Symplectic Elements Harvester for VIVO:

Description & Overview

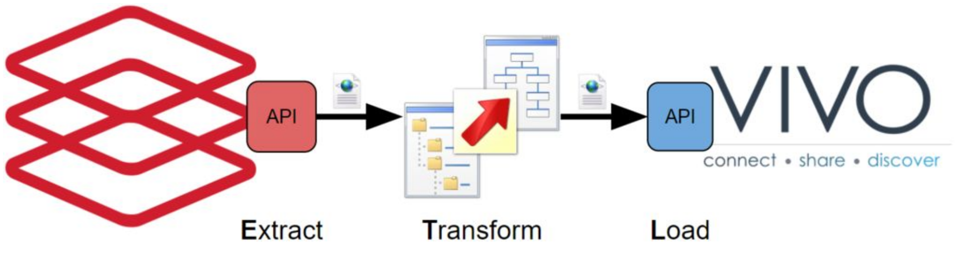
Symplectic, 4 Crinan Street, London,
N1 9XW, United Kingdom.

Contents

Introduction

The Elements' Vivo Connector allows regular updates of the data in a Vivo instance based on changes occurring within a connected Elements instance.

It is, in essence, an Extract, Transform and Load (ETL) pipeline that extracts data from the Elements' API, transforms that data into Vivo compatible Resource Description Framework (RDF) data using an eXtensible Stylesheet Language Transformation (XSLT) based pipeline, and finally loads that data into the connected Vivo instance via Vivo's Sparql Update API.



There are obviously many important detailed operations that occur during these phases, but this is a good conceptual overview.

Connector Description

The connector fundamentally consists of two components:

1. The *Harvester*
2. The *Fragment loader*

Harvester

The harvester performs several functions:

- Harvesting data from the Elements API.
- Translating the harvested data to a Vivo compatible RDF representation (triples).
- Loading those triples into a temporary triple store (Jena TDB).
- Creating change-sets by comparing the current temporary triple store to the temporary triple store from the previous run (if present).
- Turning those change-sets into small fragment files (by default <2MB).

The harvester is designed to minimize the load that is placed on the Elements API by making use of a differential harvest when pulling data from the Elements API, i.e. it will try to pull just the data that has changed since the last time the harvester was run.

The harvester can be run in multiple modes by passing an argument on the command line:

- Default (no argument) : This mode will run either a full pull if it is the first run or a differential harvest on subsequent runs
- --full : forces the harvester to perform a full re-harvest, re-pulling all the data from Elements
- --skipgroups : this will instruct a differential run NOT to re-process the Elements group/group membership structures and instead to rely on the structures from the previous run.
- --reprocess : to reprocess the existing cached data against the current XSLT mappings without calling the Elements API at all (useful when developing custom mappings).

These different modes can be combined to create a harvest schedule using cron or another scheduling utility. e.g:

- Run a --skipgroups delta every 3 hours.
- Run a normal delta every day at 4 am.
- Run a --full on the last Sunday of each month.

Fragment Loader

The Fragment loader meanwhile has just one function, to load any fragments generated by the harvester in to Vivo, via Vivo's Sparql Update API. The fragment files generated by the harvester are timestamped and indexed, so they effectively form a queue which the fragment loader works through one by one. Note that the fragment loader is designed to operate as a daemon process (i.e. constantly running in the background).

Connector Design Philosophy

The Harvester extracts data from the Elements API by looping through three primary types of Elements data:

1. Objects (these include Users, Publications, Professional Activities, etc).
2. Relationships (the links between one object and another within Elements, e.g. confirmed authorship links between a user and a publication).
3. User Groups (groups of users that can be defined within Elements, primarily for reporting purposes).

For many institutions, this will represent a significant volume of data (several GB), so to be able to perform updates to Vivo in a reasonably timely manner we need an approach that minimises the amount of data being transferred. To achieve this, wherever possible, we retrieve data in a *differential* or *delta* based manner, requesting only the data that has changed since the last time the harvester was run.

The Internal Cache.

The approach that we take to enabling differential updates is to maintain a cache of both the raw Elements data and the corresponding transformed data for everything that "could" be included in Vivo, regardless of whether a particular item is actually being sent to Vivo at the current time. This cache is initialised on the first run of the harvester which performs a "full" pull of all the relevant data from Elements. Subsequent harvester runs update this internal cache with data extracted from Elements. Differential runs update just the data that has changed within Elements since the last harvester run, whilst full runs refresh the entire cache.

Having this cache means that if a particular object's (e.g. a user or a publication) status changes, meaning it should now be included in Vivo, then the relevant data is available, up to date, and ready to use, even if the change that made it suitable for being included in Vivo did not result in the object itself being modified.

Other approaches typically suffers from the problem that changes to user group membership (which are the primary driver of what data should be included in Vivo) are not tracked within Elements. Essentially, whilst the API supports asking the question:

"Show me all publications that are currently associated with Group X where the publication has been altered since date Y"

this query will not return information about any publications that have become newly associated with Group X since date Y but have not been otherwise been altered because Elements does not track group membership changes.

Technical Details

Skipgroups mode

The Elements API has resources for both Objects and Relationships that support using a "modified-since" parameter to allow retrieval of just the items that have been altered since the specified date, meaning that a differential approach is possible. The API resources for User Groups and Group memberships, however, do not.

Consequently both full and differential harvests update all their data about User Groups and Group memberships every time. This can be computationally costly, particularly for institutions with complex group structures within Elements, so the harvester maintains a separate cache of group membership information that can optionally be used during a differential update (a "skipgroups" run). When this happens any user group membership changes will not be reflected in the data sent to Vivo and any "new" users will appear only as members of the top level "organisation" group. This is particularly important as group membership is one of the key drivers for which data should be transferred to Vivo.

Differential update limitations

When using the connector with an Elements API Endpoint using an endpoint specification earlier than v5.5 (e.g. v4.9) there are some known issues which mean we cannot guarantee that the state of the connector's internal cache after a *Differential* run will be a 100% accurate representation of the data within Elements. It is therefore essential that a "full" harvest is performed periodically to fully refresh the cache (e.g. at least once a month).
For API's running endpoint specification v5.5 or later, differential updates should be reliable, regardless of activity during the harvest run. Nonetheless we would recommend running a "full" harvest occasionally to make absolutely sure the cache is correct (e.g. once a quarter).

Generating RDF

As data is retrieved from the Elements API, the XML fragment representing each individual Element's item (e.g. an individual User, Publication, Professional Activity, Relationship or User Group) is separated out, and passed into an XSLT pipeline for transformation. The output of this is a single XML-RDF file representing the Vivo Triples that should be added to Vivo if that specific Elements item is eventually transferred to Vivo.

Files representing the items raw (i.e. Elements API XML) and transformed (RDF XML) representations are then stored in the internal cache.

There are a few special features of the Transform process:

1) When transforming a Relationship item (e.g. an "association" link between a particular User and a Distinction Object) the connector can include the Elements API XML representing the objects related by the link as extra XSLT parameters that can then be used in the transformation.
   This is particularly useful when Vivo's representation of a particular concept does not match well with Element's representation (e.g. rich context objects, etc).
2) When transforming a User Group the connector includes a block of XML representing information about the Elements users who are both explicit members of the group within Elements and part of the set of users that will be transferred to Vivo, as an additional XSL parameter. This enables the transform to include those users as group members.
3) When transforming a user who has a profile picture the connector initiates a secondary transformation passing in XML detailing the locations and filenames of the final processed picture, enabling the pipeline to generate the necessary RDF to display that picture within Vivo.

The Transform process is conceptually fairly simple, but this XSLT pipeline is where majority of the heavy lifting is done, in terms of how Elements data is cross walked into Vivo Ontology RDF-XML. Consequently these XSL mapping scripts are where anyone looking to customise the the data being passed over to Vivo should start.

Note: After any alterations to the XSLT mapping scripts, it is essential to run either a "full" or a "reprocess" harvest run to ensure that the internal cache is loaded with transformed data that corresponds to the updated mappings.

Calculating the set of data to be transferred.

The set of data to be transferred to Vivo is primarily driven from the Elements User Groups configured to be included/excluded in the harvester's config file.

Note: not specifying anything in the config file corresponds to all users and groups being included.

Based on these configured groups, and user group membership information pulled from Elements, the harvester generates a set of Elements Users to be "included" in Vivo. The RDF files corresponding to the translations of these users are then added to the collection of data to be transferred to Vivo.

The connector then runs over its internal cache of raw data about Elements Relationships.

- For relationships between a User and another Object the connector checks whether the User is in the set being sent to Vivo. If so, the RDF files corresponding to the relationship, the user and the other object in the relationship are added to the collection of data to be transferred to Vivo.
- For relationships not involving a User, the RDF files corresponding to the relationship, and both objects in the relationship are added to the collection of data to be transferred to Vivo.

Note : this check loops through ALL relationships in the cache, regardless of whether this is a full or differential harvest. Therefore it processes all relationships, not just those that were updated this run.

We now have a collection of files representing the RDF data we would like to transfer to Vivo.

Minimising the data transfer

At this point we know the entire set of RDF data that we would like to see represented in Vivo but, unless this is the first time the connector has been run, it would be inefficient to transfer all this data as much of it will already be present in Vivo from previous runs. This is particularly relevant as importing data into Vivo can be a very slow process.

To improve this we load our entire set of data into a temporary triple store (Jena TDB) using bulk load techniques, then compare it to the equivalent temporary store generated during the previous harvest run. From this comparison we generate two RDF files (*additions* and *subtractions*) representing the changes that will need to be made to Vivo to bring it into line with the current desired state. We then transfer this data into Vivo via the Sparql Update API.

Loading data via the Sparql Update API

The very last step that the harvester performs is to split the additions and subtractions files into many small fragments (each typically ~1.2Mb in size). These fragments are tagged with the date and time of the connector run that generated them. This is done as Vivo's Sparql Update API has a default limit of 2Mb (when hosted within a typically configured Apache Tomcat servlet container). The gap between 1.2 and 2Mb is because we need to leave headroom to url-encode the data in each fragment.

The harvester process completes at this point, and leaves the final job of loading these fragments into Vivo to the Fragment Loader. The Fragment Loader process is designed to run as a daemon process and has just one function. It monitors the directory into which the harvester outputs fragments and processes new fragments as they appear, ordering them based on run date and type (i.e. is it a fragment representing additions or subtractions). For each fragment the Fragment Loader URL encodes the data and POSTs it to the Sparql Update API of the configured Vivo instance.

Using the Sparql API ensures that the imported data is fully processed by Vivo. This includes both inferencing and indexing the incoming data to ensure that Vivo has all the triples it requires (e.g. the vitro:mostSpecificType) and ensure that Vivo's Solr index is up to date.

Using the Sparql Update API therefore avoids the need to perform any one off tasks (e.g. "recompute inferences") to complete the data transfer, which means that the Vivo connector can be configured to run regularly (e.g. via CRON) without requiring any manual intervention.

Note : It can be beneficial to disable inferencing and indexing when ingesting many hundreds or thousands of fragments (e.g. after the initial harvest). Whilst this will mean you will need to run the "recompute inferences" task to complete the process, it can be significantly quicker overall.