A FIREFOX EXTENSION FOR SEMANTIC ANNOTATIONS

Written by: Pham Van Vung – Student ID: 722905.

Table of content

I. Introduction 3
II. Analysis and Design of a Firefox extension for semantic annotation...................... 4
IV. Development of the JavaScript part of the extension... 19
V. Application, Results, Discussions and Future works... 25
VI. Deployment 27
References.. 28
A. A sample of joseki_config.ttl 29

I. Introduction

In every area of software development, the ease in extensibility and efficiency of the development is always a practical issue, especially in the world of JavaScript which is difficult to debug when things go wrong. Furthermore, it is always a benefit of having a generic methodology which allows the developers to develop applications in a declarative way using RDF technologies.

In this AIRLab project I would like to present a methodology which tries to satisfy the mentioned requirements. This methodology will present the development steps, design architectures, and technologies in development of extensible Firefox extension for Semantic Annotations.

The report first analyzes the generic components of Firefox extensions for semantic annotations to have clear idea of steps that should be followed to develop an extension providing semantic annotation functionalities. Basing on this analysis, the code for the generic components is implemented. Then later the programmers can just declaratively describe the components, using RDF technologies, and the code will automatically generate them based on the developers' descriptions.

The report will next describes about each of the components, how they are described in RDF and how to use them in JavaScript code if needed.

For the developers who would like to extend this framework, there is also a part describing the coding ideas of this development methodology and the possible extensions of this project.

II. Analysis and Design of a Firefox extension for semantic annotation

First of all we need to have some understandings about some terminologies used in this writing, specifically annotation, semantic annotations, semi-automatic semantic annotation, and Firefox extensions.

- Annotation, "In linguistics (and particularly in computational linguistics) an annotation is considered a formal note added to a specific part of the text" [6]. Some possible types of annotations could be in form of tags, ratings, page suggestions, etc:
    - A tag is a keyword or term assigned to a piece of information. This kind of metadata helps describe an item and allows it to be found again by browsing or searching. Tags are generally chosen informally and personally by the item's creator or by its viewer, depending on the system.
    - A rating is the evaluation or assessment of something, in terms of quality, quantity (as with an athlete being rated by his or her statistics), or some combination of both.
    - A page suggestion, is specific for annotations for websites, a page suggestion could be a next page (a website that the user should browse next starting from currently browsing page), or a previous page (a website that the user should browse before the currently browsing page, in order to understand the currently browsing page for instance).
    - Etc.
- "A semantic annotation in a document is additional information that identifies or defines a concept in a semantic model in order to describe part of that document" [7]. In the application point of view, Semantic annotation is defined as [6]:
    - a sort of meta-data and
    - the process of generation of such meta-data.
- Semi-automatic semantic annotation, is defined as "it is semantic annotation which uses machines to suggest terms and uses editors to refine choice" [8].
- Firefox extension, "Extensions add new functionality to Mozilla applications such as Firefox, SeaMonkey and Thunderbird. They can add anything from a toolbar button to a completely new feature. They allow the application to be customized

to fit the personal needs of each user if they need additional features, while keeping the applications small to download"[9].

This project will develop a Firefox extension for semantic annotations. This Firefox extension will let user add annotation to websites while browsing them with Firefox browser. The user will be able to have:

- Tagging functionality:
    - Add a tag to the browsing page
    - View the tag cloud
- Rating functionality:
    - Rate the browsing page
    - View the rating values (such as average rating value for a page, and number of rates)
- Page suggestion functionality:
    - Suggest next, previous page
    - Browse next, previous page

As mentioned, annotations are not limited to tags, ratings, or page suggestions, so this Firefox extension will also be flexible and extensible to let the developers easily add the other possible types of annotations.

As a general analysis of Firefox extensions for semantic annotations we discover that they should have the following components for each type of annotation:

- *Menu*, menu *popup*, list of menu *items*
- When we choose a menu item a *Function* will be executed
- At the Data point of view, we need to execute *queries* against one or more data source in order to *update/select* data for the extension.
- Some *XUL Dialogs* which is used to ask for user input and update the information to the current *data stores* as database for the annotation system.
- There might be some *Floating Window* which is used to display information to the user

At the data point of view, the annotations are stored using RDF. Current time, there are many ontologies for describing annotations. For instance, for rating we may use Tagging ontology [10], Rating ontology [11], sioc ontology [12], etc., to describe annotations. We should not use individual ontology for each annotation type such as Tagging Ontology for tags, Rating Ontology for rating, and so on. There should be one single ontology used to describe annotation in this project for the consistent representation and ease in programming and extensibility. After evaluating of several ontologies, it comes to a conclusion that Tagging Ontology [11] is suitable for describing out annotations. For further details about Tagging Ontology please refer to [11]. However, we are not exactly using the Tagging Ontology here to describe annotations since Tagging Ontology is for tags. However, a similar ontology is used to describe all types of annotations.

The fundamental design decisions are:

1. The annotator (user) is foaf:Agent
2. Annotations reify the n-ary relationship between a annotator, a annotation, a resource, and a date.
3. Annotations are members of a `Annotation` class. `Annotations` have ids.

Specifically, for tagging we have:

1. Taggers are `foaf:Agent`s.
2. `Tagging`s reify the n-ary relationship between a tagger, a tag, a resource, and a date. Relationships exist for each of these roles.
3. Tags are members of a `Tag` class. `Tag`s have names. We do not attempt to implement plurals (as in the `labels` schema) or synonymity at this stage, as these are subjective assessments of a tag.

One example to tag “general” to <http://localhost:2020/> is:

```
@prefix xsd: <http://www.w3.org/2001/XMLSchema#>
@prefix foaf: <http://xmlns.com/foaf/0.1/>
@prefix tags: <http://www.holygoat.co.uk/projects/tags/>
@prefix tag: <http://www.polimi.it/phamvanvung/sema/tag/>
tag:general a tags:Tag; tags:tagName "general" .
<http://localhost:2020/> tags:tag [
                        tags:associatedTag tag:general;
                        tags:taggedBy [foaf:mbox <annonymous@annonymous.com>];
                        tags:taggedOn "Fri Jun 11 2010 15:25:33 GMT+0200"^^xsd:dateTime] .
```

Similarly, we are able to describe other types of annotations in such the way.

Following is an example of rating annotation, which rates the page "http://localhost:2020/" with value "5":

```
@prefix xsd: <http://www.w3.org/2001/XMLSchema#>
@prefix foaf: <http://xmlns.com/foaf/0.1/>
@prefix rat: <http://www.polimi.it/phamvanvung/sema/ratings/>
<http://localhost:2020/> rat:rating [ rat:ratedValue "5"^^xsd:integer ;
                                      rat:suggestedBy [foaf:mbox <annonymous@anonymous.com>];
                                      rat:suggestedOn "Fri Jun 11 2010 15:31:12 GMT+0200 (ora l
```

Following is an example of page suggestion annotation, which suggests "http://localhost:2020/query1.html" to be the next page that user should browse starting from this page.

```
@prefix xsd: <http://www.w3.org/2001/XMLSchema#>
@prefix foaf: <http://xmlns.com/foaf/0.1/>
@prefix ps: <http://www.polimi.it/phamvanvung/sema/pagesuggestions/>
<http://localhost:2020/>  ps:suggestion [
                              ps:nextPage "http://localhost:2020/query1.html" ;
                              ps:suggestedBy [foaf:mbox <annonymous@annonymous.com>];
                              ps:suggestedOn "Fri Jun 11 2010 15:37:30 GMT+0200 (ora legale Europ
```

At this point, there may be an argument that why we don't use some other standard ontologies for describing these types of annotations, for instance Tagging Ontology for tagging annotations, Rating Ontology for rating annotations and so on. In the development of this extension, we use Tagging Ontology for tags and use a similar design decisions for other types of annotations. Since we found that an ontology which is similar to Tagging Ontology is generic enough in order to describe also other types of annotations. In other words, using one generic type of Ontology with similar design decisions (as mentioned) we are able to describe all other types of ontologies for extensibility and consistency.

All the data of the annotations will be stored as RDF triples in a RDF data store. In this application we are using Joseki as the data server. Joseki [5] is an HTTP engine that supports the SPARQL Protocol [14] and the SPARQL RDF Query [13] language. SPARQL is developed by the W3C RDF Data Access Working Group. Joseki Features:

- RDF Data from files and databases
- HTTP (GET and POST) implementation of the SPARQL protocol

With Joseki we can setup Sparql endpoints which are able to process update/select Sparql queries to update/select data for our applications.

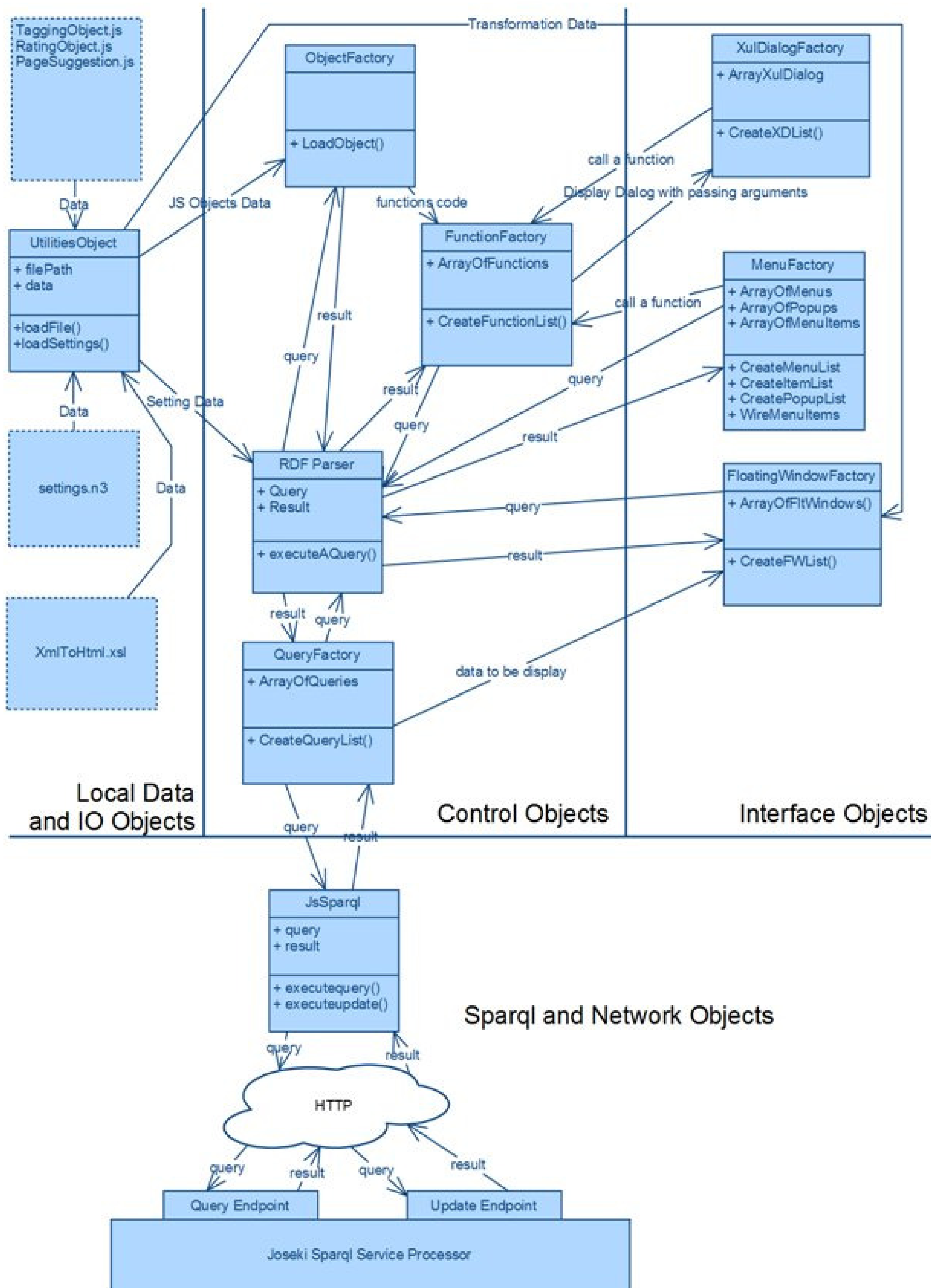

Figure 1 Architecture of a Firefox extension for semantic annotation

Figure 1 depicts the Design Architecture of the Firefox extensions for semantic annotations. It has the following components:

- A UtilitiesObject component
- An RDF Parser
- An ObjectFactory component
- A FunctionFactory component
- A QueryFactory component.
- A JsSparql component
- A MenuFactory component
- A FloatingWindow component
- A XULDialogFactory component
- A Data store component
- And other local files

An UtilitiesObject component is used to load the local files from local disk of computer which is running Firefox browser. The main function is a function called *loadData(fileName)* this file take an argument is the file name and return the content of the file in bytes. In this function, we are using the Firefox local file service to load the data from its file name to it content in bytes (*var outputFile = Components.classes["@mozilla.org/file/local;1"].createInstance(Components.interfaces.nsILocalFile);*) .The local files include

- Setting.n3 is an RDF setting file which is used to describe the settings of the extension.
- JavaScript files, these files are the code of JavaScript functions which are placed here in order to avoid complex code in the body of the functions described in the setting file. For instance, *TaggingObject* includes code for functions used in the Tagging annotation functionality and so on.
- Xml Style Sheet Language (xsl) files which are used to transform XML data (the result of the Sparql queries) into HTML to be displayed into Floating Window.

For all of the Factories which are going to be explained in the following parts, the main steps will be: first it has sparql query which is used to load information about the item

inside the factory from the RDF setting file. It uses the JavaScirpt syntax "var object = Function(params, body)" to create an object (in general, a function in JavaScript is an object) and then add that object to the factory with the name which is the name of the item using the syntax "Factory[name]=object".

The RDF parser is used to parse the data loaded from setting.n3 RDF file using sparql query. In this implementation we are using Hercules RDF Library in JavaScript: SPARQL Engine [2] which is able to execute sparql queries over the RDF data stored in n3 tripples loaded in bytes (setting.n3). All the factories are using sparql queries in order to queries information about its factories giving this object a query; it will return the results as described in the setting.n3 RDF setting file. This RDF parser is used to parse data from RDF local file loaded to it in bytes. It is different from the JsSparql object which (will be described later) is used to send queries to Joseki sparql endpoints over the network and receive the returns results.

The ObjectFactory component is used to put extra coding body for some functions which is described in FunctionFactory (will be described next). Because the functions described in FunctionFactory are in plain text, it could be complex to describe the body of a function in a string. Therefore, the functions could be coded in JavaScript files and later be loaded and use in describing functions. For instance, instead of describing the whole code for the function TagFromSelection which is used to tag a page from selected text in the page, we create a JavaScript file called TaggingObject.js which has a TaggingObject object and this object has a function called TagFromSelection. Later, the function TagFromSelection which is being described in setting.n3 will have a body is "TaggingObject.TagFromSelection();" instead of having the whole body of the function described in a string. The main coding idea of this ObjectFactory is using the eval(string) function in JavaScript which will execute a JavaScript string written in the JavaScript file which the ObjectFactory is loading.

The FunctionFactory component is used to load functions which are described in the setting file and later be used in the JavaScript code. It will use the RDF parser to parse for

information about the functions described in setting files. The main function in this object is to createFunctionList. This function queries all function information (including a name, parameter list and the body) using RDF parser and then create the function using the JavaScript syntax "var functionName = new Function(parameterList, body);". The function after creating will be stored in FunctionFactory and be used later. So we can view the FunctionFactory as an array of functions ready to use by the extension.

The QueryFactory component is used to load sparql queries which are described in the setting file. It will use RDF parser to load all the queries information such as the query name, query parameters, query body, response types (such as xml, json, plaintext), and the Floating Window (if there is) used to display the result of the out put. It will then build two main methods for each query. The first method is "getQueryString" which will return the string of the query. The second method is "execute" which will execute the query over the sparql endpoints using JsSparql component.

The JsSparql component is used to send queries to Sparql endpoints and receive results. This JsSparql component has two methods. The first one is selectquery method which is used to send a selection query to the Joseki "read" Sparql endpoint via an HTML Get method and receive the results in the body of the response text. The second one is updatequery method which is used to send an update query to the Joseki "update" Sparql endpoint via a HTML Post method.

The MenuFactory component queries the menus, menu popup, and menu items' information from the RDF setting file using sparql query. It then creates list of menus, list of menu items, and list of menu popup and save them in the MenuFactory as an object named as the item's name (MenuFacotry["ItemName"] = Item). It then wires them together using the subItem property of the items. It then adds the items to the menu context of the Firefox browser.

The FloatingWindowFactory component queries the floating window information such as name, width, height, title, etc., from RDF setting file using a sparql query and create an

object with the name is the name of the floating name and store it into the FloatingWindowFactory (FloatingWindowFactory[“name”] = item). In each item of the Factory (each floating window), it creates a method called “createFloatingWindow”. This method when called will create an HTML Div with title, close button, body, etc., and attach it to the top left of the current browsing page. The floating windows in FloatingWindowFactory are often used to display the result from Sparql query from the QueryFactory. It use a transformation file (xsl file) loaded by the ObjectFactory to transform the XML result of a sparql query and translate it into HTML text to be displayed in the FloatingWindow.

The XulDialogFactory component is use a sparql query to query information such as name, parameters, etc., of the XulDialog in order to create a XUL Dialog which is used to get input information from the user and update it into the sparql endpoint using JsSparql object. The each xul dialog is stored in the XulDialogFactory as an object with a name is the name of the xul dialog. The passing parameters are usually a response text from a sparql query used to give some initialize information (e.g., a tag cloud for tagging dialog) or the JsSparql object which is usually used to update information to the sparql endpoints.

The Data store component is a Joseki server. It is used to provide sparql read/update endpoints to the firefox extension. In this case it provides one read endpoint which receives an HTML Get method (coming from JsSparql object) with the sent content is a select query and will returns the result of the query over the stored RDF triples as the response text of the HTML request. It also provides one update endpoint which receives an HTML Post method (coming from JsSparql object) which the sent content is an update query and process the update to the RDF triples.

The local files are the JavaScript files which are used to put the complex code of body of some functions so that we can reference to the code from the body of the declarative function by just referencing the name of the function coded in these files. The main programming idea of this is to load the content of the JavaScript files using the UtilitiesObject and use the “eval(content)” method to create the object. The another local

file is the RDF setting file which is used to declare all the objects such as the menu, queries, functions, etc. This setting file then will be parse by a parser (we are using Hercules parser in this case) to parse for information about menus, queries, functions, etc. Other local files are the XSL files which are used by floating window. The content of these files are also loaded by the UtilitiesObject and then used to transform the XML result of a Sparql query into the HTML content and will be displayed in the floating window.

III. Development of the RDF Settings of the framework

This part will describe the main components of the design architecture, starting with an RDF file which describes the parts of the functionalities including the possibility to describe Menus, Functions, Menus and Menu Items, Queries for Sparql, Floating Windows, XUL Dialogs, and Objects (for some JavaScript functionalities). The following part will describe the JavaScript implementation of the components.

The purpose of this description idea is to let developers declaratively specify the parts of the Firefox extensions for semantic annotations which require less ability to program in JavaScript. The framework first deals with how to describe a function.

As mentioned in the design architecture that our Firefox extension has a component called the RDF Parser which uses the Hercules Library [2] enabling it to execute Sparql queries over RDF triples loaded in bytes (using UtilitiesObject). Therefore, we decided to use RDF to store the information about the Firefox extensions as the followings.

1. Description of Functions

A typical form of a function description is:

```
#Design of a dynamic function
func:tagFromSelection func:Function [   func:Name "tagFromSelection";
                                    func:Params "event";
                                    func:Body "ObjectFactory.TaggingObject.tagFromSelection();"
                                    ] .
```

This description is clearly self explaining. A function includes:

- A function name
- Its parameters
- The body of the function.

One may argue what is the use of having a function description while we still have to write its body with JavaScript. However, this makes the system easier for non-programmers (users can call this function easily and declaratively by using its resource reference). I.e, it can be declaratively called by other descriptions such as a menu item,

and can also be called by JavaScript code. Another practical benefit of this is the separation of the development structures.

Another argument one may have is that writing the body of the function in the form of a string is relatively complex and difficult. This implementation deals with this by using an ObjectFactory (please see next parts for ObjectFactory) which let the developers put complex codes into a file, taking care of loading the code automatically. Therefore, the developers only need to put the simple code here (say a function name) which references to the complex code loaded by the ObjectFactory.

2. Descriptions of menus:

Another part of every Firefox extension for semantic annotation is the Menu. Every menu should have a menu popup, inside which the menu items are listed. Each menu item may have a function (command) to be executed when it is clicked. Therefore, the following ones are the typical ways to describe menus, menu popup, and menu items.

```
#TaggingMenu
menu:TaggingMenu    menu:menu [ menu:id "TaggingMenu";
                                menu:label "Tagging Menu";
                                menu:subItem menu:taggingmenupopup
                              ] .
menu:taggingmenupopup menu:menupopup [  menu:id "taggingmenupopup";
                                        menu:subItem menu:tagSelected, menu:tagCloud, menu:taggingDialog
                                        ] .
#TaggingMenu Items
menu:tagSelected    menu:menuitem [     menu:label "Tag Selected";
                                            menu:id     "tagSelected";
                                            menu:oncommand func:tagFromSelection
                                ].
menu:tagCloud menu:menuitem [   menu:label "Tag Cloud";
```

A menu, menu popup, and menu item should have:

- A label
- An id

This may be not so from coding the menu in XUL: the only difference is that all menus, menu popups, and menu items need to have an Id in order for us to reference to them and wire them together. The menu popup and menu should have an extra description predicate, called subItem, which describes which menu items are the children of the menu popup and which menu popup is child of a menu. The menu item also has an extra

predicate which allows describing the command which is a reference to the Function described in previous part. This function will be executed when the menu item is clicked.

The advantages of this approach are the possibility to declaratively specify the system and wire different components together and also the clear separation of the extension's structure. For instance, if the developers want to change or edit things related to menus and interface, they just need to modify this part of the configuration.

3. Description of Sparql Queries

The next thing that is important to be able to describe is how Sparql queries are executed. For most of the provided functions, we need to execute the Sparql queries to insert or select the data from Sparql endpoints. The typical query description is as following:

```
#Queries for tagging
qry:selectTags qry:query [  qry:name  "selectTags";
                            qry:params  "thePage, conf_limitTags";
                            qry:type  "select";
                            qry:body    "'PREFIX xsd: <http://www.w3.org/2001/XMLSchema#> PREFIX tags: <
                            qry:responsetype "xml";
                            qry:callbackhandler flw:tagCloud
                            ].
```

A query should have

- A name
- A set of parameters
- A type which has the value of either "select" or "update"
- A body which is really the query that should be written as in normal JavaScript code.
- A response type is either "xml", "json" or "plaintext".
- A call back handler which could be a Floating Window (defined in the next part) which is described to display the result of the queries. The value of the callback handler could also be "custom" meaning the developer will give a custom function to handle the result of the queries or "none" meaning we do nothing with the query result or when there's no result as in case of "update" queries.

The advantages of having the possibility to describe queries are the following: first of all, we are able to specify the queries declaratively and the framework will do the real job of

accessing the endpoints, execute queries and return the results; Furthermore, developers can easily access the query information (for any purpose) via the extension itself.

4. Description of Floating Windows

As the result from analysis parts, in most of the functionalities we need ,sometimes, to show some information. One function which is very frequently used is the one that shows, inside a floating window, the results of a sparql query as HTML, based on some XSL transformation. A typical description of a floating window is as the following:

```
flw:tagCloud flw:window [   flw:id "tagCloud";
                            flw:title "Tag Cloud";
                            flw:width "500";
                            flw:height "200";
                            flw:xsl "xsls\\tagCloud.xsl"
                          ].
```

A floating window should have:

- An id
- A title which is displayed on the bar of the window
- A size (width, height)
- A xsl file

The id, title, width, and height are self explaining. The “xsl” file is the transformation file that when the floating window is used to display the result of a query to transform the result to the html displayed in the floating window.

The advantages of having the possibility to describe a floating window are that the developers only need to declare the window declaratively without programming requirements. The framework will be able to create the window, load the transformation file, transform the results of a query to html from xml result basing on the transformation file, and finally find the current document to attach this floating window to it. Moreover, there is the advantage of wiring this floating window to a query declaratively.

5. Description of XUL Dialogs

In many cases, developers need to display a XUL Dialog to the users to ask for input. The XUL dialogs defined in the Extensions have the ability to access its JavaScript functionalities; specifically which is the possibility to access the “JsSparql” object in order to execute “select”/”update” queries. A typical description of a XUL Dialog is as follows:

```
xdlg:tagDialog xdlg:dialog [    xdlg:id "tagDialog";
                                xdlg:chromeurl "chrome://semanticannotation/content/dialogs/tagDialc
                                xdlg:title "Tag Cloud";
                                xdlg:params "responseText, thePage, tagFromString"
                                ].
```

A xul dialog has:

- An id (for referencing in other places),
- A chromeurl, this is a chromeurl since this dialog must be defined by the current extension in order to have the ability to access JavaScript functionalities defined by this extension
- A title
- A set of parameters

The very main advantage of having ability to describe a XUL dialog is just about the clarity of the structure of the development.

IV. Development of the JavaScript part of the extension

After having the description of the Extension, the next task is being able to load the RDF descriptions from a setting file to create the elements and wire them together as described in order to build the extension.

The main idea of the development in JavaScript of the extension is the ability to create a function, which is also an object in JavaScript, dynamically from string descriptions. For instance we have a string describing parameters of a function as `params = "x, y"` and a string describing the body of the function as `body = "return x + y".` We can create an `"add"` function as following: `var add = new Function(params, body)` and later can call the function as `x = add(1, 2)` this will lead to `x` having the value of `3(=2+1).`The main reason for using this ability of JavaScript is that the data we get from querying an RDF file are in type of strings, and what we need in the JavaScript are the functions and therefore, this ability enables us to translate strings into functions. For further descriptions and other use of this functionality please refer to Mozila Developer Community [1].

The first part of the code are some supporting objects such as constants object `(xmlns)`, query execution object `(executeAQuery)`, utility object (`UtilitiesObject`), and browser object `BrowserObject`.

Constants (`xmlns`**)**, the very first part of the extension is to define an area where all the constants (specifically the definition of the xml namespaces) which would be repeatedly use in the coding.

Query execution object (`executeAQuery`), the extension must be able to load the setting file and execute some Sparql queries on it in order to have data on the queries, on floating windows, on functions, etc., to create them. In this extension we are using "Hercules RDF Library in JavaScript" [2] to execute Sparql queries on a file loaded in bytes. This library, by the time of writing this document, is still having some limited

ability such as not yet supports the full description of Sparql query (`OPTIONAL` feature for instance) and the data must be defined in `n3` format (not other format such as XML which is normally a good thing if we would like to use the RDF setting file as data source for Firefox template functionality as in [3]). Further noticing point is this library doesn't accept other types of whitespaces (such as new line, or tab, etc) in the body of the string literal value.

Utility Object (`UtilitesObject`**)**, which is a JavaScript object implementing some utilities such as loading the setting data from RDF file to `bytes` to be ready to use by the Hercules RDF Library to executes Sparql queries against it, or loading JavaScript files to `bytes` to be executed (using "`eval`" function in JavaScript), etc.

BrowserObject(`BrowserObject`**)**, this object defines some functionalities of the browser such as getting the URL of the currently browsing page, adding a new tab to the browser, etc.

JavaScript ability to create `Function` from its string descriptions as "`var functionName = new Function(parametersString, bodyString);`". These string descriptions are described as RDF triples in the RDF setting file. The object factories which are Function Factory, Menu Factory, Query Factory, Sparql Object, Floating Window Factory, Object Factory and a XUL Dialog Factory are built as follows.

- The setting file is loaded into bytes using `UtilitiesObject` (which use Firefox local file interface feature)
- A query is made to query for string descriptions of the to-be-created object (`params`, `body`, etc)
- The query is executed using `executeAQuery` object (which uses Hercules RDF library [2]) against the loaded bytes to get descriptions of the to-be-created object.
- Using JavaScript syntax to create the new object (as a `Function` in JavaScript as `var newObject = new Function(params, body);`).

In the following we are going to describe each of the Factories.

Function Factory (`FunctionFactory`), this factory will make a query to load all the functions defined in the setting RDF file and transform them to functions with the ability to make new `Function` from string descriptions of its parameters and body as described in the very beginning of this part. When the factory is built (after calling `buildListOfFunctions()`), to make call to a function described in the RDF settings file with the name of `functionname`, programmer will call `FunctionFactory.functionname(parameters)`.

Menu Factory (`MenuFactory`), this factory will make queries to load and create all menus, menu popup, and menu items as XUL elements. The items are created using the syntax to make new `Function` (which is also an Object in JavaScript) from string descriptions of its parameters and body. The RDF setting file is loaded into bytes using `UtilitiesObject,` then queries are executed against these bytes using `executeAQuery` object (which uses Hercules RDF library [2]) to get the string descriptions. Then the menu items will be wired with its function to be executed when it is clicked. All the elements will be wired together based on the `subItem` predicates using `appendChild` method of the XUL elements. Finally the menus are overlaid to the `Context menu` of the Firefox browser. With this factory, the menus are just needed to be described in RDF and this extension will load, create, wire them together, and overlay them to the browser automatically and dynamically.

Query Factory (`QueryFactory`), this factory is used to load and build the list of queries and the functions which should be called when developers would like to execute a query. The queries are created using the syntax to make new `Function` (which is also an Object in JavaScript) from string descriptions of its parameters and body. The RDF setting file is loaded into bytes using `UtilitiesObject,` then queries are executed against these bytes using `executeAQuery` object (which uses Hercules RDF library [2]) to get the string descriptions. If a query is defined in the RDF Setting file with a name like "`queryName`" then the programmer can get the string description of the query with the function

`QueryFactory.queryName.getQueryString(params)`. The `params` is the set of parameters required by the query defined in the query area in the RDF setting file. In order to execute the query the programmer just needs to call `QueryFactory.queryName.execute(params)`. Again, the `params` here are the parameters required by the query and a callback handler if the query callback handler (defined in the query part area in RDF Setting file) is of type "`custom`". If the callback handler defined in the query as the Floating Window, the floating window (described in next part) will be used automatically to handle the result. No handlers for the callback handler of type "`none`".

JavaScript Sparql Object (`JsSparql`), this object is used to execute queries to the sparql endpoints. It includes two methods: one for executing the selection queries (`selectionquery`) and another for executing the update queries (`updatequery`). The `updatequery` method has two extra parameters the first is the `callbackFunction` which is used to handle the result responded from the Sparql endpoint, and another is the requesting response type ("`xml`", "`json`" or "`plaintext`"). The `selectionquery` method executes its queries by sending them to the Sparql read endpoint "`sparql/myproject/read`" using the HTML GET method. The updatequery method executes its queries by sending them to Sparql update endpoint "`sparql/myproject/update`" using the HTML POST method.

FloatingWindow (`FloatingWindow`), for each floating window defined in the RDF setting file one object will be created in the floating window factory (`FloatingWindow.floatingWindowName`). The Floating Windows are created using the syntax to make new `Function` (which is also an Object in JavaScript) from string descriptions of its parameters and body. The RDF setting file is loaded into bytes using `UtilitiesObject`, then queries are executed against these bytes using `executeAQuery` object (which uses Hercules RDF library [2]) to get the string descriptions. The programmer can wire a Floating Window to a query defined in the query area in the RDF setting file. The `xsl` file in the floating window will be loaded and used to transform the query result from `xml` to `html` and put into the body of the floating window and finally

attached to the currently browsing page. The programmer can also manually call to the `createPopup` method in order to create a custom floating window in the JavaScript code.

Object Factory (`ObjectFactory`), the idea of describing the function in RDF could be little troublesome when we are writing a JavaScript function in a string. We encountered lots of troubles such as having spaces, new lines, and escaping characters, code indentation, etc., which would lead to the inability to read the code from the string (`body`). Therefore, extra JavaScript code for the function should be written in JavaScript files and loaded on to the Firefox extension. Therefore, the body of the function is only a short reference to the code written in the file. This helps the clearness of the code. Furthermore, this also helps to extend the functionalities of the extensions. For instance, in Firefox extension for semantic annotation developed, there are three extra JavaScript files (`ratingObject.js`, `suggestingObject.js`, `taggingObject.js`) used for rating, page suggestion, and tagging correspondingly. These JavaScript files are first loaded into bytes using `UtiltiesObject` and then activated using the "`eval(byte[] theCode)`" method of JavaScript. With this organization and separation of code, it is easy for developers to implement their own JavaScript files and load them automatically to the extension. If a developer want to edit any functionality it is relatively easy to isolate and inspect it separately.

XUL Dialog Factory (`XulDialogFactory`), this Factory creates a XUL Dialog for each defined as `XulDialogName` in RDF Setting file. The xul dialogs are created using the syntax to make new `Function` (which is also an Object in JavaScript) from string descriptions of its parameters and body. The RDF setting file is loaded into bytes using `UtilitiesObject,` then queries are executed against these bytes using `executeAQuery` object (which uses Hercules RDF library [2]) to get the string descriptions. The XUL Dialog could be showed by calling `XulDialogFactory.XulDialogName.show(params)`. `Params` is the set of parameters required by a XUL Dialog to initialize its interface and enable its functionalities. Normally, the developer needs to pass a response text from a query (such as list of `tags` in case of the Tagging XUL Dialog), the current page (`currentPage`) and a JavaScript object to execute its functionality. For instance, a

JavaScript sparql (`JsSparql`) object is passed to the XUL Dialog enabling it to update or select data from Sparql endpoints defined by the extension.

As a short summary, all the factories and objects are created as follows:

The Factories (FunctionFactory, XULDialogFactory, QueryFactory, FloatingWindowFactory) are created as:

- The setting file is loaded into bytes using `UtilitiesObject` (which use Firefox local file interface feature)
- A query is made to query for string descriptions of the to-be-created object (`params`, `body`, etc)
- The query is executed using `executeAQuery` object (which uses Hercules RDF library [2]) against the loaded bytes to get descriptions of the to-be-created object.
- Using JavaScript syntax to create the new object (as a `Function` in JavaScript as var `newObject = new Function(params, body);`).

ObjectFactory is created as:

- First load JavaScript code from the JavaScript local files into bytes using `UtilitiesObject`
- Second use JavaScript method "`eval(byte[] theCode)`" to activate the code in the JavaScript.

The Sparql JavaScript object has two methods:

- First method is `selectionquery` method uses HTML GET method to send the queries to the Sparql read endpoint
- Second method is `updatequery` method uses HTML POST method to send the queries to the Sparql update endpoint.

V. Application, Results, Discussions and Future works

This Firefox extension for semantic annotations is still simple, implementing a tagging, rating, and page suggesting functionalities. With tagging functionalities, the user is able to tag the currently browsing page from the text selected from it (tag selection). User can see the tag cloud of the page. User can also view a custom tagging dialog, which allows manually adding tags to the page.

With rating functionalities, user is able to rate the page (with value ranged from 1 to 5), to see the highest rated pages (pages ordered by its averagely rated value of highest is 5 and lowest of 1), and the most rated pages (pages ordered by number of peoples who rated for this page, regardless of the averagely rated value).

With page suggestion functionalities, the user is able to go to the highest suggested next page from this page and highest suggested previous page from this page. User can also view a page suggestion dialog, which allows suggesting his/her opinion about the next or previous page that a user should go starting from this page.

The steps in developing this extension are as follows:

- **Step 1**. Describe menus for tagging, rating, and page suggesting in the setting file.
- **Step 2**. For each functionality, make a code file (which is not simply coded in the string as a body of a function in the RDF setting file). There are three files (`ratingObject.js`, `suggestingObject.js`, `taggingObject.js`) correspondingly for rating, page suggesting, and tagging functionalities.
- **Step 3**. Describe the functions, which are used by the menu items. Then wire the functions with menu items using their `oncommand` predicate.
- **Steps 4**. Describe the queries needed by the functions described in step 3.
- **Steps 5**. Describe the floating windows which are used by the queries defined in step 4 and then wire the queries with them.
- **Step 6**. Code the XUL Dialogs and describe them in the setting file.

The result of these works is a Firefox extension for semantic annotation with features as mentioned at the beginning of this part.

Some further discussions are that, we kept the application reasonably simple, since the idea is to demonstrate functional flexibility of this extension for semantic annotation. The extension is a reasonably flexible and extensible Firefox extension. However, the part that would be kind of trouble is it takes little time to initialize the extension when Firefox is loaded, due to the “eager” loading manner (not “lazy” loading). At the first time the menu is loaded into Firefox browser, the extension starts to read the RDF setting files to create all the objects for menus, menu items, menu popup, queries, functions, etc and these would take time. The current reasonable remedy for this is to display an “Initializing” message to the menu to let user know that the extension is initializing its interface. The good thing is that this loading will appear only once and the program will be executed faster since all the objects are loaded and ready. The “lazy” loading method may let user see the program is loaded quickly. However, it would be slow at runtime due to the need of loading objects at runtime.

In the future, there should be more analysis on the components of the Firefox extensions for semantic annotations, in order to make the extension more reusable. For instance, the extension should be able to generate automatically aggregation functions (adding, averaging, mean, min, max, etc) for queries defined in the query area. Next, there should be more reusability discovered among the XUL Dialogs, because in current version, there’s not very much use of the extension in creating these XUL Dialogs. The developers still need to manually code the dialogs individually. Next, the extension is currently working with single update and single read endpoints. However, it is relatively simple to put a setting part for multiple endpoints. In addition, the user credentials and login/logout part would have a lot in reusability so it should be an interesting part to be developed in the extension too. The current extension doesn’t deal with users of the annotations but it does have the “user” which is currently assigned as anonymous@annonymous.com.

VI. Deployment

Installation

The installation platform was tested in Windows Vista, Windows XP and Ubuntu 9.10. There are two versions of this, one for Windows and another for Linux. The Linux version is only different from that for Windows by changing the "\\" to "/" in some parts of the code for file/directory paths. There is a readme.txt file inside each of the folder.

1. Install MySQL
2. Create a database named "annotationdb", or any other name as long as you will change it in the joseki-config.ttl file to reflect this.
3. Install Joseki (tested version is 3.4.1)
4. Add the data store (the database created) and setup the sparql read endpoint (read action) and the sparql update endpoint (update action) (please refer to **Appendix A** the joseki-config.ttl file in to have the example about setting up the enpoints to use MySQL DB as RDF Store).
5. In the sample file (joseki-config.ttl) the lines starting from line 172 to the end are the setup used to set the RDF data store to "annotatiodb" database in MySQL Server (localhost). The read action is "sparql/myproject/read", and the update action is "sparql/myproject/update"
6. Install the *.xpi, the sawindow.xpi is for Window version and salinux.xpi is for Linux version.
7. If you changed your joseki host (currently http://localhost:2020/), or your read action (currently "sparql/myproject/read"), or your update action (currently "sparql/myproject/update"), or the default user name (currenty annonymous@annonymous.com). You will need to change this default options after installing the add-on. The add-on dialog (In Firefox: Toos > Add-ons). In our add-on there is an Options Button(for Window) Preferences Button (for Linux), you can click on that and change your host, read action, update action, and user, correspondingly.

References

[1] "Function - MDC" https://developer.mozilla.org/en/Core_JavaScript_1.5_Reference/Global_Objects/Function, accessed on June 1, 2010.

[2] "Hercules RDF Library in JavaScript: SPARQL Engine Demo", http://www.arielworks.net/works/codeyard/hercules/demo/index.html, accessed on June 1, 2010.

[3] "XUL XML-based User Interface Language", http://www.xul.fr/en-xml-xul.html , accessed on June 1, 2010.

[4] "Mysql Server", http://www.mysql.com, accessed on June 1, 2010.

[5] "Joseki - A SPARQL Server for Jena", http://joseki.sourceforge.net, accessed on June 1, 2010.

[6] "The KIM Platform: Semantic Annotation", Ontext, Sirma Group Company, http://www.ontotext.com/kim/semanticannotation.html, accessed on June 9, 2010.

[7] "Semantic Annotations for WSDL and XML Schema", W3C Recommendation 28 August 2007, http://www.w3.org/TR/sawsdl/, accessed on June 9, 2010.

[8] Sudeshna. D, Tim. C, "Semantic annotation of scientific articles", DC-2009 "Semantic Interoperability of Linked Data", Semantic Annotation DC 2009.

[9] "Extensions", Mozilla Development Community, https://developer.mozilla.org/en/Extensions, accessed on June 9, 2010.

[10] "Tagging Ontology", http://www.holygoat.co.uk/projects/tags/, accessed on June 10, 2010.

[11] "Ratings Ontology", http://www.tvblob.com/index.php?option=com_content&task=view&id=128, accessed on June 10, 2010.

[12] "Sioc Ontology", http://sioc-project.org/ontology, accessed on June 10, 2010.

[13] "SPARQL Protocol for RDF", W3C Recommendation 15 January 2008, http://www.w3.org/TR/rdf-sparql-protocol, accessed on June 10, 2010.

[14] "SPARQL Query Language for RDF", W3C Recommendation 15 January 2008, http://www.w3.org/TR/rdf-sparql-query, accessed on June 11, 2010.

Appendix

A. A sample of joseki_config.ttl

```
# This file is written in N3 / Turtle

@prefix rdfs:   <http://www.w3.org/2000/01/rdf-schema#> .
@prefix rdf:    <http://www.w3.org/1999/02/22-rdf-syntax-ns#> .
@prefix xsd:    <http://www.w3.org/2001/XMLSchema#> .

@prefix module: <http://joseki.org/2003/06/module#> .
@prefix joseki: <http://joseki.org/2005/06/configuration#> .
@prefix ja:     <http://jena.hpl.hp.com/2005/11/Assembler#> .

## --------------------------------------------------------------
## This file is written in N3 / Turtle
## It is an RDF graph - order of RDF triples does not matter
## to the machine but it does help people who need to edit this file.

## Note: web.xml must be in-step with this file.
## for each service,

## Note: adding rdfs:label to blank nodes will cause Joseki
## to print that in log messages.

## See also file:joseki-config-example.ttl


## --------------------------------------------------------------
## About this configuration

<> rdfs:label "Joseki Configuration File" .

## --------------------------------------------------------------
## About this server

<#server>  rdf:type joseki:Server ;
    # Example of some initialization code.
    joseki:initialization
        [ module:implementation
            [ module:className <java:org.joseki.util.ServiceInitSimple>
;
              rdfs:label "Example initializer" ; ]
        ] ;
    .

## --------------------------------------------------------------
## Services

## Services are the points that request are sent to.
## serviceRef that will be used to match requests to services,
## not some resource URI for the description.

## Note that the service reference and the routing of incoming
## requests by URI as defined by web.xml have to align.

# Service 1
```

```
# General purpose SPARQL processor, no dataset, expects the
# request to specify the dataset (either by parameters in the
# protocol request or in the query itself).

<#service1>
    rdf:type            joseki:Service ;
    rdfs:label          "service point" ;
    joseki:serviceRef   "sparql" ;  # web.xml must route this name to
Joseki
    joseki:processor    joseki:ProcessorSPARQL ;
    .

# Service 2 - SPARQL processor only handling a given dataset
<#service2>
    rdf:type            joseki:Service ;
    rdfs:label          "SPARQL on the books model" ;
    joseki:serviceRef   "books" ;   # web.xml must route this name to
Joseki
    # dataset part
    joseki:dataset      <#books> ;
    # Service part.
    # This processor will not allow either the protocol,
    # nor the query, to specify the dataset.
    joseki:processor    joseki:ProcessorSPARQL_FixedDS ;
    .

# ---- SPARQL/Update
# A pair of services - one for SPARQL queries, one for SPARQL/Update
# Previous web.xml must also be updated to include a defintion for the
# servlet "SPARQL/Update service processor" and update requests must
# be routed to this servlet.

 <#serviceUpdate>
     rdf:type            joseki:Service ;
     rdfs:label          "SPARQL/Update" ;
     joseki:serviceRef   "update/service" ;
     # dataset part
     joseki:dataset      <#books>; ##<#mem>;
     # Service part.
     # This processor will not allow either the protocol,
     # nor the query, to specify the dataset.
     joseki:processor    joseki:ProcessorSPARQLUpdate
     .
##
## <#serviceRead>
##     rdf:type            joseki:Service ;
##     rdfs:label          "SPARQL" ;
##     joseki:serviceRef   "sparql/read" ;
##     # dataset part
##     joseki:dataset      <#mem> ;     ## Same dataset
##     # Service part.
##     # This processor will not allow either the protocol,
##     # nor the query, to specify the dataset.
##     joseki:processor    joseki:ProcessorSPARQL_FixedDS ;
##     .

## ----------------------------------------------------------------
```

```
## Datasets

<#books>   rdf:type ja:RDFDataset ;
    rdfs:label "Books" ;
    ja:defaultGraph
      [ rdfs:label "books.n3" ;
        a ja:MemoryModel ;
        ja:content [ja:externalContent <file:Data/books.n3> ] ;
      ] ;
    .

<#mem>  rdf:type ja:RDFDataset ;
    rdfs:label "MEM" ;
    ja:defaultGraph [ a ja:MemoryModel ] ;
    .

## ---------------------------------------------------------------
## Processors

joseki:ProcessorSPARQL
    rdfs:label "General SPARQL processor" ;
    rdf:type joseki:Processor ;
    module:implementation joseki:ImplSPARQL ;

    # Parameters - this processor processes FROM/FROM NAMED
    joseki:allowExplicitDataset       "true"^^xsd:boolean ;
    joseki:allowWebLoading            "true"^^xsd:boolean ;
    ## And has no locking policy (it loads data each time).
    ## The default is mutex (one request at a time)
    joseki:lockingPolicy                joseki:lockingPolicyNone ;
    .

joseki:ProcessorSPARQL_FixedDS
    rdfs:label "SPARQL processor for fixed datasets" ;
    rdf:type joseki:Processor ;
    module:implementation joseki:ImplSPARQL ;

    # This processor does not accept queries with FROM/FROM NAMED
    joseki:allowExplicitDataset       "false"^^xsd:boolean ;
    joseki:allowWebLoading            "false"^^xsd:boolean ;
    joseki:lockingPolicy              joseki:lockingPolicyMRSW ;
    .

joseki:ProcessorSPARQLUpdate
    rdfs:label "SPARQL Update processor" ;
    rdf:type joseki:Processor ;
    module:implementation joseki:ImplSPARQLUpdate ;
    joseki:lockingPolicy              joseki:lockingPolicyMRSW ;
    .

joseki:ImplSPARQL
    rdf:type   joseki:ServiceImpl ;
    module:className
        <java:org.joseki.processors.SPARQL> .

joseki:ImplSPARQLUpdate
    rdf:type   joseki:ServiceImpl ;
```

```
    module:className
        <java:org.joseki.processors.SPARQLUpdate> .

# Local Variables:
# tab-width: 4
# indent-tabs-mode: nil
# End:
<#myProjectUpdate>
   rdf:type            joseki:Service ;
   rdfs:label          "My Project SPARQL/Update" ;
   joseki:serviceRef   "sparql/myproject/update" ;
   joseki:dataset      <#myProject> ;
   joseki:processor    joseki:ProcessorSPARQLUpdate .

 <#myProjectRead>
   rdf:type            joseki:Service ;
   rdfs:label          "SPARQL" ;
   joseki:serviceRef   "sparql/myproject/read" ;
   joseki:dataset      <#myProject> ;
   joseki:processor    joseki:ProcessorSPARQL_FixedDS .

 <#myProject>
   rdf:type            ja:RDFDataset ;
   rdfs:label          "My Project" ;
   ja:defaultGraph     <#myProjectDB> .

 <#myProjectDB>
   rdf:type            ja:RDBModel ;
   ja:connection       [
                         ja:dbType "MySQL" ;
                         ja:dbURL <jdbc:mysql://localhost/annotationdb>
;
                         ja:dbUser     "root"     ;
                         ja:dbPassword     "" ;
                         ja:dbClass        "com.mysql.jdbc.Driver"
                        ] ;
   ja:reificationMode    ja:minimal ;
   ja:modelName        "DEFAULT" .
```