by

(Under the Direction of )

ABSTRACT

INDEX WORDS:

by

\*This is only for degrees previously earned! Please do not include your major with the degree name, and list the degree simply as BA, BS, MA, etc.

\*International Students must include the name of the country between the school and the date the degree was received, if it was received outside of the US.

**Click on the borders of this box, then hit the delete key to make it disappear.**

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

OF

ATHENS, GEORGIA

©

All Rights Reserved

by

Major Professor:

Committee:

Electronic Version Approved:

Maureen Grasso

Dean of the Graduate School

The University of Georgia

DEDICATION

This page is optional. You do not have to provide a dedication or acknowledgements for your thesis or dissertation. You may write whatever you would like to on this page. If you are not going to use the acknowledgements page, be sure to delete it up to the section break at the end of this section.

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 #

#

#

If you need more chapters or subheadings in any chapter, simply copy a line with a blank subheading, and paste it where you need it. (To select a whole line, move your mouse pointer to the left of the line until the pointer becomes a white arrow pointing towards the text, then click and the whole line should get selected). This also applies if you need more tables or figures in the respective lists.

\*Please do not list more than just the first subheading of each chapter.

**Select the line the box appears on by clicking in the left margin to highlight it, and then hit the delete key to make the box disappear.**

2 #

#

#

3 #

#

#

#

4 #

#

#

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

ENTER TITLE HERE

Please begin the beginning of your body text on this page (if already created in another document, you may simply copy and paste the appropriate information on this page, remembering to check that your first line of text meets the 1.75 inch top margin [i.e. appears on the third double spaced line of the page]).

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

CHAPTER 3

SYSTEM DESIGN

Before describing the web-services supported by our system 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. It is implemented using JBOSS’s RESTEasy, which is a framework for developing RESTful Java web services. It 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.

The service layer contains Java classes that have annotations binding methods to specific URI patterns and HTTP operations. We have utilized JAX-RS specified features such as parameter injection annotations to map the information mentioned in the HTTP methods to the Java class methods. To keep the marshaling and un-marshaling of request/response data decoupled from the Java objects, this 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.

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 uses the Jena Semantic Web toolkit to interact with the ontologies loaded in the web server.

Jena Semantic Web toolkit is open source software released under BSD license. It is a Java framework for building Semantic Web applications. It provides a programmatic environment to build applications using Semantic Web technologies like RDF, RDFs, OWL and SPARQL. The Jena framework includes API for reading and writing RDF, RDF/XML, OWL, N3, N-triples. It provides in-memory and persistent storage for ontologies. Jena framework also includes a SPARQL engine to query RDF data.

The Logic Layer utilizes Jena’s framework for manipulating OWL files. It uses

CHAPTER 4

**IMPLEMENTATION**

This chapter describes the design and implementation of the RESTful web services that are defined in the Service Layer. The system has four web services namely OntologyService, SchemaService, NavigationService and SPARQLService. Each service provides complete RESTful API to interact with the ontologies loaded in the web server. Each service methods have RESTEasy defined JAX-RS annotations to bind HTTP operations to them. Following are the details of each of the service.

**4.1 OntologyService**

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 an ontology as a OWL file.

SchemaService

{ontologyName}/schemaService

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

**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*

/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

**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>

CHAPTER 5

EXPERIMENTS AND RESULTS