Collaborative Project

GeoKnow - Making the Web an Exploratory Place for Geospatial Knowledge

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

Deliverable 4.3.1
Concept for Public-Private Co-Evolution

Dissemination Level	Public
Due Date of Deliverable	M20, 31/07/2014
Actual Submission Date	M22, 31/09/2014
Work Package	WP4, Spatial-Semantic Browsing, Visualisation and Authoring Interfaces
Task	T4.3
Type	Prototype
Approval Status	Approved
Version	1.0
Number of Pages	23
Filename	D4.3.1_Concept_for_public-private_Co-Evolution.pdf

Abstract:
This deliverable presents concepts and a prototype for managing and synchronizing changes between different versions of private and public interlinked datasets.

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

History

Version	Date	Reason	Revised by
0.1	11/08/2014	Initial version of this deliverable	Matthias Wauer
0.2	13/08/2014	Outline, initial structure, initial content	Matthias Wauer
0.3	15/08/2014	Changesets content	Bernd Eickmann
0.4	20/08/2014	Dataset Description content	Bernd Eickmann
0.5	28/08/2014	Chapters completed	Matthias Wauer
0.6	03/09/2014	Prepared for peer review	Matthias Wauer
0.7	10/09/2014	Peer review	Valentina Janev
0.8	16/09/2014	Final changes w.r.t. review comments	Matthias Wauer
1.0	18/09/2014	Final version approved	Claus Stadler

Author List

Organization	Name	Contact Information
Unister	Matthias Wauer	matthias.wauer@unister.de
Unister	Bernd Eickmann	bernd.eickmann@unister.de

Executive Summary

Linking internal data with the geospatial open Linked Data created in the course of GeoKnow is one of the major opportunities for companies arising from this project. Since each data set evolves separately, changes between interlinked datasets have to be synchronized somehow. This deliverable presents a first approach to building an Enterprise Knowledge Hub capable of describing, tracking, and synchronizing changes of different versions of private and public data sets. The component implemented as part of this task, which will be integrated into GeoKnow Generator until the next deliverable 4.3.2, provides several services via a REST API.

Table of Contents

1 Introduction

Most datasets evolve. Triples or entire resources are added or modified over time. Consequently, interlinked datasets have to adapt to these changes in various ways. The implications of a modification depends on the relationships between datasets and their resources.

In general, a dataset which is the result of some process on a source dataset has to be updated when the source dataset is modified. However, this can be unfeasible for entire large datasets. As a result, depending on the actual processing it can be sufficient to only revisit the resources that have actually changed. For example, if a dataset C was created by merging the datasets A and B using some already known interlinks L, then a change in a certain resource r of dataset A can only lead to a change in the merged resources in C which include properties of r. Unfortunately, for many processes it isn't that easy to decide which resources have to be adapted due to changes in a source dataset. Mostly though, there are two primary use cases:

1. a dataset has been updated to a new version, and the differences between these versions should be published so mirrors and consumers of the dataset can be updated without importing or processing the entire dataset again.
2. erroneous data in a dataset (which can also result from some process) has been detected and fixed using manual or automatic methods, and the resulting changes should be applied when the dataset is created again.

In the scope of the GeoKnow use cases, we further have to address the distinction between *public and private* datasets. There are very different processes regarding error correction for these types of datasets. While private datasets in companies usually involve some final approval of changes by responsible persons, approval of error corrections to public datasets can be based on community votings. Regardless of that, private datasets involve access restrictions which can be very elaborate. This has to be considered by a generic approach for dataset change management. While all dataset providers should benefit from error corrections, keeping track of private data and preventing disclosure of such data is a requirement of the concept to be developed. As a result, *Co-evolution* describes the synchronized evolution of spatial Linked Open Data sources and company internal datasets.

In this deliverable, we analyse the concepts related to describing and versioning datasets, managing and handling individual changes, and publishing these changes in Section 2. We then conceptualize a basic architecture and services required for these tasks in Section 3. Finally, we describe the current state of the implementation in Section 4.

1.1 Requirements

The GeoKnow Revised Common Requirements Specification (Deliverable D1.2.1) defines all general and use-case-specific requirements towards the functionality, performance, security, interfaces and data formats with regards to the GeoKnow Generator and its components. For the co-evolution related components, the following requirements have to be considered:

Table 1: Common Requirements for Public-private Co-evolution

Requirement ID	Short Description	Relevance for Public-private Co-Evolution
FR-U6	GeoKnow Generator has to be able to process subsets of datasets.	Co-Evolution services have to support processing of subsets.

Table 1 – *Continued from previous page*

Requirement ID	Short Description	Relevance for Public-private Co-Evolution
FR-B8	Users can select a subset of the configured data sources for each context.	(same as above)
DR-U1	GeoKnow Generator has to keep track of the coarse-grained provenance of a dataset.	Co-Evolution services have to support named graphs as contexts of changes, as well as modification metadata like author, date etc.
DR-U2	GeoKnow Generator has to keep track of the fine-grained provenance of a dataset.	(same as above)
IR-B1	GeoKnow Generator functionality can be accessed via REST API.	Co-Evolution services have to provide a REST API.
FR-U3	GeoKnow Generator has to integrate knowledge from various public and internal sources in RDF.	Co-evolution services have to be designed such that all of these sources can be addressed.
FR-B5	GeoKnow has to integrate internal (private) and external (public) data.	Co-Evolution services have to be aware of the access restrictions.
FR-U9	GeoKnow Generator has to support highly dynamic information.	Co-Evolution should be able to support dataset iterations quickly, without the need to process entire datasets.
SR-B6	All GeoKnow components have to support secure authorization and communication.	Co-Evolution services must be provided using HTTPS and secure authorization methods.
PR-S5	Scalability: the components should be able to scale operations across several GIS engines.	Co-Evolution services should support deployment on multiple nodes and transactional management of changes.
FR-U2	GeoKnow Generator has to log every processing step ...	Co-Evolution services have to support configurable logging.
FR-R5	Provide abstraction of the complete workflow independent from RDF.	Co-Evolution services should be provided with REST representations other than RDF for reasonable operations.

In summary, the primary technical requirements towards co-evolution services are:

- REST Web services which enable creating and modifying change requests, applying, and synchronizing changes; i.e., changeset management, changeset application, and changeset synchronization services,
- support for defining context, e.g., using named graphs and specific endpoints, but ideally a more comprehensive dataset description via the GeoKnow Generator dataset management component, and
- authentication options, ideally via the GeoKnow Generator access and authentication management component.

1.2 Goals

This deliverable is the first of a total of two deliverables in Task T4.3 on Public-private Co-Evolution. Consequently, it will present a concept for tracking, applying and distributing changes to datasets and an initial prototype implementation of the primary services. The second deliverable will then include a complete implementation of these services, integrated within the GeoKnow Generator and front-end components. Thus, in this report we focus on the following aspects of this task:

- Examining the State of the Art of dataset management and description, including the handling of different versions,
- Standard methods for tracking and applying modifications, and approaches to conflict resolution,
- Design of services for Public-private Co-Evolution architecture, and
- Prototype development of initial services.

As a non-goal, in this deliverable, we do *not* plan to implement an entire knowledge hub and entirely automated change management across different dataset providers.

2 Concepts

In this section, we describe general concepts and approaches to the following issues related to Public-private Co-Evolution:

- Dataset versioning,
- Dataset description, and
- Change management of datasets.

2.1 Dataset Versioning and Description

Versioning of information is a major concern for anyone dealing with data represented as RDF triples. Data evolves over time, sometimes triples are added, replaced or modified. A natural way to talk about this is to say that graphs are changing over time[1]. To manage this evolution we store versions of graphs as distinct named graphs. This allows, among others, to compare the output of different production cycles with each other, determining for example the improved quality of interlinking after replacing the interlinking algorithm. This obviously means that all versions of a graph have a different URI.

To preserve the notion that all these versions are versions of the same entity (which is itself no *named* graph), we introduce *graph types* - a graph type being the abstract entity that is exemplified by all versions of a graph. A graph type denotes — so to speak — the set of named graphs containing the same information, and is itself denoted by an URI. Having a stable URI for all versions of a graph enables us to model information about versions in RDF and automate the selection of graphs for each iteration of processing without changing configurations each time. Each processing job determines the instances with the most recent timestamps for all graph-types it uses at runtime.

While all this works well, we would like to move towards using an established vocabulary for data-versioning. In the following sections we shortly describe VoID (Vocabulary of Interlinked Datasets), which revolves around the notion of a dataset, and DataID (DBpedia Data ID).

2.1.1 Dataset Description using VoID

The introduction of datasets in the *vocabulary of interlinked datasets* (VoID) tries to account for the need to talk about RDF data collections in ways that transcend the technical notion of an RDF graph. Strictly speaking, a graph is an unchanging (mathematical) set of triples. Users and publishers of data are less interested in such a set, but want to talk about, e.g., the water quality data for 2013 published by the city of Leipzig, etc. Datasets try to fill that conceptual void (pun intended). A dataset is defined as a "set of RDF triples that are published, maintained or aggregated by a single provider"[2] , thereby adding a social dimension. The expressed intent of the authors goes further in that they "think of a dataset as a meaningful collection of triples, that deal with a certain topic, originate from a certain source or process, are hosted on a certain server, or are aggregated by a certain custodian"[3]. A dataset could be described using Dublin Core Metadata Terms[4] and the FOAF vocabulary[5]. VoiD additionally provides the means to specify how a dataset is published (i.e., the serialization format), how

[1] Strictly speaking, however, this would be wrong. An RDF graph is defined as a set of triples, and thus there is rather a succession of graphs for each modification of the data.

[2] http://vocab.deri.ie/void#Dataset

[3] http://www.w3.org/TR/void/#dataset

[4] http://dublincore.org/documents/2010/10/11/dcmi-terms/

[5] http://xmlns.com/foaf/spec/

the dataset could be accessed (for example, by giving a description of a dedicated SPARQL endpoint or giving the address of an RDF dump), and to add structural metadata. Structural metadata contains the specification of example resources described by the dataset, specification of the namespace of resources described, used vocabulary, possible subpartitions of the dataset, and statistics about the dataset (e.g. number of triples in the dataset). Furthermore, VoID allows to specify how a dataset relates to others. On the logical side, it allows to define linksets: linksets are subsets of datasets that consist of triples specifying links between resources in other datasets. A linkset could be described by specifying the used linking property (e.g. `owl:sameAs`), the source and target datasets of the links (e.g. from DBpedia to Geonames), etc.

The main intent of VoID is to allow publishers of public data to describe it to its potential users. It is not primarily meant to be a versioning tool for producers of data. What is missing for example is a conceptual connection between datasets and named graphs. It is only possible to specify how to publicly access the data, but not which named graphs a dataset corresponds to.

It is therefore possible to specify the location of a file containing the triples a dataset comprises. However, it is not possible to give an RDF description of which triples belong to a dataset, although this can indirectly be defined by incorporating named graphs to use in a described SPARQL endpoint using the SPARQL Service Description vocabulary[6]. This is a serious shortcoming as named graphs are – on the logical level – the way to denote triples belonging together.

2.1.2 Dataset Provenance using PROV

In order to further describe revisions of datasets and the context of the respective changes, the PROV specification[7] can help with "a core data model for provenance for building representations of the entities, people and processes involved in producing a piece of data or thing in the world"[8]. In general, the provenance ontology PROV-O can describe entities (such as resources or data sets), activities related to these entities, and agents associated with them.

For example, a person (an agent) is responsible for executing an activity (such as a GeoKnow interlinking process) which uses two entities (e.g., two datasets) in order to generate a new entity (the resulting interlinked dataset). Different versions of an entity can be described by declaring a new version of an entity a *revision* of another entity. Furthermore, using PROV-O we can state correction activities which were applied when creating a new revision using a qualified association, which can indicate an agent (anything responsible for executing the modification) and a plan (instructions for the modifications). Such resource can be annotated with the time they were generated, started, or finished.

PROV-O properties are a nice addition which represent an approach to allow a fine-grained account of who did what to a dataset. It allows to answer questions like who created a dataset, who republished it, what kind of process was used to compile the dataset from its sources, etc. While this provides a general framework for versioning datasets, PROV-O does not define how the plan (instructions for modifications) has to be modelled: "There exist no prescriptive requirement on the nature of plans, their representation, the actions or steps they consist of, or their intended goals. Since plans may evolve over time, it may become necessary to track their provenance, so plans themselves are entities."[9] Consequently, we will look into how to describe such modifications in detail in Section 2.2.

[6] http://www.w3.org/TR/sparql11-service-description/
[7] http://www.w3.org/TR/prov-overview/
[8] http://www.w3.org/TR/prov-primer/
[9] http://www.w3.org/TR/prov-o/#Plan

2.1.3 DataID

DBpedia Data ID (DataID) is another attempt to model datasets: "DBpedia Data ID is an ontology with the goal of describing LOD datasets via RDF files in a uniform way. Established vocabularies like DCAT, VoID, Prov-O and SPARQL Service Description are used for maximum compatibility."[10]

Additionally, DataID provides properties like `?s` http://dataid.dbpedia.org/ns/core#version http://dataid.dbpedia.org/ns/core#latestVersion to interlink datasets as versions of each other. Note that there is another vocabulary for that purpose, the Dataset Versioning ontology[11] which is derived from the ISO 25964 standard on "Thesauri and interoperability with other vocabularies" that could be applied here. DataID also incorporates a distinction from the DCAT vocabulary[12] between a dataset and a distribution of a dataset. It is possible to link a distribution of a dataset to a named graph (as accessible by a SPARQL endpoint).

2.1.4 Semantic Versioning

When defining versions and version changes, the de-facto standard Semantic Versioning[13] helps to indicate compatible and incompatible changes:

> Given a version number MAJOR.MINOR.PATCH, increment the:
>
> - MAJOR version when you make incompatible API changes,
> - MINOR version when you add functionality in a backwards-compatible manner, and
> - PATCH version when you make backwards-compatible bug fixes.
>
> Additional labels for pre-release and build metadata are available as extensions to the MAJOR.MINOR.PATCH format.

While this approach has originally been created for software development, the same mechanism can be applied for datasets, especially regarding progressions of vocabulary and related properties. As a prominent example, DBpedia basically applies this scheme, with its current version being 3.9 (the PATCH version has been ommitted in this case).

2.2 Changesets

Importing and processing large RDF graphs is inevitably accompanied by idiosyncratic data errors. This is mostly due to the import of faulty data, but quirks in the processing steps could also lead to errors, which cannot always immediately be dealt with in a principal manner. Thus the need for a manual data editing mechanism and a process to propagate the resulting changes to target RDF graphs arises.

2.2.1 Structure of Changesets

ChangeSets[14] are a resource-centric format to represent changes to graphs. A changeset always centers around a resource that is the 'subject of change' and specifies triples to be added or removed that pertain to that resource, which — in most cases — simply means that they contain that resource as a subject or object. The

[10] https://raw.githubusercontent.com/dbpedia/dataId/master/ontology/dataid.ttl
[11] http://purl.org/iso25964/DataSet/Versioning
[12] http://www.w3.org/TR/vocab-dcat/
[13] http://semver.org/
[14] An official description of the vocabulary for changesets could be found at http://vocab.org/changeset/schema.html

set of all triples that stand in that relation to the subject can be said to describe the resource in question and are hence called 'resource description'. The 'change' in changeset are thus primarily changes to resource descriptions.
A changeset is itself an RDF graph that contains both meta-information about the change — who published the change, when was the change published, which resource does the change pertain to, what is the reason of the change —, as well as removal and addition relations to reified statements representing the triples that should be removed or added, respectively.
In the following example in Turtle notation the represented change is the replacement of a faulty `owl:sameAs` link between a DBpedia resource http://dbpedia.org/resource/England and a Geonames resource http://sws.geonames.org/3333218/ with a correct owl:sameAs statement linking it to http://sws.geonames.org/6269131/:

```
@prefix changeset: <http://example.org/ontology/changeset/> .
@prefix statement: <http://example.org/ontology/changeset/statement/>
@prefix cs: <http://purl.org/vocab/changeset/schema#>
@prefix rdf: <http://www.w3.org/1999/02/22-rdf-syntax-ns#> .
@prefix owl: <http://www.w3.org/2002/07/owl#> .
@prefix xsd: <http://www.w3.org/2001/XMLSchema#> .
@prefix dbpedia: <http://dbpedia.org/resource/> .
@prefix geonames: <http://sws.geonames.org/> .

changeset:000565a2-d3cf-0c53-3fee-f2624b8cb0c5  a cs:ChangeSet .
changeset:000565a2-d3cf-0c53-3fee-f2624b8cb0c5  cs:createdDate  "
   2014-08-13T13:59:28+0000"^^xsd:date ;
    cs:changeReason "bugfix"^^xsd:string ;
    cs:creatorName  "a.nonymous"^^xsd:string ;
    cs:removal  statement:000565c5-995a-a37e-3fb8-cf9e1568dc58 ;
    cs:addition statement:000565c5-995b-781f-3fc7-c9b291bf39bc .

changeset:000565a2-d3cf-0c53-3fee-f2624b8cb0c5  cs:subjectOfChange
   dbpedia:England ;
    cs:verified 1 .
statement:000565c5-995a-a37e-3fb8-cf9e1568dc58  rdf:subject dbpedia:
   England ;
        rdf:predicate owl:sameAs ;
        rdf:object  geonames:3333218/.

statement:000565c5-995b-781f-3fc7-c9b291bf39bc rdf:subject dbpedia:
   England ;
            rdf:predicate owl:sameAs ;
            rdf:object  geonames:6269131/ .
```

A changeset contains both triples to be removed and triples to be added. Removals are always carried out before the additions are applied. ChangeSets are partially ordered by a predecessor property http://purl.org/vocab/changeset/schema#precedingChangeSet, that allows to define a succession of changes that are to be applied to a graph. Such a succession is called a *history*. Histories are not circular, as no change could be a predecessor of itself.
Histories could principally branch out in that a changeset could have more than one immediate predecessor. This could, for instance, occur when a resource should be divided into several resources. For example, suppose a merging process conflates the Island of Guernsey with the British Crown dependency Guernsey, an unfortunate mistake we want to correct. But first we want to modify incorrect data pertinent to both. So we first start with a changeset representing common changes, and then perform the split by introducing e.g. a new URI for the island. The divided resources are then further modified in their own continuing modification histories.

One shortcoming of the changeset vocabulary is that it only allows to represent removals of fully specified triples. For example, suppose you want to remove any `owl:sameAs` links that connects your subject of change with a Geonames resource, and then add the correct `owl:sameAs` link to the correct Geonames resource. You will have to specify any such `owl:sameAs`-triple to be removed individually. Or, which is a frequent use case working with geospatial data, you want to remove geocoordinates for a resource *and* all small deviations from that coordinates. In the standard changeset format you have to specify each possible deviation, truncation or rounding individually, which could prove cumbersome.
What is missing is a wildcard mechanism allowing to represent descriptions of triples to be removed. This is covered in Section 2.2.3.

2.2.2 Tracking and Applying Modifications

With the changeset representations in place, we need both a way to modify and publish such changesets, and a mechanism to apply them to RDF graphs.
For editing and publishing we also developed a command line builder tool that creates changesets for different named graphs. The entered changesets are then stored in a change graph corresponding to a given named graph, which was selected by the editor. In order to ensure valid histories, each changeset in such a change graph is per default attached with the predecessor property `http://purl.org/vocab/changeset/schema#precedingChangeSet` to a preceding changeset with the same subject of change that is itself no predecessor to another changeset. If no such changeset is present, then the changeset is inserted without a predecessor. If several changesets with the same subject of change and without a predecessor are present (because of a branching history), the changeset with the most recent creation date is picked as a default. This constitutes a natural default to ensure well ordered changeset histories. An editing tool should potentially also offer the possibility to supersede such a default ordering, which is currently not possible with our command line tool.
To apply the changesets to our graphs we developed the *ChangeSet Application Service*, which is a resource-centric service that receives a resource and a graph (i.e. a set of statements). The graph is then modified according to the set changesets for that resource. This mirrors the resource-centric representation format of changesets.

The first step is to retrieve the first changeset that has the given resource as its subject of change (i.e. the changeset — if any — without a predecessor changeset with the same subject of change) in a potential history of changes regarding that subject that are to be applied to the graph. After applying the represented changes the immediate successors of that changeset — if any — are selected and applied. (That possibly multiple successors are selected is to account for the possibility that the processed history has branches.) The process is then applied to the immediate successors. This process performs consecutively until no changesets are left in the ordering. After all changes have been applied successfully the resulting modified graph is returned.
This workflow obviously puts some restrictions on the histories of the changesets. The ordering must be non-circular, and there should not be more than one history for the same subject of change. Otherwise no single changeset could be identified as a starting point. This puts some restrictions on the editing tools creating changesets and, more specifically, transactional requirements on the Changeset Management Service.
The implemented way of representation and workflow lends itself naturally to a batch-processing of graphs were the process iterates over all resources in one or several input graphs, enriches and processes the resource-description, until finally the changesets are applied to the resource-description, which is the only relevant subset of the final graph as far as the changesets for that subject-of-change are concerned.

For keeping track of modifications, we can also use publish-subscribe methods. RSine[15] is a candidate for such a service on RDF datasets, which is investigated in GeoKnow Task 4.4. We plan to investigate integrating this, or similar solutions, as part of the next deliverable D4.3.2.

[15] https://github.com/rsine/rsine

2.2.3 Conflicts

As indicated earlier, changesets created for a certain version may not be directly applicable to updated versions of a dataset. One possible conflict that could arise with respect to geospatial data arises from slightly different latitude/longitude geocoordinates. There are three potential resolutions:

1. The changeset which could not be applied because a removal did not match an existing statement is ignored entirely. This approach could lead to issues when applying subsequent changesets and is not recommended.
2. A removal of changeset which does not match an existing statement is ignored, but all other removals and additions of that changeset are applied. This approach could equally result in problems, most likely including duplicate contradicting information, and is not recommended.
3. A changeset history of a resource is not applied if one of the changesets could not be applied because a removal did not match an existing statement. This can result in outdated or still-erroneous information, and should only be used with caution on well-defined types of changesets.

All three solutions are error-prone. As a result, the changeset management and application process should be extended using either or both of the following means:

a) Extending the changeset removal representation such that it supports certain patterns to be removed. This could include wildcards, such as 'any object' for a certain predicate which should be removed, or a more elaborate description of statements to be removed, which can be formulated using, e.g., a SPARQL query. Both approaches are likely not entirely backwards compatible with the original ChangeSets specification.

b) Making the changeset application process semi-automatic, such that on detected conflicts a supervisor has to decide how to proceed, i.e., which statements to remove or add. However, since this is a manual synchronous step this could block entire processes, depending on how the changeset application is performed.

3 Architecture Overview

After we discussed different options of capturing and applying changes to datasets, we can now focus on how to provide such functionality to the GeoKnow Generator. This section will focus on the architecture of an *Enterprise Knowledge Hub*, a set of components required to store, manage, apply, and distribute changesets.

3.1 Enterprise Knowledge Hub

The primary components of the Enterprise Knowledge Hub are shown in Figure 1. As a central component, the *data store* contains private and public[16] named graphs with semantic information. In addition to that, it holds changesets for private and public graphs, as well as changeset application logs, i.e., which changeset has been applied to which version of a local dataset. In the GeoKnow Generator, this component is provided by Virtuoso, which also provides a SPARQL endpoint as an interface.

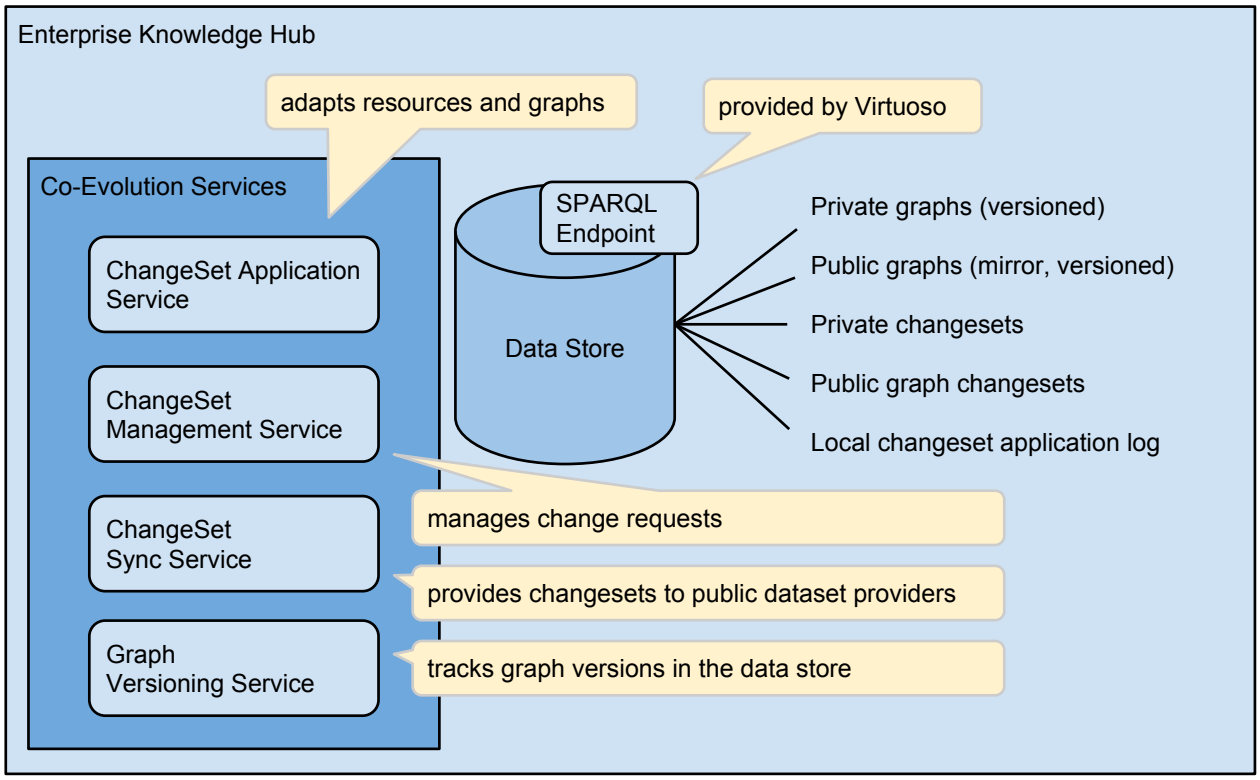

Figure 1: High-level Architecture of the Enterprise Knowledge Hub

Additionally, a number of *Co-evolution services* developed as part of this task in the GeoKnow project provides the functionality for Public-private Co-Evolution. These services will be described in the following section, with additional details on the REST functionality below. In the GeoKnow Stack (see Deliverable D2.1.1 page 33), these services can be considered part of the Evolution/Repair types of components.

[16] mirrored, i.e., containing a copy of a publicly available dataset

3.2 Services

A *ChangeSet Management Service* accepts *change requests*, i.e., requests for modifications to existing data. It makes sure that the changeset history chain is valid, as described in Section 2.2.1.

The *ChangeSet Application Service* can be used to make sure previously recorded modifications get applied to a certain version of a graph. This service can be applied in different ways, typically in one of the following scenarios:

1. Changesets are applied to a set of statements, typically related to a single resource, and the modified set of statements is returned. This mode will be implemented in the initial version of the service.
2. Changesets are applied to an entire graph, either modifying this graph in place or creating a new version of the graph.

A *ChangeSet Sync Service* provides changesets and additional options for tracking changes of a single dataset or between two datasets over a certain period in time. It is primarily used to synchronize changes of local and remote datasets. This service will have an initial service interface design in this first version, with an extension towards publish-subscribe models[17] and an implementation expected for Deliverable D4.3.2.

The *Graph Versioning Service* helps to identify subsequent versions of datasets and their respective sources, as discussed in Section 2.1. The initial implementation will be a very simple approach, which will subsequently be extended to cover more extensive use cases.

3.3 REST API

As specified in the requirements (see Section 1.1), the services have to provide a REST interface. Since GeoKnow Generator can be installed on any host, we define a default context `coevolution/rest` for the service API, i.e., for the demo at `http://generator.geoknow.eu`, the service endpoint of the REST API would be `http://generator.geoknow.eu/coevolution/rest`.

Also, the API documentation should be available alongside the service. Therefore we define a default context `coevolution/rest/api`, i.e., the service API documentation for the demo installation will be published at `http://generator.geoknow.eu/coevolution/rest/api`, with a machine readable JSON description at `http://generator.geoknow.eu/coevolution/rest/api/api-docs`. Details on the REST API is explained below in Section 4.

In order to abstract from the RDF representation (see requirement FR-R5, page 4) the ChangeSets are modeled in the ChangeRequest class. This can be serialized easily into all required representations.

[17] Deliverable D4.4.1 discusses publish-subscribe methods and provides an implementation on top of Virtuoso, which could be applied directly here. The publish-subscribe features of the service should be considered a way to managing these subscriptions.

4 Implementation

In the provided prototype, we primarily use the following technologies to provide the services:

JAX-RS 2.0 , the Java API for RESTful Web Services[18], is applied using the Jersey[19] library[20],

Sesame framework[21] is applied as an implementation of the RDF data model,

Swagger framework[22] is used as an integrated documentation and testing solution,

Spring framework is used as a dependency injection framework, and

Maven is used as a project management solution.

The Maven project `Coevolution`[23] consists of several modules:

coevolution-parent is the Maven parent project (at the root of `Coevolution`),

coevolution-core defines the interfaces and DTOs required for using the services and essential functionality,

coevolution-service implements the REST API services including annotations for creating the Web interface and documentation, and

coevolution-default-implementation provides a basic implementation of the changeset storage and application classes. It is included in the default build profile "geoknow" of the `Coevolution` project, and can be replaced by other implementations using other build profiles.

4.1 Implementation of the REST API

The REST API implementation in `coevolution-service` is split across the following subpackages of `com.unister.semweb.geoknow.coevolution`:

(root) contains a `Starter` class for executing the web application locally with an integrated Jetty server, although the default deployment option is a Java Web application container (.war file),

exceptions defines exceptions thrown by the service, such as `ChangeSetNotFoundException` or *RemovalFailedException*,

providers contains JAX-RS provider implementations of `MessageBodyReader` and `MessageBodyWriter` for (de-)serializing XML representations using the XStream[24] library and Turtle representations using parsers and writers from the OpenRDF RIO library,

providers.helper includes an implementation of an `ApplicationListener` required to scan the XStream annotations on the model classes,

[18] https://jax-rs-spec.java.net/, Java Specification Request JSR-339
[19] https://jersey.java.net/
[20] Originally the Spring-WebMVC framework was tested instead, but this led to issues with Swagger.
[21] http://www.openrdf.org/
[22] https://helloreverb.com/developers/swagger
[23] https://github.com/GeoKnow/Coevolution
[24] http://xstream.codehaus.org/

resource contains the models required for the service, such as `ChangeRequest`, and

services contains the service implementation, i.e., the actual functionality, and

rs contains the JAX-RS application and REST service implementations, along with the documentation annotations for Swagger.

The `resources` package contains the primary data model, as shown in Figure 2.

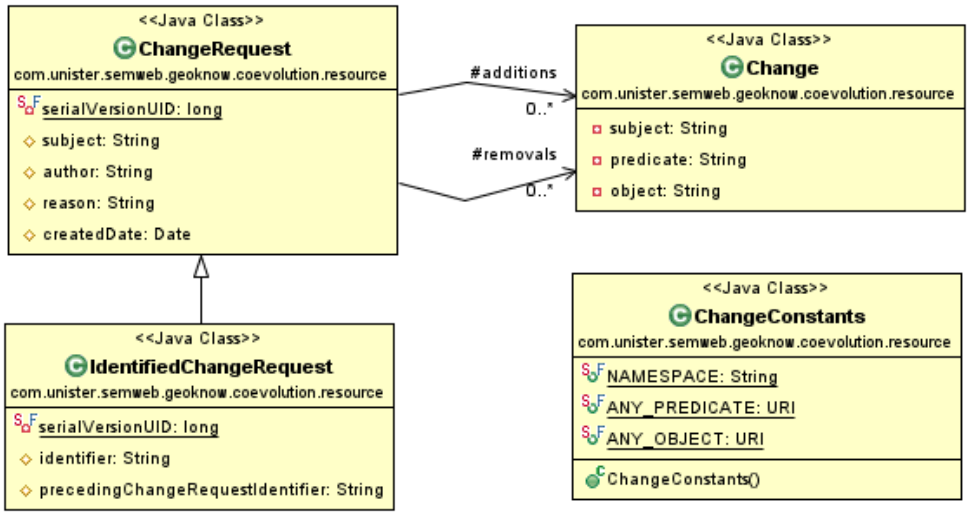

Figure 2: Classes in the `resource` package

When the web application is started, the Swagger library (and the swagger-ui package) provides the API documentation and a REST client for testing the functionality in place, as shown in Figures 3, 4, 5, 6,and 7.

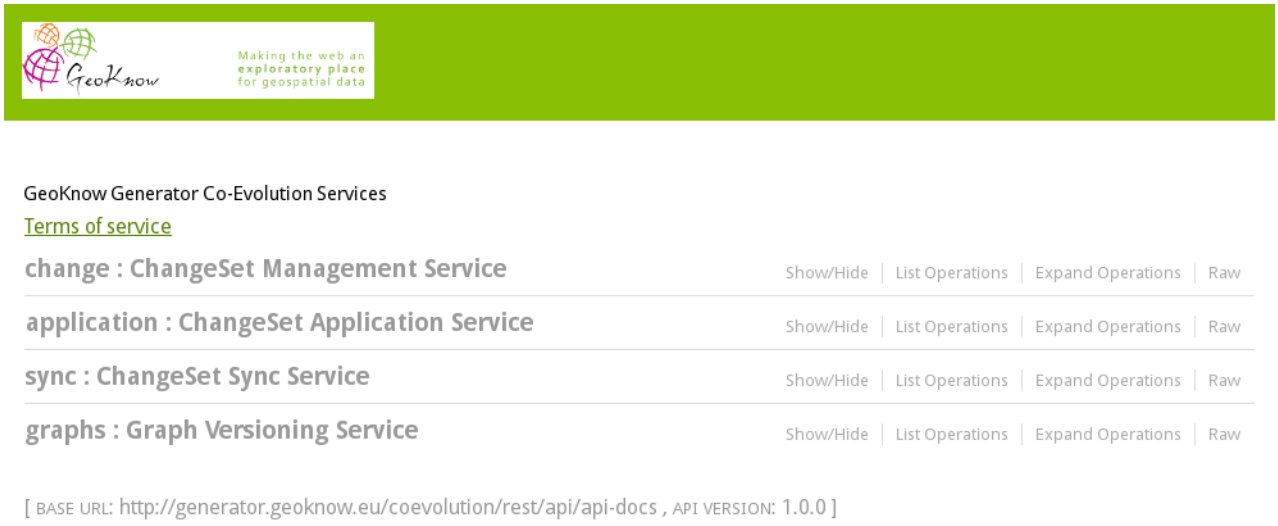

Figure 3: Co-Evolution services start page, provided by Swagger

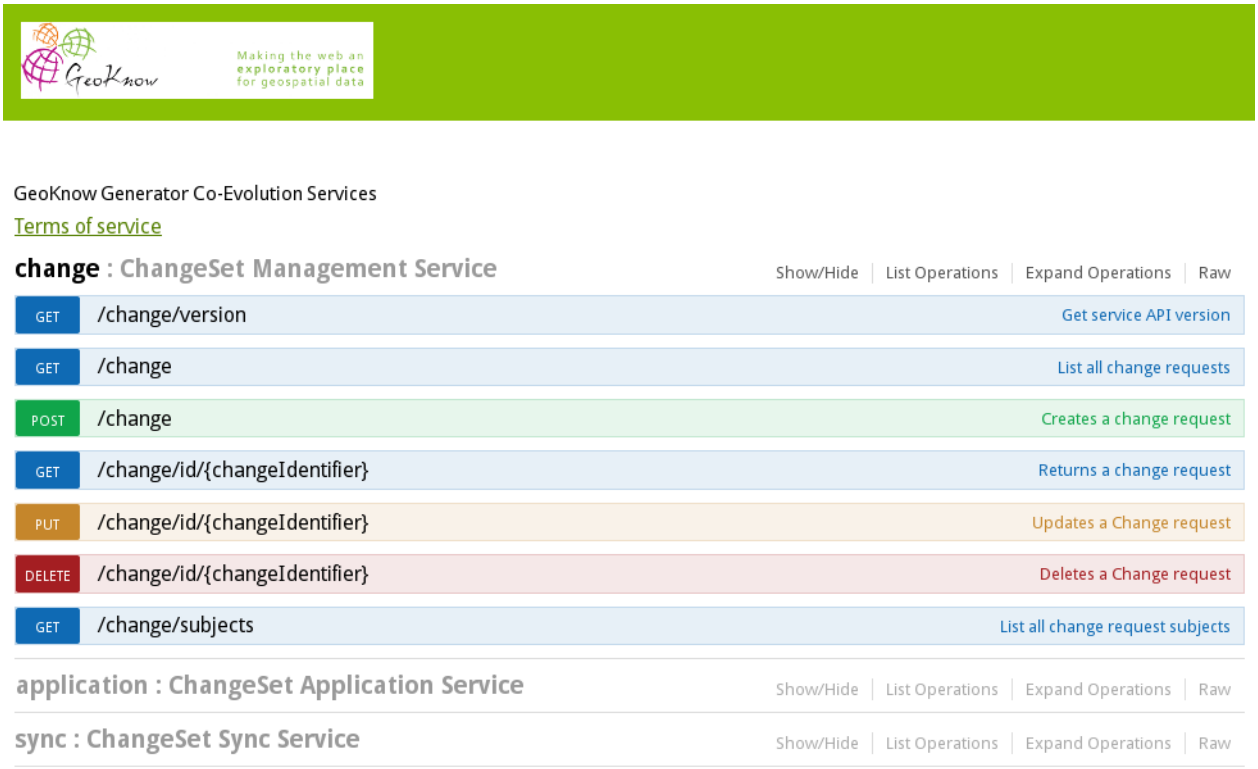

Figure 4: Operations overview of the ChangeSet Management Service, provided by Swagger

The operations provided by a Co-Evolution service can be opened by clicking on the service name. The type of operation, indicated by its colour, its resource path and a short description are listed.

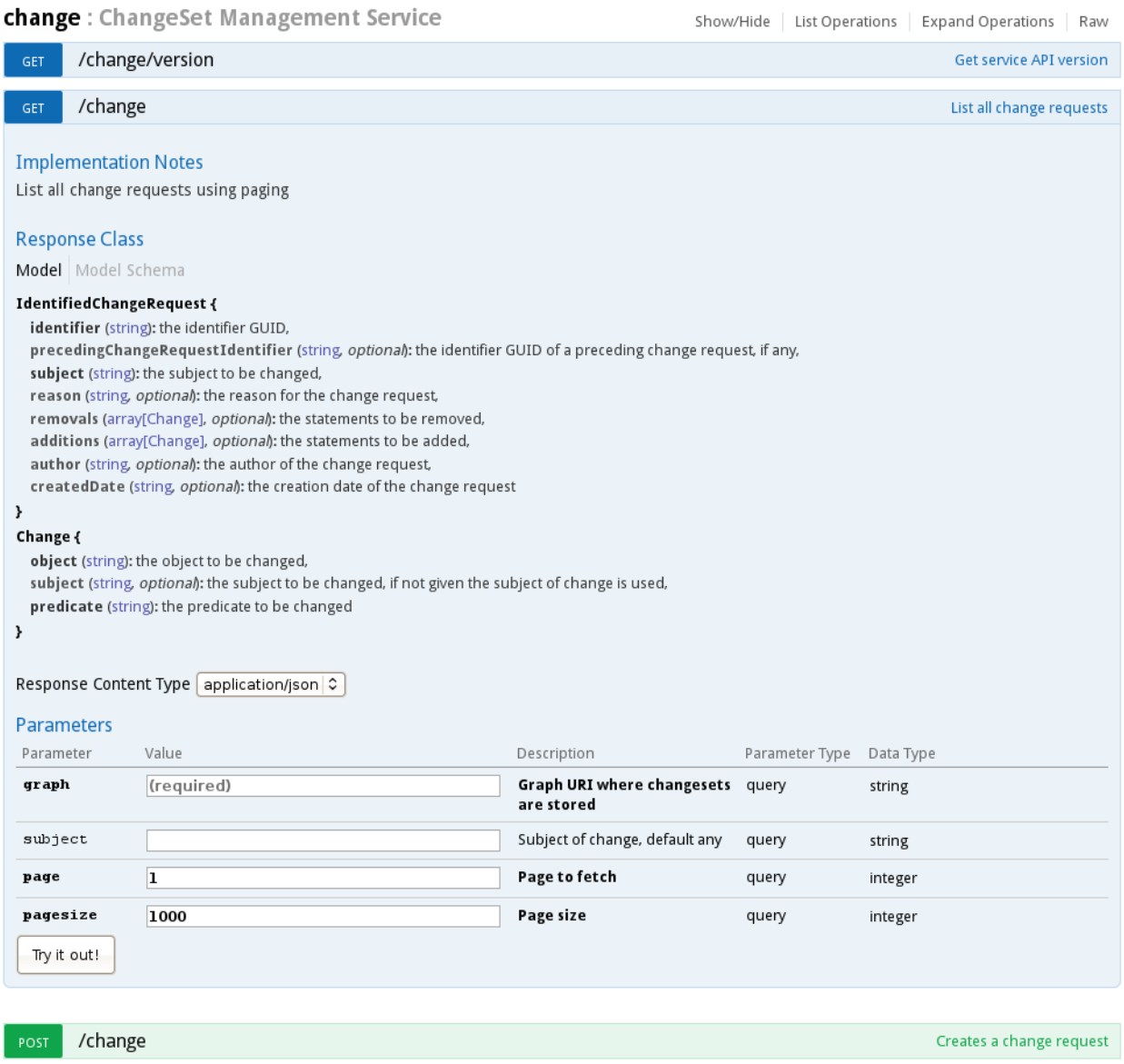

Figure 5: GET request documentation of a ChangeSet Management Service operation

After clicking on the second operation, more detailed information is listed for the operation. The response class is detailed with model information, which is all generated from annotations given in the source code. For example, the `reason` property of the `IdentifiedChangeRequest` class is implemented as follows:

```
@ApiModel( value = "Change Request", description = "Change request
   resource representation" )
@XStreamAlias("changeRequest")
public class ChangeRequest implements Serializable {
    ...
    @ApiModelProperty(value = "the reason for the change request",
       required = false)
    protected String reason;
    ...
}
```

Listing 1: Annotations of the `reason` attribute in `IdentifiedChangeRequest`

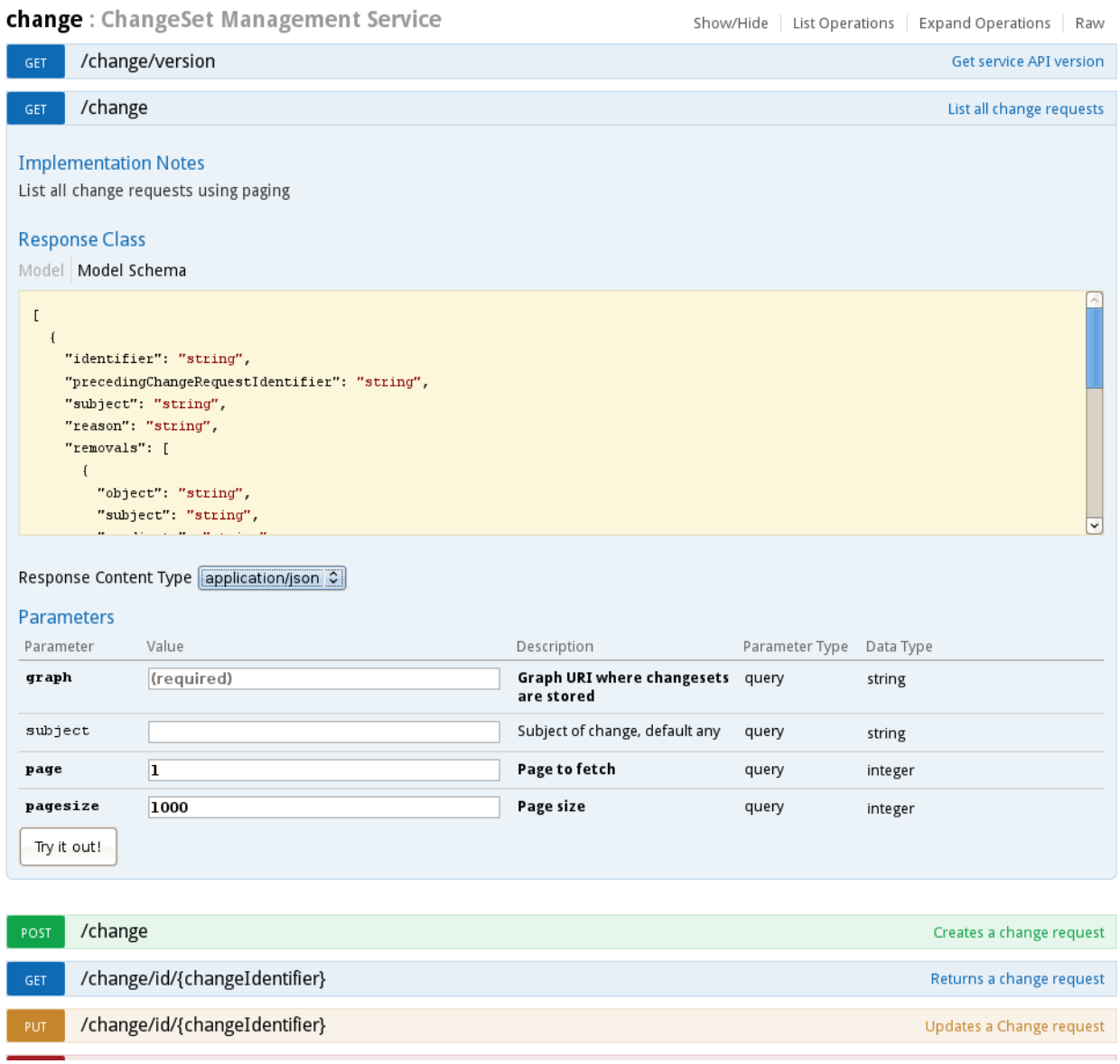

Figure 6: Documentation of a model schema and representation options for a ChangeSet Management Service operation

Switching to "Model Schema", Swagger shows a representation schema according to the selected Response Content Type, in this case a JSON representation. Below, we can enter request parameters for a request to be invoked with the "Try it out" button.

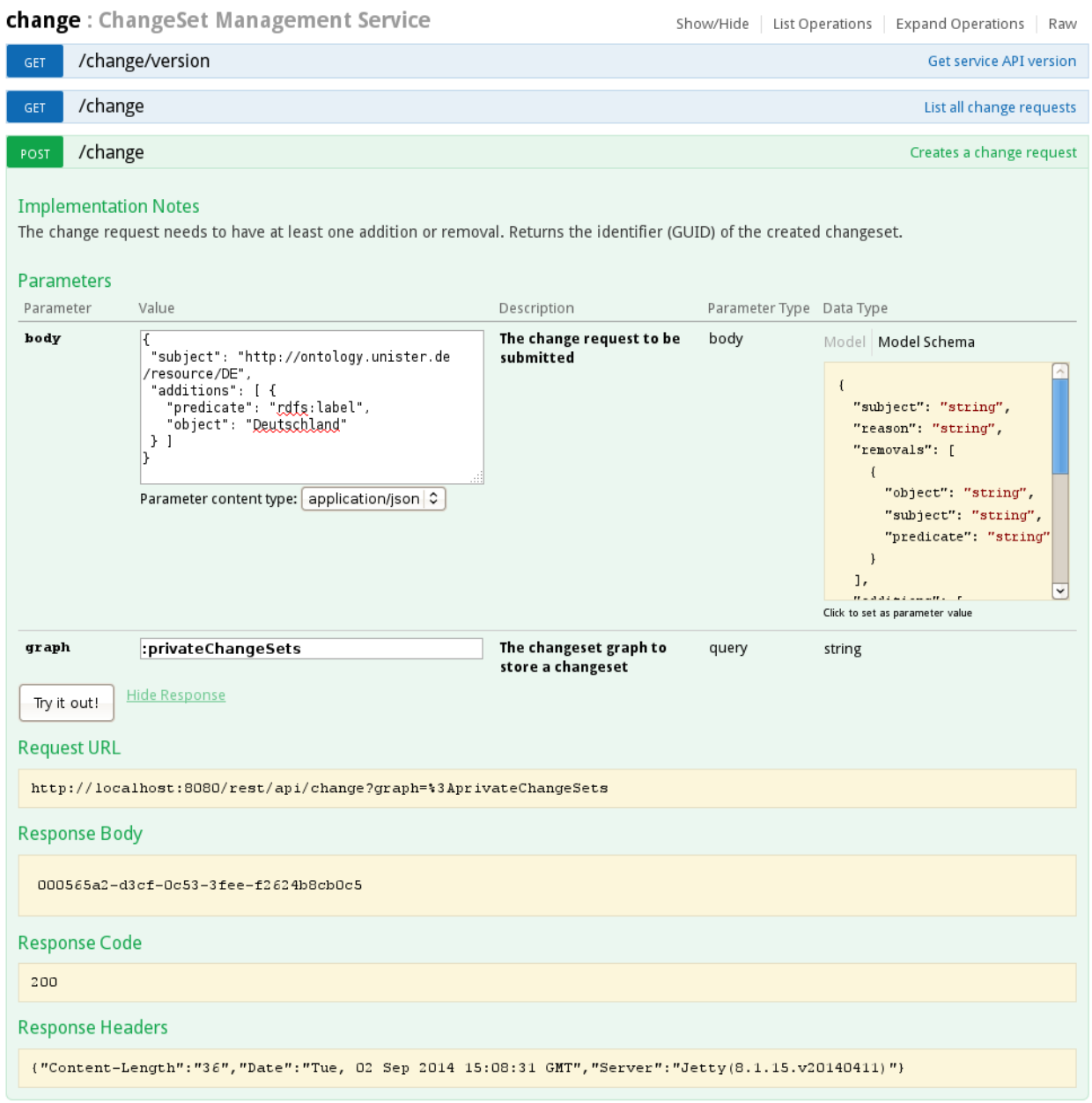

Figure 7: Example of a POST request with service response for a ChangeSet Management Service operation

In this case, we uploaded a change request with the given JSON representation, and the service returned an identifier for the created ChangeSet. The relevant implementation in the REST service is shown in Listing 2.

```
@Path("/change")
@Api(value = "/change", description = "ChangeSet Management Service",
    position = 1)
public class ChangeSetManagementRestService {
    @Inject
    private ChangeSetManagementService changeSetManagementService;

    /**
     * Creates a Change request
     */
```

```
    @ApiOperation(value = "Creates a change request", notes = "The change
        request needs to have at least one addition <i>or</i> removal.
       Returns the identifier (GUID) of the created changeset.")
    @POST
    public String createChangeRequest(
            @ApiParam(required = true, name = "change request", value = "
               The change request to be submitted") ChangeRequest
               changeRequest,
            @ApiParam(required = true, name = "graph", value = "The
               changeset graph to store a changeset") @QueryParam("graph"
               ) String graph) {
        return changeSetManagementService.create(changeRequest, graph);
    }
    ...
}
```

Listing 2: Implementation of the `createChangeRequest` operation in `ChangeSetManagementRestService`

4.2 Implementation of the Changeset Application Service

When changesets have been stored, we want to apply them either to a set of statements for a certain resource or to an entire graph. The ChangeSet Application Service currently contains an initial implementation for the first use case in the `geoknow-coevolution-changeset` module.

The implementation provides the operation `public Multimap<Modification, Statement> correct(Model instance, Resource correctionContext, Resource resultContext)`, which takes a set of statements as `instance`, the graph containing the changesets as `correctionContext`, and a target context of the changes to be added (`resultContext`). It returns a multimap of modifications, i.e. additions and successful or failed removals, and the respective statements which have been addressed. Internally, the service queries Virtuoso for the changeset chain of the subjects in the model, finding the first changeset in each chain. It then iterates over these chains an applies the changesets recursively.

Since repetitive requests to the service can lead to many SPARQL queries at Virtuoso, we implemented a caching layer prefetching a list of subjects in the given correction context, i.e., we prefetch the subjects from the graph containing the changesets. This is supported by the ChangeSet Management Service resource `/change/subjects`.

4.3 Implementation of the Graph Versioning Service

The initial implementation is rather simple, storing a timestamp and optional version tag for a certain named graph pertaining to a data set group. An example is shown in Listing 3. The service itself uses a so-called `GraphOracle` implementation in order to return the newest (or, in special cases, the second newest) named graph of a certain graph type. We expect to extend this implementation for the second deliverable on Public-private Co-Evolution, including additional metadata detailed in Section 2.

```
@prefix xsd: <http://www.w3.org/2001/XMLSchema#> .
@prefix datid: <http://dataid.dbpedia.org/ns/core#>
@prefix graph: <http://generator.geoknow.eu/coevolution/ns/graph#>
@prefix graphtype: <http://generator.geoknow.eu/coevolution/graphtype/>
@prefix graphversion: <http://generator.geoknow.eu/coevolution/
   graphversion/>

graphtype:interlinking a datid:Dataset ;
    datid:version graphversion:1407258834751 ;
```

```
    datid:version graphversion:1407258945862.

graphversion:1407258834751 a datid:Dataset ;
    datid:versionInfo "1.0.0-COEVOLUTION-TEST" ;
    graph:timestamp "1407258834751"^^xsd:long .

graphversion:1407258945862 a datid:Dataset ;
    datid:versionInfo "1.0.1-COEVOLUTION-TEST" ;
    graph:timestamp "1407258945862"^^xsd:long .
```

Listing 3: Turtle sample representation of preliminary graph version implementation

5 Summary

This deliverable reports on the initial implementation of a Public-private Co-Evolution concept. We have defined a set of services, available using a REST API, and specified their functionality for managing change request and applying changes to an RDF model. We discussed concepts and required functionality for synchronizing such changes between different public and private dataset providers, along with methods for describing different versions of a graph. This has also been encapsulated in respective services.

As this is work in progress, the implemented services of the component will be integrated with frontend services, e.g., a Web GUI for modifying resources of data sets, and extended in order to serve the purpose of synchronizing different datasets. Also, the dataset management in the GeoKnow Generator has to be integrated with the versioning service towards the next deliverable.