ODATA2SPARQL

1/6/2020

OData2SPARQL: an OData service provider for any SPARQL endpoint.

OData2SPARQL

Revision History:

Date	Version	Description	Author
14/11/2014	1.0	Initial Document	PJL
1/6/2020	2.0	Olingo revision	PJL
1/6/2020	3.0	Query generation anatomy	PJL
25/11/2016	4.0	Added Operation configuration	PJL
15/06/2017	4.1	Added application use issues	PJL
5/06/2017	4.2	BaseType Web IDE description	PJL
18/07/2017	4.3	Added support for CRUD operation	PJL
03/08/2017	4.4	Documented file locations and default cardinality rules	PJL
22/06/2018	4.5	Added support for different text searching	PJL
12/03/2020	4.6	Upgraded to fully support CRUD operations	PJL

OData2SPARQL

Table of Contents

Figures

Tables

WHY ODATA2SPARQL?

To compare SPARQL with OData is somewhat misleading. After all SPARQL has its roots as a very powerful query language for RDF data, and is not intended as a RESTful protocol. Similarly OData has its roots as an abstract interface to any type of datastore, not as a specification of that datastore. Some have said "OData is the equivalent of ODBC for the Web".

The data management strengths of SPARQL/RDF can be combined with the application development strengths of OData with a protocol proxy: OData2SPARQL, a Janus-point between the application development world and the semantic information world.

OData and SPARQL/RDF: Contradictory or Complimentary?

We all know that information is an essential ingredient to successful execution of most jobs today. In that sense we are all information-workers. However delivering information in a suitable form such that workers can take action, and hence produce results, still presents significant difficulties to the enterprise information worker:

- Only corporate information can be accessed.
- Cannot join the dots between internal and external data.
- Ad-hoc queries across structured data often restricted to predefined reports (and hence predefined questions)

OData and SPARQL/RDF: Contradictory or Complimentary?		
	OData	**SPARQL/RDF**
Strengths	• Schema discovery • OData provides a data source neutral web service interface which means application components can be developed independently of the back end datasource. • Supports CRUD • Not limited to any particular physical data storage • Client tooling support • Easy to use from JavaScript • Growing set of OData productivity tools such as Excel, SharePoint, Tableau and BusinessObjects. • Growing set of OData frameworks such as SAPUI5, OpenUI5, and SAP WebIDE • Growing set of independent development tools such as LINQPad, and XOData • Based on open (OASIS) standards after being initiated by Microsoft • Strong commercial support from Microsoft, IBM, and SAP. • OData is not limited to traditional RDBMS applications. Vendors of real-time data such as OSI are publishing their data as an OData endpoint.	• Extremely flexible schema that can change over time. • Vendor independent. • Portability of data between triple stores. • Federation over multiple, disparate, data-sources is inherent in the intent of RDF/SPARQL. • Increasingly standard format for publishing open data. • Linked Open Data expanding. • Identities globally defined. • Inferencing allows deduction of additional facts not originally asserted which can be queried via SPARQL. • Based on open (W3C) standards
Weaknesses	• Was perceived as vendor (Microsoft) protocol • Built around the use of a static data-model (RDBMS, JPA, etc) • No concept of federation of data-sources • Identities defined with respect to the server. • Inferencing limited to sub-classes of objects	• Application development frameworks that are aligned with RDF/SPARQL limited. • Difficult to access from de-facto standard BI tools such as Excel. • Difficult to report using popular reporting tools

TABLE 1: ODATA AND SPARQL/RDF: CONTRADICTORY OR COMPLEMENTARY?

- Although JSON-LD (Manu Sporny, 2014), a method of transporting Linked Data using JSON, it focusses on the transportation of linked data (RDF) as JSON rather than defining the protocol for a RESTful CRUD interface.

Benefits of OData2SPARQL proxy server

OData as a complement to RDF/SPARQL	
Complementary	• Brings together the strength of a ubiquitous RESTful interface standard (OData) with the flexibility, federation ability of RDF/SPARQL. • Allows standard queries to be published via the OData service along with the deduced classes • SPARQL/OData Interop proposed W3C interoperation proxy between OData and SPARQL (Kal Ahmed, 2013) • Opens up many popular user-interface development frameworks and tools such as KendUI and SAP WebIDE. • Acts as a Janus-point between application development and data-sources. • User interface developers are not, and do not want to be, database developers. Therefore they want to use a standardized interface that abstracts away the database, even to the extent of what type of database: RDBMS, NoSQL, or RDF/SPARQL • By providing an OData4SPARQL server, it opens up any SPARQL data-source to the C#/LINQ development world. • Opens up many productivity tools such as Excel/PowerQuery, and SharePoint to be consumers of SPARQL data such as Dbpedia, Chembl, Chebi, BioPax and any of the Linked Open Data endpoints! • Microsoft has been joined by IBM and SAP using OData as their primary interface method which means there will many application developers familiar with OData as the means to communicate with a backend data source.

TABLE 2: ODATA AS A COMPLEMENT TO RDF/SPARQL

Contradictions between OData and RDF/SPARQL			
	Issue	**Description**	**Mitigation**
Contradictions	OData 3NF versus RDF 1NF	RDF inherently supports multiple values for a property, whereas OData up to V2 only supported scalar values	OData V3 supports collections of property values, which are supported by OData4SPARQL proxy server
	RDF Language tagging	RDF supports language tagging of strings	OData supports complex types, which are used to map a language tagged string to a complex type with separate language tag, and string value.
	DatatypeProperties versus object-attributes	OWL DatatypeProperties are concepts independent of the class, bound to a class by a domain, range or OWL restriction. Conversely OData treats such properties bound to the class.	In OData4SPARQL The OWL DatatypeProperty is converted to an OData EntityType property for each of the DatatypeProperty domains.
	Multiple inheritance	OData only supports single inheritance via the OData baseType declaration within an EntityType definition.	
	Multiple domain properties	An OWL ObjectProperty will be mapped to an OData Association. An Association can be between only one FromRole	OData Associations are created for each domain. The OData4SPARQL names these associations

		and one ToRole and the Association must be unique within the namespace.	{Domain}_{ObjectProperty}, ensuring uniqueness.
	Cardinality	The capabilities of OData V3 allow all DatatypeProperties to be OData collections. However the ontology might have further restrictions on the cardinality.	OData2SPARQL assumes cardinality will be managed by the backend triple store. However in future versions, if the cardinality is restricted to one or less, then the EntityType property can be declared as a scalar rather than a collection.

TABLE 3: CONTRADICTIONS BETWEEN ODATA AND RDF/SPARQL

OData Protocol

What is the OData protocol? OData is a standardized protocol for creating and consuming data APIs. OData builds on core protocols like HTTP and commonly accepted methodologies like REST. The result is a uniform way to expose full-featured data APIs (Odata.org). Version 4.0 is being standardized at OASIS, and was released in March 2014.

To compare SPARQL with OData is somewhat misleading. After all SPARQL has its roots as a very powerful query language for RDF data, and is not intended as a RESTful protocol. Similarly OData has its roots as an abstract interface to any type of datastore, not as a specification of that datastore. Some have said "OData is the equivalent of ODBC for the Web" (OASIS Approves OData 4.0 Standards for an Open, Programmable Web, 2014).

The data management strengths of SPARQL/RDF can be combined with the application development strengths of OData with a protocol proxy: OData4SPARQL. OData4SPARQL is the Janus-point between the application development world and the semantic information world.

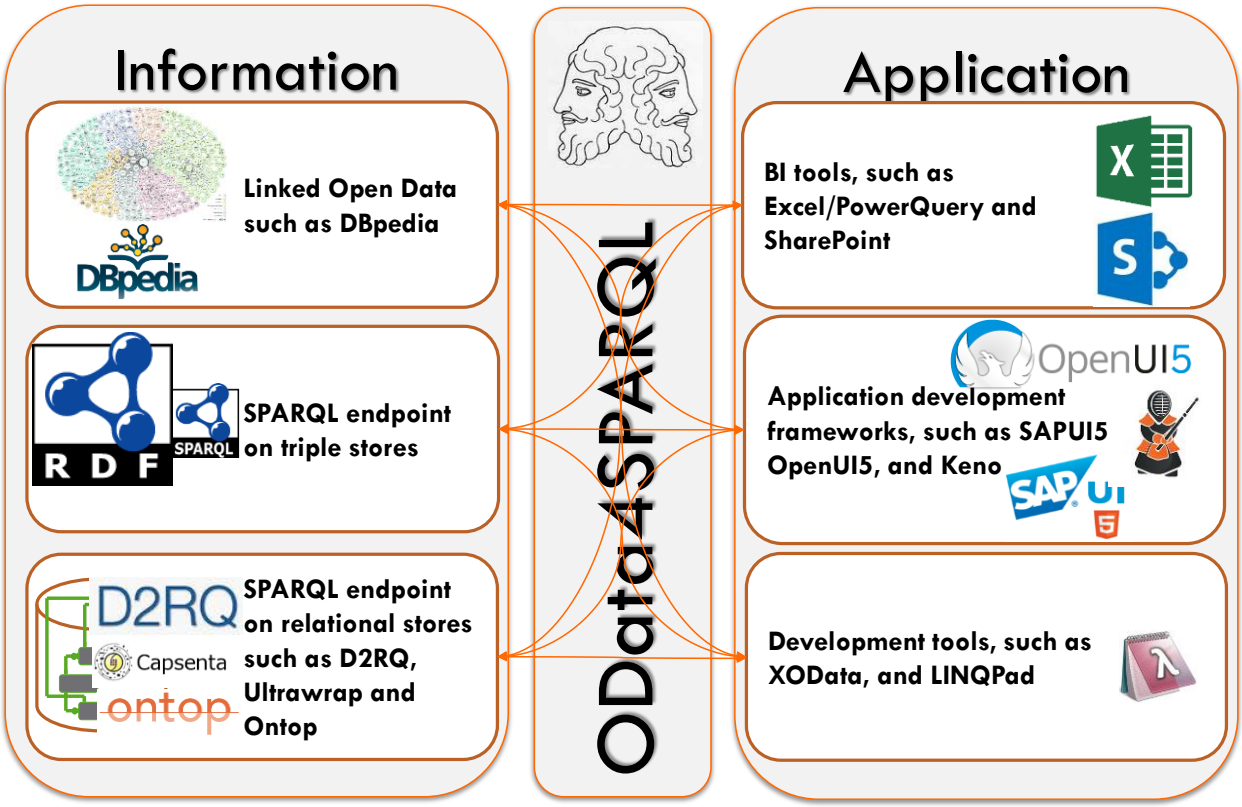


FIGURE 1: ODATA4SPARQL PROXY BETWEEN SEMANTIC INFORMATION AND APPLICATION DEVELOPMENT

Consuming OData2SPARQL

Development Tools

XODATA

A new online OData development is XOData from (PragmatiQa, n.d.). Unlike other OData tools, XOData renders very useful relationship diagrams. The Northwind RDFD model published via OData4SPARQL endpoint is shown below:

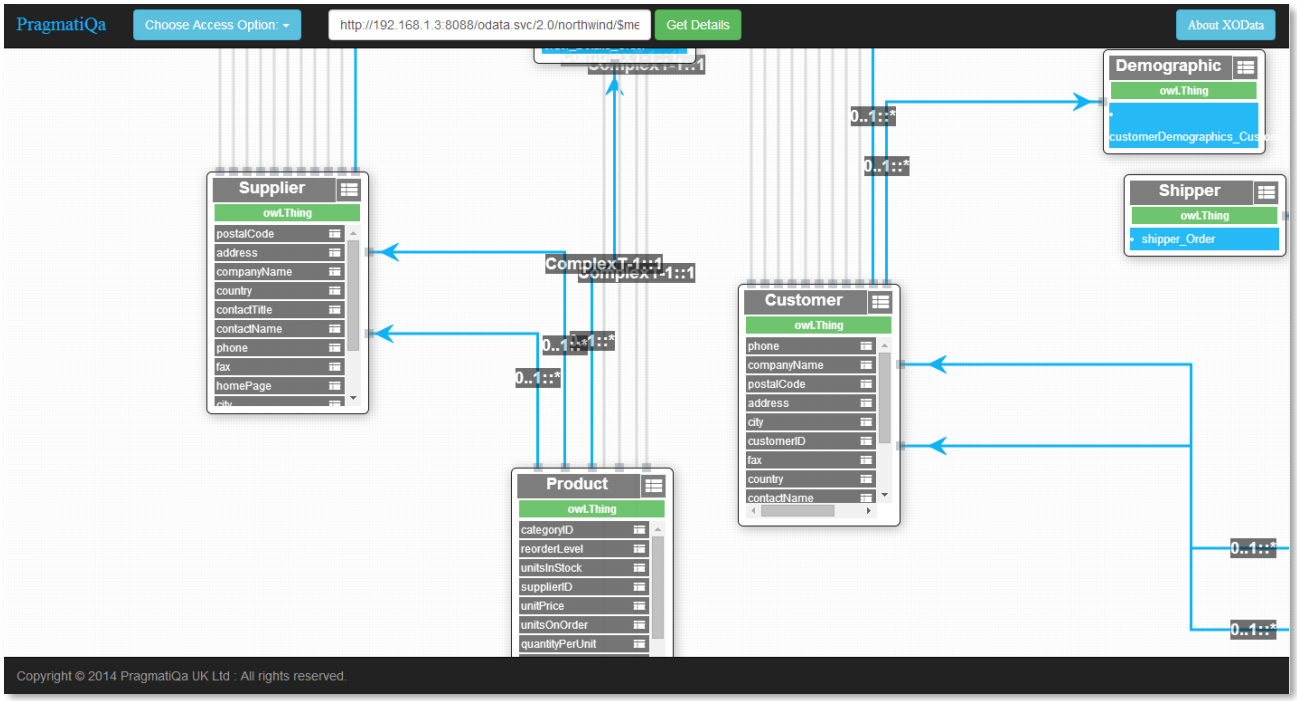


FIGURE 2: BROWSING THE EDM MODEL PUBLISHED BY OPDATA4SPARQL USING XODATA

XOData also allows the construct of queries as shown below:

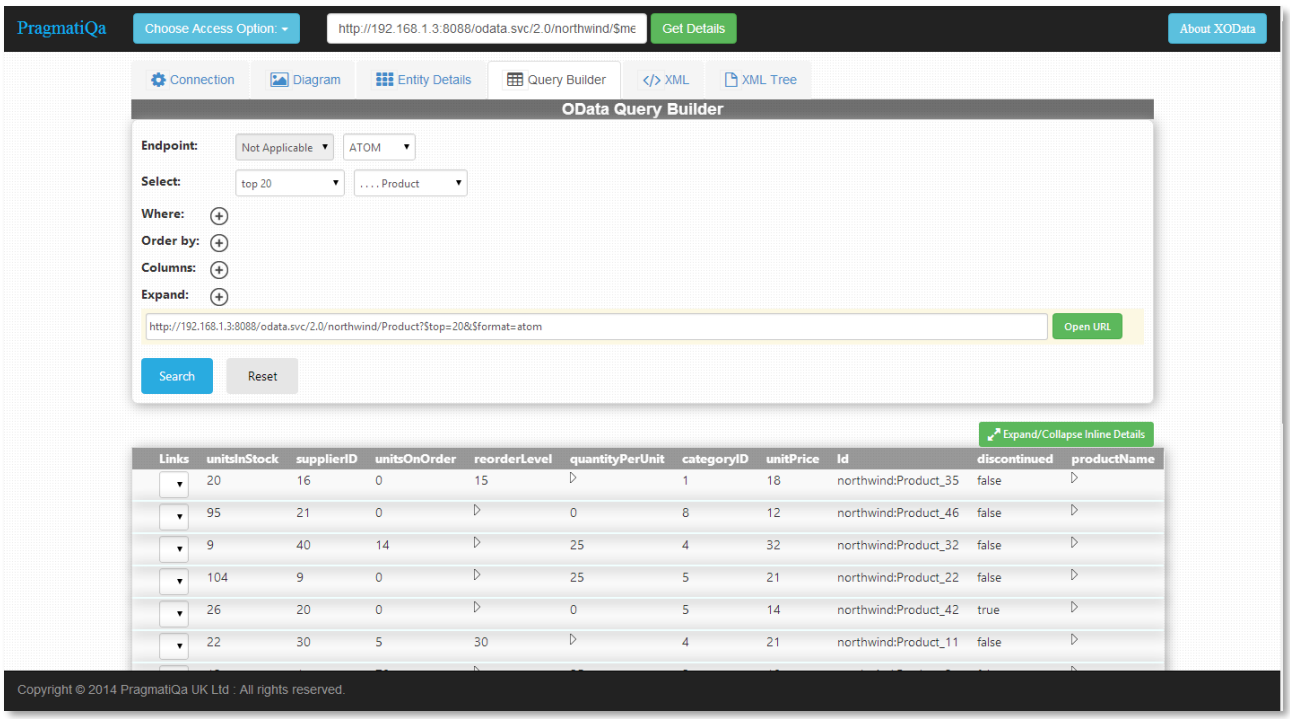


FIGURE 3: QUERYING THE ODATA4SPARQL ENDPOINTS USING XODATA

LINQPAD

(LINQPad, n.d.) is a free development tool for interactively querying databases using C#/LINQ. Thus it supports Object, SQL, EntityFramework, WCF Data Services, and, most importantly for OData4SPARQWL, OData services. Since LINQPad is centered on the Microsoft frameworks, WCF, WPF etc, this illustrates how the use of OData can bridge between the Java worlds of many semantic tools, and the Microsoft worlds of corporate applications such as SharePoint and Excel.

LINQPad shows the contents of the EDM model as a tree. One can then select an entity within that tree, and then create a LINQ or Lambda query. The results of executing that query are then presented below in a grid.

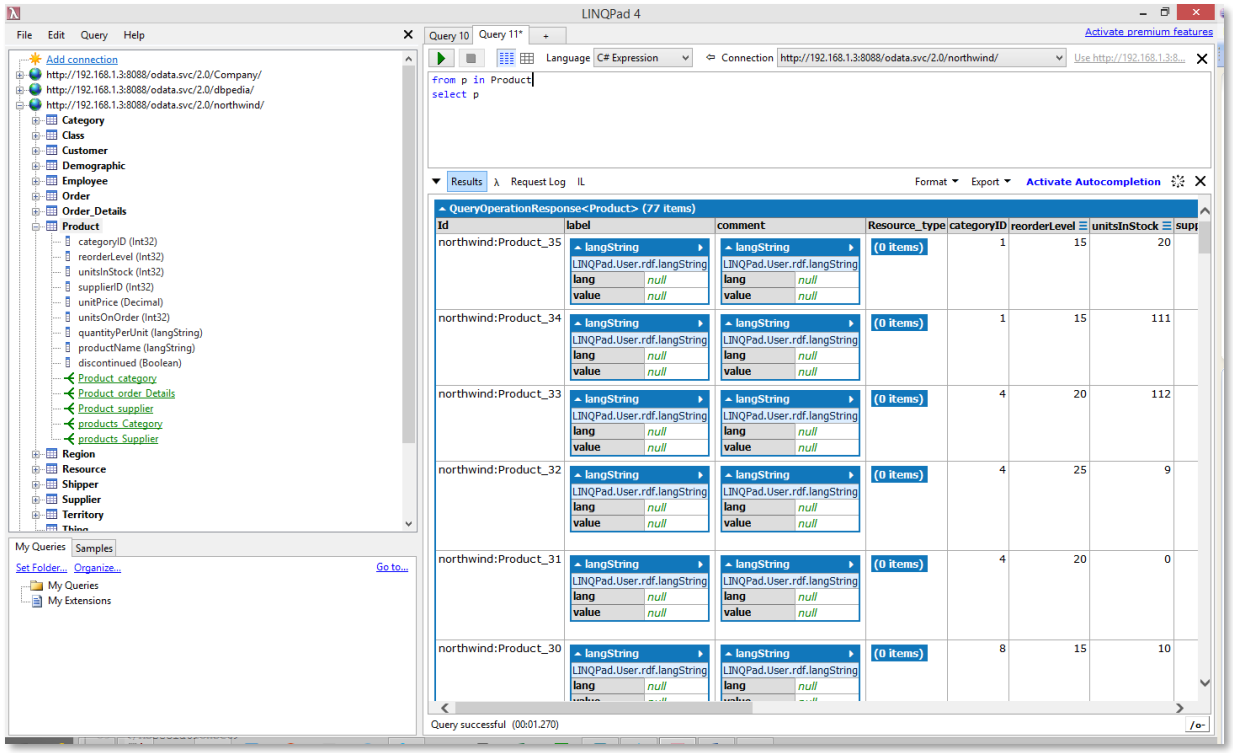


FIGURE 4: BROWSING AND QUERYING THE ODATA4SPARQL ENDPOINTS USING LINQPAD

LINQPad and XOData are good for testing out queries against any datasource. Therefore this also demonstrates using the DBpedia SPARQL endpoint as shown below:

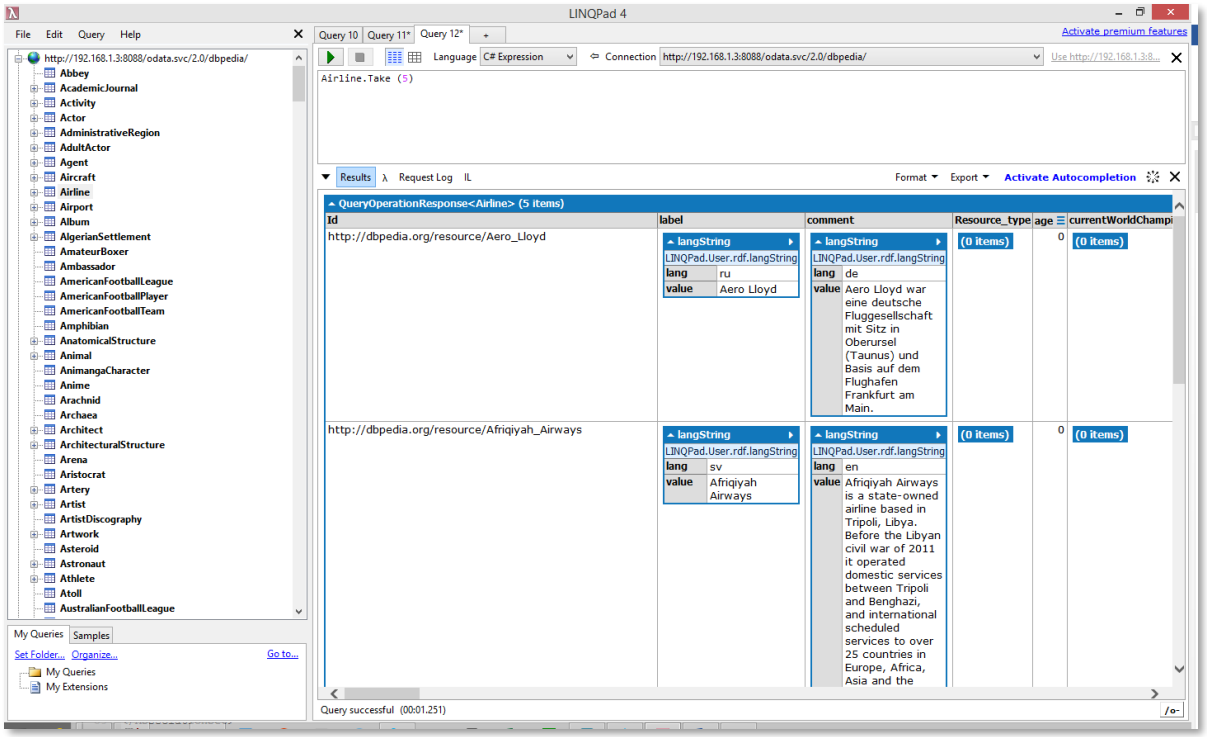


FIGURE 5: BROWSING DBPEDIA SPARQLENDPOINT USING LINQPAD VIA ODATA4SPARQL

Browsing Data

One of the primary motivations for the creation of OData4SPARQL is to allow access to Linked Open Data and other SPARQLEndpoints from the ubiquitous enterprise and desktop tools such as SharePoint and Excel.

LENS2ODATA

The objectives of lens2OData are to provide a simple method of OData query construction driven by the metadata provide by OData services

- Provides metamodel-driven OData query construction
  - Eliminates any configuration required to expose any OData service to lens2OData
- Allows searches to be saved and rerun
  - Allows ease of use by casual users
- Allows queries to be pinned to 'Lens' dashboard panels
  - Provides simple-to-use dashboard
- Searches can be parameterized
  - Allows for easy configuration of queries
- Compatible with OData2SPARQL, a service that exposes any triple store as an OData service

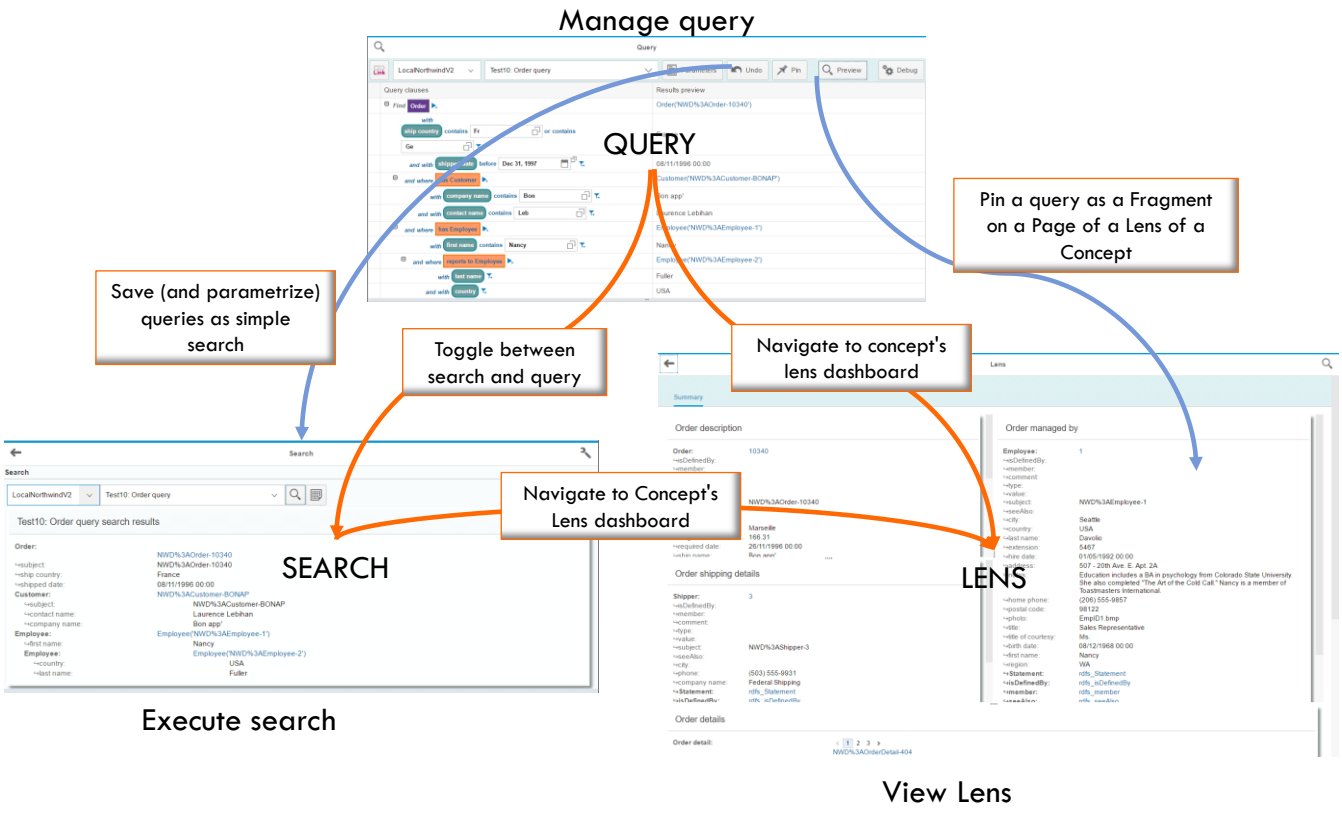


FIGURE 6: LENS2ODATA NAVIGATION

EXCEL/POWERQUERY

"Power Query is a free add-in for Excel 2010 and up that provide users an easy way to discover, combine and refine data all within the familiar Excel interface." (Introduction to Microsoft Power Query for Excel, 2014)

PowerQuery allows a user to build their personal data-mart from external data, such as that published by OData2SPARQL. The user can fetch data from the datasource, add filters to that data, navigate through that data to other entities, and so on with PowerQuery recording the steps taken along the way. Once the data-mart is created it can be used within Excel as a PivotTable or a simple list within a sheet. PowerQuery caches this data, but since the steps to create the data have been recorded, it can be refreshed automatically by allowing PowerQuery to follow the same processing steps. This feature resolves the issue of concurrency in which the data-sources are continuously being updated with new data yet one cannot afford to always query the source of the data. These features are illustrated below using the Northwind.rdf endpoint published via OData2SPARQL:

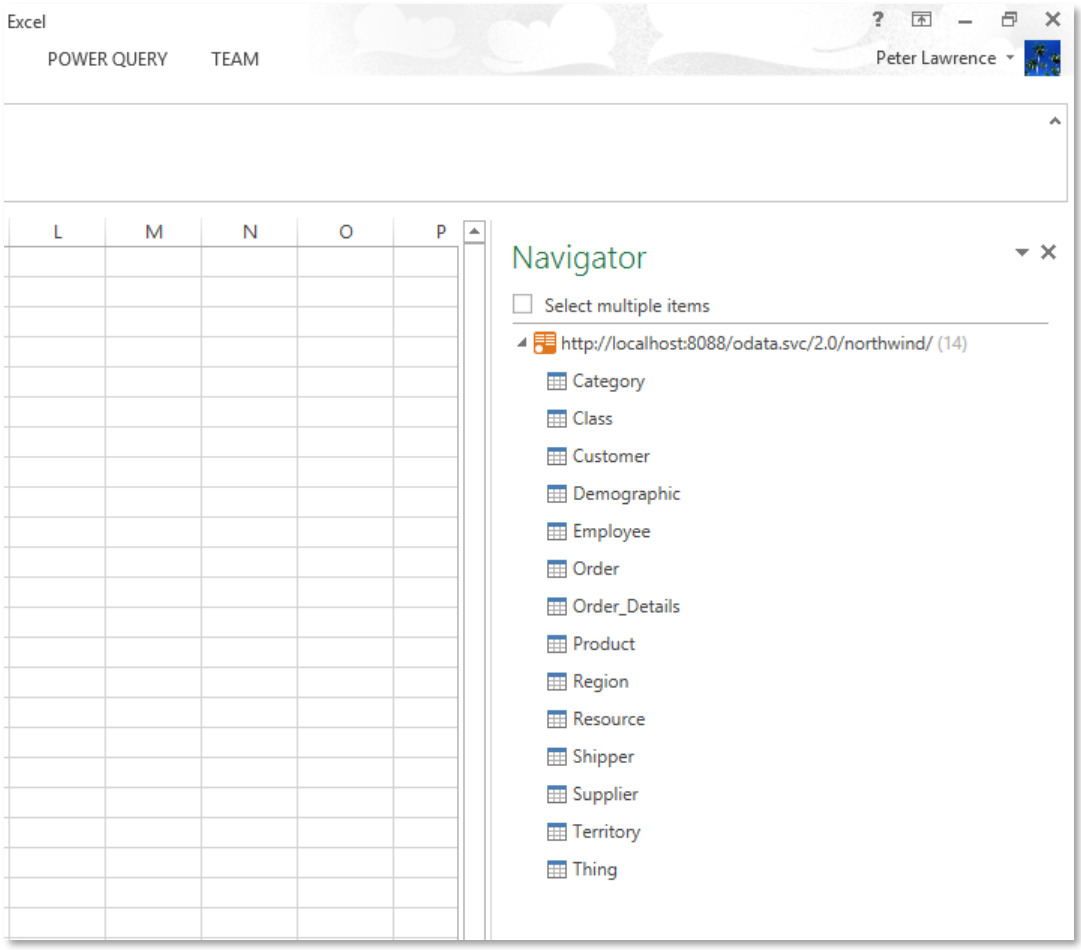


FIGURE 7: BROWSING THE ODATA4SPARQL ENDPOINT MODEL WITH POWERQUERY

Choosing an entity set allows one to start filtering and navigating through the data, as shown in the 'Applied Steps' frame on the right.

Note that the selected source is showing all values as 'List' since each value can have zero, one, or more values as is allowed for RDF DatatypeProperties.

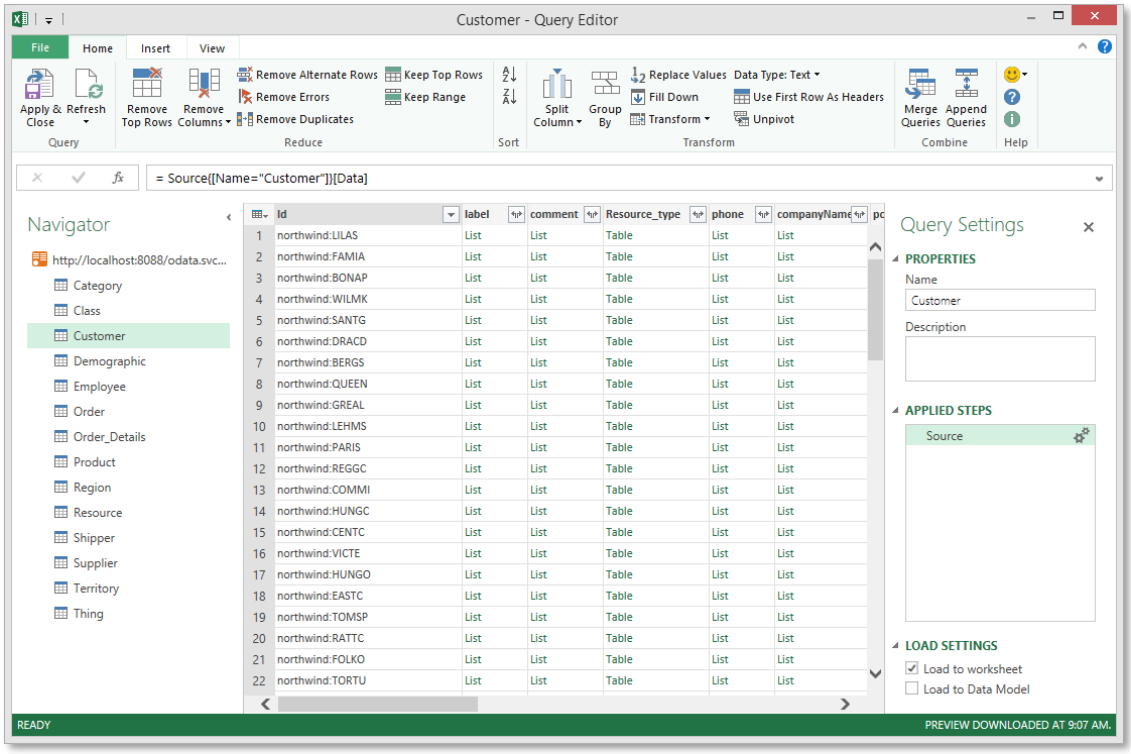


FIGURE 8: SETTING UP INITIAL SOURCE OF DATA IN POWERQUERY

As we expand the data, such as the companyProperty, we see that the Applied Steps records the steps take so that they can be repeated.

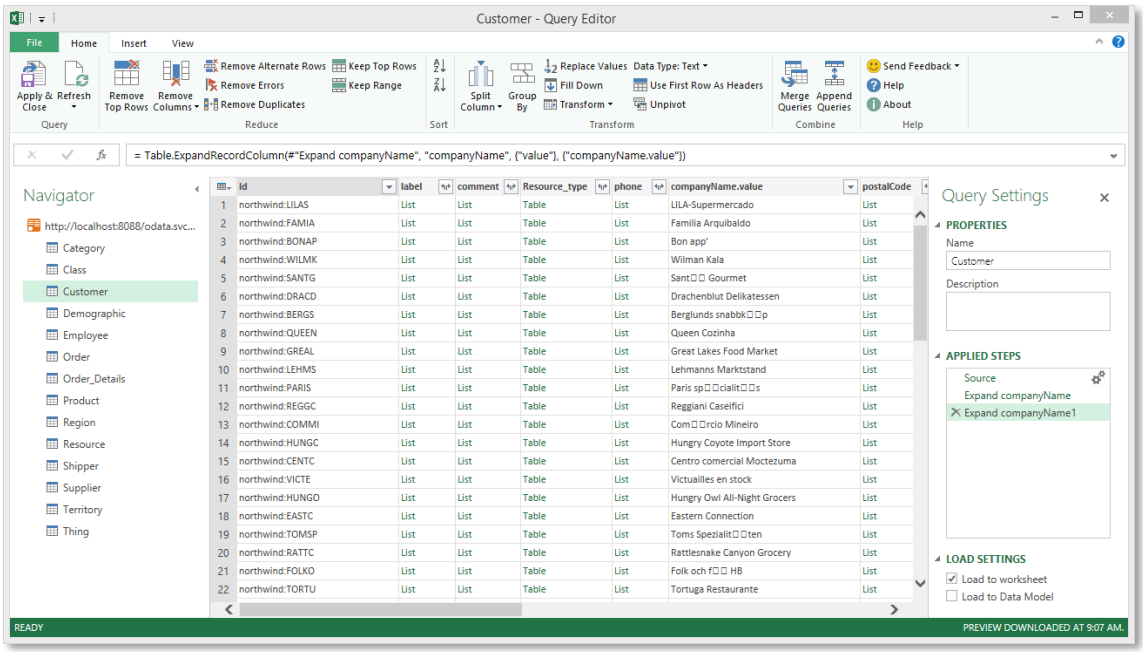


FIGURE 9: EXPANDING DETAILS IN POWERQUERY

The above example expanded a DatatypeProperty collection. Alternatively we may navigate through a navigation property such as Customer_orders, the orders that are related to the selected customer:

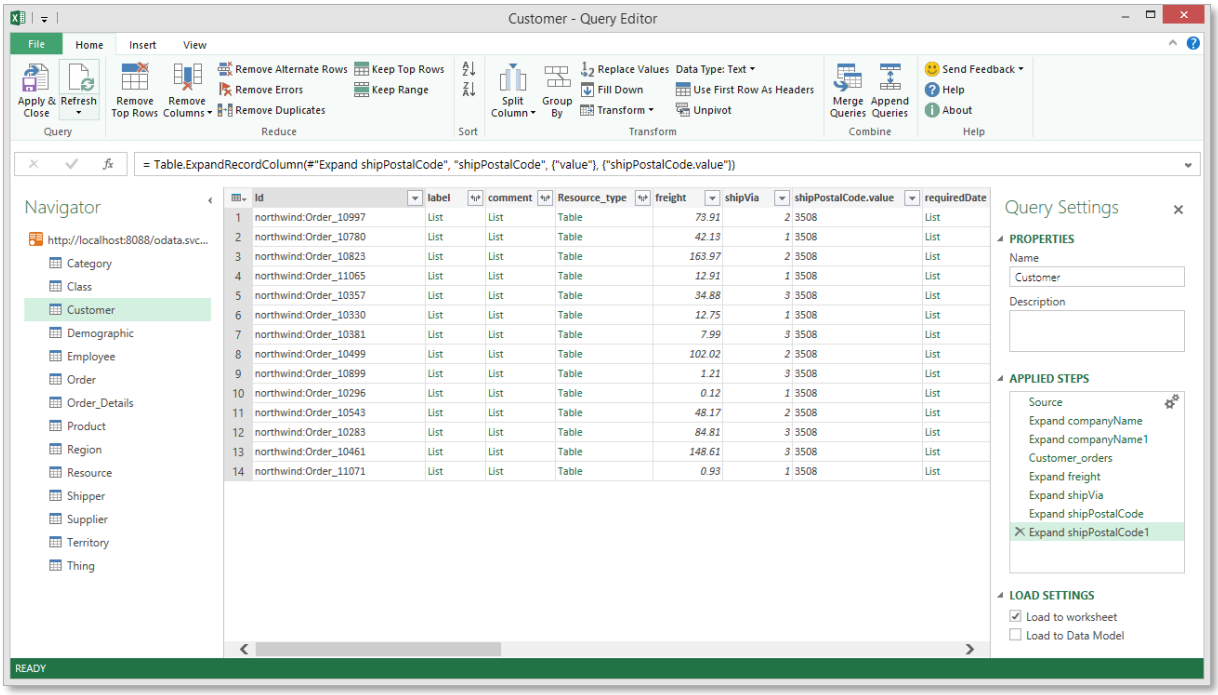
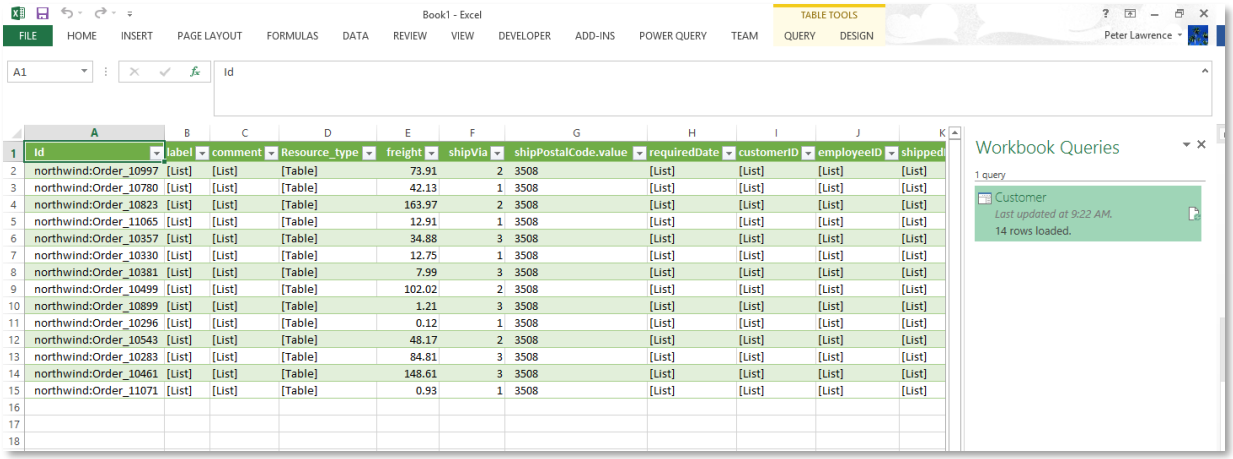


FIGURE 10: NAVIGATING THROUGH RELATED DATA WITH POWERQUERY

Once complete the data is imported into Excel:



Id	label	comment	Resource_type	freight	shipVia	shipPostalCode.value	requiredDate	customerID	employeeID	shipped
northwind:Order_10997	[List]	[List]	[Table]	73.91	2	3508	[List]	[List]	[List]	[List]
northwind:Order_10780	[List]	[List]	[Table]	42.13	1	3508	[List]	[List]	[List]	[List]
northwind:Order_10823	[List]	[List]	[Table]	163.97	2	3508	[List]	[List]	[List]	[List]
northwind:Order_11065	[List]	[List]	[Table]	12.91	1	3508	[List]	[List]	[List]	[List]
northwind:Order_10357	[List]	[List]	[Table]	34.88	3	3508	[List]	[List]	[List]	[List]
northwind:Order_10330	[List]	[List]	[Table]	12.75	1	3508	[List]	[List]	[List]	[List]
northwind:Order_10381	[List]	[List]	[Table]	7.99	3	3508	[List]	[List]	[List]	[List]
northwind:Order_10499	[List]	[List]	[Table]	102.02	2	3508	[List]	[List]	[List]	[List]
northwind:Order_10899	[List]	[List]	[Table]	1.21	3	3508	[List]	[List]	[List]	[List]
northwind:Order_10296	[List]	[List]	[Table]	0.12	1	3508	[List]	[List]	[List]	[List]
northwind:Order_10543	[List]	[List]	[Table]	48.17	2	3508	[List]	[List]	[List]	[List]
northwind:Order_10283	[List]	[List]	[Table]	84.81	3	3508	[List]	[List]	[List]	[List]
northwind:Order_10461	[List]	[List]	[Table]	148.61	3	3508	[List]	[List]	[List]	[List]
northwind:Order_11071	[List]	[List]	[Table]	0.93	1	3508	[List]	[List]	[List]	[List]

FIGURE 11: IMPORTING DATA FROM ODATA4SPARQL WITH POWERQUERY

Unlike conventional importing of data into Excel, the personal data-mart that was created in the process of selecting the data is still available.

Application Development Frameworks

There are a great number of superb application development frameworks that allow one to create cross platform (desktop, web, iOS, and Android), rich (large selection of components such as grids, charts, forms etc) applications. Most of these are based on the MVC or MVVM model both of which require a systematic and complete (CRUD) access to the back-end data via a RESTful API. Now that OData has been adopted by OASIS, the number of companies offering explicit support for OData is increasing, ranging from Microsoft, IBM, and SAP to real-time database vendors such as OSI. Similarly there are a number of frameworks, one of which is SAPUI5 (UI Development Toolkit for HTML5 Developer Center , n.d.) which has an open source version OpenUI5 (OpenUI5, n.d.).

OPENUI5

OpenUI5 is an impressive framework which makes MVVC/MVVM application development easy via the Eclipse-based IDE. Given that OData4SPARQL publishes any SPARQLEndpoint as an OData endpoint, it means that this development environment is immediately available for a semantic application development. The following illustrates a master-detail example against the Northwind.rdf SPARQL endpoint via OData2SPARQL.

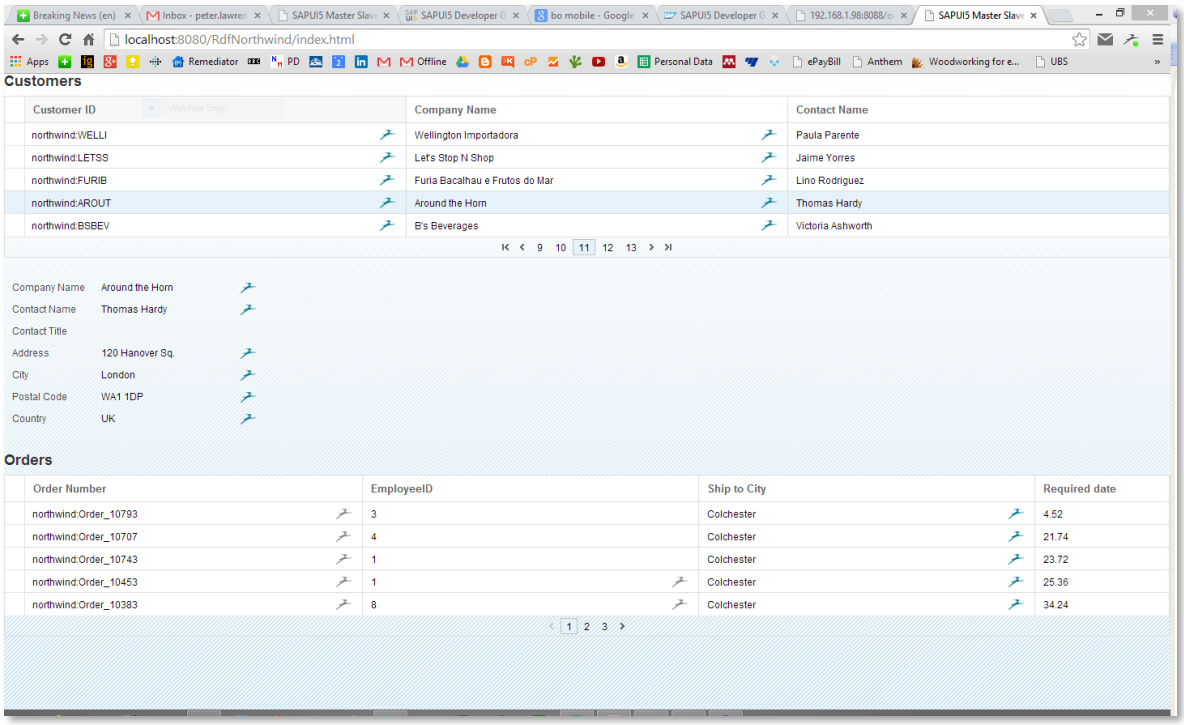


FIGURE 12: OPENUI5 APPLICATION USING ODATA2SPARQL ENDPOINT

Yes we could have cheated and used the Northwind OData endpoint directly, but the QNames of the Customer ID and Order Number reveals that the origin of the data is really RDF.

WEBIDE

SAP Web IDE is a powerful, extensible, web-based integrated development tool that simplifies end-to-end application development. Since it is built around using OData datasources as its provider, then webIDE can be used as a semantic application IDE.

WebIDE runs either as a cloud based service supplied free by SAP, or can be downloaded as an Eclipse ORION application. Since the development is probably against a local OData endpoint, then the latter is more convenient.

Instructions for installing the Web IDE Personal edition can be found here: SAP Web IDE Personal Edition

Once installed an OData service definition file (for example Northwind) can be added to the SAPWebIDE\config_master\service.destinations\destinations folder, as follows

```
Description=Northwind
Type=HTTP
TrustAll=true
```

```
Authentication=NoAuthentication
Name=Northwind
ProxyType=Internet
URL=http://localhost:8080
WebIDEUsage=odata_gen
WebIDESystem=Northwind
WebIDEEnabled=true
```

Since the web IDE does not support basetypes, it is recommended OData repository definitionhas useBaseType false and withSAPAnnotations true

An application can be built using a template, for example the CRUD Master-Detail Application. During that process the Data Connection needs to be setup: choose Service URL, select the data service (Northwind) and enter the path of the particular endpoint (/OData2SPARQL/2.0/NW/)

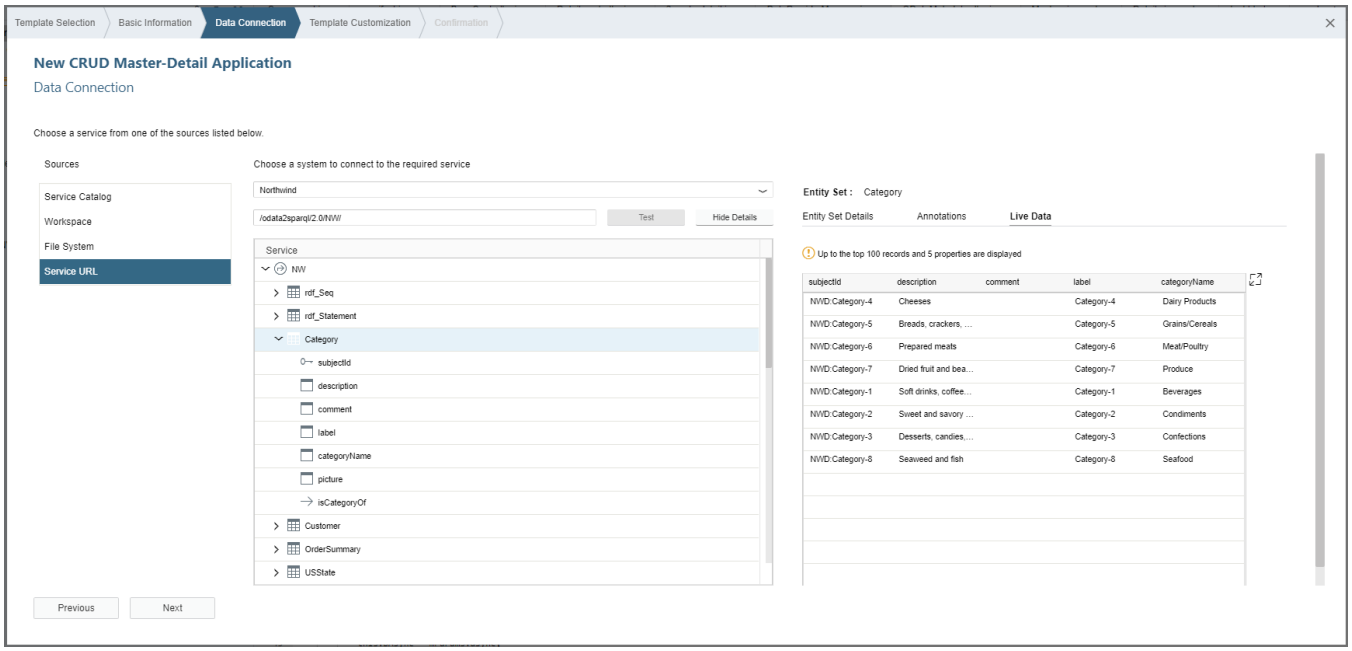


FIGURE 13: WEB IDE DATA CONNECTION DEFINITION

Since the Web IDE and OpenUI5 is truly model driven, the IDE creates a master-detail given the entities that you define in the next screen:

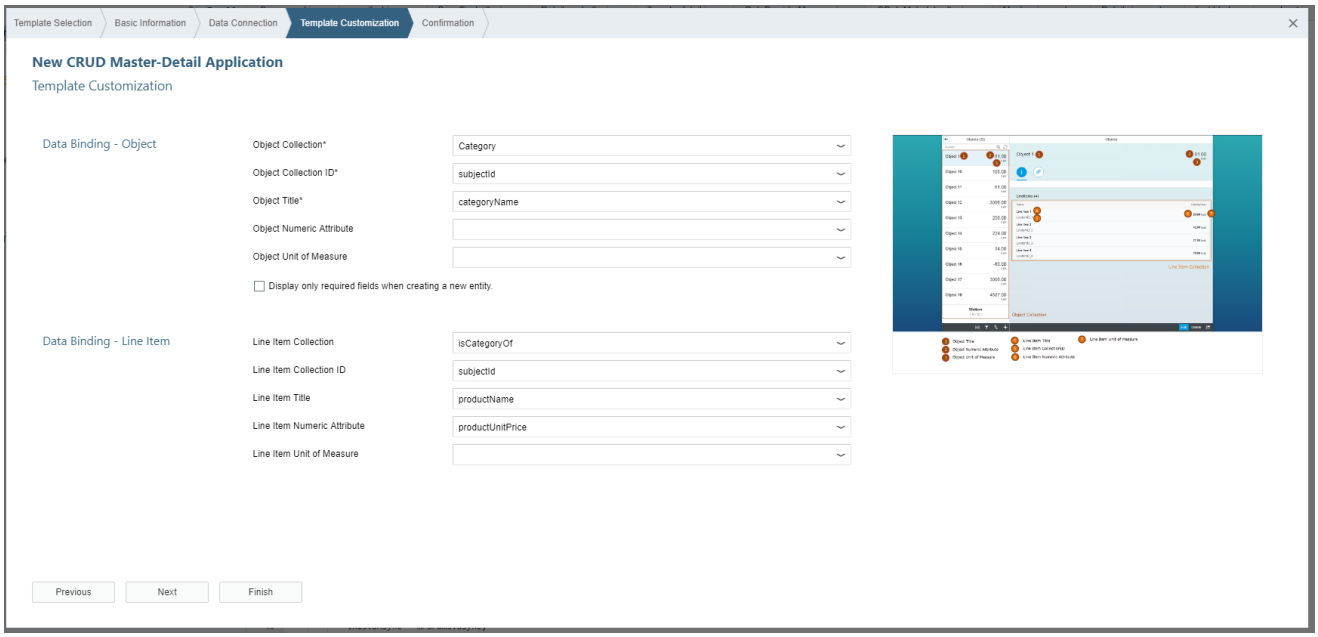


FIGURE 14: WEB IDE TEMPLATE CUSTOMIZATION

The application is now complete and can be launched from the IDE:

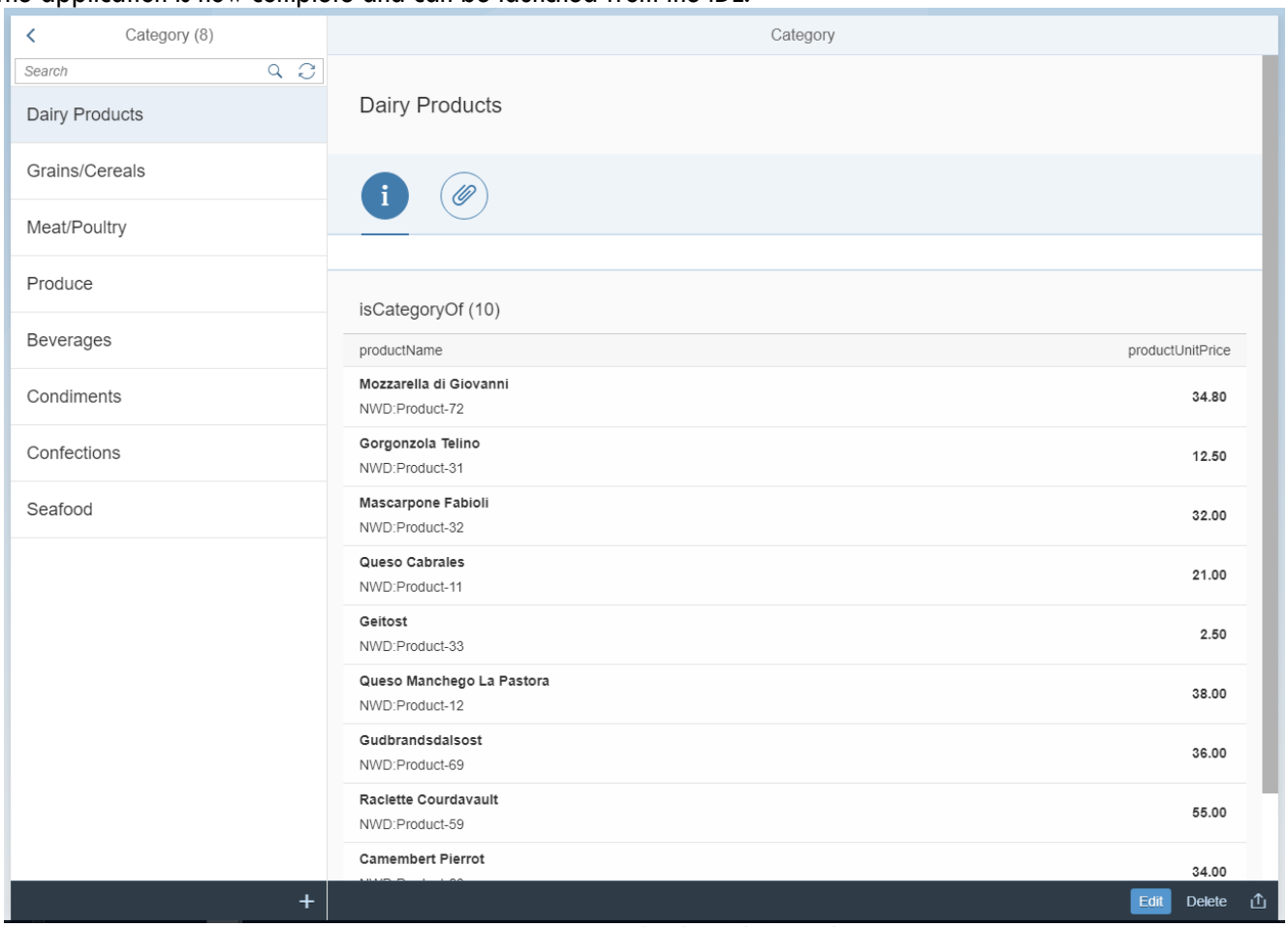


FIGURE 15: WEB IDE TEMPLATE APPLICATION EXAMPLE

WHAT DOES ODATA2SPARQL DO?

OData2SPARQL is a proxy server that provides OData V2 and V4 access to any triplestore that published a SPARQL interface. OData2SPARQL provides the following capabilities:

1. A RESTful API conforming to the OData standard via which applications can access the underlying RDF datasets.
2. A mapping of the OData metadata model to the underling vocabulary and vice versa
3. A conversion of the incoming OData query into the corresponding SPARQL query, and the conversion of the SPARQL results back into the OData results, which can be in Atom/XML or JSON format.
4. A vocabulary (OData4sparql) that allows the mapping between OData and RDF/RDFS/OWL to be described.

OData2SPARQL URI Query

The general structure of the OData URI is shown in Figure 16: OData URI Structure

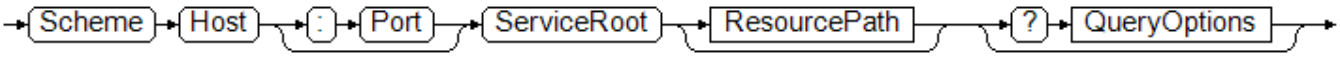


FIGURE 16: ODATA URI STRUCTURE

An example of OData2SPARQL is as follows:

http://localhost:8080/OData2SPARQL/2.0/NW/Customer?$top=1

where

ServiceRoot = OData.svc/{Version}/{EndPoint}

where

{Version} = 2.0

{EndPoint}= logical name of one of the endpoints described within the VoID model of the attached SPARQL data-sources. For example Northwind, an RDF equivalent of the 'classic' Northwind sample database. (Northwind database, 2011)

ResourcePath = Customer

and QueryOption = $top=1

Semantics of OData to SPARQL

OData is essentially a rigorously standardized REST service specification. The following table describes the semantics of the OData service calls:

OData2SPARQL Semantics									
ResourcePath	**UriType**[7]	**Example**	**ContentType**	**Body**	**GET**[4]	**POST**[3]	**DELETE**[3]	**PATCH**[3,6]	**PUT**[3,6]
EntitySet	**1**	Category	application/json	{"comment":"a comment","label":"a label","subjectId":"NWD~Category-A"}[9]	Returns all EntitySet and properties but not navigation properties	Creates new Entity within entitySet initiating with contents of body, using subjectId as the new URI of the created entity. Returns the created entity.	n/a	n/a	n/a
Entity	**2**	Category('NWD~Category-A')	application/json	{"label":"a new label"}	Returns all properties of the Entity	n/a	Removes entity and references to entity entirely from store	Updates the entity with the properties contained in the body	Updates the entity with the properties contained in the body
Entity /$ref	**2**	Category('NWD~Category-A') /$ref	application/json	{"@odata.id" :"serviceRoot/Category('NWD~Category-B')"}	Returns a reference to the entity.	n/a	Removes entity and references to entity entirely from store	n/a[8]	n/a[8]
Entity /NavigationSet	**6B**	Category('NWD~Category-A') /isCategoryOf	application/json	{"comment":"a product comment" ,"label":"a product label" ,"subjectId":"NWD~Product-A"}[9]	Returns all entities that belong to the navigationSet associated with the specified entity. If the navigationSet has an inverse, then those are included.	Creates a new entity as long as subjectId specified and adds to the navigationSet associated with the specified entity. Returns the created entity.	n/a[1]	n/a[2]	n/a[2]
Entity /NavigationSet /$ref	**7B**	Category('NWD~Category-A') /isCategoryOf/$ref	application/json	{"value":[{"@odata.id": "serviceRoot/Product('NWD~Product-A')"}, {"@odata.id": "serviceRoot/Product('NWD~Product-B')"]}	Returns an array of reference values of the entity in the form [{@odata.id: <$ref value>}...]	**Adds** new reference value(s) to the collection	Deletes all reference values from the collection	n/a	n/a
Entity /NavigationProperty[5]	**6A**	Product('NWD~Product-A') /category	application/json	{,"label":"a new label"}	Returns the entity value of the navigationProperty. Also applies if the navigationProperty has an inverse	n/a	Deletes the entity value of the navigationproperty.	Updates the entity value of the navigationProperty associated with the specified entity.	Updates the entity value of the navigationProperty associated with the specified entity.
Entity /NavigationProperty /$ref	**7A**	Product('NWD~Product-A') /category/$ref	application/json	{"@odata.id" :"serviceRoot/Category('NWD~Category-A')"}	Returns the reference value of the entity in the form @odata.id: <$ref value>	n/a	Deletes the reference to the navigationProperty entity	Adds a new reference value to the navigationproperty associatd with the entity	Adds a new reference value to the navigationproperty associatd with the entity
Entity /NavigationEntity	**2**	Category('NWD~Category-A') /isCategoryOf('NWD~Product-A')	application/json	{"comment":"a product comment" ,"label":"a product label" ,"subjectId":"NWD~Product-A"}9	Returns the entity defined in the navigationSet associated with the specified entity. This is the same as referencing the entity directly, in addition to verifying the entity belongs to the navigationSet. If it does not belong, then no data is returned even if the specified entity exists.	n/a	Deletes the defined entity from the navigationSet associated with the specified entity. It does not delete the defined entity from the store.	Updates the details of the defined entity from the navigationSet associated with the specified entity.	Updates the details of the defined entity from the navigationSet associated with the specified entity.
Entity /NavigationEntity /$ref	**7A**	Category('NWD~Category-A') /isCategoryOf('NWD~Product-A') /$ref	application/json	{"@odata.id" :"serviceRoot/Product('NWD~Product-A')"}	Returns a reference to the entity defined in the navigationSet associated with the specified entity. This is the same as referencing the entity directly, in addition to verifying the entity belongs to the navigationSet. If it does not belong, then no data is returned even if the specified entity exists.	n/a	Deletes the defined entity from the navigationSet associated with the specified entity. It does not delete the defined entity from the store.	Replaces the referenced entity from the navigationSet associated with the specified entity.	Replaces the referenced entity from the navigationSet associated with the specified entity.
Entity /Property	**5**	Category('NWD~Category-A') /label	application/json	{"value": "a changed comment"}	Returns the value of the specified property as JSON	n/a	Deletes the value of the specified property of the entity	Updates the property value with the "value" property contained in the application/json body	Updates the property value with the "value" property contained in the application/json body
Entity /Property /$value	**5**	Category('NWD~Category-A') /label /$value	text/plain	"a changed comment"	Returns the value of the specified property as text/plain	n/a	Deletes the value of the specified property of the entity	Updates the property value with the value contained in the text/plain body	Updates the property value with the value contained in the text/plain body
Entity /NavigationEntity /Property	**5**	Category('NWD~Category-A') /isCategoryOf('NWD~Product-1') /label	application/json	{"value": "a changed comment"}	Returns the value of the specified property as JSON	n/a	Deletes the value of the specified property of the navigationEntity associated with the specified entity	Updates the property value with the "value" property contained in the application/json body	Updates the property value with the "value" property contained in the application/json body
Entity /NavigationEntity /Property /$value	**5**	Category('NWD~Category-A') /isCategoryOf('NWD~Product-1') /label /$value	text/plain	"a changed comment"	Returns the value of the specified property as text/plain	n/a	Deletes the value of the specified property of the navigationEntity associated with the specified entity	Updates the property value with the value contained in the text/plain body	Updates the property value with the value contained in the text/plain body

Example based on simple RDF model:

```
:aProduct a :Product .
:aCategory a :Category .
:aProduct :category :aCategory .
:aCategory :isCategoryOf :aProduct , :anotherProduct, ... .
```

Notes

1: Some confusion regarding OData standard whether this is supported or not. Library used does not support this.
2: If the body contains a reference to the key, for example "subjectId":"NWD~Product-A", the value will be ignored. All other properties will be assumed to reference the navigationEntity key.
3: If the service has enable change tracking, then these methods will cause the changes to be logged
4: In the case of a GET the URL patterns is resourcepath?queryOptions, where the queryoptions control what is being returned as well as navigating further through the data. See https://docs.oasis-open.org/odata/odata/v4.01/odata-v4.01-part2-url-conventions.html

```
queryOptions ::=  $select=(property,)*&$expand=(navigationProperty(query)?,)+
query ::= ($select=(property,)*,)?($expand=(navigationProperty(query)?,)+)?
```

5: A navigationProperty allows only a single value for the target entity, thus it need not be specified is disambiguate it from others. In fact the standard requires that it is ***not*** specified.
6: PATCH is preferred to PUT. OData standard requires that PATCH is supported and PUT is optional. See http://docs.oasis-open.org/odata/odata/v4.01/csprd04/part1-protocol/odata-v4.01-csprd04-part1-protocol.html#sec_UpdateanEntity
7: Classifier used to categorize the ResourcePath
8: Not (yet) supported as this would be interpreted as changing the URI of a resource, which is meant to be immutable.
9: The body of an entity insert can include dependent entities, either existing entities via bind references, or new entities:

```
{
  "label": "OrderDetail-10248-20",
  "subjectId": "NWD~OrderDetail-10248-20",
  "product@odata.bind":  "Product('NWD~Product-20')"
}
```

```
{
 "label": "OrderDetail-10248-20",
 "subjectId": "NWD~OrderDetail-10248-a",
 "product":{
   "subjectId": "NWD~Product-a",
   "label":"aa",
   "comment":"aaaa"
 }
}
```

TABLE 4: ODATA2SPARQL HTML SEMANTICS

Mapping RDF to OData Models

It is frequently cited that one of the benefits of the semantic modeling is that it is the only approach that is truly model driven. To the extent that both model and the equivalent of the data schema (aka ontology) are defined using RDF, then this is indeed true especially when comparing with a traditional RDBMS or a less traditional model such as MongoDB in which the schema is decidedly hard-coded into the database.

However in part this is what OData is trying to alleviate. The Entity Data Model (EDM) is the description of the data model that is being exposed by the OData service.

Thus the first step in exposing an RDF datasource as an OData endpoint is to generate the EDM from the RDF model.

Feature	OData	RDF/SPARQL
Classes	EntityType	Class
Attributes	OData allows an entitytype to any number of property	The equivalent to OData attributes are OWL's DatatypeProperty
Relationships	Navigation property	ObjectProperty
Inheritance	Property inheritance supported via sub-class definitions, but only single inheritance	
Multi-valued properties	Supports collection properties as of V3.0 and above.	Inherently supports multi-valued properties
Language encoded	Supported via complex-types, with fields for the value and language.	Inherently supports language-tagged literals.
Extended datatypes		
Namespaces	EntityTypes within the EDM belong to a namespace	Classes
Primary identifier	Uses URL: http://inova8.com/OData/v2/northwind/Company('northwind:Company1')	Uses URI: <http://inova8.com/northwind#Company1>

TABLE 5: ODATA/EDM AND RDF MAPPING

Schema

Entity types, associations, entity sets, and association sets are all defined within a concept of schema, which are allocated a namespace in just the same way as RDF/OWL classes, datatype- and object- properties carry their namespace along with their definition.

EDM

```
<Schema xmlns="http://schemas.microsoft.com/ado/2006/04/edm"
Namespace="northwind">
```

RDF

```
<rdf:RDF
    xmlns:rdf="http://www.w3.org/1999/02/22-rdf-syntax-ns#"
    xmlns:northwind="http://inova8.com/northwind#"
    xmlns:owl="http://www.w3.org/2002/07/owl#"
```

```
xmlns:xsd="http://www.w3.org/2001/XMLSchema#"
xmlns:rdfs="http://www.w3.org/2000/01/rdf-schema#"
xml:base="http://inova8.com/northwind">
```

EntityType

EntityTypes, which are the equivalent of RDFS/OWL classes, are structures consisting of named and typed properties and with a key. For example Customer, Employee, Supplier etc. Thus they are aligned with RDFS or OWL classes. Similarly EDM EntityTypes may have a BaseType from which it is derived. This is equivalent to the RDFS concept of rdfs:subClassOf:

EDM

```
<EntityType Name="Customer" Abstract="false" BaseType="owl.Thing">
```

RDF

```
northwind:Customer  a    owl:Class ;
        rdfs:label       "Customer"^^xsd:string ;
        rdfs:subClassOf  owl:Thing .
```

KEY

In OData EDM each entity needs a Key to be defined. Of course in RDF the URL: is the universal identifier. In the mapping from RDF to EDM, we have assumed that every class inherits ultimately from the class rdfs:Resource.

EDM

```
…
<EntityType Name="Resource" Abstract="false">
      <Key>
            <PropertyRef Name="Id"/>
      </Key>
<Property Name="Id" Type="Edm.String" Nullable="false"/>
…
```

Of course within RDF the concept of the key is inherent within any statement so it does not have to be declared.

Unlike RDF, OData/EDM does not specify how the key should be constructed. In OData2SPARQL we chose to use the URL as the key value, either the full URL or the QName. Thus a Customer entitytype instance can be referenced as:

Customer('northwind:FAMIA')

Or

Customer(' http://inova8.com/northwind#FAMIA')

We could also reference them as:

Thing('northwind:FAMIA')

Or

Thing('http://inova8.com/northwind#FAMIA')

Property

OData EDM distinguishes between properties and navigation properties, in much the same way as OWL distinguishes between owl:DatatypeProperty, the equivalent of a property, and owl:ObjectProperty, the equivalent of a NavigationProperty

EDM

```
<EntityType Name="Customer" Abstract="false" BaseType="owl.Thing">
      <Property Name="phone" Type="rdf.langString" Nullable="false"/>
      …
```

RDF

```
northwind:phone  a   owl:DatatypeProperty ;
        rdfs:domain  northwind:Customer , northwind:Supplier ;
                          rdfs:label   "Phone"^^xsd:string
```

SIMPLE PROPERTY

Most literal values within RDF can be mapped to simple primitive datatype values, and hence are classified as OData/EDM simple properties. This is accomplished by aligning the RDF xsd:Datatypes with the Edm primitive datatypes such as Edm.Decimal, Edm.String etc.

Note that (OData Version 4.0 Part 3: Common Schema Definition Language (CSDL), 2014) supports the specialization of primitive types using TypeDefinitions. This would then align with the user defined rdfs:Datatypes within RDF. OData2SPARQL currently supports up to V3.0

COMPLEX PROPERTY

RDF is inherently built around simple literals. However within RDF, strings are frequently language tagged such as "San"@es. EDM/OData datatypes do not include such tagging, therefore a 'rdf.langString' complex type is defined within the Edm 'rdf' schema to allow the language tag, if supplied, to be accessible via OData:

EDM

```
<ComplexType Name="langString">
      <Property Name="lang" Type="Edm.String" Nullable="true"/>
      <Property Name="value" Type="Edm.String" Nullable="false"/>
</ComplexType>
```

Thus in RDF we might have a value "San"@es which becomes in OData:

Customer('northwind:FAMIA').city.value = 'San'
Customer('northwind:FAMIA').city.lang =' es'

COLLECTION PROPERTY

Another distinctive feature of RDF is that any property can take on zero, one, or many values all of which are valid unless there is an OWL cardinality restriction. As of OData V3.0, property types can be declared as collections of simple or complex types.

EDM

```
…
<Property Name="phone" Type="Collection(rdf.langString)"
Nullable="false"/>
```

…

Note that one implication of this is that even if the property has only one value it will be mapped to a collection of one value, rather than a scalar.

Additionally, as shown in the example, the collection may be of complex property values, in this case rdf.langStrings.

Navigation Property and Association

Just like OWL, OData distinguishes between a datatype property and an object property. OData refers to an owl:objectProperty as an edm.NavigationProperty, reflecting its origins in the entity-relationship model in which a foreign key in one table or entitytype is used to navigation to an associated entity in another association table.

OData/EDM is really quite verbose when declaring a navigation property, nevertheless the same concept as an OWL objectproperty can be described as follows.

> Associated with the EntityType are zero, one or more NavigationProperties. Each NavigationProperty is given a unique name within the EntityType as well as the Relationship which describes the record linking the two (FromRole and ToRole) EntityTypes.
>
> Additionally we need to declare the Association. Again this reveals the origin of EDM within the ER world: an Association can be thought of as a description of a relational table that contains the many-to-many foreign-key mappings.
>
> Since the Association is independent of the EntityType, it requires a unique name within the namespace. Thus OData2SPARQL uses for the name {FromRoleEntityType}_{ObjectProperty} as both a descriptive and unique name.
>
> EDM

```
<EntityType Name="Customer" Abstract="false" BaseType="owl.Thing">
      <NavigationProperty Name="Customer_orders"
      Relationship="northwind.Customer_orders" FromRole="Customer"
      ToRole="Order"/>
      …

<Association Name="Customer_orders">
      <End Multiplicity="0..1" Role="Customer"
      Type="northwind.Customer" />
      <End Multiplicity="*" Role="Order" Type="northwind.Order" />
      <ReferentialConstraint>
            <Principal Role="Customer">
                  <PropertyRef Name="Id" />
            </Principal>
            <Dependent Role="Order">
                  <PropertyRef Name="Id" />
            </Dependent>
      </ReferentialConstraint>
</Association>
```

> RDF

```
northwind:orders  a  owl:ObjectProperty ;
      rdfs:domain  northwind:Customer ;
      rdfs:range   northwind:Order ;
      rdfs:label   "orders"^^xsd:string .
```

Incoming Navigation Property and Association

Within RDF, when we have a statement like {northwind:FAMIA northwind:order northwind:order_10347} we recognize an EDM navigationproperty, northwind:order, from Customer to Order. However when navigating through instances we might have an Order and wish to discover which Customer made that Order. Unfortunately, if there is no inverse navigation property from an Order to the corresponding Custoemr, there will be no easy way to get there.

Therefore for each and every navigation from a FromRole to a ToRole, we want to construct the inverse NavigationProperty. So pursuing the above example, we need a Navigationproeprty on the ToRole, Order, that allows us to navigate back to the FromEntity, Customer. This would allow us to find the Customer who was responsible for this particular order.

```
<EntityType Name="Order" Abstract="false" BaseType="owl.Thing">
      <NavigationProperty Name="orders_Customer"
      Relationship="northwind.orders_Customer" FromRole="Order"
      ToRole="Customer"/>
      …

<Association Name="orders_Customer ">
      <End Multiplicity="0..1" Role="Order" Type="northwind.Order" />
      <End Multiplicity="*" Role="Customer" Type="northwind.Customer"
      />
      <ReferentialConstraint>
            <Principal Role="Order">
                  <PropertyRef Name="Id" />
            </Principal>
            <Dependent Role="Customer">
                  <PropertyRef Name="Id" />
            </Dependent>
      </ReferentialConstraint>
</Association>
```

EntityContainer

As well as defining the EntityTypes and Assocations, the EDM defines the different containers of these: EntitySets, and AssociationsSets. One could imagine a RDBMS with a several tables containing customers, perhaps NationalCustomer, and InternationalCustomers. These would be two containers of the EntityType Customer.

In RDF we do not have to worry about such details, however we need to define these within the EDM as it is the EntitySets that are being queried via the OData RESTful interface.

ENTITYSETS

Within OData2SPARQL we construct one EntitySet for each EntityType, with the same name as the EntityType to which it is related.

```
<EntityContainer
xmlns:annotation="http://schemas.microsoft.com/ado/2009/02/edm/annotati
on" Name="InstanceContainer" m:IsDefaultEntityContainer="true"
annotation:LazyLoadingEnabled="false">
<EntitySet Name="Customer" EntityType="northwind.Customer"/>
```

…

ASSOCIATIONSETS

Within OData2SPARQL we construct one AssociationSet for each Association, with the same name as the Association to which it is related.

```
<EntityContainer
xmlns:annotation="http://schemas.microsoft.com/ado/2009/02/edm/annotati
on" Name="InstanceContainer" m:IsDefaultEntityContainer="true"
annotation:LazyLoadingEnabled="false">
…
<AssociationSet Name="Customer_orders"
Association="northwind.Customer_orders">
<End EntitySet="Customer" Role="Customer"/>
<End EntitySet="Order" Role="Order"/>
</AssociationSet>
…
```

Cardinality

One of RDF's attractive features is that the model is not limited to the 3NF or above inherent in any RDBMS. However OData makes the following assumptions:

PROPERTIES

All properties take a single scalar (maybe nullable) value. If multiple values are required then this needs to be mapped to collections, which appeared in ODataV3

NAVIGATIONPROPERTIES

All navigation properties (aka objectproperties) require their cardinality to be defined. The cardinality of both ends of the association need to be specified as shown in this example:

```
<Association Name="managesOrder">
      <End Type="northwind.Employee" Multiplicity="0..1" Role="Employee"/>
      <End Type="northwind.Order" Multiplicity="*" Role="Order"/>
</Association>
```

OData will use this cardinality when returning its resultSets. For example if we are returning Orders associated with Employees, but there is only one order, OData will still return this as a list with one value. Conversely if we are returning Employees associated with an Order, OData will return this as a single value (or empty since it is optional).

This might appear obvious, but the distinction becomes important when using HTML5/JavaScript development environments that uses bindings based on the OData query path. If an HTML5 control, such as an Input cell, expects to be bound to a single value (such as Employee), it cannot be bound to multiple values. Similarly an HTML5 control, such as a table or list, expects to be bound to a list of values (such as an Employee's Orders) even if the list contains only one value.

OData2SPARQL will retrieve any defined cardinality from the model. The query that performs this deduction is the http://inova8.com/OData4sparql#associationQuery which returns the following

- ?maxDomainCardinality: the max cardinality associated with the domain of the property
- ?minDomainCardinality: the min cardinality associated with the domain of the property
- ?domainCardinality: the cardinality associated with the domain of the property
- ?maxRangeCardinality: the max cardinality associated with the range of the property

- ?minRangeCardinality: the min cardinality associated with the range of the property
- ?rangeCardinality: the cardinality associated with the range of the property

This query uses the following pattern to identify the cardinality:

```
OPTIONAL{
        ?domain rdfs:subClassOf ?minrdomainestriction .
        ?minrdomainestriction rdf:type owl:Restriction .
        ?minrdomainestriction owl:onProperty ?property .
        ?minrdomainestriction owl:minCardinality ?minDomainCardinality
}
```

Therefore OData2SPARQL assumes the cardinality will be declared explicitly within the ontology. Of course that will not always be the case, so OData2SPARQL will deduce the corresponding mappings as follows:

		Min-Cardinality		
		Undef	**0**	**1**
Max-Cardinality	**Undef**	<default>	0..*	1..*
	1	0..1	0..1	1..1
	*	0..*	0..*	1..*

Where the <default> takes the following values

- ObjectProperties: 0..1
- InverseProperties: 1..*

OData to SPARQL Query Mapping

The OData query consists of two parts:

- ResourcePath: this defines the particular entity or entityset that is to be return.
  - EntitySet , for example /Persons
  - Entity, for example /Persons('JohnDoe')
  - Entity/NavigationProperty, for example /Persons('JohnDoe')/children
- QueryOptions: these provide query options. Along with filtering and limiting the resultSet, OData query options include the ability to expand some entities, and define which properties of an entity to be returned.
  - $skip, will skip a number of entities, note not records
  - $top, will return only the top x entities
  - $expand, allows expanding a navigationproperty to include the resultant entities. Note that this can be nested. For example
    - $expand=children, would include the children
    - $expand =children($expand=children) would include the grandchildren
  - $select, specifies the properties to be include, rather than defaulting to all (or *)
  - $filter, specifies a condition on the properties that limits the resultset
  - $orderby, specifies the order that the resultSet will be returned

ResourcePath mapping to SPARQL

The function of the ResoourecPath is to designate the entityset or Entity that is to be explored further. The equivalent SPARQL queries are as follows:

ENTITYSET RESOURCEPATH

OData: …/Entityset

SPARQL:

```
{
    ?entity ?property ?value .
    { #resourcePath
        ?entity a ?EntitySet .
        ?entityset (rdfs:subClassOf)* :EntitySet .
    }
}
```

Note that instead of specifying the SPARQL1.1 path pattern, one could list the classes that are subClassesOf :EntitySet since they are already declared in the $metadata, as follows

SPARQL:

```
{
    ?entity ?property ?value .
    { #resourcePath
        ?entity a ?EntitySet .
        VALUES(?entityset){ (:EntitySet)(:subClassOfEntitySet)…(:subClassOfEntitySet)}
    }
}
```

ENTITY RESOURCEPATH

OData: …/Entityset(entityKey)

SPARQL:

```
{
    ?entity ?property ?value .
    { #resourcePath
        VALUE(?entity){(:entityKey)} .
    }
}
```

ENTITY/NAVIGATIONPROPERTY RESOURCEPATH

OData: …/Entityset(entityKey)/navigationProperty

SPARQL:

```
{
    ?entity ?property ?value .
    { #resourcePath
        ?entityKey ?navigationProperty ?entity .
        VALUE(?entityKey){(:entityKey)} .
    }
}
```

Note that the navigationProperty is equivalent to an owl:ObjectProperty.

QueryOptions mapping to SPARQL

TOP AND SKIP MAPPING TO SPARQL

The $top and $skip options limit the number of entities returned. Therefore these options are applied to the resourcePath SPARQL as follows:

```
OData: …&$top=:top&$skip=:skip
SPARQL:
        {
                #resourcePath graph pattern
        }
                LIMIT :top OFFSET :skip
```

Note that these options only apply when the resourcePath will return multiple entities. That is either an Entityset, or an Entity/navigationProperty that returns multiple values.

EXPAND MAPPING TO SPARQL

```
OData: …&$expand=navProp1($expand=navProp11…),navProp2…
SPARQL:
        {
                ?entity ?property ?value .

                OPTIONAL{ #expandLevel1
                        ?entity ?navProp1 ?entitynavProp1 .
                        ?entitynavProp1 ?entitynavProp1_property ?entitynavProp1_value .
                        OPTIONAL{ #expandLevel2
                                ?entitynavProp1 ?navProp11 ?entitynavProp1navProp11 .
                                ?entitynavProp1navProp11 ?entitynavProp1navProp11_property
                        ?entitynavProp1navProp11_value
                                {
                                        …
                                }
                        }
                }
                OPTIONAL{ #expandLevel1
                        ?entity ?navProp2 ?entitynavProp2 .
                        ?entitynavProp2 ?entitynavProp2_property ?entitynavProp2_value .
                        OPTIONAL{ #expandLevel2
                                …
                        }
                }
                …
                {
                        #resourcePath graph pattern
                }
        }
```

Note that the expand graph patterns are qualified as 'OPIONAL' because we would not want to restrict the entity resultSet to be just those that have all of the expanded terms. However this will be modified later when a $filter is applied to properties of expanded entities. In these cases it is assumed that the property value, and hence the expanded entity, must exist to participate in the filter condition

SELECT MAPPING TO SPARQL

So far it has been assumed that all properties of an entity, or equivalently all DatatypeProperties of a resource, will be returned. In practice the returned datatypeProperties are restricted to those that belong to the entityset.

Thus the general graph pattern is modified to limit the properties returned to those that have been designated in the $metadata to belong to the entitySet's EntityType or baseType of the EntityType. This is achieved by adding the VALUES(?property) clause as follows:

```
OData: …/Entityset(entityKey)/navigationProperty

SPARQL:
        {
                ?entity ?property ?value .
                VALUES(?property){(:property1),{:property2),….{:propertyN)} .
                {
                        #resourcePath
                }
        }
```

Now if the OData query is modified with a $select query option as follows, the corresponding SPARQL is modified to restrict the properties to the selection:

```
OData: …&$select=propertyA, propertyB

SPARQL:
        {
                ?entity ?property ?value .
                VALUES(?property){(:propertyA),{:propertyB)} .
                {
                        #resourcePath
                }
        }
```

This pattern also applies to the $select options applied to expanded entities.

FILTER MAPPING TO SPARQL

The objective of the $filter option is to restrict the entities returned in the resultSet to those that satisfy the query condition. Thus the filter is applied to the #resourcePath graph pattern as shown below:

```
OData: …&$filter=filterExpression(propertyA, propertyB, …)

SPARQL:
        {
                …
                {
                        #filterExpression(?propertyA_value, ?propertyB_value, …)
                        ?entity :propertyA ?propertyA_value .
                        ?entity :propertyB ?propertyB_value .
                        …

                        #resourcePath
                }
        }
```

Note that the corresponding filterExpression will use the values of the property found in the triple pattern of the form

```
?entity :property ?property_value .
```

This pattern also applies to the $filter options applied to expanded entities. However there are additional factors when a $filter is applied to an expanded entity:

1. The OPTIONAL applied to the expanded entity graph pattern cannot be applied
2. The expanded graph pattern needs to be included into the #resourcePath query as the filter needs to restrict the entitySets to when the $expand are applied

OData: …&$expand=navProp1($filter=filterExpression11(propertyA, propertyB, …))…
SPARQL:

```
{
    ?entity ?property ?value .
    { #expandLevel1
        ?entity ?navProp1 ?entitynavProp1 .
        ?entitynavProp1 ?entitynavProp1_property ?entitynavProp1_value .

        #filterExpression
        filterExpression(?entitynavProp1propertyA_value,
    ?entitynavProp1propertyB_value …)
        ?entitynavProp1 :propertyA ?entitynavProp1propertyA_value .
        ?entitynavProp1 :propertyB ?entitynavProp1propertyB_value .
        …
    }
    …
    {
        SELECT ?entity
        {
            { #expandLevel1
                ?entity ?navProp1 ?entitynavProp1 .
                #filterExpression
                filterExpression(?entitynavProp1propertyA_value,
            ?entitynavProp1propertyB_value …)
                ?entitynavProp1 :propertyA ?entitynavProp1propertyA_value .
                ?entitynavProp1 :propertyB ?entitynavProp1propertyB_value .
                …
            }

            #resourcePath graph pattern
        }
    }
}
```

Note that now a SELECT clause is added to the resourcePath query as we now want to restrict the resultSet from the resourcePath to those entities that satisfy the filterExpression condition.

Constructing the Results

SPARQL:

```
CONSTRUCT {
    #targetEntityIdentifier
    ?entity <http://targetEntity> true .
```

```
        #constructType
        ?entity a :EntitySet .
        #constructPath
        ?entity ?property ?property_value .
        #constructExpandSelect
}
```

Anatomy of Generated Queries

The generic structure of a generated SPARQL query mapped from an OData URI is as follows:

```
CONSTRUCT {
        #The collection that is being queried
        #The additional properties requested in the $expand section
WHERE {
        #select all triples about this instance
        #limit predicates to those specified in the metadata
        { #select entities of interest
        SELECT ?entity
                WHERE {
                #find associated entities corresponding to entity or entity set defined in the resourcepath
                }
        }
}
```

OData4SPARQL Vocabulary

The OData4SPARQL vocabulary adds some specific classes and properties necessary to model the datasets published by oadat2sparql beyond the imported vocabularies:

```
<http://inova8.com/OData4sparql>
  rdf:type owl:Ontology ;
  owl:imports <http://spinrdf.org/spin> ;
  owl:imports <http://spinrdf.org/spl> ;
  owl:imports <http://www.openrdf.org/config/sail> ;
  owl:imports <http://www.w3.org/2004/02/skos/core> ;
.
```

Classes

ODATA4SPARQL:DATASET

A dataset is the entity that is published by the OData service. As such it is also the location of various properties required to complete the definition of the service.

```
OData4sparql:Dataset
  rdf:type owl:Class ;
  rdfs:label "OData4Sparql Dataset" ;
  rdfs:subClassOf void:Dataset ;
.
```

ODATA4SPARQL:METADATA

An entity that acts as a container for a set of metadata queries used to initialize the service. OData4sparql:RDFSModel is predefined in <http://inova8.com/OData4sparql> and used as the default.

```
OData4sparql:Metadata
  rdf:type owl:Class ;
  rdfs:label "Metadata" ;
  rdfs:subClassOf owl:Thing ;
.
```

ODATA4SPARQL:OPERATION

An operation is a packaged SPARQL SELECT query that is published by the OData service that includes this operation within its vocabulary graphs. The SPARQL select is defined a SPIN property.

```
OData4sparql:Operation
  rdf:type owl:Class ;
  rdfs:label "Operation" ;
  rdfs:subClassOf spin:SelectTemplate ;
  rdfs:subClassOf owl:Thing ;
.
```

For a OData4sparql:Operation to be published in the OData2SPARQL metamodel, they need to be declared following the SPIN model for a spin:SelectTemplate, since OData4sparql:Operation are defined as a rdfs:subClassOf spin:SelectTemplate

- OData4sparql operations MUST have exactly one references to a base query
- OData4sparql operations MAY have one or multiple references to arguments (constraint), which become arguments within OData
- OData4sparql operations MUST have one or more variables in the sp:resultVariables each of which should define the corresponding spl:predicate so that this pseudo-entitySet can be linked to the remainder of the OData model
- a base query can be referenced by one or multiple OData4sparql operations

The SPIN vocabulary is being used for representing the OData4SPARQLOperation query objects. Since SPIN comes already with an RDF based representation, query objects are being stored as a set of triples.

Conventions

To be treated as a first class query object by the back-end, it must meet the following conditions:

- the base query MUST be typed at least with the SPIN superclass sp:Query and MUST have a property sp:text (storing the plain query string).
- the executable query template MUST be typed at least with the SPIN superclass OData4sparql:Operation and MUST have a property spin:body referencing the base query.
- the template arguments MUST be typed at least with the SPIN superclass spin:Argument and MUST have a property spl:predicate. The local name of the URI referenced by the spl:predicate determines the variable to be bound within the query.

- In the spirit of linked data, the SPIN vocabulary can be mixed with other vocabularies e.g. by adding comments or domain specific categorization using vocabularies such as SKOS.

ODATA4SPARQL:PREFIX

A prefix that contains both a URL and a prefix used to create QNames within models

```
OData4sparql:Prefix
  rdf:type owl:Class ;
  rdfs:comment "A prefix that contains both a URL and a prefix used to create
QNames within models" ;
  rdfs:label "Prefix" ;
  rdfs:subClassOf owl:Thing ;
.
```

ODATA4SPARQL:PROFILE

Instances of Profile are used to shape the generated SPARQL query, since different SPARQL endpoints behave differently (and adversely) to the same SPARQL.

```
OData4sparql:Profile
  rdf:type owl:Class ;
  rdfs:label "Profile" ;
  rdfs:subClassOf owl:Thing ;
.
```

Current instances include:

- OData4sparql:ALLEGROGRAPH ALLEGROGRAPH
- OData4sparql:DEFAULT DEFAULT
- OData4sparql:JENA JENA
- OData4sparql:SPARQL10 SPARQL10
- OData4sparql:SPARQL11 SPARQL11
- OData4sparql:TOPQUADRANT TOPQUADRANT
- OData4sparql:VIRTUOSO VIRTUOSO

Properties

ODATA4SPARQL:DATAREPOSITORY

The value of OData4sparql:dataRepository defines an instance of a Repository that provides the data for the service

```
OData4sparql:dataRepository
  rdf:type owl:ObjectProperty ;
  rdfs:domain OData4sparql:Dataset ;
  rdfs:label "data repository" ;
  rdfs:range rep:Repository ;
.
```

ODATA4SPARQL:INSERT-GRAPH-URI

Optional named graph URI to be used as the default graph for inserting statements. If not supplied data will be inserted into the default graph as defined by the endpoint URL. If that is not specified then RDF4J would use a context-less graph, so http://insert will be used as a default.

```
odata4sparql:insert-graph-uri
  rdf:type owl:ObjectProperty ;
  rdfs:comment "Optional named graph URI to be used as the default graph for
inserting statements. If not supplied data will be inserted into the default
graph as defined by the endpoint URL. If that is not specified then RDF4J
would use a context-less graph, so http://insert will be used as a default."
;
  rdfs:domain rep:RepositoryImpl ;
  rdfs:isDefinedBy <http://inova8.com/odata4sparql> ;
  rdfs:label "insert-graph-uri" ;
  rdfs:range xsd:anyURI ;
.
```

ODATA4SPARQL:CHANGE-GRAPH-URI

The value of odata4sparql:dataRepository defines an instance of a Repository that provides the data for the service.

```
odata4sparql:dataRepository
  rdf:type owl:ObjectProperty ;
  rdfs:comment "The value of odata4sparql:dataRepository defines an instance
of a Repository that provides the data for the service" ;
  rdfs:domain odata4sparql:Dataset ;
  rdfs:isDefinedBy <http://inova8.com/odata4sparql> ;
  rdfs:label "data repository" ;
  rdfs:range rep:Repository ;
.
```

ODATA4SPARQL:DATASETPREFIX

A OData4sparql:dataPrefix defines an instance of a prefix that will be used by the service of the dataset to which it is attached.

```
OData4sparql:datasetPrefix
  rdf:type owl:ObjectProperty ;
  rdfs:domain void:Dataset ;
  rdfs:label "dataset prefix" ;
  rdfs:range OData4sparql:Prefix ;
.
```

ODATA4SPARQL:DEFAULTPREFIX

A OData4sparql:defaultPrefix defines an instance of a prefix that will be used by the service of the dataset as the default prefix.

```
OData4sparql:defaultPrefix
  rdf:type owl:ObjectProperty ;
  rdfs:domain void:Dataset ;
  rdfs:label "default prefix" ;
  rdfs:range OData4sparql:Prefix ;
.
```

ODATA4SPARQL:DEFAULTQUERYLIMIT

A OData4sparql:defaultQueryLimit is an integer value used to limit the queries. It can be overridden by a $top OData query option defines an instance of a prefix that will be used by the service of the dataset as the default prefix.

```
OData4sparql:defaultQueryLimit
  rdf:type owl:DatatypeProperty ;
  rdfs:domain rep:RepositoryImpl ;
  rdfs:label "default query limit" ;
  rdfs:range xsd:integer ;
.
```

ODATA4SPARQL:NAMESPACE

A URI that declares the namespace to be used for the service.

```
OData4sparql:namespace
  rdf:type owl:DatatypeProperty ;
  rdfs:domain OData4sparql:Prefix ;
  rdfs:label "namespace" ;
  rdfs:range xsd:anyURI ;
.
```

ODATA4SPARQL:PREFIX

Determines the list of prefixes and corresponding namespaces to be used in the endpoint

```
OData4sparql:prefix
  rdf:type owl:DatatypeProperty ;
  rdfs:comment "Determines the list of prefixes and corresponding namespaces
to be used in the endpoint" ;
  rdfs:domain OData4sparql:Prefix ;
  rdfs:label "prefix" ;
  rdfs:range xsd:string ;
.
```

ODATA4SPARQL:SPARQLPROFILE

Profile of the SPAQL supported by this implementation. Does not belong with the dataset, as the data and vocabulary (ABox, TBox) can be via separate implementations

```
OData4sparql:sparqlProfile
  rdf:type owl:ObjectProperty ;
  rdfs:domain rep:RepositoryImpl ;
  rdfs:label "Profile of the SPARQL supported by this implementation. Does
not belong with the dataset, as the data and vocabulary (ABox, and TBox) can
be via separate implementations)" ;
  rdfs:range OData4sparql:Profile ;
.
```

ODATA4SPARQL:VOCABULARYREPOSITORY

The value of OData4sparql:vocabularyRepository defines an instance of a Repository that provides the vocabulary for the service

```
OData4sparql:vocabularyRepository
```

```
  rdf:type owl:ObjectProperty ;
  rdfs:domain OData4sparql:Dataset ;
  rdfs:label "vocabulary repository" ;
  rdfs:range rep:Repository ;
.
```

ODATA4SPARQL:VOCABULARYMETAMODEL

Defines which metaModel queries to be used which are in turn used to deduce the metadata that is to be published by the endpoint. The default is OData4sparql:OData4sparql:RDFSModel

```
OData4sparql:vocabularyMetaModel
  rdf:type owl:ObjectProperty ;
  rdfs:comment "Defines which metaModel queries to be used which are in turn
used to deduce the metadata that is to be published by the endpoint. The
default is OData4sparql:OData4sparql:RDFSModel" ;
  rdfs:domain OData4sparql:Dataset ;
  rdfs:isDefinedBy <http://inova8.com/OData4sparql> ;
  rdfs:label "vocabulary repository" ;
  rdfs:range OData4sparql:Metadata ;
.
```

ODATA4SPARQL:WITHRDFANNOTATIONS

Set to true if the OData metadata to be annotated with the RDF descriptions

```
odata4sparql:withRdfAnnotations
  rdf:type owl:DatatypeProperty ;
  rdfs:comment "true if the OData metadata to be annotated with the RDF
descriptions" ;
  rdfs:domain odata4sparql:Dataset ;
  rdfs:isDefinedBy <http://inova8.com/odata4sparql> ;
  rdfs:label "with RDF annotations" ;
  rdfs:range xsd:boolean .
```

ODATA4SPARQL:WITHSAPANNOTATIONS

Set to true if the OData metadata to be annotated with the SAP descriptions (see https://wiki.scn.sap.com/wiki/display/EmTech/SAP+Annotations+for+OData+Version+2.0)

```
odata4sparql:withSapAnnotations
  rdf:type owl:DatatypeProperty ;
  rdfs:comment "true if the OData metadata to be annotated with the SAP
descriptions" ;
  rdfs:domain odata4sparql:Dataset ;
  rdfs:isDefinedBy <http://inova8.com/odata4sparql> ;
  rdfs:label "with SAP annotations" ;
  rdfs:range xsd:boolean .
```

ODATA4SPARQL:WITHFKPROPERTIES

Set to true if property key fields should be created for navigation properties. This enables some reporting applications to more easily consume OData without relying on navigationProperty joins.

```
odata4sparql:withFKProperties
  rdf:type owl:DatatypeProperty ;
```

```
  rdfs:comment "true if property key fields should be created for navigation
properties. This enables some reporting applications to more easily consume
OData without relying on navigationProperty joins." ;
  rdfs:domain odata4sparql:Dataset ;
  rdfs:isDefinedBy <http://inova8.com/odata4sparql> ;
  rdfs:label "with FK properties" ;
  rdfs:range xsd:boolean ;
.
```

ODATA4SPARQL:WITHMATCHING

Set to true if it is intended that matching entities be merged together. The expression used for matching linksets is contained in the property odata4sparql:match

```
odata4sparql:withMatching
  rdf:type owl:DatatypeProperty ;
  rdfs:comment "true if it is intended that matching entities be merged
together. The expression used for matching linksets is contained in the
property odata4sparql:match" ;
  rdfs:domain odata4sparql:Dataset ;
  rdfs:isDefinedBy <http://inova8.com/odata4sparql> ;
  rdfs:label "with matching" ;
  rdfs:range xsd:boolean ;
.
```

ODATA4SPARQL:MATCH

Defines the match expression to be used within the generated SPARQL queries. The default pattern is { key1 (<http://www.w3.org/2004/02/skos/core#exactMatch> | ^ <http://www.w3.org/2004/02/skos/core#exactMatch>)* key2 }

```
odata4sparql:match
  rdf:type owl:DatatypeProperty ;
  rdfs:comment "Defines the match expression to be used within the generated
SPARQL queries. The default pattern is  { key1
(<http://www.w3.org/2004/02/skos/core#exactMatch> | ^
<http://www.w3.org/2004/02/skos/core#exactMatch>)* key2 }" ;
  rdfs:domain odata4sparql:Dataset ;
  rdfs:isDefinedBy <http://inova8.com/odata4sparql> ;
  rdfs:label "match" ;
  rdfs:range xsd:string ;
.
```

ODATA4SPARQL:USEBASETYPE

Set to true if the OData metadata to use baseType, false will flatten the classes so every class has its own primary key and full set of properties and associations.

```
odata4sparql:useBaseType
  rdf:type owl:DatatypeProperty ;
  rdfs:comment "true if the OData metadata to use baseType, false will
flatten the classes so every class has its own primary key and full set of
properties and associations" ;
  rdfs:domain odata4sparql:Dataset ;
  rdfs:isDefinedBy <http://inova8.com/odata4sparql> ;
```

```
        rdfs:label "use BaseType" ;
        rdfs:range xsd:boolean ;
      .
```

ODATA4SPARQL:EXPANDTOPDEFAULT

Assigns default value to $top if not explicitly defined in URL. Overcomes limitations of OpenUI5 that does not allow explicit inclusion

```
odata4sparql:expandTopDefault
  rdf:type owl:DatatypeProperty ;
  rdfs:comment "Assigns default value to $top if not explicitly defined in URL.
Overcomes limitations of OpenUI5 that does not allow explicit inclusion" ;
  rdfs:domain odata4sparql:Dataset ;
  rdfs:isDefinedBy <http://inova8.com/odata4sparql> ;
  rdfs:label "expand $top default" ;
  rdfs:range xsd:integer ;
.
```

ODATA4SPARQL:EXPANDSKIPDEFAULT

Assigns default value to $skip if not explicitly defined in URL. Overcomes limitations of OpenUI5 that does not allow explicit inclusion

```
odata4sparql:expandSkipDefault
  rdf:type owl:DatatypeProperty ;
  rdfs:comment "Assigns default value to $skip if not explicitly defined in URL.
Overcomes limitations of OpenUI5 that does not allow explicit inclusion" ;
  rdfs:domain odata4sparql:Dataset ;
  rdfs:isDefinedBy <http://inova8.com/odata4sparql> ;
  rdfs:label "expand $skip default" ;
  rdfs:range xsd:integer ;
.
```

ODATA4SPARQL:INSERT-GRAPH-URI

Optional named graph URI to be used as the default graph for inserting statements. If not supplied data will be inserted into the default graph as defined by the endpoint URL. If that is not specified then RDF4J will inserted into a context-less graph

```
odata4sparql:insert-graph-uri
  rdf:type owl:ObjectProperty ;
  rdfs:comment "Optional named graph URI to be used as the default graph for
inserting statements. If not supplied data will be inserted into the default graph
as defined by the endpoint URL. If that is not specified then RDF4J will inserted
into a context-less graph" ;
  rdfs:domain rep:RepositoryImpl ;
  rdfs:isDefinedBy <http://inova8.com/odata4sparql> ;
  rdfs:label "insert-graph-uri" ;
  rdfs:range xsd:anyURI ;
.
```

ODATA4SPARQL:CHANGE-GRAPH-URI

Optional named graph URI to be used as the default graph for tracking changes. Changes will use an extended http://topbraid.org/teamwork TopBraid TeamWork Framework ontology. If no change-graph-uri is specified then it is assumed that changes will *not* be tracked.

```
odata4sparql:change-graph-uri
  rdf:type owl:ObjectProperty ;
  rdfs:comment "Optional named graph URI to be used as the default graph for
tracking changes. Changes will use an extended http://topbraid.org/teamwork
TopBraid TeamWork Framework ontology. If no change-graph-uri is specified then it
is assumed that changes will *not* be tracked." ;
  rdfs:domain rep:RepositoryImpl ;
  rdfs:isDefinedBy <http://inova8.com/odata4sparql> ;
  rdfs:label "change-graph-uri" ;
  rdfs:range xsd:anyURI ;
.
```

ODATA4SPARQL:EXPANDOPERATIONS

Set to true if operations to be implicitly included in a $expand=* query request. It does not inhibit the explicit expansion of an operation.

```
odata4sparql:expandOperations
  rdf:type owl:DatatypeProperty ;
  rdfs:comment "True if operations to be implicitly included in a $expand=*
query request. It does not inhibit the explicit expansion of an operation" ;
  rdfs:domain odata4sparql:Dataset ;
  rdfs:isDefinedBy <http://inova8.com/odata4sparql> ;
  rdfs:label "expand Operations" ;
  rdfs:range xsd:boolean ;
.
```

ODATA4SPARQL:TEXTSEARCHTYPE

A class defining the different styles of text searching available to OData2SPARQL. The availability of different text search capabilities can result in different formulation of the SPARQL query from the OData request. The default is to use REGEX and CONTAINS SPARQL functions.

```
odata4sparql:TextSearchType
  rdf:type owl:Class ;
  rdfs:comment "A class defining the different styles of text searching
available to OData2SPARQL. The availability of different text search
capabilities can result in different formulation of the SPARQL query from the
OData request. The default is to use REGEX and CONTAINS SPARQL functions" ;
  rdfs:isDefinedBy <http://inova8.com/odata4sparql> ;
  rdfs:label "Text Search Type" ;
  rdfs:subClassOf void:Dataset ;
```

ODATA4SPARQL:BOTTOMUPSPARQLOPTIMIZATION

True if main entity or entityset query is repeated inside the expandWhere sub queries. This allows SPARQL to optimize the query because otherwise it can BIND the entity values prior to the subquery execution resulting it a full scan followed by a filter on the results. See https://github.com/blazegraph/database/wiki/SPARQL_Bottom_Up_Semantics

```
odata4sparql:bottomUpSPARQLOptimization
  rdf:type owl:DatatypeProperty ;
```

```
  rdfs:comment "True if main entity or entityset query is repeated inside the
expandWhere sub queries. This allows SPARQL to optimize the query because
otherwise it can BIND the entity values prior to the subquery execution
resulting it a full scan followed by a filter on the results. See
https://github.com/blazegraph/database/wiki/SPARQL_Bottom_Up_Semantics" ;
  rdfs:domain odata4sparql:Dataset ;
  rdfs:isDefinedBy <http://inova8.com/odata4sparql> ;
  rdfs:label "include implicit RDF" ;
  rdfs:range xsd:boolean ;
  rdfs:seeAlso
<https://github.com/blazegraph/database/wiki/SPARQL_Bottom_Up_Semantics> ;
```

TextSearchTypes

odata4sparql:RDF4J_Lucene

A data repository that supports the LuceneSAIL type of Text Searching (http://docs.rdf4j.org/programming/#_full_text_indexing_with_the_lucene_sail)

odata4sparql:Halyard_ElasticSearch

A data repository that supports the Halyard ElasticSearch type of Text Searching. For example ?subject ?predicate '<ElasticSearchQueryString^^<http://merck.github.io/Halyard/ns#search>

ODATA4SPARQL:INCLUDEIMPLICITRDF

True if RDF hasValues, and hasStatements are included in the published service. This can increase the burden when querying with $expand=*, but it does allow a 'schema-less' (aka RDF) model to be published. Default is false, meaning do not include.

```
 <owl:DatatypeProperty
rdf:about="http://inova8.com/odata4sparql#includeImplicitRDF">
    <rdfs:comment>True if RDF hasValues, and hasStatements are included in
the published service</rdfs:comment>
    <rdfs:label>include implicit RDF</rdfs:label>
    <rdfs:isDefinedBy rdf:resource="http://inova8.com/odata4sparql"/>
    <rdfs:range rdf:resource="http://www.w3.org/2001/XMLSchema#boolean"/>
    <rdfs:domain rdf:resource="http://inova8.com/odata4sparql#Dataset"/>
  </owl:DatatypeProperty>
```

ODATA4SPARQL:DELETEBODY

The DELETE{}WHERE{} SPARQL text for performing a delete action associated with the OData4SPARQL:Operation.

ODATA4SPARQL:INSERTBODY

The INSERT{}WHERE{} SPARQL for performing an insert action with the OData4SPARQL:Operation.

ODATA4SPARQL:UPDATEBODY

The DELETE{}INSERT{}WHERE{} SPARQL for performing an update action with the OData4SPARQL:Operation.

ODATA4SPARQL:UPDATEPROPERTYBODY

The DELETE{}INSERT{}WHERE{} SPARQL for performing an update of a simple property action with the OData4SPARQL:Operation.

HOW TO USE ODATA2SPARQL

The OData2SPARQL service obtains its configuration via the following steps:

1. Locate a repository of datasets
    a. On startup OData2SPARQL attempts to locate a remote repository which will contain the definitions of the datasets to be published.
    b. If no remote repository is located, OData2SPARQL will instead search the local folder within the Tomcat folder: <userdata>\inova8\OData2SPARQL\<repositoryFolder>\
        i. where <userdata> is
            1. Windows: <user>/Roaming/
            2. Linux: /var/opt/
        ii. And <repositoryFolder> is as declared defined in Web.xml param repositoryFolder or null if not defined.
    c. If no local repository is located, OData2SPARQL will use the default files located in this directory including models.ttl to initialize a local repository.
2. Query the Queries
    a. OData2SPARQL will first query the located repository which should contain the OData4sparql ontology for the Models and $metadata Inference queries.
    b. These queries are associated with an instance of OData4sparql:Metadata within the OData4sparql ontology. Currently there is one instance: OData4sparql:RDFSModel
3. Query the datasets
    a. OData2SPARQL will then use these queries to query the located repository for the dataset definitions using the OData Models Queries defined below.
4. OData2SPARQL then uses lazy-loading of the metadata associated with any dataset.
    a. When a request is made for a particular dataset, OData2SPARQL checks first to see if the details of that dataset have yet to be loaded.
    b. If not already loaded, OData2SPARQL will use the OData $metadata Inference Queries below, that were loaded in step 2 above, to infer the model to be published

Details of the queries that supply these details are in OData4SPARQL:Metadata Queries section of the Appendix

The steps to setup this configuration are described in the following sections:

1. Installation: Install the OData2SPARQL service
2. Service Configuration: Configure how the OData2SPARQL service will map from the underlying RDF triple stores
3. Models configuration: Configure the datasets that will be published by the service
4. Operation configuration: Configure the operations to be published

Installing the OData2SPARQL service

The OData2SPARQL application is provided as a WAR file which can be installed into any compatible application server, such as TOMCAT. This WAR file has been optimized for Tomcat 8 and above, and requires Java8

Service Configuration

The odat2sparql service obtains its configuration directives from the OData4SPARQL file which contains the definition of the OData4SPARQL ontology.
This ontology describes the supporting information required for the OData2SPARQL service, such as how to define the repositories which will deliver the data and vocabulary. This definition in turn uses the http://www.openrdf.org/config/sail ontology.

Additionally this file defines the SPARQL queries that will be used to deduce the OData metadata from the triple store.

At present there is one set of Metadata queries that assumes the triple store is organized as an RDFS model, RDFSModel. Other sets of metadata model queries can be defined in this file.

Configuration File Locations

There are serveral locations within which the service will seek its configuration files:

<catalina>: the webapp location

<userdata>: The userdata folder where configuration files will be stored.
Windows: System.getenv("AppData")
Linux: /var/opt/

<catalina>\OData2SPARQL.v2\WEB-INF\classes

1. **logback.xml:** Contains configuration of the desired level of logging. Logs will be written to the default logging location on the server, in the case of Tomact this will be /logs/

<catalina>\OData2SPARQL.v2\WEB-INF\classes\ontologies

This contains copies of the following ontologies which are loaded to allow configuration of the odat2sparql service. These are installed as part of the WAR file but can be view.

1. **22-rdf-syntax-ns.ttl:** A copy of the RDF ontology
2. **odata4sparql.rdf:** The OData2SPARQL model containing the queries for extracting the odata metadata from the models specified in models.ttl
3. **odata4sparql.validation.ttl:** SPIN validation rules to check one's model.ttl for completeness
4. **rdf-schema.ttl:** RDFS ontology
5. **sail.rdf:** The SAIL model that complements the RDF4J RDF mediation
6. **sp.ttl:** The SPIN vocabulary
7. **virtuoso.rdf:** A 'patch' for Virtuoso so that is recognizes owl:Class as a subClassOf rdfs:Class

<userdata>\inova8\OData2SPARQL\<repositoryFolder>

1. **Models.ttl:** This contains the model configuration file which specifies which endpoints will be published, where the data will be found, and from where the corresponding metadata should be extracted.

In order to allow multiple concurrent OData2SPARQL.v4 services running simultaneously, V4 includes in the web.xml a parameter 'repositoryFolder' whose default value is V4. If it is required that multiple OData2SPARQL.v4 services run concurrently, then each web.xml should be modified to reference a different repositoryFolder. Without this the subsequent services will report a lock on the folder's contents and will fail to start.

<userdata>\inova8\OData2SPARQL\<repositoryFolder>\repositories

This is the preferred location within which OData2SPARQL will build its persistent configuration model. Note that this will create a subfolder /system within which there maybe another folder called /lock. When the service is running this will contain lock files. When the service is stopped normally these lock files will be removed, allowing the service to restart. However if the service was stopped abnormally, these lock files will remain, preventing the service restarting. Simple remove these lock files, and then attempt to restart.

Models Configuration

The service is configured via the models.ttl file. This configuration file uses the oadat4sparql vocabulary, which in turn uses the RDF4J SAIL vocabulary.

Models Configuration File

The Models file contains the configuration details of the SPARQL sources that will be published as OData services.

The Models file is an RDFS/OWL ontology using the http://inova8.com/OData4sparql vocabulary. In principle any of the serialization formats for RDF can be used such as RDF/XML, but in practice the Turtle TTL format is much easier to read and edit.

As a RDF/OWL file editors such as Protégé or TopBraidComposer can be used to facilitate editing of this file. However the structure is simple enough to use any text editor.

Data and Vocabulary Repositories

When configuring an OData2SPARQL service, the configuration requires two repositories to be defined (they can be the same) for each endpoint to be published:

- **Data Repository**: this is the repository that contains all of the data, and will be the repository queried by an OData request. If no Vocabulary repository is defined, then this is also the repository from which the OData $metadata model will be inferred. The inference of the $metadata will search all the triples for any declared classes and properties to be published.
- **Vocabulary Repository**: this is the repository from which the OData $metadata model will be inferred using the queries defined in the OData4sparql.rdf. Since this Vocabulary Repository can be distinct from the data repository, and indeed be hosted in an entirely separate triple store if required. The Vocabulary repository needs to declare the following:
    - RDFS/OWL Classes which are mapped to OData entitypes and entitysets.
    - RDFS/OWL Data properties which are mapped to OData properties.
    - RDFS/OWL Object properties which are mapped to OData navigation properties and association sets.
    - SPIN templates which are mapped to OData function imports or operations.

  Thus it is possible to tailor the model that is published via the OData $metadata by what is defined in the Vocabulary Repository

ODATA SERVICE DECLARATION

The first section defines a particular OData service as OData4sparql:Dataset, a subClassOf void:Dataset

As part of this definition, we will need to define how that OData service will get its data and vocabulary. This declaration is done via the OpenRdf concept of a 'Repository'.

Two repositories are require: a repository that is the source of the data, and (optionally) a repository that is the source of the vocabulary.

```
# Declaration of the endpoint http://<base>/sparql/2.0/<ODataName>
# For example <ODataName> = NW
OData4sparql:<ODataName>
  rdf:type OData4sparql:Dataset ;
  rdf:type OData4sparql:Prefix ;
  # Declaration of the SPARQL source of data <Datasource>
  # For example <Datasource> = NW_Data
  OData4sparql:dataRepository :<Datasource>;
  # Declaration of any prefixes that make the OData response more readable
  OData4sparql:datasetPrefix OData4sparql:NW ;
  OData4sparql:datasetPrefix OData4sparql:OWL ;
```

```
  OData4sparql:datasetPrefix OData4sparql:RDF ;
  OData4sparql:datasetPrefix OData4sparql:RDFS ;
  OData4sparql:defaultPrefix OData4sparql:NW ;
  # Declaration of the namespace of the datasource <Namespace>
  # For example <Namespace> = <http://northwind.com/>
  OData4sparql:namespace <Namespace> ;
  OData4sparql:prefix "northwind"^^xsd:string ;
  # Declaration of the SPARQL source of vocabulary if not the <Datasource>
  # For example <Vocabulary> = NW_Vocabulary
  OData4sparql:vocabularyRepository :<Vocabulary> ;
  # Assign a label for the OData service
  rdfs:label <ODataName_Label>^^xsd:string ;
.
```

DATAREPOSITORY DECLARATION

Next we should define the implementation details of the dataRepository and the vocabularyRepository. First of all the dataRepository which will be responsible for delivery the results of the OData queries translated to SPARQL.

This declaration follows the OpenRdf definition of a repository. Currently these repositories are shown as openrdf:SPARQLRepository types of repository implementations, although in principle any of the available OpenRdf repository types could be used.

```
:<Datasource>
  rdf:type rep:Repository ;
  rep:repositoryID "<Datasource>"^^xsd:string ;
  # Definition of the implementation of the Repository <Datasource_Impl>
  # For example <Datasource_Impl> = NW_Data_Impl
  rep:repositoryImpl :<Datasource_Impl> ;
  rdfs:label "<Datasource>"^^xsd:string ;
.
:<Datasource_Impl>
  rdf:type openrdf:SPARQLRepository ;
  OData4sparql:defaultQueryLimit 1000 ;
  OData4sparql:sparqlProfile OData4sparql:DEFAULT ;
  rep:repositoryType openrdf:SPARQLRepository ;
  # Define the URLs of the SPARQL endpoints to be used for query and update
  # For example <data_query_endpoint> = <http://localhost:8890/sparql?format=xml>
  # and         <data_update_endpoint> = <http://localhost:8890/sparql?default-graph-
uri=http://northwind.com/>
  sparql:query-endpoint <data_query_endpoint> ;
  sparql:update-endpoint <data_update_endpoint> ;
  rdfs:label "<Datasource_Impl>"^^xsd:string ;
.
```

VOCABULARYREPOSITORY DECLARATION

Next is the dataRepository which is responsible for delivering the model details so that the OData metadata can be created. This can be the same as the dataRepository. However the way that this service deduces the model, it can be more performant to use a separate endpoint. Note that the service deduces the model by executing a sequence of SPARQL queries defined in the model file OData4sparql against the vocabularyRepository.

```
:<Vocabulary>
  rdf:type rep:Repository ;
  rep:repositoryID "<Vocabulary>"^^xsd:string ;
  # Definition of the implementation of the Repository <Vocabulary_Impl>
  # For example <Vocabulary_Impl> = NW_Vocabulary_Impl
  rep:repositoryImpl : <Vocabulary_Impl> ;
  rdfs:label "<Vocabulary>"^^xsd:string ;
.
```

```
:<Vocabulary_Impl>
  rdf:type openrdf:SPARQLRepository ;
  OData4sparql:sparqlProfile OData4sparql:DEFAULT ;
  rep:repositoryType openrdf:SPARQLRepository ;
  # Define the URLs of the SPARQL endpoints to be used for query and update
  # For example <vocabulary_query_endpoint> = <<http://localhost:8890/sparql?default-graph-
uri=http://northwind.com/&format=xml&timeout=0>
  # and         <vocabulary_update_endpoint> = <http://localhost:8890/sparql>
  sparql:query-endpoint <vocabulary_query_endpoint> ;
  sparql:update-endpoint <vocabulary_update_endpoint> ;
  rdfs:label "<Vocabulary_Impl>"^^xsd:string ;
.
```

ENDPOINT DECLARATIONS

There are some special consideration when defining the endpoint used for vocabulary and data access.

Different vocabulary and data endpoint

The reason for this is that the vocabulary endpoint, when queried using the metadata queries in OData4sparql.rdf, should respond with the correct classes, properties etc. This in turn depends on the graphs that are imported or available to that endpoint.

Using named-graph-uri URL parameter

Some triple stores do not explicitly support owl:imports. Therefore, even though a default-graph is importing the other graphs that are required to define the complete metadata, it might be necessary to add these as 'using-graph-uri' parameters.
An example of a triple store that does not support owl:imports is Virtuoso. In the case of Virtuoso the options are as follows:

- Option 1: Create a Virtuoso 'graph group' that contains all of the graphs to satisfy metadata or data queries.
- Option 2: Include all of the graphs necessary to satisfy metadata or data queries as part of the endpoint URL for the corresponding vocabulary or model implementations. For example

  http://localhost:8890/sparql?using-graph-uri=http://northwind.com/&using-graph-uri=http://www.w3.org/2000/01/rdf-schema%23&using-graph-uri=http://www.w3.org/1999/02/22-rdf-syntax-ns%23&using-graph-uri=http://www.w3.org/2004/02/skos/core#&format=xml&timeout=0

It is also important that the graphs referenced are actually available as triples within the triples-store. Access to the namespace is insufficient to resolve the metadata queries.

COMPLETE TEMPLATE FOR NEW SERVICE

A new datasource can be added to models.ttl by following the template below, replacing the highlighted fields with these values:

<ODataName>	The name for the OData service which will also be used as a prefix throughout. For example NW
<ODataNameSpace>	The namespace Uri for the data. For example <http://northwind.com/data/>
<ODataName_Data_URI>	The URL of SPARQL endpoint to access the data to be delivered by the OData service
<ODataName_Vocab_URI>	The URL of SPARQL endpoint to access the vocabulary to be delivered by the OData service

```
OData4sparql: <ODataName>
  rdf:type OData4sparql:Dataset ;
  rdf:type OData4sparql:Prefix ;
  OData4sparql:dataRepository :<ODataName>_Data ;
  OData4sparql:datasetPrefix OData4sparql:OWL ;
  OData4sparql:datasetPrefix OData4sparql:RDF ;
  OData4sparql:datasetPrefix OData4sparql:RDFS ;
  OData4sparql:defaultPrefix OData4sparql:ENV ;
  OData4sparql:namespace <ODataNameSpace>;
  OData4sparql:prefix "<ODataName>"^^xsd:string ;
  OData4sparql:vocabularyRepository :<ODataName>_Vocabulary ;
  rdfs:label "<ODataName>"^^xsd:string ;
.
:<ODataName>_Data
  rdf:type rep:Repository ;
  rep:repositoryID "<ODataName>_Data"^^xsd:string ;
  rep:repositoryImpl :<ODataName>_Data_Impl ;
  rdfs:label "<ODataName> Data"^^xsd:string ;
.
:<ODataName>_Data_Impl
  rdf:type openrdf:SPARQLRepository ;
  OData4sparql:defaultQueryLimit 1000 ;
  OData4sparql:sparqlProfile OData4sparql:DEFAULT;
  rep:repositoryType openrdf:SPARQLRepository ;
  sparql:query-endpoint <ODataName_Data_URI> ;
  sparql:update-endpoint <ODataName_Data_URI> ;
  rdfs:label "<ODataName> Data Impl"^^xsd:string ;
.
:<ODataName>_Vocabulary
  rdf:type rep:Repository ;
  rep:repositoryID "<ODataName>_Vocabulary"^^xsd:string ;
  rep:repositoryImpl :<ODataName>Vocabulary_Impl ;
  rdfs:label "<ODataName> Vocabulary"^^xsd:string ;
.
:<ODataName>_Vocabulary_Impl
  rdf:type openrdf:SPARQLRepository ;
  OData4sparql:sparqlProfile OData4sparql:DEFAULT;
  rep:repositoryType openrdf:SPARQLRepository ;
  sparql:query-endpoint <ODataName_Vocab_URI> ;
  sparql:update-endpoint <ODataName_Vocab_URI> ;
  rdfs:label "<ODataName> Vocabulary Impl"^^xsd:string ;
```

Model Specification Validation

1. Every dataset to be published must be of type odata4sparql:Dataset.
2. Every dataset must have a odata4sparql:dataRepository of type http://www.openrdf.org/config/repository#Repository.
3. Every dataset may have a odata4sparql:vocabularyRepository http://www.openrdf.org/config/repository#Repository.
4. Every dataRepository and vocabularyRepository must have a repository implementation of a certain type, usually http://www.openrdf.org#SPARQLRepository
5. Every repository implementation must have a definition of how to access the triplestore together with any parameters to limit to which graphs the queries will be limited.

These validations are included as spin:constraints in odata4sparql.validation. These can be run in TopBraidComposer via the Eclipse Problems view, with refresh  to reveal any problems as shown below:

Problems

5 errors, 0 warnings, 0 others

Description	Resource	Path	Location	Type
Errors (5 items)				
dataset missing vocabularyRepository	northwind.operations.validationtests.rdf	/inova8.northwind	odata4sparql:ODATA2SPARQL/odata4sparql:vocabularyRepository	SPIN Constraint
Operation must have a spin:body	northwind.operations.validationtests.rdf	/inova8.northwind	:OrderSummary_Error_MissingBody/spin:body	SPIN Constraint
Operation must have a spin:body/sp:text	northwind.operations.validationtests.rdf	/inova8.northwind	:OrderSummary_Error_MissingBodyText/sp:text	SPIN Constraint
Operation result variable missing predicate for variable ?order	northwind.operations.validationtests.rdf	/inova8.northwind	:OrderSummary_Error_MissingBodyText/spl:predicate	SPIN Constraint
Operation result variable missing predicate for variable ?order	northwind.operations.validationtests.rdf	/inova8.northwind	:OrderSummary_Error_MissingPredicate/spl:predicate	SPIN Constraint

FIGURE 17: VALIDATION PROBLEMS VIEW

Operations Configuration

SPARQL does not include the concept of database views as we find in SQL. However OData allows the declaration of 'Operations' that might or might not have side effects on the underlying database. By side-effects we mean that the invocation of the operation could cause the underlying data to change. With OData2SPARQL we restrict operations to those that have no such side-effects. In other words, the execution of canned database queries.

However OData2SPARQL does allow the association of insert, update, updateproperty, and delete SPARQL actions to be associated with each operation. (see Adding CRUD support to Operations)

The creation of an OData operation is equivalent to creating a pseudo-class within the model.

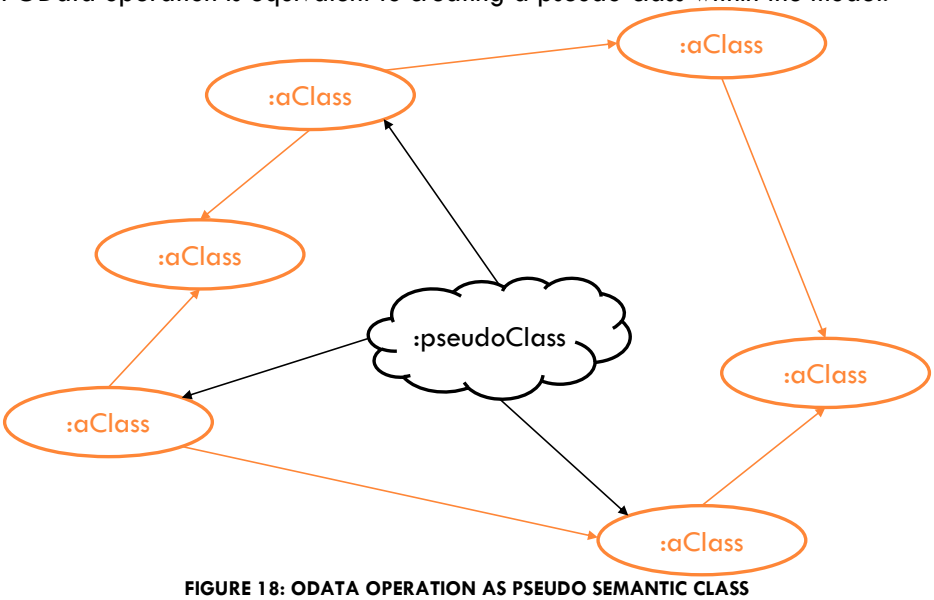


FIGURE 18: ODATA OPERATION AS PSEUDO SEMANTIC CLASS

Just like any other class, we would want instances of this class to be linked to other instances within the model. To do this we need to declare object- and datatype-properties for this pseudo-class. In that way we have a means to navigate from one instance to instances of the pseudo-class, or navigate from an instance of the pseudo-class to linked resources.

Operation Catalog

SPARQL queries can be stored as first class objects using the OData4sparql:Operation. This capability can be used to

- Create pseudo OData entitySets that aggregate data
- Create pseudo OData entitySets that transform the underlying semantic model into a different structure, such as hiding reified, but usually blank, objects.

Operation Configuration Example

As an example we might wish to embed the following query into the Northwind model.

```
select ?order ?prod (sum(?quantity * ( ?price * (1-?discount) )) as ?value)
where{
        ?orderdetail a <http://northwind.com/model/OrderDetail> .
        ?orderdetail <http://northwind.com/model/order> ?order .
        ?orderdetail <http://northwind.com/model/product> ?prod.
        ?orderdetail <http://northwind.com/model/quantity> ?quantity .
        ?orderdetail <http://northwind.com/model/unitPrice> ?price .
        ?orderdetail <http://northwind.com/model/discount> ?discount
} group by ?order ?prod
```

This SPARQL select query returns the total value of each product within a particular order. Both 'order' and 'product' would be instances of the existing Order and Product class, and therefore need to be linked via objectProperties to those classes. The 'value' would be an additional DatatypeProperty associated with this pseudo-class.

This information is shown diagrammatically below:

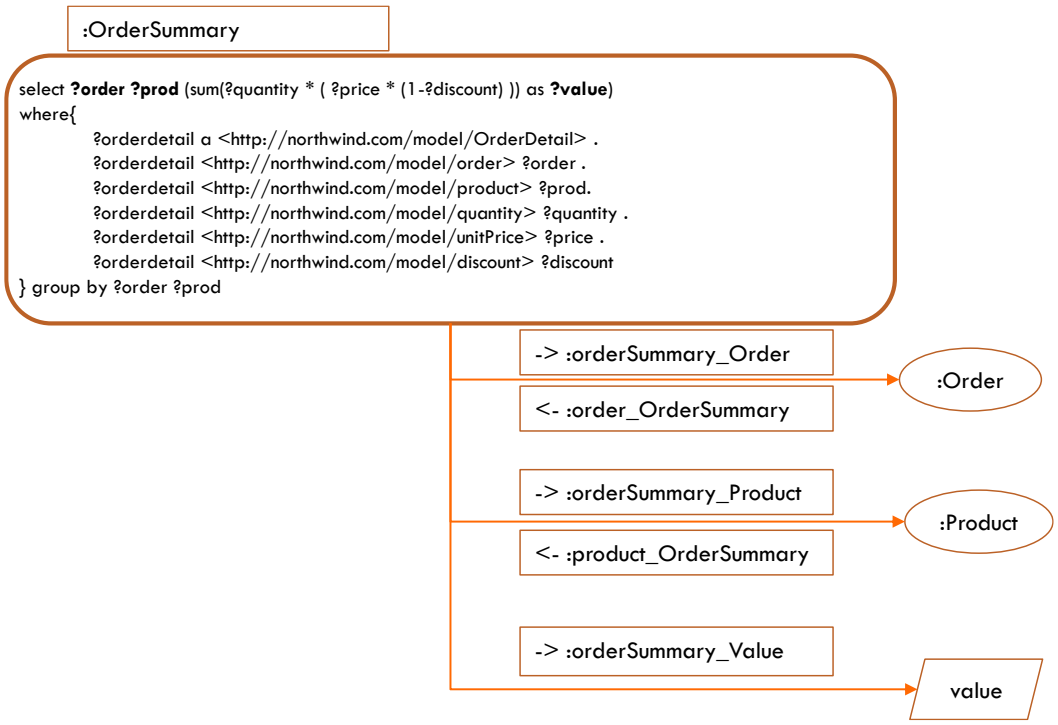


FIGURE 19: PSEUDO-CLASS WITH ADDITIONAL PREDICATES AND INVERSES

The procedure to declare this OData operation is as follows:

1. Create a new graph in the database where the operations can be defined. Note that this is not obligatory, but it does help to separate the definitions of the operations from the remainder of the vocabulary. An example would be
   `http://northwind.com/operations/`
2. Create and test the SPARQL query

```
select   ?order
         ?prod
         (sum(?quantity * ( ?price * (1-?discount) )) as ?value)
where{
   ?orderdetail a <http://northwind.com/model/OrderDetail> .
   ?orderdetail <http://northwind.com/model/order> ?order .
   ?orderdetail <http://northwind.com/model/product> ?prod.
   ?orderdetail <http://northwind.com/model/quantity> ?quantity .
   ?orderdetail <http://northwind.com/model/unitPrice> ?price .
   ?orderdetail <http://northwind.com/model/discount> ?discount
} group by ?order ?prod
```

3. Create a OData4sparql:Operation within the 'operations' ontology:

```
model:OrderSummary
   rdf:type OData4sparql:Operation .
```

4. For each selected resource in the select list create an objectProperty with the pseudo-class as its domain, and range of the class of the selected resource. Also create its corresponding inverseProperty so one can navigate in both directions:

```
   model:orderSummary_order
```

```
  rdf:type owl:ObjectProperty ;
  rdfs:domain model:OrderSummary ;
  rdfs:label "part of Order" ;
  rdfs:range model:Order ;
.

model:order_orderSummary
  rdf:type owl:ObjectProperty ;
  rdfs:label "order order summary" ;
  owl:inverseOf model:orderSummary_order ;
.
```

Note: OData2SPARQL will use any objectproperty value as part of the pseudo primary key of the operation entity. OData keys must be literals, so for each navigationProperty created, ODarta2SPARQL will create a corresponding property using the select variable name as the name of the property. Therefore do *not* make the navigationProperty name the same as the SPOARQL selected variable name

5. For each data value create a datatypeProperty with the pseudo-class as its domain.

```
model:orderSummary_Value
  rdf:type owl:DatatypeProperty ;
  rdfs:domain model:OrderSummary ;
  rdfs:label "order value" ;
  rdfs:range xsd:float ;
.
```

6. Add the SPARQL query as the spin:body value of rdf:type sp:Select. Note that this uses the SPIN vocabulary for storing the SPARQL select so it is best to use a SPIN enabled editor such as TopBraidComposer.

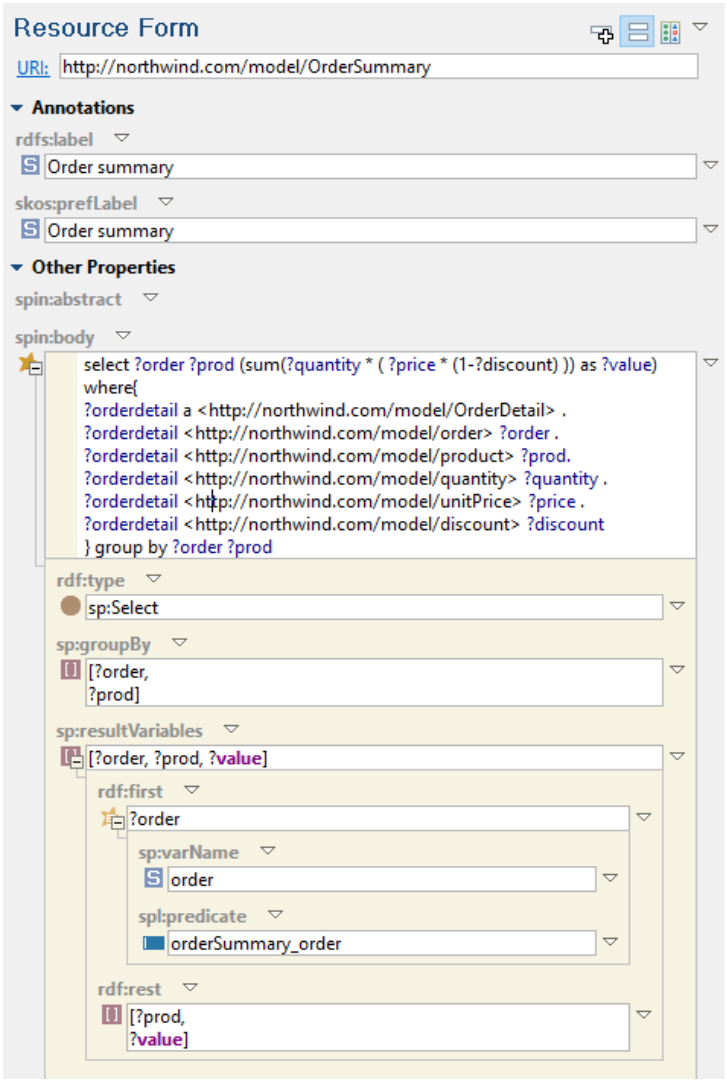


FIGURE 20: BUIDING AN ODATA OPERATION SP:SELECT

The text of the SPARQL select can be pasted into the spin:body, at which time TBC will parse the query and identify the sp:resultVariables as a rdf:List. It is important to identify the spl:predicate for each of the variables as shown in the snippet above.

7. Ensure that the vocabulary implementation in the models.ttl file has access to this new graph via its query-endpoint as shown below, where

```
:NW_Vocabulary_Impl
  rdf:type openrdf:SPARQLRepository ;
  rep:repositoryType openrdf:SPARQLRepository ;
  sparql:query-endpoint <http://localhost:8890/sparql?using-graph-
uri=http://northwind.com/&using-graph-
uri=http://www.w3.org/2000/01/rdf-schema%23&using-graph-
uri=http://www.w3.org/1999/02/22-rdf-syntax-ns%23&using-graph-
uri=http://inova8.com/OData4sparql/virtuoso&using-graph-
uri=http://northwind.com/operations/&timeout=0>
.
```

8. The operation will now appear in the OData metadata as follows:
   a. In service details:

```
<collection href="OrderSummary">
      <atom:title>OrderSummary</atom:title>
</collection>
```

b. In $metadata, note that the key is the combination of the resource variables in the select list.

```
<EntityType Name="OrderSummary">
   <Key>
         <PropertyRef Name="prod"/>
         <PropertyRef Name="order"/>
   </Key>
   <Property
         Name="prod"
         Type="Edm.String"
         Nullable="false"/>
   <Property
         Name="orderSummary_Value"
         Type="Edm.Double"
         Nullable="true"/>
   <Property
         Name="order"
         Type="Edm.String"
         Nullable="false"/>
   <NavigationProperty
         Name="orderSummary_product"
         Relationship="northwind.orderSummary_product"
         FromRole="OrderSummary"
         ToRole="Product"/>
   <NavigationProperty
         Name="orderSummary_order"
         Relationship="northwind.orderSummary_order"
         FromRole="OrderSummary"
         ToRole="Order"/>
</EntityType>
```

Parameterized Operation Configuration Example

Sometimes we might want to embed a query as an OData4sparql:Operation that accepts a parameter. An example would be a search query to which we pass the search string. An example of such a query is as follows:

```
select ?orderID ?shippedDate (sum(?quantity*?price*(1- ?discount )) as ?subTotal)
where{
      ?orderID  a <http://northwind.com/model/Order> .
      ?orderID  <http://northwind.com/model/shippedDate> ?shippedDate.
      ?orderdetail <http://northwind.com/model/order> ?orderID .
      ?orderdetail <http://northwind.com/model/quantity> ?quantity .
      ?orderdetail <http://northwind.com/model/unitPrice> ?price .
      ?orderdetail <http://northwind.com/model/discount> ?discount  .
      ?orderdetail <http://northwind.com/model/product> ?prod .
      ?prod rdfs:label ?prodlabel .
      FILTER(regex(?prodlabel, ?Summary_of_Sales_by_Years_wildcard)) .
}
group by ?orderID  ?shippedDate
```

This SPARQL select query returns the total value of a product within a particular order, where the poduct's label matches the ?Summary_of_Sales_by_Years_wildcard value via REGEX .

As before the ?orderID would be instances of the existing Order class, and therefore need to be linked via objectProperties to this class. The ?subTotal and ?shippedDate would be additional datatypeProperties associated with this pseudo-class.

The configuration of this operation proceeds identically to the previous example, however the new variable ?Summary_of_Sales_by_Years_wildcard needs to be treated differently.

Supplementary steps for configuring operation parameters

1. Define predicates for any embedded parameter:

```
model:Summary_of_Sales_by_Years_wildcard
  rdf:type owl:DatatypeProperty ;
  rdfs:label "Summary_of_Sales_by_Years_wildcard" ;
  rdfs:range xsd:string ;
.
```

   Note that OData2SPARQL uses the rdfs:label to identify the property, so in the above example `Summary_of_Sales_by_Years_wildcard` will be used.

   Note also there is no need to specify the rdfs:domain of this property.

2. Add the embedded parameter as a spin:constraint on the OData4sparql:Opereation

```
spin:constraint [
    rdf:type spl:Argument ;
    spl:predicate model:Summary_of_Sales_by_Years_wildcard ;
    spl:valueType xsd:string ;
  ] ;
```

   Note that this uses the SPIN vocabulary for storing the SPARQL select so it is best to use a SPIN enabled editor such as TopBraidComposer.

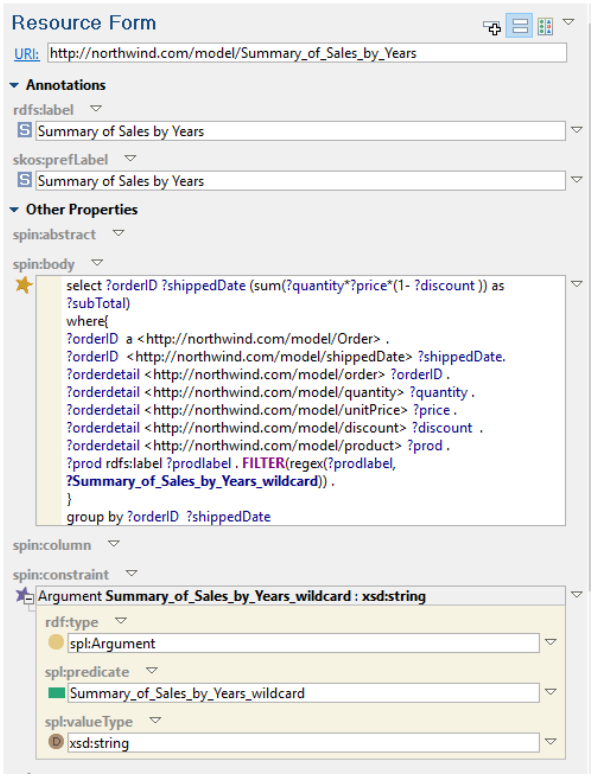


FIGURE 21: ADDING PARAMTER CONSTRAINT TO OPERATION

3. The operation will now appear in the OData metadata as follows:
   a. In service details:

```
<collection href="Summary_of_Sales_by_Years">
      <atom:title>Summary_of_Sales_by_Years</atom:title>
</collection>
```

   b. In $metadata, note that the key is the combination of the resource variables in the select list.

```
<EntityType Name="Summary_of_Sales_by_Years">
      <Key>
            <PropertyRef Name="orderID"/>
      </Key>
      <Property
            Name="Summary_of_Sales_by_Years_subTotal"
            Type="Edm.Double" Nullable="true"/>
      <Property
            Name="orderID"
            Type="Edm.String"
            Nullable="false"/>
      <Property
            Name="Summary_of_Sales_by_Years_shippedDate"
            Type="Edm.DateTime"
            Nullable="true"/>
      <NavigationProperty
            Name="Summary_of_Sales_by_Years_Order"
            Relationship="northwind.Summary_of_Sales_by_Years_Order"
            FromRole="Summary_of_Sales_by_Years"
            ToRole="Order"/>
```

```
</EntityType>
```

c. In addition a FunctionImport will appear within the EntityContainer of the $metadata:

```
<FunctionImport
      Name="Summary_of_Sales_by_Years"
      ReturnType="Collection(northwind.Summary_of_Sales_by_Years)"
      EntitySet="Summary_of_Sales_by_Years"
      m:HttpMethod="GET" IsBindable="true">
      <Parameter
            Name="Summary_of_Sales_by_Years_wildcard"
            Type="Edm.String" Nullable="false"/>
</FunctionImport>
```

4. This operation can be used within any OData URL as follows, note the assignment of a value to the ?Summary_of_Sales_by_Years_wildcard parameter :

```
../Summary_of_Sales_by_Years()?Summary_of_Sales_by_Years_wildcard='33'
```

Adding CRUD support to Operations

OData2SPARQL allows the addition of the following SPARQL actions to an operation:

- odata4sparql:deleteBody:
  - the DELETE{}WHERE{} SPARQL for performing a delete action
  - use ##DELETEVALUES template to provide the primary fields of the data to be deleted
- odata4sparql:insertBody:
  - the INSERT{}WHERE{} SPARQL for performing an insert action
  - use ##INSERTVALUES template to provide the key fields and attribute values to be inserted
- odata4sparql:updateBody:
  - the DELETE{}INSERT{}WHERE{} SPARQL for performing an update action
  - use ##INSERTVALUES template to provide the key fields and attribute values to which they should be updated
  - use ##DELETEVALUES template to provide the primary fields of the data to be updated
- odata4sparql:updatePropertyBody:
  - the DELETE{}INSERT{}WHERE{} SPARQL for performing an update of a simple property action
  - use DELETEVALUES template to provide the primary fields of the data to be updated
  - use ##UPDATEPROPERTYVALUES template to provide the new property, value pair to be updated

Each of the SPARQL bodies can reference the following templates whose purpose is to deliver VALUES values within the SPARQL statements.

The templates define the names of the variables that will be bound by OData2SPARQL. These should correspond to the variable names declared for the operation. The number of values should match the VALUES() statements otherwise a SPARQL error will result.

- ##INSERTVALUES(<list of valuables, eg ?arg>)##
  - Contains the new values of a pseudo-entity (operation) that are to be inserted or updated.
  - Each ?arg is substituted with the values posted. Otherwise UNDEF is substituted.

```
VALUES(?NEWcustomer  ?NEWcustomerCompanyName ?NEWcustomerCity  ?NEWorder ?NEWshipCity ){
##INSERTVALUES(?customer  ?customerCompanyName ?customerCity  ?order ?shipCity )##
}
```

  - Becomes

```
VALUES(?NEWcustomer  ?NEWcustomerCompanyName ?NEWcustomerCity  ?NEWorder ?NEWshipCity {
```

```
(<http://northwind.com/Customer-BERGS> "new customer name" "new customer city"
<http://northwind.com/Order-10280> "new ship city)
}
```

- ##DELETEVALUES(<list of valuables, eg ?arg>)##
  - Contains the primary key fields (at least) of a pseudo-entity (operation) that is to be updated, deleted, or a property deleted.
  - Each ?arg, if it is part of the primary key which it will be if it is an object rather than a literal value, is substituted with the values posted. Otherwise UNDEF is substituted.

```
VALUES(?OLDcustomer  ?OLDcustomerCompanyName ?OLDcustomerCity  ?OLDorder ?OLDshipCity ){
##DELETEVALUES(?customer  ?customerCompanyName ?customerCity  ?order ?shipCity )##
}
```

  - Becomes

```
VALUES(?OLDcustomer  ?OLDcustomerCompanyName ?OLDcustomerCity  ?OLDorder ?OLDshipCity ){
 (<http://northwind.com/Customer-BERGS> UNDEF UNDEF <http://northwind.com/Order-10280> UNDEF)
```

- ##UPDATEPROPERTYVALUES(?predicate ?value)##
  - The predefined arguments ?predicate and ?value are substituted with the predicate and its corresponding value as shown next

```
VALUES(?predicate  ?value){
##UPDATEPROPERTYVALUES(?predicate  ?value)##
}
```

  - Becomes

```
VALUES(?predicate  ?value){
 ( model:shipCity "London")
}
```

A detailed example is provided here: Example Updatable Operations

OData4sparql.Operation Example

The following turtle code exemplifies a complete SPIN query including an executable query template, an optional template argument as well as the base query:

```
:Summary_of_Sales_by_Years
        a       OData4sparql:Operation ;
       rdfs:label       "Summary of Sales by Years"^^xsd:string ;
       spin:body        [ a                   sp:Select ;
                         sp:groupBy          ( [ sp:varName  "orderID"^^xsd:string ] [ sp:varName
                                             "shippedDate"^^xsd:string ] ) ;
                         sp:resultVariables  ( [ sp:varName     "orderID"^^xsd:string ;
                                                 spl:predicate  :Summary_of_Sales_by_Years_Order
                                             ] [ sp:varName     "shippedDate"^^xsd:string ;
                                                   spl:predicate  :Summary_of_Sales_by_Years_shippedDate
                                                 ] [ sp:expression  [ a              sp:Sum ;
                                                                      sp:expression  [ a        sp:mul ;
sp:arg1  [ a        sp:mul ;
sp:arg1  [ sp:varName  "quantity"^^xsd:string ] ;
sp:arg2  [ sp:varName  "price"^^xsd:string ]
] ;                                                                                  sp:arg2  [ a
sp:sub ;                                                                                      sp:arg1  1 ;
sp:arg2  [ sp:varName  "discount"^^xsd:string ]
                                                                                   ]
                                                                                     ]
                                                                 ] ;
                                                sp:varName     "subTotal"^^xsd:string ;
```

```
                                                    spl:predicate  :Summary_of_Sales_by_Years_subTotal
                                                      ] ) ;
                              sp:text             "select ?orderID ?shippedDate (sum(?quantity*?price*(1-
?discount )) as ?subTotal)\r\nwhere{\r\n?orderID  a <http://northwind.com/model/Order> .\r\n?orderID
<http://northwind.com/model/shippedDate> ?shippedDate.\r\n?orderdetail <http://northwind.com/model/order>
?orderID .\r\n?orderdetail <http://northwind.com/model/quantity> ?quantity .\r\n?orderdetail
<http://northwind.com/model/unitPrice> ?price .\r\n?orderdetail <http://northwind.com/model/discount> ?discount
. \r\n?orderdetail <http://northwind.com/model/product> ?prod .\r\n?prod rdfs:label ?prodlabel .
FILTER(regex(?prodlabel, ?Summary_of_Sales_by_Years_wildcard)) .  \r\n} \r\ngroup by ?orderID
?shippedDate"^^xsd:string ;
        spin:constraint  [ a              spl:Argument ;
                           spl:predicate  :Summary_of_Sales_by_Years_wildcard ;
                           spl:valueType  xsd:string
                         ] ;
        skos:prefLabel   "Summary of Sales by Years"^^xsd:string .
```

Operation Specification Validation

1. Every operation to be published must be of type odata4SPARQL:Operation.
2. Every operation must have a spin:body.
3. Every operation's spin:body must have a sp:text of the query.
4. The query should avoid qnames, use full URL instead because those prefixes might not be available within the context that the query is run within OData
    a. If one really must use qnames make sure the prefixes are declared in the models.ttl
5. **DO NOT** include PREFIX statements within spin:body because the text will be merged within other SPARQL query, and SPARQL does not allow embedded PREFIX statements
6. **DO NOT** include LIMIT statements within spin:body because the text will be merged within other SPARQL query that will include any LIMIT statements
7. Every spin:body's sp:resultVariables needs a sp:varName and a spl:predicate defining the datatypeProperty or objectProperty of the variable.
    a. If you use TopBraidComposer and overwrite the sp:text, then you may lose the definitions of spl:predicate for each variable.

These validations are included as spin:constraints in odata4sparql.validation. These can be run in TopBraidComposer via the Eclipse Problems view, with refresh  to reveal any problems as shown below:

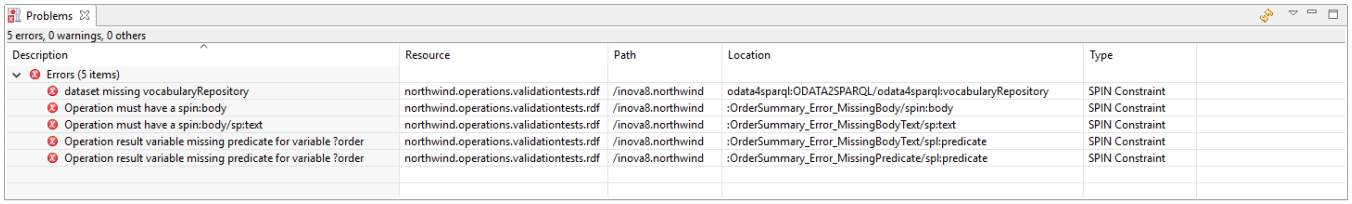


FIGURE 22: VALIDATION PROBLEMS VIEW

Managing the Service

$RESET and $RELOAD

When changes are made to the models.ttl configuration, or the source vocabularies, the OData2SPARQL service can be restarted to force a full refresh. This forces a reload of the models.ttl file, and also a refresh of the metadata loaded from the dataset vocabularies.

Alternatively the $RESET[/{dataset}] will reset the defined dataset and force a refresh from the dataset vocabularies without restarting the service

Additionally $RELOAD will clear all loaded metadata and reload the models.ttl file.

$DELTAS

Change logging can be activated by specifying a odata4sparql:change-graph-uri within the implementation section of the data repository. When activated all changes will be logged to the specified graph.

$deltas/{dataset}/clear
This option will clear the contents of the change-graph

$deltas/{dataset}/rollback (not yet implemented)
This option will roll back the changes specified in the change-graph

$deltas/{dataset}/commit (not yet implemented)
This option will commit the changes specified in the change graph, and if provenance is specified will add the change provenance to the graph. The contents of the change-graph will then be cleared.

Logging

OData2SPARQL logs its activity to the application server's (TOMCAT) logs folder to a file called OData2SPARQL.log. This log automatically rolls over to a date stamped version OData2SPARQL<date>.log.

The contents of the Logfile are controlled by the logback.xml found in the applications …/WEB-INF/classes folder. The root-level logging is set by default to "INFO".

Change Tracking

If a `odata4sparql:change-graph-uri` is specified for the data repository implementation, all changes submitted via OData2SPARQL will be tracked automatically. Changes will be logged to the graph specified in the `odata4sparql:change-graph-uri` parameter.

The change history can be managed using the following services:

$changes/Repository/Clear

This clears the change-graph, thus all records of the changes will be lost. Note that the changes themselves persist, all that is lost is the record of where what, and how.

$changes/Repository/Rollback

Rollback will iterate through the entire change graph and undo all the changes that have been logged. On completion of the rollback, the state of the graphs will have reverted to the state at the beginning of the change logging. Note that in the case of a store that is using multiple graphs, the changes will be restored to their originating graph.

$changes/Repository/Rollback?dateTime=<dateTime>

Rolls back all changes to the specified dateTime. ***Not currently deployed***

$changes/Repository/Rollback?change=<change>

Rolls back the change specified in the queryOption. A change can only be rolled back if it has not been subsequently changed. ***Not currently deployed***

$changes/Repository/Archive

Archive renames the current change graph by appending the current date/time to the graph. A new change graph is then initiated for subsequent changes

$changes/Repository/Compress

Compress removes all changes within the change history which have been subsequently changed within the same history. For example, a triple is added and then removed. This is useful for creating a record of the change between the start and end states of the graph, without any of the superfluous details of the path to get between the two states. . ***Not currently deployed***

KNOWN ISSUES

RDF to OData Metadata-mapping Issues

There can be situations where there is not perfect alignment between an ontology and the mapping to OData metadata. OData2SPARQL attempts to mitigate these issues as follows:

Properties with multiple domains

RDF independently describes properties from the classes, and then associates those properties to classes with either domain/range or restrictions. This means that the same property can be associated with more than one class. For example, drawn from the Northwind model

:Employee1 :orders :Order35

:Customer1 :orders :Order35

In each case the meaning of :orders is different. In the first example the meaning is that :Employee1 is responsible for orders including :Order35. In the second example the meaning is that :Cusomer1 has placed an order :Order35. From this description it is clear that the modeling is incorrect… we should use different predicates. However this is allowed in RDFS.

This problem will manifest itself if we wish to navigate from an order, say :Order35, to the customer who made the order. The navigation property will map to :orders, in which case we will retrieve both :Employee1 and :Customer1. We need more information to deduce that :Employee1 is not in fact the customer who ordered :Order35. One way would be to ensure that each potential customer is actually of type :Customer. However many would argue that asserting the class of each customer is taking away some of the advantages of the semantic data model: infer such information rather than asserting the fact.

If such inferencing is automatically taking place within the triple store being exposed by the SPARQL endpoint, then the OData2SPARQL query could be modified to include a check that the potential customer is indeed a :Customer.

However in OData2SPARQL a more pragmatic approach has been taken: it is assumed that good semantic modeling would imply that such ambiguities do not occur!

Classes with multiple super-classes

As of V4, OData only supports single inheritance via the BaseType attribute of an EntityType.

Incomplete Reasoning leading to incomplete results

IDEs such as Protégé and TopBraidComposer will display, for example, instances as a hierarchy of things that are subclassesOf owl:Thing. However these subClassOf relationships are often inferred and not asserted in the supplied data. This can lead to unexpected results such as:

- …/owl_Thing() returns no records
- …/skos_Concept() returns many records, especially when SKOS is included in the vocabulary.

This arises because OData2SPARQL will satisfy the …/owl_Thing() query by finding instances of owl:Thing and any instances of subClassOf owl:Thing. However if the subclasses have not been asserted then OData2SPARQL will return no results because no subclasses have been asserted.

The solution is to ensure that either inferencing is enabled on the triplestore, or such vocabulary relationships are asserted rather than relying on inferencing.

For example the following graphs could be included in the using-graph-uri:

- http://www.w3.org/1999/02/22-rdf-syntax-ns#
- http://www.w3.org/2000/01/rdf-schema#
- http://www.w3.org/2002/07/owl#
- http://www.w3.org/2004/02/skos/core#
    - Note that subClassOf not explicitly defined. However the TopBraidComposer version does include these additional triples.
- http://inova8.com/OData4sparql/virtuoso
    - Adds some additional subClassOf statements

Usage within Application Issues

Different applications use, or in some cases do not use, different features of the OData protocol

Spotfire

SPOTFIRE ODATA QUERY 'BUILDER' DOES NOT SUPPORT NAVIGATION PROPERTY NAVIGATION
Applications such as Excel PowerQuery recognize navigation properties (aka ObjectPropeties in OWL-speak) and allow a query to be constructed by navigating from one entity to another entity or entity collection via such navigation properties.

Workaround: Spotfire does not, yet, support such query construction. Instead it allows 'joins' between keys in the various classes. These properties will be automatically added if the odata4sparql:withFKProperties is set to true in the models.ttl configuration. For each navigationproperty, OData2SPARQL will create a corresponding data property containing the QName of the key which can be used to join with the 'primary key' subjectId of another class.

SAP WebIDE

ODATA2SPARQL DOES NOT SUPPORT AN ENTITY REQUEST WITHIN A BATCH
An issue with the Olingo2 Java library (https://olingo.apache.org/) causes an error to be thrown when a request for a single entity is included within a batched request, and the key is not encoded. An example of such a request would be:

```
Category('NWD:Category-3')
```

This occurs because a library class (BatchTransformatorCommon) uses java.net.URI.URI(String str) to validate the URI. This will fail.

Workaround 1: Specify that the model should not use batch. If one is using a template to create the application this can be done by adding the value useBatch in the manifest as follows:

```
"models": {
        …
        "": {
                "dataSource": "mainService",
                "settings": {
```

```
                                    "metadataUrlParams": {
                                          "sap-documentation": "heading"
                                    },
                                    "defaultBindingMode": "TwoWay",
                                    "useBatch" : false
                              }
                        }
                  }
```

Workaround 2: Ensure that the key is encoded in a batch request. For example:

```
var encodedkey = key.toString().replace(":","%3A")
```

Workaround 3: Use mapping of '~' in qname rather than ':'

Rather than map a URI to a traditional qname such as

```
Category('NWD:Category-3')
```

In V4 we have the option to map to

```
Category('NWD~Category-3')
```

The advantage of using '~' is that it is a 'mark' that has no predefined meaning in a URI whilst ':' does.

SAPUI5 DOES NOT FULLY SUPPORT CLASSES DERIVED FROM A BASETYPE

OData allows an EntityType to be derived from another EntityType by defining the BaseType from which it is derived. In the case of OData2SPARQL all classes are derived from Resource.

However this means that when SAPUI5 looks within the metadata for an EntityType's key fields it will find none defined except for Resource and any SPIN queries defined as Operations.

Workaround: Replace use of model.createKey with explicit definition of the key. For example in Detail.controller.js created when using a CRUDMasterDetail template, replace

```
var sObjectPath = this.getModel().createKey("isCategoryOf", oParameter);
```

with

```
var sObjectPath = "Category('" +
      oParameter.subjectId.toString().replace(":","%3A") + "')";
```

Workaround: Use the option odata4sparql:useBaseType to 'flatten' the hierarchy. This will cause all EntityTypes to have their own key fields, and inherit their superclass dataproperties. However the navigation properties cannot be inherited because they must be unique for the whole schema.

Excel PowerQuery

EARLY VERSIONS OF POWERQUERY DO NOT SUPPORT EDM:DATE

Earlier versions of PowerQuery will throw an error if the published $metadata contains an EDM:date type of property. This occurs when the ontology uses the xsd:date range for a property.

It is known that PowerQuery Version: 2.26.4128.242 exhibits this problem, whilst Version: 2.48.4792.941 has resolved the problem.

Workaround 1: Upgrade to latest PowerQuery

Workaround 2: Replace in the model range xsd:date with xsd:dateTime which maps to an EDM:DateTimeOffset OData type.

GLOSSARY

The following is a catalog of the working terms and their associated definitions. These terms will probably change in the final version:

Term	Description	References
OBDA Ontology Based Data Access		
Mediator		
OWL QL		
Rewriting		
Entity Alignment	Aligning entities, usually using predicates such as owl:sameAs to show two or more differently identifies entities are in fact the same.	
Conjunctive Query		
Provenance		
Service Bus	Service enablement for providers and consumers of services	
Messaging	Event based communications between components (often distributed)	
Service registry	Repository for the publication and discovery of services	
Security infrastructure	Service authorization and authentication management	
Governance	Lifecycle management control and consistency	

BIBLIOGRAPHY

D2RQ: Accessing Relational Databases as Virtual RDF Graphs . (n.d.). Retrieved from D2RQ: http://d2rq.org/

Introduction to Microsoft Power Query for Excel. (2014). Retrieved from http://office.microsoft.com/: http://office.microsoft.com/en-us/excel-help/introduction-to-microsoft-power-query-for-excel-HA104003940.aspx

Kal Ahmed, G. (2013, April 23-24). *SPARQL / OData Interop*. Retrieved from W3C: http://www.w3.org/2013/04/odw/odw13_submission_4.pdf

Lawrence, P. (2012, May 13). *Data cathedrals versus information bazaars?* Retrieved from Inova8: http://inova8.com/joomla/index.php/blog/data-cathedrals-versus-information-bazaars

Linked Data. (n.d.). Retrieved from http://linkeddata.org/home

LINQPad. (n.d.). Retrieved from http://www.linqpad.net/: http://www.linqpad.net/

Makris, K., Gioldasis, N., Bikakis, N., & Christodoulakis, S. (March 26-30, 2012). SPARQL-RW: transparent query access over mapped RDF data sources. *Proceedings of the 15th International Conference on Extending Database Technology*, (pp. 610-613). Berlin, Germany.

Manu Sporny, D. L. (2014, January 16). *JSON-LD 1.0*. Retrieved from W3C: http://www.w3.org/TR/json-ld/

Markus Stocker, A. S. (2008.). *SPARQL basic graph pattern optimization using selectivity estimation*. ACM,.

Northwind database. (2011, Aug 12). Retrieved from Northwind database: http://northwinddatabase.codeplex.com/

OASIS Approves OData 4.0 Standards for an Open, Programmable Web. (2014, March 17). Retrieved from OASIS: https://www.oasis-open.org/news/pr/oasis-approves-odata-4-0-standards-for-an-open-programmable-web

OData Version 4.0 Part 3: Common Schema Definition Language (CSDL). (2014, February 24). Retrieved from OASIS: http://docs.oasis-open.org/odata/odata/v4.0/odata-v4.0-part3-csdl.html

Odata.org. (n.d.). Retrieved from Odata.org: http://www.odata.org

-ontop- is a platform to query databases as Virtual RDF Graphs using SPARQ. (n.d.). Retrieved from Ontop: http://ontop.inf.unibz.it/

OpenUI5. (n.d.). Retrieved from OpenUI5: http://sap.github.io/openui5/

PragmatiQa. (n.d.). Retrieved from PragmatiQa: http://pragmatiqa.com/

UI Development Toolkit for HTML5 Developer Center . (n.d.). Retrieved from http://scn.sap.com/community/developer-center/front-end

APPENDIX

OData4SPARQL:Metadata Queries

The queries required by the OData2SPARQL service are defined in the OData4sparql vocabulary as properties of an OData4SPARQL:Metadata instance.
Note that the default instance is OData2SPARQL:RDFSModel

OData Models Queries

The following queries are used to analyze the models.ttl ontology, which is used to define which datasets are to be published:

HTTP://INOVA8.COM/ODATA4SPARQL#PREFIXQUERY

Determines the list of prefixes and corresponding namespaces to be used in the endpoint

- ?prefix: the prefix
- ?namespace: the corresponding namespace

HTTP://INOVA8.COM/ODATA4SPARQL#REPOSITORYQUERY

Returns the details of each and every repository published in the models.ttl file:

- ?Dataset: The URI of the dataset
- ?DatasetName: the name assigned to the dataset
- ?defaultNamespace: the default namespace of the dataset
- ?defaultPrefix: the default prefix
- ?DataRepositoryID: the identifier of the data repository to be used to resolve data queries
- ?DataRepositoryImplType: the SAIL type of implementation (usually SPARQLEndpoint)
- ?DataRepositoryImplQueryEndpoint: the endpoint URL, including any graphs names etc, to be used for querying from the dataset
- ?DataRepositoryImplUpdateEndpoint: the endpoint URL, including any graphs names etc, to be used when updating the dataset
- ?DataRepositoryImplProfile: the profile of the datasource, used to modify some characteristics of the generated SPARQL queries. One of the following, DEFSAULT used if not specified :
    - OData4sparql:ALLEGROGRAPH ALLEGROGRAPH
    - OData4sparql:DEFAULT DEFAULT
    - OData4sparql:JENA JENA
    - OData4sparql:SPARQL10 SPARQL10
    - OData4sparql:SPARQL11 SPARQL11
    - OData4sparql:TOPQUADRANT TOPQUADRANT
    - OData4sparql:VIRTUOSO VIRTUOSO
- ?DataRepositoryImplQueryLimit: default query limit
- ?VocabularyRepositoryID: the identifier of the vocabulary repository to be used to resolve data queries
- ?VocabularyRepositoryImplType: the SAIL type of implementation (usually SPARQLEndpoint)
- ?VocabularyRepositoryImplQueryEndpoint: the endpoint URL, including any graphs names etc, to be used for querying from the dataset
- ?VocabularyRepositoryImplUpdateEndpoint: the endpoint URL, including any graphs names etc, to be used when updating the dataset
- ?VocabularyRepositoryImplProfile: the profile of the datasource, used to modify some characteristics of the generated SPARQL queries. One of the following, DEFSAULT used if not specified :
    - OData4sparql:ALLEGROGRAPH ALLEGROGRAPH
    - OData4sparql:DEFAULT DEFAULT

    - OData4sparql:JENA JENA
    - OData4sparql:SPARQL10 SPARQL10
    - OData4sparql:SPARQL11 SPARQL11
    - OData4sparql:TOPQUADRANT TOPQUADRANT
    - OData4sparql:VIRTUOSO VIRTUOSO
- ?VocabularyRepositoryImplQueryLimit: default query limit
- ?withRdfAnnotations: true if the OData metadata to be annotated with the RDF/RDFS/OWL descriptions
- ?withSapAnnotations: true if the OData metadata to be annotated with the SAP descriptions

HTTP://INOVA8.COM/ODATA4SPARQL#DATASETQUERY (DEPRECATED)

A query used to return the dataset definitions form the models.ttl file

- ?dataset: the id of the dataset
- ?datasourceSparqlEndpoint : the SPARQL endpoint including any additional parameters to define graphs to be used
- ?datasourceRepository:
- ?datasourceURL
- ?datasourceProfileName: the profile of the datasource, used to modify some characteristics of the generated SPARQL queries. One of the following, DEFSAULT used if not specified :
    - OData4sparql:ALLEGROGRAPH ALLEGROGRAPH
    - OData4sparql:DEFAULT DEFAULT
    - OData4sparql:JENA JENA
    - OData4sparql:SPARQL10 SPARQL10
    - OData4sparql:SPARQL11 SPARQL11
    - OData4sparql:TOPQUADRANT TOPQUADRANT
    - OData4sparql:VIRTUOSO VIRTUOSO
- ?vocabularyName

OData $metadata Inference Queries

The model is inferred from the underlying data using the SPARQL queries defined in the OData4sparql.rdf. Currently there is one set of model inference queries corresponding to RDFS+.

HTTP://INOVA8.COM/ODATA4SPARQL#ASSOCIATIONQUERY

Provides the definitions of the associations (aka navigationProperties or objectProperties) that will be published in OData endpoint:

- ?property: URI of the association, aka objectProperty
- ?propertyLabel: the name assigned to the property
- ?domain: the domain of the property
- ?multipleDomain: the count of the number of domains to which this property belongs
- ?range:the range of the property
- ?multipleRange: the count of the number of ranges to which this property belongs
- ?maxDomainCardinality: the max cardinality associated with the domain of the property
- ?minDomainCardinality: the min cardinality associated with the domain of the property
- ?domainCardinality: the cardinality associated with the domain of the property
- ?maxRangeCardinality: the max cardinality associated with the range of the property
- ?minRangeCardinality: the min cardinality associated with the range of the property
- ?rangeCardinality: the cardinality associated with the range of the property
- ?description: a description of the property

HTTP://INOVA8.COM/ODATA4SPARQL#CLASSQUERY

Provides the catalog of classes

- ?class: URI of the class, aka owl:Class or rdfs:Class
- ?classLabel: the name assigned to the class
- ?baseType: the base type (class) from which the class inherits
- ?description: a description of the class.

HTTP://INOVA8.COM/ODATA4SPARQL#DATATYPEQUERY (DEPRECATED)

A query that defines any additional datatypes

- ?datatype: a datatype URI

HTTP://INOVA8.COM/ODATA4SPARQL#GRAPHQUERY

Returns the identities of ontologies:

- ?graph: the graph URI

HTTP://INOVA8.COM/ODATA4SPARQL#INVERSEASSOCIATIONQUERY

Provides any inverse associations (aka navigationProperties or objectProperties)

- ?inverseProperty: URI of the association, aka objectProperty that is inverseOf another property
- ?inversePropertyLabel: the name assigned to the property
- ?property: the corresponding property of which this is the inverse
- ?domain: the domain of the property
- ?multipleDomain: the count of the number of ranges to which this property belongs
- ?range: the range of the property
- ?multipleRange: the count of the number of ranges to which this property belongs
- ?maxDomainCardinality: the max cardinality associated with the domain of the property
- ?minDomainCardinality: the min cardinality associated with the domain of the property
- ?domainCardinality: the cardinality associated with the domain of the property
- ?maxRangeCardinality: the max cardinality associated with the range of the property
- ?minRangeCardinality: the min cardinality associated with the range of the property
- ?rangeCardinality: the cardinality associated with the range of the property
- ?description: a description of the property

HTTP://INOVA8.COM/ODATA4SPARQL#OPERATIONARGUMENTQUERY

Provides a list of arguments (aka parameters) required for the operation query.

- ?query: the URI of the query
- ?varName: the name of the variable as it appears in the declared query
- ?property: the identity of the property to which this variable provides values
- ?range: the range of the property (note that the domain will always be the query pseudo-class)
- ?propertyLabel: the label of the property
- ?description: a description of the property

HTTP://INOVA8.COM/ODATA4SPARQL#OPERATIONASSOCIATIONRESULTQUERY

Provides a list of the results from the operation query that are associations (aka navigationProperties or objectProperties)

- ?query: the URI of the query
- ?varName: the name of the variable as it appears in the declared query
- ?property: the identity of the property to which this variable provides values
- ?propertyLabel: the label of the property
- ?range: the range of the property (note that the domain will always be the query pseudo-class)
- ?description: a description of the property

HTTP://INOVA8.COM/ODATA4SPARQL#OPERATIONPROPERTYRESULTQUERY

Provides a list of the results from the operation query that are properties (aka datatypeProperties)

- ?query: the URI of the query
- ?varName: the name of the variable as it appears in the declared query
- ?property: the identity of the property to which this variable provides values
- ?propertyLabel: the label of the property
- ?range: the range of the property (note that the domain will always be the query pseudo-class)
- ?description: a description of the property

HTTP://INOVA8.COM/ODATA4SPARQL#OPERATIONQUERY

Provides the text of any query to be published as an operation:

- ?query: the URI of the query
- ?queryLabel: a label assigned to the query
- ?queryText: the full text of the SPARQL select query
- ?description: a description of the property SPARQL select query

HTTP://INOVA8.COM/ODATA4SPARQL#PROPERTYQUERY

A group of queries to extract the details of any property (aka datatypeProperty) from the vocabulary. The reason that these are in separate fragments is that some SPARQL processor cannot compose an efficient query when combined into a single SPARQL query.

http://inova8.com/OData4sparql#propertyQuery_Cardinality

Provides the deduced cardinality for any property:

- ?property: the URI of the property (aka datatypeProperty)
- ?maxCardinality: the property's maximum cardinality
- ?minCardinality: the property's minimum cardinality
- ?cardinality: the property's cardinality

http://inova8.com/OData4sparql#propertyQuery_Domains

Deduces the property and domain

- ?property: the URI of the property (aka datatypeProperty)
- ?propertyLabel: a label assigned to the property
- ?domain: the domain of the property
- ?description: a description of the property
- ?propertyType: the property types, such as owl:annotation.

http://inova8.com/OData4sparql#propertyQuery_Ranges

Deduces the ranges of the properties

- ?property: the URI of the property (aka datatypeProperty)
- ?range: the range of the property

Example OData to SPARQL Mapping

Example: filtering and limited property selection

As an example, a typical requirement is to select some attributes of those entities that match some filter condition:

OData URL request:

```
/Customer()?$filter=substringof('San',city/value)&$select=companyName
```

C#/LINQ query:

```
from c in Customer
where c.city.value.Contains("San")
select new {c.companyName}
```

Lambda

```
Customer
  .Where (c => c.city.Contains ("San"))
  .Select (
    c =>
      new
      {
        companyName = c.companyName
      }
  )
```

SPARQL:

```
CONSTRUCT {?Customer_s ?Customer_p ?Customer_o .}
WHERE {
        #select all triples about this instance
        ?Customer_s ?Customer_p ?Customer_o .
        #limit predicates to those requested
        VALUES(?Customer_p){(northwind:companyName)}.
        #select entities of interest
        { SELECT ?Customer_s
                WHERE {
                        #limit to instances of the corresponding type
                        ?Customer_s a ?class .
                        ?class (rdfs:subClassOf)* northwind:Customer .
                        #add the filter condition
                        ?Customer_s northwind:city ?city_value .
```

```
                    FILTER (  CONTAINS( ?city_value , "San" ) )
                }
            }
        }
```

Example: aggregation

Another example is that of counting the entity instances:

OData URL request:

```
/Customer()/$count
```

C#/LINQ query:

```
(from c in Customer
select c).Count()
```

SPARQL:

```
SELECT (COUNT( DISTINCT ?Customer_s ) AS ?COUNT)
WHERE {
    #select entities of interest
    { SELECT ?Customer_s
        WHERE {
            #limit to instances of the corresponding type
            ?Customer_s a ?class .
            ?class (rdfs:subClassOf)* northwind:Customer .
        }
    }
}
```

As can be seen the SPARQL query follows exactly the same pattern as all other OData to SPARQL query mappings

Example: query limited to particular instances

Rather than filtering, we might want to query for a particular instance of an entity. The instance will be identified by its QName within the OData request, however the URL could be used as well.

OData URL request:

```
/Customer('northwind:FAMIA')
```

C#/LINQ query:

```
Customer.Where (c =>c.Id == "northwind:FAMIA")
```

Lambda

```
Customer
  .Where (c => (c.Id == "northwind:FAMIA"))
  .Select (
```

```
        c =>
          new
          {
            Customer_orders = c.Customer_orders
          }
      )
```

SPARQL:

```
CONSTRUCT {?Customer_s ?Customer_p ?Customer_o .}
WHERE {
        #select all triples about this instance
        ?Customer_s ?Customer_p ?Customer_o .
        #limit predicates to those specified in the metadata
        VALUES(?Customer_p){(rdfs:label)…(northwind:orders)}
        #select entities of interest
        { SELECT ?Customer_s
                WHERE {
                #provide the key for the entity
                VALUES(?Customer_s) {(northwind:FAMIA)}
                }
        }
}
```

Example: Navigating through object-properties

OData, like OWL, distinguishes between datatype and object properties. The former are simply the properties of an entity, whilst the latter are navigation properties that allow navigation through a foreign-key to another entity.

OData URL request:

```
/Customer('northwind:FAMIA')/Customer_orders
```

C#/LINQ query:

```
from c in Customer
where c.Id == "northwind:FAMIA"
select new {c.Customer_orders}
```

Lambda:

```
Customer
  .Where (c => (c.Id == "northwind:FAMIA"))
  .Select (
    c =>
      new
      {
        Customer_orders = c.Customer_orders
      }
```

)

SPARQL:

```
CONSTRUCT {?Order_s ?Order_p ?Order_o .}
WHERE {
        #select all triples about this instance
        ?Order_s ?Order_p ?Order_o .
        #limit predicates to those specified in the metadata
        VALUES(?Order_p){(rdfs:label)...(northwind:order_Details)} .
        #select entities of interest
        { SELECT ?Order_s
                WHERE {
                #find associated entity corresponding to entity
                northwind:FAMIA northwind:orders  ?Order_s .
                }
        }
}
```

Example: Expanding all object properties

The OData request in the previous example is limited to a single entity. Normally associated entities are returned, not inline, but as reference links to the details of the entity. If one wants these associated entities to be included inline then the request is as follows:

OData URL request:

```
/ Customer()?$expand=Customer_orders
```

SPARQL:

The query occurs in two stages. The first query determines the set of entities, as described in the prior examples. The second query finds the inline details.

```
CONSTRUCT {?Order_s ?Order_p ?Order_o .}
        WHERE {
        #select all triples about this instance
        ?Order_s ?Order_p ?Order_o .
        #limit predicates to those specified in the metadata or the query
        VALUES(?Order_p){(rdfs:label) ... (northwind:order_Details)} .
        #select entities of interest
        { SELECT ?Order_s
                WHERE {
                #provide entities to be expanded from previous query
                VALUES(?Order_s) {( northwind:Order_10650) ...(northwind:Order_10581)}
                }
        }
}
```

Example: Expanding collections using Lambda 'Any'

The OData request in the previous example are limited to values that are 'scalars'. Within RDF any entity can have zero, one or many instances of a predicate. Thus a more correct mapping is that all entity attributes are collections of values with zero, one, or many values.
This example queries for any customer with any value of city that contains the string "San".

OData URL request:

```
/Customer()?$filter=city/any(x:substringof('San',x))
```

Lambda:

```
Customer
  .Where (c => c.city.Any (x => x.value.Contains ("San")))
  .Select (c => c)
```

SPARQL:

```
CONSTRUCT {?Customer_s ?Customer_p ?Customer_o .}
        WHERE { ?Customer_s ?Customer_p ?Customer_o .
        { SELECT ?Customer_s
                WHERE {
                #limit to instances of the corresponding type
                ?Customer_s a ?class .
                ?class (rdfs:subClassOf)* northwind:Customer .
                #add the 'any' lambda filter condition
                ?Customer_s northwind:city ?city_value .
                FILTER (  ?city_value CONTAINS(  , "San" )  )
                }
        }
}
```

Example: Expanding collections using Lambda 'All'

The OData request in the previous example are limited to values that are 'scalars'. Within RDF any entity can have zero, one or many instances of a predicate. Thus a more correct mapping is that all entity attributes are collections of values with zero, one, or many values.

This example queries for any customer such that all values of city contain the string "San".

OData URL request:

```
/Customer()?$filter=city/all(x:substringof('San',x))
```

Lambda:

```
Customer
  .Where (c => c.city.All (x => x.Contains ("San")))
  .Select (c => c)
```

SPARQL:

```
CONSTRUCT {?Customer_s ?Customer_p ?Customer_o .}
        WHERE { ?Customer_s ?Customer_p ?Customer_o .
        { SELECT ?Customer_s
                WHERE {
                #limit to instances of the corresponding type
                ?Customer_s a ?class .
                ?class (rdfs:subClassOf)* northwind:Customer .
                #add the predicates required for the lambda filter
                ?Customer_s northwind:city ?city_value .
                }
                GROUP BY ?Customer_s
                #add the 'all' lambda filter condition. Uses arithmetic 'addition' of boolean conditions
                HAVING(AVG(IF( CONTAINS( ?city_v alue , "San" ) ,1,0))=1)
        }
}
```

Example Queries

The following illustrates the SPARQL generated for the corresponding OData query

NW/Customer?$top=3

```
CONSTRUCT {
        #targetEntityIdentifier
        ?Customer_s <http://targetEntity> true .
        #constructType
        ?Customer_s a <http://northwind.com/model/Customer> .
        #constructPath
        ?Customer_s ?Customer_p ?Customer_o .
        #constructExpandSelect
}
WHERE {
        #clausesPathProperties
        {
                ?Customer_s ?Customer_p ?Customer_o .
                FILTER(!isIRI(?Customer_o) && !isBLANK(?Customer_o))
        }
        #clausesExpandSelect
        #selectExpand
        {       SELECT
                        ?Customer_s
                WHERE {
                        #filter
                        #clausesFilter
                        #clausesExpandFilter
                        #selectPath
                        {       SELECT DISTINCT
                                        ?Customer_s
                                WHERE {
                                        #filter
                                        #clausesPath
                                        ?Customer_s <http://www.w3.org/1999/02/22-rdf-syntax-ns#type> ?class .
                                        ?class (<http://www.w3.org/2000/01/rdf-schema#subClassOf>)*
<http://northwind.com/model/Customer> .
                                        #clausesFilter
                                        #clausesExpandFilter
                                } GROUP BY ?Customer_s LIMIT 3
                        }
                }
        }
} LIMIT 10000
```

NW/Customer('NWD%3ACustomer-ALFKI')

```
CONSTRUCT {
        #targetEntityIdentifier
        ?Customer_s <http://targetEntity> true .
        #constructType
        ?Customer_s a <http://northwind.com/model/Customer> .
        #constructPath
        ?Customer_s ?Customer_p ?Customer_o .
        #constructExpandSelect
}
WHERE {
        #clausesPathProperties
        {
                ?Customer_s ?Customer_p ?Customer_o .
                FILTER(!isIRI(?Customer_o) && !isBLANK(?Customer_o))
        }
        #clausesExpandSelect
        #selectExpand
        {       SELECT
                        ?Customer_s
                WHERE {
                        #filter
                        #clausesPath
                        VALUES(?Customer_s) {(<http://northwind.com/Customer-ALFKI>)}
                        #clausesFilter
                        #clausesExpandFilter
                }
        }
} LIMIT 10000
```

Example Updatable Operations

Customer_Order

A merge of the customer and their order. Somewhat contrived as the navigation properties linking these two entities provide much the same information, but this example helps to illustrate updatable operations.

OPERATION QUERY

This query merges all of the orders made by a customer, assuming there can be 0, 1, or many orders per customer.

spin:body:

```
SELECT ?customer  ?customerCompanyName ?customerCity  ?order ?shipCity
WHERE{
?customer a  <http://northwind.com/model/Customer> .
OPTIONAL{?customer  <http://northwind.com/model/customerCompanyName> ?customerCompanyName }
OPTIONAL{?customer  <http://northwind.com/model/customerCity> ?customerCity }
OPTIONAL{ ?order <http://northwind.com/model/customer> ?customer .
        OPTIONAL{ ?order  <http://northwind.com/model/shipCity> ?shipCity  }
        }
}
```

sp:resultsVariables

sp:variable	spl:predicate	rdfs:range
?customer	model:customer_Order_customer	model:Customer
?customerCompanyName	model:customer_Order_customerCompanyName	xsd:string
?customerCity	model:customer_Order_customerCity	xsd:string
?order	model:customer_Order_order	model:Order
?shipCity	model:customer_Order_shipCity	xsd:string

OPERATION DELETE

Delete will only remove the order, not the customer. When an order is deleted all line items of that order will be deleted as well to avoid orphaned OrderDetails.

The values defining what is to be DELETED are substituted for the following string

```
##DELETEVALUES(?customer  ?customerCompanyName ?customerCity  ?order ?shipCity )##

(<http://northwind.com/Customer-BERGS> UNDEF UNDEF <http://northwind.com/Order-10280> UNDEF)
```

Note that the use of the ?OLD prefix for the variables is simply a convention adopted to clarify what is being deleted and what might be inserted. The advantage of this convention is seen in the UPDATE operation.

odata4sparql:deleteBody:

```
DELETE{
        ?OLDorder <http://northwind.com/model/customer> ?OLDcustomer  .
        ?OLDorder ?OLDpredicate ?OLDobject .
        ?OLDorder ?OLDorderPredicate ?OLDorderObject .
        ?OLDorderDetail ?OLDorderDetailPredicate ?OLDorderDetailObject .
}WHERE{
# Use VALUES to set the 'incoming' values posted
VALUES(?OLDcustomer  ?OLDcustomerCompanyName ?OLDcustomerCity  ?OLDorder ?OLDshipCity ){
##DELETEVALUES(?customer  ?customerCompanyName ?customerCity  ?order ?shipCity )##
(<http://northwind.com/Customer-BERGS> UNDEF UNDEF <http://northwind.com/Order-10280> UNDEF)
}
# Use the same core query graph pattern as the Operation Query
{       ?OLDcustomer a  <http://northwind.com/model/Customer> .
        OPTIONAL{?OLDcustomer  <http://northwind.com/model/customerCompanyName> ?OLDcustomerCompanyName }
        OPTIONAL{?OLDcustomer  <http://northwind.com/model/customerCity> ?OLDcustomerCity }
        OPTIONAL{ ?OLDorder <http://northwind.com/model/customer> ?OLDcustomer .
                OPTIONAL{ ?OLDorder  <http://northwind.com/model/shipCity> ?OLDshipCity  }
                }
        }
        # Find all order predicate statements that need to be deleted
```

```
        # and then union all statements related to dependent orderDetails.
        {
                ?OLDorder ?OLDorderPredicate ?OLDorderObject
        }UNION{
                ?OLDorderDetail <http://northwind.com/model/order> ?OLDorder .
                ?OLDorderDetail ?OLDorderDetailPredicate ?OLDorderDetailObject .
        }
}
```

OPERATION INSERT

Insert allows one to construct a customer and an order. The POST will be provided as a set of property values for the Customer_Order entity. These property values are used to construct the VALUES statement

The values defining what is to be DELETED are substituted for the following string

```
##INSERTVALUES(?customer  ?customerCompanyName ?customerCity  ?order ?shipCity )##

(<http://northwind.com/Customer-BERGS> UNDEF UNDEF <http://northwind.com/Order-10280> UNDEF)
```

Note that the use of the ?NEW prefix for the variables is simply a convention adopted to clarify what is being inserted and what might be deleted. The advantage of this convention is seen in the UPDATE operation.

odata4sparql:insertBody:

```
INSERT{
        # the graph pattern from which the query graph pattern selects its data
        ?NEWcustomer a  <http://northwind.com/model/Customer> .
        ?NEWcustomer  <http://northwind.com/model/customerCompanyName> ?NEWcustomerCompanyName .
        ?NEWcustomer  <http://northwind.com/model/customerCity> ?NEWcustomerCity .
        ?NEWorder <http://northwind.com/model/customer> ?NEWcustomer .
        ?NEWorder  <http://northwind.com/model/shipCity> ?NEWshipCity  .
}WHERE{
        # Use VALUES to set the 'incoming' values posted
        VALUES(?NEWcustomer  ?NEWcustomerCompanyName ?NEWcustomerCity  ?NEWorder ?NEWshipCity ){
        ##INSERTVALUES(?customer  ?customerCompanyName ?customerCity  ?order ?shipCity )##
        (<http://northwind.com/Customer-BERGS> UNDEF UNDEF <http://northwind.com/Order-10280> UNDEF)
        }
}
```

OPERATION UPDATE

Update allows an existing 'garph pattern' to be updated. In SPARQL an UPDAYTE operation is a combination of a DELRETE followed by an INSERT as follows:

The values defining what is to be DELETE'd and then INSERT'ed are substituted for the following string

```
##DELETEVALUES(?customer  ?customerCompanyName ?customerCity  ?order ?shipCity )##
(<http://northwind.com/Customer-BERGS> UNDEF UNDEF <http://northwind.com/Order-10280> UNDEF)

##INSERTVALUES(?customer  ?customerCompanyName ?customerCity  ?order ?shipCity )##
(<http://northwind.com/Customer-BERGS> UNDEF UNDEF <http://northwind.com/Order-10280> UNDEF)
```

Note that the use of the ?NEW and ?OLD prefix for the variables is simply a convention adopted to clarify what is being inserted and deleted.

odata4sparql:updateBody:

```
DELETE{
# Delete the graph pattern for the values that are going to be updated
        ?OLDcustomer  <http://northwind.com/model/customerCompanyName> ?OLDcustomerCompanyName  .
        ?OLDcustomer  <http://northwind.com/model/customerCity> ?OLDcustomerCity .
        ?OLDorder <http://northwind.com/model/customer> ?OLDcustomer  .
        ?OLDorder  <http://northwind.com/model/shipCity> ?OLDshipCity .
}INSERT{
# Insert the new values that are provided in the PUT
        ?NEWcustomer  <http://northwind.com/model/customerCompanyName> ?NEWcustomerCompanyName .
```

```
        ?NEWcustomer  <http://northwind.com/model/customerCity> ?NEWcustomerCity .
        ?NEWorder <http://northwind.com/model/customer> ?NEWcustomer .
        ?NEWorder  <http://northwind.com/model/shipCity> ?NEWshipCity  .
}WHERE{
# Declares the values but leave the non-primary key fields (non URI) UNDEF
        VALUES(?OLDcustomer  ?OLDcustomerCompanyName ?OLDcustomerCity  ?OLDorder ?OLDshipCity ){
        ##DELETEVALUES(?customer  ?customerCompanyName ?customerCity  ?order ?shipCity )##
        (<http://northwind.com/Customer-BERGS> UNDEF UNDEF <http://northwind.com/Order-10280> UNDEF)
        }
# Use VALUES to set the 'incoming' values posted
        VALUES(?NEWcustomer  ?NEWcustomerCompanyName ?NEWcustomerCity  ?NEWorder ?NEWshipCity ){
        ##INSERTVALUES(?customer  ?customerCompanyName ?customerCity  ?order ?shipCity )##
        (<http://northwind.com/Customer-BERGS> "customerCompanyName"  "customerCity"
        <http://northwind.com/Order-10280> "shipCity")
        }
# Use the same core query graph pattern as the Operation Query to find the URIs of things to be deleted
{       ?OLDcustomer a  <http://northwind.com/model/Customer> .
        OPTIONAL{?OLDcustomer  <http://northwind.com/model/customerCompanyName> ?OLDcustomerCompanyName }
        OPTIONAL{?OLDcustomer  <http://northwind.com/model/customerCity> ?OLDcustomerCity }
        OPTIONAL{ ?OLDorder <http://northwind.com/model/customer> ?OLDcustomer .
                OPTIONAL{ ?OLDorder  <http://northwind.com/model/shipCity> ?OLDshipCity  }
                }
        }
}
```

OPERATION UPDATE PROPERTY

This is an operation that permits the update of a single property associated with the operation. To be emulated in SPARQL it is necessary to locate the original statement(s), deleted them, and then insert their replacements. Thus the SPARQL query uses the supplied DELETEVALUES from which the variables that constitute the pseudo-primary key can be used to find the corresponding objects. Then the predicate is found. One point to note is that the application code will not know the subject associated with the predicate, therefore there is a section in the query that locates the predicate, and then BINDS the subject with the ?OLDsubject variable.

Note that this pattern will work for one or more predicate, value pairs.

odata4sparql:updatePropertyBody:

```
DELETE{
# Delete the statement(s)
        ?OLDsubject  ?property ?OLDvalue .
}INSERT{
# Insert  the updated statement
        ?OLDsubject  ?property ?value .
}WHERE{
# Declares the values but leave the non-primary key fields (non URI) UNDEF
VALUES(?OLDcustomer  ?OLDcustomerCompanyName ?OLDcustomerCity  ?OLDorder ?OLDshipCity ){
##DELETEVALUES(?customer  ?customerCompanyName ?customerCity  ?order ?shipCity )##
}
##MAP## operation properties to entity properties
VALUES(?predicate ?property){( <http://northwind.com/model/customer_Order_customerCity>
<http://northwind.com/model/customerCity>  )
(<http://northwind.com/model/customer_Order_customerCompanyName>
<http://northwind.com/model/customerCompanyName>)
(<http://northwind.com/model/customer_Order_shipCity> <http://northwind.com/model/shipCity> ) }
# Declares the predicate and value pair to be updated
VALUES(?predicate  ?value){
##UPDATEPROPERTYVALUES(?predicate  ?value)##
}
# Use the same core query graph pattern as the Operation Query to find the URIs of things to be deleted
{       ?OLDcustomer a  <http://northwind.com/model/Customer>.
        OPTIONAL{ ?OLDorder <http://northwind.com/model/customer> ?OLDcustomer .     }
}
# Find all property statements that need to be deleted
{
        {
                ?OLDorder  ?property ?OLDvalue
                BIND(  ?OLDorder as ?OLDsubject )
                }UNION{
                ?OLDcustomer  ?property ?OLDvalue
                BIND(  ?OLDcustomer as ?OLDsubject )
                }
        }
}
```