YASGUI: Not Just Another SPARQL Client*

Laurens Rietveld[1] and Rinke Hoekstra[1,2]

[1] Department of Computer Science, VU University Amsterdam, The Netherlands
{laurens.rietveld,rinke.hoekstra}@vu.nl
[2] Leibniz Center for Law, University of Amsterdam, The Netherlands
hoekstra@uva.nl

Abstract. This paper introduces YASGUI, a user-friendly SPARQL client. We compare YASGUI with other SPARQL clients, and show the added value and ease of integrating Web APIs, services, and new technologies such as HTML5. Finally, we discuss some of the challenges we encountered in using these technologies for a building robust and feature rich web application.

Keywords: SPARQL, endpoints, API, Web service, Semantic Web.

1 Introduction

Developers that use traditional Web technologies are pampered with full-featured development tools such as in-browser debugging, integrated development environments, increasingly simple and lightweight services (RESTful APIs), and broad take up in industry. Semantic Web technologies have some catching up to do. The recent start of the W3C Linked Data Platform working group[1] is a good step in bringing triple-store querying closer to traditional RESTful APIs. However, the ingenuous developer wanting to have a first taste of Linked Data is scared away by austere clients for a rich but complex query language: SPARQL.

Indeed, several good RDF programming libraries exist, but uptake of these still relies on a good understanding of SPARQL and the underlying Semantic Web paradigm which can only be attained with simple, lightweight and user friendly clients for interacting with Linked Data. This observation holds for Semantic Web savvy developers as well: trying and testing SPARQL queries is often a cumbersome and painful experience: all who know the RDF namespace by heart raise their hands now! A related question that is hard to answer for many: "Where is that Linked Data?". Most of us will know the DBPedia endpoint URL, but can perhaps mention only a handful of other endpoints in total.

This paper introduces yet another SPARQL GUI (YASGUI[2]), a SPARQL client that shows the added value of combining Web 2.0 and Semantic Web technologies [1,2] for providing a more gentle Linked Data interaction environment. We find that most existing SPARQL clients do not offer functionality that

* This work was supported by the Dutch national program COMMIT.
[1] See http://www.w3.org/2012/ldp/wiki/Main_Page
[2] See http://aers.data2semantics.org/yasgui/

P. Cimiano et al. (Eds.): ESWC 2013, LNCS 7955, pp. 78–86, 2013.

goes far beyond a simple HTML form. These implementations convey a rather *narrow* interpretation of what a SPARQL client interface should do: POST (or GET) a SPARQL query string to an endpoint URL. As a result, they currently offer only a selection of the features that we, as a community, could offer to both ourselves as well as new users of Semantic Web technology.

YASGUI is a web-based SPARQL client that functions a a wrapper for both remote and local endpoints. It integrates linked data services and web APIs to offer features such as autocompletion and endpoint lookup. It supports query retention – query texts persist across sessions – and query permalinks, as well as syntax checking and highlighting. YASGUI is easy to deploy locally, and it is robust. Because of its dependency on third party services, we have paid extra attention to graceful degradation when these services are inaccessible or produce unintelligible results

The following sections give a brief overview of the featureset of the current state of the art features of SPARQL clients, followed by a more detailed description of YASGUI.

2 SPARQL Client Features

Table 1 lists eleven currently existing SPARQL clients, ranging from very basic to elaborate. The features of these clients fall into several categories, *syntactic* features (autocompletion, syntax highlighting and checking), *applicability* features (endpoint or platform dependent or independent) and *usability* (query retention, results rendering and download, quick evaluation). This section describes these features in more detail, and discusses whether and how various SPARQL clients implement these features.

2.1 Syntactic Features

Most modern applications containing textual input support *autocompletion*. Examples are the Google website which shows an autocompletion list for your search query, or your browser which (based on forms you previously filled in) shows autocomplete lists for text inputs. One advantage of autocompletion is that it saves you from writing the complete text. Another advantages is the increase in transparency, as the autocompletion suggestions may contain information the user was not aware of. The latter is especially interesting for SPARQL, where users might not always know the exact prefix he/she would like to use, or where the user might not know all available properties in a triplestore. The only SPARQL interface that currently makes use of this functionality is the FLINT SPARQL Editor[3], which uses autocompletion to suggest classes and properties.

Syntax highlighting is a common functionality for programming language editors. It allows user to distinguish between different properties, variables, strings, etc. The same advantage holds for query languages such as SPARQL, where

[3] See `http://openuplabs.tso.co.uk/demos/sparqleditor`

you would like to distinguish between literals, URIs, query variables, function calls, etc. The only SPARQL editor currently supporting syntax highlighting is the FLINT SPARQL Editor, which uses the CodeMirror JavaScript library[4] to bring color to SPARQL queries.

Most Integrated Development Environments (IDEs) provide feedback when code contains syntax errors (i.e. *syntax checking*). Feedback is immediate, which means the user can spot syntax errors in the code without having to execute it. Again, such functionality is useful for SPARQL editing as well. Immediate feedback on a SPARQL syntax means the user can spot invalid queries without having to execute it on a SPARQL endpoint. The FLINT SPARQL editor supports syntax checking by means a JavaScript SPARQL grammar and parser.

2.2 Applicability Features

There are only few clients who allows access to multiple endpoints. Most triple-stores provide a client interface, linking to that specific endpoint. They are *endpoint dependent*. Examples are 4Store [6], OpenLink Virtuoso [7], OpenRDF Sesame Workbench [4] and SPARQLer[5]. More generic clients are the Sesame2 Windows Client [4], Glint[6], Twinkle[7] and SparqlGUI[8]. Other applications fall somewhere in between. The FLINT SPARQL Editor only connects to endpoints which support cross-domain JavaScript (i.e. CORS enabled). This is a problem because not all endpoints are CORS enabled, such as FactForge, CKAN, Mondeca or data.gov. Other editors support only XML or JSON as query results, such as SNORQL[9] (part of D2RQ [3]), which only support query results in SPARQL-JSON format.

Platform (In)dependence increases the accessibility of a SPARQL client. The user can access the client on any operating system. Web interfaces are a good example: a site should work on any major browser (Internet Explorer/Firefox/Chrome), and at least one of these browsers is available for any type of common operating system. Examples are Virtuoso, 4Store and the Flint SPARQL Editor. Another example of multi-platform support is the use of a .jar file (e.g. Twinkle), as any major operating system supports Java and executing Java archives. Examples of single-platform applications are Sesame2 Windows Client and SparqlGUI: they require Windows.

2.3 Usability Features

Query retention allows for easy re-use of important or often used queries. This allows the user to close the application, and resume working on the query later. An example is the 'Query Book' functionality of the Sesame Windows Client.

[4] See `http://codemirror.net/`
[5] See `http://www.sparql.org/`
[6] See `https://github.com/MikeJ1971/Glint`
[7] See `http://www.ldodds.com/projects/twinkle/`
[8] See `http://www.dotnetrdf.org/content.asp?pageID=SparqlGUI`
[9] See `https://github.com/kurtjx/SNORQL/`

Quick evaluation or testing of a graph generated by the user should not require the hassle of installing a local triplestore. Ideally, this functionality would be embedded in the SPARQL client application itself. Most applications requiring a local installation on the users computer support this feature, such as Twinkle. The Sesame Windows Client supports file uploads as well, though it requires a local triplestore which implements the OpenRDF SAIL API.

Query results (such as JSON or XML) for SELECT queries are often relatively difficult to read and interpret, especially for a novice. A *rendering* method which is easy to interpret and understand is a table. All applications except 4Store support the rendering of query results into a table. Because of the use of persistent URIs, we would expect navigatable results for resources, e.g. in the form of drawing the URIs as hyperlinks. This feature is not supported by some applications, such as Virtuoso, Twinkle or SparqlGUI. SNORQL is an application with an elaborate way of visualizing the query results. Besides allowing the user to navigate to the page of the URI, the user can click on a link to browse the current endpoint for resources relevant to that URI.

Downloading the results as a file allows for better re-use of these results. A user might want to avoid running the same heavy query more than once, and use the results stored as a file instead. Additionally, the results of CONSTRUCT queries are often used in other applications or triplestores. Saving the user from needing to copy & paste query results clearly improves user experience as well. The only application that does not support the downloading of results, is the FLINT SPARQL editor.

Most of the clients described above are restricted to one simple task: accessing information behind a SPARQL endpoint. However, equally important to this task is assisting the user in doing so. This is something where all but one applications fail. Regrettably, the one interface with a user-friendly interface (FLINT SPARQL editor) falls short in the important feature of accessing all endpoints. We conclude that currently no single endpoint independent, accessible, user-friendly SPARQL client exists.

3 The YASGUI SPARQL Client

In this rest of this section we discuss the architecture, features and design considerations of YASGUI (Figure 1) and compare them to other clients.

3.1 Architecture

YASGUI is built using SmartGWT toolkit[10], jQuery[11], and uses new HTML5 functionalities such as local storage and client-side generation of files. Some of the newest HTML5 functionalities are not supported by outdated browsers and Internet Explorer. This degradation is handled gracefully: access via an incompatible browser results in a notification to the user and disabled features (such

[10] See `http://www.smartclient.com/product/smartgwt.jsp`
[11] See `http://jquery.com/`

Table 1. SPARQL client feature matrix

Feature	4Store	OpenLink Virtuoso	SNORQL	SPARQLer	Sesame Workbench	Sesame2 Windows Client	Glint	Twinkle	SparqlGUI	Flint SPARQL Editor	YASGUI
Autocompletion	-	-	-	-	-	-	-	-	-	+ [a]	+ [b]
Syntax Highlighting	-	-	-	-	-	-	-	-	-	+	+
Syntax Checking	-	-	-	-	-	-	-	-	-	+	+
Multiple Endpoints	-	-	-	-	-	+	+	+	+ [c]	+/- [c]	+
Query retention	-	-	-	-	-	+	+	-	+	-	+
File upload	-	-	-	-	+	+/- [d]	-	+	+	-	- [e]
Platform independent	+	+	+	+	+	-	-	+	-	+	+
Results rendering	-	+/- [f]	+	+/- [f]	+	+/- [f]	+/- [f]	+/- [f]	+/- [f]	+	+
Results download	+	+	+	+	+	+	+	+	+	-	+

[a] Autocompletion of properties and classes available in the triple store
[b] Autocompletion of prefixes/namespaces, support for properties and classes is a planned feature.
[c] Can deal with a limited number of endpoints, e.g. only CORS enabled ones.
[d] File upload requires a local triple store that implements the OpenRDF SAIL API, e.g. OpenRDF Sesame or OpenLink Virtuoso.
[e] File upload is a planned feature, using the rdfstore-js client side triple store.
[f] The rendering does not use hyperlinks for URI resources.

as downloading of files, or client-side caching of large objects). The decision to use HTML5 is motivated by the increasing support of the standard by major browsers. The server-side part of YASGUI is responsible for some of the communication with external services and endpoints. Communication with SPARQL endpoints is done using the Jena library [5]. External services used by YASGUI are CKAN[12], Mondeca[13] and Prefix.cc[14] (see section 3.2), and bitly[15] (see section 3.4).

3.2 Syntactic Features

Two existing libraries provide support for syntax *highlighting* and *checking* in YASGUI: The CodeMirror JavaScript library, which is an extensive JavaScript library for highlighting code, and a JavaScript SPARQL grammar of the FLINT SPARQL Editor. Given this grammar, CodeMirror applies the highlighting to the SPARQL query. Additionally, CodeMirror provides a well documented API

[12] See `http://semantic.ckan.net/sparql`
[13] See `http://labs.mondeca.com/endpoint/ends`
[14] See `http://prefix.cc/`
[15] See `http://bitly.com`

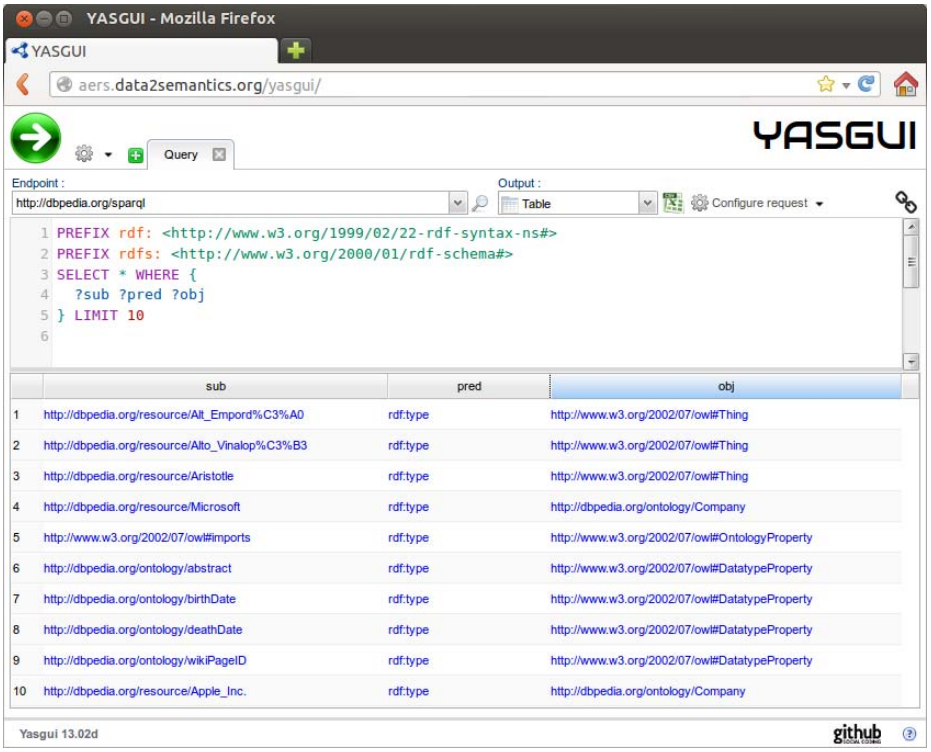

Fig. 1. Screenshot of the YASGUI interface

to parse and dissect the SPARQL query, useful for other YASGUI features such as prefix autocompletion. Both libraries are well documented, well maintained, extendible and easy to use. The existence of both libraries illustrate the availability of elaborate open source project, and the small amount of effort it takes to integrated them into an application.

YASGUI uses the RESTful API of Prefix.cc to perform *autocompletion* of namespace prefixes: full namespace URIs are completed as you type. We furthermore rely on the CKAN SPARQL endpoint for endpoint URL autocompletion and search. This autocompletion feature functions by matching the partially typed endpoint, with the list of endpoints (and their descriptions). The CKAN endpoint provides access to the CKAN datahub.io[16] catalogue of datasets. Users can either use a simple autocompletion combobox, or browse through a list of endpoints in a table. Our fallback option for CKAN is the endpoint provided by Mondeca. Mondeca hosts a project where the availability (up-time) of these endpoints is published. Both endpoints have proven to be difficult to use and access for our purposes. The CKAN endpoint is rather unreliable in up-time, and the Mondeca endpoint often return syntactically invalid XML. In the

[16] See `http://datahub.io`

implementation of YASGUI we try to handle both issues as gracefully as possible. The list of endpoints provided by CKAN or Mondeca is cached on the YASGUI server. If YASGUI fails to retrieve the list in real time from either of the endpoints, we fall back to the cached results.

Another issue with CKAN (and to a lesser extent Mondeca) is the reward model for adding and maintaining the catalogue: there is little incentive for owners of a dataset to add it to CKAN, and even less incentive keep the information up to date (e.g. when the endpoint is down or moved). As a result, CKAN is cluttered with outdated information, and some endpoints are missing. This is partly compensated by Mondeca, which allows filtering by endpoints which are up, though incorrect or missing information still persists. The reward model employed by Prefix.cc is the opposite: the content is crowd-sourced (anybody can add prefixes), and voting is used to to deal with conflicting prefix definitions. Users of prefix.cc have an incentive to keep the information up to date and as correct as possible. As a result, the information retrieved from prefix.cc is more reliable and usable than information from CKAN and Mondeca.

3.3 Applicability Features

As mentioned in section 2, client-side web applications such as the FLINT SPARQL Editor are *endpoint independent*, but only work on CORS-enabled endpoints. To overcome this limitation, YASGUI includes a server-side proxy for accessing endpoints that do not support CORS. For endpoints which do support cross domain JavaScript, YASGUI executes the queries solely from the clients side via JavaScript. The only scenario where YASGUI fails to connect to an endpoint is where a locally installed endpoint is unreachable from the web, operating on a different port than YASGUI, and CORS-disabled. Here, the YASGUI proxy is not able to access the client. Because of the CORS restriction, YASGUI is not able to access the endpoint via JavaScript as well, as it is operating on a different port. We consider this issue to be minor: because the endpoint is installed locally, the user will have access to change its CORS settings, or even run the endpoint via a different port.

Other than dealing with the accessibility issues of CORS disabled sites, endpoint independent clients should support configurable requests. For instance, some endpoints may only support the XML results format, or allow the use of additional request parameters, such as the 'soft-limit' of 4Store. Such endpoints can only be used to their full potential if users are able to specify these additional arguments manually. Therefore, YASGUI supports the specification of an arbitrary number of request parameters for every endpoint.

Finally, we had to add quite some code to deal with possible errors returned by endpoints. The SPARQL protocol specifies what the endpoint request and response should look like, but leaves error handling unspecified: what HTTP error code should be sent by an endpoint, and how should error messages be communicated. As a result, triple stores come with various ways of conveying errors. Some endpoints return the error as part of an HTML page (with the regular 200 HTTP code), or as a SPARQL query result. Others only return an

HTTP error code, where only some include a reason phrase together with the error code. The latter is a best practice for RESTful services. The absence of a standard, and the failure to adhere to best practices, makes a generic robust error handling solution messy and difficult to implement. Developing such a solution requires coding and testing by trial and error, and test queries on as many different endpoints as possible.

3.4 Usability Features

As Table 1 shows, most SPARQL clients support both *rendering* and *downloading* of query results to some extent. YASGUI does both as well. Users can render results either as a lightweight HTML table (for large numbers results), an elaborate sortable/groupable table, or show the raw query results with syntax highlighting. Tables can be downloaded as CSV, where raw query results are available for download 'as is'.

YASGUI stores the application state, making this application state persistent between browser/user sessions: a returning user will see the screen as it was when she last closed the YASGUI browser page. We elaborate on this feature by providing query permalink functionality. For a given query and endpoint combination, YASGUI creates a link. Opening the link in a browser will open YASGUI with the specified query, endpoint and request arguments filled in. We believe this is a welcome feature for people working together with a need to exchange queries. Other than regular query permalinks, YASGUI supports the generation of shortlinks as well using the Bitly URL shortener service. Using an external service such as bitly poses some limitations: their is a fair use policy, limiting the scalability of this solution. Additionally, there is a danger of link-rot as well. An alternative to an external URL shortener service is implementation of such a service as part of YASGUI. This deals with both the issues of scalability as well as link-rot. We opted for the external bitly service, as the danger of link-rot is low with a popular service such as bitly. Whenever scalability becomes an issue, the alternative of implementing our own service would still be a viable option.

4 Discussion

In the preceding sections we described YASGUI and compared it to other clients. YASGUI shows how straightforward it is combining Web APIs, libraries and new Web technologies. Compared to other clients, YASGUI is the first client that really leverages the tools and services we as a community have developed for ourselves.

In the process, we encountered 4 challenges in using WEB APIs and Linked Data together. First, maintenance of Web 2.0 content is a challenge. External services such as CKAN contain outdated, incorrect or incomplete information (section 3.2). The main challenge here is how to manage this information, e.g. using a reward model where users have an incentive to update the information, or

active curation by the service manager. Secondly, standard adherence (or lack of standards) is a challenge (section 3.3), such as the different error handling approaches implemented by endpoints. Additionally, graceful degradation (section 3.2) is an issue. The before-mentioned challenges, as well as issues such as external services breaking down, should not break the application. In the worst case, it should only break application features. Finally, it is worth considering whether or not to use online web services. Use of online web services such as url-shorteners or endpoint catalogues create external dependencies. For some services such as CKAN, graceful degradation is possible. However, for other services such as url-shorteners, this is not the case. This is a trade-off. Instead of using online web services, an application specific implementation may be preferable and more robust.

In general, our experience is that there is an abundance of libraries, new Web technologies, services, and APIs. The use of these tools increases the feature set of your application, and decreases the number of lines you have to write.

References

1. Ankolekar, A., Krötzsch, M., Tran, T., Vrandečić, D.: The two cultures: Mashing up Web 2.0 and the Semantic Web. Web Semantics: Science, Services and Agents on the World Wide Web 6(1), 70–75 (2008)
2. Battle, R., Benson, E.: Bridging the semantic Web and Web 2.0 with Representational State Transfer (REST). Web Semantics: Science, Services and Agents on the World Wide Web 6(1), 61–69 (2008)
3. Bizer, C., Seaborne, A.: D2rq - treating non-rdf databases as virtual rdf graphs. In: World Wide Web Internet and Web Information Systems, p. 26 (2004)
4. Broekstra, J., Kampman, A., Van Harmelen, F.: Sesame: An architecture for storing and querying RDF data and schema information (2001)
5. Grobe, M.: Rdf, jena, sparql and the 'semantic web'. In: Proceedings of the 37th Annual ACM SIGUCCS Fall Conference, SIGUCCS 2009, pp. 131–138. ACM, New York (2009)
6. Harris, S., Lamb, N., Shadbolt, N.: 4store: The design and implementation of a clustered RDF store. In: 5th International Workshop on Scalable Semantic Web Knowledge Base Systems (SSWS 2009), pp. 94–109 (2009)
7. Openlink Virtuoso: Universal server platform for the real-time enterprise (2009), `http://www.openlinksw.com/`