Doctoral Dissertation

Third Generation Web APIs

Bridging the Gap between REST and Linked Data

Markus Lanthaler

Institute of Information Systems and Computer Media
Graz University of Technology, Austria

Supervisor: Dr. techn. Univ.-Doz. Christian Gütl

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, March 2014

Abstract

It is becoming increasingly difficult to cope with the exponentially growing amount of data. Thus, systems are progressively being connected directly to each other to exchange, analyze, and manipulate humongous amounts of data without any human interaction. On the Web, different systems typically communicate via Web services with each other. The first generation of services was based on the flawed Remote Procedure Call (RPC) style, and was difficult to scale and maintain. Consequently, service providers started to align their offerings more closely with the architecture of the World Wide Web. Creating such services, or Web APIs as they are often called, is, however, still more an art than a science. Developers have to struggle with a number of complex design decisions and important technologies are still missing.

This dissertation discusses the main issues of current Web services and related Semantic Web approaches. It reviews the state of the art and presents the results of our research. Our first two contributions, namely SAPS and SEREDASj, were mainly research prototypes that acted as a proof of concept and allowed us to evaluate the main underlying ideas. Eventually, these approaches led to the creation of JSON-LD and Hydra. JSON-LD is a community effort to serialize Linked Data in JSON that resulted in a well-accepted and widely used standard. Hydra, on the other hand, is a lightweight vocabulary defining the necessary concepts to create Web APIs that fully conform to the Web's architecture. This thesis elucidates how JSON-LD and Hydra can be used for a domain-driven design and implementation of Web APIs. To evaluate their practicality and demonstrate that they address the issues of current Web APIs, both technologies have been integrated in a current Web development framework and a completely generic client has been implemented. Finally, this dissertation also provides an overview of some early adopters and describes how they use the technologies in practice.

Acknowledgements

Even though this dissertation was written according the academic convention that prescribes the use of *"we"* instead of *"I"*, the *"we"* is justified in many occasions. Without the help and support of many people, this thesis would not have been possible. First and foremost, I would like to express my deepest gratitude to my supervisor Christian Gütl who helped me in every imaginable way, but at the same time granted me complete freedom in my research. Without his support and trust, it would never have been possible to spend most of this research abroad—literally at the other end of the world. On a related note, I would also like to thank the Curtin University for welcoming me in Australia and Chen Wu for his support during my early days in that fantastic country. I am grateful to Michael Granitzer for reviewing my first paper on Semantic Web Services and Steve Wallis for proofreading a number of papers that I wrote while I was in Australia.

Over the years, I was very fortunate to collaborate with many smart people. I am proud to have worked with the JSON-LD and Hydra Community Groups, especially with Manu Sporny, Gregg Kellogg, Niklas Lindström, Dave Longley, Ruben Verborgh, Ryan J. McDonough, and Thomas Hoppe. Thank you for the countless technical discussions amidst an amicable atmosphere; that is certainly not a given if you have never met in person. I would also like to thank the RDF Working Group for accepting me despite being the youngest participant by far.

Finally, I would like to thank all my new friends in the little land down under for making my stay unforgettable. A very special thanks goes to Meifania Kohn who has done a terrific job in proofreading this dissertation. Thank you buddy!

I dedicate this dissertation to my family. Without their love, continuous support, and encouragement, I would certainly not be where I am today.

Contents

Chapter 1

Introduction

The World Wide Web profoundly changed many aspects of our society. It has shaped our lives so rapidly and so effectively that many of us cannot imagine life without it anymore. The Web brings the world's information to our fingertips and enables frictionless communication across continents in fractions of seconds. Never before in human history has access to information and its dissemination been easier. The initial hurdles to publish content on the Web have long since been eliminated. In fact, the Web has become a global, collaborative information space in which user-generated content dominates. In its short history, the World Wide Web has thus arguably already become an invention as important as Gutenberg's printing press.

For a long time, data has been a scarce resource but the success of the Web resulted in a fundamental shift from information scarcity to surfeit. Not only are individuals producing more content than ever before but also companies and governments are releasing unprecedented amounts of data to the public. While such open data initiatives have been mainly motivated by the desire to increase transparency and accountability, they also create substantial economic value. According to a recent study [1], the more than one million datasets that have already been made public by governments worldwide enable an estimated potential annual value of more than three trillion dollars. The same study argues that consumers will profit most of it, despite the fact that the release of data creates opportunities for whole new businesses, which can be best exemplified by the many companies that were established after access to data from the Global Positioning System (GPS) became freely available. But governments themselves also profit by providing free access to their data. Kenya, one of the poorest countries in the world, was the first African country to launch an open data portal in 2011 and claims [2] that opening up their government's procurement data could save them up to one billion dollar each year, more than they get from donors. Thus, it is also not surprising that in 2013 President Obama signed an executive order "making open and machine readable the new default for government information" [3] that does not have to be kept secret for privacy, confidentiality, or national security reasons.

While open data initiatives resulted in vast amounts of data being published, only a relatively shrinking proportion of data is being created by humans. The vast majority is created by machines or sensors, which includes our digital exhaust, i.e., the digital trails we leave by interacting with various systems. Every day, 2.5 quintillion bytes of new data are being created—so much that 90% of all data available today has been created in the last two years alone [4]. To make use of this data, simply finding it is not enough anymore as there is too much data for a human to review, absorb, and act on. We humans have become the bottleneck. Machines need to integrate data from various sources, analyze it, and distill actionable insights for us—preferably in real time. Data is thus often said to be the new oil; it is valuable but needs to be refined to be usable. To achieve that, more and more systems are connected directly to each other. They exchange, analyze, and manipulate humongous amounts of data without any human interaction. While there are many strategies to connect disparate systems, services are arguably the most flexible option as they allow systems to remain independent and self-contained. Instead of integrating systems directly, they communicate with each other via well-defined interfaces and protocols. Each subsystem in such a distributed architecture represents a service that offers a specific functionality.

Forward-looking companies soon realized the potential of services and began to redesign their systems to form so called service-oriented architectures (SOA). In a public post, Steve Yegge, a former Amazon employee, accidentally shared the now famous mandate that Amazon's CEO Jeff Bezos issued around 2002 [5] to aggressively transform all of Amazon's systems to services. Yegge summarized Bezos' mandate as follows:

- All teams will henceforth expose their data and functionality through service interfaces.
- Teams must communicate with each other through these interfaces.
- There will be no other form of interprocess communication allowed: No direct linking, no direct reads of another team's data store, no shared-memory model, no back doors whatsoever. The only communication allowed is via service interface calls over the network.
- It doesn't matter what technology they use. HTTP, Corba, Pubsub, custom protocols—doesn't matter. Bezos doesn't care.
- All service interfaces, without exception, must be designed from the ground up to be externalizable. That is to say, the team must plan and design to be able to expose the interface to developers in the outside world. No exceptions.
- Anyone who doesn't do this will be fired

Amazon and many other companies that followed such a strategy profited handsomely. Salesforce.com, e.g., generates nearly 50 percent of its revenue through Web services, i.e., services exposed on the World Wide Web; for Expedia, that figure is closer to 90 percent [6].

The first wave of services has been built according the Remote Procedure Call (RPC) style. The aim of RPC-based models is to hide all differences between local and remote computing in order to shield developers from the complexities arising from distributed architectures. But despite the fact that many successful distributed systems were built based on RPC-oriented technologies such as SOAP [7], it is known for quite some time that this approach is fundamentally flawed exactly because it ignores the differences between local and remote computing. The major differences concern the areas of latency, memory access, partial failure and concurrency as described in detail by Waldo et al. [8]. Talking about SOAP-based services specifically, they face additional problems in practice as they abuse HTTP [9] as transport protocol instead of using it as application protocol. This breaks intermediaries that rely on HTTP's application-level semantics. In Internet-scale systems, however, intermediaries for caching, filtering, monitoring, etc. are necessary to ensure good performance, scalability, maintainability, and evolvability.

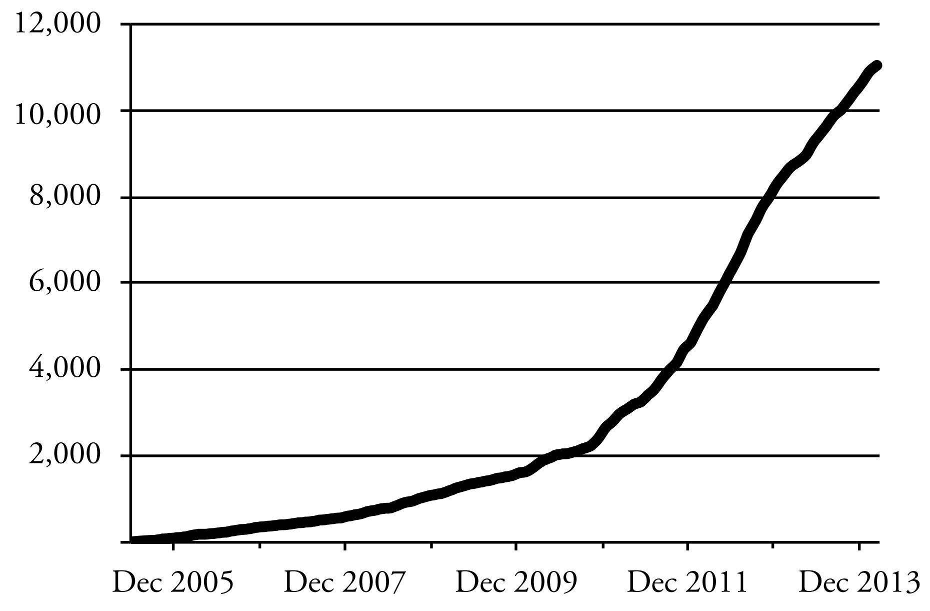


Figure 1. Total number of public Web APIs indexed by ProgrammableWeb

Relatively soon it became apparent that RPC- or SOAP-based services do not scale well and are difficult to maintain. Thus, more and more service publishers started to better align their implementations with the architecture of the World Wide Web [10], the Representation State Transfer architectural style (REST) [11], to benefit from its superior scalability. Fueled by their benefits in terms of scalability and maintainability as well as their simplicity, these second generation Web services, which are also referred to as RESTful services or Web APIs in order to distinguish them from their SOAP-based predecessors, quickly became the prevalent form and more and more companies started to retire their first generation Web Services. This, however, does not mean that the overall number of Web services decreased, quite the contrary. As shown in Figure 1, the advent of RESTful Web APIs resulted in an accelerated growth. ProgrammableWeb, the most well-known Web service directory, announced in 2013 that the number of public services grew over 10,000 [12]. Public services, however, represent only a small subset of all available Web services. There are estimates [13] that private services outnumber public services by as much as 9:1, meaning that the actual total number of services would be closer 100,000. This is an astonishing number considering that most of these Web services are proprietary snowflakes, i.e., similar, yet different enough to be not interoperable. Most of the current Web APIs violate one or more of REST's constraints which yields to several problems in practice. This mostly stems from the fact that clear guidelines and standardized technologies to implement truly RESTful services are still missing.

While it is possible to alleviate some of the problems by, e.g., creating wrapper APIs, i.e., APIs that integrate multiple APIs from a specific vertical into a single interface, standardization is unavoidable in the long term. The Web would never have experienced such an exponential growth if browsers were to be adapted to every single Web site. Similarly, a process in which clients have to be adapted to be usable with different APIs does not scale either. As Steve Vinoski argues in an excellent article [14], platforms, although efficient for their target use case, often inhibit reuse and adaptation by creating highly specialized interfaces, even if they stick to industry standards. He argues that "the more specific a service interface [is], the less likely it is to be reused, serendipitously or otherwise, because the likelihood that an interface will fit what a client application requires shrinks as the interface's specificity increases." This observation undoubtedly also applies to most current Web APIs.

What would be needed to solve these issues are standardized, Web-based technologies for the machine-to-machine communication and processing of structured data whose meaning can be understood by machines. Since we are still far from human-level artificial intelligence, the intended meaning, i.e., the semantics, of the data has to be explicitly described in a machine-processable format. Research on knowledge representation and reasoning is almost as old as the dream of intelligent machines, but in the context of Web, these ideas gained major public attention only after Tim Berners-Lee, the inventor of the World Wide Web, and others published a seminal article [15] entitled the "Semantic Web" in 2001. Since then, the World Wide Web Consortium (W3C) standardized numerous technologies to build an interoperable Semantic Web. Unfortunately, however, for a long time the Semantic Web community derailed into the artificial intelligence domain instead of concentrating on more practical data-oriented applications. Thus, most Semantic Web technologies were adopted very reluctantly (if at all) and the Semantic Web got a reputation of being overly complex and impractical. It is thus also not really surprising that even experts have quite different opinions of the likely progress toward achieving the goals of Berners-Lee's vision of the Semantic Web by the year 2020, as a survey [16] of the Pew Research Center and the Elon University revealed. Around 47% of the invited experts asserted that, “by 2020, the semantic web envisioned by Tim Berners-Lee will not be as fully effective as its creators hoped and average users will not have noticed much of a difference.” In contrast, about 41% agreed with the opposite statement: “By 2020, the semantic web envisioned by Tim Berners-Lee and his allies will have been achieved to a significant degree and have clearly made a difference to average internet users.” The remaining 12% of the experts did not venture a guess.

To refocus the Semantic Web community on the importance of the main principles of the World Wide Web in order to produce more practical solutions, in 2006 Berners-Lee published the so-called Linked Data principles [17], a short list of guidelines to publish and interlink data using basic Semantic Web technologies. This marked an important turning point in the history of the Semantic Web and resulted in the publication of numerous datasets according to these principles. Since REST principles align well with the Linked Data principles it would seem consequent to combine the strengths of both, but in practice they still remain largely separated. Instead of allowing the modification of data via RESTful service interfaces, the vast majority of data published according the Linked Data principles is read-only.

The aim of this dissertation is to bridge the gap between REST and Linked Data in order to support developers in the creation, documentation, and usage of Web APIs. The motivation is to increase developers’ productivity and to improve the quality and reusability of the created Web APIs and the data they expose. Consequently, we analyze, assess, and improve the processes and technologies used to create and access RESTful Web services. The work is divided into four phases: theoretical research and empirical analysis, design of novel solutions, implementation of prototypes, and evaluation. The initial phase of theoretical research and empirical analysis sheds light onto the potentials, the limitations, and the current situation of creating and using RESTful services and Linked Data. Based on these insights, novel approaches to describe and implement semantic RESTful services are designed and evaluated in an iterative approach. To practically evaluate the final proposed solutions, both a prototype integrating them into a popular Web development framework and a completely generic API console capable of interacting with the resulting services is implemented. The final outcome of this thesis is a set of technologies and a reference model for third generation Web APIs, i.e., Web APIs that fully embrace the architectural style of the Web and combine it with the expressive power of the Semantic Web. By addressing various issues of first and second generation Web APIs, this new breed of Web APIs allow the creation of loosely coupled, scalable systems and enable the creation of completely generic clients and tooling. By standardizing the proposed technologies, we hope to help ignite the next stage of Web API growth, similar to how the standardization of the basic Web technologies and the advent of graphical browsers led to an explosive growth of the Web in the early nineties.

1.1 Contributions

The major contributions of this thesis can be summarized as follows:

- Current state-of-the-art research. The review of related work provides a comprehensive overview of the state-of-the-art research and development in the context of RESTful Web APIs as well as semantic services. Furthermore, we present a number of popular domain application protocols and discuss current efforts to add hyperlinks and namespaces to JSON.
- SAPS and SEREDASj. With the creation of SAPS and SEREDASj we present novel approaches to combine proven technologies used in current Web APIs with Semantic Web technologies. SEREDASj, e.g., allows the data exposed by current JSON-based Web APIs to be lifted to RDF, to be manipulated with SPARQL, and to eventually be sent back to the server. These two approaches were mainly research projects and work on them has eventually been discontinued in favor of JSON-LD and Hydra.
- JSON-LD. After having begun the work to improve SEREDASj, we discovered the JSON-LD project and we were among the first to join it. Just as SEREDASj, JSON-LD's goal is to improve JSON-based Web APIs by bridging the gap to Linked Data. We made several crucial contributions to improve and shape the syntax of JSON-LD as well as its processing algorithms and application programming interface. The author of this thesis is co-author and co-editor of both specifications that have been ratified as official Internet standards by the World Wide Web Consortium (W3C). JSON-LD was well accepted and is already being used by

hundreds of millions of people across the globe, most of them without knowing it.

- Hydra. Given that JSON-LD is a data interchange format with very little semantics by itself, we created Hydra, a lightweight vocabulary specifying a number of frequently needed concepts to create and describe hypermedia-driven, RESTful Web APIs. Unlike JSON-LD, which was a collaborative effort from the very beginning, the first versions of Hydra were developed solely by the author of this dissertation. After the overall model stabilized, however, further development was moved to a steadily growing W3C Community Group.
- Prototype implementations and evaluations. As a proof of concept, we integrated JSON-LD and Hydra into a current Web development framework and implemented a completely generic API console. This allowed us to evaluate the complexity and usability of these two technologies in practice—both crucial aspects for their adoption. The complete source code has been released into public domain.
- An alternative, domain-driven approach for the design and development of Web APIs. The combination of JSON-LD, Hydra, and other RDF-based vocabularies enables an alternative, domain-driven approach covering the whole lifecycle of a Web API. By making all the knowledge about a Web API available in a reusable, machine-readable, and semantically-rich form, the approach enables the creation of much smarter clients as possible today.
- Improvement and simplification of Semantic Web standards. The standardization of JSON-LD and the consequent invitation as an expert by the W3C allowed the author of this dissertation to directly participate in the work of the RDF Working Group. He made several contributions to improve the specifications the group was working on. Eventually, the author of this thesis became co-editor of the central Semantic Web specification RDF 1.1 Concepts and Abstract Syntax. He also contributed to the new RDF 1.1 Primer to create an accessible introduction to RDF.

During the work on this doctoral dissertation, we identified a number of minor issues or missing pieces in other standardization efforts. Thus, we provided feedback for different specifications related to the main topics of this dissertation, including the upcoming revision and clarification of the HTTP/1.1 [18] and various specifications related to the "profile" link relation [19]. The latter led to the discovery of the fact that an important piece was forgotten in RFC6906 [19], namely a central registry of profile URIs to decouple clients from servers. To fix that, we requested the Internet Assigned Numbers Authority (IANA) to establish a registry for profile URIs [20]. By registering a profile URI, its ownership moves from the server to a central registry which decouples the client and server. As we will discuss in section 2.2, this is one of the main differences of the Web compared to other distributed system architectures.

1.2 Outline

This chapter provides the motivation behind our research as well as a short introduction to our main contributions. The remainder of this dissertation is structured as follows:

Chapter 2 introduces the reader to a number of basic concepts and technologies necessary for the understanding of this thesis. It includes a short overview of the history and the architecture of the World Wide Web as well as an introduction to the vision and the building blocks of the Semantic Web and Linked Data. Finally, the chapter discusses services on the Web and classifies them into two main categories, namely SOAP-based services and RESTful services.

Chapter 3 distills a number of shortcomings and issues from the current best practices for the creation, documentation, and usage of Web APIs. It looks at the slow adoption of Semantic Web technologies and discusses their (perceived) complexity, their ignorance of fundamental Web principles, and the main challenges developers face due to their underlying open-world assumption. Lastly, the chapter formalizes the research problems addressed by this thesis.

Chapter 4 reviews recent related work and research that have been conducted in the area of interface description languages, data interchange formats, and ontologies for Web services. It also presents a number of successful domain application protocols that are relevant for this thesis and discusses the efforts adding hyperlinks and namespacing support to JSON.

Chapter 5 describes the four main contributions of this thesis, namely SAPS, SEREDASj, JSON-LD, and Hydra. The description of each solution begins with an explanation of its basic concepts and principles and is followed by an illustrative example showing how it might be used in practice. Finally, the integration of each solution in the vision of a Semantic Web is discussed before a number of lessons learned are distilled and evaluated.

Chapter 6 discusses which of the problems identified in Chapter 3 have been addressed by the final two solutions JSON-LD and Hydra. It also evaluates their practicality by their integration into a current Web development framework and the implementation of a completely generic API console. Finally, the chapter provides an overview of early adopters from academia, industry, and related standardization efforts and describes how they leverage the proposed solutions.

Chapter 7 concludes the thesis by briefly revisiting and summarizing the main findings and contributions, identifying limitations of the proposed solutions, and discussing future research directions and complementary topics.

All relevant findings and contributions have already been published in peer-reviewed scientific journals and conference proceedings or have been integrated into ratified Internet standards. Since this thesis is heavily based on our previous publications, we enumerate at the beginning of each chapter the publications it is it based on.

Chapter 2

Basic Concepts and Technologies

Even though the terms Internet and World Wide Web are often used interchangeably in everyday speech, it is technically incorrect. The term *Internet* refers to the global system of interconnected computer networks. It is a network of networks using the Internet protocol suite; commonly known as TCP/IP due to the Transmission Control Protocol (TCP) [21] and the Internet Protocol (IP) [22]—the first two protocols defined back in 1981.

The *World Wide Web*, or colloquially *Web*, on the other hand, is just one of the many applications running on the Internet. Today it is by far the most popular application on the Internet and overtook other applications such as file transfer or newsgroups many years ago. The Web is an information system of interlinked hypertext documents, so called web pages, that Tim Berners-Lee proposed in 1989 to the CERN [23], his employer at that time, to build a more efficient internal information system. Luckily, he soon realized that the system could be used globally across organizations and announced the project to the wider world.

Fast forward a little more than two decades, a world without Internet and World Wide Web has become almost unimaginable. According to statistics from the International Telecommunication Union [24], three quar-

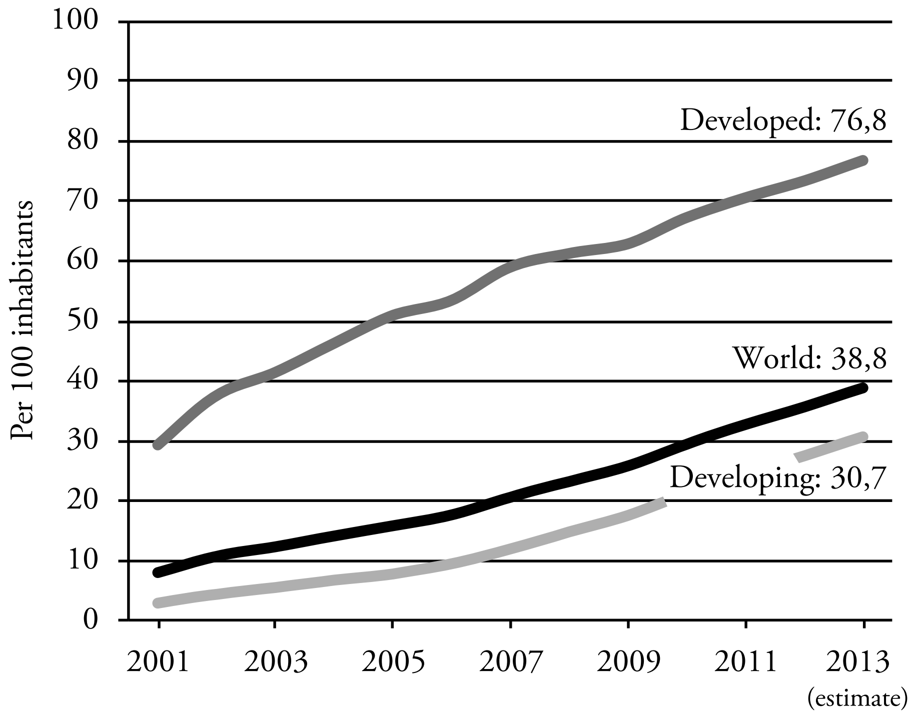


Figure 2. Internet users by development level (adapted from [24])

ters of the population of the developed world are active Internet users. In total, forty percent of the world's population is online. Considering its sheer size, its often uncoordinated development, and its heterogeneity, the Internet and the World Wide Web are the most complex systems ever built by humankind.

In the following sections, which are based on previous work published in [25] and [26]–[28], we will have a look at the architecture that enabled the Web's exponential growth and some of the main technologies to build websites and Web APIs. We will provide an overview of the vision of a "Semantic Web". In the context of this thesis, the underlying Internet technology is assumed as a given infrastructure and thus not being discussed in detail.

2.1 The Architecture of the World Wide Web

After getting the approval for his project proposal [23], Tim Berners-Lee started the development of what became the World Wide Web in October 1990 [29]. By Christmas the same year, Berners-Lee had not only specified the three main technologies which still form the foundation of today's Web, namely the Hypertext Transfer Protocol (HTTP), Universal Document Identifiers (UDIs), which was later renamed to Uniform Resource Identifiers (URIs), and the Hypertext Markup Language (HTML), but also first running prototypes of a browser and a server.

An important milestone in the history of the Web was 1993 when the CERN released the World Wide Web's technology into public domain [30]. The same year, the University of Minnesota announced [31] that it would begin to charge licensing fees for its implementation of the Gopher protocol [32] (the University of Minnesota is the inventor of Gopher) which caused many users to stop using Gopher and switch to the World Wide Web instead. Today, Gopher is often regarded as the predecessor of the World Wide Web.

The World Wide Web was conceived as a client-server system. Clients access hypertext documents which are identified by *Universal Resource Identifiers* (URI) on servers across the Internet via the *Hypertext Transfer Protocol* (HTTP). The documents themselves are expressed in the *Hypertext Markup Language* (HTML), a simple, text based markup language inspired by SGML [33] (more concretely, SGMLguid, a CERN-internal, SGML-based documentation format). So, historically, the Web can be described as a giant, globally distributed collection of hypertext documents. Links and the resulting networking effects played a fundamental role for the success of the Web. Universal Resource Identifiers provide a mechanism to enrich documents with references to other relevant documents.

The key differentiator to other hypertext systems available at the time lies in the decision to make links unidirectional instead of requiring them to be bidirectional. Practically, this means that it is impossible to prevent broken links when documents become unavailable. While link rot is certainly an undesired consequence of this decision, the advantages clearly outweigh this shortcoming. The decision is arguably one of the main reasons for the Web's superior scalability. As unidirectional links eliminate the otherwise necessary referential integrity checks, it is possible to drastically simplify the implementation of clients and servers. Furthermore, unidirectional linkage enables the decentralized, uncoordinated creation of documents. This decentralism was the main reason why the Web quickly overshadowed all previous hypertext systems and allowed it to scale in an unprecedented manner.

For a long time, the Web's architecture and its technologies were not properly standardized. The documentation merely consisted of a set of informal web pages [34], [35], draft specifications [36], and the source code for clients and servers published by the CERN. As those documents have not been kept in sync with the deployed implementations, it became harder and harder to create interoperable systems. Consequently, the pressure from industry asking for standardization of the core technologies grew and working groups writing stable specifications were established.

In 1992 Berners-Lee, Groff, and Cailliau published a paper [37] discussing "the requirements on a universal naming syntax which can be used to refer to *documents* [emphasis added]". Furthermore, concrete recommendations for a generic syntax for *Universal Document Identifiers* were presented. Somewhere during those initial standardization efforts in the early nineties a subtle yet interesting shift in the used terminology can be observed. RFC 1630 [38], which in 1994 defined for the first time the *Uniform Resource Identifier* syntax in a stable document, is primarily concerned with "objects" instead of documents. It defines "the syntax used by the World-Wide Web initiative to encode the names and addresses of *objects* [emphasis added] on the Internet". RFC 1738 [39], which was published the same year, officially standardized the syntax and replaced the term "object" with "resource". It then took four years till RFC 2396 [40] finally defined the term "resource":

> A resource can be anything that has identity. Familiar examples include an electronic document, an image, a service (e.g., "today's weather report for Los Angeles"), and a collection of other resources. Not all resources are network "retrievable"; e.g., human beings, corporations, and bound books in a library can also be considered resources. The resource is the conceptual mapping to an entity or set of entities, not necessarily the entity which corresponds to that mapping at any particular instance in time. Thus, a resource can remain constant even when its content—the entities to which it currently corresponds—changes over time, provided that the conceptual mapping is not changed in the process.

While this appears to be the first trace of what later would become Linked Data and the Semantic Web vision (which we will describe later in this chapter), the first seeds can in fact already be found in

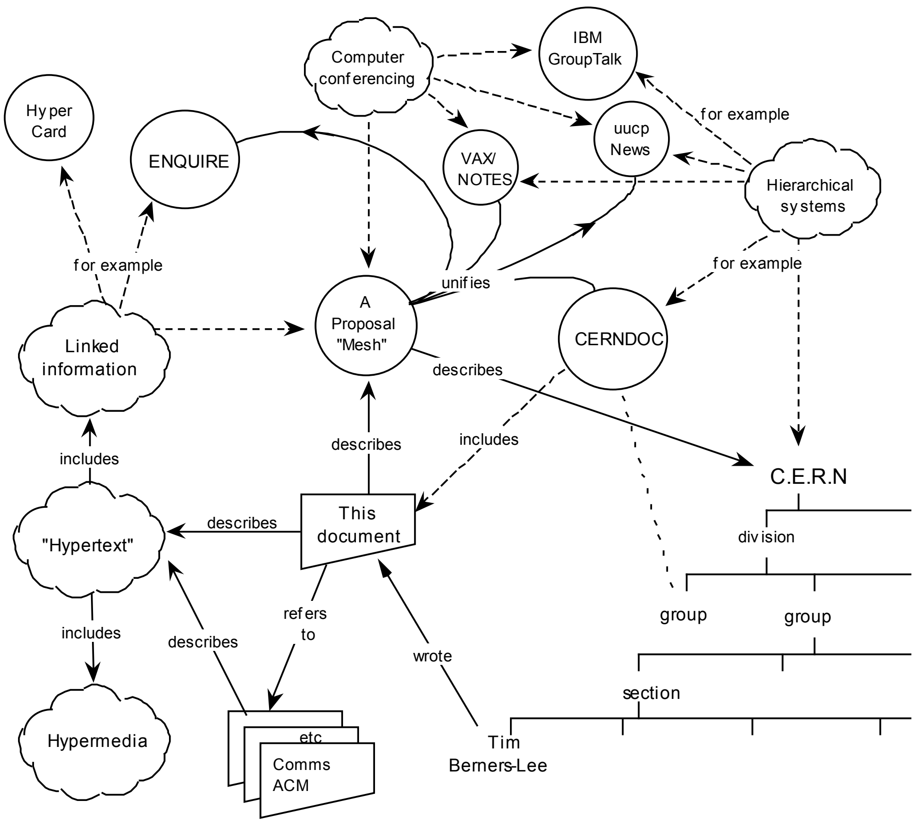


Figure 3. Information Management: A Proposal [23]

Berners-Lee's initial proposal [23] to CERN's management to build the World Wide Web. The graph shown in the document (Figure 3) not only includes documents but also real-world entities such as organizations, divisions, and even persons.

In 1995, HTML 2.0 [41] was the first version of the Hypertext Markup Language to be officially standardized and a year later, HTTP/1.0 [42] was published non-normatively as it was expected to be replaced soon by a standards track document fixing some of HTTP/1.0's issues. This happened in 1997 with the publication of HTTP/1.1 [43], the first normative specification of the Hypertext Transfer Protocol.

The snippet in Listing 1 shows the Web's three main technologies in action. A client requests the resource identified with the URI `http://example.com/doc` over HTTP/1.1 for which the server returns an HTML document linking to `http://example.org/doc2`.

In the mid-nineties, Roy T. Fielding, co-author of both the URI and the HTTP specification, started working on "an architectural model for how

```
--> Client Request
GET /doc HTTP/1.1
Host: example.com

<-- Server Response
HTTP/1.1 200 OK
Content-Type: text/html

<html>
  <head>
    <title>An Example Page</title>
  </head>
  <body>
    <p>Link to <a href="http://example.org/doc2">document 2</a>.</p>
  </body>
</html>
```

Listing 1. An HTTP request returning an HTML document

the Web should work, such that it could serve as the guiding framework for the Web protocol standards" [11]. The outcome of his work was an architectural style that he called *Representational State Transfer* (REST) [11]. According to him, REST captures all aspects of a distributed hypermedia system that are considered central to the behavioral and performance requirements of the Web. Since REST has been used to guide the design and development of the architecture of the modern Web [10] we will discuss it in more detail in the next section.

2.1.1 The Representational State Transfer Architectural Style

REST [11] is an architectural style that specifies constraints to improve performance, scalability, reliability, and resource abstraction within distributed hypermedia systems. It ignores implementation details and protocol syntaxes in order to focus on the roles of the various system components, their interaction with other components, and the interpretation of the exchanged data. The meticulously chosen design trade-offs allow the creation of extensible, maintainable, evolvable, and loosely-coupled distributed systems at Internet-scale. The fact that the Web—the largest and most successful distributed system ever built—is based on REST principles should be evidence enough of its superior characteristics.

REST is based on a traditional *client-server* architecture in which the server offers a number of services which a client can invoke by sending requests to the server. The motivation for such an architecture is the clear separation of concerns which simplifies the implementation of the two components which often has the positive side-effect of improving the scalability of the server and enabling the independent evolution of the two components (as long as their interfaces do not change). REST adds an important constraint to this architectural style, namely that the server is *stateless*, i.e., it does not manage any session state. Consequently, each request from a client to the server must contain all the information necessary for the server to understand the request. In other words, a client cannot take advantage of any stored context on the server. The server of course knows about the state of its resources but does not keep track of individual client sessions; all session state is kept entirely on the client. This is an important aspect which facilitates tasks like monitoring and logging due to the increased visibility of the interactions. It also improves reliability because the recovery from partial failures [8] is much simpler if all the necessary state information is contained in each request. In addition, scalability is improved because not having to store state between requests allows the server to quickly free resources and further simplifies implementation because the server does not have to manage resource usage across requests. It is the stateless server constraint which enables the for many applications crucial load balancing. The downside of REST's statelessness is a decreased network performance due to repetitive data since all the state information has to be transferred in every request instead of keeping it on the server between requests. It also reduces the server's control over a consistent application behavior since the application is split between the server and multiple clients with potentially different capabilities.

To mitigate the overhead caused by the statelessness of RESTful systems, support for caching has been added. The *cache* constraint requires that the data within responses is implicitly or explicitly labeled as cacheable or non-cacheable. This reduces the number of requests or results in much smaller responses that simply indicate that the data has not been changed since the last request. This has positive effects on the efficiency and scalability of RESTful systems and improves the user-perceived performance by reducing latency. The downside is that the system's reliability may be decreased due to potentially stale information.

REST's emphasis on a *uniform interface* between the system components is the central feature which distinguishes it from other network-based styles. It simplifies the overall system architecture and improves the visibility of component interactions. By decoupling implementations from the services they provide, independent evolvability of the various components is improved at the cost of degraded efficiency compared to highly specialized interfaces. REST defines four interface constraints to ensure a uniform interface: 1) identification of resources, 2) manipulation of resources through representations, 3) self-descriptive messages, and 4) hypermedia as the engine of application state.

REST is a resource-oriented architecture in the sense that the key abstraction of information in REST is a resource. Any concept can be thought of as a resource. Fielding defines a resource *R* as a "temporally varying membership function $M_R(t)$, which for time *t* maps to a set of entities, or values, which are equivalent." [11] REST's *identification of resources* constraint requires that resources are identifiable so that they can be accessed and manipulated via generic interfaces. On the Web, resources are identified by IRIs [44]. Since a resource may represent concepts which cannot be serialized into a byte stream (e.g., persons or a feeling), resources are not manipulated directly. Instead, REST is built on the concept of *manipulation of resources through representations;* i.e., an additional layer of indirection in the form of resource representations is introduced. A representation is a sequence of bytes plus some metadata. Media types standardize the data format of resource representations on the Web. Given that the communication between components is stateless and that the data format of resource representations is standardized, REST enforces *self-descriptive messages* that can be processed by intermediaries without out-of-band knowledge. The last missing piece to complete REST's uniform interface is the *hypermedia as the engine of application state* constraint (HATEOAS). It refers to the use of hyperlinks in resource representations as a way of navigating the state machine of an application. Even though it is the hypermedia constraint which allows systems to be dynamically composed and loosely coupled, it is one of the least understood constraints and thus seldom implemented correctly.

A lot of systems, regardless of claiming to be RESTful or not, rely heavily on implicit state control-flow, which is characteristic for the Remote Procedure Call style. The allowed messages and how they are interpreted depends on previously exchanged messages and thus in which implicit state the system is in. Third parties or intermediaries trying to interpret the conversation need the full state transition table and the initial state to understand the communication—something that is rarely available or not practical. This also makes it difficult or virtually impossible to recover from partial failures in large distributed systems.

To solve these issues and assure evolvability, the use of hypermedia is a core tenet of the REST architectural style. According to Fielding [45], "a REST API should be entered with no prior knowledge beyond the initial URI (bookmark) and set of standardized media types. [...] From that point on, all application state transitions must be driven by client selection of server-provided choices that are present in the received representations or implied by the user's manipulation of those representations." The Web leverages this type of interaction and state-control flow where very little is known a priori to allow the decentralization. At least humans are able to quickly adapt to new control flows that are communicated at runtime, e.g. a change in the order sequence or a new login page to access a service.

Parastatidis et al. [46] define the set of legal interactions necessary to achieve a specific, application-dependent goal as the *domain application protocol* of a service. The protocol defines the interaction rules between the different participants. Consequently, the application state is a snapshot of the system at an instant in time. This coincides with Fielding's definition [11] of application state which defines it as the "pending requests, the topology of connected components (some of which may be filtering buffered data), the active requests on those connectors, the data flow of representations in response to those requests, and the processing of those representations as they are received by the user agent." Accordingly, the overall system state consists of the application state and the server state. By using the notion of a domain application protocol, the phrase "hypermedia as the engine of application state" can now be explained as the use of hypermedia controls to advertise valid state transitions at runtime instead of agreeing on static contracts at design time. Changes in the domain application protocol can thus be dynamically communicated to clients. This brings some of the human Web's adaptivity to the Web of machines and allows the building of loosely coupled and evolvable systems. Rather than requiring an understanding of a specific URI structure, clients only need to understand the semantics or business context in which a link appears [46]. Unfortunately, however, current Web APIs rarely exhibit such features and thus it is almost impossible to achieve a comparable level of adaptivity on the Web of machines as we will see in section 3.

Annoyed by the fact that a lot of services claim to be RESTful regardless of violating the hypermedia constraint, Fielding made it very clear that hypermedia is a fundamental requirement for a RESTful architecture [45]. Given that the term REST is nonetheless frequently misused, there exist efforts in the community to establish alternative terms, such as *Hypermedia API*, to denote services respecting it.

While the hypermedia constraint is often violated, the *layered system* constraint is almost always implemented correctly and helps to reduce the coupling between components. The constraint requires that RESTful systems are composed by hierarchical layers in which components on a specific layer only provide services to components on the layer above and only use services provided by components on the layer below. This limits the knowledge of components to a single layer and it could thus be said that it lowers the upper bound of the overall system complexity. Furthermore, it allows the introduction of important intermediaries such as load balancers, shared caches, firewalls, gateways, or proxies at various points to make the system more adaptable to changing requirements without having to change the interfaces. Obviously, each additional layer increases the processing overhead and therefore latency, which results in lower user-perceived performance. That can, however, be partly compensated by adding caches.

Finally, the Representational State Transfer architectural style has a *code-on-demand* constraint which allows client functionality to be extended by code that is loaded dynamically at runtime. The constraint's main benefit is the improved extensibility of systems. It is best illustrated by current Web applications which depend heavily on dynamically loaded JavaScript code to implement functionality that is not generally available in Web browsers. It should, however, not be forgotten that loading code on demand drastically reduces visibility and may open the door to security vulnerabilities. In REST, code-on-demand is thus an optional constraint.

2.2 Contracts on the Web

In any distributed system there has to be an agreement, or more formally, a contract prescribing how the various components of the system interact; otherwise, communication is impossible. These contracts usually stipulate the data model along with its processing model and encodings, i.e., the serialization formats, the interaction model consisting of system interfaces and coordination protocols, and sometimes various policy assertions. The data encodings or formats along with their processing models enable the creation and interpretation of messages that are exchanged between the various components in order to invoke certain operations. The system interfaces and coordination protocols define the mechanisms and the order in which messages have to be exchanged to result in the desired behavior. Finally, policies may describe non-functional aspects such as service-level agreements (SLAs), pricing, security requirements, etc.

In the traditional Remote Procedure Call (RPC) model, where all differences between local and distributed computing are hidden, typically Interface Description Languages (IDL) are used to define the application-specific details on top of a standardized communication protocol. This allows automatic code generation for both the client and the server side but, in most cases, also leads to the undesirable effect of leaking implementation details from the server, who owns the contract, to the client. Given that the client and the server are tightly coupled, such systems typically rely heavily on implicit state control-flow. The allowed messages and how they have to be interpreted depends on what messages have been exchanged before and thus in which implicit state the system is. Third parties or intermediaries trying to interpret the conversation need the full state transition table and the initial state to understand the communication. This implies that states and transitions between them have to identifiable, which in turn suggests the need for (complex) orchestration technologies.

The architecture of the Web differs fundamentally from these traditional models. On the Web, contracts are based on media types and protocols. Applications can thus be built by composing various well-defined building blocks. Media types define the data and processing models as well as the serialization formats. Protocols describe interaction models that extend the capabilities of (the more or less generic) media types into the realm of specific application domains; mostly by defining specific link relations. An example illustrating this nicely is the Atom Publishing Protocol [47], which, by defining a number of link relations, extends the otherwise read-only Atom Syndication format [48] with a protocol that allows the addition or manipulation of existing feed entries.

The main difference of the Web compared to other distributed system architectures is that the contracts are centrally owned instead of being owned by the server. This allows the independent evolution of clients and servers as both are coupled to these central contracts instead of being coupled to each other. Instead of relying on upfront agreement of all aspects of interaction, parts of the contract can be communicated or negotiated at runtime. Furthermore, instead of relying on implicit state control-flows as described above, all communication is stateless, meaning that each request from the client to the server must contain all the information necessary for the server to understand the request. A client cannot take advantage of any stored context on the server as the server does not keep track of individual client sessions. The session state is kept entirely on the client. This transfer of state information paired with centrally owned contracts that can be communicated or negotiated at runtime is the essence of what Fielding describes as the Representational State Transfer (REST) architectural style [11].

The challenge in designing RESTful systems is to select the most appropriate media type(s) as the core of the application-specific contract. Sometimes this requires creating new, specialized media types. Therefore, designers of Web APIs have to decide whether to create their own specialized media type, which reduces interoperability; to use a generic one such as XML [49] or JSON [50], which, with a probability bordering on certainty, requires out-of-band contracts and thus introduces coupling; or to create a specialized media type on top of an existing, generic one. Unfortunately, even the specialization of an existing media type is not as straightforward as it might seem at first sight as we will see in section 3.1. It is also worth noting that media type specifications or media type specializations are not machine-readable but just described in natural language. Software has thus to be manually adapted if new media type (specializations) are to be supported.

2.2.1 An Alternative Approach

Both xCard [51] and xCal [52], e.g., are XML-based serializations and, as such, use XML's preferred solution to unambiguously bind elements and attributes to the semantics of a specific vocabulary, namely XML Namespaces [53]. The idea behind XML Namespaces is simple: instead of using arbitrary strings as names for elements and attributes, a vocabulary URI is defined which acts as a prefix for all names that are part of the said vocabulary. This has the advantage that elements and attributes from multiple XML markup vocabularies can be used within a single document without risking that names clash. The fact that URIs are used as identifiers allows both a centralized and a decentralized creation and management of XML namespaces; in fact, the IANA maintains a registry specifically for XML namespaces [54].

Thus, the question arises why both xCard and xCal have a dedicated media type if the semantics are already signaled by a dedicated XML namespace. The reason is simple. If HTTP messages are not typed using a media type, a processor has to look into the content of the message in order to decide how to process it. This is not problematic per se, but the real problem lies in the fact that most processors (including browsers) have no mechanisms to leverage these extension points. Instead of passing the data to the most appropriate application, they simply fall back to the basic behavior, which in the case of XML in the browser is to simply display the XML tree. Another, perhaps bigger, problem is the fact that content negotiation is based on media types which makes it impossible for a client to express its preferences if no dedicated media type exits. This problem has been known for quite some time.

Inspired by HTML's profile attribute [55] Toby A. Inkster started an effort [56] to register an optional *profile* parameter for XML's and JSON's media types to address this issue in 2009. Similar to HTML's profile attribute, the profile parameter was intended to signal that a message conforms to some additional constraints or conventions on top of the constraints and semantics imposed by the media type or to convey some additional semantics. The value of the profile parameter in Inkster's proposal had to be a single absolute URI. If multiple profiles are applicable to the content, a server should choose "the most useful" but "pay attention to any of the profiles if found in the *Accept* header during content negotiation" [56]. Unfortunately, Inkster's Internet Draft was not standardized but expired and most Web APIs continued to either use the generic media type or to mint their own specialized type.

In 2012, Erik Wilde started a new initiative to standardize a similar mechanism. Instead of trying to change XML's and JSON's media type registrations, he proposed [19] to standardize the link relation *profile* to signal additional semantics associated with a representation using an HTTP Link header [57]. While this enables servers to advertise profiles in their responses, it leaves the content negotiation problem unsolved. Wilde addressed this shortcoming in a later revision of his draft by recommending that newly defined media types should define a *profile* media type parameter if appropriate. This allows clients to signal their capabilities and preferences in the content negotiation process allowing the server to return the best matching representation. Another notable difference to Inkster's proposal is that Wilde removed the restriction to a single profile URI, meaning that multiple profiles can be easily combined.

In light of these advances, initiatives such as the effort [58] to standardize dedicated media types for JSON-based versions of vCard and iCalendar should be challenged—especially considering that most required parts already exist. The work to create a shared vocabulary has already been started a couple of years ago at the W3C [59] and JSON-LD, which is presented in detail in section 5.3, provides a way to serialize such data in a JSON-based syntax. It also features a *profile* media type parameter to signal the additional semantics and conventions at the HTTP layer. The only missing piece is the definition of a profile to specify which field names are used and how the data is structured when serialized. This is necessary as JSON-only clients depend on the structure and not directly on the semantics of the serialization. Since JSON-LD represents graphs, most of the time, there exist multiple ways to serialize the same data.

There are multiple advantages that such profile-based approach offers. First of all, the need for micro-types such as xCard would disappear. It is true that the information is basically just shifted to the profile parameter but the fact that profiles can be easily combined means that the overall need for dedicated types or profiles is reduced. This brings us to another benefit: due to their simple composability, the scope of profiles can be reduced which in turn simplifies their standardization. It is this composability which allows the separation of concerns that is often missing in media types. Leveraging profiles, generic media types defining a serialization format can be combined with the concrete semantics of a profile. Networking effects will ensure that a few well-known and widely adopted profiles will emerge. At the same time it becomes easier to bootstrap new profiles because, unlike newly established media types, they do not suffer under a cold start problem. To alleviate the risk of introducing tight coupling through the backdoor by the usage of profiles it is important that they are centrally owned, just as media types. In practice, this means that a central registry of standardized profiles is required. We thus requested the Internet Assigned Numbers Authority (IANA) to establish such a registry [20].

2.3 Linked Data and the Semantic Web

The early standardization efforts of the Web made it clear that it is more than a hypertext document system. The separation of documents into resources and representations thereof paved the way to integrate real-world entities such as persons or even imaginary or abstract concepts such as companies into the Web at an architectural level. No longer was the Web limited to simple documents. While Tim Berners-Lee's original proposal [23] already hinted such an architecture (and early standardization efforts certainly took it into consideration), he felt urged to express his vision more explicitly to a wider public at the First International World Wide Web Conference in 1994. He argued that the Web has become an "exciting world" for users but that it contains very little machine-readable information. "The meaning of the documents is clear [only] to those with a grasp of (normally) English, and the significance of the links is only evident from the context around the anchor [but to a computer] the web is a flat, boring world devoid of meaning." [60] He identified two things which would be necessary to add machine-readable semantics to the Web, namely allowing documents to contain machine-readable information and allowing links with explicit relationship values. Unsurprisingly, the World Wide Web Consortium (W3C), which Berners-Lee founded later that year to coordinate the standardization of the Web, put a focus on standardizing Semantic Web technologies. After various related efforts, this resulted in the standardization of the Resource Description Framework (RDF) in 1999 [61].

For a long time, RDF/XML was the only standardized serialization format for RDF but it was widely disliked even by XML enthusiasts (XML was at the peak of its popularity at that time). RDX/XML is neither optimized for humans nor machines but, most importantly, standard XML tools are almost useless when working with RDF/XML. This and the fact that the Semantic Web community derailed into the artificial intelligence domain instead of concentrating on more practical data-oriented applications resulted in the languishing adoption of the technology. In 2006, however, Shadbolt, Hall, and Berners-Lee published an article [62] admitting that the simple idea behind the Semantic Web vision still remained largely unrealized. Nevertheless, standardization work continued and led to a more or less complete stack of Semantic Web technologies which we will discuss in the next section. The few early Semantic Web projects lacked viral uptake and only very few used dereferenceable URIs which would have allowed the data to be browsed in a similar fashion as documents can be browsed on the Web. This motivated Berners-Lee to formulate the famous Linked Data principles [17] which can be classified as a turning point in the history of the Semantic Web. We will discuss them later in this section.

2.3.1 The Semantic Web Technology Stack

The *Semantic Web* is an extension of the traditional Web with the aim to offer information not only in the form of natural language documents but also as machine-readable data. The *Resource Description Framework* (RDF) [63] builds the foundation of the Semantic Web technology stack. It defines a simple, triple-based data model in which each statement consists of a subject, a predicate, and an object as illustrated in Figure 4. Multiple triples build a graph, and multiple graphs form a dataset.

While IRIs can be used in every element of an RDF triple, literals, i.e., basic values such as strings or numbers which are typed and optionally language-tagged, can only be used in the object position. Blank nodes, which are special local identifiers whose scope is limited to a single document or data store, can only be used in the subject and the object position; not as predicates. Despite this simple data model, RDF has the bad reputation of being overly complex. In large parts, this stems from the

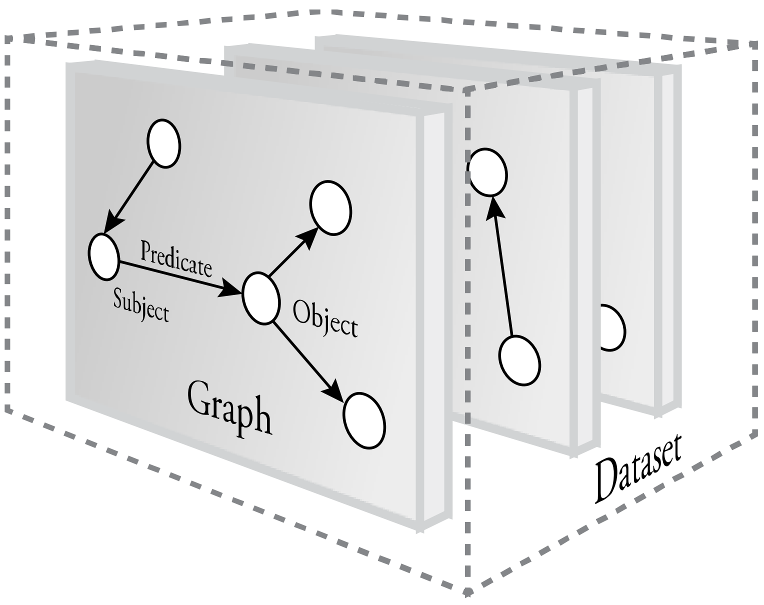


Figure 4. The RDF data model

fact that RDF is often conflated with its first, and for a long time only, serialization format RDF/XML [64]. In fact, RDF/XML is now generally believed to be one of the main reasons for the hesitant adoption of Semantic Web technologies in general. Since critics often complain that "RDF is complex artificial intelligence technology", it is also worth noting that RDF itself does not specify a mechanism for reasoning. This is left to higher layers in the stack.

In RDF, every concept is identified with an IRI, an Internationalized Resource Identifier. Since IRIs are global identifiers, two different appearances of an IRI denote the same concept. The owner of IRI [10] defines what concept the IRI denotes. This, again, can be described in RDF by reusing other, already defined concepts. A set of such concepts targeting a specific use case or application domain is typically called a vocabulary or, more formally, an ontology (we will use the two terms interchangeably throughout this thesis). The W3C standardized the two vocabularies RDF Schema (RDFS) and the Web Ontology Language (OWL) to describe new vocabularies in an interoperable way.

RDF Schema [65] defines concepts to describe classes (and class hierarchies), data types, or properties similar to object-oriented programming languages. Furthermore, it defines concepts to express sets and lists. While this may look familiar to developers used to object-oriented pro-

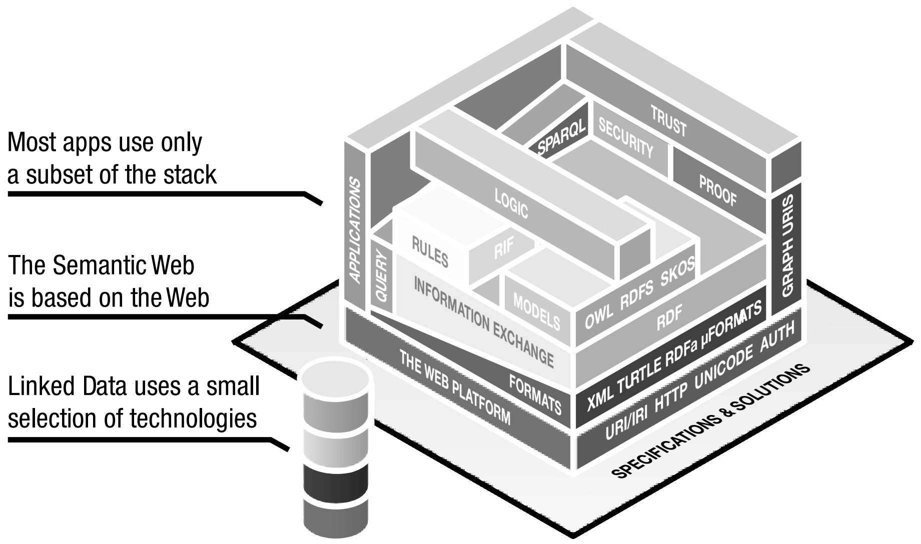

Most apps use only a subset of the stack
The Semantic Web is based on the Web
Linked Data uses a small selection of technologies
APPLICATIONS
TRUST
SPARQL
SECURITY
PROOF
LOGIC
GRAPH URIS
QUERY
RULES
RIF
INFORMATION EXCHANGE
MODELS
OWL RDFS SKOS
RDF
XML TURTLE RDFa µFORMATS
URI/IRI HTTP UNICODE AUTH
FORMATS
THE WEB PLATFORM
SPECIFICATIONS & SOLUTIONS

Figure 5. The Semantic Web technology stack (adapted from [247])

gramming languages, the devil lies in the details. Unlike programming languages, resources can be, e.g., instances and classes at the same time, classes are not disjoint (and there is no way to express disjointness explicitly), and properties can be applied to instances of any class. We will discuss these "issues" in more detail in section 3.4.

Unlike RDFS, the *Web Ontology Language* [66] allows, e.g., to create disjoint classes or unions of classes. Simply speaking, it could be classified as an extension of RDFS adding many concepts, which makes it a far more expressive modeling language (even though technically only some of OWL's profiles are extensions of RDFS). Since both RDFS and OWL play only a marginal role in the context of this thesis, we will not discuss them or their differences in detail but refer the interested reader to the respective specifications [65], [66].

In most cases, both RDFS and OWL are not used to validate data but to infer new knowledge. The properties associated to a specific entity can, e.g., be used to infer the classes it is an instance of. Historically, these inference rules have only been described in natural language in the vocabulary's specification. Thus, reasoners had to be manually adapted to support new vocabularies. RIF [67], the *Rule Interchange Format*, solves this issue by making (inference) rules machine-readable. The RIF Working Group, e.g., published a W3C WG Note showing how OWL 2 RL can be implemented using RIF [68].

Instead of trying to define a universal rule language, RIF acknowledges the fact that existing rule systems have widely varying syntaxes and semantics. In order to achieve interoperability across systems, it defines a number of "dialects", i.e., a set of languages with well-defined syntaxes and semantics. Each system translates its own language(s) to and from a RIF dialect which allows the exchange of rules across systems—provided that the systems find a dialect they both support. The intermediate representation of rules is specified in the form of an XML-based format, thus the term "format" in RIF's name. At the time of this writing, there exists only a Working Group Note describing the mapping of RIF XML documents to RDF graphs [69].

The last piece of the currently standardized Semantic Web technology stack is SPARQL. As the name suggests, SPARQL, which is a recursive acronym for SPARQL Protocol and RDF Query Language [70], not only specifies a language to query and manipulate RDF data [71]–[73] but also a protocol to invoke such queries over HTTP [74]–[76] and a number of result formats (XML, JSON, CSV, and TSV [77]–[79]), however no RDF-based formats.

2.3.1 Linked Data

All the technologies that form the Semantic Web technology stack have in common that they do not require IRIs to be dereferenceable. Instead, just as RDF itself, they treat them as opaque identifiers. Unlike the traditional Web, the early Semantic Web could not be browsed which means that, strictly speaking, it was not an extension of the Web but a separate ecosystem. In an effort to change that, Tim Berners-Lee postulated the following Linked Data principles in 2006 [17]:

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) [*sic*]
4) Include links to other URIs. so [*sic*] that they can discover more things.

These four simple principles represent an important turning point in the history of the Semantic Web. Not only did they rebrand the vision of a

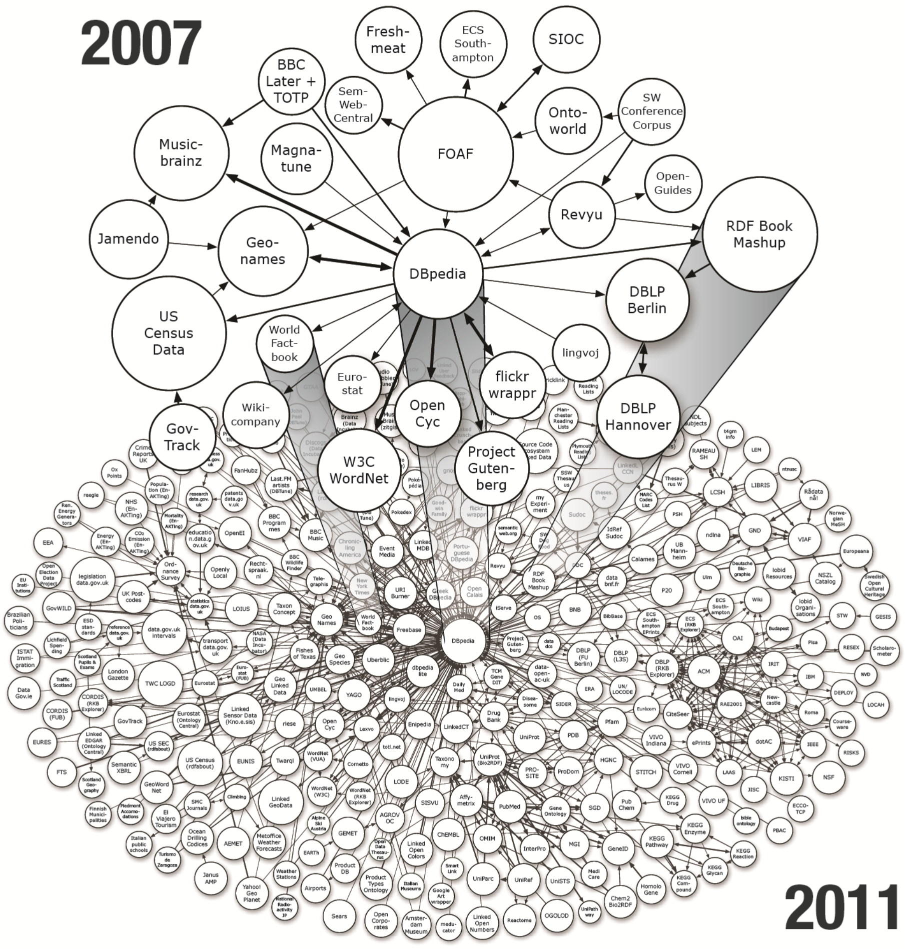


Figure 6. The growth of the Linked Open Data Cloud, 2007–2011 (source: [80])

Semantic Web with a much more concrete and graspable description of the basic principles underlying it but also refocused its applications on more practice-relevant problems, namely the publication and consumption of vast amounts of structured data. As famously illustrated by the well-known Linked Open Data cloud diagram [80] shown in Figure 6, the amount of Linked Data, and thus the Semantic Web in general, has managed to grow significantly over the last couple of years. This is not to say that the Semantic Web is now widely adopted or that there are no major pain points left hindering its adoption. Adoption is still several orders of magnitudes lower compared to the Web and, as soon as one wishes to start publishing Linked Data, several fundamental questions arise for which satisfactory answers are still missing. The most heavily debated question is about the proper use of IRIs for Linked Data.

The shift in the architecture of the World Wide Web [10] from a distributed hypertext system to a resource-oriented architecture in which resources are manipulated through representations did not require Web developers to adapt to. On the contrary, the conceptual model underlying the Web was changed to reflect the way people used the Web since its inception. However, the problem surfaces when IRIs are used to identify both a representation, i.e., the bytes on the wire, and an abstract resource such as a person. Unlike humans, logic reasoners and other inference technologies cannot disambiguate these two concepts and will consequently produce wrong conclusions. A simple example illustrating the problem is an IRI that is both used to describe a person and metadata such as licensing information about the document (representation) it returns when it is dereferenced. If such data is now integrated with data from other sources by declaring IRIs use to talk about the same person as being equal, invalid conclusions such as contradicting licensing terms may be drawn.

This problem has been discussed for over a decade and is commonly known as the httpRange-14 issue [81], the identifier assigned by the W3C's Technical Architecture Group (TAG) in its issue tracker. The resolution of that issue was an advice to the community [82] to either use fragment identifiers or HTTP `303 See Other` redirects to signal that an IRI identifies an abstract resource (formally known as a non-information resource), such as a person, instead of the returned representation, i.e., the document describing the person. While the issue has been closed, consensus that the proposed solution is practical is still uncertain and alternative solutions are proposed on a regular basis. The most promising proposals at the time of this writing involve "punning" to use the same IRI to mean different things. They use the context in which an IRI is used to determine whether the representation or the abstract concept is meant. A property specifying a person's first name, e.g., would be mapped to the abstract concept (the person) whereas a property specifying the license would be mapped to the representation. When data is integrated these separations need to be preserved meaning that it is necessary to have mechanisms which allow the equality of representations to be defined separately from the equality on the abstract concepts they describe. Tennison not only wrote an excellent blog post [83] explaining this in more detail but also published a first draft [84] of what might become a W3C Recommendation as part of her work at the W3C TAG.

Summarized, Linked Data requires Web developers not only to identify things and concepts by an IRI but also to distinguish between information resources such as documents and non-information resources such as persons in order to choose the right form of IRI (fragment identifier) or HTTP behavior when dereferenced (redirect). In this context it is interesting to note that recent commercial efforts such as Schema.org [85] or Facebook's Open Graph Protocol [86] largely stay silent on these issues and instead try to disambiguate the data computationally. Since humongous amounts of data that follows their advice are being published, they clearly influence the direction of future solutions.

2.4 Services on the Web

Many pages on Web are built by segmenting and flattening structured data from databases to HTML documents. To process the data published in such a way, brittle approaches such as screen scraping have to be used to at least partially reconstruct the raw data in order to make it machine-processable. The aim of the Semantic Web is to eliminate this limitation by creating a *Web of Data* that can be directly processed by machines. In practice, however, that goal is still rarely approached by using Semantic Web technologies. Instead, the majority of structured data is being published in the form of XML [49] or JSON [50] documents.

Commonly speaking, such offerings are termed as Web services but given that also human-targeting, HTML-based websites are, at least to a certain degree, machine-processable, that term is somewhat misleading as websites could be classified as Web services as well. Similarly, a Web service could be referred to as a "website for machines". Thus, for the scope of this thesis, we define the term Web service as *a set of HTTP-based interfaces to support interoperable machine-to-machine interaction by the exchange of structured data.* The aim of the machine-to-machine interaction is to drive business processes in order to serve particular, application-dependent goals. Since Web services put the emphasis on "machine-to-machine", the interactions are optimized for machines instead of being optimized for humans as websites are. In practice, two major classes of Web services can be identified, namely SOAP-based services and RESTful services. We will discuss both in the following sections.

2.4.1 SOAP-based Services

In an effort to improve the flexibility and dynamicity of their products, the information technology industry started to work on the formalization and standardization of Web services in the late nineties. The outcome was a complex set of specifications. XML was, mainly due to its popularity at the time, chosen as the main data format. The other three main pillars are SOAP, WSDL, and UDDI.

SOAP [7], originally defined as *Simple Object Access Protocol* at Microsoft, specifies a messaging framework consisting of a processing an extensibility model, a protocol binding framework as well as a XML-based message format. The *Web Service Description Language* (WSDL) [87] describes the interface of a Web Service and *Universal Description, Discovery and Integration* (UDDI) [88] registries allow the discovery of services and their interface descriptions.

Even though huge investments have been made, the promise of uniform service interface standards and universal service registries in the form of SOAP, WSDL, and UDDI has proven elusive. The Universal Business Registry, the main public UDDI registry, has been shut down in 2006 and most public SOAP-based Web services have been phased out shortly thereafter. There were many problems that led to this demise but the most fundamental reason is that the whole architecture is based on a Remote Procedure Call style which has been known to be flawed [8] for years. Furthermore, instead of using HTTP as an application protocol, it was misused as a transport protocol. In SOAP, e.g., data is typically retrieved by POSTing a SOAP-request to the service which then returns the desired data. This breaks intermediaries that serve as proxies or caches which typically perform their functions based on the standard semantics associated with the HTTP verbs and headers in the messages flowing through them. In practice, this reduces the scalability enormously and means that running a public service often results in prohibitive costs. Consequently, the number of public SOAP-based services has become negligibly small. Looking at company-internal services, however, the extensive tooling often outweighs these disadvantages so that they are still used in such scenarios.

Another problem when there is no coordination between the publisher and the consumer of a service is that, despite using abstractions of the data types found in the actual implementations, services interfaces described with WSDL often leak implementation details which leads to tightly coupled systems. Similarly, the mapping to the abstract types is not always easily possible and thus often results in severe interoperability problems. Especially the inherent impedance mismatch between XML Schemas (XSD) [89], [90] and object-oriented programming constructs, the so called O/X impedance mismatch, often complicate the integration of different systems. The XML Schema language has a number of type system constructs which simply do not exist in commonly used object-oriented programming languages such as Java [91]. Thus, interoperability problems arise because each SOAP stack has its own way of mapping the various XSD type system constructs to objects in the target platform's programming language and vice versa.

In summary, the problems outlined above and the complexity of the technology, which consists of far more specifications than just SOAP, WSDL, and UDDI, led to a shift towards more lightweight solutions that integrate better into the architecture of the Web, which we will describe in the next section.

2.4.2 RESTful Services

According to statistics from ProgrammableWeb [92], the premier catalog of public Web services, three out of four Web services are based on the REST architectural style. This does not mean that they fully obey REST constraints but primarily that they do use HTTP as an application protocol and that the various resources get their own IRIs. In fact, most services that claim to be RESTful (REST APIs) are not. To capture the various levels of "RESTfulness", Richardson defined a maturity model [93] but by definition, a service is either RESTful by obeying to all constraints defined by REST or it is not. Annoyed by the fact that especially the hypermedia constraint is ignored by many Web services that claim to be RESTful, Fielding wrote a blog post [45] making it clear that the constraint it is not optional. Since the term REST was so often misused, recently HTTP-based Web services are typically simply referred to as Web APIs instead. For services that do obey to REST's hypermedia constraint, the term Hypermedia API has become popular.

Web APIs typically have in common that they use a very small set of standards. Most often, it consists of just HTTP and either XML [49] or JSON [50] as the serialization format. Even though XML with its namespacing support that allows messages to be enriched with hypermedia controls or to be made self-descriptive would be better suited for RESTful services, JSON has become the preferred data interchange format for Web APIs in recent years. The downside of this simplicity is that most services are unique and only documented in natural language. This renders automatic code generation or the creation of generic tooling almost impossible. These problems are the focus of this thesis and will thus be discussed in more detail in the next chapter.

2.5 Discussion

As we have seen in this chapter, the development of the World Wide Web was often chaotic, uncoordinated, and unpredictable. Who would have thought that many of the simple technologies created by a handful of people in the first days of the Web would survive for such a long time while industry-driven, multi-million dollar projects such as SOAP-based Web services would quickly pale into insignificance? Surely technical aspects played an important role but social aspects were not less important. Compared to simple JSON-based Web APIs SOAP-based services quickly "felt" heavy and complex even though, in most cases, the complexity is completely hidden by the sophisticated tooling that has been built around these technologies. Indeed, the Web is often "more a social creation than a technical one." [94] The history of the Semantic Web is another case nicely illustrating this. Its acceptance languished for years but finally a simple Web page [17] full of typographical errors formulating four very basic principles was able to herald an important turning point. The Linked Data principles did not introduce any new technology but were a mere rebranding and clarification of the vision of a Semantic Web—a Web of Data.

The standardization of Web services failed, but the problems these efforts were trying to address are still valid. The Web has grown exponentially and massive amounts of new information are being added as we speak.

We have reached a point at which we humans are the bottleneck for the meaningful usage of all this knowledge. We need to extend the Web to make it easier for machines to process the available information and to exchange and manipulate data without human intervention in order to better assist us. The combination of Semantic Web technologies and RESTful Web APIs might help to bring us a step closer to this ambitious goal but, as already suggested, they suffer from various issues. In the next chapter we will look at those issues in detail and define the problem this thesis is addressing.

Chapter 3

Problem Definition

Web APIs are increasingly important in connecting distributed systems. They are becoming the glue keeping together systems within an organization while, at the same time, providing unprecedented, open access to data managed by these systems to the wider world. Yet, the proper design and implementation of Web APIs remain largely more an art than a science.

In this chapter, which is based on our previous work in [95], [25], [96], [97], [98], and [99], we will analyze the current best practices for the creation, documentation, and usage of Web APIs. We will distill a number of shortcomings and issues and finally formalize the research problems addressed by this thesis.

3.1 Proprietary Data Formats and Models

One of the first design decisions a developer has to make when creating a Web API is to choose the serialization format and data model. Web services created in the last decade use almost exclusively either XML or JSON as their serialization format. XML, whose first version was published in 1998, is as a markup language, i.e., a language which allows documents to be annotated with machine-processable instructions. At the beginning of the new millennium, XML was extremely popular and the preferred choice for many use cases. Not surprisingly, the first Web services also relied on XML demonstrating its flexibility and extensibility. SOAP-based services [7] use XML for data interchange alongside WSDL [87] and XML Schemas (XSD) [37-38] as description formats. This results in Web services where both the data as well as the interfaces are described in a machine-readable way which enabled the creation of powerful tooling assisting developers in the implementation of such services. Often both the server-side as well as the client-side code can be generated completely automatically out of these descriptions. Thus, the client and the server are typically tightly bound in such a system. Typically even the change from, e.g., a 64 bit integer to a 32 bit integer requires the recompilation of the client. Furthermore, the inherent impedance mismatch between XML and object-oriented programming constructs (O/X impedance mismatch) generally results in severe interoperability problems. The fundamental problem is that the XML Schema language has a number of type system constructs which simply do not exist in commonly used object-oriented programming languages such as Java. In consequence, this leads to interoperability problems because each SOAP stack has its own way of mapping the various XSD type system constructs to objects in the target platform's programming language and vice versa. Recent extensions for common languages such as Cω or LINQ (Language Integrated Query) for C# or E4X (ECMAScript for XML) for JavaScript ease the data handling enormously and avoid the inherent O/X impedance mismatch.

Nevertheless, in most cases all a developer wants to do is to interchange data—and here we are distinguishing between *data interchange* and *document interchange*. In 2002 Douglas Crockford realized that JavaScript object notation can be used as a simple data interchange format. He extracted a small subset of the JavaScript programming language [100] which he called JSON (JavaScript Object Notation) with the aim to create a lightweight, language-independent data-interchange format which is easy to parse and easy to generate. Initially, JSON was only documented on Crockford's website json.org but after requests from larger companies asking for a more stable specification he wrote an IETF Internet Draft which eventually became RFC 4627 [50] in 2006. Since then, JSON enjoys an ever-increasing popularity across the Web community [92]. Given that JSON's whole specification [50] consists of just 10 pages (with the actual content being a mere 4 pages), it is often considered to be a much simpler format and thus easier to use and understand than XML for which the XML Core Working group alone lists *XML*, *XML Namespaces*, *XML Inclusions*, *XML Information Set*, *xml:id*, *XML Fragment Interchange, XML Base*, and *Associating Stylesheets with XML* as standards [101], not even including *XML Schema Part 1* and *XML Schema Part 2*.

From a REST perspective, the current practice of using JSON could be seen as a step backwards. While XML with its namespacing support [53] allows self-describing messages to be created even when its generic media type `application/xml` (or `text/html`) is used, the same is not the case for JSON. If a JSON message is labeled with the generic media type `application/json`, all the message semantics as well as the processing model have to be documented out of band which introduces an undesired coupling between the publisher and the consumer of such data. To work around this issue, the current best practice for developing truly RESTful JSON-based Web APIs is to define a custom media type which defines the semantics of the used JSON structures. This allows JSON to be extended to support labeled hyperlinks—another painfully missing concept needed to create RESTful Web APIs with JSON. Unfortunately, however, even just creating a specialization of an existing generic media type is not as straightforward as it might seem at first sight.

On the one hand, it is not trivial to design a media type that is general enough for a broad range of applications, yet useful. On the other hand, it is difficult to find broad acceptance for a media type that is only usable in a very specific application domain. Obviously, if the media type introduces a new serialization format, no existing libraries can be used to parse its representations forcing all clients to implement parsers specifically designed for this new media type. While such an approach might provide the best possible efficiency, it does not scale when the number of services or even if just the number of entities using different media types in a single service increases. The practice of defining specialized media types for each entity type used in an application is especially problematic as it promotes the reuse of these specialized media types to design the application-level data model. More than likely, such an approach will result in tighter coupled systems at the model layer given that the same data model is shared among all system components. The fact that only very few of the more than 1,300 officially registered media types [102] are in common use should be evidence enough that their design is not trivial and requires a lot of expertise. Arguing that every RESTful service should design its own specific media type to document the contract with its clients is thus clearly impractical and far from reality. It also indicates that generally, services either stick to generic media types such as XML or JSON or do not invest the necessary time and effort to register their proprietary media types.

One of the main problems with media types is that they are organized in a very shallow, two-level deep hierarchy. This makes it impossible to define refinements or extensions in a way which would make it possible to deduce those dependencies from the media type's identifier. Given that it is also impossible to describe such dependencies in a machine-processable way in the media type's specification itself, the only available option is to directly include that knowledge into a client's code.

In principle the same applies to media types that build on top of existing, generic media types such as XML or JSON. A common pattern is to add, e.g., a `+json` suffix to the media type identifier to describe that it is based on JSON's syntax. Even though this practice has been standardized [103] (and has been so for XML for more than a decade [104]) some client libraries still do not understand this convention. To be fair, it is also not clear what libraries should do with this information; all it tells is the serialization format. In human-facing tools, such as browsers, this information might be used to render a representation as if it would have been served using the base type instead of not displaying it at all due to an unknown media type. For programming libraries the situation is much less clear as all that can be done is to parse the representation which, most of the time, is the most trivial aspect.

Looking at, e.g., XHTML, SVG, Atom, and RDF/XML it becomes clear that all these formats share is the serialization format. The processing models and even the data models are completely different. XHTML for instance deals with a document object tree while RDF/XML is used to serialize graphs. In such cases it certainly makes sense to create specific media types. If, however, the only difference lies in the semantics, i.e., the meaning of the serialized data, it is questionable whether specialized media types are required at all. The examples best illustrating this are probably xCard [51] and xCal [52] as they are doing nothing more than specifying XML-based serializations for vCard and iCalendar. Such "micro-types" are the main reason for the often criticized proliferation of media types. The concern is that an abundance of media types conflicts with REST's emphasis on a uniform interface. The more variability there is the more difficult interoperability becomes. Instead of requiring developers to create new media types for every minor semantic difference, more generic media types able to express the various semantics and mechanisms to signal them at the HTTP layer are necessary. This would allow the creation of composable contracts, improve the Web as a platform in general, and simplify the development of Web APIs in particular.

As Steve Vinoski argues in his excellent article [14], platforms, although efficient for their target use case, often inhibit reuse and adaptation by creating highly specialized interfaces, even if they stick to industry standards. He argues "the more specific a service interface [is], the less likely it is to be reused, serendipitously or otherwise, because the likelihood that an interface will fit what a client application requires shrinks as the interface's specificity increases." This observation surely applies to most current Web APIs which are, due to their specializations, rarely flexible enough to be used in unanticipated ways.

Needless to say that data integration is also made much more difficult given that data models differ widely and all the semantics are implicit. If the semantics were explicit and the data model generic, data integration would be drastically simplified. In fact, data could be integrated (semi-)automatically with other data sources. For instance, a typical mashup combining and showing data from different sources on a map could be created automatically. The widget would be able to automatically figure out which parts of the representation represent the needed coordinates and in consequence render the data on the map. This would render the creation of dashboards, an important business use case, much simpler and eliminate a lot of the usually needed data mediation code.

3.2 Static Contracts in Natural Language

In order for two or more components of a distributed system to interoperate, a contract has to be established. As we have seen in section 2.2, contracts on the Web are based on media types and protocols. In contrast to other distributed system architectures, these contracts are centrally owned and negotiated at runtime instead of being defined at design time. This involves not only the negotiation of media types but also the use of hypermedia to dynamically convey valid state transitions.

The use of *hypermedia as the engine of application state* [11] is a central aspect of the REST architectural style and when building traditional Web sites, developers intuitively use it to guide visitors through their sites. They understand that no visitor is interested in reading documentation that tells them how to handcraft the URLs necessary to access the desired pages. Developers spend considerable time to ensure that their sites are fully interlinked so that visitors are able to reach every single page in just a few clicks. To achieve that, links have to be labeled so that users are able to select the link bringing them one step closer to their goal. Often that means that multiple links with different labels but the same target are presented to make sure that a visitor finds the right path. This is most evident when looking at the checkout process of e-commerce sites which usually consists of a single path leading straight to the order confirmation page (plus a typically de-emphasized link back to the homepage or shopping cart). On this path, the user has to fill in a number of forms asking for order details such as the shipping address or the payment details. It is not a coincidence that these forms tend to use exactly the same language on completely different e-commerce sites. It is also not a coincidence that the same names for the form fields are chosen to allow the user's browser to fill the fields automatically in or, at least, offer auto-completion. HTML5 tries to push that even further by introducing an `autocomplete` attribute along with a set of tokens in order to standardize the auto-completion support across browsers [105]. All this is part of purposeful optimization with the clear goal to increase conversion rates, i.e., to ensure that visitors achieve their goal.

These practices build the foundation of today's Web, a gigantic graph consisting of billions and billions of interlinked pages. Hyperlinks are such a fundamental building block of the Web's architecture that it feels natural to browse across sites from completely different publishers. It is taken for granted that content links to other relevant content; relevant links are generally seen as a sign of quality. Surprisingly, Web services very rarely link to external data. As a matter of fact, most times even links to other resources within the service itself are missing. More often than not, developers completely ignore hypermedia when creating solutions for machine-to-machine communication.

One of the primary reasons for this is certainly that for Web APIs no accepted, standardized media type with hypermedia support exists. JSON, which is much easier to parse and has a direct in-memory representation in most programming languages, is typically favored instead of using HTML as on the human Web. Unfortunately, this often leads to the exposure of internals resulting in a tight coupling between the server and its clients. A common example for this is the inclusion of local, internal identifiers in representations instead of including links to other entities. This requires out-of-band knowledge of IRI templates to reconstruct the URLs to retrieve representations of entities referenced in such a way. Since in most cases the documentation about those IRI templates is not machine-readable, they are hardcoded into clients which means that clients break whenever the server implementation changes.

The current best practice for developing truly RESTful JSON-based Web APIs is to define a custom media type which extends JSON to support labeled hyperlinks. Effectively this means that HTML's anchor or link tags with their relation attributes (`rel`) are imitated by some JSON structure. Since there is a common need for such functionality, there have already been some efforts to standardize such extensions to JSON, but so far their adoption is very limited. Far more often these proprietary extensions are documented out-of-band on the API publisher's homepage. Apart from the description of how hyperlinks are expressed, these documentations generally also include a list of resource types (such as products and orders) describing their semantics, properties, and serializations. Last but not least, a number of link relations along with the supported HTTP operations, the expected inputs and outputs, and the consequences of invoking those operations are documented. This allows developers to get an overview of the API's service surface and to implement specialized clients.

A negative side effect of this proliferation of proprietary data formats and the use of natural language to document them is that it becomes almost impossible to create generic clients similar to browsers on the human Web. Thus, developers usually need to implement not only the server side part but also a client (library) which is then used by other developers to access the Web API. It is not surprising that these two components are often tightly coupled given that they are frequently developed in lockstep by the same team or at least the same company. Consequently developers choose to use the simplest approach to solve their problem at hand. Instead of using dynamic contracts that are retrieved and analyzed at runtime, which would, just as on the human Web, allow clients to adapt to ad-hoc changes, static contracts are used. All the knowledge about the API a server exposes is typically directly embedded into the clients. This leads to tightly coupled systems which impede the independent evolution of its components. When a service's *domain application protocol* [46], which defines the set of legal interactions necessary to achieve a specific, application-dependent goal, is defined in a static, non-machine-readable document served out-of-band, it becomes impossible to dynamically communicate changes to clients. Even though such approaches might work in the short term, they are condemned to break in the long term as assumptions about server resources will break as resources evolve over time.

3.3 Manually Written Documentation

Writing documentation is certainly a task that most developers would like to avoid. Therefore, tools for virtually every programming language have been created to formalize and streamline the process of writing (at least some minimal) documentation. In most cases, such documentation is written by directly annotating the code. Specialized tools such as Doxygen [106], so called documentation generators, analyze those annotations as well as the code itself and create consistently formatted documentations. At the same time, such tools can leverage the fact that the documentation is clearly bound to certain code fragments such as classes or methods and enable, e.g., integrated development environments (IDEs) to display assisting documentation during the development process. The documentation is automatically queried and put in the context of the programmer's task. This allows programmers to stay focused on the problem at hand instead of having to jump back and forth between their code and the documentation of the APIs of the programming libraries they are using. Given that most documentation for Web APIs is in the form of manually written HTML pages which do not follow a well-defined structure and are thus not machine-processable, it is impossible to create similar assisting tooling for Web APIs.

The Linked Data community is clearly a step ahead of the REST community in this regard. In contrast to the current practice for Web APIs, Linked Data is described in detail in a machine-readable way. All data publishers have to do, is reuse one or more of the many existing vocabularies to express their data. This not only eliminates the need to document the semantics of the data over and over again but also improves the interoperability of systems exchanging such data and the reusability of code. It would thus make sense to reuse these technologies and vocabularies for the creation and description of Web APIs. All a developer has to do is annotate the code with concepts from a vocabulary. The human-readable documentation can then be generated automatically by using the `rdfs:label` and `rdfs:comment` properties associated to that concept in the vocabulary. The positive side effect of such an approach is that machines would be able to recognize when equivalent elements are encountered and process them using existing code instead of requiring manual adaptations by the implementer. In fact, the documentation could be linked directly to the messages which would make them self-descriptive and thus reduce the need for additional human-readable documentation.

3.4 The Semantic Web: Complex, Read-Only, and No Links?

Since the Linked Data principles align well with the REST architectural style (see [107] for an extensive analysis) it would seem natural to combine their strengths as we have seen in the previous section. Nevertheless, the two remain largely separated in practice. Instead of providing Linked Data via RESTful Web services, current efforts deploy centralistic SPARQL endpoints or upload static dumps of data which rarely reflects the nature of the data, i.e., descriptions of interlinked resources. Just as public SQL endpoints are uncommon nowadays, public SPARQL endpoints are not expected to become widespread in the near future. This is because it is considerably more expensive to expose SQL or SPARQL endpoints than easier-to-optimize RESTful service interfaces.

While most of the efforts by the Semantic Web community are spent on the accurate description of resources, which could be compared to the self-descriptive messages constraint, the *linking* of data received little attention. RDF uses IRIs to identify entities but that does not imply that those IRIs are dereferenceable or that RDF has built-in support for hypermedia. In fact, neither RDF itself nor RDF Schema or OWL defines a concept to describe dereferenceable IRIs. Whether an IRI is intended to be dereferenced or not, depends implicitly on what it represents. FOAF's `homepage` property [108], e.g., suggests that its values are dereferenceable IRIs. Without further out-of-band knowledge, however, a machine would not be able to infer that information. In fact, in the early days of the Semantic Web, most of the data used IRIs that did not dereference to anything useful. The Linked Data principles postulated by Berners-Lee [17] in 2006 were an attempt to change that.

Berners-Lee urged to use IRIs to name things that dereference to useful information in a standardized format. Additionally, the returned data should contain links to other relevant data in order to create a giant graph of Linked Data that could be seen as the direct data-centric counterpart of the document-centric human Web. At the time of this writing, the Linked Data community, even though it advocates the use of dereferenceable identifiers, leaves open how to recognize them. RDF still has no built-in notion of hypermedia but uses IRIs solely as identifiers. It is therefore not surprising that the vast majority of the available data are largely read-only representations. The best a client can do is to blindly try to interact with these IRIs. To change this, a vocabulary able to describe affordances beyond simple dereferenceability would be needed.

Another aspect developers are often struggling with in practice is that in RDF properties have, just as classes and everything else that is identified with an IRI, global scope and independent semantics. In contrast, properties in the models used by most Web APIs are class-dependent. Their semantics depend on the class they belong to. In data models classes are typically described by the properties they expose whereas in RDF properties define to which classes they belong. If no class is specified, it is assumed that a property may apply to every class. This behavior stems from the fact that RDF Schema [65] and OWL [66], the two preferred languages to describe RDF vocabularies, work under an open-world assumption. In contrast, data models used by programmers typically work under a closed-world assumption. The difference is that when a closed world is assumed, everything that is not known to be true is false or vice versa. With an open-world assumption the failure to derive a fact does not automatically imply the opposite; it embraces the fact that the knowledge is incomplete. One of the effects illustrating the difference in those world views is that in data models an instance of a class also belongs to all its superclasses, but not any other class. In ontologies using an open-world assumption, the same cannot be said unless classes are explicitly defined as being disjoint.

These differences have interesting consequences. For example, the commonly asked question of which properties can be applied to an instance of a specific class cannot be answered for RDF. Strictly speaking, any property which is not explicitly forbidden can be applied. This may sound counter-intuitive and could lead to the wrong conclusion that RDF Schema and OWL cannot be used to define data models. In fact they can, but that is not their intended use.

While data models are used to describe the information in a specific, well-delimited application domain; vocabularies, as described by RDF Schema or OWL, are used to define concepts that can be shared across multiple application domains. In other words, data models are typically used to specify validity criteria and constraints for data processed within an application whereas vocabularies are used to reason over data to discover new knowledge. In this light, data models can be said to be intended for closed-world systems whereas vocabularies are intended for open, distributed systems. This may sound surprising as the motivation for most Web APIs is to build open distributed systems. However, as a matter of fact, most current Web APIs just represent small, closed-world systems that happen to be accessible over a standardized protocol with a uniform interface, i.e., HTTP. Neither the entities, nor the concepts defined by such a Web API can be reused in other systems without some special glue code. To simplify data integration and enable reuse, it would thus be sensible to describe the data and behavior exposed by a Web API using RDF vocabularies.

As famously illustrated by the well-known Linked Open Data cloud diagram [80] (see Figure 6 on page 30), the amount of Linked Data has managed to grow significantly over the last couple of years but, nevertheless, the greater vision of a Semantic Web, which has been around for more than fifteen years, still has a long way to go before mainstream adoption will be achieved. The Linked Data principles specifically and Semantic Web technologies in general, have yet to find their way into the design of RESTful Web APIs. The fundamentally different models of Semantic Web technologies with their open world assumption, the lack or immaturity of tools, and the (perceived) complexity are just some of the reasons for this lack of adoption. For a long time, the Semantic Web suffered from a classic chicken-and-egg problem as there were no clear incentives for developers to use it. This aspect is improving recently as major search engines started to index some structured data such as RDFa and microformats. Another problematic factor, especially in the enterprise space, is that the Semantic Web is perceived as a disruptive technology, making it a show-stopper for organizations needing to evolve their systems and build upon existing infrastructure investments. Changing whole systems to be based on triples whereas most developers program their systems in an entity centric, i.e., object-oriented manner is often not a viable option. Additionally, the current Semantic Web approaches usually provide just read-only interfaces to the underlying data. This clearly limits the usefulness and inhibits networking effects and engagement of the crowd.

Beside these technical issues, a lot of developers are also simply overwhelmed by the complexity, perceived or otherwise, or are just reluctant to use new technologies. The prevalent terminology, suffused with words such as *Ontology,* just seems to fuel their misconceptions. Furthermore, the fact that the Semantic Web community derailed into the artificial intelligence domain instead of concentrating on more practical data-oriented applications certainly played a major role in that regard as well. A lot of potential users were alienated by this and developed an aversion to Semantic Web technologies—a phenomenon we denoted as *Semaphobia* [95]. A solution to this problem might be a more gradual introduction to those principles and practices by the use of less disruptive technologies. Furthermore, clear incentives along with simple specifications and guidelines are necessary.

The recent introduction of Schema.org [85] and Microdata [109] nicely illustrate the potential of such an approach and the willingness of Web developers to adopt it. Instead of creating a completely new serialization format, Microdata adds a number of attributes to HTML for its semantic annotation. Schema.org, on the other hand, represents a vocabulary for a broad range of application domains ranging from events and recipes to products and people. Schema.org is a joint effort between Google, Microsoft, Yahoo!, and Yandex which all added support for both Microdata and Schema.org to their search engines to extract structured data from web pages in order to improve the precision of search results and to present the results in a visually more appealing way. Web developers benefit from higher click-through rates which increases the number of people visiting their sites. Unfortunately, a similar approach for machines talking to each other via Web services is still missing. Developers thus still have to deal with a plethora of heterogeneous data formats, data models, and service interfaces when interacting with Web APIs.

3.5 Missing Tooling

As outlined in section 3.2, developers instinctively use hypermedia when building traditional Web sites but seem to ignore it completely when building Web APIs. One of the reasons behind this might be the different level of tooling support on both the server and the client side.

Current Web development frameworks are typically based on a Model-View-Controller (MVC) architecture [110]. MVC is a design pattern that separates the presentation of information from its processing to allow code reusability and separation of concerns. The models represent the relevant entities in the system, the views create representations of those entities, and the controller is responsible for processing inputs, manipulating the models, and finally returning an updated representation by using the according views. Web frameworks often further modularize the code by dividing controllers into Front Controllers, which handle all requests for a Web site, and Page Controllers or Commands, which are only responsible for certain requests [110]. Therefore, the front controller's job is typically to parse the received HTTP request, extract the request IRI and method, and then pass the control to a specific page controller or command which then, in turn, invokes specific models and views.

In the context of this thesis, the view layer is the most critical layer as it decouples the internals of an application from its external representation.

In Web development frameworks, the view layer typically consists of templates and a rendering engine. By creating representations in a standardized and centrally owned format and targeting generic clients, i.e., Web browsers, internals are usually very well hidden behind a common standardized interface. This decouples the client from the server and enables the independent evolution of the two. Unfortunately, the situation typically looks quite different for Web APIs. Instead of a sophisticated view layer that decouples the internals from the system's external interfaces (representations), most of the time a serializer is used to marshal the in-memory representations (often complete object graphs) into a generic data format such as JSON. This would not be problematic per se, if the contract were not owned by the server instead of being centrally owned.

Since the server might decide to change the contract at any time, there are very little incentives for independent developers to invest time and money to create sophisticated and thus more flexible and dynamic service clients. On the technical side, the fact that JSON, e.g., has no built-in support for hyperlinks makes it impossible to leverage hypertext as the engine of application state (HATEOAS) [11] without additional out-of-band information. In consequence, consumers of such Web APIs expect that API publishers provide specialized programming libraries to simplify the usage of their services. The result are tightly coupled systems in which the clients are statically bound to the server's URL space and thus need to be updated in lockstep with the server. Not uncommonly, clients for object-oriented programming languages directly replicate the classes to represent the various resource types exposed by the server as native objects on the client. Such an approach clearly inhibits the independent evolution of components.

3.6 Discussion

The problem with the practices outlined in this chapter is that they result in specialized implementations targeting specific use cases and not generalizations that can be reused across application domains. Therefore, every API created with such an approach is unique and needs to be documented. Even though most of the code to access such services is very similar, there are still minor differences which make it difficult to reuse code and almost impossible to write generic clients. On the human Web this problem is addressed by a generic media type (HTML) which decouples the clients from the servers they are accessing. Admittedly, HTML could be used for Web APIs as well but its nature, which targets human facing web pages that are essentially graphical user interfaces, is fundamentally different to machine-to-machine communication. A data interchange format such as JSON is a much better fit for use cases which just require the transfer of structured data; having to parse HTML for this has typically too big of an overhead. Thus, to solve this problem also for Web APIs, a generic media type to create self-descriptive messages with inherent support for hypermedia is needed. Just adding support for hyperlinks to JSON, as most current approaches do, is not enough because it only solves part of the problem. Since the interaction with Web APIs could generally be seen as a data integration problem, other aspects, such as globally unique identifiers for both the entities and their properties, become important as well. By using semantic annotations, a client would not only be able to figure out which elements in a JSON representation represent IRIs but also what these IRIs and all the other elements mean.

We believe that it should be feasible to standardize and streamline the development of Web APIs by combining ideas and principles from both the REST architectural style and the Semantic Web. Having identified several issues and shortcomings of current best practices, we are thus able to formalize the problem statement that integrates these issues into the primary aim of this thesis, i.e., to support developers creating, documenting, and using RESTful Web APIs.

The problem addressed in this thesis consists of three clearly defined sub-problems that are summarized as follows:

- Development of a generic and extensible data interchange format or description language specifically designed for RESTful Web APIs. This requires support for hypermedia controls as well as a mechanism similar to XML namespaces to enable the creation of self-describing messages. The hypermedia support may either be included directly into the format similar to HTML or in an external vocabulary similar to the XML Linking Language [111]. The solution should address the problems discussed in this chapter.

- Design and implementation of plugins or programming libraries integrating support for the created data interchange format and vocabulary into current Web frameworks in order to simplify the creation of truly RESTful Web APIs. The aim is to make the creation of Web APIs comparably simple as the creation of a traditional website. The implementation of this proof of concept will help to evaluate the practical relevance and usability of the created solution.
- Design and implementation of a generic client accessing such Web APIs. This requires the dynamic evaluation of messages and service descriptions at runtime instead of hard-coding contracts into the client at design time as most current approaches do. The generic client will not just represent a highly valuable tool but also demonstrate some of the features realizable by building Web APIs based on the proposed data interchange format and vocabulary.

Given that the solution is targeting RESTful services, it clearly has to adhere to REST's [11] architectural constraints which can be summarized as follows: 1) stateless interaction, 2) uniform interface, 3) identification of resources, 4) manipulation of resources through representations, 5) self-descriptive messages, and 6) hypermedia as the engine of application state. Stateless interaction means that all the session state is kept entirely on the client and that each request from the client to the server has to contain all the necessary information for the server to understand the request; this makes interactions with the server independent of each other and decouples the client from the server. All the interactions in a RESTful system are performed via a uniform interface which decouples the implementations from the services they provide. To obtain such a uniform interface every resource is accessible through a representation and has to have an identifier (whether the representation is in the same format as the raw source, or is derived from the source, remains hidden behind the interface). All resource representations should be self-descriptive, i.e., they are somehow labeled with their type. Finally, the hypermedia as the engine of application state (HATEOAS) constraint refers to the use of hyperlinks in resource representations as a way of navigating the state machine of an application.

While all of these constraints are important when designing a RESTful service, the most important aspects in the context of this thesis are how resources can be accessed, how they are represented, and how they are interlinked. The solution should be expressive enough to describe how resource representation can be retrieved and manipulated, and what the meaning of those representations is. To integrate the resulting services into the Semantic Web, it should also be possible to transform resource representations to RDF. An important requirement to foster adoption, to evolve systems, and to build upon existing infrastructure is that no (or just minimal) changes on existing systems are required; this implies a requirement to support partial descriptions that can be completed later. Finally, in order to lower the entry barrier for developers, the approach has to be as simple as possible and provide instant incentives such as increased productivity or enhanced reusability.

Chapter 4

Related Work

In order for two (or more) systems to communicate successfully there has to be an agreement or contract on the used interfaces, data formats and processing models as well as the semantics. In the traditional Remote Procedure Call (RPC) model, where all differences between local and distributed computing are hidden, usually static contracts in the form of an Interface Description Language (IDL) are used to specify those interfaces. The data types that such an IDL offers are abstractions of the data types found in actual programming languages to enable interoperability between different platforms. SOAP-based services typically follow the same approach by describing the interfaces with WSDL [87] and XML Schema [112] documents. Since in such a model all the documentation is machine readable, automatic code generation on both the client and the server sides are made possible. This improves developers' productivity but also increases coupling.

In contrast, the REST architectural style is characterized by the use of contextual contracts where the set of actions varies over time. Additionally, the interface variability is almost eliminated due to REST's uniform interface. In consequence REST-based services are almost exclusively described by human-readable documentation describing the URLs as well as the data expected in requests and returned by the corresponding responses. Generally, that data is not described by specifying media types, but by the definition of specific JSON or XML structures. Due to a lack of formalism and the ambiguity of natural language, these descriptions can neither be automatically transformed into code nor be interpreted at runtime; they have to be hardcoded into clients and servers at design time.

In this section we will examine the most important related work addressing the issues outlined above and in the previous chapter. We will begin with an overview of generic interface description languages applicable to a broad range of application domains before we look at a number of ontologies and vocabularies which extend syntactic interface descriptions with semantic annotations. Such semantic descriptions typically promise higher level of automation for tasks like discovery, negotiation, composition, and invocation. Since most Semantic Web Service (SWS) technologies use ontology languages as the underlying data model they also provide the means for tackling the interoperability problem at the semantic level instead of just at the syntactic level, enabling the integration of Web services into the greater vision of the Semantic Web. We will also discuss some of the specialized media types that have been created to implement RESTful systems for specific use cases. Finally, given that JSON has become the preferred data interchange format in Web APIs, we will also provide an overview of proposals which attempt to add support for hyperlinks and namespaces to JSON.

This section is based on our previous work in [96], [26], and [28].

4.1 Interface Description Languages

Over the years, multiple interface description languages for RESTful services have been proposed. Most of them, such as WRDL [113], NSDL [114], SMEX-D [115], Resedel [116], RSWS [117], and WDL [118] were more or less ad-hoc inventions designed to solve particular problems and haven't been updated for many years. So far none of them managed to attain noticeable adoption and even approaches that were the outcome of large research projects did not manage to break out of their academic confines.

There have been very controversial discussions as to whether REST even needs service interface description languages in the traditional sense. The opponents typically argue that the definition of media types (although not machine readable) eliminates the need for additional descriptions as they describe the format as well as the semantics of exchanged message. However, as we have seen in section 2.2, the creation of specialized media types is not trivial and thus, in practice, most systems rely on generic, application-agnostic media types instead. Furthermore, the fact that new solutions trying to create machine readable documentations are proposed on an almost monthly basis clearly indicates that developers are not satisfied with the status quo.

In this section, we will discuss the most noteworthy proposals which define interface description languages for RESTful services. We do not restrict our selection to purely syntactic descriptions, but also present some approaches that either add or are based on semantic descriptions.

4.1.1 WSDL and SAWSDL

The Web Service Description Language (WSDL) [87] is an established standard to describe the contract between a Web service provider and its clients at a syntactic level. It documents the message formats (schemas), transport protocols, and locations. WSDL 2.0 introduced some significant changes to the structure of WSDL documents and added new features such as interface inheritance and extensible message exchange patterns. It was the first version which was designed with RESTful services in mind and consequently the first WSDL version to support their description; the WSDL 1.1 HTTP binding was inadequate to describe RESTful services. WSDL descriptions typically use XML Schema [112] as schema language but others, such as DTD, RelaxNG, or even non-XML type systems, would be allowed as well.

A WSDL description consists of the following four elements: `types`, `interface`, `binding`, and `service`. The `types` element describes the Web service's messages; even though other type systems are allowed practically only XML Schema is used. The `interface` describes the supported operations (this is where WSDL exposes its RPC-orientation) with the corresponding input, output, and fault messages and the respective message exchange patterns. The purpose of the `bindings` element is to specify how those messages can be exchanged. It specifies the concrete message format (the actual serialization format in contrast to the abstract definition in the `types` element) and the transmission protocol details for each operation and fault in an interface. Finally, the `service` element specifies a list of endpoints where the service can be accessed. Each endpoint is associated with a specific binding to indicate what protocols and transmission formats have to be used.

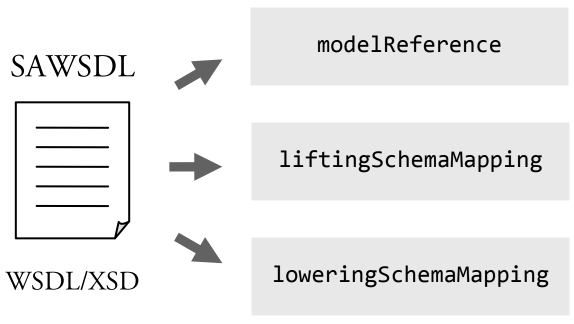


Figure 7. SAWSDL extension attributes

With the introduction of Semantic Annotations for WSDL and XML Schema (SAWSDL) [119] the W3C standardized a mechanism to associate semantics with service interfaces and message schemas. SAWSDL defines how to add semantic annotations to various parts of a WSDL document such as inputs, outputs, interfaces, and operations, but it does not specify a language for representing the semantic models. Instead, it just defines how semantic annotation is accomplished using references to semantic models such as ontologies, by providing three new extensibility attributes to WSDL and XML Schema elements as shown in Figure 7.

The `modelReference` extension attribute defines the association between a WSDL or XML Schema component and a concept in some semantic model. It is used to annotate XML Schema type definitions, element declarations, and attribute declarations as well as WSDL interfaces, operations, and faults. The other two extension attributes, named `liftingSchemaMapping` and `loweringSchemaMapping`, are added to XML Schema element declarations and type definitions for specifying mappings between semantic data and XML. SAWSDL allows multiple semantic annotations to be associated with WSDL elements. Schema mappings as well as model references can contain multiple pointers. Multiple schema mappings are interpreted as alternatives whereas multiple model references all apply. SAWSDL does not specify any other relationship between them [119].

The major critique of SAWSDL is that it comes without any formal semantics. This hinders logic-based discovery and composition of Web services described with SAWSDL but calls for "magic mediators outside the framework to resolve the semantic heterogeneities" [120]. Similarly, even though it is technically possible to use WSDL 2.0 to describe RESTful Web services, it is not perceived as suitable by developers. Thus, WSDL usage is limited to the description of traditional SOAP-based services. A reason for this lack of adoption might also be the inherent complexity of the WS-* stack compared to the lightweight model of typical RESTful services.

4.1.2 WADL

The approach of the Web Application Description Language (WADL) [121] is closely related to WSDL. With WADL a developer creates a monolithic XML file containing all the information about the service interface to make it machine-accessible. Given that WADL was specifically designed for describing RESTful services (or HTTP-based Web applications as they are called in the specification), it models the resources provided by the service and the relationships between them instead of putting operations at the center as WSDL does.

In WADL each service resource is described as a request containing the used HTTP method and the required inputs as well as zero or more responses describing the expected service response representations and HTTP status codes. The data format of the request and response representations are described by embedded or referenced data format definitions. Even though WADL does not mandate any specific data format definition language, the use of RelaxNG and XML Schema are described in the specification.

The main critique of WADL is that it is complex and thus requires developers that have a certain level of training and tool support to enable the usage of WADL. This complexity contradicts the simplicity of RESTful services. In addition, WADL urges the use of specific resource hierarchies which introduce an obvious coupling of the client and server. Servers should have the complete freedom to control their own namespace. In contrast to WSDL, no mechanism to semantically annotate service descriptions exists for WADL. A reason for WADL's missing uptake might be that in practice it offers too few advantages to justify the increased overhead, complexity and thus cost.

4.1.3 Swagger and Google's API Discovery Service

Over the years a number of similar interface description languages have been proposed and more recently most of them use JSON as their serialization format. At the time of this writing, Swagger [122], is probably the approach that received the most traction in the community due to its early release and the availability of open source tools. It follows quite a similar approach to WADL. The biggest difference is that it does not impose any specific resource hierarchy. Other than that, it allows the association of almost exactly the same information to URI templates: an HTTP method, request parameters, response type, hints for returned status codes, and natural language descriptions. Swagger is mainly intended to enrich human-facing API documentations with interactive controls so that the various operations can be tested directly in the browser. It also enables the automatic generation of client libraries. This makes it very similar to Google's API Discovery Service [123] which follows a very similar approach and is mainly used to generate client libraries in different programming languages for Google's numerous Web APIs.

All of these approaches, which also include solutions like I/O Docs [124], API Blueprint [125], and RAML [126] (some of which use Markdown or YAML instead of JSON), have in common that everything is bound to the URLs to access the various resources. This is clearly opposed to REST's hypermedia constraint which demands the dynamic discovery of resources at runtime.

4.1.4 SA-REST, hRESTS, and MicroWSMO

Compared to the approaches presented so far, SA-REST [127] follows a fundamentally different approach. Instead of creating description documents for machines and tools, SA-REST tries to exploit the fact that almost all RESTful services have textual documentation in the form of HTML pages. Its basic idea is to annotate those documents with RDFa [128] to make the information accessible to machines.

SA-REST offers the following service annotations (depicted in Figure 8): 1) `input` and 2) `output` to facilitate data mediation; 3) `lifting` and 4) `lowering` schemas to translate the data structures that represent the inputs and outputs to the data structure of the ontology, the grounding

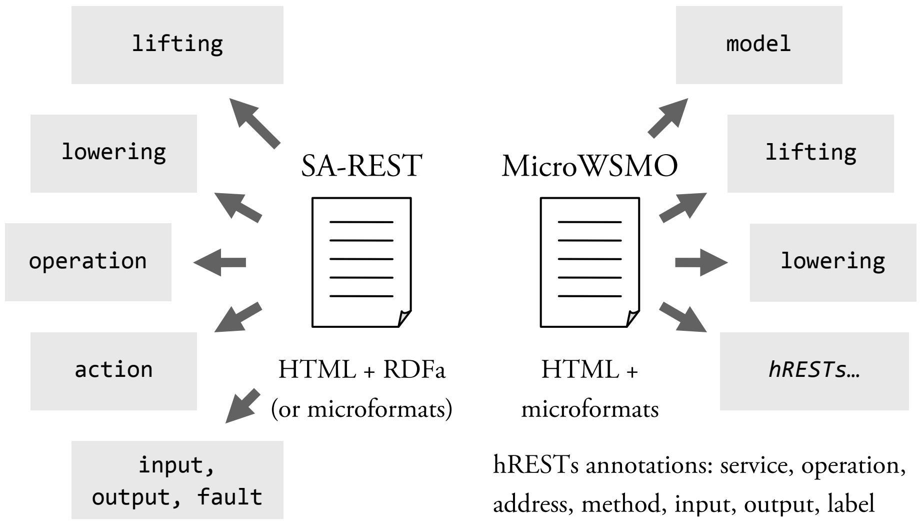


Figure 8. SA-REST and MicroWSMO (with hRESTs)

schema*;* 5) `action`, which specifies the required HTTP method to invoke the service; 6) `operation` which defines what the service does; and 7) `fault` to annotate errors.

Given that SA-REST is a derivative of SAWSDL, it is possible to transform SA-REST descriptions into WSDL 2.0 documents that are annotated with SAWSDL and vice versa (even though the information in annotated WSDL documents is not rich enough to create meaningful HTML documents). Similarly to SAWSDL, SA-REST does not enforce the choice of language for representing the ontology or conceptual model of a service.

hRESTS (HTML for RESTful Services) is an approach that is very similar to SA-REST but uses microformats [129] instead of RDFa. The main differences between the two approaches are thus not the underlying principles but rather the implementation techniques. A single HTML document enriched with hRESTS microformats can contain multiple service descriptions and conversely multiple HTML documents can together be used to document a single service (addressing the common practice of splitting service documentations into multiple HTML documents to make them more digestible).

Each service is described by a number of operations, i.e., actions a client can perform on that service, with the corresponding URL, HTTP method, the expected inputs and outputs. While hRESTS offers a relatively straightforward solution to describe the resources and the supported operations, there is some lack of support for describing the data schemas. Apart from a potential label, hRESTS does not provide any support for further machine-readable information about the inputs and outputs. Extensions such as MicroWSMO address this issue.

MicroWSMO [130] is an attempt to adapt the SAWSDL approach for the semantic description of RESTful services. Just as hRESTS, on which it relies, it uses microformats for adding semantic annotations to the HTML service documentation. Similar to SAWSDL, MicroWSMO has three types of annotations as illustrated in Figure 8: 1) `model`, which can be used on any hRESTS service property to point to appropriate semantic concepts; 2) `lifting`, and 3) `lowering`, which specify the mappings between semantic data and the underlying technical format such as XML. Therefore, MicroWSMO enables the semantic annotation of RESTful services basically in the same way as SAWSDL supports the annotation of Web services described by WSDL.

Since both MicroWSMO and SAWSDL can apply WSMO-Lite service semantics (an ontology described later in this chapter) it is important to note that REST-based services can be integrated with WSDL-based ones ([26], [131]). Therefore, tasks such as discovery, composition, and mediation can be performed independently from the underlying Web service technology.

Even though at first glance SA-REST's and hRESTS' idea seems to be fundamentally different from WSDL, their underlying models are closely related to WSDL's structure. In consequence, both SA-REST and hRESTS provide, just as WSDL, an RPC-oriented view of the service which does not really consider REST's resource orientation.

4.1.5 RESTdesc

RESTdesc [132] is a promising effort which is based on a fundamentally different realization. Instead of describing service interfaces in terms of resources or operations, it expresses functional descriptions of Web APIs in Notation3 [133], a data format extending RDF's data model by concepts such as variables. These functional descriptions are composed of preconditions which entail certain postconditions, such as the existence of an HTTP request. A client thus needs to express its goal in terms of postconditions. If the preconditions are fulfilled, it becomes possible to deduce an HTTP request that, when executed, results in the desired postconditions. It is worth noting that the HTTP request is part of the postconditions and not of the preconditions. This means that the data returned by a reasoner contains the HTTP request as if it would have been part of the input data. If several potential requests (or a chain of requests) are returned, it becomes difficult to interpret the data. This is aggravated by the fact that no tooling exists so far, not even (public) prototypes thereof.

RESTdesc's strength is the elegant description of the behavioral semantics of a service. The missing tooling, the unusual underlying data model and serialization format (Notation3 is not standardized and even within the Semantic Web community rarely used), and the dependency on semantic reasoners however makes it difficult to use RESTdesc in practice. We believe that a more gradual introduction to Semantic Web technologies is necessary to achieve widespread adoption.

4.2 Data Interchange Formats

Almost all current Web APIs use either XML or JSON as their data interchange format whereas for a long time RDF/XML was the only standardized choice for Semantic Web applications. RDF/XML has been first released in 1999 [61] at the peak of XML's hype and revised in 2004 [64]. Today, it is widely believed that RDF/XML significantly slowed the adoption of RDF and thus the whole vision of the Semantic Web. Its syntax is neither optimized for humans nor for machines nor does it allow the expression of all RDF data. In practice this means that it is generally very difficult to understand the data by just looking at the source code without further tooling. Similarly, standard XML technologies such as XPath [134], XQuery [135], or XSLT [136] are almost useless for working with RDF/XML because the same RDF graph can be serialized in many different ways. For a detailed analysis refer to Beckett's retrospective on the development of RDF/XML's revised syntax [137].

In contrast to the REST community which seems to be happy with XML and JSON (indicated by the fact that there have not been any notable efforts defining new data interchange formats), the Semantic Web community has been actively working on numerous proposals to replace RDF/XML. In 2011, the W3C started a new working group whose charter [138] included the serialization of Turtle, a W3C Team Submission [139], and to either extend it to support multiple graphs or to standardize a separate syntax doing so. Eventually, the working group decided to standardize a whole suite of new syntaxes: Turtle [140], TriG [141] (an extension of Turtle supporting multiple graphs) as well as N-Triples [142] and N-Quads [143] which could be classified as line-based counterparts of Turtle and TriG. Furthermore, the working group agreed to standardize JSON-LD [144], a serialization format based on JSON that follows a completely different approach.

In the following section, we will describe Turtle in more detail as it builds the foundation of all new RDF serialization formats. JSON-LD, which is one of the main contributions of this thesis, is described in more detail in section 5.3.

4.2.1 Turtle

In 2003, David Beckett proposed N-Triples Plus [145], a new textual, non-XML syntax for RDF based on the test case format N-Triples [146] defined by the RDF Core Working Group revising RDF/XML. As the name suggests, the syntax is, just as the 2004 revised definition of RDF [147], triple-centric which makes it much simpler to understand the serialized data. Furthermore, the simple syntactic constructs, make it easy to author such documents by hand. This can be best illustrated by an example. The snippet in Listing 2, which has been taken directly from

```xml
<?xml version="1.0"?>
<rdf:RDF xmlns:rdf="http://www.w3.org/1999/02/22-rdf-syntax-ns#"
         xmlns:dc="http://purl.org/dc/elements/1.1/"
         xmlns:ex="http://example.org/stuff/1.0/">
  <rdf:Description
      rdf:about="http://www.w3.org/TR/rdf-syntax-grammar"
      dc:title="RDF/XML Syntax Specification (Revised)">
    <ex:editor>
      <rdf:Description ex:fullName="Dave Beckett">
        <ex:homePage rdf:resource="http://purl.org/net/dajobe/" />
      </rdf:Description>
    </ex:editor>
  </rdf:Description>
</rdf:RDF>
```

Listing 2. An exemplary RDF/XML document [64]

```
@prefix rdf: <http://www.w3.org/1999/02/22-rdf-syntax-ns#> .
@prefix dc: <http://purl.org/dc/elements/1.1/> .
@prefix ex: <http://example.org/stuff/1.0/> .

<http://www.w3.org/TR/rdf-syntax-grammar>
    dc:title "RDF/XML Syntax Specification (Revised)" ;
    ex:editor [
          ex:fullname "Dave Beckett";
          ex:homePage <http://purl.org/net/dajobe/>
    ] .
```

Listing 3. The exemplary RDF/XML document from Listing 2 converted to Turtle

the RDX/XML specification [64], not only shows how verbose RDF/XML (still) is, but also that it is relatively difficult to author and read such documents without tooling support. In contrast, N-Triples Plus, which was later renamed to Turtle, feels much simpler and natural as shown in Listing 3.

Prefixes and the constructs allowing to group triples by subject or object or to express lists or blank nodes, eliminate a lot of N-Triples' verbosity. The downside of these features is the increased variability which makes parsing and processing more complex.

Turtle is seen as one of the most important efforts of the Semantic Web community. It finally provides a syntax which shows the simplicity of RDF's data model. Nevertheless, it is important to note that RDF's data model, which is based on triples, is alien to most developers. Developers typically think in terms of entities and thus an entity-centric format might be more suitable to increase the adoption of Semantic Web technologies outside the Semantic Web community. Furthermore, the fact that Turtle defines a new grammar means that custom lexers and parsers have to be build and developers cannot reuse their existing toolchains.

4.3 Vocabularies and Ontologies

Interface descriptions languages normally offer only syntactic descriptions of service surfaces. In practice, however, such syntactic descriptions are insufficient to enable the automation of tasks such as service discovery and composition. The information that an operation requires two strings and returns an integer does not offer any hint as to what the operation does. Thus, in order to solve this problem, research has been done to

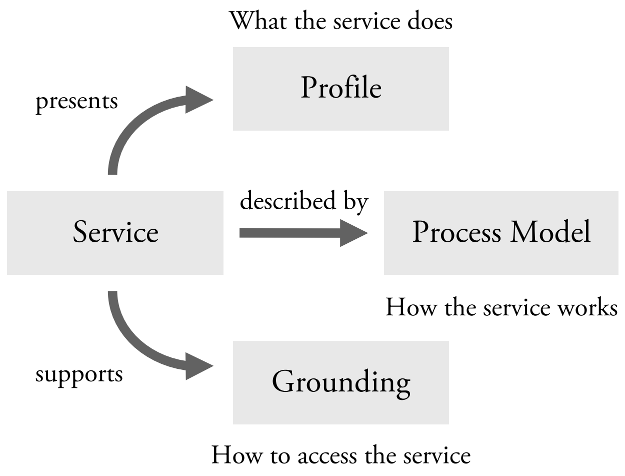


Figure 9. The OWL-S ontology

describe services also semantically. In this section we will give a brief overview of the most significant approaches.

4.3.1 OWL-S

OWL-S (Web Ontology Language for Web Services, formerly known as DAML-S) [148] is an upper ontology based on the W3C standard ontology OWL used to semantically annotate Web services. OWL-S consists of the following main upper ontologies as shown in Figure 9: 1) the *Service Profile* for advertising and discovering services; 2) the *Service (Process) Model*, which gives a detailed description of a service's operation and describes the composition (choreography and orchestration) of one or more services; and 3) the *Service Grounding*, which provides the required details about transport protocols to invoke the service (e.g. the binding between the logic-based service description and the service's WSDL description). Generally speaking, the Service Profile provides the information needed for an agent to discover a service, while the Service Model and Service Grounding provide enough information for an agent to make use of a service once found [148].

The main critique of OWL-S is its limited expressiveness of service descriptions in practice. Since it practically corresponds to OWL-DL, it allows only the description of static and deterministic aspects; it does not cover any notion of time and change, nor uncertainty. Furthermore, in contrast to WSDL, an OWL-S process cannot contain any number of completely unrelated operations [120], [149].

4.3.2 WSMO

Another approach to describe Web services semantically is the Web Service Modeling Ontology (WSMO) [150]—the outcome of work funded by numerous large European Union research projects. It defines a conceptual model and a formal language called WSML (Web Service Modeling Language) as well as a reference implementation of an execution environment (WSMX; Web Service Execution Environment) for the dynamic discovery, selection, mediation, invocation, and interoperation of Semantic Web services based on the WSMO ontology.

WSMO offers four top-level notions to describe the different aspects of Web services as shown in Figure 10: 1) *Ontologies* that define the formalized domain knowledge; 2) *Goals*, which specify objectives that a client might have when consulting a Web service; 3) *Service Descriptions* for describing functional, non-functional and behavioral aspects of a Web service; and 4) *Mediators* for enabling interoperability and handling heterogeneity between all these components at data (mediation of data structures) and process level (mediation between heterogeneous communication patterns) to allow loose coupling between services, goals, and ontologies.

In contrast to most other description formalisms, WSMO propagates a goal-based approach for SWS. It is particularly designed to allow the search for Web services by formulating the queries in terms of goals. So

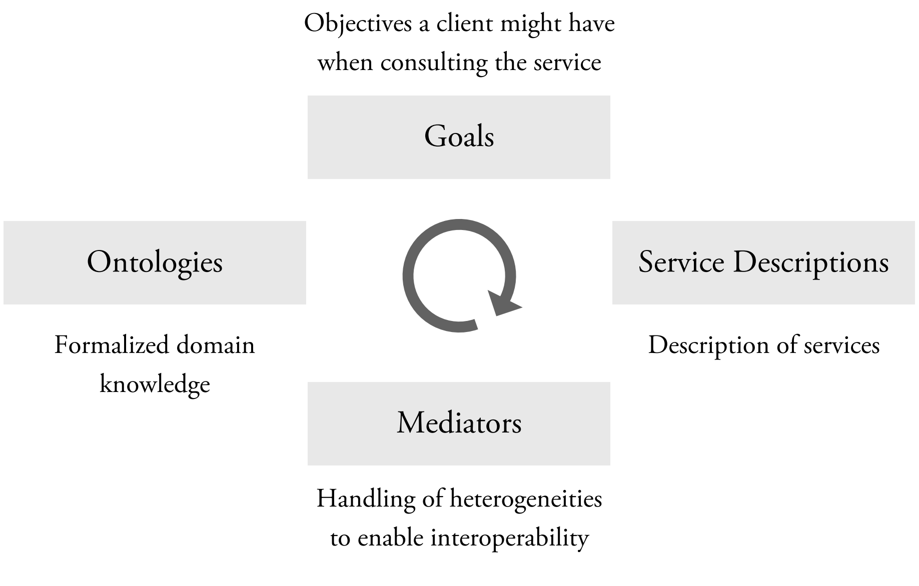

Objectives a client might have when consulting the service
Goals
Ontologies
Service Descriptions
Formalized domain knowledge
Description of services
Mediators
Handling of heterogeneities to enable interoperability

Figure 10. The WSMO ontology

the task of the system is to automatically find and execute Web services which satisfy the client's goal. This goes beyond the OWL-S idea whose principal aim is to describe the service's offers and needs.

One of the main critiques of WMO is that its development has been done in isolation of existing W3C standards. This raised serious concerns by the W3C which were expressed in the official response to the WSMO submission in 2005 [151]. To address those issues, a lightweight version called WSMO-Lite that is presented in the next section has been created. Another critique is that guidelines for developing mediators, which seem to be the essential contribution of WSMO in concrete terms, are missing.

4.3.3 WSMO-Lite

SAWSDL does not specify a language for representing the semantic models but defines how to add semantic annotations to various parts of WSDL or XML Schema documents. WSMO-Lite [152] was created as a lightweight service ontology to fill SAWSDL annotations with concrete service semantics to allow bottom-up modeling of services. It adopts the WSMO model and makes its semantics lighter. The biggest difference to WSMO is that WSMO-Lite treats mediators as infrastructure elements and specifications for user goals as dependent on the particular discovery mechanism used. In contrast, WSMO defines formal user goals and mediators. Furthermore, WSMO-Lite defines the behavior semantics only

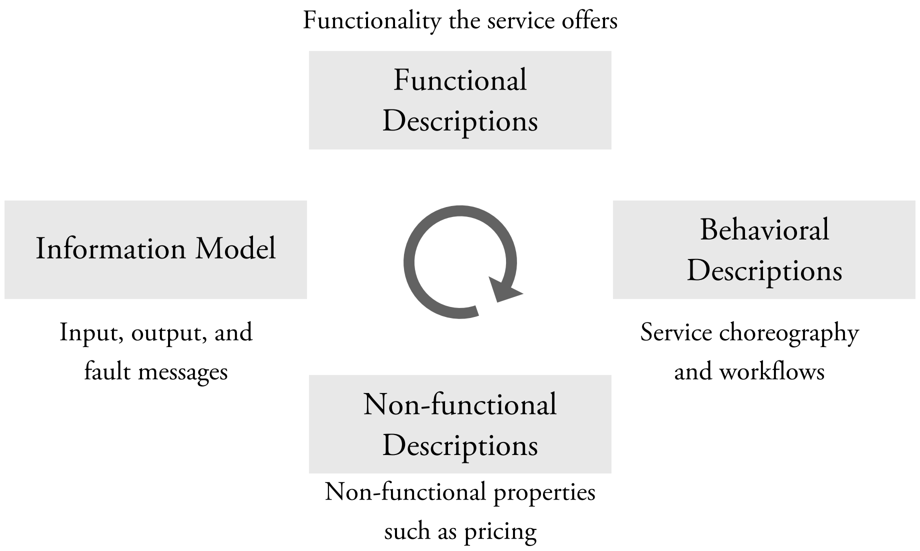

Functionality the service offers
Functional Descriptions
Information Model
Behavioral Descriptions
Input, output, and fault messages
Service choreography and workflows
Non-functional Descriptions
Non-functional properties such as pricing

Figure 11. The WSMO-Lite ontology

implicitly. WSMO-Lite also does not exclusively use WSML, as WSMO does, but allows the use of any ontology language with an RDF syntax.

WSMO-Lite describes the following four aspects of a Web service: 1) the *Information Model*, which defines the data model for input, output, and fault messages; 2) the *Functional Semantics*, which define the functionality, which the service offers; 3) the *Behavioral Semantics*, which define how a client has to talk to the service; and 4) the *Non-functional Semantics*, which define non-functional properties such as quality of service or price. A graphical representation of the ontology is shown in Figure 11.

A major advantage of the WSMO-Lite approach is that it is not bound to a particular service description format such as WSDL. As a result WSMO-Lite can be used to integrate approaches like, e.g., hRESTS and MicroWSMO with traditional WSDL-based service descriptions.

4.3.4 EXPRESS

Given that EXPRESS [153] follows a completely different strategy than the other approaches mentioned so far, we believe it is another interesting approach to look at despite the fact that we are not aware of a single usage in the wild. Instead of a domain ontology and a separate semantic description of a service, EXPRESS only requires the definition of a domain ontology in OWL [66], i.e., the formal description of the concepts and their relationship in the API's application domain. An EXPRESS "deployment engine" then analyzes that domain ontology and creates a URI space for the found classes, instances, and properties. Finally, the developer decides which HTTP methods are permitted for the various created resources in order to define the supported functionality. It is worth noting, that EXPRESS maps the HTTP methods to the CRUD operations (create, read, update, and delete) and thus, the functionality of APIs created with EXPRESS is limited to the CRUD operations. This reduces the complexity but also the possible applications of EXPRESS. Often, simple CRUD-style functionality is not enough.

Since EXPRESS follows a top-down approach in which concrete services are automatically created out of semantic descriptions, it represents a disruptive approach that cannot be used to upgrade existing services and thus makes it impossible to build upon existing infrastructure investments. For simple APIs it could, however, be an interesting approach to consider presuming that powerful tooling would be available, which is not the case at the time of this writing.

4.3.5 Linked Data Platform

Based on a member submission by IBM, the W3C decided in 2012 to start a working group with the aim to "produce a W3C Recommendation for HTTP-based (RESTful) application integration patterns using read/write Linked Data" [154]. The Linked Data Platform (LDP) vocabulary [155] defines concepts such as resources and collections but it is misses any notion of operations. Effectively this means that, at least at the current stage, the Linked Data Platform does not go beyond defining a standardized CRUD interface to manage resources in collections. It could thus be characterized as an RDF version of the Atom Publishing Protocol [47] as the interaction models are almost identical.

Collections can be used to store (more or less) opaque RDF documents. LDP has neither built-in support for the semantic description of operations other than CRUD nor does it allow the description of supported properties, classes, etc. The only way for a client to find out which properties are supported is to POST an RDF document to a collection in order to create a new resource. That resource then has to be inspected to verify that all the data has been stored and no properties have been discarded.

Given these limitations, it is questionable whether mainstream Web developers will see enough compelling reasons to adopt the approach proposed by the Linked Data Platform working group. The same functionality can be achieved with much simpler, proven approaches such as the Atom Publishing Protocol [47].

4.4 Domain Application Protocols

All the approaches mentioned so far try to be as general as possible in order to be usable for a wide range of application domains. In this section, we will present a number of specialized solutions tailored for very specific application domains. In contrast to the previously mentioned approaches, the solutions presented in this section were able to achieve some adoption across the Web. These approaches have in common that they use specialized media types (even though not all of them have been officially registered) to define message semantics and processing models for specific use cases. In other words, they represent *domain application protocols* [46].

4.4.1 Atom

Atom consists of two related standards, the Atom Syndication Format [48] and the Atom Publishing Protocol [47] (also known as AtomPub or APP). The Atom Syndication Format is an XML-based format to syndicate content in the form of so called Web feeds or news feeds. The Atom Publishing Protocol is an application-level protocol for publishing, editing, and deleting feed entries and associated media resources.

The Atom Syndication Format consists of two kinds of documents: Atom Feed Documents and Atom Entry Documents. An Atom Feed Document is, as the name suggests, the representation of an Atom feed. It contains metadata about the feed and some or all of the entries associated with the feed. An Atom Entry Document describes exactly one feed item outside the context of an Atom feed. It is worth mentioning that Atom documents must be well-formed XML but are not required to be valid XML because the specification does not include a Document Type Definition (DTD) for them. Atom is designed to be an extensible format and so foreign markup (markup which is not part of the Atom vocabulary) is allowed almost anywhere in an Atom document.

The Atom Publishing Protocol describes how a feed can be manipulated by a client. It defines, just as the Syndication Format, two kinds of documents: Category Documents and Service Documents. Category Documents are used to hold the list of Atom categories as defined in the Atom Syndication format. Those category lists are used to describe the categories that can be applied to the members of a Collection (i.e. the entries of an Atom Feed Document). The Service Document describes the location and capabilities of one or more Collections which are grouped into Workspaces. That information is needed by clients for authoring to commence.

Both the Atom Syndication Format as well as the Atom Publishing Protocol are fully based on the REST architectural style and thus integrate very well in the Web's architecture. In fact, the Atom Publishing Protocol is often cited as the poster child of RESTful service design. Its extensible design led to the adoption of AtomPub for the implementation of various kinds of Web services. The most prominent examples have been early versions of Google's Data Protocol (GData) [156] and Microsoft's Open Data Protocol (OData) [157]. They used Atom's extensibility to implement APIs for their services but unfortunately such an approach is not always feasible or desirable. It is also just a solution for the description of the service's interface; the problem of describing the exchanged data, i.e., the feed entries, still remains unresolved. Sometimes this approach also yields strange results, e.g. when a service provider just serializes an Atom feed into a JSON representation. The JSON serialization of Google's Data Protocol [156] is one of those inglorious examples. At least its subsidiary YouTube recognized the problem and is now offering an alternative JSON serialization [158].

4.4.2 OpenSearch

OpenSearch [159] was developed by A9, an Amazon.com subsidiary, and was first unveiled in 2005. It is a collection of simple formats that allow the description of search engines' interfaces as well as the publishing of search results in a format suitable for syndication and aggregation. OpenSearch allows clients such as Web browsers to invoke search queries and process the responses. By now all major Web browsers support OpenSearch and use it to add new search engines to the browser's search bar. This way the user can invoke a query directly from the browser without first having to load the search engine's homepage.

OpenSearch consists of the following four formats: 1) the description document, 2) the URL template syntax, 3) the response elements, and 4) the Query element. The OpenSearch description document describes the interface of a search engine in the form of a simple XML document. It may also contain some metadata such as the name of the search engine and its developer. The URL template syntax represents a parameterized form of the URL by which a search engine is queried. Simply speaking it describes the used GET parameters to invoke a query. An example of such a template looks as follows:

```
http://example.com/search?q={searchTerms}
```

All parameters are enclosed in curly braces and are by default considered to be part of the OpenSearch template namespace. By using the XML namespace prefix convention it is possible to add new parameter names, which enables extensibility. The OpenSearch response elements are used by search engines to augment existing XML formats such as Atom and RSS with search-related metadata. Finally, the OpenSearch Query element can be used to define specific search requests that can be performed by a search client. The Query element attributes correspond to the search parameters in a URL template. One use case is, e.g., the definition of related queries in a search result element.

4.4.3 oEmbed

oEmbed [160] provides a simple interface that allows a Web site to display embedded content (such as photos or videos) when a user posts a link to that resource without having to parse the resource directly. This makes it possible to embed, e.g., a YouTube video on a web page without having to extract the YouTube video player from the HTML page the user referenced.

The interface defined by oEmbed is trivial. A provider specifies one or more URI scheme and API endpoint pairs which a consumer then uses to issue HTTP requests to get the necessary information to embed a specific resource. The aforementioned URI scheme describes which URIs (wildcards are supported) may have an embedded representation, i.e., for which URIs the associated API endpoint might be used by a consumer to lookup the structured data used to embed the representation. The consumer then issues an HTTP GET request to the API endpoint with the URI of the representation it would like to embed along with the optional maximum width and height of the embedded resource as query parameters. It might also specify in which format it would like to get the response; possible formats are JSON and XML. Finally, the provider replies with a response containing structured data such as, among others, a title, the author's name, a thumbnail of the referenced image or the

HTML code needed to embed a video player. The client can then use this information to display the resource referenced by the user.

oEmbed is a nice example of a clear use case that lead to a clear and simple specification and thus resulted in wide adoption. Among others, YouTube, Flickr, Hulu, Slideshare, and Vimeo act as providers and there are plugins and libraries which add support for oEmbed to almost every blogging and content management system available.

4.5 Hyperlinks and Namespaces in JSON

In contrast to XML, JSON, the JavaScript Object Notation, was specifically designed as a lightweight, language-independent data-interchange format that is easy to parse and generate. Often it is thus considered to be simpler than XML—but this simplicity comes at a price. JSON has neither native support for hypermedia nor does it support namespaces or semantic annotations. There have been various proposals to solve these shortcomings; all of them have in common that they specify a set of keywords to express certain aspects such as hyperlinks.

The most prominent examples trying to add hypermedia support to JSON are probably JSON Schema [161] and its trimmed down counterpart JSON Reference [162]. Both define a special keyword `$ref` to denote a hyperlink. While, as the name suggests, JSON Schema puts that type information in a schema describing the document, JSON Reference uses the `$ref` keyword directly within the document. It can thus be seen as a static serialization of the same type but it lacks support for semantic annotation to describe its relation to the current document (which is possible with JSON Schema). Two related solutions that address this issue are HAL and Collection+JSON, but in contrast to the previously mentioned approaches which augment JSON, they represent a new media type on their own.

HAL [163] uses the `_links` keyword instead of `$ref` but, instead of setting its value directly to the link's target, it sets its value to an object whose keys are the link relations and whose values are the link targets. HAL has also support to embed external resources within a representation. Often this is important as it allows applications to greatly decrease the number of required HTTP requests.

Collection+JSON [164] is basically a JSON version of the Atom protocol suite to manage simple lists of entities. This media type not only specifies how links (which can be templated) are represented but also how HTTP can be used to manipulate the various representations.

Similar to these proposals, but with a different goal in mind, various approaches have been presented to add semantic annotations or namespace support to JSON. These two aspects can be considered to be roughly the same as the idea of semantic annotations to define the semantics of a concept in a special namespace to avoid collisions when the same terms are reused in different documents. The different proposals can be classified into two groups based on whether namespaces are supposed to be dereferenceable or not. In the first group, where namespaces are just used to avoid collisions and are thus not expected to be dereferenceable, often DNS-style names such as `com.example.projects.namespacesInJSON` are used [165]; the syntactic differences of the proposals are negligible. The second group of approaches assumes namespaces to be dereferenceable to be able to retrieve further information about them. In other words, they are based on the idea of Linked Data and, as such, they are mostly trying to create a JSON serialization format for RDF. Most of those approaches thus also offer other functionality such as data typing or string internationalization.

As part of the effort to standardize a JSON serialization format for RDF, the RDF Working group has already compared most of the existing approaches [166]; therefore we would like to refer the interested reader to that document for a detailed review of the various proposed solutions. Summarized, it can be said that most of the approaches create a new media type with specific processing models. The main difference is whether they are triple- or entity-centric and the degree by which they rely on microsyntaxes. This determines how familiar a representation looks to a JSON developer; an important aspect for the acceptance of such a format. Unfortunately, most proposed solutions fall short in this respect.

4.6 Discussion

As we have shown in this chapter, multiple approaches trying to describe RESTful services in a machine readable manner have been proposed over the years. Similarly, numerous attempts have been made to extend JSON with hypermedia controls and namespacing support to make it more suitable for the creation of self-descriptive messages. This clearly shows a desire to formalize the development and description of RESTful services but, unfortunately, none of the proposed solutions managed to achieve noticeable adoption. The only endeavors blessed with some uptake were solutions targeting small, well-defined use cases such as the description of search engine interfaces or content syndication.

We argue that the lack of acceptance of those approaches stems from the fact that they do not provide any imminent incentive and thus experience a classic chicken-and-egg problem. No services are being formally described because there are no applications making use of that information and no applications are developed because there are no such service descriptions. Furthermore, a lot of the presented proposals are complex, heavyweight solutions. Often their functioning resembles the flawed RPC-model which is a problem especially with regard to RESTful services that follow a fundamentally different architecture. Some semantic approaches introduce new languages and most of them promote top-down modeling, i.e., semantics first. RESTful Web APIs, however, are often driven by bottom-up design. Analyzing the current state of the art and taking into account our experience in creating Web services and working with Web developers, we were able to distill a number of aspects which we deem important to solve the issues described in Chapter 3.

A critical, yet often neglected feature is the support for hypermedia. Web APIs need to the able to convey valid state transitions at runtime instead of requiring developers to hardcode them into their clients. Thus, the description of URL structures or templates generally provides little advantages. Developers typically find it much simpler to hardcode against those patterns instead of processing them at runtime. This leads to a tight coupling and hinders the evolvability of Web APIs as URLs cannot be changed without breaking clients. Thus, a number of approaches presented in this chapter define constructs to serialize hyperlinks in, e.g., JSON. This is definitely a step in the right direction but doing just that is certainly not enough. It solves only part of the problem and can just as well be achieved by simpler, standardized mechanisms such as the HTTP Link header [57]. So, while support for hypermedia is without a doubt critical, features such as namespacing are important as well. Without namespacing, it becomes extremely difficult to reuse concepts across different Web APIs. This leads to the current situation in which every Web API is effectively a snowflake, i.e., every Web API is unique and requires documentation to be used. By supporting namespacing, it becomes possible to reuse and mix concepts from various sources in a single message. At the same time, the message becomes self-descriptive, which is one of REST's fundamental constraints. The problem with most formats or conventions supporting namespacing is that the messages become overly verbose. Furthermore, our experience tells us that average Web developers do not want to deal with namespaces. Thus, we believe that it would be better to let experts define "namespace bundles" similar to profiles as described in section 2.2, which would allow developers to work with the concepts as if they would all belong to a single namespace.

An interesting observation in analyzing the state of the art is that most proposed solutions are either too simplistic or overly complex. It is hard to find approaches in the middle ground of these two extremes. Solutions in the first camp typically confine themselves to CRUD-style APIs. While it is true that almost all functionality can be implemented by such interfaces, the semantics the CRUD operations offer are too weak in systems that are not centrally coordinated. Clients also need to know what the consequences of the various operations are. It is not enough to know how to create an entity, but it is also necessary to know what implications such a creation has. For instance, it is crucial to know whether the creation of an order entity results in the delivery of goods or not. Solutions in the second camp typically become overly complex by features without clear usage scenarios in practice.

A lot of proposed solutions from the Semantic Web community e.g. included descriptions of non-functional characteristics of a service to support matchmaking. This seems like a sensible decision per se but in practice it generally becomes too complex to describe abstract characteristics of multiple services in a uniform way. Often, the decision about which service to use is not based on objective characteristics but on subjective aspects such as the API publisher's reputation, relationships between various companies, or changes in competitive conditions. We therefore believe that it is essential to find to find the right trade-off between complexity and expressivity. In this thesis, we therefore concentrate on the functional aspects of Web APIs but try to keep the solution extensible enough to support more complex functionality in the future. Furthermore, we believe that it is imperative to allow the gradual introduction of new features such as namespacing or hypermedia support as well as the description of service interfaces and semantics. It should be possible to update existing services with minimal changes to the service itself. This will ensure that investments in existing systems can be leveraged and that developers do not have to change their toolchains due to disruptively different base technologies.

Chapter 5

Bridging the Gap between REST and Linked Data

Developers have to deal with a plethora of heterogeneous data formats and service interfaces for which little to no tooling support is available when using RESTful Web APIs. Similarly, when implementing services developers struggle with a number of difficult design decisions. Thus, most Web APIs are like snowflakes, i.e., similar yet not alike. Even though the differences are in general quite subtle, they render code reuse impossible to a large extent. As we have seen in Chapter 3, the usage of proprietary data formats, the reliance on static contracts written in natural language, and the fact that clients are often developed by the same team in lockstep with the service itself are the main reasons for this situation. Simply speaking, the goal of this thesis is to break this vicious cycle.

As we have seen in the previous chapters, various issues have to be solved to achieve this ambitious goal but the most important building block is a data interchange format or a description language supporting the creation of self-descriptive messages. Moreover, that format or language either needs to have built-in support for hypermedia or provide extension points to add it separately—for instance in the form of a vocabulary. It is also important to keep data integration and reuse in mind when designing a solution to these issues as it is the underlying problem to be solved in a lot of Web API usage scenarios. This is a task RDF has proven to be very apt at as its simple data model often represents the least common denominator to which various different other data models can be easily mapped to.

Even though RDF/XML, the only standardized standalone serialization format for RDF, is widely disliked and new formats, such as Turtle, have not been optimized for Web APIs, our hypothesis is that it should be feasible to standardize and streamline the development and usage of truly RESTful Web APIs by combining technologies from both the world of Web APIs and the Semantic Web. We expect that such standardization would, in the first place, result in higher productivity due to the ability to create generic tools and libraries, and to reuse already existing RDF vocabularies. Subsequently, it could lead to much more sophisticated applications. It could also potentially foster the creation of tools at higher levels of abstraction which could, hopefully, even allow non-technical experts to create solutions fulfilling their situational needs.

Based on this hypothesis, the study and analysis of the state of the art, and our experience in creating and using RESTful services, we began with experiments to design a solution for the problems identified in section 3. We first concentrated on the data interchange as it is the most visible and concrete aspect of a Web API. This led to the development of a generic data interchange format, which we complemented with a lightweight vocabulary defining the semantics of a number of concepts needed in most Web APIs. This iterative process resulted in four original contributions which are described in the following sections. JSON-LD, the final data interchange format has become an official and well-accepted W3C standard. Hydra, the vocabulary defining important concepts for RESTful Web APIs, has also been well received and has led to the establishment of a W3C Community Group working on its standardization and future extensions.

This chapter is based on previous work that has been published in [25], [27], [95]–[99], [144], [167]–[170].

5.1 SAPS

Numerous services are implemented by exposing a simple CRUD interface to manage entities of various types. All the interaction with such an API happens through the manipulation of specific entities. An e-commerce API, e.g., might enable clients to order goods by creating an order entity which references a number of article entities. While a CRUD interface is rarely enough to build sophisticated services in practice, it is a common pattern and often the least common denominator of different Web APIs. To prove the principal viability of our hypothesis that it should be possible to standardize and streamline the development of Web APIs by combining Semantic Web technologies with technologies used in RESTful services, we started experimenting with proven, standardized technologies.

The Atom Syndication Format [48] and its publishing protocol [47] are often cited as poster children of RESTful service design. Along with OpenSearch [159] they are among the few approaches that have been widely adopted. Well aware of the fact that XML was not designed for interchange of structured data but to markup mixed content and that Atom often enforces a too rigid structure, we nevertheless chose them as the base technologies for our first experiments which aimed to create a minimum viable product [171] combining Semantic Web technologies with technologies used in Web APIs. This finally lead to the development of SAPS [27] which is described in this section.

Building a solution by combining different proven technologies has the advantage of readily available, mature tooling instead of having to start from scratch. This is quite beneficial in terms of agility as the focus can be put on the high-level concepts and ideas instead of having to spend time on implementation details.

5.1.1 Basic Concepts and Principles

The basic idea of *SAPS (Semantic AtomPub-based Services)* is to combine the Atom Syndication Format [48], the Atom Publishing Protocol [47], and the OpenSearch [159] format to provide a framework for the exchange and manipulation of entities. The entities themselves, i.e., the data the client is mainly interested in, are described by semantically annotated schemas. The result is a mostly standardized technology stack as illustrated in Figure 12. If the payload is encoded in XML, even the semantic layer at the top can be realized by using standardized technologies such as XML Schema [89] and SAWSDL [119]. The sole purpose of SAPS is to define how to integrate these technologies.

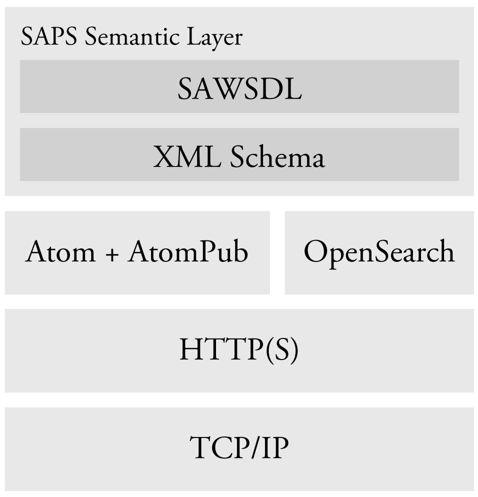


Figure 12. The SAPS layer cake

The main entry point of a Web API built with SAPS is represented by an Atom service document [47] consisting of a number of collections, i.e., Atom feeds. This allows a client to add new entities or to discover already existing entries by browsing the various collections. Without further knowledge, however, it is difficult for a machine client to figure out to which collection an entry should be added and which collection should be browsed to find a specific entity respectively. Humans typically rely on the collection's name or categories for this purpose but the natural language description is semantically too weak for a machine to work with directly. SAPS, therefore leverages the fact that Atom does not assign any meaning to the content of the `app:categories` element and reuses it to convey machine-readable hints about the data in a collection.

The category's scheme is set to an ontology's namespace prefix and the term to a specific concept in that ontology. Effectively, the concept's IRI is split into a prefix and a suffix similar to the usage of compact URIs (CURIEs) [172] in various RDF serialization formats or XML. This informs a client what kind of data it may find in a specific collection. Unfortunately, it does not convey enough information to add new entities, as it describes neither the expected syntactic structure nor the properties of such an entity (in RDF the properties define their relationship to a class and not vice versa).

The Atom Publishing Protocol normally addresses this by defining the acceptable payload formats in terms of media types but unfortunately this is normally not specific enough in practice—at least not for Web APIs.

The preferred media types for Web APIs are typically too general and, most of the time, minting new proprietary media types is neither practical nor desirable. Both XML (`application/xml`) and JSON (`application/json`), e.g., are not concrete enough to allow the automatic construction of messages. SAPS solves this by using semantically annotated schemas to define both the syntactic structure of the messages as well as the semantics of the various elements. Via the newly introduced attribute `saps:schema` the media type is augmented with a schema that can be used to generate payloads according to the requirements of the server. It is important to note that, since the schema is semantically annotated, the data model used on the client can be automatically mapped to the syntactic structure required by the server. The coupling, however, takes places at the semantic layer instead of the syntactic layer which improves the evolvability and reusability of the system as semantic concepts change much less frequently than the syntactic structures to serialize them.

Just as with Atom itself, clients of SAPS-based services typically interact with feed entries indirectly, i.e., not by dereferencing each entry's URL to retrieve its representation, but by retrieving the feed in which the entries' representations are embedded. This is much more efficient when retrieving numerous entries but the downside is that the metadata that could be found in the HTTP headers when retrieving each feed entry separately is lost. To partially mitigate this limitation SAPS introduces the `saps:etag` attribute.

SAPS's `etag` attribute is equivalent to HTTP's `ETag` header [9], a token identifying the current version of a resource representation. The `etag` attribute can be used to specify the ETag of an entry embedded in a feed so that the need to separately dereference its URL just to get the ETag is eliminated. This improves the efficiency by enabling conditional retrievals (`GET` using the `If-None-Match` HTTP header) and adds support for optimistic concurrency control when manipulating or deleting entries (e.g., `PUT` or `DELETE` requests using HTTP's `If-Match` header).

This simple model offers a complete CRUD interface to the data exposed by a Web API allowing entities to be created, retrieved, updated, and deleted by standardized technologies. While this covers a lot of the functionality typically needed in Web APIs, it ignores a common use case, namely the query of data. With the interface described so far, a client has to iterate through a collection in order to find a specific entry. In most cases, however, it would be much more efficient to query the API to find a specific entity directly. SAPS addresses this use case by describing a service's search interface(s) with the OpenSearch format. Given that Atom was designed as an extensible format, it is trivial to integrate OpenSearch. The OpenSearch document describing the search interface can either be directly embedded in a Atom service document or feed as "foreign markup" [48] or referenced by using Atom's `link` tag with the standardized `search` link relation. The OpenSearch document itself specifies a URL template which is expanded to a URL by populating it with the concrete query criteria. In SAPS, the query criteria are, most of the time, expressed by using semantic concepts from a vocabulary. The OpenSearch document defines a prefix (an XML namespace) identifying the vocabulary so that the URL template's variables can be expressed in the form of CURIEs.

Similar to the search functionality, interaction models that do not fit nicely in the collections/items structure can be built by including links with specific link relations in both feeds and feed entries. Obviously a client needs to know how to process the link relations. SAPS does not define any mechanism to describe that, but just as Atom, relies on external documentation that developers can use to implement their clients.

Since SAPS is based on Atom, it strictly follows its specification and the use of most Atom elements is self-explanatory. Some elements, however, require further clarification in the context of SAPS. For instance, in most of the cases it is not obvious how `atom:title`, `atom:author`, and `atom:summary` should be used. SAPS takes a pragmatic approach for these fields: the `title` element is used to create a human-readable representation of the item (e.g., if a product is represented its name and price could be used), the same applies for the `summary` element where required. The `author` element is a bit trickier; most of the time it will either be empty or set by the server to some constant value such as the API's name.

5.1.2 Illustrative Example

In order to give a better understanding of the basic concepts and principles explained in the previous section, we will demonstrate how an illustrative example can be realized using SAPS in this section. The example

```
<?xml version="1.0" encoding="utf-8"?>
<service xmlns="http://www.w3.org/2007/app"
        xmlns:atom="http://www.w3.org/2005/Atom"
        xmlns:saps="http://www.purl.org/saps">
 <workspace>
   <atom:title>Awesome Festivals API</atom:title>
   <collection href="/festivals/" >
     <atom:title>Upcoming Festivals</atom:title>
     <accept />
     <categories fixed="yes">
       <atom:category scheme="http://schema.org/" term="Festival" />
     </categories>
    <atom:link rel="search"
      type="application/opensearchdescription+xml"
      href="/queries/festival.xml"/>
   </collection>
   <collection href="/users/4812/orders/" >
     <atom:title>Orders</atom:title>
     <accept saps:schema="/schemas/purchase-order.xsd">
       application/xml
     </accept>
     <categories fixed="yes">
       <atom:category scheme="http://schema.org/" term="Order" />
     </categories>
   </collection>
 </workspace>
</service>
```

Listing 4. The Atom service document representing the exemplary API's entry point

implements the Web API of an imaginary website that lets users find festivals and buy tickets to attend them. Thus, users should able to find upcoming festivals, find out which artists perform at that festival, and get information about the available tickets. Furthermore, it should be possible for users to buy tickets and to see all the orders they made.

The first step when implementing the Web API is to define the resources that are needed to represent the application domain's data. By analyzing the user stories described in the example we are able to extract the following resources: festivals, performers (artists), tickets, orders, and, perhaps, payments to complete orders. These resources are accessed by different actors, e.g., users and administrators. It is obvious that the actors have different privileges and need to authenticate themselves to the system, but for the sake of simplicity we will ignore authentication and authorization issues.

Just like for a normal website we start by building the service's "homepage". In SAPS, this is done by creating an Atom service document enumerating the available collections and referencing one or more OpenSearch description documents defining the search interface. In this example, the service's homepage includes collections for the upcoming festivals and the user's orders. Neither performers, nor tickets, nor payments are included directly on the homepage because there is no imminent use case justifying it. Of course it would be possible to, e.g., also include the collection of performers directly on the homepage to make it easier for users to find all festival where a specific artist performs but this decision is at the sole discretion of the API publisher.

In the Atom service document in Listing 4, the festival collection has an empty `accept` element which specifies that the festival collection does not support the creation of new entries—at least not for the currently authenticated user. The category of the collection is set to Schema.org's [85] `Festival` class so that clients are able to understand the meaning, i.e., the semantics, of this collection. These categories could also be used to store information about the behavior as well as non-functional descriptions of the service.

Furthermore, the festival collection contains a link to an OpenSearch description document defining the interface to search for festivals either by search terms (full-text search) or by the date it begins. As shown in the OpenSearch description in Listing 5, the full-text query parameter is represented by OpenSearch's `searchTerms` variable whereas the date is linked to Schema.org's `startDate` property. This conveys the semantics of the two parameters to a client, which is then able to replace them with concrete values.

```
<?xml version="1.0" encoding="UTF-8"?>
<OpenSearchDescription xmlns="http://a9.com/-/spec/opensearch/1.1/">
  <ShortName>Festival Search</ShortName>
  <Description>Search for festivals</Description>
  <Url xmlns:schema="http://schema.org/"
    type="application/atom+xml;type=feed"
    template="http://example.com/festivals/?q=
      {searchTerms?}&amp;date={schema:startDate?}" />
</OpenSearchDescription>
```

Listing 5. An OpenSearch document describing the query interface of the festivals collection

The orders collection is, in contrast to the festivals collection, writable. It accepts new orders in the form of XML documents with the media type `application/xml` complying with the `purchase-order.xsd` XML schema. Just as for the festivals collection, the category of the orders collection is set to a concept in a vocabulary to convey the semantics of the collection's items.

With this information from the Atom service document, a client is now able to search for a festival and create a purchase order to buy a ticket. As described by the OpenSearch description document, it can, e.g., search

```
<?xml version="1.0" encoding="utf-8"?>
<feed xmlns="http://www.w3.org/2005/Atom"
 xmlns:opensearch="http://a9.com/-/spec/opensearch/1.1/">
  <title type="text">Search for "Ultra"</title>
  <updated>2013-11-14T12:29:29Z</updated>
  <author><name>Awesome Festivals API</name></author>
  <link rel="search" type="application/opensearchdescription+xml"
    href="/queries/festival.xml"/>
  <entry>
    <title>Ultra Music Festival</title>
    <summary>Outdoor electronic music festival</summary>
    <id>tag:example.org,2014:ultra-miami</id>
    <link rel="alternate" type="text/html"
       href="http://example.com/festivals/2014/ultra-miami.html" />
    <link rel="alternate" type="application/xml"
       href="http://example.com/festivals/umf84705.xml"/>
    <updated>2013-08-30T12:29:29Z</updated>
    <published>2013-05-13T08:29:29-04:00</published>
    <content type="application/xml"
      <festival xmlns="http://example.com/ns/festival/"
            xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
            xsi:schemaLocation="http://example.com/ns/festival
            http://example.com/schemas/festival.xsd">
        <id>http://example.com/festivals/umf84705</id>
        <label>Ultra Music Festival</label>
        <description>...</description>
        <from>2014-03-28</from>
        <till>2014-03-30</till>
        <performers>
          ...
        </performers>
        <tickets>
          <ticket>
            <label>General Admission</label>
            <sku>umf84705-165</sku>
            <price>399.95</price>
          </ticket>
        </tickets>
      </festival>
    </content>
  </entry>
</feed>
```

Listing 6. The result of querying the festivals collection

for a festival containing the term “Ultra” by issuing an HTTP `GET` request on `http://example.com/festivals/?q=Ultra`. The server responds with an Atom feed containing the search results as show in Listing 6.

The XML schema associated with the returned festival entity is defined is shown in Listing 7 and describes not only the syntactic structure but also

```
<?xml version="1.0" encoding="utf-8"?>
<xs:schema xmlns="http://example.com/ns/festival/"
           xmlns:xs="http://www.w3.org/2001/XMLSchema"
           xmlns:sawsdl="http://www.w3.org/ns/sawsdl"
           targetNamespace="http://example.com/ns/festival/"
           elementFormDefault="qualified" >
  <xs:element name="festival" type="festivalType" />
  <xs:complexType name="festivalType"
      sawsdl:modelReference="http://schema.org/Festival">
    <xs:sequence>
      <xs:element name="id" type="xs:anyURI" />
      <xs:element name="label" type="xs:string"
          sawsdl:modelReference="http://schema.org/name" />
      <xs:element name="description" type="xs:string"
          sawsdl:modelReference="http://schema.org/description" />
      <xs:element name="from" type="xs:date"
          sawsdl:modelReference="http://schema.org/startDate" />
      <xs:element name="till" type="xs:date"
          sawsdl:modelReference="http://schema.org/endDate" />
      <xs:element name="performers"
          sawsdl:modelReference="http://schema.org/performer">
        ...
      </xs:element>
      <xs:element name="tickets"
          sawsdl:modelReference="http://schema.org/offer">
        <xs:complexType>
          <xs:sequence>
            <xs:element name="ticket" type="ticketType" />
          </xs:sequence>
        </xs:complexType>
      </xs:element>
    </xs:sequence>
  </xs:complexType>
  <xs:complexType name="ticketType"
      sawsdl:modelReference="http://schema.org/Offer">
    <xs:sequence>
      <xs:element name="label" type="xs:string"
          sawsdl:modelReference="http://schema.org/name" />
      <xs:element name="sku" type="xs:string"
          sawsdl:modelReference="http://schema.org/sku" />
      <xs:element name="price" type="xs:decimal"
          sawsdl:modelReference="http://schema.org/price" />
    </xs:sequence>
  </xs:complexType>
</xs:schema>
```

Listing 7. The XML schema describing a festival

the mapping of the data to Schema.org using SAWSDL's `modelReference` attribute. In this case the semantic annotation is trivial as there is a direct mapping from the syntactic elements to the concepts in Schema.org. More complex mappings often need to leverage SAWSDL's `liftingSchemaMapping` and `loweringSchemaMapping` to describe the transformation using technologies such as XSLT [136], XQuery [135], or SPARQL [71].

Due to the SAWSDL annotations in both the festival and order XML schemas, the client is able to extract the SKU of the ticket and create a purchase order by sending an order XML document to the orders collection. Obviously the client has to have knowledge about the desired quantities and acceptable price ranges but that is part of the client's business logic and thus beyond the scope of SAPS. Finally, the server can guide the client to the payment process by returning a link for the payment as shown in Listing 8.

```
<link rel="next payment" href="/users/4812/orders/1684/payment"
      type="application/xml" saps:schema="/schemas/payment.xsd"
      title="Pay to complete your order" />
```

Listing 8. A typed and SAPS-annotated link to guide a client to the payment process

In order to guarantee a loose coupling of the client and the server, the schemas have to be retrieved and interpreted on-the-fly at runtime and not at design time. This is in contrast to the traditional SOAP-practice where the schemas are used at design time to generate static proxy classes to interact with the service. Additionally, it has to be assured that developers do not fall in the "RPC trap". Developers need to be aware at any point whether local or remote resources are accessed in order to treat the differences accordingly; otherwise there is an imminent danger of significantly reduced scale, greater client-server coupling, and more difficult system modification and maintenance [8], [173], [174].

5.1.3 Integration into the Linked Data Cloud

SAPS relies on SAWSDL for the semantic annotation of XML schemas. Surprisingly, however, SAWSDL, however, does not specify how the semantic annotations can be used to convert the XML instance data to RDF or vice versa. In fact, SAWSDL does not even specify a language for representing the semantic models, meaning that RDF is just an option. Similarly, SAWSDL does not prescribe any particular mapping language for its `liftingSchemaMapping` and `loweringSchemaMapping` attributes.

While SAWSDL's specification [119] contains an example illustrating how XSLT [136] and SPARQL [71] may be used to lift XML documents to RDF and lower RDF data to XML, other languages such as XQuery [135] may be used as well. The translation from XML to RDF is typically called *lifting* because data in RDF is on a higher level of abstraction than data in XML. This flexibility in regard to both the semantic model and the schema mapping languages complicates the implementation of clients and impedes interoperability, as the server and the client need to find a model and a mapping language they both support. Thus, in practice, a server may have to offer several alternatives to increase the likelihood of a match.

SAPS does not restrict the mapping languages but requires a mapping to RDF in order to integrate the services into the Linked Data cloud. In our experiments we used a pragmatic solution. We interpreted complex types as entities that are identified with the URI that is the value of the element with the type `xs:anyURI` but without `modelReference`. For the `festivalType` in Listing 7 it is thus the `id` element which holds the URI identifying the entity. The `modelReference` of the complex type defines the entity's `rdf:type`. All remaining elements of the complex type with a `modelReference` represent properties of the entity. The festival contained in the search result in Listing 6 would thus be converted to the represen-

```
@prefix rdf: <http://www.w3.org/1999/02/22-rdf-syntax-ns#> .
@prefix xs: <http://www.w3.org/2001/XMLSchema#> .

<http://example.com/festivals/umf84705>
      rdf:type <http://schema.org/Festival> ;
      <http://schema.org/name> "Ultra Music Festival"^^xs:string ;
  <http://schema.org/description> "..."^^xs:string ;
  <http://schema.org/startDate> "2014-03-28"^^xs:date ;
  <http://schema.org/endDate> "2014-03-30"^^xs:date ;
  <http://schema.org/performer> ...
  <http://schema.org/offer> [
      rdf:type <http://schema.org/Offer> ;
      <http://schema.org/name> "General Admission"^^xs:string ;
      <http://schema.org/sku> "umf84705-165"^^xs:string ;
      <http://schema.org/price> "399.95"^^xs:decimal .
  ] .
```

Listing 9. The festival returned by the query in Listing 6 translated to Turtle

tation in Turtle shown in Listing 9. Obviously such a simple approach does not work for more complex mappings which need, e.g., to combine two XML elements to a single property value in RDF as when merging a first name and last name to just a name.

5.1.4 Summary and Lessons Learned

With the definition of just two attributes SAPS is able to build an extensible framework by integrating a number of proven, standardized technologies. Simple services consisting of just a CRUD-style interface and a simple query mechanism can be realized without having to rely on any out-of-band documentation as the necessary interaction models have already been specified by the underlying standardized technologies. For more complex scenarios, however, new link relations have to be defined which means that a dependency on additional out-of-band documentation describing them is introduced.

Since SAPS is based on the Atom protocol suite it is similar to previous efforts such as Google's Data Protocol (GData) [156] or Microsoft's Open Data Protocol (OData) [157]. The difference to GData is that the allowed elements are described in a machine-readable manner in the form of a schema instead of defining them just in a human-readable form. This makes it more similar to Microsoft's OData but, in contrast to OData, SAPS uses standardized building blocks such as XML Schema and SAWSDL to do so instead of defining a completely new and proprietary data model as Microsoft does.

The fact the all major components of SAPS are already standardized leads to standard-conforming, interoperable services. Unfortunately, however, it is difficult to integrate the various components into a single product. The situation is made even worse by the fact that some important aspects, such as the conversion of XML documents to RDF via SAWSDL-annotated schemas, are underspecified. Therefore, it has to be said that the approach is of limited practical use. To be accepted by developers, a fully specified approach covering all necessary aspects is needed. Furthermore, a more gradual introduction has to be supported instead of requiring developers to change their toolchains completely and to reimplement their services from scratch. Nevertheless, from a research point of view, SAPS was a successful project as it allowed us to experiment with the underlying ideas without having to spend much effort on foundational groundwork or implementation details.

5.2 SEREDASj

The experiments with SAPS confirmed that the underlying idea of combining Semantic Web technologies with technologies used in RESTful services works but also revealed a number of practical issues. The rigid structure dictated by Atom makes it, at times, difficult to implement intuitive Web APIs. Furthermore, the model based on XML, XML Schema, SAWSDL etc. turned out to be overly complex in the context of lightweight RESTful services. The feedback we received made it strikingly clear that a more lightweight and flexible solution is required to be of practical use. While looking for alternative serialization formats, we quickly turned our attention to JSON [50] as it was becoming increasingly popular at the time.

The advantage of JSON is that in most programming languages it is much easier to work with as it can be directly parsed into an in-memory representation sharing the same structure as the data itself. This is clearly a big advantage but also imposes the risk of a tight coupling between clients and servers if they depend on the same data structures internally and externally. In an attempt to eliminate this coupling we designed SEREDASj—a language to describe *SEmantic REstful DAta Services.* The "j" at the end highlights the fact that the approach is based on JSON.

Similar to SAPS, SEREDASj is optimized for CRUD-style services but instead of forcing developers to (re)implement their services using the Atom protocol suite and XML, SEREDASj attempts to describe existing JSON-based services. It is thus not a data interchange format (or extension thereof) but a description language. SEREDASj descriptions document the semantics of the representations and the relationships between resources. Furthermore, as we will see in section 5.2.3, the descriptions can be used to lift representations to RDF, manipulate the data with SPARQL, and write the changes back to the service. This moves the coupling from the syntactic structures of the JSON representations to the semantics of the data, which not only change much less frequently but can also much more easily be shared between various services.

In this section, we will first introduce the basic underlying ideas and their realization in SEREDASj. Then, in section 5.2.2, we will illustrate its usage based on a simple example. Finally, in section 5.2.3 we will show how services based on SEREDASj can be integrated in the Linked Data cloud before we conclude the section with a brief discussion of the strengths and weaknesses of SEREDASj. These sections are based on previous work which has been published in [95], [97], [167].

5.2.1 Basic Concepts and Principles

To describe a RESTful service, SEREDASj specifies the syntactic structure of a specific JSON representation, similar to Zyp's JSON Schema draft [175]. Additionally, it allows the mapping of JSON elements to concepts in an ontology and further describes the element itself by semantic annotations. In contrast to the JSON Schema draft, SEREDASj has no validation rules as such but instead allows a developer to add arbitrary descriptions in the form of semantic annotations to a JSON element. The rationale behind this is that we believe that the data has to be understood semantically to be validated and used correctly; simple validation rules as the ones proposed by Zyp are not expressive enough and thus of limited use. In order to illustrate this, e.g., it is impossible to define in a JSON Schema that a value has to be either between ten and twenty or forty and fifty (think of something like frequency bands); it is just possible to define that it has to be between ten and fifty. A program understanding the concepts in the semantic annotations will be much more capable of validating the data. Since our use cases are fundamentally different from Zyp's this should not be understood as a critique towards Zyp's approach; quite the contrary. SEREDASj is heavily based on Zyp's JSON Schema draft to describe the syntactic structure of representations.

As illustrated in Figure 13 on the next page, a SEREDASj document consists of metadata and a description of the structure of the JSON data it describes. The metadata describes the hyperlinks related to the JSON instance data and defines prefixes to abbreviate long IRIs in the semantic annotations to CURIEs [172]. The structure of the JSON data is described in terms of nested element descriptions, which define both the syntactic structure of the data as well the semantic meaning of those syntactic constructs.

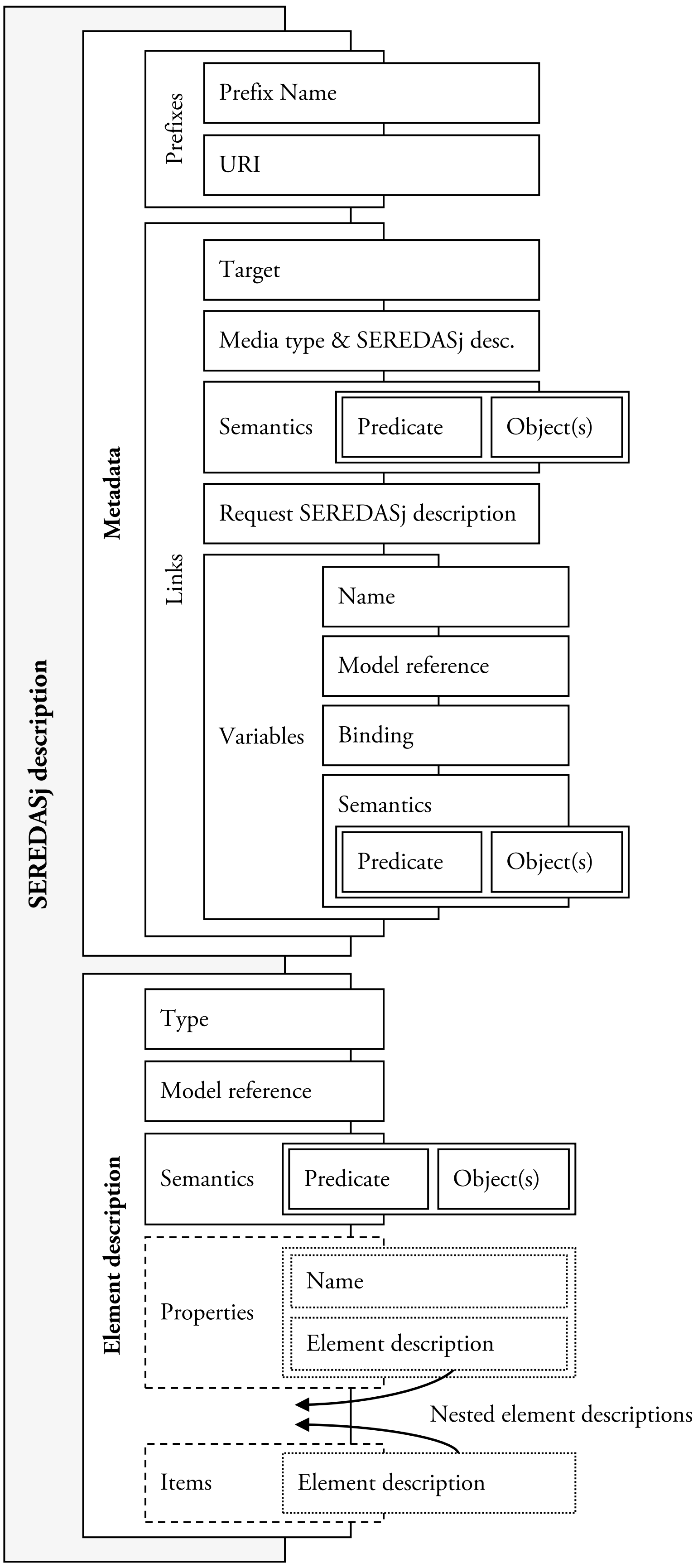


Figure 13. The SEREDASj description model

The description of links consists of a semantic description of the link, the link's target as either a concrete IRI reference or a IRI template, a definition of the variables to fill the target's IRI template, a hint about the target's media type and its SEREDASj description, and, for the construction of requests that require a payload, the SEREDASj description describing the expected payload. Just as the link itself, the IRI template variables can be described by generic semantic annotations in the form of predicate-object pairs. This can be used to describe restrictions of the value space of IRI template variables or to define a link's relation type. A link's variables can be either bound to an element in the instance data or be linked to a conceptual model, e.g., a property in an ontology.

Thus, a link description contains all the necessary information for a client to construct links by filling IRI templates with concrete values from the data or to create links depending on dynamic, client-supplied information, which is, e.g., used for query interfaces where the link contains the query criteria. Furthermore, the link description describes how a valid request body can be constructed in order to manipulate resources.

The syntactic structure of the data is described by nested element descriptions as illustrated in Figure 13. Each element description defines the element's JSON data type(s) as well as the mapping to a semantic concept. As we will see later in this chapter, this makes it possible to convert documents to RDF and vice versa. Additionally, an element description may contain semantic annotations to describe the element in more detail, and if the element represents either a JSON object or an array, it also contains a description of the object's properties respectively the array's items in term of, again, an element description.

Given that elements in different representations frequently represent the same concept, SEREDASj allows element descriptions to be reused by setting the type of an element description to the IRI of another element description, even across different SEREDASj documents. Different parts of a SEREDASj document can be referenced by using a slash-delimited fragment resolution similar to JSON Pointer [176] (which was specified after SEREDASj; otherwise we would have reused it) but without the leading slash. For instance, if the top-level structure described by a SEREDASj document called `doc.seredasj` is an object with a property called `name`, the IRI to reference the description of that property would be `doc.seredasj#properties/name`.

SEREDASj descriptions do not have to be complete, i.e., they do not need to describe every element in all details. If an unknown element is encountered in a document it is simply ignored. This way SEREDASj allows forward compatibility as well as extensibility. It should also be emphasized that a SEREDASj description does not imply a shared data model between a service and a client. It just provides a description of the service's representations to enable the translation between the service's and the client's data model. Just as in SAPS the coupling happens on the semantic layer instead of the syntactic structure of representations.

Typically, a representation is linked to its SEREDASj description via an HTTP Link header [57]. By following the Link header a client can easily find the description to interpret a representation and extract hyperlinks from the JSON data. This approach, however, works only for Web APIs where the API publisher itself decides to use SEREDASj. Thus, in order to avoid the classic chicken-and-egg problem of such new approaches, SEREDASj also allows an API user to describe a service independently of the API publisher. An API consumer can establish an overlay graph parallel to, but independent of the resources exposed by the service consisting of SEREDASj descriptions. In such a case, the client is given the entry point of the API and a link to the SEREDASj document describing it. From that point onwards the client will rely on the SEREDASj documents associated to the links it decides to follow. It thus navigates simultaneously through the resource space of the service and the SEREDASj descriptions, which have not been created by the API publisher but by a third party or by the consumer himself.

Similar to SAPS, which inherits the Atom Publishing Protocol's interaction model, SEREDASj by itself does not define any interaction model apart from untyped hyperlinks and basic CRUD operations. Instead, it completely relies on semantic annotations to describe the semantics of both hyperlinks and the data itself. In practice this means that without a concrete vocabulary describing the hyperlinks and data, the functionality of an automatic client is limited. This separation of concerns also means that the semantics are independent of the serialization format, which makes it possible to use exactly the same approach to describe, e.g., an XML-based service. Furthermore, it liberates the data from the tight corset that Atom enforces with its collection/entries model.

5.2.2 Illustrative Example

To make it easier to compare SEREDASj with SAPS and in order to better illustrate the approach, this section shows how the festival Web API described in section 5.1.2 can be implemented using SEREDASj. In contrast to SAPS, which uses Atom service documents as the API's entry point and Atom feeds for the representation of collections, SEREDASj does not distinguish between representations. It is a generic mechanism to describe arbitrary JSON documents. Thus, instead of beginning with the description of the entry point, which consists of just two links referencing the collection of festivals and orders and an IRI template to invoke search queries, we will start with the representation of a festival:

```
{
  "id": "umf84705",
  "label" : "Ultra Music Festival",
  "desc": "Outdoor electronic music festival",
  "from": "2014-03-28",
  "till": "2014-03-30",
  "performers": [
    {
      "id": "t6159",
      "name": "Tiësto"
    }
  ],
  "tickets": [
    {
      "label": "General Admission",
      "sku": "umf84705-165",
      "price": "399.95"
    }
  ]
}
```

Listing 10. Exemplary JSON representation of a festival

Without annotations the data cannot be understood by a machine, and even for a human it is not evident that a performer's ID is in fact a hyperlink to a more detailed representation of that specific performer. The SEREDASj description in Listing 11 solves these problems by describing all the important aspects of such a representation.

The metadata section in the SEREDASj document describes two links: one to get more details about the performers and one to order tickets. The link to the performer's details is defined in terms of an URI template [177] whose only variable is bound to the performer's `id` element in the data as well as the `artistId` concept in the service's vocabulary.

Furthermore, the description references SEREDASj documents describing both the representations that can be retrieved by dereferencing the link and the template to use when creating or updating artist resources. It also shows how a link can be annotated semantically. In this case, the annotation is leveraged for the conversion to RDF, which is described in detail in the next section. The second link specifies the interface to order tickets and is thus not bound to any element in the instance data but stands on its own. Again, the description describes the targets' representations and the template to create or manipulate orders. The links' semantic annotation defines the link relation (reusing Atom's `rel` attribute) a client can use to decide whether to follow the link or not. In this case, it tells the client that the link can be used to order. Such semantic annotations allow developers to implement smarter clients which follow REST's hypermedia as the engine of application state constraint. The client can dynamically choose among the server provided options by evaluating each link's semantics at runtime.

The rest of the SEREDASj document in Listing 11 describes the structure of the representation shown in Listing 10 (for the sake of brevity, we omitted details such as which properties are required and which are optional). Simply speaking, it describes the syntactic structure of the representation by nested element descriptions and maps them to concepts defined by Schema.org [85]. The mapping strategy is similar to the table-to-class, column-to-predicate strategy of the R2RML standard [178] which maps relational databases to RDF datasets. JSON objects are mapped to classes and their properties are mapped to predicates. This not only allows to translate the JSON representations to RDF, as described in the next section, but also to automatically create human-readable documentation of the data by exploiting the information about the various concepts of the used vocabulary, in this case Schema.org. The mapping to semantic concepts thus not only reduces the coupling between the client and the server by moving it to a centrally-owned contract but also liberates developers from the tedious task of writing documentation.

```
{
  "meta": {
    "prefixes": {
      "owl": "http://www.w3.org/2002/07/owl#",
      "schema": "http://schema.org/",
      "atom": "http://www.w3.org/2005/Atom",
      "ex": "http://example.com/vocab#"
    },
    "links": {
      "/artists/{id}#": {
        "mediaType": "application/json",
        "seredasjDescription": "artist.json",
        "requestDescription": "artist-createupdate.json",
        "semantics": { "[owl:sameAs]": "<#properties/performers>" },
        "variables": {
          "id": {
            "binding": "#properties/performers/id",
            "model": "[ex:artistId]"
          }
        }
      },
      "/orders/": {
        "mediaType": "application/json",
        "seredasjDescription": "order.json",
        "requestDescription": "order-createupdate.json",
        "semantics": { "[atom:rel]": "[ex:order]" }
      }
    }
  },
  "type": "object", "model": "[schema:Festival]",
  "properties": {
    "id":    { "type": "string", "model": "[ex:festivalId]" },
    "label": { "type": "string", "model": "[schema:name]" },
    "desc":  { "type": "string", "model": "[schema:description]" },
    "from":  { "type": "string", "model": "[schema:startDate]" },
    "till":  { "type": "string", "model": "[schema:endDate]" },
    "performers": {
      "type": "array", "model": "[schema:performer]",
      "items": {
        "type": "object", "model": "[schema:Person]",
        "properties": {
          "id":   { "type": "string", "model": "[ex:artistId]" },
          "name": { "type": "string", "model": "[schema:name]" }
    } } },
    "tickets": {
      "type": "array", "model": "[schema:offer]",
      "items": {
        "type": "object", "model": "[schema:Offer]",
        "properties": {
          "label": { "type": "string", "model": "[schema:name]" },
          "sku":   { "type": "string", "model": "[schema:sku]" },
          "price": { "type": "string", "model": "[schema:price]" }
  } } } }
}
```

Listing 11. SEREDASj document describing the representation in Listing 10

5.2.3 Integration into the Linked Data Cloud

As the name suggests, one of the main goals of SEREDASj is to integrate JSON-based services into the Semantic Web. In this section, we will not only show how SEREDASj descriptions can be used to convert JSON documents to RDF but also how the resulting data can be manipulated using SPARQL and how the changes can be written back to the Web API. This allows a seamless integration of RESTful services into the Linked Data cloud and goes beyond the typical read-only interfaces.

Translating SEREDASj-described JSON representations to RDF triples is a straightforward process. The translation starts at the root of the JSON representation and considers all model references of JSON objects to be RDF classes while all the other elements' model references are considered to be RDF predicates; values of those elements will be taken as objects. If a representation contains nested objects, just as the example in Listing 10, a slash-delimited URI fragment is used to identify the nested object. Semantic annotations in the form of the `semantics` property, as

```
@base <http://example.com/festivals/umf84705> .

@prefix rdf: <http://www.w3.org/1999/02/22-rdf-syntax-ns#> .
@prefix owl: <http://www.w3.org/2002/07/owl#> .
@prefix schema: <http://schema.org/> .
@prefix atom: <http://www.w3.org/2005/Atom> .
@prefix ex: <http://example.com/vocab#> .

<#> rdf:type schema:Festival ;
    ex:festivalId "umf84705" ;
    schema:name "Ultra Music Festival" ;
    schema:description "Outdoor electronic music festival" ;
    schema:startDate "2014-03-28" ;
    schema:endDate "2014-03-30" ;
    schema:performer <#performers/0> .
    schema:offer <#tickets/0> .

<#performers/0> rdf:type schema:Person ;
                ex:artistId "t6159" .
                schema:name "Tiësto" .

<#tickets/0> rdf:type schema:Offer ;
             schema:name "General Admission" ;
             schema:sku "umf84705-165" ;
             schema:price "399.95" .

</artists/t6159#> owl:sameAs <#performers/0> .
</orders/> atom:rel ex:order .
```

Listing 12. The example from Listing 10 translated to RDF

the one shown in the artist details link in Listing 11, contain the predicate and the object. The object might point to a specific element in the SEREDASj description and is eventually translated to a link in the instance data.

Listing 12 shows the result of the automatic translation of the example from Listing 10 to RDF. The event, the performers, and the tickets are nicely mapped to Schema.org's ontology. For every array item a new IRI is created by using a slash-delimited IRI fragment. Eventually, those IRIs are mapped to the performer's "real" IRI by an OWL's `sameAs` assertion [179] taken from the link's semantic annotation. This allows the JSON data to be integrated in the Linked Data cloud.

In fact, a big part of the current Semantic Web consists of data that is extracted from Web APIs, relational databases, or traditional Web sites and transformed to RDF. Unfortunately, this also means that the vast majority of the current Semantic Web is just read-only, i.e., changes cannot be stored back to the original source. Thus, we will show in the next sections how SEREDASj allows data to be updated and transferred back to the originating Web API.

In the following description we assume that all data of interest and the corresponding graph of interlinked SEREDASj descriptions have already been retrieved (whether this means crawled or queried specifically is irrelevant). The objective is then to manipulate the harvested data or to add new data by using SPARQL Update.

SPARQL Update [72] manipulates data by either adding or removing triples from a graph. The `INSERT DATA` and `DELETE DATA` operations respectively add and remove a set of triples from a graph by using concrete data (no named variables). In contrast, the `INSERT` and `DELETE` operations also accept templates and patterns. SPARQL has no operation to change an existing triple as triples are considered to be binary: the triple either exists or it does not. This is probably the biggest difference between SQL and Web APIs and complicates the translation between a SPARQL query and the equivalent HTTP requests to interact with a Web service.

Translating INSERT DATA and DELETE DATA Operations

In regard to a Web service an `INSERT DATA` operation can either result in the creation of a new resource or in the manipulation of an existing

```
do
  requests ← retrievePotentialRequests(triples)
  progress ← false
  while requests.hasNext() = true do
    request ← requests.next()
    request.setData(triples)
    request.setData(tripleStore)
    if isValid(request) = true then
      if request.submit() = success then
        resp ← request.parseResponse()
        triples.update(resp.getTriples())
        tripleStore.update(resp.getTriples())
        requests.remove(request)
        progress ← true
      end if
    end if
  end while
while progress = true
if triples.empty() = true then
  success()
else
  error(triples)
end if
```

Algorithm 1. Translate SPARQL `INSERT DATA`/`DELETE DATA` to HTTP requests

resource if just a previously unset attribute of an already existing resource is set. The same applies to a `DELETE DATA` operation which could unset an attribute of a resource or delete the whole resource. A resource is only deleted if all the triples describing the resource are deleted. This mismatch, or rather, conceptual gap between triples and resource attributes implies that constraints imposed by the Web service's interface are transferred to SPARQL's semantic layer. In consequence some operations that are completely valid if applied to a native triple store are invalid when applied to a Web API. If these constraints are documented in the interface description, i.e., the SEREDASj document, a client is able to construct valid requests or to detect invalid requests and give meaningful error messages. If these constraints are not documented, a client has no choice but to try and issue requests to the server and evaluate its responses. This is similar to HTML forms with and without client-side form validation.

In order to better explain the translation algorithm we will use the festival Web API whose interface is partially described in Listing 11 but ignore the tickets to keep the examples simple. We will assume that the CRUD operations to store and manipulate festivals and their respective performers are mapped to the HTTP verbs `POST`, `GET`, `PUT`, and `DELETE`. Festival representations can be accessed at `/festivals/{id}` URLs while the performers are accessible at `/artists/{id}` URLs. Both can be edited by PUTing an updated JSON representation to the respective URL. New festivals and artists can be created by POSTing a JSON representation to their respective collection URL.

Since SPARQL differentiates between data and template operations, we split the translation algorithm into two parts. Algorithm 1 translates SPARQL `INSERT DATA`/`DELETE DATA` operations to HTTP requests interacting with the Web service and Algorithm 2 deals with SPARQL's `DELETE`/`INSERT` operations using patterns and templates.

Listing 13 contains an exemplary `INSERT DATA` operation which we will use to explain Algorithm 1. It creates a new festival and a new artist. The festival is linked to the newly created artist as well as to an existing one.

To convert the operations in Listing 13 to HTTP requests interacting with the Web service, in the first step (line 2 in Algorithm 1) all potential requests are retrieved. This is done by retrieving all SEREDASj descriptions that contain model references corresponding to classes or predicates used in the SPARQL triples; this step also takes into consideration whether an existing resource should be updated or a new one created. Since Listing 13 does not manipulate existing resources (`/artists/t6159#` in line 11 is just used as an object), all potential HTTP requests have to create new resources, i.e., have to be `POST` requests in our example. In our example we get two potential requests, one for the creation of a new festi-

```
 1  BASE <http://example.com/>
 2
 3  PREFIX rdf:    <http://www.w3.org/1999/02/22-rdf-syntax-ns#>
 4  PREFIX schema: <http://schema.org/>
 5
 6  INSERT DATA {
 7    _:greatg rdf:type schema:Festival ;
 8             schema:name "Great Gig" ;
 9             schema:startDate "2014-02-14" ;
10             schema:endDate "2014-02-15" ;
11             schema:performers </artists/t6159#> ;
12             schema:performers _:williams .
13    _:williams rdf:type schema:Person ;
14               schema:name "Robbie Williams" .
15  }
```

Listing 13. Exemplary `INSERT DATA` operation

```
 1  → POST /festivals/
 2      { "name": "Great Gig", "from": "2014-02-14",
 3        "till": "2014-02-15", "performers": [ { "id": "t6159" } ] }
 4  ← 201 Created
 5    Location: /festivals/gg51972#
 6
 7  → POST /artists/
 8      { "name": "Robbie Williams" }
 9  ← 201 Created
10    Location: /artists/k92167#
11
12  → PUT /festivals/gg51972
13      { "name": "Great Gig",
14        "performers": [ { "id": "t6159" },
15                        { "id": "k92167" } ] }
16  ← 200 OK)
```

Listing 14. `INSERT DATA` operation from Listing 13 translated to HTTP requests

val resource and one for a new person/artist resource. These request templates are then filled with information from the SPARQL triples (line 6) as well as with information stored in the local triple store (line 7). Then, provided a request is considered to be valid (line 8), it will be submitted (line 9).

As shown in Listing 14, in our example the first valid request creates a new event (lines 1-5). Since the ID of the blank node `_:williams` is not known yet (it gets created by the server), it is simply ignored. Provided the HTTP request was successful, the response is subsequently parsed, and the new triples exposed by the Web service are removed from the SPARQL triples (line 11) and added to the local triple store (line 12). Furthermore, the blank nodes in the remaining SPARQL triples are replaced with concrete terms. In our example this means that the triples in lines 7-11 in Listing 13 are removed and the blank node subject of the triple in line 12 is replaced by the `/festivals/gg51972#` URL returned by the server. Finally, the request is removed from the potential requests list and a flag is set (line 13-14, Algorithm 1) signaling that progress has been made within the current `do while` iteration. If in one loop iteration, which cycles through all potential requests, no progress has been made, the process is stopped (line 18). In our example the process is repeated for the request to create a person, which again results in an HTTP `POST` operation (line 7-8, Listing 14). Since there are no more potential requests available, the next iteration of the `do while` loop begins.

The only remaining triple is the previously updated triple in line 12 (Listing 13), thus the only potential request this time is a `PUT` request to update the newly created `/festivals/gg51972#`. As before, the request template is filled with "knowledge" from the local triple store and the remaining SPARQL triples and eventually processed. Since there are no more SPARQL triples to process, the `do while` loop terminates and a success message is returned to the client (line 20, Algorithm 1) as all triples have been successfully processed.

Translating DELETE/INSERT Operations

In contrast to the `DATA`-form operations that require concrete data and do not allow the use of named variables, the `DELETE/INSERT` operations are pattern-based and use templates to delete or add groups of triples. These operations are processed by first executing the query patterns in the `WHERE` clause that bind values to a set of named variables. These bindings are then used to instantiate the `DELETE` and the `INSERT` templates and finally the concrete deletes are performed followed by the concrete inserts. Thus, the `DELETE/INSERT` operations are effectively transformed to concrete `DELETE DATA/INSERT DATA` operations before execution. We exploit this fact in Algorithm 2, which transforms `DELETE/INSERT` operations to `DELETE DATA/INSERT DATA` operations that are then translated by Algorithm 1 into HTTP requests.

Listing 15 contains an exemplary `DELETE/INSERT` operation which replaces the name of all persons whose ID equals `k92167` with `Lenny Kravitz`;

```
PREFIX rdf:    <http://www.w3.org/1999/02/22-rdf-syntax-ns#>
PREFIX schema: <http://schema.org/>
PREFIX ex:     <http://example.com/vocab#>

DELETE {
  ?person schema:name ?name .
}
INSERT {
 ?person schema:name "Lenny Kravitz" .
}
WHERE {
  ?person rdf:type schema:Person ;
          ex:artistId "k92167" ;
          schema:name ?name .
}
```

Listing 15. Exemplary `DELETE/INSERT` operation

```
1  select ← createSelect(query)
2  bindings ← tripleStore.execute(select)
3
4  for each binding in bindings do
5    deleteData ← createDeleteData(query, binding)
6    operations.add(deleteData)
7    insertData ← createInsertData(query, binding)
8    operations.add(insertData)
9  end for
10
11 operations.sort()
12 translateDataOperations(operations)
```

Algorithm 2. SPARQL `DELETE/INSERT` operations to HTTP requests translation algorithm

regardless of what it was before. This operation is first translated to a `DELETE DATA/INSERT DATA` operation by Algorithm 2 and then to HTTP requests by Algorithm 1.

The first step (line 1, Algorithm 2) is to create a `SELECT` query out of the `WHERE` clause. This query is then executed on the local triple store returning the bindings for the `DELETE` and `INSERT` templates (line 2). This implies that all relevant data has to be included in the local triple store (an assumption made earlier in this section), otherwise the operation might be just partially executed. For each of the retrieved bindings (line 4), one `DELETE DATA` (line 5) and one `INSERT DATA` (line 7) operation is created. In our example, the variable `person` is bound to the URL `/artists/k92167#` and `name` to `Robbie Williams`. Therefore, only one `DELETE DATA` and one `INSERT DATA` operation are created as shown in Listing 16. The operations are then sorted (line 11) as deletes have to be executed before inserts and finally translated into HTTP requests (line 12) by Algorithm 1.

```
PREFIX rdf:    <http://www.w3.org/1999/02/22-rdf-syntax-ns#>
PREFIX schema: <http://schema.org/>
PREFIX ex:     <http://example.com/vocab#>

DELETE DATA {
  </artists/k92167#> schema:name "Robbie Williams" .
}
INSERT DATA {
  </artists/k92167#> schema:name "Lenny Kravitz" .
}
```

Listing 16. `DELETE DATA/INSERT DATA` operations generated from Listing 15 by Algorithm 2

In many cases, just as demonstrated in the example, a `DELETE/INSERT` operation will actually represent a replacement of triples. Thus, the efficiency of the algorithm can be improved by performing both the `DELETE DATA` and the `INSERT DATA` locally before issuing the actual HTTP request. This optimization reduces the number of HTTP requests since attributes do not have to be reset before getting set to the desired value. In our example this consolidates the two `PUT` requests into one.

5.2.4 Summary and Lessons Learned

SEREDASj is a semantic description language that has been specifically designed for JSON-based RESTful services. It focuses on the description of JSON representations and their semantic annotation. Representations can be augmented with links and semantically annotated, which makes it possible to build hypermedia-driven APIs and convert the data to interlinked RDF, i.e., Linked Data. We have also shown how the data can be manipulated with SPARQL Update on a semantic level, abstracting away the serialization details, and how the changes can be written back to the Web API. Effectively, this introduces a standardized interface, which not only has the potential to increase the productivity of developers but also to improve code reusability and reduce the coupling between the client and the server. Unlike SAPS, SEREDASj does not require any changes on the described services and thus provides a smooth upgrade for existing Web APIs. This also implies that developers can continue to use tools and knowledge they are already familiar with.

Despite the fact that we put a strong emphasis on simplicity in order to lower the entry barrier for developers, we found that in practice the followed approach is suboptimal. Developers struggle with the separation of data into representations and descriptions thereof. The layer of interconnected SEREDASj descriptions that sits on top of the JSON representations exposed by the service increases the cognitive load put on developers substantially. These lifting and lowering schema mappings to translate between serialization formats such as JSON to namespaced, interlinked graphs in RDF, require not only an understanding of the exchanged data but also how that data gets lifted to a representation in an abstract syntax such as RDF. It is thus difficult for developers to understand the representations and the links connecting them without looking at the corresponding SEREDASj description documents at the same time. Developers also complained that the syntax, which follows JSON Schema's approach, is too verbose and that the reliance on semantic annotations to realize services whose functionality goes beyond simple CRUD operations is problematic as no vocabulary defining the necessary concepts is available—even guidelines to create one are still missing. Similarly, using the CRUD operations as the main interaction model is often problematic as their semantics are not explicit enough. Clients need to know whether the creation of, e.g., an order entity is just that or whether it also implies the delivery of goods.

Summarized, the lesson we learned from building SAPS and SEREDASj and by working with developers is that the underlying ideas are practical and accepted by developers but that the concrete realizations we chose to follow were suboptimal. In our research, we came to the conclusion that it is generally better to put more effort in the messages themselves instead of applying the intelligence to message translation mechanisms such as lifting and lowering schema mappings. What is needed is a clear separation of the solution into a generic data interchange format and vocabularies defining the semantics of both the data to be exchanged and the domain application protocol, i.e., a description of the behavioral model.

5.3 JSON-LD

Both SAPS and SEREDASj provide a more or less integrated solution for CRUD-based services but their expressivity and extensibility are limited by the lack of a clear separation between the interaction model, the semantics of the data, and the serialization format. If developers need to step outside the interaction models of the simple CRUD-based service interfaces that these approaches were optimized for, the lack of separation results in increased complexity and thus cognitive load. The experience we gained from working with SAPS and SEREDASj and teaching them to other developers revealed that it is necessary to introduce a clear separation between these aspects in order to build a scalable and extensible solution. After getting started with such work, we discovered the JSON-LD project. As it had almost the same goals that we were trying to achieve and followed a very similar approach, we joined the, back then, still small community. Consequently, we decided to discontinue our work on refining SEREDASj in favor of JSON-LD as we believed we could achieve more by joining forces instead of working on similar, yet different approaches competing to solve similar problems.

In spite of being a comparatively young project, JSON-LD has already had a turbulent history. According to Manu Sporny [180], the chair of the JSON-LD community group, the work was started internally at Digital Bazaar in March 2010. This was shortly before the W3C RDF Next Steps Workshop where the desire for a JSON-based RDF format was expressed [181]. Consequently, the RDF Working Group at W3C started working on a JSON-based RDF serialization on two fronts. It decided to quickly standardize Talis' triple-centric RDF/JSON [182] for RDF experts needing a JSON-based serialization and to incubate on JSON-LD for average Web developers without RDF background. Unsurprisingly, this strategy soon ended in general confusion as to the exact target group it is attempting to address and what the final outcome should be. The group did not share a common vision. Finally, in August 2011, Thomas Steiner, the appointed co-editor of the JSON-based RDF serialization format pulled the "emergency brake" [183] and the work in the RDF Working Group was stopped. Despite these happenings we continued to work as part of the JSON-LD community to improve the syntax. Instead of waiting for the RDF Working Group to decide on how to proceed the work was moved into a W3C Community Group, a lightweight alternative to a full-grown W3C Working Group.

Over the years we contributed several ideas to turn JSON-LD from a language that resulted in documents as the one shown in Listing 17 into a language allows documents to be created that look almost indistinguishable from the idiomatic JSON used in current Web APIs. As usual in open development projects, we started by asking questions, raising issues, and providing ideas to earn the trust and respect of the community. Over the years, the author of this dissertation went from contributing test cases and minor text proposals for the specification to become the lead editor of the JSON-LD 1.0 Processing Algorithms and API [168] and co-editor of the JSON-LD 1.0 syntax specification [144]. As JSON-LD was brought back to the RDF Working Group for standardization the author of this thesis was invited by the W3C to join the group as an expert. This allowed him to directly participate in the devel-

```
{
  "#": {
    "foaf": "http://xmlns.com/foaf/0.1/"
  }
  "@": "<http://example.com/people/markus>",
  "foaf:name": "Markus Lanthaler",
  "foaf:homepage": "<http://www.markus-lanthaler.com/>",
  "foaf:knows": {
    "@": "</people/john>",
    "foaf:name": "John Doe"
  }
}
```

Listing 17. The structure of a JSON-LD document at the time we joined the JSON LD community

opment of the RDF Working Group's other specifications. After several contributions the author of this thesis eventually also became a co-editor of the central RDF specification RDF 1.1 Concepts and Abstract Syntax [63]. Even though we were not able to achieve everything we wanted, especially not in the RDF Working Group, these opportunities gave us a unique chance to influence—and hopefully improve—the further development of the Semantic Web and Linked Data.

The insights we have gained by creating SAPS and SEREDASj clearly helped us to shape and improve JSON-LD, which we will present in this section, and to design a vocabulary, which we will present in section 5.4. The division of the solution into a data format and a vocabulary not only reduces the overall complexity but has also the positive side effect that both parts can be used independently of each other. Just as JSON-LD can be used in conjunction with any RDF vocabulary (easily also with multiple vocabularies at the same time), Hydra, the vocabulary that we will present in section 5.4, can be used with any concrete RDF syntax.

Additional to the features JSON provides, JSON-LD supports hyperlinks, universal identifiers for entities and their properties in the form of IRIs, string internationalization, definition and use of arbitrary data types, support for unordered sets and ordered lists, and, last but not least, a facility to express named graphs. These features not only simplify data integration, which is the underlying problem in many Web API usage scenarios, but also enable developers to express their data with much stronger semantics.

Just as in the previous sections, we will start with introducing JSON-LD's basic concepts and principles before we illustrate how it can be used to realize our exemplary festival API. Finally, we will show how JSON-LD documents can be interpreted as RDF and discuss JSON-LD's relationship to other Semantic Web technologies before we conclude the section with some final remarks. This section is based on material that has been published in [98], [99], [144], [168].

5.3.1 Basic Concepts and Principles

JSON-LD is an attempt to create a simple method to not only express Linked Data in JSON but also to make existing JSON documents self-descriptive. Given the reluctance of average Web developers to use Semantic Web technologies, one of the primary design goals in the development of JSON-LD was to require as little effort as possible from developers to create and understand JSON-LD documents. Therefore, great efforts were put in its simplicity, terseness, and human readability. Furthermore, it was a goal to require as little effort as possible from developers to transform their plain old JSON to semantically rich JSON-LD. Instead of the normally triple-centric approach that other common serialization formats for Linked Data use, an entity-centric approach was chosen for JSON-LD. The rationale was to resemble the programming models most developers are familiar with and to reflect the way JSON is used. This, and the fact that JSON-LD is 100% compatible with traditional JSON, allows developers to build on existing infrastructure investments—which is especially important for enterprises as it allows them to add meaning to their JSON documents in a way that is not disruptive to their operations and is transparent to their current customers. Thus, in many respects, JSON-LD forms an entire ecosystem for developers to work with Linked Data without the high entry barrier that other Linked Data and Semantic Web technology stacks entail. While initial versions [184] of JSON-LD looked more or less like a direct translation of Turtle to JSON, we changed the syntax substantially in subsequent versions to allow data to be serialized in a way that is often indistinguishable from traditional JSON. This is a remarkable feature of JSON-LD given that JSON, whose native data model is a tree, is used to serialize directed graphs which potentially even contain cycles.

Given its focus on simplicity, a developer familiar with JSON generally needs to know only two keywords in order to use JSON-LD's basic functionality, namely `@context` and `@id`. The `@context` keyword is used to include or reference a so-called *context* which allows JSON properties to be mapped to IRIs in order to make them uniquely identifiable across the Web and, typically, dereferenceable. The `@id` keyword does the same for entities by assigning identifiers to JSON objects. Thus, it can also be used to express hyperlinks between resources. Using just these two keywords, information about two persons and their relationship can be expressed as follows in JSON-LD:

```
{
  "@context": "http://example.com/contexts/person.jsonld",
  "@id": "http://example.com/people/markus",
  "name": "Markus Lanthaler",
  "homepage": "http://www.markus-lanthaler.com/",
  "knows": {
    "@id": "/people/john",
    "name": "John Doe"
  }
}
```

Listing 18. A simple JSON-LD document serializing information about two persons

The document in Listing 18 contains information about a person identified by the IRI `http://example.com/people/markus` with the name `Markus Lanthaler`. It also contains a reference to another person whose identifier is `http://example.com/people/john` (the relative IRI is resolved against the document's base IRI, which can, if needed, be set explicitly using the `@base` keyword). This reference also shows how some of the properties (in this case the name) of a referenced entity can be directly embedded. This allows developers to fine-tune the performance of Web APIs by reducing the number of HTTP requests necessary for clients to retrieve the desired information. The referenced context maps the JSON properties in the document to IRIs which enables clients to retrieve more information about them by simply following those links; this principle is known as *Follow Your Nose* [185]. Assuming that FOAF [108] is used as the vocabulary, the context would look something like Listing 19.

The fact that plain old JSON documents can be interpreted as JSON-LD by referencing a context via an HTTP Link header provides a smooth upgrade path for existing infrastructure as it allows most of the functionality without having to change the contents of an existing document. API

```
{
  "@context": {
    "name": "http://xmlns.com/foaf/0.1/name",
    "homepage": "http://xmlns.com/foaf/0.1/homepage",
    "knows": "http://xmlns.com/foaf/0.1/knows"
  }
}
```

Listing 19. The JSON-LD context mapping the properties in Listing 18 to IRIs

publishers can thus continue to serve the same documents so that existing clients do not break, while at the same time, newer, more sophisticated clients able to leverage the additional information from the context are made possible. This mechanism is even more powerful when combined with JSON-LD's keyword aliasing feature. Apart from `@context`, every keyword can be aliased to any arbitrary string. This way it is e.g. possible to use `url` instead of `@id`. Aliases are mapped to keywords just as properties are mapped to IRIs.

If documents are served with JSON-LD's media type `application/ld+json`, it is also possible to embed the context directly in the document instead of just referencing it in an HTTP Link header. This saves additional HTTP requests at the cost of increasing the document's size. The developer is thus able to control the trade-off between bandwidth usage and latency. This control is very fine-grained as it is also possible to include or reference multiple contexts by wrapping them in an array. Developers can thus reference an external context and overwrite some of the mappings locally in the document.

Looking at the example in Listing 18, the alert reader might notice that the document contains a link to the person's homepage (`http://www.markus-lanthaler.com/`) without using the `@id` construct—and also the context in Listing 19 does not contain any further information to disambiguate that IRI from a regular string. This is where the `@type` keyword comes into play. It allows type information to be assigned to properties as well as to individual values and entities. The mapping for `homepage` in the context above can therefore be rewritten to include such type information to tell clients that the property's values represent IRIs.

In Listing 21, `@id` is used as the value of `@type` to express that the data type is an IRI—all other types are identified, just as everything else, with

```
{
  "@context": {
    ...
    "homepage": {
      "@id": "http://xmlns.com/foaf/0.1/homepage",
      "@type": "@id"
    }
  }
}
```

Listing 21. Type-coercion of the `homepage` property

IRIs. The most commonly used data types are already standardized as part of XML Schema [186] and it is recommended to reuse them whenever possible to improve interoperability. It is also possible to use `@type` directly in a document to express a value or entity type. The person entity in Listing 18 can, for instance, be enriched with its type and a typed creation date:

```
{
  "@context": "http://example.com/c/person.jsonld",
  "@id": "http://example.com/people/markus",
  "name": "Markus Lanthaler",
  "@type": "http://xmlns.com/foaf/0.1/Person",
  ...
  "created_at": {
    "@value": "2012-09-05",
    "@type": "http://www.w3.org/2001/XMLSchema#date"
  }
}
```

Listing 20. Usage of `@type` to specify the type of values and entities

The first use of `@type` in the example above associates a class (FOAF's `Person` class) with the entity identified by the `@id` keyword. The second use of `@type` associates a data type (XML Schema's `date`) with the value expressed using the `@value` keyword. This is similar to object-oriented programming languages where both scalar and structured types use the same typing mechanism, even though scalar types and structured types are inherently different. As a general rule the `@type` keyword is expressing a data type to be used with scalars when `@value` and `@type` are used in the same JSON object; otherwise it is expressing an entity type, i.e., a class. If it is used within a context, it always expresses a data type.

Another use of `@value` is to language-tag strings, which is essential for multilingual applications. This can be done with the `@language` keyword which tags a string with the supplied language code. Just as the `@type` keyword, it can either be used in the context or, along with `@value`, in a document's body. The example below shows how the academic title of the person can be added in both English and German:

```
{
  "@context": "http://example.com/c/person.jsonld",
  "@id": "http://example.com/people/markus",
  "name": "Markus Lanthaler",
  ...
  "title ": [
    { "@value": "MSc", "@language": "en" },
    { "@value": "Dipl. Ing.", "@language": "de" }
  ]
}
```

Listing 22. Expressing language-tagged strings

This brings us to the only case were JSON-LD differs from traditional JSON, i.e., arrays are generally considered as being unordered sets instead of ordered lists. This stems from the fact that JSON-LD's underlying data model is based on directed graphs in which edges are inherently unordered. In most cases, this is a minor detail that only matters when JSON-LD is transformed to other serialization formats or, e.g., persisted into a database. JSON-LD, however, also has built-in support for ordered lists in the form of the `@list` keyword which can be used to express that an array has to be interpreted as an ordered list. It can either be used directly in the document by wrapping an object with only an `@list` property around the array or be mapped to a property in the context by setting `@container` to `@list` (`@set` can be used to express explicitly that an array has to be interpreted as an unordered set and can be classified as syntactic sugar). Both methods are outlined in the following example:

```
{
  "@context": {
    "propertyA": "http://example.com/vocab#a",
    "propertyB": {
      "@id": "http://example.com/vocab#b",
      "@container": "@list"
    }
  },
  "propertyA": { "@list": [ "a", "b", "c" ] },
  "propertyB": [ "a", "b", "c" ]
}
```

Listing 23. Serializing lists using `@container` and inline `@list`-objects

JSON-LD's container feature is not limited to sets and lists; there are two more container types. The first allows to index language-tagged strings by their language. This makes the usage of JSON-LD in multilingual environments much more efficient since the desired language can be accessed directly instead of having to filter an array of language-tagged strings to find the desired entry. This is illustrated in Listing 24 which indexes the person's academic title from Listing 22 by language. Since indexing provides very compelling benefits in terms of performance and ease of use, JSON-LD also allows indexing by arbitrary string values using the `@index` keyword instead of `@language`. Adding an index does not affect the semantics of the data, it just provides a mechanism to bring the data into the most advantageous form for further processing or usage.

```
{
  "@context": {
    ...
    "title": {
      "@id": "http://xmlns.com/foaf/0.1/title",
      "@container": "@language"
    }
  },
  "@id": "http://example.com/people/markus",
  "name": "Markus Lanthaler",
  ...
  "title ": {
    "en": "MSc",
    "de": "Dipl. Ing."
  }
}
```

Listing 24. Language maps allow language-tagged strings to be indexed by language

The `@reverse` keyword goes in the same direction. As the name already suggests, it allows the direction of the arc that a property is spanning to be inverted. Sometimes vocabularies define inverse properties but most of the time they do not and thus inadvertently enforce a certain serialization structure onto JSON-LD documents. By using `@reverse` such restrictions can easily be sidestepped. Thus, regardless of the fact that FOAF does not define an inverse property for `knows`, the example from Listing 18 can be serialized in a way so that the person John Doe is the top-level object, as shown in Listing 25.

As the examples shown so far already suggest, it is often cumbersome and error-prone to spell out the IRIs in full to transform a JSON document to Linked Data. To mitigate that JSON-LD has two mechanisms to

```
{
  "@context": {
    ...
    "isKnownBy": { "@reverse": "http://xmlns.com/foaf/0.1/knows" }
  },
  "@id": "/people/john",
  "name": "John Doe",
  "isKnownBy": {
    "@id": "http://example.com/people/markus",
    "name": "Markus Lanthaler",
    "homepage": "http://www.markus-lanthaler.com/"
  }
}
```

Listing 25. Reshaping documents by reversing the direction of properties

minimize the need to type full IRIs. The first one is to define prefix mappings in the context to shorten long IRIs. By using prefixes, the context in Listing 19 can be simplified by defining a prefix for FOAF's vocabulary namespace and consequently use it to considerably shorten the IRIs as shown in Listing 26.

This not only makes the context much smaller but also much more readable and hence reduces the cognitive load put on developers. Prefixes can also be used directly in properties in the body of a document. A JSON-LD processor expands all compact IRIs (that is how IRIs using prefixes are called in JSON-LD) by first splitting them into a prefix and a suffix at the colon and then concatenating the IRI mapped to the prefix to the suffix. JSON-LD's compact IRIs are thus effectively CURIEs [172] but any of their restrictions.

The second approach to minimize the amount of long IRIs goes a step further by eliminating the need to manually map terms to full IRIs altogether—at least if a single vocabulary is used. JSON-LD's `@vocab` keyword can be used to define an implicit global prefix which is used for properties that are not explicitly mapped to an IRI. Since this method

```
{
  "@context": {
    "foaf": "http://xmlns.com/foaf/0.1/",
    "name": "foaf:name",
    "homepage": "foaf:homepage",
    "knows": "foaf:knows"
  }
}
```

Listing 26. Defining prefixes to abbreviate IRIs

automatically affects every non-mapped property in a document (it is possible to override this behavior by explicitly defining which properties should not be expanded), it is recommended to use this mechanism only when a) all or at least most properties are mapped to an IRI and b) most properties are mapped to the same vocabulary sharing a common IRI prefix. By using `@vocab` the initial example can be simplified as shown in Listing 27. Please note that no external context is referenced and that the `homepage` property definition in the context just defines the type coercion but not the IRI mapping.

```
{
  "@context": {
    "@vocab": "http://xmlns.com/foaf/0.1/",
    "homepage": { "@type": "@id" }
  },
  "@id": "http://example.com/people/markus",
  "name": "Markus Lanthaler",
  "homepage": "http://www.markus-lanthaler.com/",
  "knows": {
    "@id": "/people/john",
    "name": "John Doe"
  }
}
```

Listing 27. Defining a global prefix using `@vocab`

Data serialized with JSON-LD has the form of a graph, and, at times, it becomes necessary to make statements about the graph itself rather than just about the entities, i.e., the nodes, it contains. That is exactly the purpose of the last remaining keyword: `@graph`. It makes it possible to assign properties and an identifier to the graph itself. The example in Listing 28 shows how a graph can be annotated with its creation date. It is important to note that while RDF 1.1 [63] introduces the notion of named graphs and datasets it does not define their semantics. Thus, looking at it from an RDF perspective, strictly speaking the IRI `/graphs/1` does not denote the graph consisting of the information about the two persons; it is undefined what it denotes.

As the examples already illustrated, it is possible to serialize the same data in multiple ways. This is an inevitable consequence of the underlying graph based data model. Furthermore, JSON-LD's mechanisms to make the data look like idiomatic JSON, introduce additional variability. While this flexibility has many advantages, it also makes it more difficult to process the data. Thus, additional to the serialization format, we also

```
{
  "@context": {
    ...
    "generatedAt": {
     "@id": "http://www.w3.org/ns/prov#generatedAtTime",
     "@type": "xsd:date"
    }
  },
  "@id": "/graphs/1",
  "generatedAt": "2012-09-05",
  "@graph": {
    "@id": "/people/markus",
    "name": "Markus Lanthaler",
    "homepage": "http://www.markus-lanthaler.com/",
    "knows": {
      "@id": "/people/john",
      "name": "John Doe"
    }
  }
}
```

Listing 28. Named graphs in JSON-LD

created and standardized a number of algorithms and an application programming interface (API) to simplify the processing of JSON-LD documents [168].

The algorithms allow JSON-LD documents to be *expanded*, *compacted*, and *flattened*. Expansion is the process of taking a JSON-LD document and applying all embedded and referenced contexts such that all IRIs,

```
[
  {
    "@id": "http://example.com/people/markus",
    "http://xmlns.com/foaf/0.1/name": [
      { "@value": "Markus Lanthaler" }
    ],
    "http://xmlns.com/foaf/0.1/homepage": [
      { "@id": "http://www.markus-lanthaler.com/" }
    ],
    "http://xmlns.com/foaf/0.1/knows": [
      {
        "@id": "http://www.markus-lanthaler.com/people/john",
        "http://xmlns.com/foaf/0.1/name": [
          { "@value": "John Doe" }
        ]
      }
    ]
  }
]
```

Listing 29. The example from Listing 27 converted to expanded document form

types, and values are expanded in a way so that the contexts can be eliminated from the document without losing any information. Furthermore, all properties that allow multiple values are converted to array form to harmonize their representation. By doing so, expansion thus makes it much easier to write tools and libraries on top of a JSON-LD processor as it has already processed all the information contained in the context. Listing 29 shows how the document in Listing 27 looks when converted to expanded document form. While all the properties have been replaced with the full IRIs they are mapped to and the values are expanded to either `@value`- or `@id`-objects, the overall structure of the document is still the same.

This is not the case for flattened documents. Flattening simplifies processing even more as it also normalizes the document's structure. It is thus possible to bring any JSON-LD document to a deterministic shape. This makes it possible to program against a single structure instead of having to adapt to the various possible document shapes. As shown in Listing 30, flattening removes the nesting of the two person-objects by replacing it with a hyperlink connecting them. While the data stays the same, the syntactic structure of the document is brought into a deterministic shape. This makes it much easier for machines to process the document at the cost of making it more difficult to read and understand

```
[
  {
    "@id": "http://example.com/people/markus",
    "http://xmlns.com/foaf/0.1/name": [
      { "@value": "Markus Lanthaler" }
    ],
    "http://xmlns.com/foaf/0.1/homepage": [
      { "@id": "http://www.markus-lanthaler.com/" }
    ],
    "http://xmlns.com/foaf/0.1/knows": [
      { "@id": "http://www.markus-lanthaler.com/people/john" }
    ]
  },
  {
    "@id": "http://example.com/people/john",
    "http://xmlns.com/foaf/0.1/name": [
      { "@value": "John Doe" }
    ]
  }
]
```

Listing 30. The example from Listing 27 converted
to expanded, flattened document form

for humans. Thus, we also created algorithms to reverse this process. The counterpart to expansion is compaction.

Compaction takes a JSON-LD document and applies a supplied context such that the most compact form of the document is generated, i.e., all IRIs are translated to short terms (as specified by the supplied context) and all array values with a single entry are unwrapped from that array form. Compacting Listing 29 with the initial context from Listing 19 (please note the missing type-coercion for the `homepage` property) results in a document equal to Listing 31. Compaction is, however, not always the exact inverse operation of expansion; it is, e.g., impossible to split properties that have been merged to the same IRI during expansion. Since expansion and compaction can be used together, applications can use them to harmonize data representations by translating between different contexts in order to, e.g., rename properties.

```
{
  "@context": {
    "name": "http://xmlns.com/foaf/0.1/name",
    "homepage": "http://xmlns.com/foaf/0.1/homepage",
    "knows": "http://xmlns.com/foaf/0.1/knows"
  },
  "@id": "http://example.com/people/markus",
  "name": "Markus Lanthaler",
  "homepage": { "@id": "http://www.markus-lanthaler.com/" },
  "knows": {
    "@id": "http://example.com/people/john",
    "name": "John Doe"
  }
}
```

Listing 31. The document in Listing 29 compacted with the context from Listing 19

For greater flexibility, we also started designing a *framing* algorithm [187] which allows a developer to reshape and query a document using templating and query-by-example. This declarative definition of the desired syntactic structure of the document fits very well with the way developers typically work with JSON, i.e., they program directly against the structure of the document instead of going through an API. Thus, framing usually means that all existing JSON tools and workflows can be retained and the JSON-LD data can be processed as JSON by bringing the document to the most advantageous form prior its use. Unfortunately, due to concerns from the RDF WG, whose expertise is not the definition of APIs or processing algorithms, the framing algorithm and the corresponding API were not put on the recommendation track and are thus not part of JSON-LD 1.0 as standardized at the W3C. That being said, most available JSON-LD processors implement it and there already exists a fairly complete test suite ensuring interoperability of most aspects.

Last but not least, the JSON-LD 1.0 Processing Algorithms and API specification [168] also defines how JSON-LD can be converted to RDF's abstract syntax and vice versa. We will describe the basic principles of the algorithm separately in section 5.3.3.

5.3.2 Illustrative Example

JSON-LD is just a data interchange format. As such, it is by itself not enough to implement a Web API. It also needs a vocabulary defining the semantics of the concepts serialized as JSON-LD. Most concepts needed to express the data managed by the exemplary festival Web API are already defined by Schema.org; what is missing, however, is a vocabulary to describe the hypermedia controls necessary to implement a truly RESTful Web service. Thus, in practice, developers need to define their own proprietary concepts similar to how they define their own link relations if none of the standardized relation types fits.

The JSON-LD context in Listing 32 shows the concepts we need to implement our exemplary festival API. It includes properties to link from

```
{
  "@context": {
    "ex": "http://example.com/vocab#",
    "festivals": { "@id": "ex:festivals", "@type": "@id" },
    "orders": { "@id": "ex:orders", "@type": "@id" },
    "totalItems": "ex:totalItems",
    "member": "ex:member",
    "search": "ex:search",
    "template": "ex:template",
    "mapping": "ex:mapping",
    "variable": "ex:variable",
    "property": { "@id": "ex:property", "@type": "@vocab" },
    "@vocab": "http://schema.org/"
  }
}
```

Listing 32. The JSON-LD context used to describe representations of the exemplary festival API

```
{
  "@context": "/context.jsonld",
  "festivals": {
    "@id": "/festivals/",
    "search": {
      "template": "/festivals/?q={query}&date={date}",
      "mapping": [
        { "variable": "query", "property": "searchTerms" },
        { "variable": "date", "property": "startDate" }
      ]
    }
  },
  "orders": "/users/4812/orders/"
}
```

Listing 33. The festival API's main entry point using the context from Listing 32

the API's entry point to the collection of festivals and orders, properties to express those collections, and some concepts to realize the search functionality. For the sake of simplicity, we have tried to keep the number of concepts to a bare minimum. Features such as paging or the typing of the resources have thus been omitted. All the other concepts come directly from the Schema.org vocabulary which is why the context sets `@vocab` to `http://schema.org/`.

Using this context, the API's main entry point can be realized as illustrated in Listing 33. Just as the Atom service document acting as main entry point for SAPS-based services, the JSON-LD document in Listing 33 references the main collections. While the reference to the orders collection consists of just a relative IRI, the reference to the festivals collection also includes information about how the collection can be queried. The variables in the IRI template to query the collection are mapped to properties from both the proprietary vocabulary (`searchTerms`) and Schema.org (`startDate`). In contrast to the OpenSearch description as used by SAPS, the variables themselves are just string tokens. The mapping happens separately. The reason for this seemingly arbitrary and insignificant decision is that by doing so, the JSON-LD processor will automatically take care of the expansion to the full IRIs. If the template's variables were to represent the properties directly, the processing would have to be adapted to take care of the expansion. Additionally, the transformation to other RDF serialization formats would be made more difficult as the template would have to be transformed as well—which is something no off-the-shelf JSON-LD processor would be able to do.

```
{
  "@context": "/context.jsonld",
  "@id": "",
  "totalItems": 1,
  "member": [
    {
      "@id": "/festivals/umf84705#",
      "name" : "Ultra Music Festival",
      "description": "Outdoor electronic music festival",
      "startDate": "2014-03-28",
      "endDate": "2014-03-30",
      "performer": [
        {
          "@id": "/artists/t6159#",
          "name": "Tiësto"
        }
      ],
      "offer": [
        {
          "name": "General Admission",
          "sku": "umf84705-165",
          "price": "399.95"
        }
      ]
    }
  ]
}
```

Listing 34. The representation of a search result in JSON-LD

Provided the client "understands" the used vocabularies, the representation in Listing 33 provides enough information for the client to invoke a query to find a specific festival. The result of such a query might look like the document shown in Listing 34, which, apart from `@context` and `@id`, looks like an idiomatic JSON document as it can be found in numerous current Web APIs. The difference, however, is that it is completely self-descriptive. All properties apart from `totalItems` and `member` were reused from an already existing vocabulary (Schema.org), which reduces the risk of leaking implementation details and thus introducing unnecessary coupling. An additional advantage is that, since all these properties are already documented and widely used, they do not need to be further described by the API publisher.

These examples not only highlight JSON-LD's strengths but also reveal its weaknesses regarding RESTful Web APIs. On one hand, JSON-LD makes it simple to create representations that feel like idiomatic JSON, which is important to achieve wide adoption. In many cases, existing JSON representations require very little or no changes at all. On the other hand, JSON-LD by itself is not expressive enough to implement Web APIs. It needs vocabularies which define concepts to do so. The representation of the API's entry point in Listing 33 illustrates this problem. If a client does not know what the `orders` property stands for, all it can do is to try to dereference the IRI which is the value thereof or the property itself. Unfortunately, however, no vocabulary specifically designed for RESTful Web APIs exists yet to describe the fact that a new order can be created by POSTing a representation of the order to `/users/4812/orders/`. Thus, clients effectively need to be hardcoded against such properties similar to how clients for the Atom Publishing Protocol clients are hardcoded against the `edit` link relation or the `app:collection` element.

5.3.3 Integration into the Linked Data Cloud

Ignoring a few extensions such as the support of blank node properties or data indexing, JSON-LD represents an ordinary RDF 1.1 dataset serialization format. As such, it integrates seamlessly into the Linked Data cloud. If those features are not used, a lossless conversion of JSON-LD documents to other RDF serialization formats is possible. Conversions in the other direction, i.e., from any other concrete RDF serialization format to JSON-LD, are always possible. We specified the conversion to and from RDF in detail in the JSON-LD 1.0 Processing Algorithms and API specification [168]. Thus, we will confine ourselves to a high-level description of the process in this section. To keep the explanation simple, we will use the expanded and flattened document from Listing 30, which expresses the relationship of the two persons "Markus Lanthaler" and "John Doe".

All JSON properties have been transformed to the IRIs they are mapped to during expansion. Properties (and their values) which are not mapped to IRIs, keywords, or blank node identifiers are dropped during expansion. Flattening then removed the nesting by replacing nested nodes with node references, i.e., objects consisting of just an `@id`-member. Node objects that are not identified by an IRI or blank node identifier get assigned a newly minted blank node identifier. This is possible because blank node identifiers only have local scope and can thus be systematically replaced. The remaining steps to convert such a document to RDF's abstract syntax are straightforward.

Each JSON object either represents a node, i.e., an IRI or blank node with its associated properties, or a value, i.e., a literal. If it is a node, the value of the `@id` member represents the subject of the RDF triples which can be built by taking the remaining key-value pairs. The keys, i.e., the properties' names, represent the predicates, whereas the values represent the objects. Each JSON object having an `@id` member is transformed to an IRI or blank node, depending on whether its value begins with `_:` or not. If the object has an `@value` member, it represents an RDF literal along with its type (in the form of an `@type` member) or language (in the form of an `@language` member). It may also represent an ordered list, in which case the JSON object consists of an `@list`-member. Lists are converted to linked lists using the `rdf:first`/`rdf:rest` properties as specified by RDF Schema [65]. In case JSON objects with an `@graph` property are encountered, a named graph has to be created. The value of the `@id`-member of the JSON object containing `@graph` is taken as the graph name. The value of `@graph` represents the graph itself which is converted to triples as just explained. Named graphs cannot be nested and thus there is only ever one level of nesting in the flattened JSON-LD document as well.

The result of converting the document in Listing 30 to Turtle is shown in Listing 35. In order to illustrate the triples as interpreted by RDF's abstract syntax [63] we do not use any of Turtle's syntactic shortcuts.

```
<http://example.com/people/markus>
   <http://xmlns.com/foaf/0.1/name>
      "Markus Lanthaler"^^<http://www.w3.org/2001/XMLSchema#string> .

<http://example.com/people/markus>
   <http://xmlns.com/foaf/0.1/homepage>
      <http://www.markus-lanthaler.com/> .

<http://example.com/people/markus>
   <http://xmlns.com/foaf/0.1/knows>
      <http://www.markus-lanthaler.com/people/john> .

<http://example.com/people/john>
   <http://xmlns.com/foaf/0.1/name>
      "John Doe"^^<http://www.w3.org/2001/XMLSchema#string> .
```

Listing 35. The document shown in Listing 30 converted to Turtle

It should be noted that JSON-LD allows the serialization of generalized RDF, i.e., it represents a superset of RDF because it allows blank nodes to be used as properties. Unless a flag is set, the JSON-LD processing algorithms and API will, however, by default eliminate such triples when converting JSON-LD to RDF.

5.3.4 Summary and Lessons Learned

Similar to SEREDASj, JSON-LD tries to provide a smooth upgrade path from existing JSON-based services to self-descriptive services built on Linked Data principles. This is enabled by a 100% JSON-conformant syntax which creates idiomatic representations that are very similar to how JSON is typically used by Web developers. Developers therefore neither have to change their workflows nor their toolchains or programming libraries, which considerably lowers the entry barrier to publish Linked Data in the form of RESTful services. This is exactly what JSON-LD was designed for. It has built-in support for domain semantics yet its syntax is completely independent thereof. Instead of having to design new media types, which are often just specializations of existing syntaxes, developers can thus fully concentrate on defining and describing the application's domain semantics and its behavioral model when creating Web APIs. Since every concept gets assigned an IRI whose definition can be looked up, the need for out-of-band knowledge is eliminated as the semantics are brought in band. By leveraging existing vocabularies such as RDFS [65] and OWL [66], detailed machine-processable definitions that enable automated consistency checks of domain application protocols become possible. Furthermore, since JSON-LD documents can be directly interpreted as RDF, no complex mappings or algorithms are necessary to manipulate the data and to transfer the changes back to the server. The additional complexity due to the graph-based data model and the fact that there exist multiple valid representations for the same data is significantly outweighed by the achievable benefits in terms of loose coupling, evolvability, scalability, self-descriptiveness, and maintainability.

Given that the JSON-LD syntax specification, the processing algorithms and the API were put on the recommendation track at the World Wide Web Consortium (W3C) and thus became officially endorsed Internet standards, JSON-LD could become the lingua franca for Web APIs similar to how HTML is the dominant language on the human Web. Our experiments in the context of a large-scale Web of Things project [98], [188] and the positive feedback from numerous early adopters, some of which we will present in Chapter 6, attest that the presented approach is practical. In fact, JSON-LD could be a first step toward standardizing semantic RESTful Web services and form the basis for various efforts that previously could not seem to find any common ground. As we have shown, JSON-LD itself, however, is not a complete technology stack—it needs to be used with vocabularies that define the domain semantics. Therefore, in the next section we will introduce a lightweight vocabulary that can be used with JSON-LD to created truly RESTful, hypermedia-driven Web APIs.

5.4 Hydra

RDF has, despite its use of IRIs for identifiers, no inherent support for hypermedia. Whether an IRI is intended to be dereferenced or not depends implicitly on what it represents. FOAF's [108] `homepage` property, e.g., suggests that its values are dereferenceable IRIs. The Linked Data principles postulated by Berners-Lee [17] go a step further and recommend that all IRIs are dereferenceable (unlike all other RDF concrete syntax specifications, JSON-LD's specification [144] recommends the same). This enables the creation of large interconnected graphs of data. Most of these graphs, however, are read-only representations—just as most of the document-based human Web was read-only at the beginning. To change this and add hypermedia support to RDF-driven applications, a shared vocabulary able to describe affordances beyond simple dereferenceability is needed. Hydra is an attempt to define a minimal vocabulary to address these issues. By specifying a number of concepts that are commonly used in Web APIs it can be used as the foundation to build truly RESTful, hypermedia-driven services that can be accessed with generic clients.

Simply speaking, a RESTful Web API consists of a number of interlinked resources whereby each is identified by an IRI. In order to find its way through the resource space, a client has to understand the semantics of a hyperlink, i.e., be able to identify in which relation a resource stands to another resource. Typically, those relationships and resource types themselves are domain-specific but it is, nevertheless, possible to extract a number of such link relations and resource types that are generic enough to be applicable to a wide range of application domains. Collections as defined by, e.g., Atom, are a good example for this.

Web APIs use collections to reference a number of related resources. Taking a blog as example, most developers would probably choose to expose a collection containing all individual blog posts. Similarly, all comments related to a specific post, would be grouped in a dedicated collection. Such collections typically also expose functionality such as the creation of new resources by POSTing representations to the collection's IRI or searching for specific resources in the collection by accessing the collection with specific URL parameters describing the query criteria.

Unlike JSON-LD, which was a collaborative effort from the very beginning, Hydra was a personal project for a long time. This allowed us to iterate more quickly than would have been possible if several people would have been involved at this early stages. After having published the first more or less stable version of Hydra in we decided it was time to open the further development of Hydra to a wider public and established a W3C Community Group.

Most parts of this section have already been published in [96], [99], [169], [170].

5.4.1 Basic Concepts and Principles

The basic idea behind Hydra is to provide a vocabulary which enables a server to advertise valid state transitions. A client then proceeds through the service by looking at one response at a time, each time evaluating how best to proceed given its overall goal and the available transitions. The Hydra descriptions provide enough information for the client to construct HTTP requests manipulating the server's state in order to achieve a certain application-specific goal. Since all the information about the valid state transitions is exchanged in a machine-processable way at runtime instead of being hardcoded into the client at design time, clients can be decoupled from the server and are able adapt to changes more easily.

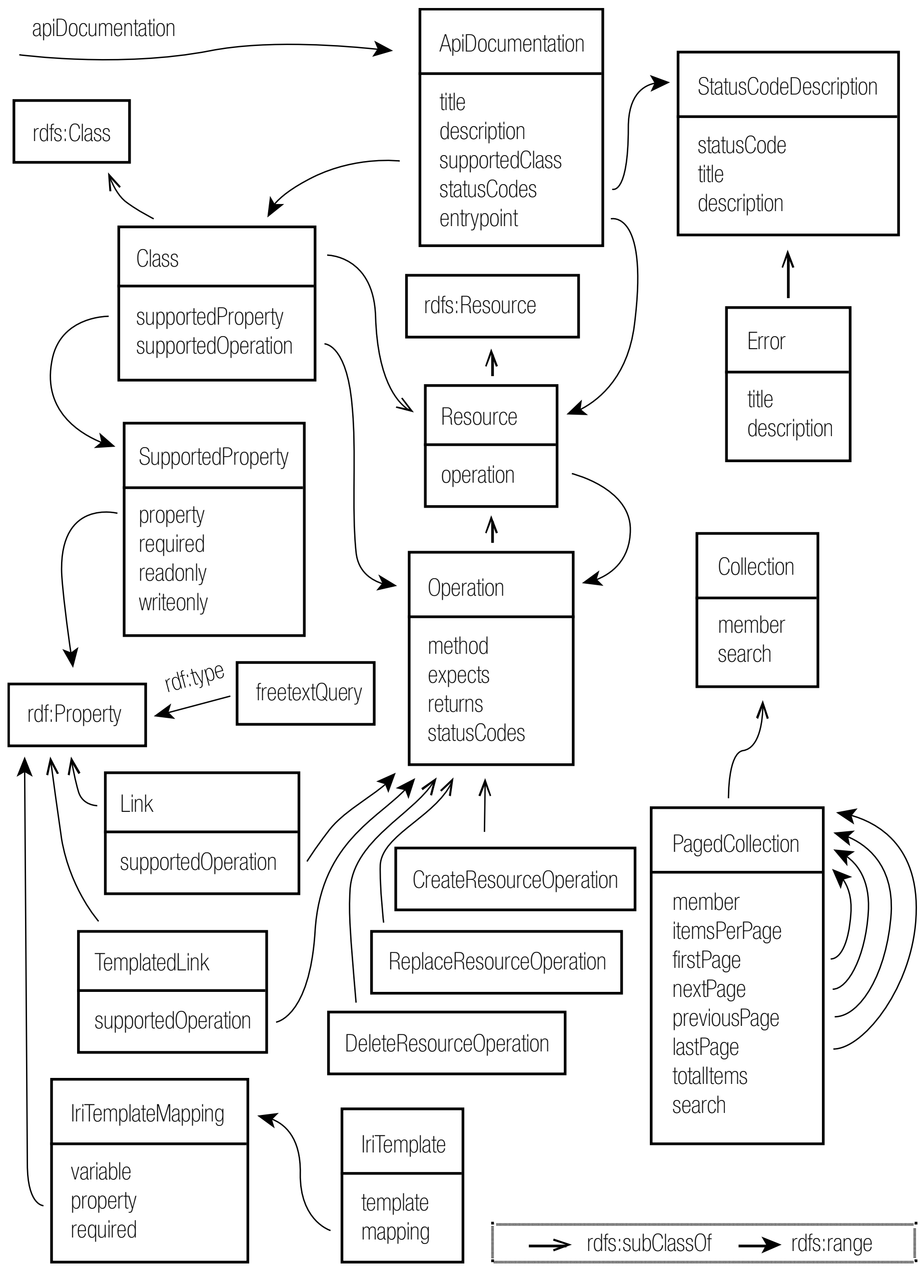


Figure 14. The Hydra core vocabulary

Figure 14 illustrates the Hydra core vocabulary (the figure's intention is to show how Hydra is used rather than its precise definition). At the center stands the `ApiDocumentation` class which represents, just as its name suggests, the documentation of a Web API. It enables a server to define the main entry point and to document the classes and properties as well the operations it supports. Furthermore, it enables HTTP status codes to be associated with additional information. Such descriptions may also be constructed and returned dynamically in response to client requests. This may sometimes be necessary as HTTP status codes are often not specific enough, making it difficult to understand the real cause of an error. For instance, a `400 Bad Request` response is rarely informative enough by itself.

Even though entities are identified by IRIs in RDF, clients cannot reliably assume that IRIs are dereferenceable. In fact, neither RDF itself nor RDF Schema or OWL defines a concept to describe dereferenceable IRIs. Hydra's `Resource` class, however, does just that. It is a subclass of RDF Schema's `Resource` class and can thus be used to signal a client that an IRI is dereferenceable and can be used to retrieve further information. This allows Linked Data to be distinguished from data where IRIs are used exclusively as identifiers. Similarly, the `Link` class can be used to define properties whose values are known to be dereferenceable IRIs.

It is not always possible for a server to create a complete link. For instance, links to query a server often require parameters which have to be filled at runtime by the client. To support such functionality, Hydra uses URI Templates [177]. An `IriTemplate` (URI templates are allowed to contain all characters that are legal in IRIs; for consistency we thus decided to name the class `IriTemplate` instead of `UriTemplate`) consists of a `template` and a number of `mapping` declarations. Each `IriTemplateMapping` maps a `variable` in the IRI template to a `property`. This allows a client to understand the meaning of the various variables and to replace them with concrete values in order to expand the IRI template to an IRI. Analogous to `Link`, there exists a property class `TemplatedLink` to create recognizable properties whose value is an `IriTemplate`.

To enable clients to interact with a Web API beyond simple GET requests, Hydra contains a notion of operations. An `Operation` represents the information necessary for clients to construct valid HTTP requests in order to manipulate the server's resource state. As such, each `Operation` consists of a required HTTP `method` and optional `expects` and `returns` types. Similar to the `ApiDocumentation` itself, operations may also document `statusCodes` that might be returned. This allows a developer to understand what to expect when invoking an operation. It is, however, not to be considered as an extensive list of all potentially returned status codes; it is merely a hint. Developers should expect to encounter other HTTP status codes and return types as well.

The alert reader might wonder why operations have no property to specify the target IRI. The reason for this is that operations are either bound to classes or link properties or directly associated with the resources they apply to. This means that the target IRI is communicated at runtime instead of being defined at design time. If an operation is bound to a class, it will apply to all its instances, which will be dereferenceable resources (they are ignored for blank nodes). Similarly, if an operation is bound to a `Link` or a `TemplatedLink`, it will apply to the corresponding IRI value.

A difficult design decision we had to make was how to inform a client which data a server expects for a certain operation. Classes would lend themselves but, as we discussed earlier, in RDF it is practically impossible to say which properties belong to a class. This, in turn, makes it impossible for a client to know which data it has to send to a server in order to achieve a certain goal. It also makes it difficult to inform a client (or a developer for that matter) what it might expect in responses from a server. We decided to choose the simplest and most pragmatic solution, i.e., to augment a class definition with `supportedProperty`, i.e., an enumeration of the properties known to be supported. This not only solves the problem at hand but also enables properties from other vocabularies to be reused directly.

Each `SupportedProperty` consists of a `property` and optionally some flags specifying whether it is `required`, `readonly`, or `writeonly`. Read-only properties cannot be modified by a client and are useful for information such as creation dates or authorship information that gets set by the server based on login credentials. Write-only properties, on the other hand, are useful for things like passwords that a client can change but not retrieve.

To ensure Hydra helps bootstrap Web API development, it includes a small number of commonly used concepts. Since a lot of APIs deal with basic CRUD functionality, Hydra has three built-in operation types, namely `CreateResourceOperation`, `DeleteResourceOperation`, and `ReplaceResourceOperation`. As their names suggest, they can be used to indicate to a client that an operation results in a resource being created, deleted, or replaced. Hydra does not restrict the mapping of these operation types to certain HTTP methods, which means that a concrete delete operation might be mapped to a `POST` request. This is an intentional design decision to not unnecessarily restrict Hydra's expressivity. The user is responsible for the mapping of operations to sensible HTTP requests respecting their semantics. It is purely the HTTP method which defines whether a method is idempotent or safe. The operation describes the result at a higher level of abstraction and can easily be reused across different Web APIs. This is one of the aspects which enable the creation of generic clients.

Similar to the predefined operation classes, Hydra defines classes for collections, another commonly used concept in Web APIs. A `Collection` is simply a container pointing to a number of `member` items. In Hydra, each of those members is a dereferenceable `Resource`. Since it is frequently desired not to serve the whole collection at once, but to separate it into pages instead, Hydra also defines a specialized `PagedCollection`. Additional to its `member` items, it may also specify the number of `itemsPerPage`, the `totalItems` and links to the `firstPage`, the `nextPage`, the `previousPage`, or the `lastPage`. This way, a client can easily navigate through a collection. Furthermore, Hydra's `search` property, whose value is an `IriTemplate`, can be used to query such a collection. The only property Hydra defines to use in such a mapping is `freetextQuery` but of course properties defined by other vocabularies can be used as well.

Table 1 and Table 2 summarize the most important information about the classes respectively the properties that Hydra defines.

Table 1. The classes defined by Hydra

Class	**Subclass of**	**Description**
`Resource`	`rdfs:Resource`	The class of dereferenceable resources.
`Class`	`hydra:Resource`, `rdfs:Class`	The class of Hydra classes. Hydra classes and their instances are dereferenceable resources.
`ApiDocumentation`	`hydra:Resource`	The Hydra API documentation class
`SupportedProperty`	`hydra:Resource`	A property known to be supported by a Hydra class.
`Operation`	`hydra:Resource`	An operation.

Class	Subclass of	Description
`CreateResource-Operation`	`hydra:Operation`	A `CreateResourceOperation` is an HTTP operation which expects an input of the type specified by `hydra:expects` and creates a resource of the type specified by `hydra:returns`.
`ReplaceResource-Operation`	`hydra:Operation`	A `ReplaceResourceOperation` is an HTTP operation which overwrites a resource. It data of the type specified in `hydra:expects` and results in a resource of the type specified by `hydra:returns`.
`DeleteResource-Operation`	`hydra:Operation`	A `DeleteResourceOperation` is an HTTP operation that deletes a resource.
`Collection`	`hydra:Resource`	A collection holding references to a number of related resources.
`PagedCollection`	`hydra:Collection`	A `PagedCollection` is a subclass of Collection with the only difference that its members are sorted and only a subset of all members are returned in a single `PagedCollection`. To get the other members, the `nextPage` and `previousPage` properties have to be used.
`Link`	`hydra:Resource`, `rdf:Property`	The class of properties representing links.
`TemplatedLink`	`hydra:Resource`, `rdf:Property`	A templated link.
`IriTemplate`	`hydra:Resource`	The class of IRI templates.
`IriTemplate-Mapping`	`hydra:Resource`	A mapping from an IRI template variable to a property.

Class	Subclass of	Description
`StatusCode-Description`	`hydra:Resource`	Additional information about a status code that might be returned.
`Error`	`hydra:Status-CodeDescription`	A runtime error, used to report information beyond the returned status code.

Table 2. The properties defined by Hydra

Property	Domain	Description
	Range	
`apiDocumentation`		A link to the API documentation
	`hydra:ApiDocumentation`	
`entrypoint`	`hydra:ApiDocumentation`	A link to main entry point of the Web API
	`hydra:Resource`	
`supportedClass`	`hydra:ApiDocumentation`	A class known to be supported by the Web API
	`hydra:Class`	
`statusCodes`		Additional information about status codes that might be returned by the Web API
	`hydra:StatusCode-Description`	
`statusCode`	`hydra:StatusCode-Description`	The HTTP status code
	`xsd:integer`	
`supportedProperty`	`hydra:Class`	A property known to be supported by the Hydra class
	`hydra:SupportedProperty`	
`property`		A property
	`rdf:Property`	

Property	Domain / Range	Description
`required`	Domain: Range: `xsd:boolean`	True if the property is required, false otherwise.
`readonly`	Domain: `hydra:SupportedProperty` Range: `xsd:boolean`	True if the property is read-only, false otherwise.
`writeonly`	Domain: `hydra:SupportedProperty` Range: `xsd:boolean`	True if the property is write-only, false otherwise.
`supported-Operation`	Domain: Range: `hydra:Operation`	An operation supported by instances of the Hydra class or the target of the Hydra link
`operation`	Domain: `hydra:Resource` Range: `hydra:Operation`	An operation supported by the Hydra resource
`method`	Domain: `hydra:Operation` Range: `xsd:string`	The HTTP method.
`expects`	Domain: `hydra:Operation` Range: `hydra:Class`	The information expected by the Web API.
`returns`	Domain: `hydra:Operation` Range: `hydra:Class`	The information returned by the Web API on success
`title`	Domain: Range: `xsd:string`	A title, often used along with a description.
`description`	Domain: Range: `xsd:string`	A description.
`member`	Domain: `hydra:Collection` Range: `hydra:Resource`	A member of the collection

Property	Domain / Range	Description
totalItems	hydra:Collection / xsd:integer	The total number of items referenced by a collection or a set of interlinked `PagedCollections`.
itemsPerPage	hydra:PagedCollection / xsd:integer	The maximum number of items referenced by each single `PagedCollection` in a set of interlinked `PagedCollections`.
firstPage	hydra:PagedCollection / hydra:PagedCollection	The first page of an interlinked set of `PagedCollections`
lastPage	hydra:PagedCollection / hydra:PagedCollection	The last page of an interlinked set of `PagedCollections`
nextPage	hydra:PagedCollection / hydra:PagedCollection	The page following the current instance in an interlinked set of `PagedCollections`
previousPage	hydra:PagedCollection / hydra:PagedCollection	The page preceding the current instance in an interlinked set of `PagedCollections`
search	/ hydra:IriTemplate	A IRI template that can be used to query a collection
freetextQuery	/ xsd:string	A property representing a freetext query.

Property	Domain / Range	Description
`template`	`hydra:IriTemplate` / `xsd:string`	An IRI template as defined by RFC6570.
`mapping`	`hydra:IriTemplate` / `hydra:IriTemplate-Mapping`	A variable-to-property mapping of the IRI template.
`variable`	`hydra:IriTemplate-Mapping` / `xsd:string`	An IRI template variable

5.4.2 Illustrative Example

As explained in the previous section, Hydra defines several key concepts for the creation of hypermedia-driven Web APIs. As shown in Listing 36, we can therefore replace almost all proprietary concepts in the context in Listing 32 with concepts defined by Hydra. This simple change is enough to eliminate most of the out-of-band knowledge required to use the API presented in section 5.3.2. Any client supporting Hydra is thus able to expand the IRI template to query the festivals collection in Listing 33 without requiring any additional information. Similarly, the

```
{
  "@context": {
    "ex": "http://example.com/vocab#",
    "festivals": { "@id": "ex:festivals", "@type": "@id" },
    "orders": { "@id": "ex:orders", "@type": "@id" },
    "hydra": "http://www.w3.org/ns/hydra/core#",
    "totalItems": "hydra:totalItems",
    "member": "hydra:member",
    "search": "hydra:search",
    "template": "hydra:template",
    "mapping": "hydra:mapping",
    "variable": "hydra:variable",
    "property": { "@id": "hydra:property", "@type": "@vocab" },
    "@vocab": "http://schema.org/"
  }
}
```

Listing 36. Hydra replaces most proprietary concepts of the context shown in Listing 32

```
{
  "@context": "http://www.w3.org/ns/hydra/core",
  "@id": "http://example.com/vocab#orders",
  "@type": "Link",
  "rdfs:range": "Collection",
  "title": "The orders collection",
  "description": "A link to the current user’s order collection.",
  "supportedOperation": {
    "@type": "CreateResourceOperation",
    "title": "Create a new order",
    "method": "POST",
    "expects": "http://schema.org/Order",
    "returns": "http://schema.org/Order"
  }
}
```

Listing 37. The definition of the `orders` property

rest of the API’s functionality can be described in a machine-readable way using Hydra.

The proprietary `orders` property, for instance, can be defined as a link pointing to a collection. Furthermore, it is possible to describe that a `POST` request to that collection can be used to create a new order entity. The whole definition can be seen in Listing 37.

This example reveals an interesting question, namely what the semantics of a specific HTTP request are. The documentation in Listing 37 only tells that a new resource is being created if an HTTP `POST` request with a payload containing an `Order` entity is invoked on the `orders` collection. These are enough semantics to describe simple CRUD-style interfaces. In our example, however, the consequence of sending such an HTTP request is that a ticket is being ordered. To convey those semantics to a client, a more specific operation type has to be used. Obviously this is out of scope for Hydra itself but nothing prevents a developer to either subclass Hydra’s `CreateResourceOperation` to specialize it or to reuse a concept someone else has already defined. Luckily in a recent initiative (long after Hydra has been first presented) it was decided to add “actions” to Schema.org which can be leveraged by Hydra-powered APIs. Thus, we can simply type the operation as `http://schema.org/BuyAction`, which is defined as the “act of giving money to a seller in exchange for goods or services rendered”. Since Hydra and Schema.org can be used so well together, we have been discussing the inclusion of Hydra into Schema.org or a closer alignment with the Schema.org team for several months. Recently we also published a first draft proposing the integration of a subset of Hydra directly into Schema.org [189].

The operation in Listing 37 expects and returns an `Order` entity. This information by itself is not enough, as RDF vocabularies typically do not link from classes to properties but vice versa. Thus, the Hydra API documentation also documents the properties known to be supported by instances of the `Order` class as shown in Listing 38. To keep the examples simple, the orders in our exemplary festival API consist of just the ticket's SKU number, its price, and a link to complete the payment. While the SKU number can be set by a client when creating the order, the price and the payment URL are filled in by the server and are therefore marked as read-only properties.

```
{
  "@context": "http://www.w3.org/ns/hydra/core",
  "@id": "http://schema.org/Order",
  "@type": "Class",
  "title": "A ticket purchase order",
  "description": "All we need is the ticket’s SKU.",
  "supportedProperty": [
    {
      "@type": "SupportedProperty",
      "property": "http://schema.org/sku",
      "required": true
    },
    {
      "@type": "SupportedProperty",
      "property": "http://schema.org/price",
      "readonly": true
    },
    {
      "@type": "SupportedProperty",
      "property": "http://schema.org/paymentUrl",
      "readonly": true
    }
  ]
}
```

Listing 38. The definition of the `Order` class

This describes the whole API in a machine-readable and interoperable manner. The few remaining proprietary concepts such as the `festivals` and `orders` properties referencing the corresponding collections could be further described using other, already existing and standardized vocabularies such as RDF Schema and OWL. For instance, using RDF Schema's `range` [65] along with OWL restrictions [190] makes it possible to describe that the `festivals` property points to a Hydra `Collection` whose `member` items are instances of Schema.org's `Festival` class. Listing 39 shows how such information can be expressed in JSON-LD (omitting the context definition).

```
{
  "@id": "http://example.com/vocab#festivals",
  "rdfs:range": [
    "hydra:Container",
    {
      "owl:equivalentClass": {
        "@type": "owl:Restriction",
        "owl:onProperty": "hydra:member",
        "owl:allValuesFrom": "schema:Festival"
      }
    }
  ]
}
```

Listing 39. Specifying the type of the Hydra collection members of the `festivals` property

5.4.3 Integration into the Linked Data Cloud

Hydra is an ordinary RDFS/OWL vocabulary and as such, it integrates seamlessly into the Linked Data cloud. In fact, it can be used to improve data in the Linked Data cloud by explicitly expressing which IRIs are dereferenceable and which are just used as identifiers. Furthermore, it allows the enrichment of the typically read-only data found in the Linked Data cloud with affordances in order to support read/write and other, more sophisticated interaction models. This opens the door for Linked Data to many applications that previously have been mainly reserved for classic Web APIs. In this context is also worth to highlight again that Hydra can be used with any concrete RDF syntax; it does not depend on JSON-LD.

5.4.4 Summary and Lessons Learned

Normally, when using Linked Data, a machine-client has no choice but to try whether a specific IRI dereferences to a document providing further information about the concept or not. The reason is that RDF lacks any notion of hypermedia or interaction models since IRIs are solely used as identifiers. This is one of the most fundamental hurdles to overcome when combining the Representational State Transfer (REST) architectural style with the Linked Data principles. Other formats such as HTML have multiple hypermedia controls that can be embedded in the representations returned by a server. Hydra therefore provides generic concepts such as links and operations that can be used to augment Linked Data representations with actionable information. Clearly, this goes far beyond what is achievable with the traditional definition of media types as the descriptions can be reasoned with by computer programs.

An important principle to follow when developing clients using such information is to be prepared that everything might change or even break. The machine-readable description of the API should be retrieved and analyzed at runtime and not embedded directly into the client. All the documentations about things such as available operations or possible errors should be seen as hints rather than static contracts. At the moment they are used they might already be outdated and the server might respond in a totally different way than expected. Clients should be able to detect and possibly recover from such errors. As a last resort, the client might need to ask its user for assistance, report an error, or automatically file a bug report.

One of the design decisions was whether these controls should be optimized to be embedded directly into every single representation, or whether a separate document should be the preferred way to describe those affordances. We choose the latter approach for a number of reasons. First of all, the responses from most Web APIs are rather uniform, meaning that in a Web API there usually exist a small number of response "types" that are all completely consistent. This is quite different from human-facing Web sites where different pages differ heavily in order to keep their users engaged. Secondly, in contrast to a human user, a machine agent has no problems to remember a number of affordances and to apply them consistently to elements contained in responses. A similar approach would be prohibitive on the human Web since the resulting cognitive load put on humans would be way too heavy. Finally, an approach collecting the affordances supported by a server in a single description document is what programmers are already familiar with. This is not only the predominant form of documentation for Web APIs, but for APIs in general as it allows developers to quickly understand the capabilities of a server or programming library without having to traverse the whole state space.

This knowledge concentration of supported affordances in a central description leads to another interesting question that is left open for most current Web APIs, namely how to discover that description. The typical approach is to fall back to a human operator which browses an API publisher's website to locate the API description. That is a valid approach given that the API description is rarely machine-readable anyway. However, if the API document is machine-readable, as it is the case for Hydra, it would be a serious limitation if the discovery of that description document would require human intervention. Therefore, Hydra uses an HTTP Link header [57] to direct a client to the corresponding API document. The link relation used in such a Link header corresponds to the IRI of Hydra's `apiDocumentation` property. This enables the dynamic discovery of the API description at runtime and works across different APIs. As soon as an API links to resources of a different API, a client can recognize the different API description and adapt itself accordingly. Since the API description is not bound to the API's host it becomes possible to rely on central, standardized API descriptions resulting in an even looser coupling between the client and the server. Furthermore, RDF's use of globally unique identifiers allows parts of API descriptions to be shared and reused, which improves interoperability and reduces costs. Hydra's predefined operation types are a first step in that direction. We believe that it is possible to extract and standardize similarly reusable concepts for a wide variety of application domains and we are already working with both the Schema.org and the Activity Streams community to do so.

Considering Hydra's focus on reusability of concepts between different APIs, the question may arise why Hydra itself does not rely more on other existing vocabularies apart from RDF Schema and OWL. The reason is simple. Hydra tries to address Web developers which do not necessarily have profound knowledge of Semantic Web technologies. As such, a simple, coherent, and self-contained vocabulary is easier to understand. Using, e.g., OWL class expressions [66] to specify required properties in the request class used in an operation would simply be too complex for average Web developers. In other cases, the potential reuse from vocabularies is too small to be justifiable. The HTTP vocabulary [191] is such an example. The only overlapping concepts are Hydra's HTTP `method` and `statusCode` properties. Such a small overlap does not represent a reasonable argument to include a dependency to a vocabulary. We did, however, align Hydra's concepts with the corresponding concepts in the HTTP vocabulary, which results in almost the same benefits without producing an unnecessary dependency.

As soon as IRIs in RDF are dereferenced to retrieve further information about a resource, the question of whether the IRI identifies the returned representation or some abstract entity arises. Hydra is deliberately silent on this issue because it is an aspect that has to be solved at a different layer. All Hydra is concerned about is to describe potential state transitions by providing concepts to describe the interaction model of a Web API. As history has shown, discussions around issue httpRange-14 [81] quickly transform into painful philosophical debates making it difficult to work on technical solutions mitigating the inherent problem. We do not believe that the Technical Architecture Group's resolution [192] is a practical way forward as the recommended mechanism is brittle and costly to implement. A quick look at the Web also reveals that it is rarely implemented and therefore cannot be relied upon. Unfortunately, until an agreed solution to this problem has been found, the only sensible advice that can be given to developers is to clearly document what IRIs and properties associated to them denote. As already mentioned earlier, Tennison's blog post describing punning [83] and her "URLs in Data Primer" [84] gives some good advice and shows how it might be done in a machine-readable way.

5.5 Discussion

In this chapter we presented the approaches we developed to simplify and standardize the creation of truly RESTful Web APIs. While JSON-LD has become an official and widely used W3C standard, Hydra's further development has been moved to a W3C Community Group to open it to the wide public. At the same time, we are already discussing its inclusion (or parts thereof) into Schema.org.

In this section, which is based on previous work published in [25], we will discuss how JSON-LD and Hydra can be used for the domain-driven design and implementation of RESTful Web APIs. We will discuss a number of crucial design decisions and show how it is possible to create Web APIs in which almost all aspects are documented in a machine-processable form. Not only does this result in an improved reusability of domain models, either as a whole or parts thereof, but also in composable contracts that enhance the interoperability between systems. The fact that all data, including the data describing the system, is managed in a unified form allows testing to commence in much earlier stages of the development process and typically increases the productivity of developers and the quality of the built solution.

Data Modeling

The first and most important step when creating a RESTful API, or an application in general, is to understand the problem domain. Based on that understanding it is then possible to design the data model representing the various domain entities and their properties. The shared understanding gained by the formalization of the data model is fundamental to enable collaboration between the various stakeholders working on the realization of a Web API. Given that REST is a resource-oriented architecture it should not come as a surprise that the modeling of the resources, i.e., the exposed entities, is a fundamental part of the design process. The outcome of this process should be a formal description of the entities, their properties, and their relationships in the problem domain. This is a task RDF has proven to be very successful at.

Standardized RDF vocabularies such as RDF Schema [65] or the Web Ontology Language (OWL) [66] formalize the necessary concepts to describe an API's data model or, more formally, ontology. The advantage of using RDF, which is based on a simple graph-based data model, is that the description can be created in exactly the same format as all other data in the system. The resulting unified view makes it possible to use the same tools for both the definition of the data models and the data itself. Another advantage of an RDF-based system is the drastically simplified reuse of domain models—either as a whole or parts thereof. Such reuse not only reduces the inherent costs and risks but also results in concrete benefits in terms of interoperability and adoption. RDF's data model uniquely embraces the inevitable heterogeneity encountered when working with data at Web scale. Furthermore, its schemalessness ensures the required agility in today's fast-moving world.

The most important aspect developers have to keep in mind is to not expose implementation details. That means that a change in the implementation on the server should not result in changes in the API it exposes over the Web. In practice, this means that developers should introduce an abstraction layer decoupling the internals from the data exposed in the Web API. There exist a number of well-known design patterns to achieve that; e.g., the Adaptor or the Composite pattern [193]. The fact that there exists a generic client (which we will present in the next chapter) from the very beginning allows API "usability tests" to be run similar to the usability tests that are typically done for Web sites. This helps to ensure that the API is usable without knowledge of server internals.

From a Linked Data perspective, a vital principle is to reuse existing vocabularies as much as possible. This allows code reuse on the client-side and simplifies data integration. Nevertheless, developers often want to keep full control over the vocabularies they use to provide a unified experience. In such cases, specific concepts should either be sub-typed or declared as being equivalent to concepts in existing vocabularies. This allows more elaborated clients to interpret the data even if they only support the already existing vocabulary. There are significant research efforts to support users in this mapping process, which is typically referred to as ontology alignment.

Related to the reuse of existing vocabularies is the reuse of existing instance data. Just as Web sites typically link to other related Web sites, data exposed by a Web API should link to other relevant data on the Web; otherwise services will continue to remain islands in the vast information sea of the Web. This is also a cost-effective opportunity for developers to provide their customers with additional data outside of their main business focus. As paradoxical as it may sound, the more data there is, and the more interconnected it is, the easier it becomes to integrate it with other data.

After the data model has been defined, it has to be decided how the data is serialized. Fortunately, there exist already a number of serialization formats for RDF. As JSON has become the prevalent serialization format used in Web APIs, it clearly makes sense to choose a format such as JSON-LD, which combines the best of both worlds the simplicity of JSON with the semantic expressivity of RDF. A special challenge lies in the fact that, in contrast to trees as used in traditional JSON, graphs can be serialized in a number of ways while still expressing the same data. While this imposes no issues for clients processing the data as JSON-LD, it requires special attention if JSON-only clients that rely purely on the structure of the serialized data have to be supported as well. The solution is to formalize the conventions used to serialize the data and document them in a profile as described in section 2.2.1. The JSON-LD processing algorithms [168] and framing [187] make it possible to define these profiles declaratively and to automatically reshape documents to bring them into the desired shape. Both JSON-only and JSON-LD aware clients can then seamlessly work with the same data representations.

Behavioral Modeling

The data model defines how data is represented in the system. This, in a sense, provides a static view of the system. To be able to access and manipulate data through an API, the domain application protocol [46], or more formally speaking, the behavioral model needs to be defined as well. Despite significant research and development efforts, most Web APIs are still solely documented in the form of human-targeting, natural-language documents. Since such documentations do not represent machine-processable information, the creation of generic clients is made almost impossible and the results are costly and hard to maintain. To address these issues we designed Hydra, a lightweight vocabulary to capture and document the behavioral model of hypermedia-driven Web APIs in a machine-processable way.

Since Hydra makes the affordances supported by the various resources exposed by a Web API explicit, it becomes possible to either build machine agents that navigate Web APIs completely autonomously or to create generic programming libraries at a much higher level of abstraction that simplify developers' lives. Given that all the descriptions are represented in the same format as the data itself, even the code to access an API can be transformed to a declarative description that can be analyzed and worked with using the same tools—a very powerful feature often referred to as the Principle of Least Power [194].

The combination of a formal data model and a holistic documentation of the behavioral model based on Hydra enable the creation of declarative contracts capturing all aspects of a Web API. It is worthwhile to note that, in the spirit of domain-driven design, it is possible to map the concepts defined in the model to those in the code implementing it. Thus, it would be possible to automatically generate code stubs from these descriptions. Automatic code generation, however, always imposes the risk of either introducing unnecessary coupling or leaking implementations details, which is especially risky if the contract is owned by the server. Solutions based on JSON-LD and Hydra mitigate this issue by allowing the problem domain to be decomposed into smaller subproblems that are significantly easier to standardize, which shifts the coupling to a central standard (or a combination of multiple standards in the form of a profile).

Test Early, Test Often

While it is important to test early in the development process it is often disproportionally difficult to do so when developing Web APIs. Apart from low-level HTTP libraries, there typically exist no off-the-shelf tools assisting developers in testing their API. The situation is similar when developing API clients. Given that the proposed approach provides a unified view of the system, where all information is represented in the same format, testing is drastically simplified.

The existence of standardized tools allows the verification of different aspects of the system at very early stages—way before the system has been implemented as a whole. This reduces risks and costs while, at the same time, improving the quality of the system. Using off-the-shelf quad stores, e.g., it is possible to ensure that the data model is expressive enough and structured in a way to facilitate its usage by the various stakeholders. By augmenting the behavioral model with sample responses for the various operations, it becomes possible to easily create mock services that can help in developing clients even when the server does not exist yet. Just as everything else, the test cases become an integrated part of the data providing a holistic view of the system. This also allows verifying that all required interactions are supported by the system being built.

To further streamline and assist the development of Web APIs, we developed generic clients for Hydra-based services which can be used to run API "usability tests" similar to usability tests as usually used for Web sites. This helps to ensure that the API is usable without knowledge of server internals. Both the human-facing single-page Web application HydraConsole, which we will present in the next chapter, and an early version of the generic programming library HydraClient have been released as open source software [170].

Documenting Services

It takes time to convince developers to use such a new approach to build their systems. Thus, everything that allows an iterative introduction of these techniques helps to foster adoption. It is, at least for the foreseeable future, still important to provide human-targeting documentation in addition to the machine-processable service descriptions. Most developers are still better in understanding prose than formal descriptions and proofs.

The process outlined in the previous sections has the unique advantage that a lot of the otherwise implicit information about the system is explicitly expressed in a machine-processable form. The information from the data model and behavioral model can be used to automatically generate large parts of human-targeting documentations. By utilizing technologies such as HTML and RDFa or HTML with embedded JSON-LD, it is even possible to combine the machine-processable and the human-targeting documentation into a single document. Since most of the lower-level details are either standardized or already documented, humans can thus focus on augmenting the documentation with information that really matters for developers: the rationales behind design decisions, the assumptions made, the mental models, and the overall goals of a Web API.

Chapter 6

Evaluation

In this chapter we will evaluate JSON-LD and Hydra by looking at them from different angles. We will begin by evaluating which of the problems discussed in Chapter 3 have been addressed. Then we will demonstrate how easily the proposed approach can be integrated into current Web frameworks. This is a very important aspect for the acceptance of new technology. If the integration is too difficult or even impossible, developers will be reluctant to use these technologies as they have to discard their existing work and start from scratch. Consequently, very few developers would decide to build Web APIs based on JSON-LD and Hydra. While this relates mainly to the server side and is thus of most interest for API publishers, the client side must also not be ignored. In fact, most of the problems of current Web APIs manifest themselves on the client side and not on the server side. Thus, in section 6.3 we will present a prototype of a completely generic client for JSON-LD/Hydra-powered Web APIs, which supports browsing the data exposed by the API and interacting with the various resources. As we will see, all the information to render the user interface is retrieved dynamically at runtime. It thus represents a highly adaptive solution similar to Web browsers. Given that JSON-LD has already been well adopted and Hydra is starting to gain traction, we will present a number of early adopters from both the industry and academia using JSON-LD and Hydra for public as well as internal Web APIs in section 6.4 before we conclude the chapter with some final remarks.

The sections 6.2 and 6.3 in this chapter are based on previous work published in [99] and [169].

6.1 Problems Addressed

As discussed in detail in Chapter 3, current Web APIs and Semantic Web technologies suffer from a number of problems. The main issues are that current Web APIs often rely on proprietary data formats and models and that the contracts are documented solely in natural language. This makes them inaccessible for machines and means that they are mostly written manually which is a tedious and error-prone process. From the perspective of the REST architectural style, these issues mostly stem from violations of two important architectural constraints, namely the requirement of messages to be self-descriptive and the usage of hypermedia as the engine of application state. Consequently, it is almost impossible to create generic, standardized tooling support for Web APIs similar to how standardized browsers exist for the human Web. Semantic Web technologies address some of these issues but are perceived as overly complex and have no inherent hypermedia support (IRIs in RDF are, strictly speaking, not hyperlinks but identifiers), which typically leads to read-only interfaces to the data.

The main idea underlying this thesis is to bridge the gap between technologies used in RESTful Web APIs and Semantic Web technologies. After several experiments, this led to the development of JSON-LD and Hydra that address the discussed problems.

RDF underpins the proposed solution by defining a simple, yet powerful and expressive data model. JSON-LD allows RDF to be serialized as JSON, the prevalent data format in current Web APIs. As illustrated in the previous chapter, in most cases JSON-LD documents look almost exactly the same as their JSON counterparts but are completely self-descriptive. Developers therefore do not have to spend any effort on defining proprietary data formats or data models. Instead, they can focus on building their solution around standardized and interoperable technologies. If desired, it also opens the door to other Semantic Web technologies such as quad stores, SPARQL query engines, and reasoners. Unlike other Semantic Web technologies, however, JSON-LD does not force developers to use them. It also works well with popular NoSQL solutions such as MongoDB [195] or Elasticsearch [196]. This makes it a highly flexible technology which looks familiar to most Web developers and provides a smooth upgrade path for existing infrastructure investments. We are confident that JSON-LD, together with the clear separation of the data model from the various serialization formats and the overall much more accessible RDF 1.1 specifications, will help to alleviate Semaphobia, the fear of developers to use Semantic Web technologies as described in section 3.4.

Hydra extends JSON-LD with hypermedia controls going far beyond simple hyperlinks. This enables the machine-readable communication of affordances at runtime instead of having to rely on static contracts written in natural language at design time. At the same time, the structured nature of these descriptions and the fact that they are based on Linked Data principles improves the reusability of definitions and thus reduces the need to manually write documentation. Developers can therefore reuse concepts defined by Schema.org instead of having to design their own vocabulary. This not only reduces the amount of work necessary to design a Web API but also improves interoperability. Furthermore, it prevents the leakage of internals, which in turn reduces the coupling between clients and the server.

The fact that all representations returned by a Web API based on JSON-LD and Hydra are self-descriptive and contain actionable hypermedia affordances makes it possible to implement powerful generic clients instead of having to rely on specialized clients or low-level HTTP libraries and tools such as cURL [197]. Given also that the API descriptions are just data, they can be used in various ways as discussed in section 5.5. It is the usage of IRIs as unambiguous and globally-valid identifiers, RDF's generic data model, and the machine-readable semantics that enable serendipity. Similar to mashups, developers will find ways to use the data in unanticipated ways and integrate it with other data to make it even more useful.

The clear separation of concerns and the fact that JSON-LD, the data format, and Hydra, the vocabulary, can be used independently is another important aspect fostering serendipity. Hydra can, e.g., also be used with other RDF serialization formats such as Turtle, thus turning hypermedia into a first-class citizen in Linked Data in general. Hydra clearly describes which IRIs in an RDF graph are just identifiers and which IRIs are, at the same time, hyperlinks that are intended to be interacted with. Consequently, it renders it possible to turn the currently mostly read-only Linked Data cloud into fully interactive Web APIs or to seamlessly integrate Web APIs into the Linked Data cloud.

6.2 Ease of Integration into Web Frameworks

Armed with JSON-LD and Hydra, we developed a prototype to demonstrate the feasibility of the approach presented in the previous chapter. It shows how easily the proposed building blocks can be integrated in real-world systems.

The prototype is based on Symfony2 [198], a Web development framework implemented in PHP [199]. It is, as most other current Web frameworks, based on the Model-View-Controller (MVC) [200] design pattern. By separating the presentation of information from its processing, MVC improves code reusability and separation of concerns. The models represent the relevant entities in the system, the views (typically defined by templates) are used to create representations of those entities, and the controller is responsible for processing inputs, manipulating the models, and finally returning an updated representation by using the associated views. Symfony2 further modularizes the code by Page Controllers, which are only responsible for certain requests [110]. Symfony2's HttpKernel and Routing components parse the received HTTP request, extract the request URL and method, and then pass the processing to a specific page controller, which in turn invokes specific models and views.

Figure 15 illustrates how Symfony2 processes an HTTP request. Incoming requests are parsed by a front controller and subsequently passed on to the framework's kernel. The kernel then invokes its routing component to retrieve the page controller responsible for the requested resource. Finally, the page controller constructs a response which is sent back to the client. Typically, the controller uses various models and a view consisting of one or more templates to construct the response.

While for human-facing Web sites the view layer is crucial and the templates vary widely, it is rarely required in Web APIs. Instead, for Web APIs the view layer is typically much simpler and consists of just a

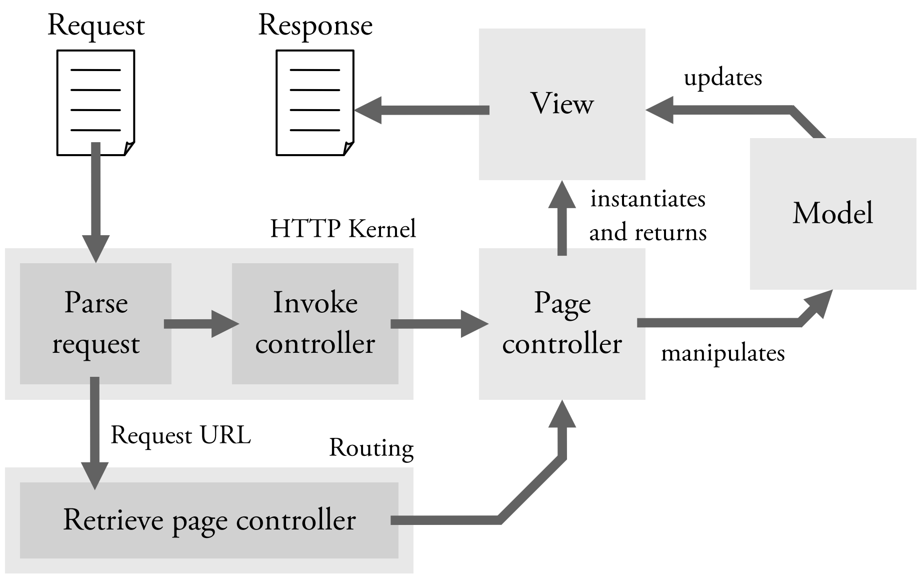


Figure 15. Symfony2's request-response flow

serializer turning the models, i.e., the entities, directly into representations following a specific format. The prototype we implemented thus replaces Symfony2's default templating engine with a serializer. Thus, instead of having to create templates to render the responses, developers can directly serialize the entities as JSON-LD. In fact, developers do not even need to call the serializer manually as the prototype directly hooks into Symfony2's request lifecycle to serialize the return value of the controller function. So, instead of having to return a response object, developers can choose to simply return the entity and our prototype will take care of the serialization.

The serialization component relies on code annotations to control the serialization of entities and their documentation. While this is more complex than simply serializing all public members of an entity (not least because PHP has no built-in support for annotations), it provides the flexibility that is often required in practice. Not all members should be exposed (all the time) and sometimes transformations, such as converting a numeric identifier into a URL, are necessary. The advantage of using annotations is that the information is kept close to the source code it describes, which makes it much easier to keep the two in sync. Symfony2 developers are generally used to annotations because several crucial components such as the routing component [201] or Symfony2's default object-relational mapper Doctrine [202] support them and recommend their use. That being said, it is worth noting that the prototype has been designed to support other mapping mechanism as well. If developers prefer to define the mapping from the internal objects to external representations by using, e.g., XML files, all they have to do is to implement a driver that reads those files and populates a class metadata registry with the extracted information.

The annotations not only specify which properties are to be exposed, but also describe the affordances supported by the entities. Thus, in addition to the serialization of entities as JSON-LD, it becomes possible to generate a machine-readable API documentation based on Hydra. As we will demonstrate in section 6.3, this makes it possible to create completely generic clients that are, e.g., able to automatically render forms in order to gather the necessary data for the creation of valid requests or to provide additional information about the semantics of the representations returned by the service. Since this data closely resembles the information available in current Web API documentations, exactly the same data can also be used to automatically generate such documentation.

To make its integration as simple as possible, we realized the prototype integrating JSON-LD and Hydra into Symfony2 in the form of a so-called bundle, i.e., a plugin in Symfony2-speak. Thanks to Composer [203] and the modularity of Symfony2, the installation involves a couple of trivial steps. After adding the HydraBundle as a Composer dependency and registering it in Symfony2's kernel, the only remaining step is to import its routes into Symfony2's routing collection. All this requires just a couple of lines of code and is documented in detail at the bundles homepage [204].

In order to demonstrate how the bundle can be used in practice, we implemented a hypermedia-driven Web API featuring an issue tracker as a case study. This not only allows us to show how easily Web APIs can be implemented using such an approach but also to describe the implementation of our prototype in more detail.

As we discussed in section 5.5, JSON-LD and Hydra can be used for the domain-driven design and implementation of RESTful Web APIs. According to that approach, the first step in the development of a Web API is to define the required domain concepts. For our issue tracker, the application domain consists of issues, comments on those issues, and users. Issues have a title, a description, a state (open/closed), a creation date, and a reference to the user who created it. Comments associated to an issue have a description, a reference to the user who created it, and a creation date. Finally, users have a name, an e-mail address, and a password. Using the user type as an example, we will show how classes can be augmented with the annotations necessary for their serialization and the generation of a machine-readable vocabulary.

As shown in the extract of the `User` class definition in Listing 40, the fields to be exposed when an instance is being serialized are annotated with an `@Hydra\Expose()` annotation. The password will never be serialized, as it is marked as write-only. It will, nevertheless, be documented in the automatically generated API documentation and be used when deserializing requests. The class itself has an `@Hydra\Id()` annotation which converts the internal identifier (an integer) to a globally valid identifier in the form of an IRI. This is done by referencing the corresponding route which essentially represents an IRI template—in this case `/users/{id}`. The class also has an `@Hydra\Operations()` annotation which documents the supported operations on this entity. In this example it references routes to replace (update) and delete users.

The `raised_issues` property in Listing 40 returns an array of the issues the user raised. The `@Hydra\Collection()` annotation tells the serializer that it should wrap that array in a Hydra `Collection` that can be accessed via the specified route. The alert reader might wonder why there is no variable mapping as in the class' ID annotation. The reason is that the serializer is smart enough to create those mappings itself if the IRI template variables correspond directly to a property of the class. In this case, there is a direct correspondence to the `id` property. This approach is commonly known as *convention over configuration* and is used to decrease the amount of code/annotations a developer has to write. The same reasoning applies to the automatic code generation of simple CRUD-controllers. All a developer has to do to generate a controller for the just defined `User` class is to invoke the following command in the shell:

```
php app/console hydra:generate:crud --entity=MLDemoBundle:User
  --route-prefix=/users/ --with-write --no-interaction
```

This will create a page controller supporting all CRUD operations and listening to requests on the `/users/` IRI prefix. If a developer omits the

```
namespace ML\DemoBundle\Entity;
use ML\HydraBundle\Mapping as Hydra;

/**
 * User
 *
 * @Hydra\Expose()
 * @Hydra\Id(
 *   route = "user_retrieve",
 *   variables = { "id" : "id" }
 * )
 * @Hydra\Operations({"user_replace", "user_delete"})
 */
class User
{
  /**
   * @var integer An internal unique identifier
   */
  private $id;

  /**
   * @var string The user's full name
   * @Hydra\Expose()
   */
  private $name;

  /**
   * @var string The user's email address
   * @Hydra\Expose()
   */
  private $email;

  /**
   * @var string The user's password
   * @Hydra\Expose(writeonly = true)
   */
  private $password;

  /**
   * The issues raised by this user
   *
   * @var ArrayCollection<ML\DemoBundle\Entity\Issue>
   * @Hydra\Expose()
   * @Hydra\Collection("user_raised_issues_retrieve")
   */
  private $raised_issues;

  // ... getters, setters, and other methods
}
```

Listing 40. An annotated entity class

parameters, a wizard will ask for the required information step-by-step. The code to retrieve the issues raised by a user cannot be generated

```
/**
 * Retrieves the issues raised by a User
 *
 * @Route("/{id}/raised_issues",
 *   name="user_raised_issues_retrieve")
 * @Method("GET")
 * @Hydra\Operation(
 *   status_codes = {
 *   "404" = "If the User entity wasn't found."
 * })
 * @Hydra\Collection()
 * @return ArrayCollection<ML\DemoBundle\Entity\Issue>
 */
public function getRaisedIssuesAction(User $entity)
{
  return $entity->getRaisedIssues();
}
```

Listing 41. An annotated controller function

automatically and has thus to be added manually. This is simple as the code in Listing 41 shows.

The `@Hydra\Operation()` annotation above shows how to document an operation. In this case, the operation would be exposed as `GetRaisedIssuesOperation` and contain the additional information when a response with a status code of `404` is returned and what it means in this context (in this case not that no raised issues exist, but that the user does not exist). In the long term, we envision that a large number of such operations are "standardized" and thus recognized by generic clients—Hydra's built-in CRUD operations are just the beginning. The methods generated by the CRUD controller code generator are automatically mapped to Hydra's built-in operations. This allows the prototype API console we implemented to pre-fill the form generated for a `ReplaceResourceOperation` with the data of the entity.

Implementing the rest of the API is just a matter of implementing the domain concepts and documenting them with the appropriate annotations. The system is then able to automatically generate both a human-readable documentation in the form of an HTML page and a machine-readable vocabulary in JSON-LD for the client. As the response in Listing 42 shows, it also allows the system to automatically serialize the entities returned by page controllers into JSON-LD documents that look almost exactly the same as responses of current JSON-based Web APIs, apart from the link to a context definition and the `@id` and `@type` keywords, which could also be aliased to something else.

```
{
  "@context": "/contexts/User.jsonld",
  "@id": "/users/1",
  "@type": "User",
  "name": "Markus Lanthaler",
  "email": "mail@markus-lanthaler.com",
  "raised_issues": {
    "@id": "/users/1/raised_issues",
    "@type": "hydra:Collection"
  }
}
```

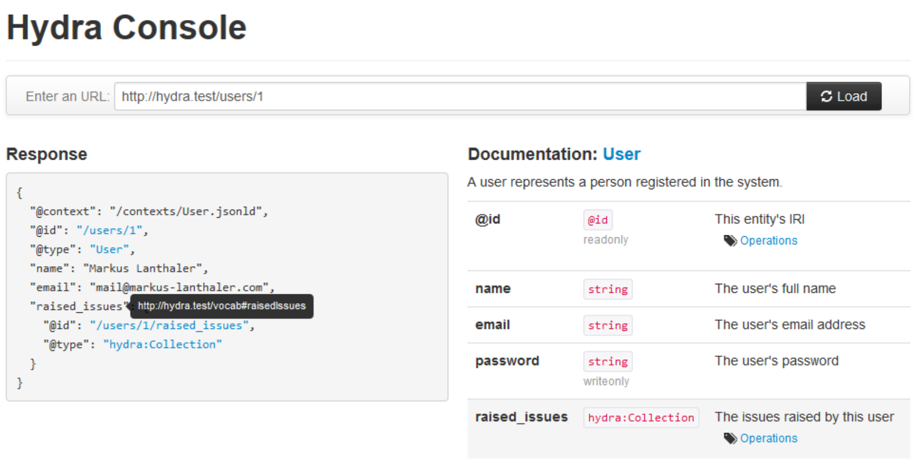
Listing 42. A sample response as rendered by the HydraBundle

6.3 Support for Generic Clients

The advantage of JSON-LD and Hydra manifests itself most apparently in the fact that it is possible to implement fully generic and adaptive clients. To demonstrate this, we implemented an API console or browser which allows the user to navigate the service presented in previous section and to invoke operations on the various resources. Furthermore, the console displays the relevant element documentation which is also used to dynamically create forms to gather the required data for the construction of valid HTTP requests.

The HydraConsole [205], as we named our API browser, is implemented as a single-page JavaScript Web application using a number of well-known libraries such as jQuery [206], Bootstrap [207], Backbone.js [208], and Underscore.js [209]. Even though it would probably have made the implementation slightly simpler we made the deliberate design decision to not use any RDF-specific library such as a triple store or a SPARQL engine. We believe that it is important to demonstrate to Web developers without Semantic Web background that it is possible to implement such a generic client without having to buy into the typical RDF stack. Instead, our implementation shows that such a client can be realized with tools and libraries most Web developers are already familiar with.

Following a similar reasoning we kept the user interface quite simple. Thus, instead of rendering the responses in the form of abstract graphs—


Hydra Console

http://hydra.test/users/1

Response

```
{
  "@context": "/contexts/User.jsonld",
  "@id": "/users/1",
  "@type": "User",
  "name": "Markus Lanthaler",
  "email": "mail@markus-lanthaler.com",
  "raised_issues": {
    "@id": "/users/1/raised_issues",
    "@type": "hydra:Collection"
  }
}
```

http://hydra.test/vocab#raisedIssues

Documentation: User

A user represents a person registered in the system.

@id	@id readonly	This entity's IRI Operations
name	string	The user's full name
email	string	The user's email address
password	string writeonly	The user's password
raised_issues	hydra:Collection	The issues raised by this user Operations

Figure 16. The HydraConsole showing a response and its documentation

as it is often done in Semantic Web tools and demos—we decided to display the retrieved JSON-LD representations more or less as they were received. As shown in Figure 16, the only formatting we apply to responses are whitespace changes, such as the addition of line breaks and indentations, and the underlining and coloring of hyperlinks in their typical blue. To aid the understanding of the rendered responses as JSON-LD we added some unobtrusive interactivity.

When the user moves his mouse over a property in the response pane to the left, a tooltip showing the IRI it is expanded to appears. Additionally, the API console dereferences the property's IRI in the background, looks for its definition in the response, displays the documentation of the class associated with the property in the pane at the right, and finally highlights the property itself in the displayed documentation. The result of this process is shown in Figure 16 in which the user's mouse is over the `raised_issues` property that, as the tooltip shows, expands to the IRI `http://hydra.test/vocab#raisedIssues`.

To realize this in-place expansion with tooltips we had to modify the expansion algorithm of our JSON-LD processor [210] to not only emit the expanded document, but also combine it with the input document. Thus, behind the scenes the sample response shown in Listing 42 and Figure 16 is transformed to the document shown in Listing 43 (context omitted). As the document in Listing 43 illustrates, properties are not removed during expansion but their value is instead transformed to an object consisting of an `__iri` and a `__value` member. The value of the

```
{
  "@context": ... ommitted for clarity ...
  "@id": {
    "__iri": "@id",
    "__value": {
      "__orig_value": "/users/1",
      "__value": { "@id": "http://hydra.test/users/1" }
    }
  },
  "@type": {
    "__iri": "@type",
    "__value": {
      "__orig_value": "User",
      "__value": { "@id": "http://hydra.test/vocab#User" }
    }
  },
  "name": {
    "__iri": "http://hydra.test/vocab#User/name",
    "__value": {
      "__orig_value": "Markus Lanthaler",
      "__value": { "@value": "Markus Lanthaler" }
    }
  },
  "email": {
    "__iri": "http://hydra.test/vocab#User/email",
    "__value": {
      "__orig_value": "mail@markus-lanthaler.com",
      "__value": { "@value": "mail@markus-lanthaler.com" }
    }
  },
  "raised_issues": {
    "__iri": "http://hydra.test/vocab#User/raisedIssues",
    "__value": {
      "@id": {
        "__orig_value": "/users/1/raised_issues",
        "__value": {
          "@id":"http://hydra.test/users/1/raised_issues"
        }
      },
      "@type": {
        "__orig_value": "hydra:Collection",
        "__value": {
          "@id": "http://www.w3.org/ns/hydra/core#Collection"
        }
      }
    }
  }
}
```

Listing 43. The sample response from Listing 42 as expanded for the rendering in the HydraConsole (context omitted for clarity)

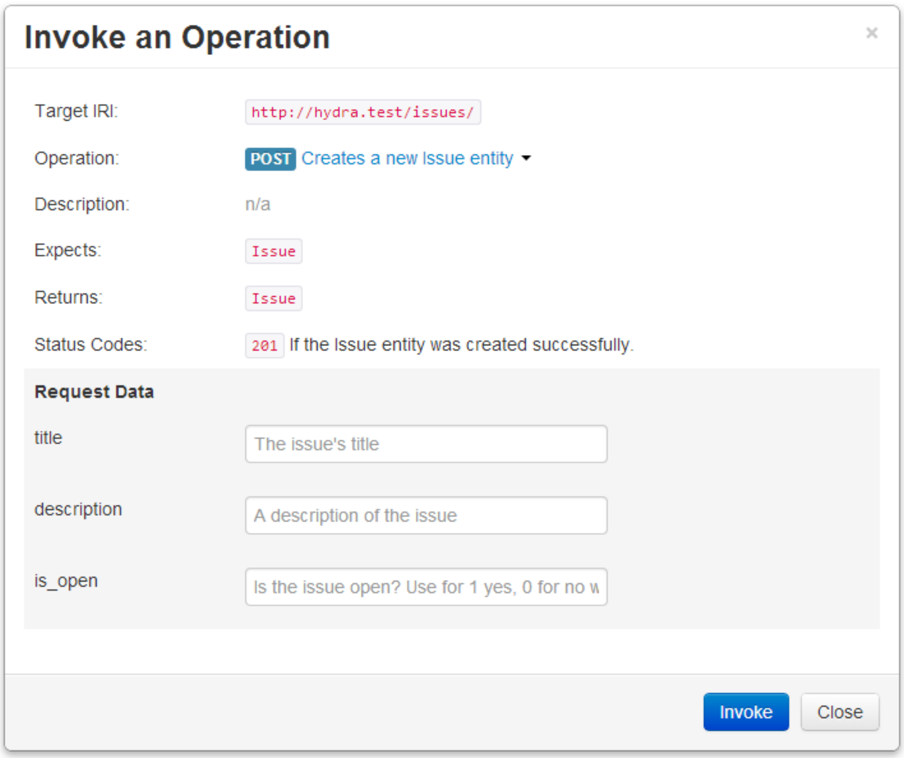
`__iri` member represents the expanded property. Therefore, the value of the `__iri` member of the `name` property is set to `http://hydra.test/`↩

vocab#User/name. As the name suggests, the __value member keeps the value of the property, which, again is split into a __value member holding the expanded value and a __orig_value member keeping the property's original, unexpanded value. This representation of both the expanded and the original, unexpanded document at the same time is what enables the HydraConsole to render the tooltips mentioned previously and to distinguish between ordinary strings and strings representing IRIs.

Given that our JSON-LD processor is implemented in PHP and not in JavaScript, it runs as a remote service that is invoked by the HydraConsole. Similarly, we implemented a simple proxy to work around the same-origin policy of browsers which, for security reasons, typically block network requests to all servers except the one that served the original web page. This was especially important at the beginning, before Hydra's namespace was moved from purl.org to w3.org, as purl.org still does not set the necessary Cross-Origin Resource Sharing (CORS) headers [211] which would allow modern browsers to make cross-origin requests.

When a user decides to load a new resource into the HydraConsole, a number of steps happen in the background. First of all, the resource is loaded via the proxy described above. If it yields a JSON-LD representation, the representation is expanded using the modified expansion algorithm in order to be rendered in the HydraConsole. Then, if the response contained an HTTP Link header to a Hydra API documentation, that documentation is loaded and framed via the proxy. If available, the documentation of the type of the top-level resource contained in the response is loaded and rendered. The result of retrieving the http://hydra.test/users/1 resource is illustrated in Figure 16.

Navigating the Web API by following hyperlinks or sending HTTP requests other than GET is as easy as clicking on a link and selecting the desired operation. The HydraConsole presents a dialog in which the user can select the desired operation. The HydraConsole takes into consideration operations embedded directly in the representation as well as operations bound to the property (i.e., the link relation) whose target the resource is and operations bound to the types the resource is an instance of. It is important to note that the client is stateless in the sense that it forgets information from previous requests. This implies that clicking on a hyperlink in different contexts may result in different operations being


Invoke an Operation

Target IRI:	`http://hydra.test/issues/`
Operation:	POST Creates a new Issue entity
Description:	n/a
Expects:	`Issue`
Returns:	`Issue`
Status Codes:	`201` If the Issue entity was created successfully.

Request Data

title	The issue's title
description	A description of the issue
is_open	Is the issue open? Use for 1 yes, 0 for no w

Figure 17. A form rendered by the HydraConsole to invoke an operation

shown. In any case, if the user selects an operation whose expected data is documented, a form to gather the required data for the creation of a valid request, as the one shown in Figure 17, is constructed on-the-fly. In order to make the navigation consisting of just `GET` requests more efficient, clicking on a link with the shift key pressed directly invokes an HTTP `GET` request on the target resource.

6.4 Adoption

Even the best technology is useless if it is not accepted by users. While this important aspect is usually neglected in research projects, it was a major concern in our work. Our focus on simplicity is fueled by studies that have shown that the perceived usefulness and the perceived ease of use are key drivers of technology acceptance and adoption [212]–[214]. The *perceived usefulness* is defined by Davis [214] as "the degree to which a person believes that using a particular system would enhance his or her job performance" whereas the *perceived ease of use* is defined as "the degree to which a person believes that using a particular system would be free of effort". Therefore, we spent a lot of time to find the right balance between feature-richness and simplicity in order to maximize the chance of adoption. While these two determinants are important, they are rarely enough by themselves. In general, further motivation is needed to overcome user resistance to change [212]. Hence, it is vital to understand the typical technology adoption lifecycle. As Tim Berners-Lee, the inventor of the Web, said, "the web is more a social creation than a technical one." [94]

Adopter groups can be categorized by their degree of resistance to a new idea or technology. In his seminal book "Diffusion of Innovations" [215], Rogers divides adopters into five categories based on their *innovativeness*: innovators, early adopters, early majority, later majority, and laggards. Innovators play an important gatekeeping role in the adoption process as they import a technology into their community (they are not necessarily respected by other members of their community). *Early adopters* on the other hand are a more integrated part of a community than their innovators. Thus, this adopter category has the highest degree of opinion leadership in most communities. Given that they are not too far ahead of the average individual in innovativeness, they typically serve as a role model in their community. Early adopters decrease the uncertainty about a new technology by adopting it and thereby help to trigger the critical mass. The *early majority* follows "with deliberate willingness in adopting innovations but seldom lead" [215]. By adopting new technologies just before the average member of a community does, they form an important link to adopters with longer innovation-decision periods. The *late majority* is driven by the pressure of

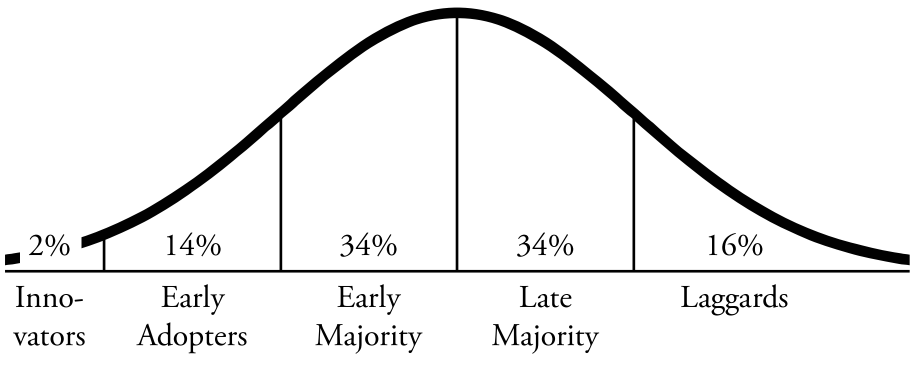


Figure 18. Adopter categorization on the basis of innovativeness [215]

peers and the early majority as well as economic necessity. Most of the uncertainty about a new technology must be eliminated before the late majority adopts it. Together, the early and late majority make up roughly two thirds of the members of a community as illustrated in Figure 18. The only remaining group is the *laggards* or *late adopters*. According to Rogers, they tend to be suspicious of innovations and thus extremely cautious in adopting them.

Following this model we reached out to innovators in different communities very early in the development of both JSON-LD and Hydra. While JSON-LD managed to attract notable early adopters such as Google and the BBC, Hydra, being a very young technology, is a step behind in the adoption lifecycle and is still concentrating on innovators. The W3C Community Group into which Hydra's further development was moved grew within the first six months of its existence to over fifty members [216]. The participants' backgrounds range from academia over startups to large companies.

In the following sections, we will present a number of notable adopters broadly categorized into academia, industry, and standardization efforts. This is by no means intended to be an exhaustive list. The motivation is to present a number of different use cases for which these technologies have been adopted and to create a historical reference of the early days of JSON-LD and Hydra.

6.4.1 Academia

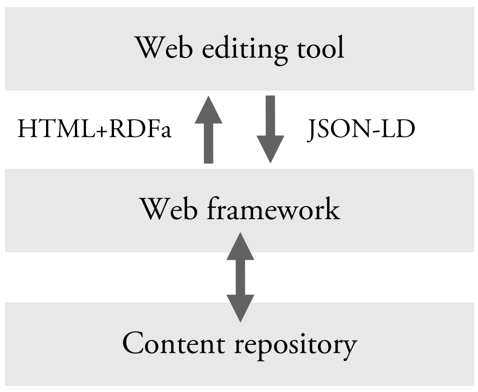


Figure 19. Decoupled CMS architecture

Among the first adopters of JSON-LD was IKS (Interactive Knowledge Stack) [217], a multi-million euro research project funded mostly by the European Union [218]. IKS was developed by a core consortium of seven research and six industrial partners. The main outcome of the project is a reference architecture for semantic content management systems [219] and a set of open source software components that both represent the reference implementation and extend traditional content management systems (CMS) with Semantic Web technologies.

The open source components can be divided into two main projects: the VIE project [220], focusing on presentation and user interaction, and the Apache Stanbol project [221], providing services such as named entity extraction and reasoning to enhance unstructured content with metadata about related entities and links. The two projects are loosely coupled via a RESTful service interface resulting in a so called “decoupled content management system” [222]. In contrast, most traditional CMS are implemented as a monolithic block. As illustrated in Figure 19, the IKS project breaks this block into a classic three-tier architecture. The communication between the logic tier and the data tier can be realized by standardized interfaces such as the Content Repository API for Java (JCR) [223]. Similarly, the communication between the logic tier and the presentation tier can be realized with standardized technologies. Given that the IKS project concentrates mainly on web content management systems, it decided to send data from the logic to the presentation tier as HTML annotated with RDFa. These semantic annotations make it possible to extract the content model of the CMS into JavaScript models on the client, i.e., the browser. The presentation tier then implements the user interface to manage the content, which includes features such as rich text editing, semantic annotations, and image handling. The changes are then sent back to the logic tier using JSON-LD. The service interface itself is hardcoded into the client and predates Hydra. It would be interesting to describe it using Hydra in future versions.

An interesting feature of VIE is the ability to load additional information about an entity from DBpedia; another project adopting JSON-LD quite early. DBpedia [224] is one of the most well-known Linked Data projects. Several hundred data sets on the Web reference DBpedia entities making it one of the central interlinking hubs in the Linked Open Data cloud [80]. Its main idea is to extract structured data from Wikipedia articles and turn it into a rich, multi-lingual knowledge base by mapping the extracted concepts to an ontology. The result is a knowledge base consisting of almost 1.5 billion facts about more than 13 million things. Using HTTP content negotiation, all that data can be retrieved from DBpedia as JSON-LD.

While these efforts either concentrate on read-only access to data or rely on out-of-band documentation for their APIs, the Educational System Group [225] of the Galileo University Guatemala goes a step further. In their Cloud Educational Interoperability Service project they are building an e-learning platform powered by popular online tools. Instead of creating, e.g., word processors or mind mapping editors themselves, they integrate popular, free tools such as Google Docs [226], MindMeister [227], or Cacoo [228] by transforming their responses to JSON-LD and describing them with Hydra. These machine-readable descriptions allow them to semi-automatically create widgets that can be used by non-technical experts to build personal learning environments.

The project is a highly interesting test bed for both Hydra and JSON-LD as it tries to convert real-world APIs into APIs powered by JSON-LD and Hydra. Instead of programming directly against the underlying API, all requests go to through a transformation layer harmonizing the APIs. The advantage is that integration becomes much more efficient and programming can happen on a higher level of abstraction. The system is built on a layered architecture that helps to concentrate the transformations on a single layer instead of having to spread them throughout the whole code base. As soon as the third-party APIs adopt JSON-LD and Hydra themselves, it becomes possible to completely eliminate that intermediary layer. Since the project is still under heavy development unfortunately no publicly accessible publications are available yet at the time of this writing.

6.4.2 Industry

Large media organizations and libraries typically maintain sophisticated metadata catalogs to manage their information. Since this is one of the core features of the Semantic Web technologies, media organizations started to embrace them relatively early. The BBC is one of the pioneers in this field by launching "the first large scale, mass media site using concept extraction, RDF, and a triple store to deliver content" [229] for the 2010 FIFA World Cup. The use of Semantic Web technologies helped, among others, to improve navigation, content re-use and re-purposing, and search engine rankings. Furthermore, the system enabled the automated publication of web pages that require minimal journalist

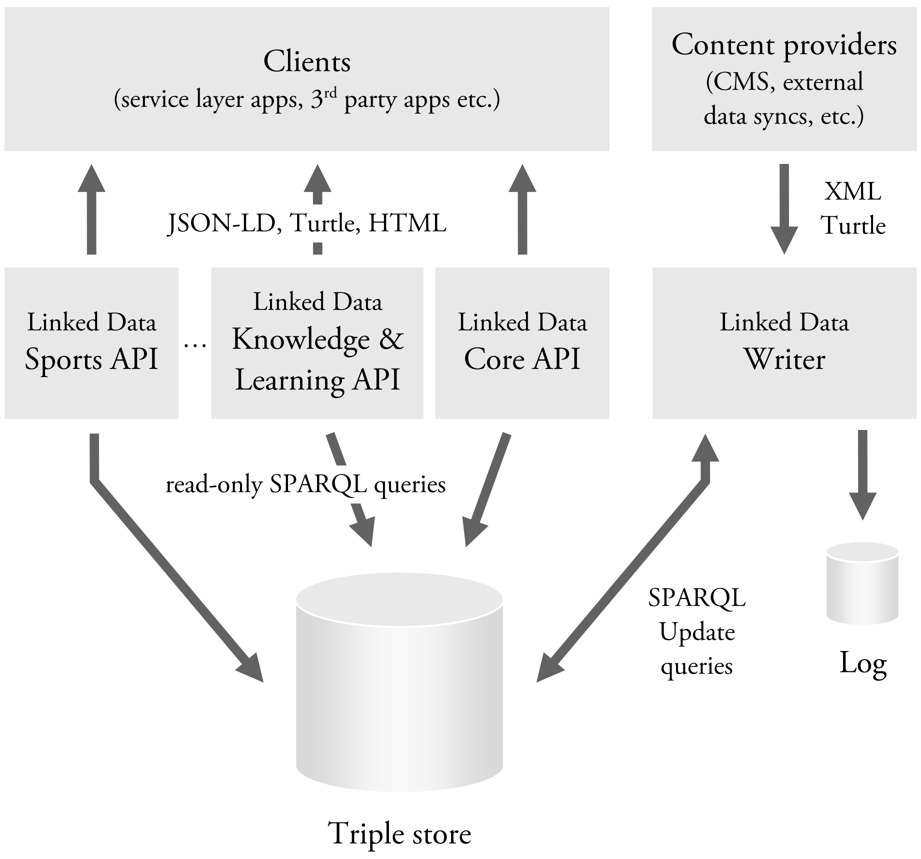


Figure 20. Architecture of the BBC Linked Data Platform

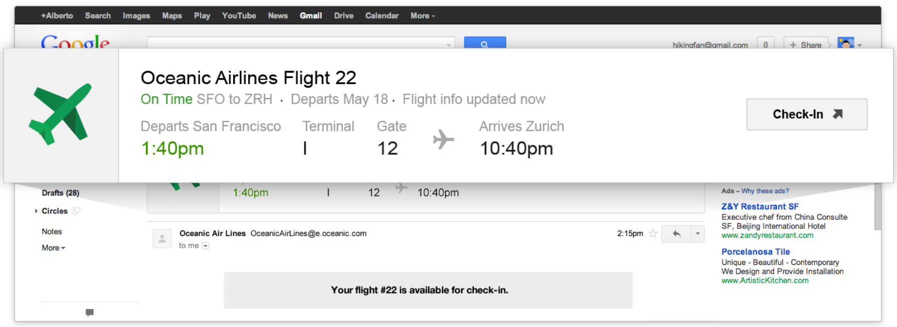
management, as they automatically aggregate and render links to relevant stories and assets.

A challenge the BBC was dealing with was the usage of RDF data in the frontend. As discussed in section 3.4, web developers find RDF and its serialization formats such as RDF/XML confusing and hard to code with. Thus, over the years the BBC refined its architecture and created a "linked data platform" (not to be confused with the Linked Data Platform [155] being standardized at the W3C) as illustrated in Figure 20. In its latest versions, Turtle and RDF/XML have been replaced with JSON-LD [230]. As David Rogers, senior technical architect at BBC Future Media, News & Knowledge, told us in personal communication, the linked data platform's APIs are planned to form the foundation of a "BBC open API" and JSON-LD will be the default format serialization format.

The platform used by globo.com, the web portal of Organizações Globo, the largest media group in Latin America, went through a similar transformation. As Ícaro Medeiros reported at the 2nd International Workshop on Web of Linked Entities (WoLE) [231], the Semantic Web team at globo.com re-architected their system to increase the data quality and to make access to data simpler, more secure, and performant by introducing a RESTful Web API built on JSON-LD. The API uses hypermedia but does not offer any hypermedia controls apart from simple hyperlinks. All operations are being described out-of-band in natural language. Since Hydra is able to address this issue, globo.com was among the first participants of the W3C Hydra Community Group [216] and is currently evaluating its usage.

Coincidentally, exactly one day after we presented Hydra for the first time to a larger audience at the 22nd International World Wide Web Conference, Google announced arguably the most high-profile adoption of JSON-LD at the Google I/O 2013 [232]. They released a new feature for Gmail, Google Search, and Google Now that leverages JSON-LD to embed structured data into e-mails. This data allows them to understand what an e-mail is about and thus to process it more intelligently. When a user, e.g., opens a flight confirmation e-mail in Gmail, all the important information about the flight is extracted from the e-mail and displayed prominently above the e-mail itself as shown in Figure 21. The same information is used to display the so-called Google Now cards on Android and iOS and is also integrated into Google's search engine so that users can easily find it. Figure 22 illustrates how the result looks like when users search for their hotel reservations.

The most interesting aspect of this new feature is that it is not limited to just displaying data. Google went a step further and also allows users to take action on e-mails without needing to open them first [233].



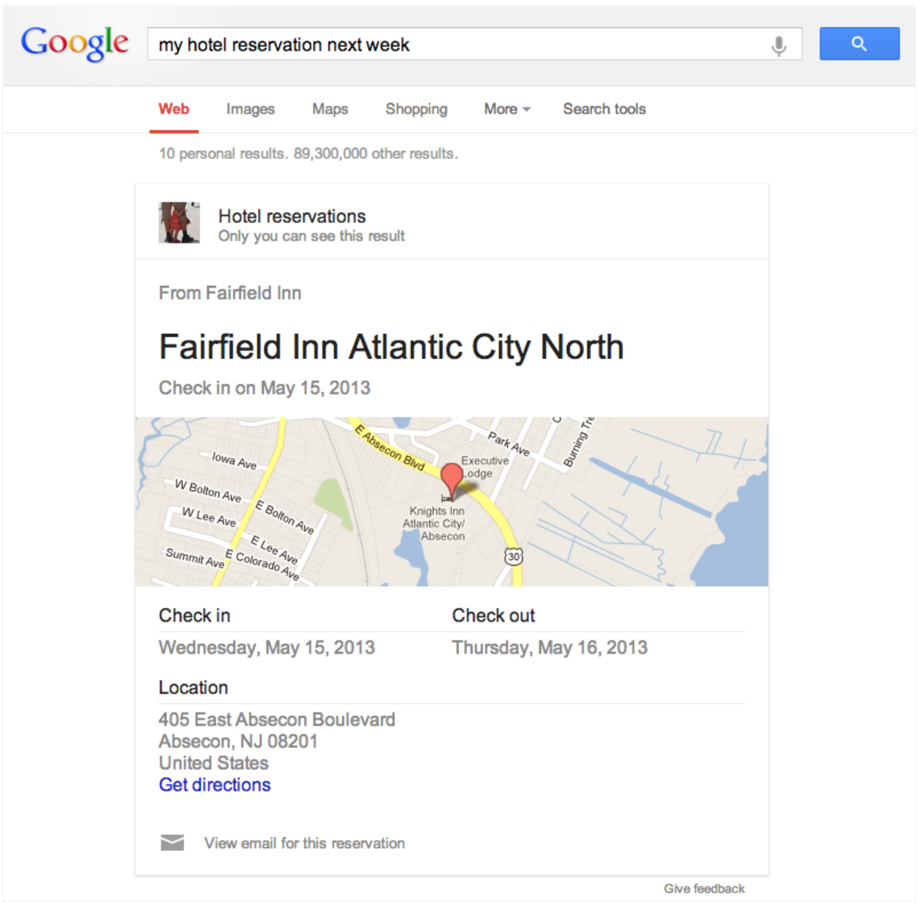
Oceanic Airlines Flight 22
On Time SFO to ZRH · Departs May 18 · Flight info updated now
Departs San Francisco 1:40pm | Terminal I | Gate 12 | Arrives Zurich 10:40pm

Oceanic Air Lines OceanicAirLines@e.oceanic.com 2:15pm
to me

Your flight #22 is available for check-in.

Ads – Why these ads?
Z&Y Restaurant SF
Executive chef from China Consulte SF, Beijing International Hotel
www.zandyrestaurant.com
Porcelanosa Tile
Unique - Beautiful - Contemporary We Design and Provide Installation
www.ArtisticKitchen.com

Figure 21. Gmail displays information embedded as JSON-LD in e-mails


my hotel reservation next week

Web Images Maps Shopping More Search tools

10 personal results. 89,300,000 other results.

Hotel reservations
Only you can see this result

From Fairfield Inn

Fairfield Inn Atlantic City North

Check in on May 15, 2013

Check in
Wednesday, May 15, 2013

Check out
Thursday, May 16, 2013

Location
405 East Absecon Boulevard
Absecon, NJ 08201
United States
Get directions

View email for this reservation

Give feedback

Figure 22. Google queries may return information embedded in e-mails

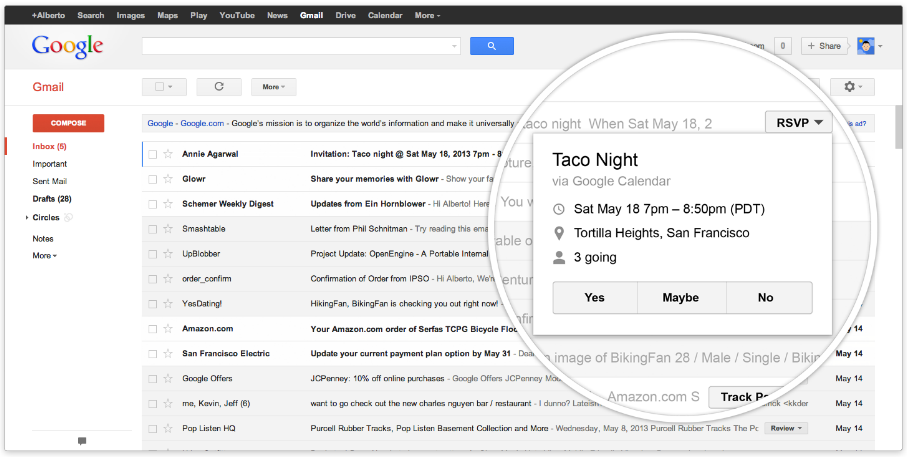
Responding to an invitation thus becomes as simple as pressing two buttons as illustrated in Figure 23. The code to render an RSVP button in Gmail is shown in Listing 44. It describes an event and declares three actions of type `RsvpAction` with different values for the `attendance` property.

While Google uses Schema.org as the vocabulary to describe actions in e-mails, the concepts have not yet been added to Schema.org. In fact, we are in contact with Google and other Schema.org partners in order to improve the vocabulary and to align it or replace it with Hydra. We also published a first draft [189] explaining how a subset of Hydra could be integrated into Schema.org. By making descriptions of actions with Schema.org less RPC-oriented, the same vocabulary could be reused to describe RESTful Web APIs.

The usage of JSON-LD by Google for Actions in Gmail and the subsequent enthusiastic announcement by the Schema.org partners [235] that JSON-LD has been added to the list of recommended serialization formats for Schema.org is an important proof for the trust in the technology.

```
<html>
  <body>
    <script type="application/ld+json">
    {
      "@context": "http://schema.org//",
      "@type": "Event",
      "name": "Taco Night",
      "startDate": "2013-05-18T19:00-07:00",
      "endDate": "2013-05-18T20:50-07:00",
      "location": {
        "@type": "Place",
        "address": {
          "@type": "PostalAddress",
          "name": "Taco Joe",
          "streetAddress": "Tortilla Heights",
          "addressLocality": "San Francisco",
          "addressRegion": "CA",
          "postalCode": "94107",
          "addressCountry": "USA"
        }
      },
      "action": [ {
          "@type": "RsvpAction",
          "handler": {
            "@type": "HttpActionHandler",
            "url": "http://example.com/rsvp?eventId=123&value=yes",
            "method": "http://schema.org/HttpRequestMethod/GET"
          },
          "attendance": "http://schema.org/RsvpAttendance/Yes"
        }, {
          "@type": "RsvpAction",
          "handler": {
            "@type": "HttpActionHandler",
            "url": "http://example.com/rsvp?eventId=123&value=no",
            "method": "http://schema.org/HttpRequestMethod/GET"
          },
          "attendance": "http://schema.org/RsvpAttendance/No"
        }, {
          "@type": "RsvpAction",
          "handler": {
            "@type": "HttpActionHandler",
            "url": "http://example.com/rsvp?eventId=123&value=maybe",
            "method": "http://schema.org/HttpRequestMethod/GET"
          },
          "attendance": "http://schema.org/RsvpAttendance/Maybe"
        }
      ]
    }
    </script>
    <p>Please let me know if you join our Taco Night on Saturday.</p>
  </body>
</html>
```

Listing 44. A marked-up e-mail declaring an event with an RSVP action (adapted from [234])



Google - Google.com - Google's mission is to organize the world's information and make it universally

Annie Agarwal — Invitation: Taco night @ Sat May 18, 2013 7pm - 8
Glowr — Share your memories with Glowr - Show your fa
Schemer Weekly Digest — Updates from Ein Hornblower - Hi Alberto! Here
Smashtable — Letter from Phil Schnitman - Try reading this ema
UpBlobber — Project Update: OpenEngine - A Portable Internal
order_confirm — Confirmation of Order from IPSO - Hi Alberto, We're
YesDating! — HikingFan, BikingFan is checking you out right now! -
Amazon.com — Your Amazon.com order of Serfas TCPG Bicycle Floo — May 14
San Francisco Electric — Update your current payment plan option by May 31 - Dear — May 14
Google Offers — JCPenney: 10% off online purchases - Google Offers JCPenney Mo — May 14
me, Kevin, Jeff (6) — want to go check out the new charles nguyen bar / restaurant - I dunno? Lateism — May 14
Pop Listen HQ — Purcell Rubber Tracks, Pop Listen Basement Collection and More - Wednesday, May 8, 2013 Purcell Rubber Tracks The Pc — May 14

Taco night When Sat May 18, 2

Taco Night
via Google Calendar
Sat May 18 7pm – 8:50pm (PDT)
Tortilla Heights, San Francisco
3 going

image of BikingFan 28 / Male / Single / Bikir
Amazon.com S

Figure 23. Actions in Gmail can be invoked directly from the inbox

As discussed in the introduction, these early adopters play a crucial role in triggering the critical mass by decreasing the uncertainty about a new technology such as JSON-LD. The official ratification of JSON-LD as a W3C standard is another important milestone to convince more risk-averse adopters with longer innovation-decision processes.

6.4.3 Standardization

Due to the need for stable base technologies, standardization efforts are typically among the last to adopt new technologies. Nevertheless, JSON-LD is already being used as the serialization format in a number of specifications and recommended by others.

The IMS Global Learning Consortium, e.g., announced in a press release in 2012 [236] to use JSON-LD as the main serialization format in the second version of their Learning Tools Interoperability standard. Similarly, the Image API 1.1 specification [237] of the International Image Interoperability Framework (IIIF) working group requires that conformant implementations use JSON-LD. Other specifications, such as Open Annotation [238] or the Open Digital Rights Language [239] do not require the usage of JSON-LD but recommend it.

Another interesting standardization effort to keep an eye on are JSON Activity Streams 2.0 [240] and the corresponding draft specifying action handlers [241]. They are relevant for both JSON-LD and Hydra. JSON

Activity Streams 2.0 is not directly based on JSON-LD but has been "designed to be closely compatible with JSON-LD" [240]. Furthermore, the IETF Internet Draft contains an appendix explaining how a JSON Activity Stream can be processed as JSON-LD. Action handlers could be classified as a mixture of the approach Google used to describe actions in e-mails and Hydra. Consequently, they are highly interesting for the further development of Hydra.

6.5 Discussion

The combination of the REST architectural style with the Linked Data principles builds a foundation to bring many of the key success factors of the human Web to the Web of machines. Instead of building Web APIs with highly specialized interfaces, all the modeling happens on a semantic layer completely independent of the underlying serialization format. By using a format such as JSON-LD to serialize the data, a gradual introduction of such a (at first sight) disruptive approach becomes possible. Apart from a few additional properties, the responses from a Web API using JSON-LD and Hydra look almost exactly the same as the ones of current JSON-based APIs. As shown in this chapter, this greatly simplifies the integration into current Web frameworks and provides a smooth upgrade path for developers that have to build on existing infrastructure investments.

The standardized data format provided by JSON-LD along with the concepts defined by Hydra enables the creation of completely generic clients. Together they are thus capable of addressing all issues that have been discussed in Chapter 3. JSON-LD has become an official World Wide Web Consortium standard that has been well accepted and adopted. Hydra, being the much younger technology, has been moved into a W3C Community Group for further development and is starting to gain traction.

Summarized, the combination of JSON-LD and Hydra is, to our best knowledge, the first practical solution to successfully bridge the gap between REST and Linked Data. By combining the REST architectural style with the Linked Data principles and the rich semantic framework established by RDF and related technologies, JSON-LD and Hydra are able to solve problems that API publishers and consumers are increasingly struggling with. This, hopefully, not only helps to solve the issues of current Web APIs, but also to advance the World Wide Web as a whole, as Martin Hepp, creator of the widely used GoodRelations e-commerce vocabulary [242], suggested in a recent tweet [243].

Chapter 7

Conclusions and Future Work

For many companies RESTful Web APIs have become an integral part of their strategy and products. Just as it became clear in the early 2000s that a company has to have a website in order to stay competitive it is nowadays almost mandatory for businesses to provide APIs. In fact, for a lot of companies the API has become the main product instead of being just an add-on for other products. Yet, by looking at the APIs such companies expose, history seems to repeat itself. Similar to how businesses at the beginning of the last decade struggled to embrace the Web as a different medium and wondered why their websites, which closely resembled their print products, failed to engage users, nowadays API publishers try to reuse their existing implementations by exposing them directly and wonder why developers have troubles accessing their services. However, instead of addressing the problem at its core, most API publishers confine themselves to cure the symptoms by also implementing the clients themselves; mostly in the form of software development kits (SDKs) or libraries. Actually, a whole industry emerged to cure the symptoms. There exist API integrators harmonizing various APIs in a specific vertical by creating generalized wrappers, API orchestration platforms allowing even non-technical users to connect different APIs to create simple applications, API portal providers formalizing API documentation, access token management, etc. by creating developers portals, API testing and monitoring solutions helping companies to fulfill their service level agreements (SLAs), API security gateways providing single sign-on solutions or enforcing access control, and API marketplaces helping publishers to monetize their services. Since such offerings are typically based on proprietary technology, their usage normally results in vendor lock-in. Switching to another vendor is not only complex, but also labor intensive and thus costly.

As analyzed in detail in Chapter 3, the root causes that led to this situation are the usage of proprietary data formats and data models and the reliance on static, manually written, natural language contracts documented out of band at design time instead of negotiating and communicating them dynamically at run time. From the perspective of the Representational State Transfer architectural style this corresponds to a violation of the self-descriptive messages constraint and a negligence of hypermedia as the engine of application state. This means that crucial information is not available in a machine-processable form, which makes it almost impossible to implement powerful, generic tooling. While Semantic Web technologies would at least be able to offer a solution to make messages self-descriptive, they suffer from a number of problems themselves—including a lack of hypermedia support. Most problematic, however, is their disruptiveness and (perceived) complexity.

For a long time the Semantic Web community derailed into the artificial intelligence domain instead of concentrating on more practical data-oriented applications. This resulted in technologies that are, admittedly, powerful, but also very alien and difficult to understand for typical developers. Gradually adopting Semantic Web technologies is very challenging as the underlying data model with its open world assumption is fundamentally different from what developers are used to. RDF/XML, which was standardized at the peak of XML's popularity, certainly tried to appeal to developers by being based on XML but it is widely disliked even by XML enthusiasts. It is neither optimized for humans nor for machines and, most critically, standard XML tools are almost useless when working with RDF/XML documents. RDF/XML can thus be considered as one of the main barriers to the adoption of Semantic Web technologies. Another important factor is that the "Web" in "Semantic Web" got little attention. Instead of building a hypermedia-driven Web of Data, IRIs in RDF were often just non-dereferenceable identifiers. In an attempt to refocus on the importance of the main principles of the Web, Tim Berners-Lee formulated the Linked Data principles in 2006. Simply speaking, he advocated the usage of dereferenceable URLs and the interlinking of data. This was an important turning point in the history of the Semantic Web and resulted in huge amounts of data being published. Unfortunately, however, this data is typically not exposed in the form of Web APIs but as static read-only dumps or centralistic SPARQL endpoints. While there have been many efforts to change that, no practical solution that addresses all the issues identified in Chapter 3 exists yet. The state of the art presented in Chapter 4 either addresses just some of the problems or is not practical enough—as confirmed by the lack of adoption. Thus, rather than focusing on a particular problem we looked at the bigger picture in this thesis.

In an iterative process we set out to bridge the gap between REST and Linked Data in order to build a practical solution for these issues. Based on experiences gained by implementing and analyzing various Web APIs, and the lessons learned from designing and experimenting with SAPS and SEREDASj, we were finally able to come up with JSON-LD and Hydra. Together, this loosely coupled combination of data format and vocabulary is able to offer a holistic solution for the identified issues. The clear separation of concerns helped to keep the complexity as low as possible while still providing all necessary functionality. For example, users that do not need the features Hydra provides because they just want to publish read-only data can choose to adopt only JSON-LD. Similarly, users which prefer another serialization format such as RDFa or Turtle may decide to adopt Hydra but not JSON-LD. This was very beneficial in terms of adoption as has been shown in Chapter 6. JSON-LD is already used by hundreds of millions of people across the globe; most of them use it without knowing it. Furthermore, it has become an official World Wide Web Consortium standard. Such quick adoption and standardization would likely have been very difficult to achieve if the two technologies were merged into one. Hydra, being the younger technology is not as widely adopted yet but is quickly gaining traction. The W3C Community Group into which its further development has been moved is continuously growing and there is interest to integrate parts of Hydra directly into Schema.org.

As demonstrated in section 6.2, current Web frameworks can be easily extended to support JSON-LD and Hydra. This is a major selling point compared to most related approaches, which are interesting but rarely practical in real-world scenarios. The design of JSON-LD ensures a smooth upgrade path from JSON tooling that couples on the syntactic structure of representations to more sophisticated tools such as the HydraConsole presented in section 6.3 that work on a higher level of abstraction. Together these two prototypes nicely illustrate what is achievable with JSON-LD and Hydra. The client, represented by the HydraConsole is completely generic. There exist no static contracts that are hardcoded into the client. Instead, all necessary information is exchanged dynamically at runtime. This allows the independent evolution of both the server and the client, and is beneficial in terms of adaptivity and reusability. By reusing existing vocabularies such as Schema.org, interoperability can be increased without having to compromise on extensibility. In many cases, developers do not have to define custom concepts at all but will be able to entirely base their services on existing vocabularies. This prevents the leakage of implementation details which in turn helps to reduce the coupling between clients and servers.

Bridging the gap between REST and Linked Data is an ambitious endeavor. The technologies and prototypes developed as part of this thesis are an important first step but there are still a number of limitations that have to be overcome to fully unfold their potential. Broadly speaking, these issues can be classified into concrete limitations of JSON-LD or Hydra, or general problems such as a lack of tooling and accessible documentation which lead to limited understanding of the underlying principles and ideas. Some of these issues are discussed below and pave the way for future work.

7.1 JSON-LD

The standardization of JSON-LD took many years and was the result of an extremely transparent, open, and consensus-driven process. Most of the development happened in a W3C Community Group which individuals as well as companies could easily join with a couple of mouse clicks. All e-mails were publicly archived, the teleconferences were scribed and recorded, the source code repository and issue tracker are open (everyone can file new issues and file so-called pull requests to contribute code). This ensured that all legitimate views and objections have been considered, which undoubtedly increased the quality of the final specification. Taking into consideration different, and sometimes conflicting, opinions is not always straightforward and frequently requires compromises to be made. Reaching consensus takes time and at some point it becomes necessary to stop the work on new features and concentrate on stabilizing and improving existing ones. This was no different in the development of JSON-LD and resulted in a number of features postponed to future versions of JSON-LD.

The indexing feature of JSON-LD is a great example of this. It is very flexible but nevertheless its realization was a compromise. When working with JSON, developers typically structure their documents in a specific way to simplify data access. In most current programming languages, parsed JSON data can be directly accessed without having to use special programming libraries. The JSON data is directly converted to an in-memory representation. JSON-LD's `@index` keyword allows data to be indexed by arbitrary strings. These strings, however, stand in no relationship with the data; they are just structural annotations (which is why the `@index` keyword was called `@annotation` for some time). In other words, indexes can be discarded without losing any information (in fact they are when JSON-LD data is being converted to RDF). In a lot of cases, however, developers would like to index their data based on the entities' IRIs or on specific properties.

Listing 45 illustrates how such a feature might look like in the future. Additional to setting the `@container` of the `knows` property to `@index`, the "index key" is set to `name`, which in this example would correspond to `http://schema.org/name`. Thus, instead of having to duplicate the data by using the `@index` feature as currently specified, it would become possible to index the data directly. This would allow more existing JSON documents to be interpreted as JSON-LD by just adding a context. It would also reduce the file size of certain JSON-LD documents as no data would have to be duplicated. Following the same motivation, various features have been proposed to simplify the conversion of certain values to IRIs. A lot of current Web APIs expose JSON which does not include the IRIs of entities directly, but just some sort of identifiers such as

```
{
  "@context": {
    "@vocab": "http://schema.org/",
    "homepage": { "@type": "@id" },
    "knows": { "@container": "@index", "@index": "name" }
  },
  "@id": "http://example.com/people/markus",
  "name": "Markus Lanthaler",
  "homepage": "http://www.markus-lanthaler.com/",
  "knows": {
    "Jane Doe": {
      "@id": "/people/jane",
      "homepage": "http://doe.example.com/jane/"
    },
    "John Doe": {
      "@id": "/people/john",
      "homepage": "http://doe.example.com/john/"
    }
  }
}
```

Listing 45. Using arbitrary properties to index data in JSON-LD

primary keys taken directly from the database. In order to convert those strings or numbers to IRIs, a client needs to have knowledge of an IRI template. Instead of having to rely on out-of-band documentation, that information could be directly embedded into a JSON-LD context. There are various proposals ranging from directly including support for IRI templates to allowing the creation of nested contexts, similar to how SEREDASj works. Describing all these proposals is beyond the scope of this section and thus we would like to refer the interested reader to our issue tracker [244] or the mailing list archives [245].

Making JSON-LD more flexible and expressive is important, but just as important is the tooling around it. The standardized JSON-LD API [168], which has been implemented in all major programming languages, lays the foundation for more sophisticated tooling to be built in the future. We have already started working on that with Framing [187]. The algorithm, however, is not fully up to date with the latest JSON-LD specification as work on it was stopped when the JSON-LD syntax and the rest of the API was brought into the RDF Working Group for standardization at the World Wide Web Consortium. Therefore, the framing algorithm does not support reverse properties or named graphs yet. It will be interesting to see how JSON-LD is used in practice and how users can be supported with tooling. Some of these observations have already led to discussions to change some aspects of the framing algorithm in order to make it more useful and practical. There also exist prototypes of tools that automatically generate JSON-LD contexts out of vocabulary definitions or attempt to convert various data formats (not just RDF-based ones) to JSON-LD.

7.2 Hydra

In contrast to JSON-LD, Hydra is not fully stable yet. The different backgrounds of the various members of the Hydra W3C Community Group and the discussions regarding the inclusion of parts of Hydra into Schema.org resulted in interesting debates, new feature proposals, and also revealed a number of weaknesses.

A frequently asked question is how Hydra can be used with media types other than JSON-LD. It is, e.g., not trivial to describe that an operation expects or returns an image file. There are various proposals on the table but no decision has yet been made of how to address this issue. One option would be to create a dedicated class—something like `Blob`—that could then be further refined by media type ranges.

The most often misunderstood and criticized aspects of Hydra, however, are the three predefined operation types. It was never the intention to include specific operation types directly in the Hydra core vocabulary but we deemed it necessary to include some very generic operations to illustrate how the whole concept can be used. Apparently, the inclusion of `CreateResourceOperation`, `ReplaceResourceOperation`, and `DeleteResourceOperation` was counterproductive. A lot of people are confused by these concepts by thinking that they are the only available operation types. Especially people without knowledge of RDF and other Semantic Web technologies do not realize that it is trivial to add additional operation types. Furthermore, the semantics of these operations are very weak. The typing of an operation as a `DeleteResourceOperation`, e.g., does not add much value if the HTTP method of that operation is set to `DELETE`. Given that the concept of "actions" has been added to Schema.org since the initial release of Hydra, it is likely that we will remove these three predefined operation types from the Hydra core vocabulary.

The assumption that collections are a much-needed feature in most Web APIs seems to vindicate. In many discussions their usefulness was reiterated and some members of the Hydra W3C Community Group proposed extensions to make it even more useful. An interesting proposal is to add a `memberTemplate` property to Hydra which associates an `IriTemplate` with a `Collection`. That way, clients would be able to directly construct the URL of specific members, which is much more efficient than having to query the collection or iterate through all members. The same motivation triggered a request for a feature to control the sorting of paged collections. At the moment, the sorting order of members of paged collections is neither specified nor can it be directly influenced by a client. It is at the sole discretion of the server.

Since we are actively discussing the inclusion of parts of Hydra directly into Schema.org, some concepts might also get renamed to more closely align with the rest of Schema.org or extended to support other use cases. An important feature request from Schema.org was to support not only HTTP operations but also operations that cause mobile applications to be launched. This could be included directly in the core vocabulary, but as Hydra was designed to be a modular suite of vocabularies form the beginning, it is more likely that such functionality would be realized by a dedicated vocabulary. Similarly, there are plans to create a vocabulary extending Hydra to allow operations to be annotated with sample requests and responses in order to facilitate API and client testing as described in section 5.5.

While extensions and refinements such as the ones presented above are important in the long term, the most pressing concern is to improve tooling support. Due to Hydra's nature, it is probably even more important than for JSON-LD. The HydraBundle for Symfony2 as well as the HydraConsole are important first steps but both have to be improved to be usable in production and complemented by additional tools and libraries. A crucial missing piece we are already working on is a client library for programmatic access to Hydra-powered Web APIs. Its aim is to provide developers with a generic library instead of requiring them to either use a special library for each API they are accessing or a very low level HTTP library. The long-term goal is to allow a more declarative, goal-oriented usage of Web APIs. If the used vocabularies are based on formal semantics (just as RDF's core vocabularies are), it becomes possible to implement reasoners that are able to infer conclusions which are not expressed explicitly in the data. That, combined with techniques such as hierarchical state machines or behavior trees that allow the creation of reusable blocks of logic, could pave the way for much smarter clients than possible today.

7.3 Related Topics

A yet unresolved issue for the creation of smarter clients, which is not directly related to JSON-LD or Hydra but more to RDF in general, is data integration. The transformation of data from different sources to RDF is just the first and simplest step. To be fully integrated, however, all the data has to be expressed eventually in a vocabulary supported by the application processing it. This process is typically called ontology alignment and is a heavily researched area. Instead of requiring a manual mapping as the HydraBundle does, a similar approach could be used to automatically map object-oriented implementations to vocabularies such as Schema.org. Currently, such alignment and integration is mostly still done in an imperative way by writing data mediation code. To be able to cope with the exponentially growing amount of data it will become inevitable to automate these processes. It is also necessary to research on how to deal with incomplete or inconsistent data and which data sources can be trusted.

Talking about the Semantic Web stack in general, the biggest remaining hurdle to a more widespread adoption apart from missing tooling is, in our opinion, a lack of accessible documentation. When writing the JSON-LD specifications we took a radically different approach than most existing specifications. We tried our best to avoid complex terminology while still being technically accurate. Instead of just enumerating and describing features, we built the entire specification around examples. The specification contains very few sections that consist of mostly normative language. Instead, much of the document reads more like a tutorial or a primer rather than a specification. This makes the specification much longer (it was indeed criticized by a minority of people for its length) but it also means that the average Web developers do not need to read any other document to understand and begin using JSON-LD; the document is completely self-contained. We tried to advocate a similar simplification for the rest of the specifications the RDF Working Group produced, but unfortunately many of our proposals were turned down. Thus, even though the new standards are much clearer and simpler than their previous versions, even our editorship of the central RDF specification RDF 1.1 Concepts and Abstract Syntax [63] did not allow us to fully achieve our goal. It is regrettable that this unique opportunity was not fully exploited at such an important turning point of the Semantic Web. Thus, much more community outreach and the publication of accessible educational material will be needed in the coming years to foster adoption. The new RDF 1.1 Primer [246], to which we contributed several ideas and feedback, is a great first step in that direction.

References

[1] J. Manyika, M. Chui, P. Groves, D. Farrell, S. Van Kuiken, and E. A. Doshi, "Open data: Unlocking innovation and performance with liquid information," *McKinsey Quarterly*, no. October, McKinsey Global Institute, 2013.

[2] E. Berkowitz and R. Paradise, "Innovation in government: Kenya and Georgia," *McKinsey Quarterly*, pp. 1–9, Sep-2011.

[3] B. Obama, "Executive Order 13642 of May 9, 2013 Making: Making Open and Machine Readable the New Default for Government Information," *Fed. Regist.*, vol. 78, no. 93, pp. 28111–28113, 2013.

[4] IBM, "Bringing big data to the enterprise." [Online]. Available: http://www.ibm.com/software/in/data/bigdata/. [Accessed: 10-Feb-2014].

[5] T. Samson, "Ex-Amazonian urges Google to sample Amazon's secret sauce," *InfoWorld*, 2011. [Online]. Available: http://www.infoworld.com/t/service-oriented-architecture/ex-amazonian-urges-google-sample-amazons-secret-sauce-175906. [Accessed: 12-Feb-2014].

[6] P. Ranade, D. Scannell, and B. Stafford, "Ready for APIs? Three steps to unlock the data economy's most promising channel," *Forbes*, 2014. [Online]. Available: http://www.forbes.com/sites/mckinsey/2014/01/07/ready-for-apis-three-steps-to-unlock-the-data-economys-most-promising-channel/. [Accessed: 04-Feb-2014].

[7] N. Mitra and Y. Lafon, "SOAP Version 1.2 Part 0: Primer (Second Edition)," *W3C Recommendation*, 2007. [Online]. Available: http://www.w3.org/TR/soap12-part0/.

[8] J. Waldo, G. Wyant, A. Wollrath, and S. Kendall, "A Note on Distributed Computing," Mountain View, California, USA, 1994.

[9] R. T. Fielding, J. Gettys, J. C. Mogul, H. Frystyk Nielsen, L. Masinter, P. J. Leach, and T. Berners-Lee, "RFC2616: Hypertext Transfer Protocol -- HTTP/1.1," *Internet Engineering Task Force (IETF) Request for Comments*, 1999. [Online]. Available: http://tools.ietf.org/html/rfc2616.

[10] I. Jacobs and N. Walsh, "Architecture of the World Wide Web, Volume One," *W3C Recommendation*, 2004. [Online]. Available: http://www.w3.org/TR/webarch/.

[11] R. T. Fielding, “Architectural Styles and the Design of Network-based Software Architectures,” University of California, Irvine, 2000.

[12] D. Berlind, “ProgrammableWeb’s Directory Hits 10,000 APIs. And Counting,” *ProgrammableWeb*, 2013. [Online]. Available: http://blog.programmableweb.com/2013/09/23/programmablewebs-directory-hits-10000-apis-and-counting/. [Accessed: 08-Oct-2013].

[13] S. Willmott, “API Predictions 2014,” 2013. [Online]. Available: http://www.3scale.net/2013/12/api-predictions-2014/. [Accessed: 10-Jan-2014].

[14] S. Vinoski, “Serendipitous Reuse,” *IEEE Internet Comput.*, vol. 12, no. 1, pp. 84–87, Jan. 2008.

[15] T. Berners-Lee, J. Hendler, and O. Lassila, “The Semantic Web,” *Sci. Am.*, vol. 284, no. 5, pp. 34–43, 2001.

[16] J. Anderson and L. Rainie, “The Fate of the Semantic Web,” Pew Research Center, Washington, D.C., 2010.

[17] T. Berners-Lee, “Linked Data,” *Design Issues for the World Wide Web*, 2006. [Online]. Available: http://www.w3.org/DesignIssues/LinkedData.html. [Accessed: 06-Jun-2010].

[18] R. T. Fielding and J. F. Reschke, “Hypertext Transfer Protocol (HTTP/1.1): Message Syntax and Routing,” *Internet Engineering Task Force (IETF) Draft*, 2014. [Online]. Available: http://tools.ietf.org/html/draft-ietf-httpbis-p1-messaging-26.

[19] E. Wilde, “RFC6906: The ‘profile’ Link Relation Type,” *Internet Engineering Task Force (IETF) Request for Comments*, 2013. [Online]. Available: http://tools.ietf.org/html/rfc6906.

[20] M. Lanthaler, “The Profile URI Registry,” *Internet Engineering Task Force (IETF) Draft*, 2014. [Online]. Available: http://tools.ietf.org/html/draft-lanthaler-profile-registry-05.

[21] Information Sciences Institute/University of Southern California, “RFC793: Transmission Control Protocol - DARPA Internet Program Protocol Specification,” *Internet Engineering Task Force (IETF) Request for Comments*, 1981. [Online]. Available: http://tools.ietf.org/html/rfc793.

[22] Information Sciences Institute/University of Southern California, “RFC791: Internet Protocol - DARPA Internet Program Protocol Specification,” *Internet Engineering Task Force (IETF) Request for Comments*, 1981. [Online]. Available: http://tools.ietf.org/html/rfc791.

[23] T. Berners-Lee, “Information Management: A Proposal,” *CERN*, 1989. [Online]. Available: http://www.w3.org/History/1989/proposal.html. [Accessed: 23-Apr-2011].

[24] International Telecommunication Union (ITU), “The World in 2013: ICT Facts and Figures,” Geneva, Switzerland, 2013.

[25] M. Lanthaler and C. Gütl, "Model Your Application Domain, Not Your JSON Structures," in *Proceedings of the 4th International Workshop on RESTful Design (WS-REST 2013) at the 22nd International World Wide Web Conference (WWW2013)*, 2013, pp. 1415–1420.

[26] M. Lanthaler and C. Gütl, "Towards a RESTful Service Ecosystem - Perspectives and Challenges," in *Proceedings of the 2010 4th IEEE International Conference on Digital Ecosystems and Technologies (DEST)*, 2010, pp. 209–214.

[27] M. Lanthaler and C. Gütl, "SAPS: Semantic AtomPub-based Services," in *Proceedings of the 11th IEEE/IPSJ International Symposium on Applications and the Internet (SAINT)*, 2011, pp. 382–387.

[28] M. Lanthaler, M. Granitzer, and C. Gütl, "Semantic Web Services: State of the Art," in *Proceedings of the IADIS International Conference on Internet Technologies & Society (ITS 2010)*, 2010, pp. 107–114.

[29] R. Cailliau, "A Little History of the World Wide Web," *W3C*, 1995. [Online]. Available: http://www.w3.org/History.html. [Accessed: 16-Oct-2013].

[30] W. Hoogland and H. Weber, "Statement Concerning CERN W3 Software Release into Public Domain," *CERN*, 1993. [Online]. Available: http://tenyears-www.web.cern.ch/. [Accessed: 17-Oct-2013].

[31] The Minnesota Gopher Team, "University of Minnesota Gopher software licensing policy," 1993. [Online]. Available: http://www.funet.fi/pub/vms/networking/gopher/gopher-software-licensing-policy.ancient. [Accessed: 17-Oct-2013].

[32] F. Anklesaria, M. McCahill, P. Lindner, D. Johnson, D. Torrey, and B. Alberti, "RFC1436: The Internet Gopher Protocol (a distributed document search and retrieval protocol)," *Internet Engineering Task Force (IETF) Request for Comments*, 1993. [Online]. Available: http://tools.ietf.org/html/rfc1436.

[33] "Information processing -- Text and office systems -- Standard Generalized Markup Language (SGML)," International Organization for Standardization (ISO), ISO 8879:1986, 1986.

[34] T. Berners-Lee, "The Original HTTP as defined in 1991," *W3C*, 1991. [Online]. Available: http://www.w3.org/Protocols/HTTP/AsImplemented.html. [Accessed: 17-Oct-2013].

[35] T. Berners-Lee, "W3 Naming Schemes," 1992. [Online]. Available: http://info.cern.ch/hypertext/WWW/Addressing/Addressing.html. [Accessed: 17-Oct-2013].

[36] T. Berners-Lee and D. Connolly, "Hypertext Markup Language (HTML)," *Internet Engineering Task Force (IETF) Draft*, 1993. [Online]. Available: http://tools.ietf.org/html/draft-ietf-iiir-html-00.

[37] T. Berners-Lee, J.-F. Groff, and R. Cailliau, "Universal Document Identifiers on the Network," *CERN*, 1992. [Online]. Available: http://citeseerx.ist.psu.edu/viewdoc/summary?doi=10.1.1.45.1836. [Accessed: 03-Mar-2013].

[38] T. Berners-Lee, “RFC1630: Universal Resource Identifiers in WWW,” *Internet Engineering Task Force (IETF) Request for Comments*, 1994. [Online]. Available: http://tools.ietf.org/html/rfc1630.

[39] T. Berners-Lee, L. Masinter, and M. McCahill, “RFC1738: Uniform Resource Locators (URL),” *Internet Engineering Task Force (IETF) Request for Comments*, 1994. [Online]. Available: http://tools.ietf.org/html/rfc1738.

[40] T. Berners-Lee, R. T. Fielding, and L. Masinter, “RFC2396: Uniform Resource Identifiers (URI) - Generic Syntax,” *Internet Engineering Task Force (IETF) Request for Comments*, 1998. [Online]. Available: http://tools.ietf.org/html/rfc2396.

[41] T. Berners-Lee and D. Connolly, “RFC1866: Hypertext Markup Language - 2.0,” *Internet Engineering Task Force (IETF) Request for Comments*, 1995. [Online]. Available: http://tools.ietf.org/html/rfc1866.

[42] T. Berners-Lee, R. T. Fielding, and H. Frystyk Nielsen, “RFC1945: Hypertext Transfer Protocol -- HTTP/1.0,” *Internet Engineering Task Force (IETF) Request for Comments*, 1996. [Online]. Available: http://tools.ietf.org/html/rfc1945.

[43] R. T. Fielding, J. Gettys, J. C. Mogul, H. Frystyk Nielsen, and T. Berners-Lee, “RFC2068: Hypertext Transfer Protocol -- HTTP/1.1,” *Internet Engineering Task Force (IETF) Request for Comments*, 1997. [Online]. Available: http://tools.ietf.org/html/rfc2068.

[44] M. Duerst and M. Suignard, “RFC3987: Internationalized Resource Identifiers (IRIs),” *Internet Engineering Task Force (IETF) Request for Comments*, 2005. [Online]. Available: http://tools.ietf.org/html/rfc3987.

[45] R. T. Fielding, “REST APIs must be hypertext-driven,” *Untangled musings of Roy T. Fielding*, 2008. [Online]. Available: http://roy.gbiv.com/untangled/2008/200brest-apis-must-be-hypertext-driven. [Accessed: 02-Jun-2010].

[46] S. Parastatidis, J. Webber, G. Silveira, and I. S. Robinson, “The Role of Hypermedia in Distributed System Development,” in *Proceedings of the 1st International Workshop on RESTful Design (WS-REST 2010)*, 2010, pp. 16–22.

[47] J. Gregorio and B. de HOra, “RFC5023: The Atom Publishing Protocol,” *Internet Engineering Task Force (IETF) Request for Comments*, 2007. [Online]. Available: http://tools.ietf.org/html/rfc5023.

[48] M. Nottingham and R. Sayre, “RFC4287: The Atom Syndication Format,” *Internet Engineering Task Force (IETF) Request for Comments*, 2005. [Online]. Available: http://tools.ietf.org/html/rfc4287.

[49] T. Bray, J. Paoli, C. M. Sperberg-McQueen, E. Maler, F. Yergeau, and J. Cowan, “Extensible Markup Language (XML) 1.1 (Second Edition),” *W3C Recommendation*, 2006. [Online]. Available: http://www.w3.org/TR/xml11/.

[50] D. Crockford, “RFC4627: The application/json Media Type for JavaScript Object Notation (JSON),” *Internet Engineering Task Force (IETF) Request for Comments*, 2006. [Online]. Available: http://tools.ietf.org/html/rfc4627.

[51] S. Perreault, “RFC6351: xCard - vCard XML Representation,” *Internet Engineering Task Force (IETF) Request for Comments*, 2011. [Online]. Available: http://tools.ietf.org/html/rfc6351.

[52] C. Daboo, M. Douglass, and S. Lees, “RFC6321: xCal - The XML Format for iCalendar,” *Internet Engineering Task Force (IETF) Request for Comments*, 2011. [Online]. Available: http://tools.ietf.org/html/rfc6321.

[53] T. Bray, D. Hollander, A. Layman, R. Tobin, and H. S. Thompson, “Namespaces in XML 1.0 (Third Edition),” *W3C Recommendation*, 2009. [Online]. Available: http://www.w3.org/TR/xml-names/.

[54] “IANA XML Registry,” *IANA*. [Online]. Available: http://www.iana.org/assignments/xml-registry-index.html. [Accessed: 22-Feb-2013].

[55] D. Raggett, A. Le Hors, and I. Jacobs, “HTML 4.01 Specification: Meta data profiles,” *W3C Recommendation*, 1999. [Online]. Available: http://www.w3.org/TR/html401/struct/global.html#h-7.4.4.3.

[56] T. A. Inkster, “The Profile Media Type Parameter,” 2009. [Online]. Available: http://buzzword.org.uk/2009/draft-inkster-profile-parameter-00.html. [Accessed: 21-Feb-2013].

[57] M. Nottingham, “RFC5988: Web Linking,” *Internet Engineering Task Force (IETF) Request for Comments*, 2010. [Online]. Available: http://tools.ietf.org/html/rfc5988.

[58] C. Daboo, “jcardcal Working Group - JSON data formats for iCalendar and vCard: Proposed charter,” *IETF Applications Area Working Group Wiki*, 2013. [Online]. Available: http://trac.tools.ietf.org/wg/appsawg/trac/wiki/jcardcal. [Accessed: 24-Feb-2013].

[59] H. Halpin, R. Iannella, B. Suda, and N. Walsh, “Representing vCard Objects in RDF,” *W3C Member Submission*, 2010. [Online]. Available: http://www.w3.org/Submission/vcard-rdf/. [Accessed: 24-Feb-2013].

[60] T. Berners-Lee, “W3 Future Directions,” *Plenary talk at the First International World Wide Web Conference*, 1994. [Online]. Available: http://www.w3.org/Talks/WWW94Tim/. [Accessed: 24-Oct-2013].

[61] O. Lassila and R. R. Swick, “Resource Description Framework (RDF) Model and Syntax Specification,” *W3C Recommendation*, 1999. [Online]. Available: http://www.w3.org/TR/1999/REC-rdf-syntax-19990222/.

[62] N. Shadbolt, W. Hall, and T. Berners-Lee, “The Semantic Web Revisited,” *Intell. Syst. IEEE*, vol. 21, no. 3, pp. 96–101, May 2006.

[63] R. Cyganiak, D. Wood, and M. Lanthaler, “RDF 1.1 Concepts and Abstract Syntax,” *W3C Recommendation*, 2014. [Online]. Available: http://www.w3.org/TR/rdf11-concepts/.

[64] D. Beckett, “RDF/XML Syntax Specification (Revised),” *W3C Recommendation*, 2004. [Online]. Available: http://www.w3.org/TR/2004/REC-rdf-syntax-grammar-20040210/.

[65] D. Brickley and R. V. Guha, “RDF Vocabulary Description Language 1.0: RDF Schema,” *W3C Recommendation*, 2004. [Online]. Available: http://www.w3.org/TR/rdf-schema/.

[66] W3C OWL Working Group, “OWL 2 Web Ontology Language,” *W3C Recommendation*, 2009. [Online]. Available: http://www.w3.org/TR/owl2-overview/.

[67] M. Kifer and H. Boley, “RIF Overview (Second Edition),” *W3C Working Group Note*, 2013. [Online]. Available: http://www.w3.org/TR/2013/NOTE-rif-overview-20130205/.

[68] D. Reynolds, “OWL 2 RL in RIF (Second Edition),” *W3C Working Group Note*, 2013. [Online]. Available: http://www.w3.org/TR/2013/NOTE-rif-owl-rl-20130205/.

[69] S. Hawke and A. Polleres, “RIF In RDF (Second Edition),” *W3C Working Group Note*, 2013. [Online]. Available: http://www.w3.org/TR/2013/NOTE-rif-in-rdf-20130205/.

[70] D. Beckett, “Re: what does SPARQL stand for? (issues#languageProtocolName),” *public-rdf-dawg@w3.org Mail Archives*, 2004. [Online]. Available: http://lists.w3.org/Archives/Public/public-rdf-dawg/2004OctDec/0453.html. [Accessed: 25-Oct-2013].

[71] S. Harris and A. Seaborne, “SPARQL 1.1 Query Language,” *W3C Recommendation*, 2013. [Online]. Available: http://www.w3.org/TR/sparql11-query/.

[72] P. Gearon, A. Passant, and A. Polleres, “SPARQL 1.1 Update,” *W3C Recommendation*, 2013. [Online]. Available: http://www.w3.org/TR/sparql11-update/.

[73] A. Seaborne, A. Polleres, L. Feigenbaum, and G. T. Williams, “SPARQL 1.1 Federated Query,” *W3C Recommendation*, 2013. [Online]. Available: http://www.w3.org/TR/sparql11-federated-query/.

[74] G. T. Williams, “SPARQL 1.1 Service Description,” *W3C Recommendation*, 2013. [Online]. Available: http://www.w3.org/TR/sparql11-service-description/.

[75] L. Feigenbaum, G. T. Williams, K. G. Clark, and E. Torres, “SPARQL 1.1 Protocol,” *W3C Recommendation*, 2013. [Online]. Available: http://www.w3.org/TR/sparql11-protocol/.

[76] C. Ogbuji, “SPARQL 1.1 Graph Store HTTP Protocol,” *W3C Recommendation*, 2013. [Online]. Available: http://www.w3.org/TR/sparql11-http-rdf-update/.

[77] S. Hawke, “SPARQL Query Results XML Format (Second Edition),” *W3C Recommendation*, 2013. [Online]. Available: http://www.w3.org/TR/rdf-sparql-XMLres/.

[78] K. G. Clark, L. Feigenbaum, and E. Torres, “SPARQL 1.1 Query Results JSON Format,” *W3C Recommendation*, 2013. [Online]. Available: http://www.w3.org/TR/sparql11-results-json/.

[79] A. Seaborne, “SPARQL 1.1 Query Results CSV and TSV Formats,” *W3C Recommendation*, 2013. [Online]. Available: http://www.w3.org/TR/sparql11-results-csv-tsv/.

[80] R. Cyganiak and A. Jentzsch, “The Linking Open Data cloud diagram,” 2011. [Online]. Available: http://lod-cloud.net/. [Accessed: 04-Mar-2013].

[81] W3C, “httpRange-14: What is the range of the HTTP dereference function?,” *W3C TAG Issues List*, 2005. [Online]. Available: http://www.w3.org/2001/tag/issues.html?type=1#httpRange-14. [Accessed: 24-Oct-2013].

[82] L. Sauermann and R. Cyganiak, “Cool URIs for the Semantic Web,” *W3C Note*, 2008. [Online]. Available: http://www.w3.org/TR/2008/NOTE-cooluris-20081203/.

[83] J. Tennison, “Using ‘Punning’ to Answer httpRange-14,” *Jeni’s Musings*, 2012. [Online]. Available: http://www.jenitennison.com/blog/node/170. [Accessed: 04-Jun-2012].

[84] J. Tennison, “URLs in Data Primer,” *W3C Working Draft*, 2013. [Online]. Available: http://www.w3.org/TR/2013/WD-urls-in-data-20130604/.

[85] “Schema.org,” 2011. [Online]. Available: http://www.schema.org/. [Accessed: 24-Aug-2013].

[86] “The Open Graph protocol,” 2010. [Online]. Available: http://opengraphprotocol.org/. [Accessed: 07-Jul-2010].

[87] R. Chinnici, J.-J. Moreau, A. Ryman, and S. Weerawarana, “Web Services Description Language (WSDL) Version 2.0 Part 1: Core Language,” *W3C Recommendation*, 2007. [Online]. Available: http://www.w3.org/TR/wsdl20/.

[88] L. Clement, A. Hately, C. von Riegen, and T. Rogers, “UDDI Version 3.0.2,” *OASIS Standard*, 2004. [Online]. Available: http://uddi.org/pubs/uddi_v3.htm.

[89] S. (Sandy) Gao 高殊镝, C. M. Sperberg-McQueen, and H. S. Thompson, “W3C XML Schema Definition Language (XSD) 1.1 Part 1: Structures,” *W3C Recommendation*, 2012. [Online]. Available: http://www.w3.org/TR/xmlschema11-1/.

[90] D. Peterson, S. (Sandy) Gao 高殊镝, A. Malhotra, C. M. Sperberg-McQueen, and H. S. Thompson, “W3C XML Schema Definition Language (XSD) 1.1 Part 2: Datatypes,” *W3C Recommendation*, 2012. [Online]. Available: http://www.w3.org/TR/xmlschema11-2/.

[91] S. Loughran and E. Smith, “Rethinking the Java SOAP Stack,” in *IEEE International Conference on Web Services (ICWS)*, 2005, vol. 5, no. July.

[92] T. Vitvar and J. Musser, “ProgrammableWeb.com: Statistics, Trends, and Best Practices,” in *Keynote of the Web APIs and Service Mashups Workshop at the European Conference on Web Services (ECOWS 2010)*, 2010, vol. 234.

[93] M. Fowler, “Richardson Maturity Model - steps toward the glory of REST,” 2010. [Online]. Available: http://martinfowler.com/articles/richardsonMaturityModel.html. [Accessed: 28-Oct-2013].

[94] T. Berners-Lee, *Weaving the Web: The Original Design and Ultimate Destiny of the World Wide Web*, 5th ed. HarperBusiness, 2000.

[95] M. Lanthaler and C. Gütl, "A Semantic Description Language for RESTful Data Services to Combat Semaphobia," in *Proceedings of the 2011 5th IEEE International Conference on Digital Ecosystems and Technologies (DEST)*, 2011, pp. 47–53.

[96] M. Lanthaler and C. Gütl, "Hydra: A Vocabulary for Hypermedia-Driven Web APIs," in *Proceedings of the 6th Workshop on Linked Data on the Web (LDOW2013) at the 22nd International World Wide Web Conference (WWW2013)*, 2013.

[97] M. Lanthaler and C. Gütl, "Aligning Web Services with the Semantic Web to Create a Global Read-Write Graph of Data," in *Proceedings of the 9th IEEE European Conference on Web Services (ECOWS 2011)*, 2011, pp. 15–22.

[98] M. Lanthaler and C. Gütl, "On Using JSON-LD to Create Evolvable RESTful Services," in *Proceedings of the 3rd International Workshop on RESTful Design (WS-REST 2012) at the 21st International World Wide Web Conference (WWW2012)*, 2012, pp. 25–32.

[99] M. Lanthaler, "Leveraging Linked Data to Build Hypermedia-Driven Web APIs," in *REST: Advanced Research Topics and Practical Applications*, C. Pautasso, E. Wilde, and R. Alarcón, Eds. Springer New York, 2014, pp. 107-123.

[100] "Standard ECMA-262 3rd Edition - December 1999," ECMA International, 1999.

[101] "XML Core Working Group Public Page– Pubblications," *XML Core Working Group*, 2012. [Online]. Available: http://www.w3.org/XML/Core/#Publications. [Accessed: 17-Aug-2013].

[102] "MIME Media Types," *IANA*. [Online]. Available: http://www.iana.org/assignments/media-types. [Accessed: 22-Feb-2013].

[103] T. Hansen and A. Melnikov, "RFC6839: Additional Media Type Structured Syntax Suffixes," *Internet Engineering Task Force (IETF) Request for Comments*, 2013. [Online]. Available: http://tools.ietf.org/html/rfc6839.

[104] M. Murata, S. St. Laurent, and D. Kohn, "RFC3023: XML Media Types," *Internet Engineering Task Force (IETF) Request for Comments*, 2001. [Online]. Available: http://tools.ietf.org/html/rfc3023.

[105] R. Berjon, S. Faulkner, T. Leithead, E. D. Navara, E. O'Connor, S. Pfeiffer, and I. Hickson, "HTML5: 4.10.19.8 Autofilling form controls: the autocomplete attribute," *W3C Candidate Recommendation*, 2014. [Online]. Available: http://www.w3.org/TR/2014/CR-html5-20140204/forms.html#association-of-controls-and-forms.

[106] D. van Heesch, "Generate documentation from source code," *Doxygen*, 2012. [Online]. Available: http://www.stack.nl/~dimitri/doxygen/. [Accessed: 31-Jul-2012].

[107] K. R. Page, D. C. De Roure, and K. Martinez, “REST and Linked Data: a match made for domain driven development?,” in *Proceedings of the 2nd International Workshop on RESTful Design (WS-REST 2011)*, 2011, pp. 22–25.

[108] D. Brickley and L. Miller, “FOAF Vocabulary Specification 0.98,” 2010. [Online]. Available: http://xmlns.com/foaf/spec/20100809.html. [Accessed: 17-Jan-2011].

[109] I. Hickson, “HTML Microdata,” *W3C Working Group Note*, 2013. [Online]. Available: http://www.w3.org/TR/2013/NOTE-microdata-20131029/.

[110] M. Fowler, *Patterns of Enterprise Application Architecture*, 1st ed. Addison-Wesley Professional, 2002.

[111] S. DeRose, E. Maler, D. Orchard, and N. Walsh, “XML Linking Language (XLink) Version 1.1,” *W3C Recommendation*, 2010. [Online]. Available: http://www.w3.org/TR/xlink11/.

[112] “XML Schema 1.0,” *W3C Recommendation*. W3C, 2004.

[113] P. Prescod, “Web Resource Description Language (‘Word-dul’).”[Online]. Available: http://www.prescod.net/rest/wrdl/wrdl.html. [Accessed: 23-Jan-2010].

[114] N. Walsh, “WITW: NSDL,” 2005. [Online]. Available: http://norman.walsh.name/2005/03/12/nsdl. [Accessed: 10-Oct-2010].

[115] T. Bray, “SMEX-D,” *ongoing*, 2005. [Online]. Available: http://www.tbray.org/ongoing/When/200x/2005/05/03/SMEX-D. [Accessed: 06-Jul-2010].

[116] J. Cowan, “Resedel,” *Recycled Knowledge*, 2005. [Online]. Available: http://recycledknowledge.blogspot.it/2005/05/resedel.html. [Accessed: 05-Jul-2010].

[117] R. Salz, “Really Simple Web Service Descriptions,” *XML.com*, 2003. [Online]. Available: http://www.xml.com/pub/a/ws/2003/10/14/salz.html. [Accessed: 07-Jul-2010].

[118] D. Orchard, “Web Description Language (WDL),” 2006. [Online]. Available: http://pacificspirit.com/Authoring/WDL/. [Accessed: 05-Jul-2010].

[119] J. Farrell and H. Lausen, “Semantic Annotations for WSDL and XML Schema,” *W3C Recommendation*, 2007. [Online]. Available: http://www.w3.org/TR/sawsdl/.

[120] M. Klusch, “Semantic Web Service Description,” in *CASCOM: Intelligent Service Coordination in the Semantic Web*, M. Schumacher, H. Schuldt, and H. Helin, Eds. Basel: Birkhäuser Basel, 2008, pp. 31–57.

[121] M. J. Hadley, “Web Application Description Language,” *W3C Member Submission*, 2009. [Online]. Available: http://www.w3.org/Submission/wadl/. [Accessed: 05-Mar-2010].

[122] “Swagger: A simple, open standard for describing REST APIs with JSON,” *Reverb Technologies*, 2013. [Online]. Available: https://developers.helloreverb.com/swagger/. [Accessed: 04-Mar-2013].

[123] "Google APIs Discovery Service," *Google Inc.*, 2013. [Online]. Available: https://developers.google.com/discovery/. [Accessed: 07-Mar-2013].

[124] Mashery, "I/O Docs," 2011. [Online]. Available: http://www.mashery.com/product/io-docs. [Accessed: 01-Sep-2013].

[125] Apiary, "API Blueprint - Connecting the dots in API development," 2011. .

[126] RAML Workgroup, "RAML^{TM} Version 0.8: RESTful API Modeling Language," 2013. [Online]. Available: http://raml.org/spec.html. [Accessed: 02-Feb-2014].

[127] J. Lathem, K. Gomadam, and A. P. Sheth, "SA-REST and (S)mashups: Adding Semantics to RESTful Services," in *International Conference on Semantic Computing 2007 (ICSC2007)*, 2007, pp. 469–476.

[128] B. Adida, M. Birbeck, S. McCarron, and I. Herman, "RDFa Core 1.1 - Second Edition," *W3C Recommendation*, 2013. [Online]. Available: http://www.w3.org/TR/rdfa-core/.

[129] R. Khare and T. Çelik, "Microformats: A Pragmatic Path to the Semantic Web CommerceNet Labs," Palo Alto, CA, USA, Tech. Rep. CN-TR 06-01, 2006.

[130] J. Kopecký, T. Vitvar, and D. Fensel, "WSMO Deliverable D38V0.1 - MicroWSMO: Semantic Description of RESTful Services," 2008.

[131] M. Maleshkova and J. Kopecký, "Adapting SAWSDL for Semantic Annotations of RESTful Services," in *On the Move to Meaningful Internet Systems: OTM 2009 Workshops*, 2009, pp. 917–926.

[132] R. Verborgh, T. Steiner, D. Van Deursen, S. Coppens, J. G. Vallés, and R. Van de Walle, "Functional Descriptions as the Bridge between Hypermedia APIs and the Semantic Web," in *Proceedings of the 3rd International Workshop on RESTful Design (WS-REST 2012) at the 21st International World Wide Web Conference (WWW2012)*, 2012, pp. 33–40.

[133] T. Berners-Lee and D. Connolly, "Notation3 (N3): A readable RDF syntax," *W3C Team Submission*, 2011. [Online]. Available: http://www.w3.org/TeamSubmission/n3/. [Accessed: 07-Mar-2013].

[134] A. Berglund, S. Boag, D. Chamberlin, M. F. Fernández, M. Kay, J. Robie, and J. Siméon, "XML Path Language (XPath) 2.0 (Second Edition)," *W3C Recommendation*, 2010. [Online]. Available: http://www.w3.org/TR/2010/REC-xpath20-20101214/.

[135] S. Boag, D. Chamberlin, M. F. Fernández, D. Florescu, J. Robie, and J. Siméon, "XQuery 1.0: An XML Query Language (Second Edition)," *W3C Recommendation*, 2010. [Online]. Available: http://www.w3.org/TR/2010/REC-xquery-20101214/.

[136] M. Kay, "XSL Transformations (XSLT) Version 2.0," *W3C Recommendation*, 2007. [Online]. Available: http://www.w3.org/TR/2007/REC-xslt20-20070123/.

[137] D. Beckett, “A retrospective on the development of the RDF/XML Revised Syntax,” Institute for Learning and Research Technology, University of Bristol, Bristol, United Kingdom, Research Report 1017, 2003.

[138] W3C, “RDF Working Group Charter,” 2011. [Online]. Available: http://www.w3.org/2011/01/rdf-wg-charter. [Accessed: 06-Oct-2013].

[139] D. Beckett and T. Berners-Lee, “Turtle - Terse RDF Triple Language,” *W3C Team Submission*, 2011. [Online]. Available: http://www.w3.org/TeamSubmission/turtle/. [Accessed: 06-Oct-2013].

[140] E. Prud’hommeaux and G. Carothers, “RDF 1.1 Turtle - Terse RDF Triple Language,” *W3C Recommendation*, 2014. [Online]. Available: http://www.w3.org/TR/turtle/.

[141] G. Carothers and A. Seaborne, “RDF 1.1 TriG - RDF Dataset Language,” *W3C Recommendation*, 2014. [Online]. Available: http://www.w3.org/TR/trig/.

[142] G. Carothers and A. Seaborne, “RDF 1.1 N-Triples - A line-based syntax for an RDF graph,” *W3C Recommendation*, 15-Jul-2014. [Online]. Available: http://www.w3.org/TR/n-triples/.

[143] G. Carothers, “RDF 1.1 N-Quads - A line-based syntax for an RDF datasets,” *W3C Recommendation*, 2014. [Online]. Available: http://www.w3.org/TR/n-quads/.

[144] M. Sporny, G. Kellogg, and M. Lanthaler, “JSON-LD 1.0 - A JSON-based Serialization for Linked Data,” *W3C Recommendation*, 2014. [Online]. Available: http://www.w3.org/TR/json-ld/.

[145] D. Beckett, “New Syntaxes for RDF,” *Paper submitted to the 2004 World Wide Web Conference (WWW2004) but rejected*, 2003. [Online]. Available: http://www.dajobe.org/2003/11/new-syntaxes-rdf/paper.pdf. [Accessed: 10-Oct-2013].

[146] J. Grant and D. Beckett, “RDF Test Cases,” *W3C Recommendation*, 2004. [Online]. Available: http://www.w3.org/TR/rdf-testcases/.

[147] G. Klyne and J. J. Carroll, “Resource Description Framework (RDF): Concepts and Abstract Syntax,” *W3C Recommendation*, 2004. [Online]. Available: http://www.w3.org/TR/rdf-concepts/.

[148] “OWL-S Semantic Markup for Web Services,” *W3C Member Submission*, 2004. [Online]. Available: http://www.w3.org/Submission/OWL-S/. [Accessed: 20-Jun-2010].

[149] R. Lara, A. Polleres, H. Lausen, D. Roman, J. De Bruijn, and D. Fensel, “A Conceptual Comparison between WSMO and OWL-S,” 2005.

[150] D. Roman, U. Keller, H. Lausen, and J. De Bruijn, “Web Service Modeling Ontology,” *Appl. Ontol.*, vol. 1, no. 1, pp. 77–106, 2005.

[151] C. Bournez, “Team Comment on Web Service Modeling Ontology (WSMO) Submission,” *W3C Submissions*, 2005. [Online]. Available: http://www.w3.org/Submission/2005/06/Comment.html. [Accessed: 10-Jul-2010].

[152] T. Vitvar, J. Kopecký, J. Viskova, and D. Fensel, "WSMO-Lite Annotations for Web Services," in *5th European Semantic Web Conference (ESWC 2008), LNCS 5021*, 2008, pp. 674–689.

[153] A. Alowisheq and D. E. Millard, "EXPRESS: EXPressing REstful Semantic Services," in *Proceedings of the 2009 IEEE/WIC/ACM International Joint Conference on Web Intelligence and Intelligent Agent Technology*, 2009, pp. 453-456.

[154] W3C, "Linked Data Platform (LDP) Working Group Charter," 2012. [Online]. Available: http://www.w3.org/2012/ldp/charter. [Accessed: 03-Sep-2013].

[155] S. Speicher, J. Arwe, and A. Malhotra, "Linked Data Platform 1.0," *W3C Last Call Working Draft*, 2013. [Online]. Available: http://www.w3.org/TR/2013/WD-ldp-20130730/.

[156] "Google Data Protocol," 2010. [Online]. Available: http://code.google.com/apis/gdata/. [Accessed: 07-Jul-2010].

[157] "Open Data Protocol," 2010. [Online]. Available: http://www.odata.org/. [Accessed: 09-Jul-2010].

[158] "Developer's Guide: JSON-C / JavaScript," 2010. [Online]. Available: http://code.google.com/apis/youtube/2.0/developers_guide_jsonc.html. [Accessed: 07-Jul-2010].

[159] D. Clinton, "OpenSearch 1.1 Draft 5," 2012. [Online]. Available: http://www.opensearch.org/Specifications/OpenSearch/1.1/Draft_5. [Accessed: 25-Jul-2012].

[160] C. Henderson, M. Malone, L. Culver, and R. Crowley, "oEmbed," 2012. [Online]. Available: http://www.oembed.com/. [Accessed: 25-Jul-2012].

[161] F. Galiegue, K. Zyp, and G. Court, "JSON Schema: core definitions and terminology," *Internet Engineering Task Force (IETF) Draft*, 2013. [Online]. Available: http://tools.ietf.org/html/draft-zyp-json-schema-04.

[162] P. C. Bryan and K. Zyp, "JSON Reference," *Internet Engineering Task Force (IETF) Draft*, 2012. [Online]. Available: http://tools.ietf.org/html/draft-pbryan-zyp-json-ref-03.

[163] M. Kelly, "JSON Hypertext Application Language," *Internet Engineering Task Force (IETF) Draft*, 2013. [Online]. Available: http://tools.ietf.org/html/draft-kelly-json-hal-06.

[164] M. Amundsen, "Collection+JSON - Hypermedia Type," 2011. [Online]. Available: http://amundsen.com/media-types/collection/. [Accessed: 29-Oct-2012].

[165] Y. Y. Goland, "Adding Namespaces to JSON," 2006. [Online]. Available: http://www.goland.org/jsonnamespace/. [Accessed: 31-Aug-2013].

[166] RDF Working Group, "JSON Serialization By Example," 2011. [Online]. Available: http://www.w3.org/2011/rdf-wg/wiki/JSON-Serialization-Examples. [Accessed: 31-Aug-2013].

[167] M. Lanthaler and C. Gütl, “Seamless Integration of RESTful Services into the Web of Data,” *Adv. Multimed.*, vol. 2012, pp. 1–14, 2012.

[168] M. Lanthaler, G. Kellogg, and M. Sporny, “JSON-LD 1.0 Processing Algorithms and API,” *W3C Recommendation*, 2014. [Online]. Available: http://www.w3.org/TR/json-ld-api/.

[169] M. Lanthaler, “Creating 3rd Generation Web APIs with Hydra,” in *Proceedings of the 22nd International World Wide Web Conference (WWW2013)*, 2013, pp. 35-37.

[170] M. Lanthaler, “Hydra Core Vocabulary Specification,” 2014. [Online]. Available: http://www.markus-lanthaler.com/hydra/. [Accessed: 22-Feb-2014].

[171] E. Ries, *The Lean Startup: How Today’s Entrepreneurs Use Continuous Innovation to Create Radically Successful Businesses*, 1st editio. New York, NY, USA: Crown Business, 2011.

[172] M. Birbeck and S. McCarron, “CURIE Syntax 1.0 - A syntax for expressing Compact URIs,” *W3C Working Group Note*, 2010. [Online]. Available: http://www.w3.org/TR/2010/NOTE-curie-20101216.

[173] S. Vinoski, “Demystifying RESTful Data Coupling,” *IEEE Internet Comput.*, vol. 12, no. 2, pp. 87–90, Mar. 2008.

[174] S. Vinoski, “RPC Under Fire,” *IEEE Internet Comput.*, vol. 9, no. 5, pp. 93–95, Sep. 2005.

[175] K. Zyp and G. Court, “A JSON Media Type for Describing the Structure and Meaning of JSON Documents,” 2010. [Online]. Available: http://tools.ietf.org/html/draft-zyp-json-schema-03.

[176] P. C. Bryan, K. Zyp, and M. Nottingham, “RFC6901: JavaScript Object Notation (JSON) Pointer,” *Internet Engineering Task Force (IETF) Request for Comments*, 2013. [Online]. Available: http://tools.ietf.org/html/rfc6901.

[177] J. Gregorio, R. T. Fielding, M. Hadley, M. Nottingham, and D. Orchard, “RFC6570: URI Template,” *Internet Engineering Task Force (IETF) Request for Comments*, 2012. [Online]. Available: http://tools.ietf.org/html/rfc6570.

[178] S. Das, S. Sundara, and R. Cyganiak, “R2RML: RDB to RDF Mapping Language,” *W3C Recommendation*, 2012. [Online]. Available: http://www.w3.org/TR/r2rml/.

[179] J. Bao, E. F. Kendall, D. L. McGuinness, and P. F. Patel-Schneider, “OWL 2 Web Ontology Language Quick Reference Guide,” *W3C Recommendation*, 2009. [Online]. Available: http://www.w3.org/TR/2009/REC-owl2-quick-reference-20091027/.

[180] M. Sporny, “Linked JSON: RDF for the Masses,” *The Beautiful, Tormented Machine*, 2011. [Online]. Available: http://manu.sporny.org/2011/linked-json/. [Accessed: 28-Apr-2011].

[181] I. Herman, “W3C Workshop — RDF Next Steps Workshop Report,” 2010. [Online]. Available: http://www.w3.org/2009/12/rdf-ws/Report.html. [Accessed: 05-Aug-2010].

[182] Talis Systems Ltd., "RDF/JSON," 2011. [Online]. Available: http://docs.api.talis.com/platform-api/output-types/rdf-json. [Accessed: 15-Jan-2012].

[183] T. Steiner, "JSON Emergency Brake," *RDF Working Group mailing list*, 2011. [Online]. Available: http://lists.w3.org/Archives/Public/public-rdf-wg/2011Aug/0131.html. [Accessed: 23-Aug-2011].

[184] M. Birbeck and M. Sporny, "JSON-LD - Linked Data Expression in JSON," *Unofficial Draft 30 May 2010*, 2010. [Online]. Available: http://json-ld.org/spec/ED/json-ld-syntax/20100529/. [Accessed: 06-Aug-2012].

[185] L. Dodds and I. Davis, "Linked Data Patterns - A pattern catalogue for modelling, publishing, and consuming Linked Data," 2012. [Online]. Available: http://patterns.dataincubator.org/book/linked-data-patterns.pdf. [Accessed: 06-Aug-2012].

[186] P. V. Biron and A. Malhotra, "XML Schema Part 2: Datatypes Second Edition." W3C, 2004.

[187] M. Sporny, G. Kellogg, D. Longley, and M. Lanthaler, "JSON-LD Framing 1.0," *W3C Community Group Draft Report*, 2013. [Online]. Available: http://json-ld.org/spec/latest/json-ld-framing/. [Accessed: 28-Mar-2013].

[188] M. Lanthaler and C. Gütl, "A Web of Things to Reduce Energy Wastage," in *Proceedings of the 10th IEEE International Conference on Industrial Informatics (INDIN)*, 2012, pp. 1050–1055.

[189] M. Lanthaler, "Integration of Hydra into Schema.org," *Unofficial Draft*, 2014. [Online]. Available: http://www.hydra-cg.com/spec/latest/schema.org/. [Accessed: 22-Feb-2014].

[190] B. Motik, P. F. Patel-Schneider, and B. Parsia, "OWL 2 Web Ontology Language Structural Specification and Functional-Style Syntax (Second Edition)," *W3C Recommendation*, 2012. [Online]. Available: http://www.w3.org/TR/owl2-syntax/.

[191] J. Koch, C. A. Velasco, and P. Ackermann, "HTTP Vocabulary in RDF 1.0," *W3C Working Draft*, 2011. [Online]. Available: http://www.w3.org/TR/2011/WD-HTTP-in-RDF10-20110510/.

[192] R. T. Fielding, "[httpRange-14] Resolved," *W3C Technical Architecture Group Mailing List*, 2005. [Online]. Available: http://lists.w3.org/Archives/Public/www-tag/2005Jun/0039.html. [Accessed: 13-Jun-2010].

[193] E. Gamma, R. Helm, R. Johnson, and J. Vlissides, *Design Patterns - Elements of Reusable Object-Oriented Software*, 1st ed. Addison-Wesley, 1994.

[194] T. Berners-Lee, "Principles of Design," *Design Issues for the World Wide Web*, 1998. [Online]. Available: http://www.w3.org/DesignIssues/Principles.html. [Accessed: 21-Feb-2013].

[195] MongoDB Inc., "MongoDB." [Online]. Available: https://www.mongodb.com/products/mongodb.

[196] Elasticsearch BV, “Elasticsearch.” [Online].
Available: http://www.elasticsearch.org/.

[197] D. Stenberg, “cURL.” [Online]. Available: http://curl.haxx.se/.

[198] “Symfony2.” [Online]. Available: http://symfony.com/.
[Accessed: 04-May-2013].

[199] “PHP: Hypertext Preprocessor.” [Online]. Available: http://www.php.net/.
[Accessed: 04-May-2013].

[200] T. M. H. Reenskaug, “The original MVC reports,” 1979. [Online].
Available: http://heim.ifi.uio.no/~trygver/2007/MVC_Originals.pdf.
[Accessed: 06-Dec-2013].

[201] “Symfony2 Routing Component.” [Online]. Available:
http://symfony.com/doc/current/components/routing/index.html.
[Accessed: 04-May-2013].

[202] “Doctrine Object-Relational Mapper.” [Online]. Available:
http://www.doctrine-project.org/projects/orm.html. [Accessed: 04-May-2013].

[203] “Composer - Dependency Manager for PHP.” [Online].
Available: http://getcomposer.org/. [Accessed: 05-May-2013].

[204] M. Lanthaler, “HydraBundle.” [Online].
Available: https://github.com/lanthaler/HydraBundle. [Accessed: 07-Jan-2014].

[205] M. Lanthaler, “HydraConsole.” [Online]. Available:
https://github.com/lanthaler/HydraConsole. [Accessed: 10-May-2013].

[206] “jQuery.” [Online]. Available: http://jquery.com/. [Accessed: 10-May-2013].

[207] “Bootstrap.” [Online]. Available: http://getbootstrap.com/.
[Accessed: 10-May-2013].

[208] “Backbone.js.” [Online]. Available: http://backbonejs.org/.
[Accessed: 10-May-2013].

[209] “Underscore.js.” [Online]. Available: http://underscorejs.org/.
[Accessed: 10-May-2013].

[210] M. Lanthaler, “JsonLD.” [Online].
Available: https://github.com/lanthaler/JsonLD. [Accessed: 10-May-2013].

[211] A. van Kesteren, “Cross-Origin Resource Sharing,”
W3C Proposed Recommendation, 2013. [Online].
Available: http://www.w3.org/TR/2013/PR-cors-20131205/.

[212] A. J. Hester, “Socio-technical systems theory as a diagnostic tool for examining underutilization of wiki technology,” *Learn. Organ.*, vol. 21, no. 1, pp. 48–68, 2014.

[213] V. Venkatesh, “Determinants of Perceived Ease of Use: Integrating Control, Intrinsic Motivation, and Emotion into the Technology Acceptance Model,” *Inf. Syst. Res.*, vol. 11, no. 4, pp. 342–365, Dec. 2000.

[214] F. D. Davis, “Perceived usefulness, perceived ease of use, and user acceptance of information technology,” *MIS Q.*, vol. 13, no. 3, pp. 319–340, 1989.

[215] E. M. Rogers, *Diffusion of Innovations*, 5th ed. New York, NY, USA: Free Press, 2003.

[216] "Hydra Community Group," *W3C Commnunity and Business Groups*. [Online]. Available: http://www.w3.org/community/hydra/. [Accessed: 19-Nov-2013].

[217] "IKS—The Semantic CMS Community." [Online]. Available: http://www.iks-project.eu/. [Accessed: 05-Jan-2014].

[218] European Commission, "IKS: Interactive knowledge stack for small to medium CMS/KMS providers," *Community Research and Development Information Service (CORDIS)*. [Online]. Available: http://cordis.europa.eu/projects/rcn/89486_en.html. [Accessed: 05-Jan-2014].

[219] F. Christ and B. Nagel, "A Reference Architecture for Semantic Content Management Systems," in *Proceeding of the Enterprise Modelling and Information Systems Architectures Workshop 2011 (EMISA'11)*, 2011, vol. 231527, no. 231527, pp. 135–148.

[220] IKS Project, "VIE.js: Semantic Interaction Framework." [Online]. Available: http://viejs.org/. [Accessed: 05-Jan-2014].

[221] "Apache Stanbol," *Apache Software Foundation*. [Online]. Available: http://stanbol.apache.org/. [Accessed: 05-Jan-2014].

[222] S. Grünwald and H. Bergius, "Decoupling Content Management," in *Proceedings of the 21st International World Wide Web Conference (WWW2012)*, 2012.

[223] "Content Repository for Java^{TM} Technology API 2.0 Specification," 2009. [Online]. Available: https://jcp.org/en/jsr/detail?id=283.

[224] J. Lehmann, R. Isele, M. Jakob, A. Jentzsch, D. Kontokostas, P. N. Mendes, S. Hellmann, M. Morsey, P. van Kleef, S. Auer, and C. Bizer, "DBpedia – A Large-scale, Multilingual Knowledge Base Extracted from Wikipedia," *Semant. Web J. (in Press.*, 2014.

[225] Galileo University, "Galileo Educational System." [Online]. Available: http://ges.galileo.edu/geswiki/research.

[226] Google Inc., "Google Docs." [Online]. Available: https://docs.google.com/.

[227] MeisterLabs GmbH, "MindMeister." [Online]. Available: http://www.mindmeister.com/.

[228] Nulab Inc., "Cacoo." [Online]. Available: https://cacoo.com/.

[229] J. O. Donovan, "The World Cup and a call to action around Linked Data," *BBC Internet Blog*, 2010. .

[230] D. Rogers, "Introducing the BBC's Linked Data Platform and APIs," *Presentation at the QCon London Software Development Conference*, 2013. [Online]. Available: http://qconlondon.com/london-2013/presentation/Introducing the BBC's Linked Data Platform and APIs. [Accessed: 05-Jan-2014].

[231] Í. Medeiros, “Linked Data at globo.com,” *Presentation at the 2nd International Workshop on Web of Linked Entities (WoLE)*, 2013. [Online]. Available: http://www.slideshare.net/icaromedeiros/linked-data-at-globocom. [Accessed: 15-Jan-2014].

[232] C. Cherubino and S. Agarwal, “Actions in the inbox, powered by schemas,” *Presentation at the Google I/O 2013*, 2013. [Online]. Available: https://developers.google.com/events/io/sessions/327735537. [Accessed: 08-Jun-2013].

[233] S. Agarwal, “Take action right from the inbox,” *Official Gmail Blog*, 2013. [Online]. Available: http://gmailblog.blogspot.com/2013/05/take-action-right-from-inbox.html. [Accessed: 08-Jun-2013].

[234] “Actions in the Inbox: Rsvp Action,” *Google Developers*, 2013. [Online]. Available: https://developers.google.com/gmail/actions/reference/rsvp-action. [Accessed: 16-Jan-2014].

[235] D. Brickley, “Schema.org and JSON-LD,” *schema blog—Official blog for schema.org*, 2013. [Online]. Available: http://blog.schema.org/2013/06/schemaorg-and-json-ld.html. [Accessed: 08-Jun-2013].

[236] IMS Global Learning Consortium, “IMS Global Learning Consortium Releases Initial Public Draft of Learning Tools Interoperability 2,” *Press Release*, 2012. [Online]. Available: http://www.imsglobal.org/pressreleases/pr121113.html. [Accessed: 16-Jan-2014].

[237] International Image Interoperability Framework (IIIF) Working Group, “International Image Interoperability Framework: Image API 1.1,” 2013. [Online]. Available: http://www-sul.stanford.edu/iiif/image-api/1.1/. [Accessed: 16-Jan-2014].

[238] R. Sanderson, P. Ciccarese, and H. Van de Sompel, “Open Annotation Data Model,” *W3C Community Group Draft*, 2013. [Online]. Available: http://www.openannotation.org/spec/core/20130208/. [Accessed: 16-Jan-2014].

[239] J. Öberg and S. Myles, “ODRL 2 JSON Encoding,” *W3C Community Group Draft*, 2014. [Online]. Available: http://www.w3.org/community/odrl/work/json/. [Accessed: 18-Jan-2014].

[240] J. M. Snell, “JSON Activity Streams 2.0,” *Internet Engineering Task Force (IETF) Draft*, 2013. [Online]. Available: http://tools.ietf.org/html/draft-snell-activitystreams-05.

[241] J. M. Snell and M. Marum, “JSON Activity Streams 2.0 - Action Handlers,” *Internet Engineering Task Force (IETF) Draft*, 2014. [Online]. Available: http://tools.ietf.org/html/draft-snell-activitystreams-actions-03.

[242] M. Hepp, “GoodRelations: An Ontology for Describing Products and Services Offers on the Web,” in *Proceedings of the 16th International Conference on Knowledge Engineering and Knowledge Management (EKAW 2008), LNCS 5268*, 2008, pp. 329–346.

[243] M. Hepp, "i think that #hydra and #json-ld are very promising candidates for really advancing the #www in 2014," *Twitter*, 2014. [Online]. Available: https://twitter.com/mfhepp/status/421320663920807937. [Accessed: 21-Jan-2014].

[244] JSON-LD Community Group, "json-ld/json-ld.org Issues." [Online]. Available: https://github.com/json-ld/json-ld.org/issues.

[245] JSON-LD Community Group, "public-linked-json@w3.org Mail Archives." [Online]. Available: http://lists.w3.org/Archives/Public/public-linked-json/.

[246] G. Schreiber and Y. Raimond, "RDF 1.1 Primer," *W3C Working Group Note*, 2014. [Online]. Available: http://www.w3.org/TR/rdf11-primer/.

[247] B. Nowack, "The Semantic Web Technology Stack (not a piece of cake...)," *BNode*, 2009. [Online]. Available: http://bnode.org/blog/2009/07/08/the-semantic-web-not-a-piece-of-cake. [Accessed: 14-Oct-2013].