Deliverable 4.3.2
Public-private Co-Evolution

<table>
<thead>
<tr>
<th>Dissemination Level</th>
<th>Public</th>
</tr>
</thead>
<tbody>
<tr>
<td>Due Date of Deliverable</td>
<td>Month 30, 31/05/2015</td>
</tr>
<tr>
<td>Actual Submission Date</td>
<td>Month 31, 15/06/2015</td>
</tr>
<tr>
<td>Work Package</td>
<td>WP4, Spatial-Semantic Browsing, Visualisation and Authoring Interfaces</td>
</tr>
<tr>
<td>Task</td>
<td>T4.3</td>
</tr>
<tr>
<td>Type</td>
<td>Prototype</td>
</tr>
<tr>
<td>Approval Status</td>
<td>Approved</td>
</tr>
<tr>
<td>Version</td>
<td>1.0</td>
</tr>
<tr>
<td>Number of Pages</td>
<td>30</td>
</tr>
<tr>
<td>Filename</td>
<td>D4.3.2_Public-private_Co-Evolution.pdf</td>
</tr>
</tbody>
</table>

Abstract: This deliverable describes the software component “Co-Evolution Services” which allows to track changes and apply tracked modifications to new dataset versions. Changes can be synchronized with conflict resolution. The component has been integrated into GeoKnow Generator and tested with versioned real-world datasets, as well as synthetic data for performance evaluations.

The information in this document reflects only the author’s views and the European Community is not liable for any use that may be made of the information contained therein. The information in this document is provided “as is” without guarantee or warranty of any kind, express or implied, including but not limited to the fitness of the information for a particular purpose. The user thereof uses the information at his/her sole risk and liability.

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

<table>
<thead>
<tr>
<th>Version</th>
<th>Date</th>
<th>Reason</th>
<th>Revised by</th>
</tr>
</thead>
<tbody>
<tr>
<td>0.1</td>
<td>12/05/2015</td>
<td>Initial version of this deliverable</td>
<td>Matthias Wauer</td>
</tr>
<tr>
<td>0.2</td>
<td>13/05/2015</td>
<td>Outline, initial structure, initial content</td>
<td>Matthias Wauer</td>
</tr>
<tr>
<td>0.3</td>
<td>19/05/2015</td>
<td>Extended service spec content</td>
<td>Bernd Eickmann</td>
</tr>
<tr>
<td>0.4</td>
<td>20/05/2015</td>
<td>Service implementation content</td>
<td>Matthias Wauer</td>
</tr>
<tr>
<td>0.5</td>
<td>22/05/2015</td>
<td>Integration content</td>
<td>Matthias Wauer</td>
</tr>
<tr>
<td>0.5</td>
<td>26/05/2015</td>
<td>Chapters completed</td>
<td>Matthias Wauer</td>
</tr>
<tr>
<td>0.6</td>
<td>09/06/2015</td>
<td>Content updated for evaluation</td>
<td>Matthias Wauer</td>
</tr>
<tr>
<td>0.6</td>
<td>10/06/2015</td>
<td>Prepared for peer review</td>
<td>Matthias Wauer</td>
</tr>
<tr>
<td>0.7</td>
<td>11/05/2015</td>
<td>Peer review</td>
<td>Valentina Janev</td>
</tr>
<tr>
<td>0.8</td>
<td>12/06/2015</td>
<td>Final changes w.r.t. review comments</td>
<td>Matthias Wauer</td>
</tr>
<tr>
<td>1.0</td>
<td>15/06/2015</td>
<td>Final version approved</td>
<td>Claus Stadler</td>
</tr>
</tbody>
</table>

Author List

<table>
<thead>
<tr>
<th>Organization</th>
<th>Name</th>
<th>Contact Information</th>
</tr>
</thead>
<tbody>
<tr>
<td>Unister</td>
<td>Matthias Wauer</td>
<td><a href="mailto:matthias.wauer@unister.de">matthias.wauer@unister.de</a></td>
</tr>
<tr>
<td>Unister</td>
<td>Bernd Eickmann</td>
<td><a href="mailto:bernd.eickmann@unister.de">bernd.eickmann@unister.de</a></td>
</tr>
<tr>
<td>Ontos</td>
<td>Alejandra García-Rojas</td>
<td><a href="mailto:alejandra-garciarojas@ontos.com">alejandra-garciarojas@ontos.com</a></td>
</tr>
<tr>
<td>InfAI</td>
<td>Claus Stadler</td>
<td><a href="mailto:cstadler@informatik.uni-leipzig.de">cstadler@informatik.uni-leipzig.de</a></td>
</tr>
</tbody>
</table>
Executive Summary

Public-private Co-Evolution describes the concept of iterative development of interlinked datasets. In other words, when independent datasets evolve over time, relationships between them (such as \texttt{owl:sameAs} references or geospatial properties between related modified resources) may need to be modified. Similarly, modifications that have been executed on previous versions of a dataset, such as manual error corrections on a DBpedia dataset dump, likely should be applied on a newly imported dump too. However, the changes might conflict with the updates.

In this deliverable, we describe the implementation, deployment and evaluation of a Co-Evolution Services component based on the interfaces and concepts defined in Deliverable D4.3.1. The component is integrated in the GeoKnow Generator user interface and enables a simple user interface for tracking and applying such modifications, based on the Facete application.

Using the Debian package deployment option discussed in Section 6, the Co-Evolution Services component has been deployed at \url{http://generator.geoknow.eu:8080/coevolution-service} and is integrated in the generator demo installation available at \url{http://generator.geoknow.eu:8080/generator/}. After an internal review, the source code will be made available at \url{https://github.com/GeoKnow/Coevolution}. 
Abbreviations and Acronyms

<table>
<thead>
<tr>
<th>Abbreviation</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>LOD</td>
<td>Linked Open Data</td>
</tr>
<tr>
<td>RDF</td>
<td>Resource Description Framework</td>
</tr>
<tr>
<td>UI</td>
<td>User Interface</td>
</tr>
<tr>
<td>TripleGeo</td>
<td>Extraction component for geospatial data in the Linked Data Stack</td>
</tr>
<tr>
<td>LIMES</td>
<td>Interlinking component in the Linked Data Stack</td>
</tr>
<tr>
<td>FAGI</td>
<td>Fusing component in the Linked Data Stack</td>
</tr>
<tr>
<td>CROCUS</td>
<td>Component for computing geospatial metrics in the Linked Data Stack</td>
</tr>
<tr>
<td>CubeViz</td>
<td>Component for visualising metrics in the Linked Data Stack</td>
</tr>
<tr>
<td>Facete</td>
<td>Component for faceted browsing on geospatial data in the Linked Data Stack</td>
</tr>
<tr>
<td>DBpedia</td>
<td>Dataset provider converting Wikipedia into RDF</td>
</tr>
<tr>
<td>PostgreSQL</td>
<td>Relational database</td>
</tr>
<tr>
<td>Generator</td>
<td>GeoKnow Generator component, <a href="http://generator.geoknow.eu">http://generator.geoknow.eu</a></td>
</tr>
</tbody>
</table>
# Contents

1 **Introduction** 7  
   1.1 Public-private Co-Evolution in the Linked Data Lifecycle 7  
   1.2 Public-private Co-Evolution in Enterprise Settings 7  

2 **Scenario and Use Case** 9  
   2.1 Workflow 9  
   2.2 Detailed Step Descriptions 9  

3 **Extended Co-Evolution Services Specification** 12  
   3.1 Co-Evolution Terminology 12  
   3.2 Extended Change Request Management 12  
   3.3 Application of Change Requests 13  
   3.4 Graph Versioning 15  
   3.5 Change Request Synchronisation 15  

4 **GeoKnow Generator Integration of Co-Evolution Services** 17  
   4.1 Graph Management 17  
   4.2 Service Integration 17  
   4.3 Change Request Creation UI 18  

5 **Co-Evolution Implementation** 20  
   5.1 REST Services 20  
   5.2 RDF Serialisation 20  
   5.3 Application Details 22  

6 **Co-Evolution Deployment** 23  
   6.1 Building Co-Evolution Services 23  
   6.2 Co-Evolution Debian Package 23  
   6.3 Co-Evolution Services Deployment 23  

7 **Co-Evolution Services Evaluation** 25  
   7.1 Tests 25  
   7.2 Scenario Evaluation 26  
   7.3 Performance Evaluation 27
8 Conclusion and Future Work

30
List of Figures

1  Co-Evolution Services architecture overview ............................................ 7
2  Co-Evolution in the Linked Data Lifecycle .................................................... 8
3  Graph management extensions in GeoKnow Generator UI ............................ 17
4  Editing change requests using the Facete tool .............................................. 19
5  Co-Evolution Services Debian package details in aptitude .......................... 24
6  Creating a new graph set in the GeoKnow Generator Settings ...................... 26
7  Creating a new graph in the hotels graph set in the GeoKnow Generator Settings . 26
8  Specifying the details of the new graph in the GeoKnow Generator Settings 27
9  The new graph has been created in the GeoKnow Generator Settings ............ 28
10 Importing a dataset in the GeoKnow Generator Import RDF data function .... 28
11 Creating a change request using the Co-Evolution Services API user interface, as opposed to the more user-friendly Facete application ......................................................... 28
12 Creating a copy of a dataset in GeoKnow Generator .................................... 29
13 Verifying that the copy has been performed properly in GeoKnow Generator settings .......................................................... 29
14 Applying change requests on the new copy graph in GeoKnow Generator .... 29
1 Introduction

Most datasets are not static. Instead, they evolve dynamically just like the actual entities they represent. For RDF datasets in the Linked Data context, triples or entire resources are added, modified, or deleted over time. Consequently, this may also affect interlinked datasets. The relationships between datasets and their resources define the implications of a modification.

In the previous Deliverable of this task, D4.3.1, we described a generic Public-private Co-Evolution concept and presented an initial implementation, consisting of a set of Web services. For requirements and goals of this task, foundational concepts and the service architecture, please refer to the report on D4.3.1. In this report, we will discuss the second iteration of this Co-Evolution Services component. First we will define the scenario and a typical workflow in Section 2. In addition to the extended service specification (Section 3), this report explains how the services are integrated with the GeoKnow Generator (Section 4) as indicated in Figure 1, discusses implementation details (Section 5), documents how the services can be built, packaged and deployed (Section 6) and finally evaluates the functionality based on a typical scenario of the E-Commerce use case (GeoKnow work package 6) which is equally applicable in many other cases (Section 7).

1.1 Public-private Co-Evolution in the Linked Data Lifecycle

Public-private Co-Evolution is a process which is a central part of the Linked Data Lifecycle¹ step “Evolution and Repair” (Figure 2). The primary goal of this task is to implement a solution for tracking, applying and distributing changes to datasets in a form which allows to re-apply changes to updated versions of datasets. This includes basic methods of conflict resolution.

1.2 Public-private Co-Evolution in Enterprise Settings

The use of interlinked public and private datasets in companies requires a certain maintenance effort since the datasets evolve independently. The proposed technology will enable any data consumer to select relevant parts of datasets and transform them as needed, thereby retaining the possibility to easily (re-)synchronize

¹http://stack.linkeddata.org
with the sources should they change. The same technology can also be used by the maintainers of the source
data sets for incorporating third party changes, such as fixes, that were made public.
2 Scenario and Use Case

For motivating and evaluating the Co-Evolution Services component, we define the following scenario and related use case which closely resembles typical tasks in data preparation for motive-based search applications in an E-Commerce use case.

The use case requires integrating private, incorporated and public open datasets while dealing with regular updates of most integrated datasets. In this scenario, we consider 3 datasets:

1. Internal “private” dataset (in our case hotel data) \( H \)
2. Internal “private” geometries (dataset stored in PostGIS) \( P \)
3. Public dataset DBpedia (in our case, a local incomplete mirror of DBpedia) \( D \)

Each of this dataset can be available in different versions, denoted as \( H_1, H_2, D_1, D_2 \), etc. For our internal processes, we define the following graphs\(^2\):

1. \( H_x P_x D_x \) for the import graphs, i.e., local versions of the source datasets
2. \( I_x \) for interlinking two datasets, e.g., using LIMES
3. \( F_x \) for fusing the datasets, e.g., using FAGI
4. \( E_x \) for adding geospatial information to them using enrichment tools, e.g., using TripleGeo or custom methods

2.1 Workflow

The use case includes the following stages of a typical workflow:

A. Data sets are imported
B. Processing takes place
C. Errors are identified and corrected
D. Updated datasets are imported
E. Recorded changes are applied
F. Processing is executed again
G. Public data sets want to incorporate changes

2.2 Detailed Step Descriptions

The steps defined in the workflow will now be explained in detail. We will refer to several existing components available in the Linked Data Stack\(^3\).

\(^2\)The \( x \) in \( D_x \) denotes a certain version of the dataset \( D \).
\(^3\)http://stack.linkeddata.org
A. Importing initial dataset versions

We want to import three initial datasets: $H_1$ (the first version of the private hotel data), $P_1$ (internal polygon data), and $D_1$ (the first version of the public DBpedia data subset). Required actions are:

- Define graphs for loading the datasets
- Import hotel RDF dump file(s) (using Generator) into virtuoso graph $H_1$
- Import DBpedia RDF dump file(s) (using Generator) into virtuoso graph $D_1$
- Configure PostgreSQL endpoint for geometries database (source of $P$)
- Import $P$ using a TripleGeo mapping file (using Generator) into virtuoso graph $P_1$

All import actions require graph versioning to be tracked, i.e. metadata on these graphs have to be stored.

B. Initial processing

We do interlinking on the initial $H_1$ and $D_1$ datasets using LIMES (into graph $I_1$) and then fuse them using FAGI (into graph $F_1$). Required actions are:

- LIMES is configured and started (using Generator)
  - source dataset $H_1$
  - target dataset $D_1$
  - result dataset $L_1$ (we are just interested in the accepted links here)
- FAGI is configured and started (using Generator)
  - source dataset $H_1$
  - target dataset $D_1$
  - result dataset $F_1$ (we are just interested in the accepted links here)

Again, these actions require graph versioning, i.e., Generator has to keep track of $L_1$ being created by LIMES on $H_1$ and $D_1$, similar for $F_1$.

C. Errors are corrected

On the fused dataset $F_1$ we use CROCUS/Cubeviz for finding potential problems (not explained in detail because not directly related to Co-Evolution Services). We fix the problems using Facete for creating change requests. Instead of fixing the issues only on $F_1$ we want to be able to visualize related data in the source datasets, e.g., for correcting bad data in DBpedia which caused the fused data in $F_1$ to be erroneous. Required actions:

- Identify a problem (e.g., using CROCUS/Cubeviz)
- Inspect the resource in Facete which contains the problem, as well as the original resources on the source graphs which contributed to the problems, example:
1. Change a geocoordinate and type in $D_1$
2. Remove interlink in $I_1$
3. Change a geocoordinate in fused dataset $F_1$
   - Save modifications of resources as change request to Virtuoso
   - Execute change applier on datasets (if not applied immediately)

**D. Updated datasets are imported**

Similar to step A, we create new graphs with updated information: $H_2$ and $D_2$. We don’t address incremental updates here. Required actions:

- Specify the new graphs as updated versions of existing graphs (in Generator, using graph versioning service)

**E. Recorded changes are applied**

Required actions:

- Select the change application component from Generator
- Specify a source graph and a set of changes to be applied
- Specify a result graph, typically an updated dataset, e.g., $H_2$ or $D_2$
- Define the conflict resolution method
- Invoke change application process

**F. Processing is executed again**

This is just like step B.

**G. Public data sets want to incorporate changes**

When we create change requests on graphs of public datasets (e.g., $D_1$), this might be helpful for DBpedia to prepare the next version of their dataset. Thus, the change sync service can provide such information to external dataset providers. Required actions for external dataset providers:

- Connect to the remote Co-Evolution Services endpoint
- Select the (public) correction context, i.e., the graph containing changes
- Specify a “from” and “to” graph version, or define filters on recorded change requests
- Retrieve respective change requests
3 Extended Co-Evolution Services Specification

Since the functionality of the Co-Evolution Services has been evolving both in terms of its interface and actual implementation, this section will first define some terminology before the extended interface is described.

3.1 Co-Evolution Terminology

Deliverable D4.3.1 described *Changesets* as a resource-centric concept to represent changes to graphs. In the scope of Public-private Co-Evolution, we extend this concept with several attributes not available in the original specification.

In order to distinguish the original from the extended specification, we coin the term *change requests* for the model used for Co-Evolution Services. A change request consists of all the properties of a Changeset (identifier URI, author, reason, created date, a set of additions and removals each as RDF statements, and a preceding changeset URI). In addition to that, we introduce the following additional attributes:

- **context**: Since we want to track changes across different named graphs managed in GeoKnow Generator, we have to specify the named graph URI to which the given statements belong.
- **verified**: Additional flag indicating whether a change should be considered a draft.

The identifier URI and the preceding change request URI should be determined by the Co-Evolution Services. Thus, when adding a change request these properties are initially omitted, and the service takes care of assigning them. Consequently we distinguish between two classes of change requests:

- **ChangeRequest** for representing individual changes without identification
- **IdentifiedChangeRequest** for representing changes with identification and (if available) a preceding change request, i.e., a change managed by Co-Evolution Services.

Regarding *change application*, we distinguish a *correction context* which contains change requests, a *result context* as the context given to newly created statements during change application (typically the target graph), and the *context* as an attribute of change requests that specifies the named graph the change was originally created for (as explained above).

With regards to *graph versioning*, we will use the term *graph set* instead of *graph types* (D4.3.1 Section 2.1) to denote an abstract entity that groups different versions of the same entity, e.g., all versions of DBpedia. The originally defined term can conflict with common understanding of a graph having types like `void:DataSet` or `void:LinkSet`.

3.2 Extended Change Request Management

The interface for change request management had to be extended in order to fully support the handling and assignment of identifiers by the Co-Evolution Services. Compared to the defined types shown in Deliverable D4.3.1. Figure 2, **IdentifiedChangeRequest** was extended with additions and removals of **IdentifiedChange**, a class that extends **Change** with an identifier URI. Additional changes with regards to the **resource** package are related to handling multiple representations, as explained in Section 5.2. This enables change requests to be accepted and returned by the service in media types application/xml, application/json, and RDF text/turtle.
3.3 Application of Change Requests

The initial interface of the change application service (D4.3.1 Section 4.2) has been changed with regards to the available methods, tracking changes, and supported extensions.

The initial service was capable of applying changes to a model representation submitted via HTTP (as message body), returning the modified model in the response entity. This proves to be insufficient in certain use cases. Hence, we extend this concept with an additional method for in-place application, i.e., applying changes directly on a named graph using SPARQL Update statements. If provenance information is crucial, the recommended approach for using this method is to create a copy of the target graph before. This is shown in Figure 12 on page 29.

Next, we added a filters parameter that enables fine-grained control of the changes to be applied. Filters consist of an attribute identifier (specifying the attribute of a change request, e.g., “reason” or “created”), an operator, e.g., “equals” or “less than”), and a value, such as “geocoordinate” or “2015-05-21T11:53:55Z”. Filters can be combined with the “and” operator, further extensions are possible as needed. Using filters, the Change Application Service can be used to apply a defined subset of change requests in a certain correction context, such as changes defined on a certain context, or verified changes created by some author after a certain date. The same filter approach can be used for fine-grained specification of changes to be synchronized.

With regards to conflict resolution, in the second iteration we address the problem of inconsistent state. This issue can occur when a change request replaces a value of an original dataset, but the value has changed in an updated version of the dataset. When applying the change request on the new dataset, it would fail to remove the original statement but still add the replacement, which could invalidate cardinality restrictions. As a result, the resulting resource would have, e.g., two geometries instead of one. The Co-Evolution Services component provides two options for resolving inconsistent state:

1. Forced change. A change request can be marked as required by using wildcards, indicating that either all statements for a certain predicate, all statements for a certain value, or all statements in general have to be modified for a given resource. This concept was already modelled initially as constants ANY_PREDICATE and ANY_OBJECT in the ChangeConstants class (D4.3.1 Figure 2).

As an example, a change request with the removal statement :r1 :p1 :ANY_OBJECT indicates that all statements of resource :r1 with predicate :p1 have to be removed, regardless of respective statements in the target model that might be correct. That way, the author of the change request can make sure there is no such statement left.

This approach has to be used with caution, as it can obscure valid updates of data. As a rule of thumb, this should be applied to correct data that is considered to be immutable, i.e., data that is not expected to change on its own.

2. Heuristic conflict resolution. Basic rules can be applied to determine whether the change request or the updates in the target dataset should prevail. In the current implementation, the heuristic is initially defined for a change application process. A potential conflict can be detected if a removal can’t be applied, i.e., the original statement to be removed does not exist. There are five potential implementations:

(a) Ignore conflict: The change request will be applied regardless of a potential conflict (default behaviour).
(b) Ignore entire change: If a potential conflict has been detected, the change request will not be applied.

---

4A potential extension of this approach is based on comparing the times of a change request and the target graph, if available, and to prefer the more recent change.
(c) Ignore change for conflicting predicate: The change request will be applied, but removals and additions of the potential conflicting statement will be ignored.

(d) Force change: If a potential conflict has been detected, the change request will be applied. In case the conflict occurred on change statements with a single removal and addition of a certain predicate and the target resource contains exactly one statement for this predicate with a different value ("cardinality one conflict"), this statement will first be removed from the target resource.

(e) Merge change: Similar to "force change", but for "cardinality one conflicts" the change applier will attempt to merge the values of the change addition and target resource statements for certain defined datatypes⁵.

For example, a change request modifying type (:changedType), label ("changedLabel"), and latitude ("2.34") of a resource is applied on a model with updated label and latitude:

```
@prefix : <http://geoknow.eu/test#> .
: r1 a :originalType ;
  :labelProperty "originalLabel" ;
  :latitudeProperty "1.23"^^xsd:double .
```

Listing 1: Original example model

```
: r1 a :originalType ;
  :labelProperty "updatedLabel" ;
  :latitudeProperty "0.00"^^xsd:double .
```

Listing 2: Updated example model

The different implementations would yield the following results⁶:

```
: r1 a :changedType ;
  :labelProperty "updatedLabel", "changedLabel" ;
  :latitudeProperty "2.34"^^xsd:double, "0.00"^^xsd:double .
```

Listing 3: Changes applied to updated model with "ignore conflict" heuristic

```
: r1 a :originalType ;
  :labelProperty "updatedLabel" ;
  :latitudeProperty "0.00"^^xsd:double .
```

Listing 4: Changes applied to updated model with "ignore entire change" heuristic

```
: r1 a :changedType ;
  :labelProperty "updatedLabel" ;
  :latitudeProperty "0.00"^^xsd:double .
```

Listing 5: Changes applied to updated model, "ignore change for conflicting predicate" heuristic

```
: r1 a :changedType ;
  :labelProperty "changedLabel" ;
  :latitudeProperty "2.34"^^xsd:double .
```

Listing 6: Changes applied to updated model with "force change" heuristic

---

⁵This could further be restricted to certain predicates, e.g., to merge geocoordinates but not prices.

⁶This example assumes that a single merge function for properties with xsd:double values was defined as average, hence the latitude property "1.17" for Listing 7. Existing vocabulary like rdfs:label and geo:lat would be used in practice.
3.4 Graph Versioning

Versioning generally assumes a notion of something that a given artifact is the version of. In the case of RDF datasets managed in GeoKnow Generator, this role is fulfilled by graph sets. Intuitively speaking, a graph set is a collection of datasets that represent stages in the evolution of a graph. A graph set itself is an RDF resource that has an URI that is formed of a given identifier and the general graph set prefix. Optionally, a graph set has a label, which might be identical to the identifier, a description and a creation date.

Datasets have in turn a graph set, which they belong to. To allow a certain degree of experimentation, this requirement was left as optional. It is, however, clear, that a dataset can only be subject to graph versioning if its graph set is registered and known. Each dataset is a named graph that has a type (typically void:Dataset or void:Linkset as described in http://www.w3.org/TR/void/), label, description, registered source graphs, licenses, authors, a homepage with further information on that dataset version and a related SPARQL endpoint. Furthermore, they have three different timestamps they are associated with, the creation time, a modification time, signifying the last changes made to the dataset, and the time the dataset was issued, that is, the time the dataset was considered to be published and thus closed to modification. Some of these properties already existed in the GeoKnow Generator named graph management functions (the “Settings” tab in the Generator web application).

The Graph Versioning Service allows to create new graph sets, retrieve existing graph sets, retrieve the metadata belonging to a graph set, add new datasets, update existing datasets, retrieve all datasets belonging to a graph set, and to retrieve the latest dataset belonging to a given graph set. The latter works using the creation time of a dataset. While this might usually gives a user what he wants, it might be convenient to add corresponding methods retrieving the last published version of a dataset or the latest modified one.

3.5 Change Request Synchronisation

Sharing changes between different GeoKnow Generator nodes enables synchronized updates of datasets and a collaborative evolution of public data. As an example, sharing the corrections made in example workflow step C (Page 9) to a local subset of DBpedia could help improve the next version of the public DBpedia if such local changes are integrated on a large scale.

Co-Evolution Synchronization provides a basic implementation for such a service, enabling remote parties to retrieve public change requests that belong to a graph set. A list of change requests, which can be further specified by using the filter mechanism used in change application, can be retrieved in different representations.

For some scenarios detecting changes between graphs (creating a “diff”) could be a useful feature. Co-Evolution Synchronization provides basic functionality for this, based on the following SPARQL queries, in order to determine additions and removals that can be used for generating a list of respective change requests for the involved subjects:

```
1 select * where {
2 graph :g1 { ?s ?p ?o } minus {
```
Listing 8: SPARQL determining additions for the transition from graph :g1 to :g2

```
3 graph :g2 { ?s ?p ?o }
4 }
```

Listing 9: SPARQL determining removals for the transition from graph :g1 to :g2

```
1 select * where {
2 graph :g2 { ?s ?p ?o } minus {
3 graph :g1 { ?s ?p ?o }
4 }
```

Note that the Virtuoso documentation mentions built-in functions for generating graph diffs. While they are not included in the current version, such integrated functionality could provide performance benefits on very large datasets. Despite these concerns, the performance of these queries appears to be very acceptable on reasonable datasets. Comparing two versions of a production dataset in the E-commerce use case (GeoKnow work package WP6) with about 10 million triples only took about 11 seconds.
4 GeoKnow Generator Integration of Co-Evolution Services

While Co-Evolution Services is a software component that can be deployed and used separately, its primary usage will be integrated with GeoKnow Generator. This section explains how Co-Evolution Services functionality is integrated into the Generator user interface and functionality.

4.1 Graph Management

Generator’s graph management has been extended by two aspects:

- Graph set management: graph sets can be created and modified, which invokes methods of the Graph Versioning Service REST interface.
- Extended named graph metadata: existing functionality for creating and modifying named graphs has been extended to cover the additional properties discussed in Section 3.4.

Figure 3 shows the extended GeoKnow Generator user interface for managing graph sets and named graphs.

Figure 3: Graph management extensions in GeoKnow Generator UI

4.2 Service Integration

Four sorts of tasks involving Co-Evolution had to be integrated in Generator:
1. Creating copies of existing graphs: this functionality is not part of the graph versioning service, instead it is implemented in Generator. However, when creating a copy of an existing graph the appropriate metadata specified in Co-Evolution Services are set appropriately. This primarily affects the source attribute.

2. Handling result graphs of processes: When new graphs are created by processes, the behaviour is described as above. For example, invoking a LIMES process will create at most two new graphs (accepted and review interlinks). Both target graphs will get appropriate metadata annotations, including the “source” and “target” graphs specified in the LIMES annotation added as dataset source.

3. Creating change requests: To enable users to specify change requests, a suitable user interface is required that uses the Change Management Service functionality. This functionality is integrated into the Facete component.

4. Applying change requests: When change requests have been recorded, a user can invoke the Change Application Service to replay the modifications on a graph. The user interface has to be capable of enabling a user to select the required parameters, such as correction context and filters.

While the graph management extensions required for integration are visible in the “Settings” tab, the service integration extensions have been added to the Workbench tab. Creating copies of graphs is added as an option in the “Extraction and Loading” phase at menu item “Import RDF data”, while the second task of handling result graphs is transparent for the user. In other words, invoking LIMES or FAGI is exactly the same for the user, the additional metadata are generated and stored automatically by the Generator.

Creating and applying change requests have been separated since these tasks are usually executed in different phases. For applying changes, there is a new phase “Evolution and Repair”. The “Apply change requests” item allows a user to specify a correction context and a target graph, as well as filters, before the process can be invoked. Creating change requests can be done via the Facete menu item in the “Search Querying and Exploration” phase.

4.3 Change Request Creation UI

Resources in named graphs managed in GeoKnow Generator can be explored using Facete. In addition to the direct editing options available in Facete (see GeoKnow Deliverable D4.1.2), the editing options based on the RDF editing extensions (REX) have been extended with an option to load and store change requests. The extensions include the following functionality:

- Additional button “Create/Update Change Request” in the editing view for storing or updating a change request for the modifications via Co-Evolution Services.
- Additional input field for selecting the correction context, i.e., the graph where change requests are stored
- A list of stored change requests in this correction context, either displaying the newest change requests (if no resource has been selected) or all change requests of a selected resource
- Additional fields for entering/modifying metadata of change requests
- Highlighting the modifications in the list of resource properties (green for added statements, red for removals) and functionality for adapting the internal models when loading an existing change request

An example of an early version with these editing options is shown in Figure 4.
Figure 4: Editing change requests using the Facete tool
5 Co-Evolution Implementation

Implementation of the Co-Evolution Services is based on the foundations and libraries already specified in Deliverable D4.3.1 Section 4: JAX-RS 2.0 for implementing REST services, Sesame for handling RDF data models, Swagger for integrated documentation and testing, Spring for dependency injection, and Maven for project management. Likewise, the modules and packages are consistent and didn’t have to be changed since the first iteration.

5.1 REST Services

Similarly, the REST services had already been established for the first release:

- ChangeSetManagementRestService
- ChangeSetApplicationRestService
- ChangeSetSyncRestService
- GraphVersioningRestService

Each REST service implements the HTTP endpoint related functionalities and delegates the actual work to a respective Service implementation. This service builds appropriate URIs for identifiers, invokes queries via different injected data access objects and generates the result objects as necessary.

5.2 RDF Serialisation

JSON and XML serialisation are supported by the given frameworks using Jackson 2 and XStream and appropriate annotations. Via content type negotiation, respective registered providers handle the serialisation and deserialisation transparently. This approach was also realised for RDF serialisation.

Compared to the providers for tree-based data models like XML and JSON, which are mostly based on reflection and annotations, the implemented RDF serialisation is realised with a custom interface RdfConvertible. The class to be (de-)serialised for a certain request has to implement this class and its two methods:

```java
@ApiModel(value = "Graph Set", description = "Graph Set resource representation")
@XStreamAlias("graphSet")
public class GraphSet implements RdfConvertible<GraphSet> {
  
  @Override
  public Model toModel() {
    LinkedHashModel model = new LinkedHashModel();
    URIImpl identifierURI = new URIImpl(Namespace.GRAPHSETS + identifier);
    model.add(identifierURI, RDF.TYPE, GRAPHSET_TYPE);
    if (getDescription() != null) {
      model.add(identifierURI, DCTERMS.DESCRIPTION, new LiteralImpl(getDescription()));
    }
    if (getLabel() != null) {
      model.add(identifierURI, RDFS.LABEL, new LiteralImpl(getLabel()));
    }
    return model;
  }

  @Override
  public void fromModel(Model model) {
    URI identifierURI = model.getURI(RDF.TYPE, GRAPHSET_TYPE);
    if (identifierURI != null) {
      identifier = identifierURI.getNamespace() + identifierURI.getString();
    }
    // Other deserialisation logic...
  }
}
```
if (getCreated() != null) {
    model.add(identifierURI, DCTERMS.CREATED, new LiteralImpl(
        RdfUtils.formatIso8601Date(getCreated()),
        XMLSchema.DATETIME));
}

return model;

@override
public GraphSet fromModel(Model model, URI subject) {
    if (subject == null) {
        // try to identify subject
        Set<Resource> subjects = model.filter(null, RDF.TYPE,
            GRAPHSET_TYPE).subjects();
        if (subjects.size() == 1) {
            subject = (URI) subjects.iterator().next();
        } else {
            throw new SerializationException("No subject given,
                but subject can't be derived from model: " +
                subjects.size() + " graphsets found in model");
        }
    }

    // extract required properties
    String description = model.filter(subject, DCTERMS.DESCRIPTION,
        null).objectString();
    String label = model.filter(subject, RDFS.LABEL, null).objectString();
    Value createdValue = model.filter(subject, DCTERMS.CREATED,
        null).objectValue();

    GraphSet graphSet = new GraphSet(
        StringUtils.substringAfter(subject.stringValue(),
            Namespace.GRAPHSETS));
    // add optional properties
    graphSet.setDescription(description);
    graphSet.setLabel(label);
    try {
        if (null != createdValue) {
            graphSet.setCreated(RdfUtils.parseIso8601Date(
                createdValue.stringValue()));
        }
    } catch (ParseException e) {
        throw new SerializationException("Invalid date: " +
            created, e);
    }
    return graphSet;
}

Listing 10: RdfConvertible implementation example: GraphSet

The actual provider implementation is capable of (de-)serialising RdfConvertible instances and serialising a java.util.Collection of RdfConvertible. It uses Rio⁸ to execute the actual parsing and

---

⁸Rio is the RDF input/output library of the Sesame RDF framework.
formatting of the serialised representation:

```java
@Component
@Provider
@Produces({ TurtleProvider.TURTLE_MEDIA_TYPE })
@Consumes({ TurtleProvider.TURTLE_MEDIA_TYPE })
public class TurtleProvider implements MessageBodyReader<Object>,
    MessageBodyWriter<Object> {
    ...
    @Override
    public void writeTo(...)
        throws IOException, WebApplicationException {
        if (arg4.toString().startsWith(TURTLE_MEDIA_TYPE)) {
            ... Model model = ((RdfConvertible<?>) arg0).toModel();
            Rio.write(model, arg6, RDFFormat.TURTLE);
            ...
        } else {
            throw new WebApplicationException(Response.Status.
                UNSUPPORTED_MEDIA_TYPE);
        }
    }
}
```

Listing 11: Turtle provider for serialisation, excerpt

5.3 Application Details

Change requests are applied in a recursive fashion, i.e., by generating a change request chain which is then applied subsequently. The chain is determined by the given precedingChangeRequest attribute of a change request (except the first change request for a subject in each correction context). Since the preceding change request relation is so vital for change application, the design of the involved resources and management methods makes sure that this property is usually not handled by the client. Instead, clients send a HTTP request to add a ChangeRequest representation, and the change management service takes care of assigning an identifier and determining the preceding change request.

The connectivity to Virtuoso is realised via Sesame 2. This way, for unit testing the application context can be configured for using an in-memory store.
6 Co-Evolution Deployment

Co-Evolution Services have been implemented as a Maven multi-module project. This facilitates easy building and reuse of the implemented functionality.

6.1 Building Co-Evolution Services

After cloning the Co-Evolution Services from git, they can be built using the command:

```
mvn package
```

By default, this builds the modules including the services war with the default profile targeted towards deployment on http://generator.geoknow.eu. For testing and local deployment (on a Virtuoso instance running on localhost), this profile has to be applied:

```
mvn package -P local
```

When the service is built, the module coevolution-service can be executed for testing:

```
mvn tomcat:run-war -P local
```

6.2 Co-Evolution Debian Package

The new coevolution-debian-package module takes care of creating a .deb package file for installing Co-Evolution Services on a Debian-based Linux server. The jdeb\(^1\) Maven plugin generates the package containing

- Debian “control” file configuring dependencies, currently only tomcat7 is required
- coevolution-service.war which is copied to /var/lib/tomcat7/webapps
- /etc/tomcat7/Catalina/localhost/coevolution-service.xml with default settings

6.3 Co-Evolution Services Deployment

The Debian package including the Web application archive is made available through the Linked Data stack repositories. After configuring this repository, Co-Evolution Services can be installed using the command:

```
apt-get install coevolution-debian-package
```

Figure 5 shows the package details after installation via aptitude. Alternatively, when building Co-Evolution Services locally, the generated package in coevolution-debian-package/target/coevolution-debian-package_{version}_all.deb can be installed using the command:

```
dpkg -i coevolution-debian-package_{version}_all.deb
```

The services are configured for deployment on generator.geoknow.eu by default. If the environment is different, the necessary changes can be made in /etc/tomcat7/Catalina/localhost/coevolution-service.xml. By default, all environment properties are commented out to use the defaults, but can be enabled as shown in Listing 12.

---

\(9\) Alternatively, the coevolution-service.war in coevolution-service/target path can be copied to a running Tomcat webapps folder as usual

\(10\) https://github.com/tcurdt/jdeb
Figure 5: Co-Evolution Services Debian package details in aptitude

```xml
  <Context path="/coevolution-service"
crossContext="false"
reloadable="true">

    <!-- docBase = "/var/lib/tomcat7/webapps/coevolution-service.war" -->

    <!-- Configuration -->
    <!-- Uncomment the following properties you want to configure,
    make sure reloadable is "true" and save the file. -->
    <!-- Tomcat should then reload the coevolution-service context,
    see /var/lib/tomcat7/logs/catalina.out. -->

    <Environment name="hostname"
      value="generator.geoknow.eu" type="java.lang.String" override="false"/>

    <Environment name="hostport"
      value="8080" type="java.lang.Integer" override="false"/>

    <Environment name="apipath"
      value="coevolution-service/rest" type="java.lang.String"
      override="false"/>

    <Environment name="virtuosourl"
      value="jdbc:virtuoso://localhost:1111/charset=UTF-8/log_enable=2/timeout=60" type="java.lang.String" override="false"/>

    <Environment name="virtuosouser"
      value="generator" type="java.lang.String" override="false"/>

    <Environment name="virtuosopassword"
      value="***" type="java.lang.String" override="false"/>

  </Context>
```

Listing 12: Default settings in coevolution-service.xml
7 Co-Evolution Services Evaluation

The Co-Evolution Services have been developed using a test-driven development approach. Most classes have been tested separately using unit tests, except trivial code (getters, setters, basic conversion, etc.). Further tests have been set up for integration testing, as explained below. Finally, the functionality in accordance with the scenario workflow described in Section 2 has been evaluated in a prototype instance on production datasets, as described in the following subsections. This includes performance measurements, in particular for the change application and synchronisation features.

7.1 Tests

Tests were generally based on the Spring Test framework, in addition to Mockito based mocking of required components and test profiles for using an in-memory sesame repository instead of a Virtuoso store. All data access objects (DAOs) have been implemented using dependency injection to facilitate this.

For integration testing, a basic test framework has been implemented around the AbstractTest class in package com.unister.semweb.geoknow.coevolution.test:

```java
@RunWith(SpringJUnit4ClassRunner.class)
@ContextConfiguration(loader=AnnotationConfigContextLoader.class)
@ActiveProfiles("test")
public abstract class AbstractTest {

    @Configuration
    static class ContextConfiguration extends AppConfig {

        @Bean
        public TripleStoreMockBean tripleStoreMockBean() {
            return new TripleStoreMockBean();
        }
    }

    @Autowired
    public TripleStoreMockBean tripleStoreMockBean;

    public AbstractTest() {
    }

    protected void addFixture(String path) throws RepositoryException, RDFParseException, RDFHandlerException, IOException {
        tripleStoreMockBean.addResourceToStore(new ClassPathResource("fixture/"+path));
    }

    protected void reset() throws RepositoryException, RDFParseException, RDFHandlerException, IOException {
        tripleStoreMockBean.reset();
    }
}
```

Listing 13: Abstract test class for integration tests

This class allows to load test fixtures (in TriG format) for each test, which enables testing functionality with a predefined set of statements. Several test fixtures have been defined, including a subset of DBpedia and hotel resources and appropriate change requests to test change management, application, graph versioning, and synchronisation functionality.
7.2 Scenario Evaluation

For evaluating the Co-Evolution Services and the integration, we further tested the functionality based on the scenario defined in Section 2. We used a subset of the hotel reviews dataset from GeoKnow Work Package 6.

First, we create a graph set representing a container for our hotel review dataset versions (Figure 6). Next, we create a new graph version of this graph set (Figure 7) and configure its details (Figure 8). When clicking “Save”, the new graph is shown in the list of named graphs in Figure 9.

Next, we can import RDF data into the new graph using existing Generator functionality (Figure 10). Next, we browse the dataset, execute some processing tools like LIMES and finally validate the dataset quality using geospatial metrics (not shown here). We identify an issue and create a respective change request (Figure 11\textsuperscript{\textsuperscript{1}}). Next, we want to update the dataset. In order to do this, we first create a copy of the dataset (Figures 12, 13). Then we invoke the change request application by selecting the respective correction context (“:changes”) and the target graph, in this case the created copy “hotel:v2”. We further specify a conflict resolution strategy and optional filters (Figure 14). Now we notice that there’s a typo in the first change. We simply create another change request and apply it directly on the hotel:v2 graph. When the new graph version has been tested

\textsuperscript{1}Here we show how the change request can be created using the Co-Evolution Services Swagger user interface. For creating a change request using the more user-friendly Facete application, see Section 4.3.
successfully we can mark it as verified (using the checkbox icon next to it in Figure 9). Applying the changes to an updated version of the dataset works just as explained above.

7.3 Performance Evaluation

While the performance test was executed on a subset of a production dataset, the performance appears to be generally acceptable also for larger datasets. The change application process scales linearly with the amount of change requests and, in the case of local application to a given model, takes advantage of subject caching to reduce lookup queries on the triple store.

For verifying that these concepts work properly, we have generated 10.000 change requests on a generated dataset with 10 million triples. Applying the 10.000 changes on the dataset took 98.6 seconds, consisting of 58.2 seconds for fetching the change requests from the correction context and 40.4 seconds for applying the changes on the target graph. This is entirely acceptable for the given use cases.
Figure 9: The new graph has been created in the GeoKnow Generator Settings

Figure 10: Importing a dataset in the GeoKnow Generator Import RDF data function

Figure 11: Creating a change request using the Co-Evolution Services API user interface, as opposed to the more user-friendly Facete application
Figure 12: Creating a copy of a dataset in GeoKnow Generator

Figure 13: Verifying that the copy has been performed properly in GeoKnow Generator settings

Figure 14: Applying change requests on the new copy graph in GeoKnow Generator
8 Conclusion and Future Work

In this deliverable we presented the second iteration of the Co-Evolution Services, a software component for managing, applying and synchronising changes to versioned public and private datasets.

The component has been integrated into GeoKnow Generator, so registered users can annotate required metadata to datasets, create change requests, and apply changes to graphs. The Facete application was used as an extension point for users to create and modify change requests. We have also created a Debian package which can be used to easily deploy the component on any Debian-based server running Apache Tomcat 7.

We can verify that the implemented component and its integration in GeoKnow Generator works as expected for a given scenario related to GeoKnow Work Package 6 and the performance is acceptable. As the evaluation shows, most of the runtime for applying changes on very large datasets is spent fetching change requests from the correction context. This could be accelerated by fetching the change requests as a batch instead of iteratively loading them. Likewise, batch updates for applying change requests could also speed up the process. Yet, such modifications only seem to be necessary when the amount of change requests is much higher and modifications have to be applied with very low latency, which is not a requirement from the given use cases. While the synchronisation of changes between GeoKnow instances in different domains has not been integrated in the Generator user interface, the required functionality is available in the Co-Evolution Services and has been evaluated for the use case.

In the future, the approach could be further integrated by automatically creating change requests based on outliers found using CROCUS metrics and the CubeViz component. Also, manual inspection of applied and ignored change requests and a more detailed provenance tracking could be helpful in other use cases.