Home page customizations

Introduction

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

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 />
```

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 then modify it accordingly so that it displays research classes. Refer to The Departments Section below for more details.

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
```

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.

```
# academic departments datagetter

<freemarker:lib-home-page.ftl> display:hasDataGetter display:academicDeptsDataGetter .

display:academicDeptsDataGetter
    a <java:edu.cornell.mannlib.vitro.webapp.utils.dataGetter.SparqlQueryDataGetter> ;
    display:saveToVar "academicDeptDG" ;
    display:query """
    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 ?deptURI (str(?label) as ?name)
    WHERE
    {
          ?deptURI rdf:type vivo:AcademicDepartment .
          ?deptURI rdfs:label ?label
    }

    """ .
```

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 section heading.

```
<#-- Renders the html for the academic departments section on the home page. -->
<#-- Works in conjunction with the homePageUtils.js file -->
<#macro academicDeptsHtml>
    <section id="home-academic-depts" class="home-sections">
        <h4>${i18n().departments}</h4>
        <div id="academic-depts">
        </div>
    </section>
</#macro>

<#-- builds the "academic departments" box on the home page -->
<#macro listAcademicDepartments>
<script>
var academicDepartments = [
<#if academicDeptDG?has_content>
    <#list academicDeptDG as resultRow>
        <#assign uri = resultRow["theURI"] />
        <#assign label = resultRow["name"] />
        <#assign localName = uri?substring(uri?last_index_of("/")) />
            {"uri": "${localName}", "name": "${label}"}<#if (resultRow_has_next)>,</#if>
    </#list>
</#if>
];
var urlsBase = "${urls.base}";
</script>
</#macro>
```

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.

The map is enabled in the runtime.properties file. Include or uncomment the line:

```
homePage.geoFocusMaps=enabled
```

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:

- 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

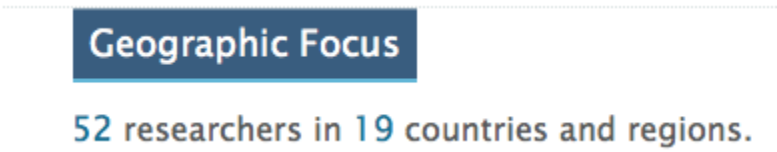
Besides the buildGlobalMap() function (discussed above), the homePageMaps.js file also includes buildCountryMap() and buildLocalMap() functions. These functions are very similar to the buildGlobalMap() function and work in the same way. When you are implementing a country map, you will want the map to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