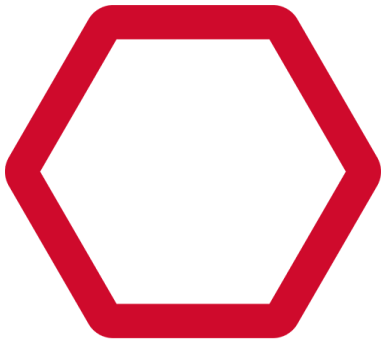


Symplectic Elements Harvester for VIVO:

Crosswalk Development Guide

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

Table of Contents

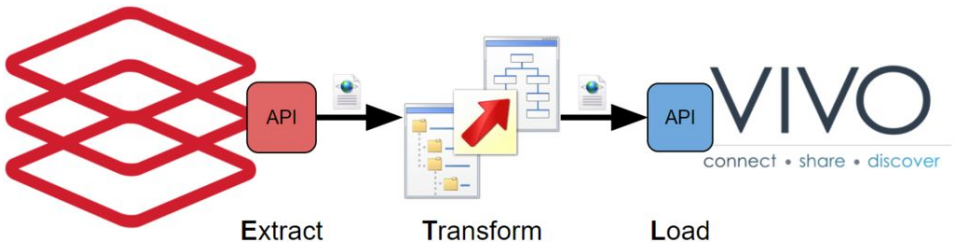
Introduction

This document covers technical details regarding how the main Harvester java process, interacts with, and passes data to, the eXtensible Stylesheet Language Transformation (XSLT) crosswalks, for translation into VIVO compatible format.

Note: *It is strongly recommended that you read the document "Symplectic Elements Harvester for VIVO : Description & Overview" prior to reading this document.*

Etl Pipeline

The Elements Vivo Connector is essentially an Extract, Transform and Load (ETL) engine.



In this document we are primarily concerned with the "Transform" step wherein Elements API output is transformed into a Vivo compatible form, i,e Resource Description Framework (RDF) data (Vivo ontology triples). To perform this translation the *ElementsFetchAndTranslate* Harvest process uses an XSLT pipeline. These XSLT files are known as the "*crosswalks*".

Note *: As covered in the "Description & Overview" document, it is the main Harvester process's responsibility to:*

- *Marshall the data coming from the Elements API.*
- *Establish what data needs to be transformed.*
- *Pass those items to the crosswalks for transformation.*
- *Calculate what data to send to Vivo.*

The main Harvester process is also responsible for handling the details of how to perform "full" or "differential" harvests. The crosswalks should not need to consider any of this.

The XSLT crosswalks should ***only*** *consider how to create the best (i.e. the most suitable) VIVO RDF representation of any data passed into them.*

The Elements Harvester for VIVO is shipped with a set of example "default" crosswalks. These are pre-configured to provide a set of mappings for Users, Publications, Grants, and a small subset of Professional and Teaching activities.

Note: *You are not required to use the default crosswalks at all, but they are a useful illustration of how the crosswalks interact with the harvester and contain a variety of useful utilities to make mapping data from Elements into Vivo easier. In the vast majority of cases it will be best to start with the default crosswalks and adapt/extend them as necessary.*

Individual Item Crosswalking

As the harvester retrieves data from the Elements API, the XML fragment representing each individual Elements item (e.g. an individual User, Publication, Professional Activity, Relationship or User Group) is separated out and passed into the XSLT crosswalks for transformation.
The crosswalks output should be a single XML-RDF file containing Vivo Triples that represent the passed-in Elements item.

The harvester then stores files of both the raw (i.e. Elements API XML) and transformed (RDF XML) representations in its internal cache. These files are effectively the *input* and *output* of the crosswalks for each processed item.

In a typical harvester installation these caches can be found at:

- %harvester-install-directory%/data/raw-records
- %harvester-install-directory%/data/translated-records

Note: *In a typical harvester installation these cached files will be stored on disk in compressed (gzipped) form. It is possible to configure the Harvester so that is will not zip these cached files (by setting* ***zipFiles = false*** *in elementsfetch.properties) but this will significantly increase the disk space utilisation.*

The "*input*" documents, as broken out from the paged API XML, generally have an **<atom:entry>** root element, beneath which there will be an **<api:object>** an **<api:relationship>** or an **<api:user-group>** element, depending on the stage currently being processed.
The internal structure of these API XML fragments is described by the API schema of the endpoint specification being used by the Elements API that the Harvester is connected to.

It is the job of the XSLT crosswalks to take these individual item inputs and return the necessary RDF-XML to represent each item appropriately in Vivo.

Multi Stage Transformation

As a harvest run is processed (regardless of run type), different types of Elements data are processed and passed into the Crosswalks at different stages.
The primary stages, in sequence are:

1. Objects
    a. User photo description
2. Relationships between objects
3. User groups
4. User group membership

Note: *Stage 1a "User photo description" is only triggered for "user" objects that have a profile picture set up within Elements.*

The Objects and Relationship stages are separated so that, for example, you only need to transform a publication once even if it has 50 internal co-authors. The separation also makes it easier, and more efficient, to handle differential updates.

Before the User group and User group membership stages can be processed, the Harvester must establish which Elements user groups are *actually* going to be transferred to Vivo (given the current Harvester configuration). Once this is known the groups and group memberships are processed taking account of the "apparent" group structure that will exist in Vivo, re-wiring parent groups and implicit group memberships where necessary.
Because of this dependency, these last two transformation stages have to be reprocessed in full every time the Harvester is run, regardless of run type (e.g. full, differential, etc)

The Harvester calls into the same XSLT file as the crosswalk entry point for all stages; this entry point is defined by "*xslTemplate*" in the main elementsfetch.properties config file. It is up to the crosswalk author to ensure that the different types of data passed in during each stage are processed appropriately.

URI Consistency

The data generated during these various stages typically contain "partial" Vivo representations of various objects. Because of this, it is vital that the different stages generate consistent URI's when describing the same object. For this reason we generally recommend using Elements "ids" when generating VIVO URIs for data. There are definitely exceptions to this rule, but it's a good rule of thumb.

As an example, consider three Elements items:

1. A user(id = 43)
2. A publication (id=78)
3. A relationship (id = 1005) (an authorship relating items 43 and 78)

In Stage 1, both of items 1 and 2 are translated into their VIVO compatible representations are created, each of which (as required by VIVO's design) are assigned a unique URI.
When the relationship is processed in Stage 3, we create an "Authorship" context object as its representation in VIVO, but within that object we need to point to the URIs of the VIVO representation of user 43 and publication 78.
If the URI's being used to refer to the same items are not consistent between the stages, then data will become disconnected within VIVO.

Interface Definitions

This section specified what data is passed into the crosswalks by the Harvester for each stage.

This includes both the primary XML input file to be processed and any additional data that the Harvester passes in to the crosswalk pipeline via defined XSLT parameters.
Not all stages use these additional parameters, but for others they are critical.

Note: *It is also possible to define "global" XSLT parameters by adding "xsl-param-******" lines in the main elementsfetch.properties file. These global parameters are passed to the crosswalks whenever the harvester invokes them. A good example is the ability to set the value of "xsl-param-baseURI" in elementsfetch.properties to specify the VIVO base URI that the crosswalks should use when generating URI's for VIVO objects.*

Stage 1: Objects

Stage Specification : Objects	
Data Processed	Any Elements object belonging to one of the categories being processed
Primary Input	/atom:entry/api:object[@category='%CATEGORY%']
Primary Input Description	The API XML representation of the Elements object to be transformed. The <api:object> will be at the "full" API detail level.
Additional XSLT Parameters	None

In this stage, data about the raw objects to be transformed (e.g. users, publications, grants, professional activities, etc) is passed from Elements to the crosswalks.

Note: *The categories of data that are considered by the Harvester are defined by "queryObjects" in the main elementsfetch.properties config file. User objects are* ***always*** *processed, even if the "users" category is not listed in queryObjects.*

Not all Elements categories will necessarily have a suitable representation in Vivo. The same may be true for some specific types of object within a category. For these cases the crosswalks should simply generate no output.

The example default crosswalks only contain transformations for "users", "publications", "grants" and a small subset of "professional and teaching activity" types. There is nothing to stop you adding other Elements categories (e.g. equipment) into "queryObjects", but unless you add new crosswalks, there will be no RDF output, and therefore nothing will appear in Vivo.

The XSLT crosswalks must define a template to match and process the passed in "api:object" data for any object categories/types that you wish to appear in Vivo. It is recommended to make use of XSLT's template matching features to specify different templates for different categories/types of data as appropriate.

Stage 1a: User Photo Description

Stage Specification : User photo description	
Data Processed	Any Elements user with a profile photo
Primary Input	/atom:entry/user-with-photo
Primary Input Description	The "user-with-photo" Element is a custom XML document constructed by the Harvester as the input to this special stage.
Additional XSLT Parameters	None

For any users with a profile photo in Elements, the Harvester retrieves the photo and processes it. It creates two resized versions of the photo in jpeg format, and stores these on disk. The harvester than creates a custom XML document to describe the generated photos and calls into the crosswalks, passing in the custom document as an input.
The crosswalks must generate the necessary triples to define where the generated images can be reached to Vivo.

For each user, the constructed input document has this structure:

```
<atom:entry>
    <user-with-photo id="$val" username="$val" proprietary-id="$val">
```

```
        <full-image-path-fragment>$val</full-image-path-fragment>
        <full-image-fileName>$val</full-image-fileName>
        <thumbnail-image-path-fragment>$val</thumbnail-image-path-fragment>
        <thumbnail-image-fileName>$val</thumbnail-image-fileName>
    </user-with-photo>
</atom:entry>
```

The attributes of the "user-with-photo" element describe the user that this data applies to, whilst the sets of *fileName and *path-fragment elements describe the literal filename (just the name not the full path) and the relative URL (relative to Vivo's base URL) where that photo file should be made available.

Note: *making sure the generated photos are available at the correct relative urls can require custom Tomcat configuration (see the Harvester installation guide for more details).*

The XSLT crosswalks must define a template to match and process the passed-in "user-with-photo". In this case the output is fairly standard. This transformation is only part of the crosswalks because it fits a generic pattern. It will almost never be necessary to alter the output of this stage from that generated by the default crosswalks.

Stage 2 : Relationships between objects

Stage Specification : Relationships between objects	
Data Processed	Any relationships between objects, where each linked object belongs to one of the Elements categories being processed (i.e queryObjects + users)
Primary Input	/atom:entry/api:relationship
Primary Input Description	Elements API XML of the relationship to be transformed. The <api:relationship> will be at the "ref" API detail level.
Additional XSLT Parameters	**extraObjects** *(optional)* This can contain an XML document containing api representations of each of the linked objects at "full" detail level

In this stage the crosswalks are passed data about Elements relationships (e.g. *publication-user-authorship*, *publication-user-editorship*.etc). The Harvester will call into the crosswalks for any Elements relationship (regardless of relationship type) for which each of the linked objects belong to one of the categories being processed ("*queryObjects*" + *users*).

Not all Elements relationship types will necessarily have a suitable representation in Vivo; for these cases the crosswalks should simply generate no output.

The example default crosswalks only contain translations for a subset of Elements relationship types. They cover most publication-user relationships, as well as grant-user relationships. They also cover user-user collaborations and a small subset of professional/teaching activity-user associations.

Note: *Activity associations (professional and teaching) are handled in a very particular way in the default crosswalks. Translation of both the activity and the relationship are deferred until a relationship containing the activity is processed here in Stage 2 (with the object data being passed in as "Additional Object Data" - see below). This is necessary because the VIVO representation of these items is so different from the Elements (item-relationship) model that the approach, of creating partial representations across different stages, falls apart.*
It is worth noting that this approach is not too much of a performance issue for these categories of data, as activities are typically only associated with one user (unlike co-authored publications).

Additional Object Data

The <api:relationship> data passed in to the crosswalks during this stage is at the "ref" API detail level. It is therefore only a short summary of the relationship and the linked objects, along with API references (API url links) to the associated objects.

In some cases this summary data will be all that is necessary to generate the appropriate RDF triples to represent the relationship in Vivo. In other cases however, particularly where the object type influences the transformation or the Vivo representation places data from the Elements objects on the "context" object within Vivo, the crosswalks need access to richer data about the linked objects at this stage.

To account for this type of requirement the Harvester is capable of passing in a custom XML document containing the "raw" full detail API description of both linked objects (retrieved from its internal cache). This data is passed to the crosswalks via an XSLT parameter named "extraObjects".

The "extraObjects" document has this general structure:

```
<extraObjects>
    <atom:entry>
        <api:object id="$val" category="$val" ...>
            <!-- full detail object entry for linked object 1 -->
        </api:object>
    </atom: entry>
    <atom:entry>
        <api:object id="$val2" category="$val2" ...>
            <!-- full detail object entry for linked object 2 -->
        </api:object>
    </atom: entry>
```

```
</extraObjects>
```

Note: this "extraObjects" data is only provided to the crosswalks for specific Elements link types: You can specify the types (by name) in "*relTypesToReprocess*" in the main elementsfetch.properties config file.
By default, if *relTypesToReprocess* is not specified, extraObjects data will only be provided for the following link types:

- publication-user-authorship
- activity-user-association
- user-teaching-association.

The latter two here are present to support the deferred manner in which the default crosswalks process "activity" objects.

The XSLT crosswalks must define a template to match and process the passed in "api:relationship" data for any relationship types that you wish to appear in Vivo. It is recommended to make use of XSLT's template matching features to specify different transformations for different relationship types, or at an even more fine grained level, as appropriate.

Stage 3: User groups

Stage Specification : User groups	
Data Processed	All Elements User Groups
Primary Input	/atom:entry/api:user-group
Primary Input Description	Elements API XML of the user group to be transformed The Elements API representation of user groups does not have multiple detail levels.
Additional XSLT Parameters	**includedParentGroup** This contains an XML fragment describing the nearest ancestral group (from the Elements group structure) that **will** be included in Vivo.

The Vivo Harvester allows the user to configure which Elements user groups should be included in Vivo by use of "*include*, "*exclude*" and "*excise*" lists.

This means that the group "tree" as transferred to Vivo may be quite different from Elements. Whole sections of the tree may be missing, individual nodes removed (and their children raised up), etc. You can even end up with multiple separate unconnected trees within Vivo.

To account for this, the User group and group membership stages are deferred until after the

Harvester has calculated which user groups are *actually* going to be sent to Vivo. The Harvester then uses additional XSLT parameters to pass in data to allow the crosswalks to generate correct information based on the group tree as it will be represented in "Vivo".
As an additional consequence, the Harvester always processes these stages in full regardless of the harvest type (i.e. full/differential/etc).

For this stage the additional parameter includedParentGroup contains a very small XML fragment of this form:

```
<group id="$val" name="$val" />
```

This represents the nearest "ancestral" group from the Elements group tree that is *actually* going to be sent to Vivo. The crosswalks should only use this value to construct any hierarchical relationships within Vivo. The crosswalks should **NOT**, for example, make use of the ***api:children/api:child*** data within the passed-in ***api:user-group*** as some of the current groups' children may well be excluded from Vivo under the current Harvester configuration.

The XSLT crosswalks must define a template to match and process the passed-in "api:user-group" data and the includedParentGroup parameter if you want user groups to appear in Vivo.

Stage 4 : User group membership

Stage Specification : User group membership	
Data Processed	All Elements users. This is a second pass through users to allow for Group Membership processing based on the groups that are actually going to be included in Vivo
Primary Input	/atom:entry/api:object[@category='**user**']
Primary Input API detail level	API XML of the user whose group membership is being processed. The <api:object> representation will be at the "full" API detail level
Additional XSLT Parameters	**userGroupMembershipProcessing = "true"** This is passed here to allow the crosswalks to filter to different templates for this stage (as opposed to those for transforming the primary objects in stage 1). **userGroups** This contains an XML document describing all the groups for which the user is an explicit member of AND which are to be included in Vivo.

Note: *We deal here with "explicit" group memberships within Elements, i.e. those where a user is a member of a particular group because either they were explicitly added to a manual group or because their HR data matched a selector (e.g a primary group descriptor or a where clause). We are not dealing with implicit memberships (i.e. those where members of a sub-group are implicitly members of that groups parents).*

As with the user groups stage, the group memberships shown in Vivo are affected by the current configuration of the "*include*", "*exclude* and "*excise*" group parameters in elementsfetch.properties. Consequently this stage is delayed until after the Harvester knows which groups are to be sent to Vivo, and it is processed in full every harvest, regardless of harvest type.

In particular, if a user is an explicit member of Elements group that has been marked as "*excluded*" but there is a parent group of that excluded group which is being "included" in Vivo, then the user's "membership" of the excluded group is "re-wired" to the nearest included parent. The Harvester process does this rewiring internally to generate a list of groups, from the subset that are being sent to Vivo, that the user is "effectively" an explicit member of.

During this stage the raw API representation of every user being sent to Vivo is passed to the crosswalks again. An XSLT parameter ("*userGroupMembershipProcessing*") is passed with its value simply set to "true", to allow you to more easily define a different transformation specific to this stage.
A secondary additional XSLT parameter ("*userGroups*") is also passed containing an XML document listing the various groups that the current user is an "effective member" of.

The "*userGroups"* XML document has this structure:

```
<usersGroups>
    <group id="$val1" name="$val1" />
    <group id="$val2" name="$val2" />
    <!-- there will be one "group" element listed for every group
    that the user is an effective member of -->
</usersGroups>
```

Note: *the XSLT parameter is named userGroups, but the XML root is user**s**Groups. This is simply an oversight, the peculiarity is currently retained for backwards compatibility reasons.*

The group membership is processed in this manner so that you have access to information like the user's HR fields and their "position" information whilst you are generating the triples that define that user's membership of groups within Vivo. This allows for the possibility of using matching techniques to try and cross reference between these various aspects of the data.
Note*: Whilst potentially powerful, "matching" approaches like this are generally difficult to set up and quite brittle. Consequently no such techniques are used in the default crosswalks.*

The XSLT crosswalks must define a template to match the passed in "api:object[@category='user']" and process the effective group membership data passed in via the "userGroups" XSLT parameter to define group membership within Vivo.
It is necessary to use the "userGroupMembershipProcessing" parameter to change template "mode" to ensure you are targeting a different transformation than that used in the Objects Stage.

The "Default" Crosswalks

The default crosswalks provided with the Elements VIVO harvester do not represent a "complete" or "perfect" mapping in any way but rather a good starting point. They hopefully serve as a useful illustration/framework, as well as providing a set of tools and utilities to develop from.

The defaults compromise ~6000 lines of XSLT split across more than 35 files. It is therefore vital that you are familiar with XSLT **2** , before attempting to adapt or extend them. It is worth paying particular attention to XSLT's template matching and import precedence features operate.

Note: *The defaults have evolved over many years, three completely different versions of the Vivo harvester, multiple versions of the Elements API and a wide variety of Vivo projects. Furthermore they have been contributed to by many hands, both within Symplectic and via Open-Source contributions. The defaults are not therefore, the cleanest or most consistent set of examples; they are however, quite powerful and configurable*

This document will not document the default crosswalks in depth, nor the mappings they create. It will instead, describe the rough structure of the crosswalks, how the files relate to each other, and note any areas of particular interest. It will also describe the main options that can be configured.

Crosswalk File Structure

Below is a diagrammatic representation of crosswalk structure (import/include indicated by level in tree)

- elements-to-vivo.xsl *(Main entry point)*
    - elements-to-vivo-config.xsl
        - elements-to-vivo-config.xml *(Main crosswalk config file)*
    - elements-to-vivo-object.xsl *(Stage 1 : implemented by imported files)*
        - elements-to-vivo-object-user.xsl
            - elements-to-vivo-object-user-vcard.xsl
        - elements-to-vivo-object-publication.xsl
        - elements-to-vivo-object-grant.xsl
        - elements-to-vivo-object-professional-activity.xsl
        - elements-to-vivo-object-teaching-activity.xsl
    - elements-to-vivo-user-photo.xsl *(Stage 1a)*
    - elements-to-vivo-relationship.xsl *(Stage 2 : implemented by imported files)*
        - elements-to-vivo-relationship-publication-author.xsl
        - elements-to-vivo-relationship-publication-editor.xsl
        - elements-to-vivo-relationship-publication-translator.xsl
        - elements-to-vivo-relationship-publication-grant.xsl
        - elements-to-vivo-relationship-user-grant.xsl
        - elements-to-vivo-relationship-collaborator.xsl
        - elements-to-vivo-relationship-professional-activity.xsl
            - elements-to-vivo-relationship-professional-activity-committee-membership.xsl
            - elements-to-vivo-relationship-professional-activity-distinction.xsl
            - elements-to-vivo-relationship-professional-activity-event-administration.xsl
            - elements-to-vivo-relationship-professional-activity-event-participation.xsl
            - elements-to-vivo-relationship-professional-activity-fellowship.xsl
            - elements-to-vivo-relationship-professional-activity-membership.xsl
        - elements-to-vivo-relationship-teaching-activity.xsl
            - elements-to-vivo-relationship-teaching-activity-course-developed.xsl
            - Elements-to-vivo-relationship-teaching-activity-course-taught.xsl
    - elements-to-vivo-group.xsl *(Stage 3)*
    - elements-to-vivo-group-membership.xsl (Stage 4)
    - elements-to-vivo-utils.xsl *(General utility functions for mapping Elements data to Vivo RDF-XML)*
        - elements-to-vivo-datatypes.xsl
            - elements-to-vivo-utils-2.xsl *(Broken out to avoid circular dependencies)*
        - elements-to-vivo-datatypes-matching.xsl
        - elements-to-vivo-fuzzy-matching.xsl
    - elements-to-vivo-template-overrides.xsl *(Override files to allow overriding of templates/functions)*
    - elements-to-vivo-util-overrides.xsl

***Note**: the two "override" files are added to the top level "elements-to-vivo.xsl" file using `<xsl:include>` statements instead of `<xsl:import>`. Therefore any templates/functions added to them will be at a higher precedence in the xslt scope chain than the definitions in the other files.*

Key Features

- **The main config file**
  (elements-to-vivo-config.xml)
  Note: *This file is fairly well self-documented - it is worth reading the comments*
  This file allows you to configure various aspects of the crosswalk's behaviour including:
    - The precedence order to use when processing items with records from multiple Elements data sources.
    - Which Elements label schemes should be mapped to Vivo (for pubs and users).
    - What "type" of Vivo object to create for a given Elements publication.
      Typically based simply on Elements object type.
    - What "type" of Vivo organisation to create for a particular org name.
      This can apply to Vivo organisations created to represent Elements groups, or just to represent some address information in Elements.
    - Etc.
- **Ability to override aspects of the config file via XSLT params**
  (elements-to-vivo-config.xsl)
  The config file is parsed by the corresponding xsl file, and in some cases the configured data is pulled into the default value of an xsl:param (e.g. baseURI, internalClass).
  For these cases you can "override" the value configured in the main crosswalk config file by using xsl-param-xxxx settings in the Harvester's elementsfetch.properties config file.
- **Ability to configure tests via XSLT params**
  (elements-to-vivo-config.xsl and elements-to-vivo-utils.xsl)
  To facilitate testing the default crosswalks allow you to alter their normal behaviour by passing additional XSLT params:
  You can pass the parameters "useRawDataFiles" and "recordDir" to have them attempt to find "Additional Object Data" by looking for it on the filesystem beneath the configured "recordDir" (referencing the expected file layout of the raw records in the internal cache, i.e. user, publication, grant, relationship, etc folders).
  The default crosswalks can also accept a "filename" as the input for any of these XSLT parameters (the referenced file then containing the relevant XML fragments):
    - Additional
    - includedParentGroup
    - userGroups
- **User translations respect privacy settings**
  (elements-to-vivo-object-user.xsl and elements-to-vivo-object-user-vcard.xsl)
  Note how the crosswalks do not map across any data that is marked as "private" in the User's profile.
- **Publication "translation contexts"**
  (elements-to-vivo-object-publication.xsl and elements-to-vivo-relationship-publication-author.xsl).
  Vivo represents some items, that Elements models as publications, as "Events" (e.g. presentation, exhibition, etc). To handle this we determine a "*translation context*" based on the type of the Vivo object being created (as determined by the configuration

mentioned above). Based on this context we fine tune both the translation of the actual object, and more relevantly type of the context object and the related linking properties used to represent associated people (e.g person-list data such as the list of authors). This same switch to use different types of context objects also needs to be made when processing the relationships that link publications to users (e.g authorships, etc). To facilitate this we ensure that we pass in the "Additional Object Data" during Stage 2 so that the relationship transformation can establish the correct "translation context" from the raw publication data.

- **Deliberate ObjectToObject URI's used for Context objects representing "resolved" people**
  (elements-to-vivo-relationship-publication-author.xsl, elements-to-vivo-relationship-publication-editor.xsl & renderLinksAndExternalPeople function in utils file).
  When processing (for example) the author list of a publication in Stage 1, we have to create a set of "Authorship" context objects (1 per author) for Vivo. These context objects are then linked to both the "publication" and to a "vCard" representation of the author's name. At a later point (during Stage 2) we will want to come along and add a link to the actual "user" to the relevant "Authorship" objects for any approved links. During Stage 1, however, we do not know the ids of the relationships that link this publication to Elements users, so we have a URI consistency problem.
  We potentially do know the user id of the Elements user that a particular author in the author list might correspond to (as this is provided in the Elements API representation of the publication if the author name has been resolved by Elements). Therefore to allow us to come along in Stage 2 and update the correct authorship object with a link to the relevant "Person" object in Vivo we deliberately construct the "Authorship" URI's using the pair of ids of the objects being linked, not the id of the relationship.
- **"Invisible link" processing**
  (all *-relationship-*.xsl files).
  Essentially only "Selected publications" (authored pubs) and the various "Investigator on/Other research activities" (grants) properties support any form of "hiding" in Vivo. For this reason the crosswalks only process "invisible" links of relevant types.
  Note: *whether the main Harvester process even attempts to process invisible links is a configuration option outside of the crosswalks.*
- **Non-existent "Activity" object translations**
  (elements-to-vivo-object-professional-activity.xsl and elements-to-vivo-object-teaching-activity.xsl)
  Both these files are essentially redundant as no mapping is performed in Stage 1 for these types of data. All translation is deferred until Stage 2 with the object data being passed in as "Additional Object Data".
- **Complex template "match" rules for "Activity" relationships**
  (all *-relationship-professional-activity-*.xsl and *-relationship-teaching-activity-*.xsl files)
  Activity objects are generally linked to their users by a single type of link ("activity-user-association" or "user-teaching-association" respectively).
  As the entire transformation is deferred until Stage 2 for these activity items, the templates that do the processing for the different types of object have to use quite specific filtering in their "match" attribute to ensure that the correct mapping is

performed.

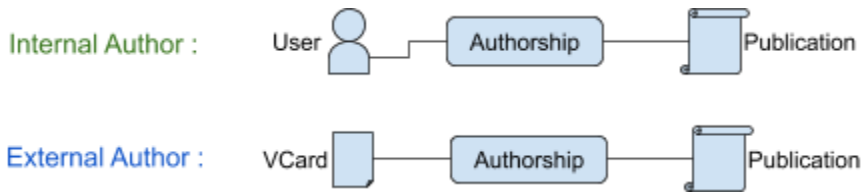
- **Utility functions in (elements-to-vivo-utils.xsl)**
  The utils file contains a swath of useful functions for a whole range of tasks including:
  - URI generation and manipulation
  - Determine correct Vivo object Type (based on Elements object data)
  - Determining the correct "translation context"
  - Creation and handling of "Organisation" objects (e.g. from address data)
  - Creation of "Date" and "DateInterval" objects
  - Handling of Elements "field" nodes from source API XML
    - Retrieving nodes following configured precedence rules from config file
      getRecordField
    - Rendering a given node as a Vivo property (based on field type)
      renderPropertyFromFieldNode
    - Combined retrieval and rendering
      renderPropertyFromField
  - Handling of "person lists" based on context
    renderLinksAndExternalPeople
  - Templates to simplify creation of "RDF" objects
- **Override files (elements-to-vivo-template-overrides.xsl and elements-to-vivo-util-overrides.xsl)**
  These files are provided devoid of any content, but they are included into the main elements-to-vivo.xsl file in such a way that any functions or templates defined in them will be at a higher "import precedence" than all the other crosswalk files.
  You can therefore add templates and functions to these files to override behaviour without needing to alter other files. This can make it easier to combine your changes if new features are added to the default crosswalks (e.g. in a future release).

Related Vivo Configuration

If you use the default crosswalks, or customised crosswalks based on the defaults, there are some parts of Vivo where it is advisable to apply certain configuration changes.

List View Configurations - VCard issues.

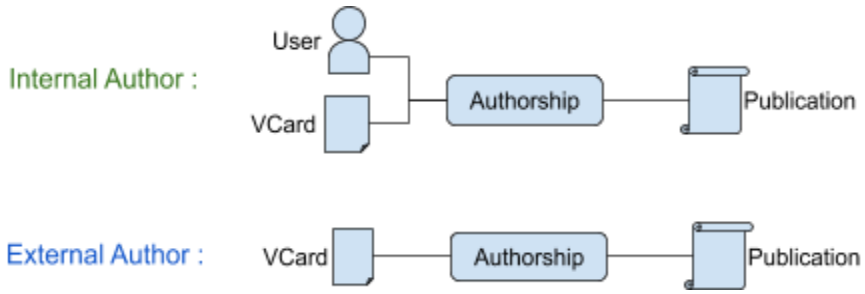
When creating "context" objects (e.g. authorships/editorships/roles), the default crosswalks make use of "VCard" objects to represent people that do not have a profile in Vivo (e.g. co-authors of an academic paper from another institution).



This is in line with what Vivo expects :

https://wiki.duraspace.org/display/VTDA/The+W3C+vCard+ontology+in+VIVO

In fact, the default crosswalks deliberately create a VCard linked to the context object regardless of whether there is an actual Vivo user present:



This is done to ensure that all authors/editors/etc are listed even if a user's relationship with a publication is marked as "hidden". It also allows for situations where a user's published name, as listed on the paper, does not match the user's current name.

Unfortunately Vivo's out of the box support for VCards in context objects is not entirely complete. For example it does not handle "VCard's" in "editorship" objects, nor can all aspects of Vivo cope with context objects containing links to both a user object and an equivalent Vcard representation of the associated user.

These problems can be addressed in a variety of ways, one of which is to customise Vivo's listViewConfig files to better handle VCard data in context objects. To that end, the "*example-integrations/vivo-list-view-configs*" directory within the harvester install kit provides some example replacements for Vivo v1.9.3:

- **listViewConfig-informationResourceInEditorship.xml**
  This adds the ability for "Editorship" links in Vivo to list VCard data.
- **listViewConfig-informationResourceInAuthorship.xml**
  This updates the view so that the VCard name will be shown in preference to the user's current label if both types of data are present.
- **listViewConfig-relatedRole.xml**
  This fixes some buggy behaviour around Vcards and makes things consistent in terms of which names are listed.

If you wish to make use of these, the "updated" files should be deployed to the "config"

directory within your deployed Vivo webapp.

Notes on Controlled Vocabularies

The default crosswalks have the ability to map Elements labels (the MeSH, Science Metrix and Fields of Research schemes by default) to Vivo “Concept” objects. In order to display these as being “part” of an external vocabulary within Vivo (e.g. so they listed as being part of the “MeSH” scheme), you need to add the contents of "add-to-vocabularySource.n3" to the file

- <vivo>/home/rdf/abox/filegraph/vocabularySource.n3

To be specific, for each “vocabulary” that you want to define you need to add two lines (representing two triples) to vocabularySource.n3.

```
<http://www.nlm.nih.gov/mesh> <http://www.w3.org/1999/02/22-rdf-syntax-ns#type>
<http://www.w3.org/2002/07/owl#Thing> .

<http://www.nlm.nih.gov/mesh> <http://www.w3.org/2000/01/rdf-schema#label>
"MeSH"^^<http://www.w3.org/2001/XMLSchema#string>  .
```

Here we are specifying that the item represented by the URI “http://www.nlm.nih.gov/mesh” is a “thing” (a vocabularySource) and that it has the label value (i.e. should be called) “MeSH”.

If you choose to map any "custom" Elements label schemes, by editing the crosswalk config file elements-to-vivo-vonfig.xml, you can specify the Vivo URI that defines the scheme then add your own lines to "vocabularySource.n3" to define how your chosen scheme is listed in VIVO.