GR4PHP: A Programming API for Consuming E-Commerce Data from the Semantic Web

Alex Stolz, Mouzhi Ge and Martin Hepp

E-Business and Web Science Research Group, Universität der Bundeswehr München
Werner-Heisenberg-Weg 39, D-85579 Neubiberg, Germany
{alex.stolz,mouzhi.ge,martin.hepp}@ebusiness-unibw.org

Abstract. Nowadays, large data collections are made available on-line for free. The liberalized data is predominantly accessed via Web services. The interaction with such Web services is facilitated by RESTful Web APIs and programming libraries that provide a convenient means for Web developers to build intelligent applications and mash-ups. However, for typical Web developers it is still hard to include the growing amount of e-commerce data on the Semantic Web in applications. Fetching useful data from public SPARQL endpoints requires substantial expertise, such as regarding the popularity of competing RDF modeling patterns, data cleansing heuristics, etc. In this paper, we propose an architecture and implementation of a PHP library for consuming e-commerce data in RDF expressed in terms of the GoodRelations vocabulary. Our approach (1) provides an efficient way of fetching information about stores, offers, product models, and opening hours from a SPARQL endpoint, and (2) hides the complexity of SPARQL and GoodRelations modeling patterns. We show that our approach significantly reduces the amount of time for a respective implementation. Furthermore, our approach has been applied successfully in a real-world application.

1 Introduction

Past research in the Semantic Web community has proposed numerous frameworks, best practices and prototypes to facilitate the development and publishing of rich content descriptions. In particular, many recent efforts deal with the proper publication and deployment of RDF data [3]. At the same moment, however, the shortage of developer-friendly programming libraries for querying the Semantic Web limits the *consumption* of semantic data, notably in the field of e-commerce.

The Semantic Web aims at creating meaningful links between resources on the Web yielding to a Web of Data processable by machines [2]. An essential building block of the Semantic Web is the Resource Description Framework (RDF), a standardized data model for representing information about resources on the Web [12]. In contrast to classical relational databases, the distributed RDF data model allows for more sophisticated and intelligent tools. For instance, the first among four Linked Data principles [3] addresses the use of

globally unique resource identifiers (URIs), which paves the way for better data integration at global scope. Furthermore, compared to relational databases with rigid schemata, resources on the Semantic Web are integrated more easily across different vocabularies. However, the gain in flexibility implies rising complexity: SPARQL [13] for example, a protocol and powerful query language for RDF, adds complexity because of the diversity of schemata that exist on the Web of Data. Consequently, for most Web developers it proves difficult to take advantage of content available on the Semantic Web. In particular, it would require them to become familiar with (1) the underlying model of the domain of interest, along with other complementing vocabularies, and (2) the syntax and semantics of a Semantic Web query language, i.e. SPARQL. Although approaches like Semantic Web programming frameworks can provide high-level assistance for developers in consuming structured data, they mostly disregard the domain perspective, e.g. in order to query product data it still requires detailed knowledge about vocabulary patterns in e-commerce. In parallel, the amount of data on the Semantic Web is subject to constant growth which is augmenting the potential benefit for Web application developers. Thus, a solution to overcome the complexity problem of the SPARQL query language with regard to Web development is needed.

The barrier of Semantic Web data consumption can be reduced for Web developers if a respective technology taps into recent Web standards, thereby eliminating learning efforts, because developers usually commit themselves to well-established technologies that feature proper documentation, community support, reference implementations, etc. Current trends in Web technology involve *Web APIs* (Application Programming Interfaces) and *programming libraries*. Web APIs constitute tools that take advantage of single Web services or mash-ups built on disparate Web services. The popularity of Web APIs is mainly due to the success of the light-weight REpresentational State Transfer (REST) paradigm [6, Chapter 5]. Compared to Web APIs, programming libraries (or programming APIs, software APIs) introduce an additional, language-specific abstraction layer that makes implementation in a particular programming language more convenient for Web developers. To our knowledge, there exist no programming libraries so far that focus on mapping between SPARQL queries and language-specific data types within the context of a specific ontology.

In this paper, we adopt from latest Web technology trends and describe the architecture and implementation of a developer-friendly programming library to fetch information about stores, offers, product models, and opening hours from a SPARQL-capable RDF store that contains e-commerce data. In particular, (1) we present a *powerful PHP library*, which empowers PHP developers with the ability to query GoodRelations [9] data from the Semantic Web. Secondly, (2) our approach aims to *hide the complexity of the Semantic Web* from PHP developers, i.e. programmers will not be confronted with details such as RDF, SPARQL, or GoodRelations, which to learn would substantially constrain their productivity. Moreover, our solution comprises manual query optimizations which requires advanced knowledge about design decisions of the underlying SPARQL engine.

The rest of this paper is organized as follows: Section 2 provides background information and reviews previous efforts to circumvent formulating SPARQL queries; Section 3 covers aspects of the programming API including the description of the functions supported, the architecture, implementation details and a demonstration; Section 4 outlines our evaluation methodology and details results and conclusions of our experiments; finally, Section 5 summarizes our work.

2 Background and Related Work

In the following, we discuss previous work for facilitating RDF consumption by use of the SPARQL query language. After that, we introduce GoodRelations, a vocabulary for describing e-commerce data on the Semantic Web that our approach relies on.

2.1 Review of Approaches to Facilitate SPARQL Usage

From a review of existing tools and publications we classified three general approaches to alleviate the complexity of formulating SPARQL queries:

NLP and Question Answering. Many question answering approaches on the Semantic Web translate natural language queries into SPARQL, which is likely the most intuitive way to allow ordinary users to do complex queries. A usability study summarizes four natural language interfaces for the Semantic Web that map natural language input to SPARQL queries [10]. The authors in [14] suggest a question answering approach over Linked Data that generates a SPARQL template according to the semantic structure of the natural language question and populates it with URIs obtained using similarity ranks. [15] propose a framework to translate natural language questions into SPARQL queries to answer questions over structured knowledge bases. [1,7] describe similar approaches for question answering in health-care and in video search.

Query Builders and Query Assistance. [4] propose a graph summary approach to assist users in formulating complex SPARQL queries over heterogeneous data sources with diverse data structures and vocabularies. Query builders represent an alternative approach to circumvent the manual crafting of SPARQL queries. For example, Openlink iSPARQL[1] provides an advanced graphical query builder that abstracts from the details of the SPARQL syntax. In literature, [11] proposed a graphical user interface over biomedical linked open data repositories to assist users in building SPARQL queries without having to know the underlying data structures. Drupal, a semantically enhanced content management system, implements a query builder module for SPARQL to integrate semantic content results into Web pages [5].

[1] http://wikis.openlinksw.com/owiki/OATWikiWeb/InteractiveSparqlQueryBuilderOverview

Programming Frameworks. Many frameworks and software libraries for the Semantic Web incorporate rudimentary query capabilities. However, most popular frameworks require users to provide raw SPARQL queries (e.g. Apache Jena, ARC2, RDFLib). Some frameworks or respective wrappers feature programmatic means to construct SPARQL queries, for instance SuRF for Python/RDFLib, or ActiveRDF for Ruby on Rails.

In contrast to related works, our approach equips PHP Web developers with a precise query mechanism to fetch e-commerce data from SPARQL endpoints, without having to become familiar with the details of the domain model or any other Semantic Web technologies. In our paper, we provide a uniform mechanism to access information from heterogeneous data sources (by virtue of SPARQL), paired with the comfortability of accessing them with a popular high-level programming library.

2.2 GoodRelations Ontology

Our research is based on GoodRelations. GoodRelations[2] [9] is a light-weight vocabulary (or ontology, schema, data dictionary) for e-commerce on the Semantic Web. Its expressivity is targeted at the description of the actual offer of a good and its related entities, i.e. the description of relationships between business entity, offer, and product or service. The ontology provides basic support for the most frequently used properties and individuals in offering descriptions such as product details (e.g. product name, product description, European article number, manufacturer), prices, and terms and conditions. Moreover, it permits to specify legal entities together with their corresponding points of sale, frequently complemented with other vocabularies such as vCard, Friend-of-a-Friend (FOAF), and WGS 84, to specify address details, geo location data and other relevant meta-data, e.g. links to Web pages.

3 Approach for Consuming Semantic E-Commerce Data

In this section, we propose the programming library (*GR4PHP*) to consume GoodRelations data from SPARQL endpoints. First, a high-level view is provided by describing the available API methods. Then, the system architecture is introduced, followed by an explanation of the interplay between the different components of the architecture. After that, the interface of the programming API is discussed in more detail. Furthermore, the steps involved in automatically building up the query that is sent to the endpoint are outlined. Finally, the usage of the programming library is demonstrated on a working example.

3.1 PHP Library for Consuming GoodRelations

Despite the dependencies on GoodRelations and SPARQL, the programming library requires neither modeling nor SPARQL skills to use it.

[2] http://purl.org/goodrelations/

The PHP API provides six abstract functions, namely *getCompany*, *getOffers*, *getProductModel*, *getStore*, *getLocation*, and *getOpeningHours*. The functions constitute a PHP-friendly way to query the most recurring data patterns within the GoodRelations domain. Search criteria supplied by developers along with the function names are internally translated into proper SPARQL queries (cf. Section 3.4). Table 1 lists the PHP functions (method signatures will be specified in Section 3.3) together with the names of the affected GoodRelations concepts. Furthermore, a concise explanation about the functions' capabilities is given. The project code, licensed under a *GNU Lesser General Public License (LGPL)*, together with a more technical description of the API are available on-line[3][4]. Furthermore, we set up a user interface[5] where potential developers can become acquainted with the library.

Table 1. Functions available in the GR4PHP library

Function	**GoodRelations concept name**	**Explanation**
getCompany	*BusinessEntity*	Fetch information (e.g. detailed address descriptions or geo positions) about a legal entity, i.e. either a company or an individual.
getOffers	*Offering*	Retrieve product or service offers and relevant details such as price specification, warranty promises, or validity constraints.
getProductModel	*ProductOrServiceModel*	Return product models and related details that are expressible in the range of the GoodRelations vocabulary. Custom characteristics are supported only via an extension mechanism as will be addressed later in Section 3.3.
getStore	*Location*	Similar to *getCompany*, but information about a point of sale are fetched rather than about a legal entity. Furthermore, a filter to fetch only locations with explicit opening hours statements that are open at query time can be set.
getLocation	*Location*	Compile a list of nearby locations given a global location number (GLN), the location name, or with respect to a particular geo position.
getOpeningHours	*OpeningHoursSpecification*	Return a list of opening hours given the GLN or location name of a point of sale.

[3] http://code.google.com/p/gr4php/
[4] http://www.ebusiness-unibw.org/tools/gr4php/doc/, menu item *GR4PHP*
[5] http://www.ebusiness-unibw.org/tools/gr4php/

The library was developed on and tested against Virtuoso[6] servers of Openlink Software. Nevertheless, most functionality provided is intended to work with arbitrary SPARQL endpoints that adhere to SPARQL 1.0 [13] with one additional requirement. I.e., most queries generated by the library take advantage of the *contains*-function which was added with the advent of SPARQL 1.1 [8].

3.2 Conceptual Architecture

The functions presented above constitute a high-level view on the *GR4PHP* component. In the following, the issue of what happens if a developer invokes one of those functions is addressed. Fig. 1 outlines the conceptual architecture of our approach.

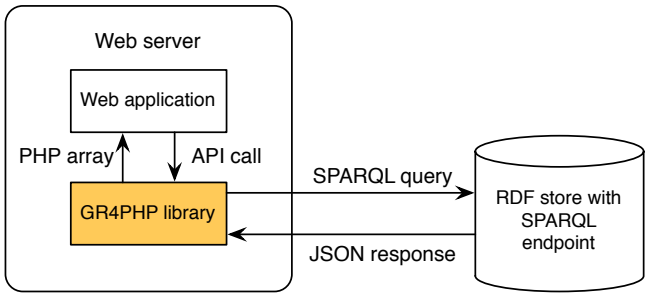

Fig. 1. Conceptual architecture

The highlighted component in Fig. 1 represents the PHP library. The library serves as an intermediary between the Web application and the RDF store, i.e. a Web application can use the API as a wrapper to indirectly communicate with the SPARQL endpoint. This way all complexity is hidden from the developer, which involves in particular (a) crafting a valid and inexpensive SPARQL query, (b) packing it into a HTTP request, (c) sending the request to the remote SPARQL endpoint, and (d) unpacking and converting the HTTP response into a PHP-friendly data type. Consequently, users are only confronted with PHP functions and a uniform way of calling them. We continue with the explanation of request and response flows involved in a function call:

Request. Depending on the type of method invoked and on the input data supplied, the library will build up a respective SPARQL query in an automatic manner. The PHP API will return an immediate error message if the parameters supplied are wrong or not feasible, without even contacting the SPARQL endpoint. If the input parameters proved valid according to the PHP API, the

[6] http://virtuoso.openlinksw.com/

generated query is further relayed to the SPARQL endpoint of the RDF store that may contain and subsequently return the relevant data.

Response. Once the results become available, the library proceeds with taking care of the response by converting the JSON results into a PHP array, ready to be processed further by the Web application. Most SPARQL endpoints provide mechanisms to return response messages with JSON-encoded results, e.g. via RESTful HTTP requests with proper URI parameters supplied.

3.3 API Interface

In the following, we specify how developers can interact with the library and invoke the supported functions. First of all, developers have to indicate the endpoint they want to connect to and execute queries on. The configuration lists a couple of popular endpoints for linked open commerce (that contain GoodRelations), but user-defined endpoint URIs can also be supplied. Furthermore, developers can provide an optional timeout value for the execution of the HTTP request. Like with any other conventional API interface, a function can be called and executed after setting up a connection. The function call may be preceded by preliminary tasks like configuration or customizations, such as binding prefixes for querying custom, application-specific properties in the dataset.

As outlined in Section 3.1, *GR4PHP* provides six distinct functions for consuming GoodRelations data from a SPARQL endpoint. All functions share a uniform interface comprising up to five input parameters, as listed below:

- *inputArray*: Provides filter constraints as an associative array with key-value-pairs to narrow down the result set; keys serve to select an appropriate template from a set of possible triple pattern templates, and values are used to match literals or to set limits (e.g. *maxPrice*) in FILTER constraints.
- *wantedElements*: Projection feature that indicates the subset of elements that should be returned as an associative array.
- *mode*: Two search modes are supported, namely *:lax* (substring matches) and a more restrictive *:strict* mode (full matches based on strings and datatypes).
- *limit*: Specifies the maximum number of results to be returned (*LIMIT* solution modifier in SPARQL). For the sake of simplicity, clauses for advanced use like *ORDER BY*, *OFFSET*, etc. were not added to the library.
- *searchProperties*: Associative array of user-defined, application-specific properties to be sought for a conceptual entity determined by the function name.

Listing 1.1 shows a method signature in PHP valid for all functions available in the library. The asterisk serves as a placeholder for the various function names. The usage of the square brackets in the signature indicates that the first parameter is mandatory, while the remaining parameters are optional and can be omitted. If optionals are missing, their default values apply. For example, if no *mode* argument is supplied, the default value *Configuration::MODE_LAX* is taken, which is a constant value that stands for *:lax* and denotes the lax search mode. Similarly, if no argument for *limit* is present, the default configuration value applies which amounts to 20.

```
function get*(array $inputArray,              // constraints array
  [array $wantedElements=FALSE],              // selection array
  [string $mode=Configuration::MODE_LAX],     // search mode
  [int $limit=Configuration::LIMIT],          // result limit
  [array $searchProperties=FALSE])            // custom properties
```

Listing 1.1. Method signature of API functions

As the project progressed, the need to query custom, application-specific properties arose. Thus, we provided a simple extension whereby unforeseeable properties from external vocabularies can be queried. Two changes were necessary for that, namely to provide a mechanism to create new prefix bindings and a slight amendment of the method signatures, i.e. the addition of an associative array (i.e. *searchProperties*) as a new parameter to specify custom properties to seek. For instance, a search criterion *foo:prop* denotes that all values of a property *prop* with a previously bound custom prefix *foo* are sought in the dataset. Similarly, the extension mechanism can be used more straightforwardly by providing full URIs for search properties instead of CURIEs peculiar to the Semantic Web.

3.4 Query Building Process

The input parameters of the API functions determine the types and constraints that eventually appear in the SPARQL query. Based on the type of function invoked and array parameters supplied the library populates a respective SPARQL template by joining graph pattern statements that apply (including triple patterns and clauses). This query building process doesn't require any further human involvement. It basically follows the natural order given by the structure of a SPARQL query [13], and thus comprises the following four stages:

1. *Projection.* First of all, the *SELECT*-part is built up. Possible values to select from consist of (a) general variables common to all functions (*uri* and *title*), (b) method-specific variables (e.g. *gln* in the context of *getStore*), and (c) values of proprietary properties (i.e. the extensibility mechanism addressed in Section 3.3). In the special context of nearby locations, an additional distance value to a reference point is computed and returned. In order to avoid expensive queries that fetch all possible select values no matter if they are going to be used by the application or not, developers can explicitly provide a list of properties to be sought.
2. *Graph pattern.* The structure of the graph pattern can be divided into three logical parts: (1) the *rdf:type* statement that defines the considered conceptual entity, (2) statements to constrain the result set according to input values supplied, and (3) an optional part composed of statements in *OPTIONAL* clauses to bind values to variables in the result form.
3. *Solution modifiers.* Besides the *DISTINCT* modifier in the result form of the SPARQL query the library also allows to limit the maximum number of results returned using the *LIMIT* solution modifier.
4. *Prefix bindings.* Finally, the algorithm prepends prefixes used in the SPARQL query. It also involves prefixes for custom vocabularies that have to be bound explicitly by the Web developer.

In addition to simple graph pattern matching, the library includes a few more complex functions that allow to manipulate query results. For instance, by using *maxPrice* it is possible to filter offers by a specified upper price limit (e.g. *list offers that cost less than 25 euros*), which is rewritten to a SPARQL query with a respective FILTER-expression. Currently, the library embodies the most relevant constraints for e-commerce that an average developer will need.

3.5 Demonstration

A typical example of using the PHP programming library for consuming e-commerce data from the Semantic Web can be implemented with only three lines of code as illustrated in Listing 1.2.

```
1 require_once('gr4php.php'); // include library
2 $connection = new GR4PHP(Configuration::ENDPOINT_URIBURNER);
3 $result_array = $connection->getStore(array("title"=>"Ravensburg"),
      array("title", "street", "city"), Configuration::MODE_LAX, 5);
```

Listing 1.2. Sample usage of the PHP library

The statement at line 1 includes the library. At line 2 the connection with the remote endpoint is set up, whereas at line 3 information about locations, constrained by the term *Ravensburg* appearing in textual properties, are fetched from the endpoint. In the given example, each of the five result sets will consist of three fields, namely *title*, *street* and *city*. Table 2 details the results as they are contained in the returned associative array.

Table 2. Query results

title	street	city
Kreissparkasse Ravensburg	Marienplatz 28	Ravensburg
Kreissparkasse Ravensburg	Meersburger Str. 1	Ravensburg
CineParC Ravensburg, Burgtheater	Marienplatz 4	Ravensburg
BODY STREET Ravensburg	Gartenstr. 25	Ravensburg
Bauernmarkt Ravensburg	Marktstr. 6	Ravensburg

The query related to code and results exemplified above is outlined in Listing 1.3 (with prefix declaration omitted). There are several patterns provided by different vocabularies to describe address details, e.g. one pattern from vCard 2001 and two additional patterns from vCard 2006; similarly, there exist numerous textual properties that are relevant for string-based matches.

```
1 SELECT DISTINCT ?title ?street ?city
2 WHERE {
3  {?uri rdfs:label ?title. FILTER(contains(?title, 'Ravensburg'))} UNION
4  {?uri gr:name ?title. FILTER(contains(?title, 'Ravensburg'))} UNION
5  {?uri gr:description ?title. FILTER(contains(?title, 'Ravensburg'))} UNION
6  {?uri rdfs:comment ?title. FILTER(contains(?title, 'Ravensburg'))} UNION
7  {?uri dc:title ?title. FILTER(contains(?title, 'Ravensburg'))}
8  {?uri a gr:Location} UNION {?uri a gr:LocationOfSalesOrServiceProvisioning}
9
```

```
 OPTIONAL {
  {?uri vc:ADR ?adr} UNION {?uri vc06:adr ?adr}
  OPTIONAL{{?adr vc06:street-address ?street} UNION {?adr vc:Street ?street}}
  OPTIONAL{{?adr vc06:locality ?city} UNION {?adr vc:City ?city}}
 }
}
LIMIT 5
```

Listing 1.3. SPARQL query for fetching stores that include the term *Ravensburg*

4 Evaluation

In order to validate the usefulness of our approach, it was necessary to compare the task of programming by means of the PHP API with the work involved in crafting corresponding SPARQL queries. For that purpose we set up an experiment where we selected two groups of developers who are aware of the GoodRelations vocabulary for e-commerce: The first group consisted of six volunteers that have advanced skills in PHP programming. We asked them to complete two programming assignments by help of our programming library, while having full access to the documentation that we provide on-line. Similarly, we asked two SPARQL-proficient developers to provide SPARQL queries with respect to the assigned tasks. Finally, we compared the average times spent by the two groups. The concrete assignments that participants were asked to complete were:

- *Task 1:* Query n=10 offers with textual properties (*gr:name*, *rdfs:label*, etc.) that contain the keyword "Bruce Springsteen" and cost less than 25 euros. Use an endpoint for linked open commerce (e.g. URIBurner) to complete this assignment and take the time.
- *Task 2:* Query n=5 business locations of the company "Containex". Again, take the time you spent for completing this assignment.

Table 3 shows the times spent for the two tasks by the eight participants. Group A represents the group that took advantage of the PHP library, group B the developers who formulated SPARQL queries instead.

Table 3. Times spent (mm:ss) for the completion of the assignments

	Group A							**Group B**		
	1	**2**	**3**	**4**	**5**	**6**	⌀	**1**	**2**	⌀
Task 1	3:15	3:30	8:00	2:30	6:00	16:00	6:32	11:30	15:00	13:15
Task 2	1:30	2:15	5:00	2:00	3:00	5:00	3:07	5:30	7:00	6:15

An independent-samples t-test was conducted to compare times in group A and in group B. There was a significant difference in the times spent for task 1 by group A (M=392.5 sec, SD=303.9 sec) and by group B (M=795 sec, SD=148.5 sec); p=0.0332. Similarly, there was a significant difference in the times spent for task 2 by group A (M=187.5 sec, SD=91.9 sec) and by group B (M=375 sec,

SD=63.6 sec); p=0.0297. These results suggest that the usage of our programming library had an effect on developer performance.

Our results suggest that when developers use our programming library they can expect higher productivity than candidates who try to achieve the same results by formulating respective SPARQL queries. Furthermore, we found out that manually crafted SPARQL queries are typically less complete in a sense of missing consideration regarding various schema patterns that frequently occur in real datasets.

Validation in practice. The first Web application that took advantage of the PHP library, *Ravensburg App*[7], further confirms the applicability of our approach. Ravensburg App is a mobile application for finding points of interest in Ravensburg city based on the GoodRelations vocabulary. The Ravensburg App developers feature their own, controlled RDF store with cleansed data from the municipality of Ravensburg city. The Ravensburg dataset is publicly available in the form of 700 individual RDF/XML documents[8]. During that project additional bug-fixes were made, the library was gradually enhanced and incoming feature requests were incorporated. Additionally, we provide some useful extensions, namely the extensibility mechanism and a means that allows to figure out whether a certain store is currently open or not.

5 Conclusions

In this paper, we presented a powerful software library for PHP developers to consume e-commerce data from the Semantic Web. Our library provides an efficient way of fetching useful information about stores, offers, product models, and opening hours from a public SPARQL endpoint. At the same time, it hides the complexity of SPARQL and schema patterns from the programmer. Programmers do not have to deal with details such as RDF modeling patterns, data cleansing heuristics, publication variants of RDF, or subtleties regarding the architecture of the SPARQL engine.

To validate our approach, we conducted an experiment. We compared the times to complete a programming assignment using our library with the times by formulating a SPARQL query. We showed that the amount of time for the implementation can be reduced significantly. The successful use in a real-world application further confirms the feasibility of our approach.

As future work we consider to repeat our preliminary experiment with a larger sample size to provide more evidence of the benefit of our approach. We believe that our PHP programming library is a proper tool to give rise to the implementation of integration of semantic data into Web pages based on e-commerce data on the Semantic Web, mobile application development considering contextual properties (e.g. geo positions), and novel product search engines.

[7] http://www.lieber-ravensburg.de/developer/
[8] http://www.wifo-ravensburg.de/rdf/sitemap.xml

Acknowledgments. Parts of the work presented in this paper have been supported by the German Federal Ministry of Research (BMBF) by a grant under the KMU Innovativ program as part of the Intelligent Match project (FKZ 01IS10022B).

References

1. Ben Abacha, A., Zweigenbaum, P.: Medical Question Answering: Translating Medical Questions into SPARQL Queries. In: Proceedings of the 2nd ACM SIGHIT International Health Informatics Symposium. pp. 41–50. Miami, Florida, USA (2012)
2. Berners-Lee, T., Hendler, J., Lassila, O.: The Semantic Web. Scientific American 284(5), 34–43 (2001)
3. Bizer, C., Heath, T., Berners-Lee, T.: Linked Data - The Story So Far. International Journal on Semantic Web and Information Systems 5(3), 1–22 (2009)
4. Campinas, S., Perry, T.E., Ceccarelli, D., Delbru, R., Tummarello, G.: Introducing RDF Graph Summary with Application to Assisted SPARQL Formulation. In: Proceedings of the 11th International Workshop on Web Semantics and Information Processing. (to appear), Vienna, Austria (2012)
5. Clark, L.: SPARQL Views: A Visual SPARQL Query Builder for Drupal. In: Proceedings of the ISWC 2010. Shanghai, China (2010)
6. Fielding, R.T.: Architectural Styles and the Design of Network-based Software Architectures. Ph.D. thesis, University of California, Irvine (2000)
7. Hakeem, A., Lee, M.W., Javed, O., Haering, N.: Semantic Video Search Using Natural Language Queries. In: Proceedings of the 17th International Conference on Multimedia. pp. 605–608. Beijing, China (2009)
8. Harris, S., Seaborne, A.: SPARQL 1.1 Query Language (2012), http://www.w3.org/TR/sparql11-query/
9. Hepp, M.: GoodRelations: An Ontology for Describing Products and Services Offers on the Web. In: Proceedings of the 16th International Conference on Knowledge Engineering. pp. 332–347. Acritezza, Italy (2008)
10. Kaufmann, E., Bernstein, A.: How Useful are Natural Language Interfaces to the Semantic Web for Casual End-users? In: The Semantic Web, 6th International Semantic Web Conference, 2nd Asian Semantic Web Conference. pp. 281–294. Busan, Korea (2007)
11. Kobayashi, N., Toyoda, T.: BioSPARQL: Ontology-based Smart Building of SPARQL Queries for Biological Linked Open Data. In: Proceedings of the 4th International Workshop on Semantic Web Applications and Tools for the Life Sciences. pp. 47–49. London, UK (2012)
12. Manola, F., Miller, E.: RDF Primer (2004), http://www.w3.org/TR/rdf-primer/
13. Prud'hommeaux, E., Seaborne, A.: SPARQL Query Language for RDF (2008), http://www.w3.org/TR/rdf-sparql-query/
14. Unger, C., Bühmann, L., Lehmann, J., Ngonga Ngomo, A.C., Gerber, D., Cimiano, P.: Template-based Question Answering over RDF Data. In: Proceedings of the 21st World Wide Web Conference. pp. 639–648. Lyon, France (2012)
15. Yahya, M., Berberich, K., Elbassuoni, S., Ramanath, M., Tresp, V., Weikum, G.: Deep Answers for Naturally Asked Questions on the Web of Data. In: Proceedings of the 21st World Wide Web Conference. pp. 445–449. Lyon, France (2012)