Hydra: A Vocabulary for Hypermedia-Driven Web APIs

Markus Lanthaler [1]
[1] Institute for Information Systems and Computer Media
Graz University of Technology
Graz, Austria
mail@markus-lanthaler.com

Christian Gütl [1, 2]
[2] School of Information Systems
Curtin University of Technology
Perth, Australia
christian.guetl@iicm.tugraz.at

ABSTRACT
Coping with the ever-increasing amount of data becomes increasingly challenging. To alleviate the information overload put on people, systems are progressively being connected directly to each other. They exchange, analyze, and manipulate humongous amounts of data without any human interaction. Most current solutions, however, do not exploit the whole potential of the architecture of the World Wide Web and completely ignore the possibilities offered by Semantic Web technologies. Based on the experiences gained by implementing and analyzing various RESTful APIs and drawing from the longer history of Semantic Web research we developed Hydra, a small vocabulary to describe Web APIs. It aims to simplify the development of truly RESTful services by leveraging the power of Linked Data. By breaking the descriptions down into small independent fragments, a new breed of interoperable Web APIs using decentralized, reusable, and composable contracts can be realized.

Categories and Subject Descriptors
H.3.4 [**Information Storage and Retrieval**]: Systems and Software – *Semantic Web, World Wide Web (WWW).* H.4.3 [**Information Systems Applications**]: Communications Applications – *Internet.* D.2.11 [**Software**]: Software Architectures – *Service-oriented architecture (SOA)*

Keywords
Web; Web service; Web API; HTTP; REST; Linked Data; RDF; vocabulary; ontology; Hydra

1. INTRODUCTION
Never before in human history has access to information been easier. Knowledge is being shared at an unprecedented scale and the Internet enables frictionless communication across continents in fractions of a second. This abundance of data is increasingly becoming a challenge for humans to cope with. To address this issue, more and more systems are connected directly to each other. They exchange, analyze, and manipulate humongous amounts of data without any human interaction.

LDOW2013, May 14, 2013, Rio de Janeiro, Brazil
Copyright is held by the author/owner(s).

In very large and loosely coupled systems, such as the Internet, the unavoidable heterogeneity has to be embraced and the fact that the data quality and meaning are fuzzy has to be accepted. The combination of Semantic Web technologies and the architectural style of the Web, REST [1], may prove to be a viable path to achieve that. Their combination enables data integration at large scale and solves some of the problems Web developers are continuously struggling with.

Unfortunately, the Linked Data principles in specific and Semantic Web technologies in general, have not yet found widespread adoption in 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. At the same time, the community is just about to truly understand REST. While some of its constraints such as stateless interaction, uniform interface, identification of resources, or manipulation of resources through representations are most of the time respected, others are rarely implemented correctly; regardless of a service claiming to be RESTful or not. Primarily the constraints that demand self-descriptive messages and require the use of hypermedia as the engine of application state are often ignored. While the Linked Data community spends most of their efforts on the accurate description of resources, which could be compared to the self-descriptive messages constraint, the *linking* of data got little attention. Linked Data advocates the use of dereferenceable identifiers but leaves it open how to recognize them as RDF has no built-in notion of hypermedia. It uses URIs solely as identifiers. The best a client can do is to blindly try to interact with these URIs.

To address these issues we developed Hydra, a lightweight vocabulary to describe Web APIs and to augment Linked Data with hypermedia controls. This enables developers to leverage RDF's expressivity with REST's benefits in terms of loose coupling, evolvability, and scalability. It also enables the creation of interoperable Web APIs that are accessible by generic clients.

The reminder of this paper is organized as follows. In section 2, we describe the inherent differences between data models as used in programming languages and vocabularies intended for the Web. After presenting and discussing Hydra in section 3, we give an overview of related work in section 4. Finally, we conclude the paper in section 5.

2. DATA MODELS VS. VOCABULARIES
Current Web APIs typically have a well-defined data model, but unfortunately it is, most of the time, only documented in natural language. This documentation typically defines a number of

JSON objects (classes) with their properties. Properties are marked as optional or required, specify their value space, and typically define whether they are read-only (e.g. server-assigned identifiers), write-only (e.g. password[s), or read-write. Advanced features such as inheritance are rarely used—or at least not exposed to the clients.

The first step to transform a classic JSON-based Web API into a Linked Data API is to uniquely identify classes and properties with Internationalized Resource Identifiers (IRIs). At that point, the first difference between RDF's data model and the class-based programming paradigm most developers are familiar with becomes obvious.

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 [2] and OWL [3], 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 could 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 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. Keeping in mind their different intended use, we designed Hydra [4], a small vocabulary extending RDF Schema by concepts required to describe Web APIs.

3. HYDRA

Even though RDF uses IRIs as identifiers, it has no inherent support for hypermedia. Whether an IRI is intended to be dereferenced or not, depends implicitly on what it represents. FOAF's `homepage` property, e.g., suggests that its values are dereferenceable IRIs. The Linked Data principles postulated by Berners-Lee [5] go a step further and recommend that all IRIs are dereferenceable. The results are large, interconnected graphs of data. Unfortunately, however, these graphs remain largely read-only representations—just as most of the document Web was read-only at the beginning. To change this, a shared vocabulary able to describe affordances beyond simple dereferenceability is needed. Hydra is an attempt to define these missing concepts.

The basic idea behind Hydra is to provide a vocabulary which enables a server to advertise valid state transitions to a client. A client can then use this information to construct HTTP requests which modify the server's state so that a certain desired goal is achieved. 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 adapt to changes more easily.

3.1 Vocabulary

Figure 1 illustrates the Hydra core vocabulary (the figure's intention is to show how Hydra is used rather than its precise definition). At its 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 this does not imply that these IRIs are dereferenceable. In fact, neither RDF itself nor RDF Schema or OWL defines a concept to describe dereferenceable IRIs. Hydra's `Resource` class does just that. It is a subclass of RDF Schema's `Resource` class and can be used to signal a client that an IRI is dereferenceable and can be used to retrieve further information. This allows distinguishing Linked Data 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 [6]. 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 `mappings`. 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

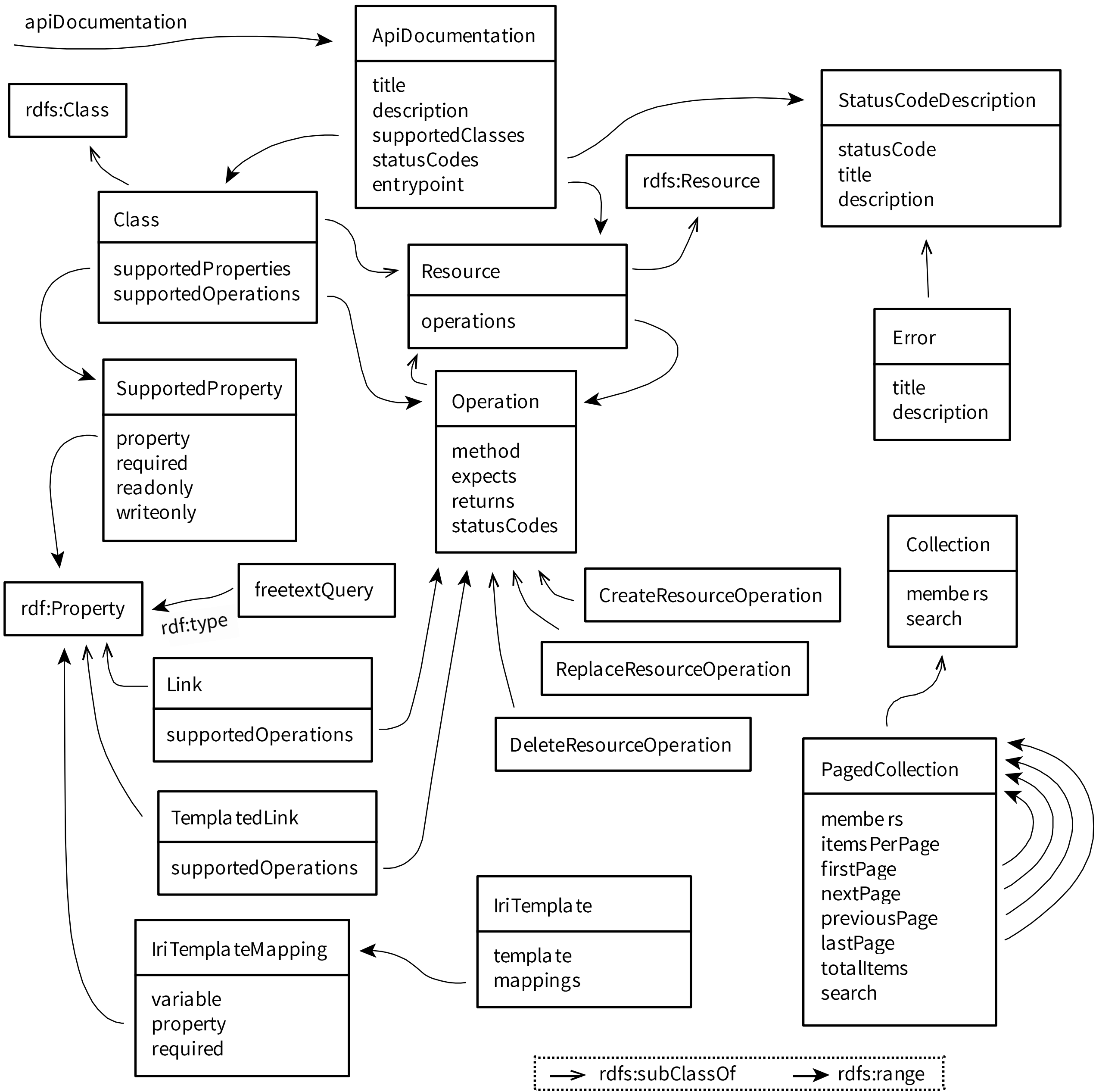

Figure 1. The Hydra core vocabulary

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 the 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. Similarly 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 has, 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 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 its `supportedProperties`. 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, e.g., 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 bootstrapping 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 types of operations, namely `CreateResourceOperation`, `DeleteResourceOperation`, and `ReplaceResourceOperation`. As their name 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 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.

Similarly to the predefined operation classes, Hydra defines classes for collections, another commonly concept in Web APIs. A `Collection` is simply a container pointing to a number of `members`. In Hydra, each of those members is a dereferenceable `Resource`. Since often it is desired to not serve the whole collection at once, but to separate it into pages instead, Hydra also defines a specialized `PagedCollection`. Additionally to its members, 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`, might be used to query such a collection. The currently only predefined property to use in such a mapping is `freetextQuery`. For small Web APIs, these built-in concepts are often enough to build and document the vast majority of the functionality.

3.2 Discussion

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 Representation State Transfer (REST) architectural style [1] with the Linked Data principles [5]. Other formats, such as, e.g., HTML have multiple hypermedia action controls that can be embedded within 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.

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 compared to 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. The reasoning behind it is to allow a developer 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 [7] 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 adopt 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. RDF's use of globally unique identifiers furthermore allows parts of API descriptions to be shared and reused which improves interoperability and reduces costs. Hydra's built-in, 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. This builds the base for the creation of generic clients as we have shown in previous work [8].

Considering Hydra's focus on reusability of concepts between different APIs, the question may arise why Hydra itself does not reuse other existing vocabularies apart from RDF Schema. 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 [3] to specify required properties in the request class used in an operation would simply be too complex. In other cases, the potential reuse from vocabularies is too small to be justifiable. The HTTP vocabulary [9] 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. In cases where related vocabularies exist but are not stable yet, we decided to postpone the decision. An example for this is the Linked Data Platform vocabulary [10] which we will discuss in more detail in the related work section.

4. RELATED WORK

While the semantic description of SOAP-based Web Services has been extensively researched, efforts targeting RESTful Web APIs have been quite limited. There exist a number of approaches for both semantic and syntactic descriptions, but most of them violate one of REST's most fundamental constraints, the use of hypermedia.

The most often discussed approach to describe RESTful services syntactically is the Web Application Description Language (WADL) [11]. It describes a service by exposing a number of URI templates with associated information such as the HTTP method and the required inputs to construct a request. This clearly indicates that hypermedia is not supported. Furthermore, WADL urges the use of specific resource hierarchies which introduces an obvious coupling between the client and the server. Servers should have the complete freedom to control their own namespace.

Swagger [12] follows a similar approach but is, in contrast to WADL which is XML-based, JSON-based. The biggest difference to WADL is that it does not impose any specific resource hierarchy. Other than that, it allows to associate 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 simply tested but it also enables the automatic generation of client libraries. This makes it very similar to Google's API Discovery Service [13] which follows a very similar approach and is mainly used to generate client libraries in different programming languages for Google's numerous Web APIs.

Since all these approaches describe a Web API just syntactically, WSDL 2.0 [14] could in principle be used as well—but typically it is perceived to be too heavy for lightweight Web APIs. Over the years, the research community proposed a number of approaches enriching the syntactic descriptions with machine-readable semantics. SA-REST and hRESTS are probably the best-known representatives of this kind. Since an extensive review of all proposed approaches and ontologies would go beyond the scope of this paper, we refer the interested reader to our previous work [15] for a detailed review.

To the best of our knowledge, there exist only two approaches apart from Hydra combining RDF's data model and expressivity with REST's use of hypermedia, namely SEREDASj and RESTdesc.

SEREDASj [16] is our previous effort to address the problem. It is a simple JSON-based format which focuses on the description of JSON resource representations and their interconnections. JSON responses can be mapped to concepts in a vocabulary and the same mechanism can be used to describe request templates. The key difference to all other approaches is that SEREDASj allows hyperlinks to be extracted from ordinary JSON responses. This allows developers to build hypermedia-driven clients and to transform JSON representations into RDF. The combination of these two features also allows the data exposed by a Web API to be modified using SPARQL queries [17]. Despite some very promising prototypes, we found, from working with different developers, that the complete separation of the description layer from the data itself is suboptimal. Effectively it creates a second layer of interconnected resources on top of the data in JSON documents. This increased the cognitive load and made it difficult for developers to understand documents without at the same time looking at the SEREDASj description document. Furthermore, the syntax, which is based on JSON Schema, was often considered to be too verbose. Ultimately, these reasons lead to the development of Hydra and the use of JSON-LD as the serialization format as described in [8].

RESTdesc [18] is a promising effort aiming at the same goals as Hydra but using, at least at first sight, a radically different approach. It expresses functional descriptions of Web APIs in Notation3 [19], 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 a HTTP request that, when executed, results in the desired post-conditions. 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. It is also not entirely clear how the result of HTTP requests that are part of a requests chain are fed back into the data, i.e., how blank nodes are replaced with IRIs. Since Hydra descriptions can easily be transformed into RESTdesc descriptions, in the long term it would be interesting to investigate the potential offered by the use of reasoning technologies. However, in the short term we believe that a more gradual introduction to Semantic Web technologies is necessary to achieve widespread adoption.

Given the fact that it is being standardized at the W3C and the recent attention it got, it is worth comparing Hydra to the Linked Data Platform (LDP) [10]. While there appears to be quite some overlap between the Hydra and LDP the underlying assumptions and goals are quite different. The Linked Data Platform defines, just as Hydra, 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 Plaform 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 [20]. 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 which creates a new resource. It then has to inspect that resource to verify that all the data has been stored and no properties have been discarded. Given these differences, the Linked Data platform cannot be compared to Hydra in any means. It is questionable whether mainstream Web developers will see enough compelling reasons to adopt such an approach given that the same functionality can be achieved with much simpler, proven approaches such as the Atom Publishing Protocol [20].

5. CONCLUSIONS

The combination of the REST architectural style and the Linked Data principles offer opportunities to advance the Web of machines in a similar way that hypertext did for the human Web. Most building blocks exist already and are in place but they are rarely used together. Hydra tries to fill that gap. It allows data to be enriched with machine-readable affordances which enable interaction. This not only addresses the problem that Linked Data is still mostly read-only, but it also paves the way for a completely new breed of interoperable Web APIs. The fact that it enables the creation of composable contracts means that interaction models of Web APIs can be reused at an unprecedented granularity.

In future work we would like to turn our attention to other aspects which are of interest for most Web APIs, e.g., authentication. Hydra was designed to be a modular vocabulary so that future companion vocabularies can be easily created to extend Hydra's expressivity. Furthermore, we would like to investigate how the availability of machine-processable information can be used in the development process. Since the functionality can be described before it is implemented, testing, e.g., can commence much earlier.

6. REFERENCES

[1] R. T. Fielding, "Architectural Styles and the Design of Network-based Software Architectures," PhD dissertation, Department of Information and Computer Science, University of California, Irvine, 2000.

[2] 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/.

[3] "OWL 2 Web Ontology Language," *W3C Recommendation*. W3C, 2009. [Online]. Available: http://www.w3.org/TR/owl2-overview/.

[4] M. Lanthaler, "Hydra Core Vocabulary Specification," 2013 (work in progress). [Online]. Available: http://www.markus-lanthaler.com/hydra/. [Accessed: 22-Feb-2013].

[5] 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].

[6] 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.

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

[8] M. Lanthaler, "Creating 3rd Generation Web APIs with Hydra," in *Proceedings of the 22nd International World Wide Web Conference (WWW2013)*, 2013.

[9] 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/HTTP-in-RDF10/. [Accessed: 15-May-2011].

[10] S. Speicher and M. Hausenblas, "Linked Data Platform 1.0," *W3C Working Draft*, 2012. [Online]. Available: http://www.w3.org/TR/2012/WD-ldp-20121025/. [Accessed: 05-Nov-2012].

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

[12] "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].

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

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

[15] 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.

[16] 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.

[17] 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.

[18] 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)*, 2012, pp. 33–40.

[19] 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].

[20] 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.