Simple VIVO

Mike Conlon

October 6, 2015

Contents

Authors – stubs, local authors, disambiguation. ORCID, other. Big topic.To add awards and honors to VIVO, we need to know several pieces of information for each award or honor:

1. Who got the award or honor?
2. What award or honor was received?
3. From what institution?
4. What is the date of the award?

Managing Enumerations

Each is represented by an enumeration:

1. person_enum for who
2. award_enum for what
3. org_enum for where
4. date_enum for when

The python program `make_enum.py` can be used to update these enumerations using data from your VIVO. To run `make_enum.py` use:

```
python make_enum.py
```

Managing Awards

You can use

```
python sv.py -a get
```

to get the awards from your VIVO into a spreadsheet called `awards.txt` You can add awards to your spreadsheet being sure to use values from each of your enumerations.

Once you have awards to add to VIVO, run:

```
python sv.py -a update
```

which will create the `award_add.rdf` and `award_sub.rdf` to update your VIVO.

Adding People, Dates, Awards and Organizations to your VIVO

To manage awards, your enumerations must be up to date. Each award, person, organization, and date must already be in your VIVO and in your enumerations. Updating your enumerations is simple – run `make_enum.py` as shown above.

If you need to add dates, rerun the dates example with a wider date range.

To add a single person, name of an award or an organization conferring an award, use the VIVO web interface. Update the enumerations, add a row to your awards.txt for the new award and run a Simple VIVO update.

For example:

John Smith earns a lifetime achievement award from the American Cancer Society in 2015. John Smith and the date 2015 are already in your VIVO, but the American Cancer Society and the lifetime achievement award are not.

1. Use the Site Admin interface to add the American Cancer Society as an organization.

2. Use the Site Admin interface to add "ACS Lifetime Achievement Award" as an award.
3. Update the enumerations. Now all four elements of the award receipt are in your VIVO and you can add a row to your awards.txt spreadsheet that looks like

   `John Smith ACS Lifetime Achievement Award American Cancer Society 2015`
4. Run

   'python ../../sv.py -a update"

and the award will be in place.

One of the great advantages of VIVO is its ability to use controlled vocabularies to manage concepts across domains. Your grants, papers, fields of study, personal research areas and others can all be coded using vocabularies that are appropriate for the job.

Managing concepts in Simple VIVO is straightforward. You edit concepts in concepts.txt as a spreadsheet. You use update to update VIVO. You use get to retrieve concepts from your VIVO.

Get started

Run `make_enum.txt` to create an enumeration of the concepts in your VIVO. This will allow you to use concept labels in your spreadsheet to refer to concepts that are "narrower" or "broader" than the concept you are editing.

```
python make_enum.txt
```

Get Concepts from VIVO

Use

```
python sv.py -a get
```

Edit your Concepts

Use a spreadsheet program such as Excel or Numbers to open `concepts.txt` Edit and save.

Update your VIVO

Use

```
python sv.py -a update
```

Each time you wish to update your concepts – add new concepts, add narrower or broader concepts to existing concepts, change the labels of concepts – you can repeat the steps above:

1. update the enumerations using `make_enum`
2. get the concepts from your VIVO into a spreadsheet using `sv.py -a get`
3. edit the concepts using a spreadsheet program
4. update VIVO using `sv.py -a update`To manage teaching in Simple VIVO, you will need to manage a list of courses and specify teaching – who taught what when.

Each is described below.

Courses

Managing a list of courses is easy. Courses have two columns – the name of the course and the concepts of the course. If your courses have course numbers, you may want to include them in the name of the course. So, for example, if the name of your course is "Introduction to Statistics" and the course number is "STA 1201," you may want to have the name in VIVO be "STA 1201 Introduction to Statistics"

You can use:

```
python -a get -c sv_courses.cfg -s courses.txt
```

to get a spreadsheet of the courses you have in VIVO. Add the names of courses, or edit the names as needed, and then update your list in VIVO using:

```
python -a update -c sv_courses.cfg -s courses.txt
```

Teaching

To manage teaching, you will need a list of courses in VIVO (see above).

For each course taught by an instructor, you will need to know:

1. The course that was taught – by name of the course
2. The person who taught the course – by orcid
3. The start date for the course – year month day by value
4. The end date for the course – year month day by value. For example "2015-12-09"

You can use:

```
python make_enum.py
```

to prepare the enumerations for your work. Then use:

```
python sv.py -a get
```

to get a spreadsheet of your courses. Add to it, edit, remove as needed.

You can then use:

```
python sv.py -a update
```

to update VIVO to include your changes.

Repeat these four steps – make_enum, get, edit, update – whenever you would like to edit your teaching records.Dates in VIVO are just like other things in VIVO – they have URI and they have attributes. Unlike other things, however, VIVO creates the same dat many time, giving each its own URI.

Using Simple VIVO, we can create a set of dates for VIVO that are available for reuse in managing all other entities.

For example, when entering a publication in VIVO, one is asked to enter the date of the publication. Rather than create a new date for each publication, we can create a set of dates that are reused for *all* publications. This have several advantages over creating dates for each publication:

1. It avoids semantic confusion. Having two dates that represent the same thing (the same date) but have different URI is not a best practice. We could assign "sameAs" predicates to equate all all the various entities representing the same date, but this just layers complexity on an unfortunate situation.
2. It is simpler. There is only one "December 18, 1993." All things that need to refer to this date need to refer to the same entity in VIVO.
3. It saves space. As VIVO grows and adds publications, it adds dates. But this is unnecessary. There is a "slow growing" number of dates (one per day!)
4. It supports look-up by date. Rather than find all the entities with the same date value and then finding all the things associated with each of those values, we can find all things associated with the single entity that represents the date of interest.

Date Precision

VIVO Supports the concept of "date precision." This is a little unusual in the big world of data management, so let's take a look at what VIVO can represent. VIVO provides three precisions for date values. A date can be just a year, or just a year and a month (common in publication dating) or a year and a month and a day (common in most administrative function). In some cases, systems other than VIVO are "required" by their internal formats to provide a day, month and year for all dates. Many systems default the unknown components of the date to "1". This means that when looking at a date (without a precision) of "1993-01-01" we are unable to discern whether the date value is informative about the month and day, or whether these are just place holders. VIVO is capable of representing, with certainty, the level of precision in the date value. The date value "1993-01-01" with a date precision of "http://vivoweb.org/ontology/core#yearPrecision" is the same as a date value of "1993" with the same year precision. The month and day values are non-informative.

VIVO date precision	Example	Since Jan 1, 1800 until Dec 31, 2050
http://vivoweb.org/ontology/core#yearMonthDayPrecision	"1993-12-18"	91,676
http://vivoweb.org/ontology/core#yearMonthPrecision	"1993-12"	3,012
http://vivoweb.org/ontology/core#yearPrecision	"1993"	251

Pre-loading VIVO with dates

We can pre-load VIVO with dates using the dates example in Simple VIVO. We can set a range of dates that will be useful for our VIVO, generate them and update our VIVO with them. Subsequent updates can use the dates that are already in VIVO, rather than making more.

examples/dates

The dates folder in examples has everything we need to pre-load our VIVO with dates.

`gen_dates.py` is an optional Python program for generating a spreadsheet of dates data. By default it generates dates from Jan 1, 1800 to Dec 31, 2050. This should be more than enough for many VIVOs. Dates are generated for each year, for each year/month combination and for each date in the range. By modifying `gen_dates.py` it can produce dates in any range.

`dates.txt` is a tab delimited spreadsheet ready for the pump.

`sv.cfg` is a config file that defines the dates update. You will need to edit this to supply query parameters for your site. Your system administrator will be able to supply you with appropriate query parameters.

`date_def.json` is a definition file for putting dates in VIVO. It defines three columns: 1) uri, 2) Precision, and 3) Date.

`datetime_precision_enum.txt` is an enumeration of the VIVO datetime precisions that allow the use of short codes "y", "ym" and "ymd" for the precisions described above.

To run the example, use:

```
python sv.py
```

The result will be two files: `date_add.rdf` and `date_sub.rdf` Use the VIVO system administration interface to add `date_add.rdf` to your VIVO. The result will be 94,939 dates. You may never have to be concerned about dates again.

Here we describe how to manage educational background using Simple VIVO. Degrees, institution granting the degree, person receiving the degree, date of the degree, and field of study. Should be very similar in concept and complexity to positions.

To add education backgrounds to VIVO, we need to know several pieces of information for each degree:

1. Who obtained the degree
2. What degree was obtained?
3. From what institution?
4. What is the date of the degree?
5. What is the field of study for the degree?

Managing Enumerations

Each is represented by an enumeration. This guarantees that all the information regarding education in VIVO is "coded" – there are no open-ended text fields associated with educational background. If something can not be found in the enumerations, it will need to be added.

1. person_enum for who
2. degree_enum for what
3. school_enum for where
4. date_enum for when
5. field_enum for field of study

The python program `make_enum.py` can be used to update four of these enumerations from your VIVO. To run `make_enum.py` use:

```
python make_enum.py
```

The field of study enumeration is managed by hand – if you have fields of study that you would like to be able to represent in VIVO, edit the field_enum.txt file with a text editor and add a row for each such field of study.

get and update as always

1. Edit your `sv.cfg` file to provide the query parameters for your VIVO. Your system administrator can help you with this.

2. Run get:

   ```
   python sv.py -a get
   ```

The result will be a degrees.txt file which you can open in a spreadsheet to edit the degrees for your people. Every value you enter in each of the columns should be a value in one of the enumerations. Each column has its own enumeration.

Edit your enumerations as needed to provide additional values for degrees and fields of study.

To add universities or other schools to your VIVO and make them available for your education work, use the orgs example to add the organizations, then run make_enum to update your enumerations. The new schools are now available for putting educational background in VIVO.

Once you have a spreadsheet with degrees that you would like to use to update VIVO:

1. Run update

   ```
   python sv.py -a update
   ```

The information in your `degrees.txt` file will be used to update degree information in VIVO.

Grants are a bit more complex than some of the other entities we have been managing using Simple VIVO. Grants have the following columns:

1. Local award ID – that is, what number, or other identifier, does your institution give this grant
2. Title – the title of the grant. This should be spelled out – do not use abbreviations. There is no length limit.
3. Direct costs – all years – a number without a dollar sign and with decimals as needed. For example `43506.45`
4. Total award amount – all years, direct plus indirect. Same format as direct costs.
5. Principal investigators – one or more principal investigators. Investigators are specified using their orcid ID and separated by your intra field delimiter.
6. Co-principal investigators – any number of co-principal investigators. Specified using orcid, separated by intra field delimiter.
7. Investigators – any number of investigators. Specified using orcid, separated by intra field delimiter.
8. Concepts – any number of concepts identifying subject areas for the grant. Concepts are specified by name as they appear in the`concept-enum.txt` file. Concepts are separated by the intra field delimiter
9. Start date in the form yyyy-mm-dd
10. End date in the form yyyy-mm-dd
11. Administering unit. Specified by name as they appear in `dept_enum.txt`
12. Sponsor. Specified by name as they appear in `sponsor_enum.txt`
13. Sponsor's award ID – how does the sponsor refer to this grant. The National Institutes of Health in the US, for example, gives its grants award ID that look like R01DK22343
14. URI in VIVO. Blank if you are adding the grant. VIVO URI returned by get.

Enumerations

Simple VIVO for grants uses several enumerations to identify data in VIVO. Each enumeration is made by`make_enum.py` as described below.

1. `concept_enum.txt` lists the concepts in your VIVO by name
2. `date_enum.txt` lists the year-month-day dates in your VIVO by date value
3. `dept_enum.txt` lists your institutions organizations by name
4. `orcid_enum.txt` lists your investigators by ORCID. ORCID is used to identify your investigators. Each should have an ORCID, and the ORCID for each investigator should be in your VIVO. See examples/people for adding ORCID to people.

5. `sponsor_enum.txt` lists the sponsors (funding organizations) in your VIVO.

Each attribute is optional. You may add a grant with very little information and provide improved values for attributes in subsequent updates.

As always, your spreadsheet values will be non-blank if you would like to specify a value for the attribute, blank if you would like to have VIVO remain unchanged, and "None" if you would like to remove the attribute's value from VIVO.

The Basic steps

1. Use `make_enum.py` to prepare the enumerations for grants

   'python make_enum.py"

2. Use get to retrieve the grants from your VIVO to a spreadsheet

   `python sv.py -a get`

3. Edit your spreadsheet to improve your grant data, and add new grants
4. Use update to put your improved grant data back in VIVO

   `python sv.py -a update`

5. Repeat the steps each time you wish to improve your grant data

Intro to the Pump

The VIVO Pump (under development), provides a means for managing VIVO data using spreadsheets. Rows and columns in spreadsheets represent data in VIVO. The Pump update method takes spreadsheet data, and a definition file, and returns VIVO RDF (add and subtract) for updating VIVO with the values in the spreadsheet. The Pump get method uses the definition file and the data in VIVO to create a spreadsheet. Schematically:

```
update method:  Spreadsheet Data    =>    The Pump      =>    VIVO RDF   =>   VIVO
                                       + definition file

get method:     Spreadsheet Data    <=    The Pump      <=    VIVO RDF   <=   VIVO
                                       + definition file
```

The same definition file can be used for get and update, and the spreadsheet resulting from a get can be used in a subsequent update, providing a "round trip" capability. That is, you can "get" data, improve it manually or programmatically, and use it to "update" VIVO.

Work in Progress

The Pump is under active development, and beta testing at the University of Florida.

See Issues for work in progress, features to be added, and bugs to be addressed.

Testing the Pump

- Test Case Overview
- Simple VIVO Tests

The Pump API

The Pump API # Intro to the Pump

The VIVO Pump (under development), provides a means for managing VIVO data using spreadsheets. Rows and columns in spreadsheets represent data in VIVO. The Pump update method takes spreadsheet data, and a definition file, and returns VIVO RDF (add and subtract) for updating VIVO with the values in the spreadsheet. The Pump get methods uses the definition file and the data in VIVO to create a spreadsheet. Schematically:

```
Spreadsheet Data    =>    The Pump      =>    VIVO RDF (the update method)
                        + definition file

Spreadsheet Data    <=    The Pump      <=    VIVO RDF (the get method)
                        + definition file
```

The same definition file can be used for get and update, and the spreadsheet resulting from a get can be used in a subsequent update, providing a "round trip" capability. That is, you can get" data, improve it manually or programmatically, and use it to "update" VIVO.

Work in Progress

The Pump is under active development, and beta testing at the University of Florida.

See Issues for work in progress, features to be added, and bugs to be addressed.

Testing the Pump

- Test Case Overview
- Simple VIVO Tests

The Pump API

The Pump API

Managing Journals in Simple VIVO

Journals a re fundamental entity for VIVO. To add publications to VIVO using Simple VIVO, the journals for the publications must already be present in your VIVO. You will want to load your VIVO with journal titles and add journals as required for your publications.

Adding journals to your VIVO

Simple VIVO provides a set of more than 6,600 academic journals, each with an ISSN. Many also have an EISSN.

To add these journals to your VIVO, use:

```
python -v -a update -s journal_data.txt
```

The result will be a two files: `journal_add.rdf` and `journal_sub.rdf` Use the VIVO System Admin interface to add the RDF in `journal_add.rdf` to your VIVO. `journal_sb.rdf` should be empty. If not, use the VIVO System Admin interface to subtract the RDF in `journal_sub.rdf` from your VIVO.

Managing Journals in your VIVO

1. Use get to retrieve the journals from your VIVO to a spreadsheet

   ```
   python sv.py -a get
   ```

2. Edit your spreadsheet to improve your journal data, and add new grants. You may need to remove duplicates, correct spelling or otherwise improve the name of the journal, or add an EISSN if one is missing.

3. Use update to put your improved journal data back in VIVO

   ```
   python sv.py -a update
   ```

4. Repeat the steps each time you wish to improve your journal dataVIVO can store information about locations such as cities, buildings, and campuses. One Simple VIVO definition can be used to manage all your locations, or you may choose to use source files specific to different kinds of locations – one for cities, one for buildings, one for campuses.

Many types of things are locations. The most common are campuses, buildings, countries, and cities, but see the file `examples\locations\location_types.txt` for a complete list of VIVO location types.

Regardless of the type of location, each location has:

1. a name
2. one or more types such as building, campus, country or city
3. within – an indication that one location is within another. For example, a city is within a country, a building is within a campus.
4. latitude. Negative is south.
5. longitude. Negative is west.

We can use one Simple VIVO scenario to manage all of our locations.

1. See `locations.txt` for sample locations.

2. Using a spreadsheet, edit this file to remove locations that you do not wish to have in your VIVO.
3. Add locations of interest for your VIVO.
4. Run python sv.py -a update to put locations in your VIVO.
5. Use python sv.py -a get to get a spreadsheet of all the locations currently in your VIVO.

That's it.

Two additional examples are provided. Buildings can be managed using the methods described below. You may find that having buildings in their own spreadsheet makes working with them easier than having all your locations in a single spreadsheet.

A second example provides a data set of United States cities with populations of over 100,000. This data set can be loaded into your VIVO and used to provide references for events and other VIVO entities that reference locations.

Manage Campus Buildings using Simple VIVO

Buildings are locations. Each has a name, a URL (may point to an enterprise system, an existing university web page about the building, historical information, or any other information resource about the building), a latitude and longitude.

Note

A sample file of buildings at the University of Florida is provided in uf_buildings.txt

How to use

1. Use the sv_buildings.cfg config file to get your buildings from VIVO, as in: python sv.py -c sv_buildings.cfg -a get
2. Edit the resulting buildings.txt file to add your buildings
3. Update VIVO using python sv.py -c sv_buildings.cfg -a update
4. Add the resulting RDF to VIVO

Add US Cities to VIVO

Add cities in the United States with population over 100,000 in 2015 to VIVO. List came from Wikipedia,updated with cities over 100,000 in Puerto Rico.

Each city has a name, a state, latitude and longitude.

Look up the state in VIVO through an enum. If not found, throw an error – all the states need to be in VIVO. VIVO is missing District of Columbia and Puerto Rico. These need to be added by hand before the cities can be added.

Look up the city in VIVO. Update the state, lat and long as needed.

Note

There's nothing US specific about this code. The data file contains US cities. Edit the data file to add cities of interest in provinces or states of interest.

The def file assumes that the wgs ontology has been added into your VIVO. See World Geodetic System in Wikipedia and Basic Geo Vocabulary and the RDF file

How to use

1. Add District of Columbia to your VIVO as a State or Province. Add to state_enum.txt

2. Add Puerto Rico to your VIVO as a State of Province. Add to state_enum.txt

3. Add the cities to your VIVO using an sv update as shown below: python sv.py -c sv_cities.cfg -s us_cities.txt -a update

4. Add the resulting RDF to VIVO

5. To see your city data and manage it in the future, you can do an sv get python sv.py -c sv_cities.cfg -a get mentoring relationships between mentors and mentees. Any context.You will want to have organizations in your VIVO for a number of reasons:

6. Organizations are the units of your institution – your colleges, academic departments, administrative units, and laboratories. By representing your organization and its structure (which organizations are parts of others) you will be able to produce reports and visualizations based on your organizational structure, and answer questions such as number of faculty per organization, number of mentors, expertise at the organizational level, productivity and other other assessments.
7. Organizations are where your people had positions prior to being at your institution.
8. Organizations are where your people were educated.
9. Organizations provided grant funding to your organization.
10. Organizations where your faculty have collaborators that they work with on scholarship.
11. Organizations are employers of your graduates.

To put organizations in VIVO, you will want to have a spreadsheet of the organizations that you can use to manage the organizational data. As organizations come and go, add and remove them from your spreadsheet. As more information about organizations becomes available, or as organizational information changes (contact information, web pages, etc), make the updates in your spreadsheet. Use Simple VIVO to update VIVO using the values in your spreadsheet (see below).

Simple VIVO always allows three things to appear in a cell in a spreadsheet containing attributes for an organization:

1. The value of the attribute. See below. The cell contains a name, or a phone number, or a zip code.
2. The word None. When None appears are the value of an attribute, Simple VIVO is directed to remove whatever attribute value it finds in VIVO. For examples, see [[Simple VIVO|]]
3. An empty cell, that is, nothing is in the cell. When Simple VIVO sees nothing in the cell, it does nothing – it neither adds nor subtracts a value. If your spreadsheet is mostly empty, it tells VIVO to mostly do nothing.

Only filled cells are acted upon. Cells with values are checked against VIVO. If the value in the cell is different than the value in VIVO, the value in VIVO is replaced with the value in the cell. If the value in the cell is the same as the value in VIVO, nothing is done. Cells withe the word None act to erase values from VIVO. Whatever value is found in VIVO is removed, and no new value is provided.

examples/orgs

examples/orgs contains the files you need to manage organizations using Simple VIVO.

Your spreadsheet data will have the columns shown in the table below. If you need more or fewer columns, your VIVO site administrator should be able to help you modify the `org_def.json` file distributed with Simple VIVO to provide the organizational attributes you need.

Column name	Purpose
uri	The VIVO uri of the organization. VIVO assigns a uri to every organization in VIVO.
remove	Used to remove organizations from your VIVO.
name	The name of the organization precisely as it will appear in VIVO
home_page	the URL of the home page of the organization
email	The email address of the organization
phone	The phone number of the organization
within	The name of another organization which is the parent of this one
address1	Address line 1 for the organization

Column name	Purpose
address2	Address line 2 for the organization
city	The name of the city of the organization
state	The name of the state of the organization
zip	the sip code (postal code) of the organization
abbreviation	An abbreviation for the organization. “NIH” for the National Institutes of Health, for example
overview	A paragraph of text regarding the organization
type	The organization’s types. See below. Many organizations have multiple types.
successor	The name of the successor to the organization if it no longer exists

Creating `org_enum.txt`

Run `make_enum.py` using:

```
python make_enum.py
```

Updating org values in VIVO

1. Get your orgs from vivo using `python sv.py -a get`
2. Edit `orgs.txt` using a text editor or a spreadsheet. The file is delimited using the inter field delimiter, which defaults to the tab character.
3. Update your orgs using ‘`python sv.py -a update`
4. Add `org_add.rdf` to VIVO using the Site Admin Load RDF feature
5. Remove `org_sub.rdf` from VIVO using the Site Admin Remove RDF feature
6. Update the enumeration using `python make_enum.py`
7. Repeat these steps to make changes to your organizational data

Adding orgs to VIVO

Adding orgs is simple – just edit your spreadsheet to add a row. Put as much data about the organization as you have on the row in the appropriate columns. Leave the rest of the columns blank. In particular, leave the `uri` column blank. Simple VIVO will assign a uri to your organization when it is added.

Follow the rest of the steps described for updating yourorg values. Adds and updates can be combined in the same spreadsheet.

Removing orgs from VIVO

Removing orgs is also very simple – just edit your spreadsheet and put the word “remove” in the `action` column. Each organization that has `remove` in the action column will be removed from VIVO.

Maintaining subsets of orgs

You may find that you would like to have different kinds of organizations in different spreadsheets to simplify data management. For example, you may wish to have your internal organizations in one spreadsheet and external organizations in another.

To maintain subsets, follow the steps below:

1. Create appropriate folders for your work. You may want to manage the two collections from two different folders.
2. Copy the def file into each folder
3. Modify the def file to select the orgs of interest.
4. Copy the sv.cfg file into each folder
5. Modify the sv.cfg file to use the name of the definition file for your subset. Modify the rdfprefix and source file name as appropriate to help you manage the subset.
6. Run your gets and updates as above in each folder for each subset.

Future

We expect to be able to manage photos of organizations in the future. Check back here to see if Simple VIVO supports photos. Each person in VIVO has contact information and other identifying information as expected.

This Simple VIVO example supports 19 attributes for each person. Each one is optional.

1. display_name
2. orcid – enter the person's orcid id in the form nnnn-nnnn-nnnn-nnnn
3. types – enter the person's type(s) using the abbreviations in the `person_types.txt` enumeration. `fac` for faculty, `pd` for postdoc, etc.
4. research_areas – enter the person's research area(s) using the
5. overview – enter a plain text overview for the person. Do not cut and paste from Microsoft Word.
6. name_prefix – name prefix such as "Dr"
7. first_name – person's first name
8. middle_name – person's middle name
9. last_name – person's last name
10. name_suffix – person's name suffix such as "Jr" or "III"
11. title – person's title
12. phone – person's primary phone number
13. email – person's primary email
14. home_page – person's home page URL
15. street_address – mailing address street for the person
16. city – mailing address city
17. state – mailing address state or province
18. zip – mailing address postal code
19. country – mailing address country

As with everything in Simple VIVO, the attributes are optional. For any particular person, enter as many attribute values as you have. It is straightforward to update the person with additional attributes in the future.

Enumerations

1. `concept_enum.txt` – used to list possible subject areas for the advising relationship
2. `person_types.txt` – used to list possible person types with abbreviations for each

Adding and Updating people

1. Update the enumerations using `python make_enum.py`

2. Get the existing people from VIVO using `python sv.py -a get`
3. Edit the people as needed – correcting values as needed, adding values to people in the spreadsheet, and adding new people. To add new a new person, add a row to the spreadsheet, leave the uri column blank, and add as many attributes as you have for the person. All the attributes you add should be found in the appropriate enumerations (see above). If the value you specify is not found, it will not be added. To add new values to enumerations, first add them to VIVO, then update the enumerations, then add them to the person.
4. Update VIVO with your improved spreadsheet using `python -a update`

Repeat these steps as needed to add additional mentoring relationships to VIVO.Positions have five attributes:

1. Start date – from an enumeration of dates of the form yyyy-mm-dd
2. End Date – from an enumeration of dates of the form yyyy-mm-dd
3. Organization of Position – from an enumeration of organizations by name
4. Person in position – from an enumeration of people by orcid
5. Job title – open text

The steps to manage positions

1. Update your enumerations using `python make_enum.py`
2. Get positions from your VIVO using `python sv.py -a get`
3. Edit your spreadsheet `positions.txt` correcting and adding information for existing positions, and adding new positions. Use a text editor or spreadsheet program
4. Update the position information in your VIVO using `python sv.py -a update`

Repeat the steps as necessary. As we gain experience with Simple VIVO and the underlying Pump software, we'll collect ideas for future features here. There are no promises here, and some ideas that seemed worthwhile at the time, may fade in importance as experience grows.

Some of the issues are for the "end user," which for Simple VIVO is typically the VIVO data manager. Some are for programmers using the Pump. And some are for the developers of the software to allow the software to grow and improve in a reasonable way.

Features are numbered simply to make referring to them easier. The numbering does not imply an ordering.

Note: Some people may ask "why have future features in the wiki rather than in the issues? After all, you can mark issues as"future" and exclude them from various lists. And the answer is that I found that mixing the issues together was inconvenient and misleading. GitHub does not have the ability to save queries of the issues, so each time I wanted to see the current issues, I had to explicitly exclude the issues marked "later". Eventually I decided to gather the "later" features here and close them out of the issues list, reserving the issues list for features under active development.

Possible Future Features

1. Add an ontology diagram to each example to show what is going on.
2. Rename the "remove" column to "action" and provide remove as one of several actions. Support a merge action by providing a syntax to specify a mater and merge to the master. "a1" and "a" where "a" specifies a master and "a1" indicates "merge to a". Could ten use any string as a mater identifier and any string with a number as merge to the master identifier. Could specify many merges in one update.
3. Add SPARQL queries and visualizations to each example to reward data managers for getting their data into VIVO.

4. Add handler for photos. When a user specifies a photo filename as data, a handler should be available as part of the definition that can process the file, create a thumbnail, put the thumbnail and photo in appropriate places in the VIVO file system and generate the appropriate RDF.
5. Refactor two_step and three_step as recursive. Pump should support any path length through recursive application of path. Need to be clever about "one back" and one forward" and then it should be feasible. Will take a good lucid morning.
6. Simple VIVO should read and write from stdin and stout respectively. Support stdin and stdout as sources and sinks for source data. As in: `filter | filter | python sv.py > file`
7. Simple VIVO should put data directly into VIVO with the need to write a file of RDF to be loaded manually.
8. Add support for auto-generated labels. When adding awards and honors, the VIVO interface auto generates a label for the AwardReceipt. Without that label, the award receipt displays its uri in the interface. When adding degrees, the VIVO interface does the same thing – auto generates a label for the AwardedDegree. We could just add a label field to each def and require the data manager to supply the label. And a simple Excel function would auto generate labels in the spreadsheet.
9. Support round tripping of the output of serialize. The Pump serialize method was envisioned as a means of round tripping a text description of the state of the Pump for the purpose of restarting or recreating a Pump. To do this we will need:
    1. A complete description of the Pump state in serialize – all instance variable values, update_def
    2. A means to define Pump via the output of serialize, as in `q = Pump.define(p.serialize())` should `define` Pump q to be the same as Pump p.
10. Add an example regarding human subject studies. UF will allow its approved human subject studies to be posted in VIVO and associated with investigators. This is a great resource for understanding scholarship in clinical research. Will need to verify data model (draw a Vue figure, run it through ontology groups), then gather data, map, test.
11. Implement American Universities data management in `uf_examples`. American Universities is a web site at UF used to list "all" accredited universities in the United States offering bachelor's degrees and above. See http://clas.ufl.edu/au
    1. Create separate def for american universities. Web address, name, type
    2. Add `rdf:type` for american universities to UF VIVO
    3. Add python script for producing american universities web insert As a UF Example. It uses a UF ontology extension
12. Refactor the CSV object as a true class. Rewrite all CSV access. Class should support column order and empty datasets. Is this the biggest topic? Has handlers for pulling data from PubMed, CrossRef and eventually SHARE.All Pump arguments are keyword. Each has a default. They can appear in any order.

`queryuri` gives the URI of the SPARQL API interface to your VIVO. That is, the URI that will be used to make queries to your VIVO. This URI is new in VIVO 1.6. The Pump requires VIVO 1.6 or later. The sample value is for a VIVO Vagrant, a test instance.

`username` gives the username (VIVO may call this "email") for an account that is authorized on your VIVO to execute the VIVO SPARQL API.

`password` gives the password of the account that is authorized on your VIVO to execute the VIVO SPARQL API. That is, the password to the account specified by the `username`

`prefix` gives the SPARQL prefix for your VIVO. SPARQL queries typically use PREFIX statements to simplify the specification of URI in SPARQL statements. The value of the prefix parameter should be the collection of PREFIX statements used at your VIVO. Note that formatting of the parameter – each PREFIX statement begins on a new line, including the first one. Each is indented by four spaces. This is recommended.

`action` is the desired Pump action. There are four choices: update, get, summarize and serialize. Often these are specified on the command line to override the value in the config file and make explicit what action Simple VIVO is performing.

`verbose` is set to `true` to have the Pump produce output regarding the status and progress of its actions. The default is `false` which produces limited output showing the the datetime the action started, the number of entities effected, and the datetime the action was completed.

`inter` is the inter field separator character used for your spreadsheets. a CSV file would have `inter = ,`, a tab separated spreadsheet (recommended) would have `inter = tab`. Note the word "tab". the Simple VIVO config file uses this to specify a tab as a field separator. Another popular choice for field separator is the "pipe" character "|". Pipe and tab are used as separators because these characters do not appear in literal values in VIVO.

`intra` is the intra field separator character used to separate multiple values in a single spreadsheet cell. The default is a semicolumn.

`nofilters` set to `true` indicates that filters specified in the definition file should not be used. Set to `false` indicates filters are to be used as normal.

`defn` specifies the name of the file that contains the definition file, in JSON format, to be used by Simple VIVO.

`src` specifies the name of the input source file (for update) or output file (for get).

`uriprefix` indicates the format of any URI to be created by the update process. URI will have random digits following the prefix. So, for example, if your prefix is `http://vivo.school.edu/n`, Simple VIVO will create new entities with URI that look like `http://vivo.school.edu/nxxxxxxxx` where `xxxxxxxx` are random digits.The methods and instance variables of the Pump are used by programmers to define and perform the work of the pump.

Pump Methods

The Pump has four methods. None of the pump methods have parameters. Each is called simply by name. See below.

summarize

The summarize() method describes the operation that will be performed by the Pump as configured. It lists the values of the instance variables, the enumerations, the get_query, and the update_def. A single string with newlines is produced.

```
p = Pump()
print p.summarize()
```

serialize

The serialize methods produces a string version of the instance variables and update definition.

```
p = Pump()
print p.serialize()
```

Future: Should be suitable for round tripping back into the Pump as a state.

get

get() performs the Pump get operation, querying VIVO according to the definition, creating a spreadsheet data structure, and writing that structure to the output file specified by the Pump instance variables. get() returns the number of rows in the output spreadsheet.

“‘python p = Pump() nrows = p.get()

update

update() performs the Pump update operation, using source data provided in a spreadsheet and a definition file supplied as JSON. The update queries VIVO for current values, adds and removes entities as directed in the spreadsheet and adds, removes and updates attributes values as supplied in the spreadsheet. update() returns an add graph and a subtract graph, which can be serialized using rdflib. For example:

`python [add_graph, sub_graph] = p.update() add_file = open(args.rdfprefix + '_add.rdf', 'w') print >>add_file, add_graph.serialize(format='nt') add_file.close()`The VIVO Pump is deceptively simple. A spreadsheet is used to update elements in VIVO. What could go wrong? Here we provide an overview of tests cases of various kinds.

Entity Level Tests

1. No-op. Round trip the entities by doing a get and an an update from the resulting spreadsheet. Nothing should be added or subtracted. NOTE: If filters are used, the get will “improve” the data and the first round trip will make filter improvements to the data. Subsequent round trips will be operating on the improved data and should then be a no-op.
2. Add entity. Create new entities as per the definition.
3. Change entities. See below. Changes are at the element level.
4. Delete entities. This is done using the “remove” column in the spreadsheet.

Element level Tests

Individual elements in the spreadsheet are identified by row (entity) and column (path definition). Path definitions lead to objects in VIVO triples. Tests are conducted with and without qualifiers.

Unique	Length	ADC	Qual	Examples
Yes	1	Add	No	Building Abbreviation
Yes	1	Change	No	Building Abbreviation
Yes	1	Delete	No	Building Abbreviation
Yes	2	Add	No	Org Mailing Address Zip
Yes	2	Change	No	Org Mailing Address Zip
Yes	2	Delete	No	Org Mailing Address Zip
Yes	3	Add	No	Grant Start Date
Yes	3	Change	No	Grant Start Date
Yes	3	Delete	No	Grant Start Date
No	1	Add	No	Person type(s); Person Research Area(s)
No	1	Change	No	Person type(s); Person Research Area(s)
No	1	Delete	No	Person type(s); Person Research Area(s)
No	2	Add	No	Grant Investigators; Org URL
No	2	Change	No	Grant Investigators; Org URL

Unique	Length	ADC	Qual	Examples
No	2	Delete	No	Grant Investigators; Org URL
No	3	Add	No	Do we have an example?
No	3	Change	No	?
No	3	Delete	No	?

Functional and component tests

The definitions support various functions through components of the software. Examples include enumeration and filtering. Each must be tested.

1. Component tests. Each software component needs unit tests.
2. Filter test. Unit tests of filters. Round trip testing of filters.
3. Enumeration tests. Unit tests of enumerations. Round trip testing of enumerations.
4. Handler tests. Each handler (photo, TBA) will need unit and integration tests. Service to the profession. Can this be readily generalized?The Simple VIVO configuration (config) file tells Simple VIVO about your VIVO, what you would like to have done, and how you would like to do it. Simple VIVO command line parameters override the values in the config file. In this way, the config file can say what you usually do, and the command line can say what you would like done in this particular run.

For example, if you have a config file that tells Simple VIVO everything it needs to know to update the people in your VIVO, you can run Simple VIVO using

```
python sv.py
```

No parameters are needed. But if you would like to see all the internal messages that Simple VIVO generates while it is working, you can use:

```
python sv.py -v
```

You can get a complete list of all the command line parameters by using

```
python sv.py -h
```

By default, Simple VIVO expects your config file to be named `sv.cfg`. But you can call it whatever you would like and tell Simple VIVO the name of your config file on the command line.

For example, suppose you name your config file 'uf_production_sv.cfg', which might be a good name if you are at the University of Florid and your config file is describing the production environment. You would run Simple VIVO using:

```
python sv.py -c uf_production_sv.cfg
```

Config file parameters

A sample config file `sv.cfg` is shown below.

Sections

Sections have names of the form `[sectionname]` There are three section headers in the config file. They can appear in any order.

`[sparql]` specifies parameters to be used to access your VIVO.

`[pump]` specifies parameters to be used to perform Pump operations.

```
[sparql]
queryuri = http://localhost:8080/vivo/api/sparqlQuery
username = vivo_root@school.edu
password = v;bisons
prefix =
    PREFIX rdf:   <http://www.w3.org/1999/02/22-rdf-syntax-ns#>
    PREFIX rdfs:  <http://www.w3.org/2000/01/rdf-schema#>
    PREFIX xsd:   <http://www.w3.org/2001/XMLSchema#>
    PREFIX owl:   <http://www.w3.org/2002/07/owl#>
    PREFIX vitro: <http://vitro.mannlib.cornell.edu/ns/vitro/0.7#>
    PREFIX bibo: <http://purl.org/ontology/bibo/>
    PREFIX event: <http://purl.org/NET/c4dm/event.owl#>
    PREFIX foaf: <http://xmlns.com/foaf/0.1/>
    PREFIX obo: <http://purl.obolibrary.org/obo/>
    PREFIX skos: <http://www.w3.org/2004/02/skos/core#>
    PREFIX uf: <http://vivo.school.edu/ontology/uf-extension#>
    PREFIX vitrop: <http://vitro.mannlib.cornell.edu/ns/vitro/public#>
    PREFIX vivo: <http://vivoweb.org/ontology/core#>
[pump]
action = get
verbose = false
inter = tab
intra = ;
nofilters = false
defn = position_def.json
src = position_update_data.txt
uriprefix = http://vivo.school.edu/individual/n
```

Parameters

Parameters take the form `name = value`. All parameters are optional. Parameters must appear in the appropriate section. Within the section, the parameters may appear in any order. The parameters that may appear in each section are defined below.

Section	Possible parameters
sparql	queryuri, username, password, prefix
pump	action, verbose, inter, intra, nofilters, defn, src, uriprefix

Each is described below.

`queryuri` gives the URI of the SPARQL API interface to your VIVO. That is, the URI that will be used to make queries to your VIVO. This URI is new in VIVO 1.6. The Pump requires VIVO 1.6 or later. The sample value is for a VIVO Vagrant, a test instance.

`username` gives the username (VIVO may call this "email") for an account that is authorized on your VIVO to execute the VIVO SPARQL API.

`password` gives the password of the account that is authorized on your VIVO to execute the VIVO SPARQL API. That is, the password to the account specified by the `username`

`prefix` gives the SPARQL prefix for your VIVO. SPARQL queries typically use PREFIX statements to simplify the specification of URI in SPARQL statements. The value of the prefix parameter should be the collection of PREFIX statements used at your VIVO. Note that formatting of the parameter – each PREFIX statement begins on a new line, including the first one. Each is indented by four spaces. This is recommended.

Note: All parameters are optional. If a parameter is not specified, the Pump software will supply a default value. But the default values for the Pump software are set for a VIVO Vagrant, not your VIVO. You should consider the four parameters above to be required for your `sv.cfg`.

`action` is the desired Pump action. There are four choices: update, get, summarize and serialize. Often these are specified on the command line to override the value in the config file and make explicit what action Simple VIVO is performing.

`verbose` is set to `true` to have the Pump produce output regarding the status and progress of its actions. The default is `false` which produces limited output showing the the datetime the action started, the number of entities effected, and the datetime the action was completed.

`inter` is the inter field separator character used for your spreadsheets. a CSV file would have `inter = ,`, a tab separated spreadsheet (recommended) would have `inter = tab`. Note the word "tab". the Simple VIVO config file uses this to specify a tab as a field separator. Another popular choice for field separator is the "pipe" character "|". Pipe and tab are used as separators because these characters do not appear in literal values in VIVO.

`intra` is the intra field separator character used to separate multiple values in a single spreadsheet cell. The default is a semicolumn.

`nofilters` set to `true` indicates that filters specified in the definition file should not be used. Set to `false` indicates filters are to be used as normal.

`defn` specifies the name of the file that contains the definition file, in JSON format, to be used by Simple VIVO.

`src` specifies the name of the input source file (for update) or output file (for get).

`uriprefix` indicates the format of any URI to be created by the update process. URI will have random digits following the prefix. So, for example, if your prefix is `http://vivo.school.edu/n`, Simple VIVO will create new entities with URI that look like `http://vivo.school.edu/nxxxxxxxx` where `xxxxxxxx` are random digits.Simple VIVO is a command line interface to the Pump.

Command Line Tests

Function	Verbose	Def	Spreadsheet	Example
get	Yes	None	None	sv -v
get	Yes	None	Good	sv
get	Yes	None	Poor	sv
get	Yes	Good	None	sv
get	Yes	Good	Good	sv
get	Yes	Good	Poor	sv
get	Yes	Poor	None	sv
get	Yes	Poor	Good	sv
get	Yes	Poor	Poor	sv
get	No	None	None	sv

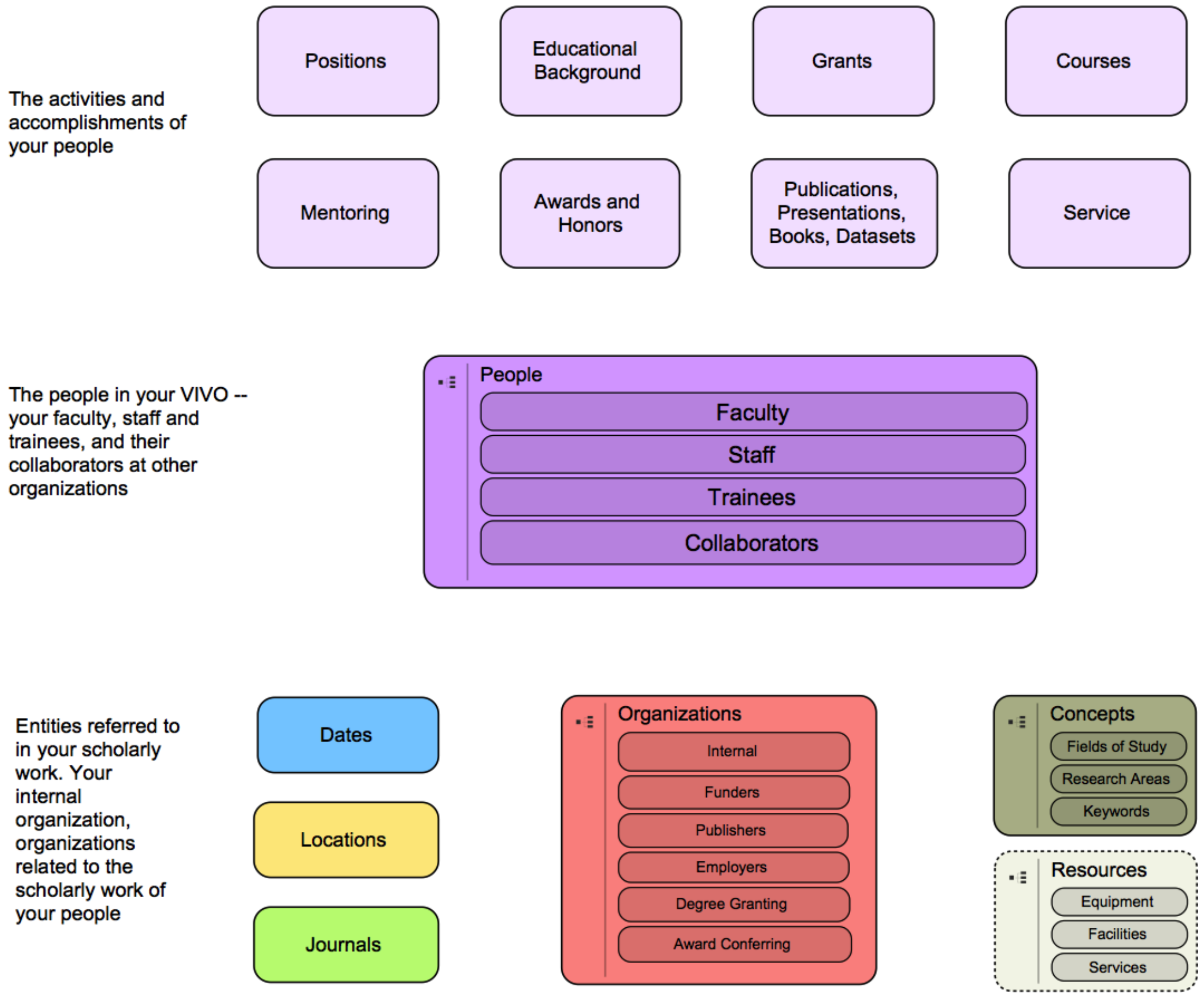
Function	Verbose	Def	Spreadsheet	Example
get	No	None	Good	sv
get	No	None	Poor	sv
get	No	Good	None	sv
get	No	Good	Good	sv
get	No	Good	Poor	sv get building_def.json bar
get	No	Poor	None	sv get foo
get	No	Poor	Good	sv get foo buildings.txt
get	No	Poor	Poor	sv get foo bar
update	Same	Same	Same	Same 18 cases
summarize	Same	Same	Same	Same 18 cases
serialize	Same	Same	Same	Same 18 cases*a command line tool for managing data in VIVO using spreads*

Could VIVO ever be "simple"?

VIVO provides an integrated view of the scholarly work of your organization. The scholarly work of your organization is complex – faculty, research staff, their activities and accomplishments. And these are connected to each other and to institutions, journals, dates, and concepts.

Let's think about VIVO as being about people. In the figure below, we see people in the center of the diagram. Things "below" the people are things that exist in the academic environment and can exist in your VIVO without reference to people. As we think about managing data in VIVO, and in building a VIVO, these things represent the place to start. We can put these things in VIVO and expect them to be there when we begin to put in people and journals.

The things at the top of the figure are the details of the scholarly record of people. They represent the things that will appear on a person's curriculum vitae.


The activities and accomplishments of your people
Positions
Educational Background
Grants
Courses
Mentoring
Awards and Honors
Publications, Presentations, Books, Datasets
Service
The people in your VIVO -- your faculty, staff and trainees, and their collaborators at other organizations
People
Faculty
Staff
Trainees
Collaborators
Entities referred to in your scholarly work. Your internal organization, organizations related to the scholarly work of your people
Dates
Locations
Journals
Organizations
Internal
Funders
Publishers
Employers
Degree Granting
Award Conferring
Concepts
Fields of Study
Research Areas
Keywords
Resources
Equipment
Facilities
Services

VIVO is great at representing all these things and more. But to keep it "simple," we will focus on the domains in the diagram.

The Basic Idea

What if we could manage the data in VIVO using spreadsheets?

That's the basic idea behind "Simple VIVO" a tool for using the VIVO Pump to take spreadsheet data and put into VIVO (called an update) and to get data from VIVO and put it in a spreadsheet (called a get).

Spreadsheets have rows and columns. The rows in the spreadsheet will correspond to "things" in VIVO. Depending on the scenario, your spreadsheet might contain people, publications, grants, courses or other kinds of things.

The columns in your spreadsheet will correspond to attributes of things in VIVO. So, for example, if you are working with people, your columns might contain attributes such as "name" and "phone number".

For each row in each column, your spreadsheet has a cell. Cells correspond to the values that will be in VIVO. In Simple VIVO you can have one of three different things in each cell:

1. Your cell contains a value. That value will be used to update VIVO.

2. Your cell contains the word “None.” None is a special word in Simple VIVO. It means that VIVO will be updated to remove whatever value is currently in VIVO. If you have None in a cell for the phone number of a person, then whatever phone number is currently in VIVO will be removed.
3. Your cell is empty or blank. Blank is also a special value for Simple VIVO. It means “do nothing.” Whatever value VIVO might have for the attribute is left unchanged.

Using these three things, you can manage data in your VIVO using a spreadsheet.

An Example

Suppose we have data in our VIVO on our faculty. Three of the faculty members are shown below:

URI	Name	Phone Number
http://my.school.edu/individual/n123321	Jones, Catherine	345-8999
http://my.school.edu/individual/n467823	Pinckey, William	(404) 345-8991
http://my.school.edu/individual/n858832	Hernandez, Juan	

In reviewing this data, we find we would like to make the following changes:

1. Catherine’s phone number should have an area code
2. William’s last name is misspelled and should not have a “c”
3. William has left the university and his phone number should be removed
4. Juan’s phone number should be provided.

We prepare an update set of data containing the following entries:

URI	Name	Phone Number
http://my.school.edu/individual/n123321		(404) 345-8999
http://my.school.edu/individual/n467823	Pinkey, William	None
http://my.school.edu/individual/n858832	Hernandez, Juan	(404) 345-8993

This update spreadsheet can be considered a set of instructions to Simple VIVO. We can read it say the following:

1. For the individual with URI http://my.school.edu/individual/n123321, leave the name unchanged (it is blank and blank means “do nothing”) and change the phone number to (404) 345-8999.
2. For the individual with URI http://my.school.edu/individual/n467823, change the name as shown and remove the phone number (None means remove whatever value you find)
3. For the individual with URI http://my.school.edu/individual/n858832, change the name as shown (Simple VIVO will notice that the name you gave is the name that is already in VIVO and so no change will actually be made. This is very handy. It means you can get data from VIVO and change only the items that need improving. All the other items can be left as they came from VIVO. The update will leave them untouched) and change the phone number to (404) 345-8993.

When the update is performed, the data in VIVO will look like:

URI	Name	Phone Number
http://my.school.edu/individual/n123321	Jones, Catherine	(404) 345-8999

URI	Name	Phone Number
http://my.school.edu/individual/n467823	Pinkey, William	
http://my.school.edu/individual/n858832	Hernandez, Juan	(404) 345-8993

Everything looks good. The names and phone numbers are all correct.

Data In

Getting data into VIVO is simple using Simple VIVO. You put the data you would like to add in a spreadsheet and save it as a "CSV File" – a comma separated value file. You can also use a tab separated file (TSV) or use any delimiter of your choosing (specified in the configuration parameters of Simple VIVO. For now, let's assume your configuration is using a comma separated file.

You tell Simple VIVO the name of your spreadsheet and the name of your definition file. That's it. Simple VIVO will add the data in your spreadsheet to the data in VIVO.

Let's say you wanted to add some new publications to your VIVO and your publications are stored in a spreadsheet called pubs.csv. Let's further assume that you have a definition file that defines your spreadsheet – many such definition files are provided with Simple VIVO – see [[Provided Data Scenarios|simple-vivo#provided-data-scenarios]] below. You would use:

```
python sv.py -a update -defn pubs_def.json -src pubs.csv
```

The data in `pubs.csv` will now be in VIVO. Simple.

Let's look at the items on the command line to see what each signifies:

`python sv.py` is the way we tell our system to run the Simple VIVO program. Your system administrator can confirm that this will work on your system.

`-a update` tells Simple VIVO that you want to update VIVO, using data from a source spreadsheet according to a definition.

`-defn pubs_def.json` tells Simple VIVO that you have a definition file called `pubs_def.json` that you would like to use to define the columns in your spreadsheet for the purpose of updating VIVO.

`-src pubs.csv` tells Simple VIVO that you have a data file called `pubs.csv` that will provide values to be added to VIVO.

Data Out

Getting data out of VIVO is simple using Simple VIVO. You specify the name of a definition file and the name of the file you want to store the data in. Simple VIVO uses the definition file to access your VIVO, retrieve the data, and return a spreadsheet.

To retrieve a list of the faculty at your institution, you would use:

```
python sv.py -a get -defn faculty_def.json -src faculty.csv
```

The result is a new file called `faculty.csv` that contains the data for the faculty as defined by `faculty_def.json`

Each of the items on the command line are described below:

`python sv.py` is the way in which a python program such as `sv.py` is executed on your system. Your system administrator can determine if this is how you will start Simple VIVO on your system.

`-a get` indicates to Simple VIVO that you want to get data from VIVO. `-a` stands for "action". So you are requesting the `get` action.

`-defn faculty_def.json` indicates to Simple VIVO that you will be using the `faculty_def.json` definition file. This file must be available on your system. If Simple VIVO can not find the definition file you specify, it will provide an error message to that effect.

`-src faculty.csv` indicates to Simple VIVO that the output spreadsheet will be called `faculty.csv`. Once Simple VIVO runs, you will have a new file called `faculty.csv`. You will be able to open the spreadsheet in Excel, Numbers, a text editor or other program.

Round Tripping

Getting data into VIVO and getting data out look very similar. The only difference is the action. The same definition file is used to get the data and to update the data. This insures that a spreadsheet that was produced by a "get" can be used by an "update".

It is common to get data from VIVO, improve it in some way, and then provide it back to VIVO. Suppose we were managing the organizations of our institution. We may have noticed that several of the institutions in our VIVO were missing URLs to their web pages, or missing phone numbers. We could use the steps below to improve the organizational data in our VIVO:

1. Get the organizational data from VIVO:

   ```
   python sv.py -a get -defn org_def.json -src org.csv
   ```

2. Open `orgs.csv` in Excel or your favorite spreadsheet editor and make changes as needed – provide missing phone numbers, URLs or making other changes such as correcting spelling in names or organizations.

3. Put the improved data back in VIVO:

   ```
   python sv.py -a update -defn org_def.json -src org.csv
   ```

These steps are very common – we get data from VIVO, improve it, and put the improved data back in VIVO.

Provided Data Scenarios

Follow one of the provided scenarios to manage data in your VIVO. The most common scenarios for managing data regarding the scholarship of your institution are provided.

- Dates
- Organizations
- Concepts
- People
- Positions
- Educational Background
- Awards and Honors
- Publications
- Journals
- Publishers

- Authors
- Grants
- Sponsors
- Teaching
- Mentoring
- Service

Adding Scenarios

Adding new scenarios requires a knowledge of the VIVO ontologies, as well as knowledge of the [[Pump Definition File|the-pump-definition-file]].
Studying the provided scenarios and the descriptions provided in this wiki will get you started. You will find support on-line on the VIVO email lists and in the VIVO Wiki. The Pump API is the collection of exposed elements that define how other software and users interact with the Pump.

The Pump API consists of:

1. The Pump arguments. All the things that are set by default and/or are available to the programmer and the user to set to define the actions of the Pump. All arguments are accessible via command line arguments of the main program `sv` (Simple VIVO) which is used to call the Pump.
2. The Pump methods and instance variables. Pump methods and instance variables are accessible to the programmer and so constitute part of the Pump API.
3. The Pump Definition file. Each Pump requires a definition file, in JSON format, to define the transformations the Pump will perform to and from the flat file representation to the semantic web representation.

Elements of the distribution that do not constitute parts of the Pump API, include **everything else**, including, but not limited to:

1. Data
2. Examples
3. The `vivopump` library, which consists of functions used by the Pump.
4. Test cases
5. Config files
6. Documentation files

Definition of the Pump API is critical for understanding Semantic Versioning used in the Pump.Every use of the Pump is defined by a pump definition file, which describes the correspondence between the data in the spreadsheet and the data in VIVO. Rows in the spreadsheet correspond to entities in VIVO. The rows in a spreadsheet might represent people, for example. Columns in the spreadsheet represent "paths" to attributes in VIVO. This will become clearer as we see examples of how paths are used to define the way in which the Pump will get or update values in VIVO. The cells in the spreadsheet represent attribute values in VIVO.

The pump definition file is always in JSON format.

Minimum definition

A minimum definition includes the required elements (marked with an asterisk below) in the required structure.

The definition file shown below is for a simple example maintaining cities (VIVO calls these PopulatedPlaces) each with two columns – a name and a US state. For example, Chicago is in Illinois.

The entity_def defines the rows of the spreadsheet. The column_defs define the columns of the spreadsheet. Each column def is a list of “paths.” Each path is a dictionary. Each dictionary has two named elements – “predicate” and “object”. The predicate element is also a dictionary, as is the object element. This is the required structure. In some cases, a definition will require “closures” to define additional relationships between columns.

The remainder of this document describes the elements, their purpose and whether they are required.

```
{
    "entity_def": {
        "entity_sparql": "?uri a vivo:PopulatedPlace .",
        "type": "vivo:PopulatedPlace"
    },
    "column_defs": {
        "name": [
            {
                "predicate": {
                    "single": true,
                    "ref": "rdfs:label"
                },
                "object": {
                    "literal": true
                }
            }
        ],
        "state": [
            {
                "predicate": {
                    "single": true,
                    "ref": "obo:BFO_0000050"
                },
                "object": {
                    "literal": false,
                    "enum": "states_enum.txt"
                }
            }
        ]
    }
}
```

entity_def*

The entity_def includes elements used to describe the rows of the spreadsheet.

entity_sparql*

Entity sparql will be used by the Pump to select the entities to represent the rows in your spreadsheet. You may have a simple entity_sparql such as:

```
?uri a foaf:Person .
```

The rows in your spreadsheet will be all people in your VIVO.

But you may need further restriction, as in:

```
?uri a vivo:FacultyMember . ?uri ufVivo:homeDept <http://vivo.ufl.edu/ndividual/n123123> .
```

where we are asking for all faculty members in the Chemistry department. Using a restricted `entity_def`, you can limit the number of rows in your spreadsheet, or partition the data management task into pieces to be done by different people.

`entity_sparql` always refers to the entities to be select as "`?uri`. `entity_sparql` must be valid SPARQL statements, each ending with a period as shown above.

type*

Indicates the rdf:type of each row of the spreadsheet. In an update, each new entity will have this type. The VIVO inferencer will supply "parent" types from the ontology. So, for example, indicating that the rows are type PopulatedPlace, the inferencer will supply the parents: Continuant (obo), Entity (obo), Geographic Location (vivo), Geographic Region (vivo), Geopolitical Entity (vivo), Immaterial Entity (obo), Independent Continuant (obo), Location (vivo), Populated Place (vivo), Spatial Region (obo), Thing.

order_by

Indicates the name of a column or columns that will be used to order the resulting spreadsheet. If no order by is given, the resulting data will be provided in URI order.

column_defs*

The column_defs specify the columns of the spreadsheet. There is one column name for each column in the spreadsheet.

Two column names are reserved words, used by the Pump:

Name	Purpose
uri	will contain the URI of the entity.
remove	The remove column is optional.

If present, it can be used to remove entities from VIVO during an update by putting the word "remove" on a row in the spreadsheet in the remove column.

For a get, the column_defs define the output spreadsheet. The resulting spreadsheet will contain one column for each column_def, plus a URI column. The columns in the spreadsheet will appear in the order of the column_defs in the definition file.

For an update, the the source spreadsheet may contain columns not defined in the definition file. These columns are ignored. The definition file may contain column_defs not found in the source file. These column_defs are ignored. RDF is generated only for columns found in *both* the column_defs and the columns of the source spreadsheet.

Column name*

Column names are specified in the definition file as shown in the example ("name" and "state"). Column names can be any length, upper and or lower case, with or without punctuation and spaces.

Whatever appears as a column name in the definition file will appear as a column name in the output spreadsheet produced by get.

In an update, the column name in the source must match the column name in the definition file *exactly*. Simple column names are preferred as a consequence.

The column_def is a list* Each column_def is a list, as indicated by the brackets

. These lists describe the "path" from the entity to the attribute represented by the column value. Paths may be of length 0, 1, 2 or 3. Lists of length zero are used to specify desired columns that will not be processed. For example:

```
notes: [
]
```

is valid and indicates that the spreadsheet has (update) or will have (get) a column called "notes". Nothing will be done with this column.

A length one list indicates that the entity has a predicate which has an object. Many VIVO attributes can be specified using length one paths. A common example is label. The label of an entity is specified using a triple of the form

```
<entity_uri>  rdfs:label  "label_literal_value"
```

See the cities example above. It specified that the first column of the spreasheet will be called "name". This column is then defined to be a length 1 path (the list has one element). The element is a dictionary as indicated by the brackets. The dictionary has two key values, "predicate" and "object". This is *required*. Every column definition is a list, every list element is a dictionary, every dictionary has two key values, "predicate" and object".

So for a length one path, this looks like:

```
"name": [
    {
        "predicate": {
        },
        "object": {
        }
    }
]
```

So for a length two path, this looks like:

```
"name": [
    {
        "predicate": {
        },
        "object": {
        }
    },
    {
        "predicate": {
        },
        "object": {
        }
    }
]
```

There are two dictionaries in the list. Each dictionary has exactly two elements. One element must have the key “predicate,” the other has the key “object.” The length two path definition indicates that the entity has a predicate that refers to an intermediary object, and that object in turn has a predicate and that the object of the second predicate is the attribute that appears in the spreadsheet. While this appears to be complex at first, it becomes clearer as examples are studied, and figures showing these relationships are examined.

Length three paths are similar. The length three column_def looks like:

```
"name": [
    {
        "predicate": {
        },
        "object": {
        }
    },
    {
        "predicate": {
        },
        "object": {
        }
    },
    {
        "predicate": {
        },
        "object": {
        }
    }
]
```

And the meaning is analogous to the length two path. The entity has a predicate the refers to an intermediate object that in turn has a predicate referring to a second intermediate object that in turns refers to an object that is the attribute that appears in the spreadsheet. VIVO has a number of three length paths attributes as you will see in the examples.

There are no four length paths in VIVO and length four paths are not supported by the Pump. Definition files with length four paths will be flagged as invalid.

predicate* The “predicate” key name is required in each element of the path. See above. By setting values of the predicate (see below) you can define the path from the entity to the value in the spreadsheet.

ref* Used to specify the uri of the predicate.

Example: You wish to manage people. You have a column in the spreadsheet called “name.” You want to associate values in that column to the `rdfs:label` of the person entity. To do this, you set the “ref” attribute of the “predicate” entry to “rdf:label” You may use all the prefixes that you are supplying to the Pump through its `prefix` argument.

single* Used to indicate whether the predicate is single valued (true) or multi-valued (false).

If you specify “true” for single, the Pump will warn you if it finds multi-valued values for the predicate. Your value will replace *all* the values in VIVO.

If you specify “false” for single, the Pump will expect you to supply all the values for the predicate. If you specify one value, *all* the values in VIVO will be replaced with the value you specify. If you specify 3 values and VIVO has two, the Pump will compare the values you specify with the values in VIVO and add and remove values as needed to insure that the values you specify are the values with that will be in VIVO.

Example: VIVO has values “a” and “b” for an attribute. You specify the attribute is “single”: false. You specify values “b”;“c”;“d” VIVO will remove “a”, leave “b” as is, and add “c” and “d” as values for the attribute.

A common case for multiple values is type of an entity. Many entities have multiple types. An organization may be a funding organization and a research organization, for example. When specifying “single”: false, you must specify all the values.

“boolean” is used to indicate that the column is a simple indicator of whether the predicate and the object value should be added or removed from VIVO. If ‘’ appears in the column, no action is taken. If ’0’ or ‘None’ or ‘n’ or ‘no’ or ‘false’ appears in a boolean column, the object value is removed from VIVO if present. If any other value is found in the column, the object value is added to VIVO if not currently in VIVO. Whenever “boolean” is specified, the “object” (see below) must have a “value” (see below). Failure to provide a value for an object with a boolean predicate is considered an invalid definition.

include Used to specify values that must be included by default in a multi-valued predicate.

include is optional. Use include when you want to be sure that the multi-valued predicate always includes the values you give in include. include is a list. You can specify as many values as you need in an include list.

Not: You do not need to include inferred values in an include list. The VIVO inferencer will supply additional types for entities.

object* The “object” key name is required in each element of the path. See above. The “object” key name

datatype Specifies the datatype of the literal object. xsd datatypes are supported:

datatype	Usage
xsd:integer	When literal values must be integers. Ranks, for example.
xsd:string	When literal values are strings.
xsd:datetime	For all VIVO dates and date times
xsd:boolean	Rare
xsd:decimal	For decimal numbers suh as dollar amounts
xsd:anyURI	For literal values that are URI, wuch as web page addresses

literal* True if the object is a literal value, False if the object is a reference to an entity.

name Used to indicate the name of an intermediate object in a multi-step path

enum Used to specify enumerations that are used to substitute one value for another. Enumerations are very handy in VIVO data management.

filter Filters are simple functions that take the value supplied and “improve” it, typically by correcting misspellings, formatting, expanding abbreviations, improving the use of upper and lower case, and other simple improvements.

The Pump provides filters which can be used to improve the data quality of VIVO.

Filter	Usage
improve_title	Used on publication and grant titles
improve_course_title	SPEC TOP IN AGRIC => Special Topics in Agriculture
improve_jobcode_description	AST PROF => Assistant Professor
improve_email	Returns RFC standard email addresses
improve_phone_number	Returns ITU standard phone numbers
improve_date	Returns yyyy-mm-dd
improve_dollar_amount	Strings which should contain dollar amounts (two decimal digits)
improve_sponsor_award_id	For NIH style sponsor award ids
improve_deptid	For department identifiers (eight digits, may start with zero)
improve_display_name	When displaying the whole name (not name parts) of a person

Example for publication titles:

```
"title": [
    {
        "predicate": {
            "single": true,
            "ref": "http://www.w3.org/2000/01/rdf-schema#label"
        },
        "object": {
            "filter": "improve_title",
            "literal": true
        }
    }
]
```

handler (future) A handler can be added to an object to indicate that special processing is required in addition to the normal operation of the pump. An example is photo processing. The name of the photo file is the object literal value, but this is not sufficient for a photo to appear in VIVO. The photo file must be copied to a specific location on the VIVO server. A photo handler can perform this action.

Note: Handlers are a future feature and are not available in the current version of the Pump.

type Intermediate objects in multi-length paths will be generated as needed by an update. These new intermediate objects will be given the type as specified. The VIVO inferencer will supply all parent types.

value value is used when the predicate processing is boolean. A value for the object is specified using the value parameter. if the column contains a '0' or 'None' or 'false' or 'n' or 'no', the value is removed from VIVO if present. If any other value is found in the column, the value is added to VIVO if not currently there. Blank column values are not processed.

Example: A new research area 'x85' is created and 50 faculty are identified that have the research area. A spreadsheet of faculty members and an x85 column is defined as boolean with the value 'http://definitinsource/x85'. The speadsheet contains a '1' in the x85 column for each of the faculty members with the x85 research area. All other data for the faculty members is unchanged. The pump will add hasResearArea assertions for each of the fifty faculty.

lang The RDF Lang attribute can be specified for the object. An example is shown below

```
"name": [
    {
        "predicate": {
            "single": true,
            "ref": "http://www.w3.org/2000/01/rdf-schema#label"
        },
        "object": {
            "literal": true,
            "lang": "en-US"
        }
    }
],
```

Object literal values will have the lang attribute in the RDF.

qualifier Use a qualifier to specify an additional condition which must be met to specify the path to the attribute.

label Use the label to specify a label to be added to the intermediate object constructed by the Pump during an update.

closure_defs

Closure definitions are identical in all ways to column_defs. They are processed after all column_defs regardless of where they appear in the definition file.

Closures are used to define additional paths required to specify the semantic relationships between between entities. In most cases, closures are not needed – paths from the row entity can be defined through the column_defs from row entity to the column value, whether literal or reference. But in some cases, there are relationships between entities referred to from the row entity.

For example, in teaching, the row entity, TeacherRole, has a clear path to the teacher and to the course. But there is also a relationship to be defined between the person and the course. One might argue that this relationship is unnecessary and that would be correct. The relationship could be inferred. But our semantic inference is not yet capable of making the inference between person and course, given the relationships between TeacherRole and person, and TeacherRole and course. A closure is to to make the assertion between Person and Course.

Just as with any normal column_def, the closure defines a path from the row entity to the column value. In the teaching example, we could define a closure from the TeachingRole to the Course through the Person, thereby defining the relationship between the Course and the Person. Or we could define the closure a length 2 path from TeacherRole to Person through the Course, thereby defining the relationship between the Person and the Course. The two paths produce the same result – the Person participates in the Course and the Course has a participant of the Person. The Closure defines this relationship.

The syntax of the closure_def is identical to the syntax of the column_def. Any number of closures can appear and in any order. All elements of the column_def are available in closure_defs and work the same way.

The closure_def is just a second chance to use the column data to define additional paths required to represent the relationships between the entities.

For the teaching example, the closure defines a second path from the TeachingRole (row entity) to the course (column entity) through the instructor. The `closure_def` is shown below:

```
"closure_defs": {
    "course": [
        {
            "predicate": {
                "single": true,
                "ref": "obo:RO_0000052"
            },
            "object": {
                "literal": false,
                "type": "foaf:Person",
                "name": "instructor"
            }
        },
        {
            "predicate": {
                "single": true,
                "ref": "obo:BFO_0000056"
            },
            "object": {
                "literal": false,
                "enum": "data/course_enum.txt"
            }
        }
    ]
},
```

Notes:

1. The course is "reused" by the closure to define a second path from the TeacherRole to the course. See the column_def for the first path.
2. The first step in the path is through the instructor. The object needs a name as an intermediate.
3. The instructor step in the path does not need a qualifier. It is coming from the row entity and there is only one entity that has the predicate relationship to the row entity. The University of Florida (UF) uses the Pump and Simple VIVO to manage its enterprise VIVO. Both the Pump and Simple VIVO were developed at UF to meet the need of upgrading VIVO from the 1.5 ontology to versions of VIVO supporting the 1.6 ontology (VIVO software version 1.6 and above). The University had tens of thousands of lines of code that were dependent on the 1.5 ontology – code that put data into VIVO and code that took data out of VIVO. These two basic functions became the update and get functions that we see in the Pump and Simple VIVO.

The University of Florida examples have several characteristics that distinguish them from the previous Simple VIVO examples:

1. The examples are large scale. UF is a large institution with over 900 institutional organizations, 6,000 faculty, and approximately 6,000 publications and 2,00 new grants per year.
2. The examples demonstrate the clean-up of enterprise data for display in VIVO. Like most institutions, the data of the University of Florida is optimized for internal transactional processing. It was never intended for integrated, external display of the scholarly work of the organization. Much effort has been put into preparing for integration and display of the scholarly work.
3. The examples use data source external to the organization. Examples show the use of PubMed and other external sources to augment the institution's data.
4. The examples make use of ontology extensions. UF finds it convenient to extend the ontology for several purposes – to better represent its work, and to provide local identifiers for entities in VIVO.

The UF domains

1. Orgs
   - Organization data management – management via spreadsheet. No institutional data
2. People
   - Ensure UF business rules
   - Manage UFCurrentEntity
   - Support excluding people from automated edited
3. Position Management
   - Support UF terminology and business rules
   - Exclude positions in protected departments
   - Improve position titles
   - Manage types
4. Papers
   - Ingest from PubMed IDs
   - Ingest from DOI
   - Ingest from CSV (bibTex to CSV)
5. Courses
   - Include common course number. Serve as the abstraction layer for course sections, much as awards have AwardReceipt and Award, and degrees have AwardedDegree and AcademicDegree, UF recognizes CourseSection (as taught) and Course (model).
6. Course Sections
   - Link to courses, instructors, terms
7. Sponsors
   - With UF sponsorid
8. Grants
   - Linked to departments, datetimes, cois, pis, invs The Pump can be used from the command line, using a delivered main program called Simple VIVO (`sv`)

```
python sv.py -defn my_definition_file.json -src my_spreadsheet_file.csv -a get
```

By default, Simple VIVO reads parameters from a configuration file. The configuration file tells Simple VIVO the address of your VIVO and the username and password needed to perform updates.

The Pump can also be used from a Python program by importing the pump, creating a pump object and calling a method on it. In the example below, a Pump object is created with a definition file and a source file. The `get()` method uses the definition file to build queries that are run in your VIVO and produce the source file.

```
import pump
p = Pump(defn='my_def.json', src='my_src.txt')
p.get()
```

In the example below, a Pump is created with a definition file name and a source file name. An update is performed. The update uses the definition to identify data in VIVO and compares it to data in the source file. Source file data values are used to update VIVO. Following the update, VIVO contains the values from the source file.

`python import pump p = Pump(defn='my_def.json', src='my_src.txt') p.update()`HomeVIVO Pump is a tool for managing VIVO data using spreadsheets.

Additional Resources

- VIVO GitHub
- VIVO Home Page
- VIVO Project Wiki
- Some VIVO Things

VIVO is a membership supported project of Duraspace