Changing the appearance of VIVO

Author: Jim Blake
Date: 04-May-2015 15:12
URL: https://wiki.duraspace.org/display/VIVO/Changing+the+appearance+of+VIVO

Table of Contents

1 Introduction

1.1 Making changes to VIVO

The VIVO application is a popular tool for research networking. Most VIVO sites put their own changes into VIVO, in order to create a distinctive appearance, or to satisfy their particular needs.

VIVO supports an assortment of tools and techniques for making these changes. Some changes can be accomplished while VIVO is running, simply by setting values on a form. Other changes require you to add or modify configuration files that control the application. Still other changes are accomplished by editing the VIVO code, re-building, and re-deploying the application.

1.1.1 What is included here?

This document describes the most common ways of modifying VIVO. The changes affect the appearance, layout, and content of the pages in the application.

1.1.2 What is not included here?

- Data operations
    - VIVO is only as good as the data it holds, and how the data is structured. The task of populating VIVO with data is very different on each site. Techniques for ingesting data into VIVO are covered in a separate document.
- The ontology
    - VIVO recognizes classes, instances, and properties base on the statements in its ontology. To a large extent, this determines how VIVO behaves. This document does not discuss changes to the ontology.
- The Java code
    - VIVO is open source software, so all changes are permitted. However, changes to the Java code are not discussed here. The sole exception is the custom editing forms, which do require some Java.
- The search index
    - The search index in VIVO can be configured to exclude certain classes of individuals. It can also be customized to include additional data fields. This document does not discuss how to customize the search index.
- The supporting technologies
    - VIVO relies on a triple-store and a search engine. It can be used with an external authentication system. Options for configuring or changing these technologies are discussed in the installation guide.

1.2 VIVO is already customized

Customization is built in to the heart of VIVO. VIVO itself is a customization of a more basic product called Vitro.

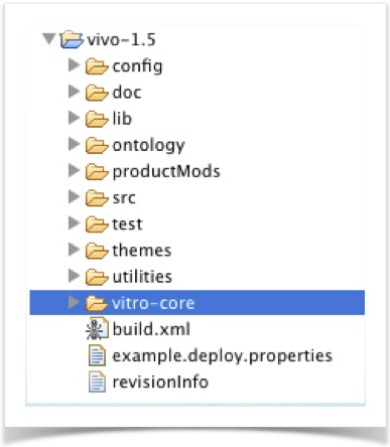
Here is how Vitro has been customized to become VIVO

Vitro	VIVO
No ontology	Includes an ontology for Research Networking
Minimal theme	Rich theme.
Default display rules	Annotations are used to: • Assign data properties to groups • Arrange property groups on the page
Default permissions	Display and editing permissions are customized, based on the ontology
Default editing forms	Editing is customized to the ontology
Default search index	Search index contains additional fields, specific to VIVO
Default functionality	Additional functionality: visualizations, interface to Harvester, QR codes, etc.
In total: A general-purpose tool for working with Semantic Data.	**In total:** A specialized tool for Research Networking

1.2.1 VIVO and Vitro

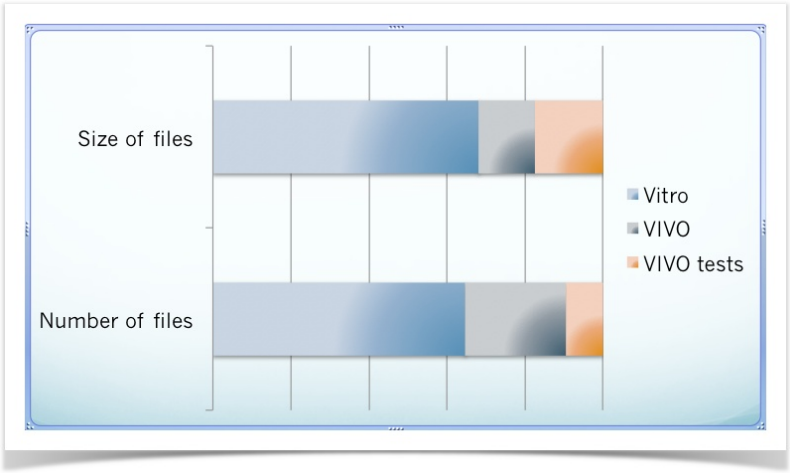
VIVO includes Vitro

When the VIVO distribution files are unpacked, the Vitro files are inside the main directory



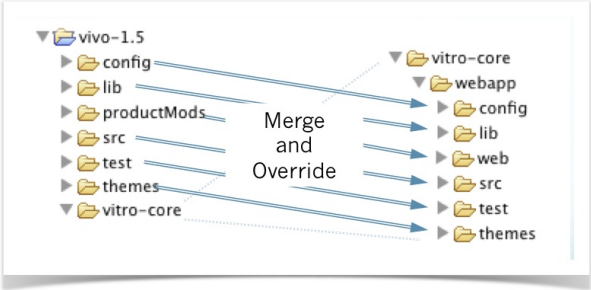
Most of VIVO is Vitro

This graphic from VIVO release 1.5 shows how much of VIVO is actually Vitro. Vitro makes up more than 60 percent of the VIVO distribution. If you were to remove the VIVO acceptance tests, about 80 percent of VIVO would actually be Vitro.

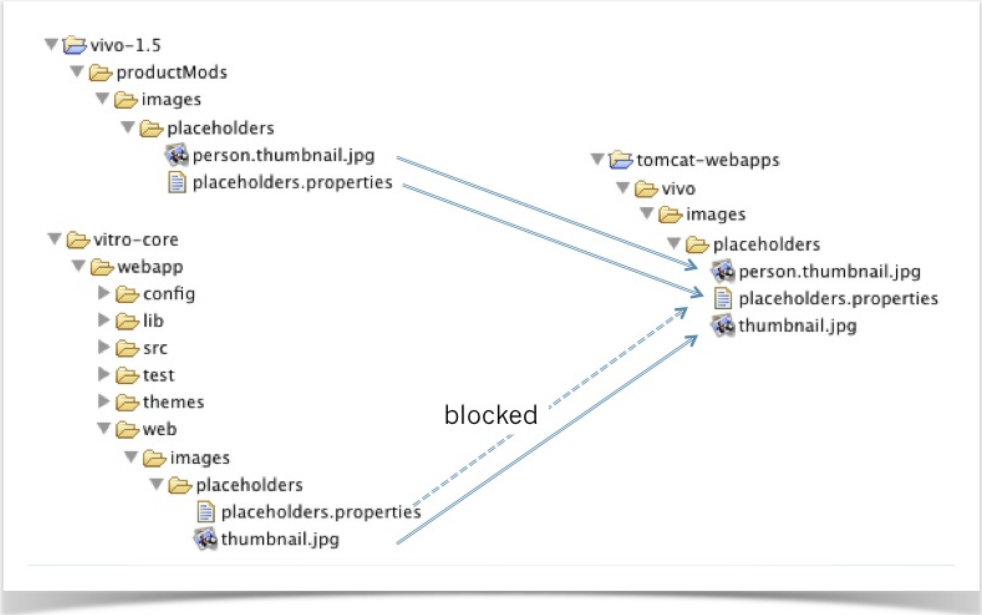


1.2.2 How VIVO is merged into Vitro

The build process in VIVO begins by overlaying the VIVO directory structure on the Vitro directory structure.



If a file in VIVO has the same name and directory path as a file in Vitro, the file in VIVO will replace (override) the file in Vitro, which is blocked. VIVO files that do not override Vitro files are added to the merged directories. The build process continues by compiling, testing, integrating and packaging the merged directories.



1.3 Adding your own customizations

How do you add your changes to VIVO? Perhaps more important, how do you keep your changes when you upgrade to a newer release of VIVO?

1.3.1 Working in the GUI

When you use forms in VIVO, the values you enter are kept in the triple-store. They will be retained when you upgrade to a new release. If the new release uses a different format to store the values, your changes will be migrated to the new format.

1.3.2 RDF files

Some customizations require that you add or modify an RDF file in your VIVO home directory. In general, it's best to create a new file to contain the RDF statements, so you can easily carry your changes to a new VIVO release.

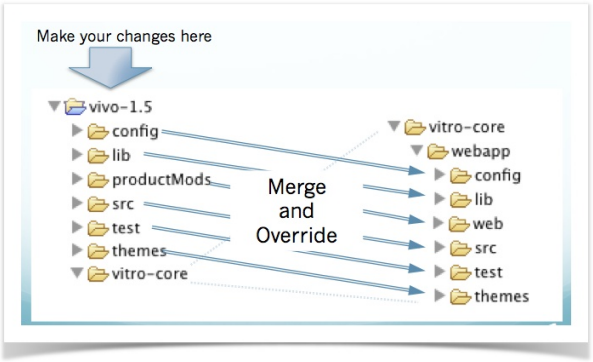
A "clean" build of VIVO will erase the RDF files in your VIVO home directory. You will need to re-create these files after the migration.

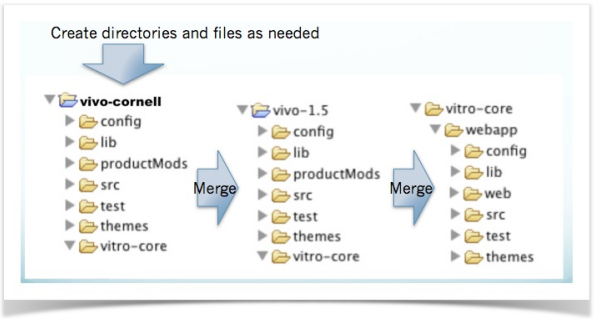
1.3.3 Changes to the source files

As with the RDF files, you should favor new files over changes to existing files. This will make it easier to carry your changes to a new release.

1.3.4 A third tier?

The discussion of VIVO and Vitro shows how the two code bases are combined during the build process. Some VIVO sites keep their local modifications in a third layer. This layer (or tier) is combined with VIVO and Vitro during the build.





The three-tier approach is a little harder to set up than the standard build, but it has the advantage of keeping all of your local modifications separate from the VIVO distribution. When the time comes to upgrade to a new release, there is no question about what files contain your local changes.

The two-tier build is fine, if your changes are limited to:

- Annotations on the ontology
- A custom theme
- Page management
- Language support

Three-tiers might be better if you will be using

- Custom list views
- Custom short views
- Custom entry forms
- Java changes

The VIVO Installation Instructions contain more details about how to set three tiers and there is a Git project template available.

1.4 Tool summary

1.4.1 Required skills

The customization tools require different levels of knowledge. Some are as simple as filling out a web form. Most require the ability to write HTML, with additions from the Freemarker template engine. Some require Java programming.

As the tools are described, these terms will be used to specify the skills needed:

	Knowlege required
Basic	Requires an understanding of VIVO concepts.
Web development	The usual technologies for writing web sites, including HTML, CSS, and JavaScript. Knowledge of the Freemarker template engine.
RDF	Modify or create RDF data files, using RDF/XML, Turtle, or N3 format.
SPARQL	Create queries against the triple-store, using SPARQL.
Java	Create or modify Java code.
OpenSocial	Create or modify OpenSocial gadgets, written in JavaScript.

1.4.2 The tools

	What does it do?	How?	Required skills
Creating a custom theme	Create your own "brand" for VIVO. • Change colors, logo, headings, footers, and more.	CSS files, JavaScript files, and templates for HTML.	Web development
Annotations on the ontology	Control how data is displayed. • Property groups, labels, display order, hidden properties, and more.	Interactive.	Basic
Home page customizations	Choose from home page options. • Add a geographic focus map.	Edit your home page template to include a selection of sub-templates.	Web development
Page management	Add new pages to VIVO. • Static pages, navigation pages, or dynamic reports.	Interactive.	Web development, optional SPARQL
Profiles for classes	Use one type of profile page for people and another for organizations.	Create page templates. Configure VIVO to associate them with classes.	Web development, RDF
Multiple profile types for foaf: Person	Provide a choice of formats for profile pages. • Each page owner selects the format for his own page.	Edit page templates. Perhaps connect to a Website image capture service.	Web development

	What does it do?	How?	Required skills
Enriching profile pages with SPARQL queries	Display additional data on a profile page.	Write a SPARQL query. Create a template to display the results. Configure VIVO to use it.	Web development, SPARQL, RDF
Enhancing page templates with SPARQL queries	Display additional data in any page template.	Write a SPARQL query. Modify a template to display the results. Configure VIVO to use it.	Web development, SPARQL, RDF
Custom list views	Change how certain properties are displayed • Change the layout for that property • Display additional data with each value.	Write a SPARQL query. Create a template to display the results. Configure VIVO to use it.	Web development, SPARQL, RDF
Custom short views	Change how search results are displayed • Display depends on the type of result (Person, Document, etc.). Also change display on index pages and browse pages.	Write a SPARQL query. Create a template to display the results. Configure VIVO to use it.	Web development, SPARQL, RDF
Custom entry forms	Create data entry forms • Add or edit complex data structures.	Write a generator class in Java. Create a template for the editing form.	Web development, SPARQL, RDF, Java
Using Open Social Gadgets	Create optional content for profile pages. • Each page owner configures the gadgets for his own page.	Create gadgets from JavaScript, or install existing gadgets.	Web development, OpenSocial

	What does it do?	How?	Required skills
Language support	Languages other than English • Use VIVO in Spanish • Allow viewers to choose their preferred language. • Implement other languages.	Create files of phrases in the desired language, or install existing files.	Basic

2 How VIVO creates a page

Some examples of how VIVO assembles web pages from Freemarker templates.

2.1 The home page

Like the title page of a book, it is not unusual for the home page of a web site to be different from all other pages. In the default VIVO theme, the most significant difference is that the search box is moved from the header to a more prominent location on the page.

The following templates are used in the home page.

```
pageSetup.ftl
page-home.ftl
    head.ftl
        stylesheets.ftl
        headScripts.ftl
    identity.ftl
        languageSelector.ftl
    menu.ftl
        developer.ftl
    footer.ftl
        scripts.ftl
        googleAnalytics.ftl
```

Template	Purpose	From
`pageSetup.ftl`	Sets some class and formatting parameters.	Included in every page, by `TemplateProcessingHelper.java`
`page-home.ftl`	The special template used for the home page.	Specified as the page template by `HomePageController.java`, overriding the default `page.ftl`.
`head.ftl`	Creates the HTML <HEAD> tag.	Included by `page-home.ftl`.
`stylesheets.ftl`	Inserts links to CSS stylesheets.	Included by `head.ftl`.
`headScripts.ftl`	Inserts links to JavaScript files that must appear in the <HEAD> section of the page. These are somewhat unusual, since most JavaScript links appear at the end of the page.	Included by `head.ftl`.
`identity.ftl`	Draws the heading of the heading of the page, including the VIVO logo and the `Index` and `Log in` links.	Included by `page-home.ftl`.
`languageSelector.ftl`	Allows the user to select their preferred language. If the site supports only one language, this template has no effect.	Included by `identity.ftl`.
`menu.ftl`	Displays the page links (`Home`, `People`, etc.) at the top of the page .	Included by `page-home.ftl`.

Template	Purpose	From
`developer.ftl`	Displays the developer panel, used when testing and monitoring VIVO operation. If developer mode has not been enabled, this templates produces nothing.	Included by `menu.ftl`.
`footer.ftl`	Draws the footer of the page, including the copyright notice, and the `About` and `Support` links.	Included by `page-home.ftl`.
`scripts.ftl`	Inserts links to JavaScript files. Compare to `headScripts.ftl`.	Included by `footer.ftl`.
`googleAnalytics.ftl`	Inserts JavaScript code to work with Google Analytics. By default, this is commented out, since each site will need to insert their own ID values in order to produce meaningful results.	Included by `footer.ftl`.

2.2 A profile page

By numbers, the vast majority of pages on a VIVO site are profile pages. These are all likely to be structured around the properties of each individual. However, the format can be very different depending on whether that individual is a person, an organization, or a research grant.

The following templates are used in this particular profile page. As explained in the notes, the choice of templates is driven in part by the content of the page.

```
pageSetup.ftl
page.ftl
    head.ftl
        stylesheets.ftl
        headScripts.ftl
    identity.ftl
        languageSelector.ftl
    search.ftl
    menu.ftl
        developer.ftl
    individual--foaf-person.ftl
        individual-setup.ftl
        individual-orcidInterface.ftl
        individual-contactInfo.ftl
        individual-webpage.ftl
            propStatement-webpage.ftl
        individual-visualizationFoafPerson.ftl
        individual-adminPanel.ftl
        individual-positions.ftl
            propStatement-personInPosition.ftl
        individual-overview.ftl
        individual-researchAreas.ftl
        individual-geographicFocus.ftl
    individual-openSocial.ftl
    individual-property-group-tabs.ftl
        individual-properties.ftl
            propStatement-hasRole.ftl
        individual-properties.ftl
            propStatement-dataDefault.ftl
            propStatement-hasInvestigatorRole.ftl
            propStatement-hasInvestigatorRole.ftl
        individual-properties.ftl
            propStatement-fullName.ftl
    footer.ftl
        scripts.ftl
        googleAnalytics.ftl
```

Template	Purpose	From
`pageSetup.ftl`	*as above.*	
`page.ftl`	The master template for most VIVO pages.	Specified by `FreemarkerH`

`head.ftl`
`    stylesheets.ftl`
`    headScripts.ftl`
`identity.ftl`
`    languageSelector.ftl`
	as above.	

Template	Purpose	From
`search.ftl`	Draws the search box in the header of the page.	Included by pa
`menu.ftl` `developer.ftl`	*as above.*	
`individual--foaf-person.ftl`	The main body of the profile page .	VIVO is config body of a profi **You can char** Class-specific This is specifie `initialTBo` recognized by `Individual` `Individual`
`individual-setup.ftl`	Sets some basic values for the following templates to use.	Included by `ir` `foaf-perso`
`individual-orcidInterface.ftl`	Implements the VIVO integration to ORCiD. If this integration is not enabled, this template has no effect.	Included by `ir` `foaf-perso`
`individual-contactInfo.ftl`	Displays the person's phone numbers and email addresses.	Included by `ir` `foaf-perso`
`individual-webpage.ftl`	Displays the person's preferred web pages.	Included by `ir` `foaf-perso`
`propStatement-webpage.ftl`	Displays a link to one of the person's preferred web pages.	VIVO is config displaying pre **change this c** View Configur This is specifie `listViewCon` specified in `Pr` `vivoListVie`
`individual-visualizationFoafPerson.ftl`	Displays the visualization links for co-authors, co-investigators	Included by `ir` `foaf-perso`

Template	Purpose	From
`individual-adminPanel.ftl`	Displays links for a VIVO administrator to use when editing this person's information	Included by `ir` `foaf-persor`
`individual-positions.ftl`	Displays the positions that this person currently holds.	Included by `ir` `foaf-persor`
`propStatement-personInPosition.ftl`	Displays one position that this person currently holds.	VIVO is config displaying pos **configuration** Configurations This is specifie `listViewCor` , which is spec

`individual-overview.ftl`
`individual-researchAreas.ftl`
`individual-geographicFocus.ftl`
	Display additional information about the person.	Included by `ir` `foaf-persor`

`individual-openSocial.ftl`	Implements the VIVO integration to OpenSocial gadgets. If this integration is not enabled, this template has no effect.	**You can conf OpenSocial g** Using OpenSc Included by `ir` `foaf-persor`
`individual-property-group-tabs.ftl`	Displays the groups of properties for this person.	Included by `ir` `foaf-persor`

`individual-properties.ftl`
`propStatement-hasRole.ftl`
`individual-properties.ftl`
`propStatement-dataDefault.ftl`
`propStatement-hasInvestigatorRole.ftl`
`propStatement-hasInvestigatorRole.ftl`
`individual-properties.ftl`
`propStatement-fullName.ftl`
	Each invokation of `individual-properties.ftl` displays a property group.
	
	Each subordinate template displays one property for this person.
		Each referenc `individual-` by `individual-`
		
		VIVO is config templates whe roles, and nan **configuration** Configurations

Template	Purpose	From

`footer.ftl`
`scripts.ftl`
`googleAnalytics.ftl`
	as above.	

2.3 The People page

The page management GUI provides an easy way for VIVO administrators to create simple pages. These pages may also be added to the menu bar. By default, VIVO is configured with eleven such pages. Five of them are listed in the menu.

The following templates are used in the People page, and in other pages that allow users to browse through a class group.

```
pageSetup.ftl
page.ftl
    head.ftl
        stylesheets.ftl
        headScripts.ftl
    identity.ftl
        languageSelector.ftl
    search.ftl
    menu.ftl
        developer.ftl
    page-classgroup.ftl
        menupage-checkForData.ftl
        menupage-browse.ftl
        menupage-scripts.ftl
    footer.ftl
        scripts.ftl
        googleAnalytics.ftl
```

Template	Purpose	From

`pageSetup.ftl`
`page.ftl`
`head.ftl`
`stylesheets.ftl`
`headScripts.ftl`
`identity.ftl`
`languageSelector.ftl`
`search.ftl`
`menu.ftl`
`developer.ftl`
	as above.	

`page-classgroup.ftl`	Combines the components to create an AJAX-driven page that browses among the classes in a class group.	VIVO is configured to use this template in the `People` menu page page. **You can change this configuration: see** Page management**.** This template is invoked by `ClassGroupPageData.java`, which is assigned to the `People` page in `menu.n3`.
`menupage-checkForData.ftl`	Checks to see if the page will be empty. Displays messages suitable to a VIVO administrator or to another user, depending on who is viewing the page.	Included by `page-classgroup.ftl`.

Template	Purpose	From
`menupage-browse.ftl`	Creates the page context that will be filled by AJAX calls.	Included by `page-classgroup.ftl`.
`menupage-scripts.ftl`	Creates or links to the JavaScripts used in browsing among classes of individuals.	Included by `page-classgroup.ftl`.

`footer.ftl`
`scripts.ftl`
`googleAnalytics.ftl`
	as above.	

2.4 A back-end page

VIVO provides several pages that allow administrators to edit the classes and properties in the ontology, and to create or adjust class groups and property groups. These pages are built around the older JSP (Java Server Pages) technology, although the header and footer are created from the same Freemarker templates as other pages.

The following templates and JSPs are used in creating this page.

```
basicPage.jsp
    head.ftl
        stylesheets.ftl
        headScripts.ftl
    identity.ftl
        languageSelector.ftl
    search.ftl
    menu.ftl
        developer.ftl
    formBasic.jsp
        classgroup_retry.jsp
    footer.ftl
        scripts.ftl
        googleAnalytics.ftl
```

Template	Purpose	From
`basicPage.jsp`	The master template for the VIVO back-end pages.	Specified by `ClassgroupRetryController.java`.
`head.ftl stylesheets.ftl headScripts.ftlidentity.ftl languageSelector.ftlsearch.ftl menu.ftl developer.ftl`	*as above.*	
`formBasic.jsp`	A generic frame that provides title and buttons for an edit.	Specified by `ClassgroupRetryController.java`.
`classgroup_retry.jsp`	Shows the fields that may be edited for a class group.	Specified by `ClassgroupRetryController.java`.
`footer.ftl scripts.ftl googleAnalytics.ftl`	*as above.*	

3 Customization Tools

A description of each tool and how to use it.

3.1 Creating a custom theme

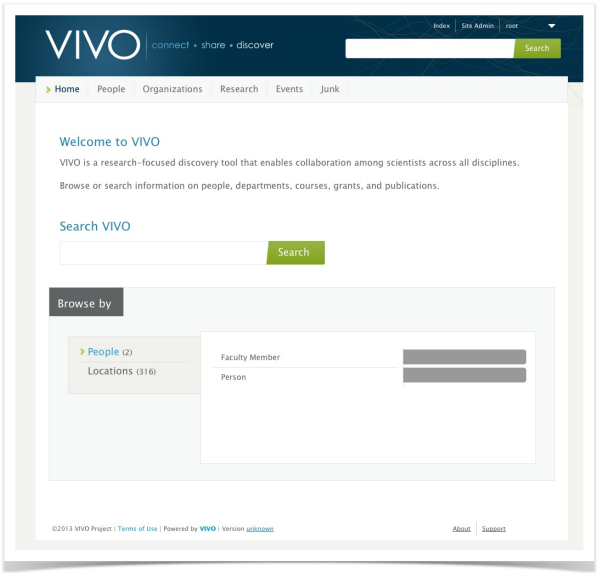
Create your own "brand" for VIVO. Change colors, logo, headings, footers, and more.

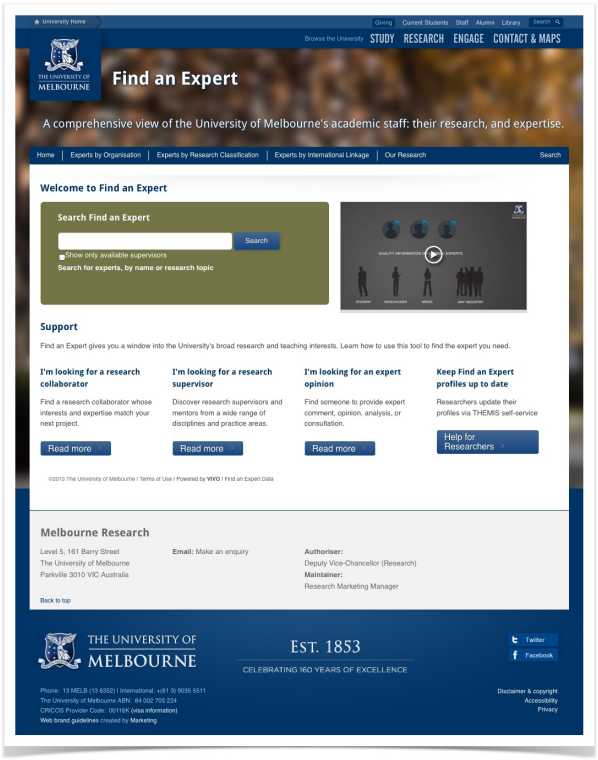
3.1.1 Overview

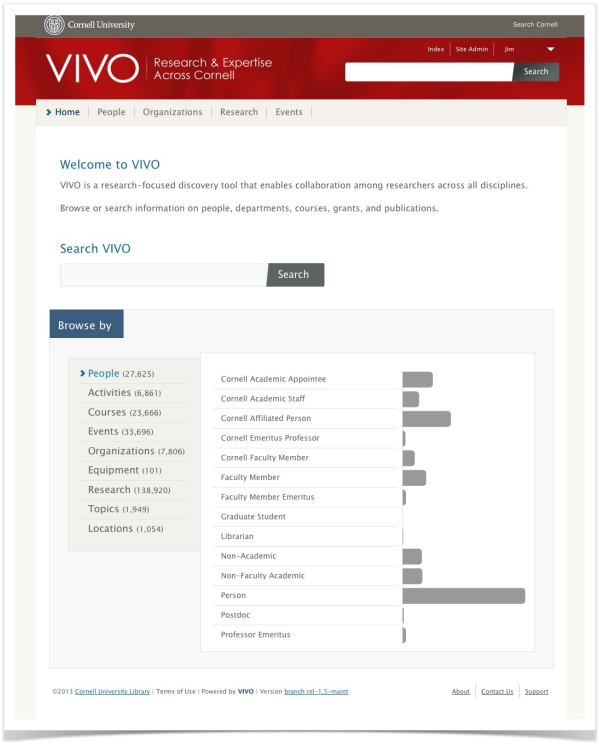
What can it do for you?

Change the "look and feel" of your VIVO installation. Change the styling, the images, the layout, the text, and more. Modify the header and footer on all pages.

Before and After







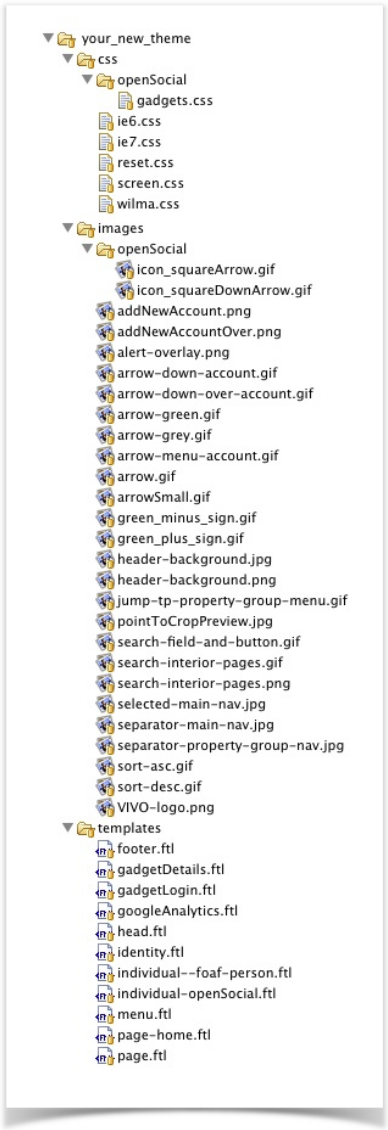
What do you need to know?

- Standard web-site technologies: HTML, CSS and maybe JavaScript.
- Something about the Freemarker template engine.
- Where the theme files are stored in VIVO, and how to reference them.

Getting started

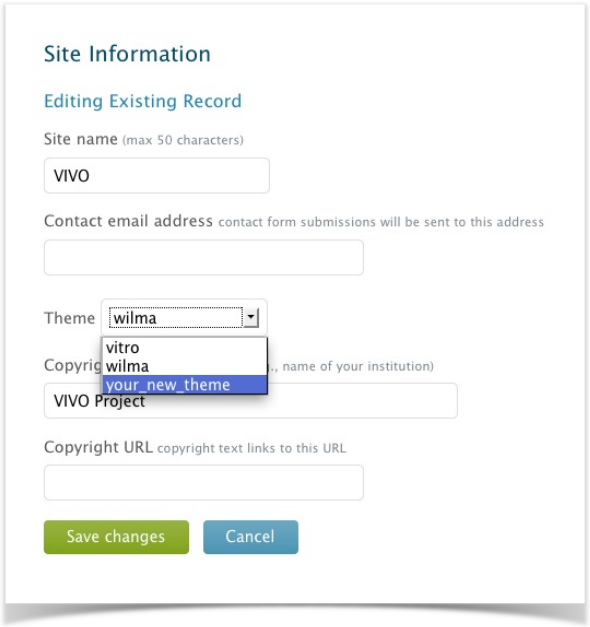
VIVO comes with a standard theme, called Wilma. Copy the theme into a new directory in the VIVO source tree, and give it a name.

Your new theme will contain CSS files, image files, and Freemarker templates.



This image is from VIVO release 1.5.1 – the exact contents of your theme may be different.

Run the Ant build to deploy your new theme to the Tomcat container. Restart the VIVO Tomcat process. You can then go to the ***Site Admin*** page and choose ***Site Information***, to select your theme as the current one.



3.1.2 The structure of pages in VIVO

The pages in VIVO are built around three different frameworks. Each of these uses the same header and footer, to provide consistency. In addition to including the header and footer, the pages frequently include smaller templates to provide detail.

These are the basic frameworks:

The home page	As the point of entry for VIVO, the home page is special. It is based on the Freemarker template `page-home.ftl`
All other public pages	Based on the Freemarker template `page.ftl`
"back-end" pages	Pages used for editing the ontology, or manipulating the raw data of VIVO are based on a JSP named `basicPage.jsp`

3.1.3 Some significant templates

page.ftl

`page.ftl` is the default base template. The rest of the theme templates listed are components of `page.ftl` (included either directly or indirectly). Closer inspection of `page.ftl` reveals a stripped down file that declares minimal markup itself and instead reads as a list of includes for the component templates.

> There are still page in VIVO which are not constructed entirely in FreeMarker. Breaking down the base template into components allows consistent markup to be used for any page rendered by the application, no matter how the page is constructed.

On the VIVO home page, `page-home.ftl` is used instead of `page.ftl`. It serves much the same purpose, but allows you to create a different layout for your home page than for the other pages in VIVO.

Once the transition is complete in a future VIVO release and all pages are rendered entirely using FreeMarker templates, the restrictions on `page.ftl` will be lifted and the preference on whether the base template should be broken down into smaller component templates will be left to the theme developer. Until that time, it is critical that the following components be maintained:

head.ftl

This component template is responsible for everything within the <head> element. Note that the open and closing tags for the <head> element are defined in `page.ftl` and wrap the include for `head.ftl`. There are several includes within `head.ftl` that should be carried over to any new theme to maintain expected functionality:

- `<#include "stylesheets.ftl">` - ensures that the necessary stylesheets called by templates downstream will be added to the page via <link> elements
- `<#include "headscripts.ftl">` - ensures that the scripts called by templates which must be in the <head> will be added to the page via <script> elements

identity.ftl

This component template is responsible for rendering the VIVO logo, secondary navigation and search input field at the top of the page. There are no mandatory includes from `identity.ftl` that need to be carried over but there are 2 template variables that are of particular interest (`${user}` and `${urls}`).

menu.ftl

This component template is responsible for rendering the primary navigational menu for the site. In wilma, it also happens to declare the open tag for the main content container. There are no mandatory includes from `menu.ftl`. The `${menu}` template variable is crucial since it contains an array of menu items needed to build the primary navigational menu.

footer.ftl

This component template is responsible for rendering the copyright notice, revision information, secondary navigation, and link for the contact form. There is a single include that should be maintained:

- `<#include "scripts.ftl">` - ensures that the non head scripts (those that don't need to be placed in the <head>) called by the templates will be added to the page via <script> elements

Several template variables of interest include `${copyright}`, `${user}`, and `${version}`.

googleAnalytics.ftl

This component template is primarily intended for the 7 partner institutions on the NIH grant, but it is available for anyone who is interested in using Google Analytics to track visits to a VIVO installation. It is included by `footer.ftl`. Simply uncomment the <script> element and provide your Google Analytics Tracking Code.

Adjust the markup as necessary in `page.ftl` and these component templates to achieve the desired content structure, and modify the stylesheets to meet layout needs and style your site. Remember that changes should be made in the source directory and that you will need to redeploy the project before the changes are reflected in the live website.

For more information about VIVO and web analytics, see VIVO Web Analytics.

You can find more information about the structure of the VIVO theme in How VIVO creates a page.

3.1.4 Making changes

Modify files in the theme

You can edit the Freemarker templates and the CSS files in the theme with any text editor. You can replace the image files with images that you choose.

Add files to the theme

Add CSS, JavaScript, or image files

As you modify the templates, you may want to use additional images, CSS files, or JavaScript files. When your templates refer to these files, they will use the Freemarker variable `urls.theme`, as shown in these examples:

```
<!-- an image file -->
<img src="${urls.theme}/images/arrow-green.gif"/>
<!-- a CSS file -->
<link rel="stylesheet" href="${urls.theme}/css/screen.css" />
<!-- a JavaScript file (create a js directory in your theme) -->
<script type="text/javascript" src="${urls.theme}/js/my.js"></script>
```

Add Freemarker templates

If your modifications use new Freemarker templates, you can refer to them more simply. Freemarker already knows where your theme directory is located.

```
<#include "my-new-template.ftl">
```

Override files that are not in the theme directory

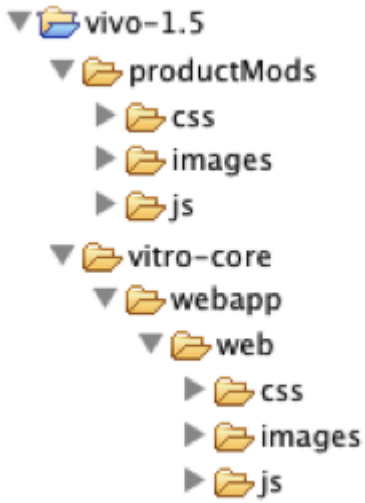
In order to keep the theme directory uncluttered, VIVO keeps most of the front-end files in a separate location. Changes to the theme usually involve the files in the theme directory, but you can override other files as well.

Override CSS, JavaScript or image files that are not in the theme directory

You may notice that templates refer to files that are not in the theme directory. They use references based on the Freemarker variable `urls.base` instead of `urls.theme`, like this:

```
<!-- an image file -->
<img src="${urls.base}/images/arrowIcon.gif"/>
<!-- a CSS file -->
<link rel="stylesheet" href="${urls.base}/css/login.css" />
<!-- a JavaScript file -->
<script type="text/javascript" src="${urls.base}/js/browserUtils.js"></script>
```

These refer to files in the `vitro-core/webapp/web` directory, which you can override in the `vivo/productMods` directory. If you look, you will see that `vivo/productMods` already contains some files that override those in `vitro/webapp/web`. This is because VIVO itself is a customization of Vitro.



Override Freemarker templates that are not in the theme directory

Like adding templates, overriding templates is simplified. You can override a file in `vitro-core/webapp/web/templates/freemarker`, or one of its sub-directories, by creating a file with the same name and path under `vivo/productMods/templates/freemarker`.

But VIVO treats all available Freemarker templates as belonging to the same flat namespace, whether they are in the theme directory or in the `templates/freemarker` directory, or one of its sub-directories. So a file named `vitro-core/webapp/web/templates/freemarker/page/partials/footer.ftl` can be overridden by a file called `footer.ftl` in the theme directory.

And it is.

Working on the theme

When you make changes to VIVO, you should make the changes in your VIVO distribution directory, run the build script, restart Tomcat, and test the changes. If you are doing full customizing of VIVO, this cycle might be best. If you are only working on the theme, you can speed things up.

- Tell the build script to skip the unit tests: they don't test the theme.
    - `ant deploy -Dskiptests=true`
- Don't restart Tomcat.
    - VIVO always serves the most recent version of CSS files, image files, and JavaScript files. You don't need to restart Tomcat to make that happen.
    - However, your browser may cache these files so you won't see the most recent version. Here are some suggestions for bypassing your browser cache.
- Tell VIVO to reload Freemarker templates each time they are requested.
    - By default, VIVO will wait one minute before checking for a change in a Freemarker template.
    - In VIVO 1.5.2 or before, add this to deploy.properties: `Environment.build=development`
    - In VIVO 1.6, add this to build.properties: `developer.defeatFreemarkerCache=true`
    - In VIVO 1.6.1 or later, use The Developer Panel to defeat the Freemarker cache.
    - This is not a good idea for your production VIVO instance - only for developing your theme.

Some developers prefer to make theme changes inside the `tomcat/webapp/vivo` directory. This eliminates the need to run the build script, but opens the threat of having the changes over-written the next time the build script runs.

When to restart Tomcat

If you make changes to any of the `runtime.properties` or to any of the RDF files in your VIVO home directory, you must restart Tomcat in order to see the effect of the changes.

If you make changes to any of the Java code, you must run the build script and restart Tomcat.

If you make changes to any of the source files in the theme, including images, CSS, JavaScript or Freemarker templates, you must run the build script, but you do not need to restart Tomcat.

3.2 Annotations on the ontology

Use the interactive editing screens to change the appearance of properties, classes, property groups and class groups.

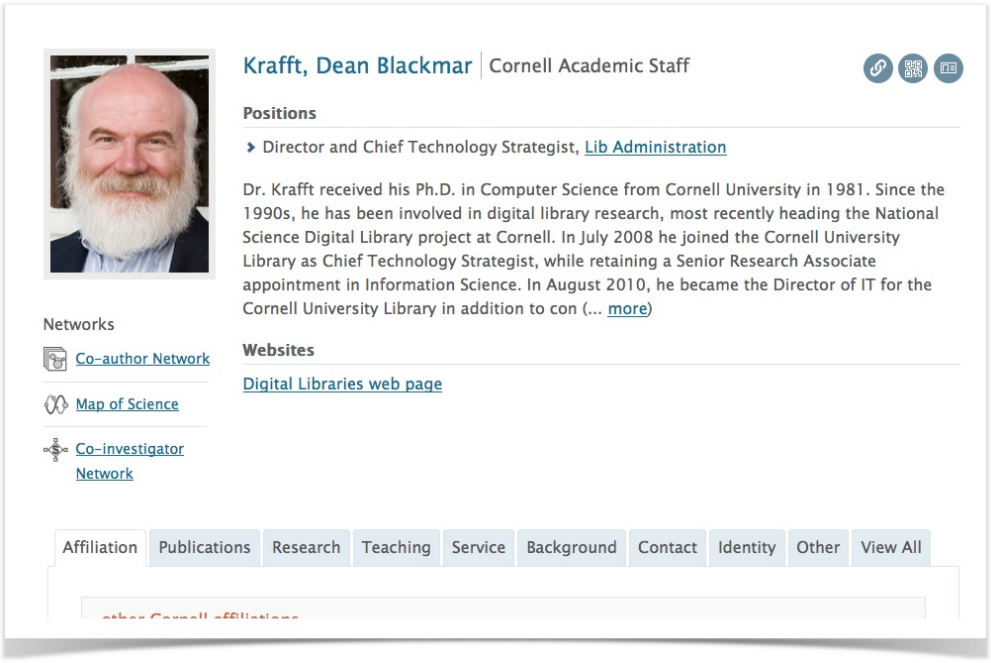
3.2.1 Edit property groups

Divide a long list of properties into groups. Name the groups and assign a display order.

Overview

Before and After

Rename "Affiliation" to "Allegiances", and change the order of "Publications" and "Research".



What do you need to know?

How to follow the GUI for property groups.

Getting started

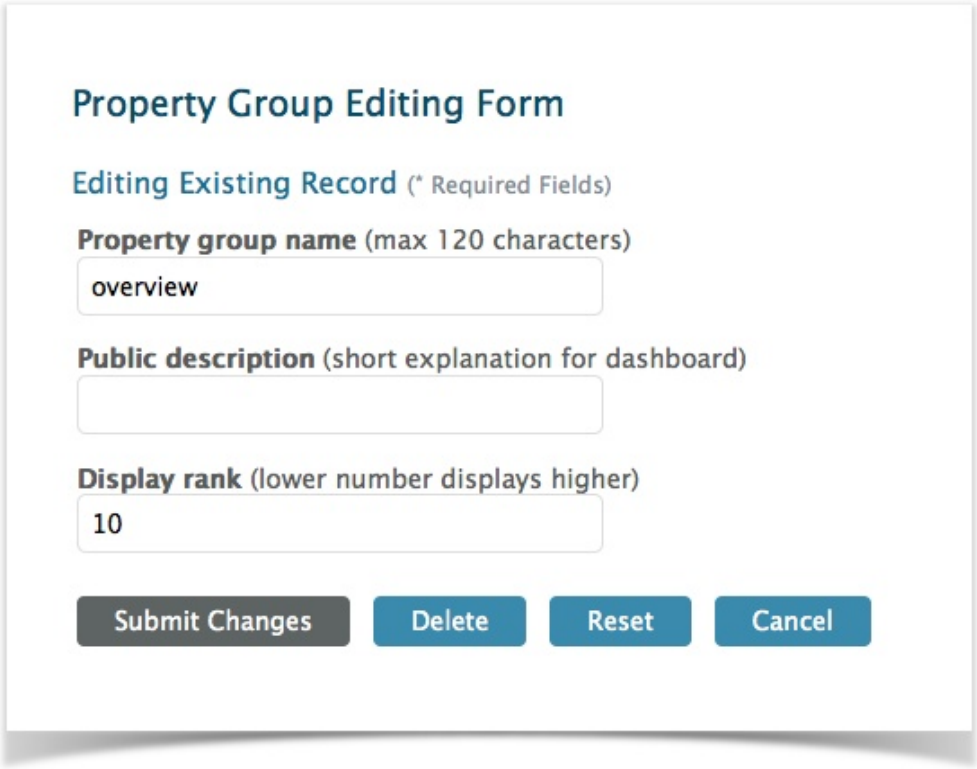
VIVO comes with a default set of property groups. You can rename them, reorder them, create new groups or delete existing ones. You can also move properties from one group to another, but that is covered in Edit the appearance of properties.

What to do

From the VIVO ***Site Admin*** page, navigate to the ***Property Groups*** page.

You can add a new property group, or click on the name of an existing group to edit it.

You can change the name of a group, change its display rank, or even delete it.



When an profile page is displayed, the property groups are shown in order of ascending display rank.

Properties that aren't included in any of the groups are displayed on the profile page as part of the group named `Other`. This is not an actual property group; it is simply a display convention.

If you delete a property group, the properties that were in it will be displayed in `Other`, until you assign them to new groups.

3.2.2 Edit the appearance of properties

Change how VIVO displays properties on a profile page.

Overview

What can it do for you?

Change how VIVO displays the properties of an individual.

For each property, you can change

- the display label
- the public and private descriptions
- which property group it belongs to
- the display rank within the property group
- who can see the values

- who can edit the values
- whether the values will be published in linked open data requests
- whether the display will be collated by sub-properties (object properties only)

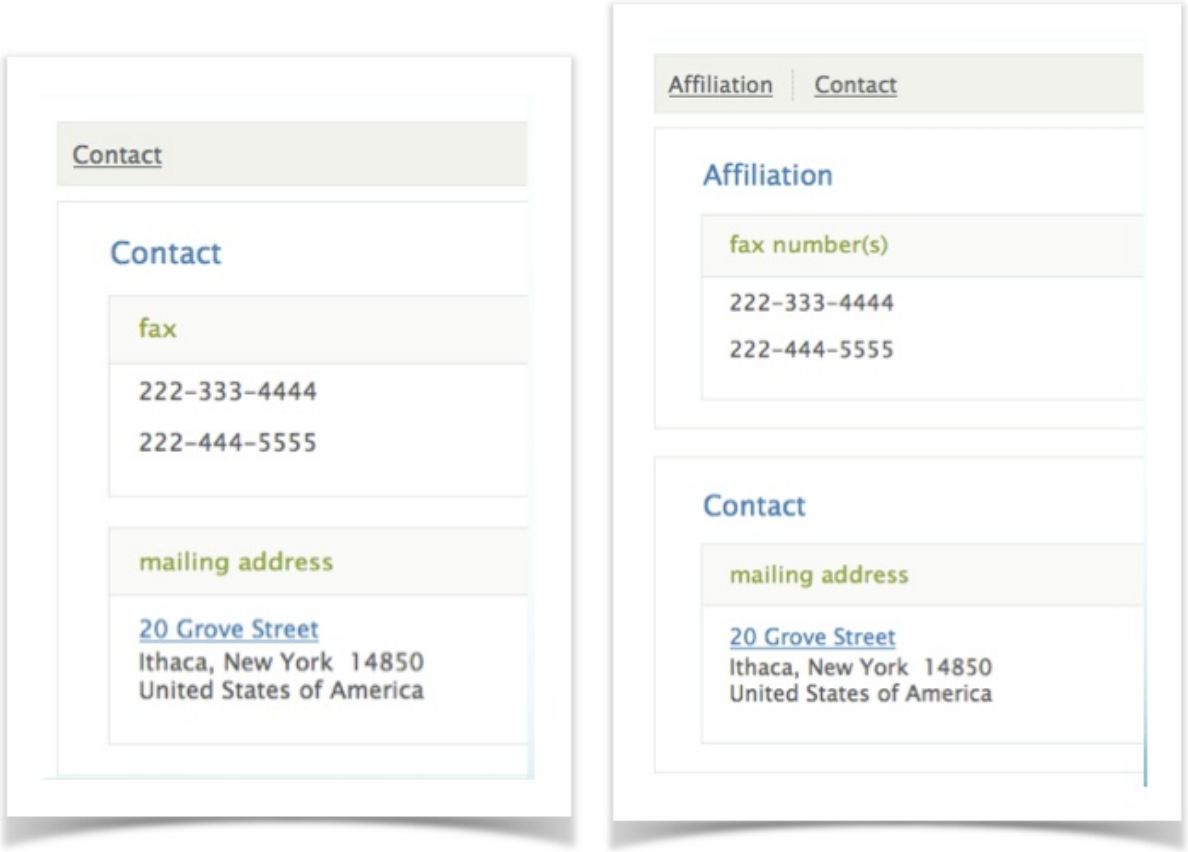
You can also change things like the namespace and parent property, but these are actually changes to the ontology.

Notice that there are necessary differences between the editing options for a data property and those for an object property. Since an object property describes the relationship between two individuals, it is richer than a data property which has only a text value.

The property editing form also allows you to assign a custom entry form to a property, as described in Customize: date entry forms

Before and After

Modify the "fax" property, changing the display label to "fax number(s)", and moving it to the "Affiliation" property group.



What do you need to know?

How to follow the GUI for property editing.—

Getting started

VIVO comes with a default set of properties. You can edit them to suit your display requirements, or make more extensive modifications, by customizing the ontology.

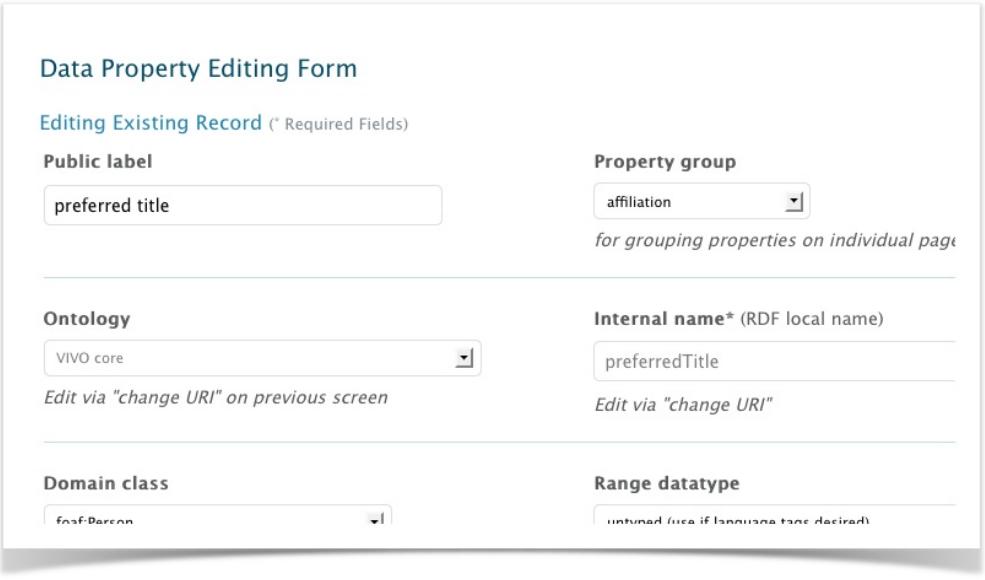
What to do

There are several ways to navigate to the ***Property Editing Form*** for a particular property. Perhaps the most common way is to show the profile page for an individual, turn on ***verbose property display***,

click on the name of the property you want to edit,

from the ***Property Control Panel***, choose to edit the property

When the property editing form appears, make the changes you want to see, and click on `Submit Changes`. Navigate to a profile page and you will see the effect of your changes immediately.



Notes

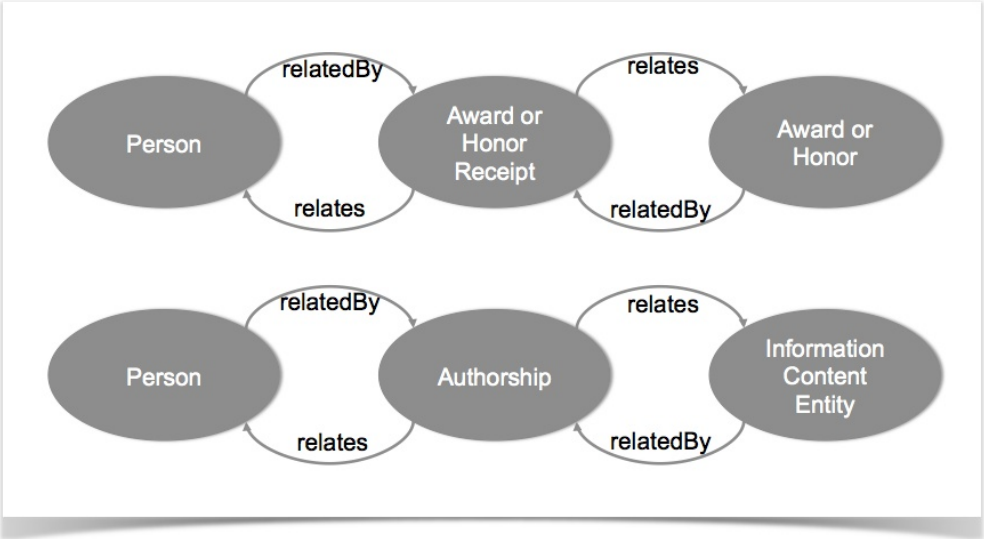
Properties may appear differently depending to someone who is authorized to edit them. Try logging out to see how your changes will appear to the general user.

3.2.3 Create and edit faux properties

When a property in an ontology is too broadly used, creating "faux" properties to distinguish among uses.

Overview

Since VIVO 1.6, the emphasis in the ontology has been of fewer properties connecting more classes. For example, we see that the relationship between a Person, an Authorship, and an Article is very similar to the relationship between a Person, an Award Receipt and an Award.



The profile pages in VIVO have been organized by properties. This re-use of properties makes it difficult to organize information on the page. Not only do we want to see at a glance the difference between an authorship and a received award; we may also want to display them in different areas of the page, using different custom views, etc.

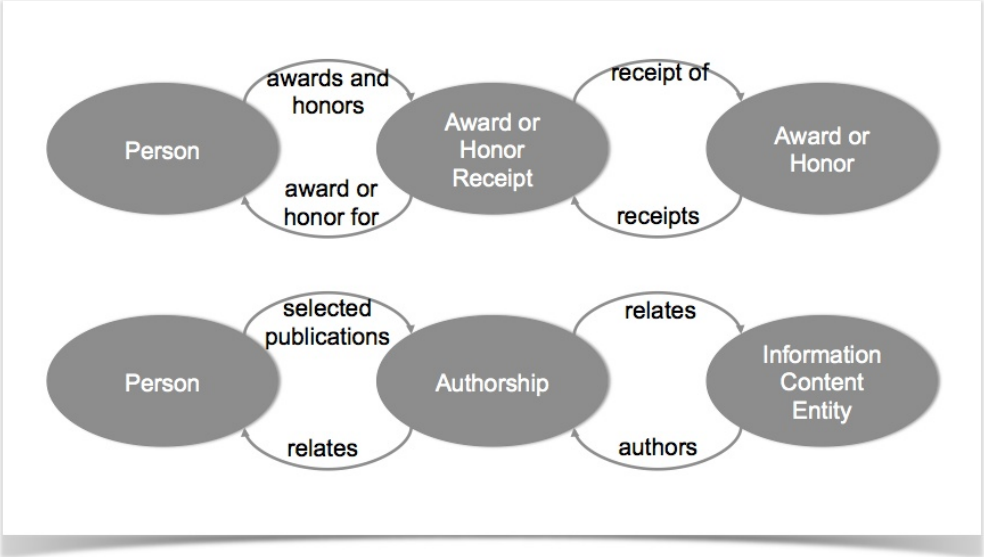
VIVO allows us to create "faux" properties, as restrictions on object properties. The faux property has the same property URI as its base property. It has a domain and a range that are restrictions of the domain and range of the base property. Once we have established these criteria, we can assign display properties to the faux property, just as if it were its own object property with its own URI.

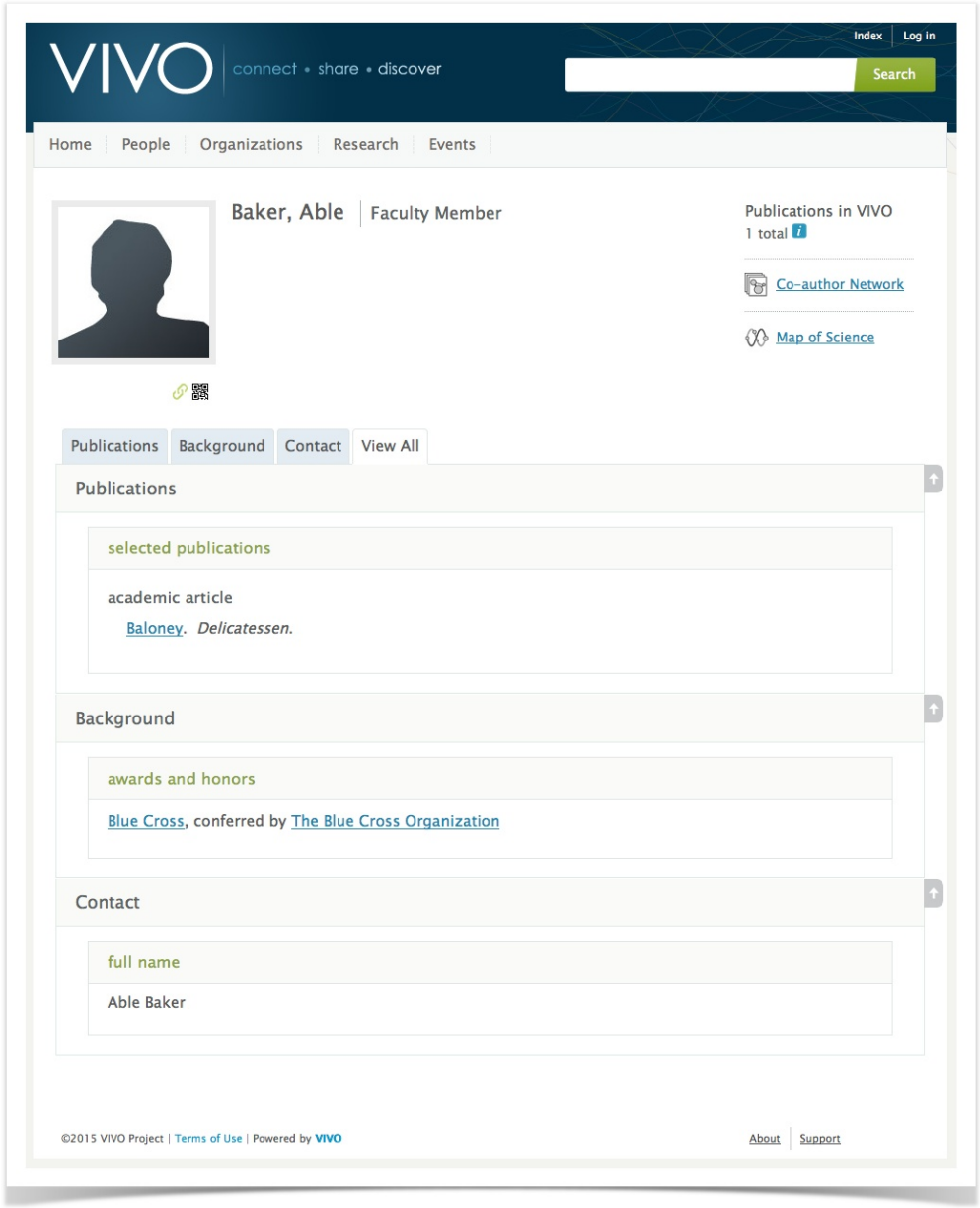
In this way, we can define faux properties as follows:

URI	Domain	Range	label	property group
relatedBy	Person	Award or Honor Receipt	awards and honors	Background
relatedBy	Person	Authorship	selected publications	Publications

URI	Domain	Range	label	property group
relatedBy	Award or Honor	Award or Honor Receipt	receipts	Overview
relatedBy	Information Content Entity	Authorship	authors	Overview
relates	Award or Honor Receipt	Person	award or honor for	Overview
relates	Award or Honor	Award or Honor Receipt	receipt of	Overview

Now, these same relationships display quite differently:





In general, it makes sense to partition all of a property into faux properties. Then the base property is set to be invisible (except to the root user), and the faux properties display the desired information.

What do you need to know?

How to follow the GUI for faux properties.

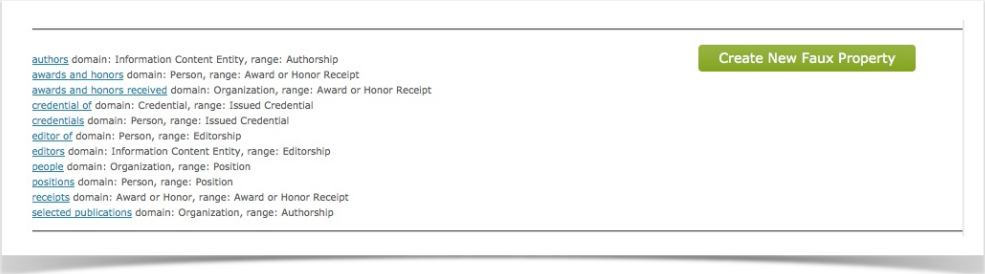
Getting started

VIVO comes with a default set of faux properties. You can edit them to suit your display requirements, or make more extensive modifications.

What to do

There are several ways to navigate to the Faux Property Editing Form. Perhaps the most common way is to show the profile page for an individual, and turn on **verbose property display**.

In this example, we see that `positions` is a faux property of `vivo:relatedBy`. If you click on the link for `positions`, you will see the Faux Property Editing Form for `positions`. On the other hand, if you click the link for `vivo:relatedBy`, you will see the Object Property Editing Form for `vivo:relatedBy`. In addition to the parameters for `vivo:relatedBy`, you will also see links to each of the faux properties that are based on it:



From each of these pages, you can modify or delete the faux property that is displayed.

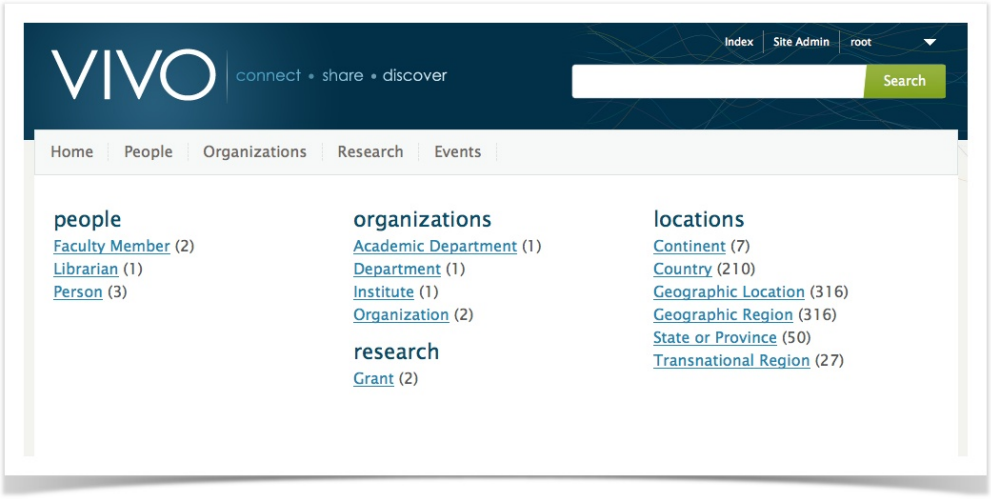
3.2.4 Edit class groups

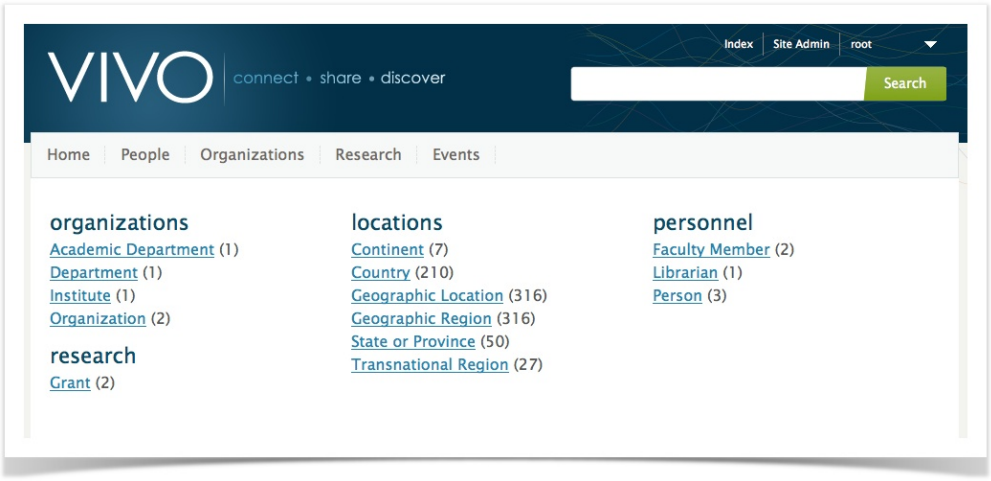
Control the labels of class groups, and the order in which they appear.

Overview

Before and After

Rename "people" to "personnel", and move it to the end of the display order.





What do you need to know?

How to follow the GUI for class groups.

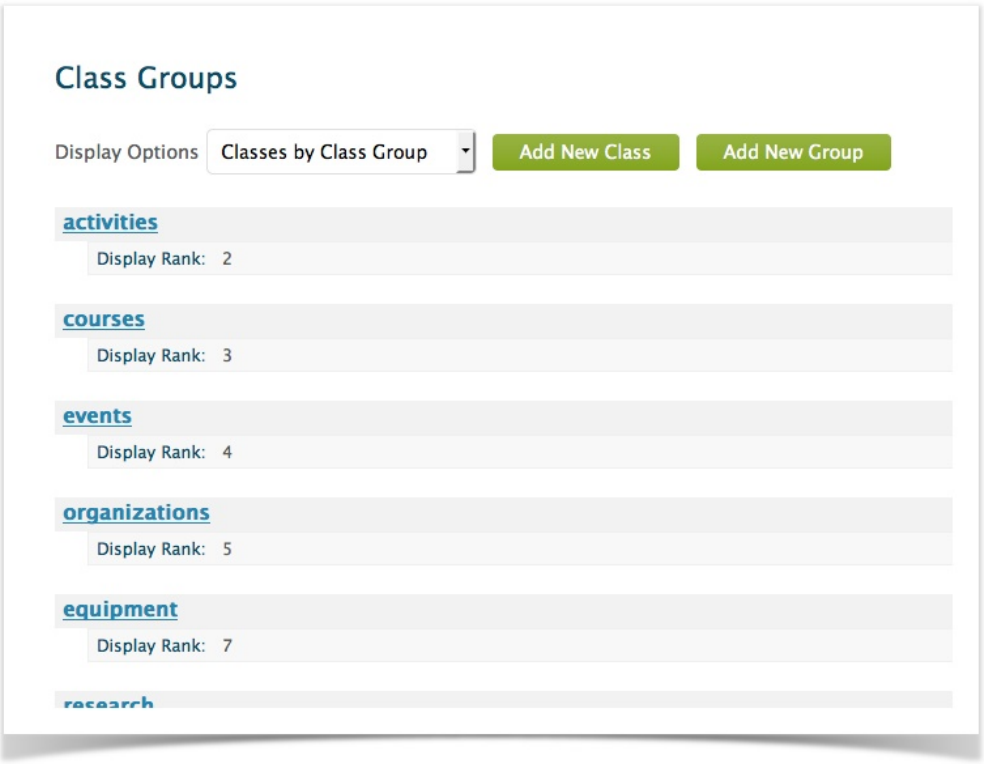
Getting started

VIVO comes with a default set of class groups. You can rename them, reorder them, create new groups or delete existing ones. You can also move classes from one group to another, but that is covered in Edit classes.

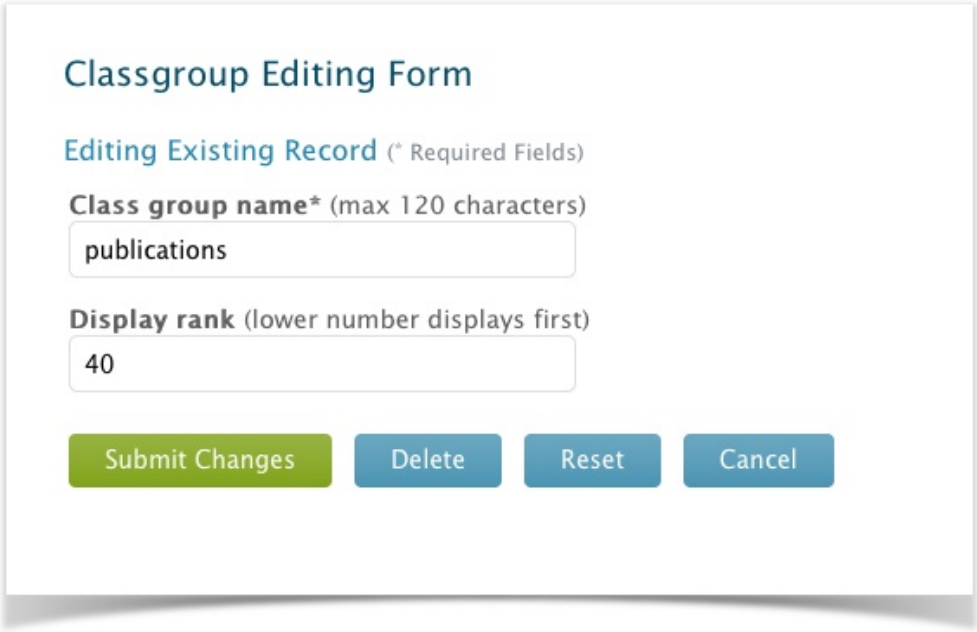
What to do

From the VIVO ***Site Admin*** page, navigate to the ***Class Groups*** page.

You can add a new class group, or click on the name of an existing group to edit it. You can also create a new class, but that involves editing the ontology.



You can change the name of a group, change its display rank, or delete it.



When the index page is displayed, the class groups are shown in order of ascending display rank.

Classes that aren't included in any of the groups are not displayed on the index page. If you delete a class group, the classes that were in it will not be displayedon the index page unless you assign them to new groups.

Class groups and their membership can also affect the contents of managed pages. "Browse"-style pages display the contents of some or all of the classes in a single group, so the structure of your class groups will affect how these pages can be configured. Find more information in the section on Page management.

Class groups are also used to provide facets in search results.

3.2.5 Edit the appearance of classes

Change how VIVO displays classes, individually or in class groups.

Overview

What can it do for you?

Change how VIVO displays a class of individuals.

For each class, you can change

- the display label
- the public and private descriptions
- which class group it belongs to
- who can see the values
- who can edit the values

- whether the values will be published in linked open data requests

You can also change things like the namespace and parent class, but these are changes to the ontology.

The class editing form also allows you to assign a custom entry form to a class, as described in Custom entry forms

Before and After

Rename "Faculty Member" to "Member of the Faculty", and move it from the "People" class group to the " Research" class group.

What do you need to know?

How to follow the GUI for class editing.—

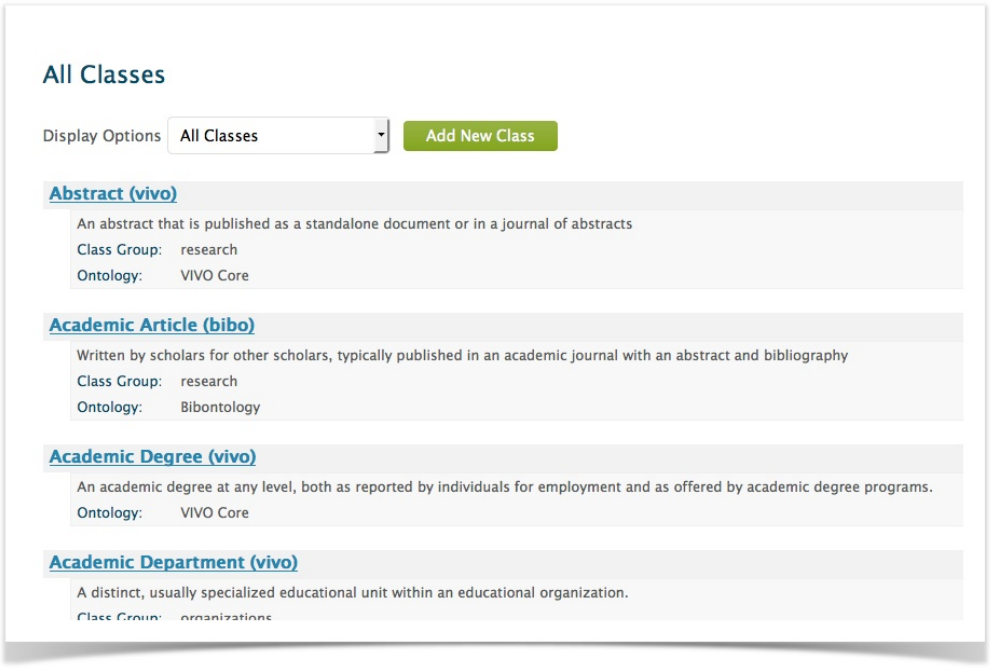
Getting started

VIVO comes with a default set of classes. You can edit them to suit your display requirements, or make more extensive modifications, by customizing the ontology.

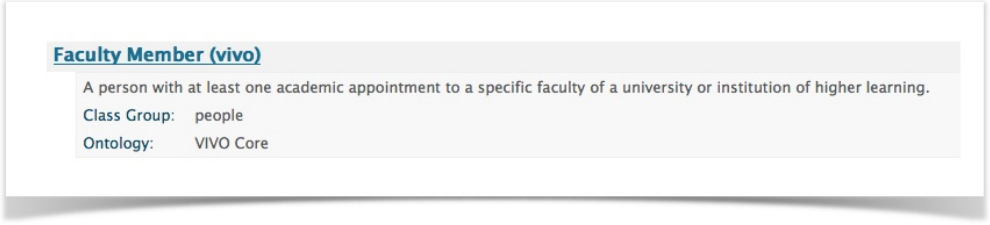
What to do

From the VIVO ***Site Admin*** page, navigate to the ***Class Hierarchy*** page.

If you know the ancestry of the class you want to change, you can navigate to it through the hierarchy. Otherwise, you may want to set the display options to show `All Classes`, and scroll directly to the class you want.

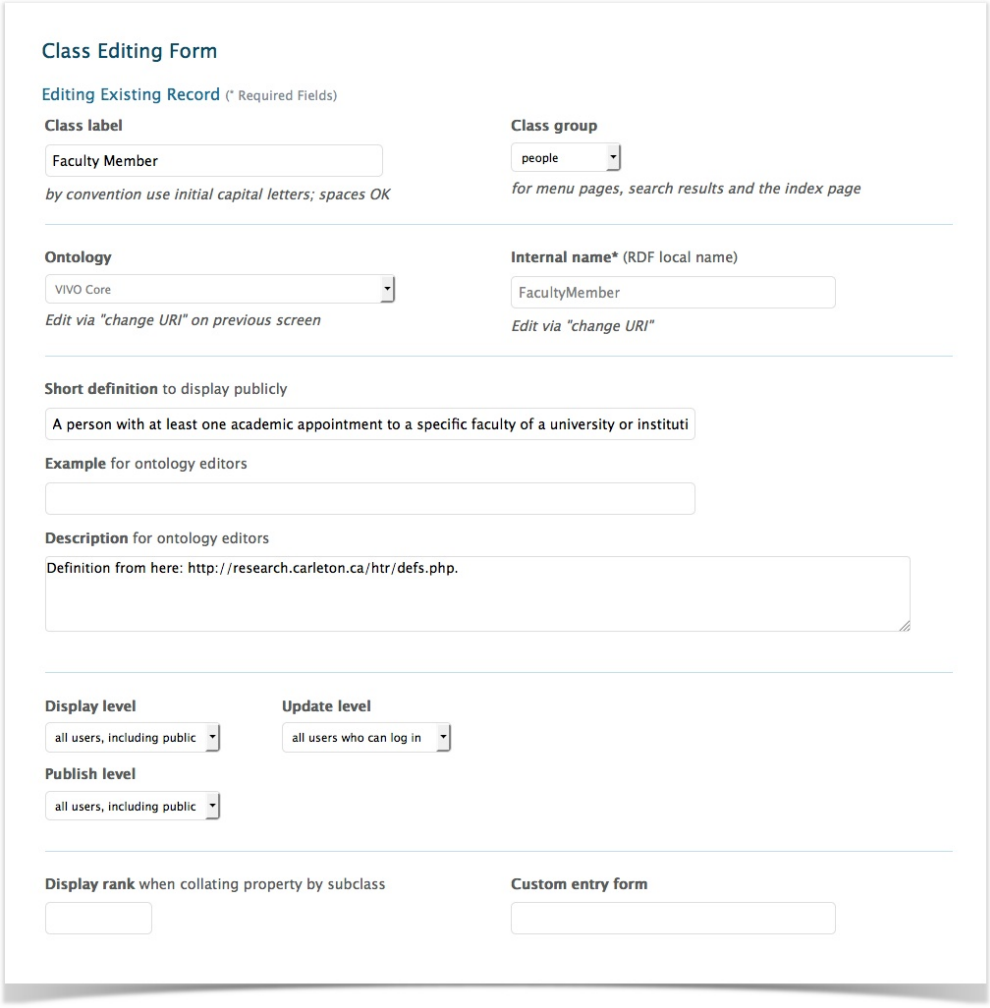


Click on the name of the class you want to edit.



This shows you the ***Class Control Panel***. Click on the `Edit Class` button, and you will see the ***Class Editing*** **Form**.

You can change the class label, or the membership in a class group. You can also change the definition and description of the class.



When you have made the changes, click on `Submit Changes`, and navigate to a page where you can see the results.

Notes

There is no need to restart or rebuild VIVO to see the effects of your changes.

3.3 Home page customizations

Adjust the dynamic sections on the home page, including the class group displays and the geographic focus map.

3.3.1 Introduction

For Release 1.6 the VIVO Home Page has been redesigned. The Search field beneath the "welcome" text now allows the user to limit the results of a search to a specific class group, such as people, organizations, etc. In addition, the browse-by-class-group display has been removed from the Home Page and replaced by multiple features, which include: a list of four randomly selected faculty members, including their titles and thumbnail images; a display of statistical data about the VIVO installation, such as the number of people, activities and organizations; and, optionally, a global map showing researchers' areas of geographic focus.

This wiki page will explain how you can modify the "Research," "Faculty" and "Departments" sections of the home page, as well as how to expand the map section to include country-specific and state or province-specific maps.

3.3.2 The page-home.ftl Template File

The new sections of the home page are all referenced as macros in the page-home.ftl template file. The macros themselves are all located in the lib-home-page.ftl file, which is imported into the page-home.ftl file via this line:

```
<#import "lib-home-page.ftl" as lh>
```

The code below is from the page-home.ftl template and shows how the macros are referenced. So, for example, if you wanted to modify the order in which these sections appear on the home page, you would move the macro references accordingly.

```
68        <!-- List of research classes: e.g., articles, books, collections, conference papers -->
69        <@lh.researchClasses />
70
71        <!-- List of four randomly selected faculty members -->
72        <@lh.facultyMbrHtml />
73
74        <!-- List of randomly selected academic departments -->
75        <@lh.academicDeptsHtml />
76
77        <#if geoFocusMapsEnabled >
78            <!-- Map display of researchers' areas of geographic focus. Must be enabled in runtime.properties -->
79            <@lh.geographicFocusHtml />
80        </#if>
81
82        <!-- Statistical information relating to property groups and their classes; displayed horizontally, not vertically-->
83        <@lh.allClassGroups vClassGroups! />
84
85        <#include "footer.ftl">
86        <#-- builds a json object that is used by js to render the academic departments section -->
87        <@lh.listAcademicDepartments />
```

3.3.3 The Research Section

It's possible that your VIVO installation has defined some of its own classes within the Research Class group . Cornell's VIVO, for example, has a Library Collection class and a Media Contributions class. If your installation does include its own classes in this group, you can display these in the Research section of the home page by modifying the researchClasses macro in the lib-home-page.ftl file. As shown in line 128 below , the classes that get displayed are hard-coded into the macro. Simply exchange the name of your classes with some or all of the ones below. You could also add your classes to the existing list.

```
119 <#macro researchClasses classGroups=vClassGroups>
120 <#assign foundClassGroup = false />
121 <section id="home-research" class="home-sections">
122     <h4>${i18n().research_capitalized}</h4>
123     <ul>
124         <#list classGroups as group>
125             <#if (group.individualCount > 0) && group.displayName == "research" >
126                 <#assign foundClassGroup = true />
127                 <#list group.classes as class>
128                     <#if (class.individualCount > 0) && (class.name == "Academic Article" || class.name == "Book" || class.name ==
  "Chapter" ||class.name == "Conference Paper" || class.name == "Proceedings" || class.name == "Report") >
129                         <li role="listitem">
130                             <span>${class.individualCount!}</span> 
131                             <a href='${urls.base}/individuallist?vclassId=${class.uri?replace("#","%23")!}'>
132                                 <#if class.name?substring(class.name?length-1) == "s">
133                                     ${class.name}
134                                 <#else>
135                                     ${class.name}s
136                                 </#if>
137                             </a>
138                         </li>
139                     </#if>
140                 </#list>
141                 <li><a href="${urls.base}/research" alt="${i18n().view_all_research}">${i18n().view_all}</a></li>
142             </#if>
143         </#list>
144         <#if !foundClassGroup>
145             <p><li>${i18n().no_research_content_found}</li></p>
146         </#if>
147     </ul>
148 </section>
149 </#macro>
```

It would be possible to display a random selection of classes rather than a hard-coded list, the same way that the Departments section displays a randomly selected list of academic departments. To do this, you would have to copy the macros and java script used for the academic departments, and them modify it accordingly so that it displays research classes. Refer to The Departments Section below for more details.

3.3.4 The Faculty Section

There's very little customization that can be done to the faculty section of the home page, excluding css changes and relocating the section to another part of the home page. The one configurable piece is the number of faculty members that get displayed. This change is made in the homePageUtils.js file. Locate the getFacultyMembers function and modify the pageSize variable (shown in line 29 below).

```
22      function getFacultyMembers() {
23          var individualList = "";
24
25          if ( facultyMemberCount > 0 ) {
26              // determine the row at which to start the solr query
27              var rowStart = Math.floor((Math.random()*facultyMemberCount));
28              var diff;
29              var pageSize = 4; // the number of faculty to display on the home page
30
```

3.3.5 The Departments Section

The list of academic departments is a randomly selected list that relies on a data getter as well as two macros in the lib-home-page.ftl file. The data getter is defined in the homePageDataGetters.n3 file. If you want to display something other than academic departments, you need to update the SPARQL query portion of the data getter, shown in lines 18-30 below. Substitute the class you want to display for vivo: AcademicDepartment.

```
11  # academic departments datagetter
12
13  <freemarker:lib-home-page.ftl> display:hasDataGetter display:academicDeptsDataGetter .
14
15  display:academicDeptsDataGetter
16      a <java:edu.cornell.mannlib.vitro.webapp.utils.dataGetter.SparqlQueryDataGetter> ;
17      display:saveToVar "academicDeptDG" ;
18      display:query """
19      PREFIX rdfs: <http://www.w3.org/2000/01/rdf-schema#>
20      PREFIX rdf: <http://www.w3.org/1999/02/22-rdf-syntax-ns#>
21      PREFIX vivo: <http://vivoweb.org/ontology/core#>
22
23      SELECT DISTINCT ?deptURI (str(?label) as ?name)
24      WHERE
25      {
26            ?deptURI rdf:type vivo:AcademicDepartment .
27            ?deptURI rdfs:label ?label
28      }
29
30      """ .
```

It is possible to expand the query to include more than one class. To do so without having to make any other macro or template changes, use UNION clauses in your query, as follows:

```
PREFIX rdfs: <http://www.w3.org/2000/01/rdf-schema#>
PREFIX rdf: <http://www.w3.org/1999/02/22-rdf-syntax-ns#>
PREFIX vivo: <http://vivoweb.org/ontology/core#>
SELECT DISTINCT ?theURI (str(?label) as ?name)
WHERE
{{
   ?theURI rdf:type vivo:AcademicDepartment .
   ?theURI rdfs:label ?label .
}
UNION
{
   ?theURI rdf:type vivo:Association .
   ?theURI rdfs:label ?label .
}}
```

The following code snippet shows the two macros used to render the Departments section. If you change the data getter to use a different class, you do not have to change any variable or macro names. The only change you'll need to make is to the heading of this section so that it correctly reflects the class being displayed. Line 155 (below) is where you would make the change. (Note the use of the internationalization variable. As part of your change, you may want to update the i18n/all.properties file to include your new secion heading.

```
151 <#-- Renders the html for the academic departments section on the home page. -->
152 <#-- Works in conjunction with the homePageUtils.js file -->
153 <#macro academicDeptsHtml>
154     <section id="home-academic-depts" class="home-sections">
155         <h4>${i18n().departments}</h4>
156         <div id="academic-depts">
157         </div>
158     </section>
159 </#macro>
160
161 <#-- builds the "academic departments" box on the home page -->
162 <#macro listAcademicDepartments>
163 <script>
164 var academicDepartments = [
165 <#if academicDeptDG?has_content>
166     <#list academicDeptDG as resultRow>
167         <#assign uri = resultRow["theURI"] />
168         <#assign label = resultRow["name"] />
169         <#assign localName = uri?substring(uri?last_index_of("/")) />
170             {"uri": "${localName}", "name": "${label}"}<#if (resultRow_has_next)>,</#if>
171     </#list>
172 </#if>
173 ];
174 var urlsBase = "${urls.base}";
175 </script>
176 </#macro>
```

3.3.6 The Geographic Focus Map

The new map on the home page uses circular markers to show the countries and regions that researchers in a VIVO installation have chosen as their areas of geographic focus (vivo:GeographicFocus). Clicking on a marker takes the user to that country or region's profile page, which shows the list of researchers in that location. The map is built using the Leaflet.js java script library, map tiles provided without charge by ESRI, and geographical data stored in a JSON file.

How the Map Works

When the home page gets loaded, three java script files relating specifically to the map are sourced in: leaflet.js, latLongJson.js and homePageMaps.js. The first is the java script library that does the actual map rendering, from sourcing in the map tiles to placing the markers on the map. The second file contains a JSON array containing geographic data such as the names of countries and regions, their latitude and longitude, and some additional information that is used to build the GeoJSON object. The last file, homePageMaps.js, contains the functions that serve as the driver for rendering the map. The following outline covers the sequence of those events.

1) The getGeoJsonForMaps() function uses an AJAX request to call the GeoFocusMapLocations.java class. The purpose of this class is to run the SPARQL query that retrieves the names of the countries and regions that researchers have selected as areas of geographic focus as well the number of researchers associated with each area.

2) Once the SPARQL query results are returned to the getGeoJsonForMaps() function, it then parses the results and uses several function calls to build the GeoJSON array that gets used by the Leaflet java script. For example, the getLatLong() function call gets the longitude and latitude of a geographic area from the latLongJson.js file. The GeoJSON array, which is stored in a variable named "researchAreas," takes this format:

```
{ "type": "FeatureCollection",
  "features": [{'geometry': {'type': 'Point', 'coordinates': '-64.0,-34.0'},
                'type': 'Feature',
                'properties': {'mapType': 'global',
                               'popupContent': 'Argentina',
                               'html': '1',
                               'radius': '8,
                               'uri': 'http%3A%2F%2Faims.fao.org%2Faos%2Fgeopolitical.owl%
23Argentina'}},
                {'geometry': {'type': 'Point', 'coordinates': '-2.0,54.0'},
                'type': 'Feature',
                'properties': {'mapType': 'global',
                               'popupContent': 'United Kingdom',
                               'html': '6',
                               'radius': '10,
                               'uri': 'http%3A%2F%2Faims.fao.org%2Faos%2Fgeopolitical.owl%
23United_Kingdom'}},
                ... ]
}
```

3) Once the researchAreas variable is set, the buildGlobalMap() function is called. The main portion of that function is shown below:

```
176         var mapGlobal = L.map('mapGlobal').setView([25.25, 23.20], 2);
177             L.tileLayer('http://server.arcgisonline.com/ArcGIS/rest/services/World_Shaded_Relief/MapServer/tile\/{z}\/{y}\/{x}.png', {
178                 maxZoom: 12,
179                 minZoom: 1,
180                 boxZoom: false,
181                 doubleClickZoom: false,
182                 attribution: 'Tiles &copy; <a href="http://www.esri.com/">Esri</a>'
183         }).addTo(mapGlobal);
184 
185         L.geoJson(researchAreas, {
186 
187             filter: checkGlobalCoordinates,
188             onEachFeature: onEachFeature,
189 
190             pointToLayer: function(feature, latlng) {
191                 return L.circleMarker(latlng, {
192                     radius: getMarkerRadius(feature),
193                     fillColor: getMarkerFillColor(feature),
194                     color: "none",
195                     weight: 1,
196                     opacity: 0.8,
197                     fillOpacity: 0.8
198                 });
199             }
200         }).addTo(mapGlobal);
201 
202         L.geoJson(researchAreas, {
203 
204             filter: checkGlobalCoordinates,
205             onEachFeature: onEachFeature,
206 
207             pointToLayer: function(feature, latlng) {
208                 return L.marker(latlng, {
209                     icon: getDivIcon(feature)
210                 });
211             }
212         }).addTo(mapGlobal);
```

Here are some key points to note about the previous code:

1.
   - The "L." references in the above code are calls to to Leaflet java script library.
   - The setView function in line 176 uses latitude and longitude coordinates to center the display of the map.
   - Also in line 176, 'mapGlobal' (in L.map('mapGlobal')...) is the name of the element in which Leaflet will render the html for the map.

The geographicFocusHtml Macro

As noted earlier, the lib-home-page.ftl file contains the macros that are used to build the new sections on the home page. Here is the geographicFocusHtml macro:

```
178 <#-- renders the "geographic focus" section on the home page. works in      -->
179 <#-- conjunction with the homePageMaps.js and latLongJson.js files, as well -->
180 <#-- as the leaflet javascript library.                                      -->
181 <#macro geographicFocusHtml>
182     <section id="home-geo-focus" class="home-sections">
183         <h4>${i18n().geographic_focus}</h4>
184         <#-- map controls allow toggling between multiple map types: e.g., global, country, state/province. -->
185         <#-- VIVO default is for only a global display, though the javascript exists to support the other   -->
186         <#-- types. See map documentation for additional information on how to implement additional types.  -->
187         <#--
188             <div id="mapControls">
189                 <a id="globalLink" class="selected" href="javascript:">Global Research</a> | 
190                 <a id="countryLink" href="javascript:">Country-wide Research</a> | 
191                 <a id="localLink" href="javascript:">Local Research</a>
192             </div>
193         -->
194         <div id="researcherTotal"></div>
195         <div id="timeIndicatorGeo">
196             <span>${i18n().loading_map_information}   
197                 <img  src="${urls.images}/indicatorWhite.gif">
198             </span>
199         </div>
200         <div id="mapGlobal" class="mapArea"></div>
201        <#--
202             <div id="mapCountry" class="mapArea"></div>
203             <div id="mapLocal" class="mapArea"></div>
204        -->
205     </section>
206 </#macro>
```

Note line 200: this is the element where Leaflet renders the map.

Customizing the Look of the Map

There are three principal ways to customize the look of the map:

1. Change the source of the map tiles that provide the "atlas"
2. Change the colors of the markers
3. Change the size of the markers

Change the source of the map tiles

This is the most significant modification that you can make. The map currently uses tiles provided by ESRI, which has other map tiles for you to use. Mapquest is another source of free map tiles, as is Google. OpenCloud is a source of map tiles but they charge a small fee.

To change the tiles, you need to update the L.tileLayer definition in the buildGlobalMap() function. This is shown in line 177 above. Simply change the URL to the URL of the service providing your map tiles. (That service may also use a slightly different API.)

Change the colors of the markers

You can change the marker colors in the getMarkerFillColor() function in homePageMaps.js. Note that there are separate colors for countries and regions. If you do change the colors of the markers, you will also have to update the legend that appears in the lower left corner of the map. The circles in this legend are actually image files (map_legend_countries.png and map_legend_regions.png), so you will have to create new image files to match the colors you have chosen for markers.

Change the size of the markers

The size of the markers is the value that is set in the "radius" property in the GeoJSON array. This value is actually calculated in the GeoFocusMapLocations.java class. You can either update this class or add a new function to homePageMaps.js and modify the radius value in that java script file.

Enabling the Country and State/Province Maps

Currently, the home page map section only shows one map view: a global view with markers displayed for regions and countries. However, the code is available to include two additional views, one for a specific country and one for a specific state or province within a country. These are the steps you need to follow to implement the other two map views.

1. Update the geoFocusHtml macro
2. Update the coordinates in the setView() function
3. Update the getResearcherCount() function
4. Update the latLongJson.js file
5. Update the SPARQL query in the GeoFocusMapLocations.java class
6. Update your VIVO data as necessary

Update the geoFocusHtml macro

If you are using multiple map views, than you need to uncomment the mapControls element in the geoFocusHtml macro (`<div id="mapControls">`). If you are only implementing two views (global and country), then you will want to ensure that the "localLink anchor tag is commented out (`<a id="localLink " href="javascript:">`). These anchor tags, along with corresponding java script in the homePageMaps.js file, allow the user to toggle between the implemented map views. (No change to the js file is necessary.)

Next you need to uncomment the elements where the additional map views will be rendered: `<div id="mapCountry" class="mapArea">` and/or `<div id="mapLocal" class="mapArea">`. Again, only uncomment the elements you are implementing.

Update the coordinates in the setView() function

Besides the buildGlobalMap() function (discussed above), the homePageMaps.js file also includes buildCountryMap() and buildLocalMap() functions. These functions are very similar to the buildGlobalMap() function and work in the same way. When you are implementing a country map, you will want the map to be centered on that country.

```
var mapCountry = L.map('mapCountry').setView([46.0, -97.0], 3);
```

The coordinates above, `[46.0, -97.0]`, center the map on the United States. If you want this map to be centered on a different country, you will have to change these coordinates accordingly. The third value in the line of code above, `3`, is the zoom value and sets the default for when the map is loaded. Note that the default zoom value for the global map is `2` and the default for the local map could be any thing from `4` to `8` depending on the location you are displaying.

Update the getResearcherCount() function

For all three types of views, the map includes summary text that shows the total number of researchers and geographical areas in the results, as show below:

Depending on the map views you implement, and the actual country or areas they display, you may want to modify the wording that gets displayed here. This is done in the getResearcherCount() function in of the homePageMaps.js file. (Note that the text here uses internationalization variables, so you may need to update the i18n/all.properties file as well.)

Update the latLongJson.js file

The latLongJson.js file contains data for countries, transnational regions and states within the United States. Therefore, if your installation wants to implement a country map other than the U.S., you will need to update the latLongJson.js file to include the necessary data. For example, if the country to be displayed is Australia, the latLongJson.js file would need to include data on the states and territories of Australia. The JSON array in this file takes data in this format:

```
{"name": "Victoria", "data": {"mapType": "country", "geoClass": "state", "latitude": "-37.4713",
"longitude": "144.7851"}}
```

Note that the mapType corresponds to the map view, in this case "country" as opposed to "global, while the geoClass corresponds (loosely) to the VIVO ontology class. ("Loosely," because the class is actually " StateOrProvince.")

Implementing a state/province map would mean updating the latLongJson.js file to include data for the geographic areas within a state. For U.S. states, examples would include counties, townships and even more general areas such as the Hudson or Mohawk valleys in New York (two areas of geographic focus for Cornell researchers). In this third case the mapType must be set to "local."

Update the SPARQL query in the GeoFocusMapLocations.java class

For performance and practical reasons, the SPARQL query in the GeoFocusLocations.java class excludes states and provinces. (Since only the global map is displayed by default, there is no reason to include state and provinces in the query results.) To update the query to include states and provinces, simply remove this line from the query:

```
FILTER (NOT EXISTS {?location a core:StateOrProvince})
```

If you want to implement a state/province map, you may need to update the query further to ensure that the local geographic areas are included in the query results. Although there are VIVO classes for counties and " populated places," your VIVO installation might have additional refinements to the ontology.

Update your VIVO data as necessary

The SPARQL query in the GeoFocusLocations.java class does not simply return a count for the numbers of researchers that have selected a country (for example) as an area of geographic focus. The query also roll-ups the counts for "child" locations into the "parent" location. For example, if 5 researchers have the U.S. as their area of geographic focus, and another 5 researchers have individual states as their focus, the query will return a count of 10 for the U.S. This is accomplished through the vivo:geographicallyContains object property. Similarly, country counts are rolled up into regional counts through the geo:hasMember object property. *It's possible that you will need to curate your VIVO data to ensure that the necessary object property relationships exist in your installation. This is especially true with the local geographical areas.*

3.4 Page management

Change the content of VIVO's managed pages, or create new ones, without changing any files or restarting VIVO.

3.4.1 Overview

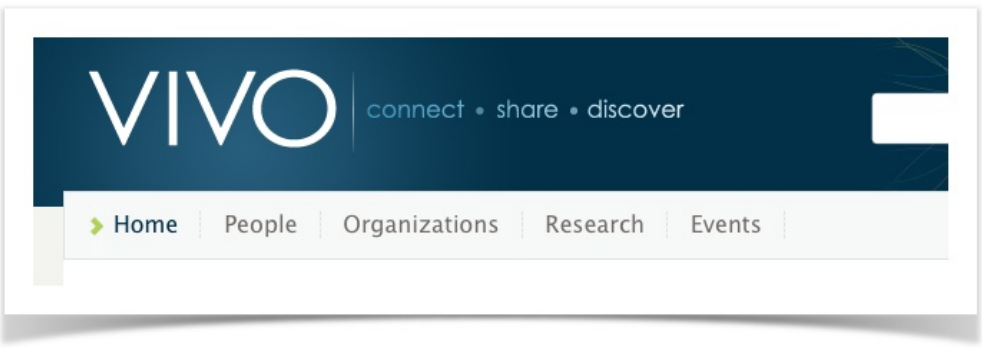
What can it do for you?

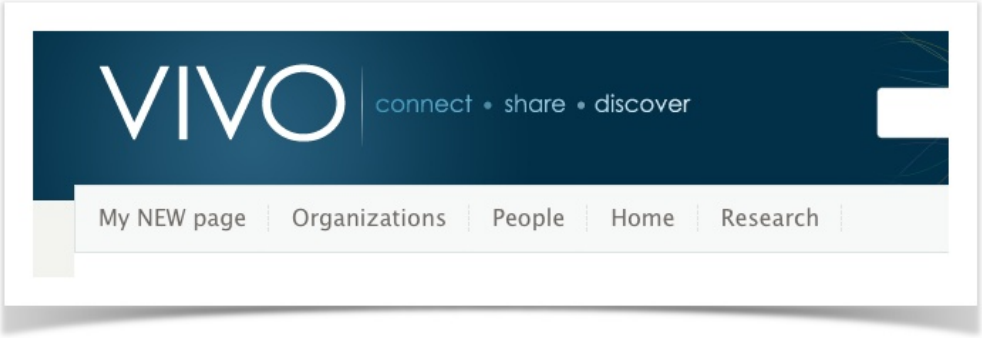
- —Create “browse” pages, static pages, or pages that display the results of a query (reports)
- —Remove existing pages
- —Manipulate the page menu

It's easy to surmise that Page Management only allows you to make changes to the menu. And its true that you can create new menu pages, rearrange the menu, or remove items from it.

But you can also use Page Management to create pages that aren't in the menu. You assign a simple URL to each page, so you can link to them from your other pages. The content of the pages can be simple HTML, or the results of a SPARQL query, or a "browse" page for individuals in VIVO.

Before and After





What do you need to know?

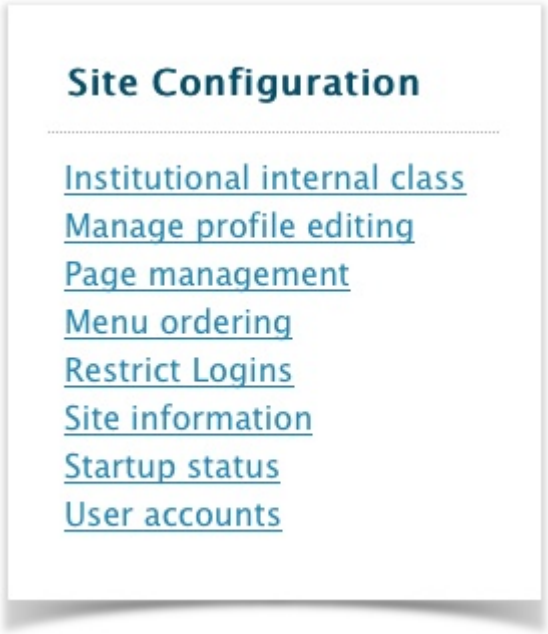
- —How to follow the GUI for page management
- —Optional – how to write Freemarker templates
- —Optional – how to write SPARQL queries

Getting started

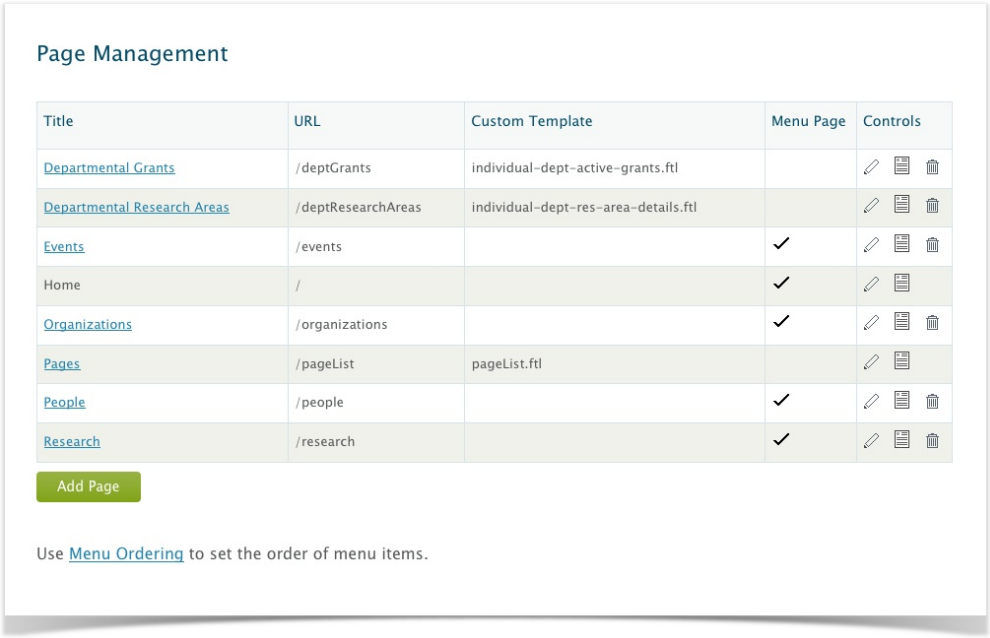
VIVO comes with a set of managed pages, including the ones that you see in the menu on each page.

3.4.2 What to do

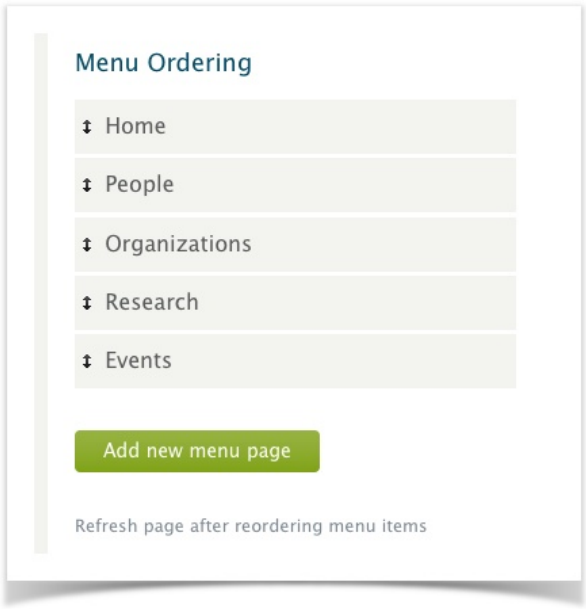
Go to the ***Site Admin*** page, and choose ***Page management***.



Use the links provided to create new pages, or edit existing ones.



Click on the Menu Ordering link to re-arrange the menu. Drag entries up or down to establish the order you want. When you refresh the page, or go to another, you will see your changes in the menu.



3.5 Class-specific templates for profile pages

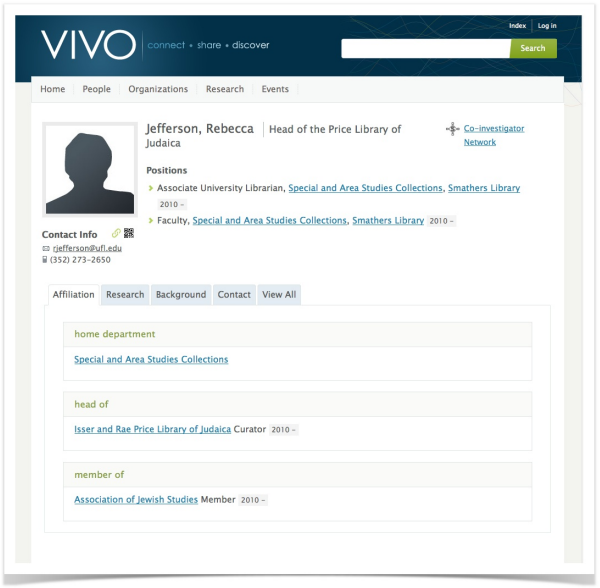
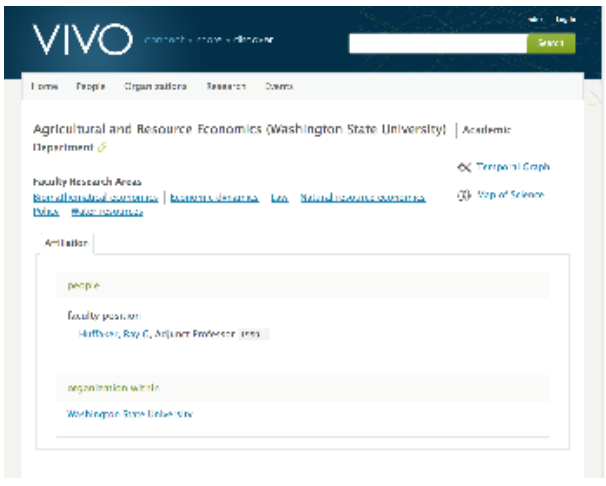
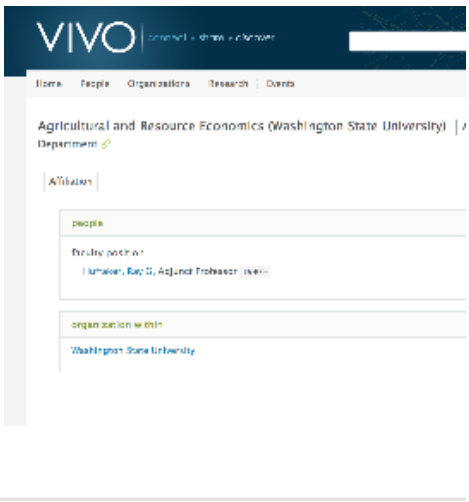
Create profile templates for specific classes, such as Organization, Faculty Member, or Grant.

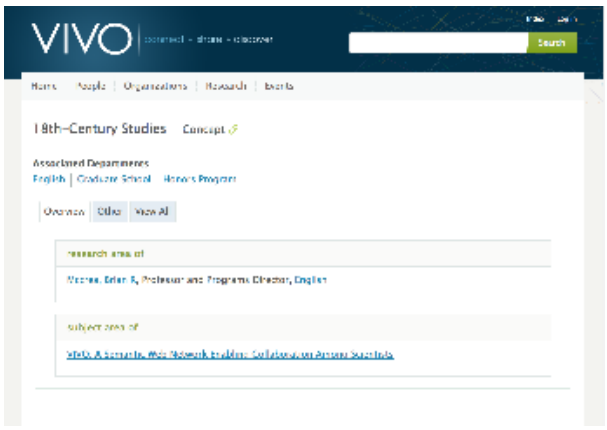
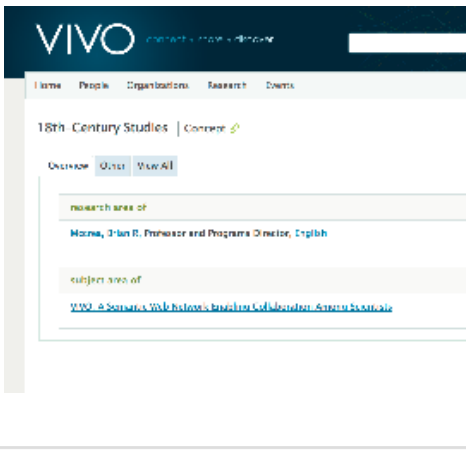
3.5.1 Overview

When the profile page for an individual is created, it includes the standard header and footer, but most of the content is built by the Freemarker template `individual.ftl`.

Sometimes you would prefer for a particular class of individuals to have a particular style of profile page.For example, you want the contact information for a `foaf:Person` to appear right below their picture. Or you want to see a link to their co-investigator network near the top of the page.

VIVO lets you specify different templates for different classes of individuals. The standard distribution includes three of those specifications. Here are examples of profile pages for a Person, and Organization, and a Concept, with and without specified templates.

	specified template	default template
Person		
Organization		

	specified template	default template
Concept		

A specified profile template applies to individuals of the specified class, and to individuals of all sub-classes. So a page specified for `foaf:Person` will also apply to its sub-classes, like `vivo:FacultyMember`.

3.5.2 How to do it

There is no page in VIVO that will allow you to set or change these template specifications. Here are two ways to set it up.

Changes on an empty data model

When you start an empty VIVO instance for the first time, it will load the files in the `rdf/tbox/firsttime` directory into the `asserted-tbox` model.

The file `initialTBoxAnnotations.n3`, in this directory, contains the triples that specify these profile templates. They are scattered in the file, and mingled with other triples, but if you look, you can find these statements:

```
foaf:Organization
      vitro:customDisplayViewAnnot
              "individual--foaf-organization.ftl"^^xsd:string .
foaf:Person
      vitro:customDisplayViewAnnot
              "individual--foaf-person.ftl"^^xsd:string .
skos:Concept
      vitro:customDisplayViewAnnot
              "individual--skos-concept.ftl"^^xsd:string .
```

You can remove triples from this file prior to the first startup of VIVO, or add triples to create other template specifications.

Changes to an existing data model

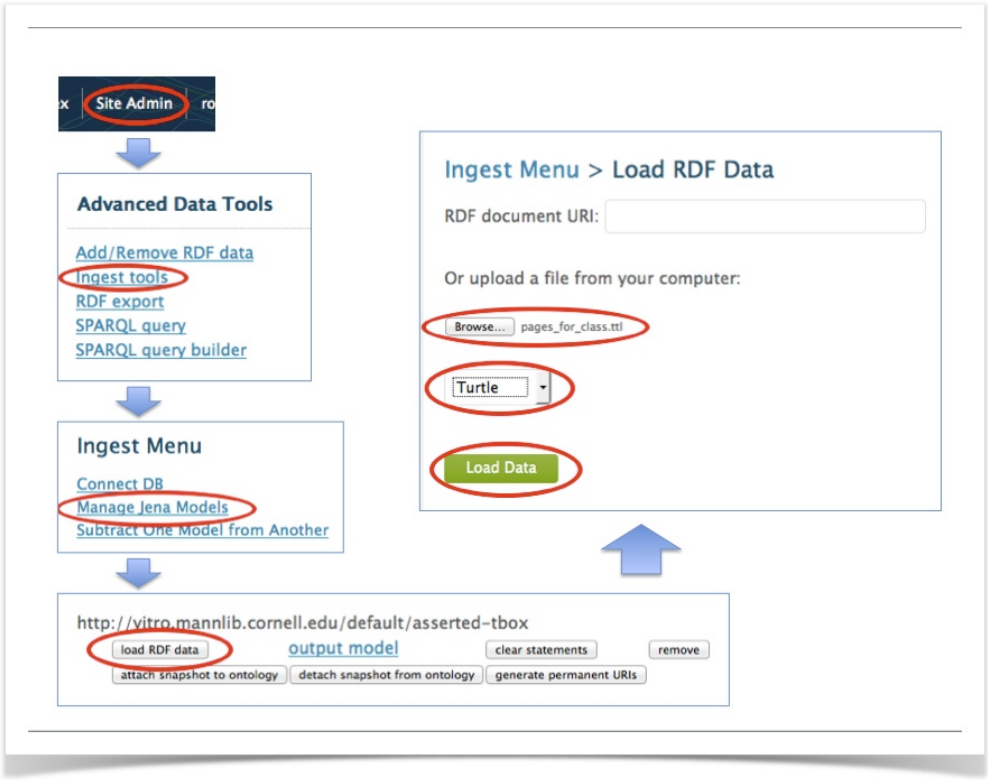
If VIVO has already been started, the files in `rdf/tbox/firsttime` will not be read again. You can use the advanced data tools on the Site Administrator's page to make changes.

Adding specifications

- Create a specialized Freemarker template.
- Prepare an RDF file containing the triples you want to add. Here is an example, in Turtle format:

```
@prefix bibo:    <http://purl.org/ontology/bibo/> .
@prefix xsd:     <http://www.w3.org/2001/XMLSchema#> .
bibo:Article
      vitro:customDisplayViewAnnot
              "individual--bibo-article.ftl"^^xsd:string .
```

- Login to VIVO as an admin user
- Follow the header link to the `Site Admin` page
- On the `Site Admin` page, under `Advanced Data Tools`, choose `Ingest tools`.
    - The link to `Add/Remove RDF data` is tempting, but it does not allow us to load into a specific model.
- On the `Ingest Menu` page, choose `Manage Jena Models`.
- In the list of available models, locate the controls for `http://vitro.mannlib.cornell.edu/default/asserted-tbox`, and choose `load RDF data`.
- On the `Load RDF Data` page, use the `Browse` control to locate your RDF file, and select the type of RDF format you used. Click the `Load Data` button.

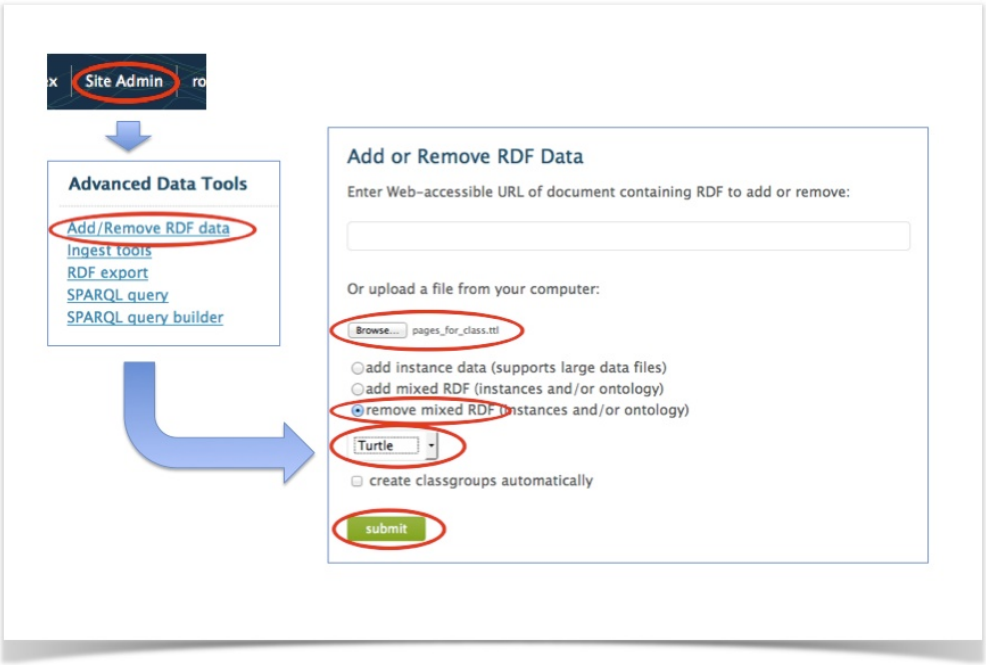


If there is a problem with the load, you will see a screen that shows an error message. Unfortunately, if the load is successful, you will see no indication. You will simply be returned to the list of available models.

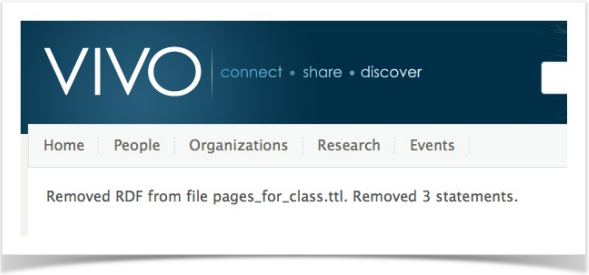
You must restart VIVO to see the effect of your changes.

Removing specifications

- Create an RDF file containing the triples you want to remove. In this example, we will remove the triple that was added above, so we will use the same file.
- Login to VIVO as an admin user.
- Follow the header link to the `Site Admin` page.
- On the `Site Admin` page, under `Advanced Data Tools`, choose `Add/Remove RDF data`.
- On the `Add or Remove RDF Data` page, use the `Browse` control to locate your RDF file, choose to `remove mixed RDF`, and select the type of RDF format you used. Click the `submit` button.



If there is a problem with your data file, you will see a screen showing an error message. If the removal is successful, you will see a message like the following:



Note that the message tells you how many triples you asked to have removed, without regard to whether they actually existed in the data model.

You must restart VIVO to see the effect of your changes.

Changing specifications

There is no direct way to replace triples in the data model. Use the preceding steps to remove the triples you don't want, and to add new triples to replace them.

Other mechanisms

Expert VIVO users will be aware of many other ways of adding or removing triples.

Remember that VIVO must be restarted, since the list of specific templates is created at startup.

3.6 Multiple profile types for foaf:Person

Provide multiple styles for profile pages, and let your users choose which style they prefer for their own profile.

3.6.1 Introduction

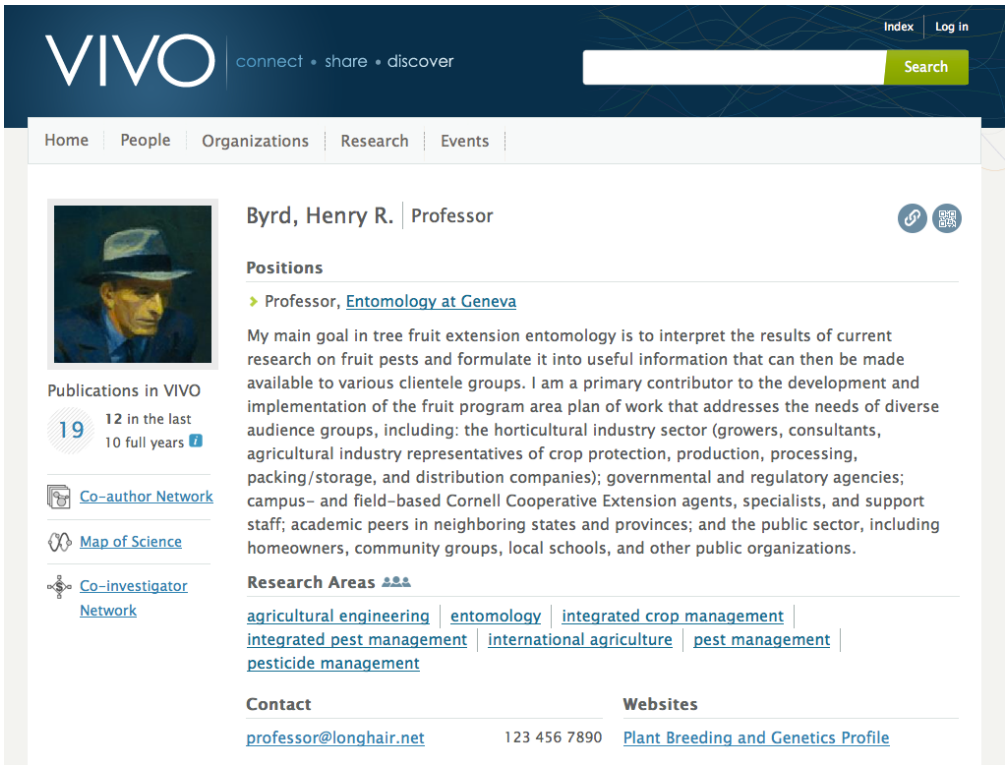
VIVO now supports multiple profile pages for foaf:Persons. This feature, which is optional so installations can continue to use just the individual--foaf-person.ftl template, currently consists of two profile page types: a standard view, which is a redesigned version of the foaf:Person template in previous releases; and a quick view, which emphasizes the individual's own web page presence while providing summary VIVO information , such as current positions and research areas. The profile quick view requires the use of a web service that captures images of web pages. This web service is not included with the VIVO software, so an installation will either have to develop their own service or use a third-party service, usually for a small fee depending on the number of images served. Examples of these services include WebShotsPro, Thumbalizr and Websnapr .

3.6.2 The Profile Page Types

As noted above, there are currently two supported profile page types. Here are examples of those two views

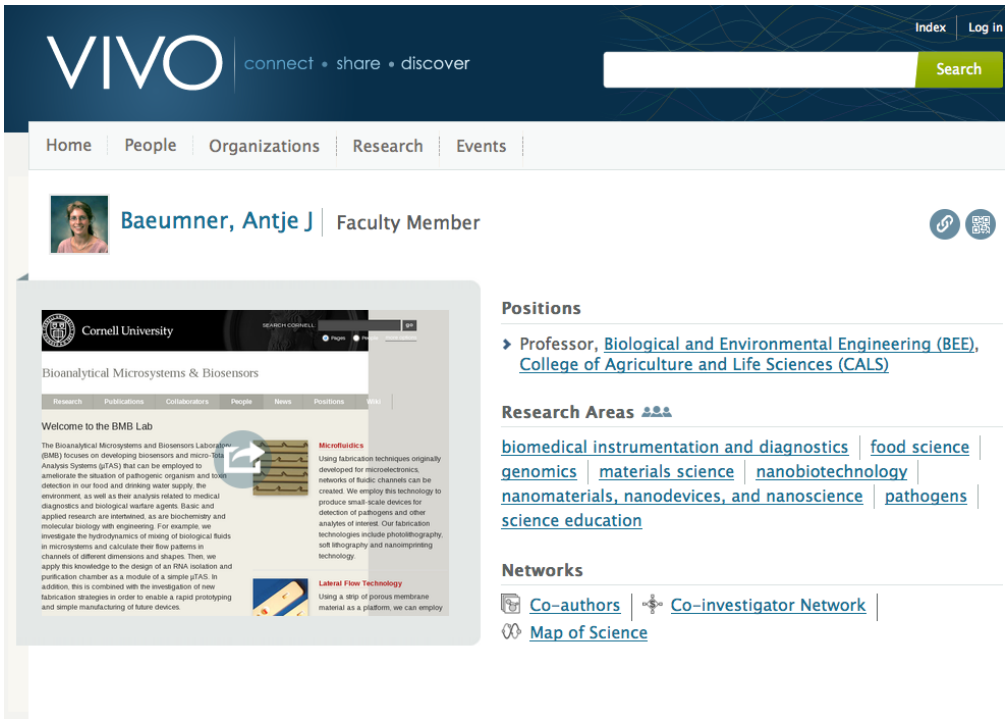
The Standard View

The standard view is similar to the default foaf:Person template except that the information displayed at the top of the page is divided into only two primary columns instead of three. The actual template name for this page type is individual--foaf-person-2column.ftl.

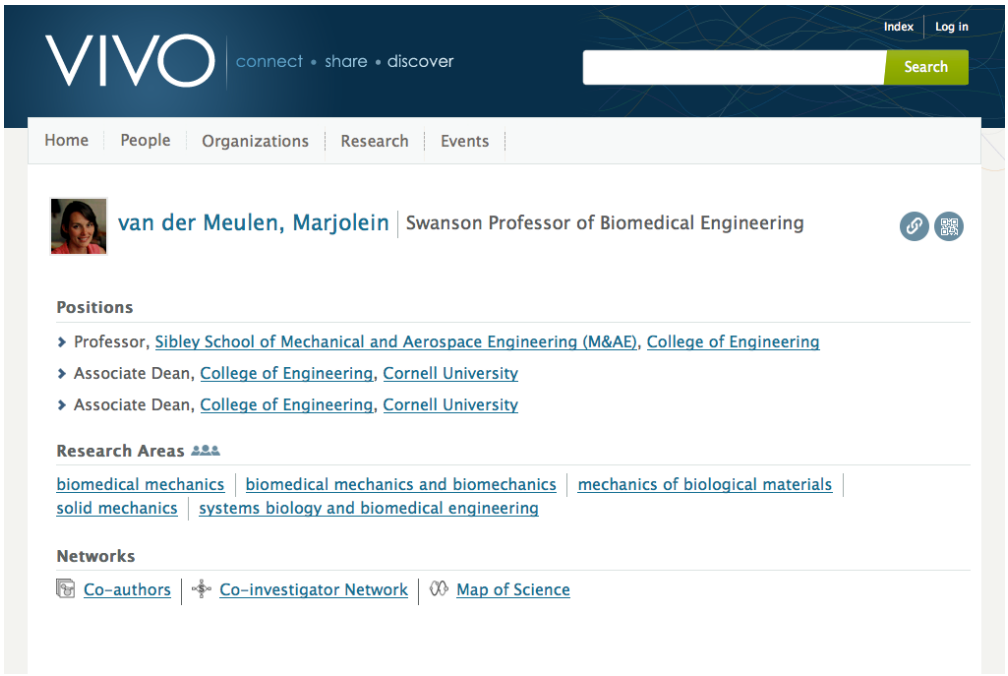


The Quick View

As illustrated below, the quick view puts a visual emphasis on the individual's own web presence. In this case, the person only has one web page displayed. When there is more than one, the primary web page is displayed as shown and any additional web pages are displayed as thumbnails beneath the primary one.



It's possible that there will be some individuals who do not have a web page to display. In that situation the quick view will display as follows.



3.6.3 Implementing Multiple Profile Pages

Here are the steps required to implement the multiple profile pages feature.

1. Develop or a website image capture service
2. Update the runtime.properties file
3. Override the default foaf:Person template
4. Update the webpage quick view template
5. Set the Profile Page Type for your foaf:Persons

Step 1. Develop or a Website Image Capture Service

Since there are currently only two page views, and one of those emphasizes the individual's own web site, to implement the multiple profile pages feature requires that an installation either develop its own web service for capturing images of web sites or select a third-party service for this purpose. As noted in the introduction, these services include WebShotsPro, Thumbalizr and Websnapr.

A third option, however, would be to modify the quick view template (individual--foaf-person-quickview.ftl) so that it does not display a web page image (as in the third screen shot above). This template file is located in the `productMods/templates/freemarker/body/individual` directory.

Step 2. Update the runtime.properties File

Set the multiViews.profilePageTypes to "enabled" and ensure that it is not commented out.

Step 3. Override the Default foaf:Person Template

There are two ways to override the default individual--foaf-person.ftl template, which is located in the `themes/wilma/templates` directory: (1) rename the file, or (2) remove it from that directory.

Step 4. Update the Webpage Quick View Template

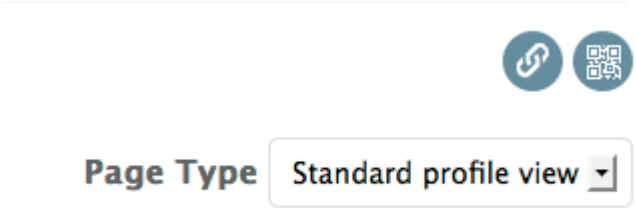
The template that displays the web page image in the quick view is named propStatement-webpage-quickview.ftl. As delivered, this template uses a placeholder link (or links) to display the individual's web page (or pages), while the code that calls the web service is currently commented out. Here is that section of the template:

```
40 <#--  This section commented out until the web service for the web page snapshot is implemented. -->
41 <#--  The assumption is made that the service will require the url of the web page and possibly  -->
42 <#--  an image size as well. Delete the placeholder link once the web service is implemented.    -->
43 <#--
44 <span id="span-${identifier}" class="webpage-indicator-gv">
45     ${strings.loading_website_image}. . .   <img  src="${urls.images}/indicatorWhite.gif">
46 </span>
47 <a title="${i18n().click_to_view_web_page(linkText)}" href="${statement.url}">
48     <img id="img-${identifier}" class="org-webThumbnail" src="http://your.web.service/getsTheImage?url=${statement.url}${imgSize}"
   alt="${i18n().screenshot_of_webpage(statement.url)}" style="display:none"/>
49 </a>
50 <#if imgSize == "" >
51     </li>
52     <li class="weblinkLarge">
53         <a title="${i18n().click_to_view_web_page(linkText)}" href="${statement.url}">
54             <img id="icon-${identifier}" src="${urls.images}/individual/weblinkIconLarge.png"  alt="${i18n().click_webpage_icon}"
   style="display:none"/>
55         </a>
56 <#else>
57     </li>
58     <li class="weblinkSmall">
59         <a title="${i18n().click_to_view_web_page(linkText)}" href="${statement.url}">
60             <img id="icon-${identifier}" src="${urls.images}/individual/weblinkIconSmall.png"  alt="${i18n().click_webpage_icon}"
   style="display:none"/>
61         </a>
62 </#if>
63 -->
64 <#-- Here is the placeholder link, 4 lines long -->
65     <a href="${statement.url}" title="${i18n().link_text}">
66         ${linkText}
67     </a>
68     <script>$("a[title='${i18n().link_text}']").parent('li').css("float","none");</script>
69 <#else>
70     <a href="${profileUrl(statement.uri("link"))}" title="${i18n().link_name}">${statement.linkName}</a> (${i18n().no_url_provided})
71 </#if>
72
```

Note the highlighted text on line 48. The URL in the src attribute is where you call either the web service you developed or the third-party service. The APIs for these services are fairly standard. Besides the URL of the web site that will be the source of the screen shot, the code in this template assumes that the API also takes an image size. For example, some services can provide small, medium and large images; others may only provide a large image and a thumbnail image. Once you've updated line 48 to call your web service, remember to comment out or remove the placeholder link, lines 65-68.

Step 5. Set the Profile Page Type for your foaf:Persons

When multiple profile pages are implemented, the default view is the standard profile view. You can change an individual's profile page type through the GUI by accessing the Page Type drop down:



You can also set the profile page type by ingesting RDF. An N3 triple, for example would consist of the following parts:

- the subject would be the URI of the individual, such as

`<http://localhost:8080/individual/n7829>`;

- the predicate would be the hasDefaultProfilePageType object property,

`<http://vitro.mannlib.cornell.edu/ontologies/display/1.1#hasDefaultProfilePageType>`;

- and the object would be the type of profile,

`<http://vitro.mannlib.cornell.edu/ontologies/display/1.1#quickView>` (or `#standard`).

The ProfilePageType class is defined in the display model. Refer to the profilePageType.n3 file for details.

3.6.4 Using the Standard View Without Implementing Multiple Profile Pages

It's possible that an installation may want to use the standard view instead of the default foaf:Person template, but does not want to implement multiple profile pages. This can be done by simply (1) overriding the default foaf:Person template (just as in Step 3 above) and (2) ensuring that the multiViews.profilePageTypes properties in the runtime.properties file is either commented out or set to " disabled."

3.7 Enriching profile pages using SPARQL query DataGetters

Write SPARQL queries to provide additional information to profile pages, depending on the class of the individual.

3.7.1 Introduction

The VIVO software, beginning with version 1.6, supports the development of SPARQL query data getters that can be associated with specific ontological classes. These data getters, in turn, can be accessed within Freemarker templates to provide richer content on VIVO profile pages. For example, the profile page for an academic department lists only the names of the faculty within that department and their titles, but with a SPARQL query data getter it is now possible to extend the faculty information to display all of the faculty members' research areas.

3.7.2 The Steps and an Example

There are five mandatory steps involved in developing and implementing a class-specific SPARQL query data getter. In this wiki page we'll walk through an example and provide details on each of these steps.

1. Define the customization

2. Write the SPARQL query

3. Produce the N3 for the data getter

4. Create a Freemarker template

5. Incorporate the new template into the application

Step 1. Define the Customization

This first step might seem obvious but it's helpful to define as specifically as possible the change being made to VIVO. For our example, we'll use the one mentioned in the introduction. On academic department pages, we'll provide a list of all the faculty members' research areas and we'll display these beneath the department overview near the top of the page. In addition, we want the listed research areas to be links that will take us to a detail page that shows all of the faculty who have selected a given research area. This last requirement, being able to drill down to a details page, requires both an additional template and data getter, and so we'll need an optional sixth step: Create the Drill-down Page Using Page Management.

Step 2. Write the SPARQL Query

Having defined our requirements, we now need to write a query that will return the data we want -- specifically, the rdfs labels of the research areas and, because we want to be able to drill-down on these labels, the URI of the research areas. An obvious place to write and test a query is the SPARQL Query page that you can access from the Site Admin page. Here's our test query:

```
PREFIX rdfs: <http://www.w3.org/2000/01/rdf-schema#>
PREFIX vivo: <http://vivoweb.org/ontology/core#>
SELECT DISTINCT (str(?researchAreaLabel) AS ?raLabel) ?ra
WHERE {
        <http://localhost:8080/individual/n2936> vivo:organizationForPosition ?posn .
        ?posn  vivo:positionForPerson ?person .
        ?person vivo:hasResearchArea ?ra .
        ?ra rdfs:label ?researchAreaLabel
}
ORDER BY ?raLabel
```

There are two points to note here. In line 3 of the query we convert the label variable to a string to prevent any duplicate labels from appearing; and in line 5 we use the specific URI for an academic department. This URI allows us to test the query, but it will have to be replaced by a "generic" subject in our next step.

Step 3. Produce the N3 for the Data Getter

Once the SPARQL query has been tested, we define the data getter using triples stored in a .N3 file. This file is then placed in the WEB-INF directory in the VIVO source code, as follows: `rdf/display/everytime/deptResearchAreas.n3`.

The N3 for our data getter consists of two parts: (1) the triple that associates our data getter with the AcademicDepartment class and (2) the triples that define the data getter itself. Here is the former:

```
<http://vivoweb.org/ontology/core#AcademicDepartment> display:hasDataGetter display:
getResearchAreaDataGetter .
```

And here are the triples that define the `getResearchAreaDataGetter` data getter:

```
display:getResearchAreaDataGetter
    a <java:edu.cornell.mannlib.vitro.webapp.utils.dataGetter.SparqlQueryDataGetter>;
    display:saveToVar "researchAreaResults";
    display:query """
      PREFIX rdfs: <http://www.w3.org/2000/01/rdf-schema#>
      PREFIX vivo: <http://vivoweb.org/ontology/core#>
      PREFIX afn:  <http://jena.hpl.hp.com/ARQ/function#>
      PREFIX foaf: <http://xmlns.com/foaf/0.1/>
      SELECT DISTINCT (str(?researchAreaLabel) AS ?raLabel) ?ra
      WHERE {
         ?individualURI vivo:relatedBy ?posn .
         ?posn a vivo:Position .
         ?posn  vivo:relates ?person .
         ?person a foaf:Person .
         ?person vivo:hasResearchArea ?ra .
         ?ra rdfs:label ?researchAreaLabel
      }
      ORDER BY ?raLabel
      """ .
```

Note that we have exchanged our specific department URI with the variable `?individualURI`. `?individualURI` is a "built-in" variable; that is, when the data getter is executed the value of this variable is set to the URI of the individual whose page is being loaded. So in our example, because we have associated the data getter with the AcademicDepartment class, when the IndividualController loads an academic department, the URI of that department gets set as the value of the `?individualURI` variable in our query.

Also note line 3 of the data getter:

```
display:saveToVar "researchAreaResults".
```

The "save to" variable `researchAreaResults` is what we use to access the query results in our template.

Step 4. Create a Freemarker Template

Now that we've created our data getter, `getResearchAreaDataGetter`, and have a "save to" variable with which to access the query results, we create a Freemarker template -- individual-dept-research-areas.ftl -- and use the <#list> function to loop through and display the results. The following markup is all that's needed in this new template.

```
<#if researchAreaResults?has_content>
    <h2 id="facultyResearchAreas" class="mainPropGroup">
        Faculty Research Areas}
    </h2>
    <ul id="individual-hasResearchArea" role="list">
        <#list researchAreaResults as resultRow>
            <li class="raLink">
                <a class="raLink" href="${urls.base}/deptResearchAreas?deptURI=${individual.uri}
&raURI=${resultRow["ra"]}" title="research area">
                    ${resultRow["raLabel"]}
                </a>
            </li>
        </#list>
    </ul>
</#if>
```

In the very first line we check to ensure that the query actually produced results. If not, no markup of any kind gets rendered. Otherwise, we give the new template section a heading, define an unordered list (<ul>) to contain the research areas, and then loop through the results. Note that the research area labels are contained within an anchor tag (<a>) because we want to be able to use these as links to a list of the faculty members for each research area. The URL in the href attribute includes what looks like a servlet name, `/deptResearchAreas,` and two parameters: `deptURI` and `raURI`. The `deptURI` parameter is the URI of the department that has been loaded by the IndividualController, and this value is accessible through the template variable ${individual.uri}. The `raURI` parameter is the URI of the research area, the value of which is available in our query results. These parameters and the servlet name will be used to develop the drill-down page that lists the faculty members in a department that have an interest in a specific research area.

Step 5. Incorporate the New Template into the Application

Now that we have a template to display the list of research areas, we need to update the individual.ftl template to source in the new template. Since individual.ftl is used to render individuals of many different classes, we include an <#if> statement to ensure that the individual-dept-research-areas.ftl template only gets included when the individual being loaded is an AcademicDepartment:

```
<#if individual.mostSpecificTypes?seq_contains("Academic Department")>
    <#include "individual-dept-research-areas.ftl">
</#if>
```

Step 6. Create the Drill-down Page Using Page Management (optional)

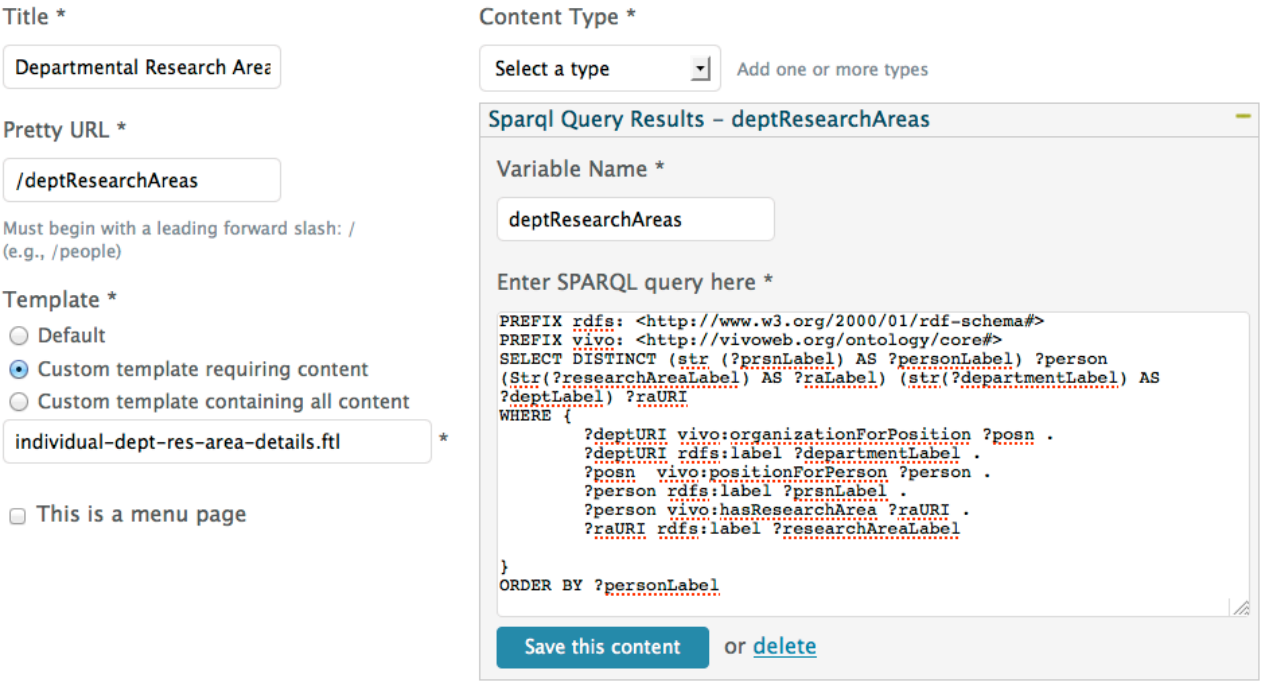
To this point, we have created a class-specific SPARQL query data getter, which retrieves the research areas of the faculty in a given academic department; developed a new template to render the results of our data getter; and updated the individual.ftl template to display the list of research areas. In Step 1, however, we defined requirements that include the ability to drill down from a selected research area to display a list of the faculty members in the department who have an interest in that research area. This is also done using a SPARQL query and new template. But in this case the query does not need to be associated with a specific class and defined in an .N3 file. Instead, we can create a SPARQL query page using the Page Management functionality.

As noted in Step 4, the anchor tags in the list of research areas include an `href` attribute that takes this format:

```
href="${urls.base}/deptResearchAreas?deptURI=${individual.uri}&raURI=${resultRow["ra"]}"
```

When creating the SPARQL query page in Page Management, as shown in the illustration below, we set the "Pretty URL" field to `/deptResearchAreas`. This portion of the `href` attribute, then, is not the name of an actual servlet but it effectively functions as one, and it is also associated with the template that we also define in Page Management: individual-dept-res-area-details.ftl. When a user clicks on one of the listed research areas, this is the template that the application will load.

Note the SPARQL query that is defined in the illustration below. It uses as variables the same parameters that are part of the `href` above: `deptURI` and `raURI`. Like the `?individualURI` discussed in Step 3, the values of these two parameters will become the values of the corresponding variables in the SPARQL query.



Now that the SPARQL query page has been created in Page Management, we still need to create the individual-dept-res-area-details.ftl template. Just as in Step 4, where we used the "save to" variable to access the query results in the individual-dept-research-areas.ftl template, we now use the variable defined in the "Variable Name" field (above) to access the results of that SPARQL query. Here is the content of the new template:

```
<#if deptResearchAreas?has_content>
    <section id="pageList">
        <#list deptResearchAreas as firstRow>
            <div class="tab">
                <h2>${firstRow["raLabel"]}</h2>
                <p>
                    Here are the faculty members in the ${firstRow["deptLabel"]} department with
an interest in this research area.
                </p>
            </div>
            <#break>
        </#list>
    </section>
    <section id="deptResearchAreas">
        <ul role="list" class="deptDetailsList">
            <#list deptResearchAreas as resultRow>
                <li class="deptDetailsListItem">
                    <a href="${urls.base}/individual${resultRow["person"]?substring(resultRow["p
erson"]?last_index_of("/"))}"
                    title="faculty name">
                        ${resultRow["personLabel"]}
                    </a>
                </li>
            </#list>
        </ul>
    </section>
</#if>
```

Once again we use an <#if> statement to check for results. But this time we use the <#list> function twice: once to retrieve just the first row, which is used to provide a heading and some introductory text; and a second time to list all of the faculty members with an interest in the selected research area.

3.8 Enhancing Freemarker templates with DataGetters

Write SPARQL queries to add information to any Freemarker template.

3.8.1 Overview

Since VIVO 1.6, it is possible for a Freemarker template to display data that is not normally provided to it.

You can create an RDF file that describes a custom `DataGetter` object, and associates it with the desired template. Each time that template is used, the `DataGetter` will be executed, and the data will be stored in a variable, so the template can display it.

This does not require changes to the Java code. You create the RDF file in your VIVO distribution directory and modify the template in your theme.

3.8.2 An example

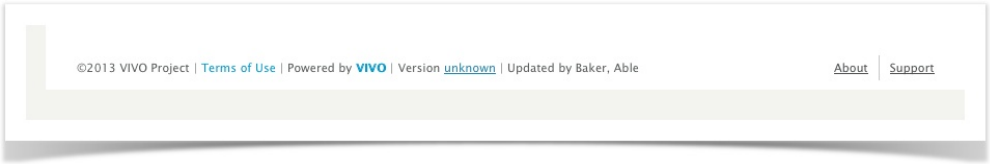
Let's assume that we need to display information about the most recent data ingest operation. We want to display the name of the Person who supervised the ingest. We would like to display this on every page.

As part of the ingest process, we can load statements like this into the data model:

```
<http://vivo.mydomain.edu/individual/n5242>
    <http://vivo.mydomain.edu/individual/isMostRecentUpdater>
    "true" .
<http://vivo.mydomain.edu/individual/n5242>
    <http://www.w3.org/2000/01/rdf-schema#label>
    "Baker, Able" .
```

We would like for VIVO to display the name of this individual on every page, so the footer will change from this:

to this:



3.8.3 Creating the DataGetter

VIVO allows you to define and use `DataGetter` objects in several contexts. `DataGetters` come in many flavors, but the most commonly used is the `SparqlQueryDataGetter`, which lets you define a SPARQL query, and store the results of that query for your Freemarker template to display.

By adding statements to your data model, you can define a `SparqlQueryDataGetter` object, and associate it with a Freemarker template. Here is the definition that is used in this example:

```
@prefix display: <http://vitro.mannlib.cornell.edu/ontologies/display/1.1#> .
@prefix rdf: <http://www.w3.org/1999/02/22-rdf-syntax-ns#> .
@prefix rdfs: <http://www.w3.org/2000/01/rdf-schema#> .
<freemarker:footer.ftl> display:hasDataGetter display:updatedInfoDataGetter .
display:updatedInfoDataGetter
    a <java:edu.cornell.mannlib.vitro.webapp.utils.dataGetter.SparqlQueryDataGetter> ;
    display:saveToVar "updatedInfo" ;
    display:query """
    PREFIX rdfs: <http://www.w3.org/2000/01/rdf-schema#>
    PREFIX local: <http://vivo.mydomain.edu/individual/>
    SELECT (str(?rawLabel) AS ?updater)
    WHERE {
        ?uri local:isMostRecentUpdater ?o ;
             rdfs:label ?rawLabel .
    }
    LIMIT 1
     """ .
```

These statements can be added to your data model in any of several ways. For this example, I stored these lines in a file called `data_getter_for_example.n3` and placed it in the VIVO distribution directory under `rdf/display/everytime`. Files placed in this directory are loaded when VIVO starts, but are not persisted when VIVO stops. This allows you to edit or remove the file without leaving residual statements in your data model.

Notice that:

- The first statement says that the Freemarker template `footer.ftl` has a `DataGetter`, defined in subsequent lines.
- The definition of the `DataGetter` states:
    - the type of the data getter,
    - the SPARQL query that will be executed
    - the Freemarker variable that will hold the results.

The results of the SPARQL query are stored in a Freemarker variable, in this case `updatedInfo`. The variable will contain a Sequence of Hashes, where each Hash represents one line of the SPARQL result. Within each Hash, result values are specified as key/value pairs.

For more information on Sequences and Hashes, consult the Freemarker manual:

- Retrieving data from a Sequence
- Retrieving data from a Hash

3.8.4 Modifying the template

Here is the standard `footer.ftl` template, as used in VIVO 1.6:

```
1  <#-- $This file is distributed under the terms of the license in /doc/license.txt$ -->
2
3  </div> <!-- #wrapper-content -->
4
5  <footer role="contentinfo">
6      <p class="copyright">
7          <#if copyright?>
8              <small>&copy;${copyright.year?c}
9              <#if copyright.url?>
10                 <a href="${copyright.url}" title="copyright">${copyright.text}</a>
11             <#else>
12                 ${copyright.text}
13             </#if>
14              | <a class="terms" href="${urls.termsOfUse}" title="terms of use">Terms of Use</a></small> |
15         </#if>
16         Powered by <a class="powered-by-vivo" href="http://vivoweb.org" target="_blank" title="powered by VIVO"><strong>VIVO</strong></a>
17         <#if user.hasRevisionInfoAccess>
18              | Version <a href="${version.moreInfoUrl}" title="version">${version.label}</a>
19         </#if>
20     </p>
21
22     <nav role="navigation">
23         <ul id="footer-nav" role="list">
24             <li role="listitem"><a href="${urls.about}" title="about">About</a></li>
25             <#if urls.contact?>
26                 <li role="listitem"><a href="${urls.contact}" title="contact us">Contact Us</a></li>
27             </#if>
28             <li role="listitem"><a href="http://www.vivoweb.org/support" target="blank" title="support">Support</a></li>
29         </ul>
30     </nav>
31 </footer>
32
33 <#include "scripts.ftl">
```

Insert these lines between lines 19 and 20:

```
        <#if (updatedInfo?first.updater)??>
            | Updated by ${updatedInfo?first.updater}
        </#if>
```

The SPARQL result is obtained and stored into the Freemarker variable `updatedInfo` each time the `footer.ftl` template is loaded for display. The name we want is in the first row of the SPARQL result, keyed to the name `updater`.

3.8.5 Summary

Enhancing Freemarker templates is one more way to use the VIVO `DataGetter` mechanism. When you associate a `DataGetter` with a Freemarker template, that `DataGetter` will be run each time the template is invoked. This is true whether the template is specified by the controller, or included in another template. You can modify the template to display the data from the `DataGetter`, but it is prudent to include an `<#if>` tag, so your template won't fail if the data is not found.

3.9 Custom List View Configurations

On a profile page, expand the data that is displayed for object and data properties.

3.9.1 Introduction

Custom list views provide a way to expand the data that is displayed for object and data properties. For example, with the default list view the hasPresenterRole object property would only display the rdfs:label of the role individual; but with a custom list view, the "presentations" view includes not only the role but also the title of the presentation, the name of the conference where the presentation was given and the date the presentation was given. This wiki page provides guidelines for developing custom list views as well as an example of a custom list view.

3.9.2 List View Configuration Guidelines

The following our guidelines for developing custom list views.

Registering the List View

A custom list view is associated with an object property in the RDF files in the directory `/vivo/rdf/display/everytime`. To register a list view, create a new `.rdf` or `.n3` file in that directory. The file must be well-formed RDF/XML or N3.

Here is an example of registering a new association in a file named `newListViews.n3`:

```
<http://vivoweb.org/ontology/core#authorInAuthorship>
<http://vitro.mannlib.cornell.edu/ontologies/display/1.1#listViewConfigFile>
"listViewConfig-authorInAuthorship.xml" .
```

With this triple the `authorInAuthorship` object property is associated with a list view configuration that is defined in a file named `listViewConfig-authorInAuthorship.xml`.

Place the N3 file containing this triple (or the well-formed RDF/XML file) in the `/vivo/rdf/display/everytime` directory, redeploy VIVO and restart tomcat to put the new custom list view in effect.

Required Elements

The list view configuration file requires three elements:

1. list-view-config: this is the root element that contains the other elements
2. query-select: this defines the SPARQL query used to retrieve data
3. template: this holds the name of the Freemarker template file used to display a single property statement

Optional Elements

The list-view-config root element can also contain two optional elements:

1. query-construct: one or more construct queries used to construct a model that the select query is run against
2. postprocessor: a Java class that postprocesses the data retrieved from the query before sending it to the template. If no post-processor is specified, the default post-processor will be invoked.

Construct Queries

Because SPARQL queries with multiple OPTIONAL clauses are converted to highly inefficient SQL by the Jena API, one or more construct queries should be included to improve query performance. They are used to construct a model significantly smaller than the entire dataset that the select query can be run against with reasonable performance.

The construct queries themselves should not contain multiple OPTIONAL clauses, to prevent the same type of inefficiency. Instead, use multiple construct queries to construct a model that includes all the necessary data.

In the absence of any construct queries, the select query is run against the entire dataset. If your select query does not involve a lot of OPTIONAL clauses, you do not need to include construct queries.

The construct queries must be designed to collect all the data that the select query will request. They can be flexibly constructed to contain more data than is currently selected, to allow for possible future expansion of the SELECT and to simplify the WHERE clause. For example, one of the construct queries for core:hasRole includes:

```
CONSTRUCT {
    ?role ?roleProperty ?roleValue .
    ...
} WHERE {
    ?role ?roleProperty ?roleValue .
    ...
}
```

That is, it includes all the properties of the role, rather than just those currently selected by the select query.

The ordering of the construct queries is not significant.

The Select Query

General select query requirements

Use a SELECT DISTINCT clause rather than a simple SELECT. There can still be cases where the same individual is retrieved more than once, if there are multiple solutions to the other assertions, but DISTINCT provides a start at uniqueness.

The WHERE clause must contain a statement ?subject ?property ?object, with the variables ?subject and ?property named as such. For a default list view, the ?object variable must also be named as such. For a custom list view, the object can be given any name, but it must be included in the SELECT terms retrieved by the query. This is the statement that will be edited from the edit links.

Data which is required in public view, optional when editing

Incomplete data can result in a missing linked individual or other critical data (such as a URL or anchor text on a link object). When the user has editing privileges on the page, these statements are displayed so that the user can edit them and provide the missing data. They should be hidden from non-editors. Follow these steps in the select query to ensure this behavior:

- Enclose the clause for the linked individual in an OPTIONAL block.
- Select the object's localname using the ARQ localname function, so that the template can display the local name in the absence of the linked individual. Alternatively, this can be retrieved in the template using the localname(uri) method.

- Require the optional information in the public view by adding a filter clause which ensures that the variable has been bound, inside tag <critical-data-required>. For example:

```
OPTIONAL { ?authorship core:linkedInformationResource ?infoResource }
```

- This statement is optional because when editing we want to display an authorship that is missing a link to an information resource. Then add:

```
<critical-data-required>
    FILTER ( bound(?infoResource) )
</critical-data-required>
```

- The Java code will preprocess the query to remove the <critical-data-required> tag, either retaining its text content (in public view) or removing the content (when editing), so that the appropriate query is executed.

Collated vs. uncollated queries

The query should contain <collated> elements, which are used when the property is collated. For uncollated queries, the fragments are removed by a query preprocessor. Since any ontology property can be collated in the Ontology Editor, all queries should contain the following <collated> elements:

- A ?subclass variable, named as such, in the SELECT clause. If the ?subclass variable is missing, the property will be displayed without collation.

```
SELECT DISTINCT <collated> ?subclass </collated> ...
```

- ?subclass must be the first term in the ORDER BY clause.

```
ORDER BY <collated> ?subclass </collated> ...
```

- Include the following in the WHERE clause, substituting in the relevant variables for ?infoResource and core:InformationResource:

```
<collated>
    OPTIONAL { ?infoResource a ?subclass
               ?subclass rdfs:subClassOf core:InformationResource .
    }
</collated>
```

Postprocessing removes all but the most specific subclass value from the query result set.

Alternatively (and preferably):

```
<collated>
    OPTIONAL { ?infoResource vitro:mostSpecificType ?subclass
               ?subclass rdfs:subClassOf core:InformationResource .
    }
</collated>
```

Automatic postprocessing to filter out all but the most specific subclass will be removed in favor of this implementation in the future.

Both collated and uncollated versions of the query should be tested, since the collation value is user-configurable via the ontology editor.

Datetimes in the query

To retrieve a datetime interval, use the following fragment, substituting the appropriate variable for ?edTraining:

```
OPTIONAL {
    ?edTraining core:dateTimeInterval ?dateTimeInterval
    OPTIONAL { ?dateTimeInterval core:start ?dateTimeStartValue .
               ?dateTimeStartValue core:dateTime ?dateTimeStart
    }
    OPTIONAL { ?dateTimeInterval core:end ?dateTimeEndValue .
               ?dateTimeEndValue core:dateTime ?dateTimeEnd
    }
}
```

The variables ?dateTimeStart and ?dateTimeEnd are included in the SELECT clause.

Many properties that retrieve dates order by end datetime descending (most recent first). In this case, a post-processor must apply to sort null values at the top rather than the bottom of the list, which is the ordering returned by the SPARQL ORDER BY clause in the case of nulls in a descending order. In that case , the variable names must be exactly as shown to allow the post-processor to do its work.

The Template

To ensure that values set in the template on one iteration do not bleed into the next statement:

- The template should consist of a macro that controls the display, and a single line that invokes the macro.
- Variables defined inside the macro should be defined with <#local> rather than <#assign>.

To allow for a missing linked individual, the template should include code such as:

```
<#local linkedIndividual>
    <#if statement.org??>
        <a href="${url(statement.org)}">${statement.orgName}</a>
    <#else>
        <#-- This shouldn't happen, but we must provide for it -->
        <a href="${url(statement.edTraining)}">${statement.edTrainingName}</a> (no linked
organization)
    </#if>
</#local>
```

The query must have been constructed to return orgName (see above under "General select query requirements"), or alternatively the template can use the localname function: ${localname(org)}.

If a variable is in an OPTIONAL clause in the query, the display of the value in the template should include the default value operator ! to prevent an error on null values.

3.9.3 List View Example

This example will walk through the custom list view for the core:researchAreaOf object property. This property is displayed on the profile page for research area individuals.

Associate the property with a list view

In this example we're using RDF/XML to associate the researchAreaOf object property (line 1) with a specific list view configuration (line 2):

```
<rdf:Description rdf:about="http://vivoweb.org/ontology/core#researchAreaOf">
    <display:listViewConfigFile rdf:datatype="http://www.w3.org/2001/XMLSchema#string">
listViewConfig-researchAreaOf.xml</display:listViewConfigFile>
</rdf:Description>
```

The list view configuration

The root <list-view-config> element in our listViewConfig-researchAreaOf.xml file contains the required < query-select> and <template> elements as well as two optional <query-construct> sections and an optional <postprocessor> element.

This is the <query-select> element:

```
    <query-select>
        PREFIX afn:  &lt;http://jena.hpl.hp.com/ARQ/function#&gt;
        PREFIX core: &lt;http://vivoweb.org/ontology/core#&gt;
        PREFIX rdfs: &lt;http://www.w3.org/2000/01/rdf-schema#&gt;
        PREFIX vitro: &lt;http://vitro.mannlib.cornell.edu/ns/vitro/0.7#&gt;
        PREFIX foaf: &lt;http://xmlns.com/foaf/0.1/&gt;
        SELECT DISTINCT
                       ?person
                       ?personName
                       ?posnLabel
                       ?orgLabel
                       ?type
                       ?personType
                       ?title
        WHERE {
                 ?subject ?property ?person .
                 ?person core:personInPosition ?position .
                 OPTIONAL { ?person rdfs:label ?personName }
                 OPTIONAL { ?person core:preferredTitle ?title }
                 OPTIONAL { ?person vitro:mostSpecificType ?personType .
                            ?personType rdfs:subClassOf foaf:Person
                 }
                 OPTIONAL { ?position rdfs:label ?posnLabel }
                 OPTIONAL { ?position core:positionInOrganization ?org .
                            ?org rdfs:label ?orgLabel
                 }
                 OPTIONAL { ?position core:hrJobTitle ?hrJobTitle }
                 OPTIONAL { ?position core:rank ?rank }
        }
        ORDER BY ?personName ?type
    </query-select>
```

Here is the first <query-construct> element:

```
    <query-construct>
        PREFIX rdfs: &lt;http://www.w3.org/2000/01/rdf-schema#&gt;
        PREFIX core: &lt;http://vivoweb.org/ontology/core#&gt;
        CONSTRUCT {
            ?subject ?property ?person .
            ?person core:personInPosition ?position .
            ?position rdfs:label ?positionLabel .
            ?position core:positionInOrganization ?org .
            ?org rdfs:label ?orgName .
            ?position core:hrJobTitle ?hrJobTitle
        } WHERE {
            {
                ?subject ?property ?person
            } UNION {
                ?subject ?property ?person .
                ?person core:personInPosition ?position
            } UNION {
                ?subject ?property ?person .
                ?person core:personInPosition ?position .
                ?position rdfs:label ?positionLabel
            } UNION {
                ?subject ?property ?person .
                ?person core:personInPosition ?position .
                ?position core:positionInOrganization ?org
            } UNION {
                ?subject ?property ?person .
                ?person core:personInPosition ?position .
                ?position core:positionInOrganization ?org .
                ?org rdfs:label ?orgName
            } UNION {
                ?subject ?property ?person .
                ?person core:personInPosition ?position .
                ?position core:hrJobTitle ?hrJobTitle
            }
        }
    </query-construct>
```

The second <query-construct> element:

```
    <query-construct>
        PREFIX rdfs: &lt;http://www.w3.org/2000/01/rdf-schema#&gt;
        PREFIX core: &lt;http://vivoweb.org/ontology/core#&gt;
        PREFIX foaf: &lt;http://xmlns.com/foaf/0.1/&gt;
        PREFIX vitro: &lt;http://vitro.mannlib.cornell.edu/ns/vitro/0.7#&gt;
        CONSTRUCT {
            ?subject ?property ?person .
            ?person rdfs:label ?label .
            ?person core:preferredTitle ?title .
            ?person vitro:mostSpecificType ?personType .
            ?personType rdfs:subClassOf foaf:Person
        } WHERE {
            {
                ?subject ?property ?person
            } UNION {
                ?subject ?property ?person .
                ?person rdfs:label ?label
            } UNION {
                ?subject ?property ?person .
                ?person core:preferredTitle ?title
            }  UNION {
                ?subject ?property ?person .
                ?person vitro:mostSpecificType ?personType .
                ?personType rdfs:subClassOf foaf:Person
            }
        }
    </query-construct>
```

Next is the required <template> element:

```
<template>propStatement-researchAreaOf.ftl</template>
```

And here is the <postprocessor> element:

```
<postprocessor>
edu.cornell.mannlib.vitro.webapp.web.templatemodels.individual.ResearchAreaOfPostProcessor</
postprocessor>
```

Note: the <postprocessor> is included here only to show the syntax. The actual listViewConfig-researchAreaOf.xml file in the VIVO code base does not use a custom post-processor.

The Freemarker Template

Finally, here are the contents of our Freemarker template, propStatement-researchAreaOf.ftl.

```
<#import "lib-sequence.ftl" as s>
<@showResearchers statement />
<#-- Use a macro to keep variable assignments local; otherwise the values carry over to the
     next statement -->
<#macro showResearchers statement>
    <#local linkedIndividual>
        <a href="${profileUrl(statement.uri("person"))}" title="${i18n().person_name}">${
statement.personName}</a>
    </#local>
    <#if statement.title?has_content >
        <#local posnTitle = statement.title>
    <#else>
        <#local posnTitle = statement.posnLabel!statement.personType>
    </#if>
    <@s.join [ linkedIndividual, posnTitle, statement.orgLabel!"" ] /> ${statement.type!}
</#macro>
```

3.10 Creating short views of individuals

Short views of individuals appear in search results, on index pages, and in browse pages. But should the short view of an Academic Department contain the same information as the short view of a Faculty Member?

3.10.1 Overview

What does it do?

Custom short views are used in three different contexts within VIVO, to give you more control over how an individual is displayed in that context.

You can configure VIVO to display different classes of individuals in different ways. As an example, you might choose to display a Faculty Member in a grey color if she has no current appointments.

Custom short views will frequently be different in the three different contexts in which they are available. For example, you might want to show the image of a Person on a search result, but you might not want to display that image in the Person index pages.

How is it created?

A short view is defined by two elements. First there will be a group of RDF statements in this file:

`vitro/webapp/web/WEB-INF/resources/shortview_config.n3`

In a VIVO release, this file is empty (except for a few comments), and the default short views are used in all cases. You will add RDF to this file as you define your custom short views. The RDF statements will name the class of Individual, the Freemarker template, and any SPARQL queries that are used to get the data you need to display.

The other thing you will need is the Freemarker template itself, to render your custom view.

An example of some custom short views can be found in this directory:

`vivo/utilities/acceptance-tests/suites/ShortViews/`

The directory contains a copy of `shortview_config.n3`, and some Freemarker templates. These files are essentially the same as the examples on this page.

3.10.2 Details

The class association

A statement associates a custom short view with a VIVO Class from the ontology. For example:

```
vivo:FacultyMember
    display:hasCustomView   mydomain:facultySearchView .
```

This means that the specified short view will be used for any Individual that has `vivo:FacultyMember` as a most specific class.

In the current implementation, only the most specific classes of an Individual are recognized by the short views. So if you want a custom short view to be used for all Person objects, you must define it on Person and on FacultyMember and on FacultyMemberEmeritus, etc. This will be addressed in a future release.

The customViewForIndividual definition

An object is given a URI and declared to be a `customViewForIndividual`

It may apply to one or more of the contexts: `SEARCH, INDEX`, or `BROWSE`.
It must have an associated template, and may have one or more associated DataGetters.

Here is an example:

```
mydomain:facultySearchView
    a                       display:customViewForIndividual ;
    display:appliesToContext "SEARCH" ;
    display:hasTemplate      "view-search-faculty.ftl" ;
    display:hasDataGetter    mydomain:facultyDepartmentDG .
```

This custom view applies in the SEARCH context. It specifies a DataGetter, which will be invoked to find data each time this short view is rendered. It also specifies the Freemarker template that will render the view.

The SparqlQueryDataGetter definition

The DataGetter must also be defined in the RDF, like this:

```
mydomain:facultyDepartmentDG
    a                   datagetters:SparqlQueryDataGetter ;
    display:saveToVar   "details" ;
    display:query       """
       PREFIX rdfs: <http://www.w3.org/2000/01/rdf-schema#>
       PREFIX vivo: <http://vivoweb.org/ontology/core#>
       SELECT ?deptName
       WHERE {
          ?individualUri  vivo:hasMemberRole      ?membership .
          ?membership     vivo:roleContributesTo  ?deptUri .
          ?deptUri
             a           vivo:AcademicDepartment ;
             rdfs:label  ?deptName .
       }
       LIMIT 20
       """ .
```

Besides the type and the URI, this object specifies a SPARQL query, and the name of a Freemarker variable where the results of the query will be stored.

When the SPARQL query is executed, the value of `?individualUrl` will be bound to the actual URI of the Individual being displayed. The values returned from the query will be stored in an array of Freemarker Hash containers, with each one representing a row of the SPARQL query result. The Hash will contain the values returned by the query, keyed to the variable names used in the query.

When this array of Hash containers has been constructed, it is stored in the variable named by the DataGetter declaration; `details` in this case.

The DataGetter is optional in a custom short view. If no DataGetter is specified, then the Freemarker template will only have the standard set of data available to it.

Conversely, multiple DataGetters may be specified on a short view. If this is done, each DataGetter should assign to a different Freemarker variable, to avoid problems with overwriting data.

The Freemarker template

A default template exists for each of the short view contexts: `SEARCH`, `INDEX` and `BROWSE`.
If no custom short view is defined for an Individual, the default template will be used to render the Individual.

The custom template will likely be based on the default template for that context. For example, the default template for search results is called `view-search-default.ftl` and looks like this:

```
<#import "lib-vivo-properties.ftl" as p>
<a href="${individual.profileUrl}" title="individual name">${individual.name}</a>
<@p.displayTitle individual />
<p class="snippet">${individual.snippet}</p>
```

Our modified template is this:

```
<#import "lib-vivo-properties.ftl" as p>
<a href="${individual.profileUrl}" title="individual name">${individual.name}</a>
<@p.displayTitle individual />
<#if (details[0].deptName)?? >
    <span class="display-title">Member of ${details[0].deptName}</span>
</#if>
<p class="snippet">${individual.snippet}</p>
```

So, if a department name was found for this Faculty member, it will be displayed. If more than one was found, the remainder will be ignored, since the template only displays the first one.

The default template can be modified in the theme

Besides taking advantage of custom short views, the theme author may also choose to override the templates for the default short views. This would merely require creating a new template with the same name as the one being overridden, and putting this new template into the template directory of your theme.

3.10.3 Some examples

The scenario

When a FacultyMember appears in a short view, we would like to add the name of his/her department. This information isn't directly available, so we will need to obtain it from a SPARQL query.

This should work in all three contexts, `SEARCH`, `INDEX`, and `BROWSE`.

This will only apply to FacultyMembers. Other individuals will use the default short views.

If the FacultyMember is not a member of a department, the short view should just omit the name, without causing an error.

SEARCH example

The default template

The default short view for the SEARCH context looks like this:

And it produces results like this:

Specifying the custom short view

In the `shortview_config.n3` configuration file, create these structures:

```
vivo:FacultyMember
    display:hasCustomView   mydomain:facultySearchView .
mydomain:facultySearchView
    a                           display:customViewForIndividual ;
    display:appliesToContext    "SEARCH" ;
    display:hasTemplate         "view-search-faculty.ftl" ;
    display:hasDataGetter       mydomain:facultyDepartmentDG .
mydomain:facultyDepartmentDG
    a                   datagetters:SparqlQueryDataGetter ;
    display:saveToVar   "details" ;
    display:query       """
       PREFIX rdfs: <http://www.w3.org/2000/01/rdf-schema#>
       PREFIX vivo: <http://vivoweb.org/ontology/core#>
       SELECT ?deptName
       WHERE {
          ?individualUri  vivo:hasMemberRole      ?membership .
          ?membership     vivo:roleContributesTo  ?deptUri .
          ?deptUri
             a            vivo:AcademicDepartment ;
             rdfs:label   ?deptName .
       }
       LIMIT 20
       """ .
```

Create the template `view-search-faculty.ftl` to look like this:

```
<#import "lib-vivo-properties.ftl" as p>
<a href="${individual.profileUrl}" title="individual name">${individual.name}</a>
<@p.displayTitle individual />
<#if (details[0].deptName)?? >
    <span class="display-title">Member of ${details[0].deptName}</span>
</#if>
<p class="snippet">${individual.snippet}</p>
```

The new search results look like this:

INDEX example

The default template

The default short view for the `INDEX` context looks like this:

```
<#import "lib-properties.ftl" as p>
<a href="${individual.profileUrl}" title="name">${individual.name}</a>
<@p.mostSpecificTypes individual />
```

And it produces results like this:

Specifying the custom short view

In the `shortview_config.n3` configuration file, create these structures:

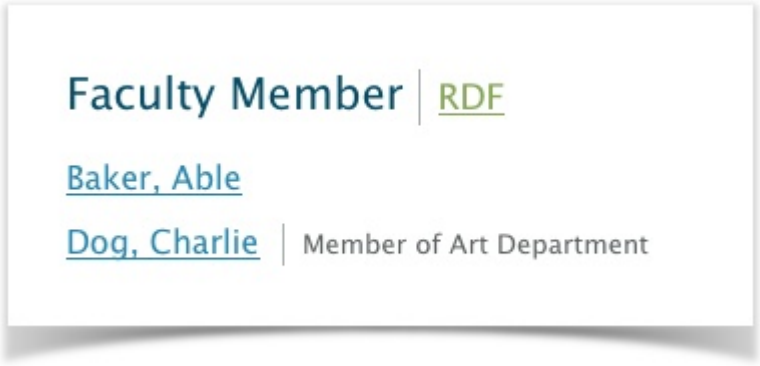
```
vivo:FacultyMember
    display:hasCustomView   mydomain:facultyIndexView .
mydomain:facultyIndexView
    a                          display:customViewForIndividual ;
    display:appliesToContext   "INDEX" ;
    display:hasTemplate        "view-index-faculty.ftl" ;
    display:hasDataGetter      mydomain:facultyDepartmentDG .
mydomain:facultyDepartmentDG
    a                   datagetters:SparqlQueryDataGetter ;
    display:saveToVar   "details" ;
    display:query       """
       PREFIX rdfs: <http://www.w3.org/2000/01/rdf-schema#>
       PREFIX vivo: <http://vivoweb.org/ontology/core#>
       SELECT ?deptName
       WHERE {
          ?individualUri  vivo:hasMemberRole      ?membership .
          ?membership     vivo:roleContributesTo  ?deptUri .
          ?deptUri
             a            vivo:AcademicDepartment ;
             rdfs:label   ?deptName .
       }
       LIMIT 20
       """ .
```

Note that the DataGetter is the same as in the previous example. If two custom short views want to use the same DataGetter, there is no need to code it twice. If both of these examples are tried at the same time, the two custom short views would refer to the same DataGetter.

Create the template `view-index-faculty.ftl` to look like this:

```
<#import "lib-vivo-properties.ftl" as p>
<a href="${individual.profileUrl}" title="individual name">${individual.name}</a>
<@p.displayTitle individual />
<#if (details[0].deptName)?? >
    <span class="display-title">Member of ${details[0].deptName}</span>
</#if>
```

The new index looks like this:



BROWSE example

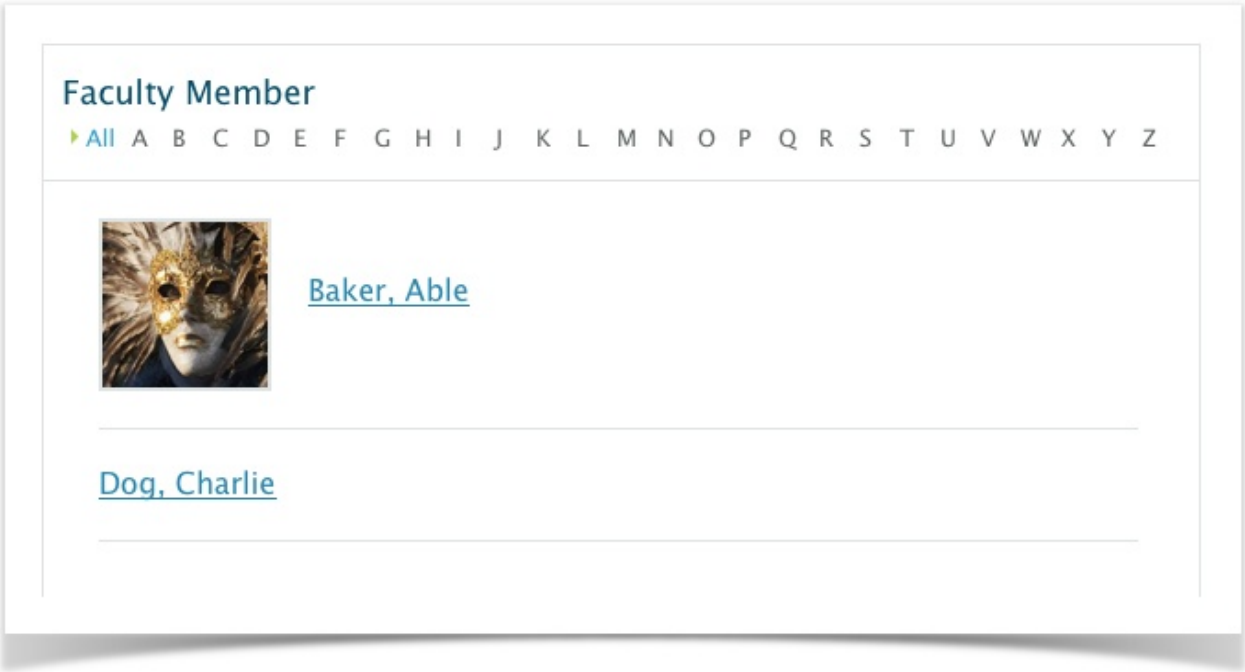
The default template

The default short view for the BROWSE context looks like this:

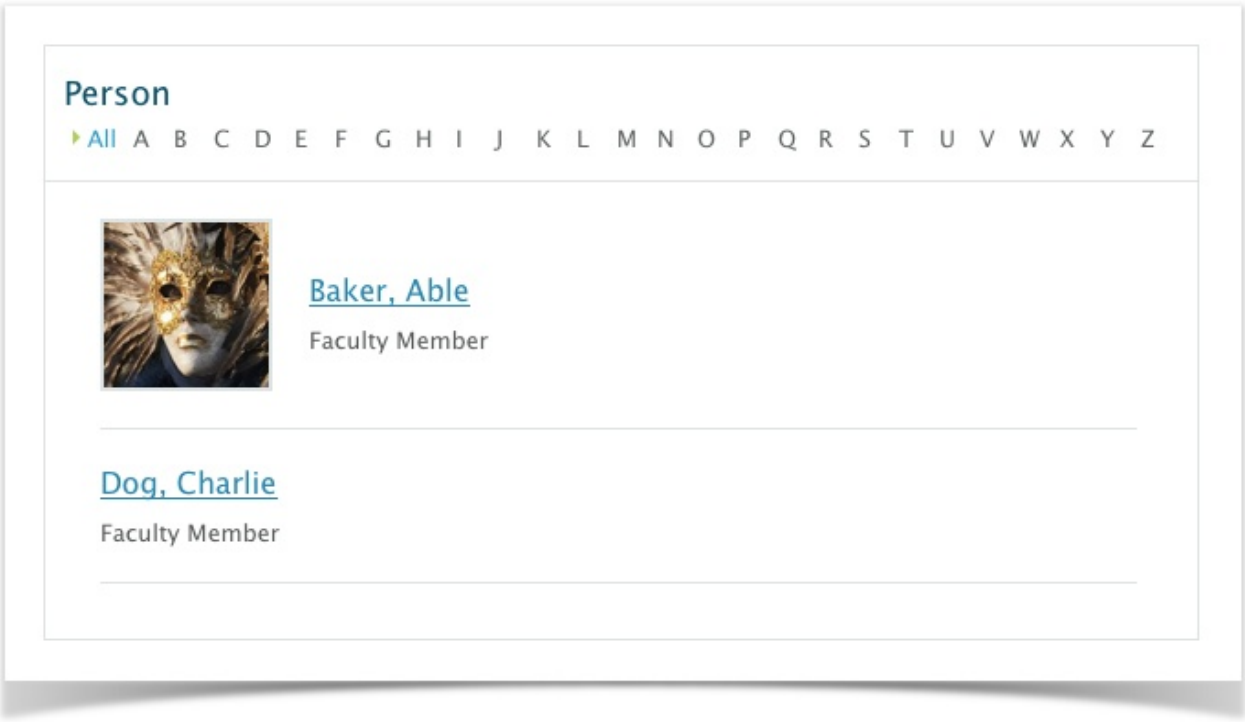
```
<#import "lib-properties.ftl" as p>
<li class="individual" role="listitem" role="navigation">
<#if (individual.thumbUrl)??>
    <img src="${individual.thumbUrl}" width="90" alt="${individual.name}" />
    <h1 class="thumb">
        <a href="${individual.profileUrl}" title="${i18n().view_profile_page_for}
              ${individual.name}}">${individual.name}</a>
    </h1>
<#else>
    <h1>
        <a href="${individual.profileUrl}" title="${i18n().view_profile_page_for}
              ${individual.name}}">${individual.name}</a>
    </h1>
</#if>
<#assign cleanTypes =
    'edu.cornell.mannlib.vitro.webapp.web.TemplateUtils$DropFromSequence'?new()(
individual.mostSpecificTypes, vclass) />
<#if cleanTypes?size == 1>
    <span class="title">${cleanTypes[0]}</span>
<#elseif (cleanTypes?size > 1) >
    <span class="title">
        <ul>
            <#list cleanTypes as type>
            <li>${type}</li>
            </#list>
        </ul>
    </span>
</#if>
</li>
```

Notice that it contains some conditional logic, producing different results depending on whether there is an image file associated with the Individual, or whether the type being browsed is the most specific type for the individual.

The default template produces results like this:



Or this:



Specifying the custom short view

In the `shortview_config.n3` configuration file, create these structures:

```
vivo:FacultyMember
    display:hasCustomView   mydomain:facultyBrowseView .
mydomain:facultyBrowseView
    a                         display:customViewForIndividual ;
    display:appliesToContext   "BROWSE" ;
    display:hasTemplate        "view-browse-faculty.ftl" ;
    display:hasDataGetter      mydomain:facultyDepartmentDG ;
    display:hasDataGetter      mydomain:facultyPreferredTitleDG .
mydomain:facultyDepartmentDG
    a                   datagetters:SparqlQueryDataGetter ;
    display:saveToVar   "details" ;
    display:query       """
       PREFIX rdfs: <http://www.w3.org/2000/01/rdf-schema#>
       PREFIX vivo: <http://vivoweb.org/ontology/core#>
       SELECT ?deptName
       WHERE {
          ?individualUri  vivo:hasMemberRole      ?membership .
          ?membership     vivo:roleContributesTo  ?deptUri .
          ?deptUri
             a            vivo:AcademicDepartment ;
             rdfs:label   ?deptName .
       }
       LIMIT 20
       """ .
mydomain:facultyPreferredTitleDG
    a                  datagetters:SparqlQueryDataGetter ;
    display:saveToVar  "extra" ;
    display:query      """
          PREFIX rdfs: <http://www.w3.org/2000/01/rdf-schema#>
       PREFIX vivo: <http://vivoweb.org/ontology/core#>
       SELECT ?pt
       WHERE {
          ?individualUri <http://vivoweb.org/ontology/core#preferredTitle> ?pt
       }
       LIMIT 1
       """ .
```

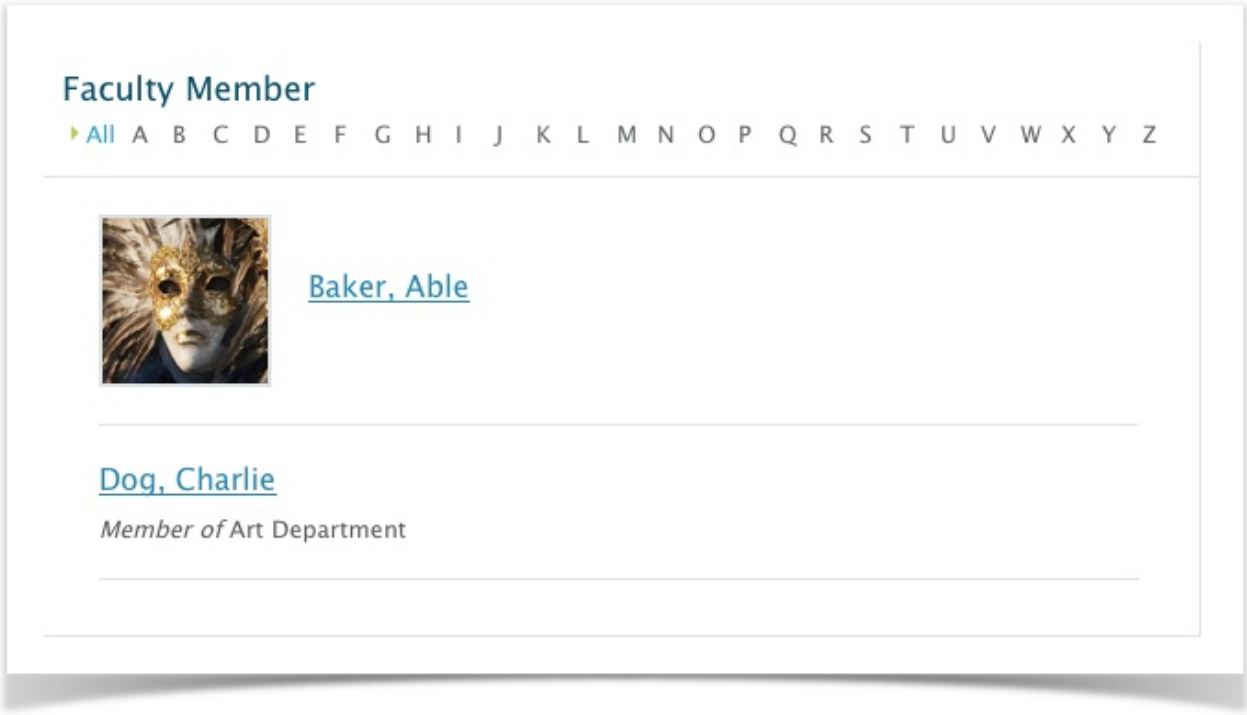
Note that the first DataGetter is the same as in the previous examples. If two custom short views want to use the same DataGetter, there is no need to code it twice. If two or more of these examples are tried at the same time, the short views would each refer to the same DataGetter.

Also note that this short view uses two DataGetters. One stores its results in "details" and the other stores its results in "extra", so the freemarker template will have access to both sets of results.

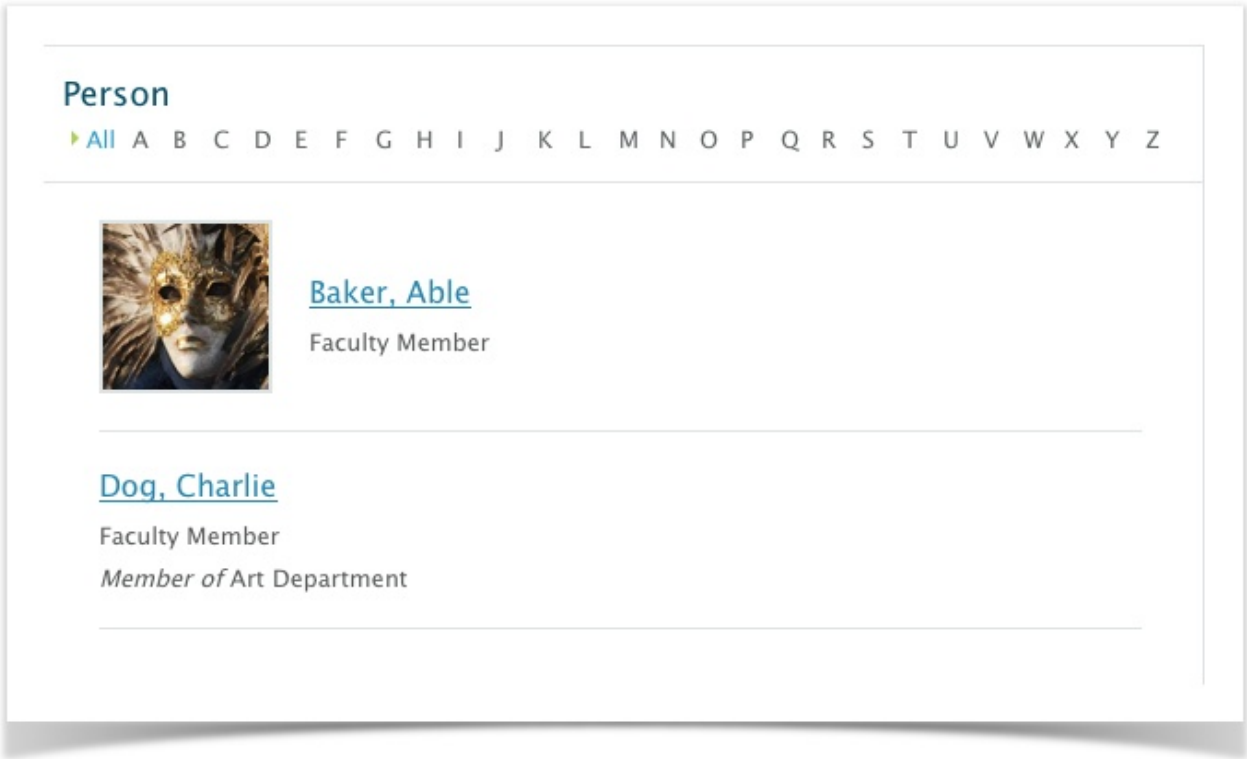
Create the template `view-browse-faculty.ftl` to look like this:

```
<#import "lib-properties.ftl" as p>
<li class="individual" role="listitem" role="navigation">
<#if (individual.thumbUrl)??>
    <img src="${individual.thumbUrl}" width="90" alt="${individual.name}" />
    <h1 class="thumb">
        <a href="${individual.profileUrl}" title="View the profile page for
              ${individual.name}}">${individual.name}</a>
    </h1>
<#else>
    <h1>
        <a href="${individual.profileUrl}" title="View the profile page for
              ${individual.name}}">${individual.name}</a>
    </h1>
</#if>
<#if (extra[0].pt)?? >
    <span class="title">${extra[0].pt}</span>
<#else>
    <#assign cleanTypes =
       'edu.cornell.mannlib.vitro.webapp.web.TemplateUtils$DropFromSequence'?new()(
individual.mostSpecificTypes, vclass) />
    <#if cleanTypes?size == 1>
        <span class="title">${cleanTypes[0]}</span>
    <#elseif (cleanTypes?size > 1) >
        <span class="title">
            <ul>
                <#list cleanTypes as type>
                <li>${type}</li>
                </#list>
            </ul>
        </span>
    </#if>
</#if>
<#if (details[0].deptName)?? >
    <span class="title"><em>Member of</em> ${details[0].deptName}</span>
</#if>
</li>
```

The new browse results look like this:


Faculty Member
All A B C D E F G H I J K L M N O P Q R S T U V W X Y Z
Baker, Able
Dog, Charlie
Member of Art Department

Or this:


Person
All A B C D E F G H I J K L M N O P Q R S T U V W X Y Z
Baker, Able
Faculty Member
Dog, Charlie
Faculty Member
Member of Art Department

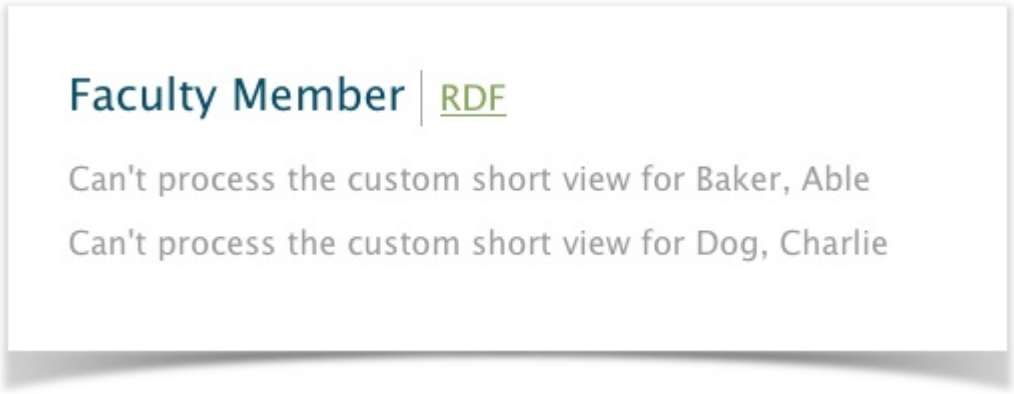
3.10.4 Troubleshooting

Errors in the template?

If the freemarker template for a short view contains syntax errors, the request will not throw an exception, which will be written to the VIVO log (`vivo.all.log`). The page will still render, but with an error message in place of the intended short view.

For example, in search results:

In an index page:



In browse results:

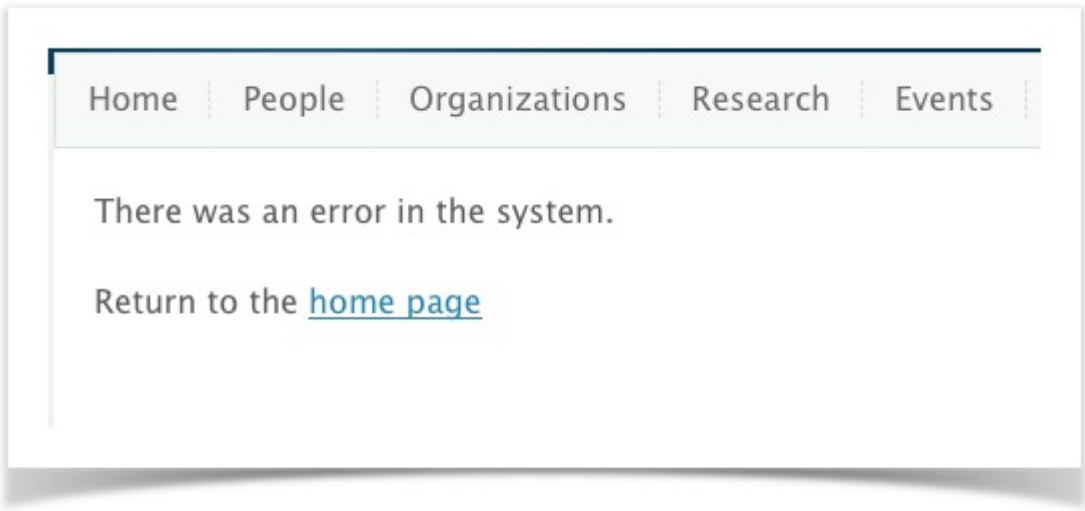
Errors in the Query?

If the SPARQL query defined in the configuration file contains syntax errors, the log will contain a stack trace for the exception. The information in the exception may be cryptic, but will at least tell you where the error is located within the query.

For example:

```
2012-10-23 17:25:26,499 ERROR [FreemarkerHttpServlet] com.hp.hpl.jena.query.QueryParseException:
    Encountered " <VAR1> "?individualUri "" at line 6, column 1.
    Was expecting:
    "{" ...
```

The page will not render properly. Instead it will show a standard error screen:



Errors in the config file?

Other errors in the configuration file may give less obvious results. For example, if your customView object calls for a data getter that does not exist, the page will attempt to render without that data. If the data from that data getter is optional, you will see no error indicator except for a message in `vivo.all.log`

3.10.5 Notes

Waiting for the Application and Display Ontology

This implementation of short views is intended to be temporary, pending the implementation of the Application and Display Ontology (A&DO).

Much of the RDF that is entered in the configuration file (`shortview_config.n3`) should be replaced by triples in the A&DO. It's not clear where the SPARQL queries will be specified.

Classes are not inferred

Short views are applied based on the most-specific classes of the Individual. No inference is done when trying to find applicable views. So if an Individual has a type of FacultyMember, then a short view that applies to Person will not be used. Even though the Indvidual should also have a type of Person, it will not be among the most specific types for the Individual, and so does not apply. If you want a short view to apply to all sub-classes of Person, you must explicitly list each of these sub-classes in `shortview_config.n3`

This is expected to change when the Application and Display Ontology is used.

More than one applicable short view

In theory, it is possible that an Individual may qualify for two short views simultaneously. An Individual could have two most specific types (say FacultyMember and ExemptEmployee), and both of those types might have configured short views. Or, in the degenerate case, there might be multiple short views configured for a single type.

In these cases, one of the applicable short views will be arbitrarily selected.

Hard-coded BROWSE view for People

In the `BROWSE` context, if no short view is found for a given class URI, but that class is included in the People classgroup, a hard-coded short view is applied. This is to maintain compatibility with previous versions.

It is hoped that the Application and Display Ontology will be expressive enough to configure this behavior within the standard mechanism. Until then, it is coded into the class `edu.cornell.mannlib.vitro.webapp.services.shortview.FakeApplicationOntologyServi`

3.11 Creating custom entry forms

A custom form allows your users to edit complex data structures on a single page, instead of editing one triple at a time.

3.11.1 Overview

Custom entry forms allow VIVO to transcend the general-purpose, utilitarian editing scheme of Vitro. Without custom entry forms, VIVO users must edit each RDF triple individually. With a custom entry form, users can edit a complex data structure on a single page.

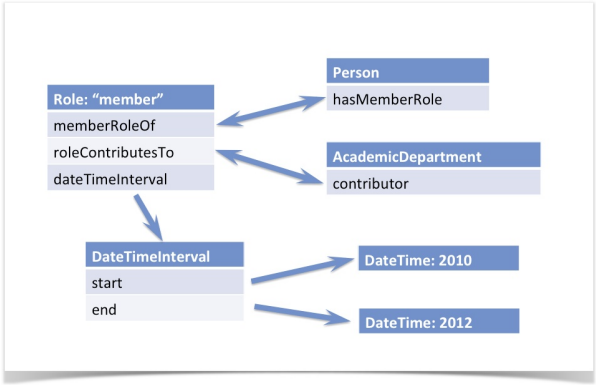
VIVO is distributed with a dozens of custom entry form generators. You may want to modify these form generators, or add more of your own.

An example

Say you wish to establish that a particular person is a member of a particular academic department. This relationship can be expressed in a single RDF statement.

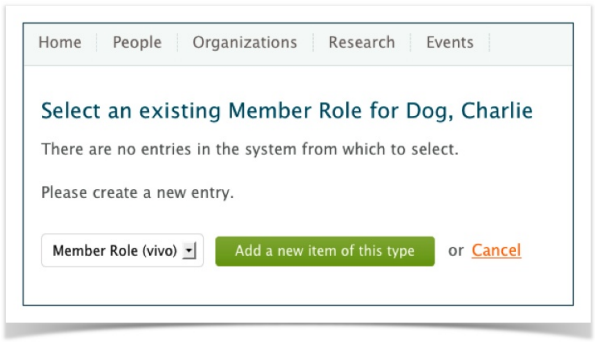
But what if the academic department doesn't exist in VIVO yet? You will want to create that department, and assign a name to it. You may also want to record the role that your person plays in that department, when their membership began, and when it ended (if it is not ongoing).

The data structure for this information looks something like this:

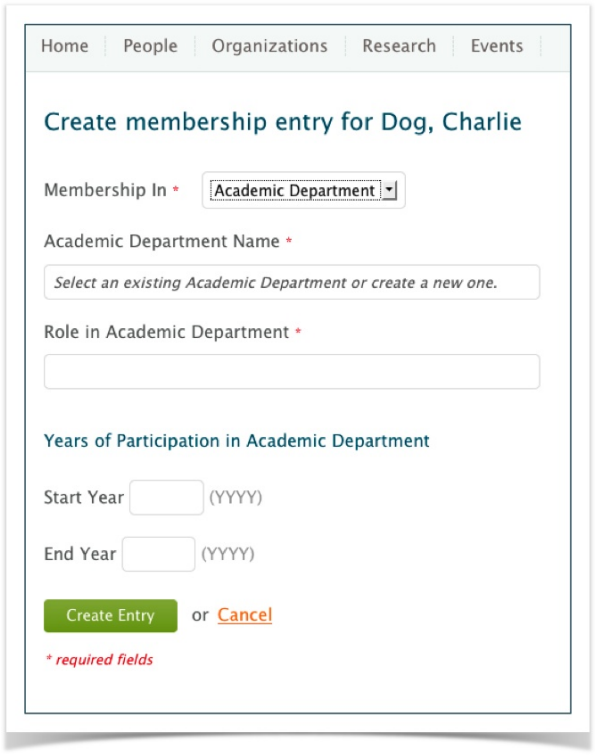


Note that this reflects the VIVO 1.5 Ontology. The data structures have changed significantly in VIVO 1.6 and beyond.

Without a custom entry form, you would need to record each piece of data individually.



VIVO includes a custom form generator for this relationship. The custom entry form looks like this:



How is it created?

The creation of custom entry forms is an arcane and eldritch art, for which little documentation is available.

Each form requires a Java class known as a `EditConfigurationGenerator`. The generator describes the data structure being created, lists the SPARQL queries used, and includes a reference to the Freemarker template that will render the form.

You can start by examining the existing generators in this directory `[VIVO]/src/edu/cornell/mannlib/vitro/webapp/edit/n3editing/configuration/generators`

and the Freemarker templates found here `[VIVO]/productMods/templates/freemarker/edit/forms`

There is also a short page of technical description called Implementing custom forms using N3 editing.

—

3.12 Using OpenSocial Gadgets

Create JavaScript-based gadgets to display on VIVO pages, or use existing gadgets from the ORNG library.

3.12.1 Overview

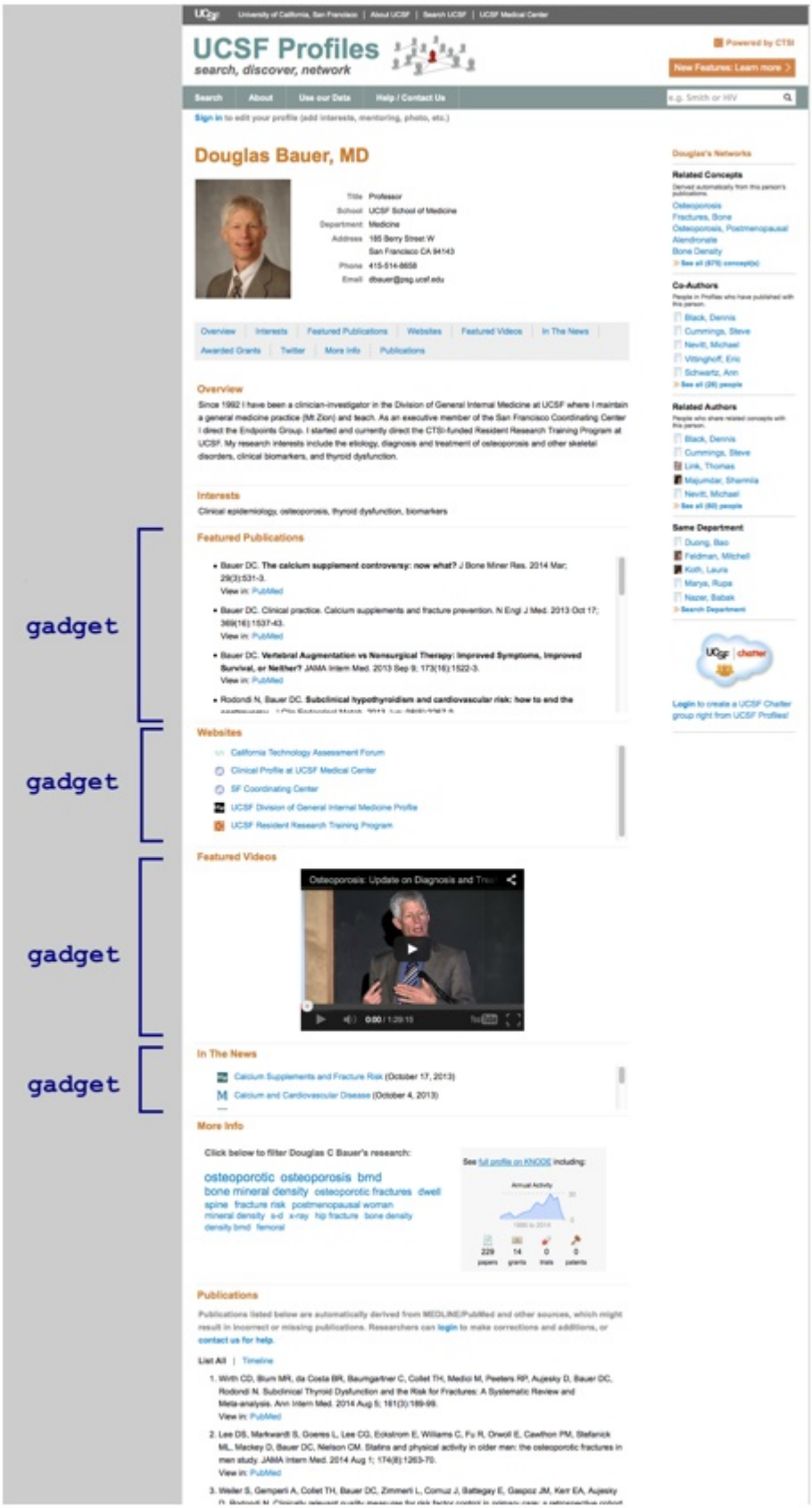
What can you do?

Your site administrators can configure a collection of "gadgets" for your VIVO installation. From that collection, each faculty member can decide which gadgets he will show on his profile page, and how they should be configured.

Perhaps it would be better to describe the gadgets as "page sections", because you can use CSS styling to make the gadget seamlessly match your theme. The result is profile pages that still look unified, but are at least partially configurable by the individual faculty member.

An example

Here is a portion of a profile page from UCSF Profiles. Each gadget is there because the page owner selected it and configured it.


gadget
gadget
gadget
gadget
UCSF Profiles
search, discover, network
Douglas Bauer, MD
Overview
Interests
Featured Publications
Websites
Featured Videos
In The News
More Info
Publications

OpenSocial

The OpenSocial standard was developed to make it easy for developers to add functionality to social networking systems like Google and MySpace. OpenSocial has lost popularity in social networking, but is becoming more favored in enterprise systems.

The Clinical and Translational Science Institute at UCSF created a project to host OpenSocial gadgets in the Harvard Profiles system. In keeping with the cross-platform origins of OpenSocial, the CTSI team decided to adapt their gadgets for use in VIVO as well.

ORNG

Social networking systems provide very little information about their participants. The group at CTSI wanted to combine the display tools of OpenSocial with the data structure of VIVO RDF. They accomplished this through an extension to the standard, which they called Open Research Networking Gadgets, or ORNG.

3.12.2 Adding gadgets to VIVO

The gadgets used at UCSF are provided in a library. Some are written specifically for UCSF, or specifically for the Harvard Profiles platform. However, many are available for use in VIVO.

You can also create your own gadgets. The gadgets are written in JavaScript, and you can use the existing gadgets as coding examples.

Under your control

The VIVO administrators select which gadgets will be available for the site. They also decide where the gadgets will appear on a profile page, if enabled.

Under control of your faculty

Each page owner may choose to enable individual gadgets for their page. A gadget may be written to accept settings that allow further configuration of its content and appearance.

3.12.3 Getting started

The VIVO Installation Instructions contain a section on how to add OpenSocial gadgets to a VIVO site. This will require some setup, and re-deploying VIVO. Once those steps are completed, your gadget library is configured by settings in a MySQL database table, and the gadget appearance is controlled by your Freemarker templates and CSS files.

For more information about Open Research Networking Gadgets, see the ORNG web site.

3.13 Excluding Classes from the Search

A VIVO/Vitro instance can be customized to exclude a class of individuals from the search index. All instances of a class can be excluded from the search index by adding a vitroDisplay:excludeClass property between vitroDisplay:SearchIndex and the URI of the class that you would like to exclude. This will have the effect of not displaying any individual with this class in search results, on the index page, in browse pages and as options for entry in some forms. The search exclusions are controlled by RDF statements in the display model.

Steps to create a new search exclusion:

1. Create a file with your new exclusion
2. In your deploy directories, place that file in either vivo/rdf/display/everytime or vitro/webapp/rdf/display /everytime
3. stop tomcat, deploy, and restart tomcat
4. login to VIVO as an admin and rebuild the search index.

Example on the contents of an RDF file to define exclusions:

```
@prefix vitroDisplay: <http://vitro.mannlib.cornell.edu/ontologies/display/1.1#> .
vitroDisplay:SearchIndex
    vitroDisplay:excludeClass <http://example.org/ns/ex/Hat> .
```

3.14 VIVO support for languages other than English

To a large extent, VIVO supports Spanish-language installations. Support for other languages is not difficult to implement.

3.14.1 Overview

Frequently, we are asked about a Spanish-language VIVO, or Portuguese, or a VIVO that supports multiple languages. The VIVO development team is working in this direction.

What do we mean by "Language Support"?

Multiple language support can mean many things. When a VIVO site supports a language other than English , that support includes:

- Text that is displayed in the VIVO pages.
    - For example, menus, selections, prompts, tool-tips and plain text.
- Terms in the Ontology, which are frequently displayed as links or section headings.
    - Labels and descriptions of properties and classes

- Text in the data model.
    - For example, if a book title is available in both French and English, a French-speaking user sees the French title. If a title is available only in English, it is displayed, without regard to the user's preference in languages.

Languages can be selected in a variety of ways, depending on the installation parameters:

- A VIVO installer can configure VIVO to use one of the supported languages.
- Different users may see different languages, depending on the settings in their web browser.
- Different users may select a language from a list of available languages.

Language support in VIVO is being implemented in phases:

- Phase 1 includes read-only support of public pages:
    - Pages that are visible to users who are not logged in.
    - Also includes support of some administrative pages.
    - This is currently available.
- Phase 2 will also provide read-write support of profile pages:
    - Users will be able to edit language-specific data in profile pages.
- Phase 3 will support administrative pages
    - Creating user accounts, manipulating RDF data and other administrative functions.
- Phase 4 will support "back-end" pages.
    - Used to edit the ontology, or to do low-level editing on individual entities.

Currently, VIVO language files are available for English and Spanish. If you need support for another language, please inquire of the VIVO Implementation mailing list, to see if another group is already developing the files you need.

If you have a particular use case, or if you would be willing to assist in translating, please contact the VIVO development team.

Contacting the VIVO development team

You can discuss this with the VIVO developers on one of the regular team calls, or by submitting a note to the contact page at vivoweb.org.

3.14.2 Adding a language to your VIVO site

Adding language files to VIVO

VIVO is distributed with English as the only supported language. Releases 1.6 and 1.7 also included a set of "pseudo-language" files, as a demonstration of how language support is implemented.

Additional language files are available in the Git repositories at https://github.com/vivo-project/Vitro-languages and https://github.com/vivo-project/VIVO-languages.

If the repository contains files for the language you want, in the VIVO release that you are using, you can just download those files and install them, as described in the Installing VIVO release 1.8.

Updating languages files for a newer release

If the language you want has been implemented for VIVO, but is not available for the release that you want, you can adapt the existing language files for the newer release.

Perhaps of more importance, you can submit these updated language files to the VIVO community for others to use.

You can find some notes on this process at Updating language files for the next release. You can also contact the VIVO development team, to discuss this with the developers.

Translating VIVO into your language

First, contact the VIVO development team: we would love to talk to you. We will tell you if anyone else is working on your language, and we will be happy to help with any questions you may have.

When you are ready to go ahead, you must determine the "locale" of your translation. Then you prepare translations of twenty-one files, as shown below.

The locale

Your locale is an internationally recognized code that specifies the language you choose, and the region where it is spoken. For example, the locale string `fr_CA` is used for French as spoken in Canada, and `es_MX` is used for Spanish as spoken in Mexico. Recognized codes for languages and regions can be found by a simple Google search. Here is a list of locales that are recognized by the Java programming language. You may also use this definitive list of languages and regions, maintained by the Internet Assigned Numbers Authority.

The locale code will appear in the name of each file that you create. In the files that contain RDF data, the locale code will also appear at the end of each line.

When the locale code appears in file names, it contains an underscore (`en_US`). When it appears inside RDF data files, it contains a hyphen (`en-US`).

The language files

You can get the Spanish or the English files from the VIVO and Vitro language repositories, to use as a template for your own files.

The example that follow assume that you are creating files for the Estonian language, as spoken in Estonia, with the locale `et_EE`.

Text strings (.properties)

These files contain about 1500 words and phrases that appear in the VIVO web pages. The page templates contain more than just text – they contain programming logic and display specifiers.

These words and phrases have been removed from the page templates, so no programming knowledge is required to translate them.

There is one file for phrases used in Vitro, the core around which VIVO is built, and one file for phrases that are specific to VIVO. In the example, these two files are both called `all_et_EE.properties`.

Example file names

```
[Vitro]/webapp/languages/et_EE/i18n/all_et_EE.properties
[VIVO]/languages/et_EE/themes/wilma/i18n/all_et_EE.properties
```

Example content

```
minimum_image_dimensions = Minimaalne pildi mõõdud: {0} x {1} pikslit
cropping_caption = Profiilifoto näeb alloleval pildil.
```

Freemarker Templates (.ftl)

Almost all of the text in the Freemarker templates is supplied by the text strings in the properties files. However, some Freemarker templates are essentially all text, and it seemed simpler to create a translation of the entire template. These include the `help` and `about` pages, the `Terms of Use` page, and the emails that are sent to new VIVO users.

Example file names

```
[Vitro]/webapp/languages/et_EE/templates/freemarker/search-help_et_EE.ftl
[Vitro]/webapp/languages/et_EE/templates/freemarker/termsOfUse_et_EE.ftl
[Vitro]/webapp/languages/et_EE/templates/freemarker/userAccounts-acctCreatedEmail_et_EE.ftl
[Vitro]/webapp/languages/et_EE/templates/freemarker/
userAccounts-acctCreatedExternalOnlyEmail_et_EE.ftl
[Vitro]/webapp/languages/et_EE/templates/freemarker/
userAccounts-confirmEmailChangedEmail_et_EE.ftl
[Vitro]/webapp/languages/et_EE/templates/freemarker/
userAccounts-firstTimeExternalEmail_et_EE.ftl
[Vitro]/webapp/languages/et_EE/templates/freemarker/userAccounts-passwordCreatedEmail_et_EE.ftl
[Vitro]/webapp/languages/et_EE/templates/freemarker/
userAccounts-passwordResetCompleteEmail_et_EE.ftl
[Vitro]/webapp/languages/et_EE/templates/freemarker/
userAccounts-passwordResetPendingEmail_et_EE.ftl
[VIVO]/languages/et_EE/templates/freemarker/aboutMapOfScience_et_EE.ftl
[VIVO]/languages/et_EE/templates/freemarker/aboutQrCodes_et_EE.ftl
[VIVO]/languages/et_EE/templates/freemarker/mapOfScienceTooltips_et_EE.ftl
```

Example content

```
<section id="terms" role="region">
    <h2>kasutustingimused</h2>
    <h3>Hoiatused</h3>
    <p>
        See ${termsOfUse.siteName} veebisait sisaldab materjali; teksti informatsiooni
        avaldamine tsitaadid, viited ja pildid ikka teie poolt ${termsOfUse.siteHost}
        ja erinevate kolmandatele isikutele, nii üksikisikute ja organisatsioonide,
        äri-ja muidu. Sel määral copyrightable Siin esitatud infot VIVO veebilehel ja
        kättesaadavaks Resource Description Framework (RDF) andmed alates VIVO at
        ${termsOfUse.siteHost} on mõeldud avalikuks kasutamiseks ja vaba levitamise
        tingimuste kohaselt
        <a href="http://creativecommons.org/licenses/by/3.0/"
                target="_blank" title="creative commons">
            Creative Commons CC-BY 3.0
        </a>
        litsentsi, mis võimaldab teil kopeerida, levitada, kuvada ja muuta derivaadid
        seda teavet teile anda laenu ${termsOfUse.siteHost}.
    </p>
</section>
```

RDF data (.n3)

Data in the RDF models includes labels for the properties and classes, labels for property groups and class groups, labels for menu pages and more.

Example file names

```
[VIVO]/languages/et_EE/rdf/applicationMetadata/firsttime/classgroups_labels_et_EE.n3
[VIVO]/languages/et_EE/rdf/applicationMetadata/firsttime/propertygroups_labels_et_EE.n3
[VIVO]/languages/et_EE/rdf/display/everytime/PropertyConfig_et_EE.n3
[VIVO]/languages/et_EE/rdf/display/firsttime/aboutPage_et_EE.n3
[VIVO]/languages/et_EE/rdf/display/firsttime/menu_et_EE.n3
[VIVO]/languages/et_EE/rdf/tbox/firsttime/initialTBoxAnnotations_et_EE.n3
```

Example content

```
<http://vivoweb.org/ontology#vitroClassGrouppeople>
    <http://www.w3.org/2000/01/rdf-schema#label> "inimesed"@et-EE .
<http://vivoweb.org/ontology#vitroClassGrouppublications>
    <http://www.w3.org/2000/01/rdf-schema#label> "teadus"@et-EE .
<http://vivoweb.org/ontology#vitroClassGrouporganizations>
    <http://www.w3.org/2000/01/rdf-schema#label> "organisatsioonid"@et-EE .
<http://vivoweb.org/ontology#vitroClassGroupactivities>
    <http://www.w3.org/2000/01/rdf-schema#label> "tegevused"@et-EE .
```

The selection image (.png, .jpeg, .gif)

If you allow the user to select a preferred language, you must supply an image for the user to click on. Typically, this image is of the flag of the country where the language is spoken.

Example file names

```
[VIVO]/languages/et_EE/themes/wilma/i18n/images/select_locale_et_EE.gif
```

Example content

How can I contribute my language files to the VIVO community?

If you are planning to create a translation of VIVO, you should coordinate with the VIVO developers. When your files are ready, you can make them available to the development team in any way you choose. Note that the VIVO project will release your files under the Apache 2 License. They will require a Contributor Agreement stating that you agree to those terms.

3.14.3 Adding language support to your local modifications

If you make changes to the VIVO code or templates, you may want to add language support to your changes. This is only necessary if your site supports multiple languages, or if you plan to contribute your code to the VIVO community.

Language in the data model

The usual form of language support in RDF is to include multiple labels for a single individual, each with a language specifier.

In fact, any set of triples in the data model are considered to be equivalent if they differ only in that the objects are strings with different language specifiers. If language filtering is enabled, VIVO will display the value that matches the user's preferred locale. If no value exactly matches the locale, the closest match is displayed.

Consider these triples in the data:

```
<http://abc.edu/individual/subject1> <http://abc.edu/individual/property1> "coloring" .
<http://abc.edu/individual/subject1> <http://abc.edu/individual/property1> "colouring"@en-UK .
<http://abc.edu/individual/subject1> <http://abc.edu/individual/property1> "colorear"@es .
```

VIVO would display these values as follows:

User's preferred locale	displayed text
en_UK	colouring

User's preferred locale	displayed text
en_CA	colouring
es_MX	colorear
fr_FR	coloring

⚠ In release 1.7, there is still only limited language support for editing values in the GUI. It is possible to edit language-specific labels for individuals. Language-specific values for other properties must be ingested into VIVO.

Language support in VIVO pages

This section deals with the framework of the VIVO pages: the page titles, the prompts, the tool tips, the error messages; everything that doesn't come from the data model. These pieces of text are not stored in RDF, so we need a different mechanism for managing them.

The mechanism we use is based on the Java language's built-in framework for Internationalization. You can find more information in the Java tutorials for resource bundles and properties files.

"Internationalization" is frequently abbreviated as "I18n", because the word is so long that there are 18 letters between the first "I" and the last "n".

In the I18n framework, displayed text strings are not embedded in the Java classes or in the Freemarker template. Instead, each piece of text is assigned a "key" and the code will ask the framework to provide the text string that is associated with that code. The framework has access to sets of properties files, one set for each supported language, and it will use the appropriate set to get the correct strings.

For example, suppose that we have:

- The text that will appear in an HTML link, used to cancel the current operation, with the key `cancel_link`.
- The title of a page used to upload an image, with the key `upload_image_page_title`.
- The text of a prompt message, telling users how big an image must be, with the key `minimum_image_dimensions`.

The default properties file might show the English language versions of these properties, like this:

Excerpt from all.properties

```
cancel_link = Cancel
upload_image_page_title = Upload image
minimum_image_dimensions = Minimum image dimensions: {0} x {1} pixels
```

Notice that the actual image dimensions are not part of the text string. Instead, placeholders are used to show where the dimensions will appear when they are supplied. This allows us to specify the language-dependent parts of a message in the properties file, while waiting to specify the language-independent parts at run time.

A Spanish language properties file might show the Spanish versions of these properties in a similar manner:

Excerpt from all_es.properties

```
cancel_link = Cancelar
upload_image_page_title = Subir foto
minimum_image_dimensions = Dimensiones mínimas de imagen: {0} x {1} pixels
```

To use these strings in Java code, start with the I18n class, and the key to the string. Supply values as needed to replace any placeholders in the message.

Using I18n strings from Java code

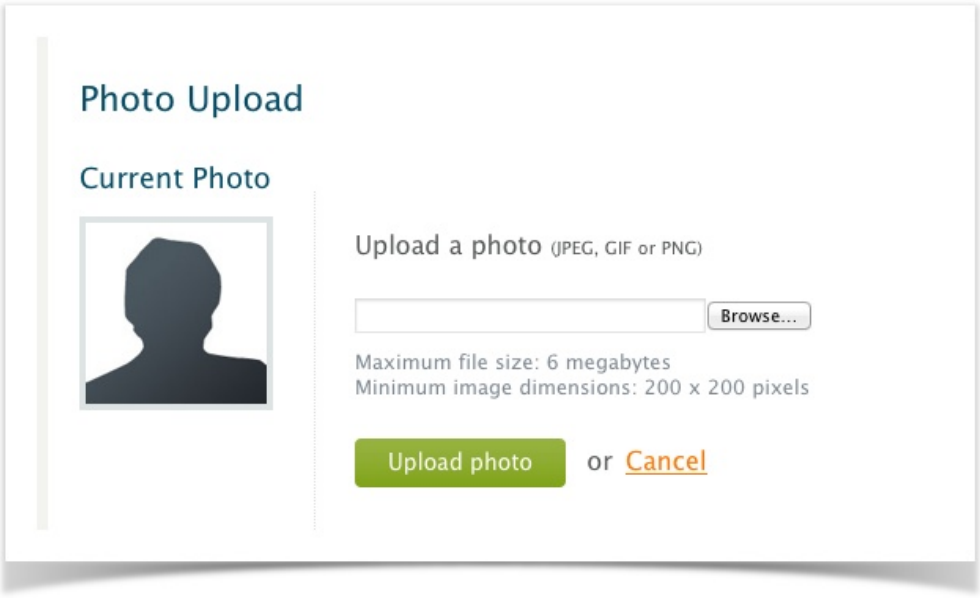
```
protected String getTitle(String siteName, VitroRequest vreq) {
    return I18n.text(vreq, "upload_image_page_title");
}
private String getPrompt(HttpServletRequest req, int width, int height) {
    return I18n.text(req, "minimum_image_dimensions", width, height);
}
```

Similarly, using text strings in a Freemarker template begins with the `i18n()` method.

Using I18n strings in a Freemarker template

```
<#assign text_strings = i18n() >
<a href="../cancel" >
    ${text_strings.cancel_link}
</a>
<p class="note">
    ${text_strings.minimum_image_dimensions(width, height)}
</p>
```

Here is the appearance of the page in question, in English and in Spanish:



Structure of the properties files

The properties files that hold text strings are based on the Java I18n framework for resource bundles. Here is a tutorial on resource bundles.

Most text strings will be simple, as shown previously. However, the syntax for expressing text strings is very powerful, and can become complex. As an example, take this text string that handles both singular and plural:

A complex text string

```
deleted_accounts = Deleted {0} {0, choice, 0#accounts |1#account |1<accounts}.
```

The text strings are processed by the Java I18n framework for message formats. Here is a tutorial on message formats. Full details can be found in the description of the MessageFormat class.

Local extension: application vs. theme

The Java I18n framework expects all properties files to be in one location. In VIVO, this has been extended to look in two locations for text strings. First, it looks for properties files in the current theme directory. Then, it looks in the main application area. This means that you don't need to include all of the basic text strings in your theme. But you can still add or override strings in your theme.

If your VIVO theme is named "frodo", then your text strings (using the default bundle name) would be in

- `webapp/themes/frodo/i18n/all.properties`
- `webapp/i18n/all.properties`

If you specify a complex locale for VIVO, this search pattern becomes longer. For example, if your user has chosen Canadian French as his language/country combination, then these files (if they exist) will be searched for text strings:

- `webapp/themes/frodo/i18n/all_fr_CA.properties`
- `webapp/i18n/all_fr_CA.properties`
- `webapp/themes/frodo/i18n/all_fr.properties`
- `webapp/i18n/all_fr.properties`
- `webapp/themes/frodo/i18n/all.properties`
- `webapp/i18n/all.properties`

When VIVO finds a text string in one of these files, it uses that value, and will not search the remaining files.

Language in Freemarker page templates

Here is some example code from `page-home.ftl`

Excerpt from page-home.ftl

```
<section id="search-home" role="region">
    <h3>${i18n().intro_searchvivo} <span class="search-filter-selected">filteredSearch</span></
h3>
    <fieldset>
        <legend>${i18n().search_form}</legend>
        <form id="search-homepage" action="${urls.search}" name="search-home" role="search"
method="post" >
            <div id="search-home-field">
                <input type="text" name="querytext" class="search-homepage" value=""
autocapitalize="off" />
                <input type="submit" value="${i18n().search_button}" class="search" />
                <input type="hidden" name="classgroup"  value="" autocapitalize="off" />
            </div>
            <a class="filter-search filter-default" href="#" title="${i18n().intro_filtersearch}
">
                <span class="displace">${i18n().intro_filtersearch}</span>
            </a>
            <ul id="filter-search-nav">
                <li><a class="active" href="">${i18n().all_capitalized}</a></li>
                <@lh.allClassGroupNames vClassGroups! />
            </ul>
        </form>
    </fieldset>
</section> <!-- #search-home -->
```

This code lays out all of the formatting and markup, but the actual strings of text are retrieved from the property files, depending on the current language and locale. Here are the English-language strings used by this code:

English properties used in the example

```
intro_searchvivo = Search VIVO
search_form = Search form
search_button = Search
intro_filtersearch = Filter search
all_capitalized = All
```

Language-specific templates

Most Freemaker templates are constructed like the one above; the text is merged with the markup at runtime . In most cases, this results in lower maintenance efforts, since the markup can be re-structured without affecting the text that is displayed.

In some cases, however, the template is predominantly made up of text, with very little markup. In these cases, it is simpler to rewrite the entire template in the chosen language.

The Freemarker framework has anticipated this. When a template is requested, Freemarker will first look for a language-specific version of the template that matches the current locale. So, if the current locale is `es_MX` , and a request is made for `termsOfUse.ftl`, Freemarker will look for these template files:

Search order for `termsOfUse.ftl` Current locale is `es_MX`
`termsOfUse_es_MX.ftl`
`termsOfUse_es.ftl`
`termsOfUse.ftl`

Language in Java code

Java code has access to the same language properties that Freemarker accesses. Here is an example of using a language-specific string in Java code:

Excerpt from UserAccountsAddPageStrategy.java

```
  FreemarkerEmailMessage email = FreemarkerEmailFactory.createNewMessage(vreq);
  email.addRecipient(TO, page.getAddedAccount().getEmailAddress());
  email.setSubject(i18n.text("account_created_subject", getSiteName()));
```

The properties files contain this line:

English language properties used in the example

```
 account_created_subject = Your {0} account has been created.
```

Note how the name of the VIVO site is passed as a parameter to the text message.

Language in JSPs

Up through VIVO release 1.7, no attempt has been made to add language support to JSPs.

Language in JavaScript files

There is no convenient way to access the `i18n` framework from JavaScript files. One workaround is to assign values to JavaScript variables in the Freemarker template, and then access those values from the JavaScript.

For example, the template can contain this:

Excerpt from page-home.ftl

```
    <script>
        var i18nStrings = {
            countriesAndRegions: '${i18n().countries_and_regions}',
            statesString: '${i18n().map_states_string}',
    </script>
```

And the script can contain this:

Excerpt from homePageMaps.js

```
        if ( area == "global" ) {
            text = " " + i18nStrings.countriesAndRegions;
        }
        else if ( area == "country" ) {
            text = " " + i18nStrings.statesString;
        }
```

3.14.4 Tools you can use

i18nChecker

This is a set of Ruby scripts that are distributed with VIVO, in the `utilities/languageSupport/i18nChecker` directory. Use them to scan your language properties files and your freemarker templates. The scripts look for common errors in the files.

Scanning language properties files:

- Warn if a specialized file has no default version.
- Warn about duplicate keys, keys with empty values.
- Warn about keys that do not appear in the default version.
- If the "complete" flag is set,
    - Warn if the default version is not found.
    - Warn about missing keys, compared to the default version.

Scanning Freemarker templates:

- Warn about visible text that contains other than blank space or Freemarker expressions.
- Visible text is:
    - Anything that is not inside a tag and not between <script> tags
    - title="" attributes on any tags
    - alert="" attributes on <img> tags
    - alt="" attributes on <img> tags
    - value="" attributes on <input> tags with submit attributes

4 Customization: Appendices

Tips and techniques to help in the process of customization.

4.1 Overview

This page describes some tools and techniques that can help customizing your VIVO user interface.

4.2 Use the Developer Panel

Many of these techniques involve The Developer Panel, which is described in the VIVO Programmer's Notes section of the wiki.

You can change settings on The Developer Panel interactively, while VIVO is running, or you can use a `developer.properties` file in your VIVO home directory.

A typical developer.properties file

```
developer.enabled=true
developer.permitAnonymousControl=true
developer.defeatFreemarkerCache=true
```

When any feature of The Developer Panel is active, you will see this indicator in the header of your VIVO pages:

This is to remind you that developer options may slow down your VIVO, and should not be used in production.

4.3 Iterate your code more quickly

4.3.1 Reduce the VIVO build time

A full rebuild of VIVO may be necessary if you are changing the Java source code, or the contents of RDF files. However, if you are only making changes to the Freemarker templates, you can run the build script like this:

```
ant deploy -skiptests=true
```

By choosing `ant deploy` instead of `ant all` or `ant clean deploy`, you are selecting an incremental build which will not re-compile all of the Java classes, or copy all of the unchanged files. By setting the option `-skiptests=true`, you are choosing not to run the unit tests. This is reasonable because the unit tests to not apply to the Freemarker templates.

After making your changes to the templates, you should perform a full build with unit tests.

```
ant all
```

4.3.2 Don't restart VIVO until you need to

VIVO will detect changes to the templates without requiring a restart. However, you will probably want to defeat the Freemarker cache (see below).

Also, VIVO will serve the latest version of CSS, JavaScript, or image files. For these files, however, you may need to clear the cache in your browser. Instructions for doing this will differ, depending on which browser you are using. If you don't know how to reset the cache in your browser, you may want to consult this web site: http://clearyourcache.com/, or just search the web for "clear browser cache".

If you change any other types of files, you will need to restart VIVO after running the build script.

4.3.3 Defeat the Freemarker cache

As mentioned above, VIVO will detect changes to Freemarker templates. By default, however, VIVO will not detect the changes immediately. The Freemarker framework caches the templates that it uses, and won't even look to see if a template has changed until 1 minute after it was last read from disk. In a production system, of course, that makes the accessing much more efficient. When you are making frequent changes, it's an annoyance.

Use The Developer Panel to defeat the Freemarker cache.

4.3.4 Customizing listViewConfigs

Ted Lawless has written an open-source Python script to assist with viewing the output of a listViewConfig without having to rebuild the entire Vivo app.

Also, you can skip the unit tests when building VIVO, as shown in Reduce the VIVO build time. Unit tests do not apply to listViewConfigs.

4.4 Reveal what VIVO is doing

4.4.1 Insert template delimiters in the HTML

It's not always clear which template has created a particular piece of your HTML page. Templates include other templates, templates are invoked in custom list views, short views, etc. You can use The Developer Panel to insert comments in the HTML that tell you where each template begins and ends.

For example, this section of a page was produced mostly by the `identity.ftl` template. The `languageSelector.ftl` template is included, but does not generate any HTML. The next section is produced by the `menu.ftl` template, and so on.

```
...
    <body >
            <!-- FM_BEGIN identity.ftl -->
<header id="branding" role="banner">
  <h1 class="vivo-logo"><a title="VIVO | connect share discover" href="/vivo">
    <span class="displace">VIVO</span>
  </a></h1>
  <nav role="navigation">
    <ul id="header-nav" role="list">
<!-- FM_BEGIN languageSelector.ftl -->
<!-- FM_END languageSelector.ftl -->
        <li role="listitem"><a href="/vivo/browse" title="Index">Index</a></li>
        <li role="listitem"><a href="/vivo/siteAdmin" title="Site Admin">Site Admin</a></li>
        <li>
          <ul class="dropdown">
            <li id="user-menu"><a href="#" title="user">Jim</a>
              <ul class="sub_menu">
                <li role="listitem"><a href="/vivo/accounts/myAccount" title="My account">My
account</a></li>
                <li role="listitem"><a href="/vivo/logout" title="Log out">Log out</a></li>
              </ul>
            </li>
          </ul>
        </li>
      </ul>
    </nav><!-- FM_END identity.ftl -->
            <!-- FM_BEGIN menu.ftl -->
</header>
...
```

4.5 Improve your SPARQL Queries

Ted Lawless and Steve McCauley at Brown point out these resources for learning SPARQL:

- Understanding SPARQL is important when it comes to modifying the listViewConfig files. Bob DuCharme's "Learning SPARQL" is the go-to reference: http://www.learningsparql.com/
- For understanding SPARQL, it also helps to understand how it is related to SQL, the query language for relational databases: http://www.cambridgesemantics.com/semantic-university/sparql-vs-sql-intro

But not all SPARQL techniques are efficient in practice. Brown has open-sourced their VIVO customization, where you can see some aspects of SPARQL that has worked for them:

https://github.com/Brown-University-Library/vivo/tree/master/productMods/config

This includes constructing local display properties against which they select.