Keep Your Triples Together: Modeling a RESTtful, Layered Linked Data Store

Niko Popitsch
University of Vienna, Liebiggasse 4/3-4, A 1010 Vienna, Austria
niko.popitsch@univie.ac.at

ABSTRACT

Linked data (LD) is an increasingly important way for publishing structured data on the Web. Applications that represent their (meta) data as LD benefit from the flexibility, the semantic elaborateness and the simplicity of this approach. There are, however, several issues with such a solution in real-world scenarios, e.g., (i) mechanisms for writing LD such as SPARQL UPDATE are still under development, (ii) the generic, flexible nature of the used RDF data model makes the development of data- and user interfaces complex, and, (iii) descriptions of LD resources are treated as "atomic" entities, no mechanisms for accessing or delivering partial resource descriptions via HTTP are available.

In this work, we introduce a model for a LD store that enforces structuring of its data in two dimensions: named graphs are used to separate logical *aspects* of the contained data into "horizontal" *layers*. Extraction functions are used to decompose these graphs into "vertical" *records*, collections of sub-graphs "centered" around some resource of interest. This model can be implemented using established technologies and formats from (Semantic) Web research and we discuss a prototypical implementation in this paper. Our prototype implements a REST interface for read/write access to the stored records. Further, clients can access partial records based on HTTP header settings.

The higher structuredness of the data in such a store, when compared to arbitrary RDF graphs, brings practical advantages for implementing data or user interfaces. The strong data compartmentalization can be exploited in many ways, e.g., for more fine-grained access control or for concurrent write access to various layers of the store.

Categories and Subject Descriptors

H.4 [**Information Systems Applications**]: Miscellaneous

General Terms

Theory, Design

1. INTRODUCTION

The goal of the *linked data (LD)* initiative is to build a global network of interlinked data that can be processed by both, human and machine actors. LD *resources*, basically anything that can be named, such as people, genes, products or movies, are identified by HTTP URIs that can be dereferenced (accessed). One of the core LD ideas is that dereferencing such a URI leads to useful *representations* (descriptions) of the respective resource that usually serve a particular purpose, such as human or machine consumption. Technically spoken, LD services often return either HTML or RDF resource representations (descriptions). The decision what representation is returned depends on HTTP header settings: sending an HTTP GET request to the URI `http://example.org/X` would in one case return an HTML representation, in another (e.g., when the *HTTP Accept* header is set to "text/rdf+n3") an RDF document. This enables clients to negotiate with the server in what content format they want to access a resource's description.

LD provides some advantageous features for using it as a (meta) data access layer for common (Web) applications. In this paper, we discuss benefits and shortcomings of such an approach with the help of the following two scenarios:

Scenario 1: A consumer-to-consumer (C2C) Web application.

Imagine a C2C Web application that enables users to directly sell items to other consumers. Such a service would, at its core, provide descriptions of users (profiles) and items including user names, email addresses or item prices. Users may contribute item profiles but also links to related items or other Web resources. Further, the service might include some reputation mechanism based on user votings as common in current Web 2.0 services.

Scenario 2: A file metadata store.

As a second scenario, consider a local store for file-related metadata (such as textual descriptions, keywords, semantic tags and links to other files or external Web resources) that contains metadata contributed by various applications (photo management software, word processors, link mining algorithms). As a concrete example for such an application, consider some music management software that extracts ID3 tags from MP3 files and stores them in this metadata store. Such a store[1] could then expose its metadata as LD in order

[1]We are actually implementing such a metadata store based on the model discussed in this paper.

to share it with remote consumers but also as a common access interface in a local, heterogeneous environment itself. Users or applications could, for example, query this store (locally or remotely) to learn what music is available in this system.

Representing the core resources (user profiles, items, files) of these two exemplary scenarios as LD has several advantages: First, the used data and schema modeling languages (RDF, OWL, etc.) allow the flexible yet exact expression of these data and their semantics. In scenario 1, for example, users may unambiguously specify what items they wish to sell by using respective semantic vocabularies. Second, reusable semantic vocabularies in combination with a common data model (RDF) greatly facilitate the integration of these data with related data sets. E.g., a service might search the metadata storage in scenario 2 for all descriptions of human users (e.g., document authors, music composers, people depicted on a photo) by considering resources described with common vocabularies for describing people or their user profiles (e.g., FOAF). It may then retrieve additional metadata about these users from other data sets. Third, LD provides a uniform way to address and access stored resource descriptions. Resources (items, user profiles, files) would be identified by globally unique URIs, which enables external actors to reference them unambiguously and thus also enables them to express additional knowledge about them. Further, resource descriptions are accessible via standard HTTP requests, no specialized transmission protocol/layer is required. A standardized query language (SPARQL) for querying LD is available.

However, the relatively young LD approach still suffers from some major shortcomings. Above all, a standardized approach for *writing* LD itself is currently missing, which may be considered a major issue [1, 8]. Current LD (e.g., in the LOD cloud) is usually created by some mapping process that converts "conventional" data sources (e.g., RDBs) to RDF, as in the case of DBpedia. Such data cannot be updated by sending RDF graphs to the LD endpoint, but rather by changing the underlying data source, in the case of DBpedia by modifying data stored in a database using the Wikipedia Web interface. However, this forces applications to switch between two "worlds" for writing and reading data which adds undesirable complexity.

Further, descriptions of LD resources are currently treated as atomic entities by current LD servers. When a content-format was negotiated successfully, the whole resource description is delivered by the server in the respective format. In some situations it would, however, be beneficial if a server could deliver only *parts* of a (possibly large) resource description for reasons of restricted access or to reduce processing, I/O and transmission costs.

In this paper, we discuss a novel model for a storage layer directly on top of LD that enables CRUD (create, read, update, delete) operations on sub-graphs of LD descriptions. Named graphs are used to model multi-layered descriptions of things (*records*), the various layers of a record represent arbitrary *partial aspects* of item descriptions. Such records are treated as units that can be **c**reated, **r**ead, **u**pdated and **d**eleted in a storage that may be realized using, for instance, a common triple-store implementation. Serialized records are read/written via an HTTP REST interface [6], record layers are realized as named graphs which enables efficient access and clean compartmentalization of these data. CRUD operations on records can be restricted to (groups of) layers that can be specified using an HTTP 1.1 standard compliant selection mechanism based on *range* headers.

We start the remainder of this paper by formally introducing our model, followed by an informal discussion of its advantages and shortcomings. We continue by discussing our prototypical implementation and our approach in the context of related work.

2. FORMAL MODEL

Our model disallows the use of blank nodes and strongly discourages the use of RDF reification in the used RDF graphs. Both are also discouraged for LD publishing in general [3]. The main reasons for this are that (i) bnodes are not referenceable and make graph merging much more complex and (ii) that reification results in increased storage requirements and makes querying more difficult. We start this section by introducing some derivations of well-known sets from the RDF specifications by excluding bnodes:

DEFINITION 2.1 (INITIAL DEFINITIONS). *Let $\mathbb{U}$ be the set of URI references, $\mathbb{B}$ the set of RDF blank nodes and $\mathbb{L}$ the set of literals as defined in [13]. Further let $\mathbb{T} = (\mathbb{U} \cup \mathbb{B}) \times \mathbb{U} \times (\mathbb{U} \cup \mathbb{B} \cup \mathbb{L})$ be the set of RDF triples and $\mathbb{T}' = \mathbb{U} \times \mathbb{U} \times (\mathbb{U} \cup \mathbb{L})$ the set of triples without blank nodes.*
Further let $\mathcal{P}(X)$ be the powerset of an arbitrary set X. Then $\mathbb{G} = \mathcal{P}(\mathbb{T})$ is the set of all RDF graphs and $\mathbb{G}' = \mathcal{P}(\mathbb{T}')$ is the set of all RDF graphs without bnodes. Further, analogously to the definitions in [4], we define a named graph without bnodes *as a pair $(u, g) \in (\mathbb{U} \times \mathbb{G}')$.*

Having introduced these basic sets, we continue by describing our concepts of *facets*, RDF graphs "centered" around a subject resource. This subject resource is identified in a sub-graph by being the only one having an asserted OWL class `y2:RecordFacet`, that stems from our vocabulary introduced in Section 4. A second type of facets, so-called *extension facets*, are defined analogously using a second OWL class `y2:RecordExtensionFacet`. Facet graphs may not contain other facet subjects but apart from that they are not further restricted as shown in Definition 2.2.

DEFINITION 2.2 (FACETS). *We call $f^x_{core} \subseteq \mathbb{G}'$ a core facet of a resource identified by the URI $x \in \mathbb{U}$ iff $y, z \in \mathbb{U}$ and $\exists!(y, \textit{rdf:type}, \textit{y2:RecordFacet}) \in f_x^{core}$ with $y = x$ and $\nexists(z, \textit{rdf:type}, \textit{y2:RecordExtensionFacet}) \in f^x_{core}$. We call $f^x_{ext} \subseteq \mathbb{G}'$ an extension facet of x iff $y, z \in \mathbb{U}$ and $\exists!(y, \textit{rdf:type}, \textit{y2:RecordExtensionFacet}) \in f^x_{ext}$ with $y = x$ and $\nexists(z, \textit{rdf:type}, \textit{y2:RecordFacet}) \in f^x_{ext}$.*

A *slice* of a subject x is a named graph created by merging[2] one core facet of x with an arbitrary number of extension facets (Definition 2.3).

DEFINITION 2.3 (SLICES). *Let $\oplus$ be an RDF merge operator as defined in [11]. We call the named graph $(n, S^x) \subseteq (\mathbb{U} \times \mathbb{G}')$ a slice with name n of the core facet f^x_{core} and a (possibly empty) set of extension facets $\{f^{y_1}_{ext}, \dots, f^{y_m}_{ext}\}$ iff $S^x = f^x_{core} \oplus_{i=1}^{m} (f^{y_i}_{ext})$.*

A *layer* is a named graph created by merging multiple slices sharing a common name (Definition 2.4).

[2]A merge of a set of RDF graphs that do not share blank nodes is simply the set union of the contained triples.

DEFINITION 2.4 (LAYER). *We call the named graph* $(n, L) \subseteq (\mathbb{U} \times \mathbb{G}')$ *a layer with name n of the (possibly empty) set of slices* $\{(n, S_1^{x_1}), \ldots, (n, S_m^{x_m})\}$ *iff* $L = \oplus_{i=1}^{m}(S_i^{x_i})$.

A *store* is a set of layers including one mandatory *core layer* named by a special URI reference $n_{core} \in \mathbb{U}$ that is a configuration parameter of this store (Definition 2.5).

DEFINITION 2.5 (STORE). *A store* $\mathbb{S}^{n_{core}}$ *is a set of layers* $\{(n_{core}, L_{core}), (n_1, L_1), \ldots, (n_m, L_m)\}$ *with distinct names* n_i*.* (n_{core}, L_{core}) *is called the core layer of the store.*

Finally, a *record* is an arbitrary set of slices sharing a common subject x. It thus represents multiple *partial aspects* of this subject resource. In order to integrate records into a store, its slices should be named according to the layers of the store. In particular, such records have to contain a *core slice* named by n_{core}.

DEFINITION 2.6 (RECORD). *A record* R^x *is a set of slices* $\{(n_{core}, S_{core}^x), (n_1, S_1^x), \ldots, (n_m, S_m^x)\}$ *with distinct names* n_i *and a common subject* x*.* (n_{core}, S_{core}^x) *is called the core slice of the record.*

Integration of such a record into a store is then achieved by simply merging all slice-graphs with the respective (equally named) layer-graphs in the store.

3. INFORMAL DISCUSSION

At its core, a *store* in our model is a set of named graphs with some associated metadata. Such a store could thus be realized by a single RDF data set [15], where the default graph corresponds to the core layer of the store. Most current triple store implementations support this concept and provide a SPARQL service for querying such data sets. Note, however, that the simplest useful store compliant with our model contains only one single (core) layer and can be represented by one single RDF graph. Further note, that is is also easily possible to realize multiple stores in one single triple store, as long as distinct layers have distinct names. In such a scenario, it would even be possible that different stores share common layers, which would, however, require some kind of concurrency handling to preserve data integrity.

The named graphs in a data set are *layers* that should be used to group logical aspects of the stored data. For scenario 1, one could, for example, foresee a layer for storing the user profiles, another one for storing item descriptions and a third layer for storing annotations (e.g., reviews) regarding both. The benefit of this logical separation is that it is now straightforward to deliver or modify these data aspects separately or attach differing access policies to them. If, for example, one would like to make certain parts of an item profile readable only to a particular group of (registered) users, the item descriptions themselves could be split up into two actual layers: One would be publicly readable while the other would require a valid login. The same is obviously applicable for restricting write access.

3.1 Two-dimensional compartmentalization

The proposed data compartmentalization has further practical advantages. For example, a system could allow concurrent access to these layers which would, for example, clearly be beneficial in scenario 2 where multiple applications write exclusively to "their" layers.

While layers structure a store "horizontally", *records* group resource-related descriptions from all layers, thus being a kind of orthogonal ("vertical") structuring concept. A (complete) record describes all available *partial aspects* of a resource and may thus be conceived as the current "view" on a resource in the respective linked data set.[3] However, records are also layered (layers of a record are called *slices* to make them distinguishable). This enables the retrieval and manipulation of partial records that represent only certain aspects of a resource.

3.2 Faceted Records

Records are logical entities in our store to which CRUD operations are applied. This does not mean, that our store cannot be queried like any other LD set, e.g., using SPARQL queries that fulfill certain triple patterns. Records are further assembled by so-called *facets*, sub-graphs of a particular slice of a record. A single, mandatory *core facet* (per slice) describes the data that is exclusively related to the particular subject of the record: when the record is removed from the store, these triples are removed in any case. A slice may, however, further contain an arbitrary number of extension facets. These facets may be used for descriptions that are *shared* between multiple records.

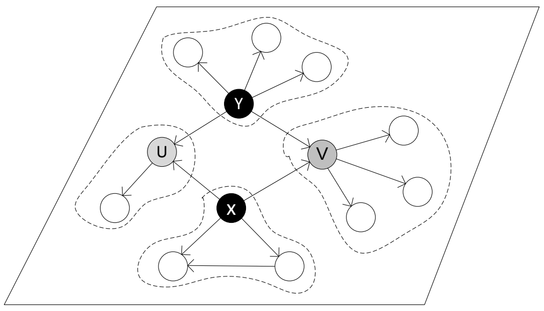

Figure 1: A store layer containing 2 records (X, Y) which have 2 core facets (denoted by the dashed lines around X and Y) and 2 extension facets (dashed lines around U and V) each. In the depicted situation, the reference count of these extensions is 2 each. Deleting one of the two records would remove only its core facet and reduce the reference counts to one. Consecutively removing the other record would then remove this record's core facet as well as both extension facets.

Consider, for example, the mentioned ID3 extractor in scenario 2. When representing extracted ID3 tags as RDF, this component could create a core facet for describing the MP3 file, using literals to describe directly related metadata such as the title of this piece of music. For representing an MP3's artist, however, an own resource with associated metadata (e.g., including `owl:sameAs` links to respective DBpedia resources) could be created and linked via an object-property to the MP3 resource. This artist sub-graph represents data that is potentially shared between multiple records as many MP3s may share the same artist(s).

[3]Note, that we actually do call a set of *layers* a "*view*" in our implementation.

In this example, the sub-graph describing the MP3 would be the *core facet* while the artist sub-graph(s) would be extension facets. Figure 1 depicts such a situation. The resources X and Y represent two MP3 files that are associated with two artist resources (U, V) each. The difference between core and extension facets becomes clear when considering what happens when the respective records are deleted from the store. As core facets describe exclusive metadata about the record, they are deleted in any case. Extension facets, however, may only be deleted if they are not *referenced* by any other record. Thus a deletion of record X in Figure 1 leaves its extension records untouched. A subsequent deletion of Y, however, would trigger the deletion of U and V as well.

3.3 Facet extractors

The core issue regarding our model is *how* (core and extension) facets are extracted from a given RDF graph (i.e., a store layer or a record slice). As described above, facets are "centered" around some resource. More formally, we are looking for a function
$\epsilon : \mathbb{U} \times \mathbb{G}' \longrightarrow \mathcal{P}(\mathbb{G}')$ that extracts a set of *facets* describing a record-resource (identified by a URI) from a given RDF graph without bnodes. Several methods for the extraction of a resource-centered sub-graph have been proposed:

Simple extractors.
A simple way to extract a useful sub-graph is to consider only triples that have the resource as subject. Stickler [17] and others call this kind of representation "asymmetric", whereas a "symmetric" description would also include inbound arcs (i.e., triples that have the center-resource at the object position). Other proposed methods, such as *Concise Bounded Descriptions (CBD)* [17], *RDF molecules* [5] and *Minimum Self-contained Graphs (MSG)* [19] are basically reduced to these simple asymmetric or symmetric "shells" when considering graphs without bnodes.

Pattern-based methods.
More complex graphs may be extracted using SPARQL CONSTRUCT queries where the graph patterns of the selection clause match the graph template used for constructing the result graph. Note, that formalizing such queries requires prior knowledge about the "structure" of the extracted sub-graphs in order to formulate the proper query graph pattern. Further note that such an extraction function would require the execution of one SPARQL query per extraction process which might be costly. Fresnel Selector Language (FSL) [14] expressions are another method for sub-graph extraction based on predefined patterns.

Explicit methods.
A further possibility is to use named graphs: one could reuse the URI of the center-resource also as the name of a named graph that contains all triples describing the respective facet. This explicit method for defining a sub-graph does not restrict what triples are contained in a facet graph and does not require a special function for extracting them. This approach is, however, not applicable in our scenario for the following reasons: (i) we already use named graphs for representing layers. As facets are sub-graphs of layers this scenario would require a concept for embedding named graphs into named graphs. We are currently not aware of an appropriate storage infrastructure for this; (ii) it would further require a format for serializing such hierarchically organized named graphs which is currently not available; (iii) it would lead to a large number of named graphs (one per facet), something that is not efficiently handled by current triple store implementations; (iv) further, compartmentalizing the layer graph into named sub-graphs would make querying more complex and probably slow.

Custom algorithms.
Finally, custom algorithms can be used to extract sub-graphs of arbitrary complexity. Such individual implementations have the benefit that they might also resort to external data and contextual information for this task and might be specialized for a particular domain.

Whatever sub-graph extraction function is chosen, the fundamental requirement is that it enables lossless graph decomposition and merging. This means that adding, removing or updating records in a store should not interfere with other records (except for updating shared descriptions, i.e., extension facets). One way to achieve this is to avoid that facets overlap within a layer, i.e., that they do not share triples. This is, for example, achieved when the simple *asymmetric shell* mentioned above is used (which, however, allows only the representation of name/value pairs and links). The *symmetric shell* would obviously not show this property as each extracted facet would contain all triples having the center-resource as subject or object.

As distinct layers represent varying aspects of a data set, it is likely that they will require differing models for expressing their data. Allowing different extractors for distinct layers is thus a logical consequence from this. Consider, for example, scenario 2: here, the different actors will use very different metadata models that probably differ in their complexity. While one application might get along with simple name/value pairs (and could use a simple yet effective *asymmetric shell* extractor for the layer it writes to), another, such as the above-mentioned MP3 handler, would probably require a custom-tailored extraction function that extracts, e.g., the extended "artist" facets from an RDF graph by resorting to a certain semantic vocabulary (e.g., a property from this vocabulary could be used to connect a MP3 resource to its artist resources).

Reference counting.
In our model, each layer in a store is assigned exactly one *record extractor* that implements the above-mentioned extraction function ϵ as well as a "reference counting" function $\rho : \mathbb{G}' \times (\mathbb{U} \times \mathbb{G}') \longrightarrow \mathbb{N}^0$. This function accepts an extension facet and a layer as parameters and returns the number of core facets contained in the layer that reference this extension facet. This reference count (*refcount*) is then used by the store to decide whether an extension facet is removed together with its core facet or not (Figure 1). A simple method for this is to count the number of triples that directly link core facet subjects with a particular extension facet subject. In case of the above-mentioned ID3 example, a *refcount* could be determined by counting only the triples with the predicate that was used to connect core and extension facets. Another possibility is to explicitly represent the reference count in a *literal* attached to the extension subject. In this case, however, respective functionality is required that keeps this value up-to-date.

Record index.

Although the data in our model is strongly compartmentalized into layers, an index of all records contained in a store can easily be retrieved. Reconsider that each record contains a mandatory core facet which has to include a triple that asserts the RDF type of the record's core facet using a special OWL class. This core record slice will then be merged with the core layer of the store and thus a list of all contained records is easily retrievable from this graph using a trivial SPARQL query. Such an *index* is useful in many scenarios, e.g., for search engines or other indexing applications.

4. VOCABULARY AND IMPLEMENTATION

We have implemented a prototypical LD store as described in this paper. The central classes and interfaces of our implementation are depicted in Figure 2. A store starts an embedded Web server that implements a REST interface realizing the CRUD operations that can be applied to the store. We have further developed a simple OWL light vocabulary (the "Y2 model vocabulary"), an excerpt of which is depicted in Figure 3, for expressing the core concepts of our model in RDF.

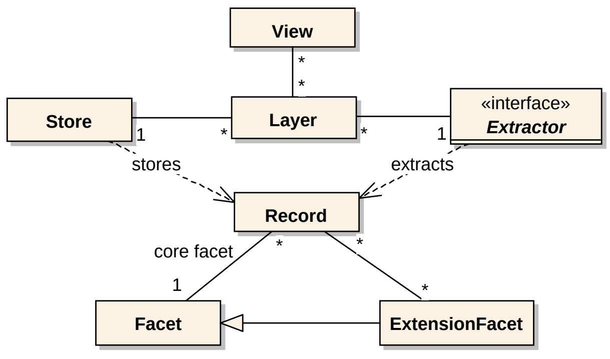

Figure 2: Main classes and interfaces of our prototypical implementation.

A store is consequently represented as a resource identified by a dereferenceable HTTP URI. Dereferencing this URI (with content-type "text/rdf+n3") leads to an RDF description of this store that includes references to all its contained *layers*. Dereferencing a layer URI leads to a description containing: (i) a `dataLocation` URI that can be used to link, for example, to a local file storing this layer's RDF model; (ii) a `readonly` property that, when set to `true`, disallows write access to this layer via the REST interface; (iii) a `recordExtractor` property that is used to link to a resource describing an extractor. As we used Java for our prototype implementation, we foresaw an `implementingClass` property that stores the FQN of the Java class that actually implements an extractor. This class, that complies with a simple extractor interface (cf. Figure 2), is loaded and instantiated dynamically by our store implementation. The extractor instance is then used to extract a collection of facets (including exactly one core facet) from a given layer RDF graph. Our implementation comes with some basic but easily extensible extractor implementations that decompose a layer into non-overlapping facets as described above.

Two OWL classes are used to represent `RecordFacets` and `RecordExtensionFacets`. It is notable that resources contained in the merged RDF graph, that is returned when dereferencing a record URI, may be rdf-typed by both classes as a resource may be the center resource of a core facet in one layer and the center resource of an extension facet in another one.

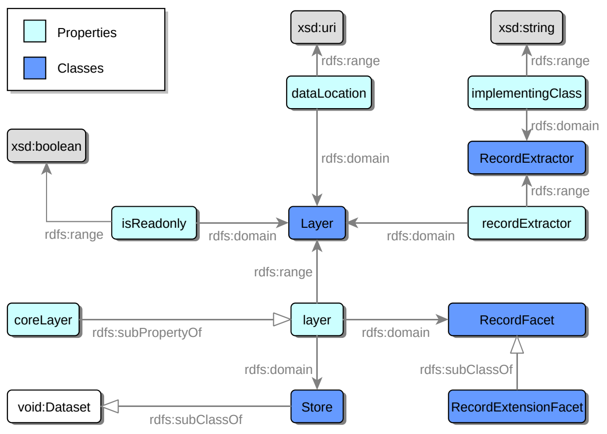

Figure 3: Y2 model vocabulary.

4.1 REST interface

Our store implements a REST interface for executing CRUD operations on its contained records. Reading records from a store can thus be done using standard HTTP GET requests. The store responds to such requests by (i) extracting all according record facets from each layer using the respective extractors, (ii) merging the respective record slices to one RDF graph[4] and (iii) serving this graph in serialized form via HTTP[5].

As with most other LD services clients can negotiate the content type of the returned representation (HTML or RDF) with our server. As our model is layered, however, we added a second kind of "negotiation" possibility. Clients may request that the server assembles the returned description only from a sub-set (called a *view*) of layers. This is done by passing a list of layer names (i.e., URIs) in the HTTP *range* headers of the request.

The HTTP range mechanism [7] allows the access of partial resource representations. The main intention of this mechanism is the reduction of unnecessary data transfer. HTTP 1.1 enables clients to select byte-ranges of resource representations (e.g., 500-999/1234 selects the second 500 bytes of a 1234 bytes long representation) and a range-header-aware server answers such requests with a message containing an HTTP 206 (Partial Content) response code. The HTTP range mechanism explicitly foresees the possibility to define custom "range units" that can be used to specify a range[6]. We therefore propose this mechanism for selecting what layers a delivered record description should

[4]It is is also possible to access a decomposed version of a record serialized in TriG via content negotiation.

[5]Note, that our model does not specify what description is returned for non-core facet resources. Our implementation handles this case by returning their *asymmetric shell*.

[6]Note that RFC 2616 (HTTP 1.1) is a bit vague regarding whether custom range units may be used for the Content-Range header. Although the running text states that such custom ranges can be used, they are not foreseen in the EBNF of section 14.16 Content-Range. We understand that

contain. The following command, for example, requests the annotation layer representation of resource X in N3 notation:

```
curl -H "Accept: text/rdf+n3"
  -H "Range: layers=<http://mystore.com/y2/conf/anno>"
  -X GET http://mystore.com/y2/r/X
```

Record creation.

Records can be created by sending RDF descriptions to their HTTP URIs using HTTP PUT or HTTP POST requests[7]. Our current implementation accepts only record descriptions serialized as TriG [2], a simple text format for serializing named graphs. The server de-serializes the record from a TriG document in the following way: (i) for each named graph in the TriG document, a corresponding layer (i.e., a layer with the same name) is looked-up in the store; (ii) the record extractor of these layers is used to extract the record facets from the named graph; (iii) the extracted facets are merged to create the record slices.

A record might be *invalid* with respect to such a create operation, e.g., if not all named graphs have corresponding store-layers, if no core slice was found or if some layers are not writable. In such a case, the server answers with a respective HTTP status code (e.g., *416 Requested Range Not Satisfiable* or *403 Forbidden*). Otherwise, the record is added to the store by merging its slice graphs with the respective layer graphs. This results in the integration of layered records into a layered LD store as depicted in Figure 4.

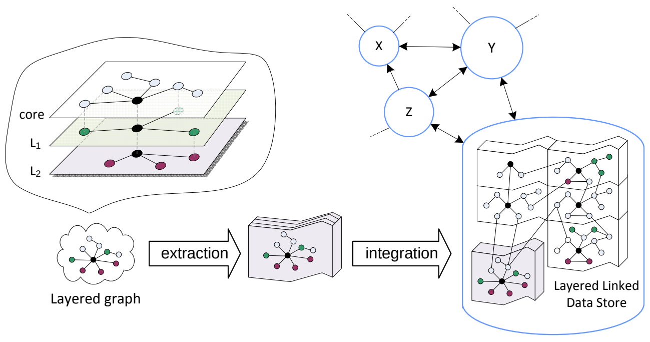

Figure 4: A record with three layers (*core*, L_1, L_2) is extracted and integrated into a layered LD store that is linked to other LD sources. Record subject resources are depicted as black circles.

Record deletion and update.

Analogously, the server accepts HTTP DELETE requests to the respective resource URIs. The server extracts the respective record from its layers and deletes these triples. Note that the above-mentioned range negotiation is used for the whole REST interface. Thus, replacing the HTTP method by "DELETE" in the above-mentioned *curl* request results in deleting only the annotation layer of resource X, leaving all other layers untouched.

Sending HTTP PUT or POST requests to the URIs of already existing records results in an update operation. In this case, the old record is deleted from the storage and the new one is parsed from the request body and created in the store. Note, that this may also result in an update of shared extension facets.

this issue is work in progress and designed our solution with regard to the latest IETF draft published in [7].

[7]Note, that this obviously works only if the respective URIs are part of this stores' dereferenceable URI space.

Bulk creations/deletions.

In order to support bulk creation and deletion operations (useful, e.g., for store synchronization, etc.), there is also the possibility to send a collection of records to a special resource of a store: its *RDF sink*. This resource is backed by a servlet that accepts HTTP POST and DELETE requests. Collections of records are parsed from the TriG document embedded in the HTTP request and the respective storage operations (create, update or delete) are executed on each contained record.

5. RELATED WORK AND DISCUSSION

Obviously, consuming and updating LD sets via SPARQL in general and the currently discussed SPARQL 1.1 Update [9] extension in particular are related work to ours. Our proposed model should, however, not be considered in any way as some kind of replacement for SPARQL but rather as a complementing strategy for storing and accessing LD that is pre-structured in some form. The record extraction functions that determine the "silhouettes" of the various record layers in our model enforce a common structure of the stored graphs respectively their facet sub-graphs. This may be beneficial when consuming these data as less-generic data handling interfaces (e.g., import interfaces, GUIs) are required. On the downside, this also restricts the modeling possibilities for the LD in such stores. Although this restriction is alleviated by the possibility to provide multiple layers, each containing differently structured aspects of the data, our approach cannot be a general but rather a domain-specific solution for LD. Generally spoken, the main difference between our solution and SPARQL is, that the latter acts on the triple level of RDF graphs while we introduce logical, decomposable entities and act on the descriptions of these entities (i.e., on layered records).

The form of these entities is determined by extractors on the server side which is comparable to the SPARQL DESCRIBE semantics. In both approaches, the client requires no prior knowledge about the structure of the returned description. As our records are decomposable into layers, we additionally enable filtering of such descriptions by layer and thus also the retrieval of partial descriptions of a LD resource. SPARQL also provides methods for filtering triples from a result set, even on a per-named-graph basis (namely, graph patterns and FILTER constraints) which could, arguably, also be used for the retrieval of partial records. The difference is, however, that in this case the data consumer explicitly decides how these partial records look like while in our approach the *writer(s)* actually determine the contents of a record slice. This is also reflected in the ways how such partial descriptions are requested: In SPARQL, the respective filter and selection statements would be part of the query itself (and thus part of the HTTP message payload), while our solution relies on negotiation via HTTP headers. This allows it to handle the availability or accessibility of partial records on the transport protocol layer which might be beneficial in certain situations, e.g., for caching purposes.

Writing to our store is also handled differently from SPARQL UPDATE. Although clients may send any RDF data to our service, only properly extracted sub-graphs will be further processed. The parts of the transmitted descriptions that do not match the structure predefined by the given extractor functions are filtered out. This is different in SPARQL UPDATE where there are no restrictions for the client to specify what would be updated in a *graph store*.

Nevertheless, our store is in the end represented by a single RDF dataset that can be accessed via SPARQL like any other graph store. Actually, we do believe that a combination of both access methods could be beneficial in many scenarios where both levels of granularity (triple and record) are useful. Reconsidering scenario 1, SPARQL could be used to query a store as usual (e.g., "return all URIs of items that cost less than x USD"), accessing these URIs via the REST interface could then be used to exchange whole logical entities (e.g., the respective item descriptions, as foreseen by the store implementer), and SPARQL UPDATE could be used to update the store in situations where client-control over the data representation is useful (e.g, users could annotate their profiles with own FOAF graphs that are sent to a particular graph layer using SPARQL UPDATE requests).

Tabulator.

A generic user interface that uses SPARQL to build a kind of readable and writable "data wiki" is Tabulator [1]. *Tabulator* is not restricted to a single data source and treats LD in a generic fashion on the triple level, i.e., no logical containers comparable to our records are available. The decision where modified triples are written to depends on their provenance; new item-related triples are added to the documents that mentioned them. Writable RDF documents are flagged by HTTP headers as such. Technically, WebDAV and SPARQL UPDATE are used to write RDF data.

Tabulator does not introduce any actual restrictions to the structure of the edited RDF graphs. This genericity is the main strength but also weakness of this approach in our opinion. Any data source can be filled with any triples which basically dissolves this compartmentalization mechanism as one cannot predict what data is found in what source anymore. This, however, raises questions of practicability. Besides the difficulties to develop appropriate read and write interfaces, the existing "hard" boundaries between data sources are usually exploited in many ways, e.g., for deciding on the provenance and reliability of the data or for access restrictions. It is further exploited to actually navigate to the data one wants to consume: certain data sources are "known" to contain certain kinds of data. For this reason one does not have to search the Web every time but may bookmark Web resources as consuming them at a later point in time is likely to return the same or comparable information.

Social rules may practically restrict what data is written in what "structure" to what data sources with Tabulator-like software and this might work quite well, e.g., for a data wiki. Other applications, however, might require some explicit and controllable mechanism for data structuring. Not only the structure of these data but also their compartmentalization (respectively, the possibility to group data sharing some common feature) is important in real-world situations. One practically highly relevant application for this is access control. Both our introductory scenarios obviously require some restrictions of read and write access to the stored resource descriptions. E.g., in scenario 1, parts of the resource descriptions, such as user names or item labels, would be publicly available while others, such as email addresses, would be accessible only to registered users. Further, while possibly all registered users may create new items, write access to user profiles or item descriptions would be restricted to the "owners" of these resources. In scenario 2, where many applications write into one common metadata store, it should be possible to manipulate and access the contributed metadata independently to avoid that the various applications mutually overwrite these data.

Finally, the generality of the Tabulator approach raises user interface design questions: How to build user-friendly GUIs for entering new/editing existing RDF data? One possible approach to this is shown by RDFauthor [18], an approach for editing RDF graphs embedded in RDFa annotated HTML pages. RDFauthor also uses SPARQL UPDATE for propagating changes to a respective SPARQL endpoint. It consists basically of a JavaScript API that parses RDFa annotations from HTML pages and automatically builds a user interface using appropriate input widgets for the various properties. This user interface hides much of the complexity of the underlying RDF data model and is more user-friendly than the Tabulator. Nevertheless, such generic user interfaces can never reach the usability of specialized GUIs as they are possible when one knows about the basic structure of the edited data, especially when patterns more-complex than simple name/value pairs are used.

6. CONCLUSIONS

In this paper we proposed a model for a structured and layered read/writable LD store. We introduced logical compartmentalization of resource descriptions in two dimensions: "horizontal" layers and "vertical" records. Records are layered RDF graphs, each layer representing some logical aspect of the description of a record's subject. This subject is a LD resource: dereferencing its HTTP URI leads to a representation of this resource (the record) available in different formats (N3, TriG, XHTML). Partial records can be requested by providing lists of layer names in HTTP range headers which leads to partial descriptions of only these aspects of a resource. An index of all records contained in a store, usable by indexing applications, such as semantic search engines, is automatically maintained.

Our proposal relies only on well established technologies and formats from (Semantic) Web research, such as named graphs, TriG, HTTP and REST, and can be implemented using standard triple store and Web server technology. We have developed a prototypical Java implementation of this model based on the Jena Tuple Database (TDB)[8] and the Jetty Web server[9]. A demo of our implementation is available at `http://purl.org/y2/`.

Our proposed solution does not touch the complex topics of security, privacy, access control and related issues. Although these are clearly missing features for using our implementation in real-world scenarios as described in this paper, we are confident that existing research on these topic may be integrated with our solution (cf., e.g., [12, 10, 16]).

We have discussed advantages and disadvantages of our model in comparison to generic methods that directly operate on the triple level based on SPARQL (UPDATE). Our solution is no replacement for such generic solutions. It operates on a different "level" of abstraction and may be used as a complementing strategy for manipulating LD.

The main intention of our model is to introduce some structuring into LD resource representations while not restricting the flexibility and expressiveness of the graph-based RDF model too much. These structures are determined by extraction functions that are pluggable into a LD store. Extractors may define the space of allowed structures for a record description based on all levels of the *Semantic Web Stack* and even based on external (e.g., contextual) data. They must, however, ensure (together with the way the data is stored) that lossless graph decomposition is possible to enable CRUD operations on RDF subgraphs. Developing non-trivial, efficient extraction functions that fulfill this property is the hardest thing when implementing our model for real-world data sources. However, when this task succeeds, one is benefited with a store whose data structuredness lies somewhere between the fixed structures of object-oriented or frame-based systems and the extreme flexible structures achievable by arbitrary RDF graphs.

7. REFERENCES

[1] T. Berners-Lee, J. Hollenbach, K. Lu, J. Presbrey, and M. Schraefel. Tabulator redux: Browsing and writing linked data. In *WWW 2008 Workshop: Linked Data on the Web (LDOW2008)*, 2008.

[2] C. Bizer and R. Cyganiak. The trig syntax. Technical report, FU Berlin, 7 2007.

[3] C. Bizer, R. Cyganiak, and T. Heath. How to publish linked data on the web. `http://www4.wiwiss.fu-berlin.de/bizer/pub/LinkedDataTutorial/`, July 2008. Accessed april 2011.

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

[5] L. Ding, T. Finin, Y. Peng, P. P. da Silva, and D. L. McGuinness. Tracking rdf graph provenance using rdf molecules. In *Proceedings of the 4th International Semantic Web Conference*, November 2005.

[6] R. T. Fielding. *REST: Architectural Styles and the Design of Network-based Software Architectures.* Doctoral dissertation, University of California, 2000.

[7] R. Fielding (ed.), J. Gettys, J. Mogul, H. Frystyk, L. Masinter, P. Leach, T. Berners-Lee, Y. Lafon (ed.), and

[8] `http://openjena.org/wiki/TDB`

[9] `http://jetty.codehaus.org/jetty/`

J. Reschke (ed.). HTTP/1.1, part 5: Range Requests and Partial Responses draft-ietf-httpbis-p5-range-14, April 2011.

[8] A. Garrote and M. Moreno. Restful writable apis for the web of linked data using relational storage solutions. In *WWW 2011 Workshop: Linked Data on the Web (LDOW2011)*, 2011.

[9] P. Gearon (ed.), A. Passant (ed.), and A. Polleres (ed.). Sparql 1.1 update, w3c working draft 12 may 2011. Technical report, W3C, 2011.

[10] O. Hartig and J. Zhao. Publishing and consuming provenance metadata on the web of linked data. In *IPAW*, pages 78–90, 2010.

[11] P. Hayes (ed.) and B. McBride (ed.). Rdf semantics. Technical report, W3C Recommendation 10 February 2004, 2004.

[12] J. Hollenbach, J. Presbrey, and T. Berners-Lee. Using rdf metadata to enable access control on the social semantic web. In *Proceedings of the Workshop on Collaborative Construction, Management and Linking of Structured Knowledge (CK2009)*, 2009.

[13] G. Klyne (ed.), J. J. Carroll (ed.), and B. McBride (ed.). Resource description framework (rdf): Concepts and abstract syntax. Technical report, W3C, 2004.

[14] E. Pietriga (ed.). Fresnel selector language for rdf (fsl). Technical report, W3C, 2005.

[15] E. Prud'hommeaux (ed.) and A. Seaborne (ed.). SPARQL query language for rdf. w3c recommendation 15 january 2008. Technical report, W3C, 2008.

[16] O. Sacco and A. Passant. A privacy preference ontology (ppo) for linked data. In *Proceedings of the Linked Data on the Web Workshop (LDOW2011), CEUR-WS Workshop at 20th International World Wide Web Conference*, 2011.

[17] P. Stickler. Cbd - concise bounded description. `http://www.w3.org/Submission/CBD/`, June 2005. Accessed april 2011.

[18] S. Tramp, N. Heino, S. Auer, and P. Frischmuth. Making the semantic data web easily writeable with rdfauthor. In *ESWC (2)*, pages 436–440, 2010.

[19] G. Tummarello, C. Morbidoni, P. Puliti, and F. Piazza. Signing individual fragments of an rdf graph. In *Special interest tracks and posters of the 14th international conference on World Wide Web*, WWW '05, pages 1020–1021, New York, NY, USA, 2005. ACM.