Collaborative Project

GeoKnow - Making the Web an Exploratory for Geospatial Knowledge

Project Number: 318159 **Start Date of Project:** 2012/12/01 **Duration:** 36 months

Deliverable 4.2.2
Spatial curation interface

Dissemination Level	Public
Due Date of Deliverable	Month M30, 31/05/2015
Actual Submission Date	Month M36, 30/11/2015
Work Package	WP4, Spatial-Semantic Browsing, Visualisation and Authoring Interfaces
Task	T4.2
Type	Prototype Release
Approval Status	Approved
Version	1.0
Number of Pages	31
Filename	D4.2.2_Spatial_curation_interface.pdf

Abstract In this deliverable we present an approach and a supporting software framework for curating large and small numbers of resources by means of SPARQL-intensive workflows realized with a batch processing framework. We demonstrate the effectiveness of the framework in a sophisticated geocoding scenario.

Project funded by the European Commission within the Seventh Framework Programme (2007 - 2013)

Version	Date	Reason	Revised by
0.0	03/02/2015	Initial version	Clemens Hoffmann
0.1	10/03/2015	Collection of Use Cases	Clemens Hoffmann
0.2	15/05/2015	Conceptual work	Clemens Hoffmann
0.3	20/05/2015	Revision of the conceptual work	Claus Stadler
0.4	09/07/2015	Revision of the architecture	Claus Stadler
0.5	15/09/2015	Added change tracking and undo	Claus Stadler
0.6	05/10/2015	Revision of the architecture	Claus Stadler
0.7	16/10/2015	Proof reading	Jens Lehmann
0.8	21/11/2015	Peer review	Vuk Mijovic
0.9	29/11/2015	Addressed peer review comments	Claus Stadler
1.0	30/11/2015	Approval	Jens Lehmann

Author List

Organization	Name	Contact Information
InfAI	Claus Stadler	cstadler@informatik.uni-leipzig.de
InfAI	Clemens Hoffmann	cannelony@gmail.com
InfAI	Jens Lehmann	lehmann@informatik.uni-leipzig.de

Executive Summary

In this software prototype deliverable, we present *Lodtenant*, a system for curating RDF data by means of workflows realized as batch processes. We show that by the introduction of additional SPARQL functions, SPARQL can even be used for fetching JSON (or XML) data from a (geocoder) REST service, and for transforming the response to triples in virtually any (spatial) vocabulary. Thereby we focus on making the system as easy to use as possible: For the workflow configuration, JSON can now be used as an alternative to spring's usual Java or XML approaches. As with the other approaches, the JSON document defines Java beans and their dependencies. However, the benefits of this format are: The possibility to expand shorthands in the JSON document before actually processing the bean definitions. While this can be partly done in XML as well, for JSON such transformations can be succinctly written in JavaScript. The most essential benefit of JSON however is, that browsers support this format natively, such that data exchange between a server component and a Web client becomes greatly simplified. Our software prototype is made available as (Debian) packages in the Linked Data Stack, the source code is published on Github, and corresponding software artifacts are released in the the central Maven repository, making our system easily accessible to users and developers.

Abbreviations and Acronyms

LOD	Linked Open Data
RDF	Resource Description Framework
SPARQL	SPARQL Protocol and RDF Query Language
JSON	JavaScript Object Notation

Table of Contents

1 Introduction

In general, curation involves maintenance, preservation and addition of value to assets. The natures of these assets can be very diverse such as cultural heritage sites, books, digital assets and data sets. Specifically, *data curation includes all the processes needed for principled and controlled data creation, maintenance, and management, together with the capacity to add value to data.*[1]. Under this perspective, activities related to Linked Data curation could be seen as broad as to potentially comprise all of the Linked Data life cycle's phases, depicted in Figure 1. Further, accomplishing curation tasks often requires carrying activities out in certain steps

Figure 1: The Linked Data life cycle

that form workflows. In this deliverable, we study how RDF-intensive tasks can be represented as extraction-transform-load (ETL) workflows. Our approach is implemented in the *Lodtenant* software prototype. Our contributions are as follows:

- We devise models for atomic and chunked SPARQL based operations.
- We further devise a simple, yet effective and flexible SPARQL-based approach for data integration of third party REST APIs by extending SPARQL's function set with support for HTTP requests and JSON queries.
- Within our framework, we devise approaches for enriching RDF data with spatial information, thereby taking support for various spatial vocabularies into account.
- We provide implementations of aforementioned concepts using the well known spring batch framework, that has also been used for managing certain jobs in the GeoKnow generator.
- We allow workflows to be represented as JSON documents (in addition to spring's default XML support) which simplifies the development of Web frontends for editing workflow specifications.
- We demonstrate the usefulness of our framework using a sophisticated geocoding/enrichment scenario.

The remainder is structured as follows: In section 2 we present the goals for the development of our software prototype. Afterwards, in section 3 we present preliminaries, most notably in regard to workflows with spring batch. Subsequently, in section 4 we present the concepts that drive our curation system followed

[1] Renée J. Miller, "Big Data Curation" in 20th International Conference on Management of Data (COMAD) 2014, Hyderabad, India, December 17-19, 2014

by a description of its architecture and implementation in section 5. In section 6 we detail how Lodtenant can be obtained and used, followed by section 7 that demonstrates our system in a complex geocoding scenario. Related work is presented in section 8. Finally, we conclude this deliverable in section 9 where we also discuss limitations and future work.

2 Goals

Our goals are summarized as follows:

- Our system must work with SPARQL infrastructure, i.e. it must support SPARQL queries, update statements and endpoints.
- Representing transformations with SPARQL statements is preferred over other approaches, such as using programming languages. However, scripting should be possible, such as with JavaScript.
- Applications must be capable of submitting jobs to the workflow engine and query its state.
- Curation actions should as much as possible be represented as sequences of SPARQL (Update) queries.
- The framework should offer a high degree of flexibilty in arranging curation actions.
- The system must be capable of logging all remote HTTP requests, such that e.g. excessive number of requests can be identified early.
- The system must be capable of keeping track of changes made to SPARQL endpoints.

3 Preliminaries

In this section, we summarize important notions and relevant aspects of the used technologies.

SPARQL endpoint and SPARQL service

We adopt the terminology of the SPARQL Service Description standard[2], which states: A *SPARQL endpoint* is a *URI* at which a SPARQL service resides. A *SPARQL service* is a service implementation that supports the SPARQL protocol.

3.1 Spatial Vocabularies

Despite the strive of the Linked Data community to endorse standard vocabularies, even nowadays, spatial data in the Semantic Web can take many forms. In some cases, prominently DBpedia, the same unit of geometric information is redundantly materialized using different spatial vocabularies and datatypes. For instance, the resource for Leipzig[3] has its reference point represented component-wise using WGS84's[4] *geo:lat* and *geo:long* as well as a single literal using the the non-standard *geo:geometry* property. LinkedGeoData[5] at present still mixes GeoVocab[6] with GeoSPARQL[7]. As a consequence, a curation tool needs to provide support for virtually arbitrary vocabulary mixes, as to allow establishing compatibility with existing datasets and dependent tool chains.

3.2 Batch Workflows with Spring Batch

Spring Batch[8] is a Java framework for creating batch processes. A depiction of its reference model is shown in Figure 2. The main entity is the *Job* which comprises a set of *Steps*. A step encapsulates an action in a batch workflow. There are two essential types of steps: the *tasklet* step and the *chunked* step.

A tasklet step simply encapsulates an action about which spring batch makes no assumptions. In our case, basic SPARQL update queries are realized at steps of this type.

Chunked steps feature additional structure for reading, processing and writing items. Items are read using an *ItemReader*, processed with an *ItemProcessor*, and the processed result is written out using an *ItemWriter*. For these kind of steps, Spring batch readily ships with tooling to remember the last item read and the last position written to. This information can be used for displaying status messages as well as restarting jobs after failure or suspension. It is also possible to create steps that only run as a whole, such as the execution of a SPARQL Update query. Launching a job works as follows: First, a job object has to be created from a job specification, e.g. represented as an appropriate spring XML document. A job thereby acts as a template which can be instantiated by parameterization. For this reason spring batch introduces the classes *JobParameters* and *JobInstance*. For example, consider a job that creates RDF conversions of a monthly updated CSV file published on the Web. In this case, the job would comprise the necessary steps for performing the conversion, whereas for each month there would be a job instance. Finally, launching a JobInstance yields a *JobExecution*, whereas

[2] http://www.w3.org/TR/sparql11-service-description/
[3] http://dbpedia.org/resource/Leipzig
[4] http://www.w3.org/2003/01/geo/wgs84_pos#
[5] http://linkedgeodata.org/triplify/node1681920624
[6] http://geovocab.org/
[7] http://www.opengeospatial.org/standards/geosparql
[8] http://projects.spring.io/spring-batch/

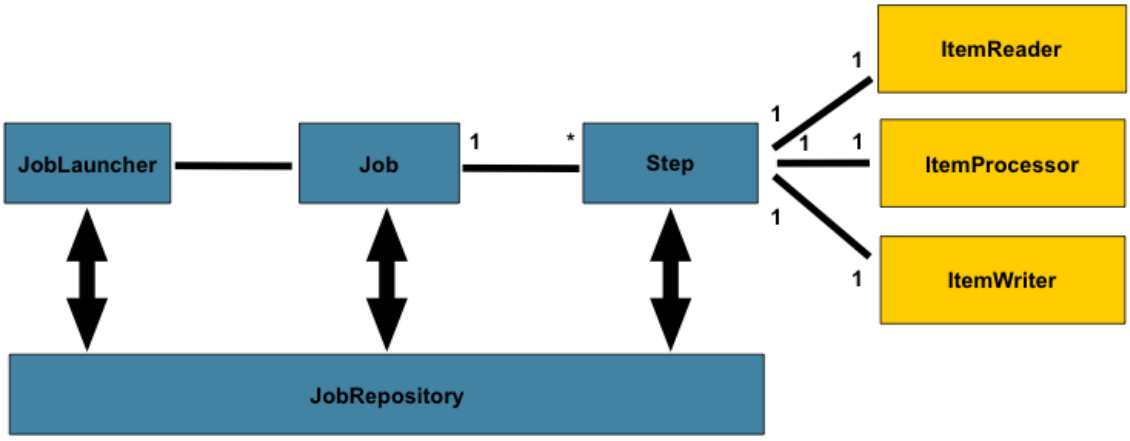


Figure 2: The Spring Batch Reference Model

a *StepExecution* is created for each step being run. Spring batch uses the execution objects to keep track of the jobs' and steps' states, such as the start and end of execution and success/failure events. Further, jobs may store their own custom values in *JobExecutionContext* and *StepExecutionContext* objects associated with the execution, respectively. All this information is usually stored in a (relational) database backend.

The most relevant features of Spring Batch for our work are: (a) the state of batch jobs are stored in a relational database which, using RDB2RDF solutions, can be exposed as a SPARQL endpoint, and (b) it comes with support for *chunked processing*, which e.g. opens the possibility to continue jobs after temporary problems, such as downtime of SPARQL endpoints or the exceedance of API usage limits.

Although the spring batch reference model provides a framework for creating batch workflows in general, it does not ship with facilities for RDF data processing. Our contributions in this regard are presented in section 5.

4 Lodtenant Concepts

As Lodtenant is a tool designed for RDF data, SPARQL is the natural choice for a language that facilitates corresponding data access and modification. Consequently, we introduce several SPARQL-based notions in order to support chunked processing and nested lookups of related RDF data.

4.1 SPARQL-based notions

The SPARQL 1.1 specification[9] is a collection of documents that, among other things, defines syntax and semantic for SPARQL query[10] and update[11] statements.

SPARQL Statement

A SPARQL statement can be either a SPARQL query and SPARQL update statement.

SPARQL Concepts

A SPARQL concept (in the following simply referred to as concept) is a pair (p, v), comprised of a SPARQL group graph pattern p (with optional enclosing braces) and a variable v thereof. As such, a concept intensionally describes a set of resources. Concepts can be used to create corresponding SPARQL queries for counting resources and retrieving them. An example for a notion is *{ ?s | ?s a o:Project }*. Creating a standard SPARQL query from a concept can be easily accomplished with the template shown in Listing 1.

```
1 SELECT DISTINCT ${concept.var}
2 WHERE ${concept.pattern}
```

Listing 1: Template for creating a standard SPARQL query from a concept

Binary Relations

A SPARQL binary relation (short: relation) $(p, s, t) \in P \times V \times V$, with P the set of group graph patterns and V the set of SPARQL variables, comprises a SPARQL graph pattern p, a source variable s, and a target variable t. Relations are used to navigate from a concept, i.e. set of resources, to a related one.

Partitioned SPARQL Statements

In order for our workflow system to be capable of processing a large number of items, such as RDF ressources, a workload may need to be partitioned. In this work, we focus on SPARQL CONSTRUCT, INSERT and DELETE statements, which have in common that they operate on sets of quads. We therefore introduce the notion of a partitioned SPARQL statement as a pair (statement, var) where var is used to partition the statement's result RDF graph.

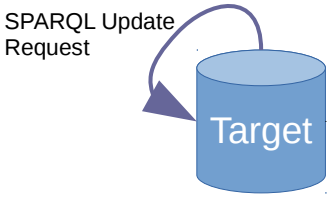
[9] http://www.w3.org/TR/sparql11-overview/
[10] http://www.w3.org/TR/sparql11-query/
[11] http://www.w3.org/TR/sparql11-update/

4.2 RDF Step Processing Models

For processing RDF data as part of workflows, models for representing the individual steps are required. In this section, we devise three step types for transferring data between SPARQL services and performing updates.

Simple SPARQL Update Requests

The most basic step type introduced by Lodtenant is the one for executing a SPARQL update statement on a certain SPARQL service, as shown in Figure 5.




```
{ $sparqlUpdate: {
  name: 'materializeFeatureTypes',
  target: '#{ target }',
  update: 'INSERT { ?s a ogc:Feature } WHERE { ?s ogc:hasGeometry ?g }'
} }
```

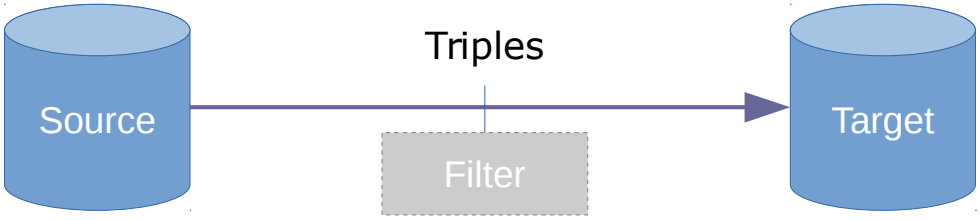
Figure 3: Depiction of the SPARQL update step

Piping RDF data

A SPARQL pipe fetches data from a source sparql service and either inserts or removes it from a target one, as depicted in Figure 4. The most essential configuration parameters are therfore *source*, *target* and *query*. By default, insert mode is assumed. The basic chunked step parameters *name*, *chunk* and *read* default to the shown values and can be adjusted as desired. Note, that it is also possible to provide a SPARQL expression over the variables *?s*, *?p*, and *?o* in order to filter out triples where the expression evaluates to false. The reason this feature was added was to support certain older triple store versions that may yield RDF terms that are not standard conforming.

Performing Diffs with RDF data from distributed sources

The diff step, depicted in Figure 5, enables powerful transformations using SPARQL. Given an initial set of resources by means of resolving a given concept on a SPARQL service, a temporary in-memory graph is built by fetching sets of triples related to each resource. Modifications to that in-memory graph can be made with SPARQL update statements, that may also make use of custom functions registered to the Jena framework. Finally, a diff is automatically created and applied to the target endpoint by comparing the state of the temporary graph before and after its modifications.




```
{ $sparqlPipe: {
  name: 'loadData',
  source: '#{ source }',
  target: '#{ target }',
  query: 'Construct Where { ?s ?p ?o }',
  read: 1000,
  chunk: 1000,
  filter: 'term:valid(?s) && term:valid(?p) && term:valid(?o)',
  delete: false
} }
```

Figure 4: Depiction of the SPARQL pipe step

In many cases it is necessary to combine information from multiple SPARQL datasets that are possibly distributed accross the Web. For example, consider someone having created links between airports in DBpedia and LinkedGeoData and loaded them into a local quad store. Now, as part of a workflow action, he needs to combine information from these three sources.

While the SPARQL standard features the `SERVICE` keyword for basic federated queries, this concept does not interact nicely with Lodtenant's configuration approach: As each SPARQL service is represented as a Java object (bean), it is quite cumbersome to properly extract endpoint and dataset information from the bean and inject this information into a query template string while avoiding the possible introduction of syntax errors.

Our approach is as follows:

- First, one defines an initial set of resources by specifying a concept and a SPARQL service on which to resolve the concept.
- Then, it is possible to (a) fetch triples/quads by executing a set of partitioned CONSTRUCT queries for this set of resources on arbitrary SPARQL services and/or (b) navigate to a related set of resources via a relation again resolved on an arbitrary SPARQL service.
- When navigating to a related set of resources, the second step can be repeated recursively.

As this approach allows one to "hop" from one SPARQL service to another, we refer to this model as a *Hop*. An example is shown in Figure 6: Given an initial set of airport resources on DBpedia, corresponding triples from LinkedGeoData are fetched based on a local store of sameAs links. A JSON representation is shown in Listing 2.

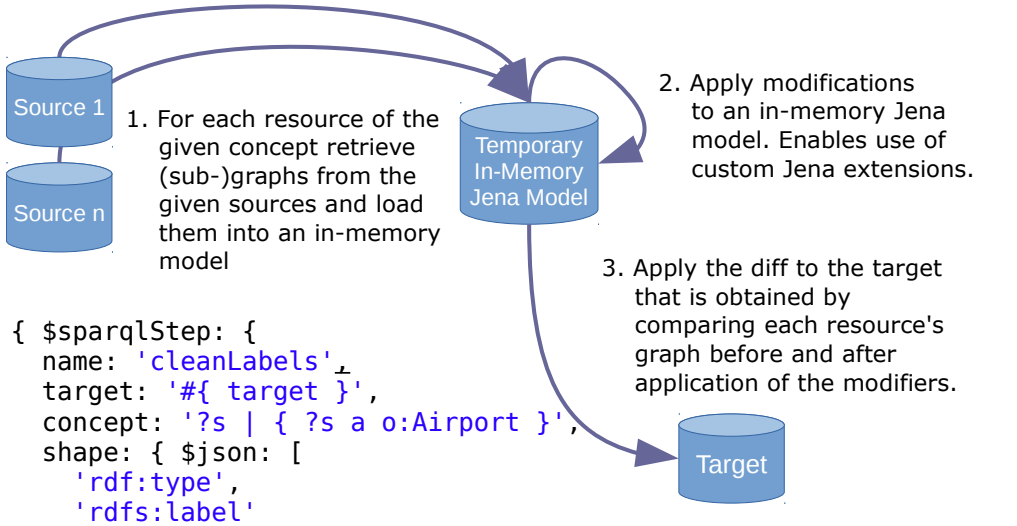


Figure 5: Depiction of the SPARQL update step

```
{
  service: '#{ dbpedia }',
  concept: '?d | { ?d a dbr:Airport }',
  hop: { $hop: {
    queries: [],
    relations: [
      // Relation items: [ relationString, sparqlService, { queries:, relations:
          } ]
      [ '?d ?l | ?s owl:sameAs ?l', '#{ local }', {
        queries: [
          // Query items: [ partitionedQuery, sparqlService ]
          ['?l | CONSTRUCT { ?l ?p ?o }', '#{ linkedGeoData }']
        ],
        relations: []
      }],
  } }
}
```

Listing 2: JSON representation of hops in a Lodtenant specification

Note, that the approach could be extended to support an arbitrary number of variables. Concepts would then be written as:

$$v_1, \ldots, v_n \mid \text{group-graph-pattern}$$

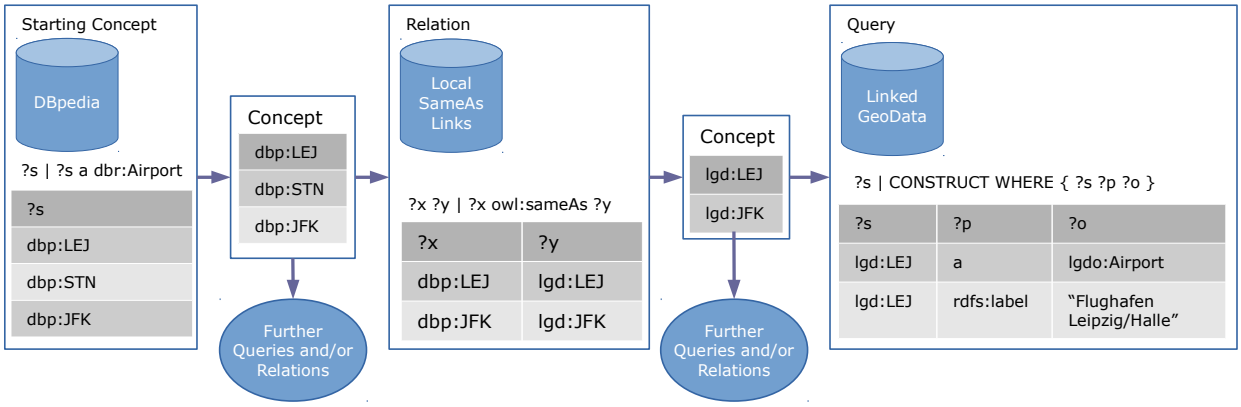


Figure 6: Creating RDF graphs from distributed sources using relations and concepts

Whereas relations would generalize to a mapping between a list of source variables to a list of target variables:

$$s_1, \ldots, s_n \rightarrow t_1, \ldots, t_m \mid \text{group-graph-pattern}$$

However, as in accordance with the Linked Data paradim URIs already *uniquely* identify entities, in constrast to e.g. relational data where composite keys are common, multi-variable support seems to be hardly needed.

It is noteworthy, that another approach would be the integration of a query federation framework, such as TopFed[12]: With this approach, a virtual sparql service could be configured from a set of underlying sparql services. However, in order for these query federation systems to work efficiently, they need metadata about the kind of data the sources contain. Although, in theory, this information can be generated automatically, there are practical constraints in analyzing large datasets, such as DBpedia, in an ad-hoc fashion over a public SPARQL endpoint.

[12] https://code.google.com/p/topfed/

5 Implementation

In this section we first present Lodtenant's architecture, detail relevant components and describe important features.

5.1 Architecture

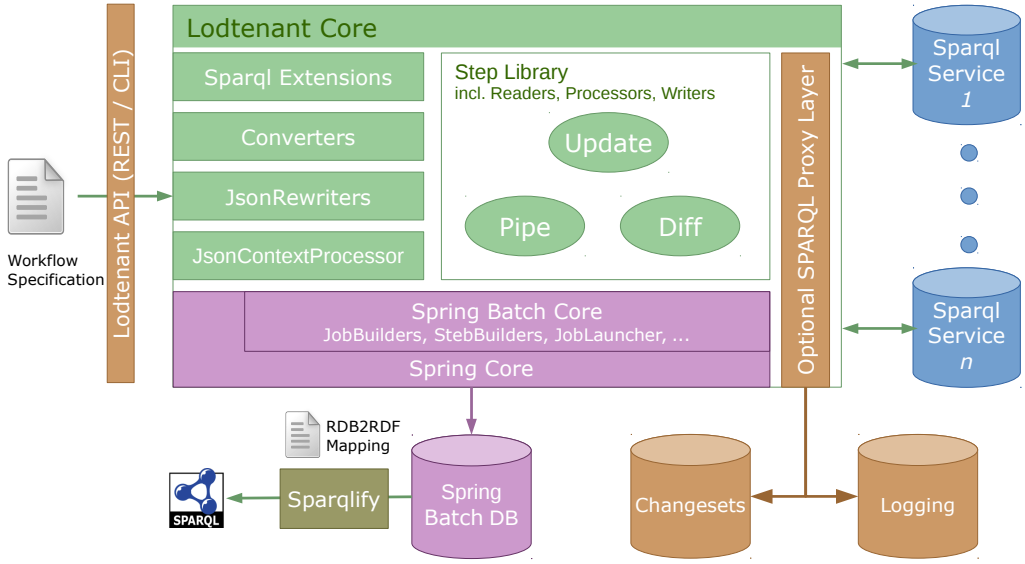


Figure 7: Architecture of Lodtenant

The components of Lodtenant are shown in Figure 7 and are described as follows:

- *Spring Core*: Spring Batch and Lodtenant utilize the well known Spring framework; especially features such as dependency injection and Java config.
- *Spring Batch*: A framework for realizing batch processes. The framework keeps track of executing jobs in a (relational) database.
- *Sparqlify*: Our SPARQL-to-SQL rewriter that enables live access to RDF data generated from the Spring batch's database based on a RDB2RDF mapping definition.
- *JsonContextProcessor*: Our adapter that allows for Spring bean definitions to be written in JSON, with the intention of simplifying creating Web interfaces for the system.
- *JsonRewriters*: A set of simple JSON transformations that enable the use of convenient short hands. For example, the *$sparqlService* shorthand avoids having to recall the corresponding java class *org.aksw.jena_sparql_api.core.SparqlService*.
- *StepLibrary*: A set of factory beans for creating SPARQL-based batch processing step objects.
- *Lodtenant API*: Lodtenant comes with a command line interface and a REST API. Both interfaces enable the management of workflows.

- *Sparql Services*: The primary type of data source Lodtenant is desigened for. Note, that support for other datasources can be easily added by creating/importing appropriate existing Spring beans.
- *Optional SPARQL proxy layer*: When Lodtenant processes the workflow definition, it can determine which SPARQL services are referenced and proxy them. The main use cases for this are *dry-run* and *tracking*. In dry-run mode, update requests are intercepted and changes are applied only in memory. Tracking means, that any changes due to update requests are tracked in a quad store using the Changeset vocabulary. Tracking information can be used to e.g. revert all the changes made by a workflow execution. Note, that these features are in an experimental state.

5.2 A simple Workflow

A workflow specification for loading data from a file is shown in Listing 3. This document defines three beans named *source*, *target* and *job*. The special keys *$sparqlFile*, *$sparqlService* and *$simpleJob* map to Java classes that inherit from Spring's FactoryBean interface, whereas the given properties in the JSON document map to these classes' properties. This means, that once the JSON document is processed, the Java objects that implement the actual behavior are obtained from the configured factory beans. The type *$sparqlFile* wraps a file with Jena's Graph interface, which enables the direct, non-indexed execution of SPARQL queries on that file. While this is inefficient for general SPARQL queries, this approach can be used to create and filter a stream of triples using SPARQL queries that do not contain any joins. A sophisticated example is presented in section 7.

```
1  source: { $sparqlFile: '/tmp/geo-coordinates_en.nt' },
2
3  target: { $sparqlService: ['http://localhost:8890/sparql',
4    'http://dbpedia.org/'] },
5
6  job: { $simpleJob: {
7    name: 'dbpediaLoadingJob',
8
9    steps: [{ $sparqlPipe: {
10     name: 'loadDataFromFileStep',
11     source: '#{ source }',
12     query: 'CONSTRUCT WHERE { ?s ?p ?o }',
13     target: '#{ target }',
14   } }]
15 } }
16 }
```

Listing 3: A simple workflow for loading data from a file

5.3 Logging

Often it is necessary to capture information about certain indicators during workflow execution in order to determine potential abnormalities. In this work, we provide the foundation for detecting excessive and unusually long running HTTP requests. In general, Lodtenant is capable of writing out RDF log messages for each HTTP request. Example log messages are shown in Listing 4.

```
:log-entry-http-20150519-152923-813
  a :HttpLogEntry ;
  dc:created "..."^^xsd:dateTime ;
  :protocol "http" ;
  :host "dbpedia.org" ;
  :path "sparql ;
```

Listing 4: Log messages generated from HTTP and SPARQL requests

```
:log-entry-sparql-20150519-152924-143
  a SparqlLogMsg ;
  :relatedRequest :log-entry-http-20150519-152923-813 ;
  dc:created "..."^^xsd:dateTime ;
  :endpoint "http://dbpedia.org/sparql" ;
  :query "SELECT * { ?s ?p ?o }" ;
  :executionTimeInMs 41 ;
  :success false ;
  :errorMsg "Syntax error at 3:10" .
```

5.4 Tracking changes and Rollback

The Lodtenant is capable of proxying the SPARQL services in a workflow with a change tracking wrapper. At present, this uses a basic component that rewrites SPARQL INSERT / DELETE requests, such that updates are not performed directly. Instead, the proxy layer performs three steps: First, it creates SELECT queries from the WHERE clause of the update requests and uses it to instantiate quads based on the INSERT / DELETE patterns. Second, every quad of the change request, it is checked whether it causes a change in the store, i.e. no events are raised for insertions of existing quads and removals of non-existent ones. Finally, all registered listeners are notified. One of our listener implementations updates a quad store from these events using a slightly modified version of the ChangeSet vocabulary[13], that in addition also keeps track of the SPARQL endpoints on which the changes occurred. Further, the job instance identifier is stored with the change sets, which enables retrieval of all changes performed by a certain job instance and thus rollback. At present, rollback simply undos all of a job instance's tracked changes, regardless of other change sources. Also, there are no special transaction management facilities.

5.5 Important SPARQL Extensions, Java Interfaces and Classes

In this section we summarize important aspects of our implementation. In Table 1, we list our Jena ARQ SPARQL extensions, that can be used in SPARQL update statements in the SPARQL diff step. In Table 2, present essential Java classes. Thereby, the value *jena* in the source column means that this is a class of the Apache Jena project[14], whereas *jsa* stands for our *Jena SPARQL API*[15] project that introduces a vast amount of abstractions and extensions. Further, Table 3 lists available shorthands for simplifying the specification of Java beans in JSON, and Table 4 lists available implicit conversions that will be used during processing of the workflow specification.

[13] http://vocab.org/changeset/
[14] https://jena.apache.org/
[15] https://github.com/AKSW/jena-sparql-api

Function	Description
`http:get(url)`	Perform a HTTP GET request at the given url. The result value's type depends on the content type of the response.
`http:encode_for_qs(str)`	Encode a given string argument for use in a URL's query string.
`json:parse(str)`	Parse a given string as a JSON document. The result is a typed literal with the non-standard datatype *xsd:json*.
`json:path(json)`	Run a JSON path[16] query against the given JSON argument.
`json:unnest(jsonArray)`	This is a property function that takes a JSON array as input, and yields appropriate SPARQL solution bindings for each array item. Property functions are invoked by using their URI in the predicate position of a SPARQL triple pattern, as in `?jsonArray json:unnest ?item`.
`term:valid(term)`	Validates the given term. Especially, if this argument is an HTTP URL, an URL validator will be applied[17].

Table 1: Jena ARQ SPARQL extensions introduced by Lodtenant. All functions raise a SPARQL type error on invalid arguments types or values.

[16] https://github.com/jayway/JsonPath
[17] https://commons.apache.org/proper/commons-validator/apidocs/org/apache/commons/validator/UrlValidator.html

Class	Source	Description
Query	jena	Main class for representing SPARQL queries at the syntax level. A different set of classes is used for the algebraic representation.
Element	jena	Base interface for SPARQL graph patterns.
Expr	jena	Base interface for SPARQL expressions.
UpdateRequest	jena	Base class for representing SPARQL update requests, i.e. INSERT, DELETE and MODIFY.
DatasetDescription	jena	Class for specifying a list of default graph URIS and a list of named graph URIs.
SparqlService	jsa	Interface for accessing query and update capabilities of a SPARQL service.
QueryExecutionFactory	jsa	Interface for preparing the execution of query statements from query objects or strings.
UpdateExecutionFactory	jsa	Interface for preparing the execution of update statements from query objects or strings.
Concept	jsa	Class that represents a SPARQL concept as described in section 3.
Relation	jsa	Class that represents a SPARQL relation as describen in section 3.
Modifier<T>	jsa	Utility interface for the application of arbitrary modifications to an object. Defines the method `void modify(T obj)`.
FactoryBeanStepSparqlUpdate		Factory bean class for steps that perform SPARQL updates
FactoryBeanStepSparqlPipe	jsa	Factory bean class for steps that perform SPARQL pipes, i.e. transfer triples from a source SPARQL service to a target one.
FactoryBeanStepSparqlDiff	jsa	Factory bean class for steps that implement our SPARQL diff approach.

Table 2: Excerpt of Lodtenant's essential Java interfaces and classes

Shorthand	Description
$sparqlService: [endpoint (SparqlService), datasetDescription (DatasetDescription), authenticator (Authenticator)]	Define a SPARQL service with a preconfigured endpoint URL, SPARQL dataset and authenticator.
$sparqlFile: fileName (File)	Wraps a given file as a read only (i.e. queryable) sparql service.
$dataSource	Configures a spring DriverManagerDataSource object for connecting to a relation database.
$simpleJob	Creates a spring batch SimpleJob object for executing a simple sequence of steps.
$sparqlPipe	Step for transferring data between sparql services.
$sparqlUpdate	Step for executing a SPARQL update query.
$sparqlDiff	Step for performing a SPARQL-based diff.
$hop: json (Gson)	Create an object for retrieval of distributed SPARQL data from the given JSON specification.
$json: json (Gson) ...	Creates a Gson object from the JSON subtree given as the argument.
$prefixes: { p: url }	Utility to set global prefixes.
$sparqlMacros: { 'functionURI(args)': 'definitionExpression' }	Utility to create custom SPARQL functions, as shown in Listing 5

Table 3: Overview of Lodtenant's JSON short hands

```
1 {
2   sparqlMacros: { $sparqlMacros: {
3     'o:nominatim(?x)': 'http:get(concat("http://nominatim.linkedgeodata.org/?
          format=json&q=", http:encode_for_qs(?x)))'
4   } }
5 }
```

Listing 5: Definition of SPARQL function macros

The JSON rewriters are part of the module *org.aksw.jena_sparql_api.batch.json.rewriters*[18].

[18] https://github.com/AKSW/jena-sparql-api/tree/master/jena-sparql-api-batch/src/main/java/org/aksw/jena_sparql_api/batch/json/rewriters

Source type	Target type	Description
String	Concept	Parse a string of format "var | groupGraphPattern" as a concept.
	DatasetDescription	Create a dataset description whose default graph is that of the given string.
	Expr	Parse the string as a SPARQL expression.
	Modifier	Parse the string as a SPARQL update statement and wrap it as a Modifier
	Query	Parse the string as a SPARQL query statement.
	Relation	Parse a string of format "sourceVar targetVar | groupGraphPattern" as a relation.
	UpdateRequest	Parse the string as a SPARQL update statement.
SparqlService	QueryExecutionFactory	Returns a sparql service's query execution factory.
	UpdateExecutionFactory	Returns a sparql service's update execution factory.
List<String>	DatasetDescription	Create a DatasetDescription with the given list of strings becoming its default graphs.

Table 4: Overview of the sparql-batch module's default converters

5.6 Dry run mode

This is an experimental feature. When the dry run mode flag is set, Lodtenant wraps all sparql services, such that SPARQL update modifications are performed in memory. Queries are then executed by Jena's ARQ engine against a virtual SPARQL dataset comprised of the original endpoint's data and the in-memory changes. Technically, this is accomplished with simple custom implementations of Jena's DatasetGraph `find` and `findNG` (ng = named graph) methods. While this approach is not expected to scale to large amounts of data, it is still useful for testing and debugging purposes. Note, that ultimatively the step implementations need to respect the dry run flag.

6 Obtaining and using Lodtenant

6.1 Setup

Lodtenant is available on Github[19] and the Linked Data Stack Debian Repository[20]. The latter can be set up following the instructions on the webiste, which are duplicated in Listing 6 for reference.

```
# download the repository package
wget http://stack.linkeddata.org/ldstable-repository.deb
# install the repository package
sudo dpkg -i ldstable-repository.deb
# download the Linked Data Stack public key in order to trust the repository
sudo apt-key adv --keyserver keyserver.ubuntu.com --recv-keys B3A3E13B
# update the repository database
sudo apt-get update
```

Listing 6: Setting up the Linked Data Stack Debian Repository

Lodtenant's command line and Web interface can then be installed with the commands shown inListing 7. By default, the Web interface will be hosted under http://localhost:8080/lodtenant.

```
sudo apt-get install lodtenant-cli
sudo apt-get install lodtenant-tomcat7
```

Listing 7: Installing Lodtenant from the Linked Data Stack

Both packages depend on the `lodtenant-common` package that sets up spring batch database, virtuoso, and provides appropriate default configuration files under `/etc/lodtenant/`. The Tomcat 7 Web application in addition uses by default the configuration at `/etc/tomcat7/Catalina/localhost/lodtenant.xml`.

6.2 The Lodtenant Command Line Interface

The main steps necessary to execute a workflow is to provide a job specification, instantiate it with a set of parameters and launch it, as shown in Listing 8.

[19] https://github.com/GeoKnow/Lodtenant

[20] stack.linkeddata.org/components/debian-repository/

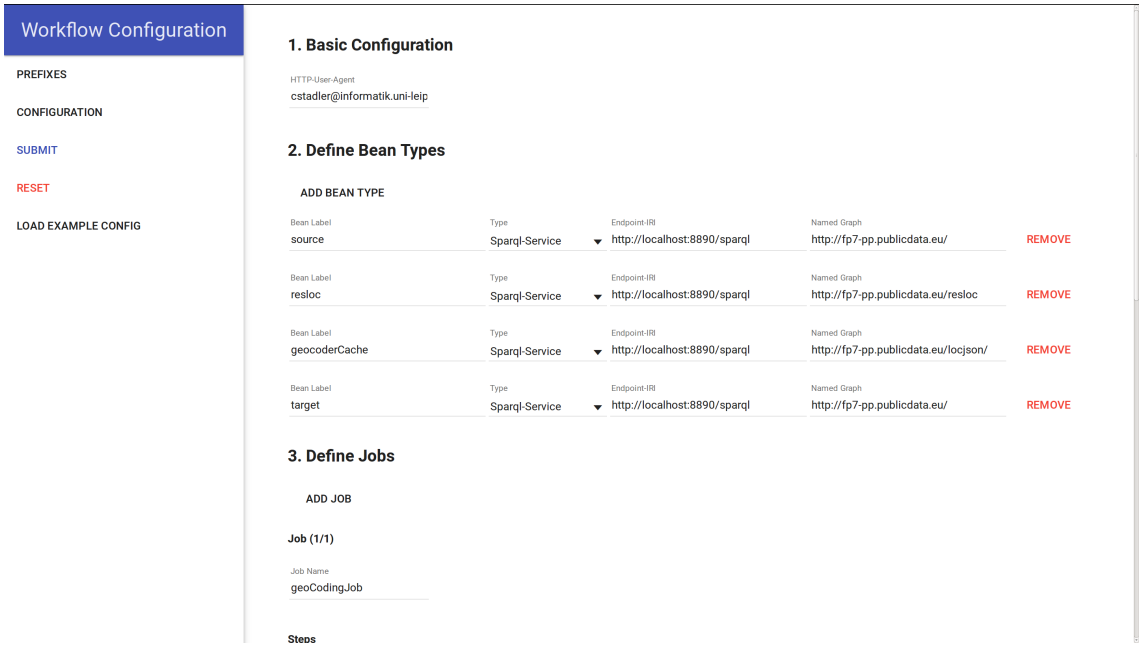


Figure 8: Screenshot of the Lodtenant Web application

```
lodtenant --register workflow.js --prepare '{}' --launch

# Enable dry run mode
lodtenant --dry-run ...

# Query the status of the spring batch database
lodtenant --query 'Select * { ?s ?p ?o }'

# This statement will try to undo changes that were tracked for the given
# job instance id
lodtenant --undo jobInstanceId

# Control the execution of a specific job instance
lodtenant --{start, stop, restart, abort} jobInstanceId
```

Listing 8: Lodtenant Usage Examples

6.3 The Lodtenant Web Application

The Web application, depicted in Figure 8, simplifies the creation of workflows by offering form-templates for the different processing steps. Unsurprisingly, the form fields closely match those of the JSON documents. At present, the Web application only offers a form-based graphical user interface for exporting the JSON workflow definition. Future work will be a closer integration with spring batch's functionality, such as a dashboard of recent and current jobs, support for submitting jobs directly and controlling their life cycle.

7 Enrichment and Geocoding with Lodtenant

In this section, we introduce a dataset on which we then demonstrate the use of Lodtenant for solving several curation-related tasks. As an example, we use the dataset about *ICT research projects under EU-FP7*[21]. An RDF version of this dataset is hosted at[22]. Listing 9 shows an example resource of that dataset, which is explained as follows: As part of FP7, a research project named *LATC* was funded, in which a set of partners, among them *InfAI*, received a certain amount of money. Each parter may receive additional amounts of funding from other projects. Each partner has an address comprised of URIs denoting the city and country. By concatenation of these address components, a location string was derived that was sent to an OpenStreetMap-based geocoder service. Using the data from the reponse, an *owl:sameAs* link to a resource in OpenStreetMap could be established, which in turn contains the geocoordinates.

```
r:project/256975
  rdfs:label "LATC" ;
  o:funding r:funding/256975-998797652 .

r:funding/256975-998797652 ;
  rdfs:label "Funding of partner INSTITUT FUR [...] INFORMATIK EV in project
      LATC" ;
  o:partner r:partner/998797652 .

r:partner/998797652
  rdfs:label "INSTITUT FUR ANGEWANDTE INFORMATIK EV" ;
  r:address/998797652 .

r:address/998797652
  o:city r:city/Germany-LEIPZIG ;
  o:country fp7-pp-r:country/Germany .

r:city/Germany-LEIPZIG
  rdfs:label "LEIPZIG" .

r:country/Germany
  rdfs:label "Germany" .

r:city/Germany-LEIPZIG
  owl:sameAs lgd:node1681920624 .

lgd:node1681920624
  geo:lat "51.340462"^^xsd:double
  geo:long "12.374705"^^xsd:long
```

Listing 9: An example resource of the FP7 research project dataset. For brevity, QNames with '/' in their local names were still prefixed, although this is not standard turtle syntax.

The basic set up of our example geocoding workflow description is shown in Listing 10. First, custom prefixes are registered, followed by SPARQL services and the job

[21] fhttp://open-data.europa.eu/en/data/dataset/ict-research-projects-under-eu-fp7
[22] http://fp7-pp.publicdata.eu/sparql

definition. Note, that all prefixes of the RDFa initial context[23] are available by default.

```
{
  prefixes: { $prefixes: {
    'o': 'http://fp7-pp.publicdata.eu/ontology/',
    'cache', 'http://example.org/cache',
    'tmp', 'cache', 'http://example.org/tmp'
  } },

  source: { $sparqlService: [ 'http://fp7-pp.publicdata.eu/sparql',
    'http://fp7-pp.publicdata.eu/'] },

  geocoderCache: { $sparqlService: [ 'http://localhost:8890/sparql',
    'http://fp7-pp.publicdata.eu/locjson/'] },

  target: { $sparqlService: ['http://localhost:8890/sparql',
    'http://fp7-pp.publicdata.eu/'] },

  job: { $simpleJob: {
    name: 'geoCodingJob',

    steps: [
      ...
    ]
  } }
}
```

Listing 10: Basic setup

As a first step, we clear any existing data in our target graph, as shown in Listing 11.

```
{ $sparqlUpdate: {
  name: 'clearData',
  target: '#{ target }',
  update: 'DELETE WHERE { ?s ?p ?o }'
} },
```

Listing 11: Clearing the target graph

Next, Listing 12 shows how to transfer all data from the remote store to our local one.

```
{ $sparqlPipe: {
  name: 'loadData',
  source: '#{ source }',
  query: 'CONSTRUCT WHERE { ?s ?p ?o }',
  target: '#{ target }',
} },
```

Listing 12: Loading data from a remote source

We want to freshly geocode the data, hence we need to clear existing geometric data as shown in Listing 7.

[23] http://www.w3.org/2011/rdfa-context/rdfa-1.1

```
{ $sparqlUpdate: {
  name: 'clearLocations',
  target: '#{ target }',
  update: 'DELETE { ?s ?p ?o } WHERE { ?s ?p ?o . Filter(?p In (geo:lat, geo:long, o
      :addressString, tmp:lgdLink )) }'
} },
```

In Listing 7 we enrich the target dataset with address strings that can be directly used for lookups at geocoding services. Note, that for some entities in the dataset only the country is known. For this reason there is extra cosmetic processing to avoid address strings that start with a white space.

```
{ $sparqlUpdate: {
    name: 'createAddresses',
    target: '#{ target }',
    update: '\
INSERT { \
  ?s o:addressString ?a \
} \
WHERE { \
  ?s\
    o:address [\
      o:country [ rdfs:label ?col ] ; \
      o:city [ rdfs:label ?cil ] \
    ] \
  BIND(str(?cil) As ?cils) \
  BIND(concat(?cils, if(strlen(?cils) > 0, " ", ""), ?col) As ?a) \
}'
} },
```

Based on the address strings, in Listing 13 we now create triples that are inserted into the geocoder cache. The subject of these triples are thereby derived from the address strings, such that equal address strings yield equals triples. This means that any information already in the cache merges naturally.

```
{ $sparqlPipe: {
    name: 'createLocationStringResources',
    source: '#{ target }',
    target: '#{ geocoderCache }',
    query: 'CONSTRUCT { ?x cache:addressString ?a } WHERE { ?s o:addressString ?a .
        \
          Bind(uri(concat("http://example.org/location/", encode_for_uri(?l))) As ?x)
              }'
} },
```

Listing 13: Adding address strings to the geocoder cache

The next step is to geocode all address strings that have not been geocoded yet. In the first part of Listing 14 we first select all address strings from the cache for which no corresponding *cache:geocodeJson* triple exists. There are three things to note here: First, when requesting polygons, such as by setting Nominatim's *polygon_text=1*, geocoder responses may become several megabytes in size. On some triple stores this may exceed limitations on the size of RDF literals, hence we added an appropriate filter expression to demonstrate how to skip such large results. Our function *http:encode_for_qs* encodes the

given string for a URL query string. Second, this step will perform geocoding on all non-geocoded address strings in the cache, not just the ones that are part of the source dataset. And third, the chunk set has been set to 1, such that each geocoding result gets immediately stored in the geocoder cache. If the chunk size was higher, in the event of failure, all lookup results for that chunk would be lost.

```
{ $sparqlStep: {
    name: 'geocodeLocations',
    chunk: 1,
    service: '#{ geocoderCache }',
    concept: '?a | { ?x cache:addressString ?a . Optional { ?x cache:geocodeJson ?j
        } Filter(!Bound(?j)) }',
    hop: { $hop: {
        queries: [
            '?a | CONSTRUCT WHERE { ?x cache:addressString ?a }'
        ]
    } },
    modifiers: [
'\
INSERT { \
    ?x cache:geocodeJson ?j \
} \
WHERE { \
\
  { SELECT ?l (http:get(concat("http://nominatim.linkedgeodata.org/?format=json&
      email=cstadler%40informatik.uni-leipzig.de&polygon_text=1&q=", http:
      encode_for_qs(?l))) AS ?tmpJ) { \
    { SELECT DISTINCT ?l { \
      ?x tmp:hasLocation ?l \
    } } \
  } } \
  ?x cache:addressString ?l . \
  Bind(if(strlen(str(?tmpJ)) > 5000000, "[]", ?tmpJ) As ?j) \
}'
]
} },
```

Listing 14: Performing lookups to the geocoder service

Listing 15 shows how for each address string in the source, a lookup for the corresponding JSON document the geocoder cache is performed. The first element of the json array is then used to craft a LinkedGeoData URL, which follows the schema `http://linkedgeodata.org/triplify/{osmType}{osmId}`.

```
{ $sparqlStep: {
    name: 'createLgdUrls',
    chunk: 1,
    source: '#{ source }',
    concept: '?l | ?s o:addressString ?l',
    hop: { $hop: {
      queries: [
        [ '?l | CONSTRUCT WHERE { ?x tmp:geocodeJson ?j ; cache:addressString ?l }',
            '#{ geocoderCache }']
      ],
      relations: [
        [ '?l ?s | ?s tmp:location ?l', {
           queries: [
             [ '?s | CONSTRUCT WHERE { ?s cache:addressString ?l }', '#{
                 geocoderCache }']
        ] } ]
      ]
    } },
  modifiers: [
'\
INSERT { \
    ?s owl:sameAs ?z \
} WHERE { \
  ?s tmp:location ?l . \
  ?x tmp:hasLocation ?l . \
  ?x tmp:geocodeJson ?j . \
  BIND(json:path(?j, "$[0].osm_type") AS ?osmType) \
  BIND(json:path(?j, "$[0].osm_id") AS ?osmId) \
  BIND(uri(concat("http://linkedgeodata.org/triplify/", ?osmType, ?osmId)) AS ?z) \
}'],
    target: '#{ target }'
} },
```

Listing 15: Enrich the original dataset with sameAs links to LinkedGeoData

Finally, Listing 16 demonstrates how to convert geometric information represented in the WGS84 vocabulary to GeoSPARQL.

```
{ $sparqlUpdate: {
    name: 'wgs84ToGeoSPARQL',
    target: '#{ target }',
    update: '\
INSERT { \
  ?s geo:geometry ?g . ?g ogc:asWKT ?w \
} WHERE { \
  SELECT \
   ?s \
   (uri(concat(?s, "geometry")) As ?g) \
   (concat("POINT(", str(?x), ", ", str(?y), ")") As ?w) { \
      ?s geo:long ?x ; geo:lat ?y. \
   } \
} '
} }
```

Listing 16: Additionally serialize centroid information using the GeoSPARQL vocabulary

8 Related Work

There are severale ETL tools available. In this section we review tools which we consider most relevant to our work: GeoKettle[24], *OpenRefine*[25], Unified Views[26].

GeoKettle

is an ETL tool for spatial data, hence it can read data from various sources, apply transformations and write out the data to various targets in various formats. Also, it ships with a graphical workflow designer. Its functionality clearly overlaps with that of spring batch. An essential difference is, that GeoKettle is more of an application for which plugins can be written, whereas spring batch is a framework geared towards integration into other applications. In fact, most of Lodtenant's functionality is part of our Jena SPARQL API library project and thus readily available for reuse in other projects.

The most important contribution of our work are the design of SPARQL based workflow steps and our SPARQL function extensions. In principle it appears, that these concepts can be implemented in most of the ETL tools, however, our choice of Spring Batch is motivated by the facts that this framework was already in use in the consortium, hence additional workflows could be monitored with existing infrastructure, as well as that Spring batch is itself a well known framework for ETL processes with lots of available tooling. Many features of GeoKettle related to spatial data can be accomplished with appropriate SPARQL function extensions, such as setting the Spatial Reference Identifier of geometries.

OpenRefine

originated from the *GoogleRefine* tool which was handed to the Open Source community. In essence it features a tabular data model, similar to spread sheet applications such as Excel, however with the twist that cells contain JSON objects. This approach facilitates easy addition of additional information to cell values. Expressions and plugins can be used to compute new cell values and transform existing ones. For example, given a column that contains names of entities such as countries, named entity resolution plugins can be used to resolve these names to DBpedia URLS and associate those with the original cell values. While there exist plugins for loading RDF data into tabular model and writing the data back to RDF, our approach allows modifying RDF data with SPARQL without the need to transform data to and from a tabular structure, and thus removes one layer of indirection.

Unified Views

is an ETL tool/workflow engine developed originally in the LOD2 project. It is similar to GeoKettle, OpenRefine and our work. It features an extensive set of *data processing units* (DPUs) for transforming and enriching RDF data. However, to the best of our knowledge, it neither exploits SPARQL as a scripting language as we did in this work, nor does it offer the spring batch integration.

[24] http://sourceforge.net/projects/geokettle/
[25] http://openrefine.org/
[26] http://unifiedviews.eu/

9 Conclusions and Future Work

In this deliverable we presented Lodtenant, an ETL workflow system based on spring and spring batch with several extensions for exploiting SPARQL infrastructure. We created a JSON processor for spring bean definitions with the goal of simplifying interactions between a Web frontend and Lodtenant's backend, and a first version of such a Web application has been created. We introduced a set of spring batch step types dedicated to SPARQL processing, namely sparqlPipe, sparqlUpdate and sparqlDiff. Thereby, a model for combining data from multiple SPARQL services has been devised. Further, we introduced several SPARQL function extensions, including a JSON datatype, the functions `json:path`, `json:unnest` for JSON manipulation and the function `http:get` for retrieving remote data. We demonstrated the combined use of these components in a geocoding workflow example, where we also showed the possibility to make use of arbitrary spatial vocabularies.

Future work will continue in the following directions: First, existing features will be stabilized. For instance, logging and error reporting could be made more user friendly. Second, some of the presented concepts could be enhanced, such as adding support for additional parameters in the SPARQL diff step for better control over which triples are to be added and/or removed. Another example is the optimization of the SPARQL pipe step by exploiting some of a quad store's vendor specific functionality. Finally, our plan is to achieve a better integration of the Lodtenant Web application with the spring batch framework, and to also integrate the *quartz scheduler*[27] for recurrent job execution.

In the Big Data space, stream processing is a popular approach for achieving scalability. And in fact, batch processing can be seen as a subset of stream processing: The itemReader acts as a source, the itemWriter as a sink and in between there is the itemProcessor. However, this model does not allow e.g. passing items from a source down on different processing paths, that possibly eventually write to multiple sinks.

Apache Storm[28] is a framework that supports the definition of complex processing topologies. Data sources are thereby called *spouts*, and yield *tuple* objects, which can be passed to processing units called *bolts*. Bolts in turn can act as tuple sources for other bolts. Spring XD[29] is a recent competitor to Storm that promises to unify batch and stream processing. Therefore, we intend to investigate how our approach could be implemented in this framework. As our work already builds on spring batch, it appears that this can be done with little efforts.

However, it is important to note, that taking advantage of these systems at present come at the expense of increasing the complexity of a workflow definition, such as the need for highly specialized processing steps rather than our proposed uniform processing with SPARQL statements.

[27] https://quartz-scheduler.org/
[28] https://storm.apache.org/
[29] http://projects.spring.io/spring-xd/