RIO: RESTful INTERFACE TO ONTOLOGY

by

CHINMAY KALE

(Under the Direction of Krzysztof J. Kochut)

ABSTRACT

The vision of the Semantic Web is to transform the World Wide Web from a web of linked documents to a web of linked data. Using the RESTful style, Web applications can navigate among resources, discover new resources, modify them and perform other tasks. If we view an ontology as a set of triples forming a graph, it is similar in its organization to the World Wide Web. Hence, using a set of RESTful style services, we should be able to perform similar operations on an ontology. In this thesis, we present a design and a prototype implementation of RIO, a RESTful Interface to Ontologies. RIO provides a RESTful interface to manage, edit, and query OWL ontologies. RIO also provides a novel way of navigation within an ontology based on URIs representing ontology paths. In addition, RIO provides a unique way to execute SPARQL queries in a RESTful way.

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

RIO: 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

RIO: 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
December 2011

DEDICATION

To my parents, brother and loved ones.

ACKNOWLEDGEMENTS

I would like to thank my major advisor, Dr. Krys J. Kochut, for being an inspiring mentor and a supportive advisor for past three years. I would also like to thank my committee, Dr. Miller and Dr. York for their support, time and valuable suggestions.

I thank my friends Uthaya and Ankur, for their help and ideas to improve my thesis. Last, but certainly not the least, I would thank all my loving friends and family for their support without which this work wouldn't have been possible.

TABLE OF CONTENTS

Page

LIST OF FIGURES

Page

CHAPTER 1

INTRODUCTION

1.1 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 intended for human use. 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 so 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, which represent the relationship between the resources.

1.2 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 such as "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 [1]. Ontologies represent shared domain-specific knowledge, which can be shared between machines and people. Ontologies can be expressed in increasingly expressive languages such as: RDF-Schema [2], Web Ontology Language (OWL) [3]. Both these languages allow ontologies to be modeled as directed labeled graphs [4] where the nodes of the graphs are the concepts and the labeled edges are the relationships between the concepts.

1.3 URL, URI and URN

URI stands for Uniform Resource Identifier. It is a string of characters used to identify resources in a distributed system such as Internet. Such a representation of resources in a distributed system enables interaction between the resources over network. A URI identifies a resource either using a name or location. Depending upon this, the URI can be classified into URL, Unified Resource Location or Unified Resource Name.

The goal of Semantic Web is to empower Web-based agents with the ability to process and understand the data instead of merely just displaying it [5] . On the other hand ontologies are the formal specification and description of concepts of a particular domain. Thus, ontologies can provide knowledge to Web-based agents. And with the help of this knowledge it will be possible for the agents to process and understand the data that is exchanged amongst them.

CHAPTER 2

BACKGROUND

2.1 Ontology Languages

Ontologies are used to model domain knowledge in a machine-readable fashion. This knowledge can range from some basic statement to complex axioms. To cater to the difference in the level of knowledge representation, there are increasingly expressive ontology languages:

- RDF/RDFS (RDF Schema)

  RDFS [6] allows users to model knowledge using resources which might be classes, properties or instances. Group of similar instances belong to a class. Every instance has its type specified using `rdf:type` property. RDFS allows hierarchical relation between classes with the `subClassOf` relationship. A property is applied to a class and can be considered as its attribute. The domain of a property specifies the classes to which the property can be applied. The range of a property restricts the classes of instances, which can be a valid value for that property.

- Web Ontology Language (OWL)

  OWL [7] is one of the most prominent languages for publishing and sharing domain knowledge through ontologies. It adds to the expressiveness of RDFS by providing a framework for framing more complex knowledge components. We will not introduce the

complete description of the OWL language, as this goes beyond the scope of the thesis. The reader should consult [8] for the complete specification of OWL. Some of the RDFS lacking features provided by OWL are:

- Ability to state equivalence between two resources.
- Ability to state the cardinality of properties.
- Expressing the range of a property to be a closed set.
- Ability to state disjointedness of classes.
- Ability to express classes as unions and intersections of other classes

2.2 Ontology Servers

We were unable to name a formal definition of ontology servers after a review of literature in this research area. [9] state that the notion of Ontology Server originated from the research of ontology development tools. Most of the previous works describe an ontology server, implicitly or explicitly, as a kind of isolated or integrated tool for building ontologies [10, 11]. Whereas some studies discuss ontology servers as ontology repositories [12, 13]. On the other hand, some other studies discuss both ontology repository as well as server functionality, so in this context, the server is described as an information system [14, 15]. Based on the literature review, we believe that a server that provides an interface to manage, browse, edit and navigate ontologies can be called an ontology server.

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 the 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 and ontology translator.

Many servers expose their APIs through web-services to have the advantage of interoperability. Two most popular styles of implementing web-services are SOAP and REST. W3C defines SOAP as " a lightweight protocol specification for exchanging structured information in a decentralized, distributed environment" [16]. XML technologies are used by SOAP to define an extensible messaging framework. The framework provides a message construct that can be exchanged over a variety of underlying protocols. SOAP, also defines a processing model that specify rules for processing SOAP messages, an extensibility model, that defines the SOAP features and modules and a protocol binding framework, that defines the underlying protocol binding framework and also specifies the rules for defining the binding to the underlying protocol that can be used to exchange SOAP messages between client and server. The client to invoke a SOAP based web-service has to send information encoded in XML according to the SOAP specification. On the other hand REST web-services uses the ubiquitous HTTP protocol for exchanging the information as well as invoking the web-service. As most of the modern languages and web-browser have native support for HTTP protocol, REST web services are more scalable. We will describe more about REST web-services later in this chapter.

For our purpose, we assume that 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 of them:

- Jena

  Jena [17] 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 [18] is a free open source framework for storage, inferencing and querying RDF data. It provides features similar to Jena. Sesame's focus is on the RDF data storage and query, but without much support for OWL and related inferencing tools.

- Redland

  Redland [19] 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 also called, as Virtuoso [20] 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.

- BRAHMS

  BRAHMS [21] is a RDF store primary geared for high performance semantic association discovery. It is a main-memory RDF store. It provides read-only access to client-applications. RDF triples are indexed and provide very fast access and semantic association discovery.

2.3 SPARQL endpoint

SPARQL [22] is an RDF query language and data access protocol 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 [23] is the core specification of SPARQL query language. Together with this language specification is the SPARQL Query XML Results Format specification [24], which describes an XML format for serializing the results of a SPARQL query. The third specification is he SPARQL Protocol for RDF (SPROT) specification [25] 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 and few others. SPARQL endpoints can be categorized as generic endpoints and specific end- 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 [22] 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.

2.4 Introduction to REST

Representation State Transfer (REST) was introduced and described by Roy Fielding in his doctoral dissertation [26]. In the dissertation he put forth the architectural principles of the Web. He presented these architectural principles as a framework of constraints. According to him these framework of constraint describe how large-scale distributed information systems such as Web are built and operated. He stated that the core of such distributed systems is its resources and the interplay between them. In his dissertation, he advocated using a limited set of operations with uniform semantics to build a ubiquitous infrastructure that can support any type of application. He referred to this architectural style as *REpresentational State Transfer,* or REST.

According to him, this framework is the reason for how scalable, mash-up able, usable and accessible the Web is. With these observations, he states that if distributed systems are designed using these constrained, they will have above stated advantages.

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 dereferenceable irrespective of 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 provide 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 [27], this constraint is the most important reason for supporting loose 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 our 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 add 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.

For the thesis, we have built an ontology server, RIO that provides a RESTful Interface to Ontologies. It provides RESTful sub-services for ontology management, browsing, editing, navigation and execution of SPARQL query.

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 a web of linked documents to a web of linked data. Maturing Semantic Web technology stack fuels the increasing interest in publishing semantically linked data. Within recent years we have witnessed creation of very large ontologies such as Dbpedia [28], YAGO [29], UniProt [30] being published. On the other hand many domain specific ontologies such as GlycO [31], ProkinO [32] are also being published. The applications that interoperate among various such domains seek for an alignment among the ontologies from these domains. As a 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 applications such as editors, browsers and visualizers have a hard time processing such large ontologies. Ontology navigation can be of great help to solve these problems. Navigating ontology to a point of interest can provide the ontology client 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 Create-Read-Update-Delete (CRUD) operations. Also these 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 documents can be seen analogous to concepts (classes, properties, instances) in ontology. Each document in Web of linked document has a unique addressable URI and so do concepts in ontology. Using REST web-services, we have applications that can navigate between various documents, discovery new documents, modify different documents and perform other similar tasks, in the Web of linked documents. With this as our motivation, we believe that we can perform similar tasks using REST web-services within an ontology.

SPARQL protocol [22] 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.

The vision of Semantic Web is to transform WWW from a web of linked documents to a web of linked data. Topology of WWW is similar to topology of ontologies. Using REST web-services, we have applications that can navigate between various documents, discovery new documents, modify different documents and perform other similar tasks, in the Web of linked documents. Hence, with REST-Web Services, we should be able to perform similar operations on an ontology. With this motivation, in this thesis we present design and implementation of RIO, a RESTful Interface to Ontology server framework. RIO provides RESTful interface to manage, edit and modify an OWL ontology. RIO also provides a novel way of programmatic navigation within the Ontology. Apart from this, RIO provides a unique way to execute SPARQL query is a RESTful way.

CHAPTER 4

RELATED WORK

Many free open-source as well as commercial ontology servers are available or being developed. Web-Protégé [33], AllegroGraph [34], Ontology-browser [35], Virtuoso [20], and KAON [36], BRAHMS [21] are few such ontology servers. These ontology servers mainly provide functionalities to maintain and manage ontologies. They primarily vary by the way they have implemented the maintenance operations, features they provide and the type of Application Programming Interfaces (API) they expose. It is very common and intuitive for an ontology server to expose these functionalities as SOAP or REST APIs. Providing these known standard web-service APIs allow client stubs to interact with the server dynamically. However some existing ontology servers provide custom APIs that are developed for the client applications. Their functionalities are tuned to the application served hence they may not support some general functionalities expected in an ontology servers. Web-Protégé is one of those application centric ontology servers. It is an open-source server providing lightweight, web-based ontology editor with a web browser based graphical user interface. The server side component is developed using Protégé-OWL [37] API services exposed using SOAP web services. There are few general-purpose ontology servers that mainly focus on ontology storage functionality, such as Virtuoso, Allegograh, Redland. Virtuoso provides features such as a relational database engine, web-server and a file server. Surprisingly only few ontology servers currently adopt REST. One of them is

Allegrograh ontology server, which provides a RDF triple store and access to the triple store via REST web services. It is a high performance persistent graph database engine developed by Franz Inc. However AllegroGraph has no provision to edit, browse or navigate OWL ontologies. Ontology-Browser is another rest kind ontology server providing features such as browsing OWL ontologies, executing SPARQL queries, and dynamically loading ontologies in the server. Though it adopts REST architecture, the web-services exposed by the server do not follow REST principles. Also it allows read-only access to the ontologies and there is no support for editing or updating any concept from ontology. Both AllegroGraph and Ontology-Browser provide SPARQL endpoints, but their implementation of the SPARQL endpoint is not RESTful.

Most of the ontology navigation services are provided using graphical user interface. Such tools render ontology as a node-link diagram and navigation is provided using a click-expand-navigate model. Web-Protégé [33], NavEditOW [38] are few such ontology servers which provide browser based ontology navigation as described allow. None of the above mentioned ontology servers provide an API for ontology navigation.

The focus of currently available ontology servers is on ontology development, storage and management. On the other hand, RIO server focuses upon providing a REST interface for editing and navigating ontologies. Also it provides a novel way of executing SPARQL queries in a RESTful way.

CHAPTER 5

SYSTEM DESIGN

The RIO server is an ontology server implemented in Java using the Jena framework. It provides REST web services to applications, which require navigating, managing, performing CRUD operations and executing SPARQL queries on ontologies. Each of these features is exposed as sub-service through REST web-services. The figure below depicts the overall system architecture of RIO server.

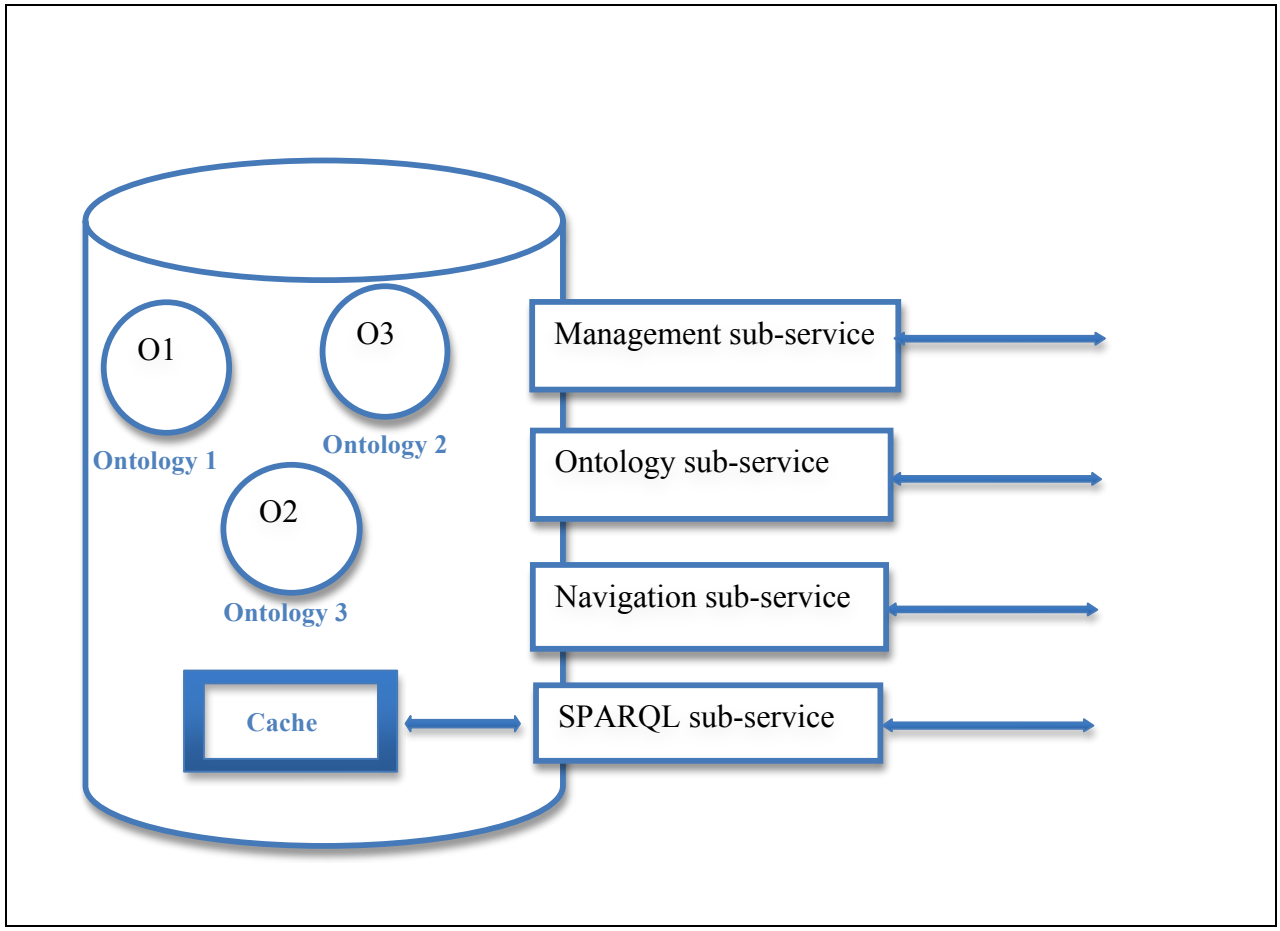

Figure 1: System Architecture of the RIO server.

RIO is a J2EE specification compliant server and has the capability of serving REST web-services invoked via HTTP protocol. Multiple OWL ontologies can reside in the server at the same time. The server has four sub-service interfaces namely Ontology Management sub-services, Ontology sub-services, Navigation sub-services and SPARQL query execution sub-services. Each sub-service is designed strictly according to REST principles. Any service call made to this ontology server does not allow referring multiple ontologies together. However any ontology, which has been imported into another ontology, can be referred together with the later. Following is a detailed description of each service bundle interface. For explaining the design of the URIs we will be giving an example of URIs used to load, navigate, interact and execute SPARQL queries over wine ontology. Wine ontology [39] is an ontology developed at University Jaume I of Castellón, Spain.

5.1 Ontology Management Sub-Service

This sub-service provides functions to manage the OWL ontologies that are deployed in the server. The client application invokes this sub-service, if it wants to load an ontology that is currently not loaded in the server. It also provides a utility for the client application to take a snapshot of any ontology that is currently loaded by requesting to serialize it into an OWL file. This feature is provided because RIO provides sub-service interface to execute CRUD operations on the loaded ontologies, so at any given point the client application can download a modified ontology as an OWL file from the server. Another utility provided by this sub-service is to remove a loaded ontology from RIO.

Also this sub-service loads some default ontology during server startup. The default ontologies are configured using a configuration file. This sub-service reads this file and loads the default ontologies during server startup.

Another useful utility provided by this sub-service is to perform validation and consistency checks. The client application invokes this utility to get a report of results of consistency and validation checks performed on any ontology loaded in RIO. This utility performs a global check across the schema and instance data for inconsistencies. If any inconsistencies are encountered, a report is returned to the client application.

Following is the URI design for invoking sub-services from this interface.

1) Load Ontology

PUT
Request:
Resource: /ontMgmt/{ontologyName}
Content Type: application/binary
Body:
The OWL file stream representing the ontology that is being loaded.
Response:
Success: 201 "Ontology Loaded"
Error: 404 "Bad request. "

Example:

```
PUT /ontMgmt/wine
```

2) Remove Ontology

DELETE
Request:
Resource: /ontMgmt/{ontologyName}
Response:
Status:
Success: 200 "OK"
Error: 404 " The ontology requested in not loaded in the server."

Example:

```
DELETE ontMgmt/wine
```

3) Validate Ontology

For each failed validation or inconsistency check, a report is returned to the client application in the following format. The Type tag defines the type of the failed validation or inconsistency check. The Description tag encloses a brief description of the failed validation or inconsistency check.

```
GET
Request:
Resource: ontMgmt/{ontologyName}/validate
Response:
Content Type: application/xml
Body:

<Reports>
      {<Report />

            <Type> type < Type />

            <Description> description … </Description>

      </ Report>...}
</Reports>
```

4) Display all ontologies loaded in the server.

For each ontology loaded in the server, an Ontology tag is returned. The Ontology tag specifies the local name of the ontology and its URI.

```
GET
Request:
Resource: ontMgmt/display
Response:
```

```
Content Type: application/xml
Body:

< Ontologies>

      {< Ontology name=”ont1” uri=” uri of ont” />…}

</ Ontologies>
```

Example:

```
GET ontMgmt/display

<?xml version="1.0" encoding="UTF-8"?>

<Ontologies>
  <Ontology name="wine"
      uri="http://krono.act.uji.es/Links/ontologies/wine.owl"/>
 <Ontology name="pizza"
      uri="http://www.co-ode.org/ontologies/pizza/pizza.owl" />
 <Ontology name="glyco"
      uri="http://glycomics.ccrc.uga.edu/ontologies/GlycO" />
</Ontologies>
```

5.2 Ontology Sub-Service.

This sub-service is designed to provide a RESTful interface for interacting with the concepts of an ontology. In OWL ontology the classes, properties, instances and restrictions are the concepts of interest. This sub-service provides methods to perform CRUD operations on these concepts. Each of these concepts is treated as resource in REST terminology and they have a unique URI. These 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. Currently JSON/XML encoded request/response is supported.

1) Sub-Services for Interacting with a class.

This sub-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 URI (as explained above).

a) Accessing a class

The client application passes a class name in the URI. This request returns information about the class passed in the URI. For the class, list of its super classes, sub-classes, properties and instances is returned in xml format (defined below) as response body. If the class does not exist in the ontology, an error with appropriate HTTP status code is returned to the client application. For the class mentioned in the request URL a Class tag is returned. The Class tag has SuperClasses, SubClasses, Instances, and Properties tag.

```
GET
Request:
Resource:  ontService/{ontologyName}/class/{className}
Response:
Content Type:    application/xml
Body:

<Class name="Class1" uri= http://serverAddress/#Class1" />

   <SuperClasses>
     {<SuperClass name="SuperClass1"
                uri="http://serverAddress/#SuperClass1" />…}
    </SuperClasses>

    <SubClasses>
     {<SuperClass name="SuperClass1"
                uri="http://serverAddress/#SuperClass1" />…}
    </SubClasses>

    <Properties>
     {<Property name="Prop1"
                 uri="http://serverAddress/#Prop1" />…}
    </Properties>
```

```
    <Instances>
      {<Instance name=”Inst1”
              uri=”http://serverAddress/#Instance1” />..}
     </Instances>
</Class>

Errors:
    • 404 “no such class exist”
    • 404 “requested ontology is not loaded”
```

Example:

```
GET ontService/wine/class/CheeseNutsDessert


<Class name="CheeseNutsDessert"
uri="http://krono.act.uji.es/Links/ontologies/food.owl#CheeseNutsDe
ssert">
    <SubClasses/>
    <SuperClasses>
       <SuperClass name="Dessert”
                   uri="http://krono.act.uji.es/Links/ontologies/
                        food.owl#Dessert"/>
    </SuperClasses>

    <Instances>
      <Instance name="Cheese"
                uri="http://krono.act.uji.es/Links/ontologies/
                    food.owl#Cheese"/>
      <Instance name="Nuts"
               uri="http://krono.act.uji.es/Links/ontologies/
                    food.owl#Nuts"/>
    </Instances>

   <Properties>
        <Property name="hasSugar"
                  uri="http://krono.act.uji.es/Links/ontologies/
                       wine.owl#hasSugar"/>
        <Property name="madeFromFruit"
                  uri="http://krono.act.uji.es/Links/ontologies/
                       food.owl#madeFromFruit"/>
        <Property name="hasMaker"
                  uri="http://krono.act.uji.es/Links/ontologies/
                       wine.owl#hasMaker"/>
        <Property name="madeIntoWine"
```

```
                    uri="http://krono.act.uji.es/Links/ontologies/
                         wine.owl#madeIntoWine"/>
          <Property name="hasFlavor"
                    uri="http://krono.act.uji.es/Links/ontologies/
                        wine.owl#hasFlavor"/>
        <Property name="locatedIn"
                  uri="http://krono.act.uji.es/Links/ontologies/
                      wine.owl#locatedIn"/>
        <Property name="hasBody"
                  uri="http://krono.act.uji.es/Links/ontologies/
                      wine.owl#hasBody"/>
       <Property name="producesWine"
                  uri="http://krono.act.uji.es/Links/ontologies/
                  wine.owl#producesWine"/>
    </Properties>
</Class>
```

b) Creating a class

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. The request body should be in following format. For the class that has to be added, a Class tag should be present. The Class tag has the name attribute required whereas the URI attribute is optional. The Class tag can have at most one SuperClass tag. For the SuperClass tag name attribute is required, URI attribute is optional. If the SuperClass tag is present, the class being created is added as sub-class of the class mentioned by the SuperClass tag, otherwise it is added as a top-level class (sub-class of owl: Thing).

```
PUT
Request:
Resource:  ontService/{ontologyName}/class/{className}
Content Type:    application/xml
Body:

      <Class name="Class1" uri= http://serverAddress/#Class1" />
```

```
            <SuperClass name="SuperClass1"
                  uri="http://serverAddress/#SuperClass1"  />
      </Class>


Response:
Status
Success:   201  "Class added".

Errors:
The request body encoded in XML is validated and following errors,
if encountered, are returned to client application
    • 404 "class already exists"
    • 404 "name attribute missing from class tag"
    • 404 "name attribute missing from superclass tag"
    • 404 "requested ontology is not loaded"
```

Example

```
PUT ontService/wine/class/newAmericanWine
      <Class name="newAmericanWine" />
            <SuperClass name="AmericanWine"  />
      </Class>
```

c) Updating a class

This requests, updates a class. The information required to update the class is sent, as the request body encoded in XML. The request body should be in the following format. The class to be updated is included in the Class tag. It should be a class that currently exists in the ontology, if not an error will be returned. The name attribute is required, and the local name of the class to be updated is to be included there. The URI attribute is

optional. The Class can contain at most one SuperClass tag. If the SuperClass tag is not included, the class is updated to be a top-level class (sub-class of owl: Thing). If the SuperClass tag is mentioned, it should have the name attribute mentioning any existing super-class of the class that has to be updated. The SuperClass tag should contain Update tag. The name attribute is required for the Update tag whereas the uri attribute is optional. The name attribute of Update tag mentions the name of the class that would be added as the new super-class for the class that is being updated. If the class mentioned in the name attribute of Update tag already exists in the ontology, it is re-used, otherwise a new class is created with that name and added as the super-class of the class that is being updated.

POST
Request:
Resource: ontService/{ontologyName}/class/{className}
Content Type: application/xml
Body:

```
<Class name="Class1"
        uri= http://serverAddress/#Class1" />


     <SuperClass name="SuperClass1"
                 uri="http://serverAddress/#SuperClass1"

           <Update name="newSuperClass"
                   uri="http://serverAddress/#newSuperClass"/>

      </SuperClass>

</Class>
```

Response:
Status
Success:
- 200 "Ok" Requested Class updated.

Errors:

The request body encoded in XML is validated and following errors, if encountered, are returned to client application

- 404 “class does not exists”
- 404 “name attribute missing from class tag”
- 404 “name attribute missing from superclass tag”
- 404 “super-class mentioned in request does not exist”
- 404 “name attribute missing from Update tag”
- 404 “requested ontology is not load

Example

```
POST ontService/wine/class
      <Class name="newAmericanWine" />

            <SuperClass name=”AmericanWine”  >
                  <Update name=”AlsatianWine” />
            </SuperClass>

      </Class>
```

d) Deleting a class

This request deletes the classes mentioned in the URL from the ontology. If multiple classes are request for deletion, the class names have to be comma delimited. If only one class is to be deleted, no need to delimit it with a comma. If any of the classes requested for deletion do not exist in the ontology, an error with the appropriate HTTP status code is returned to the client application.

DELETE
Request:
Resource: `ontService/{ontologyName}/class/{className}`
Response:
Status
Success:

- 200 “Ok” All requested classes are deleted

Errors:

- 404 “ requested ontology is not loaded”

Example

```
DELETE ontService/wine/class/newAmericanWine
```

2) Sub-Service for interacting with properties

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

a) Accessing Properties

The client application passes a list of comma-delimited names of properties in the URL. This request returns information about each property passed in the URL. For each property, the list of its super properties, sub-properties, domain and range is returned in XML format (defined below) as response body. If multiple properties are requested, the property names have to be comma delimited. If only one property is to be requested, no need to delimit it with a comma. If any of the comma-delimited property does not exist in the ontology, an error with appropriate HTTP status code is returned to the client application. For the property mentioned in the request URL, its domains, ranges, sub-properties and super-properties are returned in a XML format described below.

```
GET
Request:
Resource:  ontService/{ontologyName}/property/{propertyName}
Response:
Content Type:    application/xml
Body:

<Property name="P1" uri= http://serverAddress/#P1" />

     <SuperProperties>
           {<SuperProperty name ="SuperProp1
                        uri="http://serverAddress/#SuperProp1"/>…}
      </SuperProperties>

     <SubProperties>
           {<SubProperty name ="SubProp1
                        uri="http://serverAddress/#SubProp1"/>…}
      </SubProperties>


      <Domain>
           {<Class name="Class2"
                   uri="http://serverAddress/#Class2" />…}
      </Domain>

      <Range>
           {<Class name="Class3"
                   uri="http://serverAddress/#Class3" />…}
      </Range>

</Property>

Errors:
   • 404 " no such property exist"
   • 404 " requested ontology is not loaded"
```

Example:

```
GET ontService/wine/property/hasDrink

<Property name="hasDrink"
        uri="http://krono.act.uji.es/Links/ontologies/
             food.owl#hasDrink" type="object"/>

     <SuperProperties/>
```

```
    <SubProperties/>

     <Domain>

      <Class name="MealCourse"
             uri="http://krono.act.uji.es/Links/ontologies/
                  food.owl#MealCourse"/>
     </Domain>

     <Range>

      <Class name="MealCourse"
             uri="http://krono.act.uji.es/Links/ontologies/
                  food.owl#MealCourse"/>
     </Range>

</Property>
```

b) Creating properties

This request creates the property in the ontology. Currently the client application can create only a data or an object property. The information for the property to be added in the ontology has to be provided in XML format as described below. The Property tag includes the name and the type of the property that is to be created. The name and type attribute of the Property tag is required whereas the URI attribute is optional. The type attribute currently accepts only object or data. The Property tag can have at most one SuperProperty tag. For the SuperProperty tag name attribute is required, URI attribute is optional. If the SuperProperty tag is present, the property being created is added as a sub-property of the property listed by the SuperProperty tag, otherwise, it is added as a top-level property. The Domain tag contains a list of classes that will be added as the domain of the property. The name attribute of the Domain tag is a required property where as the uri attribute is optional. The Range tag contains either a list of classes or a

DataType tag depending upon the type of the property that is to be added. If the type of property is “data”, then the Range tag should contain the DataType tag. The DataType tag encloses the data type for the ranges of data values. Currently following data type values are supported and can be enclosed in DataType tag.

- Numeric – integer, float, decimal, nonPositive, nonnegative.
- String – string, token, language.
- Boolean – Boolean
- URI – anyURI
- Time – dateTime

If any other string is enclosed other than the above mentioned, an error is returned. If the type of the property is “object”, the Range tag encloses a list of Class tag. These classes are added as the range of the property being added. 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.

```
PUT
Request:
Resource:  ontService/{ontologyName}/property/{propertyName}
Content Type:    application/xml
Body:

<Property name="P1" uri= http://serverAddress/#P1” type=””/>

      <SuperProperty name =”SuperProp1
                    uri=”http://serverAddress/#SuperProp1”/>

     <Domain>
           {<Class name=”Class2”
                   uri=”http://serverAddress/#Class2” />…}
     </Domain>

     <Range>
           {<Class name=”Class3”
```

```
                    uri="http://serverAddress/#Class3" />…}
            <DataType>datatype</DataType>
      </Range>

</Property>
Response:
Status
Success:
    • 201  "Property created"
Errors:
    • 404 " no such property exist".
    • 404 "super property does not exist".
    • 404 "domain class does not exist".
    • 404 "range class does not exist".
    • 404 "name attribute was missing".
    • 404 "type attribute was missing".
    • 404 " requested ontology is not loaded"
```

Example:

```
PUT ontService/wine/property/hasNewDrink


<Property n      ame="hasNewDrink" type="object"/>



     <SuperProperty name="hasDrink"/>

      <Domain>

          <Class name="MealCourse" />

      </Domain>

      <Range>

         <Class name="MealCourse" />
      </Range>

</Property>
```

c) Updating properties

This requests, updates a property. The update service allows only updating of the domain and range of the property. Updating the type of the property is not permitted.

The information for the property that has to be updated is passed as request body in xml format as described below. The Property tag includes the property that is to be updated. The name attribute is required in the Property tag whereas the URI attribute is optional. The Property tag includes Domain and Range tags. The Domain tag includes a Class tag that mentions the domain class for this property that has to be updated and the Class tag contains an Update tag including the domain class with which this property will be updated. The Range tag can contain a Class tag or a DataType tag, depending on the type of the property that is being updated. The Class tag includes the range class for this property that has to be updated and it encloses an Update tag mentioning the range class with which this property has to be updated. Similarly the DataType encloses an Update tag that includes the data type value that is to be added as the range value for the property. Please refer to "creating a property", to check for supported data type values. The Update tag should mention a class that is already present in the ontology, new class will not be created.

```
POST
Request:
Resource:  ontService/{ontologyName}/property/{propertyName}
Content Type:    application/xml
Body:

<Property name="P1" uri= http://serverAddress/#P1" />


  <Domain>
       <Class name="Class1" uri="http://serverAddress/#Class1" />
            <Update name="M1" uri=http://serverAddress/#M1"  />
        </Class>
  </Domain>

  <Range>
       <Class name="Class1" uri="http://serverAddress/#Class1" />
            <Update name="M1" uri=http://serverAddress/#M1"  />
        </Class>
```

```
  </Range>

 </Property>

Response:
Status
Success:
    • 200  “Ok” Requested property updated

Errors:
    • 404 “ no such property exist”
    • 404 “domain class does not exist”
    • 404 “range class does not exist”
    • 404 “name attribute was missing”
    • 404 “ requested ontology is not loaded”
```

Example:

```
POST ontService/wine/property/hasNewDrink

<Property name="hasNewDrink" type="object"/>

      <Range>
            <Class name=”MealCourse” >
                  <Update name="DessertCourse" />
            </Class>
      </Range>

</Property>
```

d) Delete a property

This request removes the property included in the URL from the ontology.

DELETE
Resource: `ontService/{ontologyName}/property/{propertyName}`

Response:
Status
Success:
- 200 “Ok” Requested property deleted

Error:

- 404 " requested ontology is not loaded"

Example

```
DELETE ontService/wine/property/hasNewDrink
```

3) Sub-Service for interacting with instances of a class

This set of services provide an interface to browse, add or delete any class's instance from the requested ontology. The client application sends a URL that has the ontology name and the name of the class. All the above-mentioned operations are performed for the instances of this class.

a) Accessing instances of a class

This request retrieves instances of the class included in the URL. If the class included in the URI does not exist in the ontology, an error with appropriate HTTP status code is returned to the client application.

```
GET
Request:
Resource:  ontService/{ontologyName}/instancesOf/{className}
Response:
Content Type:    application/xml
Body:

<Class name="Class1" uri= http://serverAddress/#Class1" />

      <Instances>
            {<Instance name="Inst1"
                        uri="http://serverAddress/#Inst1" />…}
      </Instances>
```

```
</Class>

Error:
   • 404 " requested ontology is not loaded".
   • 404 "requested class does not exist".
```

Example:

```
GET ontService/wine/instanceOf/WineBody

<Class name="WineBody"
      uri="http://krono.act.uji.es/Links/ontologies/
                  wine.owl#WineBody" />

      <Instances>
            <Instance name="Light"
                  uri="http://krono.act.uji.es/Links/ontologies/
                  wine.owl#Light"/>
            <Instance name="Medium"
                  uri="http://krono.act.uji.es/Links/ontologies/
                  wine.owl#Medium"/>
            <Instance name="Full"
                  uri="http://krono.act.uji.es/Links/ontologies/
                  wine.owl#Full"/>
      </Instances>
</Class >
```

b) Creating an instance for a class

This service adds an instance for the class included in the URI. Information for the class for which the instance is to be added, is provided using the request body encoded in XML. While adding the instance, the client application can also specify the values for the properties of the class. The format of request body is as shown below. For the class included in the URI, only one instance is added per request. The Class tag should include the name of the class for which the new instance will be added. The Class tag contains one Instance tag. The Instance tag should include the name attribute whereas

URI attribute is optional. The Instance tag includes a list of Property tags. The Property tag requires name, value and type attributes whereas the URI attribute is optional. The name attribute includes the name of the property, the type attribute includes the type of the property and the value attribute includes the value, this property will have for this instance.

```
PUT
Request:
Resource:  ontService/{ontologyName}/instancesOf/{className}
Content Type:    application/xml
Body:

<Class name="Class1" uri= http://serverAddress/#Class1" />
  <Instance name ="Inst1" uri="http://serverAddress/#Inst1>
       <Properties>
          {<Property name="p1"
                    uri=http://serverAddress/#prop1 value="v1"
                    type="object"/>…}
       </Properties>
      <Instance>

</Class>

Response:
Status
Success:
    • 201 "Requested instance created".
Errors:
    • 404 "no such class exist".
    • 404 "property does not exist".
    • 404 "instance already present".
    • 404 "value class does not exist".
    • 404 "name attribute was missing".
    • 404 "type attribute was missing".
    • 404 "requested ontology is not loaded"
```

Example

```
PUT ontService/wine/instanceOf/class/AmericanWine

      <Class name="AmericanWine" />

            <Instance name=”newWine”  >
                  <Property name=”locatedIn”
                              type=”object”
                              value=”USRegion”/>
            </Instance>

      </Class>
```

c) Delete instance of a class**.**

This operation is not permitted for this resource. Please refer the services explained at #4 Handling instances, deleting an instance.

4) Sub-Service for interacting with instances.

This set of services provides functionality to delete instances from an ontology. The request returns the response body encoded in XML in following format. The Instance tag includes the information of the instance name included in the URI.

a) Access an instance.

```
GET
Request:
Resource:  ontService/{ontologyName}/instance/{instanceName}
Response:
Content Type:    application/xml
Body:

<Instance name="Class1" uri= http://serverAddress/#Inst1” />

      <Classes>
            {<Class name=”Inst1”
                    uri=”http://serverAddress/#Class1” />…}
      </Classes>
</Instance >
```

```
Errors:
    • 404 “instance does not exist “
    • 404 “ontology does not exist”
```

Example:

```
GET ontService/wine/instance/newWine

<Instance name="newWine"
      uri="http://krono.act.uji.es/Links/ontologies/
                  wine.owl#newWine” />

      <Classes>
            <Class name=”AmericanWine”
                  uri="http://krono.act.uji.es/Links/ontologies/
                  wine.owl#AmericanWine"/>
      </Classes>
</Instance >
```

b) Creating an instance

This operation is not permitted for this resource. Please refer the service explained at #3

Handling instances of a class, for creating an instance.

c) Delete an instance

This request removes the instance included in the URL from the ontology.

DELETE
Request:
Resource: `ontService/{ontologyName}/instance/{instanceName}`
Response:
Status
Success:

- 200 “Ok” Requested instance deleted

Error:

- Ontology is not loaded -> 404 “ requested ontology is not loaded”

Example

```
DELETE ontService/wine/instance/newWine
```

5) Sub-Service for interacting with restrictions of a class

This set of services provide an interface to browse, add or delete any class's restrictions from the requested ontology. The client application sends a URL that has the ontology name and a class name. All the above-mentioned operations are performed on the restrictions of this class.

a) Accessing restrictions of a class

This request retrieves restrictions of all the class whose name is included in the resource URI of the request. If the class does not exist in the ontology, an error with appropriate HTTP status code is returned to the client application. For the class included in the URL, a Class tag is returned. The Class tag contains a list of Restriction tags. Each Restriction tag has a type attribute, which mentions the type of the attribute. The values that type attribute can contain are `allValuesFrom`, `someValuesFrom`, `hasValue`, `maxCardinality` and `minCardinality`. The Restriction tag contains the Property tag, which mentions the property on which the restriction is. The Property tag encloses either a Value tag or a Class tag depending upon the type of the restriction. If the restriction type is value restriction, then Class tag is present where as if the restriction type is cardinality restriction then Value tag is present.

GET

Request:
Resource: ontService/{ontologyName}/restrictionsFor/{className}
Response:
Content Type: application/xml
Body:

```
<Class name="Class1" uri= "http://serverAddress/#Class1" />

      <Restrictions>
          {<Restriction type="someValuesFrom">
              <Property name="p1" uri=http://serverAddress/#p1/>
               <Value>val</Value>
                <Class name="C1" uri=http://serverAddress/#C1/>
           </Restriction>…}
      </Restrictions>

  </Class>
```

Errors:

- 404 "class does not exist".
- 404 "requested ontology is not loaded"

Example:

```
GET ontService/wine/restrictionsFor/Juice

<Class name="Juice" uri="http://krono.act.uji.es/Links/ontologies/
                          food.owl#Juice" />
    <Restrictions>
      <Restriction type="minCardanility">
           <Property name="madeFromFruit"
                    uri="http://krono.act.uji.es/Links/ontologies/
                           food.owl#madeFromFruit"/>
            <Value>1</Value>
      </Restriction>
     </Restrictions>

  </Class>
```

b) Creating restriction for a class

This service adds restrictions for the classes. Following is the format for the request body encoded in XML. For the class for which restriction is to be added a Class tag

mentioning the name of the class is required. The Class tag has a list of Restriction tags. Every Restriction requires having the type attribute. The type attribute mentions the type of the restriction. Only allowed values for type attribute are `allValuesFrom`, `someValuesFrom`, `hasValue`, `maxCardinality` and `minCardinality`. The Restriction tag encloses Property tag, which mentions the property on which the restriction applied. The Property tag encloses either a Value tag or a Class tag depending upon the type of the restriction. If the restriction is of value constraint type, the Class tag is required mentioning the name of class whereas if the restriction is of cardinality restriction the Value tag is required enclosing the value.

```
PUT
Request:
Resource:  ontService/{ontologyName}/restrictionsFor
Content Type:    application/xml
Body:

<Class name="Class1" uri= http://serverAddress/#Class1" />

       <Restrictions>
           {<Restriction type="someValuesFrom">
              <Property name="p1" uri=http://serverAddress/#p1/>
               <Value>val</Value>
              <Class name="C1" uri=http://serverAddress/#C1/>
           </Restriction>…}
       </Restrictions>
</Class>

Response:
Status
Success:
    • 201  "All restrictions created".
Errors:
    • 404 "no such class exist".
    • 404 "property does not exist".
    • 404 "value class does not exist".
    • 404 "name attribute was missing".
    • 404 "type attribute was missing".
    • 404 "requested ontology is not loaded"
```

Example:

```
PUT ontService/wine/restrictionsFor/Juice

<Class name="Juice" uri="http://krono.act.uji.es/Links/ontologies/
                          food.owl#Juice" />
    <Restrictions>
      <Restriction type="minCardanility">
            <Property name="hasSugar"
                     uri="http://krono.act.uji.es/Links/ontologies/
                            wine.owl#hasSugar"/>
             <Value>1</Value>
      </Restriction>
     </Restrictions>

  </Class>
```

c) Delete a restriction for a class.

This operation is not permitted for this resource. Please refer #6 Handling restriction of a class on a property.

6) Sub-Service for interacting with restrictions of a class on a property.

This set of services provides the utility to interact with the restrictions of a class on a particular property.

a) Accessing restriction

This operation is not permitted for this resource. Please refer #6 Handling restrictions for a class to access restrictions.

b) Creating restrictions

This operation is not permitted for this resource. Please refer #6 Handling restrictions for a class to create restrictions

c) Deleting restriction

This service deletes a restriction for the class and the property whose names are included in the resource URI. If either the class or the property is not present in the ontology included in the URI, an error with appropriate status code is returned to the client application.

Request:
Resource: `ontService/{ontName}/restrictionOf/{clsName}/{propName}`
Response:
Status
Success:

- 200 "Ok" Restriction deleted.

Errors:

- 404 "requested ontology is not loaded"
- 404 "class does not exist"
- 404 "property does not exist"

Example:

```
DELETE ontService/wine/restrictionsOf/Juice/hasSugar
```

5.3 Navigation Sub-Service

The OWL ontology is a directed labeled [4] graph with concepts as the nodes and properties being the edges. Ontology navigation is used to get a zoomed in view of a node (classes or instances) of interest. This navigation through ontology is analogous to graph traversal. Popular navigation tools such as OntoGraf [40], Jambalaya [41] provides a click-expand-navigate approach. In this approach, the user clicks on a node which is the starting point and this node is expanded into one or multiple neighboring nodes, then the user clicks on one of

the newly rendered node which then expands, and so on. User does this process iteratively till he finds his node of interest. Effectively, the user navigates from the starting node along the edges to a destination node. This ordered set of edges is the path from the starting node to the destination. Such a path naturally fits into the URI format and the output achieved after navigating along this path can be viewed as path to a resource in URI terminology. The Navigation service provides a REST web service interface to embed such a path into HTTP request URI and process it to return the destination nodes as the response. The URI for invoking the navigational service is

<Starting node>/<forward slash delimited edges, which constitute the path> ? limit={value}.

- where the Starting node can be a class or an instance.

Navigating along the path is a pipeline process, where each stage represents a navigation step along the one edge from the path. At each stage, we navigate from a set of input nodes to a set of nodes, known as output nodes, which are reachable via the corresponding edge. Output nodes from one stage are applied as input to the subsequent stage. For the first stage, the starting node is considered as the input node. Navigating in such manner can result into discovery of large number of nodes. To limit this, client application can provide a query parameter called limit in the URI. The final set of nodes will be limited to the limit included in as the query parameter by the client application.

Following is the request format for invoking this service and brief explanation about how it works.

Request syntax:

path/{ontologyName}/**class/{className}/{property}$^{+}$?limit={value}**

`path/{ontologyName}/`**`instance/{individualName}/{property}`**`+?limit={value}`

`{className}/{instance}` provides us the starting point for navigation. We than go on navigating through the ontology along the **property path** specified by the associations/properties in the HTTP request.

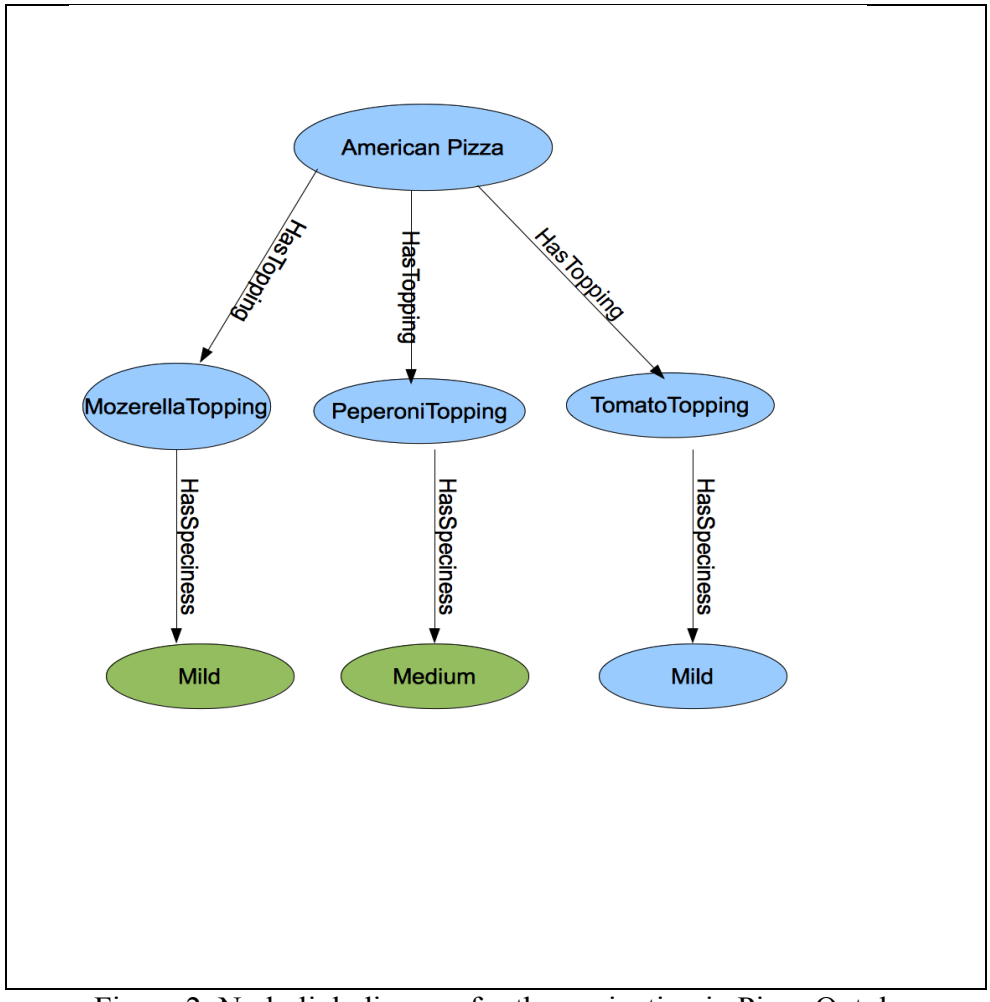

Figure 2: Node-link diagram for the navigating in Pizza Ontology

The navigation starts with the indicated class or instance. The request URI specifies using the keyword class or instance if the starting point is class or an instance. We navigate using the first property mentioned to all the classes associated with our starting point class and we collect them as interim result set. Then we navigate using the second property mentioned in the path from each of the class from the interim results from the first query and replace the current contents of the interim results with the newly explored classes. We continue navigation in such a manner till we have processed the path or any property/association from the path resulted into an empty interim result. To make things more clear consider an example from the pizza ontology.

path/pizza/class/AmericanPizza/hasTopping/hasSpeciness?limit=5

Here the URI specifies that our starting point in this case, `AmericanPizza` is a class. We first locate the `AmericanPizza` in the `pizza` ontology. Then we use the first property specified in the path query, which is `hasTopping` and navigate along this property to get all the classes that are associated to `AmericanPizza` using the `hasTopping` property and add them into the interim result set. In this example after navigating from `AmericanPizza` using `hasTopping` property, we get `PeperoniTopping` and `MozzarellaTopping` classes.

AmericanPizza/hasTopping/ => [PeperoniTopping, MozzarellaTopping]

We then apply the second property in the path query (`hasSpiciness`) on each `PeperoniTopping` and `MozzarellaTopping` and navigate from each of these classes along the property `hasSpeciness`.

[PeperoniTopping, MozzarellaTopping]/hasSpiciness => [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 or instance (starting point) and then 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 such as `subClassOf`, `superClassOf`, `instancesOf`, `equivalntClasses`, `disjointClasses`, `complementClasses`.

The response of this request depends upon the navigational path. If the result of navigation results into a set of classes, then all the information of the class is returned encoded in XML as shown below.

```
GET
Request:
Resource:
path/{ontName}/class/{className}/{prop} ?limit={value}
Or
path/{ontName}/instance/{instanceName}/{prop}+?limit={value}
Response:
```

```
Content Type:    application/xml
Body:

<Classes>
 {<Class name="Class1" uri= http://serverAddress/#Class1" />

   <SuperClasses>
     {<SuperClass name="SuperClass1"
               uri="http://serverAddress/#SuperClass1" />…}
    </SuperClasses>

    <SubClasses>
     {<SuperClass name="SuperClass1"
               uri="http://serverAddress/#SuperClass1" />…}
    </SubClasses>

    <Properties>
     {<Property name="Prop1"
               uri="http://serverAddress/#Prop1" />…}
    </Properties>

    <Instances>
     {<Instance name="Inst1"
              uri="http://serverAddress/#Instance1" />..}
     </Instances>
  </Class>…}
</Classes>
```

If the result of the navigation is set of instances, then the response encoded in XML has following format.

```
<Instance name="Class1" uri= http://serverAddress/#Inst1" />

      <Classes>
            {<Class name="Inst1"
                     uri="http://serverAddress/#Class1" />…}
      </Classes>
</Instance >
```

Errors:

- Class does not exist -> 404 “no such class exist”.
- Ontology is not loaded -> 404 “requested ontology is not loaded”

5.4 SPARQL Query Sub-Service

This sub-service provides an interface to execute SPARQL queries. We have come up with a novel approach to expose SPARQL execution service as a REST web-service (which adheres to REST principles). To execute one SPARQL query and get the result set, the client application has two send two separate HTTP requests. The client application "posts" the SPARQL query that is to be executed as a request body encoded in XML, in the format explained below. The HTTP POST request creates the result set resource on the server. The server sends back the URI to the created result set and a time parameter as a response to the first request. The result set resource is cached on the server for a time period equivalent to time parameter included in the response of the POST request. To retrieve the result set, the client application has to send a HTTP GET request with the URI of the result set within the time period. The server removes the result set after the time period has elapsed. If the client application requests for the result set after the time period has elapsed, an error with appropriate HTTP status code is sent back. This service accepts a SPARQL query in form the request body encoded in XML. The XML format to invoke this service is explained below. The response returned to the client is encoded in XML and is in following format. The Result tag includes URI and TimeOut tags. The URI tag notifies the client application the URI for the result set produced by the execution of the query requested. The TimeOut tag notifies the client application for how much time the result set will be cached on the server. The client application will have to send a HTTP GET request passing the identifier before the timeout time has elapsed to get the result from the server. The time mentioned by the timeout attribute is in milliseconds.

Following is the description of the URI design of this sub-service

a) Execute SPARQL query

The URI template to invoke this service is

```
sparqlService/{ontologyName}
```

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

```
POST
Request
Resource:  sparqlService/{ontologyName}
Content Type:    application/XML
Body:

<Query>
      { SPARQL Query }
</Query>
```

```
Response
Content type:    application/XML
Body:

<ResultSet>

      <URI> http://serverAddress/resultSetIdentifier </URI>
      <TimeOut> xyz </TimeOut>

</ResultSet>
```

b) Accessing the result of SPARQL query

This service provides the client with the result set of the SPARQL query that the client application previously executed. The client application has to invoke this service using

the URI included in the response of the HTTP POST request that the client application sent to execute the query.

GET

Request

Resource

`URI included in the response of the POST request`

Response

The format of the response provided by this service depends upon the format the client application requested in the HTTP header of the request. More precisely, the server performs content negotiation and returns the result set in the format the client application mentioned in the Accept field of the HTTP request header.

CHAPTER 6

IMPLEMENTATION

This chapter describes the implementation of the RIO server.

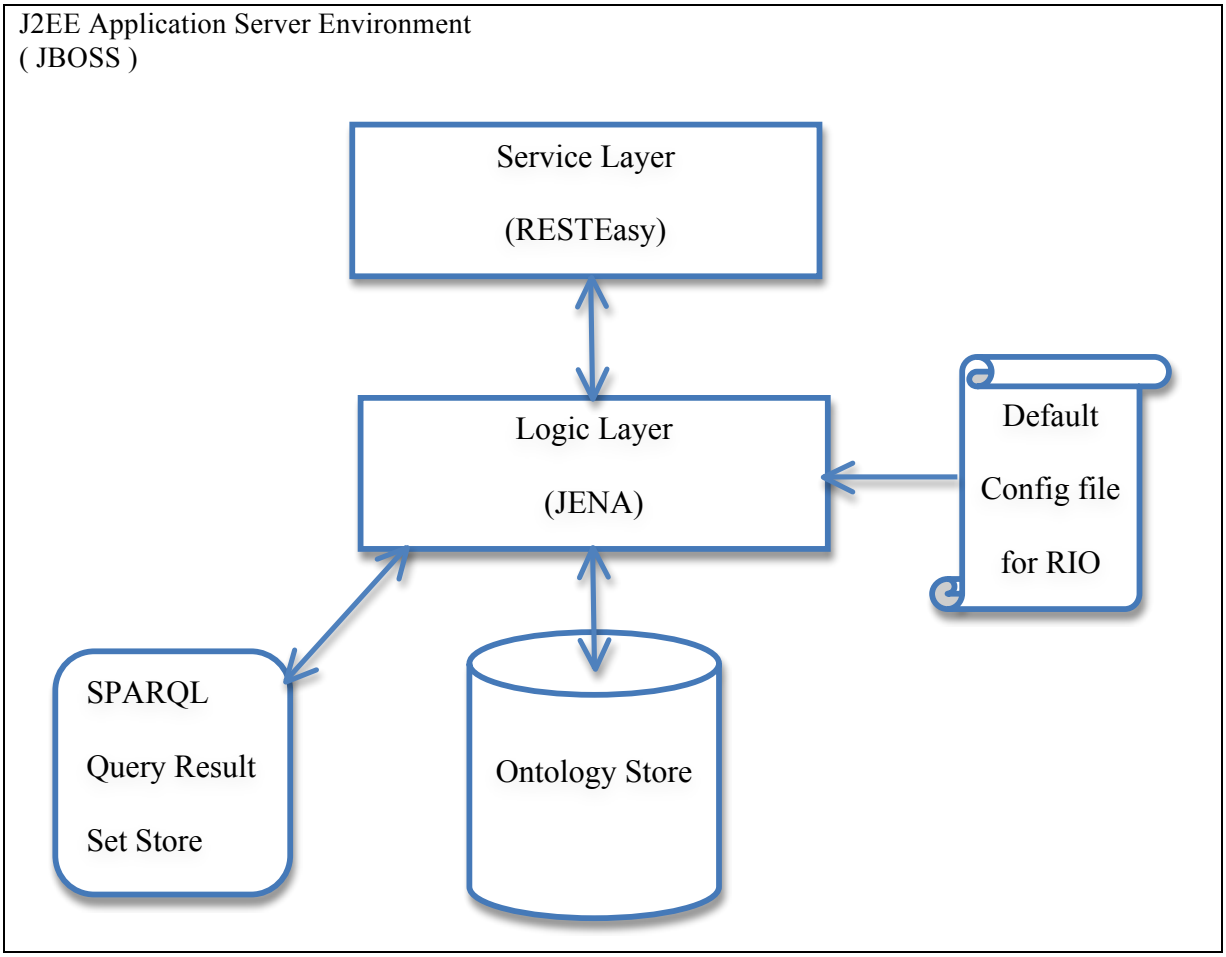

Figure 3: Architecture of RIO

The ontology server is a J2EE web server with the capability of processing RESTful requests. Implementation of RIO can be divided into logical layers as shown in Figure 3.

The service layer handles RESTful request. The logic layer interacts with the ontology store. The ontology store is a main-memory storage where all the ontologies currently loaded in RIO are present. The default configuration file contains the details of the ontologies that are to be loaded during server startup. The SPARQL query result set store, caches SPARQL query result sets temporarily. Following is a brief overview of each component of RIO.

The service layer is implemented using JBOSS's RESTEasy [42], which is a framework for developing RESTful Java web services. RESTEasy is an open source software distributed under Apache Software License 2.0. RESTEasy is a full certified and portable implementation of the JAX-RS specification [43]. JAX-RS is the Java Community Process specification released in 2008. It provides a Java API for RESTful web services over the HTTP protocol. As mentioned in the System Design chapter the ontology server provides four types of services. They are implemented as Java interfaces and are called OntologyManagementService, OntologyService, NavigationService and SPARQLService. Each service provides a RESTful API to interact with the ontologies loaded in the web server. These services are implemented using JBOSS's RESTEasy [42], which is a framework for developing RESTful Java web services.

Every Java service resource is mapped to a unique URI. For example a URI `/ontoService` represents the `OntologyService` 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 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. They also validate the request and check if it adheres to the format expected by the API. If not, an error is returned to the client along with appropriate HTTP status code. The message writer on the other hand wraps the result of the invoked service 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.

The logic layer is implemented using JENA semantic web toolkit. During the server startup, logic layer reads the default configuration file. The default configuration file contains details of the ontologies, which are to be loaded into the ontology store. RIO currently hosts all the ontologies in main memory. Each ontology is loaded into memory using JENA Java API without any semantic reasoning capability support. We made this design decision, as most of the services exposed by RIO do not need any semantic reasoning, except the validate service (refer Chapter #5). We use in-built reasoners provided by JENA API for the validate service. When any client application invokes the validate service, logic layer, converts the current in-memory model of the requested ontology into a inferred in-memory model using JENA's built-in OWL micro reasoners [44] . This inferred model is used by the validate service to check for any inconsistencies in the ontology.

As discussed in Chapter # 5, we have a novel way of executing SPARQL queries in RESTful way. The result set created after a successful execution of a SPARQL query is temporarily cached in the SPARQL query result set store. The time for which the result sets will be cached is set at server startup through the configuration file, and this time is called

`resultSetTimeOut`. As soon as a result set is present in the result set store for `resultSetTimeOut` time, that result set is evicted from the cache.

CHAPTER 7

EXPERIMENTS & EVALUATION

To evaluate RIO, we deployed it in JBOSS application server (version 5.1). We configured RIO to load four ontologies by default, namely Pizza ontology, Wine ontology, GlycO ontology and ReactO ontology. We used tools such as FireFox add-on Poster [45] and unix shell utility – CURL [46] that can create HTTP requests and parse HTTP response to test all the services provided by RIO.

Following are the tests that we did with the wine ontology.

- Ontology Management sub-service.

  We loaded the wine ontology by uploading the OWL file of wine ontology using the load service from ontology management sub-service. Also tested the save ontology service by serializing the wine ontology into an OWL file.

- Ontology sub-service.

  We tested this sub-service using the wine ontology. For testing various services provided by this sub-service, we created a class, added some properties to it, added few restrictions to it, and added an instance to the newly created class. We created the class newAmericanWine and added it as the sub-class of `AmericanWine` class. We created `AmericanRedWine` as its instance. We also performed tests to retrieve various classes, properties, and instances from the wine ontology.

- Navigation sub-service.

We tested the navigation service using the example explained in the Chapter #5 (under Navigation sub-service). We loaded the pizza ontology into RIO. We used `http://om.cs.uga.edu/rio/path/American/hasTopping/hasSpiciness` as the URI to test navigation within the pizza ontology with the `American` class as the starting point. We also used the GlycO ontology to test the navigation service provided by RIO.

- SPARQL Query sub-service

For testing SPARQL queries we used wine and the pizza ontologies. We submitted various queries with different values in `Accept` header of the client request to test the content negotiation feature provided by the navigation service.

Apart from these tools, we developed a client application in JAVA called `OntoStat.java` for testing the services provided by RIO. We programmed the client application to gather statistics of number of concepts present in any ontology that is loaded in RIO. The client application is developed using RESTEasy client framework [47]. `OntoStat` accepts the name of the ontology from command line, whose statistics are to be calculated. It then sends RESTful requests to RIO to retrieve all classes, properties and instances for that particular ontology and provides a count of each of these concepts.

RIO uses JENA API for interacting with the ontologies. Hence performance of RIO depends upon performance of JENA framework. Currently all the ontologies loaded in RIO are in-memory models. Needless to say, this has the drawback of reliability. During unfortunate events like server crashes, the modifications done to any ontology loaded in the server will be lost. To overcome, this problem, we have provided a service, which can serialize any

ontology loaded in RIO to an OWL file (as discussed in Chapter 5). The client applications can utilize this service to create a snapshot of any ontology loaded in RIO.

RIO loads all the ontologies without any inferencing support. So no schema validations with inferencing are executed, for every operation performed on the ontology using RIO's services. To overcome this, RIO provides a validate service as described in Chapter #5 (under Ontology Management Sub-Service section.) This service returns a detailed report of any present inconsistencies in the ontology. RIO uses the validity checking provided by JENA framework, to validate the model. The client application then can use modification services provided by the Ontology sub-service to rectify any inconsistencies that are present. Currently we have not performed any performance evaluation of RIO. We plan to do so in future.

CHAPTER 8

CONCLUSION AND FUTURE WORK

This thesis demonstrated a new way of building an ontology server and exposing its functionality using REST architecture principles. We successfully demonstrated a novel way of programmatic ontology navigation in a RESTful way. In this thesis, we also successfully demonstrated a unique way of implementing a SPARQL endpoint that adheres to REST architecture principles. We also displayed how operations such as ontology management, ontology modification and ontology editing can be done using REST web-services. RIO server demonstrates how REST web-services are a natural fit to provide an interface for interacting with ontologies. We can have ontology independent client application such as ontology visualizers and browsers built using RIO. Due to RIO, any client application that has the capability of sending/receiving HTTP request/response can leverage the advantages of ontologies.

Currently RIO has rich but basic set of functionalities. RIO server can be extended to add following features.

8.1 Persistent Storage for Ontologies

Currently all the ontologies loaded in the server are in-memory models. Persistent triple store support is currently not provided by the server. The current server implementation can

be easily extended to support persistent triple store for storing ontologies. Due to the modular design of the ontology server, adding this support won't be a tedious task.

8.2 Regular Expression Support

Currently the navigational queries do not accept regular expressions. We plan to add regular expression in future. Some work related to infinite loops would be required to add support of regular expressions to navigational queries.

8.3 Support RDF/XML format

The ontology server currently accepts request in JSON and XML format. And all the responses to the client are encoded in XML or JSON format. In future we plan to support RDF/XML format.

8.4 Performance Evaluation

At the time of writing this thesis, we did not perform any performance evaluation of the services provided by RIO. In future we plan to evaluate the performance of the various services provided by RIO.

REFERECES

1. Gruber, T.R., *A translation approach to portable ontology specifications.* Knowledge acquisition, 1993. **5**(2): p. 199-220.
2. Brickley, D. and R.V. Guha, *Resource Description Framework (RDF) Schema Specification 1.0: W3C Candidate Recommendation 27 March 2000.* 2000.
3. McGuinness, D.L. and F. Van Harmelen, *OWL web ontology language overview.* W3C recommendation, 2004. **10**.
4. Hayes, J. and C. Gutierrez, *Bipartite Graphs as Intermediate Model for RDF* *The Semantic Web – ISWC 2004*, S. McIlraith, D. Plexousakis, and F. van Harmelen, Editors. 2004, Springer Berlin / Heidelberg. p. 47-61.
5. Lee, T.B., J. Hendler, and O. Lassila, *The semantic web.* Scientific American, 2001. **284**(5): p. 34-43.
6. *RDF Schema*. Available from: http://www.w3.org/TR/rdf-schema/.
7. *OWL, Web Ontology Language*. Available from: http://www.w3.org/TR/owl-features/.
8. *OWL Language Reference*. Available from: http://www.w3.org/TR/owl-ref/.
9. 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.
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. Eklund, P., N. Roberts, and S. Green. *Ontorama: Browsing rdf ontologies using a hyperbolic-style browser*. in *Proc. First International Symposium on Cyber Worlds (CW.02), IEEE Computer Society*. 2002. IEEE.

12. Pan, J., S. Cranefield, and D. Carter, *A lightweight ontology repository*, in *Proceedings of the second international joint conference on Autonomous agents and multiagent systems*2003, ACM: Melbourne, Australia. p. 632-638.

13. Harrison, R. and C.W. Chan. *Distributed ontology management system*. in *18th Annual Canadian Conference on Electrical and Computer Engineering Saskatoon, IEEE*. 2005. IEEE.

14. Reinberger, M.L. and P. Spyns, *STAR Lab Technical Report.* STAR, 2004. **2004**(16): p. 16.

15. Li, Y., et al., *Beyond Ontology Construction; Ontology Services as Online Knowledge Sharing Communities*

The Semantic Web - ISWC 2003, D. Fensel, K. Sycara, and J. Mylopoulos, Editors. 2003, Springer Berlin / Heidelberg. p. 469-483.

16. *SOAP Version 1.2*. Available from: http://www.w3.org/TR/soap12-part1/.

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

18. 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.

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

20. Erling, O. and I. Mikhailov, *RDF Support in the Virtuoso DBMS.* Networked Knowledge-Networked Media, 2009: p. 7-24.

21. Janik, M. and K. Kochut, *BRAHMS: A WorkBench RDF Store and High Performance Memory System for Semantic Association Discovery*

The Semantic Web – ISWC 2005, Y. Gil, et al., Editors. 2005, Springer Berlin / Heidelberg. p. 431-445.

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

23. *SPARQL Query Language for RDF*. 2008.

24. *SPARQL Query Results XML Format*. 2008; Available from: http://www.w3.org/TR/rdf-sparql-XMLres/.

25. *SPARQL Protocol for RDF*. 2008; Available from: http://www.w3.org/TR/rdf-sparql-protocol/.

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

27. 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.

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

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

30. Bairoch, A., et al., *The Universal Protein Resource (UniProt)*. Nucleic Acids Research, 2005. **33**(suppl 1): p. D154-D159.

31. Thomas, C.J., A.P. Sheth, and W.S. York, *Modular Ontology Design Using Canonical Building Blocks in the Biochemistry Domain*, in *Proceeding of the 2006 conference on Formal Ontology in Information Systems: Proceedings of the Fourth International Conference (FOIS 2006)*2006, IOS Press. p. 115-127.

32. Gosal, G.P.S., *ProKinO: Design and Development of Ontology on Protein Kinases*, 2010, University of Georgia.

33. Tania Tudorache, J.V., and Natalya F. Noy, *Web-Prot eg e: A Lightweight OWL Ontology Editor for the Web.* OWL Experiences and Directions Workshop (OWLED 2008), 2008. **5th**.

34. Aasman, J., *Allegro graph*, 2006, Technical Report 1, Franz Incorporated.

35. *Ontology-Browser.*

36. Bozsak, E., et al., *KAON — Towards a Large Scale Semantic Web*

E-Commerce and Web Technologies, K. Bauknecht, A. Tjoa, and G. Quirchmayr, Editors. 2002, Springer Berlin / Heidelberg. p. 231-248.

37. Horridge, M., et al., *A Practical Guide To Building OWL Ontologies Using The Protégé-OWL Plugin and CO-ODE Tools Edition 1.0.* The University Of Manchester, 2004.

38. Bonomi, A., et al., *NavEditOW – A System for Navigating, Editing and Querying Ontologies Through the Web*

Knowledge-Based Intelligent Information and Engineering Systems, B. Apolloni, R. Howlett, and L. Jain, Editors. 2007, Springer Berlin / Heidelberg. p. 686-694.

39. *Wine Ontology*. Available from: http://krono.act.uji.es/Links/ontologies/wine.owl/view.

40. *OntoGraf*. Available from: http://protegewiki.stanford.edu/wiki/OntoGraf.

41. *Jambalaya*. Available from: http://www.thechiselgroup.org/jambalaya.

42. *RESTEasy*. Available from: http://www.jboss.org/resteasy.

43. Pericas-Geertsen, S. and M. Potociar, *JAX-RS: Java™ API for RESTful Web Services*. 2011.

44. *JENA, OWL Micro Reasoner*. Available from: http://jena.sourceforge.net/javadoc/com/hp/hpl/jena/reasoner/rulesys/OWLMicroReasoner.html.

45. *FireFox add-on Poster*. Available from: https://addons.mozilla.org/en-US/firefox/addon/poster/.

46. *CURL*. Available from: http://www.unix.com/man-page/Linux/1/curl/.

47. *RESTEasy Client Framework*. Available from: http://docs.jboss.org/resteasy/2.0.0.GA/userguide/html/RESTEasy_Client_Framework.html.