Documentation SurfTheOWL

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. 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.

Directory **SurfTheOWL:** 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 dajngo and displayed in the browser. All styling is done via css from the subdirectory "`static`" with the file "`SurfTheOWL.css`".

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….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.

At the start in line 9 get the OWL loaded, if the name of the OWL changes it must change also here and in line 10. 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 as 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.

If you call the local server in the web browser you are redirected from urls.py and call 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.jtml. For the action that's trigger by the form filed the second URL "/Surfing" is used with the request method POST. The ObjectFriendlyName serves as the displayed name (because convenience) and the ObjectName is the real value.

By selecting one OWL class in the select field in the web browser you are selecting the ObjectFreindlyName, as value the ObjectName is set. By clicking on the "SEARCH" button you send the selected ObjectName to the URL "/Surfing" as variable searched_class. Urls.py redirects the request to views.py and calls the function search(request). 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. With the data_tree 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. If the className is contained in the OWL, line 251 tow 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 the OWL classes. The main key in the classes_dict is the given function attribute, 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) is called.

The function generates two dictionaries' 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 an other 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 Key dose not have subclasses the function get_data:instances(className) is called, which returns the Datatype ore the other special Object referring.

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 OWL. 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 children(className)is than looped offer the list elements which contains a datatype ore a special Object is manipulated in the list 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. Now the function children(className) return the children_classes_dict, the children_keys, the friendly_names_dict and the list other_object_refer_pair 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. 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) to the function search(request) in views.py. There the html code is generated and served to the browser.

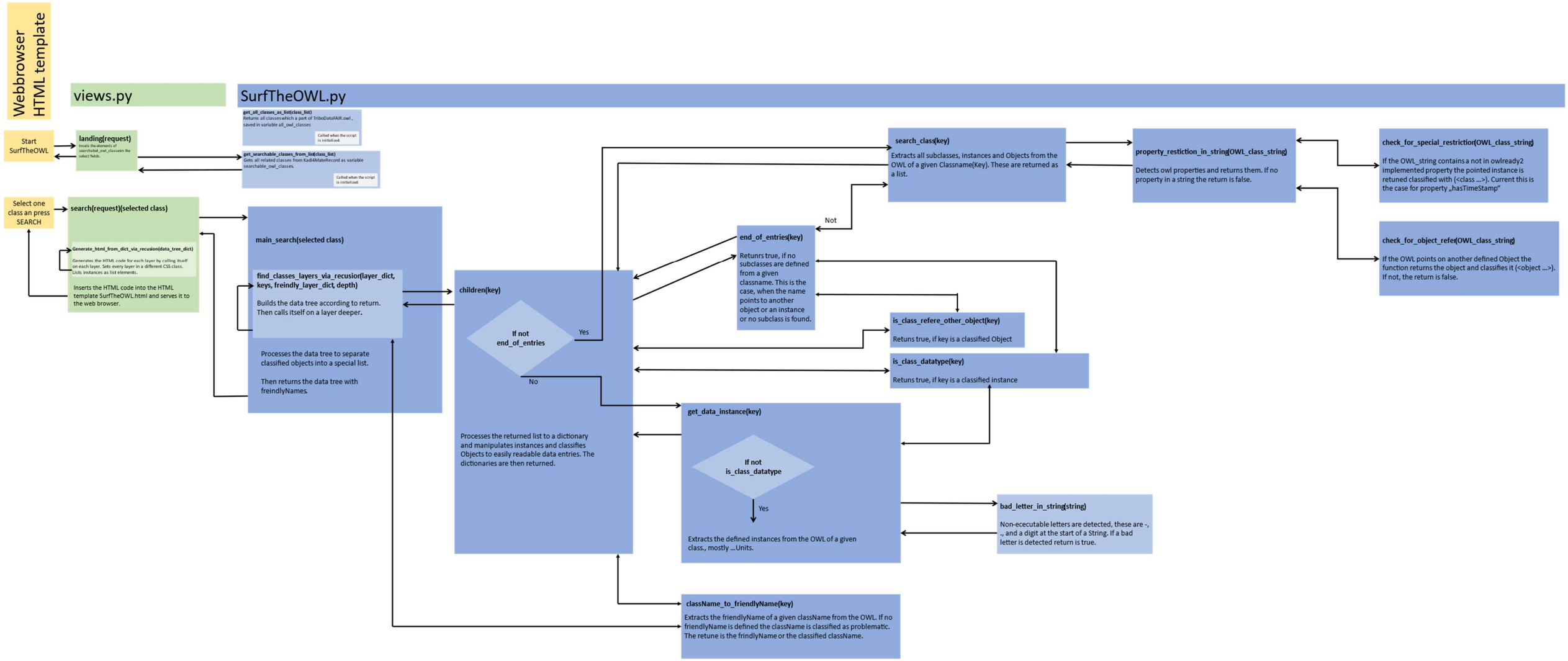

Figure 1: schematic of the code, SurfTheOWL