RESTful INTERFACE TO ONTOLOGIES

by

CHINMAY KALE

(Under the Direction of Krzysztof J. Kochut)

ABSTRACT

INDEX WORDS: RESTful Web Services, Ontologies, SPARQL Endpoints, Ontology Servers, And Semantic Web

RESTful INTERFACE TO ONTOLOGY

by

CHINMAY KALE

B.E. Computer Engineering, University of Pune, India, 2006

A Thesis Submitted to the Graduate Faculty of The University of Georgia in Partial Fulfillment of the Requirements for the Degree

MASTER OF SCIENCE

ATHENS, GEORGIA

2011

© 2011

CHINMAY KALE

All Rights Reserved

RESTful INTERFACE TO ONTOLOGY

by

CHINMAY KALE

Major Professor: Krzysztof J. Kochut

Committee: John A. Miller

William York

Electronic Version Approved:

Maureen Grasso

Dean of the Graduate School

The University of Georgia

DEDICATION

To my parents and loved ones.

ACKNOWLEDGEMENTS

Again, this page is optional. You do not have to provide an acknowledgements section in your thesis or dissertation. You may use this section to express acknowledgements of those who have helped you with this document and your academic career.

TABLE OF CONTENTS

Page

ACKNOWLEDGEMENTS #

LIST OF TABLES #

LIST OF FIGURES #

CHAPTER

1 INTRODUCTION 1

1.1 Semantic Web 1

1.2 Ontologies 1

1.3 REST Web Services 2

2 RELATAED WORK 5

3 SYSTEM DESIGN 6

3.1 Service Layer 7

3.2 Logic Layer 9

3.3 Ontology Layer 9

4 IMPLEMENTATION 10

4.1 Ontology Service 11

4.2 Schema Service 11

4.3 Navigation Service 32

4.4 SPARQL Query Service 35

5 #

#

#

#

#

6 #

#

#

7 #

REFERENCES #

APPENDICES

A #

B #

C #

LIST OF TABLES

Page

Table 1: #

Table 2: #

Table 3: #

Table 4: #

Table x: #

LIST OF FIGURES

Page

Figure 1: #

Figure 2: #

Figure 3: #

Figure 4: #

Figure x: #

CHAPTER 1

INTRODUCTION

* 1. Semantic Web

In the World Wide Web (www) a web page can be accessed by its Uniform Resource Locator (URL) through the hypertext transfer protocol (HTTP). Most of the resources on the www are written in HTML, which conveys their rendering information to the web browsers. Therefore most of the information on the www is for human consumption. Machines for automatic information processing and integration cannot use the information present in the web pages. Semantic Web aims at representing information on the web in a way that the computers can understand the meaning of the information. This is accomplished by embedding machine-readable information in the existing web pages. The machine-readable syntax makes the content easy to process the information while making it more amenable to exchange between heterogeneous applications. The semantic web can be thought of as a huge graph where resources are connected to other resources through meaningful edges.

* 1. Ontologies

There can be different ways in which semantics can be added to information. Ranked from the weakest formalisms to the strongest they are as follows:

* Controlled Vocabularies

Controlled vocabularies are a limited set of enumerated terms, which are agreed upon based on the particular use case. Only the terms from the enumerated set can be used to add metadata.

* Taxonomies

Taxonomy is a controlled vocabulary with relations like “subclass of” and “superclass of” between the enumerated terms.

* Thesaurus

A Thesaurus adds to taxonomy by giving the ability to state if two terms are equivalent, homographic or associative [NISO, 2005]

* Ontologies

An ontology is a formal specification of a shared conceptualization [reference here ].Ontologies represent shared domain-specific [[1](#_ENREF_1)]knowledge, which can be shared between machines and people. Ontologies can be expressed in increasingly expressive languages like: RDF-Schema [[1](#_ENREF_1)], Web Ontology Language (OWL) [Reference here].

* 1. REST Web Services

The popularity for Web services is growing with the demand for the interoperability between software applications across frameworks and platforms. W3C defines Web service as “a software system designed to support interoperable machine-to-machine interaction over a network. It has an interface described in a machine-processable format. Other systems interact with the Web service in a manner prescribed by its description”. REST[[2](#_ENREF_2)] is a Web service design model that focuses on system’s resources. Each resource is represented by a URI (Uniform Resource Identifier), which enables uniform identification of resources. It follows a defined set of naming scheme.

The design principles followed by an implementation of a REST Web service are: Stateless, use of HTTP methods, use of URIs and XML formats. It is called a stateless operation as the request contains all the information needed to generate a response independent of the session. Also, a REST Web service always uses one of the HTTP methods: POST, GET, PUT or DELETE to add, remove or get a resource. Each resource is represented by a URI in a REST Web service. This enables use of hyperlinks to browse resources. And the data format typically used is XML. XML is used to format request and response messages.

REST Web services have quite a few advantages because of which REST design model is gaining popularity gradually and is being adopted more for major Web service implementations. Some of the benefits of using REST Web services are:

* They are lightweight.
* The results are human readable.
* They are easy to implement.

Another widely adopted Web service design is based on SOAP. SOAP stands for Simple Object Access Protocol. It is a XML based protocol that enables the exchange of information amongst software applications over HTTP. Depending on the application at times SOAP is preferred over REST for the following reasons:

* It is mostly easier to consume a SOAP Web service.
* The type checking is much stricter in SOAP.
* Because of wide Enterprise level adoption there are a lot of development tools available to enable some of the more complicated tasks in implementation of a SOAP Web service.

Overall, the REST design pattern provides the flexibility and scalability that is integral to this project. We will go into further details on how RESTful Web services are leveraged to provide an interface to or download ontology as an owl file, to browse the ontology schema and to update the classes and properties.

CHAPTER 2

BACKGROUND

2.1 Ontology Servers

An ontology server is a tool that supports distributed, collaborative editing and browsing of ontologies. Ontology servers can be classified into two groups based on the functionalities that they offer: (a) tool development and (b) Application Programming Interfaces (APIs). During the infancy of Semantic Web, most of the ontology servers developed were with the primary focus of ontology development. As more and more ontologies were created and their sizes increased, ontology servers providing application programming interface for interacting with the ontology started surfacing in the research community. The application-programming interface aids development of any kind of application on top of the ontology repository such as ontology browser, ontology editor, ontology translator etc.

So ontology servers are ontology repositories that provide services to interact with ontologies. These repositories are geared to storing and returning RDF triples in response to queries. Such repositories are also called as triple stores. Triple stores can be classified into two types depending upon the way they store RDF triples (a) in-memory triple stores (b) persistent triple store. Persistent ontology repositories usually store the RDF data in Relational Database Management Systems. Both of these approaches have their advantages and disadvantages. In-memory triple store have space limitations and cannot be used for storing huge amounts of data. On the other hand in-memory triple stores have efficient reasoners available. There are many free open-source as well as commercial triple stores available. Following is a brief overview of few

* Jena

Jena[[3](#_ENREF_3)] is a free open source Java platform for building Semantic Web applications. It provides both in-memory as well as persistent triple store storage. It uses JDBC for connecting to persistent triple stores. Jena also provides reasoning capabilities. For better performance, Jena requires data to be present in-memory for reasoning. Jena framework also provides a SPARQL query engine.

* Sesame

Sesame[[4](#_ENREF_4)] is a free open source framework for storage, inferencing and querying RDF data. It provides similar features like Jena. Sesame’s focus is on the RDF data storage and query, but without much support for OWL and related inferencing tools.

* Redland

Redland[[5](#_ENREF_5)] is a set of free C language libraries that provide support for RDF.

It provides a RDF parser library called Raptor for parsing RDF/XML or N-triples and storing them in RDF triples. Although Redland does not provide a strong support for reasoning and inferencing, it does work with C language. When speed is a major concern, Redland framework can be the choice.

* Virtuoso

Virtuoso Universal Server, Virtuoso is a database engine that combines the functionality of traditional RDBMS, ORDBMS, RDF, XML, free-text, Web application server, and file server into a single server product package.

2.2 SPARQL endpoint

SPARQL is an RDF query language and data access proto- col for the Semantic Web. Its name is a recursive acronym that stands for *SPARQL Protocol and RDF Query Language.* The W3C Recommendation of SPARQL consists of three separate specifications. The first one *SPARQL Query Language specification* is the core specification of SPARQL query language*.* Together with this language specification is the *SPARQL Query XML Results Format specification*, which describes an XML format for serializing the results of a SPARQL query. The third specification is he *SPARQL Protocol for RDF (SPROT) specification* that uses WSDL 2.0 to define simple HTTP and SOAP protocols for remotely querying RDF databases.

A SPARQL endpoint is a SPROT conformant interface. It provides a service for client applications to query knowledge bases using the SPARQL query language. After execution of the SPARQL query, the results are transmitted to client applications. A SPARQL endpoint can be configured to return results in a number of different formats. For instance, when used by human users in an interactive way, it presents the result in the form of a HTML table. When accessed by applications, the results are serialized into machine-process able formats, such as RDF/XML or Turtle format etc. SPARQL endpoints can be categorized as genericendpoints and specificend- points. A generic endpoint works against any RDF dataset, which could be stored locally or accessible from the Web. A specific endpoint is tied to one particular dataset, and this dataset cannot be switched to another endpoint.

SPARQL protocol[[6](#_ENREF_6)] uses WSDL 2.0 to define simple HTTP and SOAP bindings for remotely querying RDF data. Client applications use SPARQL protocol to interact with SPARQL endpoints.

2.3 Introduction to REST

Representation State Transfer (REST) was introduced and described by Roy Fielding in his doctoral dissertation. As part of his doctoral studies, Roy Fielding generalized the Web’s architectural principles and presented them as a framework of constraints or an architectural style. Through this framework, Fielding described how distributed information systems such as the Web are built and operated. He described the interplay between resources, and the role of unique identifiers in such systems. He also talked about using a limited set of operations with uniform semantics to build a ubiquitous infrastructure that can support any type of application. Fielding referred to this architectural style as *REpresentational State Transfer,* or REST.

REST is a set of constraints that inform the design of hypermedia system. REST claims that following these constraints will result in architecture that is scalable, mash up able, usable and accessible. The following constraints are the core of REST architectural style.

* Resource Identification:

All resources of a system should have a unique identifier and the resources should be addressable using this identifier. To have addressability, the identifiers should be global and should be dereference able irrespective have their context.

* Unique Interface:

This constraint states that all the interactions between the system's resources and the client applications should be carried out through a uniform constrained interface. This interface should expose a small set of well-defined methods to manipulate the resources.

* Self-Describing Messages:

This constraint builds upon the second constraint. As the second constraint states that all interactions with resources should be exposed through a uniform interface, REST architecture demands the resources should have representations that represent the important aspects of the resource. These representations have to be designed in such a way, that any client applications can get the relevant state of the resource by inspecting their representations. Also, by exchanging these representations via the uniform interface, any changes to the resource or its state should be communicated.

* Hypermedia Driving application state:

This constraint states that the representations, described in third constraint, should be linked, so that the applications that have the capability to understand these representations will be able to find these links. As the semantics of these links are described by the representations, these applications will also be able to understand them. These links help these applications in identifying new resources and also they allow them with the possibility of making certain state transitions. In short this constraint states to use Hypermedia As The Engine Of Application State (HATEOAS). According to [[7](#_ENREF_7)], this constraint is the most important reason for supporting lose coupling, as identifiers can be discovered at runtime and used through the uniform interface without the need of any agreements between the interacting parties.

* Stateless Interaction:

This constraint states that every interaction between the client and server should be self-contained and isolated. The server should not maintain any state of the client, which would allow interactions to depend upon both the exchanged representation and on the session associated with the client. This constraint is necessary to ensure the scalability of the servers is bound only by the number of concurrent client requests and not by the total number of clients that they have to interact with.

If any system is designed and implemented using these constraints, such systems are called RESTful applications. In the System Design chapter of this thesis, we will show how we have incorporated all these constraints in your ontology server.

REST is protocol agnostic. But due to HTTP’s ubiquitous nature, most of the systems adhering to REST principles use HTTP protocol as transport layer. The idea behind REST principle of uniform interface is to stick to the finite set of operations of the application protocol that your system uses to distribute your system’s services. This means utilizing the HTTP methods for exposing the services offered by the system. HTTP specification lists eight methods, out of which four are important to design RESTful services. They are GET, POST, PUT and DELETE.

* The GET request method offers read-only access to resources. It is used to query the server for specific information. It is idempotent and safe operation. GET method does not change the state of the resource.
* The POST request method offers a way to send data to the server. It is a non-idempotent operation. It is usually modeled to create or modify a resource.
* The PUT request method also offers a way to send data to the server. But it differs from the POST method as its idempotent. It is usually modeled to update the state of the resource.
* The DELETE request method offers a way to remove resources. It is idempotent as well.

Application systems provide RESTful web-services by having unique identifiers for the resources they want to expose and support these four HTTP methods to perform operations on the resources.

CHAPTER 3

MOTIVATION

The current de facto global information system World Wide Web (WWW) is a web of linked documents. The vision of Semantic Web is to transform WWW from web of linked documents to web of linked data. Maturing Semantic Web technology stack fuels the increasing interest in publishing semantically linked data. Within recent years we have witnessed huge ontologies like Dbpedia[[8](#_ENREF_8)], YAGO[[9](#_ENREF_9)], Good Relations being published. On the other hand many domain specific ontologies like GlycO, Mouse, ProkinO are also being published. The applications that interoperate among various such domains seek for an alignment among the ontologies from these domains. As a common result a unified ontology is created. Though the individual size of these domain ontologies not very large in size, the resultant unified ontology tends to be enormously large and complex. Consequently the number of huge and complex ontologies and applications based on them is rising.

Ontology is a directed graph and its topology can become very complex especially for larger ontologies. This makes it very difficult to comprehend or render them. Ontology administrators face difficulties in managing and maintaining large ontologies. Similarly ontology consumer applications like editors, browsers; visualizers have a hard time processing such large ontologies. Ontology navigation can be of create help to solve these problems. Navigating ontology to a point of interest can provide the ontology consumer application a zoomed-in view of the point of interest. Using navigation techniques the applications using large ontologies can focus on a small sub-graph from the ontology, which is of their interest.

In last few years, ontologies have moved from theory to practice to real world applications. Ontologies are now not limited to academia but they are finding their way into enterprise applications. Most common operations required in any enterprise application are the CRUD operations. Also these CRUD operations are handy if they are exposed through web-services. And if the web-services are of type REST, then any client application that has capability of sending and receiving HTTP request and response respectively can consume the web-services offering CRUD operations.

REST architecture is not protocol specific, but uses HTTP protocol as its transport layer. The HTTP protocol is the de facto for Web of linked documents. Web of linked documents is very much similar to ontology in terms of its topology. Both are directed sub-graphs with nodes connected by directed edges. The documents or resources in Web of linked data can be seen analogous to concepts (classes, properties, instances) in ontology. As discussed in the background chapter in REST architecture all resources of a system are exposed via a uniform interface or the URI and CRUD operations are performed on these resources using HTTP operations. Each document in Web of linked document has a unique addressable URI and so do concepts in ontology. Hence we can conclude that REST web-services are a natural fit to expose CRUD operations on the concepts in ontology.

SPARQL protocol[[6](#_ENREF_6)] uses WSDL 2.0 to define simple HTTP and SOAP bindings for remotely querying RDF data. There are two HTTP bindings defined in the SPARQL protocol specification - *queryHttpGet* and *queryHttpPost.* The specification instructs to use queryHttpGet except in cases where the URL encoded query exceeds practical limits. In such cases the specifications says queryHttpPost should be used. The current SPARQL specification supports SELECT, CONSTRUCT, DESCRIBE and ASK queries. Both SELECT and CONSTRUCT queries are read-only. The SELECT query after successful execution creates a new temporary RDF graph, called the result-set, that contains all or a subset of the variables bound in the query pattern match, whereas the CONSTRUCT query creates a new RDF graph by substituting variables in a set of triple templates. So both of these query constructs are creating a resource. Clearly using queryHttpGet binding for these queries violates REST principles.We would discuss in this thesis our approach to make SPARQL query execution RESTful.

CHAPTER 2

RELATED WORK

Many previous attempts have been made to develop a ontology server as stated in the survey [Reference]. Most of the servers mentioned in the survey are developed with the purpose of ontology development. They provide a collaborative platform for various domain knowledge experts to collaborate and create ontologies. Servers like Ontolingua Server[[10](#_ENREF_10)], Ontosaurus[[11](#_ENREF_11)] and OntoRama[[12](#_ENREF_12)] provide implicitly or explicitly isolated tool for ontology building[[13](#_ENREF_13)]. But such ontology servers are of little use to real world applications, as these servers would be primarily use by ontologists and not real world applications.

There are other ontology servers which allow only browsing of ontologies. The ontologies loaded in the server are read-only and cannot be edited. One of such server is Ontology-Browser; a project that grew out of OWL-Doc project provides a ontology server for browsing ontologies. The user can navigate around OWL ontologies and Linked Open Data online. It supports features like loading any content on the fly into the server, execution of SPARQL queries. But as stated above it does not allow editing or addition of any entity of the ontology.

SPARQL is the standard query language for RDF data. Queries can be formulated, are submitted to a single processing facility, which then returns a result set. The SPARQL protocol defines means of conveying SPARQL queries from query clients to query processors. Most of the current implementation of the SPARQL protocol is not RESTful in the sense of the Resource-Oriented Architecture (ROA) [[14](#_ENREF_14)].

CHAPTER 3

SYSTEM DESIGN

Before describing the web-services supported by our ontology server, it is important to understand the design of the system. This chapter describes the design and approach that we have come up with for developing the ontology server.

The ontology server is a J2EE specification compliant web-server. The system has multi-tier approach like a normal J2EE web application. Following diagram displays the overall system architecture.

Ontologies

Logic Layer

(JENA/ARQ)

Service Layer

(RESTEasy)

3.1 Service Layer

The first layer is the Service layer. The ontology server provides four types of services namely, ontology management service, services for CRUD operations on ontology, services for execution of navigational queries and services for execution of SPARQL queries. Each of these services is designed using RESTful Architectural Principles. Following is a detailed design overview for each service.

3.1.1 Ontology Management Service

This module provides services to manage the ontologies that are deployed in the server. The client application can invoke a service from this module, if they want to load an ontology that is currently not loaded in the server. This module also provides a service with which the client application can take a snapshot of any ontology that is currently loaded. This feature is provided because the ontology server provides services to execute CRUD operations on the loaded ontologies, so at any given point the client application can download a modified ontology as a OWL file from the server.

Also this module can load some ontology during server startup. These are the default ontologies that the server administrators can configure during server startup.

3.1.2 CRUD Operation Services.

This module is designed to provide a RESTful interface for performing CRUD operations on ontologies. In ontology the classes, properties and instances are the resources of interest. We further divided these resources into sub-resources like super-classes, sub-classes, super-properties and sub-properties. This module provides methods to perform CRUD operations on these resource and sub-resources of ontology. Each of these resources are uniquely identified by a URI. The CRUD operations are mapped to four HTTP operations namely POST, GET, PUT and DELETE. The combination of one of the HTTP operation and a URI invokes a service method from this module.

3.1.3 Navigational Service

The ontology is a directed graph with classes and instances as the nodes and properties as the edges. This module provides an interface to navigate within this directed graph along the edges.

The navigation through ontology can be seen as path query.

3.1.4 SPARQL Query Service

This module provides an interface to execute SPARQL queries. If a client application wants to execute a SPARQL query, it has to send two requests to the server. In the first request the client application sends the query that is to be executed on the ontology mentioned in the URL of the request. After executing the query, the result set is not immediately returned to the client application. Instead the result set is cached at the server for a specific amount of time. A unique identifier identifying the result set and the time for which the result set will be cached is returned to the client application in form of response. To obtain the result set the client application has to send another request within the time mentioned in response of the first request. With the second request the client application also have to send the unique identifier for the result set and the format in which it wants the result set to the server. This module then accordingly processes the second request from the client and returns the result set. We designed this two fold request-response sequence to incorporate the RESTful architectural principles.

3.2 Logic Layer

The Logic Layer comprises of methods that contain the logic to perform the task, the client application’s requested for. This layer implements all the interfaces that are defined by the service layer. Also this layer is an interface between the service layer and the ontologies. This layer performs all the CRUD operations exposed by the service layer.

3.3 Ontology Layer

The Ontology Layer is where the ontologies reside. The ontologies can be in a persistent storage system or could be in-memory.

CHAPTER 4

**IMPLEMENTATION**

This chapter describes the implementation of the ontology server. The ontology server is a J2EE web server with the capability of processing RESTful web services. As mentioned in the System Design chapter the ontology server provides four types of services. They are implemented as Java interfaces and are called OntologyService, SchemaService, NavigationService and SPARQLService. Each service provides a RESTful API to interact with the ontologies loaded in the web server. The service layer is implemented using JBOSS’s RESTEasy, which is a framework for developing RESTful Java web services. The logic layer is implemented using Jena Semantic Web toolkit.

4.1 Implementation of Service Layer: RESTEasy

RESTEasy is open source software distributed under Apache Software License 2.0. RESTEasy is a full certified and portable implementation of the JAX-RS specification. JAX-RS is the Java Community Process specification released in 2008. It provides a Java API for RESTful web services over the HTTP protocol.

Every Java service interface is mapped to a unique URI. For example a URI /schemaService represents the SchemaService Java interface. This mapping is achieved using JAX-RS annotations defined by RESTEasy.

All the Java services support four HTTP operations GET, POST, PUT, DELETE. A combination of any one of these HTTP operations and a URI uniquely identifies a Java service method from the Java service interfaces. So for every URI, all of the four HTTP operations are supported.

To keep the marshaling and un-marshaling of request/response data decoupled from the Java objects, the service layer uses message body readers and writers. These message body readers parse the request body to extract the information sent by the client. It also validates the request and check if it adheres to the format expected by the API. If it is not, exception is returned to the client along with appropriate HTTP status code. The message writer on the other hand wraps the result of the Logic Layer into a format that the client can accept (as mentioned in the Accept Header field of the HTTP request). Once the request has been parsed and validated the service layer transfers the control to the logic layer.

Following are the details of each of the service.

4.1OntologyService

This service provides APIs to load any ontology or take a snapshot of any loaded ontology. By taking a snapshot here I mean is that the user can download from the web server ontology as a OWL file.

4.2 SchemaService

This service provides CRUD operations for the ontology schema as well as the instances in the ontology. Each method in the service is bound to one unique URL and HTTP operation using the RESTEasy annotations. Also request parameters are injected into the method parameters using the annotations. To understand see lets see an example.

Consider the URL schemaService/{ontologyName}/allClasses.

Here the value of the ontologyName is variable and is injected with the value that the client application provides in form of the URL. So when the client application sends a request with the URL schemaService/pizza/allClasses, the string pizza is injected into the variable ontologyName. Thus the client application states which ontology they want to use.

Following methods are supported by the schema service

1. Classes

This service provides an interface to browse, add, update or delete any class from the requested ontology. The client application provides the name of the ontology they want to query in form of the URL (as explained above). Also the client application passes a list of comma-delimited names of classes.

**GET**

This request returns information about each class passed in the URL. For each, list of its super classes, sub-classes, properties and instances is returned in xml format (defined below) as response body.

Request:

*Resource*

schemaService/{ontologyName}/classes/<comma separated names of classes>

Response:

*Content Type:*

application/xml

*Body*:

For each class mentioned in the request url a Class tag is returned. All the classes are enclosed in Classes tag, as shown below.

<Classes>

{<Class name="Class1" uri= <http://../..#Class1>” />} m

<SuperClasses>

{{<SuperClass name=”SuperClass1” uri=”<http://../..#SuperClass1>” />} n

</SuperClasses>

<SubClasses>

{<SuperClass name=”SuperClass1” uri=”<http://../..#SuperClass1>” />} n

</SubClasses>

<Properties>

{<Property name=”Prop1” uri=”http://../..#Prop1” />}n

</Properties>

<Instances>

{<Instance name=”Inst1” uri=”http://../..#Instance1” />}n

</Instances>

</Class>

</Classes>

m: Occurs for every class passed in the request URL.

n: Can occur n number of times.

Errors:

* Class does not exist -> 404 “ no such class exist”
* Ontology is not loaded -> 404 “ requested ontology is not loaded”
* Any other delimiter used instead of comma -> 400

**POST**

This request creates classes in the ontology mentioned in the request URL. Information for each class that has to be created is provided as the request body in xml format as described below.

Request:

*Resource*

{ontologyName}/classes/<comma separated classes>

*Content Type:*

application/xml

*Body:*

The request body should be in following format.

<Classes>

{<Class name="Class1" uri= <http://../..#Class1>” />

<SuperClasses >

{{<SuperClass name=”SuperClass1” uri=”<http://../..#SuperClass1>” />} n

</SuperClasses>

<SubClasses>

{<SubClass name=”SuperClass1” uri=”<http://../..#SuperClass1>” />} n

</SubClasses>

<Instances>

{<Instance name=”Inst1” uri=”http://../..#Instance1” />}n

</Instances>

</Class>} m

</Classes>

m Number of classes passed in the request URL.

n: Can occur any number of times.

Errors:

* Class does not exist -> 404 “ no such class exist”
* Ontology is not loaded -> 404 “ requested ontology is not loaded”
* Any other delimiter used instead of comma -> 400

**PUT**

This requests, updates each class passed in the URL. Information for each class, that has to be updated is passed as request body in xml format as described below.

Request:

*Resource*

{ontologyName}/classes/<comma separated classes>

*Content Type:*

application/xml

*Body:*

The request body should be in following format.

<Classes>

{<Class name="Class1" uri= <http://../..#Class1>” />

<SuperClasses>

{<SuperClass name=”SuperClass1” uri=”<http://../..#SuperClass1>” >

<Update name=”N1” uri=”<http://../..#N1> ”/>

</SuperClass>} n

</SuperClasses>

<SubClasses>

{<SubClass name=”SuperClass1” uri=”<http://../..#SuperClass1>” />

<Update name=”M1” uri=<http://../..#M1>” />

</SubClass>} n

</SubClasses>

<Properties>

{<Property name=”Prop1” uri=”<http://../..#Prop1>” />

<Update name=”P1” uri=”<http://../..#P1>” />

</Property>} n

</Properties>

<Instances>

{<Instance name=”Inst1” uri=”<http://../..#Instance1>” />

<Update name=”I1” uri=”[http://../.. #I1](http://../..#I1)” />} n

</Instances>

</Class>} m

</Classes>

m: Number of classes passed in the request URL.

n: Can occur n number of times.

**DELETE**

This request deletes the classes mentioned in the URL from the ontology.

Request:

*Resource*

/classes/<comma separated names of classes>

2) Sub-Classes

This service provides an interface to browse, add, update or delete any class from the requested ontology. The client application sends a URL that has the ontology name and a list of comma-delimited classes. The sub-classes of these comma-delimited classes are modified.

**GET**

This request retrieves sub-classes of all the classes mentioned in comma-delimited fashion in the URL.

Request:

*Resource*

/classes/<comma separated names of classes>

Response:

*Content Type:*

application/xml

*Body*:

For each class mentioned in the request URL a Class tag is returned. Each class tag has a list of SubClass tag enclosing all the sub-classes for this particular class. All the classes are enclosed in Classes tag.

<Classes>

{<Class name="Class1" uri= <http://../..#Class1>” />

<SubClasses>

{<SubClass name=”SuperClass1” uri=”<http://../..#SuperClass1>” />} n

</SubClasses>

</Classes>} m

</Classes>

m Number of classes passed in the request URL.

n: Can occur n number of times.

Errors:

* Class does not exist -> 404 “ no such class exist”
* Ontology is not loaded -> 404 “ requested ontology is not loaded”
* Any other delimiter used instead of comma -> 400

**POST**

This request creates sub-classes in the ontology mentioned in the request URL. The classes for which sub-classes are to be added are listed in comma-delimited fashion in the request URL.

Request:

*Resource*

{ontologyName}/subClassesOf/<comma separated classes>

*Content Type:*

application/xml

*Body:*

Information for each sub-class that has to be created is provided as the request body in xml format as described below. For each class mentioned in the URL, a Class tag is required. Each class tag contains a list of SubClass tag. The SubClass tag describes the sub-class that has to be added to the ontology. If the class mentioned in the SubClass tag is not present in the ontology, it will be added.

<Classes>

{<Class name="Class1" uri= <http://../..#Class1>” />

<SubClasses>

{<SubClass name=”SuperClass1” uri=”<http://../..#SuperClass1>” />} n

</SubClasses>

</Classes>} m

</Classes>

m Number of classes passed in the request URL.

n: Can occur n number of times.

Errors:

* Class does not exist -> 404 “ no such class exist”
* Ontology is not loaded -> 404 “ requested ontology is not loaded”
* Any other delimiter used instead of comma -> 400

**PUT**

This requests, updates sub-classes of each class mentioned in the URL. Information for each class that has to be updated is passed as request body in xml format as described below.

Request:

*Resource*

{ontologyName}/subClassesOf/<comma separated classes>

*Content Type:*

application/xml

*Body:*

Information for each sub-class that has to be updated is provided as the response body in xml format as described below. For each class mentioned in the URL, a Class tag is required. Each class tag contains a list of SubClass tag. The SubClass tag describes the sub-class that has to be updated and each SubClass tag contains a Update tag mentioning the sub-Class that has to be added as the sub-class. The Update tag should mention a class that is already present in the ontology, new class will not be created.

<Classes>

{<Class name="Class1" uri= <http://../..#Class1>” />

<SubClasses>

{<SubClass name=”SuperClass1” uri=”<http://../..#SuperClass1>” />

<Update name=”M1” uri=<http://../..#M1>” />

</SubClass>} n

</SubClasses>

</Classes>} m

</Classes>

m: Number of classes passed in the request URL.

n: Can occur n number of times.

Errors:

* Class does not exist -> 404 “ no such class exist”
* Sub-Class does not exist -> 404 “no such sub-class exist”
* Ontology is not loaded -> 404 “ requested ontology is not loaded”
* Any other delimiter used instead of comma -> 400

**DELETE**

This request removes the sub-classes of the classes mentioned in the URL from the ontology.

Request:

*Resource*

/subClassesOf/<comma separated names of classes>

1. Super-Classes

**GET**

This request retrieves super-classes of all the classes mentioned in comma-delimited fashion in the URL.

Request:

*Resource*

/superClassesOf/<comma separated names of classes>

Response:

*Content Type:*

application/xml

*Body*:

For each class mentioned in the request URL a Class tag is returned. Each class tag has a list of SuperClass tag enclosing all the super-classes for this particular class. All the classes are enclosed in Classes tag.

<Classes>

{<Class name="Class1" uri= <http://../..#Class1>” />

<SuperClasses>

{<SuperClass name=”SuperClass1” uri=”<http://../..#SuperClass1>” />} n

</SuperClasses>

</Classes>} m

</Classes>

m Number of classes passed in the request URL.

n: Can occur n number of times.

**POST**

This request creates super-classes in the ontology mentioned in the request URL. The classes for which super-classes are to be added are listed in comma-delimited fashion in the request URL.

Request:

*Resource*

{ontologyName}/superClassesOf/<comma separated classes>

*Content Type:*

application/xml

*Body:*

Information for each super-class that has to be created is provided as the request body in xml format as described below. For each class mentioned in the URL, a Class tag is required. Each class tag contains a list of SuperClass tag. The SuperClass tag describes the super-class that has to be added to the ontology. If the class mentioned in the SuperClass tag is not present in the ontology, it will be added.

<Classes>

{<Class name="Class1" uri= <http://../..#Class1>” />

<SuperClasses>

{<SuperClass name=”SuperClass1” uri=”<http://../..#SuperClass1>” />} n

</SuperClasses>

</Classes>} m

</Classes>

m Number of classes passed in the request URL.

n: Can occur n number of times.

Errors:

* Class does not exist -> 404 “ no such class exist”
* Ontology is not loaded -> 404 “ requested ontology is not loaded”
* Any other delimiter used instead of comma -> 400

**PUT**

This requests, updates super-classes of each class mentioned in the URL. Information for each class that has to be updated is passed as request body in xml format as described below.

Request:

*Resource*

{ontologyName}/superClassesOf/<comma separated classes>

*Content Type:*

application/xml

*Body:*

Information for each sub-class that has to be updated is provided as the response body in xml format as described below. For each class mentioned in the URL, a Class tag is required. Each class tag contains a list of SubClass tag. The SubClass tag describes the sub-class that has to be updated and each SubClass tag contains an Update tag mentioning the super-class that has to be added as the super-class. The Update tag should mention a class that is already present in the ontology, new class will not be created.

<Classes>

{<Class name="Class1" uri= <http://../..#Class1>” />

<SuperClasses>

{<SuperClass name=”SuperClass1” uri=”<http://../..#SuperClass1>” />

<Update name=”M1” uri=<http://../..#M1>” />

</SuperClass>} n

</SuperClasses>

</Class>} m

</Classes>

m: Number of classes passed in the request URL.

n: Can occur n number of times.

Errors:

* Class does not exist -> 404 “ no such class exist”
* Super-Class does not exist -> 404 “no such super-class exist”
* Ontology is not loaded -> 404 “ requested ontology is not loaded”
* Any other delimiter used instead of comma -> 400

**DELETE**

This request removes the super-classes of the classes mentioned in the URL from the ontology.

Request:

*Resource*

/superClassesOf/<comma separated names of classes>

4) Properties

This service provides an interface to browse, add, update or delete any property from the requested ontology

**GET**

This request retrieves properties of all the classes mentioned in comma-delimited fashion in the URL.

Request:

*Resource*

/properties/<comma separated names of properties>

Response:

*Content Type:*

application/xml

*Body*:

For each property mentioned in the request URL, its declaring classes, its domains and ranges are returned in a XML format described below.

<Properties>

{<Property name="P1" uri= <http://../..#P1>” />

<DeclaringClasses>

{<Class name=”Class1” uri=”<http://../..#Class1>” />} n

</DeclaringClasses>

<Domain>

{<Class name=”Class2” uri=”http://../..#Class2” />} n

</Domain>

<Range>

{<Class name=”Class3” uri=”http://../..#Class3” />} n

</Range>

</Property>} m

</Properties>

m Number of classes passed in the request URL.

n: Can occur n number of times.

Errors:

* Property does not exist -> 404 “ no such property exist”
* Ontology is not loaded -> 404 “ requested ontology is not loaded”
* Any other delimiter used instead of comma -> 400

**POST**

This request creates properties mentioned in the URL.

Request:

*Resource*

{ontologyName}/properties/<comma separated properties>

*Content Type:*

application/xml

*Body:*

Information for each property to be added in the ontology has to be provided in XML format as described below. The DeclaringClasses tag contains a list of classes to which this property will be associated. The Domain tag contains a list of classes that will be added as the domain of the property. The Range tag contains a list of classes that will be added as the range of the property. Classes mentioned as Domain or Range for the property need to be existent classes in the ontology, no new classes are created using this service.

<Properties>

{<Property name="P1" uri= <http://../..#P1>” />

<DeclaringClasses>

{<Class name=”Class1” uri=”<http://../..#Class1>” />} n

</DeclaringClasses>

<Domain>

{<Class name=”Class2” uri=”http://../..#Class2” />} n

</Domain>

<Range>

{<Class name=”Class3” uri=”http://../..#Class3” />} n

</Range>

</Property>} m

</Properties>

m Number of properties passed in the request URL.

n: Can occur n number of times.

Errors:

* Property does not exist -> 404 “ no such property exist”
* Ontology is not loaded -> 404 “ requested ontology is not loaded”
* Any other delimiter used instead of comma -> 400

**PUT**

This requests, updates properties mentioned in the URL.

Request:

*Resource*

{ontologyName}/properties/<comma separated classes>

*Content Type:*

application/xml

*Body:*

Information for each property that has to be updated is passed as request body in xml format as described below. For each property mentioned in the URL, a Property tag is required. The Property tag contains DeclaringClasses, Domain and Range tags. The DeclaringClasses tag contains a Class tag that mentions the declaring class that has to be updated and the Class tag contains an Update tag mentioning the class with which the property will be updated. The Domain tag contains a Class tag that mentions the domain class for this property that has to be updated and the Class tag contains an Update tag mentioning the domain class with which this property will be updated. The Range tag contains a Class tag that mentions the range class for this property that has to be updated and the Class tag contains an Update tag mentioning the range class with which this property has to be updated. The Update tag should mention a class that is already present in the ontology, new class will not be created.

<Properties>

{<Property name="P1" uri= <http://../..#P1>” />

<DeclaringClasses>

{<Class name=”Class1” uri=”<http://../..#Class1>” />

<Update name=”M1” uri=<http://../..#M1>” />

</Class>} n

</DeclaringClasses>

<Domain>

{<Class name=”Class1” uri=”<http://../..#Class1>” />

<Update name=”M1” uri=<http://../..#M1>” />

</Class>} n

</Domain>

<Range>

{<Class name=”Class1” uri=”<http://../..#Class1>” />

<Update name=”M1” uri=<http://../..#M1>” />

</Class>} n

</Range>

</Property>} m

</Properties>

m Number of properties passed in the request URL.

n: Can occur n number of times.

Errors:

* Property does not exist -> 404 “ no such property exist”
* Ontology is not loaded -> 404 “ requested ontology is not loaded”
* Any other delimiter used instead of comma -> 400

**DELETE**

This request removes the properties mentioned in the URL from the ontology.

Request:

*Resource*

/properties/<comma separated names of properties>

5) Sub-properties

This service provides an interface to browse, add, update and delete sub-properties of properties mentioned in the URL.

**GET**

This request retrieves sub-properties of all the comma-delimited properties mentioned in the URL.

Request:

*Resource*

/subPropertiesOf/<comma separated names of properties>

Response:

*Content Type:*

application/xml

*Body*:

For each property mentioned in the request URL a Property tag is returned. Each property tag has a list of SubProperties tag enclosing all the sub-properties for this particular property. All the properties are enclosed in Properties tag.

<Properties>

{<Property name="Prop1" uri= <http://../..#Prop1>” />

<SubProperties>

{<SubProperty name=”SuperProp1” uri=”<http://../..#SuperProp1>” />} n

</SubProperties>

</Property>} m

</Properties>

m Number of properties passed in the request URL.

n: Can occur n number of times.

Errors:

* Property does not exist -> 404 “ no such property exist”
* Ontology is not loaded -> 404 “ requested ontology is not loaded”
* Any other delimiter used instead of comma -> 400

**POST**

This request creates sub-properties in the ontology mentioned in the request URL. The properties for which sub-classes are to be added are listed in comma-delimited fashion in the request URL.

Request:

*Resource*

{ontologyName}/subClassesOf/<comma separated classes>

*Content Type:*

application/xml

*Body:*

Information for each sub-property that has to be created is provided as the request body in xml format as described below. For each property mentioned in the URL, a Property tag is required. Each property tag contains a list of SubProperty tag, which describes the sub-class that has to be added to the ontology. If the Property mentioned in the SubProperty tag is not present in the ontology, it will be added.

<Properties>

{<Property name="Prop1" uri= <http://../..#Prop1>” />

<SubProperties>

{<SubProperty name=”SuperProp1” uri=”<http://../..#SuperProp1>” />} n

</SubProperties>

</Property>} m

</Properties>

m Number of properties passed in the request URL.

n: Can occur n number of times.

Errors:

* Property does not exist -> 404 “ no such property exist”
* Ontology is not loaded -> 404 “ requested ontology is not loaded”
* Any other delimiter used instead of comma -> 400

**PUT**

This requests, updates sub-properties of each property mentioned in the URL. Information for each property that has to be updated is passed as request body in xml format as described below.

Request:

*Resource*

{ontologyName}/subPropertiesOf/<comma separated properties>

*Content Type:*

application/xml

*Body:*

Information for each sub-property that has to be updated is provided as the request body in xml format as described below. For each property mentioned in the URL, a Property tag is required. Each Property tag contains a list of SubProperty tag. The SubProperty tag describes the sub-property that has to be updated and each SubProperty tag contains an Update tag mentioning the sub-Property with which this property has to be updated with. The Update tag should mention a property that is already present in the ontology, new properties will not be created.

<Properties>

{<Property name="Prop1" uri= <http://../..#Prop1>” />

<SubProperties>

{<SubProperty name=”SuperProp1” uri=”<http://../..#SuperProp1>” />

<Update name=”M1” uri=<http://../..#M1>” />

</SubProperty>} n

</SubProperties>

</Property>} m

</Properties>

m: Number of properties passed in the request URL.

n: Can occur n number of times.

Errors:

* Property does not exist -> 404 “ no such property exist”
* Sub-Property does not exist -> 404 “no such sub-property exist”
* Ontology is not loaded -> 404 “ requested ontology is not loaded”
* Any other delimiter used instead of comma -> 400

**DELETE**

This request removes the sub-properties of the properties mentioned in the URL from the ontology.

Request:

*Resource*

/subPropertisOf/<comma separated names of classes>

4.3 Navigation Service

This service provides an interface to navigate from one class or instance of ontology to another class or instance along the properties. The client application provides the path query in form of a RESTful URI. Following is the query syntax for invoking this service and brief explanation about how it works.

Query syntax:

navigate/{ontologyName}/**class/{className}/{property}+**

navigate**/**{ontologyName}/**individual/{individualName}/{property}**

{className}/{individualName} provides us the starting point for navigation. We than go on navigating through the ontology according the **paths** specified by the associations/properties in the query path.

How does it work?

We start with one class/individual, which is our starting point. We navigate using the first property mentioned to all the classes associated with our starting point class and we collect them. Then we apply the second property mentioned in the path query on each of the class from the resulting from the first query. To make things more clear consider an example from the pizza ontology.

**navigate/pizza/class/AmericanPizza/hasTopping/hasSpeciness/**

We first locate the AmericanPizza in the pizza ontology.

Then we use the first property mentioned in the path query, which is hasTopping in this example.

We get all the classes that are associated to AmericanPizza using the property hasTopping

**AmericanPizza/hasTopping/ => [PeperoniTopping, MozerellaTopping]**

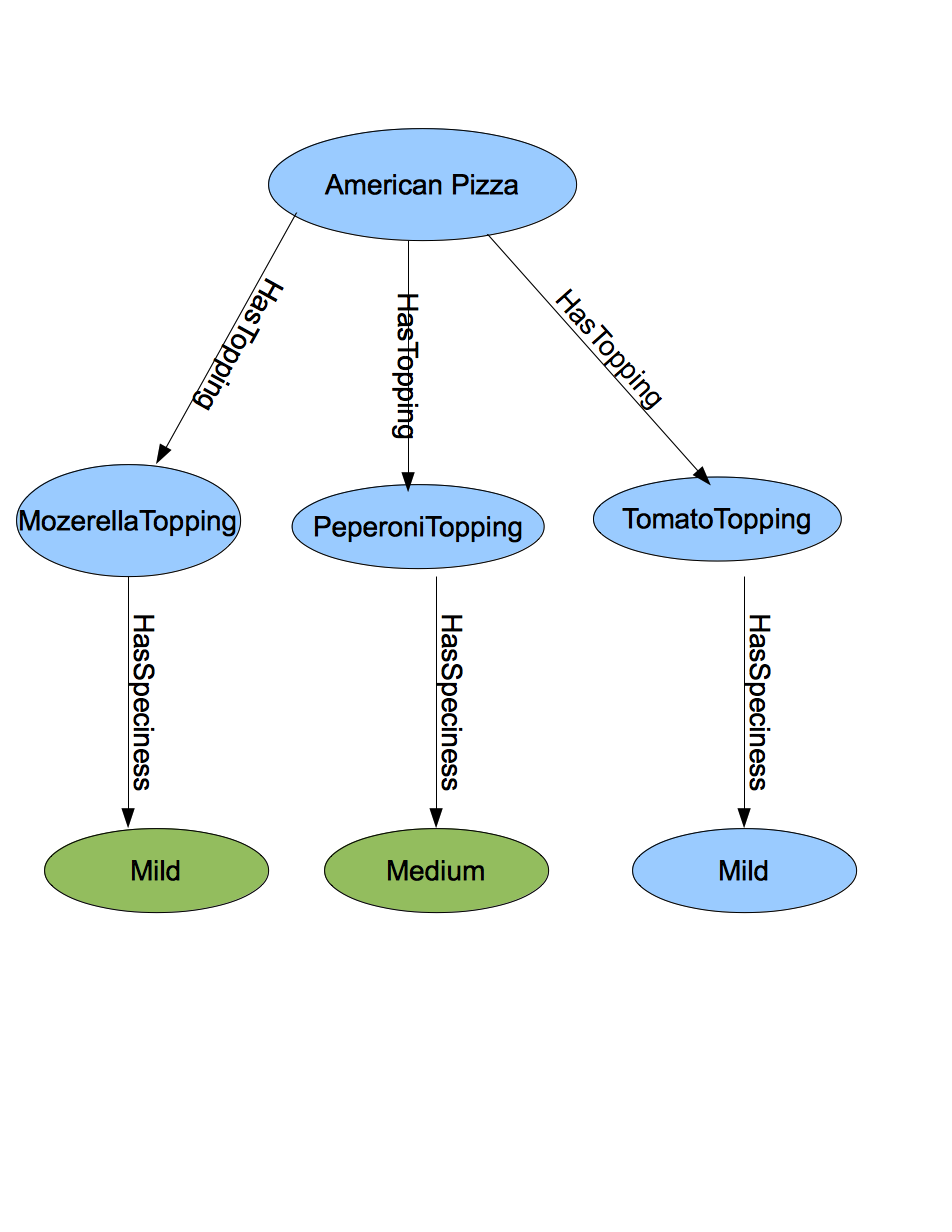
In this case we get PeperoniTopping and MozerellaTopping classes associated to AmericanPizza using the property hasTopping.

We then apply the second property in the path query (hasSpeciness) on each PerperoniTopping and MozerellaTopping.

**[Peperoni, Mozerella]/hasSpiceness => [Mild, Medium]**

So the output of the navigational query is **[Mild, Medium]** and it is returned to the user.

In short, we start with one class (starting point) and the apply first property to get a result of interim classes on which we then apply the second property to get another new set of interim result on which we apply the third property and so on. We do this till at any state we don’t get any interim result or we are done processing all the properties mentioned in the path, whichever one occurs first.  Apart from properties user can also specify relations like subClassOf, superClassOf, instancesOf, equivalntClasses, disjointClasses, complementClasses. Please



4.4 SPARQL Query Service

This module provides an interface for the client application to execute a SPARQL query.

The URI template to invoke this service is

sparql/{ontologyName}

Where the {ontologyName} is the ontology that the user wants to query.

Following are the HTTP operations that this service supports

**POST**

This service accepts a SPARQL query in form the request body encoded in XML. The XML format to invoke this service is explained below.

Request

*Resource*

sparql/{ontologyName}

*Content Type*

application/XML

*Body*:

The request has to be encoded in XML as follows. The SPARQL query is enclosed within Query tags.

<Query>

{ SPARQL Query }

</Query>

Response

*Content type*

application/XML

*Body*

The response returned to the client is encoded in XML and is in following format. The Result tag has two attributes. The id attribute notifies the client application the unique identifier for the result set produced by the execution of the query requested. The timeout attribute notifies the client application for how much time the result set will be cached on the server. The client application will have to make a GET request passing the identifier within the timeout time to get the result from the server. The time mentioned by the timeout attribute is in seconds.

<ResultSet>

<Result id=”xyz” timeout=”180” />

</ResultSet>

**GET**

This service provides an interface for the client application to retrieve the result set of a earlier SPARQL query executed. The client application provides a unique identifier to identify the result set along with the format in which it wants the result set.

Request

*Resource*

sparql/{ontologyName}/[[3](#_ENREF_3)]/{format}

Response

The response provided by this service depends upon the format the client application passed in the request URL.

REFERECES

1. Brickley, D. and R.V. Guha, *Resource Description Framework (RDF) Schema Specification 1.0: W3C Candidate Recommendation 27 March 2000.* 2000.

2. Fielding, R.T., *Architectural styles and the design of network-based software architectures*, 2000, Citeseer.

3. McBride, B., *Jena: A semantic web toolkit.* Internet Computing, IEEE, 2002. **6**(6): p. 55-59.

4. Broekstra, J., A. Kampman, and F. Van Harmelen, *Sesame: A generic architecture for storing and querying rdf and rdf schema.* The Semantic Web—ISWC 2002, 2002: p. 54-68.

5. Beckett, D., *Redland RDF application framework.* Institute for Learning and Research Technology, University of Bristol, 2004.

6. Clark, K.G., L. Feigenbaum, and E. Torres, *SPARQL protocol for RDF.* World Wide Web Consortium (W3C) Recommendation, 2008.

7. Pautasso, C. and E. Wilde, *Why is the web loosely coupled?: a multi-faceted metric for service design*, in *Proceedings of the 18th international conference on World wide web*2009, ACM: Madrid, Spain. p. 911-920.

8. Auer, S., et al., *Dbpedia: A nucleus for a web of open data.* The Semantic Web, 2007: p. 722-735.

9. Suchanek, F.M., G. Kasneci, and G. Weikum. *Yago: a core of semantic knowledge*. 2007. ACM.

10. Farquhar, A., R. Fikes, and J. Rice, *The ontolingua server: A tool for collaborative ontology construction.* International Journal of Human-Computers Studies, 1997. **46**(6): p. 707-727.

11. Swartout, B., et al. *Ontosaurus: a tool for browsing and editing ontologies*. 1996.

12. Eklund, P., N. Roberts, and S. Green. *Ontorama: Browsing rdf ontologies using a hyperbolic-style browser*. 2002. IEEE.

13. Ahmad, M.N. and R.M. Colomb, *Managing ontologies: a comparative study of ontology servers*, in *Proceedings of the eighteenth conference on Australasian database - Volume 63*2007, Australian Computer Society, Inc.: Ballarat, Victoria, Australia. p. 13-22.

14. Wilde, E. and M. Hausenblas, *RESTful SPARQL? You name it!: aligning SPARQL with REST and resource orientation*, in *Proceedings of the 4th Workshop on Emerging Web Services Technology*2009, ACM: Eindhoven, The Netherlands. p. 39-43.