Semantic Web Integration with SPARQL Autocomplete

Conference Paper · April 2015

DOI: 10.13140/RG.2.2.16563.43044

Aleksandar Andreevski
Ss. Cyril and Methodius University in Skopje
1 PUBLICATION **5** CITATIONS
SEE PROFILE

Riste Stojanov
Ss. Cyril and Methodius University in Skopje
36 PUBLICATIONS **124** CITATIONS
SEE PROFILE

Milos Jovanovik
Ss. Cyril and Methodius University in Skopje
61 PUBLICATIONS **212** CITATIONS
SEE PROFILE

Dimitar Trajanov
Ss. Cyril and Methodius University in Skopje
158 PUBLICATIONS **624** CITATIONS
SEE PROFILE

Semantic Web Integration with SPARQL Autocomplete

Aleksandar Andreevski, Riste Stojanov, Milos Jovanovik and Dimitar Trajanov
Faculty of Computer Science and Engineering
Ss. Cyril and Methodius University in Skopje, Macedonia
andreevski.aleksandar@students.finki.ukim.mk, {riste.stojanov, milos.jovanovik, dimitar.trajanov}@finki.ukim.mk

***Abstract*—This project is intended to ease the writing process of dynamic SPARQL queries for applications. Its goal is to make an autocomplete form that can be reused in different applications and will be up to date with the latest ontologies, thus making the process of using Linked Open Data closer to application developers in general. This is done by having a server with an API that returns a JSONP format for each SPARQL query sent by the user application, i.e. the autocomplete form. The autocomplete feature is implemented with AngularJS and helps the user with writing the SPARQL keywords, the ontology classes and properties.**

***Index Terms*—Semantic Web, SPARQL, AngularJS, Autocomplete**

I. Introduction

The current Web is comprised of text and media files such as pictures, audio and video files. This kind of data representation makes it very easy for humans to understand and absorb the knowledge from that data. However, it is not suitable for the computers, since they see the data structurally without ”understanding” its meaning. Thus, the computers don't understand this data, and they can not fetch, select, filter, combine, aggregate data for us, without someone analyzing that data. This is where the Semantic Web comes in. The Semantic Web represents a system that makes it easier for machines to process and ”understand” web data and also to respond to complex human requests based on their meaning. All that the humans need to provide is relevant information about that data. The Semantic Web is bridging the gap between the human understandable meaning and computer understandable structure with the standards that define representation structure of the data meaning.

The knowledge in the Semantic Web is mostly represented through several formats, such as RDF[1], RDFS[1] and OWL[2]. One can query this knowledge using the SPARQL query language[3]. SPARQL is able to query files containing RDF data regardless if they are exposed on the Web or stored in a local database. Furthermore, SPARQL can query multiple data sources at once and dynamically build a virtual RDF graph from all those sources. It is intended for revealing facts from the semantic data by the applications and expert users that know its syntax.

The base of the SPARQL syntax is similar to the SQL syntax, and their difference is that SPARQL is designed to work with path expressions for graph querying. In the query languages such as SPARQL, the one that writes the query should know the underlying data and its relations. Even though the base of the SPARQL syntax is simple, its main complexity comes from the huge amount of data that is being queried, since each data set can contains millions of triples from different domains. It is hard for a human to remember all the resources from some data set, and it is often practice for the SPARQL users to write several additional queries in order to reveal the underlying data, represented with resources identified by their URIs. In SPARQL queries, the resources must be referenced with their URIs.

In this paper we describe a SPARQL autocomplete system that solves the problem of remembering resource URIs by the users, and significantly speeds up the query writing process. In Section II we describe the SPARQL query building systems with their features. In Section III we describe our SPARQL autocomplete system with the technical and architectural features, and we compare it with similar systems in Section IV. The last section Section V gives the conclusion and describes the future steps for this project.

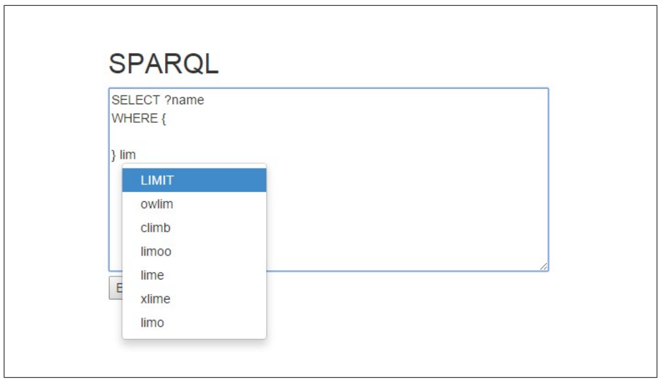
II. Related Work

There are a few SPARQL query builders that exist. In general they can be split in two groups:

a) Graphical query builders: These query builders lean towards the idea of graphical representation of the resources, variables, type of query. They can be in a form of building a graph [4] where every node is a variable, resource or literal and by choosing the type of query, the system (program) will build the query for you. The other form is building the query based on filling in forms of what the user wants the query to achieve, which would the variables be, selecting the resources from lists, etc. These query builders are wizard-like or have an interface that starts building the query like a tree starting from the query type as a root, and then restricting the possible terms and resources to choose from with each choice made in the branches [5].

One of the main advantages of the graphical query builders is that the person using them does not have to know much about SPARQL, or can just have a basic knowledge of it to construct a query. The user interface is pretty straight-forward and easy to learn for new users. The downside is that in some builders the query takes more time to build due to the conversion from its graphical form to a proper SPARQL query.

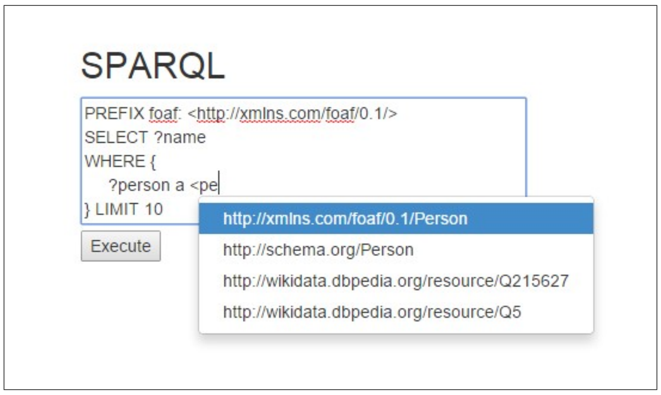
[1]http://www.w3.org/RDF/



```
SELECT ?name
WHERE {

} lim
```

Fig. 2. *Syntax words autocomplete.*


SPARQL
Execute
http://xmlns.com/foaf/0.1/Person
http://schema.org/Person
http://wikidata.dbpedia.org/resource/Q215627
http://wikidata.dbpedia.org/resource/Q5

```
PREFIX foaf: <http://xmlns.com/foaf/0.1/>
SELECT ?name
WHERE {
   ?person a <pe
} LIMIT 10
```

Fig. 3. *Ontology terms URL autocomplete.*

b) Autocomplete or Textcomplete query builder: These builders are meant for users who have knowledge of SPARQL query writing. They will not create the query for you; instead they will help you writing it. The general idea is to give the user suggestions what to write next or in some builders even check the validity of the current state of the query. Flint[2] and Squebi[3] are text editors for building SPARQL queries that enable this kind of autocomplete and query validation. Although these kind of query builders may not be visually rich as the graphical query builders, they do provide in-depth control of the query and are also faster in terms of query creation time. The query is built by the user and it is validated with every next letter the user types, so the query is ready instantaneously after each new letter. One problem with this kind of builders is that the autocomplete is only for reserved syntax words and maybe some small set of the most popular prefixes, classes and properties. This leaves the user with the problem of finding the URIs (URLs) of the desired resources so he/she can build the desired query. Most of the graphical query builders share this problem too.

Regardless of the type of the query builders, they can be hosted on the Web as part of a web page or they can be a desktop editor or an application. The editor and application types are bound to the user system and the user can use them from there only. The ones hosted on the Web are more flexible in that aspect. They are service-oriented, can be used from everywhere and will always be up-to-date.

III. SPARQL Autocomplete System

The system presented in this paper provides SPARQL query builder that autocompletes the user text with suggestions on what to write next. It also allows query execution against any SPARQL endpoint provided by the user. The SPARQL queries are written in the HTML textarea, which is the main component in the project. The query is automatically completed with SPARQL syntax terms, the classes and properties from the most widely used ontologies, and also supports the most commont ontology prefixes. When a prefix is chosen, it is automatically added in the prefix clause of the query along with its full URI.

[2]http://openuplabs.tso.co.uk/demos/sparqleditor
[3]https://github.com/tkurz/squebi

The SPARQL autocomplete editor is developed as a reusable web component which can be integrated in any web page and can query any SPARQL endpoint. Since there are endpoints that doesn't support the JSONP[4] protocol, and the browsers doesn't support cross site scripting[6], the editor communicates with a dedicated server via the JSONP protocol that provides the autocomplete hints for the users. The architecture of the system is shown in Figure 1.

A. *SPARQL Autocomplete Server*

The server is responsible for indexing and storing the classes and the properties from the most common SPARQL endpoints. In the current version, the server provides the classes and properties provided by the Linked Open Vocabulary project[5], but it is not limited to them. The server also provides an interface for adding new SPARQL endpoints; when a new endpoint is added, the system queries it and retrieves all the classes and properties available through it. For the prefixes, the server provides the one registered in the Linked Open Vocabulary project, combined with the prefixes defined in the crowd sourced prefix.cc platform[6].

The server has scheduled tasks that re-index the classes, properties and prefixes from the registered sources. In the current version, these scheduled tasks are executed weekly.

The server provides several functonalities. The first one is to serve the script for the autocomplete editor widget. This widget is described in details in Section III-B. The second function is to provide the resources needed for autocompleting the user query. The server supports two modes for communication. In the first mode it transfers JSON objects that are suitable for all clients. However, if the editor is reused in a page with a different domain than the one of the server, this way of communication does not work, since the browsers block the cross side scripting. Because of this, the server also supports communication via JSONP, where the data is sent to a callback provided by the caller.

[4]http://en.wikipedia.org/wiki/JSONP
[5]http://lov.okfn.org
[6]http://prefix.cc/

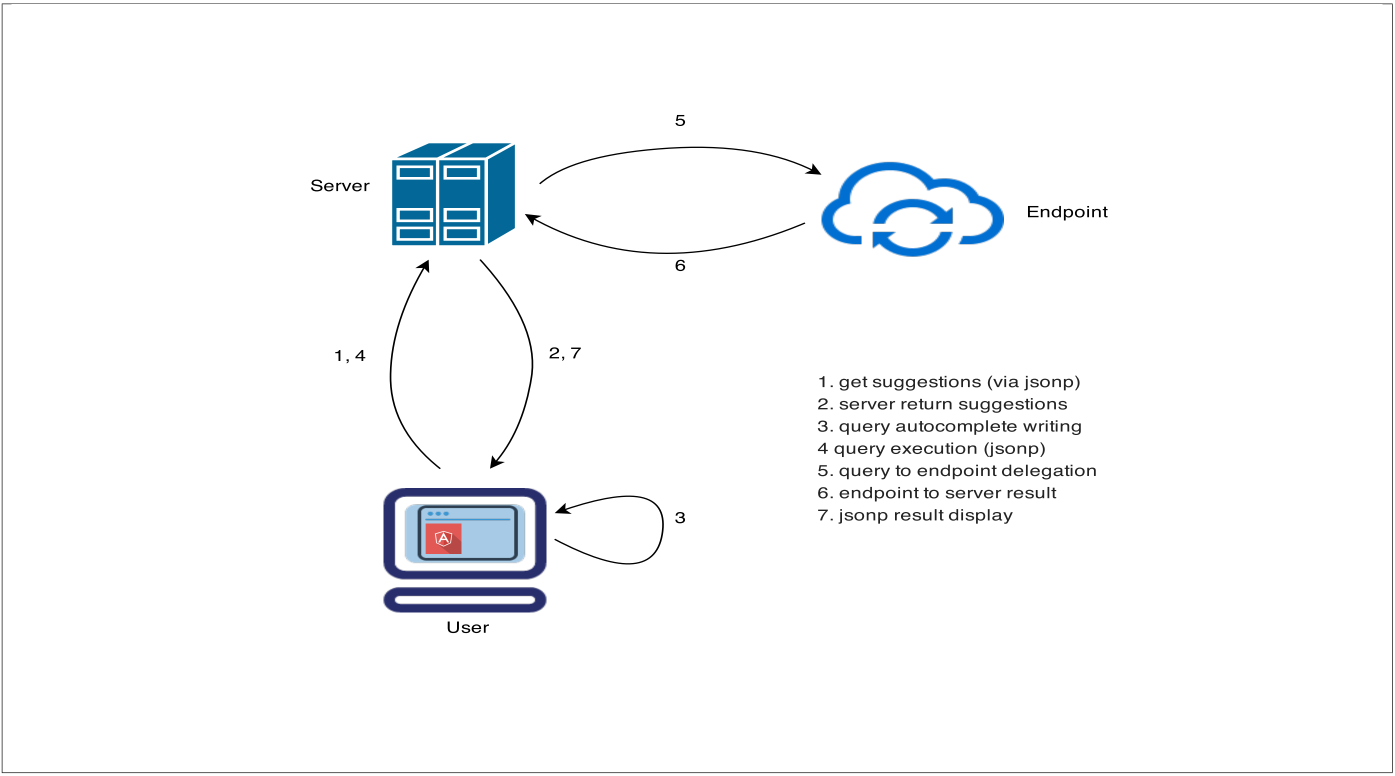

5
Server
Endpoint
6
1, 4
2, 7
1. get suggestions (via jsonp)
2. server return suggestions
3. query autocomplete writing
4 query execution (jsonp)
5. query to endpoint delegation
6. endpoint to server result
7. jsonp result display
3
User

Fig. 1. *System architecture.*

B. SPARQL Autocomplete Editor

The SPARQL autocomplete editor is developed as an Angular JS[7] directive, which can be reused in every web project. The directive provides configuration of the accepted SPARQL endpoints, a default endpoint and the appearance of the component. It can be reused in every web project, and its only dependencies are AngularJS and Text Complete JQuery plugin[7]. This way, each developer can reuse the features of this editor in his/her project. In this mode of functioning, the directive is hosted on a different domain than the autocomplete server, and in order to communicate with eachother, the JSONP protocol is used. This way, the server sends the result as an argument to a callback Javascript function.

Additionally, the system provides an easier way of integration via the widget. This widget is included with an embedded Javascript, hosted by the autocomplete server. This script takes the ID of the target HTML element as an argument, and once it is loaded, it displays the editor at the position of the provided element. The embedded Javascript has all of the dependencies with it, so there is no additional configuration necessary.

The SPARQL autocomplete editor is a custom directive that displays a textarea with auto-completion dropdown list shown while the user types in it. The directive additionally provides list of available endpoints and allows query execution against the selected endpoint.

The autocomplete functionality uses the open source JQuery plugin(Footnote 7) that allows definition of suggested words based on regular expression for matching the text typed in by the user. The component allows definition of a function for narrowing the suggestions, the template for the suggestion list, what to replace after the autocompletion and many more options. There are three matching patterns implemented in the current version:

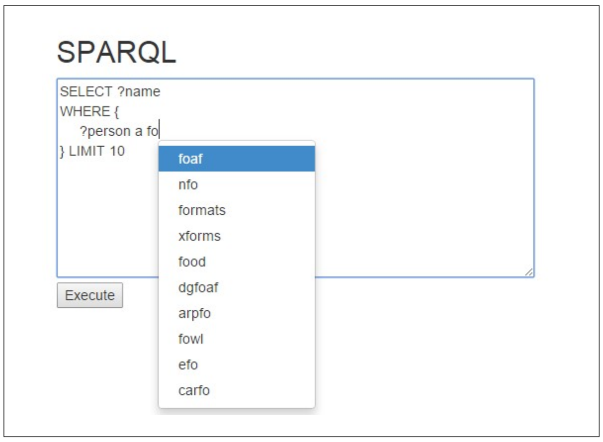
- SPARQL syntax words
- <resource uri>
- ontology prefix:resource

All of these words are retrieved from the SPARQL autocomplete server and are loaded into the directive. The SPARQL syntax words are hard-coded in an array, because they represent a fixed set of words defined by the SPARQL W3C standard. These words are suggested after a single typed letter, as shown in Figure 2.

The next group of words are shown when the user types the '<' symbol and means that the user can search the ontology terms by their URIs. Once the '<' is typed, a set of URIs is retrieved by the AngularJS service. The service loads these URIs by sending a request to the server who then delegates the response to the AngularJS service. After the response is retrieved by the service, it will be processed and returned as an array to the autocomplete directive. Figure 3 shows the URL autocompletion.

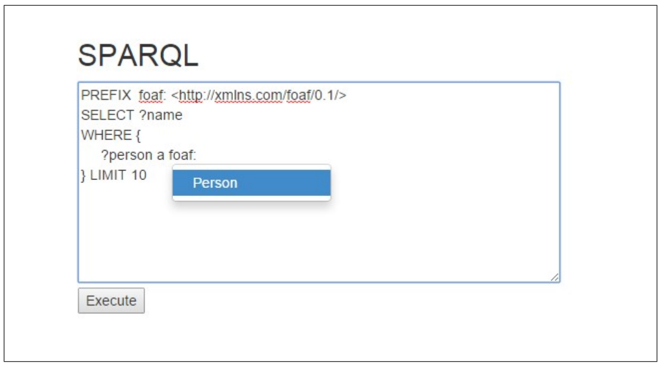
The last word group is the the ontology prefixes group followed by the ontology resources. Autocomplete suggestions are given just after one typed letter. Until the ':' symbol is typed or the user chooses one of the prefixes from the list, the autocomplete shows suggestions only for the ontology prefixes. They are retrieved from http://prefix.cc but this time

[7]http://yuku-t.com/jquery-textcomplete/



```
SELECT ?name
WHERE {
    ?person a fo
} LIMIT 10
```

Fig. 4. *Prefixes autocomplete.*



```
PREFIX foaf: <http://xmlns.com/foaf/0.1/>
SELECT ?name
WHERE {
    ?person a foaf:
} LIMIT 10
```

Fig. 5. *Sufixes autocomplete.*

through a direct AJAX[8] call from the AngularJS service itself, not the server. The response contains the most popular ontology prefixes along with their URIs. The prefixes are then sent to the autocomplete script, while the prefix-uri mapping is kept in the service (see Figure 4). After the ':' is being typed, the autocomplete directive tries to match whether the prefix URI is a prefix of some of the URIs we have stored (the URIs retrieved from http://seminant.com/). If there is a match, those matches are shown as suggestions and when the user chooses one of them it is shown in the query as "prefix:resource", while the whole URI is added at the beginning of the query in the prefixe clause (see Figure 5).

IV. Discussion

In the Section III a full description of our autocomplete query builder was given. In this section a comparison is made with the SPARQL query builders and editors from the plain autocomplete category. Let us start with what this editor can not do. Our editor lacks the ability to perform a query validation, while Flint supports this feature by having all the syntax words and the ontology terms categorized so it can apply ordering rules to them. Another drawback is the lack to format the query. This version does not adds tabs or new lines after writing a triple in the query like in Squebi. Coloring of words is also not supported because of the constraints of a HTML textarea and is not intended to support this feature.

Now let us see what is the advantage of our editor over the others. All of the other cited editors have only a few ontologies that they offer as autocomplete suggestions. They are hard-coded in their implementation, while our system fetches them as described previously. The other builders only have autocompletion for the few most common prefixes and some ontology terms URLs. We offer the most used ontologies, about one thousand in size, their prefixes, URLs and terms. That was the general idea for starting this kind of project. This was the common thing all the other SPARQL editors had as a drawback. All of their focus was on either formatting, coloring, visualizing the query or the results. As for our autocomplete system, the goal was to make the autocompletion more powerful, thus easing the query writing process.

V. Conclusion and Future Work

In this paper a review was given of our SPARQL autocomplete system. The idea behind it was described and then elaborated through the project structure and logic. Finally, a comparison was made with the other SPARQL query builders and listed its flaws and advantages over them. After the previous section, it was concluded that our autocomplete system reduces the time to construct a query by not having to search for the ontology terms and resources by ourselves; they are rather provided as suggestions.

In our future work we plan to change the preview of the results. For now we use an AngularJS Table containing links of triplets. We intend to create a separate AngularJS component (directive) to show the result in a graph. The general plan is to have the graph show the resources and to display some metadata about a node on hover. We also plan to support navigation from a node to other nodes (resources) that have an outgoing connection to the one that we navigated from.

References

[1] D. Brickley and R. V. Guha, "Resource description framework (rdf) schema specification 1.0: W3c candidate recommendation 27 march 2000," 2000.
[2] D. L. McGuinness, F. Van Harmelen *et al.*, "Owl web ontology language overview," *W3C recommendation*, vol. 10, no. 10, p. 2004, 2004.
[3] S. Harris and A. Seaborne, "Sparql 1.1 query language," *W3C Recommendation*, vol. 21, 2013.
[4] A. Russell and P. Smart, "Nitelight: A graphical editor for sparql queries," 2008.
[5] O. Ambrus, K. Möller, and S. Handschuh, "Konduit vqb: a visual query builder for sparql on the social semantic desktop," in *Workshop on Visual Interfaces to the Social and Semantic Web*, 2010.
[6] K. Spett, "Cross-site scripting," *SPI Labs*, pp. 1–20, 2005.
[7] P. B. Darwin and P. Kozlowski, *AngularJS web application development*. Packt Publ., 2013.
[8] J. J. Garrett *et al.*, "Ajax: A new approach to web applications," 2005.