GETTING STARTED WITH LINQ TO RDF

Semantic web applications in .NET

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.

Andrew Matthews

6/19/2007

GETTING STARTED WITH LINQ TO RDF

Semantic web applications in .NET

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. It's beyond the scope of this document to explain the whole pyramid of standards and technologies needed to support the semantic web. Instead I'll give context enough for you to know what steps are required to get your Semantic Web Application working.

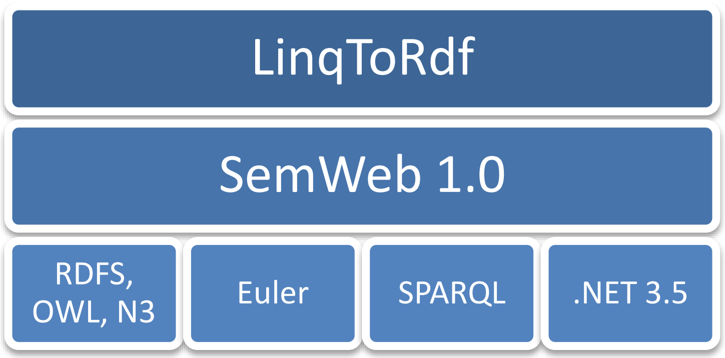


Figure 1 Major technologies employed with LinqToRdf

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. The current version at time of writing is targeted at Visual Studio .NET 2008 beta 2.

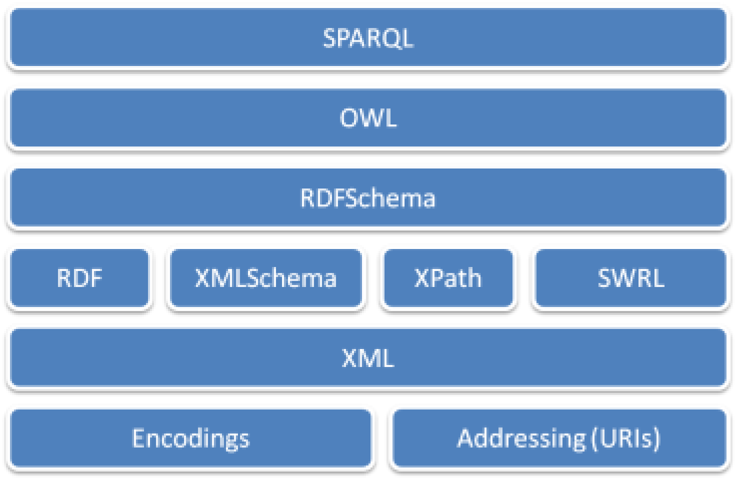


Figure 2 The semantic web technology stack

Figure 2, above, shows the hierarchy of technologies that are involved in the semantic web.

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

LinqToRdf requires an IDE that supports the latest version of LINQ, which at the time of writing is Visual Studio .NET 2008 Beta 2. The 3.5 version of the .NET framework that comes with beta 2 has a go-live license, so it should not change substantially prior to release.

Get the latest release, if you haven't already, and install it.

Where to get LinqToRdf

You can download the latest release from Google code (latest version at time of writing is LinqToRdf-0.4.msi) . If you want to use the visual designer in VS.NET 2008 beta 2, you will also need to download and install LinqToRdfDesigner-0.4.msi, also available for download from the Google Code website. Check the linqtordf-discuss discussion forum for announcements of newer releases.

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*". It is recommended that you stay with that location. 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.

CREATING AN ONTOLOGY

The details of the OWL standard are beyond the scope of this document. The standards document is found at the W3C[1]. There are various tools available for creating RDF, and it is important to know that the SemWeb library can understand the Notation 3 syntax[2][3], which is a human readable (non-XML) variant of RDF. The examples we'll 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 Protégé. LinqToRdfDesigner will be covered in a later section. This section shows how the LinqToRdf framework can be used without any other support.

Let's start with a simple ontology for recording MP3 files. The ontology file should have an extension of n3. Let's 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#> .
```

[1] The best place to start is with http://www.w3.org/2004/OWL/ - the specification for OWL. You should also be aware of http://www.w3.org/TR/2000/CR-rdf-schema-20000327/ which is the specification for RDFS (RDF Schema).
[2] See Tim Berners-Lee's Guide for further information: http://www.w3.org/2000/10/swap/Primer
[3] For a quick reference guide to the syntax of N3, you should also look at http://aabs.wordpress.com/semantic-web/n3guide which provides numerous examples of how to achieve common tasks.

```
@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 let's 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 &c). Because the prolog section previously defined a default namespace, these declarations are now in the `<http://aabs.purl.org/ontologies/2007/04/music#>` namespace. That's all that's 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: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: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: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. ☺ I'll 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 doesn't 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 that's 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 we'll 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).

That's it. That's all there is to creating an ontology. Later on, we'll 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.

HOSTING YOUR ONTOLOGY

Since the uptake of semantic web technologies has been pretty patchy in the .NET domain your best bet for industrial strength RDF triple stores will (for now) lie in the Java domain, and there are various triple store solutions that can be used. For this guide I shall stick to .NET by using Joshua Tauberer's 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, which runs on Java. LinqToRdf is platform independent.

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 doesn't matter – it doesn't 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 SPARQL Protocol implementation with a path on your
        website. With this, you get a SPARQL 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. That's all that's needed to turn your ASP.NET into a semantic web triple store! Yes, it's 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. Here's 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.

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. We'll see an easier way to do this later on, but for now we're 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 { return title; }
                set { title = value; }
        }
        [OwlResource(OntologyName = "Music", RelativeUriReference="artistName")]
        public string ArtistName
        {
                get { return artistName; }
                set { artistName = value; }
        }
        [OwlResource(OntologyName = "Music", RelativeUriReference="albumName")]
        public string AlbumName
        {
                get { return albumName; }
                set { albumName = value; }
        }
        [OwlResource(OntologyName = "Music", RelativeUriReference="year")]
        public string Year
        {
                get { return year; }
                set { year = value; }
        }
        [OwlResource(OntologyName = "Music", RelativeUriReference="genreName")]
        public string GenreName
        {
                get { return genreName; }
                set { genreName = value; }
        }
        [OwlResource(OntologyName = "Music", RelativeUriReference="comment")]
        public string Comment
        {
                get { return comment; }
                set { comment = value; }
        }
        [OwlResource(OntologyName = "Music", RelativeUriReference="fileLocation")]
        public string FileLocation
        {
                get { return fileLocation; }
                set { fileLocation = value; }
        }
        [OwlResource(OntologyName = "Music", RelativeUriReference="rating")]
        public int Rating
        {
                get { return rating; }
                set { rating = value; }
        }
        private string title;
        private string artistName;
        private string albumName;
        private string year;
        private string genreName;
        private string comment;
        private string fileLocation;
        private int rating;
        public Track(TagHandler th, string fileLocation)
        {
                this.fileLocation = fileLocation;
                title = th.Track;
                artistName = th.Artist;
                albumName = th.Album;
                year = th.Year;
                genreName = th.Genere;
                comment = th.Comment;
        }
        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[4]. 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/2007/04/music#Track. 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.

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

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

[4] Navigate to http://linqtordf.googlecode.com/svn/trunk/src/linqtordf/Attributes.cs for more information

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 we're ready to perform the LINQ query. We'll get all of the tracks from 2007 that have a genre name of "*Rory Blyth: The Smartest Man in the World*". We'll create a new anonymous type to store the results in, and we're 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};
```

Then we'll just iterate over the results and wait for a keypress before quitting.

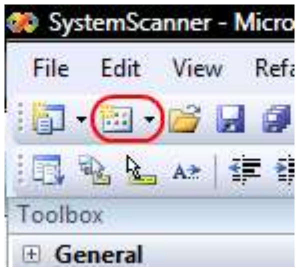
```
foreach(var track in q){
        Console.WriteLine(track.Title + ": " + track.FileLocation);
}
Console.ReadKey();
```

That's all there is to it. Of course there's a lot more going on behind the scenes, but the beauty of LINQ is that you don't need to see all of that while you're only interested in getting some Tracks back! In the references section I'll give some links that you can go to if you want to know what's going on under the hood.

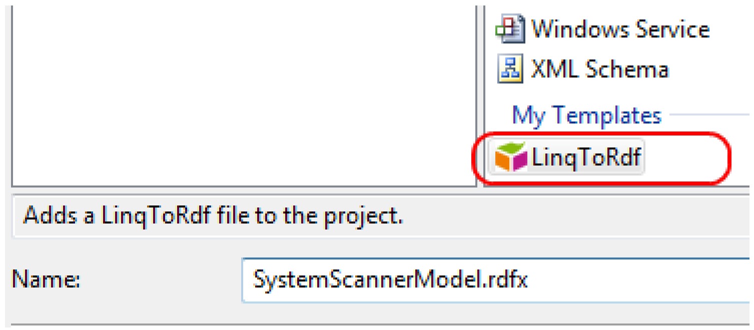
USING THE GRAPHICAL DESIGNER TO DESIGN AN ONTOLOGY

The process described in previous sections can be done quite 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. It's 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 won't 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.



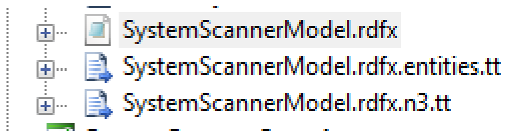
The item templates selection dialog will them appear. You will locate the LinqToRdf item template at the bottom, under My Templates. If it is not there, then check that the LinqToRdfDesigner has been installed on your machine.



Adds a LinqToRdf file to the project.

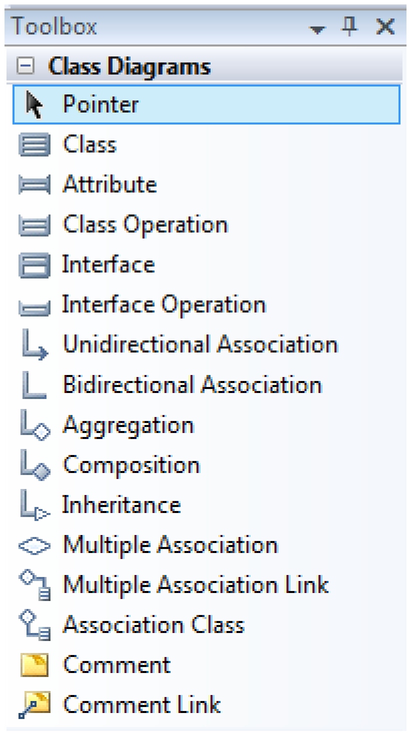
Name: SystemScannerModel.rdfx

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

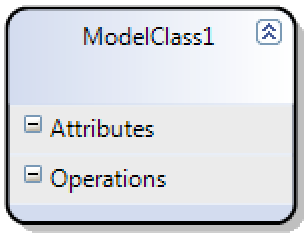

SystemScannerModel.rdfx
SystemScannerModel.rdfx.entities.tt
SystemScannerModel.rdfx.n3.tt

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 you .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.

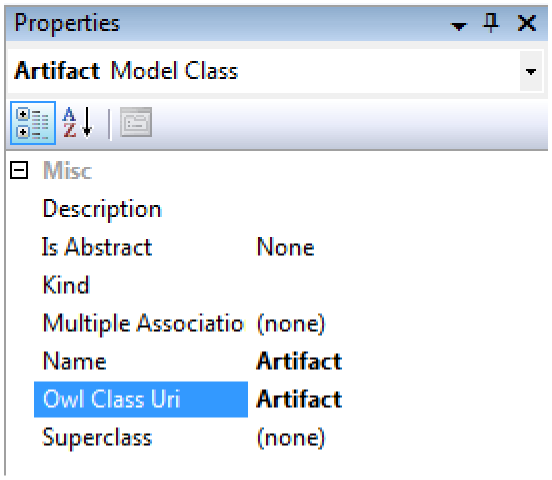


Drag a class from the toolbox onto the design surface.



By default, it will be called something 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. We’ll 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.

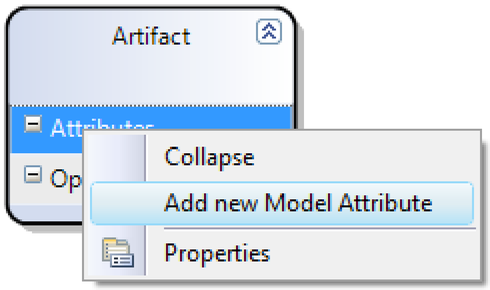

Properties

Artifact Model Class

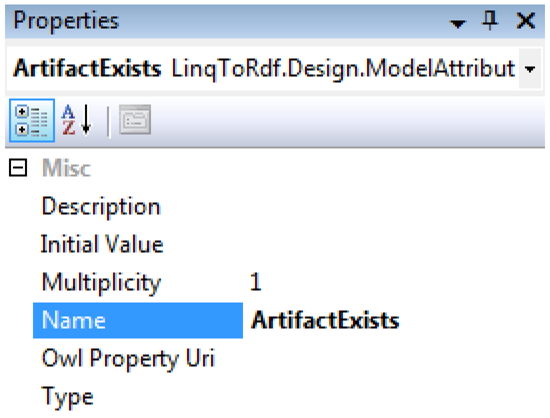
Misc

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

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



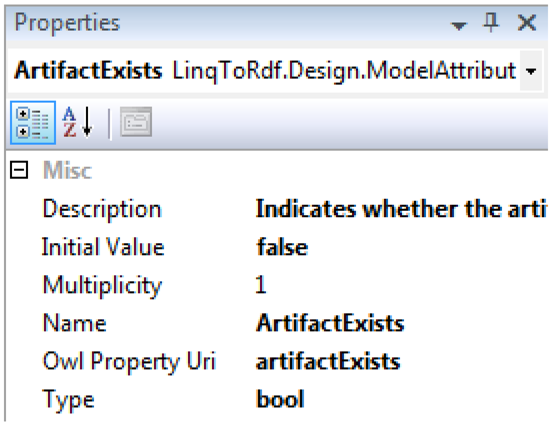
Add an attribute (a field in .NET parlance) to the class and call it ‘ArtifactExists’. Again, use the properties window to set up some properties for the attribute. Initially it will look like this.


Properties

ArtifactExists LinqToRdf.Design.ModelAttribut

Misc	
Description	
Initial Value	
Multiplicity	1
Name	ArtifactExists
Owl Property Uri	
Type	

Change the properties to be like this.

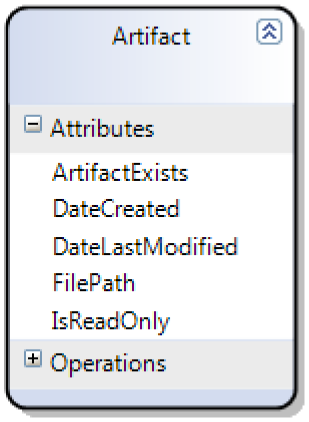

Properties

ArtifactExists LinqToRdf.Design.ModelAttribut

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

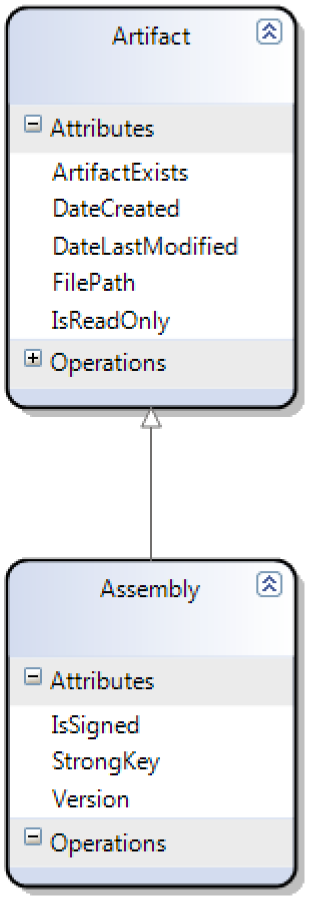
What we're saying here is that the .NET property 'ArtifactExists' of .NET type bool corresponds to the OWL URI 'artifactExists' which LinqToRdf will convert into the XML Schema Datatype 'xsdt:boolean'. You can ignore the description property, for the moment, since it's not currently used in the designer 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.



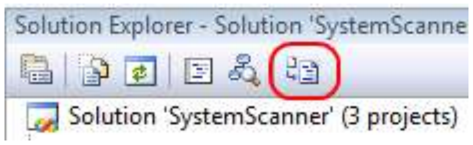
Their types should be easy enough to guess, and the OWL URIs are just the same, except using camel rather than Pascal case, since that is the norm in N3.

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.

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


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

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

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

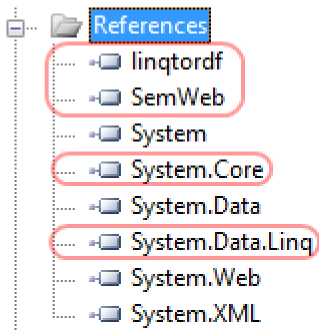
If you look inside the code generated for the entities, it should look like this.

```
namespace SystemScannerModel
{

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

REFERENCE MATERIAL