EVN Developer Guide

Page Contents

EVN Tutorials

This guide describes options and advanced techniques for customizing aspects of TopBraid EVN. This is for advanced users, administrators, and developers who can address topics in EVN systems, semantic technologies, and development tools including the TopBraid Composer - Maestro Edition (TBC-ME) environment.

In general, any product modifications or extensions should be developed in a non-production environment and thoroughly tested under relevant conditions before deploying into production. Additionally, support considerations apply for both production usage and technical migrations as needed for product upgrades.

If you are interested in the possibilities for tailoring TopBraid EVN to better suit your needs, please contact TopQuadrant to explore the following options:

- Having TopQuadrant quickly configure a customized EVN solution to meet your detailed requirements. We will be pleased to quote and provide affordable customization and tailoring services.
- Enabling your organization to develop and maintain customizations for EVN by guiding and training your selected personnel to perform the variety of customization capabilities.

Preparing Ontology, Taxonomy and Crosswalk Data for Import

As described in the Importing RDF into the Taxonomy Editor section of the User Guide chapter, EVN expects that RDF imported into the taxonomy editor conforms to the W3C SKOS standard. That section describes specific details about classes and properties that EVN expects to see such as `skos:ConceptScheme` and `skos:hasTopConcept`. If you have SKOS RDF that does not fit this model—for example, `skos:ConceptScheme` is absent —you can use SPARQL CONSTRUCT or UPDATE queries and TopBraid Composer to add or convert to the necessary SKOS data. TopBraid Composer includes a range of features for developing and executing these queries, as well as for saving them in a SPARQLMotion script if you need to run a given CONSTRUCT query or set of queries on a regular basis. Taxonomy Editor also expects the use of `skos:prefLabel.` If your RDF contains `rdfs:label` values, they will be converted to `skos:prefLabel` values.

EVN Ontology Editor has no special requirements on the classes, properties or instances that must be present. The only exception is the presence of `rdfs:subClassOf` statements for classes, which are needed for them to appear in the class tree.

EVN Crosswalks contain RDF triples of the following form:

```
example:Vocabulary1_Resource1 skos:closeMatch :example:Vocabulary2_Resource1.
```

Valuable data for use in your vocabularies and asset collections may not be stored in RDF, but instead in another model or format. The Importing spreadsheet data section of the User Guide chapter describes how to import spreadsheets representing a taxonomy hierarchy in a choice of patterns; if your spreadsheet does not fit one of these patterns, TopBraid Composer offers several approaches for converting a spreadsheet into RDF that can then be converted to SKOS RDF with CONSTRUCT queries. In the TopBraid Composer > Help, see **Importing Data Sources > Import external information > Import Spreadsheets** for an overview of these approaches and links to more detailed information.

You can also use tabular (ho hierarchy) spreadsheets to import data, including creating relationships between resources. For crosswalks the spreadsheet should have only two columns representing the two sides of the crosswalk. No mapping will be required.

You can create an ontology from a spreadsheet - using the header row as properties and the worksheet name as a class.

If your data is in one or more relational databases, TopBraid lets you create connectors that make it possible to work with that data as RDF triples so that you can then convert it to SKOS. In Composer's online help, select **Working with Data Back Ends > Working with Relational Databases** to learn more.

SPARQLMotion scripts let you create scripts by dragging and dropping from a wide selection of specialized modules into visual data manipulation pipelines. You can run these scripts interactively from within Composer or as web services deployed to your EVN server. SPARQLMotion module choices such as ConvertJSONToRDF and ConvertXMLByXSLT give additional options for taking advantage of non-RDF data that may be available to you. In the Composer online help, select **Application Development Tools > SPARQLMotion Scripts** for an overview of SPARQLMotion and select **Reference > SPARQLMotion Module Library Reference** for a complete list of available modules. These modules can also be used within SPARQL Web Pages (SWP) scripts. Unlike SPARQLMotion where data transformation scripts are developed using a visual drag and drop approach, SWP is a textual scripting language using custom HTML tags and JavaScript.

TopBraid's Semantic XML feature gives you another option for converting XML to RDF, storing enough information about the input XML to let you round-trip the data back to a valid XML document. Select **Working with XML and Table files > Creating, Importing, Querying, Saving XML documents with Semantic XML** to learn more about this feature.

If you are unsure which choices in the TopBraid Composer toolset would let you best take advantage of the data available to you, contact your EVN support representative.

Once you create a custom import script with TopBraid Composer, you can make it available to all EVN users by following instructions in the Projec tPlug-ins section.

Advanced Debugging

Debugging of complex relationships within your data may require more sophisticated investigations than the EVN query form allows. You can send the data any SPARQL query you like via the SPARQL endpoint, which has a web-based form at /EVN/tbl/sparql where you can enter queries. (For example, for an EVN installation at http://mycompany.com/EVN, the endpoint would be at http://mycompany.com/EVN/tbl/sparql.)

For an RDF-based test-case framework, see *DASH Test Cases Vocabulary*: http://datashapes.org/testcases.html

Even more advanced investigations can be done by FTPing models to a local copy of TopBraid Composer (TBC-ME). The following is a series of general steps that can be used in this scenario. Note these steps will not work for TDB backend.

1. FTP the EVN vocabulary project to a folder on the machine running TBC-ME.
2. Use Import Existing Projects into Workspace to place the project into TBC-ME's workspace. This give you access to the data through the back-end.
3. Modification can be made directly to the data graph as needed, because save operations will save to the data back end.

   If you use TopBraid Composer and not EVN to make these changes, they will not be recorded using EVN change tracking features, and may conflict with activity taking place using the web-based interface, so should be avoided.
4. Test in localhost to make sure the changes are correct.

EVN Integration Points

TopBraid EVN's ability to import and export data gives you some batch-oriented options for using data from other systems as well as providing managed vocabulary data and metadata for use by other systems. For more dynamic integration with other systems, EVN offers two classes of options: web services and a SPARQL endpoint.

As a SPARQL endpoint

Add `/evn/tbl/sparql` to your EVN server's domain name and port number to get the URL of its SPARQL endpoint. Each asset collection is stored as a separate named graph on the server; to find a given asset collection's graph name, mouse over its name on the EVN home screen and note the projectGraph parameter in the URL that the name links to. **NOTE:** The SPARQL standard treats references to non-existent graphs the same as references to empty ones, which means that incorrect graph URIs might not raise any obvious errors.

For example, if you have EVN installed at the domain name ourserver.com on port 8080, entering the following query into the TopBraid Composer SPARQL view would send a query to that endpoint requesting a list of Continent resources in the Geography vocabulary:

```
SELECT ?continent WHERE
{ SERVICE <http://ourserver.com:8080/evn/tbl/sparql>
   {GRAPH <urn:x-evn-master:geo>
 {?continent a <http://topquadrant.com/ns/examples/geography#Continent>}
  }
}
```

Instead of using a SPARQL client, such as the TBC-ME SPARQL view to send the query to the endpoint with the SERVICE keyword, you (or, more likely, an application you are building) can run the query by embedding an escaped version of it in a URL built on the endpoint URL. For example, let's say you wanted to send the following query to the endpoint `http://ourserver.com:8080/EVN/tbl/sparql` :

```
SELECT ?continent WHERE
   {GRAPH <urn:x-evn-master:geo>
 { ?continent a <http://topquadrant.com/ns/examples/geography#Continent>}
}
```

You can add an escaped version as a query parameter value to the endpoint URL shown above, resulting in the following URL (split here for display, but to be treated as a single line). The best practice is to use a URL encoding function to properly format the query for HTTP:

```
http://ourserver.com:8080/evn/tbl/sparql?query=SELECT+?continent+WHERE+{GRAPH+<urn:x-evn-master:geo>+
{?continent+a+<http://topquadrant.com/ns/examples/geography%23Continent>}}
```

An option to specifying the named graph with the GRAPH keyword, is to name the graph in a default-graph-uri parameter added to the URL. For example, let's look at this simpler query that does not mention the graph to query:

```
SELECT ?continent WHERE
{?continent a <http://topquadrant.com/ns/examples/geography#Continent>}
```

Escaping this and adding it to the SPARQL endpoint URL with the named graph specified using the default-graph-uri parameter gives us the following URL (split here for display, but to be treated as one line):

```
http://localhost:8080/evn/tbl/sparql?default-graph-uri=urn:x-evn-master:geo&query=SELECT+?continent+WHERE+
{?continent+a+<http://topquadrant.com/ns/examples/geography%23Continent>}
```

For further information on these options and on ways to specify the result set format, see the TBC-ME online help at TopBraid Composer > TopBraid Live Integration > TopBraid Live SPARQL Endpoint.

SPARQL endpoint update

SPARQL Update statements can be executed on EVN project graphs. By default, SPARQL Endpoint Updates are disabled. To enable updates set the Advanced Parameter: **Enable SPARQL updates** to true in **Server Administration > Server Configuration Parameters**. Access controls are also available for SPARQL Endpoint Updates. For more information on setup for SPARQL Endpoint update, see SPARQLEndpointUpdate

SPARQL Update statements use the `sparql` servlet with the `update` parameter for specifying the SPARQL Update query. Note that the query must therefore explicitly specify a graph context to execute the update in using the GRAPH <...> syntax. An example update call is shown in the following URL:

```
tbl/sparql?update=INSERT{GRAPH <urn:x-evn-master:schemaexample>{<http://topbraid.org/examples/topquadrant%23TBEVN> <http://www.w3.org/2000/01/rdf-schema%23comment> "a comment"}}WHERE {}
```

Note this assumes a user is logged in, either via a browser or a basic authentication request. SPARQL Updates can be included in the change history by appending the user name to the end of the graph name as in the following example. The change will be added to the change graph with the currently logged in user as the creator of the change.

```
tbl/sparql?update=INSERT{GRAPH <urn:x-evn-master:schemaexample:Administrator>{<http://topbraid.org/examples/topquadrant%23TBEVN> <http://www.w3.org/2000/01/rdf-schema%23comment> "a second comment"}}WHERE {}
```

Access controls can also be defined for SPARQL Update using the Permission Group Management page and the SPARQLUpdateAllowGrp and SPARQLUpdateDenyGrp groups. For more information see the Permission Group Management document.

Application Configuration

This section illustrates some of the ways that EVN can be extended to customize the user experience. The most basic and probably most powerful way of adjusting EVN to your specific needs is by defining your own schemas for different types of resources (as classes and properties) through the editor. The UI is flexible enough to handle any number of custom properties of any data type.

TopBraid EVN was designed to offer, out-of-the-box, most of the core capabilities needed for enterprise vocabulary and asset management functions identified by our market research and interactions with a wide variety of organizations.

Recognizing that some organizations and users may require a certain degree of modifications and additions to the user interface and functionality —ranging from simple *tailoring* (for example, adding a logo, changing some color scheme choices) to *customization of operations* (for example, modifying or adding a use case)— TopBraid EVN includes several levels and capabilities for customization. Built on the TopBraid Suite platform, designed for creating and modifying flexible applications, EVN's features as well as its look and feel can be customized through:

- Adding or modifying business actions through SPARQLMotion scripts. SPARQLMotion is a visual scripting language for data processing and orchestration. Many new operations can be added by developing new scripts.
- Changing the look and feel or content of EVN on entry pages such as the EVN home page, management, and working copy management pages using SPARQL Web Pages, a SPARQL-based framework for describing user interfaces.

If you are interested in the possibilities for tailoring TopBraid EVN to better suit your needs, please contact TopQuadrant to explore the following options:

- Having TopQuadrant quickly configure a customized EVN solution to meet your detailed requirements. We will be pleased to quote and provide affordable customization and tailoring services.
- Enabling your organization to develop and maintain customizations for EVN by guiding and training your selected personnel to perform the variety of customization capabilities outlined below.

The rest of this section describes some of the most frequently used customization options.

Extension Process

Creating some of EVN customizations and extensions requires using TopBraid Composer-Maestro Edition (TBC-ME), which serves as the IDE for TopBraid EVN. TBC-ME is a development toolkit for TopBraid server products. Some experience with SPARQL Web Pages (SWP) is required to build EVN extensions. The general process of customizing and extending EVN using TBC-ME is as follows:

1. Create a project that will only contain your extension files. This project will need to be uploaded onto the server for deployment.
2. In that project, create a SPARQL Web Pages file (RDF/SWP) with TopBraid Composer. Make sure that file ends with `.ui.*` (for example, `myextensions.ui.ttlx`), so that EVN will use it as part of the global registry of SWP components. Unless you restart, you may need to click **System > Refresh TopBraid system registries** to ensure that TBC knows about your new `.ui.` file.
3. In the SWP file, create the extension by making the appropriate additions to your .ui.ttlx file, for example based on the examples mentioned below.
4. You can test your components incrementally by launching EVN from within TBC on localhost.
5. Once you want to use them on the server, use the TopBraid Live project upload feature to put the project containing the SWP file (and supporting files if needed) into production. Refresh the workspace through the EVN administration console after uploading. A server restart may be advisable.

6. TopQuadrant will attempt to ensure compatibility of the extension points with each release. In practice this means that you will only need to redeploy your customization project to TopBraid EVN after you have done a fresh reinstall of a new version. In the case of major technology upgrades it may be unavoidable to have to make adjustments to your customizations. Please test thoroughly after installing a new release.

The file evn-plugins.ui.ttlx contains examples that you can use as a starting point to understand how to implement other extensions. You can download that file and save it into your workspace. Then use **Model > Refresh global SWP graphs** to activate the example extensions and launch EVN to play with them. The following sub-sections briefly describe how to change CSS styled and all examples from the `evn-plugins.ui.ttlx` . In TBC, you can use **Model > Find all locally defined resources** to get a clickable list of these extensions.

Common extension tasks

Adding SPARQL query templates for use in Taxonomy Editor

To add new taxonomy template query choices to the selections described in Running template queries in the Taxonomy help pages, use TopBraid Composer Maestro Edition to create some that follow the pattern of the existing ones and then upload the project containing your new template queries to the EVN server.

The easiest way to accomplish this is by creating a new RDF file, checking "File will export SPIN Functions or Templates (.spin extension)" on the "Create RDF/OWL/SPIN File" dialog box.

Once the file has been created, import EVN.topbraidlive.org/spin/skostemplates.spin.ttl into it, and then create a new subclass of skostemplate: IntegratedTemplates, using the existing subclasses that you see as a model. You will see that the rdfs:label property of each stores the text displayed as the template's name on the EVN **Execute t**emplate query... drop-down list, its rdfs:comment value provides the tooltip, and the spin: body property stores the query to execute. Define run-time parameters by adding spl:Argument values to the spin:constraint property.

After saving the file, upload the project containing it to the EVN server, and the new menu template will be available to your users on the **Execute t**emplate query... menu.

For tips on testing your query against data stored on EVN before uploading its project to make it one of the query template selections, see EVN As a SPARQL Endpoint.

Adding SPARQL endpoints to the copy taxonomy concept from SPARQL endpoint list

The EVN user guide chapter's section on EVN Taxonomy Editing: Copying Data from Remote Sources and Other Taxonomies describes how to copy taxonomy concepts from SPARQL endpoints. SPARQL endpoints listed in the Copy Concept from SPARQL Endpoint dialog box are stored in a model that uses the SPARQL Service Description schema.

Working with these models requires TopBraid Composer, the administrative interface to EVN. To create one, create a SPARQL Web Pages file (that is, one with .ui. in its name, such as EVN-endpoints.ui.ttlx) that imports sd.ttl from the Common folder of TopBraid Composer's TopBraid project in the Navigator view.

To add a choice to the Copy Concept from SPARQL Endpoint list of endpoints, define an instance of sd:Service in your SPARQL Web Pages model. For each, specify the URI of the endpoint as an sd:endpoint value and the name of the endpoint that you want to appear on the dialog box as an rdfs:label value.

This model must be made available to the EVN server; the simplest way to do so would be to create it in its own project and to then upload that project to the EVN server.

Creating and maintaining new SPIN constraint libraries (DEPRECATED; use SHACL)

TopBraid EVN's includes a number of pre-defined constraint templates that let users specify data constraints - Constraint Libraries. You can use TopBraid Composer to define new SPIN constraint templates. Once a template is defined and uploaded to the server, it is available for selection.

To do this, use TopBraid Composer to create a model with ".spin." in its filename (for example, `myConstraints.spin.ttl`) that has `spin:constraint` values added to any classes whose definitions are visible in that model (`skos:Concept` for the taxonomy editor and any class you like for the ontology editor). Import the SPIN/spin.ttl library from the TopBraid SPIN project into the model with your class definitions to make the s `pin:constraint` property available on the class editing form. The skosspin.spin.ttl model included in the SKOS folder of the TopBraid project imports spin.ttl and the skos-core.ttl W3C standard SKOS model in order to implement several of the constraints described in the W3C SKOS specification, and these constraints can serve as a model for new ones that you create. See the "Adding constraints" section of Getting Started with SPARQL Rules (SPIN) for additional information on using TopBraid Composer to create new constraints.

Deploying that model to your EVN installation's workspace will make that model appear in the manage tab on the vocabulary/asset collection's home page under Constraint Libraries. Each model created in this manner will show up as a separate choice on the list, so splitting constraints across models will let you use specific subsets of your total constraint collection in different combinations as necessary for your application.

If you keep these models in a dedicated project in TopBraid Composer, that project can be redeployed to the TopBraid Live server running the EVN application to put those changes into effect as described in the TopBraid Composer online help under **TopBraid Composer > TopBraid Live Integration > Overview of TopBraid Live Integration**.

Changing CSS Styles

In order to customize the TopBraid EVN user interface styling, follow this process:

1. Identify the CSS style that you want to change. This is usually done by looking at the source code or the DOM tree of the HTML with a tool such as FireBug.
2. Create a folder ending with `.www` in the workspace—for example, `my.www`. Restart TBC to make sure that it knows about that folder.
3. In that folder create a CSS file, for example, `my.css`.
4. Add your custom CSS In that CSS file.
5. Create a `.ui.` file (File -> New RDF/SWP File), and add `evn.topbraidlive.org/EVN/EVN.ui.ttlx` to its imports.
6. In that file, select the `SWA:Elements` class and add an empty row to the `ui:headIncludes` property.
7. In that row, enter a line such as `<link href="lib/my/my.css" rel="stylesheet" type="text/css"/>`.

Here is an example CSS snippet to override the default logo:

```
.edg-product-logo {
    background-image: url(images/CustomizedLogo.png);
}
```

To verify that this worked, you should see the link to your CSS file in the generated source of EVN.

Note that the `/lib/...` folder is a redirect to any folder ending with `.www` in your workspace. This way you can also add images and other resources such as JavaScript files.

Customizing the login screen

To implement a background image, locate the following code in logon.html:
<html>
<head>
<meta http-equiv="Content-Type" content="text/html; charset=UTF-8"/>
<style type="text/css">

And beneath it, insert the following code:

```
html, body {
   min-height: 100%;
}
body {
   background-image: url(http://path-to-your-background.jpg);
   background-repeat: no-repeat;
   background-position: 0 0;
   background-size: cover;
}
@media (min-width: 1024px), (min-height: 576px) {
   body { background-size: auto; }
}
```

Update the background-image url, add the @media's min-width and min-height dimensions to match the image.

To add an image to the login window, browse to the bottom of the file and locate the following section:
</style>
<title>TopBraid Live Login Page</title>
</head>

<body>

<!-- <img src="http://your-custom-logo-here.jpg" alt="Logo"> -->

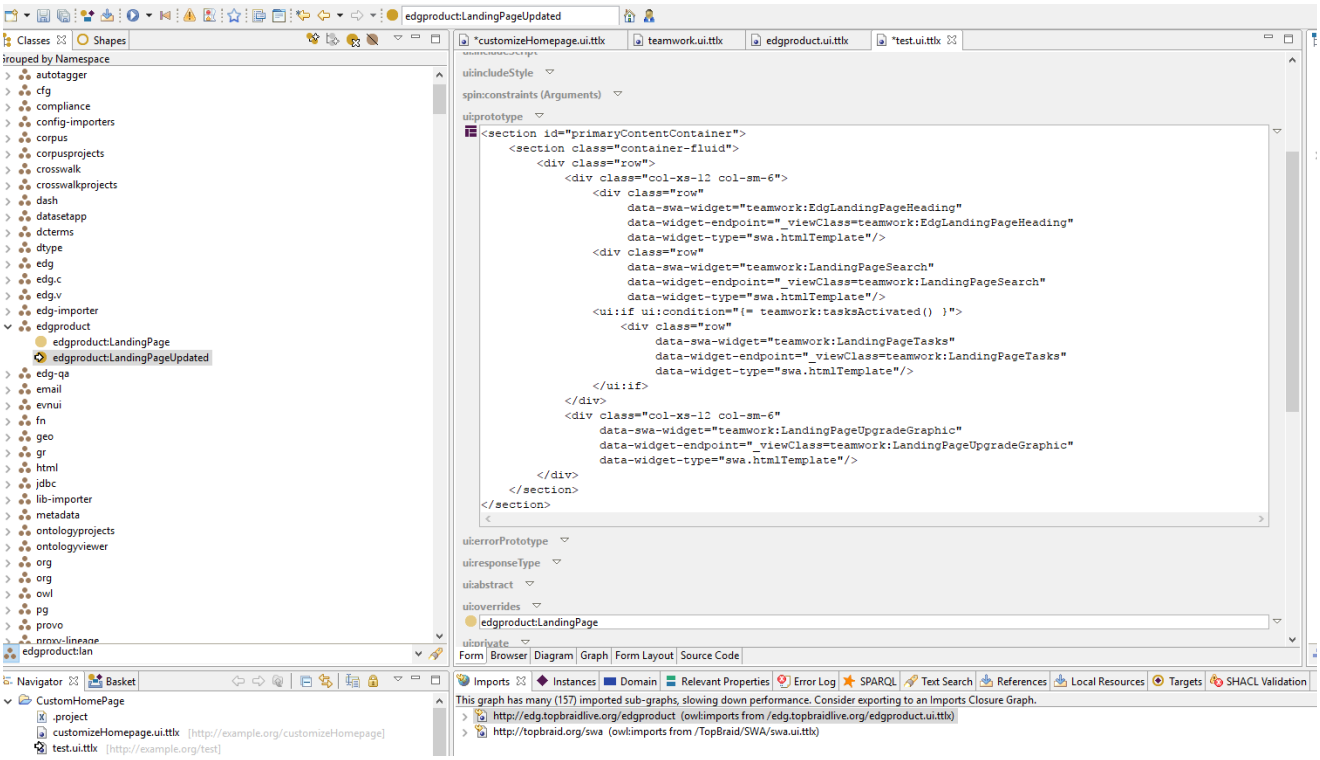
Uncomment the <img> tag and update the src attribute to point to the location where the image is hosted.

Images cannot be hosted from within the application server. Authentication prevents any assets outside of the .html file from being accessed anonymously. Because of this, they must be hosted on a separate web server. You can also alter the html file to use colors, text and fonts.

Customizing the Landing Page

1. In TBCME File -> New -> RDF/SWP File
2. Import edgproduct.ui.ttlx into the newly created file
3. From classes view, navigate to edproduct:LandingPage
4. Right Click -> Clone
5. Set the ui:overrides in the newly created cloned class to edgproduct:LandingPage

At this point we should have a clone of the landing page that is overriding the standard one, and are ready to make some changes:



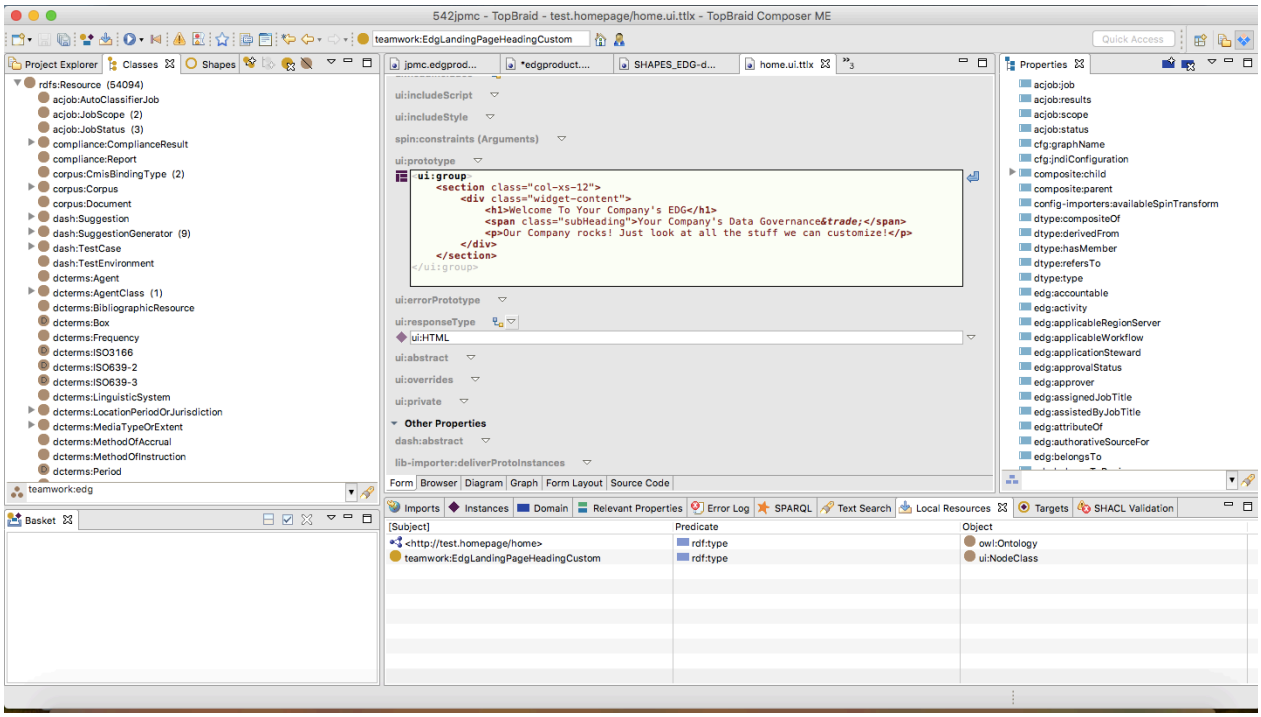
Replacing the EDG Heading Information -

The 'Welcome to TopBraid EDG' part of the landing page is coming from teamwork:EdgLandingPageHeading - so we can do the same sort of clone and override operation to change this content:

1. Navigate to teamwork:EdgLandingPageHeading
2. Right Click -> Clone, and set the ui:overrides on the clone to teamwork:EdgLandingPageHeading
3. Make some changes - update the body to something like this:

edgproduct.ui.ttlx

```
<ui:group>
   <section>
       <div class="col-xs-12">
           <h1>This is my Custom header EDG</h1>
           <span class="subHeading">Totally Customizable</span>
           <p>How cool is that?</p>
       </div>
   </section>
</ui:group>
```



4. Save, and refresh the homepage.

5. You should see your changes reflecting:

Customizing Icons

TopBraid EVN allows the icons of resources in the tree components to be changed, even assigning different icons to representing different classes for easier visual identification by end users.

To do this, first define a CSS style using the steps described above, pointing that style to a background image of your choice. Then, store the name of that style at the class that you want to change the icon of, using the property `swa:typeIcon`. Point the evnui:Editor SWA application at your CSS file, and the tree will use that icon for all instances of the associated class.

For example, to reset the concept hierarchy icon for instances of the Country class in the Geography vocabulary with a myCountryIcon.jpg file stored in the same directory as your CSS file, you could add the following lines to it:

```
.CountryStyle {

background-image: url('myCountryIcon.jpg');

background-position: center;

background-size: 100%;

}
```

Then, import the Geography schema file geo.tdb into your custom ui.ttlx file and assign the g:Country class a `swa:typeIcon` value of "CountryStyle". Finally, import evn.topbraidlive.org/taxonomy/evn.ui.ttlx into your custom .ui.ttlx file and add a ui:headIncludes value of the following to the evnui:Editor class, adjusting the name and path to point at your own CSS file:

```
<link href="lib/my/my.css" rel="stylesheet" type="text/css"/>
```

On the concept hierarchy, you will see your new icon in place of the default one for the specified class:

If no direct type icon exists, it will walk up the superclass tree. In the case of the Concept hierarchy, it will eventually reach the icon defined for the class `skos:Concept`.

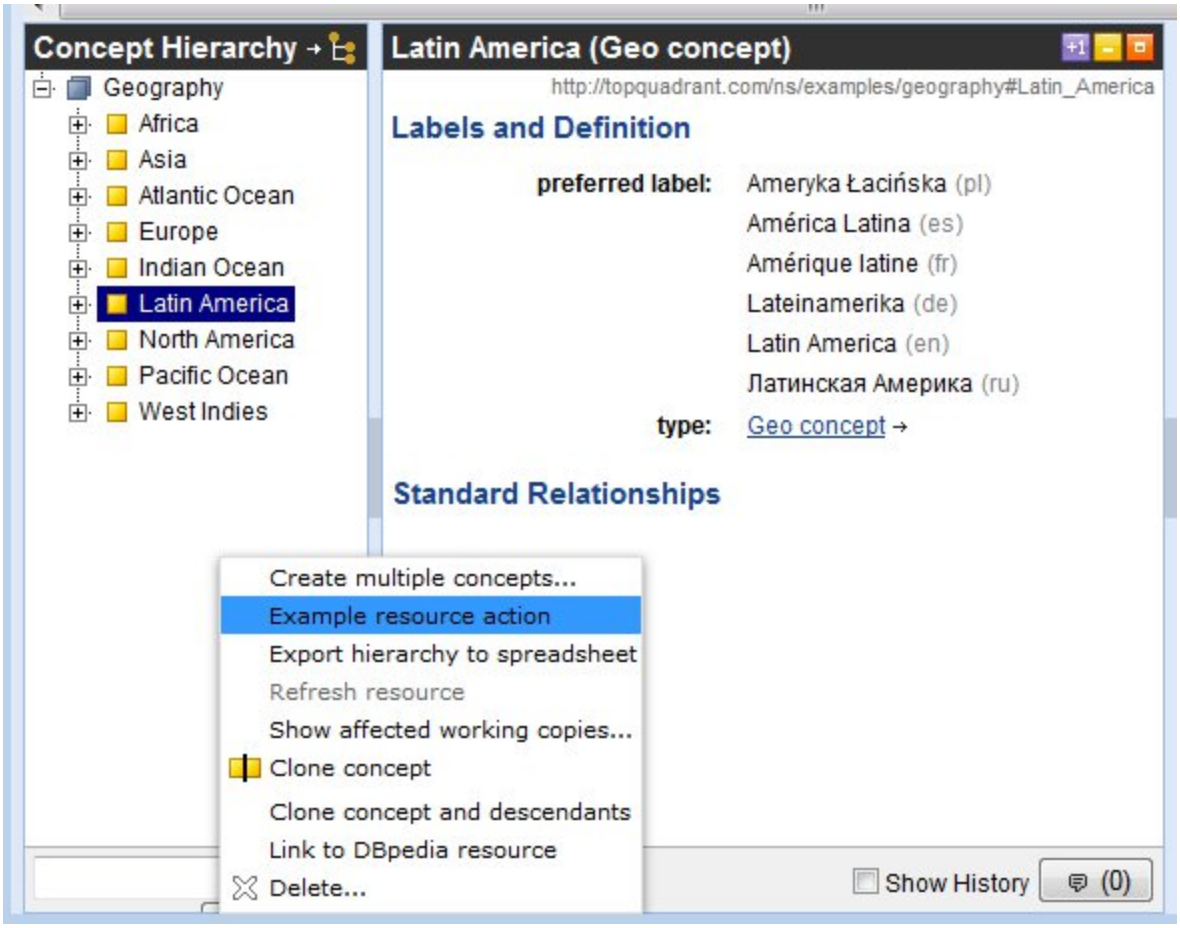
Adding New Menu Choices

For an example of adding a new menu choice you can follow the procedure below.

In the evn-plugins.ui.ttlx example file described at the beginning of this section, navigate to **evn-plugins:ExampleGlobalAction** to see how to contribute items to the global context menu in the header area. This example adds the "Example global action" menu choice. When selected, it counts the number of concepts in the model.

These global menu items can be used for a large variety of tasks, and the example illustrates how to make an Ajax call-back to the server along the way.

The context menu at the bottom of the screen lists menu choices that act on the currently selected resource. See **evn-plugins: ExampleResourceAction** in the example file for an example of how to add an item to this menu. Those action items are often used to perform edits on the selected resource, and the example shows how such edits are implemented using `ui:update`. Below we can see the "Example resource action" item that this example adds to the menu; selecting it displays a dialog box that queries the user for "related" and "date" values that it then adds to the resource:

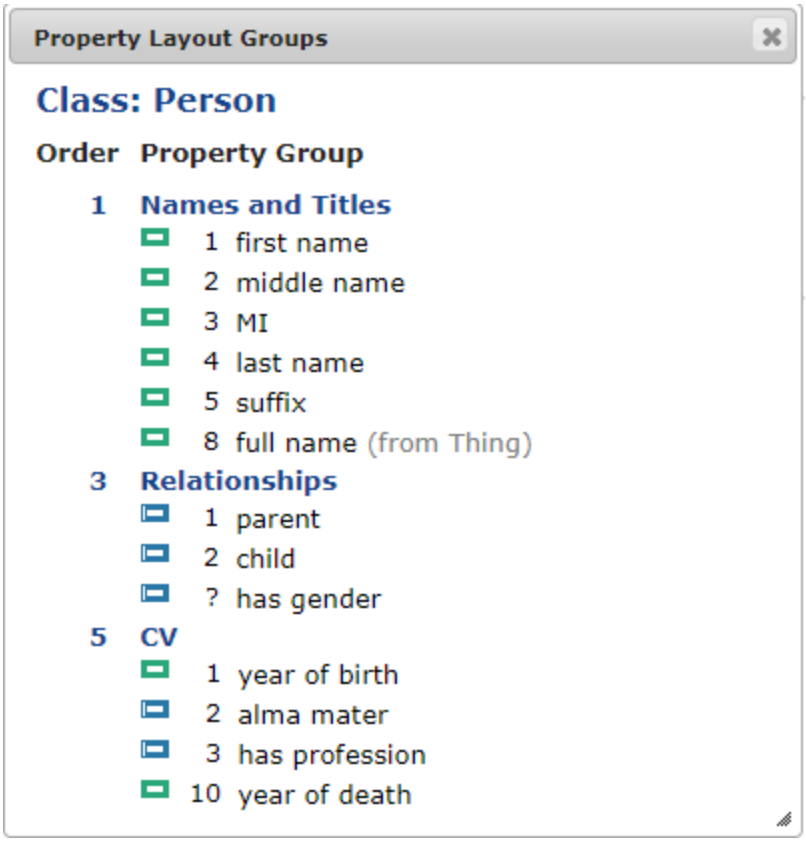


Further details are provided in the `rdfs:comment` values of the examples.

Customizing Form Layouts

Versions 5.4+: Using SHACL constraints

From TopBraid version 5.4 onward, SHACL constraints are the recommended way of defining the layout of property groups in forms. For a given class, its form layout consists of the property groups used by its `sh:property` constraints (via `sh:group`), and `sh:order` is used for relative ordering of both the groups and the properties. This property grouping and ordering for form layouts can be customized in the Ontology editor for classes that have property shapes defined. For a selected class, in its details pane, select **Actions (gear button) > View/Edit SHACL property layout groups...**



This opens an interactive, non-modal window showing the selected class's property-groups, properties, and their relative-order values (integers or decimals). The pop-up window works in parallel with the main editor such that changes in one are reflected in the other.

In addition to grouping properties in to sections, you can also define a display name for a property that is different from the label. This can be particularly useful for inverses. And you can select different edit, search and view widgets. By default, widgets will correspond to the type of the value that a property has.

Versions 4.5 and up: Using form editor or SWP

TopBraid 4.5 introduced a graphical form layout editor, allowing end users to define which properties show up for a given class and in what order. Utilizing the editor, EVN users can create custom form layouts without having to use TopBraid Composer. This feature continues to be supported for advanced use cases and form configurations. Majority of form configurations can be accomplished by using SHACL, as described in the section above.

Forms created using web based form layout editor are stored in the `uiconfig.ui.ttl` file in the `dynamic` folder under the `server.topbraidlive.org` project. Note that form layouts are global to EVN. If a class has a custom form, all instances of the class will be using this form - irrespective of the vocabulary/asset collection they are in.

For advanced use cases please take a look at the SWA documentation (available from **Help > SWA Help/Examples**) for details on how to define custom form layouts via SWP. You can create an initial form with the web based form layout editor and then, if necessary, extend it further using TopBraid Composer.

Forms are attached to classes using `ui:instanceView` property. Since `uiconfig.ui.ttl` file doesn't import any ontologies, to find the class to which your form is attached to, you need to do one of the following:

- Temporary import your ontology file, then you will see the class in the Classes view of TopBraid Composer.
- Type the URI of the class in the global navigation field of the TBC toolbar, make sure to use <> around the URI.
- Use SPARQL. Run the following query in the SPARQL view. Then, in the results, select the subject (class) you are interested in.

```
SELECT *
  WHERE {
    ?subject ui:instanceView ?object .
  }
```

SWA documentation (**Help > SWA Help/Examples**) explains how to create nested widgets when you want to fold into a form information about related resources. For example, for an organization, in its address field you may want to directly display the street address, city and state - as opposed to a link to a resource holding this information. To do so, set the address property to be a blank node as shown below. This statement can be added to the `uiconfig.ui.ttl` file and will be effective globally. EVN will interpret it as a direction to nest the form for this property.

> *<http://www.example.org/address> swa:blankNodeProperty true .*

The layout of the nested form will be determined by the class in the range of the "blank node" property. In our example, this may be a class Address. When used in editing, if user enters information in the fields of the nested form, EVN will automatically create a blank node (e.g., instance of an Address) as a subject to hold information entered a nested form.

Another useful example of customization you can find in SWA documentation explains how to define dynamic ranges for some of the form fields. For example, if you need the values for countries to be dependent on the values for states.

Further, the SWP library used by EVN, the TopBraid SWP Application Components Library SWA, provides a flexible plugin mechanism that allows developers to add custom widgets to forms. Those widgets can insert arbitrary HTML, replacing the default widgets that render values as simple labels or hyperlinks. The SWA documentation (in the **Help > SWA Help/Examples**) provides some background. In the `evn-plugins.ui.ttlx` example file, navigate to **evn-plugins:ExampleMapObjectViewer** for an example Google Maps component. You can use this with the Geography example taxonomy that is shipping with TopBraid Composer.

Adding descriptive text below section titles

When using SHACL to define forms, the property-group sections of either the view or edit forms, custom text (e.g., a line of description, instructions, etc.) can be added below each section title by setting either of the corresponding properties: **view group description** or **edit group description** (cf. in code: `tosh:viewGroupDescription` and `tosh:editGroupDescription`).

Controling order and collapse/expand status of form sections

When using SHACL, you can elect to have a form section closed by default by setting the **open** field to false and **openable** field to true. Section will be initially presented as collapsed and users will be able to open it.

Configuring search text properties

The EVN advanced search forms by default include a global text field called "Search any Text" that searches across all string properties of the matching instances. For large models with a lot of strings, this may be quite slow. It is therefore possible to limit the properties on a class-by-class basis, for example by searching only within a preferred and alternative names, but not descriptions or other text properties.

To configure this behavior and specify the properties to be searched by this field, execute the following steps in TopBraid Composer:

1. Create an empty SWP file with a name ending with `ui.ttlx` —for example, GeoSearchForm.ui.ttlx

2. Import the SWA namespace (swa.ui.ttlx) and the ontology defining the class that **Search Text Properties** will search field into the file.
3. For each property to be searched by the **Search Text Properties** field, add a triple of the form `{?class swa:fullTextSearchProperty ?property}` to the file, where `?class` is a class you want to limit the search for and `?property` is the property to include in the list of properties to search.
4. Upload the file to the EVN server when done.

Adding custom actions on create

Custom plugins are used to modify behavior when new items are created.

For example, you may want EVN to perform additional initialization whenever a new taxonomy concept is created—to assign a new unique ID for each new concept or perform any other desired initialization. EVN provides a plugin mechanism that allows developers to add such initializations. See **evn-plugins:ExampleCreateConceptPlugins** for an example of such a plugin.

You can use a similar plugin to adjust the dialog box that appears when any new entity is created. For example, URIs of newly created instances of certain classes could be auto-generated. See **evn-plugins:UUIDCreateResourceDialogPlugin** for an example of such a plugin.

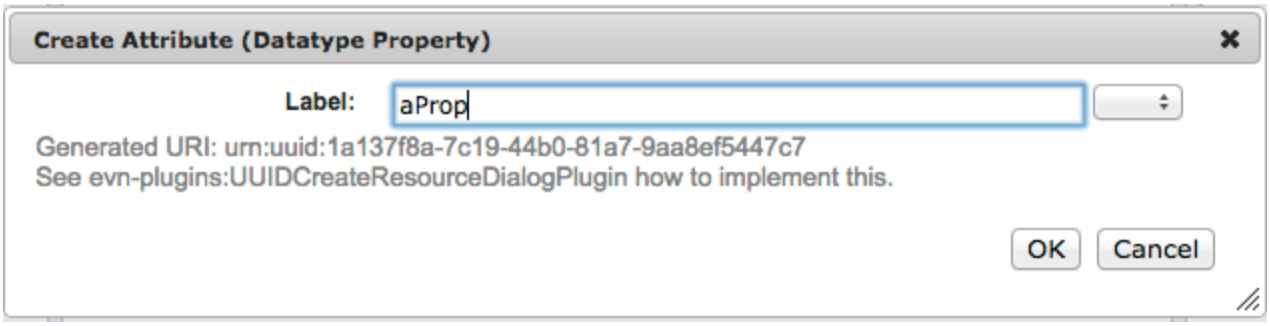
Similarly, EVN can be configured to perform additional initializations whenever a new vocabulary/asset collection has been created through the user interface. Among others, this can be used to automatically pre-populate the user permissions (in the .tch. graph) with certain user groups. See **evn-plugins:ExampleCreateProjectPlugin** for an example of such a plugin.

Plugins for initializing new resources

EVN can be configured to perform additional initialization whenever a new resource is created—for example, to assign a new unique ID for each new concept. EVN provides a plugin mechanism that allows developers to add such initializations. See **evn-plugins:ExampleCreateConceptPlugins** for an example of such a plugin.

Custom plugins for dialog boxes

The dialog box that appears when new items are created can be customized. This applies to any item e.g., a taxonomy concept, a class, an attribute, a relationship, an instance of any class. For example, if we auto-generate URIs of newly created items, we wouldn't want to have the URI field be editable in the dialog. Instead, we want to generate and display it. See **evn-plugins:UUIDCreateResourceDialogPlugin** for an example of such a plugin.



Custom plugins for creating vocabularies/asset collections

EVN can be configured to perform additional initializations whenever a new asset collection has been created through the user interface. Among other tasks, this can be used to automatically pre-populate the user permissions (in the .tch. graph) with certain user groups. See **evn-plugins:ExampleCreateProjectPlugin** for an example of such a plugin. The following image shows the custom dialog box from the example in `evn-plugins.ui.ttlx`. This dialog uses a UUID() generated namespace and does not allow the user to change that namespace. Defining static namespaces is also possible.

Custom plugins for cloning vocabularies/asset collections

EVN can be configured to perform additional steps after a vocabulary/asset collection has been duplicated via the Create Cloned Version action. This is done by creating a new subclass of the abstract class teamwork:ClonePlugins, and by defining the custom behavior using SWP in the ui: prototype of the new class. The arguments for prototype can be looked up in teamwork:ClonePlugins. To enable the new plugin for a particular project type, it must be added as a value of teamwork:clonePlugin to the teamwork:ProjectType instance.

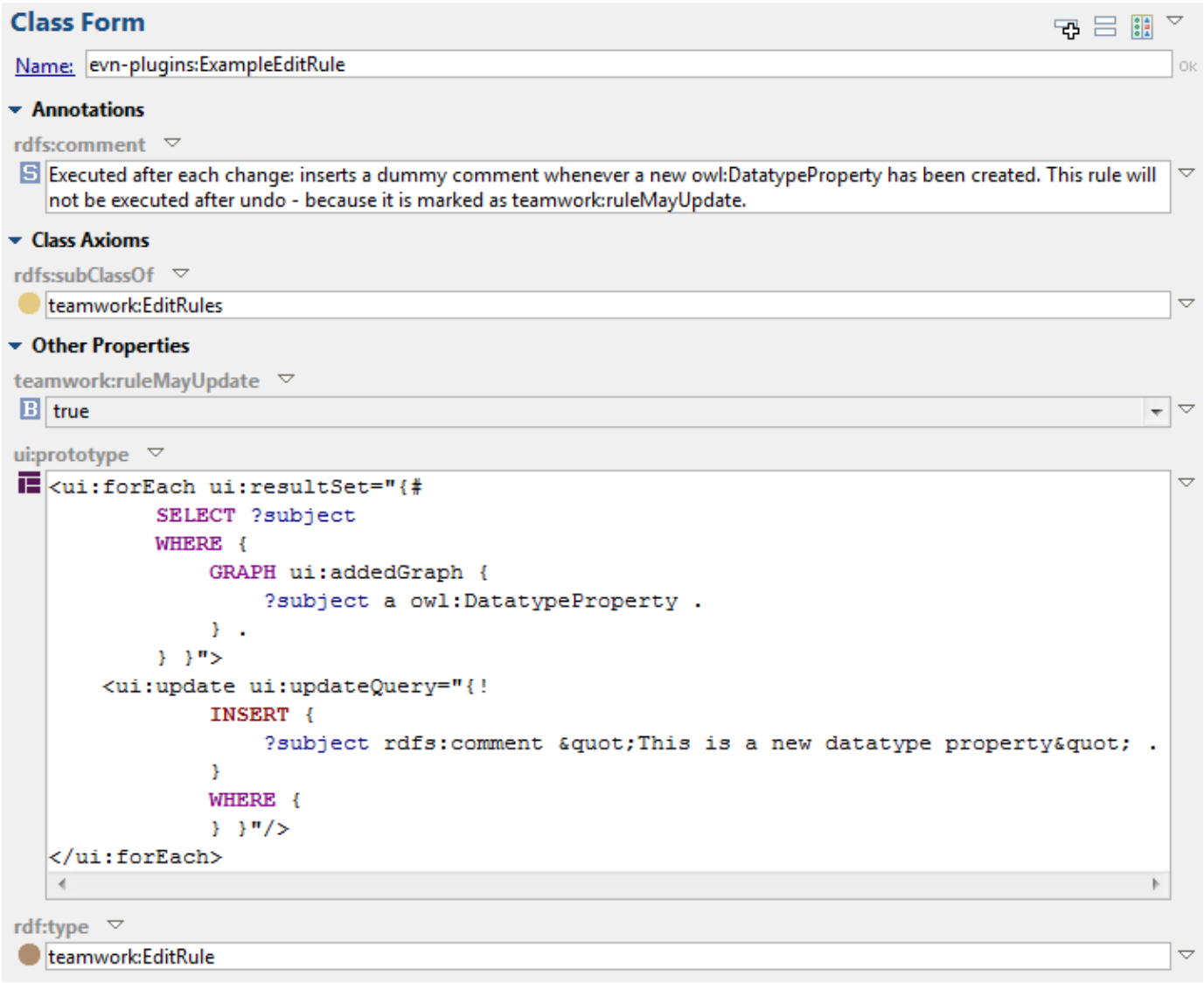
Adding Event-Condition-Actions Rules

TopBraid EVN includes a simple Event-Condition-Action (ECA) rule engine that can be used to trigger rules whenever a change was performed on an EVN project. Example use cases of this feature include anything that needs to happen as a side effect of edit operations:

- updating an external index (such as Solr)
- creating automatically generated triples

- aligning inverse properties
- sending automated notifications to co-workers
- ...

The following screenshot defines a rule that inserts an `rdfs:comment` whenever a new `owl:DatatypeProperty` has been created by someone.



As shown above, these ECA rules are essentially SWP elements that define their behavior in their `ui:prototype`. To create your own rule, simply create a globally registered `.ui.*` file that imports the teamworkrules system ontology, and then create a subclass of `teamwork:EditRules`. The rule engine of EVN will execute the `ui:prototype` of any globally registered edit rule after each edit. During the execution of those prototypes, there are two dedicated named graphs with a special meaning: The graph `ui:addedGraph` will contain the newly added triples, and `ui:deletedGraph` will contain those triples that have just been deleted. For example, if someone has pressed Save Changes on an EVN form, replacing one `skos:altLabel` with another, then the `ui:addedGraph` will contain the new value, and `ui:deletedGraph` will contain the old value. You can use SPARQL to query those change triples and perform side effects with `ui:update` or SPARQLMotion modules.

Note that the default graph of the WHERE clause is the active EVN graph but *without* its imports. To operate on the imports closure, use a WHERE clause such as:

```
WHERE {
{
BIND (ui:graphWithImports(ui:currentQueryGraph()) AS ?graph) .
}
GRAPH ?graph {
...
}
}
```

EVNuses this rule engine out of the box to update a Solr index after each edit (if Solr has been activated for the current project). For more information and examples, see the Blog entry that introduced the feature.

Status change rules

It is possible to define SWP-based rules that are triggered whenever the status of a working copy changes, e.g. from Uncommitted to Frozen. The `evn-plugins.ui.ttlx` file contains an example of such a rule, which will send an email to all editors or managers of a given working copy whenever its status changes. Take a look at `evn-plugins:SendEmailsOnTagStatusChangeRule`. In a nutshell, the rule's prototype will be executed and side effects can be performed there. The prototype of the rule can find out what has changed via the special named graphs `ui:addedGraph` and `ui:deletedGraph`. Most SPARQLMotion modules such as `sml:SendEMails` can be used within SWP.

Customizing the Governance Roles

To customize the list of governance roles:

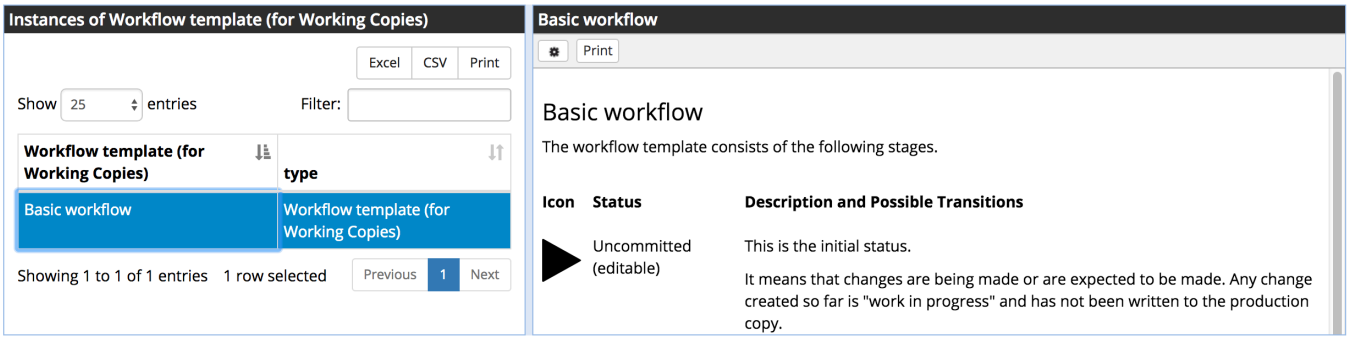
1. In TBCME, create a new .ttl file, open it, and import edg.topbraidlive.org/1.0/schema/SCHEMA_EDG-governance-core-v1.0.ttl
2. To disable existing governance roles:
    1. Navigate to the class edg:Stewardship
    2. Navigate to the property constraint for the role to be disabled, e.g., edg:Stewardship-applicationSteward
    3. Set sh:deactivated to true
3. To add a new governance role:
    1. Navigate to edg:WorkflowParticipantProperty
    2. Create a new instance of this class
    3. Choose its URI, rdfs:label, and rdfs:comment to represent the new governance role
    4. Navigate to the class edg:Stewardship
    5. On this class, create a new property constraint with the new property as predicate
    6. Optionally, navigate to the new constraint and set its sh:order to adjust the order in which the governance roles will appear in the EDG/EVN user interface
4. In EDG, navigate to the Import/Export tab of the Governance Model
5. Import RDF File, select the created file, check “Record each new triple in change history”

To undo the customization at a later stage, go to the Change History of the Governance Model, find the Change corresponding to the import, and revert it.

Adding Custom Workflow Templates

The stages that a workflow goes through are defined by in a workflow template. The Workflow Templates page allows users to view and, if permitted, modify the workflow templates. The workflow templates are defined in RDF in a dedicated graph, accessible via the function `teamwork:workflowsGraph()` in SPARQL queries. Users with a Teamwork Administrator Role in the EVN Configuration Parameters page are permitted to edit workflow template definitions.

Editing workflow templates is a low-level operation that should be handled with care. It is technically possible to modify workflow templates that are already in use and, as a result of this, an existing working copy may be in a state that gets disconnected from the other states in the template, or its whole workflow template may get deleted. We therefore strongly recommend that the development and editing of workflow templates is performed by expert users only, and tested in a sandbox environment (such as a TBC-ME instance) before deployment to a production server. Alternatively, consider asking TopQuadrant support staff for help.



Download as Turtle File Upload Turtle File

The editing of workflow templates is currently performed via TBC or source code editing. For Teamwork Administrators, the basic mechanism is:

1. Go to Workflow Templates on the production server
2. Click `Download as Turtle File` to a local file
3. Edit this file using TobBraid Composer (or a text editor of your choice). Note that also attached to this page (see below), is an example template file, which has workflow diagram layouts, with positioning, widths, and heights.
4. In a test environment, use `Upload Turtle File` to replace the current definitions
5. If any errors are detected, the Upload feature present a report of all the errors found in the file. These errors must be addressed in order for the file to be successfully uploaded.
6. Carefully test the workflows in your test environment
7. When satisfied, use `Upload Turtle File` to replace the definitions on the production server

Defining workflow template status

Each workflow template defines a start status (e.g. Uncommitted) and a set of transitions that specify which steps are possible and by whom, until the workflow is ended (e.g. by committing). The following list enumerates frequently used properties at workflow templates:

- **rdf:type:** workflow templates must have rdf:type `teamwork:TagWorkflowTemplate` (for general asset workflows), `teamwork:ExistingResourceTagWorkflowTemplate` (for workflows that are specific to an existing asset), or `teamwork:NewResourceTagWorkflowTemplate` (for workflows that create a new asset).
- **rdfs:label:** a human-readable label that is used to name the workflow template in the user interface
- **teamwork:defaultTagWorkflowTemplateForProjectType:** enumerates the asset collection types for which the workflow template is the default. Asset collection types that don't have any default workflow template fall back to the system-wide default workflow template. For example, use `edg:GlossaryProjectType` as a value to make the workflow template the default for EDG Glossaries.
- **teamwork:suitableWorkflowForProjectType:** enumerates the asset collection types for which the workflow template is suitable for use. If left empty, it applies to all types. For example, use `edg:GlossaryProjectType` as the value to make the workflow template only applicable to EDG Glossaries.
- **teamwork:suitableWorkflowForSubjectArea:** the subject areas that asset collection must have so that the workflow template is suitable for them. If left empty, then it applies to all subject areas. Details about defining subject areas can be found at EVN Governance Model.
- **teamwork:editorWorkflowParticipantProperty:** lists the workflow participant properties (e.g. `edg:responsible`) whose users are automatically granted editing permission. By default, workflow participants only get viewer permission.
- **teamwork:managerWorkflowParticipantProperty:** lists the workflow participant properties (e.g. `edg:responsible`) whose users are automatically granted manager permission. By default, workflow participants only get viewer permission.
- **teamwork:initialStatus:** specifies the initial status of a Workflow. Example values include `teamwork:Uncommitted` or `teamwork:New`.
- **teamwork:tagShape:** (advanced feature) SHACL shapes that the working copy needs to conform to before it can be committed. These are validated using the union of the working copy and the TCH graph as data graph, and the workflows definitions graph as shapes graph. The focus node is the `teamwork:Tag` instance. There is an example below.

An additional property for templates that have the type `teamwork:ExistingResourceTagWorkflowTemplate`:

- **teamwork:editedResourceShape:** the shape that edited resources need to have. Used to filter applicable workflows for a given resource. If multiple values are present then they are interpreted as a union, i.e. a candidate resource must conform to any one of them. An example of this is `teamwork:editedResourceShape [ sh:class ex:Person ]` to limit a workflow for instances of `ex:Person`.

Defining workflow template transitions

A workflow is started in the status specified by `teamwork:initialStatus` in the template. From there, the possible transitions are defined by the values of `teamwork:transition`. These values must be URI or blank nodes of type **teamwork:TagStatusTransition**, and can have the following properties:

- **teamwork:fromStatus:** the status that the workflow (working copy) must be in for the transition to be applied.
- **teamwork:toStatus:** the status that the workflow (working copy) will be in after the transition is applied.

- **teamwork:transitionLabel:** the display label of the transition, e.g. "Accept changes to production".
- **sh:order:** an optional number that is used to order the transitions in the Workflow tab. Transitions with smaller numbers show up higher, with 0 being the default.
- **teamwork:requiredGovernanceRole:** the minimum governance role (e.g. informed) that a user must have on an asset collection to perform the transition. For example, the required governance role can be an instance of `edg:JobRole`, which means that any user who has `edg:assignedJobRole` `edg:JobRole` is allowed to execute the transition. The user will in this case see a corresponding button in the UI, and under My Workflows.
- **teamwork:requiredProjectPermissionRole:** the minimum permission role (e.g. editor) that a user must have on an asset collection (master graph) to perform the transition. These are the roles defined on the User Roles tab of an asset collection.
- **teamwork:requiredTagPermissionRole:** the minimum permission role (e.g. editor) that a user must have on a working copy to perform the transition. These are the roles defined on the User Roles tab of a working copy/workflow.
- **teamwork:minVoteCount:** the minimum number of votes required before the transition can be performed.
- **teamwork:voteAutoTransitions:** if true then the transition will be performed automatically, as soon as sufficient votes have been registered. This property can only be used if `teamwork:minVoteCount` is present.
- **teamwork:autoTransitionHours:** the number of hours after which a working copy will automatically transition to another state. For example this can be used to give users a chance to comment without explicitly waiting on such feedback.

Example workflow-template file with diagram layouts

This example workflow template file (`create-dataset-schema.ttl`) shows diagram layout specifications with (x,y) positioning, widths, and heights: 

Example of teamwork:tagShape

This example illustrates the use of `teamwork:tagShape` to specify that a working copy can only be committed if it contains exactly one new instance of `edg:GlossaryTerm` and no other subject. If `teamwork:tagShape` statements are present for a workflow template, then the Commit button will force the user to first go through a dedicated page that checks whether the working copy conforms to the specified shapes. If violations are detected, the user may still proceed, yet he or she has been warned about the consequences.

```
  edg:SimpleGlossaryTermWorkflowTemplate
    rdf:type teamwork:NewResourceTagWorkflowTemplate ;
    teamwork:initialStatus teamwork:Uncommitted ;
    teamwork:tagShape glossary-term-workflow:TagHasExactlyOneGlossaryTermShape ;
    ...

glossary-term-workflow:TagHasExactlyOneGlossaryTermShape
    rdf:type sh:NodeShape ;
    sh:sparql [
        sh:message "Working copy contains changes about {?other} while it should only be about {?newTerm}"
;
        sh:prefixes <http://edg.topbraidlive.org/1.0/samples/workflows/glossary-term-workflow> ;
        sh:select """
            SELECT DISTINCT $this ?other ?newTerm
            WHERE {
                ?change teamwork:tag $this .
                ?change a teamwork:Change .
                ?change teamwork:added ?added .
                ?added teamwork:subject ?newTerm .
                ?added teamwork:predicate rdf:type .
                ?added teamwork:object edg:GlossaryTerm .
                ?otherChange teamwork:tag $this .
                ?otherChange a teamwork:Change .
                ?otherChange teamwork:added|teamwork:deleted ?aod .
                ?aod teamwork:subject ?other .
                FILTER (?other != ?newTerm) .
            }""" ;
    ] ;
    sh:sparql [
        sh:message "Working copy does not create a new instance of edg:GlossaryTerm." ;
        sh:prefixes <http://edg.topbraidlive.org/1.0/samples/workflows/glossary-term-workflow> ;
        sh:select """
            SELECT $this
            WHERE {
                FILTER NOT EXISTS {
                    ?change teamwork:tag $this .
                    ?change a teamwork:Change .
                    ?change teamwork:added ?added .
                    ?added teamwork:subject ?gt .
                    ?added teamwork:predicate rdf:type .
                    ?added teamwork:object edg:GlossaryTerm .
                }
```

```
        }""" ;
] .
```

Getting Started with Custom Data Quality Rules in EVN

Constraint Templates in EVN define parameterized business rules for valid data values. The business rule created from the reusable template will in turn be executed when data is created or modified. EVN will alert the user if the rule fails as a result of a data value violation. New business rules may be introduced at any time and any collection's **Reports > Programs and Suggestions** report will uncover any old data that violates the newly added, or modified, rule. For more information please see Using SHACL Data Constraints In The TopBraid Web Products guide.

The TopBraid Teamwork Framework

Getting Started with TTF

This section provides a basic overview of the TopBraid Teamwork Framework (TTF) and its motivation. See also the dynamically generated list of Teamwork SPARQL functions, accessible from TopBraid Composer's Help Menu.

You will need to understand TTF if you want to:

- Programmatically access change history or working copies
- Create new vocabulary types for **$ProductFullName**
- Make certain modifications to options available for pre-built vocabulary types

Why TTF?

TTF is a development and runtime framework for web-based multi-user read-write applications for different types of data assets or vocabularies.

For example, an application to work with SKOS-based Taxonomies consists of pages to create a new taxonomy, to manage user permissions, to track and review changes, to edit the taxonomy itself, to produce various reports and to run data exports and imports. Many of these capabilities are also applicable to other kinds of vocabularies, not just SKOS taxonomies. The role of TTF is to generalize those capabilities so that it becomes more efficient to develop similar applications in a componentized way.

Overview of TTF

TTF **projects** consist of two workspace files that connect to a database for data storage. The first database contains the actual data of the "master" graph. The second database, ending with ".tch" contains metadata that is used to keep track of changes, permissions and other vocabulary specific settings. Details of these files and their content are provided by Graphs, Permissions and Change Tracking.

TTF **vocabulary and asset types** (sometimes called "project types") are the kinds of models that are available to the end user.

Each vocabulary type is defined as a plugin to the teamwork framework. In a nutshell, each vocabulary type is backed by an instance of the class `teamwork:ProjectType` which is stored in an SWP file and points to the specific editors, plugins and other features that distinguish this vocabulary type from others. If you are interested in adding your own vocabulary types, see section: Vocabulary/Asset (Project) Types. You can also modify features available for the pre-defined vocabulary types.

EVN home pages list all installed vocabulary/asset types. When a user clicks on the available type, they see a list of all vocabularies/assets of that type. From there, they have access to the available features.

Graphs, Permissions and Change Tracking

A combination of two related graphs, one containing the data and another one containing the change history of the data plus other information, is called a Teamwork project. This should not to be confused with the term Eclipse project. When using EVN, every vocabulary/asset is a Teamwork project. For the purposes of this discussion, Vocabulary/Asset is synonymous with Teamwork project.

This section introduces the two kinds of RDF graphs involved in a Teamwork project.

Anatomy of a Teamwork project

Each Teamwork project consists of two graphs that have corresponding `.sdb` connection files in the TopBraid workspace. You can explore those files by opening the files created via EVN within TBC-ME. (When doing this, note that you should not close either of the two .sdb files because this may invalidate the connection with the localhost server).

- The **master graph** (e.g. `XY.sdb`) contains the actual data such as the edited SKOS concepts, also known as a "production copy".
- The **TCH (or team) graph** (e.g. `XY.tch.sdb`) contains metadata about the project, including the change history, available working copies, user permissions and comments.

TopBraid identifies Teamwork projects by their graph URIs: `urn:x-evn-master:XY` is the master graph for the vocabulary with the short id "XY". Its metadata would be stored in `urn:x-evn-master:XY.tch`. The teamwork namespace contains SPARQL functions that can be used to derive those URNs from ids and to extract ids from a given URN. These functions are the preferred way of working with those graphs. You can find a complete list of those functions in Appendix: Teamwork SPARQL Functions Reference.

The **project id** (sometimes called **graph id**) is the short name of the vocabulary, usually consisting of letters only.

Permissions

All users with access to the surrounding TopBraid Live server can log into the Teamwork pages. However, who is allowed to do what can be controlled by the system administrator as well as the corresponding vocabulary managers.

Teamwork supports the following three roles per vocabulary:

- **viewer**: can only view a vocabulary or working copy, and make comments.
- **editor**: same as viewer but also has editing rights to make changes to a vocabulary or working copy. See Working Copies (Tags) for a definition and description of working copies.
- **manager**: same as editor but also has administrative control over a vocabulary or working copy, including the permission to grant or deny permissions to other users.

Any user can create a new vocabulary, and will become the initial manager of this project. At any given time, there must be at least one manager per vocabulary. The vocabulary home page can then be used to grant additional permissions to other known users. All of those permissions are stored in the TCH graph. For example if Administrator is the manager of the Teamworks project with the id XY, then the TCH graph will contain the following triple:

```
<urn:x-evn-master:XY.tch> teamwork:manager <urn:x-tb-users:Administrator>
```

As shown above, the URIs of the users are always following the same pattern. Use the SPARQL function `smf:currentUserName()` to access the currently logged in user, and `smf:userWithName(?userName)` to construct the user URIs as above.

In addition to assigning permission roles to individual users, it is also possible to assign roles to security roles from LDAP or tomcat. The UI provides simple drop down boxes for this. In terms of the internal RDF representation in the TCH graph, an LDAP role is represented with a URI such as`urn:x-tb-role:ROLE`.

You can get the union of all master graphs that the given user (in the example, : Administrator) has at least read access to by using special named graphs of the pattern `urn:x-evn-union<:user>`. For example, `urn:x-evn-union:Administrator`. This can be convenient for global search across multiple graphs. Note that such graphs are strictly read-only.

Change tracking

A change is an atomic unit of modifications to a given graph. They are usually created by the user through buttons and menu items such as the Create Concept buttons in EVN or the Save Changes button of the SWA edit forms. Whenever the user sends such a request to the server, master or working copy graph will be updated and the TCH graph will be updated with a new instance of `teamwork:Change`. An example Change object is shown below (in Turtle):

```
<urn:x-change:2013-04-03T00-32-05.915ZAdministrator>
        a       teamwork:Change ;
        rdfs:comment "Create Class with URI http://example.org/vocabulary1/ExampleClass"^^xsd:string ;
        dct:created "2013-04-03T00:32:05.915Z"^^xsd:dateTime ;
        sioc:has_creator <urn:x-tb-users:Administrator> ;
        teamwork:added
        [ teamwork:object owl:Class ;
                teamwork:predicate rdf:type ;
                teamwork:subject <http://example.org/vocabulary1/ExampleClass>
          ] ;
          teamwork:added
          [ teamwork:object "Sub3"^^xsd:string ;
                teamwork:predicate rdfs:label ;
                teamwork:subject <http://example.org/vocabulary1/ExampleClass>
          ] ;
          teamwork:added
          [ teamwork:object owl:Thing ;
                teamwork:predicate rdfs:subClassOf ;
                teamwork:subject <http://example.org/vocabulary1/ExampleClass>
          ] ;
          teamwork:status teamwork:Committed .
```

This is a set of changes indicating that the user:

- Created a new class with the id `http://example.org/vocabulary1/ExampleClass`
- Set the label of the new class to "Sub3"
- Made the class a subclass of `owl:Thing`

A change may also point to a working copy via `teamwork:tag` if it has been made while the user was editing a working copy. Each added or deleted triple is stored in reified form via `teamwork:subject`, `teamwork:predicate` and `teamwork:object` as shown above. This makes it possible to find all changes for a given resource via a SPARQL query over the TCH graph. Note that `teamwork:deleted` is used to point at triples that were deleted.

The `teamwork:status` triple of a change indicates whether the change has been committed to the master copy or whether it only exists in a working copy - in which case the value would be `teamwork:Uncommitted`.

Working copies (tags)

A **working copy** (internally often called **tag**) is a specific collection of changes that have been made to the master graph, but have not been saved in the master graph. Since all these changes are stored only in the TCH graphs, the working copy is a virtual construct that is computed on demand only. TopBraid has a mechanism that can temporarily apply a set of changes to an RDF graph (without modifying the underlying graph). All changes that are part of a working copy are associated with the working copy via the property `teamwork:tag`.

In order to access a specific working copy for a given graph (with id "XY"), and tag "myTag") use a graph URI such as `urn:x-evn-tag:XY:myTag:Administrator`. In general, although it is possible to create such URIs "manually", the preferred mechanism is through the SPIN function `teamwork:queryGraph` as in the following example:

```
SELECT ?concept
WHERE {
        BIND (teamwork:queryGraph(true, "XY", "myTag", "Administrator") AS ?graph) .
        GRAPH ?graph {
                ?concept a skos:Concept .
        }
}
```

Comments and Tasks

EVN forms have a button to add a comment and/or task to the currently selected resource. These items are stored in the TCH graph, as instances of `sioc:Post` where the `sioc:about` property points at the resource. You can explore details by looking at the TCH graph.

Vocabulary/Asset Collection (Project) Types

This section explains how developers can add their own types of collections as alternatives to the pre-existing collection types defined for TopBraid EVN, such as SKOS *Taxonomies,* or *Content Tag Sets*. All collections of the same type share the same edit application and the same plugins. Note that defining custom collection types is a licensed feature of EVN. To upgrade a license file, please contact TopQuadrant.

Creating a new vocabulary/asset collection (project) type

Each of the default collection types that come with TopBraid EVN are plug-in modules within the more general Teamwork Framework. It is possible to add new modules by creating an instance of the class `teamwork:ProjectType` in a globally registered SWP file (ending with `.ui.*`). This file should import the file `teamwork.ui.ttlx`. The best practice is to create a new Eclipse workspace project and place your file within it.

Note that while you will be able to develop and test new project types in TopBraid Composer, once you deploy your code to the EVN server, new types of collections will only be available if your solution is licensed for creation of new asset collection types.

Simply create an instance of `teamwork:ProjectType` and fill in the properties similar to how it's done in the default Ontology application (from `ontologyprojects.ui.ttlx`). You need to enter values for `teamwork:singularLabel` and `teamwork:pluralLabel`. You also need to enter value for the `teamwork:vocabularyType`. These values must be subclasses of the `teamwork:Vocabulary` class. For the ontology projects this value is ontologyprojects:Ontology. Under teamwork:Product select the one instance for EVN and add the "yourprojectype": ProjectType to the defaultProjectType property

Once you have filled in the minimum properties for your new ProjectType, make sure that the framework "knows" about your new file by selecting **Model > Refresh global SWP graphs...**. Then you should already see a new tab for your vocabulary type on theEVNhomepage, on the same level as any other pre-existing vocabulary types such as Ontologies.

Create-new pages

The section for your new vocabulary/asset collection types on the home page will have a link to create a new vocabulary/asset collection of your type. The framework will use the singular or plural labels that you have specified wherever possible, so the link may be "Create New My Custom Collection" if that's how you have called it. The link will by default go to a default create page where the user can enter label, default namespace and a description of the new asset collection.

If you need additional parameters at creation time, take a look at how this is done in the case of Crosswalks, in `crosswalkprojects:CreateProjectPage`.

Project plug-ins

As you might know, the main page of any collection's production or working copy contains links to various "utilities" features such as Import /Export, Report, etc. These come from plug-ins. You may want to explore the standard plug-ins shipped as part of the Teamwork Framework, e.g. `teamwork:TurtleFileExportPlugin`.

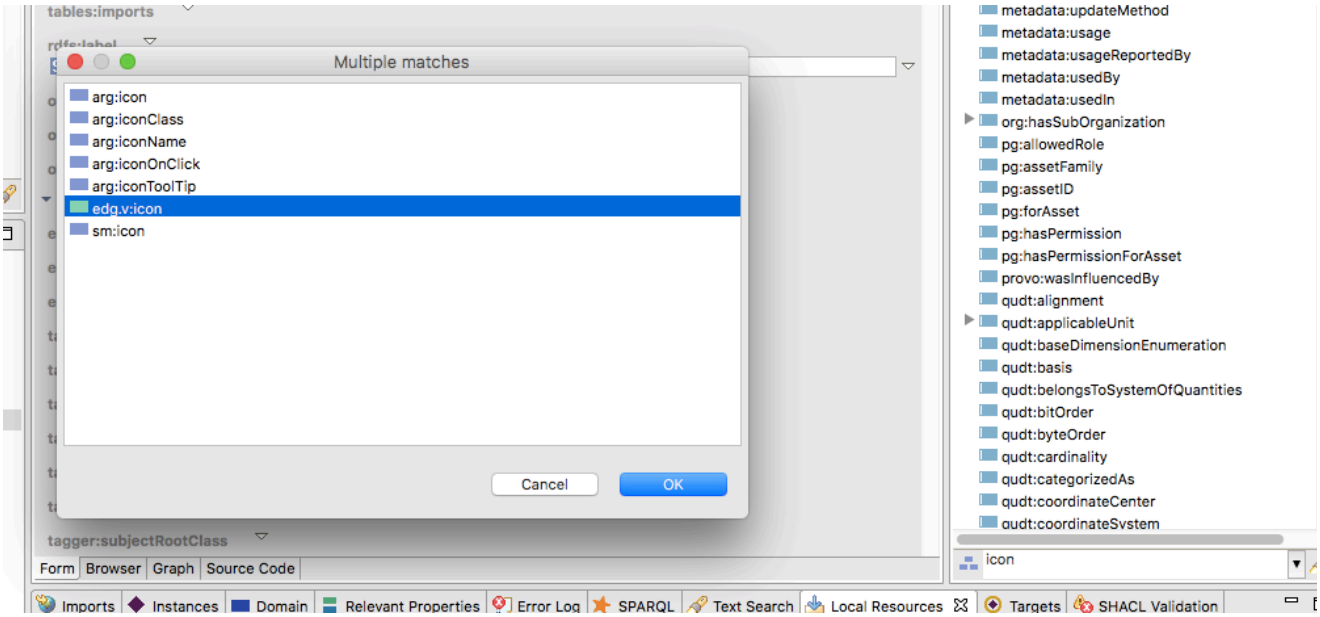
In a freshly created `teamwork:ProjectType`, no plug-in would show up. You need to select those plug-in that you want to expose via the `teamwork:projectPlugin` property.

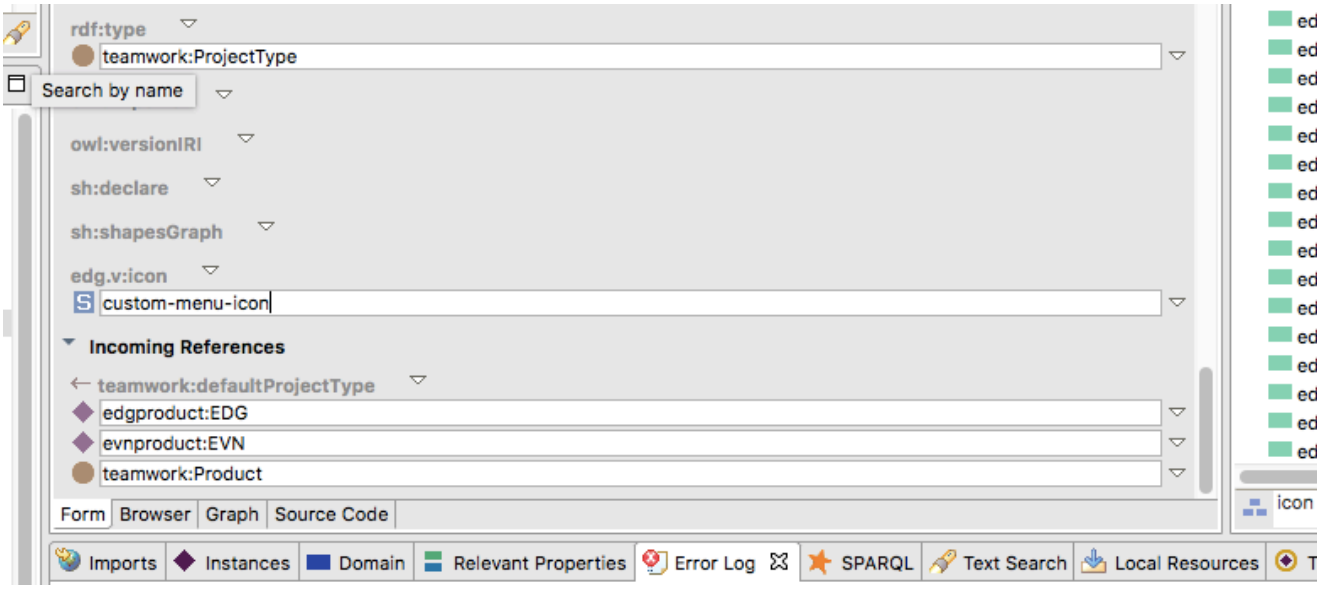
There is also a generic SPARQLMotion-based importer mechanism that applies to all vocabulary types. Such importers receive an uploaded file as input and produce triples that shall be added to the current vocabulary or working copy. In order to create such importers, store them in a file ending with `.sms.*`. Import the file `teamworkscripts.ttl` and use the properties defined therein to link your `sm:Function` either with a vocabulary type or specific vocabularies for which the function shall show up. The script needs to end with `sml:ReturnRDF` and take exactly one string argument - the body of the uploaded file.

Custom icon for project type

To add a custom icon to the project type:

Search the properties pane for edg.v:icon and drag this property onto your new project type resource form, give it a value.





The value for custom icons must start with the string "custom-" or they will be ignored and the default icon (a circle) will be used. Add a style block which defines your custom icon style like seen below. This can live in any external stylesheet. We suggest you create your own "*.www" directory to hold these types of customizations.

```
#primaryNav .custom-menu-icon {
background-image: url("https://maxcdn.icons8.com/Share/icon/Messaging//online1600.png");
}
```

(Ensure that the sytlesheet is loaded in the application pages.)

Extending the *Problems and Suggestions* Reports

The *Problems and Suggestions* report feature implements data quality rules for collections (see a collection type's User Guide **...Utilities > Reports > Problems and Suggestions Report** for documentation). This feature includes an extension point allowing developers to plug in their own algorithms written in Java or SWP. For an example of SWP-based suggestion generators, in TBC open `tosh.ui.ttlx`, select `tosh:ReportsGenerator` and go to **Resource > Find usages in workspace**. Use a copy of the example(s) as a starting point.

Notifications

The Teamwork Framework includes a notifications mechanism that can be configured to send notification emails (and other notification) to subscribers. The process to set this up consists of two steps:

1. Assign organizations, persons, users or user groups to RACI roles using the Metadata tab
2. Select which notification events shall be triggered for which RACI role

New notification types can be added by creating subclasses of `teamwork:Notifications` in a `.ui.` file. There are two ways of triggering notifications:

- Manually, by calling the SWP element `teamwork:sendNotifications`
- By hooking into direct changes against the master graph (subclassing `teamwork:MasterEditNotifications`) or the TCH graph (subclassing`teamwork:TCHEditNotifications`). In those cases, an `arg:expression` needs to be specified to determine whether the notification must be sent for the given set of changes.

The file `teamworknotifications.ui.ttlx` contains plenty of examples.

In order to have notifications sent to other channels than emails, you need to create a new SWP file with `.ui.ttlx` ending, and define a subclass of`teamwork:sendNotifications` that points at `teamwork:sendNotifications` via `ui:overrides`. In the `ui:prototype` of that subclass, you may repurpose the incoming arguments such as `?text`, for example, to call `sml:ExportToJMS` for output into a JMS bus.

Editors

Editors are the SWP applications that are loaded when the user clicks on **Edit Production XY** or **Edit Working Copy**. While these editors can in principle be any type of web interface, it is often the easiest to start with the procedure below.

Creating an Editor

The easiest way or creating a new editor is to take an existing editor as a starting point. Use **File > New > SWP Application from Template** and select from one of the available options. For example, if you want to base your new editor on the ontology editor, select **ontologyapp**. Then link your `teamwork:ProjectType` with that editor via the `teamwork:projectEditorClass` property. A side effect of this procedure is that you can in principle reuse the same editor class for different vocabulary types.

From there on, your copy of the Ontology editor application can be modified by adding, changing or removing gadgets and their surrounding windows. The SWA user guide should provide you enough information to get started with that.