we present a domain-driven approach as a first attempt to streamline and standardize the development of RESTful Web APIs before we conclude the paper in section 5.

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 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, usually Interface Description Languages (IDL) are used to define the application-specific details on top of a, most of the time, standardized communication protocol. This allows automatic code generation for both the client and the server side and leads, in most cases, 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 in turn implies that states and transitions between them have to identifiable which in turn implies the need for (complex) orchestration technologies.

The architecture of the Web breaks fundamentally with 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 in to the realm of specific application domains; this is mostly done by defining specific link relations. An example illustrating this nicely is the Atom Publishing Protocol, which, by defining a number of link relations, extends the otherwise read-only Atom Syndication format with an interface allowing to add new or to manipulate existing 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 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 paired with centrally owned contracts that can be negotiated or extended at runtime is the essence of what Fielding describes as the Representational State Transfer (REST) architectural style [2].

The challenge in designing RESTful systems is to select the most appropriate media type(s) as the core of the application-specific contract which, sometimes, requires creating new, specialized media types. Designers of Web APIs thus have to decide whether to create their own specialized media type, which reduces interoperability; use a generic one such as XML or JSON, 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 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 just the number of entities using different media types in a single service increases. The often seen 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 [3] 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, only two levels 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 `+xml` suffix to the media type identifier to describe that it is based on XML's syntax. Even though this practice has been standardized [4] (and been so for XML for more than a decade) 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 could be done is to parse the representation which, most of the time, is the most trivial aspect.

Looking, e.g., at 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 [5] and xCal [6] 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 will allow the creation of composable contracts, improve the Web as a platform in general, and simplify the development of Web APIs in particular.

2.1 An Alternative Approach

Both xCard and xCal 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 [7]. 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 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 decentralized and a centralized creation and management of XML namespaces; in fact, the IANA maintains a registry specifically for XML namespaces [8].

The question arises why both xCard and xCal have a dedicated media type if the semantics are already signaled using 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 to decide how to process the message. 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 is, in the case of XML in the browser, 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 [9], in 2009 Toby A. Inkster started an effort [10] to register an optional *profile* parameter for XML's and JSON's media types to address this issue. Similarly 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 IRI. 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" [10]. 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 [11] to standardize the link relation *profile* to signal additional semantics associated with a representation using an HTTP Link header [12]. 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, the recent initiative [13] to standardize dedicated media types for JSON-based versions of vCard and iCalendar should be challenged [14]—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 [15] and JSON-LD [16] 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 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 which is that, due to their simple composability, the scope of profiles can be reduced which simplifies their standardization. It is this composability which allows the separation of concerns which 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. Finally, given that profiles have the potential to make the implicit semantics found in current Web APIs explicit, search engines and other automated agents might start to crawl directly the APIs instead of HTML representations of the same data.

3. THE WEB IS A GRAPH

The Web is a distributed hypermedia system at its core. It consists of interlinked resources (mostly in the form of HTML documents)

forming a giant graph of knowledge. HTTP, the Hypertext Transfer Protocol, is used to navigate and manipulate that graph and Uniform Resource Identifiers (URIs) are used to uniquely identify and, in most cases, also directly locate the nodes (resources) the graph consists of. These aspects, the identification of resources and the use of hypermedia to guide interaction, are the base for REST's uniform interface constraint. The two missing architectural constraints in this description are the manipulation of resource through representations and the requirement for messages to be self-descriptive. Given that well-adopted standards for Web APIs supporting them are still missing, it is not surprising that exactly those two constraints are violated most often.

Even though there exist almost 500 media types in the standards tree and another 900 in the vendor and personal trees in the media type registry maintained by IANA [3], only very few support hypermedia. The most often used data formats in Web APIs, XML and JSON, are no exception to this. Given that the Web is intended to be a distributed hypermedia system, this is astonishing—to say the least.

For XML with its namespacing support the problem can be alleviated by reusing elements from a well-known markup vocabulary. The W3C created the XML Linking Language (XLink) [17] to fill this gap, but often Atom's link element is used instead. In JSON, the problem is not as easily solvable as it also lacks support for namespacing. As we discussed in previous work [18], there exist various efforts to add hypermedia support to JSON, but so far no clear winner emerged. The consequence is that most Web APIs invent their own proprietary conventions to represent links; needless to say that almost none of them are tried to be standardized.

JSON-LD [16] is an upcoming W3C standard trying to address these shortcomings of JSON. It provides a generic, JSON-based serialization format for graphs with inherent support for hypermedia. Just as XML, its namespacing feature allows mixing elements from various vocabularies to create mixed documents or extend JSON-LD's functionality. It also enables the creation of self-descriptive messages. Meticulously chosen design decisions were made to keep JSON-LD as simple as possible. The result is that representations in JSON-LD usually look almost exactly the same as they would in plain old JSON. To ensure that clients do not have to depend on the structure and the fieldnames used in a representation, a number of algorithms and a simple API [19] for some basic document transformations haven been developed. This is often necessary since, in contrast to a tree data model for which there only exists one possible serialization, a graph can be serialized in multiple ways. It also means that representations can be optimized for JSON-only clients while at the same time providing much more flexibility for clients understanding JSON-LD. The fact that all properties and entities are uniquely identified by IRIs greatly simplifies some of the most difficult tasks when dealing with data at Internet-scale, most notably data integration.

4. PUTTING EVERYTHING TOGETHER: DOMAIN-DRIVEN WEB SERVICE DESIGN

Designing Web APIs that take advantage of REST's benefits in terms of scalability, maintainability, and evolvability is still more an art than a science. Nevertheless, we are convinced that it is possible to at least distill a number of guidelines and procedures to assist developers in the complex design process.

Based on the experiences gained by implementing various RESTful Linked Data-based APIs and drawing from a longer history of Semantic Web and hypermedia research, we will present an alternative, domain-driven approach as a first attempt to address this issue in the next sections. We will give some suggestion for concrete technologies and discuss their consequences.

4.1 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. A commonly agreed 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 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 or the Web Ontology Language (OWL) 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 defining the data models and to work with the data itself. Another advantage of an RDF-based system is the dramatically simplified reuse of domain models—either as a whole or of parts thereof. Such reuse not only dramatically 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 Internet-scale. Furthermore, its schemalessness ensures the required agility in today's fast-moving world.

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 earlier. Both JSON-only and JSON-LD aware clients can then seamlessly work with the same data representations.

4.2 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 [20] or, more formally speaking, the behavioral model needs to be defined as well. Despite significant research and development effort, the problem of describing

RESTful Web APIs in a machine-processable way is still to be solved. Consequently, most Web APIs are solely documented in the form of human-targeting, natural-language documents. Based on the experience gained from our previous efforts to address this problem, we designed Hydra [21], a lightweight vocabulary to capture and document the behavioral model of hypermedia-driven Web APIs in a machine-processable way.

Simply speaking, Hydra defines a number of concepts, such as collections, commonly used in Web APIs and provides a vocabulary that can be used to describe the domain application protocol of services. Operations stand at the core of a Hydra-powered description and allow the semantics of specific HTTP operations to be documented. Specific operations can either be bound to entity classes (types) or link relations (properties) or be used directly within data representations. This allows agents to discover the affordances supported by the various entities in a Web API. As result, it becomes possible to either build machine agents that navigate Web APIs completely autonomously or to create programming libraries with a much higher level of abstraction simplifying developers' lives. Since 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* [22]. A complete description of Hydra would go beyond the scope of this paper and we would thus like to refer the interested reader to [21] and [23] for more information.

The combination of a formal data model as described in the previous section and a holistic documentation of the behavioral model based on Hydra enables the creation of declarative contracts capturing all aspects of a Web APIs. It is worthy 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. In a prototype we presented in previous work [24], the mapping took place in the form of code annotations. Entities were augmented with instructions on how to serialize them to JSON-LD. Furthermore, these annotations built the base to automatically generate the code for simple controllers implementing the basic CRUD functionality. Automatic code generation always imposes the risk of either introducing unnecessary coupling or leaking implementations details—this is especially risky if the contract is owned by the server. The presented approach mitigates this problem by allowing the problem domain to be decomposed into smaller sub-problems that are significantly easier to standardize, shifting the coupling to a central standard (or a combination of multiple standards in the form of a profile).

4.3 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 exist no off-the-shelf tools assisting developers in testing their API. The situation looks similar for 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 dramatically 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 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 all other data, 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. Hydra's core vocabulary [21] has no built-in support for such sample responses yet, but it is planned to create an extension which addresses it.

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 and the generic programming library HydraClient are available freely as open source software [21].

4.4 Documenting Services

It takes time to convince developers to use such new models to build their systems. Thus, everything that allows an iterative introduction of these techniques helps to foster adoption. It will, at least for the foreseeable future, still be 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 directly used to create large parts of human-targeting documentations automatically. By utilizing technologies such as HTML and RDFa 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 matter for developers: the rationales behind design decisions, the assumptions made, the mental models, and the overall goals of a Web API.

Recent advances in artificial intelligence suggest that in the near future it will even become possible to automate the creation of these natural-language descriptions. Narrative Science [25], e.g., has already built a commercial product which is able to generate natural-language stories and reports for a wide variety of applications out of highly structure data. Extending it to API documentations, which tend to be very similar, might just require a small step.

5. CONCLUSIONS

In this position paper we presented an alternative, domain-driven approach to design RESTful Web APIs. We discussed a number of crucial design decision and showed how it is possible to create Web APIs where 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 of 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. This hopefully increases the quality of system build using such an approach.

In future work we would like to develop tools assisting developers in the various development stages. The generic API console and client library presented in previous work was a first step into that direction. We would also like to further experiment with the implications of composable contracts based on profiles. Finally, we would like to investigate to which degree human-readable documentations are still needed, given that most parts of an API can be standardized and remaining variability can be captured in machine-readable descriptions. In the spirit of the principle of least power, we do believe that in most cases it should be possible to ship APIs without any additional human-targeting documentation—Web sites typically do not need to be explained either.

6. REFERENCES

[1] 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/.

[2] 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.

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

[4] 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.

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

[6] 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.

[7] 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/.

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

[9] 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.

[10] 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].

[11] 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.

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

[13] 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].

[14] M. Lanthaler, "Re: [apps-discuss] iCalendar and vCard in JSON: WG draft charter," 2013. [Online]. Available: http://www.ietf.org/mail-archive/web/apps-discuss/current/msg08873.html. [Accessed: 24-Feb-2013].

[15] 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].

[16] M. Sporny, G. Kellogg, and M. Lanthaler, "JSON-LD 1.0 – A JSON-based Serialization for Linked Data," *W3C Editor's Draft*, 2013. [Online]. Available: http://www.w3.org/TR/json-ld/. [Accessed: 29-Mar-2013].

[17] 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/.

[18] 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) at the 21st International World Wide Web Conference (WWW2012)*, 2012, pp. 25-32.

[19] M. Lanthaler, G. Kellogg, and M. Sporny, "JSON-LD 1.0 Processing Algorithms and API," *W3C Editor's Draft*, 2013. [Online]. Available: http://www.w3.org/TR/json-ld-api/. [Accessed: 22-Feb-2013].

[20] 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.

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

[22] 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].

[23] 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.

[24] 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, 2013 (in press).

[25] Narrative Science. [Online] Available: http://narrativescience.com/