Graphity: generic processor for declarative Linked Data applications

Martynas Jusevičius

Graphity

martynas@graphity.org

Abstract. In this paper we describe a novel approach to read-write Linked Data design. By combining URI-to-query mapping with SPIN, XSLT, and RDF/POST, many advanced Web application features can be implemented declaratively. Such architecture is standard-compliant and provides a rapid way to build life-science Linked Data apps from reusable components.

Keywords: Life sciences · Linked Data · Declarative · Web Applications · RDF · SPARQL · Graphity · SPIN · XSLT · Linked Data Platform · REST · OWL

1. Introduction

The volume of life-science related RDF data is set to grow [1]. Data publishers providing Linked Data access often choose to develop new software, which is costly. Usability expectations are high. We address these issues with a novel software design that makes the components reusable and the development rapid by reducing the functionality into simple operations on semantic data.

In Graphity [2], we implemented a declarative approach which maps read-write Linked Data HTTP access to SPARQL operations on RDF stores. The W3C has done related work based on XML [3], but has not applied it to RDF. The Linked Data Platform [4] has similar aims, but lacks the mapping to SPARQL.

2. Approach

In RESTful applications, resources that share the same URI structure usually share the same representation pattern. In our case, the representation is RDF, with views defined in SPARQL. Let us consider a simple Linked Data request, to which the server responds with a query result:

```
GET <resource> → DESCRIBE <resource>
```

We can generalize this into a mapping between resource URIs and SPARQL queries by using templates of the form:

```
/{path} → DESCRIBE ?this
```

Special query variable `?this` stands for the request URI that matches the template. A collection of such mappings is application-specific and called a sitemap.

3. Implementation

Each application instance embeds a processor, which interprets the sitemap on each HTTP request and executes queries on a SPARQL endpoint. No domain-specific object layer is present: the triplestore acts as an MVC model via HTTP, while the processor is the controller, and XSLT is the optional view layer. All components operate on RDF natively and directly.

1.1 Sitemap and templates

Resource template is an OWL class, which maps a URI template to a SPARQL query. For that we use regex-like JAX-RS syntax [5] and SPIN RDF [6] syntax, respectively:

```
gp:Resource a owl:Class, gp:Template ;
  rdfs:subClassOf foaf:Document ;
  gp:uriTemplate "/{path: .*}" ;
  spin:query gp:Describe .

gp:Describe a sp:Describe, sp:Query ;
  sp:text """DESCRIBE ?this"""^^xsd:string .
```

This example shows a catch-all URI template mapped to a default query. In real-world applications both the URI templates and the queries are more specialized. The query forms are limited to `DESCRIBE` and `CONSTRUCT`, as the required result is RDF graph.

1.2 Processing model

Each request URI is first matched against the set of URI templates in the sitemap, using the JAX-RS precedence algorithm. `?this` in the query of the matching template is bound to request URI. The query is then executed on the endpoint, and the result is returned as the representation of the requested resource. If no template matches, `404 Not Found` is returned.

It is often useful to arrange resources in a hierarchical fashion. Folders and files in the filesystem is well known example. We implement this using container resources that have children resources. Container queries must contain `SELECT` subqueries to provide paginated access to its children by dynamically setting `LIMIT` and `OFFSET`

modifiers. Ordering is implemented using `ORDER BY`. Default modifier values are specified in the resource template and overridden by request query parameter values:

```
<#Container> a owl:Class, gp:Template ;
  gp:limit 20 ;
  gp:offset 100 ;
  gp:orderBy "title"^^xsd:string .
```

Previous/next page resources are added to container responses automatically, following the REST principles:

```
<container?limit=20&offset=120&orderBy=title>
```

1.3 Representations

Representations should be available in all RDF serializations supported by the underlying I/O framework via content negotiation. The processor must select best-matching media type based on the `Accept` header of the request.

The application can produce non-RDF representations, including binary. (X)HTML output is achieved via XSLT transformation of RDF/XML, server- and client-side. Layout modes can be configured per resource template.

Caching can also be configured per resource template:

```
<#CachedDocument> a owl:Class, gp:Template ;
  gp:cacheControl "public, max-age=86400" .
```

1.4 Data input

`POST` HTTP request to a container is used to create a new child resource, while `PUT` is used to replace existing representation. The update is done using SPARQL Update template attached to resource template using `spin:update` property. However, Graph Store Protocol [7] is often more convenient .

Blank nodes in the request RDF payload are skolemized. For example, building URI for node with `dct:identifier` value `42` and URI template `/books/{identifier}` yields `/books/42`. Unmatched bnodes are left untouched.

Quality of the incoming data is controlled using SPIN constraints [8]. If the data is invalid, an error response is returned to the client and no further processing is done.

A common, but still problematic use case in read-write Linked Data applications is retrieving user input natively as RDF. Luckily, that is covered by RDF/POST [9].

1.5 Access control

Access control can be implemented using the W3C ACL ontology [10]. Graphity provides a transparent filter that authorizes each request using an `ASK` SPARQL query for public or authenticated agent access.

User accounts and authorizations are stored in a RDF repository separately from the domain data, prohibiting the end-users from modifying them. The filter has access to both of them by means of federation (the `SERVICE` SPARQL keyword).

4. Conclusions

The architecture of a generic processor interpreting declarative application-specific sitemaps enables a new way to reuse application components without writing new software. It is also very scalable, as the system is stateless and functional.

Core Web application functionality such as URI routing, data representation and input, and access control can be implemented using standard RDF/OWL constructs only. Such apps can run on different processors and platforms, can be imported, merged, forked, managed collaboratively, transformed, queried etc.

The Graphity processor is open-source and works with any SPARQL 1.1 triplestore. The commercial platform layer provides multi-tenant functionality and has been successfully used to build rich Linked Data applications for product information management [11] and library data [12].

Our goal is to formalize this approach as a W3C submission. We invite you to join the discussion at the W3C Declarative Linked Data Apps Community Group [13].

References

1. S. Jupp et al. The EBI RDF Platform: Linked Open Data for the Life Sciences.
2. Graphity. http://graphityhq.com
3. Declarative Web Applications Current Status, http://www.w3.org/standards/techs/dwa
4. Linked Data Platform 1.0. http://www.w3.org/TR/ldp/
5. The @Path Annotation and URI Path Templates. http://docs.oracle.com/cd/E19798-01/821-1841/6nmq2cp26/index.html
6. SPIN - Modeling Vocabulary. http://spinrdf.org/spin.html
7. SPARQL 1.1 Graph Store HTTP Protocol. http://www.w3.org/TR/sparql11-http-rdf-update/
8. The Data Quality Constraints Library Language Reference. http://semwebquality.org/ontologies/dq-constraints
9. RDF/POST Encoding for RDF. http://www.lsrn.org/semweb/rdfpost.html
10. WebAccessControl. http://www.w3.org/wiki/WebAccessControl
11. NXP Data, http://data.nxp.com
12. De Danske Aviser. http://dedanskeaviser.dk
13. Declarative Linked Data Apps CG, http://www.w3.org/community/declarative-apps/