Aligning Web Services with the Semantic Web to Create a Global Read-Write Graph of Data

Markus Lanthaler [1]
[1] Institute for Information Systems and Computer Media
Graz University of Technology
Graz, Austria

Christian Gütl [1,2]
[2] School of Information Systems
Curtin University of Technology
Perth, Australia

Abstract—**Despite significant research and development efforts, the vision of the Semantic Web yielding to a Web of Data has not yet become reality. Even though initiatives such as Linking Open Data gained traction recently, the Web of Data is still clearly outpaced by the growth of the traditional, document-based Web. Instead of releasing data in the form of RDF, many publishers choose to publish their data in the form of Web services. The reasons for this are manifold. Given that RESTful Web services closely resemble the document-based Web, they are not only perceived as less complex and disruptive, but also provide read-write interfaces to the underlying data. In contrast, the current Semantic Web is essentially read-only which clearly inhibits networking effects and engagement of the crowd. On the other hand, the prevalent use of proprietary schemas to represent the data published by Web services inhibits generic browsers or crawlers to access and understand this data; the consequence are islands of data instead of a global graph of data forming the envisioned Semantic Web. We thus propose a novel approach to integrate Web services into the Web of Data by introducing an algorithm to translate SPARQL queries to HTTP requests. The aim is to create a global read-write graph of data and to standardize the mashup development process. We try to keep the approach as familiar and simple as possible to lower the entry barrier and foster the adoption of our approach. Thus, we based our proposal on SEREDASj, a semantic description language for RESTful data services, for making proprietary JSON service schemas accessible.**

*Keywords—**Web services; Web APIs; Semantic Web; Linked Data; SEREDASj; SPARQL; JSON; REST; Internet***

I. INTRODUCTION

Despite the recent uptake of the Linked Data movement, the vision of a Semantic Web, which has been around for more than fifteen years, still has a long way to go before mainstream adoption will be achieved. The reasons for this are manifold. First of all the Semantic Web suffers from a classical chicken-and-egg problem as there are no clear incentives for developers to use it. This aspect is improving recently as major search engines started to index some structured data such as RDFa and microformats. Another factor, especially in the enterprise space, is that the Semantic Web is perceived as a disruptive technology, making it a show-stopper for organizations needing to evolve their systems and build upon existing infrastructure investments. Additionally, the current Semantic Web approaches usually provide just read-only interfaces to the underlying data. This clearly limits the usefulness and inhibits networking effects and engagement of the crowd. Beside these technical issues, a lot of developers are simply overwhelmed by the complexity, perceived or otherwise, or are 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* [1]. To help developers get past this fear, and to show them that it is baseless, 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. RESTful data services could prove to be a viable solution to these problems.

Contrary to the adoption of the Semantic Web, Web services, especially RESTful ones using JSON (JavaScript Object Notation) [2] as the serialization format, are increasingly popular. Recently the term *Web API* has emerged as the collective name for RESTful Web services in order to distinguish them from their flawed [3] SOAP-based counterparts. Thus, throughout this paper, the terms Web API and RESTful Web service are considered interchangeable.

According to ProgrammableWeb's statistics [4], 2010 has seen a twofold increase in new APIs per month compared to the year before. 74% of the APIs are now RESTful and 45% of them use JSON as the data format; some of the biggest service providers' APIs such as, e.g., Facebook's Graph API [5], are now JSON-only. It is interesting to note that more and more companies are starting to build APIs as their primary product instead of just providing APIs as extensions of existing products; their whole business model is now based upon their APIs. In spite of their growing adoption, Web APIs still have some shortcomings.

The major problem of RESTful services is that no agreed machine-readable description format exists to document them. 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. In consequence, machine-to-machine communication is often based on static knowledge and tight coupling to resolve those issues. 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. After all, a Web service can be seen as a special Web page meant to be consumed by an autonomous program as opposed to a human being. Another issue is the prevalent use of proprietary schemas to represent data by these Web APIs. This inhibits the ability of generic browsers or crawlers to access and understand this data.

Since REST's principles align well with Linked Data principles, it would seem natural to combine their strengths. Nevertheless, the two remain largely separated in practice. Instead of providing Linked Data via RESTful Web services, current efforts deploy centralistic SPARQL endpoints or Tabulator-like interfaces, or simple upload static dumps of the data in order to provide access to the data. This rarely reflects the nature of the data, i.e., descriptions of interlinked resources. Just as public SQL endpoints are uncommon nowadays, public SPARQL endpoints are not expected to become widespread in the near future. This is because it is considerably more expensive to expose SQL or SPARQL endpoints than easier-to-optimize RESTful service interfaces. We thus propose an approach were Web services are integrated with data in RDF and accessed by using SPARQL.

In [1] we laid out the foundation for the integration of RESTful data services and the Web of Data by creating a semantic description language for JSON services. We choose to base our approach on JSON instead of XML to avoid the inherent impedance mismatch between XML and object oriented programming constructs (the so called O/X impedance mismatch). JSON, the JavaScript Object Notation [2] was specifically designed for data interchange. It is a lightweight, language-independent data-interchange format that is easy to parse, and easy to generate. Furthermore, it is much less complex than XML as its specification [2] consists of merely four pages (ten pages in total, including the table of content and other mostly irrelevant material).

The primary contribution of this work is a new model for read-write capable Linked Data applications by integrating JSON services into the Web of Data. This aims to generalize the application respectively mashup development by homogenizing the proprietary schemas used in today's Web services to a common data format, namely RDF. The approach is based on SEREDASj, a semantic description language for JSON services we introduced in [1].

The reminder of the paper is organized as follows. First, in section II, we give an overview of related work. Then, in section III we briefly present SEREDASj. After discussing the requirements in section IV, we propose a novel approach to integrate Web services into the Linked Data cloud in section V by introducing an algorithm to translate SPARQL queries to HTTP requests in section VI. Finally, section VII concludes the paper and gives an overview of future work.

II. Related Work

As previously outlined, one of the limitations of the current Semantic Web is that it usually just provides a read-only interface to the underlying data. SPARQL [6], the standardized query language for RDF just defines how to retrieve data, not how to manipulate it. This limitation is addressed by SPARQL Update [7], which is still a working draft. So, while several Semantic Web browsers, such as Tabulator [8], Oink [9] or Disco [10], 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 [11] and ISAViz [12] but, to our best knowledge, Tabulator Redux [13] is the only editor that allows the editing of graphs derived from multiple sources.

To mitigate this situation, the *pushback project* [14] was initiated in 2009 (unfortunately 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. This surely makes a lot of sense as large parts of the current Web of Data are generated from non-RDF databases by tools such as D2R [15] or Triplify [16]. 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 referred to as *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 no agreed standards for describing RESTful services; neither syntactically nor semantically. Exposing these Web APIs as Linked Data is therefore more an art than a science and thus the numerous proposals for describing RESTful services follow quite different approaches.

SA-REST [17] and hRESTS [18] are two approaches that enrich the, mostly already existing human-readable HTML documentation with RDFa or microformats to make it machine-processable. The biggest difference between them is that SA-REST has some built in support for semantic annotations whereas hRESTS provides nothing more than a label for the inputs and outputs. SA-REST uses the concept of lifting and lowering schema mappings to translate the data structures in the input and outputs to the data structure of an ontology, the grounding schema, to facilitate data integration. MicroWSMO [19] is an extension for hRESTS adding similar semantic annotations.

The Web Application Description Language (WADL) [20] falls in another category as it is closely related to WSDL. WADL describes a service by creating a monolithic XML file containing all the information about the service interface. This syntactic description of the service can then be semantically annotated by, for instance, SBWS (Semantic Bridge for Web Services) [21]. In principle, a RESTful service could even be described by using WSDL 2.0 with SAWSDL and an ontology like OWL-S or WSMO-Lite. As a complete description of these ontologies and interface description formats is beyond the scope of this paper we would like to refer you to [22].

The problem with all of the above described approaches is that they rely heavily on RPC's (Remote Procedure Call) flawed [3] 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 results 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 [23], the Resource Linking Language. It is a language to describe RESTful services with emphasis on the hypermedia characteristics of the REST model. This allows, e.g., a crawler to automatically retrieve the data exposed by Web APIs. One of the aims of ReLL is to transform this 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 any semantic annotations but relies on XSLT transformations to do so. This clearly limits ReLL's expressivity as it is not able to describe the resource representations semantically.

To overcome these and other shortcomings we introduced SEREDASj [1], the description language for *SEmantic REstful DAta Services*. It is similar to ReLL in that it focuses on the description of resource representations and their interconnections. It also allows, just as ReLL+XSLT, to transform these representations to RDF. But instead of being based upon XML as ReLL is, SEREDASj is based on JSON. By having built-in support for semantic annotations in contrast to the transformation process being driven by XSLT transformations, SEREDASj supports more use cases. It is, e.g., possible to create whole service documentations out of SEREDASj descriptions. It thus follows exactly the opposite approach as SA-REST and hRESTS which annotate existent documentations. The reasoning behind this is that developers usually prefer to write code than documentation. Given that this work is based on SEREDASj we describe it in more detail in the next section.

III. SEREDASJ

SEREDASj [1] specifies the syntactic structure of a specific JSON representation. Additionally, it allows to reference JSON elements to concepts in an ontology and to further describe the element itself by semantic annotations. Figure 1 depicts the structure of a 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. 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 as well as semantic annotations to

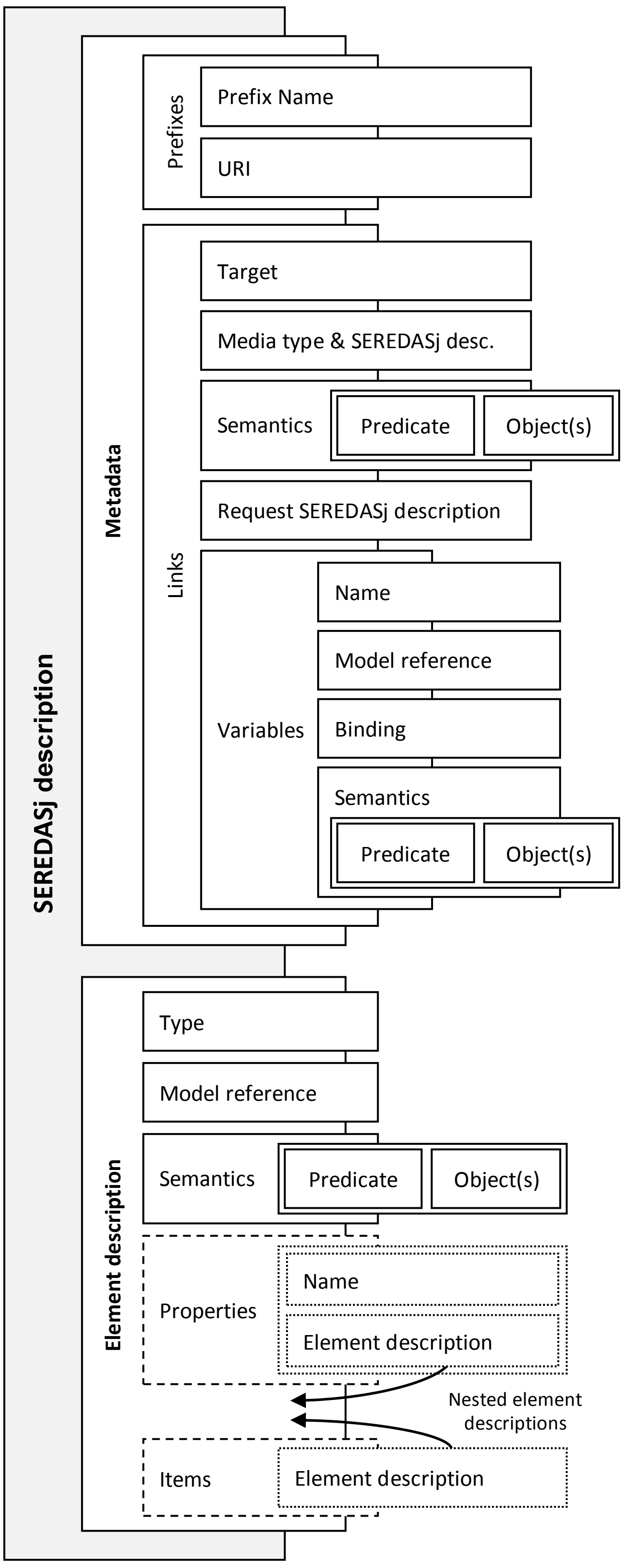


Figure 1. The SEREDASj description model

describe the link, e.g., 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, e.g., 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. This represents a current limitation as it requires the request bodies to be serialized in JSON. Currently there are also no mechanisms to describe, e.g., required HTTP headers which are often needed for features such as authentication. We plan to address these limitations by the creation of a lightweight ontology to describe exactly those aspects.

The description of the structure of instance representations (denoted as *element description* in Figure 1) defines the allowed JSON data type(s) as well as links to conceptual models. Furthermore, it may contain semantic annotations to describe an element 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 these 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 reference different parts of a model, a slash-delimited fragment resolution is used.

By describing all the important aspects of a resource representation, SEREDASj not only allows extracting hyperlinks, but also creating a human-readable documentation of the data format and translating the JSON representation to a RDF representation. This requires a mapping of the JSON structure to an ontology. The mapping strategy is similar to the table-to-class, column-to-predicate strategy of current relational database-to-RDF approaches; JSON objects are mapped to classes, all the rest to predicates.

SEREDASj descriptions don't have to be complete, i.e., they do not need to describe every element in all details. If an unknown element is encountered in an instance representation, it is simple ignored. This way, SEREDASj allows forward compatibility as well as extensibility and diminishes the coupling.

IV. Requirements

Currently, in mashup development, developers have to deal with a plethora of heterogeneous data formats and service interfaces for which little 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 the issues and limitations outlined in the previous sections and taking into account our experience in creating mashups and web applications, we derived a set of requirements for our proposal which we see as a first step towards this ambitious goal.

To be widely accepted the model 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. 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. For instance, a typical mashup combining and showing data from different sources on a map could be created automatically. The widget would be able to automatically figure out which fields in the representation represent the needed coordinates and in consequence render the data on the map. This would render the creation of dashboards, an important business use case, much simpler and eliminate a lot of the usually needed data mediation code. Another feature demanding a semantic description of a service interface, especially for JSON-based services, is the ability to create more flexible and dynamic service consumers or clients. JSON, for instance, has no built-in support for hyperlinks which makes it impossible to build services following the *hypertext as the engine of application state (HATEOAS)* constraint without additional out-of-band information. This leads to the undesirable consequence of tighter coupled services. By using semantic annotations, the client will not only be able to figure out which elements in a JSON representation represent URIs, but also what these URIs and all the other elements mean. To be able to evolve systems and build upon existing infrastructure, it is an important requirement that no, or just minimal changes of the existing system are required

Summarized, the most important aspects are how resources can be accessed, how they are represented, and how they are interlinked. A mashup should then be able to retrieve and manipulate representations of these resources.

V. Integrating Web APIs into the Web of Data

Based on the requirements outlined in the previous section, we propose a new reference model for integrating traditional Web service interfaces into the Web of Data in this section. This might be used to standardize and streamline the mashup creation process and should result in a global read-write graph of data.

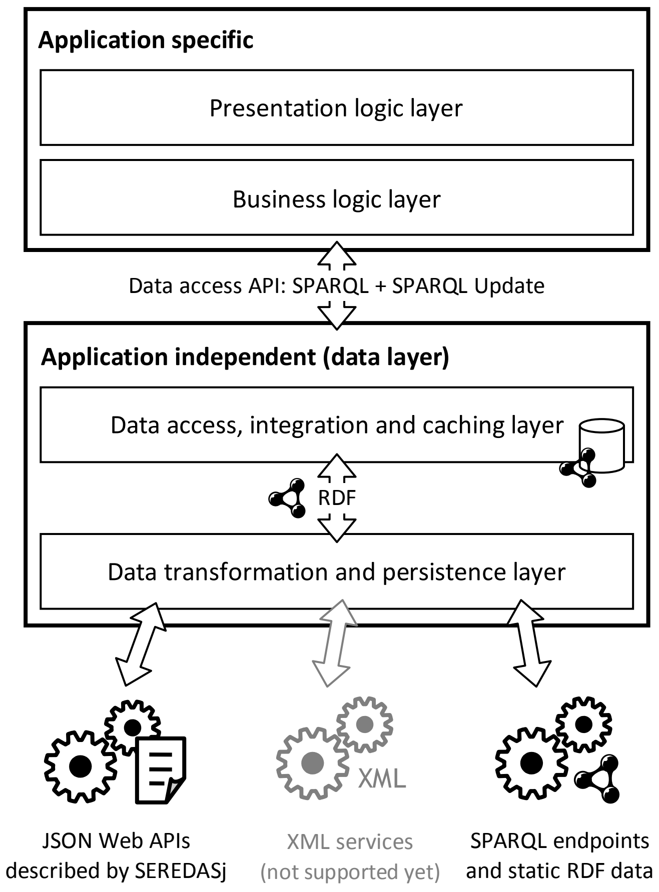

Application specific
Presentation logic layer
Business logic layer
Data access API: SPARQL + SPARQL Update
Application independent (data layer)
Data access, integration and caching layer
RDF
Data transformation and persistence layer
XML
JSON Web APIs described by SEREDASj
XML services (not supported yet)
SPARQL endpoints and static RDF data

Figure 2. A reference model for integrating Web APIs into the Web 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 traditional, JSON-based Web services, which are 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. 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 data which is a fundamental requirement in network-based applications.

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 [6] and SPARQL Update [7]. 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.

VI. Translating SPARQL Update to HTTP Requests

As previously outlined, a big part of the current Semantic Web consists of data transformed from Web APIs or relational databases to RDF. In [1] we demonstrated how data exposed by SEREDASj-described Web services can be harvested and transformed into RDF. Thus, in this paper we concentrate on how data can be updated and how new data can be inserted into those "legacy" data sources. In 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 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 a SPARQL query and the equivalent HTTP requests to interact with a Web service.

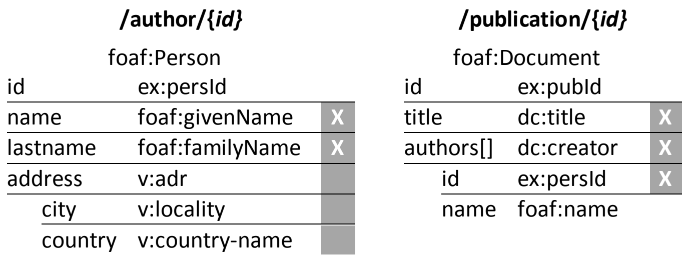
A. *Translating INSERT DATA and DELETE DATA*

In regard to a Web service, an `INSERT DATA` operation, e.g., 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 representation 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, i.e., 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 publication management Web service whose interface is shown in Figure 3. Its only function is to store publications and their respective authors 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 a SEREDASj document (this is a limitation that will be addressed in future work).

The author representations can be accessed by `/author/`*`{id}`* URIs while the publications are accessible by `/publication/`*`{id}`* URIs. Both can be edited by PUTing an updated JSON representation to the respective URI. New authors and publications can be created by POSTing a JSON representation to the collection URI. The representations for updated respectively new resources consist of the fields marked in gray; the ones denoted with an *X* are mandatory. Fields, such as the IDs, which are not highlighted, are not part of requests as they are managed by the Web service and can't be changed by a client. All this information as well as the mapping to the ontologies as shown in Figure 3 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.


/author/*{id}*

foaf:Person

id	ex:persId	
name	foaf:givenName	X
lastname	foaf:familyName	X
address	v:adr	▨
city	v:locality	▨
country	v:country-name	▨

/publication/*{id}*

foaf:Document

id	ex:pubId	
title	dc:title	X
authors[]	dc:creator	X
id	ex:persId	X
name	foaf:name	

Figure 3. Publication management service interface

```
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

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

To translate the operations in Listing 1 into HTTP requests suitable to interact with a 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 1 does not reference existing resources (`auth:cg789` in line 11 is just used as an object), all potential HTTP requests have to create new resources, i.e., have to be `POST` requests. In our trivial example we get two potential requests, one for the creation of a new publication resource and a second for a new author 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), i.e., it contains all the mandatory data, it will be submitted (line 9). As shown in Listing 2, the first valid request creates a new publication (lines 1-3). Since the ID of the blank node `_:author1` is not known yet (it gets created by the server), it is simple 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 8, 9, and 11 in Listing 1 are removed and

```
1   PREFIX foaf: <http://xmlns.com/foaf/0.1/>
2   PREFIX dc:   <http://purl.org/dc/elements/1.1/>
3   PREFIX v:    <http://www.w3.org/2006/vcard/ns#>
4   PREFIX ex:   <http://example.com/onto/>
5   PREFIX auth: <http://example.com/author/>
6
7   INSERT DATA {
8     _:public1 a foaf:Document ;
9               dc:title "My first paper" ;
10              dc:creator _:author1 ;
11              dc:creator auth:cg789 .
12    _:author1 a foaf:Person ;
13              foaf:givenName "Markus" ;
14              foaf:familyName "Lanthaler" ;
15              v:adr _:addr1 .
16    _:addr1 v:country-name "Italy" .
17  }
```

Listing 1. Examplary `INSERT DATA` operation

```
1   → POST /publication/
2       { "title": "My first paper",
3         "authors": [ { "id": "cg789" } ] }
4   ← 201 Created
5     Location: /publication/p489/

6   → POST /author/
7       { "name": "Markus", "lastname": "Lanthaler",
8         "address": { "country": "Italy" } }
9   ← 201 Created
10    Location: /author/ml980

11  → PUT /publication/p489
12      { "title": "My first paper",
13        "authors": [ { "id": "cg789" },
14                     { "id": "ml980" } ] }
15  ← 200 OK
```

Listing 2. `INSERT DATA` operation translated to HTTP requests

the blank node in the triple in line 10 is replaced by the newly created `/publication/p489` URI. Finally, the request is removed from the potential requests list and a flag is set (line 14, Listing 2) signaling that progress has been made within the current `do while` iteration. If in one loop iteration, which cycles through all potential requests, no progress has been made, the process is stopped (line 18). In our example the process is repeated for the *create author* request which again results in a `POST` request (line 6-10, Listing 2). 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 10 (Listing 1), thus, the only potential request this time is a `PUT` request updating the newly created `/publication/p489/`. As before, the request template is filled with "knowledge" from the local triple store and the remaining SPARQL triple and eventually processed. Since there are no more SPARQL triples to process, the `do while` loop terminates and a success message is returned to the client (line 20, Listing 2) as all triples have been successfully processed.

B. *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 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 3 contains an exemplary `DELETE/INSERT` operation which replaces the country name of all authors whose given name is "Christian" and whose family name is "Gütl" with "Austria". 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 4), one `DELETE DATA` (line 5) and one `INSERT DATA` (line 7) operation are created. In our example the result consists of a single binding, namely `<author/cg789>` for `x`, `</author/cg789#address>` for `addr` (this is SEREDASj's way

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

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

```
1   DELETE {
2     ?addr v:country-name ?country .
3   }
4   INSERT {
5     ?addr v:country-name "Austria" .
6   }
7   WHERE {
8     ?x a foaf:Person ;
9        foaf:givenName "Christian" ;
10       foaf:familyName "Gütl" ;
11       v:adr ?addr .
12    ?addr v:country-name ?country .
13  }
```

Listing 3. Examplary `DELETE/INSERT` operation

```
DELETE DATA {
 </author/cg789#address> v:country-name "Österreich" .
}
INSERT DATA {
 </author/cg789#address> v:country-name "Austria" .
}
```

Listing 4. `DELETE DATA`/`INSERT DATA` operations generated by Algorithm 2 out of Listing 3

to address parts within a JSON representation), and `"Österreich"` for the variable `country`. Therefore, only one `DELETE DATA` and one `INSERT DATA` operation are created as shown in Listing 4. Finally, these operations are sorted (line 11; deletes have to be executed before inserts) and translated into HTTP requests (line 12) by Algorithm 1.

In many cases, 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 set to `NULL` before getting set to the desired value. In our example this consolidates the two `PUT` requests to one.

VII. Conclusions and Future Work

In this paper, we presented a novel approach to manipulate data exposed by a RESTful Web service via SPARQL Update. We introduced two algorithms to translate SPARQL Update operations to HTTP requests interacting with a SEREDASj-described Web API. This creates a standardized interface which not only increases the developer's productivity but also improves code reusability.

There are still heated discussions in the community about how machine-readable interface descriptions, such as SEREDASj, affect the coupling between a service and its clients. Our viewpoint is pragmatic. Descriptions inherently introduce coupling but without descriptions no interpretation of the exchanged data and thus no automatic invocation is possible. We argue that whether these descriptions are machine readable or not, does not affect the degree of coupling between a service and its clients.

In future work we would like to extend SEREDASj's expressivity by creating a lightweight ontology to describe different aspects, such as, e.g., the authentication mechanism, of a service interface. We would also like to extend the model to other content types beyond JSON starting with the widely used XML and application/x-www-form-urlencoded format. Another current limitation that has to be addressed is the need to first index all data in a local triple store; potentially this could be done on the fly. Moreover, research needs to be done on the conceptual gap between SPARQL's triple model and the Web services' resource model as constraints imposed by the Web services' interfaces are transferred to SPARQL's semantic layer. This leads to the rejection of operations which would be completely valid when executed on native triple store.

References

[1] M. Lanthaler. and C. Gütl, "A semantic description language for RESTful data services to combat Semaphobia," in Proc. 5th IEEE Int. Conf. on Digital Ecosystems and Technologies (DEST), Daejeon, Korea, IEEE, 2011, in press.

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

[3] M. Lanthaler and C. Gütl, "Towards a RESTful service ecosystem - perspectives and challenges," in Proc. 4th IEEE Int. Conf. on Digital Ecosystems and Technologies (DEST), Dubai, United Arab Emirates: IEEE, 2010, pp. 209-214.

[4] T. Vitvar and J. Musser, "ProgrammableWeb.com: Statistics, trends, and best practices," Keynote of the Web APIs and Service Mashups Workshop at the European Conf. on Web Services, 2010.

[5] Facebook, Graph API, 2011, http://developers.facebook.com/docs/reference/api/.

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

[7] SPARQL 1.1 Update. W3C Working Draft, 2010, http://www.w3.org/TR/2010/WD-sparql11-update-20101014/.

[8] T. Berners-Lee, Y. Chen, L. Chilton, et al., "Tabulator: Exploring and analyzing linked data on the semantic web," in Proc. 3rd Int. Semantic Web User Interaction Workshop (SWUI 2006).

[9] O. Lassila, "Browsing the Semantic Web," in Proc. 5th Int. Workshop on Web Semantics (WebS 2006), pp. 365-369.

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

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

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

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

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

[15] C. Bizer and R. Cyganiak, "D2R Server – Publishing relational databases on the Semantic Web," poster at the 5th Int. Semantic Web Conf., 2006.

[16] S. Auer, S. Dietzold, J. Lehmann, S. Hellmann, and D. Aumueller, "Triplify – Lightweight linked data publication from relational databases," in Proc. 18th Int. Conf. on World Wide Web (WWW 2009), pp. 621-630.

[17] J. Lathem, K. Gomadam, and A.P. Sheth, "SA-REST and (S)mashups: Adding semantics to RESTful services," in Proc. Int. Conf. on Semantic Computing 2007 (ICSC2007), pp. 469-476.

[18] J. Kopecký, K. Gomadam, and T. Vitvar, "hRESTS: An HTML microformat for describing RESTful Web services," in Proc. 2008 IEEE/WIC/ACM Int. Conf. on Web Intelligence and Intelligent Agent Technology, pp. 619-625.

[19] M. Maleshkova and J. Kopecký, "Adapting SAWSDL for semantic annotations of RESTful services," in On the Move to Meaningful Internet Systems: OTM 2009 Workshops, Springer, pp. 917-926.

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

[21] R. Battle and E. Benson, "Bridging the semantic Web and Web 2.0 with Representational State Transfer (REST), " in Web Semantics: Science, Services and Agents on the World Wide Web, vol. 6, 2008, pp. 61-69.

[22] M. Lanthaler, M. Granitzer, C. Gütl, "Semantic Web services: State of the Art," in Proc. IADIS Int. Conf. on Internet Technologies & Society (ITS 2010), pp. 107-114.

[23] R. Alarcón and E. Wilde, "Linking data from RESTful services," in Proc. 3rd Workshop on Linked Data on the Web, 2010.