Documentation SurfTheOWL as Django Project

Requirements:

Python 3 or higher:

Owlready2 installation via command "`pip install owlready2`"

Django installation via command "`pip install django`"

Start Developern Server

Double click the script "`start SurfTheOWL.py`" a localserver is set up and your browser calls the localserver. Additionally, a small GUI opens in which you can terminate the local server, by clicking on the big red button.

Structure

The Script **manage.py:** is the master script which controls the localsever and manages the different scripts.

Directory **SurfTheOWL_django_project:** is the main directory of the website. It contains the settings.py in which all setup relevant data is stored. The most important script in her is urls.py it contains all URLs and directs request to the right views.py function, in the app directory. Urls.py is the script which is responsible for the routing on the website.

Directory **SurfTheOWL** is the app directory**:** views.py holds the displayed responses according to the URL requests. SurfTheOWL.py is the actual script which interacts with the "`TriboDataFair….owl`" Ontology. In the subdirectory "`templates`" is the html template "`SurfTheOWL.html`" which is manipulated by Django and displayed in the browser. All styling is done via css from the subdirectory "`static`" with the file "`SurfTheOWL.css`". The script "`FindDublicateIDs.py`" is used to determinate if a class in the Ontology has a duplicate TDO id.

Start SrufTheOWL.py: is a little script which let you conveniently start the local server an opens the local server in your main web browser. Just double click it,

db.splite3: contains a not used sqlite3 database.

Website workflow

When the local server is started, the `TriboDataFAIR_Ontology.owl` is loaded and the function `get_seachable_classes_from_list(Kadi4Mate_objects)` is called. The function delivers all related OWL objects which have an association with TriboDataFAIR.Kadi4MateRecord as a list of pairs. `[[ ObjectName , ObjectFriendlyName]]` as variable `searchable_owl_classes`. the `ObjectFriendlyName` is used to display it in the browser, the `ObjectName` is necessary for the code to find the related Objects like subclasses (liked by properties) and instances of a given Class.

At the start in line 8 the path to the used `.owl` is defined, this path is used in line 10 to load the Ontology. If the name of the OWL or the path changes, the path (l. 9) and nam e(l. 7) must be changed. In addition to the Function "`get_seachable_classes_from_list(Kadi4Mate_objects)`" the Function "`get_all_classes_as_list(TriboDataFAIR.classes()")` is called and saved a variable "`all_owl_classes`". The list contains all classes which are contained in the OWL and provides the opportunity to make sure that a wanted class is an actual OWL object. This functionality is later used when we search all related classes of one class, to prevent that the code wants to call a not existing Class.

If you call the local server in the web browser you are redirected from urls.py and call function "`welcome()`" in `views.py`. This function returns the static `Welcomt.html` template, in which is checked if the loaded Ontology contains a duplicate TDO id. This functionality is provided by the script "`FindDublicatIDs.py`". This script reads the provided Ontology as a simple `xml` file, by looping over all xml entries, and if the entry has a `pesindentID` that id is pushed in a Array. With a loop over the Array the Array is asked if the current loop ID is contained, if it is the ID is pushed in the Array `duplicate_ids`, which can after the loop be downloaded as JSON file on the appearing Button in `Welcome.html`.

If your are clicking on the Start Button you are calling the URL "`./landing`". Urls.py redirects the request and calls the function `landing()` in `views.py`. This function gets the variable `searchable_owl_classes` from `SurfTheOWL.py` and puts it in select fields in the html response SurfTheOWL.html. The select fields is inside a form container within the html-template `SurfTheOWL.html`.

By choosing one element in the select box and clicking the SEARCH Button the selected Class name is send to `ulrs.py`, which redirect it to the function `search()` in `views.py`. In the background of the selected Element is the Object Name bound to it, which is the value that is needed and send as variable `searched_class` to the `search()` function via POST. The variable searched_class is extracted and the function `main.search(searched_class)` in `SurfTheOWL.py` is called. It returns a JSON which contains the hole data tree. The hole data tree is processed with the function `generate_html_from_dict_via_recusion(data_tree, depth)`, which generates a html code which structures the data according to its position by using recursion to loop through all layers. The so generated html code is inserted in the html templated `SurfTheOWL.html`. The html-templated is than send as return to the web browser.

How the SurfTheOWL.py search for all related Classes functions.

The functionality is called via the function `main_search(className)` with a className as attribute. At first the defined contextual type of the Kadi4MateRecord-Class, by calling it with the provided owlread2 functionalities to refer defied Annotations by simply call it with de defined name.

If the className is contained in the OWL (l. 251) two empty dictionaries are generated, `classes_dict` and `friendly_classes_dict`. `classes_dict` is used to build the data tree with the real class names and `friendly_classes_dict` is used to build simulations to `classes_dict` a dictionary with the friendly Names of a given OWL classes. The main key in the `classes_dict` is the given function attribute which you selected in the select boxy, the first key in the `friendly_classes_dict` is the corresponding `friendlyName`. The Loop through all layers is done by a recursive acting function `find_classes_layers_via_recursion(layer, keys, friendly_layer, depth)`, the abortion of the recursion is implemented with a abortion statement, "`if depth == 10`" and depth represents the actual layer depth. Then for each key in the given Layer of the `classes_dict` the function `children(key)` are called. This function generates two dictionaries itself `children_classes_dict` and `friendly_names_dict`, the first is used to contain the subclasses ClassNames the second is used to contain the subclasses FriendlyClassNames. At first it is checked if the key has subclasses by calling the function `end_of_entries(className)`, which returns true if the key has no subclasses `(search_class(className) == true)` ore it is another special Object `(is_classs_refer_other_object(className) == true)` ore the key is a datatype (int, float, …) `(is_class_datatype(className) == true)`. If the key has subclasses these are searched and convert in a child dictionary by calling the function `dict.fromkeys(search_class(className))`. If the class have subclassed an instance both data is extracted and put in both dictionaries. The class instances are extracted by the function get`_data_instances(className)` which returns the Instance-name and the Instance-friendly-name so as the instance comment if it is provided by the Ontology.

The function `search_class(className)` uses the method "`.is_a`" provided by the package owlready2 to get the subclasses with it set restrictions from the Ontology. This list is further manipulated to also get not only the subclasses but also the referring to a special Object and the datatypes (<class (datatype)>, <object (object)>) and returns this list.

The so returned list to the function `children(className)` is than looped to get elements which contains a datatypes ore a special Objecst. If they are found these elements are manipulated to meet the requirements of an understandable dictionary. Therefore, the functions `is_class_datatype(className)` and `is_class_refer_other_object(className)` is used to provide the needed functionality. The `friendly_names_dict` is generated by calling the function `className_to_friendlyName(className)` which returns the defined `friendlyName` of a Class. if it is provided by the OWL. Comments defined in the Ontology are extract with the function `get_commetn_by_className(className)` which returns the defined Comment. These Comments are pushed in the `class_comment_dict` with the className as Key. The TDO id of a class is extracted with the function `get_ID_by_className(className)` and pushed in the `id_dict` dictionary with the className as key.

Now the function `children(className)` return the `children_classes_dict`, the `children_keys`, the `friendly_names_dict`, the list `other_object_refer_pair`, the `id_dict`, `class_comment_dict` and the `Instances_comment_dict` to the function `main_search(className)`. Here the `children_classes_dict` is assigned to the `classes_dict`, the `friendly_names_dict` is assigned to the `friendly_class_name` in the `main_search(…)` function. The main `id_dict`, `class_comment_dict` and `instances_comment_dict` are append with the returned according dictionary. The list

`children_keys` are used to call the recursive functionality and only loop over subclasses which have a subclass, because datatypes and special objects represents the maximal depth of this data-limb. If all subclasses are extracted the function `main_search(className)` manipulates the referred other objects and returns the dictionaries (`classes_dict, friendly_classes_dict, special_objects_freindly, id_dict, comment_dict, contextual_type_definition_friendly, instance_comment_dict`) to the function `search(request)` in views.py. There the html code is generated and served to the browser. The recursive loop over the friendly_name_dict arrange the Elements in the html code in div container, which has a style indentation to make the hierarchy clear. Therefore, also a dotted line is used which is arranged on the container side and a sort vertical on the top. These lines appear in the Browser as if they are one unit, the lines also represent the dependencies. If a Element can determined as Magnitude (unit and datatype) the both elements are cast in a table an displayed beside each other. To the className a tooltip is added which contains the according information form the `id_dict` and `comment_dict`. Individuals from instances are generated as table. If the instances are contained in the `instances_comment_dict`, these comments are added to the instances as tooltip.

Webbrowser HTML template

views.py

SurfTheOWL.py

Start SurfTheOWL

welcome(request)

OWL_master_name
OWL_path

FindDublicateIDs.py
check_ids(owl_file_path)
Returns a list with founded dublicate TDO ids

Press Start Button

landing(request)
inserts the elements of searchabel_owl_classes in the select fields.

get_all_classes_as_list(class_list)
Returns all classes which a part of TriboDataFAIR.owl, saved in variable all_owl_classes
Called when the script is initialized.

get_searchable_classes_from_list(class_list)
Gets all related classes from Kadi4MateRecord as variable searchable_owl_classes.
Called when the script is initialized.

Select one class an press SEARCH

search(request)(selected class)

Generate_html_from_dict_via_recusion(data_tree_dict)
Generates the HTML code for each layer by calling itself on each layer. Sets every layer in a different CSS class. Lists instances as list elements.

Inserts the HTML code into the HTML template SurfTheOWL.html and serves it to the web browser.

received ids and comments are inserted in html as tooltips

Magnitueds are extracted and the unit and datatype are arranged beside each other in the html code

main_search(selected class)

find_classes_layers_via_recusion(layer_dict, keys, freindly_layer_dict, depth)

Builds the data tree according to return. Then calls itself on a layer deeper.

Processes the data tree to separate classified objects into a special list.
Orders the tree hierarchy to meet the Kadi4Mate one.

Then returns the data tree with freindlyNames. Gets and retunes the contectual type of the record.
Also returns the id_dict, comment_dict, instances_comment_dict.

children(key)

If not end_of_entries

Yes

Not

Processes the returned list to a dictionary and manipulates instances and classifies Objects to easily readable data entries. The dictionaries are then returned.

search_class(key)

Extracts all subclasses, instances and Objects from the OWL of a given Classname(Key). These are returned as a list.

end_of_entries(key)

Retunrs true, if no subclasses are defined from a given classname. This is the case, when the name points to another object or an instance or no subclass is found.

property_restiction_in_string(OWL_class_string)

Detects owl properties and returns them. If no property in a string the return is false.

check_for_special_restriction(OWL_class_string)

If the OWL_string contains a not in owlready2 implemented property the pointed instance is retuned classified with (<class ...>). Current this is the case for property „hasTimeStamp"

check_for_object_refer(OWL_class_string)

If the OWL points on another defined Object the function returns the object and classifies it (<object ...>). If not, the return is false.

is_class_refere_other_object(key)

Returns true, if key is a classified Object

is_class_datatype(key)

Returns true, if key is a classified instance

get_data_instance(key)

If not is_class_datatype

Yes

Extracts the defined instances from the OWL of a given class., mostly ...Units.
Also Extracts the comment of a instance, if one is provided
Returns instances , instances Comments dictonarie

bad_letter_in_string(string)

Non-ececutable letters are detected, these are -, ., and a digit at the start of a String. If a bad letter is detected return is true.

get_comment_by_className(class_name)
Returns the comment of a given class

get_ID_by_className(class_name)
Returns the TDO id of a given class

className_to_friendlyName(key)

Extracts the friendlyName of a given className from the OWL. If no friendlyName is defined the className is classified as problematic. The retune is the frindlyName or the classified className.

Figure 1: schematic of the code, SurfTheOWL