
      resource="http://example.org/stadiums#anfield"></div>
    <div rel="schema:containsPlace">
      <div typeof="rdfs:Resource">
        <div property="schema:name" content="Goodison Park"></div>
      </div>
    </div>
  </div>
  <div typeof="schema:Place" about="http://example.org/stadiums#Anfield">
    <div rel="schema:address">
      <div typeof="schema:PostalAddress">
        <div property="schema:addressCountry" xml:lang="de"
          content="Vereinigtes Koenigreich"></div>
        <div property="schema:addressLocality" content="Liverpool"></div>
        <div property="schema:streetAddress" content="Anfield Rd"></div>
        <div property="schema:addressCountry" xml:lang="en" content="United
          Kingdom"></div>
      </div>
    </div>
    <div property="schema:name" content="Anfield"></div>
    <div property="schema:image" datatype="xsd:anyURI"
      content="https://upload.wikimedia.org/wikipedia/
      commons/8/86/Anfield%2C_20_October_2012.jpg"></div>
  </div>
</div>

```

---

Listing 2: Formulation of an RDF graph in RDFa



## List of Abbreviations

**API** ... Application Programming Interface  
**BCP** ... Best Current Practice  
**CoAP** ... Constrained Application Protocol  
**CRUD** ... Create, Read, Update, Delete  
**CURIE** ... Compact URI  
**CWA** ... Closed World Assumption  
**DNS** ... Domain Name System  
**FTP** ... File Transfer Protocol  
**HAL** ... Hypertext Application Language  
**HATEOAS** ... Hypermedia As The Engine of Application State  
**HMI** ... Human Machine Interaction  
**HTML** ... Hypertext Markup Language  
**HTTP** ... Hypertext Transfer Protocol  
**IANA** ... Internet Assigned Numbers Authority  
**IETF** ... Internet Engineering Task Force  
**IoT** ... Internet of Things  
**IP** ... Internet Protocol  
**IRI** ... Internationalized Resource Identifier  
**JSON** ... JavaScript Object Notation  
**JSON-LD** ... JSON for Linked Data  
**LD** ... Linked Data  
**LOD** ... Linked Open Data  
**MIME** ... Multipurpose Internet Mail Extensions  
**MQTT** ... Message Queuing Telemetry Transport  
**OWA** ... Open World Assumption  
**OWL** ... Web Ontology Language  
**PoC** ... Proof-of-concept  
**RDF** ... Resource Description Framework  
**REST** ... Representational State Transfer  
**RFC** ... Request for Comments  
**RPC** ... Remote Procedure Call  
**SAWSDL** ... Semantic Annotations for WSDL  
**SOA** ... Service Oriented Architecture  
**SOAP** ... Simple Object Access Protocol  
**SPARQL** ... SPARQL Protocol And RDF Query Language  
**TCP** ... Transmission Control Protocol  
**TLS** ... Transport Layer Security

## 5 Conclusions

**Turtle** ... Terse RDF Triple Language

**UCS** ... Universal Character Set

**UDP** ... User Datagram Protocol

**URI** ... Uniform Resource Identifier

**URL** ... Uniform Resource Locator

**URN** ... Uniform Resource Name

**US-ASCII** ... US-American Standard Code for Information Interchange

**UTF-8** ... UCS Transformation Format

**W<sub>3</sub>C** ... World Wide Web Consortium

**WoT** ... Web of Things

**WWW** ... World Wide Web

**XML** ... Extensible Markup Language

**YAML** ... YAML Ain't Markup Language

## List of Figures

2.1	WWW architecture diagram from 1990. . . . .	4
2.2	The Semantic Web Technology Stack. . . . .	5
2.3	Fundamental data model of RDF. . . . .	14
2.4	Example of an RDF graph. . . . .	16
2.5	The structure of OWL 2. . . . .	24
2.6	The WSDL 2.0 description framework. . . . .	30
2.7	The Richardson Maturity Model. . . . .	34
2.8	Exemplary web service description in plain text. . . . .	36
2.9	The fundamental data model of HAL. . . . .	42
2.10	Graph illustrating the Hydra core vocabulary in its current form. . . . .	45
2.11	RESTdesc automatic service composition. . . . .	48
2.12	Abstract OAuth 2.0 Protocol Flow. . . . .	50
3.1	General workflows concept. . . . .	54
3.2	Overall architecture abstraction. . . . .	56
3.3	Basic components of the User Interface. . . . .	58
3.4	Basic components of the Developer Interface. . . . .	59
3.5	Basic components of the Third-Party Service Interface. . . . .	61
3.6	Exemplary workflow description resource. . . . .	63
3.7	Exemplary hydra:Operation class object. . . . .	65
3.8	Exemplary instantiation of workflow items. . . . .	66
3.9	Exemplary mapping of workflow item instances. . . . .	68
3.10	Overall graph of an exemplary workflow description. . . . .	69
3.11	Exemplary workflow item description resource. . . . .	71
3.12	Exemplary workflow item input description resource. . . . .	73
3.13	Exemplary workflow item output description resource. . . . .	73
3.14	Overall graph of an exemplary workflow item description. . . . .	74
4.1	Response of the endpoint resource. . . . .	96
4.2	Detailed description of a workflow resource. . . . .	98
4.3	Execute a workflow. . . . .	99
4.4	Retrieve an existing workflow instance. . . . .	100





# Listings

2.1	Basic XML element annotation	12
2.2	XML element attributes	12
2.3	XML comment	12
2.4	Basic JSON key-value pair annotation	13
2.5	JSON datatype syntax	13
2.6	Formulation of an RDF statement in Turtle syntax	14
2.7	Formulation of an RDF graph in Turtle	18
2.8	Formulation of an RDF graph in JSON-LD	22
2.9	Example of a WADL service description	38
2.10	Swagger skeleton code	39
2.11	Exemplary account balance HAL resource	43
2.12	Derivation of a RESTdesc service description	47
2.13	Example of a RESTdesc service description	47
2.14	RESTdesc service description with pre- and postcondition	48
3.1	Basic structure of the workflow description concept	62
3.2	Exemplary hydra:Operation class object	64
3.3	Exemplary instantiation of workflow items	66
3.4	Example of input/output mapping of workflow items	67
3.5	Basic structure of a workflow item description	71
3.6	Input and output property description of a workflow item	72
3.7	Basic structure of a hydra:ApiDocumentation class object	75
3.8	Hydra description of an API in the form of a templated URI query	76
3.9	Exemplary hydra:entrypoint	77
3.10	Example for a plain JSON reponse	78
3.11	Context injection template	78
3.12	Exemplary context-injection	79
3.13	JSON-LD-converted JSON response after context-injection	79
3.14	Example for a nested class definition	80
3.15	Context-injection converted nested class definition	81
3.16	Context injection template	82
3.17	Nested objects response with ambiguities	83
3.18	Disambiguated JSON-LD converted response	83
3.19	Proposed Hydra OAuth extension	84
1	Formulation of an RDF graph in RDF/XML	112
2	Formulation of an RDF graph in RDFa	113



## Bibliography

- [1] Corporate Communications. *BMW Group driving the transformation of individual mobility with its Strategy NUMBER ONE ; Next*. Munich, 2016-03-16. URL: <https://www.press.bmwgroup.com/global/article/detail/T0258269EN/bmw-group-driving-the-transformation-of-individual-mobility-with-its-strategy-number-one-next?language=en> (visited on 06/20/2016) (cit. on p. 1).
- [2] T. Berners-Lee. *The World Wide Web: Past, Present and Future*. Tech. rep. W3C, 1996. URL: <https://www.w3.org/People/Berners-Lee/1996/ppf.html> (visited on 09/30/2016) (cit. on p. 3).
- [3] Tim Berners-Lee, James Hendler, Ora Lassila, et al. "The semantic web." In: *Scientific american* 284.5 (2001), pp. 28–37 (cit. on pp. 5, 44).
- [4] T. Berners-Lee. *Universal Resource Identifiers in WWW: A Unifying Syntax for the Expression of Names and Addresses of Objects on the Network as used in the World-Wide Web*. RFC 1630. IETF, 1994. URL: <https://www.ietf.org/rfc/rfc1630.txt> (visited on 06/17/2016) (cit. on p. 6).
- [5] J. Kunze. *Functional Recommendations for Internet Resource Locators*. RFC 1736. IETF, 1995. URL: <https://tools.ietf.org/html/rfc1736> (visited on 06/17/2016) (cit. on p. 6).
- [6] K. Sollins and Masinter L. *Functional Requirements for Uniform Resource Names*. RFC 1737. IETF, 1994. URL: <https://tools.ietf.org/html/rfc1737> (visited on 06/17/2016) (cit. on p. 6).
- [7] T. Berners-Lee, R. T. Fielding, and Masinter L. *Uniform Resource Identifier (URI): Generic Syntax*. RFC 3986. IETF, 2005. URL: <https://tools.ietf.org/html/rfc3986> (visited on 06/17/2016) (cit. on pp. 6, 7).
- [8] M. Duerst and M. Suignard. *Internationalized Resource Identifiers (IRIs)*. RFC 3987. IETF, 2005. URL: <https://tools.ietf.org/html/rfc3987> (visited on 06/17/2016) (cit. on pp. 7, 10).
- [9] D. Thaler, T. Hansen, and T. Hardie. *Guidelines and Registration Procedures for URI Schemes*. BCP 35. IETF, 2015. URL: <https://www.rfc-editor.org/bcp/bcp35.txt> (visited on 09/30/2016) (cit. on p. 7).
- [10] M. Birbeck and S. McCarron. *CURIE Syntax 1.0: A syntax for expressing Compact URIs*. Ed. by W3C. 2010. URL: <https://www.w3.org/TR/curie/curie.pdf> (visited on 09/30/2016) (cit. on p. 8).

## Bibliography

- [11] T. Berners-Lee, R. T. Fielding, and H. Frystyk. *Hypertext Transfer Protocol – HTTP/1.0*. RFC 1945. IETF, 1996. URL: <https://tools.ietf.org/html/rfc1945> (visited on 10/26/2016) (cit. on p. 9).
- [12] R. T. Fielding and J. Reschke. *Hypertext Transfer Protocol (HTTP/1.1): Message Syntax and Routing*. RFC 7230. IETF, 2014. URL: <https://tools.ietf.org/html/rfc7230> (visited on 10/26/2016) (cit. on p. 9).
- [13] R. T. Fielding and J. Reschke. *Hypertext Transfer Protocol (HTTP/1.1): Semantics and Content*. RFC 7231. IETF, 2014. URL: <https://tools.ietf.org/html/rfc7231> (visited on 10/26/2016) (cit. on pp. 9, 10).
- [14] R. T. Fielding and J. Reschke. *Hypertext Transfer Protocol (HTTP/1.1): Conditional Requests*. RFC 7232. IETF, 2014. URL: <https://tools.ietf.org/html/rfc7232> (visited on 10/26/2016) (cit. on p. 9).
- [15] R. T. Fielding, Y. Lafon, and J. Reschke. *Hypertext Transfer Protocol (HTTP/1.1): Range Requests*. RFC 7233. IETF, 2014. URL: <https://tools.ietf.org/html/rfc7233> (visited on 10/26/2016) (cit. on p. 9).
- [16] R. T. Fielding, M. Nottingham, and J. Reschke. *Hypertext Transfer Protocol (HTTP/1.1): Caching*. RFC 7234. IETF, 2014. URL: <https://tools.ietf.org/html/rfc7234> (visited on 10/26/2016) (cit. on p. 9).
- [17] R. T. Fielding and J. Reschke. *Hypertext Transfer Protocol (HTTP/1.1): Authentication*. RFC 7235. IETF, 2014. URL: <https://tools.ietf.org/html/rfc7235> (visited on 10/26/2016) (cit. on p. 9).
- [18] M. Belshe, R. Peon, and Thomson M. *Hypertext Transfer Protocol Version 2 (HTTP/2)*. RFC 7540. IETF, 2015. URL: <https://tools.ietf.org/html/rfc7540> (visited on 10/26/2016) (cit. on p. 9).
- [19] J. Gregorio et al. *URI Template*. RFC 6570. IETF, 2012. URL: <https://tools.ietf.org/html/rfc6570> (visited on 10/26/2016) (cit. on p. 11).
- [20] M. Nottingham. *Web Linking*. RFC 5988. IETF, 2012. URL: <https://tools.ietf.org/html/rfc5988> (visited on 10/26/2016) (cit. on p. 11).
- [21] T. Bray et al. *Extensible Markup Language (XML) 1.0 (Fifth Edition)*. Ed. by W3C. 2008. URL: <https://www.w3.org/TR/xml/> (visited on 10/27/2016) (cit. on p. 12).
- [22] T. Bray. *The JavaScript Object Notation (JSON) Data Interchange Format*. RFC 7159. IETF, 2014. URL: <https://tools.ietf.org/html/rfc7159> (visited on 10/27/2016) (cit. on p. 12).
- [23] G. Schreiber and Y. Raimond. *RDF 1.1 Primers*. Ed. by W3C. 2014. URL: <https://www.w3.org/TR/rdf11-primer/> (visited on 09/30/2016) (cit. on p. 13).
- [24] R. Cyganiak, D. Wood, and M. Lanthaler. *RDF 1.1 Concepts and Abstract Syntax*. Ed. by W3C. 2014. URL: <https://www.w3.org/TR/rdf11-concepts/> (visited on 09/30/2016) (cit. on p. 14).

- [25] A. Stolz, B. Rodriguez-Castro, and M. Hepp. *RDF Translator: A RESTful Multi-Format Data Converter for the Semantic Web*. Tech. rep. 2013. URL: <http://www.stalsoft.com/publications/rdf-translator-TR.pdf?format=pdf> (visited on 06/30/2016) (cit. on p. 17).
- [26] F. Gandon and G. Schreiber. *RDF 1.1 XML Syntax*. Ed. by W3C. 2014. URL: <https://www.w3.org/TR/rdf-syntax-grammar/> (visited on 09/30/2016) (cit. on p. 17).
- [27] I. Herman et al. *RDFa 1.1 Primer - Third Edition*. Ed. by W3C. 2015. URL: <https://www.w3.org/TR/rdfa-primer/> (visited on 09/30/2016) (cit. on p. 17).
- [28] M. Sporny. *RDFa Lite 1.1 - Second Edition*. Ed. by W3C. 2015. URL: <https://www.w3.org/TR/rdfa-lite/> (visited on 09/30/2016) (cit. on p. 17).
- [29] B. Adida et al. *RDFa Core 1.1 - Third Edition*. Ed. by W3C. 2015. URL: <https://www.w3.org/TR/rdfa-core/> (visited on 09/30/2016) (cit. on p. 17).
- [30] D. Beckett et al. *RDF 1.1 Turtle: Terse RDF Triple Language*. Ed. by W3C. 2014. URL: <https://www.w3.org/TR/turtle/> (visited on 09/30/2016) (cit. on p. 18).
- [31] Markus Lanthaler and Christian Gütl. "On using JSON-LD to create evolvable RESTful services." In: *Proceedings of the Third International Workshop on RESTful Design*. ACM. 2012, pp. 25–32 (cit. on p. 19).
- [32] Ian Davis, Thomas Steiner, and Arnaud Le Hors. *RDF 1.1 JSON Alternate Serialization (RDF/JSON)*. Ed. by W3C. 2013. URL: <https://www.w3.org/TR/rdf-json/> (visited on 10/26/2016) (cit. on p. 19).
- [33] Markus Lanthaler and Christian Gütl. "A semantic description language for RESTful data services to combat semaphobia." In: *5th IEEE International Conference on Digital Ecosystems and Technologies (IEEE DEST 2011)*. IEEE. 2011, pp. 47–53 (cit. on pp. 19, 26).
- [34] Manu Sporny, Gregg Kellogg, and Markus Lanthaler. *JSON-LD 1.0: A JSON-based Serialization for Linked Data*. Ed. by W3C. 2014. URL: <https://www.w3.org/TR/json-ld/> (visited on 10/26/2016) (cit. on p. 19).
- [35] Markus Lanthaler, Gregg Kellogg, and Manu Sporny. *JSON-LD 1.0 Processing Algorithms and API*. Ed. by W3C. 2014. URL: <https://www.w3.org/TR/json-ld-api/> (visited on 10/26/2016) (cit. on p. 19).
- [36] ontology. *Chicago: Encyclopaedia Britannica*. Encyclopaedia Britannica, 2016. URL: <https://www.britannica.com/topic/ontology-metaphysics> (cit. on p. 23).
- [37] Woody Pidcock. "What are the differences between a vocabulary, a taxonomy, a thesaurus, an ontology, and a meta-model?" In: (Jan. 2003). URL: <http://www.metamodel.com/article.php?story=20030115211223271> (cit. on p. 23).
- [38] W3C OWL Working Group. *OWL 2 Web Ontology Language*. Ed. by W3C. 2012. URL: <https://www.w3.org/TR/owl2-overview/> (visited on 10/21/2016) (cit. on p. 23).

## Bibliography

- [39] Markus Kroetzsch, Frantisek Simancik, and Ian Horrocks. “A Description Logic Primer.” In: *CoRR abs/1201.4089* (2012). URL: <http://arxiv.org/abs/1201.4089> (cit. on p. 25).
- [40] Janna Quitney Anderson and Harrison Rainie. *The fate of the Semantic Web*. Pew Internet & American Life Project, 2010 (cit. on p. 26).
- [41] T. Berners-Lee. *Linked Data*. 2006. URL: <https://www.w3.org/DesignIssues/LinkedData.html> (visited on 09/30/2016) (cit. on p. 26).
- [42] Web Applications Working Group. *What are APIs?* Ed. by W3C. 2008. URL: <https://www.w3.org/2008/webapps/> (visited on 07/12/2016) (cit. on p. 28).
- [43] Joel Farrell and Holger Lausen. *Semantic Annotations for WSDL and XML Schema*. Ed. by W3C. 2007. URL: <https://www.w3.org/TR/sawSDL/> (visited on 10/21/2016) (cit. on p. 30).
- [44] L. Clement et al. *UDDI Version 3.0.2*. Ed. by OASIS. 2004. URL: [http://www.uddi.org/pubs/uddi\\_v3.htm](http://www.uddi.org/pubs/uddi_v3.htm) (visited on 10/28/2016) (cit. on p. 31).
- [45] R. T. Fielding. “Architectural Styles and the Design of Network-based Software Architectures.” PhD thesis. 2000 (cit. on pp. 32, 107).
- [46] Jim Webber, Savas Parastatidis, and Ian Robinson. *REST in practice: Hypermedia and systems architecture*. O’Reilly Media, Inc., 2010. ISBN: 0596805829 (cit. on p. 34).
- [47] Ruben Verborgh et al. “Survey of semantic description of rest apis.” In: *rest: Advanced Research Topics and Practical Applications*. Springer, 2014, pp. 69–89 (cit. on pp. 36, 91, 92).
- [48] Marc Hadley. *Web Application Description Language*. Ed. by W3C. 2009. URL: <https://www.w3.org/Submission/wadl/> (visited on 10/21/2016) (cit. on p. 37).
- [49] SmartBear. *What is Swagger?* 2016. URL: <http://swagger.io/getting-started/> (visited on 07/17/2016) (cit. on p. 39).
- [50] K. Zyp and G. Court. *JSON Schema: interactive and non interactive validation*. Internet-Draf. IETF, 2013. URL: <https://tools.ietf.org/html/draft-fge-json-schema-validation-00> (visited on 10/26/2016) (cit. on p. 40).
- [51] Mike Kelly. *JSON Hypertext Application Language*. RFC. IETF, 2016, p. 11. URL: <https://tools.ietf.org/html/draft-kelly-json-hal-08> (visited on 06/17/2016) (cit. on p. 42).
- [52] M. Lanthaler. *Hydra Core Vocabulary: A Vocabulary for Hypermedia-Driven Web APIs*. Ed. by Hydra W3C Community Group. 2016. URL: <http://www.hydra-cg.com/spec/latest/core/> (visited on 10/28/2016) (cit. on p. 44).
- [53] Ruben Verborgh et al. “Description and interaction of RESTful services for automatic discovery and execution.” In: *2011 FTRA International workshop on Advanced Future Multimedia Services (AFMS 2011)*. Future Technology Research Association International (FTRA). 2011 (cit. on p. 46).

- [54] J. Reschke. *HTTP Authentication-Info and Proxy-Authentication-Info Response Header Fields*. RFC 7615. IETF, 2015. URL: <https://tools.ietf.org/html/rfc7615> (visited on 10/30/2016) (cit. on p. 49).
- [55] R. Shekh-Yusef, D. Ahrens, and S. Bremer. *HTTP Digest Access Authentication*. RFC 7616. IETF, 2015. URL: <https://tools.ietf.org/html/rfc7616> (visited on 10/30/2016) (cit. on p. 49).
- [56] J. Reschke. *The 'Basic' HTTP Authentication Scheme*. RFC 7617. IETF, 2015. URL: <https://tools.ietf.org/html/rfc7617> (visited on 10/30/2016) (cit. on pp. 49, 51).
- [57] N. Sakimura et al. *OpenID Connect Core 1.0 incorporating errata set 1*. Tech. rep. 2014. URL: <https://tools.ietf.org/html/rfc7615> (visited on 10/30/2016) (cit. on p. 49).
- [58] D. Hardt. *The OAuth 2.0 Authorization Framework*. RFC 6749. IETF, 2012. URL: <https://tools.ietf.org/html/rfc6749> (visited on 10/30/2016) (cit. on p. 49).
- [59] E. Hammer-Lahav. *The OAuth 1.0 Protocol*. RFC 5849. IETF, 2010. URL: <https://tools.ietf.org/html/rfc5849> (visited on 10/30/2016) (cit. on p. 49).
- [60] M. Jones and D. Hardt. *The OAuth 2.0 Authorization Framework: Bearer Token Usage*. RFC 6750. IETF, 2012. URL: <https://tools.ietf.org/html/rfc6750> (visited on 10/30/2016) (cit. on p. 52).
- [61] Quan Z. Sheng et al. "Web services composition: A decade's overview." In: *Information Sciences* 280 (2014), pp. 218–238. ISSN: 00200255. DOI: [10.1016/j.ins.2014.04.054](https://doi.org/10.1016/j.ins.2014.04.054) (cit. on pp. 91, 105).