Exploiting the RDF-based Linked Data Web using .NET via LINQ

Introduction

Over the course of the last year, the Semantic Web has continued to gain prominence and begun to receive coverage in the broader media. Large RDF data sources are now being published and exposed publicly, providing a concrete manifestation of the progress being made. In tandem with the work being done on the server-side to support publishing of ever larger RDF datasets, client-side tools and APIs are continuing to evolve. Numerous RDF browsers are available which, although not always user-friendly, allow these RDF datasets to be explored. APIs such as Sesame (Java/JDBC-based), Jena (Java/JDBC-based) and Redland (C-based) provide a foundation for building traditional (i.e. non-browser based) desktop clients. However, because of their language bindings, these APIs predominantly favour non-Windows clients. One fledgling effort which attempts to counter this bias is LINQ to RDF from Andrew Matthews.

While Microsoft's ADO.NET Data Services (previously known as project Astoria) provides a Redmond vision of exposing data on the Web, it as yet has no support for RDF and is limited to Windows server platforms. LINQ to RDF (aka LinqToRdf) constitutes an early effort to fill this gap and provide a bridge between Windows desktop applications and the Semantic Web. In addition it introduces a facility often missing from existing Semantic Web APIs, the facility to handle RDF data as a collection of objects rather than raw triples. LinqToRdf reconstitutes fetched RDF data into entity instances which much better reflect the underlying object graph model being represented in RDF.

In this note, I provide an outline of LinqToRdf and a simple example of its use to retrieve data from the MusicBrainz music metadatabase via an OpenLink Virtuoso Quad Store. The primary intention is draw attention to this excellent effort and hopefully persuade some of you in the Semantic Web community to contribute to this endeavour. The example also illustrates the use of the Virtuoso Sponger, an "RDFizer" forming part of the RDF toolset furnished by OpenLink Virtuoso Universal Server, to convert the raw MusicBrainz data to RDF on-the-fly.

Resources

LinqToRdf is largely the effort of one individual, Andrew Matthews, though the lower levels of LinqToRdf rely on underpinnings provided by Joshua Tauberer's SemWeb library, a Semantic Web/RDF Library for C#/.NET. Background information and status updates can be found in Andrew's blog. The project itself is hosted on Google Code. Virtuoso is available as a closed source product or in open source form.

MusicBrainz As A Test Data Source

Our test data is taken from the MusicBrainz music database which captures information about artists, their recorded works, and the relationships between them. Entries for recorded works capture at a minimum the album title, track titles, and the length of each track. In MusicBrainz, artists always have a unique ID. So, for example, the URLs http://musicbrainz.org/artist/4d5447d7-c61c-4120-ba1b-d7f471d385b9.html or http://musicbrainz.org/artist/72c090b6-a68e-4cb9-b330-85278681a714.html take you directly to entries for John Lennon or John Cale respectively. Although you can navigate to these pages by entering the artist's name in the MusicBrainz search page, we use these URLs as our starting point.

If you were to look at one these pages in your browser, you would see that the information about the artist contains no RDF data. One method of generating the RDF representation we require is an RDFizer. The Virtuoso Sponger is one such tool which provides a pluggable architecture for RDFizers. Individual RDFizers, targetting different non-RDF data sources, are installed as Sponger Cartridges. Virtuoso provides many pre-written cartridges, for data sources such as Flickr, eBay and Amazon, or you can write your own. A pre-written cartridge is also available for MusicBrainz but, rather than use the regular version for the LinqToRdf example, we use a slightly modified version better suited to our requirements. More details of the modified sponger cartridge are presented later.

There are several ways to invoke the Sponger, supporting a range of application environments and consumers. Options include:

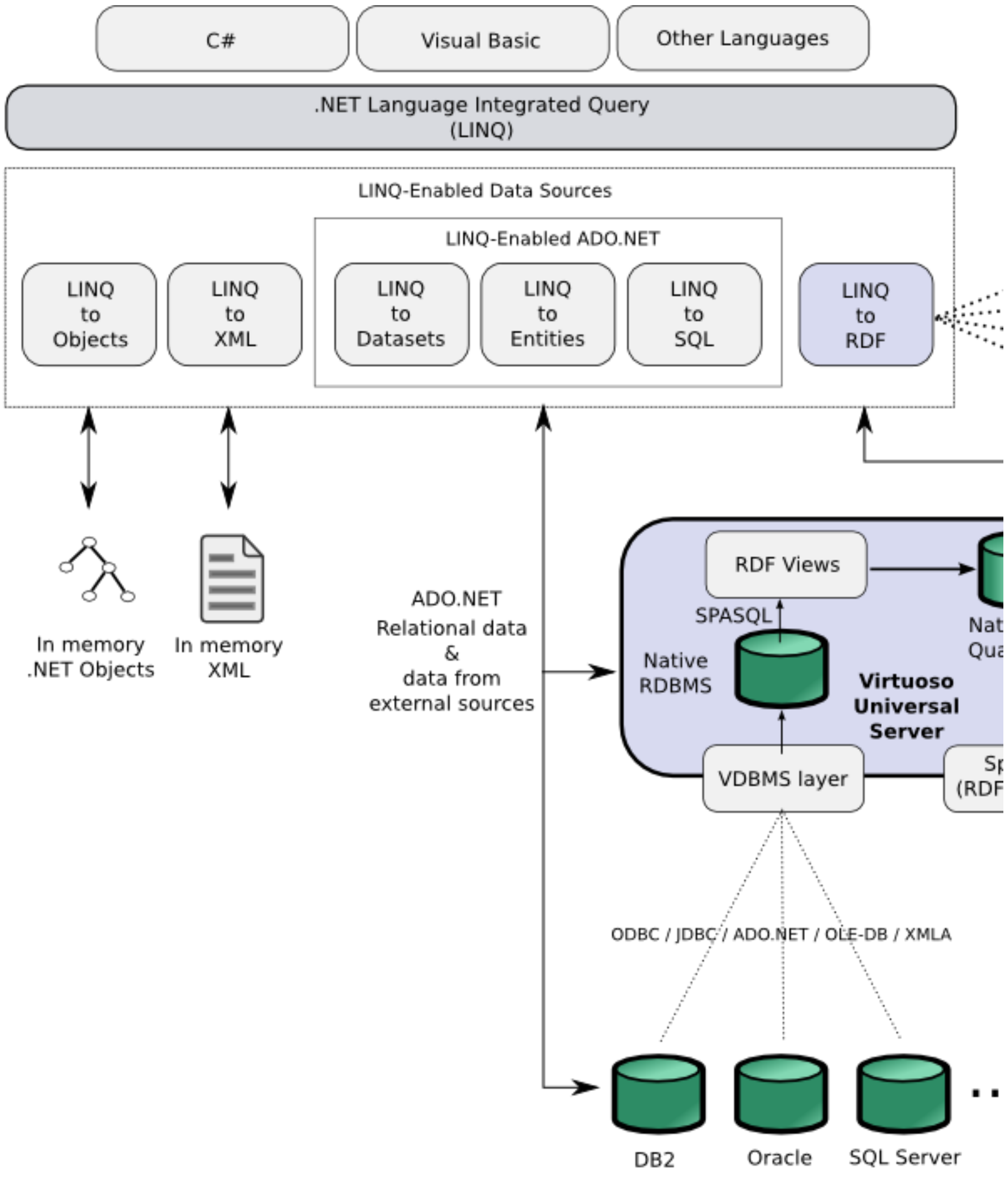
- Virtuoso SPARQL query processor
- Virtuoso RDF Proxy Service
- OpenLink RDF client applications
- ODS-Briefcase (ODS-Briefcase is a component of OpenLink Data Spaces)
- Directly via Virtuoso's SQL Procedure language Virtuoso/PL

The method we'll use is to invoke the MusicBrainz Sponger cartridge through the Virtuoso RDF proxy. This service is exposed at the "/proxy" endpoint of any Virtuoso installation (e.g. http://localhost:8890/proxy). To generate RDF data from the MusicBrainz entry for John Cale, we simply reference the appropriate MusicBrainz Web page via the RDF proxy. Assuming you have a Virtuoso instance installed locally, two ways to do this are:

- Paste the URL http://localhost:8890/proxy?force=rdf&url=http://musicbrainz.org/artist/72c090b6-a68e-4cb9-b330-85278681a714.html into your Web browser, or
- Use the curl command:

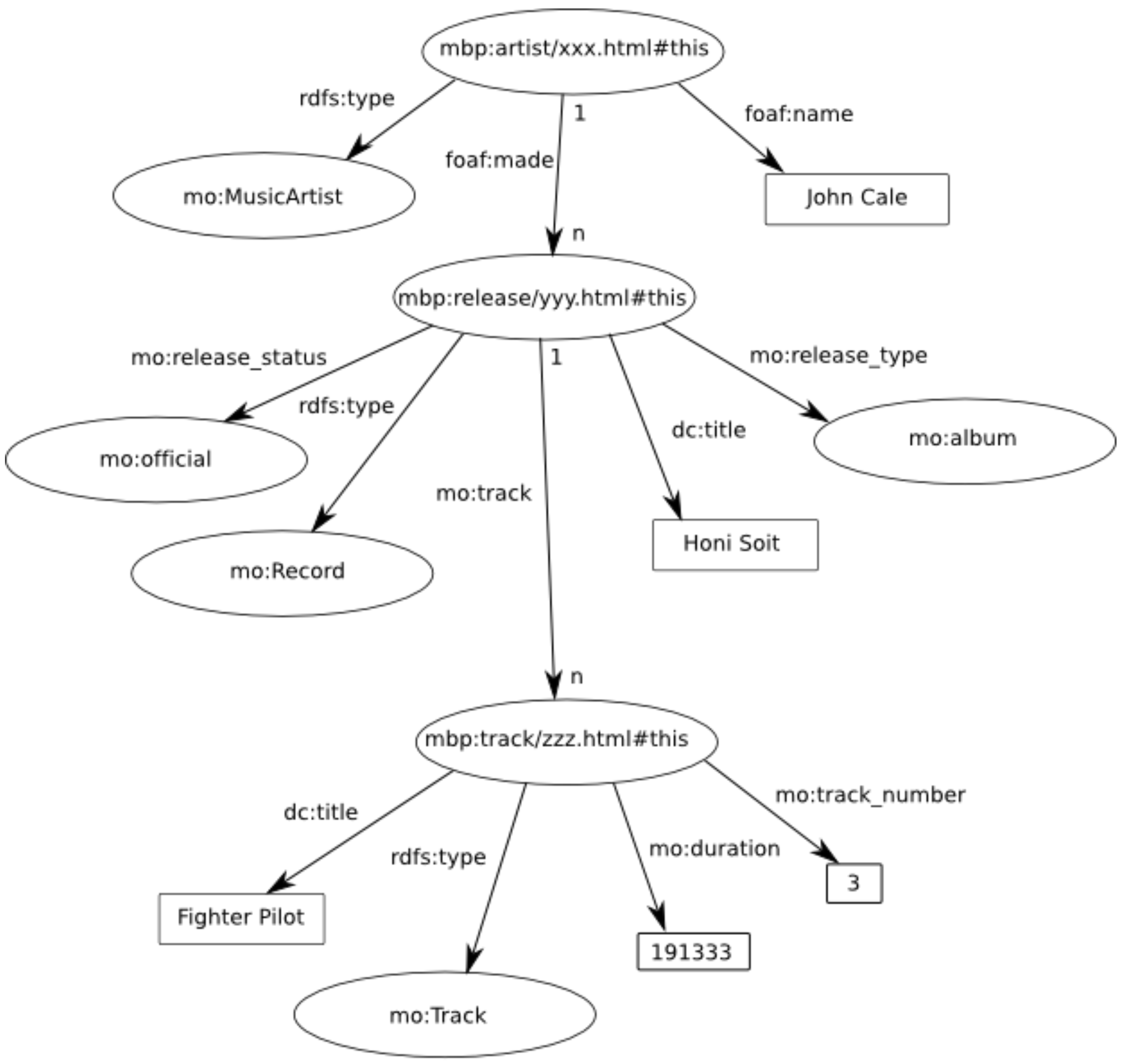
  ```
  curl -H "Accept: text/xml"  "http://localhost:8890/proxy?force=rdf&url=http://musicbrainz.org/artist/72c090b6-a68e-4cb9-b330-85278681a714.html"
  ```

The client application used does not matter. What is important is that in the course of returning RDF to the client, the Sponger service caches the sponged data in the Virtuoso Quad store. Our LinqToRdf application can then query the Quad Store using SPARQL over HTTP as illustrated below.



MusicBrainz Test Data - RDF Representation

Although MusicBrainz defines its own XML Metadata Format to represent music metadata, the MusicBrainz sponger converts the raw data to a subset of the Music Ontology, an RDF vocabulary which aims to provide a set of core classes and properties for describing music on the Semantic Web. The subset used is depicted in the following RDF graph representing a John Cale album.



mbp: http://localhost:8890/proxy/rdf/http://musicbrainz.org/
mo: http://purl.org.ontology/mo/
dc: http://purl.org/dc/elements/1.1/
rdfs: http://www.w3.org/2000/01/rdf-schema#
foaf: http://xmlns.com/foaf/0.1/

Artist ID: xxx = 72c090b6-a68e-4cb9-b330-85278681a714b330b
Release ID: yyy = 9d0cadc4-69cd-4692-b6c4-9458522733e1
Track ID: zzz = 94cd2383-c828-4835-9c83-645148379136

With the prefix mo: denoting the Music Ontology at http://purl.org/ontology/mo/, it can be seen that artists are represented by instances of class mo:Artist, their albums, records etc. by instances of class mo:Release and tracks on these releases by class mo:Track. The property foaf:made links an artist and his/her releases. Property mo:track links a release with the tracks it contains.

LINQ to RDF Object Model

Having outlined the structure of the test RDF data, we can now turn our attention to how LinqToRdf represents and queries this data model from the .NET realm, with the aim of creating a skeletal C# application to return a list of tracks from a selected album using LINQ.

Data Sources & DataContexts

LinqToRdf shares some similarities with LINQ to SQL, Microsoft's LINQ interface to SQL Server. In LINQ to SQL, the DataContext class represents the main entry point for the framework. It is the source of all entities from a database and wraps a database connection. Similarly, LinqToRdf uses an RdfDataContext class as the source of all entities from a triple store. It includes a query results cache and when used with a remote triple store, wraps a triple store connection. RdfDataContext also includes a class factory method IRdfQuery<T> ForType<T>( ) that creates ontology query objects for the type T.

Our example application defines class MusicDataContext as a subclass of RdfDataContext:

```
public class MusicDataContext : RdfDataContext
{
  public MusicDataContext(TripleStore store) : base(store)
  {
  }

  public MusicDataContext(string store) : base(new TripleStore(store))
  {
  }
  . . .
```

A MusicDataContext instance is initialized with a TripleStore instance, or a string identifying a TripleStore, where TripleStore is a class for storing the location and query idiom for an RDF data store. The supporting assembly SemWeb supports four types of triple store: a local in-memory N3 store, a local persistent (file-based) N3 store and both local and remote SPARQL-queryable stores. Here we illustrate only the latter. In this case http://ripley:8890/sparql/ is the HTTP SPARQL endpoint of a Virtuoso RDF store.

```
namespace RdfMusic
{
  class Program
  {
    static void Main(string[] args)
    {
      var ctx = new MusicDataContext(@"http://ripley:8890/sparql/");
      ctx.DefaultGraph =
        "http://musicbrainz.org/artist/72c090b6-a68e-4cb9-b330-85278681a714.html"
```

The DefaultGraph property identifies the RDF graph created by the Virtuoso Sponger to hold data sponged about artist John Cale via the proxy http://ripley:8890/proxy/. The DefaultGraph property is necessary because an RDF quad store can typically host multiple graphs as an "inverted forest". The default graph is used in the FROM clause of any SPARQL query used to retrieve entities for this context.

The records and tracks in our music collection are represented by two C# classes, Record and Track. Two public properties of MusicDataContext provide access to the Records and Tracks collections via strongly typed instances of IQueryable<T>, the key interface a LINQ provider must implement to enable LINQ queries to be executed against the data sources it supports. IQueryable<Record> and IQueryable<Track> are ontology query objects created by class factory method ForType<T>( ).

```
public class MusicDataContext : RdfDataContext
{
  . . .
  public IQueryable<Record> Records
  {
    get
    {
      return ForType<Record>();
    }
  }

public IQueryable<Track> Tracks
{
  get
  {
    return ForType<Track>();
  }
}
```

Next we need a mechanism to map between an OWL class defined in the RDF domain and an equivalent C# class in the LINQ domain representing the same entity type.

Attribute-Based Mapping

Like LINQ to SQL, LinqToRdf uses .NET attributes to map the data source object model to a LINQ object model. It defines custom attributes which you use to decorate C# classes and describe the mapping between the two object models.

LINQ To RDF Custom Attributes

LINQ to RDF defines two custom attributes, Ontology and OwlResource. The Ontology attribute specifies at the assembly level what ontologies are in use and where to find them. The OwlResource attribute maps a C# class to an OWL class or an OWL property to equivalent property in a C# class. The parameters supported by these attributes are summarized below:

Ontology attribute	
Parameter	Usage
Name	Name of the ontology to be used internally by LINQ to RDF.
Prefix	The preferred prefix for the ontology URI.
BaseUri	The base URI used in the ontology.

OwlResource attribute	
Parameter	Usage
OntologyName	Must match the Name property of some assembly level instance of OntologyAttribute
RelativeUriReference	The relative URI of the OWL resource that the target (class/property) of this attribute corresponds to.

Armed with these attributes, we can tie the LINQ and RDF domains together. In our application's assembly information, we identify the RDF/OWL ontologies being employed.

Assembly attributes:

```
  [assembly: AssemblyTitle("RdfMusic")]
  [assembly: AssemblyVersion("1.0.0.0")]
  . . .
  [assembly: Ontology(
    BaseUri = "http://purl.org/ontology/mo/",
    Name = "Music",
    Prefix = "music"
  )]

  [assembly: Ontology(
    BaseUri = "http://purl.org/dc/elements/1.1/",
    Name = "DublinCore",
    Prefix = "dc"
  )]
```

Although the Music Ontology is our primary ontology, properties from other ontologies are also used, e.g. foaf:made and dc:title. These other ontologies are also identified through assembly attributes. Next, each ontology class or property we wish to expose at the application level is mapped to an equivalent C# class or public class property.

Record C# class:

```
[OwlResource(OntologyName = "Music", RelativeUriReference = "Record")]
public class Record : OwlInstanceSupertype
{
  [OwlResource(OntologyName = "DublinCore", RelativeUriReference="title")]
  public string Title { get; set; }

  [OwlResource(OntologyName = "Music", RelativeUriReference = "release_type")]
  public string ReleaseType { get; set; }

  [OwlResource(OntologyName = "Music", RelativeUriReference = "release_status")]
  public string ReleaseStatus { get; set; }
  . . .
}
```

Track C# class:

```
[OwlResource(OntologyName="Music", RelativeUriReference="Track")]
public class Track : OwlInstanceSupertype
{
  [OwlResource(OntologyName = "Dublincore", RelativeUriReference = "title")]
  public string Title { get; set; }

  [OwlResource(OntologyName = "Music", RelativeUriReference="track_number")]
  public int TrackNumber { get; set; }

  [OwlResource(OntologyName = "Music", RelativeUriReference="duration")]
  public int Duration { get; set; }

  . . .
}
```

Retrieving a Collection of Tracks

Details of how to configure the MusicBrainz sponger cartridge are presented later. Assuming you have done this and cached data to a Virtuoso quad store using the proxy service as described above, you should, after compiling the above C# classes, be able to retrieve a collection of all John Cale's tracks through LINQ and inspect the track properties:

```
var ctx = new MusicDataContext(@"http://ripley:8890/sparql/");
ctx.DefaultGraph = @"http://musicbrainz.org/artist/72c090b6-a68e-4cb9-b330-85278681a7
var tracks = (from t in ctx.Tracks
              orderby t.Title
              select t);
Console.WriteLine("Track count: {0}\n", tracks.Count());

foreach (var track in tracks)
{
  Console.WriteLine("Title: {0}", track.Title);
  Console.WriteLine("Duration(msecs): {0}", track.Duration);
}
```

Handling Relationships

Domain-specific object models are typically defined as hierarchies or graphs. Indeed RDF was designed to represent such graphs naturally. In our simple example, records and tracks form an obvious hierarchy and the relationship between the two is contained in the mo:track property.

In LINQ to SQL, objects refer to each other by using property references or collections of references that you navigate by using dot notation. The references to other objects or collections of other objects in your class definitions correspond directly to foreign-key relationships in the database. LINQ to SQL defines an Association attribute together with the EntityRef<TEntity> and EntitySet<TEntity> types to help represent such relationships, where EntityRef<TEntity> and EntitySet<TEntity> support the singleton and collection sides respectively of a one-to-many relationship.

LinqToRdf can also be made to support relationship navigation using dot notation, again through a combination of properties, and the EntityRef and EntitySet types. An illustration of how this can be achieved is shown in the listing below which shows how the Record class exposes its tracks through the public Tracks property.

```
[OwlResource(OntologyName = "Music", RelativeUriReference = "Record")]
public class Record : OwlInstanceSupertype
{
  . . .
  private EntitySet<Track> _Tracks = new EntitySet<Track>();
  [OwlResource(OntologyName = "Music", RelativeUriReference = "track")]
  public EntitySet<Track> Tracks
  {
    get
    {
      if (_Tracks.HasLoadedOrAssignedValues)
        return _Tracks;
      if (DataContext != null)
      {
        string recordUri = this.InstanceUri;
        string trackPredicateUri = this.PredicateUriForProperty(MethodBase.GetCurrent
        _Tracks.SetSource(
          from t in ((MusicDataContext)DataContext).Tracks
          where t.StmtObjectWithSubjectAndPredicate(recordUri, trackPredicateUri)
          select t);
        return _Tracks;
      }
      return null;
    }
  }
  . . .
}
```

In the above property definition, the LINQ filter clause

```
          where t.StmtObjectWithSubjectAndPredicate(recordUri, trackPredicateUri)
```

is translated by the LinqToRdf internals to a SPARQL WHERE clause of the form:

```
WHERE {
$t a mo:Track .
. . .
<recordUri> <trackPredicateUri> $t .
}
```

where <recordUri> is the URI identifying the Record instance for which we want a list of tracks, and <trackPredicateUri> is the predicate associated with the Tracks property through an OwlResource attribute, i.e. mo:track. So, at the application level, after retrieving an record, the Tracks property returns a collection populated with all the tracks on that record, as we would expect.

```
var ctx = new MusicDataContext(@"http://ripley:8890/sparql/");
ctx.DefaultGraph = "http://musicbrainz.org/artist/72c090b6-a68e-4cb9-b330-85278681a

var album = (from r in ctx.Records
             where r.Name.StartsWith("Honi Soit")
             select r).First();

Console.WriteLine("Record title: {0}", record.Title);
Console.WriteLine("Record track count: {0}", record.Tracks.Count);

foreach (var track in record.Tracks)
{
  Console.WriteLine("Track {0}: {1} [{2} msecs]",
                    track.TrackNumber, track.Title, track.Duration);
}
```

Navigating this relationship in the opposite direction, i.e. from a Track to the parent Record, is supported through a Record property in the Track class:

```
[OwlResource(OntologyName="Music", RelativeUriReference="Track")]
public class Track : OwlInstanceSupertype
{
  . . .
  private EntityRef<Record> _Record { get; set; }
  [OwlResource(OntologyName = "Music", RelativeUriReference = "track")]
  public Record Record
  {
    get
    {
      if (_Record.HasLoadedOrAssignedValue)
        return _Record.Entity;
      if (DataContext != null)
      {
        var ctx = (MusicDataContext)DataContext;
        string trackUri = this.InstanceUri;
        string trackPredicateUri = this.PredicateUriForProperty(MethodBase.GetCurre
        _Record = new EntityRef<Record>(
          from r in ((MusicDataContext)DataContext).Records
          where r.StmtSubjectWithObjectAndPredicate(trackUri, trackPredicateUri)
          select r);
        return _Record.Entity;
      }
      return null;
    }
 }
 . . .
```

which allows us to do the following:

```
var track = (from t in ctx.Tracks select t).First();

Console.WriteLine("Record title: {0}", track.Record.Title);
Console.WriteLine("Record track count: {0}", track.Record.Tracks.Count);
```

In the definition of the Track class Record property above, the use of the extension method StmtSubjectWithObjectAndPredicate reflects the direction of the predicate, mo:track, linking the parent Record to each child Track. In this case,

```
where r.StmtSubjectWithObjectAndPredicate(trackUri, trackPredicateUri)
```

translates to a SPARQL WHERE clause:

```
WHERE {
$r a mo:Record .
. . .
$r <trackPredicateUri> <trackUri> .
}
```

If the predicate linking the two ran in the opposite direction, e.g. the predicate was "isTrackOn" instead of "track", we would use StmtObjectWithSubjectAndPredicate instead. With this pair of extension methods we can cope with ontologies which don't define reciprocal predicates to link two enties in both directions.

Supporting Tools

Two related projects running in tandem with LinqToRdf aim to automate the creation of object models. LinqToRdfDesigner is a Visual Studio 2008 hosted tool which supports visual editing of ontologies and the generation of accompanying model/class definition files in OWL (N3) and C#. RdfMetal is a command-line code generator for LINQ to RDF, in a similar vein to LINQ to SQL’s SQLMetal. It has only recently been announced and is at a very early implementation stage. Given the URL of a remotely managed ontology accessible through a SPARQL endpoint, it generates partial C# classes and mappings for the classes defined by the ontology.

Summary

Hopefully this note has provided an indication of how LinqToRdf addresses the challenge of bridging the .NET and Semantic Web worlds. With LINQ positioned as a generic query language for .NET, LinqToRdf seeks to fill an obvious gap. Clear shortcomings exist when compared to LINQ to SQL, such as the lack of a facility to update remote triple stores through LINQ. However, this issue is still being addressed in SPARQL (the update language for SPARQL, SPARQL/Update or SPARUL, although implemented in Virtuoso, is not yet a standard.) Nevertheless LinqToRdf provides the beginnings of a viable bridge from the Semantic Web to .NET.

Footnote - The MusicBrainz Sponger Cartridge for LinqToRdf

As outlined above, the Virtuoso Sponger is RDF middleware for transforming non-RDF data into RDF "on the fly". Its goal is to use non-RDF Web data sources as input and create RDF as output, so enabling the use of non-RDF data sources as Semantic Web data sources. Architecturally, it is comprised of a number of cartridges which are themselves comprised of a metadata extractor and an ontology mapper. The ontology mapper generates RDF instance data from extracted non-RDF data using ontologies associated with a given data source type. They are typically XSLT or Virtuoso PL based.

Virtuoso supplies numerous cartridges for extracting RDF data from popular Web resources and file types in the form of the VAD (Virtuoso Application Distribution) package rdf_cartridges_dav. Included in this set is a MusicBrainz cartridge which comprises two Virtuoso/PL procedures, RDF_LOAD_MBZ and RDF_LOAD_MBZ1 and an XSLT stylesheet mbz2rdf.xsl. The last two components are fine for our purposes, but the main cartridge procedure RDF_LOAD_MBZ requires minor modification to generate RDF in a form suitable for consumption by our LinqToRdf example.

The required Virtuoso/PL code, listed below, can be executed by pasting it into the iSql (interactive SQL) panel of Virtuoso's browser-based Conductor interface.

```
--no_c_escapes-
create procedure DB.DBA.RDF_LOAD_MBZ (
  in graph_iri varchar, in source_uri varchar,  in dest varchar,
  inout _ret_body any, inout aq any, inout ps any, inout _key any,
  inout opts any)
{
  declare artistId, vReleaseUris, vReleaseIds, releaseId any;
  declare tmp, mdata, rset any;
  declare qry, artistGraph, artistUri, proxyIri, stat, msg varchar;

  -- On error, report no data can be sponged
  declare exit handler for sqlstate '*'
  {
    -- dbg_printf ('%s', __SQL_MESSAGE);
    return 0;
  };

  -- Extract the artist's MusicBrainz ID (MID) from the source URI
  tmp := regexp_parse ('http://musicbrainz.org/artist/([^\.]+)', source_uri, 0);
  if (length (tmp) < 4)
    return 0;

  artistId := subseq (source_uri, tmp[2], tmp[3]);
  -- dbg_printf('artist id: %s', artistId);

  -- Sponge all the artist's releases by querying the MusicBrainz XML Web Service.
  if (dest is null)
    delete from DB.DBA.RDF_QUAD where G = DB.DBA.RDF_MAKE_IID_OF_QNAME (graph_iri);
  DB.DBA.RDF_LOAD_MBZ_1 (graph_iri, source_uri, dest, 'artist', artistId, 'sa-Album s

  -- Now we want to find the MIDs of all the releases.  For each MID, we can then que
  -- XML Web Service again, this time for the tracks contained on each release (i.e.
  -- way because MusicBrainz doesn't appear capable of returning in a single query al
  -- together with all the tracks for each release.
  --
  -- The Sponger will have created a graph for the artist with a URI of the form:
  -- <http://musicbrainz.org/artist/4d5447d7-c61c-4120-ba1b-d7f471d385b9.html>
  -- and which contains statements: <artist_uri> foaf:made <release_uri>
  -- We can find all the artist's releases by querying the cached sponged data with a
  -- The returned objects will be mo:Record instances with URI's along the lines of:
  -- <http://demo.openlinksw.com/proxy/rdf/http://musicbrainz.org/release/cc59710f-d5
  -- where prefix mo: identifies the Music Ontology <http://purl.org/ontology/mo/>.

  proxyIri := DB.DBA.RDF_SPONGE_PROXY_IRI();
  artistGraph := sprintf('<http://musicbrainz.org/artist/%s.html>', artistId);
  artistUri := sprintf('<%shttp://musicbrainz.org/artist/%s.html#this>', proxyIri, ar
  qry := sprintf(
      'sparql prefix foaf: <http://xmlns.com/foaf/0.1/> select distinct ?release from
      artistGraph, artistUri);
  --dbg_printf('qry: %s', qry);
  exec (qry, stat, msg, vector (),
     200,   /* Max no. of rows to return. MusicBrainz query resultset limit is 100 ro
     mdata, /* Result-set metadata */
     rset); /* Result-set */

  vReleaseUris := vector();
  foreach (any rset_row in rset) do
  {
    -- dbg_printf('rset_row[0]: %s', rset_row[0]);
    vReleaseUris := vector_concat(vReleaseUris, vector(rset_row[0]));
  }

  vReleaseIds := vector();
  foreach (any releaseUri in vReleaseUris) do
  {
    -- Scan vReleaseUris, extracting distinct release MIDs
    tmp := regexp_parse ('musicbrainz.org/release/([^\.]+)', releaseUri, 0);
    if (length (tmp) < 4)
```

Alternatively the above procedure can be created under a different name, RDF_LOAD_MBZLINQ say, and the Sponger pointed to it by changing the settings for the MusicBrainz cartridge displayed in Conductor's "RDF Cartridges" configuration screen. (Full details of how to use Conductor to configure Sponger cartridges and view or edit their XSLT stylesheets can be found in the Virtuoso Sponger whitepaper.)

You will no doubt be asking why this alternative MusicBrainz cartridge is needed. Both cartridges extract data through the MusicBrainz XML Web Service using, as the basis for the initial query, the artist ID extracted from the original URI submitted to the RDF proxy. The original cartridge returns a list of the artist's releases. However, to obtain a list of tracks for each release, it's necessary to issue a separate query to the cartridge, once for each release, for instance http://ripley:8890/proxy?force=rdf&http://musicbrainz.org/release/<release_id>. The need for separate queries reflects limitations in the MusicBrainz Web Service interface which is incapable of returning the information we require in a single query. The modified cartridge overcomes this limitation but is restricted to handling a specific class of query, viz. for the records and tracks by a particular artist. The original cartridge is capable of handling all the query types supported by MusicBrainz but in a drill-down usage scenario. For a compelling example of the cartridge being used in this way try the OpenLink Data Explorer, a browser extension which uses the Sponger extensively.