Linked Data Templates

Ontology-driven approach to read-write Linked Data

Martynas Jusevičius

AtomGraph

<martynas@atomgraph.com>

Abstract

In this paper we summarize the architecture of Linked Data Templates, a uniform protocol for read-write Linked Data.

We start by overviewing existing approaches as well as protocol constraints, and propose a declarative approach based on URI-representation mapping, defined as templates. We then introduce an abstract model for RDF CRUD interactions based on template matching, outline the processing model, and provide an example.

We conclude that LDT provides a new way to build applications and can be used to implement the ontology-driven Semantic Web vision.

Keywords: HTTP, REST, RDF, Linked Data, SPARQL, XSLT, SPIN, declarative, data-driven

1. Introduction

Linked Data is a vast source of information available in RDF data model. In this paper we describe Linked Data Templates: a generic method for software agents to publish and consume read-write Linked Data. By doing so, we facilitate a distributed web as well as redefine the software development process in a declarative manner.

Web applications duplicate a lot of domain-specific code across imperative programming language implementations. We abstract application logic away from source code and capture it in a machine-processable representation that enables reuse, composition, and reasoning. Our goal is to permit software to operate on itself through metaprogramming to generate higher-level applications — a feature not available in imperative languages. Having a uniform homoiconic representation, we reduce application state changes to uniform Create, Read, Update, Delete (CRUD) interactions on Linked Data representations [1].

Linked Data representations describe application resource properties. A consuming process can query and change resource state by issuing requests to resource URIs. On the producing end, representations are generated from an RDF dataset, either stored natively in a triplestore or mapped from another data model. The exact logic of how representations are generated and stored is application-specific.

By establishing a mapping between URI address space and the RDF representation space, we can model application structure in terms of RDF classes that map URIs to RDF queries and updates. We choose RDF-based ontologies as the LDT representation, as it is the standard way to define classes and satisfies both the machine-processability and the homoiconicity requirements. Ontology as a component for application structure is what distinguishes LDT from other Linked Data specifications such as Linked Data Platform.

In the following sections, we explain the motivation behind LDT and describe the LDT architecture in more detail. First we establish that applications can be driven by ontologies that can be composed. We then introduce templates, special ontology classes that map URI identifiers to request-specific SPARQL strings. We proceed to describe how a process matching templates against request URI is used by RDF CRUD interactions that generate Linked Data descriptions from an RDF dataset and change the dataset state. Lastly, we map the interactions to the HTTP protocol and show how the client can use HTTP to interact with LDT applications.

2. Distributed web as read-write Linked Data

2.1. A protocol for the web of data

Smart software agents should navigate the web of data and perform tasks for their users — that has been an early optimistic vision of the Semantic Web [2]. Ontologies and ontology-driven agents were central to this vision, to provide the means to capture domain logic in a way that sustains reason. It was largely forgotten when the high expectations were not met, and the community focus shifted to the more pragmatic Linked Data.

In order to enable smart agents, we need to provide a uniform protocol for them to communicate. Such a protocol, understood by all agents in the ecosystem, decouples software from domain- or application-specific logic and enables generic implementations. The web has thrived because HTTP is such protocol for HTML documents, and web browsers are generic agents.

REST readily provides uniform interface for a protocol, which has been successfully implemented in HTTP. The interface is defined using 4 constraints [3]:

- identification of resources
- manipulation of resources through representations
- self-descriptive messages
- hypermedia as the engine of application state

Since the Linked Data resource space is a subset of REST resource space, we can look at how these constraints apply to standard Linked Data technologies:

- URIs identify resources
- RDF is used as resource representation. Resource state can be changed using RDF CRUD interactions.
- RDF formats are used for self-describing Linked Data requests and responses

2.2. Ontology-driven Linked Data

The main point of interest for this paper is RDF CRUD interactions, the central yet underspecified component of read-write Linked Data. The only standard in this area is W3C Linked Data Platform 1.0 specification, which defines a set of rules for HTTP interactions on web resources, some based on RDF, to provide an architecture for read-write Linked Data on the web [4]. It has several shortcomings:

- It is coupled with HTTP and provides no abstract model for RDF CRUD
- In order to accommodate legacy systems, it does not mandate the use of SPARQL. SPARQL is the standard RDF query language and does provide an abstract model [5].
- It does not offer a standard way for agents to customize how CRUD interactions change resource state

The Linked Data API specification defines a vocabulary and processing model for a configurable, yet read-only API layer intended to support the creation of simple RESTful APIs over RDF triple stores [6]. Hydra Core Vocabulary is a lightweight vocabulary to create hypermedia-driven Web APIs and has similar goals to combine REST with Linked Data principles, but does not employ SPARQL and focuses on JSON-LD [7].

As we can see, currently popular Linked Data access methods are either read-only or do not use SPARQL. They use lightweight RDF vocabularies, but not formal ontologies that enable reasoning, as does the original ontology-driven Semantic Web vision. Although there has been a fair share of research and development in the area of ontology-driven applications [8] [9], it focuses mostly on domain and user interface modeling, and not representation manipulation modeling.

2.3. LDT design

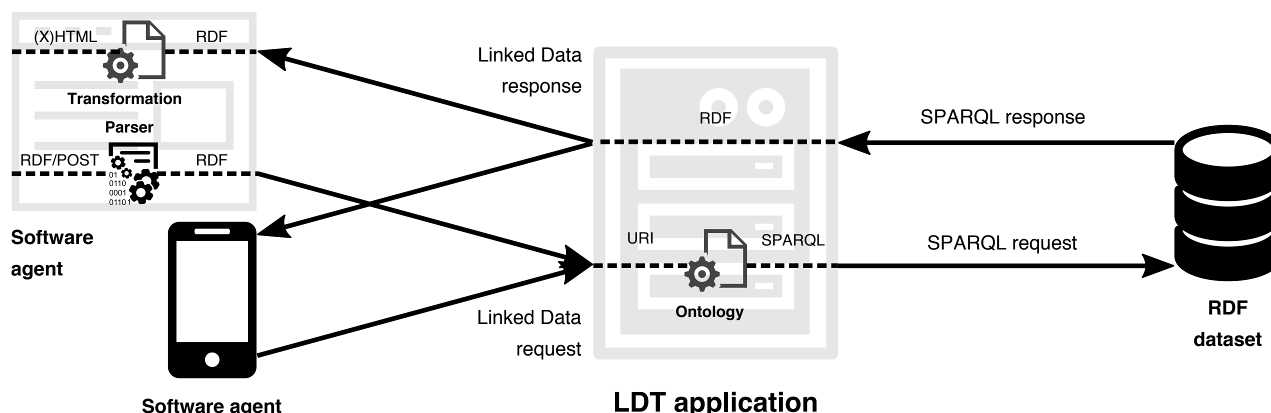
We propose Linked Data Templates, an ontology-driven approach to read-write Linked Data. It builds on the following constraints:

- there is a mapping between URI address space and the RDF representation space. It is used to determine resource's representation from its URI identifier.
- applications are read-write and backed by SPARQL 1.1 compatible services to decouple them from database implementations
- application structure is defined in an ontology to enable reasoning and composition
- application state is driven by hypermedia (HATEOAS) to satisfy REST constraints

XSLT is a homoiconic high-level language for the XML data model [10]. We wanted to follow this approach with a Linked Data specification, and as a result, XSLT heavily influenced the template-based design of LDT. We draw multiple parallels between XSLT stylesheets and LDT ontologies, XML source documents and RDF datasets, XPath patterns and URI templates etc.

AtomGraph Processor[2] is an open-source implementation of LDT. The commercial AtomGraph Platform[3] provides a multi-tenant environment and has been successfully used to build rich LDT applications for product information management and library data.

Figure 1. Main components of LDT architecture[1]



3. Application ontologies

[Definition: An LDT *application* represents a data space identified by its base URI, in which application resource URIs are relative to the base URI.] The only external interface an application provides is RESTful Linked Data: application produces RDF representations when resource URIs are dereferenced, and consumes RDF representations when requested to change resource state.

Application structure, the relationships between its resources, is communicated through representations. Representations are generated from, and stored in, an RDF dataset. Two different applications should be able to use the same dataset yet expose different structures because they produce representations and change state differently. It follows that application structure can be defined as instructions for representation processing.

An ontology is an efficient way to define such structure declaratively. We use OWL to define LDT application ontologies with RDF query and state change instructions specific to that application. We use SPARQL to encode these instructions, because it is the standard RDF query and update language and can be conveniently embedded in ontologies using SPIN RDF syntax. Using SPARQL service as the interface for the dataset, applications are independent from its implementation details.

An LDT application ontology may comprise several ontologies, contained in different RDF graphs. For a given application, one of these serves as the *principal ontology*. Ontologies are composed through the standard `owl:imports` mechanism, with the additional concept of import precedence, which makes ontologies override each other depending on the import order.

LDT does not make any assumptions about the application structure. There is however a useful one: a resource hierarchy consisting of a container/item tree. It is similar to the container/resource design in LDP, but based on the SIOC ontology instead [11]. The vast majority of Web applications can be modeled using this structure.

4. Templates

[Definition: A *template* is a declarative instruction contained in an ontology, defined using LDT vocabulary[1], and driving RDF CRUD processing.] It is a special ontology class that maps a certain part of the application URI space to a certain SPARQL string. A template can be viewed as a function with URI as the domain and SPARQL as the range, which are the two mandatory parts of the template, detailed below.

A template domain is defined using `ldt:path` property and a regex-based JAX-RS URI template syntax [12]. It is a generic way to define a class of resources based on their URI syntax: if an URI matches the template, its resource is a member of the class. Starting with a catch-all template that matches all resources in an application, we can specialize the URI pattern (e.g. by adding fixed paths) to narrow down the class of matching resources.

A template range is defined using `ldt:query` property and SPIN RDF syntax, while updates use `ldt:update`

[1] Icons used in the diagram made by Freepik http://www.flaticon.com
[2] AtomGraph Processor - https://github.com/AtomGraph/Processor
[3] AtomGraph Platform - http://atomgraph.com
[1] LDT vocabulary is planned to have http://www.w3.org/ns/ldt# namespace in the final specification

property [13]. URI that matches URI template is passed to SPARQL using a special variable binding `?this` (path variables from the URI template match, if any, are not used since URIs are opaque). Starting with the default query `DESCRIBE ?this`, we can specialize it with a graph pattern, for example to include descriptions of resources connected to `?this` resource. The query forms are limited to `DESCRIBE` and `CONSTRUCT`, as the required result is RDF graph.

An important feature of LDT templates is *annotation inheritance*, which enables code reuse but requires reasoning. It mimics object-oriented multiple inheritance: a class inherits annotation properties from its superclasses via the `rdfs:subClassOf` relation, unless it defines one or more properties of its own which override the inherited ones. SPIN takes a similar object-oriented world-view and uses subclass-based inheritance.

5. Processing model

We have established that application state is queried and changed using Linked Data requests that trigger RDF CRUD in the form of SPARQL. We can constrain the requests to 4 types of CRUD *interactions* that map to either SPARQL query or update. An interaction is triggered by a Linked Data request and results in query or change of application state by means of SPARQL execution [14].

Table 1. LDT interaction types

Interaction type	SPARQL form	Generated from
Create	`INSERT DATA`	request RDF entity
Read	`DESCRIBE`/ `CONSTRUCT`	`ldt:query`
Update	`DELETE; INSERT DATA`	`ldt:update`; request RDF entity
Delete	`DELETE`	`ldt:update`

`DESCRIBE` and `CONSTRUCT` forms are generated from `ldt:query` SPARQL templates; `DELETE` is generated from `ldt:update` SPARQL template. `INSERT DATA` is generated from the RDF in the request entity, either as triples or as quads. Update interaction combines two updates into one SPARQL request.

[Definition: We refer to the software that uses application templates to support the interaction as an LDT *processor*.] A processor consists of several sub-processes that are triggered by a Linked Data request and executed in the following order:

1. A validation process validates incoming RDF representations against SPIN constraints in the ontology. Invalid data is rejected as bad request. Only applies to Create and Update.
2. A skolemization process matches request RDF types against ontology classes and relabels blank nodes as URIs. Only applies to Create.
3. A matching process matches the base-relative request URI against all URI templates in the application ontology, taking import precedence and JAX-RS priority algorithm into account. If there is no match, the resource is considered not found and the process aborts.
4. A SPARQL generation process takes the SPARQL string from the matching template and applies `?this` variable binding with request URI value to produce a query or an update, depending on the interaction type. `BASE` is set to application base URI.
5. A SPARQL execution process executes the query/ update on the application's SPARQL service. If there is a query result, it becomes the response entity.
6. A response generation process serializes the response entity, if any. It uses content negotiation to select the most appropriate RDF format, sets response status code, adds ontology URI, matched template URI and inheritance rules as header metadata.

For the container/item application structure it is convenient to extend this basic model with pagination, which allows page-based access to children of a container. It requires `SELECT` subqueries and extensions to query generation and response generation processes.

Having access to application ontologies, LDT clients can infer additional metadata that helps them formulate successful requests. For example, SPIN constructors can be used to compose new resource representations from class instances, while SPIN constraints can be used to identify required resource properties. Applications with embedded clients become nodes of a distributed web, in which data flows freely between peers in either direction.

5.1. HTTP bindings

The mapping to HTTP is straightforward — each interaction has a corresponding HTTP method:

Table 2. LDT interaction mapping to HTTP

Interaction type	Request method	Success statuses	Failure statuses
Create	POST	201 Created	400 Bad Request
			404 Not Found
Read	GET	200 OK	404 Not Found
Update	PUT	200 OK	400 Bad Request
		201 Created	404 Not Found
Delete	DELETE	204 No Content	404 Not Found

It should be possible to use the PATCH method for partial modifications instead of replacing full representation with PUT, but that is currently unspecified.

5.1.1. Example

In the following example, an HTTP client performs an Update-Read request flow on linkeddatahub.com application, which supports LDT. Only relevant HTTP headers are included.

First, the client creates a resource representing Tim Berners-Lee by submitting its representation:

```
PUT /people/Berners-Lee HTTP/1.1
Host: linkeddatahub.com
Accept: text/turtle
Content-Type: text/turtle

@base <http://linkeddatahub.com/people/Berners-Lee> .
@prefix ldt:  <http://www.w3.org/ns/ldt#> .
@prefix foaf: <http://xmlns.com/foaf/0.1/> .
@prefix owl:  <http://www.w3.org/2002/07/owl#> .

<> a ldt:Document ;
  foaf:primaryTopic <#this> .

<#this> a foaf:Person ;
  foaf:isPrimaryTopicOf <> ;
  owl:sameAs
  <https://www.w3.org/People/Berners-Lee/card#i> .
```

Let's assume the match for /people/Berners-Lee request URI is the :PersonDocument template in the application ontology:

```
@base        <http://linkeddatahub.com/ontology> .
@prefix :     <#> .
@prefix ldt:  <http://www.w3.org/ns/ldt#> .
@prefix owl:  <http://www.w3.org/2002/07/owl#> .
@prefix rdfs: <http://www.w3.org/2000/01/rdf-schema#> .
@prefix sp:   <http://spinrdf.org/sp#> .

# ontology
: a ldt:Ontology ;
  owl:imports ldt: .

# template
:PersonDocument a rdfs:Class, ldt:Template ;
  ldt:path "/people/{familyName}" ;
  ldt:query :DescribeWithPrimaryTopic ;
  ldt:update :DeleteWithPrimaryTopic ;
  rdfs:isDefinedBy : .

# query
:DescribeWithPrimaryTopic a sp:Describe, ldt:Query ;
  sp:text
      """PREFIX  foaf: <http://xmlns.com/foaf/0.1/>
DESCRIBE ?this ?primaryTopic
WHERE
  { ?this ?p ?o
    OPTIONAL
      { ?this foaf:primaryTopic ?primaryTopic }
  }""" .

# update
:DeleteWithPrimaryTopic a sp:DeleteWhere, ldt:Update ;
  sp:text
       """PREFIX  foaf: <http://xmlns.com/foaf/0.1/>
DELETE {
  ?this ?p ?o .
  ?primaryTopic ?primaryTopicP ?primaryTopicO .
}
WHERE
  { ?this ?p ?o
    OPTIONAL
      { ?this foaf:primaryTopic ?primaryTopic .
        ?primaryTopic ?primaryTopicP ?primaryTopicO
      }
  }""" .
```

The variable binding (?this, <http://linkeddatahub.com/people/Berners-Lee>) is applied on the DELETE associated with the template. It is combined with INSERT DATA generated from the request RDF entity

into a single update request. Application base URI is set on the final SPARQL string which is then executed on the SPARQL service behind the application:

```
BASE          <http://linkeddatahub.com/>
PREFIX  foaf: <http://xmlns.com/foaf/0.1/>
PREFIX  owl:  <http://www.w3.org/2002/07/owl#>
PREFIX  ldt:  <http://www.w3.org/ns/ldt#>

DELETE {
  <people/Berners-Lee> ?p ?o .
  ?primaryTopic ?primaryTopicP ?primaryTopicO .
}
WHERE
  { <people/Berners-Lee> ?p ?o
    OPTIONAL
      {
        <people/Berners-Lee>
                  foaf:primaryTopic ?primaryTopic .
        ?primaryTopic ?primaryTopicP ?primaryTopicO
      }
  } ;

INSERT DATA {
  <people/Berners-Lee>
    a ldt:Document .
  <people/Berners-Lee>
    foaf:primaryTopic <people/Berners-Lee#this> .
  <people/Berners-Lee#this>
    a foaf:Person .
  <people/Berners-Lee#this>
    foaf:isPrimaryTopicOf <people/Berners-Lee> .
  <people/Berners-Lee#this>
    owl:sameAs
    <https://www.w3.org/People/Berners-Lee/card#i> .
}
```

We assume the representation did not exist beforehand, so it is created instead of being updated (an optimized implementation might have skipped the DELETE part in this case). The application responds with:

```
HTTP/1.1 201 Created
Location: http://linkeddatahub.com/people/Berners-Lee
```

The client can choose to follow the link to the newly created resource URI, and retrieve the same representation that was included with the initial PUT request:

```
GET /people/Berners-Lee HTTP/1.1
Host: linkeddatahub.com
Accept: text/turtle
```

We omit the response, but note that the application would use the DESCRIBE query associated with the matching template to generate the representation.

6. Future work

The use of OWL and SPARQL is probably the biggest advantage and limitation of LDT at the same time. RDF ontology and query tools as well as developers are scarce for mainstream programming languages with the possible exception of Java, making implementations expensive and adoption slow. Query performance is a potential issue, albeit constantly improving and alleviated using proxy caching. On the other hand, OWL and SPARQL provide future-proof abstract models on which LDT builds.

We are working around slow adoption of Linked Data by providing a hosted LDT application platform[1]. It uses metaprogramming to implement complex data management features such as application and resource creation, autocompletion, access control, provenance tracking, faceted search — all done through a user interface, exposing as little technical RDF details as possible.

We envision an ecosystem in which applications by different developers interact with each other: ask for permissions to access or create data, send notifications to users, automate interactions etc.

7. Conclusions

In this paper we have described how read-write Linked Data applications can be modeled using standard RDF/OWL and SPARQL concepts. Linked Data Templates enable a new way to build declarative software components that can run on different processors and platforms, be imported, merged, forked, managed collaboratively, transformed, queried etc. Experience with AtomGraph software has shown that such design is also very scalable, as the implementation is stateless and functional. We expect that substantial long-term savings in software engineering and development processes can be achieved using this approach.

We have shown that SPARQL is the crucial link that reconciles ontology-driven Semantic Web and read-write Linked Data. Using SPARQL, Linked Data Templates define a protocol for distributed web of data as uniform RDF CRUD interactions. LDT already provide features

[1] AtomGraph - http://atomgraph.com

from the original Semantic Web vision, such as ontology exchange between agents, and we are confident it has the potential to implement it in full.

Bibliography

[1] *Create, read, update and delete*. Wikipedia, The Free Encyclopedia.
https://en.wikipedia.org/wiki/Create,_read,_update_and_delete

[2] *The Semantic Web*. Tim Berners-Lee, James Hendler, and Ora Lassila. Scientific American. 1 May 2001.
http://www.scientificamerican.com/article/the-semantic-web/

[3] *Representational State Transfer (REST)*. Roy Thomas Fielding. 2000.
https://www.ics.uci.edu/~fielding/pubs/dissertation/rest_arch_style.htm#sec_5_1_5

[4] *Linked Data Platform 1.0*. Steve Speicher, John Arwe, and Ashok Malhotra. World Wide Web Consortium (W3C). 26 February 2015.
https://www.w3.org/TR/ldp/

[5] *SPARQL 1.1 Query Language*. Steve Harris and Andy Seaborne. World Wide Web Consortium (W3C). 21 March 2013.
https://www.w3.org/TR/sparql11-query/

[6] *Linked Data API Specification*.
https://github.com/UKGovLD/linked-data-api/blob/wiki/Specification.md

[7] *Hydra Core Vocabulary*. Markus Lanthaler. 20 March 2016.
http://www.hydra-cg.com/spec/latest/core/

[8] *Agents and the Semantic Web*. James Hendler. 2001.
http://www.cs.rpi.edu/~hendler/AgentWeb.html

[9] *Ontology-Driven Apps Using Generic Applications*. Michael K. Bergman. 7 March 2011.
http://www.mkbergman.com/948/ontology-driven-apps-using-generic-applications/

[10] *XSL Transformations (XSLT) Version 2.0*. Michael Kay. World Wide Web Consortium (W3C). 23 January 2007.
https://www.w3.org/TR/xslt20/

[11] *SIOC Core Ontology Specification*. Uldis Bojārs and John G. Breslin. DERI, NUI Galway. 25 March 2010.
http://rdfs.org/sioc/spec/

[12] *JAX-RS: Java™ API for RESTful Web Services*. Marc Hadley and Paul Sandoz. Sun Microsystems, Inc.. 17 September 2009.
https://jsr311.java.net/nonav/releases/1.1/spec/spec3.html#x3-300003.4

[13] *SPIN - Modeling Vocabulary*. Holger Knublauch. 7 November 2014.
http://spinrdf.org/spin.html

[14] *Architecture of the World Wide Web, Volume One*. Ian Jacobs and Norman Walsh. World Wide Web Consortium (W3C). 15 December 2004.
https://www.w3.org/TR/webarch/#interaction