Using the RdfMetal code generator

Andrew Matthews
matthews.andrew@gmail.com
http://code.google.com/p/linqtordf

July 9, 2008

Abstract

RdfMetal is part of the LinqToRdf tool suite. It reads an ontology in a semantic web database and generates LinqToRdf compatible code from it. This walkthrough shows how to use it to generate code for your project.

1 Overview & Usage

RdfMetal works by querying a remote SPARQL enabled triple store to get information about the classes defined in a user specified ontology. It stores the results in a metadata file for use during a code generation phase. During a code generation phase the tool will read the metadata creating C# source code from it.

The two phases of process (metadata retrieval and code generation) are independent and can be invoked separately or together. The common thread that links them is the metadata file stored at the end of the metadata retrieval phase.

RdfMetal takes these options:

```
$ ./RdfMetal.exe /?
RdfMetal  0.1.0.0 - Copyright (c) Andrew Matthews 2008
A code generator for LinqToRdf

Usage: RdfMetal [options] is used with the following options
Options:
  -e -endpoint:PARAM      The SPARQL endpoint to query.
  -h -handle:PARAM        The ontology name to be used in
                          LinqToRdf for prefixing URIs and
  disambiguating class names and
  properties.
  -? -help                Show this help list
     -help2               Show an additional help list
  -i -ignorebnodes        Ignore BNodes. Use this if you only
```

```
                          want to generate
                          code for named classes.
-m -metadata:PARAM        Where to place/get the collected
                          metadata.
-n -namespace:PARAM       The XML Namespace to extract classes
                          from.
-N -netnamespace:PARAM    The .NET namespace to place the
                          generated source in.
-o -output:PARAM          Where to place the generated code.
-r -references:PARAM      A comma separated list of namespaces to
                          reference in the generated source.
   -usage                 Show usage syntax and exit
-V -version               Display version and licensing informa-
                          tion
```

If you provide a SPARQL endpoint, `RdfMetal` will connect to it and retrieve whatever domain model it can find there. If you provide a filename for the metadata to be stored in, it will save the metadata as XML in that file. If you want you can stop at that point and generate code later. If you don't provide a metadata storage location the data will be passed directly to the code generation part of the application, that will generate source from it. To generate source, you must provide an output location for the source. If no output location is provided, then no source is created and the application stops.

The steps you need to follow are:

- Locate the SPARQL endpoint where the data is stored

  Restrict the ontology namespace URI to the object model you are interested in.

- Invoke `RdfMetal` to retrieve the class definitions from the remote SPARQL endpoint.

- Add the source `RdfMetal` generates to your project

- Reference the `LinqToRdf.dll` assembly in your project

- Query the object model using LINQ.

Walkthrough

1.1 Installing `RdfMetal`

`RdfMetal` is part of the `LinqToRdf` tool suite. If you have the latest version of `LinqToRdf`, then you should have a copy. If not, you should download and install the latest featured release of `LinqToRdf` at `http://code.google.com/p/linqtordf`. After installing `LinqToRdf` you should have a program files directory at `c:/Program Files/Andrew Matthews /LinqToRdf`. Inside which is all that you need to be able to run RdfMetal. It's a command line tool, so you will need to either put that directory in the path, or refer to the RdfMetal executable using the full path, as we'll do in this walkthrough.

1.2 Finding the endpoint URI

The first thing you will require is the SPARQL endpoint URL of the semantic web database you're planning to work with. For this walkthrough, we'll be using the sample triple store that is used by the LinqToRdf project for testing. In this case, you query it through the URL http://localhost/linqtordf/SparqlQuery.aspx. You can find out more about hosting your own ontologies in the LinqToRdf manual, or in my article Understanding SPARQL.

You need to know this endpoint URL because RdfMetal will send a series of SPARQL queries to the database requesting information about the classes stored there and their properties and relationships.

1.3 Create a pre-build step in your project

In this walkthrough, we use RdfMetal to generate code as part of a pre-build step. That means the metadata will be retrieved and used to generate source code every time your project gets built. This probably overkill, especially if the remote ontology is not subject to frequent changes. In that case you might want to perform these steps manually as required.

This command line invokes RdfMetal using the build step macro $(ProjectDir) defined in Visual Studio. Typically, you define the SPARQL endpoint as an initial phase, and then cache metadata gathered in an XML file. Once the metadata has been gathered, C# source code can be generated from it. If the target ontology is static, you may choose not to continue using RdfMetal after the initial source code generation. Here, we're simply gathering the metadata every time and storing it in a source file called music.cs.

```
"c:\Program Files\Andrew Matthews\RdfMetal\RdfMetal.exe"
    -e:http://localhost/linqtordf/SparqlQuery.aspx
    -i -n http://aabs.purl.org/ontologies/2007/04/music\#
    -o "$(ProjectDir)music.cs" -N:RdfMetal.Music -h music
```

2 Using RdfMetal with Visual Studio

RdfMetal was designed to used from a pre-build event in Visual Studio. The example above, shows how it can be used to generate source for the small ontology used to test LinqToRdf. It queries a local website called LinqToRdf. (You can get instructions on how to set up such a self hosted ontology from the LinqToRdf manual.) I'm restricting the classes to only those defined in the http://aabs.purl.org/ontologies/2007/04/music# namespace and am generating code in the RdfMetal.Music .NET namespace. The internal name

I'm using is 'music', and that's the preferred prefix to be used in any generated RDF or SPARQL.

When RdfMetal is run it outputs a list of the classes that it is generating source for, then writes 'done'. Here's some output from the music ontology:

```
------ Build started: Project: RdfMetalTestHarness
C:\. . .\linqtordf\src\RdfMetal\bin\Debug\RdfMetal.exe
  -e:http://localhost/LINQTORDF/SparqlQuery.aspx -i -n
  http://aabs.purl.org/ontologies/2007/04/music\# -o
  "C:\. . .\linqtordf\src\unit-testing\
  RdfMetalTestHarness\music.xml" -N:RdfMetal.Music -h music

ProducerOfMusic
SellerOfMusic
NamedThing
TemporalThing
Person
Band
Studio
Music
Album
Track
Song
Mp3File
Genre
done.
```

Each of these classes is defined in the store.n3 file in the unit tests directory in the LinqToRdf solution. The source that it generates will be in the file music.cs. The output is too long and repetitive to be worth including in full, but here's some edited highlights. The generator creates a DataContext class containing standard query properties for each of the class types found in the metadata extraction process. In this case the queries are included for Album and Track

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

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

```
    public IQueryable<Album> Albums
    {
      get { return ForType<Album>(); }
    }

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

In most cases the classes generated are empty, but in the test data the Track and Album classes have several DatatypeProperties as well as ObjectProperties. Here's the code generated for the `Track` class.

```
[OwlResource(OntologyName = "music",
             RelativeUriReference = "Track")]
public class Track : OwlInstanceSupertype
{
    #region Datatype properties

    [OwlResource(OntologyName = "music",
                 RelativeUriReference = "title")]
    public string title { get; set; } // Track
    [OwlResource(OntologyName = "music",
                 RelativeUriReference = "artistName")]
    public string artistName { get; set; } // Track
    [OwlResource(OntologyName = "music",
                 RelativeUriReference = "albumName")]
    public string albumName { get; set; } // Track
    [OwlResource(OntologyName = "music",
                 RelativeUriReference = "genreName")]
    public string genreName { get; set; } // Track
    [OwlResource(OntologyName = "music",
                 RelativeUriReference = "comment")]
    public string comment { get; set; } // Track
    [OwlResource(OntologyName = "music",
                 RelativeUriReference = "fileLocation")]
    public string fileLocation { get; set; } // Track

    #endregion

    #region Incoming relationships properties

    #endregion
```

```
    #region Object properties

    [OwlResource(OntologyName = "music",
                 RelativeUriReference = "isTrackOn")]
    public string isTrackOnUri { get; set; }

    private EntityRef<Album> _isTrackOn { get; set; }

    [OwlResource(OntologyName = "music",
                 RelativeUriReference = "isTrackOn")]
    public Album isTrackOn
    {
        get
        {
            if (_isTrackOn.HasLoadedOrAssignedValue)
                return _isTrackOn.Entity;
            if (DataContext != null)
            {
               var ctx = (musicDataContext) DataContext;
               _isTrackOn = new EntityRef<Album>(
                     from x in ctx.Albums
                     where x.HasInstanceUri(isTrackOnUri)
                     select x);
               return _isTrackOn.Entity;
            }
            return null;
        }
    }

    #endregion
}
```

Note the use of the `EntityRef` in the `_isTrackOn` field and the `isTrackOn` property to provide navigation across the object graph.

```
[OwlResource(OntologyName = "Music",
             RelativeUriReference = "Album")]
public class Album : OwlInstanceSupertype
{
  public Album()
  {
  }

  [OwlResource(OntologyName = "Music",
               RelativeUriReference="name")]
```

```
  public string Name { get; set; }

  private EntitySet<Track> _Tracks = new EntitySet<Track>();
  [OwlResource(OntologyName = "Music",
               RelativeUriReference = "isTrackOn")]
  public EntitySet<Track> Tracks
  {
    get
    {
      if (_Tracks.HasLoadedOrAssignedValues)
        return _Tracks;
      if (DataContext != null)
      {
        _Tracks.SetSource(from t in
          ((MusicDataContext)DataContext).Tracks
           where t.AlbumName == Name
           select t);
      }
      return _Tracks;
    }
  }
}
```

Again notice the use of `EntitySet`s to provide navigation. This time the navigation is from parent to child.

Once the code is generated all you need to do is incorporate it into your project and use it like any other `LinqToRdf` code. For instructions on how to do that, consult the `LinqToRdf` manual for guidance.