RESTful Write-oriented API for Hyperdata in Custom RDF Knowledge Bases

Jacek Kopecký, Carlos Pedrinaci
Knowledge Media Institute
The Open University, UK
j.kopecky@open.ac.uk, c.pedrinaci@open.ac.uk

Alistair Duke
Research and Technology
BT Innovate & Design, UK
alistair.duke@bt.com

Abstract

The Linked Data movement has been very successful in bringing large amounts of open RDF data on the Web. Much research and development work has been done on publishing Linked Data, while update access to linked data sources has been neglected. This paper presents a configurable approach to creating custom RESTful APIs for writing and updating linked data. In the spirit of hypertext, the approach fully embraces hyperlinking and resource descriptions, resulting in an update-enabled hyperdata.

1. Introduction

The success of the Linked Data movement [2] is marked by the availability of large amounts of RDF data sources openly on the Web. However, the Linked Data principles are aimed at making data available for reading; the publishing of write-oriented APIs has been neglected so far.

The continued growth of the Linked Data cloud demonstrates that many existing databases can be made openly available. Much work has been done to support publishing of data, leading for example to standardization efforts on languages for mapping relational databases to RDF datasets [6]. When publishing an existing database, a read-only RDF view as linked data is quite sufficient — there must already be processes and APIs in place for populating and maintaining the underlying database.

In the cases when a new database is being created, it is natural to consider whether the data (or its subset) should be openly available as Linked Data. If so, it may be appropriate to use RDF technologies for the back-end of the new database. Then it is straightforward to follow the Linked Data principles and publish the RDF data as a set of retrievable graphs, potentially with a SPARQL endpoint.

For writing into RDF-based data stores, there needs to be an update API. Triple stores provide generic APIs (e.g. the Sesame `uploadData` servlet[1]), and the W3C is working on the SPARQL Update language and protocol [9].

However, such generic update APIs including SPARQL Update are aimed rather at the internal database update interface, which should be wrapped in an application layer that enforces consistency and security. There are several reasons for this. First, a data update seldom remains only a simple database write — it often also triggers custom actions, such as propagation of changes into dependent data.

Second, imposing security limits on allowed types of updates through a general-purpose update language requires low-level access policy languages (e.g. [7]), while a custom application wrapper layer can structure a security policy according to the high-level (and coarse-grained) structure of the underlying data.

Third, a custom API can validate the updates, and guide (or constrain) its users in the structure of the accepted data (for example preventing well-meaning users from using the wrong ontology by mistake), while in generic triple stores, update validation is an uncommon feature.

Finally, the application must be in charge of creating the identifiers (URIs) for new pieces of data: according to the Linked Data principles, the identifier of a piece of data should be resolvable to its description, i.e., on a web site of the application. SPARQL Update does not provide the equivalent of an `AUTO_INCREMENT` field in an SQL database, and leaving the creation of identifiers to clients is undesirable due to the potential for conflicts, therefore the application wrapper needs to handle the creation of new identifiers.

In this paper, we present a RESTful approach to defining such application-specific update APIs, driven by a simple declarative configuration. The approach focuses heavily on *discoverability* of the update capabilities, so that when a client reads some data, it learns also how to update it — how to delete a particular piece of data, how to change it, or how to add another one like it. Our approach is based on named graphs with metadata that link the graphs to the contained data. The resulting data enriched with named graph metadata can be called *hyperdata*,[2] essentially meaning linked

[1] `http://www.openrdf.org/doc/sesame/users/ch08.html`

[2] The term, predating the Web, has been used in connection to the Web of Data, for example in `http://www.novaspivack.com/technology/the-semantic-web-collective-intelligence-and-hyperdata`.

data with self-describing write support.

In Section 2 below, we introduce a use case that led us to develop the customizable RDF update API. In Section 3, we present our approach to defining custom hyperdata update APIs, and in Section 4, we discuss our proof-of-concept implementation used to realize the back-end data store of the use case. Section 5 discusses works related to linked data and update APIs. Finally, Section 6 concludes the paper, with remarks on future work.

2. Use Case: Offers4All Back-end Data Store

Our write-oriented hyperdata API approach was developed to support the back-end data storage for a case study scenario in the research project SOA4All.[3] The project was concerned with supporting the creation and deployment of service mashups, i.e., services that are themselves constructed from existing Web Services or APIs. SOA4All implemented tools to simplify the discovery and composition of existing Web Services and APIs and also the deployment and execution of the resulting service mashups. Within the project, one case study focused on telecommunications-based mashups with a particular emphasis on business processes where telecommunications APIs (such as messaging, call setup and control) are used to improve the flow of information between companies and their customers.

The case study developed an illustrative scenario called "Offers4All". The Offers4All service allows companies such as retail organisations, entertainment providers, travel and hotel companies to advertise offers to subscribers of the service. These offers might be "last-minute" travel or entertainment deals or predefined campaign offers from retail organisations. The Offers4All service allows an offer provider to create a new offer by describing what the offer is and who it is targeted at. An appropriate set of subscribers are then chosen and are made aware of the offer via their preferred communication channel.

The Offers4All service is backed by a database that stores the details of each offer that is created by an offer provider. Details of an offer include a textual description that can be sent to users, an offer category and a target profile indicating what users should be targeted with the offer, and information about how interested users can respond to the offer in order to take it up. The target profile can include criteria such as location dependency, i.e., is this offer targeted at people near a location (which would be appropriate for example for offers of restaurants or entertainment venues), or wealth / income dependency i.e. is this offer targeted at people who are likely to be wealthy (or not). Predictions of this nature can be made from Linked Open Data sources such as the UK census data which provides average income data for UK postcodes.

In addition to offer data, the database stores data about offer providers and users. Offer providers have contact details and locations to which their offers can be related, and of course the actual offers. Users also have contact details (potentially for multiple devices or locations) and preferences about these. They can specify what offer categories they are interested in, and they can also choose to "like" an offer which allows social-networking-style recommendations to be used to increase the uptake of offers.

Finally, the database stores offer consumption data, which includes events such as a user being informed about an offer, a user choosing to respond to an offer, or a user forwarding an offer to another user. The storage of such events is important e.g. for billing or for determining the success statistics of offers and offer providers.

The data in the database is in RDF, structured according to the Linked Data principles. Two distinct roles access and modify the data: offer providers and consumers; they naturally have different access rights over the data.

To support the database interactions common in the scenario, such as new offer providers or users registering with the system, or the launching, distribution and consumption of offers, the database is façaded by three custom APIs: a Users API that stores information about users and offer providers, an Offers API for storing information about offers, and a Consumption API that handles the consumption data. To illustrate the structure of the APIs, the following is an operation-oriented view on a part of the Users API, listing the functionalities available for managing users:

- listUsers() returns a list of the known users
- addUser(data) creates a new user record
- getUser(id) returns a user record
- getUserLikes(id) returns the offers liked by the user
- addUserLike(id, uri) adds a new liked offer
- deleteUserLike(like-id) removes a liked offer
- deleteAllUserLikes(id) clears the list of liked offers
- Dislike — similar four operations as above, manipulating the list of offers explicitly disliked by a user
- Interest, Disinterest — similar operations, for offer categories of interest to the user, and for those categories that the user prefers to filter out
- Contact — similar operations for contact information
- addUserInformation(id, data) amends a user record with new arbitrary data
- deleteUserInformation(val-id) removes an arbitrary statement about the user
- replaceUserInformation(val-id, data) updates an arbitrary statement about the user

The last two operations handle data that is not foreseen in the API. In Offers4All, we used these two operations to write and update the estimate of the user's wealth status, when the

[3] `http://soa4all.eu/`

scenario was extended this way.

The remainder of the three APIs has the same general structure of operations. In the following section, we abstract from these operations to a generic configurable update API.

3. Write-oriented Hyperdata API

Each API in the use case is a container for instances of one or more classes. In this section, we will use the Users API, which manages the instances of the classes uc:User and uc:OfferProvider, to illustrate the concrete hyperdata structure and operations of the API.

Below, in Section 3.1 we define the types of resources that make up our hyperdata API, then in Section 3.2 we describe the metadata used to describe the named graphs that make up the API, and in Section 3.3 we specify the methods that should be supported by the various resources.

In the text, we use a `typewriter` font to write example URIs, abbreviated by stripping the common location prefix, e.g. `http://example.com/data`. For instance, `/users` should be read as `http://example.com/data/users`. Further, we use a sans-serif font when mentioning RDF identifiers, with several namespace prefixes: uc: for the ontology developed for the use case, ex: for example data, and g: for a graph vocabulary.

3.1. Resource Structure of the API

The principles of REST [4] call for structuring an application around its main resources. We have identified the following types of resources (with examples from the use case) that should be represented by a custom write-oriented API for RDF data:

- **class resources** (`/users`, `/providers`) for every class managed by the API, with methods to list all the instances of the given class, and to add new instances;
- **instance resources** (`/users/{id}`) for every instance of a given class, with methods to get information about the instance, and to add new information (triples) about the instance;
- **property resources** (`/users/{id}/likes`) for selected significant properties (as discussed later), with methods to get, add and remove the value(s) of the given property;
- **value resources** (`/users/{id}/likes/{like-id}`) for the property values of the instances, with methods to delete or update the particular value.

The resources above are RDF named graphs [3]. Class and property resources act as *collections* where new items can be added, creating the IDs used in the URIs. Instance and value resources are concrete items in those collections.

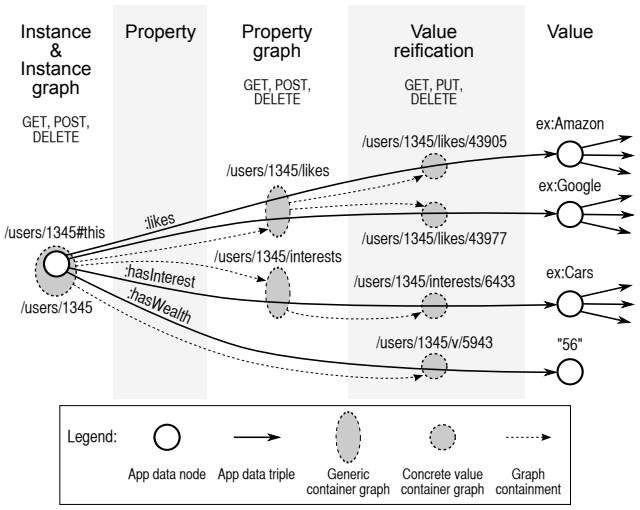

Figure 1. Hyperdata structure of the API

Figure 1 illustrates the structure of the data managed by the Users API, especially including the named graph resources for manipulating the statements. When a new user record is submitted to `/users`, a new ID is created, in the figure `/users/1345`. To follow the Linked Data principles, the submitted instance must itself be identified through a URI that is dereferencable and leads to the appropriate named graph; therefore, the API assigns instance URIs of the form of the instance resource URIs with the appended fragment identifier `#this`, such as `/users/1345#this`.

In the figure, the following information was submitted about the new user:

```
_:x a uc:User;
   uc:likes ex:Amazon, ex:Google;
   uc:hasInterest ex:Cars;
   uc:hasWealth "56".
```

For the purpose of the illustration, we use ex:Amazon and ex:Google as the identifiers of two offers that the user likes, and ex:Cars as an offer category.

The various graph resources are interlinked: class resources link to the contained instance resources; instance resources link to all their property and value resources; and property resources link to all the value resources for the given property.

The concrete structure of a given API is guided by a simple **configuration** which names the classes whose instances are managed by the API, and for each class, the significant properties. Every configuration item also has a manually chosen short name that is used in the URIs. Table 1 shows the configuration that defines the Users API.

Note the last configuration entry, which we call a *wildcard property*. Its presence in the configuration lets the API accept any other properties that are not explicitly mentioned

Config. entry	Ontology element	Short name
class	uc:User	`users`
property	uc:likes	`likes`
property	uc:dislikes	`dislikes`
property	uc:hasInterest	`interests`
property	uc:hasDisinterest	`disinterests`
property	uc:hasContact	`contacts`
property	*	`v`

Table 1. Configuration for the Users API

in the configuration; in the use case, we use this for the uc:hasWealth property. There is no property resource for the wildcard property, but the short name of the wildcard property is used in the name of the concrete value resources, e.g. `/users/1345/v/5943` in Figure 1.

3.2. Named Graph Metadata

To indicate the relation between the actual data and the named property and value graphs, we employ a vocabulary for graphs along with the RDFS *reification* vocabulary. For illustration, the data in `/users/1345` would include the triples in Listing 1, where lines 1–4 contain the actual data about the particular user (`/users/1345#this`), and the following lines show a subset of the graph metadata.

Line 7 indicates that the graph is the description of the user instance, making it possible for a client to infer that an HTTP DELETE request can remove the instance.

Line 8 links the instance graph with one of the property graphs (`/users/1345/likes`) and a wildcard property value graph (`/users/1345/v/5943`) — other graphs that would be linked here are omitted for brevity. Line 9 links the instance graph with the high-level class graph.

Lines 11–18 describe a property graph: it contains a concrete value graph, and a reified triple pattern (lines 14–17) that indicates that the graph includes statements of the form `/users/1345#this` uc:likes *something* (note the blank node on line 17). The triple pattern is meant to indicate what kind of data can be POSTed to the property resource, and what subset of the data about the user can be expected when GETting the property resource.

Finally, lines 20–26 and 28–34 describe the two concrete value graphs. The client can use PUT or DELETE on these graphs to update or remove a particular statement.

Such graph descriptions and links support *discoverability* of the update API in the true spirit of hypertext, and turn the underlying RDF data into what we call *hyperdata*. In fact, the API also has a **root resource** (`/`) intended for further discoverability — it lists the class resources managed by the given API, describing them with triple pattern like *something* rdf:type uc:User, which indicates the ontology class whose instances are managed by the resource.

```
1  </users/1345#this> a uc:User ;
2    uc:likes ex:Amazon, ex:Google ;
3    uc:hasInterest ex:Cars ;
4    uc:hasWealth "56".
5
6  </users/1345> a g:Graph ;
7    g:defines </users/1345#this> ;
8    g:contains </users/1345/likes>, </users/1345/v/5943> ;
9    g:isContainedIn </users> .
10
11 </users/1345/likes> a g:Graph ;
12   g:contains </users/1345/likes/43905> ;
13   g:contains [
14     a rdf:Statement ;
15     rdf:subject </users/1345#this> ;
16     rdf:predicate uc:likes ;
17     rdf:object [ ]
18   ] .
19
20 </users/1345/likes/43905> a g:Graph ;
21   g:contains [
22     a rdf:Statement ;
23     rdf:subject </users/1345#this> ;
24     rdf:predicate uc:likes ;
25     rdf:object ex:Amazon
26   ] .
27
28 </users/1345/v/5943> a g:Graph ;
29   g:contains [
30     a rdf:Statement ;
31     rdf:subject </users/1345#this> ;
32     rdf:predicate uc:hasWealth ;
33     rdf:object "59"
34   ] .
```

Listing 1. Example graph description triples

3.3. API Methods

We've already touched on the HTTP methods allowed by the various types of resources in our API. Below, we specify the main allowed methods in more detail.

Class resources support GET to list the instances, and POST to add a new instance. The RDF data sent to POST must describe a single instance of the appropriate class. The instance will get a new assigned ID; if it already has a URI (i.e., it is not a blank node), the old URI will be added to the data as owl:sameAs the newly assigned ID.

The POST data can contain any statements about the submitted instance, and also further statements that are *forward-reachable* from the instance. (A statement is forward-reachable from a node if its object is the node, or if its object is the subject of another statement forward-reachable from the node.) However, the submitted data must not contain statements about other instances managed by the same API; this ensures that data managed by the API is submitted through the appropriate resources. Further, if the API does not allow the wildcard property, the submitted data can only contain statements about the instance that use one of the explicitly configured properties.

Allowing arbitrary statements *forward-reachable* from the instance means that the data about the instance is not limited to simple literal values or bare URIs: for example in our use case, the contact information for a user may include structured data for a mailing address.

Instance resources support GET to retrieve all the information about an instance, DELETE to remove the instance, and POST to add new statements. The POSTed RDF data must use the correct ID assigned to the instance and only contain statements forward-reachable from it.

Property resources support GET to retrieve all the values of the given property for the given instance, DELETE to remove all the values, and POST to add new values. The POSTed RDF data must use the correct property and only contain statements forward-reachable from the instance.

Value resources support PUT to replace the current value (with the same input data structure as POST on property resources), and DELETE to remove the current value (along with the value resource graph itself).

Whenever data is deleted or replaced, the operation also removes all data in the instance's graph that is no longer forward-reachable from the instance. In effect, a DELETE on a value resource would normally remove the triple described as a reification in the resource's metadata and any further statements about the removed triple's object.

In summary, the API has a configurable four-level hierarchical structure: i) class resources manage instances; ii) instance resources are the main Linked Data resources, supporting some manipulation of the instance data; iii) property resources simplify access to and creation of property values; and iv) value resources manipulate concrete property values. The resources are interlinked and support the appropriate HTTP methods; in effect, the API gives read-write access to Linked Data in a RESTful and self-describing manner that we call *hyperdata*.

4. Implementation

We have developed a proof-of-concept configuration-driven triple-store wrapper called "Hyperdata API",[4] written in Java with the Jersey framework for building RESTful Web services, and with RDF2Go as an abstraction over a triple store, in our case OWLIM.[5]

Each of the 5 types of resources is implemented in a separate stateless Java class. Since the structure of the implementation is very simple and mirrors the structure of the hyperdata API, it can easily be extended with configurable security policies and application code triggers for validation and data handling actions. Such extensions are part of our planned future work.

Compared to the description in the preceding section, our implementation has several notable limitations: i) we use a custom vocabulary for graphs because there is no widely accepted standard (this is under discussion in the W3C[6]); ii) application data cannot include the graph vocabulary or RDFS reification vocabulary because it might conflict with our uses in the hyperdata graph links and descriptions; and iii) the implementation contains no security mechanisms apart from what is available through Web server configuration. None of these limitations harms the functionality and applicability of the API in our use case.

The resource descriptions and links in the API are expressed as RDF triples in the returned data, and our implementation also materializes them in the underlying triple store. In effect, the triple store must handle significant overhead on top of the size of the application data managed by the API. In our deployment, the overhead has not proved detrimental to the performance of the system, but we have not conducted scalability testing.

Beside the issues of scalability, the materialization of the hyperdata linking and graph description metadata also complicates potential reconfiguration. Changes in the configuration of the API, such as an addition of explicitly configured properties, or changes in short names, necessitate a rebuilding of the metadata, and possibly changes to the identifiers of graphs and instances in the data. In practice, such changes are virtually unavoidable. Our implementation currently provides no support for handling configuration changes.

Finally, our experience with the API implementation has shown that RDF/XML is not the most convenient format for API access, and should be complemented by alternative data formats, such as custom XML or JSON. The issue is likely mainly the result of the relative complexity of handling RDF graph data in client software, in contrast to handling XML or JSON tree data. In response to this concern, the method POST on property resources in our implementation also accepts two other media types beside RDF/XML: `text/uri-list` allows a simple way of adding one (or more) URIs directly as the new value(s) of the given property on the given instance, and `text/plain` is a simple way of adding one untyped literal value. The same two extra media types are also supported by PUT on value resources, meaning the replacement of the current value.

5. Related Work

Update APIs for Linked Data commonly either use a query language such as SPARQL Update [9], or they simply use HTTP methods for updating named graphs.

Works that do direct updates to named graphs rely on a division of the whole data set into resources according to the Linked Data guidelines. Among such works is [5], which supports graph collections and individual named graph re-

[4] `http://kmi.open.ac.uk/technologies/name/hyperdata`

[5] `http://www.ontotext.com/owlim`

[6] `http://www.w3.org/2011/rdf-wg/wiki/TF-Graphs`

sources. While its configuration capabilities are extensive, [5] does not seem to support subgraphs and subcollections such as the property and value resources in our approach. In effect, in terms of our use case, either the update API would not support partial update operations such as the addition or removal of individual property values within an instance graph, or it would split user records into many explicit named graphs (where the interlinking would have to be done by the client). This issue is shared by all the named-graph-oriented approaches we know.

Also in this class of approaches, the W3C is working on SPARQL Graph Store HTTP Protocol [8], which is a straightforward recommendation on how to interpret HTTP methods on named graph resources. Interestingly, it uses POST for adding data to a graph (RDF merging); in effect, the protocol has no notion of a collection of graphs. The Users API, implemented with the graph store protocol, would require the clients to assign identifiers to new instances, which, as discussed in the introduction of this paper, is undesirable.

In summary, we are aware of no related work that focuses on simple configurability and discoverability of the resulting update API, while supporting partial updates.

Among the works that employ SPARQL Update, we will mention the extension of the RDF browser Tabulator with writing capabilities in [1]. In this approach, a server may indicate that a resource can be updated with either WebDAV or SPARQL Update, and Tabulator will allow its user to edit the data received from that server, and to send the update to the server. While a public SPARQL Update endpoint may sometimes be available, we've argued in the introduction that custom APIs are often better. As far as we know, Tabulator currently cannot deal with custom update APIs, but it could be extended to recognize the graph metadata provided by our approach and to offer editing capabilities that would invoke our API.

Finally, we noted in the preceding section that RDF should be complemented in our API by alternative (and simpler) data formats, such as custom XML and JSON. This can be achieved by combining our approach with efforts such as the Linked Data API.[7] The LD-API focuses on developer-friendly APIs for reading Linked Data, particularly with support for the JSON format. Our approach mainly focuses on updates and API discoverability, and as such, it is complementary to LD-API and similar efforts.

6. Conclusions and Future Work

While publishing linked data is a heavily-researched task, making linked data stores available for writing is a so-far neglected aspect of the Web of Data. In this paper, we have presented a proposal for structuring update-oriented custom APIs over RDF stores, with particular focus on hypertext-like discoverability of update capabilities.

In its operation, our API enriches the application data with metadata about named graphs that support read/write access to the data and its subsets. We suggest the term *hyper-data* to describe such update-enabled linked data, to contrast it to simple *linked data* that is read-oriented.

As part of our future work, we plan to analyze the discoverability aspect of our approach, and to align the graph metadata vocabulary with emerging standards. One of the effects that we are aiming for is that the discoverability, supported by a standard vocabulary, will enable tools such as Tabulator to provide edit/update capabilities over a wider range of resources than they do currently.

Additionally, we plan to enhance the implementation with support for security policies, strong data validation, and application triggers.

Acknowledgments

This work was done as part of the case study activities of the European research project SOA4All (`soa4all.eu`).

References

[1] T. Berners-Lee, J. Hollenbach, Kanghao Lu, J. Presbrey, E. Prud'hommeaux, and mc schraefel. Tabulator Redux: Browsing and Writing Linked Data. In *Proceedings of the WWW 2008 Workshop on Linked Data on the Web*, Beijing, China, 2008.

[2] C. Bizer, T. Heath, and T. Berners-Lee. Linked Data - The Story So Far. *Intl Journal on Semantic Web and Information Systems (IJSWIS), Special Issue on Linked Data*, 2009.

[3] J. J. Carroll, C. Bizer, P. Hayes, and P. Stickler. Named graphs, provenance and trust. In *Proceedings of the 14th international conference on World Wide Web*, WWW '05, pages 613–622, New York, NY, USA, 2005. ACM.

[4] R. T. Fielding. *Architectural styles and the design of network-based software architectures*. PhD thesis, University of California, Irvine, 2000. Chair: Richard N. Taylor.

[5] A. Garrote and M. N. Moreno García. RESTful writable APIs for the web of Linked Data using relational storage solutions. In *Proceedings of the WWW 2011 Workshop on Linked Data on the Web*, Hyderabad, India, 2011.

[6] R2RML: RDB to RDF Mapping Language. Working Draft, W3C, March 2011. Available at `http://www.w3.org/TR/r2rml/`.

[7] P. Reddivari, T. Finin, and A. Joshi. Policy based access control for an RDF store. In *Proceedings of the Policy Management for the Web workshop*, Chiba, Japan, May 2005. In conjunction with the 14th International World Wide Web Conference.

[8] SPARQL 1.1 Graph Store HTTP Protocol. Working draft, W3C, May 2011. Available at `http://www.w3.org/TR/sparql11-http-rdf-update/`.

[9] SPARQL 1.1 Update. Working draft, W3C, May 2011. Available at `http://www.w3.org/TR/sparql11-update/`.

[7] `http://code.google.com/p/linked-data-api/`