Hyperdata: Update APIs for RDF Data Sources (Vision Paper)*

Jacek Kopecký

Knowledge Media Institute, The Open University, UK
j.kopecky@open.ac.uk

Abstract. The Linked Data effort has been focusing on how to publish open data sets on the Web, and it has had great results. However, mechanisms for updating linked data sources have been neglected in research. We propose a structure for Linked Data resources in named graphs, connected through hyperlinks and self-described with light metadata, that is a natural match for using standard HTTP methods to implement application-specific (high-level) public update APIs.

1 Vision

A major function of Web APIs is to give users a way to contribute to data sources (whether they be social networks, photo sharing sites, or anything else) through rich scripted web sites, rather than through simple web forms, and also through external (even 3rd-party) tools. Facebook API, Flickr API and so on, support interactive Web interfaces as well as mobile apps or desktop tools.

Some of the data in these apps then gets published as Linked Data, a machine-friendly representation suitable for combining with other data. Commonly, there is a technologies disconnect, though, between the Linked Data read-only view on the data source (which employs RDF and URIs), and the update APIs (with JSON or XML, and non-URI identifiers).

In this paper, we describe a vision of *hyperdata*[1] — data that is not only hyperlinked and self-describing in terms of its schema, but also self-describing on how it can be updated.

As we've discussed in [1], update access cannot practically be provided through protocols such as SPARQL Update. Indeed, public update access should be through a data-source-specific application layer that enforces consistency and security. There are several reasons for this: 1) data dependencies, where an update needs to propagate into dependent data, 2) security, where low-level access policies for RDF stores are harder to manage than if they were policies on the

* This work has been funded by the European projects SOA4All (www.soa4all.eu) and PlanetData (www.planet-data.eu).

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

level of application-specific resources, 3) data constraints and consistency validation, and guiding the users in the structure of the accepted data (for example preventing well-meaning users from using the wrong ontology by mistake), and 4) creation of identifiers, because SPARQL Update does not (as yet) 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.

With self-describing read-write hyperdata, applications that consume Linked Data could easily add update functionalities, currently generally missing from mash-ups and other Linked-Data-based apps. Further, data browsers such as Tabulator, which currently supports SPARQL Update and WebDAV [2], would be able to provide edit/update capabilities over a wider range of resources.

Linked Data may be compared to Web 1.0: the latter was mostly read-only documents, and the former is mostly read-only RDF views on some databases. The Web of Data should be more like Web 2.0, with many sites allowing (and even relying on) contributions from their users. With hyperdata, that will be possible, because hyperdata is not only linked to other data, but also to its update APIs.

In our vision, the optimal update API should fit well with the structure of Linked Data (including the application of the principle of *following your nose* to discovering update capabilities), it should rely as much as possible on the methods of HTTP (adhering to REST's *uniform interface* constraint), and it should easily accommodate application-specific update authorization, validation and propagation logic.

2 Use Case Description

Our hyperdata approach was developed within a use case of the European research project SOA4All. The use case is an application called "Offers4All" that allows diverse companies to advertise offers to subscribers of the service (more detail in [1]). These offers might be "last-minute" travel deals, predefined campaign offers of restaurants, and so on. The Offers4All application 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.

The application is backed by an RDF database that stores information about offer providers, their offers, and the users registered to receive the offers. Users can specify what offer categories they are interested in, and they can also choose to "like" some offers which allows social-networking-style recommendations to be used to increase the uptake of offers. Naturally, users can also specify various contact details, such as an email address and a mobile phone number.

For read and update access, the database is façaded by a custom API, whose functionalities can be seen as the following types of operations:

- listUsers() returns a list of the known users
- getUser(id) returns user data

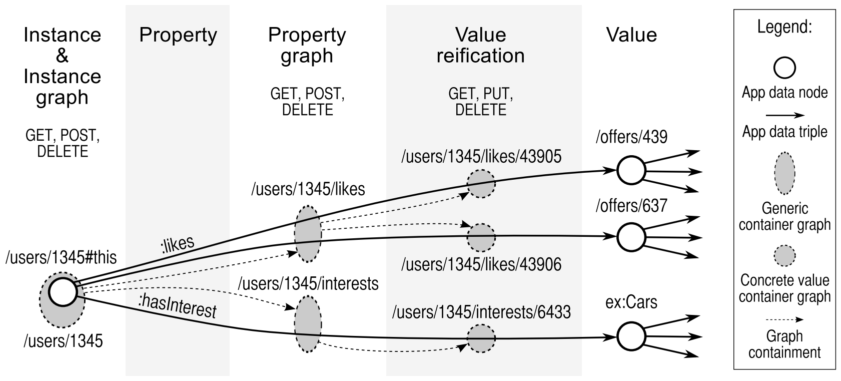

Fig. 1. Hyperdata structure of the API

- addUser(data) creates a new user record
- getUserInterests(id) returns the offer categories of interest to the user
- addUserInterest(id, uri) adds to the user's list of interests
- deleteUserInterest(interest-id) removes one from the user's list of interests
- deleteAllUserInterests(id) clears the list of interests
- *and so on for the various properties of the various objects in the database, incl. "likes" and contact information*

The granularity of these operations corresponds to the intended uses of the system: these are the types of operations that clients of such a database want to perform, and they are a good input for analyzing access control.

Following the principles of Linked Data and REST (useful even if the data is not published openly), the database is split into a number of resources: a single container *users resource*, multiple *user resources* (one per known user), container *user interests resources* (one per known user), and concrete *interest value resources* (one per a stated interest of a user), etc.

In a read-only data source, this fine level of granularity could be seen as too much, as retrieving all the data about a user does not present much overhead even if the client is only interested in the user's interests. However, with all these resources in place, update operations naturally map to HTTP methods.

Figure 1 shows the RDF graph of a user who likes two specific offers and has interest in one category. For brevity, the figure doesn't show the container resource for users. Along with the actual data triples, the figure also displays the self-description aspects, discussed in the next section.

3 Hyperdata Approach

The API in our use case consists of the following generic four types of resources: 1) containers of instances (users, offers etc.), 2) the instances themselves, 3) containers of property values, 4) concrete property values. Listing 1 illustrates the

```
1   </users/1345#this>  a  uc:User ;             uc: likes   </offers/439>, </offers/637> .
2
3   </users/1345>  a  g:Graph ;                  g: defines  </users/1345#this> ;
4      g: contains  </users/1345/likes> ;        g: isContainedIn  </users> .
5
6   </users/1345/likes>  a  g:Graph ;
7      g: contains  </users/1345/likes/43905>, </users/1345/likes/43906> ;
8      g: defines  [   a  rdf:Statement ;
9          rdf: subject  </users/1345#this> ;   rdf: predicate  uc: likes   ;   rdf: object  [  ]
10     ]  .
11
12  </users/1345/likes/43905>  a  g:Graph ;
13     g: defines  [   a  rdf:Statement ;
14         rdf: subject  </users/1345#this> ;   rdf: predicate  uc: likes   ;   rdf: object  </offers/439>
15     ]  .
```

Listing 1. Example graph description triples (truncated)

self-description metadata and hyperlinks, also shown in Figure 1; it starts on line 1 with (a subset of) the actual data about the particular user.

Line 3 indicates the graph that 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 4 links the instance graph with one of the property graphs (`/users/1345/likes`), and with the high-level class graph.

Lines 6–10 describe the property graph: it contains concrete value graphs, and a reified triple pattern (lines 8–10) that indicates that the graph includes statements of the form `/users/1345#this` uc:likes *something* (note the blank node as object). The triple pattern is meant to indicate what kind of data can be POSTed to the property resource to add a value, and what subset of the data about the user can be expected when GETting the property resource.

For adding a property value, the client can POST several kinds of data: an RDF graph, a list of URIs, or a literal value. Primarily, the POSTed data can be an RDF graph that contains a triple `/users/1345#this` uc:likes *something*, and any triples about the *something*. To prevent adding arbitrary statements, all the triples in the submitted data must be about the instance (the particular user) or about the values of other triples in the graph — we use the phrase that all the triples are "forward-reachable" from the instance. Further, the data must not contain any triples about other instances managed by this particular hyperdata store (for example about an offer) because submissions of such triples must go through that instance's update API.

Alternatively to submitting RDF data, there are media types that give the client a simpler way for submitting a new property value: if the client wants to add a property value that is some resource (e.g. `/offers/439#this`), the URI can be submitted as `text/uri-list` and it will be added as a direct value. And finally, a new literal value (not appropriate for uc:likes but possible for other properties) can be submitted simply as `text/plain`.

The listing concludes on lines 12–15 with a description of a concrete value graph. The client can use PUT or DELETE here to update or remove a particular statement. PUT here has the same options for payload formats as POST for submitting new property values above — it can be an RDF graph, a URI list or a plain literal value.

The metadata uses a few very simple concepts to communicate much information: a Graph is a resource that besides GET may also accept update and delete requests (actually available methods can be discovered with HTTP OPTIONS).

The meaning of updates depends on the contents of the graph, described through reified statements. The reified statement may indicate a concrete triple like on line 14 (meaning that it represents a specific value, to be updated with PUT or removed with DELETE), or it may use blank nodes to indicate a collection (accepting POST with new items). The listing indicates a collection of property values for uc:likes on line 9. A reified statement of the form *something* rdf:type uc:User would be shown on the user container graph to indicate that it contains instances of the given ontology class, and that's what can be POSTed.

4 Prototype Implementation

We have developed a proof-of-concept triple-store wrapper (also described in [1]) that uses very simple configuration to realize the hyperdata API. Configured with a set of "classes of interest", whose instances the API manages, and "properties of interest" on those classes, the wrapper implements the necessary read and update resources to cover the structure of the hyperdata graph. The code generates and maintains all the self-description metadata as data is submitted and updated.

Currently, the metadata is stored in the underlying triple store along with the application data. If this overhead should become a performance or scalability issue, the metadata could be stored in a separate triple store (so that it does not affect reading and querying performance), or generated at runtime as the data is being accessed. On-the-fly generation of the metadata would decouple the data from the current configuration of the update API; however, it could itself present performance overhead. We have not evaluated performance and scalability issues in the scope of the use case.

The prototype does not address the issue of concurrent updates from multiple clients, beyond serializing the addition of instances; however, the HTTP specification defines the "entity tag" mechanism that supports conditional updates, performed only if the resource has not changed since the client has last seen it.

5 Conclusion

The Web included update capabilities from the start—the first browser[2] was also an editor—but still Web 1.0 was mostly read-only. A significant boom came with the advent of the Web 2.0 with its attitude that anybody on the Web can—and should be allowed to—contribute. The Web of Data so far remains on the Web 1.0 level where contributions to it mostly happen outside it. Hyperdata

[2] `http://www.w3.org/People/Berners-Lee/WorldWideWeb.html`

APIs can bring update capabilities to the Web of Data, and make it more like Web 2.0. After all, Web 2.0 gave us Wikipedia, the heart of Linked Data.

Our prototype implementation is very basic, but the structure of the code and its configuration seems amenable to extensions towards access control policies, data validation and custom update propagation/processing (including logging and versioning). Also in future work, we would like to develop a client access library for hyperdata, and to extend Tabulator to support it. Client support will let us better evaluate the communication and client-side-processing overhead cost of all the metadata.

The hyperdata vision relies on the assumption that self-description of update capabilities can help clients adapt to changes in evolving hyperdata APIs, as hyperlinking allows clients to discover locations of new data, and to adapt to changed locations of expected data sources. This assumption needs to be evaluated on further case studies.

References

1. Kopecký, J., Pedrinaci, C., Duke, A.: RESTful Write-oriented API for Hyperdata in Custom RDF Knowledge Bases. In: Proceedings of the International Conference on Next Generation Web Service Practices (NWeSP), Salamanca, Spain (2011)
2. Berners-Lee, T., Hollenbach, J., Kanghao Lu, Presbrey, J., Prud'hommeaux, E., 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)