RAP: RDF API for PHP

Radoslaw Oldakowski, Christian Bizer, Daniel Westphal

Freie Universität Berlin, Institut für Produktion, Wirtschaftsinformatik und OR
Garystr. 21, D-14195 Berlin, Germany
cax@zedat.fu-berlin.de, bizer@wiwiss.fu-berlin.de,
westphal@wiwiss.fu-berlin.de

Abstract. RAP - RDF API for PHP is a Semantic Web toolkit for PHP developers. It offers features for parsing, manipulating, storing, querying, serving, and serializing RDF graphs. RAP was started as an open source project by the Freie Universität Berlin in 2002 and has been extended with code contributions from the Semantic Web community. Its latest release (V0.9.1) includes among others: a statement-centric API for manipulating RDF graphs as a set of statements, a resource-centric API for manipulating RDF graphs as a set of resources with properties, an up-to-date RDF/XML parser and serializer, support for other serialization techniques (i.e. N-Triple, Notation3, RDF embedded in XHTML), in-memory or database model storage, an inference engine supporting RDF-Schema reasoning and some OWL entailments, an integrated RDF server, and support for the RDQL query language.

1 Introduction

The next generation of web technology called *Semantic Web* aims at representing data in a more machine-understandable way, allowing enhanced information discovery, data exchange and integration as well as advanced automation of a variety of tasks. The core technology of the *Semantic Web* is the *Resource Description Framework (RDF)* which provides means for structural and semantic description of data [12].

The core idea of RDF is to represent information as sets of triples which can be exchanged over the Web. Every triple (RDF Statement) consists of the subject (resource being described), predicate (named property), and object (property value). Resources and predicates are represented by *URI*s – a global identification mechanism. A set of such triples is called an RDF graph. This abstract syntax of RDF is serialized using several concrete syntaxes like RDF/XML [3], N3 [4], or N-Triple[9].

2 What is RAP?

RAP - RDF API for PHP is a Semantic Web toolkit for PHP developers. It offers features for parsing, manipulating, storing, querying, serving, and serializing RDF graphs. RAP was started as an open source project by the Freie Universität Berlin in

2002 and has been extended with code contributions from the Semantic Web community.

The core of RAP are two implementations of statement storages which hold RDF graphs either in memory or in a relational database. Around these storages RAP provides rich programming interfaces for manipulating RDF graphs on different abstraction layers. Furthermore, RAP supports RDFS inference as well as some OWL entailments, allowing programmers to work with implicit (virtual) statements. Various tools complement the RAP package: an up-to-date RDF/XML parser and serializer, further I/O modules for alternative serialization techniques (i.e. N3, N-Triple, RDF embedded in XHTML), an integrated RDF server, and a graphical user-interface for managing database-backed RDF models as well as an implementation of the RDQL query language.

3 Working with RDF Graphs

In RAP, RDF graphs are represented as instances of class `Model`. The elements within a Model are `Statements`; each Statement comprises three `Nodes`: the subject, predicate and object. A Node represents a `Resource` identified by a URI, a `BlankNode` (also known as bNode or anonymous node), or a `Literal`.

RAP offers three programming interfaces for manipulating RDF graphs: the statement-centric *Model API* which allows manipulating an RDF graph as a set of statements, the resource-centric *ResModel API* for manipulating an RDF graph as a set of resources having properties, and the ontology-centric *OntModel API* which provides extra functionality for handling ontologies.

RAP supports the comparison and combination of multiple graphs. There are primitives in each of the above APIs checking whether or not two graphs are equal, or have common statements. Moreover, uniting two graphs, subtracting one graph from another one as well as creating an intersection of two graphs are also supported. There is no restriction regarding the storage mechanism of the graph, which means that one can for instance compare and combine an in-memory graph with a persistent graph.

3.1 Statement-centric Programming Interface

The *Model API* exposes an RDF graph as a set of RDF statements. This API is very similar to the statements storage structure and leads to a very small overhead in accessing the graph.

The core methods for modifying RDF graphs support adding, deleting, and replacing of single statements inside a graph. `StatementIterators` allow sequential access to all statements within a graph.

The most significant part of this API are the `find()`and `findAsIterator()` methods providing a fast and straightforward way to query RDF statements. The former method delivers a new model, the latter returns an iterator over all the statements of the queried model, which match the triple pattern (S, P, O). S/P/O can either be instances of the subclasses of `Node` or be equal NULL (meaning anything).

For example, the pattern (S, NULL, NULL) with S being an instance of `Resource` will match all statements describing this particular resource S.

There are also variations of the find primitive, which for instance, return only the first matching statement or merely return the number of statements matching the specified triple pattern. Another example of the simple query is the method `findRegex()` in which S/P/O of the triple pattern are specified using Perl-style regular expressions. More sophisticated queries than the simple pattern search can be expressed in the graph matching query language RDQL (see Section 7).

3.2 Resource-centric Programming Interface

The resource-centric API represents an RDF graph as a set of resources having properties. This interface enables to manipulate and navigate through an RDF graph in a much more comfortable way. For example, if a resource is known to be of type `rdf:Collection`, then viewing the corresponding resource as a collection allows easier access to it's members without having to deal with the sophisticated list-structure.

This *ResModel API* is implemented on top of the statement-centric interface. Thus, each `ResModel` always has an underlying in-memory or persistent statement store and is only providing a resource-centric view on this model. To ensure data consistency, there is no caching being done between the layers. Each method call is translated into a series of `find()`, `add()`, or `remove()`calls of the underlying model. Therefore, working with `ResModels` is slightly slower than using the *Model API* directly, but offers the comfort of accessing the information about resources in an object-orientated way. The *ResModel API* is very similar to the Jena Model API [7] allowing programmers, which are used to Jena, to readily write RAP code.

3.3 Ontology-centric Programming Interface

The ontology-centric API is an extension of the resource-centric interface. It adds support for ontological primitives: classes (in a class hierarchy), properties (in a property hierarchy) and individuals. The properties defined in the ontology language map to accessor methods. For example, if a resource is known to be an `rdfs:Class` in the given RDF graph it has a method to list its super-classes which correspond to the values of the `rdfs:subClassOf` property. This interface supports not only the RDF-Schema ontology language but also parts of OWL by using a loadable vocabulary. Thus, a new class is generated as an `rdfs:Class` or an `owl:Class` depending on the vocabulary currently loaded.

Likewise in the case of the resource-centric layer no information is stored in the `OntClass` object itself, but if any method of this class is called, the information needed is retrieved directly from the underlying RDF statements.

4 Inference

RAP includes a forward- and a backward-chaining RDFS reasoner and supports some OWL constructs. The reasoners are implemented as extensions of the in-memory triple storage by adding the ability to infer additional (implicit) statements. Thus, the inference process is totally hidden to the interfaces working with the in-memory storage and allows full access within all abstraction layers. This also enables to issue RDQL queries to an RDF graph containing inferred statements, thus allowing queries over entailments.

RAP's reasoners support following RDFS and OWL constructs: `rdfs:subclass`, `rdfs:subproperty`, `rdfs:range`, `rdfs:domain`, `owl:sameAs`, and `owl:inverseOf`. Programmers are given the choice of two different inference algorithms. Whereas the forward-chaining algorithm stores both its base graph and inferred triples in memory, the backward-chaining algorithm merely stores its base graph in memory and, in contrast to the former approach, creates inferred triples on the fly during query execution.

Up to our knowledge, RAP provides the only reasoning-engine implemented in PHP. Although its performance cannot compete with other reasoning engines implemented in Java or C, because PHP in general is comparatively slower, RAP's inference support works fine with medium sized graphs and relatively small RDFS schemata.

4.1 Forward-chaining Inference

The implementation of the forward-chaining inference strategy is represented by the class `InfModelF`. If a new statement from the RDFS or OWL namespace is added to the model, a corresponding `InfRule` is added to the model's rule-base. If a new statement, which matches the trigger of an `InfRule`, is added to the model then the entailment will be recursively computed until no more rules match the statement or statements inferred from it. The base statement and all inferred statements are added to the model.

Materializing all possible inferred statements means that adding base statements to the model is relatively slow. On the other hand find operations are very fast, because the `find()` method only has to look into the statement index and return the matching statements (including already materialized inferred statements). Therefore, this inference strategy is best used with models which do not change much and which are frequently queried.

4.2 Backward-chaining Inference

The implementation of the backward-chaining inference strategy is represented by the class `InfModelB`. In contrast to the former approach, in this case no inferences are done when a new statement is added to the model. Instead, when a query is executed against an RDF graph only the necessary inferences for this particular query are done on the fly: a find-pattern is executed with the supplied parameters first. Afterwards, it is checked against an inference-rules index, whether or not there are any rules that could entail statements matching the find-pattern. If there are such rules, the find-pattern is rewritten and a new search is performed. The new statements are inferred and added to the result set. This iterative process is repeated until there are no rules left that could produce matching statements.

`InfModelB` recognizes subclassing loops in ontologies. It also uses some shortcuts which speed up the search process: it recognizes branches in the ontology that do not lead to any additional statements and avoids running into these 'dead ends' for the second time. This inference strategy is optimal for large, changing models which are queried occasionally.

5 Input/Output

RAP offers basic primitives for reading and writing RDF graphs in a choice of Semantic Web languages. The languages supported are RDF/XML [3], N3 (also known as Notation3) [4], and N-Triple [9]. Furthermore, RAP's GRDDL parser allows extracting RDF-data from XHTML documents. GRDDL stands for "Gleaning Resource Description from Dialects of Languages" and uses XSLT to extract RDF-data from XHTML [10].

The RDF/XML-Serializer allows programmers to choose between different output modes. Users can decide, for example, whether triples should be grouped by subject, entities used for URIs, literal values always written as escaped CDATA, qualified names used for reserved RDF elements, or literal values of single properties serialized as XML attributes.

RAP also offers a convenient method for displaying an RDF model in a browser window in form of an HTML table containing a set of triples. In this table both asserted and inferred statements are presented and distinguished from each other.

6 Storing RDF Graphs

The core of RAP are two implementations of statement storages, which hold RDF graphs either in memory or in a persistent store. Working with in-memory models, however, has one major disadvantage: after finishing the execution of a PHP script, all models created and manipulated would be lost, unless saved to a file. But even if serialized to file, the document containing RDF data would have to be parsed any time a PHP script would be executed and additionally the search index built if efficient queries should be performed. Both processes are rather time-consuming, especially while working with large in-memory models.

To address this problem RDF API for PHP supports persistent storage of RDF models in a relational database. Storing models in a database not only saves main memory, but moreover allows quick access to RDF data by using the internal indexing and query optimization capabilities of the database.

6.1 In-memory Storage

The in-memory implementation stores an RDF graph as a collection of statements in an array in the system memory. To improve query performance RAP builds by default three search indices over statement's subject, predicate, and object. However, programmers can also customize the indices used, accordingly to their queries. They are given the choice to generate an index over the combined labels of subject, predicate, and object or just subject and predicate, or subject and object. As soon as new statements are added to or removed from the store the statement index is instantly updated in order to accelerate upcoming queries.

6.2 Persistent Storage

The core of RAP's database backend is built by two classes: `DbStore` and `DbModel`. The former is used to set the database connection as well as create, store, list, and retrieve RDF models, whereas the latter provides methods for manipulating each model.

To ensure wide portability RAP utilizes the ADOdb Database Abstraction Library for PHP [1] which supports a great variety of different databases (including MySQL, Oracle, DB2, MS SQL-Server, MS Access, ODBC) and allows connecting to multiple databases at the same time.

6.2.1 Denormalized Database Schema

Among various RDF APIs, there are different approaches to storing RDF data in a relational database stretching from plain schemas, where statements are stored in one table in a form resembling N-Triples, to highly normalized layouts consisting of several tables enabling efficient implementation of sophisticated RDF capabilities.

The goal for RAP's database backend was to provide an optimized for speed, portable solution. For this purpose RAP uses a denormalized database schema, where all resources and literals are stored in full in table *statements*. The performance of this solution was compared with a normalized layout having the main table *rdf_statements* which stored only numerical identifiers pointing to the corresponding entries in tables: *rdf_resources*, and *rdf_literals*. Benchmark tests have showed that the simple schema was 2-3 times faster than the normalized one and the advantage was rising with the increasing number of statements. Moreover, the trade-off between better performance and increased database size was acceptable, especially since RAP mostly targets manipulating medium-sized RDF models.

6.2.2 Manipulating Persistent Models

Once an instance of `DbModel` was created users can manipulate it, i.e. add, remove, or replace statements, serialize the model into different formats (plain text, RDF/XML, N3, N-Triples), save it to file or output in a browser window, compare and combine it with other `MemModel`s or `DbModel`s, reify, or query it. For this purpose RDF API for PHP allows using the same method names as in the case of manipulating an in-memory model. Although the internal implementation of these methods differs from those of the class `MemModel`, all the differences are hidden for the user. Hence, one can expect the same results while manipulating RDF models regardless of the storage mechanism chosen.

Note that RDF statements of a persistent model are not loaded into memory while a `DbModel` is created. Most of the methods of this class are able to query and manipulate the triples directly in the database. Delegating some tasks to the database management system saves server resources and reduces the CPU usage. Only few functions (e.g. `saveAs()`) require all the statements of a model to be loaded into memory for further processing.

6.2.3 RDF DB Utils

In the RAP toolkit there is also RDF DB Utils included - a graphical user-interface for managing database-backed RDF models. It allows convenient browsing through a selected persistent model to view, edit, or delete statements. Querying a model is realized by either providing a simple SPO-pattern or by using the more expressive RDQL query language.

7 RDQL

RDQL (RDF Data Query Language) [17] is a query language for extracting information from RDF graphs. Queries are formulated by specifying a subgraph, with missing parts having assigned variable names, which is matched against an RDF graph. RDQL is implemented in several RDF toolkits and has been submitted to the W3C for standardisation [17]. In order to ensure the greatest possible compatibility RAP's RDQL implementation follows the current de facto standard set by the Jena [7] implementation.

7.1 RDQL Syntax

An RDQL query consists of a graph pattern, expressed as a list of triple patterns (S, P, O). S/P/O can either be named variables or RDF values (URIs or Literals). Literals may additionally be constrained by their language and datatype. Furthermore, an RDQL query can have a set of constraints on the values of query variables. Filter expressions supported by RAP are: arithmetic conditions (including multiplicative and additive operators), string equality expressions, and Perl-style regular expressions. Multiple constraints can be combined using logical operators. A list of variables required in the answer set is specified in the SELECT clause of an RDQL query. To make the query easier to read and write for humans, RDQL provides a way to shorten the length of URIs by defining a string prefix.

```
SELECT ?person
WHERE (?person, <info:age>, ?age)
AND ?age >= 26
USING info FOR <http://example.org/people#>
```

The above triple pattern matches all statements having predicate `http://example.org/people#age`. The variable `?person` will be bound to the label of the statement subject, the variable `?age` to the literal value of the statement object. The query returns all values of `?person` from statements matching the specified pattern and having the object value greater or equal 26.

7.2 Query Execution

The execution of an RDQL query in RAP consists of four main steps. At first, the query string is parsed into an in-memory representation and subsequently passed to the corresponding engine (`RdqlMemEngine` for in-memory models and `RdqlDbEngine` for persistent models). In the second step, the query engine searches for triples matching all patterns from the `WHERE` clause of the RDQL query and returns a set of variable bindings. Next, this result set is filtered by evaluating Boolean expressions specified in the `AND` clause. The last step is the processing of the final result, i.e. creating of objects (`Resource`, `BlankNode`, `Literal`) or their plain text serialization as values of the query variables, according to the desired format. RAP users are given the opportunity to decide whether the query should deliver an RDF graph instead of variable bindings.

If an RDQL query is performed on a persistent model, the searching for tuples matching all patterns is delegated to the database. Moreover, instead of calling the database for the result set matching each single pattern and then joining them in memory, the `RdqlDbEngine` translates almost the entire RDQL query[1] into an SQL statement and sends it to the database for execution. This approach enables RAP to use the database indexing and optimization capabilities.

8 NetAPI

The RAP NetAPI is an RDF server for publishing RDF models on the web making them accessible to remote clients and applications. It implements a subset of the W3C member submission RDF NetAPI [13], providing similar functionality as the Joseki RDF server [11]. The advantage of using the RAP NetAPI is that it can be run on all web servers supporting PHP (currently 15 million domains).

[1] Except for the AND clasue, because some kinds of expressions (e.g. regular expressions, arithmetical operations) are either not standardized or not supported by all databases. Therefore, the filtering of the result set of the database query is carried out in memory.

Models published on the web have URLs. They can be accessed by either simple find(SPO) queries or by RDQL queries, using HTTP GET. RAP NetAPI also offers operations for adding and removing subgraphs from a remote RDF graph (using HTTP POST), thus allowing development of collaborative applications based on shared models.

9 Related Work

A detailed comparison of the features of Semantic Web toolkits for different programming languages can be found in [5]. For PHP there are three other RDF toolkits: the 'ARC - appmosphere RDF Classes' [2] which provide less functionality than RAP but have a better performance on core tasks like parsing RDF/XML to an in-memory array, the PEAR:RDF 0.1.0 [15] packages which are a port of RAP Version 0.8.1 to the PEAR repository according to the PEAR coding style, and PHPXML Classes 1.1 [16] which also include a basic RDF/XML parser and an RDQL implementation but have not been updated since 2002.

10 Conclusion

RAP offers PHP developers wide range of functionalities for parsing, manipulating, storing, querying, and serializing RDF data. Whereas the *Model API* with its core functionality is optimized for speed, the *ResModel API* with its support for sophisticated RDF constructs like containers and collections as well as the *OntModel API* with its support for ontology-specific concepts allow much more convenient manipulation of RDF data. Furthermore, by giving access to RDFS inference capabilities as well as some OWL entailments, RAP offers developers a toolkit for writing applications working with implicit (virtual) statements. RDF can be stored in a variety of databases using a flexible abstraction layer. The RDQL implementation enables developers to use the expressive power of this query language to find the desired information. Support for various serialization techniques allows flexible exchange of RDF models with other applications storing and processing RDF data. Moreover, the NetAPI provides an RDF server for accessing and querying RDF models in a distributed environment and thus enabling collaborative applications based on shared models. By being implemented in PHP and owing to its portability to various databases RAP can be used as foundation for deploying Semantic Web applications on cheap commercial web space.

Current development activities are concentrated on extending RAP for Named Graphs [8] by porting NG4J [6] to PHP, as well as on implementing the SPARQL [14] query language. Another feature waiting to be implemented in RAP is the support for signing RDF models on syntax or model level using XML-Signature or the Semantic Web Publishing (SWP) vocabulary [8].

RAP is available under GNU Lesser General Public License (LGPL) from http://www.wiwiss.fu-berlin.de/suhl/bizer/rdfapi/. The package has been downloaded over 2.100 times since September 2002.

11 References

[1] *ADOdb Database Abstraction Library for PHP*. http://adodb.sourceforge.net/

[2] *ARC - appmosphere RDF classes*. http://www.appmosphere.com/pages/en-arc

[3] D. Beckett: *RDF/XML Syntax Specification (Revised).* W3C Recommendation. 10 February 2004. http://www.w3.org/TR/rdf-syntax-grammar/

[4] T. Berners-Lee: *Primer: Getting into RDF & Semantic Web using N3*. World Wide Web Consortium, 2000. http://www.w3.org/2000/10/swap/Primer.html

[5] C. Bizer, D. Westphal: *Developers Guide to Semantic Web Toolkits for different Programming Languages*. 15 February 2005. http://www.wiwiss.fu-berlin.de/suhl/bizer/toolkits/

[6] C. Bizer, R. Cyganiak, R. Watkings: *NG4J - Named Graphs API for Jena V0.4*. 2nd European Semantic Web Conference (ESWC 2005), Heraklion, Greece. May 2005. http://www.wiwiss.fu-berlin.de/suhl/bizer/ng4j/

[7] J. Carroll, et. al: *Jena: Implementing the Semantic Web Recomandations.* Bristol. 2003. http://www.hpl.hp.com/techreports/2003/HPL-2003-146.pdf

[8] J. Carroll, C. Bizer, P. Hayes, P. Stickler: *Named Graphs, Provenance and Trust*, at the 14th International World Wide Web Conference (WWW2005), Chiba, Japan. May 2005. http://www.wiwiss.fu-berlin.de/suhl/bizer/pub/Carroll_etall-WWW2005.pdf

[9] J. Grant, D. Beckett: *RDF Test Cases*. W3C. 10 February 2004.

[10] D. Hazaël-Massieux, D. Connolly: *Gleaning Resource Descriptions from Dialects of Languages (GRDDL).* W3C Coordination Group Note. 13 April 2004. http://www.w3.org/TR/grddl/

[11] *Joseki – Jena RDF Server*. http://www.joseki.org/

[12] G. Klyne, J. Carroll: *Resource Description Framework (RDF): Concepts and Abstract Syntax*, W3C Recommendation. 10 February 2004. http://www.w3.org/TR/rdf-concepts/

[13] G. Moore, A. Seaborne: *RDF Net API, W3C Member Submission.* 2. October 2003. http://www.w3.org/Submission/rdf-netapi

[14] E. Prud'hommeaux, A. Seaborne: *SPARQL Query Language for RDF*, W3C Working Draft. 19 April 2005. http://www.w3.org/TR/rdf-sparql-query/

[15] *PEAR:RDF*. http://pear.php.net/package/RDF

[16] *PHP XML Classes*. http://phpxmlclasses.sourceforge.net/

[17] A. Seaborne: RDQL - *A Query Language for RDF*, W3C Member Submission, 9 January 2004. http://www.w3.org/Submission/RDQL/