# dotNetRDF Design Document

## dotNetRDF Version 0.4.1

Library: dotNetRDF.dll  
Version: 0.4.1  
Target Date: April 2011  
Author: Rob Vesse  
Proposed Implementer: Rob Vesse  
Last Updated:

## Required Features

* Node API restructuring (Completed)
* Parser Subsystem restructuring (Completed)
* Enable Native SPARQL Update support for 4store

## Time Permitting Features

* Thread safe dataset management for Leviathan (re: Active & Default Graph)
* Improved persistent graphs (Completed)

## Known Issues/Bugs to Fix

* None at present

# Design

## Node API Restructuring (Completed)

It has come to our attention that having concrete classes for each node type and no specific node interfaces limits extensibility. Do a major refactor where each of the 5 node types gets a dedicated interface and all operations on nodes that use typed nodes are modified to use interface typed nodes.

## Parser Subsystem Restructuring

### 1 – Rewrite BlockingStreamReader (Completed)

* Rewrite the BlockingStreamReader to wrap a TextReader rather than just wrapping a StreamReader
* Add buffering to the reader internally so that it reads in a chunk of characters at a time rather than getting one character at a time
* Refactor StreamParams so that it is a special case of a new base class that just takes a TextReader for reading from Stores

### 2 – Add a new interface IRdfHandler (Completed)

* Rather than parsing directly into a Graph add an intermediate interface which allows parsing to arbitrary handlers. Methods related to creation of Nodes will most likely want abstracting into own separate interface (perhaps INodeFactory) and have IRdfHandler extend this:

IRdfHandler.StartRdf();

IRdfHandler.EndRdf();

IRdfHandler.HandleTriple(Triple t);  
IRdfHandler.HandleNamespace(String prefix, Uri namespaceUri);

IRdfHandler.HandleBaseUri(Uri baseUri);  
INodeFactory.CreateUriNode(Uri u);  
INodeFactory.CreateBlankNode() and INodeFactory.CreateBlankNode(String id);

INodeFactory.CreateLiteralNode() – all relevant forms

* Add some basic implementations
  + Abstract base implementation which implements the Node creation using an INodeFactory instance provided to the constructor or a temporary Graph if required
  + One that just asserts into an IGraph implementation – ensure that StartRdf() and EndRdf() are implemented in such a way as to ensure that parsing into a non-empty Graph causes a merge to happen
  + A writer which uses a given TripleFormatter to format Triples to an output stream
  + An implementation which uses an IGenericIOManager which supports triple level updates to write the Triples (in batches) to a Graph in an underlying store
* Add additional Load() methods to IRdfReader which take an IRdfHandler instance instead of an IGraph instance.
* Adapt BaseParserContext to use IRdfHandler instead of IGraph as the target of parsing
* Adapt all Parsers to utilise IRdfHandler instead of IGraph directly

## Enable Native SPARQL Update support for 4store

Should be fairly trivial since 4store now supports the majority of SPARQL Update – just add and implement the IUpdateableGenericIOManager interface to FourStoreConnector. May need to parse commands locally before sending them to the 4store to work around the lack of support for DELETE WHERE { } forms

## Thread Safe Dataset Management for Leviathan

As it stands currently while we have the useful ISparqlDataset abstraction it is now thread safe as currently implemented since a query on one thread may modify the dataset while a query on another thread is executing and thus alter the results of the others query. One option is to have the LeviathanQueryProcessor (and the LeviathanUpdateProcessor) effectively clone the dataset for each query/update evaluation since this would be a relatively cheap operation i.e. just copy the IInMemoryQueryableStore reference and the current Active and Default Graphs.

## Improved Persistent Graphs (Completed)

Abandon the current BackgroundPersistedGraph approaches (mark obsolete) in favour of the ModifableGraphWrapper (from dotNetRDF.Alexandria.dll) approach which does the persistence on disposal. Add a Discard() method which can be used to reverse the changes.

Refactor the basic implementation so that it is an abstract class which provides abstract methods to be implemented which perform the persistence.

* SupportsTripleLevelUpdates – Boolean property used to indicate whether the derived class supports the InsertTriples and DeleteTriples method
* InsertTriples(IEnumerable<Triple> ts) – Inserts a set of triples
* DeleteTriples(IEnumerable<Triple> ts) – Deletes a set of triples
* PersistChanges() – Persists the changes all at once (only used if Insert/Delete are not supported)

Add an implementation which used the existing ModifiableGraphWrapper persistence methodology to persist using an IGenericIOManager and the UpdateGraph()/SaveGraph() method.

Add an in-memory implementation where essentially the abstract functions do nothing (since there will be no changes to persist)