Accessing Site-Specific APIs Through Write-Wrappers From The Web of Data

Oana Ureche, Aftab Iqbal, Richard Cyganiak, and Michael Hausenblas

Digital Enterprise Research Institute, NUI Galway, Ireland
{firstname.lastname}@deri.org

Abstract. Web of Data formats such as linked data, RDFa and microformats are read-only technologies. The issue of updating Web data directly from an RDF-based environment has so far gained limited attention. We present a preliminary report on the idea of "pushing back" modifications of RDF data to the original source, focusing on the specific case where data resides behind site-specific Web APIs.

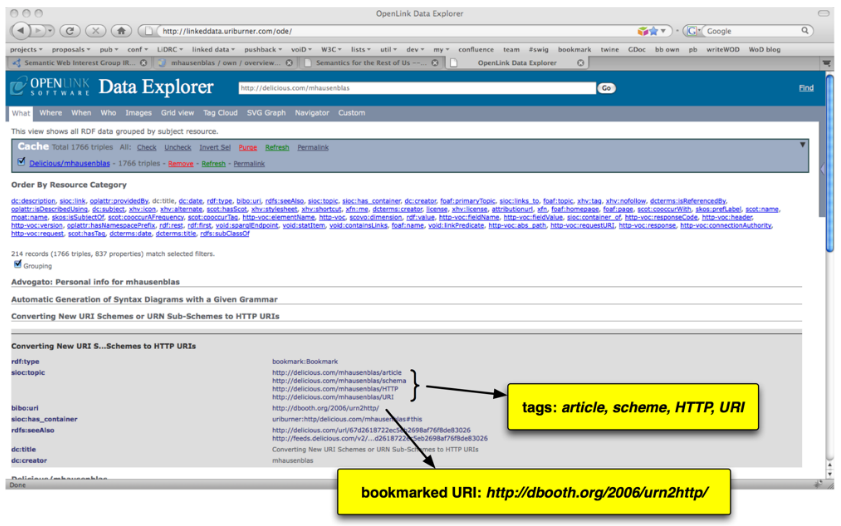
1 Introduction

The *linked data* meme [?] in general, and the *Linking Open Data* community project[1] in particular triggered the publication of vast amounts of RDF-based data; billions of RDF triples, interlinked with hundreds of millions of typed links are available as part of the Web of Data [?]. Furthermore, search engines, such as Google and Yahoo! started to index structured data (essentially, RDFa and microformats in HTML). A significant fraction of the linked data available on the Web is generated by RDF wrappers around popular Web 2.0 sites, such as Flickr [?], MySpace [?] and delicious [?]. From a consumer point of view this is a positive development, enabling one to build (Web) applications [?,?] that can exploit Web data from a large variety of sources through a uniform interface.

However, linked data, RDFa and microformats are read-only technologies. *Manipulation* of Web data—that is, creating, updating or deleting data—is supported through *site-specific APIs*, if at all. In the following we will use the term site-specific API synonymously for Web 2.0 API and Web services in the sense of [?]. An example of a site-specific API is the delicious.com API [?] for managing bookmarks. Taking into account that there are more than 1300 site-specific APIs at time of writing[2], and contrasting this with the number of SPARQL endpoints[3] currently available (less than 50), one might be interested in accessing site-specific APIs from an RDF-based environment, as there is considerably more data available in the former case. With RDF-based environment we mean applications that already operate on RDF data, such as Tabulator [?] or OpenLink's Data Explorer (ODE) [?].

[1] http://linkeddata.org
[2] http://www.programmableweb.com/apis
[3] http://esw.w3.org/topic/SparqlEndpoints

Let us take a look at a concrete use case. Sean is browsing the Web of Data using Tabulator. While surfing the Web of Data, he finds a bookmark from his delicious.com account. When Sean first bookmarked the URI, he assigned a few general tags. As he has gained more experience in the field since, he now wants to update the bookmark at hand (in terms of tags, description, etc.—cf. also Fig. **??**). Sean logs into the delicious.com Web application, locates the bookmark, performs the desired changes and eventually the RDF representation will be updated. When reviewing the actions necessary to complete Sean's "update" task, one may wonder if there are not more efficient means to do so. This is even more evident if Sean wants to keep his tags synchronised between several Web 2.0 sites, or if he wants to do mass updates to his bookmarks.


OpenLink Data Explorer
http://linkeddata.uriburner.com/ode/
projects proposals pub conf LiDRC linked data pushback voiD W3C lists util dev my confluence team #swig bookmark twine GDoc bb own pb writeWOD WoD blog
Semantic Web Interest Group IR... mhausenblas / own / overview... Semantics for the Rest of Us --... OpenLink Data Explorer
OPENLINK SOFTWARE Data Explorer
http://delicious.com/mhausenblas Go Find
What Where When Who Images Grid view Tag Cloud SVG Graph Navigator Custom
Cache Total 1766 triples All: Check Uncheck Invert Sel Purge Refresh Permalink
Delicious/mhausenblas - 1766 triples - Remove - Refresh - Permalink
Grouping
Done
tags: *article, scheme, HTTP, URI*
bookmarked URI: *http://dbooth.org/2006/urn2http/*

This view shows all RDF data grouped by subject resource.

Order By Resource Category

dc:description, sioc:link, oplattr:providedBy, dc:title, dc:date, rdf:type, bibo:uri, rdfs:seeAlso, sioc:topic, sioc:has_container, dc:creator, foaf:primaryTopic, sioc:links_to, foaf:topic, xhv:tag, xhv:nofollow, dcterms:isReferencedBy, oplattr:isDescribedUsing, dc:subject, xhv:icon, xhv:alternate, scot:hasScot, xhv:stylesheet, xhv:shortcut, xfn:me, dcterms:creator, license, xhv:license, attributionurl, xfn, foaf:homepage, foaf:page, scot:cooccurWith, skos:prefLabel, scot:name, moat:name, skos:isSubjectOf, scot:cooccurAFrequency, scot:cooccurTag, http-voc:elementName, http-voc, scovo:dimension, rdf:value, http-voc:fieldName, http-voc:fieldValue, sioc:container_of, http-voc:responseCode, http-voc:header, http-voc:version, oplattr:hasNamespacePrefix, rdf:rest, rdf:first, void:sparqlEndpoint, void:statItem, void:containsLinks, foaf:name, void:linkPredicate, http-voc:abs_path, http-voc:requestURI, http-voc:response, http-voc:connectionAuthority, http-voc:request, scot:hasTag, dcterms:date, dcterms:title, rdfs:subClassOf

214 records (1766 triples, 837 properties) match selected filters.

Advogato: Personal info for mhausenblas

Automatic Generation of Syntax Diagrams with a Given Grammar

Converting New URI Schemes or URN Sub-Schemes to HTTP URIs

Converting New URI S...Schemes to HTTP URIs

rdf:type	bookmark:Bookmark
sioc:topic	http://delicious.com/mhausenblas/article http://delicious.com/mhausenblas/schema http://delicious.com/mhausenblas/HTTP http://delicious.com/mhausenblas/URI
bibo:uri	http://dbooth.org/2006/urn2http/
sioc:has_container	uriburner:http/delicious.com/mhausenblas#this
rdfs:seeAlso	http://delicious.com/url/67d2618722ec5eb2698af76f8de83026 http://feeds.delicious.com/v2/...d2618722ec5eb2698af76f8de83026
dc:title	Converting New URI Schemes or URN Sub-Schemes to HTTP URIs
dc:creator	mhausenblas

Fig. 1. The data from `http://delicious.com/mhausenblas` viewed in ODE.

Thus, the key question is: *How can data residing behind site-specific APIs efficiently be manipulated from an RDF-based environment?* In this work, we focus on the issue of mapping from RDF-based update operations to invocations of site-specific API operations. The key challenge is the mismatch between the RDF model and the local semantics of site-specific APIs.

Based on the concept of write-wrappers introduced in Section **??** we elaborate on design issues in Section **??**. Then, in Section **??** we report on an experimental implementation of a write-wrapper. We review related work in Section **??** and eventually discuss our findings in Section **??**.

2 Write-Wrappers for site-specific APIs

As we have reported in [**?**], read-only RDF wrappers are a common method to integrate data obtained from site-specific APIs into the Web of Data. The approach presented herein is hence based on *write-wrappers*. The input to a write wrapper is an operation, such as *update*, along with some RDF data (see Fig. **??**). The wrapper maps the operation and data to one or more invocations of a site-specific API. Write-wrappers are a high-level update mechanism because they map a uniform RDF-based interface into different heterogenous, and likely not RDF-based, APIs. Write-wrappers are usually *proxies* in the REST sense [**?**].

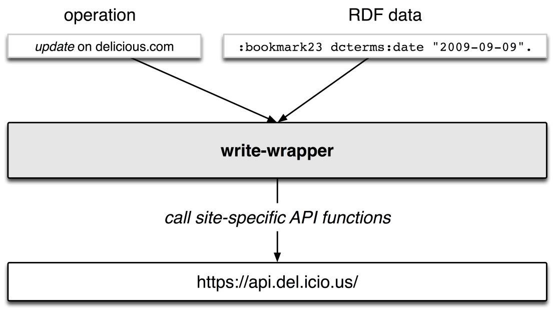


Fig. 2. The interfaces of a write-wrapper.

There are different alternatives for conveying the input operation and data. Options include:

– RESTful HTTP interfaces, which use the four uniform HTTP methods GET, POST, PUT, DELETE.
– XML-RPC interfaces.
– SPARQL Update[4].

The output-interface of the write wrapper, on the other side, is determined by the site-specific API. The write-wrapper needs to handle a variety of cases, depending on the style of the API (RESTful, XML-RPC, etc.), the offered authentication mechanisms, error handling and so forth. However, these cases can be seen as implementations of the same basic steps. Therefore, it should be possible to employ a generic mechanism to perform the mapping and API access.

RDF clients can already update triple stores through SPARQL Update [**?**]. In order to allow clients to treat site-specific APIs and RDF triple stores in the same manner, we decided to build our update mechanism on SPARQL Update. We thus introduce the notion of a **pushback** (Fig. **??**). A pushback conceptually consists of:

1. A SPARQL Update *input interface* towards the client;
2. A *generic mapping mechanism* that maps the SPARQL Update requests to site-specific API function calls, using site-specific, parameterized mappings;

[4] `http://www.w3.org/TR/sparql-features/#sparql-update`

3. An *output interface* that is able to call Web services and pipe results back to the client.

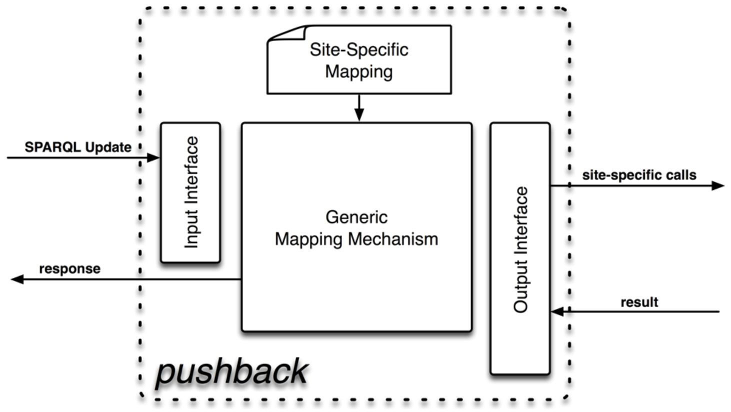


Fig. 3. Architecture diagram of a pushback RDF write-wrapper.

As an initial step towards a generic pushback solution, in this paper we discuss design considerations for the components of a pushback and further discuss an experimental implementation for a specific case.

3 Design Considerations

3.1 Input Interface

The first component of a pushback is the *SPARQL Update input interface.* SPARQL Update is at time of writing under standardisation, hence a moving target. Beside this, the SPARQL Update is straight-forward to implement; an array of existing libraries and platforms, for example, ARC2[5] (for PHP) or ARQ/Jena[6] (for Java) exist.

Type of Operation. Essentially, we have identified two options to encode the operation in a request from the RDF-based client, that is, the update request semantics. First, one can use the SPARQL Update operations such as `INSERT`, `DELETE`, etc. to convey the semantics. Second, one can use the RDForms vocabulary[7] over SPARQL Update. The second approach is described in detail in [?].

[5] `http://arc.semsol.org/`
[6] `http://jena.sourceforge.net/`
[7] `http://rdfs.org/ns/rdforms`

3.2 Generic Mapping Mechanism

The second module of a pushback, the *mapping mechanism for both the data and operations* could take many forms. We have contemplated the following options:

1. Rules expressed in an appropriate dialect of the *Rule Interchange Format* (RIF)[8],
2. Embedded scripting language fragments that directly invoke the target API,
3. An extension of the *Web Application Description Language* (WADL)[9],
4. The creation of a new declarative mapping language.

This area is subject to further research, as we are only starting to understand the requirements and possible limitations of the candidate languages. So far we have identified the following functions that a mapping mechanism must support in order to be used in a pushback:

1. Map input operations to target function call(s) of the site-specific API;
2. For each of the target function calls, map input data elements to function call parameters;
3. Validate the parameter values against the contract of the API (required, optional, etc.);
4. Declare how users are authenticated to the site-specific API;
5. Identify whether an API function call has succeeded or failed, and represent error messages or conditions.

We will give an example of a simple mapping in the following. Assuming that the input operation is based on RDForms over SPARQL Update and that the site-specific API is an XML-RPC-style one, the 1:1 mapping could look like the following:

```
CRUDOperationCreate{
 method: POST
 call: https://api.del.icio.us/v1/posts/add?
 url.key : &url
 description.key : &description
 extended.key : &extended
 tags.key : &tags
 dt.key : &dt
 replace.key : &replace
 shared.key : &shared
}
```

Parameter Validation Site-specific API method calls usually encode a number of parameters. Table **??** illustrates, by example, some parameters characteristics for the `add` function call[10] of the delicious.com API. Explicitly, the

[8] http://www.w3.org/2005/rules/wiki
[9] https://wadl.dev.java.net/wadl20061109.pdf
[10] http://delicious.com/help/api#posts_add

	structured	**unstructured**
required	`N/A`	`&description`
optional	`&dt`	`&extended`

Table 1. Parameters characteristics for a delicious.com API function.

`&description` parameter, referring to the bookmark's title, is required and unstructured, whereas the `&dt` parameter (date stamp of the bookmark) is optional, but structured because it requires a full ISO8601[11] date format. These parameters have to be checked against their characteristics by the pushback. Note that some parameters are not updatable. The RDForm will not display a field for such parameter.

Authenticating users In the context of Web services, authentication of a client is performed by a server to determine the identity of the requesting agent. In order to perform authentication, one needs to know about the identity of an agent. Site-specific APIs support different authentication schemes, such as HTTP Basic, Digest, or API keys.

Error Handling Site-specific APIs signal errors in very different ways. For example, the delicious.com API only returns `<result code='done'>` in case the call was successful and `<result code='something went wrong'>` otherwise. The site-specific mapping must be sufficiently expressive to tell these situations apart.

3.3 Output Interface

Eventually, the third component, the *Web service output interface*, poses some challenges on its own, as it very much depends on the type of Web service. We have identified the following forms, a site-specific API function call may be performed:

1. For RESTful APIs, one can use generic libraries and tools such as *curl*;
2. Based on a mapping that is rooted in an interpreted language such as PHP, one can directly execute the scripts, with the data as well as the operations supplied as parameter;
3. Utilising WADL—WADL provides a machine processable description of RESTful Web applications and services. Using WADL, the implementation is not tied to a specific programming language. There are tools available[12] that generate code automatically out of a WADL description.

We note further that every site-specific API provides potentially different methods of authenticating users and returning error messages. We will discuss this in greater detail below.

[11] `http://www.cl.cam.ac.uk/~mgk25/iso-time.html`
[12] `http://tomayac.de/rest-describe/latest/RestDescribe.html`

4 Experimental Implementation

In this section, we provide a qualitative evaluation of an experimental implementation of a pushback against a simple API access:

Simple API access The first implementation is our evaluation base-line. In this implementation the call to a site-specific API is done directly through a basic programming script. The input data, arguments, operation and user credentials are hard-coded in the application.

API access from the Web of Data The second implementation accesses a site-specific API through a write wrapper. For the sake of completeness, the input is an RDForm. The data of the RDForm is gleaned into a SPARQL Update operation. The output is the call to a Web service. The call to a Web service takes the language specific script form. Moreover, this implementation includes form validation, secure on line log in and operation expressed using only the RDForms vocabulary. This approach uses only the SPARQL INSERT query. Note that the choice to ultimately maintain, delete or update triples from the RDF back-end is done after the SPARQL INSERT query, which requires a close observation over the RDF back-end.

Our evaluation purpose is to compare the complexity vs. the performance of the two approaches. While the second approach is obviously more complex than the first one, it is interesting to find out the difference between the processing time for both approaches. First, we will compare the two implementations by their feature characteristics. Second, we will measure the processing time for the both implementations for several consecutive accesses to a site-specific API.

The second approach includes error handling because it validates the user given fields' values. Although the authentication transparency suffers in the second approach, the security risk is minimal compared to the first approach. The second approach suffers from server-side complexity, due to the RDF back-end monitoring. Lastly, none of the two approaches offers data integrity, that is, transaction support for concurrent editing operations.

The chart in Figure **??** shows the performance of the two implementation approaches. For the evaluation we chose the delicious.com API. Due to the restriction of "at least one second between queries" required by delicious.com, we set up a two seconds throttle between requests. The terms "create" and "delete" refer to the type of the request: "create a bookmark", respectively "delete a bookmark". We observe that the time grows linearly with the number of requests performed for both approaches and both types of requests. Moreover, the create request for the second implementation approach takes twice as much to process than the create request for the first approach. The delete request takes about the same time for both implementations. While the create request encodes seven fields in the SPARQL request and thus more information to process, the delete request encodes only one field for the URL parameter, which leads to the same processing time for both approaches when the number of requests is low.

The experiment shows that a write-wrapper does not add a lot of overhead which provides evidence for the feasibility of the overall approach.

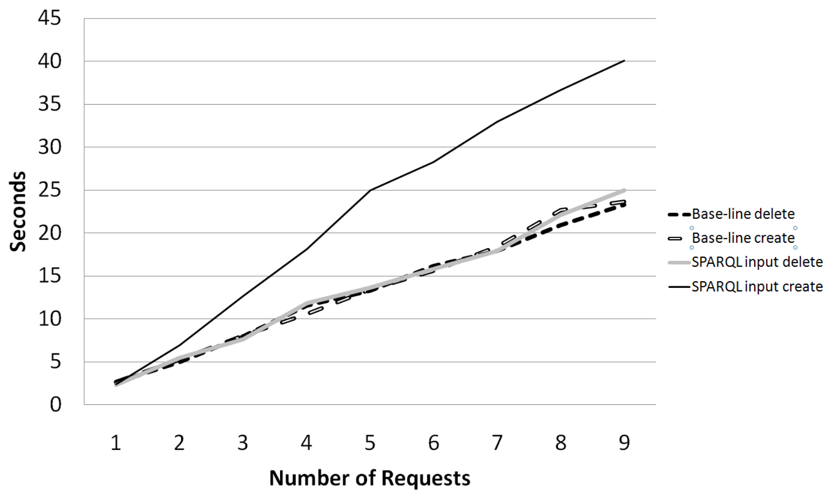


Fig. 4. Performance chart: time required to perform several requests.

5 Related Work

Basic Update Mechanisms. Three basic update mechanisms are available in the Web of Data context, (i) native HTTP methods, (ii) WebDAV, and (iii) SPARQL Update. The HTTP protocol [?] defines methods for querying and managing Web resources (such as `GET`, `POST`, `DELETE`, etc.). The Web Distributed Authoring and Versioning protocol (WebDAV) [?] extends HTTP with additional methods for collaborative authoring of resources, like `COPY` and `MOVE`. Neither HTTP methods nor WebDAV are optimal for updating RDF-based data, because of the coarse granularity of their operation, which work on entire documents. The W3C SPARQL Working Group is currently standardising update operations to allow remote manipulation of RDF data[13]. While it is subject to discussion if the proposed protocol is RESTful in the sense of the *Resource-Oriented Architecture (ROA)* [?], we rely on SPARQL Update as the basis of our update mechanism, because it allows clients to treat RDF-based back-ends, such as triple stores, and non-RDF back-ends, such as site-specific APIs, in a uniform manner.

RDF Mappings. Relational-to-RDF (RDB2RDF) mapping approaches [?] have been widely studied. The problem of mapping SPARQL update operations to site-specific API invocations can be seen as a comparable problem, with the main difference that not only data has to be mapped, but also operations. This has serious consequences, as for example one has to pipe back errors or exceptions to the client, handle concurrent editing, authorization, and authentication.

Alternative Update Approaches. The Tabulator data browser implements an update protocol based on per-triple update [?]. Dietzold et al present

[13] `http://www.w3.org/TR/sparql-features/`

a method for client-side in-place editing of data embedded as RDFa in a web page, where a statement-level diff is sent back to the server [?]. Both approaches are complementary to our work, because the performed client-side updates are, or can be, expressed as SPARQL Update operations that could be propagated to site-specific APIs using pushbacks.

Semantic web services. Major work towards connecting web services to languages such as RDF and OWL has been undertaken in the Semantic Web Services (SWS) field. In SWS terminology, the area addressed in our work is *Service Grounding*: how to invoke a service given some input that is expressed in terms of a domain ontology, and how to interpret the results in terms of the domain ontology. This is typically achieved by referring to a WSDL file plus *lowering* and *lifting transformations*, which map semantic information to and from the message syntax expected by the service. This approach is standardized in the SAWSDL [?] W3C Recommendation. XSLT is used as transformation language, although the possibility of using other languages is noted.

A SAWSDL-annotated WSDL file contains all necessary information for configuring a write wrapper around the service. But the preferred method of documenting RESTful APIs is prose text, and WSDL files are rarely provided by service operators. Setting up a write wrapper thus requires manual authoring of a WSDL file, an extremely tedious activity even for simple services, especially when compared to creating a WADL file for the same service. Thus, an extension of WADL with annotations similar to those in SAWSDL seems like a more attractive alternative. We also want to further explore alternatives to XSLT for expressing the mappings, because processing RDF/XML input with XSLT is awkward, and input and output of RESTful services are often not expressed in XML.

SA-REST [?] has been proposed as another alternative to SAWSDL specifically for RESTful services. It works by embedding annotations directly into the HTML developer documentation of the service. The supported annotations are kept very simple though, and do not cover details required by the services we attempted to map, such as authentication, input validation or the recognition of errors reported by the server.

6 Conclusion

We have argued in this paper that there is a growing need in accessing site-specific APIs from an RDF based environment. We have introduced the concept of a pushback, a SPARQL Update-based write-wrapper that is able to translate RDF data into site-specific API calls. Further, we have elaborated on the design of the three main components of a pushback, the SPARQL update interface, the generic mapping mechanism, and the output interface. For each one of them we have provided an initial design solution and implementation. Moreover, we have highlighted the limitations of the proposed design. Finally, we have compared direct access to a site-specific API and a prototype implementation of our approach.

In this paper we are not claiming to offer a comprehensive answer to our initial question *how can site-specific APIs efficiently be accessed from an RDF-based environment?*, but discuss possibilities and limitations, building a basis for our further research. We have, for example, not yet addressed issues around concurrent data updates and plan to elaborate as well on transactions to ensure data integrity. Finally, we plan to have a deeper look into the handling of results and how to pipe back errors to the client.

Acknowledgements

Our work has partly been supported by the European Commission under Grant No. 217031, FP7/ICT-2007.1.2, project "Domain Driven Design and Mashup Oriented Development based on Open Source Java Metaframework for Pragmatic, Reliable and Secure Web Development" (Romulus).