The LinqToRdf Tutorial

Andrew Matthews
matthews.andrew@gmail_dot_com
http://aabs.wordpress.com/semantic-web

July 27, 2008

Abstract

This document guides you through the steps needed to write a simple semantic web application in C#. It covers the tools and systems prerequisites, the techniques and the expected behavior. All examples are in C#, although LinqToRdf should work on any .NET language that supports LINQ.

Contents

List of Figures

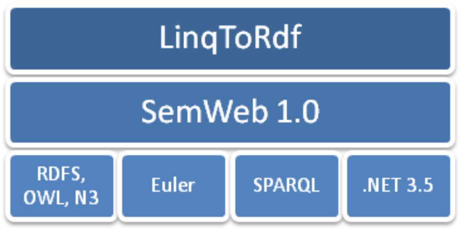


Figure 1: Major technologies employed with LinqToRdf.

1 Introduction

1.1 What are the components of a Semantic Web Application?

Within the context of this document, 'Semantic Web Application' means "application that uses RDF and related technologies". That is – an application that represents information as a graph structure using RDF. Its beyond the scope of this document to explain the whole pyramid of standards and technologies needed to support the semantic web. Instead Ill give context enough for you to know what steps are required to get your Semantic Web Application off the ground.

LinqToRdf uses the SemWeb.NET framework by Joshua Tauberer, which provides a platform for working with OWL and SPARQL. It also uses the .NET 3.5 namespace `System.Linq` which will be released as part of Visual Studio .NET 2008.

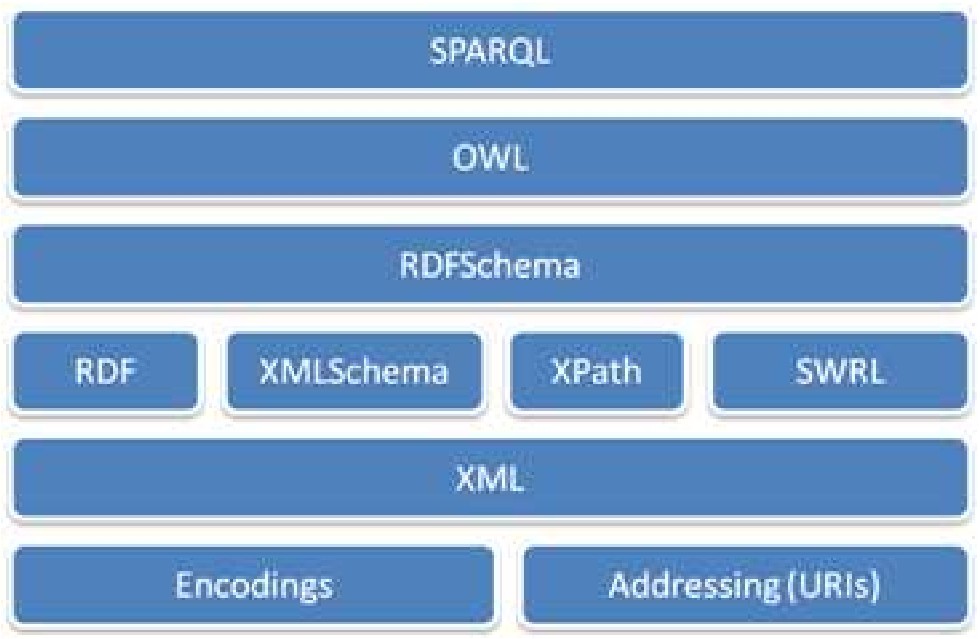


Figure 2: The hierarchy of technologies used in the semantic web.

2 Programming with LinqToRdf

2.1 What do you need to do semantic web programming with LinqToRdf?

LinqToRdf requires a runtime that supports the latest version of LINQ, which at the time of writing is .NET 3.5 or Mono 1.9. LinqToRdf has not been tested with Mono yet, but since Mono 1.9 now supports C# it should be able to run.

2.2 Where to get LinqToRdf

You can download the latest release from Google code (latest version at time of writing is LinqToRdf-0.8.msi) . That installer also contains an installer for te LinqToRdf designer - an extension to visual studio 2008 that will provide a simple UML design surface for producing ontologies and domain models. Check the linqtordf-discuss discussion forum for announcements of newer releases.

2.3 Installation procedure

Installation is a simple matter of double clicking the MSI file, and deciding where to install the assemblies for LinqToRdf. The default location will be in a `LinqToRdf-0.x` directory under “ `c:\Program Files\Andrew Matthews`”.

Once LinqToRdf is installed, you should then install LinqToRdfDesigner, which will register the DSL (Domain Specific Language) add-in to Visual Studio. Note that LinqToRdf and LinqToRdfDesigner will only work with Visual Studio 2008 and above.

2.4 Creating an ontology

The details of the OWL standard are beyond the scope of this document. The standards document is found at the W3C . There are various tools available for creating RDF, and it is important to know that the SemWeb library can understand the Notation 3 syntax , which is a human readable (non-XML) variant of RDF. The examples well be using in the rest of this document use Notation 3 (or N3 for short). If your requirements are modest the LinqToRdf graphical designer should be sufficient for most tasks. If you need to create more complex ontologies, or if you need them to be created in OWL, RDF or RDFS format, then you might want to consider tools such as Protg. LinqToRdfDesigner will be covered in a later section. This section shows how the LinqToRdf framework can be used without any other support. Lets start with a simple ontology for recording MP3 files. The ontology file should have an extension of n3. Lets call ours music.n3. First you define the XML namespaces you will be working with:

```
@prefix rdf:  <http://www.w3.org/1999/02/22-rdf-syntax-ns\#> .
@prefix daml: <http://www.daml.org/2001/03/daml+oil\#> .
@prefix log: <http://www.w3.org/2000/10/swap/log\#> .
@prefix rdfs: <http://www.w3.org/2000/01/rdf-schema\#> .
@prefix owl:  <http://www.w3.org/2002/07/owl\#> .
@prefix xsdt: <http://www.w3.org/2001/XMLSchema\#>.
@prefix : <http://aabs.purl.org/ontologies/2007/04/music\#> .
```

This imports some of the standard namespaces for OWL, RDF, XML Schema datatypes and others. It also defines a default namespace to be used for all classes and properties that are going to be defined in the rest of the document. Now lets define some classes:

```
:Album a owl:Class.
:Track a owl:Class.
:title rdfs:domain :Track;
    rdfs:range  xsdt:string.
:artistName
    rdfs:domain :Track;
    rdfs:range  xsdt:string.
:albumName
    rdfs:domain :Track;
    rdfs:range  xsdt:string.
:year
    rdfs:domain :Album;
    rdfs:range  xsdt:integer.
:genreName
    rdfs:domain :Track;
    rdfs:range  xsdt:string.
:comment
    rdfs:domain :Track;
    rdfs:range  xsdt:string.
:isTrackOn
    rdfs:domain :Track;
    rdfs:range  :Album.
:fileLocation
    rdfs:domain :Track;
    rdfs:range  xsdt:string.
```

This defines a class Track of type owl:Class. After the class declaration, I defined some properties on the Track Class (:title, :artistName etc). Because the prolog section previously defined a default namespace, these declarations are now in the ¡http://aabs.purl.org/ontologies/2007/04/music#¿ namespace. Thats all thats required to define our simple ontology. Now we can to create some data for MP3 files.

Create another file called mp3s.n3 and add the following:

```
@prefix ns1: <http://aabs.purl.org/ontologies/2007/04/music\#> .
ns1:Track_-861912094 <http://www.w3.org/1999/02/22-rdf-syntax-ns\#type> ns1:Track ;
    ns1:title "History 5 | Fall 2006 | UC Berkeley" ;
    ns1:artistName "Thomas Laqueur" ;
    ns1:albumName "History 5 | Fall 2006 | UC Berkeley" ;
    ns1:year "2006" ;
    ns1:genreName "History 5 | Fall 2006 | UC Berkeley" ;
    ns1:comment " (C) Copyright 2006, UC Regents" ;
    ns1:isTrackOn ns1:Album_2 ;
    ns1:fileLocation "C:\\Users\\andrew.matthews\\Music\\hist5_20060829.mp3" .
ns1:Track_-1378138934 <http://www.w3.org/1999/02/22-rdf-syntax-ns\#type> ns1:Track ;
    ns1:title "History 5 | Fall 2006 | UC Berkeley" ;
    ns1:artistName "Thomas Laqueur" ;
    ns1:albumName "History 5 | Fall 2006 | UC Berkeley" ;
    ns1:year "2006" ;
    ns1:genreName "History 5 | Fall 2006 | UC Berkeley" ;
    ns1:comment " (C) Copyright 2006, UC Regents" ;
    ns1:isTrackOn ns1:Album_2 ;
    ns1:fileLocation "C:\\Users\\andrew.matthews\\Music\\hist5_20060831.mp3" .
ns1:Track_583675819 <http://www.w3.org/1999/02/22-rdf-syntax-ns\#type> ns1:Track ;
    ns1:title "Rory Blyth: The Smartest Man in the World\u0000" ;
    ns1:artistName "Rory Blyth\u0000" ;
    ns1:albumName "Rory Blyth: The Smartest Man in the World\u0000" ;
    ns1:year "2007\u0000" ;
    ns1:genreName "Rory Blyth: The Smartest Man in the World\u0000" ;
    ns1:comment "Einstein couldn't do it again if he lived today. He'd be
      too distracted by the allure of technology, and by all those buttheads
      at Mensa trying to prove how smart they are." ;
    ns1:isTrackOn ns1:Album_2 ;
    ns1:fileLocation "C:\\Users\\andrew.matthews\\Music\\iTunes\\iTunes
      Music\\Podcasts\\Rory Blyth_ The Smartest Man in the Worl\\A Few
      Thoughts on the Subject of Gen.mp3" .
```

These entries were taken randomly from a list of podcasts that I subscribe to. In addition, I wrote a program to create them, but you could do it by hand if you want to. Ill show you in a little while how I created the entries in the `mp3s.n3` file. In mp3s.n3 I defined a namespace called ns1. It refers to what was the default namespace in music.n3. That means that references to entities from the ontology like Track will be called ns1:Track rather than :Track as they were called in `music.n3`. It doesnt matter what you call the prefix for the namespace, just so long as the URI that it maps to is the same as was used in the ontology definition file. I called it `ns1`, because thats what my import program wanted to do. The point is that the type `ns1:Track` in this file refers to the `:Track` class defined in `music.n3`. The triple store that well get to shortly will be able to make sense of that in order to know that a ns1:Track has a title, artist etc. It is also able to work out the types of the properties (which just happens to be string for the moment).

Thats it. Thats all there is to creating an ontology. Later on, well get onto the more complicated task of linking types together using ObjectProperties, but for now you have an ontology and some data that uses it.

2.5 Hosting your ontology

Since the uptake of semantic web technologies has been pretty patchy in the .NET domain so far, your best bet for industrial strength RDF triple stores will (for now) be either Jena (`jena.sf.net`) or OpenLink (`openlinksw.com`), though there are various triple store solutions that can be used. For this guide I shall stick to .NET by using Joshua Tauberers SPARQL enabled HttpHandler for ASP.NET, which is sufficient to demonstrate how LinqToRdf can connect to a SPARQL compatible triple store. LinqToRdf has been tested with the Joseki SPARQL interface to Jena, and the OpenLink Virtuoso platform. LinqToRdf is platform independent in the sense that it will happily work on any standards compliant SPARQL server.

To use the HttpHandler as a triple store for music.n3, create an ASP.NET application in visual studio. Place the following into configuration section of the web.config of the project:

```
<configSections>
    <section name="sparqlSources"
    type="System.Configuration.NameValueSectionHandler, System,
    Version=2.0.0.0, Culture=neutral, PublicKeyToken=b77a5c561934e089"/>
</configSections>
```

Next, add the following to the body of the body of the config file beneath the root:

```
<sparqlSources>
    <add key="/[your vdir here]/SparqlQuery.aspx"
       value="n3:[your path here]\mp3s.n3"/>
</sparqlSources>
```

The name SparqlQuery.aspx doesnt matter it doesnt exist. You can call it whatever you want. What this does is link the URL for the file SparqlQuery.aspx to the SPARQL HttpHandler that we next add to the system.web section of the web.config:

```
<httpHandlers>
   <!-- This line associates the \spq{} Protocol implementation with a path on
   your website. With this, you get a \spq{} server at
   http://yourdomain.com/sparql.  -->
   <add verb="*" path="SparqlQuery.aspx"
   type="SemWeb.Query.SparqlProtocolServerHandler, SemWeb.Sparql" />
</httpHandlers>
```

This uses the HttpHandler defined in SemWeb to accept SPARQL queries and run them against the triples defined in mp3s.n3. Thats all thats needed to turn your ASP.NET into a semantic web triple store! Yes, its that easy. To use the HttpHandler you give the URL (/[your vdir here]/SparqlQuery.aspx in the example above) defined above in a TripleStore object that is passed to the RDF context object. Heres an example taken from the LinqToRdf test suite.

```
TripleStore ts = new TripleStore();
ts.EndpointUri = @"http://localhost/linqtordf/SparqlQuery.aspx";
ts.QueryType = QueryType.RemoteSparqlStore;
```

the LinqToRdf SparqlQuery object will use this to direct queries via HTTP to the triple store located at ts.EndpointUri.

2.6 Linking to the ontology from .NET

Now we have an ontology defined, and somewhere to host it that understands SPARQL we can start using LinqToRdf. References to ontologies are defined at the assembly level to prevent repetition (as was the case in earlier versions of

LinqToRdf). You create an Ontology Attribute for each of the ontologies that are referenced in the application or ontology. Below is an example from the SystemScanner reference application.

```
[assembly: Ontology(
    BaseUri = "http://aabs.purl.org/ontologies/2007/10/"
        +"system-scanner\#",
    Name = "SystemScanner",
    Prefix = "syscan",
    UrlOfOntology =
    "file:///C:/etc/dev/semantic-web/linqtordf/doc/"
        +"Samples/SystemScanner/rdf/sys.n3")]
[assembly: Ontology(
    Prefix = "rdf",
    BaseUri = "http://www.w3.org/1999/02/22-rdf-syntax-"
        +"generatedNamespaceChar\#",
    Name = "RDF")]
[assembly: Ontology(
    Prefix = "rdfs",
    BaseUri = "http://www.w3.org/2000/01/rdf-schema\#",
    Name = "RDFS")]
[assembly: Ontology(
    Prefix = "xsdt",
    BaseUri = "http://www.w3.org/2001/XMLSchema\#",
    Name = "Data Types")]
[assembly: Ontology(
    Prefix = "fn",
    BaseUri = "http://www.w3.org/2005/xpath-functions\#",
    Name = "XPath Functions")]
```

Each ontology has a number of named properties that can be set. In the example above, we have only set all of the properties for the system-scanner ontology itself. The others are standard namespaces that are well-known to the underlying components of LinqToRdf. They only require a prefix, BaseUri and Name property to be set. The Prefix property is a suggestion only to LinqToRdf of how you want the namespaces to be referenced. If LinqToRdf finds a clash between prefixes, then it will make a prefix name up instead.

The BaseUri is the fully qualified BaseUri used to form full resource URIs within the triple store. The Name property is used internally to refer to the ontology symbolically. The other attribute OwlResource has an OntologyName property that refers to the name of the ontology defined at the assembly level. Since the prefix is subject to change without notice, the Name property is used instead to embed a class, field or property into a specific named ontology.

Next you should create a new class called Track in a file called Track.cs. Well see an easier way to do this later on, but for now were going to do it the hard way.

```
using LinqToRdf;
namespace RdfMusic
{
[OwlResource(OntologyName="Music", RelativeUriReference="Track")]
public class Track : OwlInstanceSupertype
{

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

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

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

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

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

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

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

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

    public Track(TagHandler th, string fileLocation)
    {
        FileLocation = fileLocation;
        Title = th.Track;
        ArtistName = th.Artist;
        AlbumName = th.Album;
        Year = th.Year;
        GenreName = th.Genere;
        Comment = th.Comment;
    }

    private EntityRef<Album> _Album { get; set; }
    [OwlResource(OntologyName = "Music",
     RelativeUriReference = "isTrackOn")]
    public Album Album
    {
        get
        {
            if (_Album.HasLoadedOrAssignedValue)
                return _Album.Entity;
            if (DataContext != null)
            {
                var ctx = (MusicDataContext)DataContext;
                string trackUri = this.InstanceUri;
                string trackPredicateUri =
this.PredicateUriForProperty(MethodBase.GetCurrentMethod());
                _Album = new EntityRef<Album>(
                    from r in ((MusicDataContext)DataContext).Albums
                    where r.StmtObjectWithSubjectAndPredicate(trackUri,
     trackPredicateUri)
                    select r);

                return _Album.Entity;
            }
            return null;
        }
    }
    public Track()
    {

    }
}
```

The class is just the same as any other entity class except that the class and its properties have been annotated with OwlResource attributes . The critical bit to get right is to use the same URI in `OntologyAttribute` as we used in `music.n3` and `mp3s.n3` for the namespace definitions. Using `OntologyAttribute` plus `OntologyName` allows you to define all attributes as relative URIs which makes for a much more readable source file. The `OwlResourceAttribute` defines our .NET class `RdfMusic.Track` to correspond with the OWL class `http://aabs.purl.org/ontologies/2007/`
Likewise the `FileLocation` property defined on it corresponds to the RDF datatype property `http://aabs.purl.org/ontologies/2007/04/music#fileLocation`. The Boolean true on these attributes simply tells LinqToRdf that the URIs are

relative. It then knows enough to be able to work out how to query for the details needed to fill each of the properties on the class Track.

This approach is deliberately as close as possible to LINQ to SQL. It is hoped that those who are already familiar with DLINQ (as LINQ to SQL used to be known) will be able to pick this up and start working with it quickly. In DLINQ, instead of URIs for resources defined in an ontology, you would find table and column names.

Thats all you need to be able to model your ontology classes in .NET. Now we can move on to the techniques needed to query your RDF triple store.

2.7 Querying the ontology using SPARQL

The steps to start making queries are also pretty simple. First just create a simple LINQ enabled console application called `MyRdfTest`. Open up `Program.cs` up for editing, and add namespace import statements for `System.Linq`, `LinqToRdf` and `SemWeb`:

```
using System;
using LinqToRdf;
using System.Linq;
```

In `Main`, create a `TripleStore` object with the location of the SPARQL server:

```
private static void Main(string[] args)
{
    TripleStore ts = new TripleStore();
    ts.EndpointUri = @"http://localhost/linqtordf/SparqlQuery.aspx";
    ts.QueryType = QueryType.RemoteSparqlStore;
```

`TripleStore` is used to carry any information needed about the triple store for later use by the query. In this case I set up an IIS virtual directory on my local machine called `linqtordf`, and followed the steps outlined early. The `QueryType` just indicates to the query context that we will be using SPARQL over HTTP. That tells it what types of connections, commands, XML data types and the query language to use.

Now were ready to perform the LINQ query. Well get all of the tracks from 2007 that have a genre name of "Rory Blyth: The Smartest Man in the World". Well create a new anonymous type to store the results in, and were only interested in the `Title` and the `FileLocation`.

```
var q = from t in new RDF(ts).ForType<MyTrack>()
    where t.Year == "2007" &&
    t.GenreName == "Rory Blyth: The Smartest Man in the World"
    select new {t.Title, t.FileLocation};
```

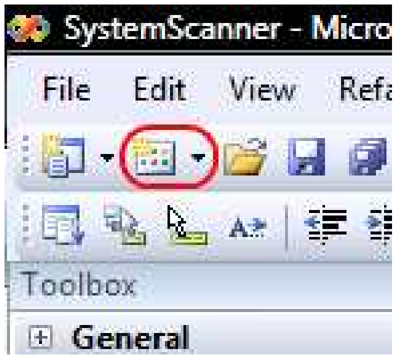


Figure 3: The New Items button.

Then well just iterate over the results and wait for a keypress before quitting.

```
foreach(var track in q){
    Console.WriteLine(track.Title + ": " + track.FileLocation);
}
Console.ReadKey();
```

Thats all there is to it. Of course theres a lot more going on behind the scenes, but the beauty of LINQ is that you dont need to see all of that while youre only interested in getting some `Tracks` back! In the references section Ill give some links that you can go to if you want to know whats going on under the hood.

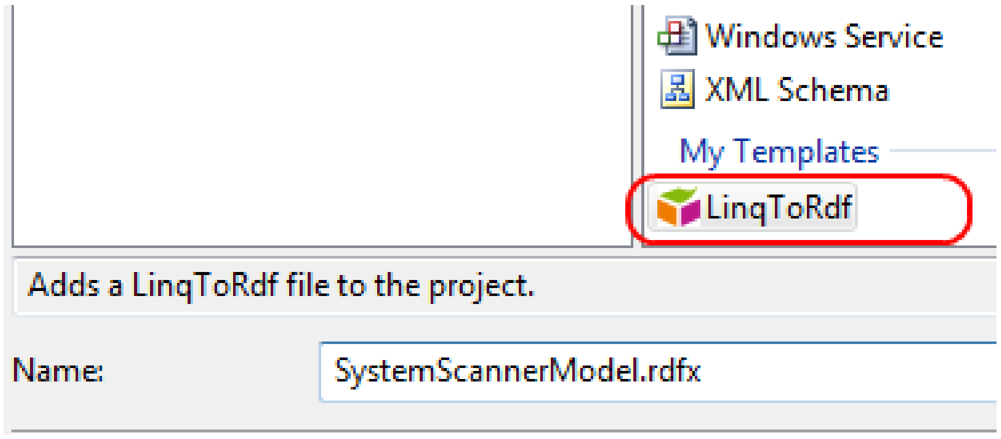
2.8 Using the graphical designer to design an ontology

The process described in previous sections can be done easily using the `LinqToRdfDesigner`. This section describes the tasks required to set up a very simple ontology of two classes called `Artifact` and `Assembly`. The program associated with it is available from the google code site via subversion. Its purpose is to gather various bits of information about the running system and store them as objects in an N3 file for later use. We wont explore too much of the application except where it depends on LinqToRdf or LinqToRdfDesigner.

The first task is to create a design surface to draw the domain model on. The file extension for LinqToRdfDesigner files is `rdfx`. Click on the New Item button on the standard toolbar.

The item templates selection dialog will then appear. You will locate the LinqToRdf item template at the bottom, under My Templates. If it is not there, then check that the LinqToRdfDesigner has been installed on your machine.

Change the name of the file to `SystemScannerModel.rdfx`. This will create several files in your project.



Adds a LinqToRdf file to the project.

Name: SystemScannerModel.rdfx

Figure 4: The file templates selection dialog showing the LinqToRdf designer template.

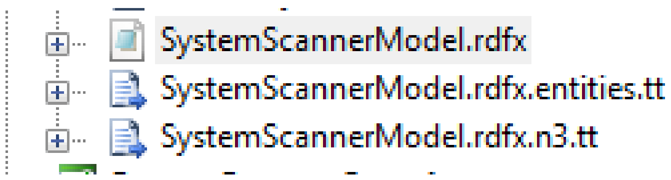

SystemScannerModel.rdfx
SystemScannerModel.rdfx.entities.tt
SystemScannerModel.rdfx.n3.tt

Figure 5: The resulting files using the name that you provide.

SystemScannerModel.rdfx is the design surface file. SystemScannerModel.rdfx.entities.tt is a text template file that is used to generate the C# file containing the base definitions of your .NET entity model. SystemScannerModel.rdfx.n3.tt is the text template that is used to generate the N3 format ontology definition that will be used by LinqToRdf and its components.

Double click on the rdfx file to open the design surface. You should now notice that the toolbox contains elements for modeling your classes.

Drag a class from the toolbox onto the design surface.

By default, it will be called something like ModelClass1. Rename it to Artifact. This class also needs a name within the ontology. In semantic web applications, classes are named using URIs. You will need to choose a URI for the class. Well get onto that in a moment. For now, press the F4 button an enter Artifact in the Owl Class Uri property in the property window.

Your properties window should now look like this.

Right click on the class shape in the designer and click on the Add New Model Attribute menu option.

Add an attribute (a field in .NET parlance) to the class and call it ArtifactExists. Again, use the properties window to set up some properties for the attribute. Initially it will look like this.

Change the properties to be like this.

What were saying here is that the .NET property ArtifactExists of .NET type bool corresponds to the OWL URI artifactExists which LinqToRdf will convert into the XML Schema Datatype xsdt:boolean. You can ignore the description property, for the moment, since its not currently used in the designer text templates. Later on we will see how these extra properties can be used within the text templates to generate documentation for both C# and N3.

The rest of the attributes attached to the artifact class are shown below.

Their types should be easy enough to guess, and the OWL URIs are just the same, except using camel rather than Pascal case, since that is the norm in N3.

Next drag another class from the toolbox onto the design surface and call it 'Assembly'.

Use the inheritance tool from the toolbox to connect from it to the class Artifact. This will connect the classes using direct inheritance in C# and owl:subclass in N3. Your domain model is now taking shape.

Youre now at a stage where you have something worth converting into code! Click on the transform all templates button on the Solution Explorer toolbar.

This processes the text templates that were created earlier, supplying them with the object model that youve built up over the last few steps. The output should look like this.

If you look inside the code generated for the entities, it should look like this.

```
namespace SystemScannerModel
{

    [OwlResource(OntologyName = "$fileinputname$",
        RelativeUriReference = "Artifact")]
```

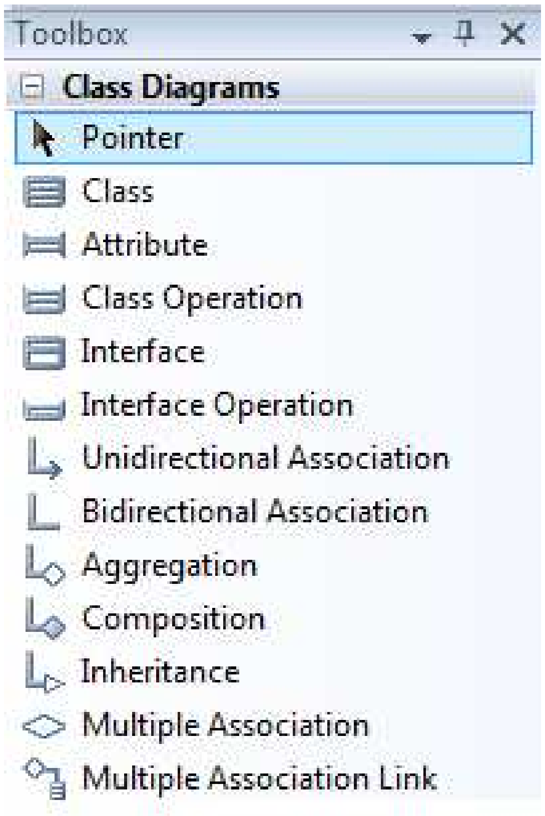


Figure 6: The toolbox when you're editing a LinqToRdf design.

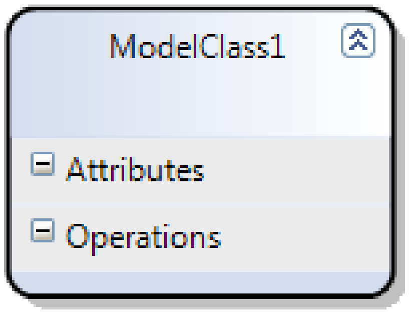
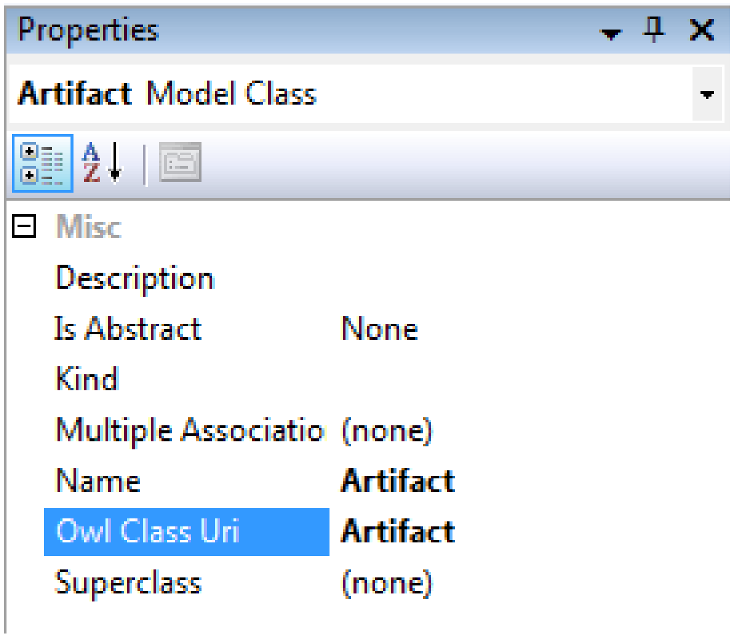


Figure 7: The default appearance of a class when first dragged onto a design surface.


Properties

Artifact Model Class

Misc

Description	
Is Abstract	None
Kind	
Multiple Associatio	(none)
Name	**Artifact**
Owl Class Uri	**Artifact**
Superclass	(none)

Figure 8: Editing the properties of a class on the design surface.

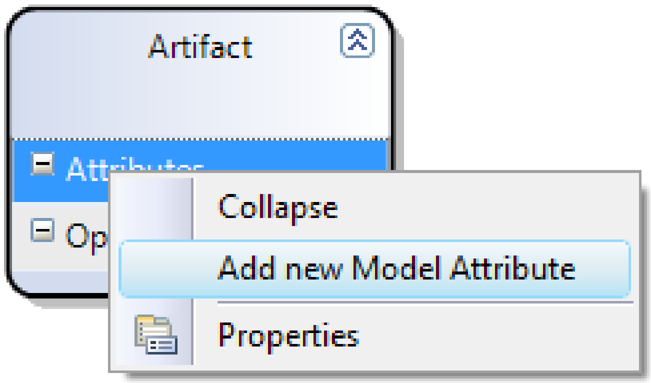
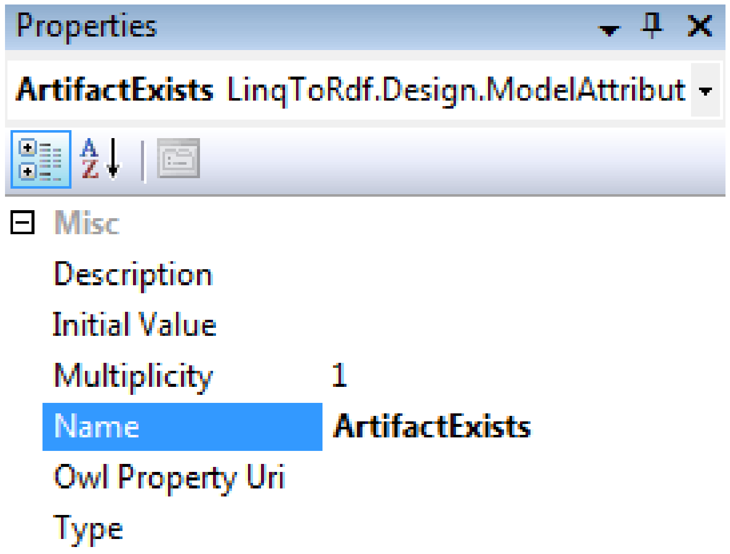


Figure 9: Adding a new model attribute.



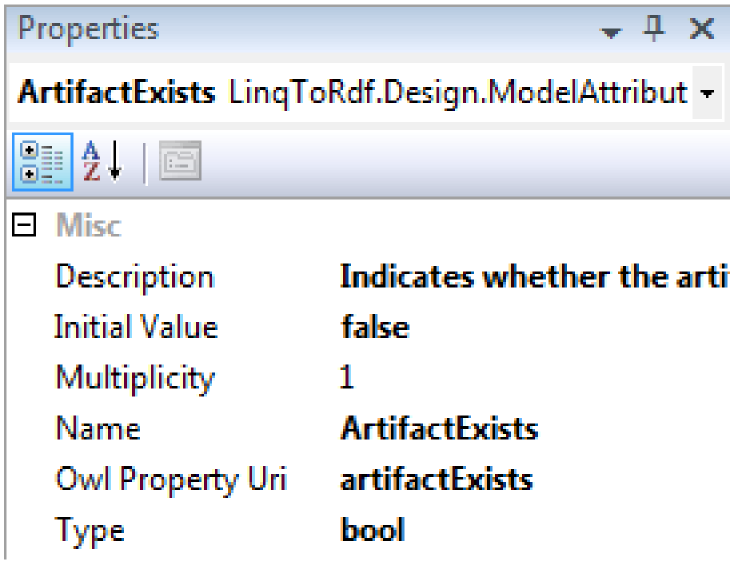
Properties

ArtifactExists LinqToRdf.Design.ModelAttribut

Misc

Description	
Initial Value	
Multiplicity	1
Name	ArtifactExists
Owl Property Uri	
Type	

Figure 10: The default properties of a class property.



Properties

ArtifactExists LinqToRdf.Design.ModelAttribut

Misc

Description	Indicates whether the arti
Initial Value	false
Multiplicity	1
Name	ArtifactExists
Owl Property Uri	artifactExists
Type	bool

Figure 11: A properly specified attribute has a reference to the property URI that matches the object or data property in the ontology.

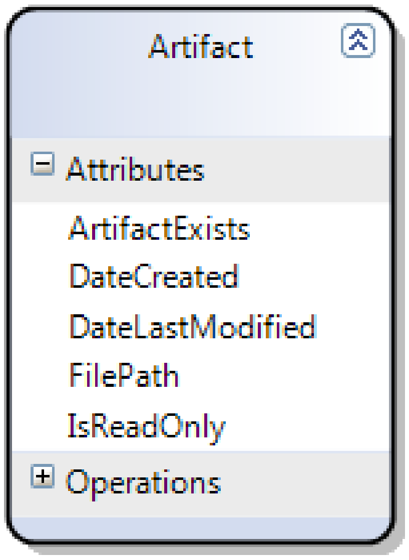


Figure 12: The class with all of the properties defined.

```
public partial class Artifact : OwlInstanceSupertype
{
    [OwlResource(OntologyName = "SystemScannerModel ",
    RelativeUriReference = "artifactExists")]
    public bool ArtifactExists { get; set; }
    [OwlResource(OntologyName = "SystemScannerModel ",
    RelativeUriReference = "dateCreated")]
    public DateTime DateCreated { get; set; }
    [OwlResource(OntologyName = "SystemScannerModel ",
    RelativeUriReference = "dateLastModified")]
    public DateTime DateLastModified { get; set; }
    [OwlResource(OntologyName = "SystemScannerModel ",
    RelativeUriReference = "filePath")]
    public string FilePath { get; set; }
    [OwlResource(OntologyName = "SystemScannerModel ",
    RelativeUriReference = "isReadOnly")]
    public bool IsReadOnly { get; set; }
}

[OwlResource(OntologyName = "$fileinputname$",
    RelativeUriReference = "assembly")]
public partial class Assembly : Artifact
{
    [OwlResource(OntologyName = "SystemScannerModel ",
    RelativeUriReference = "isSigned")]
```

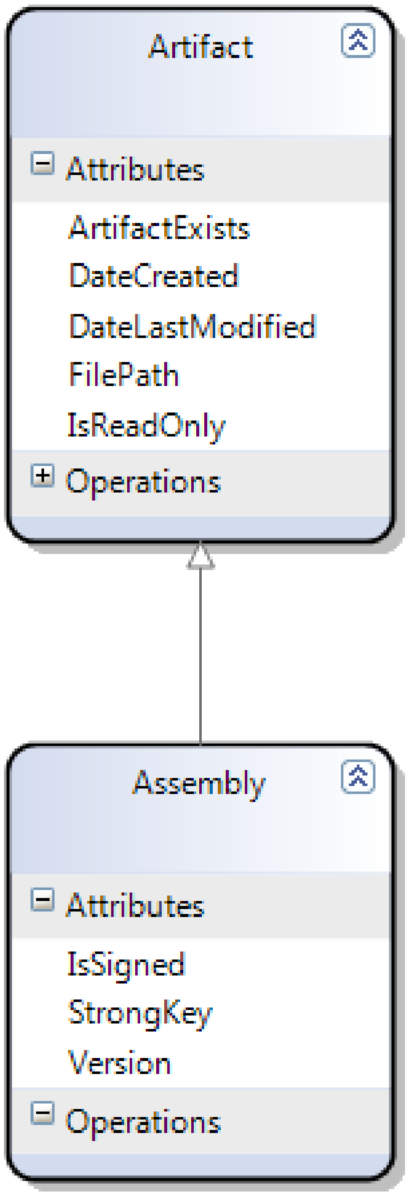


Figure 13: Two classes related by an inheritance relationship.

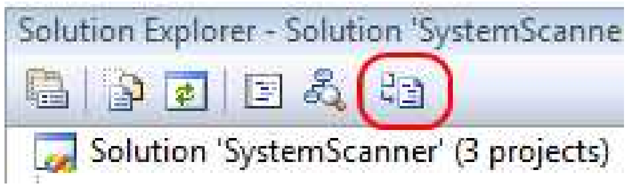
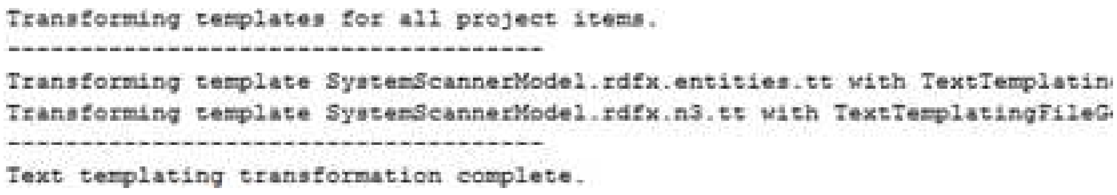

Solution Explorer - Solution 'SystemScanne
Solution 'SystemScanner' (3 projects)

Figure 14: Invoking the code generator, using the 'transform all templates' button.



```
Transforming templates for all project items.
----------------------------------------
Transforming template SystemScannerModel.rdfx.entities.tt with TextTemplatin
Transforming template SystemScannerModel.rdfx.n3.tt with TextTemplatingFileG
----------------------------------------
Text templating transformation complete.
```

Figure 15: The output from running the code generator.

```
        public bool IsSigned { get; set; }
        [OwlResource(OntologyName = "SystemScannerModel ",
        RelativeUriReference = "strongKey")]
        public string StrongKey { get; set; }
        [OwlResource(OntologyName = "SystemScannerModel ",
        RelativeUriReference = "version")]
        public string Version { get; set; }
    }
}
```

Here is an example of a unit test that grabs the data from the assembly. `ArtifactStore` is a wrapper around a SemWeb `MemoryStore`.

```
string loc = @"C:\\etc\\dev\\prototypes\\linqtordf\\"
    +"SystemScanner\\rdf\\sys.artifacts.n3";
ArtifactStore store = new ArtifactStore(loc);
Dictionary<string, AssemblyName> tmpStore =
    new Dictionary<string, AssemblyName>();
Extensions.Scan(GetType().Assembly.GetName(), tmpStore);
foreach (AssemblyName asmName in tmpStore.Values)
{
    store.Add(new SystemScannerModel.Assembly(asmName.GetAssembly()));
}
Assert.AreEqual(344, store.TripleStore.StatementCount);
```

Before this code will build you need to add a few assembly references to

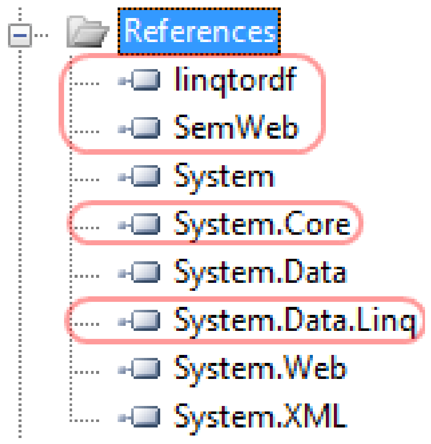


Figure 16: Necessary assembly references for working with LinqToRdf.

LinqToRdf, SemWeb and LINQ to your project.

Now rebuild your solution. Assuming there are no other compile time errors in the system it should build OK. You are now able to perform LinqToRdf queries against the ontology using the techniques described previously.

3 Navigating relationships in LinqToRdf

The representation and querying of relationships in RDF is very different from the model employed in relational databases. At first glance, looking at LINQ to SQL, you might think that that LINQ itself was tied to the relational model, but that is not the case. LINQ is a fully extensible platform allowing us to represent other kinds of relationship through the use of query operators.

In LinqToRdf you can use the `StmtObjectWithSubjectAndPredicate` and `StmtSubjectWithObjectAndPredicate` to instruct it to navigate the relationships using SPARQL syntax. here is an example of `StmtObjectWithSubjectAndPredicate` in action:

```
var ctx = new MusicDataContext(@"http://localhost/linqtordf/SparqlQuery.aspx");
var album = (from a in ctx.Albums
              where a.Name.StartsWith("Thomas")
              select a).First();

string recordUri = album.InstanceUri;
string trackPredicateUri = album.PredicateUriForProperty(MethodBase.GetCurrentMethod());
var tracks = from t in ((MusicDataContext)DataContext).Tracks
where t.StmtSubjectWithObjectAndPredicate(recordUri, trackPredicateUri)
select t;
```

Given an object called `album` whose instance URI we already know, we can select through properties that we know reference that URI. The definition of property `Album` in class `Track` looks like this:

```
[OwlResource(OntologyName = "Music", RelativeUriReference = "isTrackOn")]
public Album Album
{
    get
    {
//...
```

The combination of `t`, `Album` and `StmtSubjectWithObjectAndPredicate` is enough for LinqToRdf to be able to generate SPARQL with a triple like this:

```
$t ns1:isTrackOn ns1:Album_2 .
```

Which will filter all tracks except those that are tracks on `ns1:Album_2`. To support this, LinqToRdf automatically retrieves and stores the instance URIs for all ontology classes. If you use projections, then you don't get the `InstanceUri` property, so you will have to construct your collections manually.

LinqToRdf also stores a reference to the DataContext from an instance was retrieved with the instance itself. An instance can therefore define parent child collections like this:

```
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)
        {

            string recordUri = this.InstanceUri;
            string trackPredicateUri =
      this.PredicateUriForProperty(MethodBase.GetCurrentMethod());
            _Tracks.SetSource(
                from t in ((MusicDataContext)DataContext).Tracks
                where t.StmtSubjectWithObjectAndPredicate(recordUri,
 trackPredicateUri)
                select t);
        }
        return _Tracks;
    }
}
```

This uses the EntitySet container that can be primed with a query based on the `DataContext` and `InstanceUri` of `this`. It will store the query and only invoke it on first demand. You need to beware that the `DataContext` has not gone out of scope before the query gets invoked. That constraint is in common with LINQ to SQL.

```
using System.Data.Linq;
using System.Linq;
using System.Reflection;
using ID3Lib;
using LinqToRdf;

namespace RdfMusic
{
  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>(); }
    }
  }

  [OwlResource(OntologyName = "Music",
    RelativeUriReference = "Track")]
  public class Track : OwlInstanceSupertype
  {
    public Track(TagHandler th, string fileLocation)
    {
      FileLocation = fileLocation;
      Title = th.Track;
      ArtistName = th.Artist;
      AlbumName = th.Album;
      Year = th.Year;
      GenreName = th.Genere;
      Comment = th.Comment;
    }

    public Track()
    {
    }

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

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

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

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

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

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

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

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

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

    [OwlResource(OntologyName = "Music",
      RelativeUriReference = "isTrackOn")]
    public Album Album
```

```
  {
    get
    {
      if (_Album.HasLoadedOrAssignedValue)
        return _Album.Entity;
      if (DataContext != null)
      {
        var ctx = (MusicDataContext) DataContext;
        string trackUri = InstanceUri;
        string trackPredicateUri =
  this.PredicateUriForProperty(MethodBase.GetCurrentMethod());
        _Album = new EntityRef<Album>(
          from r in ((MusicDataContext) DataContext).Albums
          where r.StmtObjectWithSubjectAndPredicate(trackUri,
    trackPredicateUri)
          select r);

        return _Album.Entity;
      }
      return null;
    }
  }
}

[OwlResource(OntologyName = "Music",
  RelativeUriReference = "Album")]
public class Album : OwlInstanceSupertype
{
  private readonly EntitySet<Track> _Tracks =
    new EntitySet<Track>();

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

  [OwlResource(OntologyName = "Music",
    RelativeUriReference = "isTrackOn")]
  public EntitySet<Track> Tracks
  {
    get
    {
      if (_Tracks.HasLoadedOrAssignedValues)
        return _Tracks;
      if (DataContext != null)
      {
        string recordUri = InstanceUri;
```

```
          string trackPredicateUri =
    this.PredicateUriForProperty(MethodBase.GetCurrentMethod());
          _Tracks.SetSource(
            from t in ((MusicDataContext) DataContext).Tracks
            where t.StmtSubjectWithObjectAndPredicate(recordUri,
      trackPredicateUri)
            select t);
        }
        return _Tracks;
      }
    }
  }
}

public static void Main()
{
    var ctx = new MusicDataContext(
      @"http://localhost/linqtordf/SparqlQuery.aspx");
    var track = (from t in ctx.Tracks
                 where t.StmtObjectWithSubjectAndPredicate(
   "http://aabs.purl.org/ontologies/2007/04/music#Track_-861912094",
           "http://aabs.purl.org/ontologies/2007/04/music#isTrackOn")
                 select t).First();

    Debug.WriteLine("Track was " + track.Title);
    Debug.WriteLine("Album was " + track.Album.Name);

    foreach (var t in track.Album.Tracks)
    {
        Debug.WriteLine("Album Track was " + track.Title);
    }
}
```