SPARQL Gateway Specification V0.3

Arthur Ryman
Last Modified: 2012-07-12

Status

This document is under development. The API information described in this document is provisional and subject to change.

Revision History

- V0.2.0 Completed RDF vocabulary, resource formats, and REST API for gw:Gateway, gw:EndpointList, gw:QueryList, gw:Endpoint, gw:Query, and gw:Variable.
- V0.2.1 Added copyright notice.
- V0.3 Added specification for data service resource navigation from the gw:Gateway resource. Updated RDF namespace to http://jazz.net/ns/reporting/sparqlgateway#

Introduction

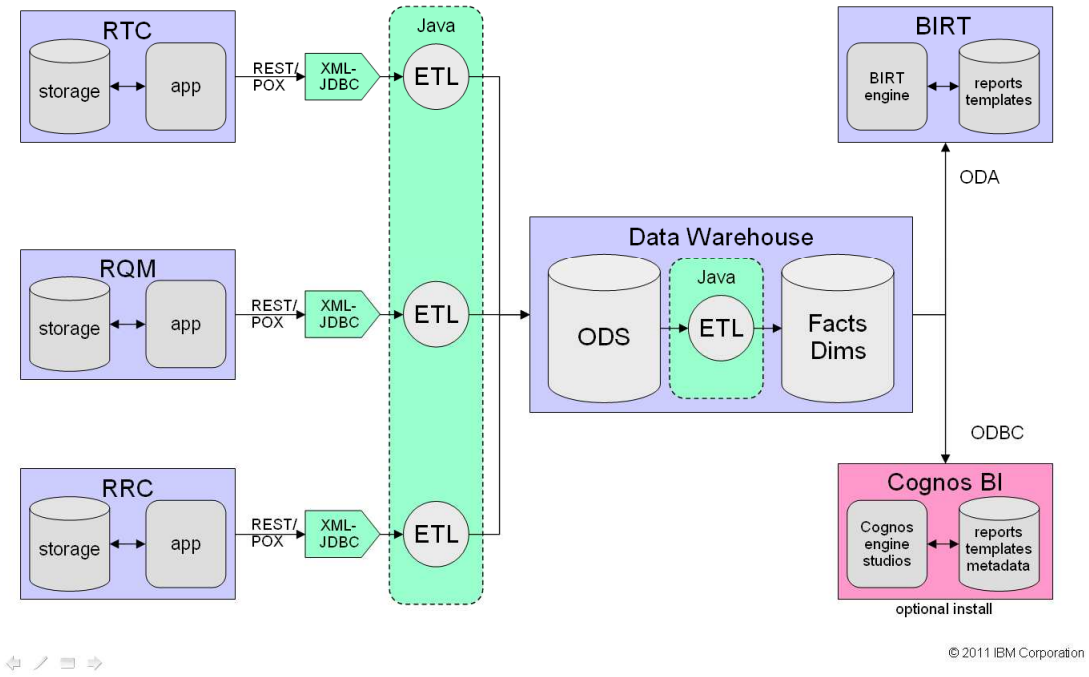
This specification describes a new component that is part of a system for reporting on data that is held in software development tools. Although suited to reporting on software development artifacts such as requirements, defect reports, and test plans, it can be used for many similar types of artifacts, e.g. those from systems engineering.

Reporting can be viewed as a three-tiered system. Tier 1 is the Application tier which consists of the software development tools. These tools are the sources of data that users want to see reports about. Tier 2 is the Integration tier which collects the data from the Application tier and stores it in a format that can be efficiently queried. Tier 3 is the Presentation tier which sends queries to the Integration tier and renders the results as reports that users can view.

Our “As is” XML-based Reporting Architecture

Rational currently has a reporting architecture that uses REST APIs and Plain Old XML (POX) representations in the Application tier and traditional data warehousing using relational databases and SQL in the Integration tier. The Presentation tier consists of the Cognos BI and Eclipse BIRT reporting engines.

The following diagram illustrates this architecture for the Rational Collaborative Lifecycle Management (CLM) 3.0.1 release as of June 2011. The Application tier contains three tools: Rational Team Concert (RTC), Rational Quality Manager (RQM), and Rational Requirements Composer (RRC). The Integration tier contains Java ETLs and the data warehouse. The Presentation tier contains BIRT, Cognos BI, and Rational Publishing Engine (not shown).



The tools in the Application tier implement a REST API that conforms to the Rational Reportable REST specification. Knowledge of Reportable REST is required for an understanding of the SPARQL Gateway. See [REPORTABLE-REST] for further information.

This reporting architecture takes advantage of highly mature database and reporting technologies, and it can execute cross-product analytical queries efficiently. However, data warehouses are complex to implement and maintain, and they do not support low-latency traceability queries. We believe can improve this situation by using Linked Data technology.

Our “To be” RDF-based Reporting Architecture

This document describes the SPARQL Gateway, which is a software component that enables reporting on Linked Data sources. Linked Data is the foundational technology that Rational has selected for integrating artifacts across the development lifecycle.

With Linked Data, development tools in the Application tier are web applications that expose their artifacts as web resources, i.e. each development artifact is identified by an HTTP URI. Linked Data uses Resource Description Framework (RDF) as the data model for describing resources. A client of a development tool can therefore request an RDF representation of any development artifact via an HTTP GET request that contains an Accept header for an RDF media type, e.g. use application/rdf+xml for RDF/XML. The

representations of many development artifacts from many tools can be stored in an RDF database known as a triple store which integrates the data and enables queries across all the data using the powerful RDF query language SPARQL. Knowledge of SPARQL is required for an understanding of this specification. See [SPARQL] for more information.

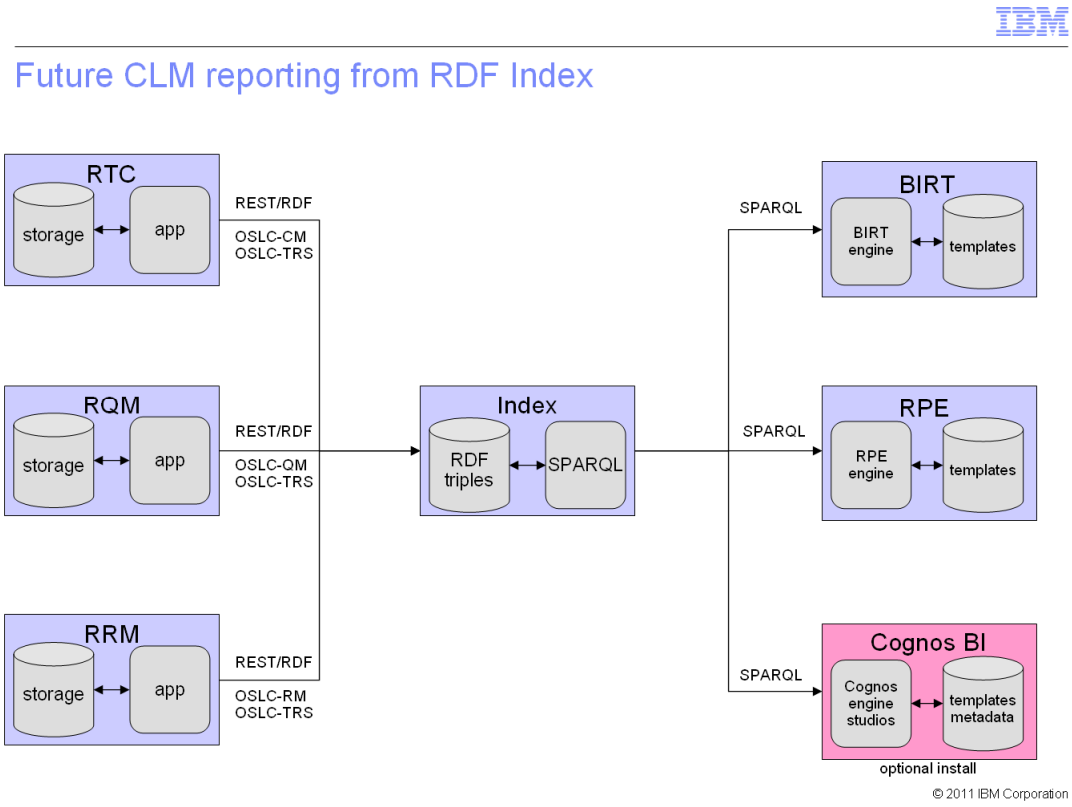
Rational is specifying RDF vocabularies for many development domains at Open Services for Lifecycle Collaboration (OSLC) so that data from tools that support a given domain can be represented in a uniform way.

Rational is also specifying a new REST API at OSLC known as Tracked Resource Set (OSLC-TRS) that enables tools to expose change activity on their resources. OSLC-TRS is like a news feed for development artifacts that contains information about creation, modification, and deletion events. A knowledge of OSLC-TRS in NOT required for an understanding of this specification

Rational is developing an RDF indexer that uses OSLC TRS to efficiently index sets of development resources into an RDF triple store where they can be queried using SPARQL. The indexer, triple store, and SPARQL endpoint become the new Integration tier.

The following diagram illustrates the future RDF-based reporting architecture. In this example, the Application tier contains RTC, RQM, and RRC. However, they now implement OSLC domain specifications, RTC implements OLSC Change Management (OSLC-CM), RQM implements OSLC Quality Management (OSLC-QM), and RRC implements OSLC Requirements Management (OSLC-RM). In addition, each tool implements OSLC-TRS. The Integration tier contains the index. The index containers and indexing process that uses the OSLC-TRS API for each tool to determine which resource to load into the triple store. The triple store provides access to its content via a SPARQL endpoint. The Presentation tier contains BIRT, Cognos, and Rational Publishing Engine (RPE). RPE is a document generation tool. In the future architecture, each engine in the Presentation tier sends SPARQL queries to the index to get data, and then renders it as reports or documents.

Currently BIRT, Cognos, and RPE do not have the ability to generate SPARQL queries. Cognos does have a prototype which we've tested, but there is no plan to make it a product. Rational has begun to design SPARQL support in RPE, but again this is in the early stages.

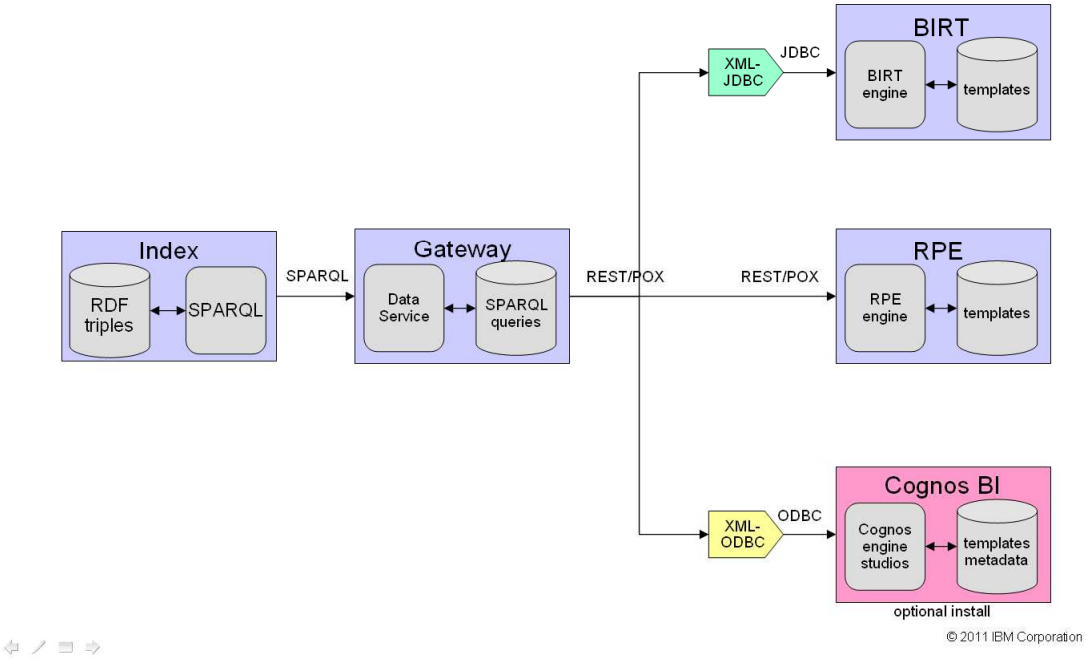


The problem is then how can we use our existing Presentation tier engines with this new SPARQL-based Integration tier? The answer is to develop a SPARQL Gateway that mediates between the SPARQL endpoint and the traditional reporting engines.

Overview of the SPARQL Gateway

The main idea behind the gateway is that it makes a SPARQL query look like a Reportable REST Data Service. The following diagram illustrates the architecture of the Integration tier with the gateway. The gateway creates a Reportable REST data service each SPARQL deployed into it. RPE can consume this type of data service directly. BIRT and Cognos can consume this data service via the Insight XML JBBC/ODBC driver, which requires the definition of an XDC file that maps the XML data to relational data.

SPARQL gateway



The gateway is a web application. We'll assume that the gateway is implemented as a Java servlet, although any similar web technology can be used.

The gateway provides a REST API and a web UI for managing all of its resources. These include SPARQL endpoints and SPARQL queries. The REST API and web UI support the creation, modification, deletion and retrieval of the resources managed by the gateway. The web UI delegates actions to the REST API.

In the following discussion, I'll use a suggested URI pattern for queries, endpoints, and associated resources. However, client applications should not assume any particular pattern. Instead, all URIs should be discoverable, either from a service description document for the gateway, or from links in RDF representations of the gateway resources. The contents of the RDF representations will be given later.

The gateway has a base URI which represents the gateway application as a whole. An HTTP GET request should return useful information about the gateway and links to its main resources. The response should be either HTML or RDF, depending on the Accept header. If HTML is requested, then the gateway should provide a very simple user interface for managing the gateway and accessing its resources. Here is the gateway base URI:

```
http://example.org/gateway
```


The gateway manages a set of SPARQL endpoints. Each endpoint has a URI that supports the SPARQL protocol. The gateway manages other descriptive information about each endpoint, e.g. a title and description, so that uses can easily select the endpoint that they want to query. Here is the URI for the set of endpoint descriptions:

```
http://example.org/gateway/endpoint
```

An HTTP GET request on the endpoint set URI gives a list of the endpoint descriptions. An HTTP POST to this URI creates a new endpoint description. Here is the URI of an endpoint description:

```
http://example.org/gateway/endpoint/3
```

An endpoint description is retrieved, modified, and deleted by sending HTTP GET, PUT, and DELETE requests to its URI.

The gateway manages a set of SPARQL queries. These may be written by hand or generated by some other tool. The gateway assigns each query a URI which supports the Reportable REST API. Here is the URI for the set of queries:

```
http://example.org/gateway/query
```

An HTTP GET request on the query set gives a list of the queries. An HTTP POST to this URI creates a new query. Here is the URI of a query:

```
http://example.org/gateway/query/42
```

A query is retrieved, modified, and deleted by sending HTTP GET, PUT, and DELETE requests to its URI.

The gateway can execute the query and return the SPARQL results with no transformation. The function is useful for testing. The URI that returns the SPARQL results for query 42 is:

```
http://example.org/gateway/query/42/sparqlresults
```

The primary function of the gateway is to make the query behave as a Reportable REST data service. The URI for the data service defined by query 42 is:

```
http://example.org/gateway/query/42/dataservice
```

The gateway manages some metadata about each SPARQL query. This metadata may be inferred by parsing the SPARQL query or it may be provided explicitly with the SPARQL query or by the administrator. For simplicity, let's assume that the SPARQL query is a SELECT query. The metadata includes the names and datatypes of the columns of the result set.

The gateway uses this metadata to generate an XML Schema to describe the Reportable REST result. It also uses this metadata to generate other artifacts as described below. The URI of the XML schema that describes the Reportable REST data service for query 42 is:

```
http://example.org/gateway/query/42/dataservice/xsd
```


When the gateway receives an HTTP request on a Reportable REST URI associated with a SPARQL query, it performs the following functions:

1. The gateway parses the query parameters on the request URI. These query parameters are in the XPath syntax defined by the Reportable REST specification. The query parameters specify filtering conditions.
2. The gateway transforms the XPath filtering conditions into SPARQL graph patterns and adds then to the WHERE clause of the SPARQL query.
3. The gateway sends the modified SPARQL query to the SPARQL endpoint.
4. The gateway receives the result of the SPARQL query.
5. The gateway transforms the result of the SPARQL query into an XML format that is conformant with the Reportable REST specification and returns it in the HTTP response.

This describes the main behavior of the gateway.

A Reportable REST data source can be directly consumed by Rational Publishing Engine. It can also be consumed by the Rational Insight XML JDBC/ODBC driver if an XML Data Configuration (XDC) file is provided. The gateway can generate an XDC file. Similarly the gateway can generate a Cognos Framework Manager model and a Cognos report template. These artifacts can be deployed into the reporting environment to enable the generation of a default report. The deployment of these artifacts is outside the scope of the gateway. The gateway generates these associated artifacts to eliminate the manual steps normally required for consuming a data source, thus improving the consumability of the solution. Here are the URIs of the XDC file, Framework Manager model, and Cognos report definitions for the data service of query 42:

```
http://example.org/gateway/query/42/dataservice/xdc
http://example.org/gateway/query/42/dataservice/fm
http://example.org/gateway/query/42/dataservice/report
```

Example

This section is a simple defect reporting example to illustrate the operation of the gateway.

RDF Data

For simplicity, let's consider a set of facts about defects. Each defect is a web resource and therefore has a URI. In this example, the defect URIs are of the form http://example.org/bug/1234. Each defect has a short description, a severity, and a creation date.

We'll use dcterms:title as the predicate for the short description where dcterms: is the prefix for the Dublin Core terms vocabulary. The value of the short description is a literal string.

We'll use exbugs:severity as the predicate for the severity where exbugs: is the prefix for http://example.org/bugs#. Severity values are integers from 1 to 5.

We'll use dcterms:created as the predicate for the creation date. Creation date values are date-times as defined by XML Schema.

Here is some sample data in Turtle format:

```
@prefix dcterms: <http://purl.org/dc/terms/> .
@prefix exbugs: <http://example.org/bugs#> .
@prefix xsd: <http://www.w3.org/2001/XMLSchema#> .

<http://example.org/bug/1234>
      dcterms:title "NPE in parser" ;
      dcterms:created "2011-06-23T12:00:00"^^xsd:dateTime ;
      exbugs:severity 3 .

<http://example.org/bug/1235>
      dcterms:title "Out of memory in web UI" ;
      dcterms:created "2011-06-24T15:00:00"^^xsd:dateTime ;
      exbugs:severity 1 .

<http://example.org/bug/1236>
      dcterms:title "Sales tax calculation is wrong" ;
      dcterms:created "2011-06-24T15:30:00"^^xsd:dateTime ;
      exbugs:severity 4 .
```

SPARQL SELECT Query

The following sample SPARQL query returns the defect data as a 4-column table:

```
PREFIX dcterms: <http://purl.org/dc/terms/>
PREFIX exbugs: <http://example.org/bugs#>

SELECT ?defect ?created ?severity ?title
      WHERE {
            ?defect
                  dcterms:title ?title ;
                  dcterms:created ?created ;
                  exbugs:severity ?severity
      }
```

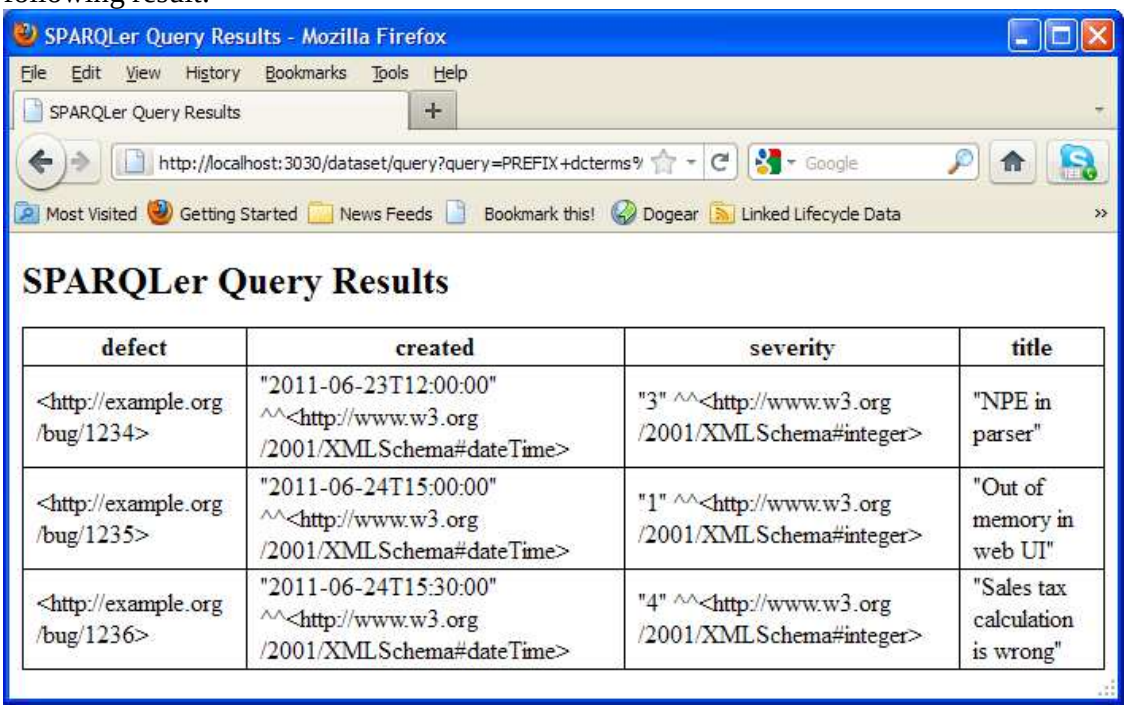
SPARQL SELECT Query Result

The result is a table with 3 rows and 4 columns. The SPARQL specification defines an XML format for the result of a SELECT query.

You can set up a SPARQL endpoint using the Fuseki server. See [FUSEKI] for more information on how to stand up a Fuseki server. Start up the server using, e.g., the following command:

```
java -Xmx1200M -jar fuseki-sys.jar  --update --mem /dataset
```


Open a web browser on the server using the URI http://localhost:3030/, go to the control panel, select the dataset, load the sample data, and run the sample query. You will get the following result:



SPARQLer Query Results

defect	created	severity	title
<http://example.org /bug/1234>	"2011-06-23T12:00:00" ^^<http://www.w3.org /2001/XMLSchema#dateTime>	"3" ^^<http://www.w3.org /2001/XMLSchema#integer>	"NPE in parser"
<http://example.org /bug/1235>	"2011-06-24T15:00:00" ^^<http://www.w3.org /2001/XMLSchema#dateTime>	"1" ^^<http://www.w3.org /2001/XMLSchema#integer>	"Out of memory in web UI"
<http://example.org /bug/1236>	"2011-06-24T15:30:00" ^^<http://www.w3.org /2001/XMLSchema#dateTime>	"4" ^^<http://www.w3.org /2001/XMLSchema#integer>	"Sales tax calculation is wrong"

This HTML table is generated by applying /xml-to-html.xsl, an XSLT stylesheet, to the XML result of the SPARQL SELECT query. This XML format is defined by the SPARQL specification. It has content type application/sparql-results+xml. Here is the result as XML:

```
<?xml version="1.0"?>
<sparql xmlns="http://www.w3.org/2005/sparql-results#">
  <head>
    <variable name="defect"/>
    <variable name="created"/>
    <variable name="severity"/>
    <variable name="title"/>
  </head>
  <results>
    <result>
      <binding name="defect">
        <uri>http://example.org/bug/1234</uri>
      </binding>
      <binding name="created">
        <literal
datatype="http://www.w3.org/2001/XMLSchema#dateTime">2011-06-
23T12:00:00</literal>
      </binding>
      <binding name="severity">
        <literal
datatype="http://www.w3.org/2001/XMLSchema#integer">3</literal>
      </binding>
```



```
      <binding name="title">
        <literal>NPE in parser</literal>
      </binding>
    </result>
    <result>
      <binding name="defect">
        <uri>http://example.org/bug/1235</uri>
      </binding>
      <binding name="created">
        <literal
datatype="http://www.w3.org/2001/XMLSchema#dateTime">2011-06-
24T15:00:00</literal>
      </binding>
      <binding name="severity">
        <literal
datatype="http://www.w3.org/2001/XMLSchema#integer">1</literal>
      </binding>
      <binding name="title">
        <literal>Out of memory in web UI</literal>
      </binding>
    </result>
    <result>
      <binding name="defect">
        <uri>http://example.org/bug/1236</uri>
      </binding>
      <binding name="created">
        <literal
datatype="http://www.w3.org/2001/XMLSchema#dateTime">2011-06-
24T15:30:00</literal>
      </binding>
      <binding name="severity">
        <literal
datatype="http://www.w3.org/2001/XMLSchema#integer">4</literal>
      </binding>
      <binding name="title">
        <literal>Sales tax calculation is wrong</literal>
      </binding>
    </result>
  </results>
</sparql>
```

Reportable REST Data Service

The main purpose of the gateway is to create a Reportable REST data service based on the SPARQL query so that it can be consumed by our reporting engines. The gateway must therefore generate an XML result format that conforms to the requirements of the Reportable REST specification.

Note that since Reportable REST is based on XML, it is very suitable for SPARQL SELECT queries. Similarly, since the OSLC Reporting Profile is based on RDF, it is very suitable for SPARQL CONSTRUCT queries. For now we'll focus on SELECT queries.

The XML format of the SELECT result is not suitable for a Reportable REST service. Instead, the SELECT format should be transformed to the following:


```
<?xml version="1.0"?>
<results>
	<result>
		<defect>http://example.org/bug/1234</defect>
		<created>2011-06-23T12:00:00</created>
		<severity>3</severity>
		<title>NPE in parser</title>
	</result>
	<result>
		<defect>http://example.org/bug/1235</defect>
		<created>2011-06-24T15:00:00</created>
		<severity>1</severity>
		<title>Out of memory in web UI</title>
	</result>
	<result>
		<defect>http://example.org/bug/1236</defect>
		<created>2011-06-24T15:30:00</created>
		<severity>4</severity>
		<title>Sales tax calculation is wrong</title>
	</result>
</results>
```

This simpler format is closer to what a human designer would create. It has that advantage that each result column name becomes an element name. This makes it easier for humans to create XPath expressions that pull out the data and make it usable for inclusion in documents and reports. In addition, we can create an XML Schema (xsd) document that defines the structure of this simpler format and allows the datatypes to be validated by an XSD validating parser.

Here is an XSLT file that transforms the SPARQL results into the Reportable REST format:

```
<?xml version="1.0" encoding="UTF-8"?>
<!--
	This stylesheet transforms the output of a SPARQL SELECT query to
a form compatible with
	Reportable REST.
 -->
<xsl:stylesheet version="1.0"
	xmlns:xsl="http://www.w3.org/1999/XSL/Transform"
xmlns:sparql="http://www.w3.org/2005/sparql-results#"
	exclude-result-prefixes="sparql">

	<xsl:output method="xml" indent="yes"></xsl:output>

	<xsl:template match="/sparql:sparql/sparql:results">
		<results>
			<xsl:apply-templates
select="sparql:result"></xsl:apply-templates>
		</results>
	</xsl:template>

	<xsl:template match="sparql:result">
```

```
            <result>
                  <xsl:apply-templates
select="sparql:binding"></xsl:apply-templates>
            </result>
      </xsl:template>

      <xsl:template match="sparql:binding">
            <xsl:element name="{@name}">
                  <xsl:value-of select="*"></xsl:value-of>
            </xsl:element>
      </xsl:template>

</xsl:stylesheet>
```

The XSLT file can be applied to the SELECT format using the Java Transformation API for XML (TrAX). See the Eclipse Web Tools Platform book for an example.

XSLT is a suitable implementation technology if the documents being transformed are not too large. For very large documents, it is more appropriate to use a streaming technology such as the Java Streaming API for XML (StAX).

Data Service XML Schema

A Reportable REST data service should also provide metadata in the form of an XML schema (XSD) that describes the data. This metadata is requested by appending the following query string onto to the data service URI: metadata=schema. For example, here is the URI to request the XSD for the data service defined by query 42:

```
http://…/query/42/dataservice?metadata=schema
```

When the gateway receives this request it should redirect the response to the following URI:

```
http://…/query/42/dataservice/xsd
```

In general, it is not possible to infer the XML schema datatypes associated with the variables in a SPARQL SELECT query so the gateway allows them to be explicitly specified. In addition, the gateway allows a descriptive label for each variable to be specified. This descriptive label is added to the XML schema as appinfo using a convention that is specified by Reportable REST. This label information is used by authoring tools to improve the user experience. In our example, supposed the query includes the following metadata:

Name	Type	Label
Defect	xsd:anyURI	Link
Created	xsd:dateTime	Creation Date
Severity	xsd:integer	Severity
Title	xsd:string	Summary

Here is the XML schema that describes the data service for query 42:

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



```
<xsd:schema xmlns:xsd="http://www.w3.org/2001/XMLSchema">

      <xsd:element name="results">
            <xsd:complexType>
                  <xsd:sequence>
                        <xsd:element ref="result" minOccurs="0"
maxOccurs="unbounded"></xsd:element>
                  </xsd:sequence>
            </xsd:complexType>
      </xsd:element>

      <xsd:element name="result">
            <xsd:complexType>
                  <xsd:sequence>
                        <xsd:element ref="defect"></xsd:element>
                        <xsd:element ref="created"></xsd:element>
                        <xsd:element ref="severity"></xsd:element>
                        <xsd:element ref="title"></xsd:element>
                  </xsd:sequence>
            </xsd:complexType>
      </xsd:element>

      <xsd:element name="defect" type="xsd:anyURI">
            <xsd:annotation>
                  <xsd:appinfo>
                        <label>Link</label>
                  </xsd:appinfo>
            </xsd:annotation>
      </xsd:element>

      <xsd:element name="created" type="xsd:dateTime">
            <xsd:annotation>
                  <xsd:appinfo>
                        <label>Creation Date</label>
                  </xsd:appinfo>
            </xsd:annotation>
      </xsd:element>

      <xsd:element name="severity" type="xsd:integer">
            <xsd:annotation>
                  <xsd:appinfo>
                        <label>Severity</label>
                  </xsd:appinfo>
            </xsd:annotation>
      </xsd:element>

      <xsd:element name="title" type="xsd:string">
            <xsd:annotation>
                  <xsd:appinfo>
                        <label>Summary</label>
                  </xsd:appinfo>
            </xsd:annotation>
      </xsd:element>


</xsd:schema>
```


This XML schema has a simple structure. It contains an element definition for each variable. These element definitions use the variable name as the element name, the specified type as the element type, and the specified label as the `<xsd:appinfo>` annotation. In addition there is a root results element and a child result element.

Data Service Result Set Filtering

In general, data sources may contain very large amounts of data. The Reportable REST specification provides a mechanism that allows clients to restrict the amount of data returned in a request to the required by the client. This is accomplished by appending to the request URI a query string that contains filter conditions. The filter contain syntax uses a subset of XPath 2.0 to specify the data elements that are being tested by the condition.

In the gateway, the XML structure of the result set is very simple, so the XPath expressions are also simple. The document root element is `<results>,` which has zero or more child `<result>` elements, each of which contains a child element for each variable.

For example, suppose the client only requires the set of all bugs that have severity = 1. The XPath expression `results/result[severity='1']` selects only those result elements that contain a severity element whose value is 1. The XPath expression `results/result[severity='1']/*` selects all the child elements of only those result elements that have a severity of 1. The following URI contains a query string with this filter:

```
…/dataservice?fields=results/result[severity='1']/*
```

When the gateway receives this request, it creates a modified version of the SPARQL query that contains a FILTER clause. Here is the modified query:

```
PREFIX dcterms: <http://purl.org/dc/terms/>
PREFIX exbugs: <http://example.org/bugs#>

SELECT ?defect ?created ?severity ?title
     WHERE {
          ?defect
               dcterms:title ?title ;
               dcterms:created ?created ;
               exbugs:severity ?severity .

          FILTER ( str(?severity) = '1' )
     }
```

The gateway translated the Reportable REST query string:

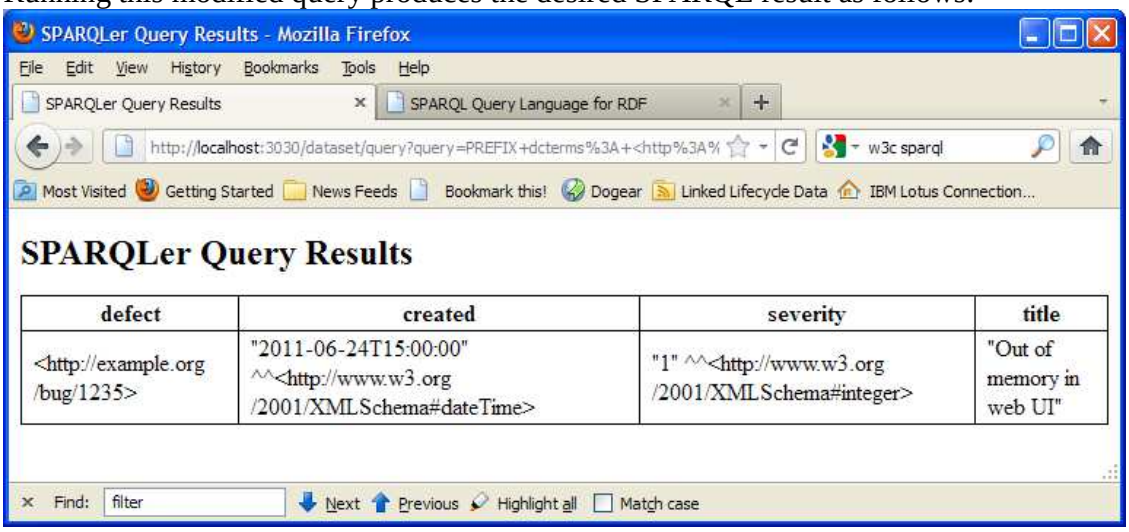
```
fields=results/result[severity='1']/*
```

into the SPARQL filter clause:

```
FILTER ( str(?severity) = '1' )
```

Note that the SPARQL `str` function is applied to the `?severity` variable in order to convert it to a string so that it can be correctly compared to the string value `'1'` passed in on the query string.

Running this modified query produces the desired SPARQL result as follows:



Data Service Resource Navigation

Section 3.4 of the Reportable REST specification [REPORTABLE-REST] describes how data service resources may be discovered by navigation from a root data service resource. The root data service resource for the SPARQL Gateway is the data service associated with the gateway resource, e.g.

```
http://example.com/gateway/dataservice
```

When the service receives an HTTP GET request for this resource, it should respond with an XML document of the following structure:

```
<?xml version="1.0" encoding="UTF-8"?>
<Gateway xmlns="http://jazz.net/ns/reporting/sparqlgateway/dataservice"
       xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
       xsi:schemaLocation="http://jazz.net/ns/reporting/sparqlgateway/dataservi
ce http://example.com/gateway/dataservice/xsd">
       <title>example.org SPARQL Gateway Service</title>
       <QueryList>
              <title>example.org SPARQL Query List</title>
              <Query href="http://example.com/gateway/query/1/dataservice">
                     <title>Chemical Elements in DBpedia</title>
              </Query>
              <Query href="http://example.com/gateway/query/2/dataservice">
                     <title>Unresolved Defects</title>
              </Query>
              <Query href="http://example.com/gateway/query/7/dataservice">
                     <title>Failing Test Cases</title>
              </Query>
```



```
            <Query href="http://example.com/gateway/query/8/dataservice">
                  <title>Requirements with no Test Cases</title>
            </Query>
            <Query href="http://example.com/gateway/query/14/dataservice">
                  <title>Analysis of Features by Component</title>
            </Query>
      </QueryList>
</Gateway>
```

This document lists all the data service resources associated with queries. The document also includes the titles of the gateway, the query list, and each query to aid the user. The titles are represented in RDF as the values of the dcterms:title property.

The key feature of this document is the use of href attributes which link to the actual data service resources. Clients that support Reportable REST provide a user interface that lets users select the linked data service resources. This is more user-friendly than requiring users to copy and paste URLs.

The document also links to an XML schema which the gateway makes available at the URL

`http://example.com/gateway/dataservice/xsd`

The service redirects to the above URL if the user uses the following URL with a query string:

`http://example.com/gateway/dataservice?metadata=schema`

Here is the XML schema:

```
<?xml version="1.0" encoding="UTF-8"?>
<schema xmlns="http://www.w3.org/2001/XMLSchema"
      xmlns:tns="http://jazz.net/ns/reporting/sparqlgateway/dataservice"
      targetNamespace="http://jazz.net/ns/reporting/sparqlgateway/dataservice"
>
      <element name="Gateway" type="tns:GatewayType"></element>

      <element name="QueryList" type="tns:QueryListType"></element>

      <element name="Query" type="tns:QueryType"></element>

      <element name="title" type="string"></element>

      <complexType name="GatewayType">
            <sequence>
                  <element ref="tns:title" maxOccurs="1"
minOccurs="1"></element>
                  <element ref="tns:QueryList" maxOccurs="1"
minOccurs="1"></element>
            </sequence>
      </complexType>

      <complexType name="QueryListType">
            <sequence>
```

```
                    <element ref="tns:title" maxOccurs="1"
minOccurs="1"></element>
                    <element ref="tns:Query" minOccurs="0"
maxOccurs="unbounded"></element>
             </sequence>
       </complexType>

       <complexType name="QueryType">
             <sequence>
             <element ref="tns:title" maxOccurs="1" minOccurs="1"></element>
             </sequence>
             <attribute name="href" type="anyURI" use="required"></attribute>
       </complexType>

</schema>
```

SPARQL Gateway RDF Vocabulary

The preceding discussion introduced many types of resources, their properties and their relations with other resources. In this section I define a set of RDF vocabulary terms for these concepts. These RDF vocabulary terms are used in the resource representations that are supported by the REST API of the gateway.

The gateway returns an RDF representation of a resource when it receives an HTTP GET request that contains an Accept header with an RDF content type. All gateway implementations must support RDF/XML which has the content type application/rdf+xml. Gateways should also support Turtle which has the content types text/turtle and application/x-turtle.

All of the RDF vocabulary terms defined here are part of the namespace

`http://jazz.net/ns/reporting/sparqlgateway#`

which we abbreviate as `gw.` The following prefixes are used throughout this document:

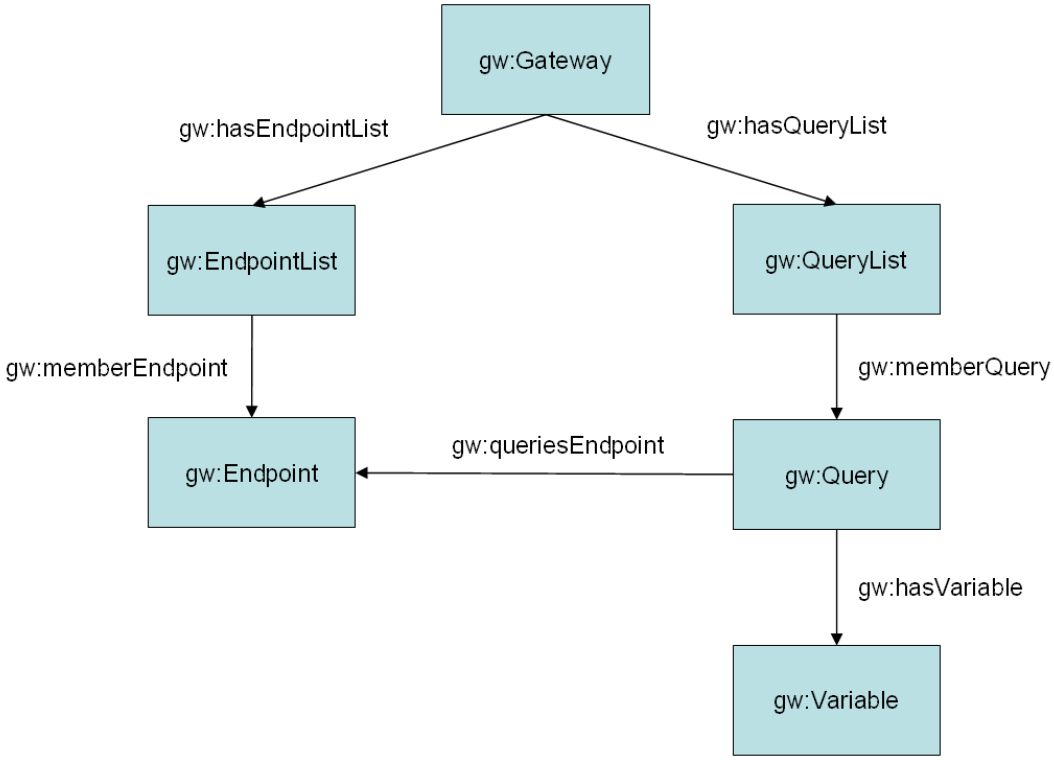
Prefix	URI	Description
gw:	http://jazz.net/ns/reporting/sparqlgateway#	SPARQL Gateway
rdf:	http://www.w3.org/1999/02/22-rdf-syntax-ns#	Resource Description Framework
dcterms:	http://purl.org/dc/terms/	Dublin Core Terms
xsd:	http://www.w3.org/2001/XMLSchema#	XML Schema Datatypes
exbugs:	http://example.org/bugs#	Example bug vocabulary

RDF Classes

The following classes are used by the gateway:

Class	Description
gw:Gateway	A SPARQL gateway service.
gw:EndpointList	A list of SPARQL endpoint descriptions.
gw:QueryList	A list of SPARQL query definitions.
gw:Endpoint	A SPARQL endpoint description.
gw:Query	A SPARQL query definition.
gw:Variable	A SPARQL query variable definition.

The following diagram illustrates the main relations between these classes.



The properties of these classes are described below.

Note that there are also some common properties for keeping track of the creation and modification dates, the creator, etc. These are omitted for brevity. A production quality implementation of the gateway should implement these properties, manage authentication and access control, etc.

gw:Gateway

The root resource of the gateway has class gw:Gateway. In our example, the following URI has class gw:Gateway:

```
http://example.org/gateway/service
```

Here is an example RDF representation of it:

```
@prefix dcterms: <http://purl.org/dc/terms/> .
@prefix gw: <http://jazz.net/ns/reporting/sparqlgateway#> .

@base <http://example.org/gateway/> .

# the gateway service
<service> a gw:Gateway ;
     dcterms:title "example.org SPARQL Gateway Service" ;
     dcterms:description "This service lets reporting tools
at example.org access SPARQL endpoints." ;
     gw:hasEndpointList <endpoint> ;
     gw:hasQueryList <query> ;
     gw:hasDataService <dataservice> ;
     gw:hasDataServiceXsd <dataservice/xsd> .
```

A gw:Gateway resource has the following properties:

Property	Value	Description
dcterms:title	literal plain text or XHTML	The brief, one-line description of this gateway. This property is modifiable via the REST API.
dcterms:description	literal plain text or XHTML	The description of this gateway. This property is modifiable via the REST API.
gw:hasEndpointList	URI of the gw:EndpointList resource	The list of endpoint descriptions managed by this gateway. This property is system-generated and immutable.
gw:hasQueryList	URI of the gw:QueryList resource	The list of query definitions managed by this gateway. This property is system-generated and immutable.
gw:hasDataService	URL	The URL of the Reportable REST V1 data service associated with the gateway. This URL acts as the root for navigation to the data services associated with queries. This property is system-generated and immutable.
gw:hasDataServiceXsd	URL	The URL of the XML Schema for the data service associated with the gateway. This property is system-generated and immutable.

gw:EndpointList

The list of all endpoint resources managed by the gateway service has class gw:EndpointList. In our example, the endpoint list has the following URI:

```
http://example.org/gateway/endpoint
```

Here is an example RDF representation of it:

```
@prefix dcterms: <http://purl.org/dc/terms/> .
@prefix gw: <http://jazz.net/ns/reporting/sparqlgateway#> .

@base <http://example.org/gateway/> .

# the endpoint list
<endpoint> a gw:EndpointList ;
     dcterms:title "example.org SPARQL Endpoint List" ;
     dcterms:description "This is the list of SPARQL
endpoints that reporting tools at example.org can access."
;
     gw:nextIdentifier "7" ;
     gw:inGateway <service> ;
     gw:memberEndpoint <endpoint/1> ;
     gw:memberEndpoint <endpoint/4> ;
     gw:memberEndpoint <endpoint/5> .
```

A gateway service always has exactly one endpoint list. It has the following properties:

Property	Value	Description
dcterms:title	literal plain text or XHTML	The brief, one-line description of the endpoint list. This property is modifiable via the REST API.
dcterms:description	literal plain text or XHTML	The description of the endpoint list. This property is modifiable via the REST API.
gw:nextIdentifier	literal plain text	The identifier that the endpoint list will assign to the next endpoint resource that it creates. Identifiers are unique within an endpoint list and are never reused. This property is system-generated.
gw:inGateway	URI of a gw:Gateway resource	The gateway that manages this endpoint list. This property is always present and occurs exactly once. This property is system-generated and immutable.
gw:memberEndpoint	URI of a gw:Endpoint resource	An endpoint resource managed by the gateway. There may be zero or more of these properties. This property is system-generated.

gw:Endpoint

In our example, the following URIs has class gw:Endpoint:

```
http://example.org/gateway/endpoint/1
```

Here is an example endpoint:

```
@prefix dcterms: <http://purl.org/dc/terms/> .
@prefix gw: <http://jazz.net/ns/reporting/sparqlgateway#> .

@base <http://example.org/gateway/> .

# endpoint 1
<endpoint/1> a gw:Endpoint ;
     dcterms:identifier "1" ;
     dcterms:title "DBpedia" ;
     dcterms:description "Infobox data collected from
wikipedia.org" ;
     gw:inEndpointList <endpoint> ;
     gw:sparqlEndpointLocation <http://dbpedia.org/sparql>
.
```

An endpoint list has zero or more endpoint resources. Each endpoint resource has the following properties:

Property	Value	Description
dcterms:title	literal plain text or XHTML	The brief, one-line description of the endpoint. This property is modifiable via the REST API.
dcterms:description	literal plain text or XHTML	The description of this endpoint. This property is modifiable via the REST API.
dcterms:identifier	literal plain text	The identifier of the endpoint. Identifiers are unique within an endpoint list and are never reused. This property is system-generated and immutable.
gw:inEndpointList	URI of a gw:EndpointList resource	The endpoint list that manages this endpoint resource. This property is always present and occurs exactly once. This property is system-generated and immutable.
gw:sparqlEndpointLocation	URL	A SPARQL endpoint URL where queries can be sent. This property is modifiable via the REST API.

gw:QueryList

In our example, the following URI has class gw:QueryList:

```
http://example.org/gateway/query
```

Here is an example RDF representation of it:

```
@prefix dcterms: <http://purl.org/dc/terms/> .
@prefix gw: <http://jazz.net/ns/reporting/sparqlgateway#> .

@base <http://example.org/gateway/> .

# the query list
<query> a gw:QueryList ;
     dcterms:title "example.org SPARQL Query List" ;
     dcterms:description "This is the list of SPARQL
queries that reporting tools at example.org can access." ;
     gw:nextIdentifier "15" ;
     gw:inGateway <service> ;
     gw:memberQuery <query/1> ;
     gw:memberQuery <query/2> ;
     gw:memberQuery <query/7> ;
     gw:memberQuery <query/8> ;
     gw:memberQuery <query/14> .
```

A gateway service always has exactly one query list. It has the following properties:

Property	Value	Description
dcterms:title	literal plain text or XHTML	The brief, one-line description of the query list. This property is modifiable via the REST API.
dcterms:description	literal plain text or XHTML	The description of the query list. This property is modifiable via the REST API.
gw:nextIdentifier	literal plain text	The identifier that the query list will assign to the next query resource that it creates. Identifiers are unique within a query list and are never reused. This property is system-generated.
gw:inGateway	URI of a gw:Gateway resource	The gateway that manages this query list. This property is always present and occurs exactly once. This property is system-generated and immutable.
gw:memberQuery	URI of a gw:Query resource	A query resource managed by the gateway. There may be zero or more of these properties. This property is system-generated.

gw:Query

In our example, the following URI has class gw:Query:

```
http://example.org/gateway/query/1
```


Here is an example RDF representation of it:

```
@prefix dcterms: <http://purl.org/dc/terms/> .
@prefix gw: <http://jazz.net/ns/reporting/sparqlgateway#> .
@prefix rdfs: <http://www.w3.org/2000/01/rdf-schema#> .
@prefix xsd: <http://www.w3.org/2001/XMLSchema#> .

@base <http://example.org/gateway/> .

# query 1
<query/1> a gw:Query ;
     dcterms:identifier "1" ;
     dcterms:title "Chemical Elements in DBpedia" ;
     dcterms:description "Queries all topics in
Category:Chemical_elements, and gets their English labels
and comments." ;
     gw:inQueryList <query> ;
     gw:queriesEndpoint <endpoint/1> ;
     gw:sparqlQuery
"""PREFIX dcterms: <http://purl.org/dc/terms/>
PREFIX dbpedia: <http://dbpedia.org/resource/>

SELECT *
WHERE {

?element dcterms:subject
<http://dbpedia.org/resource/Category:Chemical_elements> .

?element <http://www.w3.org/2000/01/rdf-schema#label>
?label .
FILTER ( lang(?label) = "en" ) .

?element <http://www.w3.org/2000/01/rdf-schema#comment>
?comment .
FILTER ( lang(?comment) = "en" ) .

} order by ?label""" ;
     gw:hasSparqlResults <query/1/results> ;
     gw:hasDataService <query/1/dataservice> ;
     gw:hasDataServiceXsd <query/1/dataservice/xsd> ;
     gw:resultsLabel "Table of Chemical Elements" ;
     gw:resultLabel "Chemical Element" ;
     gw:hasVariable <query/1#element> , <query/1#label>,
<query/1#comment> .

<query/1#element> a gw:Variable ;
     dcterms:identifier "element" ;
```



```
     gw:position 1 ;
     gw:hasDatatype xsd:anyURI ;
     rdfs:label "URI" .

<query/1#label> a gw:Variable ;
     dcterms:identifier "label" ;
     gw:position 2 ;
     gw:hasDatatype xsd:string ;
     rdfs:label "Name" .

<query/1#comment> a gw:Variable ;
     dcterms:identifier "comment" ;
     gw:position 3 ;
     gw:hasDatatype xsd:string ;
     rdfs:label "Description" .
```

A query list has zero or more query resources. Each query resource has the following properties:

Property	Value	Description
dcterms:title	literal plain text or XHTML	The brief, one-line description of the query. This property is modifiable via the REST API.
dcterms:description	literal plain text or XHTML	The description of the query. This property is modifiable via the REST API.
dcterms:identifier	literal plain text	The identifier of the query. Identifiers are unique within an query list and are never reused. This property is system-generated and immutable.
gw:inQueryList	URI of a gw:QueryList resource	The query list that manages this query resource. This property is always present and occurs exactly once. This property is system-generated and immutable.
gw:queriesEndpoint	URI of a gw:Endpoint resource	An endpoint resource where the query will be sent. This property is modifiable via the REST API.
gw:sparqlQuery	literal plain text	The SPARQL query string. This property is modifiable via the REST API.
gw:hasSparqlResults	URL	The URL of the SPARQL results in XML format for the query. This property is system-generated and immutable.
gw:hasDataService	URL	The URL of the Reportable REST V1 data service associated with the query. This property is system-generated and immutable.

gw:hasDataServiceXsd	URL	The URL of the XML Schema of the data service associated with the schema. This property is system-generated and immutable.
gw:resultsLabel	literal plain text	A human-readable label for the set of all results for the query. This property is modifiable via the REST API.
gw:resultLabel	literal plain text	A human-readable label for an individual result for the query. This property is modifiable via the REST API.
gw:hasVariable	URI of a gw:Variable resource	A SPARQL variable that is returned by the SELECT clause of the query. This property is modifiable via the REST API. The variables MUST match the SPARQL query string.

gw:Variable

In our example, the following URI has class gw:Variable:

```
http://example.org/gateway/query/1#element
```

These resources of contained in the RDF graph of gw:Query resources. See the preceding example of gw:Query for its RDF representation.

A query has one or more variables. Each variable has the following properties:

Property	Value	Description
dcterms:identifier	literal plain text	The name of the variable. This property is modifiable via the REST API. The name MUST match the variable name in the SPARQL query string.
gw:position	literal integer	The integer position of the variable within the query. The positions are unique within the query and range from 1 to the number of variables. This property is modifiable via the REST API. The positions of all variable MUST be consecutive integers starting from 1 and match the order of the variables in the SPARQL query string.
gw:hasDatatype	URI of a rdfs:Datatype resource	The XML schema datatype of the variable in the results, e.g. xsd:integer or xsd:anyURI. This property is modifiable via the REST API.
rdfs:label	literal plain text	A human-readable label for the variable. This property is modifiable via the REST API.

REST API

The following table shows the allowed REST methods on the top-level resources:

Resource	GET	PUT	POST	DELETE
gw:Gateway	yes	yes	no	no
gw:EndpointList	yes	yes	yes (gw:Endpoint)	no
gw:QueryList	yes	yes	yes (gw:Query)	no
gw:Endpoint	yes	yes	no	yes
gw:Query	yes	yes	no	yes

The following additional rules apply:

- An endpoint MUST NOT be deleted if any query refers to it.
- The SPARQL query string in a query MUST be syntactically valid.
- The variables in a query MUST exactly match the variables in the SPARQL query string.
- System-generated properties MUST NOT be included in POST requests.
- System-generated properties MAY be included in PUT requests, in which case they MUST exactly match there current values. They MAY be omitted.

References

[FUSEKI] Fuseki, http://openjena.org/wiki/Fuseki

[LINKED-DATA] Linked Data – Design Issues, http://www.w3.org/DesignIssues/LinkedData.html

[REPORTABLE-REST] Reportable Rest Services Interfaces, http://open-services.net/pub/Main/ReportingHome/Reportable_Rest_Services_Interfaces-OSLC_Submission.pdf

[SPARQL] SPARQL Query Language for RDF, http://www.w3.org/TR/rdf-sparql-query/

