Hindawi Publishing Corporation
Advances in Multimedia
Volume 2012, Article ID 586542, 14 pages
doi:10.1155/2012/586542

Research Article

Seamless Integration of RESTful Services into the Web of Data

Markus Lanthaler[1] and Christian Gütl[1, 2]

[1] *Institute for Information Systems and Computer Media, Graz University of Technology, 8010 Graz, Austria*
[2] *School of Information Systems, Curtin University of Technology, Perth WA 6102, Australia*

Correspondence should be addressed to Markus Lanthaler, markus.lanthaler@student.tugraz.at

Received 4 November 2011; Accepted 15 January 2012

Academic Editor: Nabil Tabbane

We live in an era of ever-increasing abundance of data. To cope with the information overload we suffer from every single day, more sophisticated methods are required to access, manipulate, and analyze these humongous amounts of data. By embracing the heterogeneity, which is unavoidable at such a scale, and accepting the fact that the data quality and meaning are fuzzy, more adaptable, flexible, and extensible systems can be built. RESTful services combined with Semantic Web technologies could prove to be a viable path to achieve that. Their combination allows data integration on an unprecedented scale and solves some of the problems Web developers are continuously struggling with. This paper introduces a novel approach to create machine-readable descriptions for RESTful services as a first step towards this ambitious goal. It also shows how these descriptions along with an algorithm to translate SPARQL queries to HTTP requests can be used to integrate RESTful services into a global read-write Web of Data.

1. Introduction

We live in an era where exabytes of data are produced every single year; never before in human history had we to deal with such an abundance of information. To cope with this information overload, more sophisticated methods are required to access, manipulate, and analyze these humongous amounts of data. Service-oriented architectures (SOAs) built on Web services were a first attempt to address this issue, but the utopian promise of uniform service interface standards, metadata, and universal service registries, in the form of SOAP, WSDL, and UDDI has proven elusive. This and other centralized, registry-based approaches were overwhelmed by the Web's rate of growth and the lack of a universally accepted classification scheme. In consequence, the usage of SOAP-based services is mainly limited to company-internal systems and to the integration of legacy systems. In practice, however, such a clear and crisp definition of data is rare. Today's systems integrate data from many sources. The data quality and meaning are fuzzy and the schema, if present, are likely to vary across the different sources. In very large and loosely coupled systems, such as the Internet, the gained adaptability, flexibility, and extensibility, in a transition away from strict and formal typing to simple name/value pairs or triples, outweighs the resulting loss off "correctness."

Thus, it is not surprising that RESTful services, and there especially the ones using the lightweight JavaScript Object Notation (JSON) [1] as the serialization format, are increasingly popular. According to ProgrammableWeb, 74% of the Web services are now RESTful and 45% of them use JSON as the data format [2], but, in spite of their growing adoption, RESTful services still suffer from some serious shortcomings.

The major problem is that, for RESTful services or *Web APIs,* a recently emerged term to distinguish them from their traditional SOAP-based counterparts, no agreed machine-readable description format exists. All the required information of how to invoke them and how to interpret the various resource representations is communicated out-of-band by human-readable documentations. Since machines have huge problems to understand such documentations, machine-to-machine communication is often based on static knowledge resulting in tightly coupled system. The challenge is thus to bring some of the human Web's adaptivity to the Web of machines to allow the building of loosely coupled, reliable, and scalable systems.

Semantic annotations could prove to be a viable path to achieve that, but, while the vision of a Semantic Web has been around for more than fifteen years, it still has a long way to go before mainstream adoption will be achieved. One of the reasons for that is, in our opinion, the fear of average Web developers to use Semantic Web technologies. They are often overwhelmed by the (perceived) complexity or think they have to be AI experts to make use of the Semantic Web. Others are still waiting for a killer application making it a classical chicken-and-egg problem. A common perception is also that the Semantic Web is a disruptive technology which makes it a showstopper for enterprises needing to evolve their systems and build upon existing infrastructure investments. Obviously, some developers are also just reluctant to use new technologies. Nevertheless, we think most Web developers fear to use Semantic Web technologies for some reason or another; a phenomenon we denoted as *Semaphobia* [3]. To help developers get past this fear, and to show them that they have nothing to fear but fear itself, clear incentives along with simple specifications and guidelines are necessary. Wherever possible, upgrade paths for existing systems should be provided to build upon existing investments.

That is exactly what made the Linked Data movement so successful. It simplified the technology stack and provided clear incentives for annotating data. In consequence, it is not surprising that after being ignored by the majority of the Web developers for a long time, lightweight semantic annotations finally start to gain acceptance across the community. Facebook's Open Graph protocol, for example, was implemented in over 50,000 Web sites within the first week of its launch [4] and the current estimates are that roughly 10% of all Web pages are annotated with it.

It would just seem consequent to combine the strengths of both, REST and the Linked Data principles, but in practice they still remain largely separated. Instead of providing access to Linked Data via a RESTful service interface, current efforts deploy centralistic SPARQL endpoints or simply upload static dumps of RDF data. This also means that most current Semantic Web projects just provide read-only interfaces to the underlying data. This clearly inhibits networking effects and engagement of the crowd.

To address these issues, we developed a novel approach to semantically describe RESTful data services which allows their seamless integration into a Web of Data. We put a strong emphasis on simplicity and on not requiring any changes on the Web service itself. This should lower the entry barrier for future Web developers and provide a viable upgrade path for existing infrastructure. At the same time, the approach is extensible and flexible enough to be applicable in a wide application domain.

The reminder of the paper is organized as follows. In Section 2, we give an overview of related work. Then, in Section 3, we present the requirements and the design of SEREDASj, our approach to semantically describe RESTful services. Section 4 shows how SEREDASj can be used to integrate different RESTful services into the Web of Data, and finally, Section 5 concludes the paper and gives an overview of future work.

2. Related Work

In contrast to traditional SOAP-based services, which have agreed standards in the form of WSDL and SAWSDL [5] to be described, both, syntactically and semantically, no standards exist for RESTful services. In consequence, RESTful services are almost exclusively described by human-readable documentations describing the URLs and the data expected as input and output. There have been made many proposals to solve this issue; SA-REST [6], hRESTS [7], and WADL [8] are probably the best-known ones.

The Web Application Description Language's approach (WADL) [8] is closely related to WSDL in that a developer creates a monolithic XML file containing all the information about the service's interface. Given that WADL was specifically designed for describing RESTful services (or HTTP-based Web applications as they are called in WADL's specification), it models the resources provided by the service and the relationships between them. 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, just 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 between the client and the server. Servers should have the complete freedom to control their own namespace.

hRESTS (HTML for RESTful Services) [7] follows a quite different approach as it tries to exploit the fact that almost all RESTful services already have a textual documentation in the form of Web pages. hRESTS' idea is hence to enrich those, mostly already existent, human-readable documentations with so-called microformats [9] to make them machine-processable. 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 (it is a common practice to split service documentations into different HTML documents to make them more digestible). Each service is described by a number of operations, that is, actions a client can perform on that service, with the corresponding URI, HTTP method, the 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 used data schemas. Apart from a potential label, hRESTS does not provide any support for further machine-readable information about the inputs and outputs. Extensions like SA-REST [6] and MicroWSMO [10] address this issue.

MicroWSMO is an attempt to adapt the SAWSDL approach for the semantic description of RESTful services. It uses, just as hRESTS, on which it relies, microformats for adding semantic annotations to the HTML service documentation. Similar to SAWSDL, MicroWSMO has three types of annotations: (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 in which SAWSDL supports the annotation of Web services described by WSDL.

Another approach for the semantic description of RESTful services is the before-mentioned SA-REST [6]. It relies on RDFa for marking service properties in an existing HTML service description, similar to hRESTS with MicroWSMO. As a matter of fact, it was the first approach reusing the already existing HTML service documentation to create machine-processable descriptions of RESTful services. The main differences between the two approaches are indeed not the underlying principles but rather the implementation technique. SA-REST offers the following service elements: (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 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.

In principle, a RESTful service could even be described by using WSDL 2.0 [11] with SAWSDL [5] and an ontology like OWL-S or WSMO-Lite. OWL-S (Web Ontology Language for Web Services) [12] 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: (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; (3) the *Service Grounding*, which provides the needed 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, taken together, provide enough information for an agent to make use of a service, once found [12]. 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. Besides that, an OWL-S process cannot contain any number of completely unrelated operations [13, 14], in contrast to WSDL.

WSMO-Lite [15] is another ontology to fill SAWSDL's annotations with concrete service semantics. SAWSDL itself does not specify a language for representing the semantic models but just defines how to add semantic annotations to various parts of a WSDL document. WSMO-Lite allows bottom-up modeling of services and adopts, as the name suggests, the WSMO [16] model and makes its semantics lighter. 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; (4) the *Nonfunctional Descriptions*, which define nonfunctional properties such as quality of service or price. A major advantage of WSMO-Lite is that it is not bound to a particular service description format, for example, WSDL. Consequently, it can be used to integrate approaches like, for example, hRESTS (in conjunction with MicroWSMO) with traditional WSDL-based service descriptions. Therefore, tasks such as discovery, composition, and data mediation could be performed completely independent from the underlying Web service technology.

Even though at a first glance all the above-described ideas seem to be fundamentally different from WSDL, their underlying model is still closely related to WSDL's structure. In consequence, all presented approaches heavily rely on RPC's (Remote Procedure Call) flawed [17] operation-based model ignoring the fundamental architectural properties of REST. Instead of describing the resource representations, and thus allowing a client to understand them, they adhere to the RPC-like model of describing the inputs and outputs as well as the supported operations which result in tight coupling. The obvious consequence is that these approaches do not align well with clear RESTful service design.

One of the approaches avoiding the RPC-orientation, and thus more suitable for RESTful services, is ReLL [18], the Resource Linking Language. It is a language to describe RESTful services with emphasis on the hypermedia characteristics of the REST model. This allows, for example, a crawler to automatically retrieve the data exposed by Web APIs. One of the aims of ReLL is indeed to transform crawled data to RDF in order to harvest those already existing Web resources and to integrate them into the Semantic Web. Nevertheless, ReLL does not support semantic annotations but relies on XSLT for the transformation to RDF. This clearly limits ReLL's expressivity as it is not able to describe the resource representations semantically.

There are many other approaches that allow, just as ReLL, to transform data exposed by Web APIs to RDF. In fact, large parts of the current Web of Data are generated from non-RDF databases by tools such as D2R [19] or Triplify [20] but one of the limitations of the current Semantic Web is that it usually just provides read-only interfaces to the underlying data. So, while several Semantic Web browsers, such as Tabulator [21], Oink [22], or Disco [23], have been developed to display RDF data, the challenge of how to edit, extend, or annotate this data has so far been left largely unaddressed. There exist a few single-graph editors including RDFAuthor [24] and ISAViz [25] but, to our best knowledge, Tabulator Redux [26] is the only editor that allows the editing of graphs derived from multiple sources.

To mitigate this situation, the *pushback project* [27] was initiated in 2009 (it is not clear whether this project is still active) to develop a method to write data back from RDF graphs to non-RDF data sources such as Web APIs. The approach chosen by the pushback project was to extend the RDF wrappers, which transform non-RDF data from Web APIs to RDF data, to additionally support write operations. This is achieved by a process called *fusion* that automatically annotates an existing HTML form with RDFa. The resulting *RDForm* then reports the changed data as RDF back to the pushback controller which in turn relays the changes to the RDF write wrapper that then eventually translates them into an HTTP request understandable to the Web API. One of the major challenges is to create the read-write wrappers as there are, as explained before, no agreed standards for describing RESTful services; neither syntactically nor semantically. Exposing these Web APIs as read-write Linked Data is, therefore, more an art than a science.

3. Semantic Description of RESTful Services

A machine-readable documentation of a service's interface and the data it exposes is a first step towards their (semi-) automatic integration. In this section, we first discuss the requirements for a semantic description language for RESTful services and then present SEREDASj, a novel approach to address this ambitious challenge.

3.1. Requirements. Analyzing the related work and taking into account our experience in creating RESTful services and integrating them into mashups, we derived a set of core requirements for a semantic description language.

Since the description language is targeted towards RESTful services, it clearly has to adhere to REST's architectural constraints [28] 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 (whether the representation is in the same format as the raw source, or is derived from the source, remains hidden behind the interface) and has to have an identifier. All resource representations should be self-descriptive, that is, they are somehow labeled with their type which specifies how they are to be interpreted. 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.

To be widely accepted, the approach has to be based on core Web standards. That means it should use Uniform Resource Identifiers (URIs) for identifying resources, the Hypertext Transfer Protocol (HTTP) for accessing and modifying resource representations, and the Resource Description Framework (RDF) as the unified data model for describing resources. To ease tasks such as data integration, a uniform interface to access heterogeneous data sources in a uniform and intuitive way, has to be provided as well. This, in turn, will lead to reusability and flexibility which are important aspects for the adoption of such a new approach. By having semantically annotated data, a developer could also be supported in the data integration and mediation process which is not only important in enterprise scenarios but also for the creation of mashups. All too often the required data mediation code is longer than the actual business logic. By having semantically annotated data, it is possible to integrate it (semi-) automatically with other data sources.

While all of these constraints are important when designing a RESTful service, the most important aspects for a semantic description language are how the resources can be accessed, how they are represented, and how they are interlinked. The description language 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 service into the Semantic Web, the description language should also provide means to transform the representations in RDF triples. In order to be able to evolve systems and build upon existing infrastructure, an important requirement is that no (or just minimal) changes on the existing system are required; this implies a requirement to support partial descriptions. Last but not least, the approach should be as simple as possible to lower the entry barrier for developers and to foster its adoption.

3.2. SEREDASj. Considering the requirements described in the previous section, we designed SEREDASj a language to describe *SEmantic REstful DAta Services*. The "j" at the end should highlight that we based the approach on JSON. JSON's popularity in Web APIs is not the only reason for that.

The inherent impedance mismatch (the so-called O/X impedance mismatch) between XML, which is used in traditional SOAP-based Web services, and object-oriented programming constructs often results in severe interoperability problems. The fundamental problem is that the XML Schema language (XSD) has a number of type system constructs which simply do not exist in commonly used object-oriented programming languages such as, for example, Java. This leads in consequence 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.

In most use cases addressed by Web services, all a developer wants to do is to interchange data—and here we are distinguishing between data interchange and document interchange. JSON was specifically designed for this: it is a lightweight, language-independent data-interchange format which is easy to parse and easy to generate. Furthermore, it

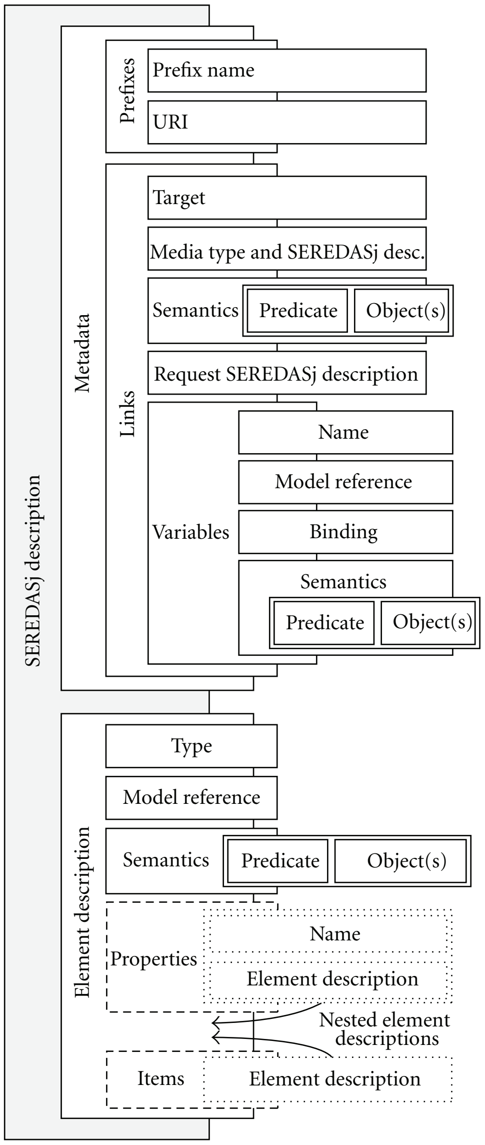

Figure 1: The SEREDASj description model.

is much easier for developers to understand and use. JSON's whole specification [1] consists of ten pages (with the actual content being a mere four pages) compared to XML where the XML Core Working group alone [29] lists *XML*, *XML Namespaces*, *XML Inclusions*, *XML Information Set*, *xml:id*, *XML Base*, and *Associating Stylesheets with XML* as standards; not even including *XML Schema Part 1* and *XML Schema Part 2*.

Summarized, JSON's simplicity, ease of integration, and raising adoption across the Web community [2] made it the first choice for our description language, but we would like to highlight that the principles of our approach are applicable to any serialization format.

To describe a RESTful service, SEREDASj specifies, similar to schemas, the syntactic structure of a specific JSON representation. Additionally, it allows to reference JSON elements to concepts in a vocabulary or ontology and to further describe the element itself by semantic annotations. Figure 1 depicts the structure of an SEREDASj description.

A description consists of metadata and a description of the structure of the JSON instance data representations it describes. The metadata contains information about the hyperlinks related to the instance data and prefix definitions to abbreviate long URIs in the semantic annotations to CURIEs [30]. The link descriptions contain all the necessary information for a client to retrieve and manipulate instance data. Additionally to the link's target, its media type and the target's SEREDASj description, link descriptions can contain the needed SEREDASj request description to create requests and semantic annotations to describe the link, for example, its relation to the current representation. The link's target is expressed by link templates where the associated variables can be bound to an element in the instance data and/or linked to a conceptual model, for example, a class or property in an ontology. The link template's variables can be further described by generic semantic annotations in the form of predicate-object pairs. The links' SEREDASj request description allows a client to construct the request bodies used in `POST` or `PUT` operations to create or update resources.

The description of the structure of instance representations (denoted as element description in Figure 1) defines the JSON data type(s) as well as links to conceptual models. Furthermore, it may contain semantic annotations to describe an element further and, if the element represents either a JSON object or array, a description of its properties, respectively, items in term of, again, an element description. The structure of the JSON instance arises out of nested element descriptions. To allow reuse, the type of an element description can be set to the URI of another model definition or another part within the current model definition. To address different parts of a model, a slash-delimited fragment resolution is used. In Listing 1, for instance, `event.json#properties/enddate` refers to the end date property defined by the SEREDASj document `event.json`.

In order to better illustrate the approach, a simple example of a JSON representation and its corresponding SEREDASj description are given in Listing 1. The example is a representation of an event and its performers from an imaginary event site's API. 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. SEREDASj solves those problems by describing all the important aspects of such a representation. In consequence, it is not only possible to extract the hyperlinks, but also to create a human-readable documentation of the data format (as shown in [3]) and to translate the JSON representation to an RDF representation.

The SEREDASj description in Listing 1 contains two link definitions. The first one specifies the link to the performers' representations via their ID. It uses a URI template whose variable is bound to `#properties/performers/id`. This link definition also shows how further semantic annotations can be used; this is described in detail in Section 4.1. The second link specifies a search interface and is thus not

Instance Data
http://example.com/event/e48909

```
{
  "id": "e48909",
  "name": "Dick Clark's New Year's Rockin' Eve",
  "startdate": "2011-12-31",
  "enddate": "2012-01-01",
  "performers": [
    { "id": "p84098", "name": "Lady Gaga",
        "birthdate": "1986-03-28" }
  ]
}
```

SEREDASj Description
http://example.com/models/event.json

```
{
  "meta": {
    "prefixes": {
      "owl": "http://www.w3.org/2002/07/owl#",
      "so": "http://schema.org/",
      "ex": "http://example.com/onto#",
      "iana": "http://www.iana.org/link-relations/"
    },
    "links": {
      "/person/{id}": {
        "mediaType": "application/json",
        "seredasjDescription": "person.json",
        "semantics": {
          "owl:sameAs": "<#properties/performers>"
        },
        "variables": {
          "id": {
            "binding": "#properties/performers/id",
            "model": "[ex:id]"
          }
        },
        "requestDescription": "person-createupdate.json"
      },
      "/events/search{?query}": {
        "mediaType": "application/json",
        "seredasjDescription": "eventlist.json",
        "semantics": {
          "[iana:relation]": "[iana:search]" },
        "variables": {
          "query": { "model": "[so:name]" }
        }
      }
    }
  },
  "type": "object",
  "model": "[so:Event]",
  "properties": {
    "id": {
      "type": "string", "model": "[ex:id]" },
    "name": {
      "type": "string", "model": "[so:name]" },
    "startdate": {
      "type": "string", "model": "[so:startDate]" },
    "enddate": {
      "type": "string", "model": "[so:endDate]" },
```

Listing 1: Continued.

```
        "performers": {
          "type": "array",
          "model": "[so:performers]",
          "items": {
            "type": "object", "model": "[so:Person]",
            "properties": {
              "id": {
                "type": "string", "model": "[ex:id]" },
              "name": {
                "type": "string", "model": "[so:name]" },
              "birthdate": {
                "type": "string", "model": "[so:birthDate]" }
            }
          }
        }
      }
    }
```

Listing 1: An exemplary JSON representation and its corresponding SEREDASj description.

bound to any element in the instance data; instead, the variable's model reference is specified. Again, this link is semantically annotated so that an agent will know that this link specifies a search interface. These semantic annotations allow developers to implement smarter clients understanding the relationships of resources and thus following REST's hypermedia as the engine of application state constraint.

The following description of the representation's structure basically maps the structure to the ontology defined by schema.org [31]. The mapping strategy is similar to the table-to-class, column-to-predicate strategy of current relational database-to-RDF approaches [32]; JSON objects are mapped to classes, all the rest to predicates. By reusing schema.org's ontology wherever possible, the developer is able to exploit the already available human-readable descriptions for the various elements and generate completely automatically a human-readable documentation.

SEREDASj descriptions do not have to be complete, that is, they do not need to describe every element in all details. If an unknown element is encountered in an instance representation, it is simply ignored. This way, SEREDASj allows forward compatibility as well as extensibility and diminishes the coupling. In this context, 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 ease the mapping to the client's data model.

4. Seamless Integration of RESTful Services into a Web of Data

Currently mashup developers have to deal with a plethora of heterogeneous data formats and service interfaces for which little to no tooling support is available. RDF, the preferred data format of the Semantic Web, is one attempt to build a universal applicable data format to ease data integration, but, unfortunately, current Semantic Web applications mostly provide just read-only interfaces to their underlying data. We believe it should be feasible to standardize and streamline the mashup development process by combining technologies from, both, the world of Web APIs and the Semantic Web. This would, in the first place, result in higher productivity which could subsequently lead to a plethora of new applications. Potentially it could also foster the creation of mashup editors at higher levels of abstraction which could, hopefully, even allow non-technical experts to create mashups fulfilling their situational needs.

Based on SEREDASj which we introduced in the previous section, we would like to propose a new reference model for integrating traditional Web service interfaces into a global read-write graph of data. Figure 2 shows the architecture of our approach.

We broadly distinguish between an application-specific (at the top) and an application-independent layer (at the bottom). The application-independent layer at the bottom is used as a generic data access layer. It separates the application and presentation logic from the common need to manage and manipulate data from a plethora of different data sources. This separation of concerns should result in better reusability and increased development productivity.

Data from JSON-based Web services described by SEREDASj are translated into RDF data and stored along with data from native RDF sources such as SPARQL endpoints, static RDF dumps, or RDF embedded in HTML documents in a local triple store. This unification of the data format is the first step for the integration of these heterogeneous data sources. We use RDF because it reflects the way data is stored and interlinked on the Web, namely, in the form of a graph. The fact that it is schema-free and based on triples makes it the lowest common denominator for heterogeneous data sources, flexible, and easily evolvable. In addition to acting as a data integration layer, this local triple store is also used for caching the

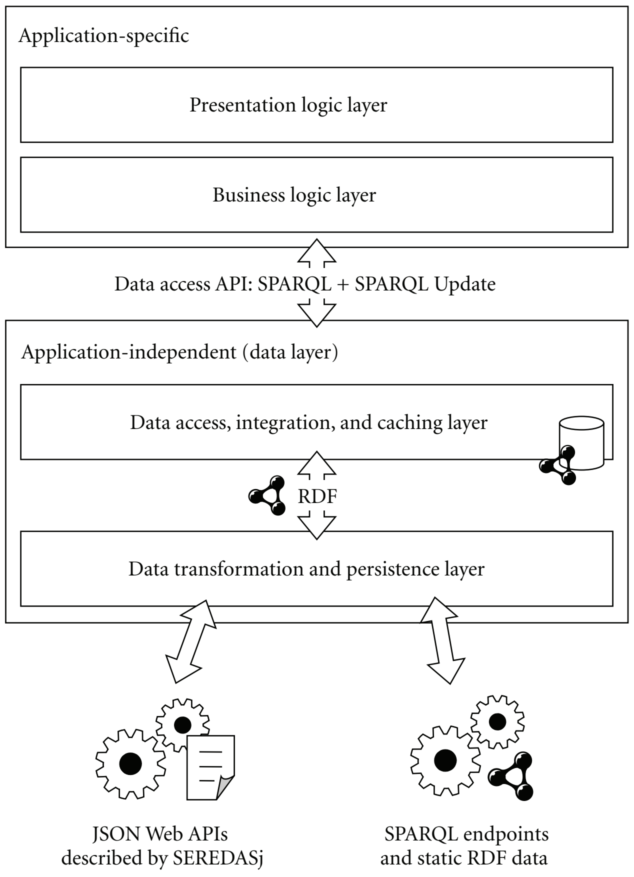
Application-specific
Presentation logic layer
Business logic layer
Data access API: SPARQL + SPARQL Update
Application-independent (data layer)
Data access, integration, and caching layer
RDF
Data transformation and persistence layer
JSON Web APIs described by SEREDASj
SPARQL endpoints and static RDF data

Figure 2: A reference model for integrating Web APIs into the Web of Data.

data which is a fundamental requirement in networked applications. Furthermore, centralized processing is much more efficient than federated queries and the like. Just look at, for example, Google's centralized processing compared to federated database queries and please keep in mind that we are not arguing against achievable speed increases by parallelization.

All data modifications are passed through the data access and persistence layer and will eventually be transferred back to the originating data source. The interface connecting the data access layer and the business logic layer has to be aware of which data can be changed and which cannot since some data sources or part of the data representations might be read-only. Depending on the scenario, a developer might choose to include a storage service (either a triple store or a traditional Web API) which allows storing changes even to immutable data. It is then the responsibility of the data integration layer to "replace" or "overwrite" this read-only data with its superseding data. Keeping track of the data's provenance is thus a very important feature.

In order to decouple the application-specific layer from the application-independent data layer, the interface between them has to be standardized. There exist already a standard and a working draft for that, namely, SPARQL [33] and SPARQL Update [34]. We reuse them in order to build our approach upon existing work. Of course, an application developer is free to add another layer of abstraction on top of that—similar to the common practice of using an O/R mapper (object-relational mapper) to access SQL databases.

While this three-tier architecture is well known and widely used in application development, to our best knowledge it has not been used for integrating Web services into the Semantic Web. Furthermore, this integration approach has not been used to generalize the interface of Web services. Developers are still struggling with highly diverse Web service interfaces.

4.1. Data Format Harmonization. Translating SEREDASj described JSON representations to RDF triples, the first step for integrating them into the Linked Data Cloud, is a straightforward process. The translation starts at the root of the JSON representation and considers all model references of JSON objects and tuple-typed arrays to be RDF classes, while all the other elements' model references are considered to be RDF predicates where the value of that element will be taken as object. If a representation contains nested objects, just as the example in Listing 1, a slash-delimited URI fragment is used to identify the nested object. Semantic annotations in the form of the `semantics` property, as the one shown in the performer's link in Listing 1, 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.

The automatic translation of the example from Listing 1 to RDF is shown in Listing 3. The event and its performers are nicely mapped to schema.org ontology. For every array item, a new object URI is created by using a slash-delimited URI fragment. Eventually, those URIs are mapped to the performer's "real" URI by the link's semantic annotation. Please note that the query link is not included in the RDF representation. The reason for this is that the query variable is not bound to any instance element and thus its value is unknown. In consequence, the translator is unable to construct the URI.

4.2. Integration with Other Data Sources. As explained in the previous section, the conversion to RDF is a first step towards integration of data from different sources. To be fully integrated, the data from all sources eventually has to use the same semantic annotations, that is, the same vocabulary and the same identifiers. Traditionally, this homogenization has been done in an imperative way by writing data mediation code. The Semantic Web technology stack on the other hand embraces the inevitable heterogeneity and provides means to address this issue in a declarative way by creating new knowledge in the form of, for example, schema or identifier mappings. By studying the contents of data and the relationships between different data items, it is sometimes possible to infer (semi-) automatically that two seemingly different items are really the same.

It is straightforward to integrate the data from our example in Listing 3 with data about Lady Gaga stored in, for example, DBpedia (a project aiming to extract structured content from the information contained in Wikipedia). All we have to do is to map some of schema.org concepts and

```
PREFIX foaf: <http://xmlns.com/foaf/0.1/>
PREFIX xsd: <http://www.w3.org/2001/XMLSchema#>
PREFIX dbpprop: <http://dbpedia.org/property/>

SELECT ?s
WHERE {
  ?s foaf:name ?name;
    dbpprop:birthDate ?dob.
  FILTER(str(?name) = "Lady Gaga").
  FILTER(str(?dob) = "1986-03-28") }
```

Listing 2: SPARQL query to find Lady Gaga's identifier in DBpedia.

```
@base <http://example.com/event/e48909>.

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

@prefix owl: <http://www.w3.org/2002/07/owl#>.
@prefix so: <http://schema.org/>.
@prefix ex: <http://example.com/onto#>.

<#> rdf:type so:Event.
<#> ex:id "e48909".
<#> so:name "Dick Clark's New Year's Rockin' Eve".
<#> so:startDate "2011-12-31".
<#> so:endDate "2012-01-01".
<#> so:performers <#performers/0>.

<#performers/0> rdf:type so:Person.
<#performers/0> ex:id "p84098".
<#performers/0> so:name "Lady Gaga".
<#performers/0> so:birthDate "1986-03-28".

<http://example.com/person/p84098> owl:sameAs
  <#performers/0>.
```

Listing 3: The example in Listing 1 translated to RDF.

our local identifier to concepts and Lady Gaga's identifier in DBpedia. Schema mappings are already provided by DBpedia (http://mappings.dbpedia.org/) so all we have to do is to find DBpedia's identifier and map it to our local identifier. An inference engine could do this easily by running the query shown in Listing 2 at DBpedia's SPARQL endpoint. The result is the URI we are looking for: `http://dbpedia.org/resource/Lady_Gaga`. After mapping that URI to our local identifier by using OWL's `sameAs` concept, we can easily query all the data about Lady Gaga from DBpedia as it would be part of our Web service; the data layer in Figure 2 is responsible to take care of all the necessary details.

4.3. Storing Changes Back to the Source. Just as DBpedia, a big part of the current Semantic Web consists of data transformed from Web APIs or relational databases to RDF or by data extracted from Web sites. In consequence, the vast majority of the current Semantic Web is just read-only, that is, changes cannot be stored back to the original source. Thus, in this section, we will show how SEREDASj allows data to be updated and transferred back to the originating

Web service (obviously we are not able to update static Web pages).

For the following description, we assume that all data of interest and the resulting Web of interlinked SEREDASj descriptions have already been retrieved (whether this means crawled or queried specifically is irrelevant for this work). The objective is then to update the harvested data or to add new data by using SPARQL Update.

SPARQL Update manipulates data by either adding or removing triples from a graph. The `INSERT DATA` and `DELETE DATA` operations add, respectively, 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 to SQL and Web APIs and complicates the translation between an SPARQL query and the equivalent HTTP requests to interact with a Web service.

4.4. Translating Insert Data and Delete Data. In regard to a Web service, an `INSERT DATA` operation, for example, can either result in the creation of a new resource or in the manipulation of an existing one if a previously unset attribute of an existing resource is set. The same applies for a `DELETE DATA` operation which could just unset one attribute of a resource or delete a whole resource. A resource will only be deleted if all triples describing that resource are deleted. This mismatch or, better, 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 which 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, that is, the SEREDASj document, in the form of semantic annotations, a client is able to construct valid requests, respectively, to detect invalid requests and to 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 response. This is similar to HTML forms with, and without client side form validation in the human Web.

In order to better explain the translation algorithm, and as a proof of concept, we implemented a simple event guide Web service based on the interface described in Listing 1. Its only function is to store events and their respective performers via a RESTful interface. The CRUD operations are mapped to the HTTP verbs `POST`, `GET`, `PUT`, and `DELETE` and no authentication mechanism is used as we currently do not have an ontology to describe this in an SEREDASj document (this is a limitation that will be addressed in future work).

The event representations can be accessed by `/event/{id}` URIs while the performers are accessible by `/person/{id}` URIs. Both can be edited by PUTing an updated JSON representation to the respective URI. New events and performers/persons can be created by POSTing a JSON representation to the collection URI. All this information as well as the mapping to the respective vocabularies is described machine-readable by SEREDASj documents.

Since SPARQL differentiates between data and template operations, we split the translation algorithm into two parts. Algorithm 1 translates SPARQL 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 4 contains an exemplary `INSERT DATA` operation which we will use to explain Algorithm 1. It creates a new event and a new performer. The event is linked to the newly created performer as well as to an existing one.

To translate the operations in Listing 4 into HTTP requests suitable to interact 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 which 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 4 does not reference existing resources (`pers:p84098` in line 10 is just used as an object), all potential HTTP requests have to create new resources, that is, have to be `POST` requests. In our trivial example, we get two potential requests, one for the creation of a new event resource and a second for a new person/performer 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 valid (line 8), that is, it contains all the mandatory data, it will be submitted (line 9). As shown in Listing 5, the first valid request creates a new event (lines 1–3). Since the ID of the blank node `_:bieber` is not known yet (it gets created by the server), it is simply ignored. Provided the HTTP request was successful, in the next step the response is 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 line 7–10 in Listing 4 are removed and the blank node in the triple in line 11 is replaced by the newly created `/event/e51972` URI. 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 request to create a person which again results in a `POST` request (line 6–8, Listing 5). 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 11 (Listing 4), thus, the only potential request this time is a `PUT` request to update the newly created `/event/e51972`. 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

```
1  do
2    requests ← retrievePotentialRequests(triples)
3    progress ← false
4    while requests.hasNext() = true do
5        request ← requests.next()
6        request.setData(triples)
7        request.setData(tripleStore)
8        if isValid(request) = true then
9          if request.submit() = success then
10           resp ← request.parseResponse()
11           triples.update(resp.getTriples())
12           tripleStore.update(resp.getTriples())
13           requests.remove(request)
14           progress ← true
15         end if
16       end if
17   end while
18 while progress = true
19 if triples.empty() = true then
20   success()
21 else
22   error(triples)
23 end if
```

Algorithm 1: SPARQL DATA operations to Web API translation algorithm.

```
PREFIX owl: <http://www.w3.org/2002/07/owl#>
PREFIX foaf: <http://xmlns.com/foaf/0.1/>
PREFIX so: <http://schema.org/>
PREFIX ex: <http://example.com/onto#>
PREFIX pers: <http://example.com/person/>

INSERT DATA {
    _:greatg a so:Event;
            so:name "Great Gig";
            so:startDate "2012-08-03";
            so:performers pers:p84098;
            so:performers _:bieber.
    _:bieber a so:Person;
            so:name "Justin Bieber";
            so:gender "male";
            so:birthDate "1994-03-01".
}
```

Listing 4: Examplary INSERT DATA operation.

to the client (line 20, Algorithm 1) as all triples have been successfully processed.

4.5. 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 using templates to delete or add groups of triples. These operations are processed by first executing the query patterns in the `WHERE` clause which bind values to a set of named variables. Then, these bindings are used to instantiate the `DELETE` and the `INSERT` templates. Finally, the concrete deletes are performed followed by the concrete inserts. The `DELETE/INSERT` operations are, thus, in fact, transformed to concrete `DELETE DATA/INSERT`

```
→ POST /event/
    { "name": "Great Gig",
     "performers": [{ "id": "p84098" }]}
← 201 Created
  Location: /event/e51972

→ POST /person/
    { "name": "Justin Bieber", "gender": "male",
     "birthdate": "1994-03-01" }
← 201 Created
  Location: /person/p92167

→ PUT /event/e51972
    { "name": "Great Gig",
     "performers": [ { "id": "p84098" } ,
                     { "id": "p92167" } ] }
← 200 OK
```

Listing 5: INSERT DATA operation translated to HTTP requests.

```
select ← createSelect(query)
bindings ← tripleStore.execute(select)

for each binding in bindings do
  deleteData ← createDeleteData(query, binding)
  operations.add(deleteData)
  insertData ← createInsertData(query, binding)
  operations.add(insertData)
end for

operations.sort()
translateDataOperations(operations)
```

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

```
DELETE {
 ?per so:gender ?gender.
}
INSERT {
  ?per so:gender "female".
}
WHERE {
  ?per a so:Person;
     so:name "Lady Gaga";
     so:birthDate "1986-03-28";
     so:gender ?gender.
}
```

Listing 6: Examplary DELETE/INSERT operation.

```
DELETE DATA {
  </person/p84098> so:gender "unknown".
}
INSERT DATA {
  </person/p84098> so:gender "female".
}
```

Listing 7: DELETE DATA/INSERT DATA operations generated by Algorithm 2 out of Listing 6.

DATA operations before execution. We exploit this fact in Algorithm 2 which transforms DELETE/INSERT operations to DELETE DATA/INSERT DATA operations which are then translated by Algorithm 1 into HTTP requests.

Listing 6 contains an exemplary DELETE/INSERT operation which replaces the gender of all persons whose name “Lady Gaga” and whose birth date is March 28, 1986, with “female” 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 work), otherwise, the operation might be executed just partially. For each of the retrieved bindings (line 3), one `DELETE DATA` (line 4) and one `INSERT DATA` (line 6) operation are created. In our example, the result consists of a single binding, namely, `</person/p84098>` for `per` and some unknown value for `gender`. Therefore, only one `DELETE DATA` and one `INSERT DATA` operation are created as shown in Listing 7. Finally, these operations are sorted (line 9) as deletes have to be executed before inserts and eventually translated into HTTP requests (line 10) by Algorithm 1.

In many cases, just as demonstrated in the example, a `DELETE/INSERT` operation will actually represent a replacement of triples. Thus, both, the `DELETE DATA` and the `INSERT DATA` operation are performed locally before issuing the 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 to one.

5. Conclusions and Future Work

In this paper, we presented SEREDASj, a new approach to describe RESTful data services. In contrast to previous approaches, we put strong emphasis on simplicity to lower the entry barrier. Web developers can use tools and knowledge they are mostly already familiar with. Since SEREDASj does not require any changes on the described Web service, it provides a viable upgrade path for existing infrastructure. We also introduced two algorithms to translate SPARQL Update operations to HTTP requests interacting with an SEREDASj-described Web API. This creates a standardized interface which not only increases the developer's productivity but also improves code reusability.

A limitation of the current proposal is that it is restricted to resources represented in JSON; no other media types are supported at the moment. In future work, support should be extended to other formats such as, for example, XML. Potentially, this could be done by mapping XML representations to JSON as there are already promising approaches such as the JSON Markup Language (JsonML) [15] to do so. This would allow to transparently support XML representations without changing the current approach. Similarly, URI templates could be used to support the popular *application/x-www-form-urlencoded* media type.

In future work, we would also like to create a tool suite for developers to support the creation of SEREDASj descriptions and, if needed, the automatic creation of domain ontologies with techniques similar to the ones used to create domain ontologies from relational databases [32]. Moreover, we would like to research aspects such as service discovery and composition which includes issues like authentication that might require the creation of a lightweight ontology to be described.

References

[1] The application/json Media Type for JavaScript Object Notation (JSON), Request for Comments 4627, Internet Engineering Task Force (IETF), 2006.

[2] T. Vitvar and J. Musser, “ProgrammableWeb.com: statistics, trends, and best practices,” in *Proceedings of the 4th International Workshop on Web APIs and Services Mashups*, 2010.

[3] M. Lanthaler and C. Gütl, “A semantic description language for RESTful data services to combat Semaphobia,” in *Proceedings of the 5th IEEE International Conference on Digital Ecosystems and Technologies (DEST ’11)*, pp. 47–53, IEEE, 2011.

[4] S. L. Huang, “After f8—resources for building the personalized Web,” Facebook Developer Blog, 2010, http://developers.facebook.com/blog/post/379.

[5] Semantic Annotations for WSDL and XML Schema (SAWSDL), W3C Recommendation, 2007.

[6] J. Lathem, K. Gomadam, and A. P. Sheth, “SA-REST and (S)mashups: adding semantics to RESTful services,” in *Proceedings of the International Conference on Semantic Computing(ICSC ’07)*, pp. 469–476, IEEE, September 2007.

[7] J. Kopecký, K. Gomadam, and T. Vitvar, “hRESTS: an HTML microformat for describing RESTful Web services,” in *Proceedings of the IEEE/WIC/ACM International Conference on Web Intelligence and Intelligent Agent Technology (WI ’08)*, pp. 619–625, 2008.

[8] M.J. Hadley, Web Application Description Language (WADL), 2009.

[9] R. Khare and T. Çelik, “Microformats: a pragmatic path to the semantic web, 2006,” Tech. Rep. 06-01, CommerceNet Labs, Palo Alto, CA, USA, http://wiki.commerce.net/wikiima-ges/e/ea/CN-TR-06-01.pdf.

[10] J. Kopecký and T. Vitvar, D38v0.1 MicroWSMO: Semantic Description of RESTful Services, 2008, http://wsmo.org/TR/d38/v0.1/20080219/d38v01_20080219.pdf.

[11] Web Services Description Language (WSDL) Version 2.0, W3C Recommendation, 2007.

[12] OWL S: Semantic Markup for Web Services, W3C Member Submission, 2004, http://www.w3.org/Submission/OWL-S/.

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

[14] R. Lara, D. Roman, A. Polleres, and D. Fensel, “A conceptual comparison of WSMO and OWL-S,” in *Proceedings of the European Conference on Web Services (ECOWS ’04)*, vol. 3250, pp. 254–269, Erfurt, Germany, 2004.

[15] JSON Markup Language (JsonML), 2011, http://jsonml.org/.

[16] D. Roman, U. Keller, H. Lausen, and J. D. Bruijn, “Web service modeling ontology,” *Applied Ontology*, vol. 1, no. 1, pp. 77–106, 2005.

[17] J. Waldo, G. Wyant, A. Wollrath, and S. Kendall, “A note on distributed computing,” Tech. Rep., Mountain View, Calif, USA, 1994.

[18] R. Alarcón and E. Wilde, “Linking data from RESTful services,” in *Proceedings of the 3rd Workshop on Linked Data on the Web*, 2010.

[19] C. Bizer and R. Cyganiak, “D2R server—publishing relational databases on the Semantic Web,” in *proceedings of the 5th International Semantic Web Conference (ISWC ’06)*, 2006.

[20] S. Auer, S. Dietzold, J. Lehmann, S. Hellmann, and D. Aumueller, "Triplify—lightweight linked data publication from relational databases," in *Proceedings of the 18th International Conference on World Wide Web (WWW '09)*, pp. 621–630, 2009.

[21] T. Berners-Lee, Y. Chen, L. Chilton et al., "Tabulator: exploring and analyzing linked data on the semantic web," in *3rd International Semantic Web User Interaction Workshop (SWUI '06)*, 2006.

[22] O. Lassila, "Browsing the Semantic Web," in *Proceedings of the 5th International Workshop on Semantic (WebS '06)*, pp. 365–369, 2006.

[23] C. Bizer and T. Gauß, Disco—Hyperdata Browser, http://www4.wiwiss.fu-berlin.de/bizer/ng4j/disco/.

[24] D. Steer, RDFAuthor, http://rdfweb.org/people/damian/RDFAuthor/.

[25] E. Pietriga, IsaViz: a visual authoring tool for RDF, http://www.w3.org/2001/11/IsaViz/.

[26] T. Berners-Lee, J. Hollenbach, K. Lu, J. Presbrey, E. Pru d'ommeaux, and M.M. Schraefel, "Tabulator Redux: writing into the semantic web," Tech. Rep. ECSIAM-eprint14773, University of Southampton, Southampton, UK, 2007.

[27] pushback—Write Data Back From RDF to Non-RDF Sources, http://www.w3.org/wiki/PushBackDataToLegacySources.

[28] R.T. Fielding, *Architectural styles and the design of network-based software architectures*, Ph.D. dissertation, Department of Information and Computer Science, University of California, Irvine, Calif, USA, 2000.

[29] XML Core Working Group Public Page—Pubblications, XML Core Working Group, 2011, http://www.w3.org/XML/Core/#Publications.

[30] CURIE syntax 1.0: a syntax for expressing compact URIs, W3C Working Group note. W3C, 2010, http://www.w3.org/TR/curie/.

[31] Google Inc., Yahoo Inc., and Microsoft Corporation., Schema.org, http://www.schema.org/.

[32] F. Cerbah, "Learning highly structured semantic repositories from relational databases: the RDBToOnto tool," in *Proceedings of the 5th European Semantic Web Conference (ESWC '08)*, pp. 777–781, Springer, 2008.

[33] SPARQL Query Language for RDF. W3C Recommendation, 2008, http://www.w3.org/TR/2008/REC-rdf-sparql-query-20080115/.

[34] SPARQL 1.1 Update. W3C Working Draft, 2011, http://www.w3.org/TR/2011/WD-sparql11-update-20110512/.