RESTful writable APIs for the web of Linked Data using relational storage solutions

Antonio Garrote
Universidad de Salamanca, Spain
agarrote@usal.es

María N. Moreno García
Universidad de Salamanca, Spain
mnmg@usal.es

ABSTRACT

Linked Data is rapidly becoming an important mechanism to expose structured data in the web. The ability to inter-link data sets from different providers using standard description vocabularies and the same data model, opens new possibilities in the way these data can be used. Despite of its growth, Linked Data principles have not found yet widespread application in the design of data APIs for web applications. The lack of a write support for linked data repositories, the barrier imposed by the required technological change and the immature state of client and server semantic infrastructure are some of the main causes for this lack of adoption.
This paper introduces a possible alternative for building writable web APIs according to Linked Data principles, using the already deployed technology stack present in most web applications. We propose the use of R2RML to lift relational data into the RDF model as well as to map SQL manipulation data queries into SPARQL update queries. Additionally a RDF vocabulary describing a RESTful interface for the mapped data that can be easily consumed from web clients is proposed. The combination of both aspects allows web developers to offer a familiar web API compatible with linked data APIs that can be deployed along with the already existing interface.

Categories and Subject Descriptors

H.3.5 [**Online Information Services**]: Web-based services; D.2.12 [**Interoperability**]: Data mapping

General Terms

Languages, Design

1. INTRODUCTION

The Linked Data initiative (LD) [5], [2], [1] is making the semantic web a reality for the mainstream web developer. Despite of its success, and the increasing volume of Linked Data available in the web, the incipient use of LD in many web applications is restricted to a static repository of data that is queried or crawled as needed. The use of LD as the main API for today's web 2.0 applications is not a viable alternative yet. A quick search in a popular repository of web APIs [1] shows only 34 RDF enabled APIs in contrast with more than 1000 JSON interfaces. One of the main causes of this situation is that the web of data, in its present form, only supports a read interface. Other causes range from the complexity introduced by the RDF data model and the lack of support for semantic technologies like SPARQL in common web application clients like web browsers and smartphone SDKs, to the insufficient performance and scalability concerns of key semantic web infrastructure components like triple stores. [24]

Most of Web 2.0 application's APIs are built on a technological stack that typically comprises the following elements:

- a persistence layer backed by a relational database management system (RDBMS) or a mixture of a RDBMS and some kind of *NoSQL* technology.
- a mapping framework exposing the relational model as a set of more or less compliant RESTful web services.
- Plain JSON objects as the most prominent serialization format for application data.
- A security and access control framework based on mechanisms like API keys and the OAuth protocol [11].

To make possible the transition from this model for building web APIs to a LD enabled one, technological options that make easy the building of LD APIs must be available for the web developer. One important step in this direction is the *R2RML: RDB to RDF Mapping Language* (R2RML) [22] W3C's recommendation working draft. R2RML consists of a generic vocabulary that can be used to map a relational schema into RDF triples.

In this paper we will use the R2RML specification as a starting point to build a mechanism capable of automating the transformation of the relational persistence layer in a Web 2.0 API, into an API compliant with the LD principles, that can be offered as an alternative by a service provider.

The main contributions of this paper can be summarized as follows:

- It describes a generic translation of SPARQL 1.1 Update [23] queries into SQL queries over a relational schema, provided a R2RML mapping for that schema.

Copyright is held by the author/owner(s).
LDOW2011, March 29, 2011, Hyderabad, India..

[1] http://www.programmableweb.com/

- It introduces a common RESTful [10] interface for LD enabled web APIs that transforms HTTP request into SPARQL queries.

- We collect a light vocabulary for the declarative description of APIs suitable for being automated as a software library.

Different approaches for the translation of SPARQL Update queries into SQL data manipulation operations have recently been proposed. ONTOACCESS [12] introduces a translation mechanism based in its own mapping language R3M. The transformation algorithm maps whole tables and columns into classes and properties in a certain ontology. This approach simplifies the translation of SPARQL queries since there is a direct relationship between any valid triple and a single table in the database schema. Another recent translation mechanism is described in [18] as an extension to the D2RQ [2] mapping language called D2RQ++ introducing support for mapping blank nodes and dealing with database constraints in the insertion and deletion of triples. D2RQ++ also proposes the use of an external RDF triple store to deal with triples in conflict with the database schema. Main differences between ONTOACCESS, D2RQ++ and our approach are the consequence of the different characteristics of the mapping languages used. These differences also impose constraints on the kind of RDF graphs that can be stored in the underlying relational system. R2RML allows to map *variable* columns in the relational schema that can store any kind of URI or literal directly into the database. This makes possible to describe mappings capable of storing any possible triple in a RDF graph relational database without requiring an auxiliary triple store without requiring an external storage solution as proposed by the authors of DRQ++. The drawback of these flexibility is that situations where a triple of quad pattern can be inserted in more than one mapped must be addressed in the construction of the translation mechanism. R2RML also includes support for named graphs in the mapping language. We add support for this feature in our proposed solution that will be used extensively to expose subsets of the RDF triples stored in the database as RESTful resources.

This paper is organized in two main sections. In the first one, the SPARQL to SQL translation using R2RML is described. In the second part, a RESTful API for read-write LD APIs, a vocabulary for the description of these APIs and the operational details of the transformation of HTTP requests into SPARQL queries according to this API are introduced. The combination of both parts can be used to build LD APIs using present day relational technologies.

A prototype implementation of the described SPARQL mapping mechanism for R2RML as well as an executable implementation of the API are currently being developed. They can be found at [3].

2. SPARQL 1.1 OPERATIONS OVER RELATIONAL DATA

W3C's R2RML proposal consists of a generic vocabulary that can be used to map a certain relational schema into RDF triples. A formal description of a simplified R2RML mapping is shown in listing 1 as an EBNF grammar.

```
<R2RMLMapping> ::= { <TableMapping> } ;

<TableMapping> ::= ( table:String ,
                     <subject:TermMapping>,
                     <graph:TermMapping>,
                     <propertyObj:{ TripleMapping }>) ;

<TripleMapping>  ::= { (<property:TermMapping>,
                        <column:TermMapping>,
                         [rr:datatype],
                         [rr:laguage]) } ;

<TermMapping> ::= <VariableMapper> | <ConstantMapper> ;

<ConstantMapper> ::=  rr:property | rr:constantValue
                    | rr:columnGraphIRI ;

<VariableMapper> ::=  rr:propertyColumn | rr:column
                    | rr:columnGraph ;
```

Listing 1: R2RML mapping

This model formalizes a R2RML mapping as a collection of *TableMappings* for each RDBMS mapped table. Each *TableMapping* describes how the data stored in that table can be transformed into *RDF quads* with subject, predicate, object and an associated named graph. The components of a quad are generated using a *TermMapping* defined in the *TableMapping*. Each *TermMapping* can be constant or variable. Constants *TermMappings* point to an URI or RDF literal for the value of the quad component while variable *TermMappings* refer to a column in the relational schema where the value for the quad component can be retrieved.

W3C's SPARQL 1.1 Update proposal [23] describes a standard query language to retrieve and modify RDF graphs with a syntax similar to that of SQL. The main construct of SPARQL queries are quad patterns that can be matched against triples stored in RDF graphs. Listing 2 formalizes the notion of triple pattern stored in a named graph as a *QuadPattern* with variable and constant components.

```
<QuadPattern> ::= (<subject:Term>,
                   <property:Term>,
                   <object:Term>,
                   <graph:Term>) ;

<Term> ::= <VariableTerm> | <ConstantTerm>;

<ConstantTerm> ::= URI | RDF Literal ;

<VariableTerm> ::= {?a, $a, ?b, $b...} ;
```

Listing 2: Quad patterns

R2RML describes a transformation from the relational model into the RDF model. In order to use R2RML mappings to build a generic transformation mechanism for SPARQL 1.1 Update operations over a RDF graph into SQL operations over an equivalent relational model, we need a way to find the inverse transformation for a R2RML mapping as shown in figure 1.

Listing 1 describes such an algorithm that expresses the inverse transformation as a set of *QuadMatchers* generated from a set of R2RML *TripleMappers*.

The output of the function *buildQuadMatchers* consists of a collection of *QuadMatchers*: tuples describing a pattern that can be used to map a RDF compatible *QuadPattern* to a RDBMS relation. The listing 2 describes a procedure to check if a *QuadPattern* is compatible with a *QuadMatcher*. The constant *NULL* value is used to identify matchers and patterns in the default graph.

To allow the manipulation of triples through a RESTful HTTP interface, it is necessary to find SPARQL 1.1 queries compatible with the semantics of the HTTP uniform interface methods. *SELECT*, *INSERT DATA*, and

[2]http://www4.wiwiss.fu-berlin.de/bizer/d2rq/
[3]https://github.com/antoniogarrote/clj-r2rml

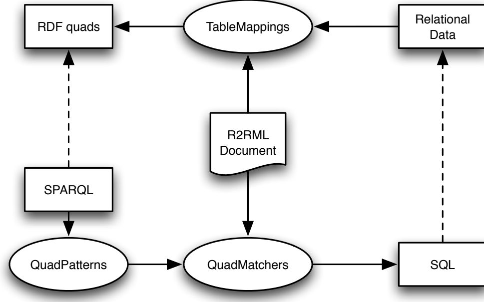

Figure 1: Different transformations encoded in a R2RML mapping

Algorithm 1 Building of quad matchers for a R2RML mapping

```
Function: buildQuadMatchers
Input: mapping : R2RMLMapping
Output: Collection of QuadMatchers
Begin
  quadMatchers ← {}
  for tableMapping in mapping do
    table ← tableMapping.table
    subjetTerm ← tableMapping.subject
    for tripleMapping in tableMapping.tripleMapping do
      property ← tripleMapping.property
      object ← tripleMapping.object
      graph ← tripleMapping.graph
      quadMatchers ∪ (table, graph, subject, property, object)
    end for
  end for
  return quadMatchers
```

Algorithm 2 Procedure to check if a quad pattern and a quad matcher built from a R2RML mapping are compatible

```
Function: compatible?
Input: p : quadPattern, m : QuadMatcher
Output: True or False
Begin
  compatible ← True
  for patternComp in p, matcherComp in m do
    if var?(patternComp) ∨ var?(matcherComp) then
      compatible ∧ True
    else
      compatible ∧ (patternComp = matcherComp)
    end if
  end for
  return compatible
```

DELETE SPARQL 1.1 operations will be used to map the *GET*, *POST*, *PUT* and *DELETE* operations of the HTTP methods.

2.1 SPARQL SELECT operation

The transformation of SPARQL *SELECT* queries into efficient SQL queries has been an active research topic in the last years. A relational algebra for SPARQL has been proposed [8], and different translation mechanisms have been defined [7], [9], [14].

Our approach is based on [7]. This translation schema uses two functions α and β, capable of retrieving the table and the column where the component of any RDF triple pattern is stored in the database. The transformation mechanism imposes the restriction that this table must be unique for every triple pattern in a SPARQL *SELECT* query. Nevertheless, this assumption cannot be taken for granted for a generic R2RML mapping.

In listing 3 a couple of procedures describing an algorithm for translating an arbitrary *QuadPattern* into a SQL *SELECT* query using a collection of *QuadMatchers* are introduced. This algorithm takes as input a triple pattern for a certain RDF graph in a SPARQL select query, parsed as a *QuadPattern* and a R2RML mapping transformed into a collection of *QuadMatchers*, and outputs a SQL *SELECT* query composed of the UNION of different SQL *SELECT* sub-queries for each *QuadMatcher* compatible with the *QuadPattern*. If no *QuadMatcher* is compatible with the pattern, the algorithm returns a failure. Projections and conditions for SQL queries can be built checking the variable components of the pattern and the matcher.

Algorithm 3 Composition of a SELECT query for a quad pattern and a set of quad matchers

```
Function: select
Input: quad : QuadPattern, matchers : QuadMatcher
Output: SQL query or FAIL
Begin
  compatibleMatchers ← mapCompatibleQuadMatchers(quad, matchers)
  query ← FAIL
  if compatibleMatchers ≠ ∅ then
    subselects ← mapSubselects(quad, compatibleMatchers)
    query ← join("UNION", subselects)
  end if
  return query

Function: mapSubselects
Input: quad : QuadPattern, matchers : QuadMatcher
Output: SQL subquery
Begin
  subselects ← {}
  for matcher in matchers do
    table ← matcher.table
    projections ← genProjections(quad, matcher)
    conditions ← genConditions(quad, matcher)
    sql = "SELECT DISTINCT" + join(",", projections) + "FROM" +
    table
    if conditions ≠ ∅ then
      sql + "WHERE" + join(",", conditions)
    end if
    subselects ∪ sql
  end for
  return subselects
```

The output of this algorithm can be inserted in [7] algorithm to build SQL queries for complex SPARQL queries involving *AND*, *OPT*, *UNION* and *FILTER* constructs.

2.2 SPARQL 1.1 Update INSERT DATA operation

SPARQL 1.1 Update *INSERT* operations have the form shown in listing 3. *INSERT* queries can be directly transformed into a collection of *QuadPatterns* where all the components have a constant value.

```
INSERT DATA {
```

```
GRAPH <graph_uri>
{ triples }
}
```

Listing 3: SPARQL 1.1 Update INSERT operation

Insertion of *QuadPatterns* for an *INSERT* query and R2RML mapping, requires to find the collection of *QuadMatchers* compatible with each *QuadPattern*. Many different *QuadMatchers* can be compatible with each pattern, making possible the insertion of the triple in different tables of the data base. It is also possible that there exists in the database columns with *NULL* values that can be updated instead of inserting new rows.

Listing 4 shows an algorithm that can be used to insert triples into a data base, creating or updating table rows according to the information of a R2RML mapping.

Algorithm 4 Generation of an insertion query for a quad pattern and a set of quad matchers

```
Function: insert
Input: quads : QuadPattern, matchers : QuadMatcher
Output: SQL DML query or FAIL
Begin
  sortedQuads ← sortBySubject(quads)
  contexts ← initialContext(quads)
  for quad in sortedQuads do
    compMatchers ← mapCompatibleQuadMatchers(quad, matchers)
    if compMatchers = ∅ then
      return FAIL
    end if
    contexts′ ← nextLevel(quad, compMatchers, contexts)
    contexts ← minSchemaContexts(contexts′)
  end for
  sql ← generateInsertionSQL(first(contexts))
  return sql
```

The algorithm works building a tree of possible ways of inserting the triples in the database, and selecting the terminal node that minimizes a cost metric.

The function *initialContexts* queries the database and retrieves all the existing rows with the same subject as the quads to be inserted. The resulting compatible rows are stored as the only context structure in the *contexts* list, becoming the root node of a contexts tree. Then main loop of the algorithm retrieves all the *QuadMatchers* compatible with the next quad to be inserted and generates a new level of contexts. Each generated context in the list of contexts represents a possible way of inserting the quad in the database consistent with a compatible *QuadMatcher*. The function *nextLevel* tries to update the existing rows to be inserted with the values for the columns resulting of applying a compatible *QuadMatcher* to the *QuadPattern* to insert. If no row of the context can be updated, for example, because the existing row already inserts or updates the column with a different value, a new row marked to be inserted, is added to the context containing the new columns.

The function *minSchemaContexts* trims the contexts that do not minimize the cost function shown in listing 5. This metric uses the number of rows and columns inserted in the database as factors. It grants that triples are inserted into tables where rows with the same subject are stored in the same row if possible. This feature is important if the RDBMS tables are also used by an object-relational mapping framework storing each object in a single row of the database.

Finally the function *generateInsertionSQL* just translates the first of the equivalent minimum contexts into a series of *INSERT* or *UPDATE* operations over the RDBMS.

The figure 2 shows graphically the insertion of two quads in the default graph using the R2RML mapping in the listing 4 for two relations *test1(id,a,b)* and *test2(s,p,o)*.

Algorithm 5 Insertion cost metric

```
Function: insertionCost
Input: context : SchemaUpdateContext
Output: cost : integer
Begin
  columns ← 0
  for rowMatch in context do
    columns ← columns + count(rowMatch.columns)
  end for
  return ((1 + count(context.rows)) * columns)
```

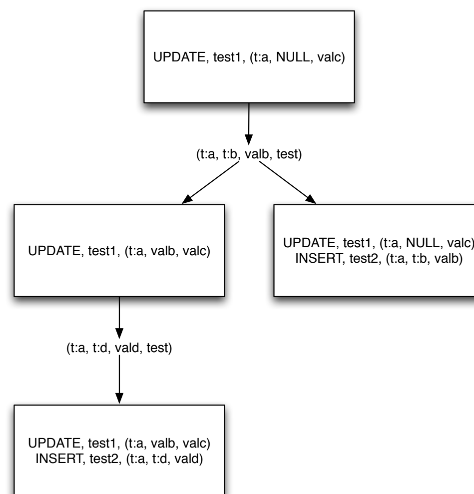

Figure 2: Insertion of two quads

```
@prefix lda: <http://restful_linked_data_api.org#> .

_:maping1 rr:table "test1" ;
          rr:subjectMap [ rr:column "id" ] ;
          rr:propertyObjectMap [ rr:property <test:a> ;
                                 rr:column "a" ;
                                 rr:columnGraphIRI <test> ] ;
          rr:propertyObjectMap [ rr:property <test:b> ;
                                 rr:column "b" ;
                                 rr:columnGraphIRI <test> ] .

_:maping2 rr:table "test2" ;
          rr:subjectMap [ rr:column "s" ] ;
          rr:propertyObjectMap [ rr:propertyColumn "p" ;
                                 rr:column "o" ;
                                 rr:columnGraphIRI <test> ] .
```

Listing 4: Example R2RML mapping

2.3 SPARQL DELETE operation

SPARQL 1.1 Update *DELETE* operations have the form shown in listing 5. *DELETE* operations can also be directly transformed into a set of SQL operations. As a first step, the *WHERE* clause of the query must be executed as a *SELECT* operation, and the retrieved bindings applied to the *modify template* of the *DELETE* query so they can be transformed into a collection of *QuadPatterns*.

```
DELETE {
 GRAPH <graph_uri>
 { .. modify template .. }
}
WHERE
{
 GRAPH <graph_uri>
 { .. pattern .. }
}
```

Listing 5: SPARQL 1.1 Update INSERT operation

Algorithm 6 removes the compatible quads stored in a RDBMS according to a SPARQL 1.1 Update DELETE op-

eration and a R2RML mapping. The algorithm updates the columns for RDF properties and objects with *NULL* value instead of removing the whole triple, since the subject column can be shared by other triples stored in the same row. After removing all the triples, the function *removeEmptyRows* delete all the rows in the tables of the mapping where all the property and object columns have a *NULL* value.

Algorithm 6 Composition of a query to remove a quad pattern matching a set of quad matchers

```
Function: delete
Input: quads : {QuadPattern}, matchers : {QuadMatcher}
Output: SQL DELETE DML query
Begin
  sql ← ""
  for quad in quads do
    compatibleMatchers ← mapCompatibleQuadMatchers(quad, matchers)
    for matcher in compatibleMatchers do
      columnMatches ← genColumnMatches(quad, matcher)
      if count(columnMatches) > 0 then
        table ← matcher.table
        sql ← sql + "UPDATE" + nameSQL(table) + "SET"
        conds = {}
        values = {}
        for columnMatch in columnMatches do
          if isNotSubject?(columnMatch) then
            values ∪ nameSQL(columnMatch.column) + " = NULL"
          end if
          conds ∪ nameSQL(columnMatch.column) + " = "
            +nameSQL(columnMatch.value)
        end for
        sql ∪ join(values, ",") + "WHERE" + join(conds, "AND") + ";"
      end if
    end for
  end for
  return sql
```

2.4 Limitations and additional extensions

The described mechanism has omitted important features of RDF that can be added to the simplified version presented in this section. Some of the features include:

- RDF data types or literals
- Mapping of URIs to table index values.
- Blank nodes

RDF data types can be easily added to the mapping algorithm using the translation of SQL data types into XML Schema data types provided by the *A Direct Mapping of Relational Data to RDF* [15] W3C's working draft. Using this translation, the function that checks the compatibility between typed *QuadPatterns* and typed *QuadMatchers* can also check if the type for the column matches the type of the pattern component. Both structures can also be extended adding support for an optional language in the positions of subject, predicate or object and adding additional checks in the compatibility function

The problem of mapping URIs in triples to their final value in a relational table can be solved extending R2RML with a new predicate *rr:URIMappingExpression* similar to *rr:inverseExpression*. This new property will introduce a transformation for the value of the URI into the suitable SQL value, that can be used into SPARQL operations.

Support of blank nodes has been proved to be problematic when the column storing the blank node is marked as *AUTO_INCREMENT*. In these cases, the solution found involves the generation of a pseudo-unique integer value that is assigned to the newly created blank node identifier.

Another important limitation of the proposed mapping is that it can only be used with updatable tables. The original read-only scope of R2RML makes easy to translate tuple values into URIs using a logical table consisting of a complex SQL query as the starting point of the mapping. This is not possible when supporting modifications in the tables and mechanisms like the proposed *rr:URIMappingExpression* property must be used to transform URIs into the final values to be stored in table rows.

3. DECLARATIVE MAPPING OF RDF GRAPHS AS RESTFUL SERVICES

The previous section of this document has described a mechanism to manage RDF graphs stored in a relational database using SPARQL queries built using a R2RML mapping. In this section we will introduce a RDFS vocabulary describing a RESTful API, aligned with the principles of the LD initiative, that can be used to access a RDF graph by any client software supporting the HTTP protocol.

In the same way that R2RML describes a mapping from the relational model into the RDF model, the proposed API vocabulary describes a mapping from a set of HTTP requests to the SPARQL query language.

The described interface must meet certain requirements:

- It must be a viable alternative for present day web applications, exposing new capabilities in a familiar way for web developers not used to semantic web technologies.
- It must be compliant with REST architectural principles [10] as well as with Linked Data recommendations [5].
- It must support not only the retrieval of data but also the creation, update and destruction of resources.
- It must reuse existing work and vocabularies in the area of Linked Data APIs where possible.

3.1 API alternatives for the web of LD

There are currently different proposals to describe interfaces allowing access to RDF graphs for web clients. A possible categorization of these APIs could be:

- **SPARQL endpoints**
- **RDF over HTTP**
- **Entity Attribute Value (EAV) mappings over HTTP**

SPARQL endpoints offer a very general interface to access RDF graphs using the expressivity power of the SPARQL query language. The *SPARQL protocol for RDF* W3C recommendation standardized a SOAP based protocol to access such an endpoint. Despite of its genericity, SPARQL endpoints and the SPARQL protocol for RDF violates some of the constraints imposed by the Linked Data and REST architectural styles, for example, the non dereferenceability of the URIs stored in the graph. The use of SPARQL endpoints also imposes important requirements on the clients accessing the service. They must deal with the building of

SPARQL queries, the creation of the correct SOAP requests and the parsing of RDF/XML responses.

The exchange of RDF graphs using the HTTP protocol according to the REST architecture is another alternative to build a HTTP interface for RDF graphs. In this case, the central abstraction is the named graph (NG) [6]. The RDF dataset stored in a service is partitioned into several graphs identified by an URI and this URI is exposed as a HTTP accessible resource. HTTP methods are mapped to SPARQL 1.1/Update operations creating, editing, retrieving and destroying NGs. the *SPARQL 1.1 Uniform HTTP Protocol for Managing RDF graphs* W3C's recommendation [17] or the Pubby frontend [20] are examples of such interfaces. This kind of APIs are compatible with REST and Linked Data principles but support less expressive queries over the RDF graph. From the client point of view, using this kind of APIs requires the support of the RDF data model and some of the RDF serialization formats provided by the service (Turtle, N3, RDF/XML, JSON).

A simpler kind of HTTP interfaces for RDF graphs consists of EAV mappings for RDF graphs that are exchanged using the HTTP protocol. In this kind of APIs, the server hides the RDF data model mapping sets of triples sharing the same subject as pairs of key-values objects. These objects are encoded in HTTP requests parameters and they are returned in HTTP responses, using serialization formats like JSON objects with plain attribute names. Examples of this kind of APIs are the Linked Data API Proposal (LD-API) [3] and RDF backed versions of the Open Data Protocol API [19]. EAV interfaces are compliant with REST principles and they have a familiar interface for most web developers. They usually introduce ad-hoc mechanisms to deal with practical web development issues like the *pagination* of resources in collections. On the other hand, they present some problems from the Linked Data perspective like the use of hidden URIs in objects attributes, what could prevent the effective linking among services.

The API and the description vocabulary we are going to introduce are based on the exchange of RDF graphs using the HTTP protocol according to the *SPARQL 1.1 Uniform HTTP Protocol for Managing RDF graphs* W3C's draft recommendation. Using this protocol, information resources in the service can be described as triples stored in a named graph that can be exposed through the service API. This level of granularity is equivalent to the one that can be found in most RESTful APIs [25]. Additionally, we will introduce some convenient features commonly found in EAVs APIs, like the LD-API proposal, to facilitate the use of the API by restricted web clients.

3.2 Declaration of linked resources

Listing 10 contains a RDF graph encoded using the Turtle syntax that describes a couple of resources: a collection of blogs and each blog. These resources expose using the HTTP protocol the relational data stored in the *BLOGS* SQL table accessed using the R2RML mapping shown in the same listing.

```
@prefix testblog: <http://example.org/blog#> .
@prefix lda: <http://restful_linked_data_api.org#> .
@prefix api: <http://purl.org/linked-data/api/vocab#> .
@prefix rr: <http://www.w3.org/ns/r2rml#> .
@prefix sioct: <http://rdfs.org/sioc/types#> .
@prefix dc: <http://purl.org/dc/elements/1.1/> .
@prefix dcterms: <http://purl.org/dc/terms/> .
@prefix xsd: <http://www.w3.org/2001/XMLSchema#>.

testblog:
  a lda:API ;
  lda:exposes testblog:blogs, testblog:blog .

testblog:blogs
  a lda:Resource ;
  api:uriTemplate "http://testblog.org/lodapi/blogs" ;
  lda:endpoint
  [ a lda:R2RMLSparqlEndpoint ;
    lda:has_r2rml_mapping testblog:bogsMapping ;
    lda:has_r2rml_graph rr:columnGraphIRI ];
  lda:has_operation lda:GET, lda:POST ;
  lda:named_graph_creation_mechanism testblog:blogsMappingUriMinter

testblog:blogsMappingUriMinter
  a lda:NamedGraphCreationMechanism ;
  lda:uri_template "http://testblog.org/lodapi/blogs/{id}" ;
  lda:mapped_uri_parts
  [ lda:mapped_component_value "id" ;
    lda:uri_generator lda:UniqueIdInt ] .

testblog:blog
  a lda:Resource ;
  api:uriTemplate "http://testblog.org/lodapi/blogs/{id}" ;
  lda:endpoint
  [ a lda:R2RMLSparqlEndpoint ;
    lda:hasR2RMLMapping testblog:blogsMapping ;
    lda:hasR2RMLGraph rr:columnGraph ];
  lda:has_operation lda:GET, lda:PUT, lda:DELETE .

testblog:blogsMapping
  a rr:TriplesMap ;
  rr:logicalTable "blogs" ;
  rr:class sioct:Weblog ;
  rr:subjectMap [ a rr:IRIMap;
                  rr:column "id" ];
  rr:propertyObjectMap [ rr:property dc:creator;
                         rr:column "author" ];
  rr:propertyObjectMap [ rr:property dc:title;
                         rr:column "title" ];
  rr:propertyObjectMap [ rr:property dcterms:created;
                         rr:column "created_at" ;
                         rr:datatype xsd:dateTime ].
```

Listing 6: Blogs resources mapping

The main parts of the resource declaration are

- *api:uriTemplate* Designates a dereferenceable URI indentifying a set of Named graphs that can be addressed using the mapped HTTP operations. The property is reused from the LD-API specification, as well as the semantics for validating an URI against an URI template.
- *lda:endpoint* a SPARQL endpoint capable of processing the SPARQL Update query generated by the mapping of the HTTP method. Two main kinds of endpoints are valid: a list of R2RML *triplesMap* that are interpreted as a collection of patterns in a SPARQL query according to the first part of this paper or a *void:sparqlEndpoint* an external endpoint supporting the *SPARQL 1.1 Uniform HTTP Protocol for Managing RDF graphs* protocol.
- *lda:has_operation* contains the collection of HTTP operations that will be valid on the named graph being exposed as a resource. *lda:GET*, *lda:PUT*, *lda:POST*, and *lda:DELETE* are supported. These operations are interpreted according to the *SPARQL 1.1 Uniform HTTP Protocol for Managing RDF graphs* protocol.
- *lda:named_graph_creation_mechanism* describes how new named graphs are created in the service endpoint. The mechanism must specify an URI template for the new NG and what parts of the URI template for the named graph will be generated. Two generation mechanisms are supported *lda:UniqueIdInt*, *lda:UUID*. The first one generates a new unique integer, and the second one an UUID.

The proposed language for mapping resources to SPARQL endpoints makes possible to associate the same R2RML mapping to different named graphs. This can be achieved using the *lda:hasR2RMLGraph* property in the declaration of *R2RMLSparqlEndpoints*. If the *rr:table-graph-ri* property is specified, the R2RML mapping is augmented with that property and the resolved URI template for that resource. On the other hand, if the *rr:column-graph* property and the name of a column are used, the mapped value of that column will be matched against the URI of the resource in the generated SPARQL query. This feature can be used to restrict the results of the generated queries or to link the triples stored in different relational tables with foreign keys and expose them as a single RESTful resource.

3.3 Service Processing Model

Software implementations supporting the description of resources using the previous vocabulary must accept HTTP requests and process them following three main stages:

- Mapping the HTTP request to a SPARQL query
- Execution of the SPARQL query in the associated SPARQL endpoint
- Formatting of the resulting RDF graph as the representation of the resource agreed in the HTTP content negotiation process

The semantics of the SPARQL query to be build for each HTTP method matches the semantics described in the *SPARQL 1.1 Uniform HTTP Protocol for Managing RDF graphs* protocol draft.

3.3.1 GET requests

GET requests retrieve the whole graph associated to the resource using the SPARQL query

```
CONSTRUCT { ?s ?p ?o }
WHERE
{
  GRAPH <graph_uri> { ?s ?p ?o }
}
```

Listing 7: SPARQL query for a HTTP GET operation

as specified in the *SPARQL 1.1 Uniform HTTP Protocol for Managing RDF graphs* protocol.

This query can be executed in a RDBMS with a R2RML mapping, building a *SELECT* SPARQL query and returning the retrieved variable bindings as the triples of the RDF graph to return.

Support for pagination is provided via two special HTTP request parameters *_pageSize* and *_page*. The meaning of these parameters is equivalent to the one specified in the LD-API proposal. They are translated into values for the **LIMIT** and **OFFSET** clauses of the constructed SPARQL query. If these parameters are present the SPARQL query is also modified to be sorted by subject using the **ORDER BY ?s** SPARQL clause.

3.3.2 POST requests

POST requests create a new named graph in the associated SPARQL endpoint, inserting into that graph the RDF triples encoded in the HTTP request.

The SPARQL 1.1 Update query constructed according to the the *SPARQL 1.1 Uniform HTTP Protocol for Managing RDF graphs* protocol has the form

```
INSERT DATA
{
  GRAPH <graph_uri> {.. RDF payload .. }
}
```

Listing 8: SPARQL query for a HTTP POST operation

where *graph_uri* is minted using the mapping information associated to the *api:uriTemplate*, and *lda:mapped_uri_parts* properties.

A similar problem to the generation of the named graph URI is the generation of the URI for the resource to be created. This URI will be used as the subject of the triples containing the meta-data for the resource. Clients creating the new resource must submit a RDF graph containing a RDF graph with at least a single blank node identifying the new resource to be created. The API implementation will replace the blank node with a new minted URI of the form *graph_uri#self*, before inserting the triple graph in the SPARQL end point. If the RDF graph in the HTTP request payload contains more than one blank node, the identifier of the blank node for the resource to be created must be passed as an URL encoded parameter with name *_self*.

A distinction must be made between the named graph created to store the triples of the resource and the resource itself. The generated URI for the named graph must be deferencable as it is an information resource whose representation can be retrieved by the clients requesting a certain representation. On the other hand, the subject of the knowledge being encoded as an RDF graph is a non information resource that should not be directly dereferenced. The authority to create this new URI for the resource belongs to the provider of the service API [13]. The use of a blank node allows the client to provide a description of the resource without knowing in advance the URI that will be introduced by the server.

If the RDF payload of the HTTP request contains triples with subjects consisting of non blank node identifiers, the API implementation must return a *403 forbidden* error response.

If the creation of the named graph is successful, the API implementation must return a *201 created* response code with the location header pointing the URI of the newly created named graph according to the *SPARQL 1.1 Uniform HTTP Protocol for Managing RDF graphs* draft.

3.3.3 PUT requests

PUT HTTP requests are handled by the API implementation specification as a request to replace the knowledge associated to a named graph by knew knowledge encoded as a RDF graph. According to the *SPARQL 1.1 Uniform HTTP Protocol for Managing RDF graphs* protocol, the implementation must translate the HTTP request into a SPARQL request of the form

```
DROP SILENT GRAPH <graph_uri >;
INSERT DATA {
  GRAPH <graph_uri> { .. RDF payload .. }
}.
```

Listing 9: SPARQL query for a HTTP PUT operation

This query can be executed using the mapping of SPARQL *INSERT DATA* query for a R2RML mapping discussed in the previous section. The *DROP* SPARQL query can be executed using a *DELETE* query with a pattern matching any triple in the graph. The deletion of all the triples in the graph will also imply the deletion of the graph and the failure of any posterior query against the same graph.

The RDF payload of the HTTP request must contain only RDF triples with the URI of the underlying resource, matching the pattern *graph_uri#self* and blank node IDs. The request must fail with a *403 forbidden* response code otherwise.

3.3.4 DELETE requests

DELETE HTTP requests are interpreted by the API implementation as requesting the deletion of the named graph and all the contained triples. The generated SPARQL query according to the *SPARQL 1.1 Uniform HTTP Protocol for Managing RDF graphs* protocol is

```
DROP GRAPH <graph_uri>.
```

Listing 10: SPARQL query for a HTTP PUT operation

This operation can be implemented using the *DELETE* operation discussed in the previous section using a pattern that matches any triples in the named graph.

3.3.5 JSON support in requests and responses

JSON is the data format of choice for most of today's Web 2.0. In order to provide an useful interface for web developers, it is important to offer a JSON encoding of RDF graphs in requests and responses that can be used with the same easiness as in common entity-attribute-value JSON objects but, at the same time, can be transformed into a valid RDF graph.

JSON-LD [16] provides such a representation. Using a feature named type-coercion, RDF graphs can be encoded as plain javascript objects with an additional nested object containing the mapping from keys and values to RDF properties, URIs and literals.

Any HTTP request and response requesting a JSON media type for the encoding of the request payload or returned representation, must use type coerced JSON-LD as the exchange format.

3.3.6 Results formatting

After obtaining a successful response from the SPARQL endpoint the API implementation must return the retrieved knowledge to the client using the media type representation agreed using the content negotiation mechanism built into the HTTP protocol.

The media types specified in the LD-API specification must be accepted by any implementation of this API.

The API implementation must also accept a number of common parameters that can be passed to the API implementation to make the protocol more useful for restricted HTTP clients like web browser javascript applications.

- *_callback* parameter can be passed as a request parameter forcing the server to return a JSON-LD encoded representation of the resulting RDF graph passed as the single argument in the invocation of the javascript function passed as value of the parameter. This technique known as JSONP allows javascript applications being executed inside a browser to bypass the single domain restriction imposed by the browser.
- *_format* specifies a format for the requested representation. The parameter takes precedence over the value of the *Accept* HTTP header.

4. CONCLUSIONS AND FUTURE WORK

In this paper we have attempted to describe a possible technical solution for the problem of building writable LD APIs that can be adopted by present day web developers using the standard technological stack deployed in today's web applications. Special attention has been paid to offering an interface suitable for common web APIs clients like javascript web applications and smartphones.

The state of the current implementation relies on several technologies and standards in an immature state. *SPARQL 1.1 Update*, *SPARQL 1.1*, the *SPARQL 1.1 Uniform HTTP Protocol for Managing RDF graphs*, *LD-JSON* and *R2RML* are still recommendation drafts susceptible of change. The solution here proposed must consequently be adapted to the changes in these proposals. Additional work needs to be done to improve the performance of the mapping of SPARQL queries using R2RML. Proper benchmarking against triple stores and equivalent relational solutions need to be carried on to ensure that the translation mechanism is a viable solution.

The problem of exposing APIs with closed world semantics, adding some kind validation mechanism in the creation of resources in the API is also an important research task, that is required for the proposed API to be suitable in many common use cases. The generation of machine processable descriptions of LD APIs using vocabularies like voID [21] is also an important feature that LD APIs must address.

Technological solutions like the presented in this paper can play an important role in the transition from Web 2.0 APIs towards LD compliant APIs. The use of a technological stack already deployed and the declarative nature of the solution can lower the entry barrier for many web developers, for example, it makes possible to offer both kind of APIs in the same application. On the other hand, the use of a well defined interface that can be used with different kind of SPARQL end points, makes possible the future substitution of a relational backend mapped using R2RML by some technology specialized in the manipulation of RDF graphs.

Nevertheless, the factor that can push the adoption of LD style APIs is the added value new web applications taking advantage of the use of shared vocabularies and the capacity of linking resources in one API with other resources can bring to application users. Related work in authentication schemes for the web of data like WebID [4] and access privileges is also required to make possible the interaction of users with resources across different linked APIs.

5. REFERENCES

[1] Linked data, connect distributed data across the web. `http://linkeddata.org/`.

[2] Linking open data, w3c sweo community project. `http://www.w3.org/wiki/SweoIG/TaskForces/CommunityProjects/LinkingOpenData`.

[3] linked-data-api: Api and formats to simplify use of linked data by web-developers, December 2010. `http://code.google.com/p/linked-data-api/wiki/Specification`.

[4] Webid w3cś space, January 2011. `http://www.w3.org/wiki/WebID`.

[5] Tim Berners-Lee. Linked data, July 2006. `http://www.w3.org/DesignIssues/LinkedData.html`.

[6] Jeremy J. Carroll, Christian Bizer, Pat Hayes, and Patrick 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.

[7] Artem Chebotko, Shiyong Lu, and Farshad Fotouhi. Semantics preserving sparql-to-sql translation. *Data Knowl. Eng.*, 68:973–1000, October 2009.

[8] Richard Cyganiak. A relational algebra for SPARQL. 2005.

[9] Brendan Elliott, En Cheng, Chimezie Thomas-Ogbuji, and Z. Meral Ozsoyoglu. A complete translation from sparql into efficient sql. In *Proceedings of the 2009 International Database Engineering & Applications Symposium*, IDEAS '09, pages 31–42, New York, NY, USA, 2009. ACM.

[10] Roy T. Fielding. *Architectural Styles and the Design of Network-based Software Architectures*. PhD thesis, University of California, Irvine, 2000.

[11] E. Hammer-Lahav. The oauth 1.0 protocol. RFC 5849, April 2010.

[12] Matthias Hert, Gerald Reif, and Harald C. Gall. Updating relational data via sparql/update. In *Proceedings of the 2010 EDBT/ICDT Workshops*, EDBT '10, pages 24:1–24:8, New York, NY, USA, 2010. ACM.

[13] Ian Jacobs. Architecture of the world wide web, volume one (uri ownership). Technical report, 2004. `http://www.w3.org/TR/webarch/#uri-ownership`.

[14] Jing Lu, Feng Cao, Li Ma, Yong Yu, and Yue Pan. An effective sparql support over relational databases. In Vassilis Christophides, Martine Collard, and Claudio Gutierrez, editors, *Semantic Web, Ontologies and Databases*, volume 5005 of *Lecture Notes in Computer Science*, pages 57–76. Springer Berlin / Heidelberg, 2008.

[15] Juan Sequeda Marcelo Arenas, Eric Prud'hommeaux. A direct mapping of relational data to rdf. W3C working draft, W3C, November 2010. `http://www.w3.org/TR/rdb-direct-mapping/`.

[16] Manu Sporny Mark Birbeck. Json-ld - linked data expression in json. Technical report, 2011. `http://json-ld.org/`.

[17] Chimezie Ogbuji. Sparql 1.1 uniform http protocol for managing rdf graphs. W3C working draft, W3C, October 2010. `http://www.w3.org/TR/sparql11-http-rdf-update/`.

[18] Sunitha Ramanujam, Vaibhav Khadilkar, Latifur Khan, Steven Seida, Murat Kantarcioglu, and Bhavani Thuraisingham. Bi-directional translation of relational data into virtual rdf stores. In *Proceedings of the 2010 IEEE Fourth International Conference on Semantic Computing*, ICSC '10, pages 268–276, Washington, DC, USA, 2010. IEEE Computer Society.

[19] Microsoft Research. Open data protocol. `http://www.odata.org/developers/protocols/overview`.

[20] Chris Bizer Richard Cyganiak. Pubby: A linked data frontend for sparql endpoints, June 2007. `http://www4.wiwiss.fu-berlin.de/pubby/`.

[21] Michael Hausenblas et alt. Richard Cyganiak. Describing linked datasets with the void vocabulary. Technical report, W3C, 2010. `http://www.w3.org/2001/sw/interest/void/`.

[22] Seema Sundara Richard Cyganiak and Souripriya Das. R2rml: Rdb to rdf mapping language. W3C working draft, W3C, October 2010. `http://www.w3.org/TR/r2rml/`.

[23] Alexandre Passant Simon Schenk, Paul Gaeron. Sparql 1.1 update. W3C working draft, W3C, October 2010. `http://www.w3.org/TR/sparql11-update/`.

[24] W3C. Large triple stores report. `http://www.w3.org/wiki/LargeTripleStores`.

[25] Erik Wilde and Michael Hausenblas. Restful sparql? you name it!: aligning sparql with rest and resource orientation. In *Proceedings of the 4th Workshop on Emerging Web Services Technology*, WEWST '09, pages 39–43, New York, NY, USA, 2009. ACM.