

# GSOC10 - Storage Service Implementations Based on Semantic Content Repository

## DSpace 2.0 Storage Service Implementations Based on Semantic Content Repository - Yigang Zhou

Develop DSpace storage service implementations based on semantic content repositories (TripleStore). - Yigang Zhou

1	<a href="#">DSpace 2.0 Storage Service Implementations Based on Semantic Content Repository - Yigang Zhou</a>
2	<a href="#">Abstract</a>
3	<a href="#">Architecture</a>
4	<a href="#">TupeloService</a>
4.1	<a href="#">TupeloService Functions</a>
4.2	<a href="#">TupeloService Access Policy</a>
4.3	<a href="#">Life Cycle Control of Tupelo Context</a>
5	<a href="#">Discussions</a>
5.1	<a href="#">TripleStore RDF API Battles</a>
5.2	<a href="#">Mappings Between DSpace and Tupelo</a>
5.2.1	<a href="#">Map StorageEntity into Tupelo Resource.</a>
5.2.2	<a href="#">Map StorageProperty into Tupelo metadata information triple.</a>
5.2.3	<a href="#">Map StorageBinary into Tupelo Blob Resource.</a>
6	<a href="#">Project Plan</a>
7	<a href="#">Future Work</a>

## Abstract

On the one hand, DSpace 2.0 has a generalized storage service API which allows a DSpace 2.0 repository to use many possible systems to store digital repository data. On the other hand, semantic content repositories (triplestores) such as Mulgara, Sesame and Tupelo are available for semantic data storage, which are suitable for storing blobs and metadata from DSpace represented in the form of RDF triples. In this project, I will develop DSpace storage service implementations based on semantic content repositories. Finally, I will cooperate with Andrius Blažinskas who is working on another GSoC 2010 project of back-porting DSpace 2.0 storage interfaces to 1.x, to make triplestore storage service ready to use for DSpace 1.x.

Project Title:	DSpace 2.0 Storage Service Implementations Based on Semantic Content Repository
Student:	Yigang Zhou, Wuhan University, P.R. China
Mentors:	Mark Diggory
Contacting author:	<i>egang</i> DOT <i>zhou</i> AT <i>gmail</i> DOT <i>com</i>
SCM Location for Project:	<a href="http://scm.dspace.org/svn/repo/sandbox/gsoc/2010/triplestore/">http://scm.dspace.org/svn/repo/sandbox/gsoc/2010/triplestore/</a>

## Architecture

The design principles of the architecture should be:

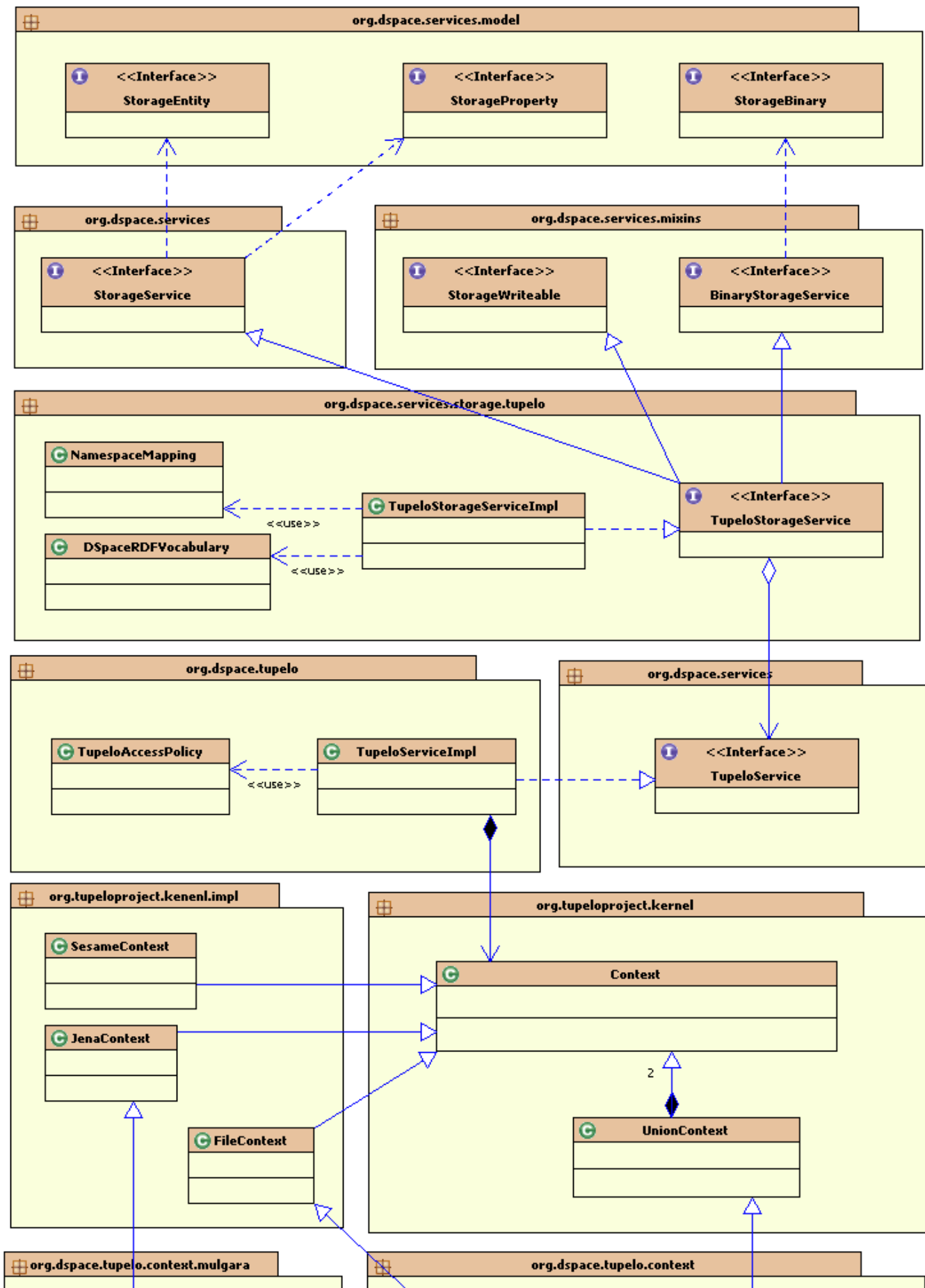
1. The triplestore StorageService can be compatible to all kinds of semantic data storages (e.g. Sesame, Jena, etc.), through different configuration settings.
2. Other new semantic data storages (e.g. Mulgara) can be easily plugin into the architecture without much efforts and need not modify code of the API.
3. The triplestore StorageService/BinaryStorageService should be able to accommodate both StorageEntity/StorageProperty for entity metadata information in the form of RDF triples and StorageBinary for blobs (binary/textual data).

This architecture is quite similar with JackrabbitStorageService, which sits in front of all kinds of PersistenceManagers for different databases.

As is shown in Figure 1, we have a `TupeloService` holding a reference to a `Context` (i.e. a triplestore instance in `Tupelo`) object to support low level triple/blob operations. The `Context` is actually a `UnionContext`, which combines a sub `Context A` (e.g. `SesameContext` or `MulguraContext`) for RDF triples and a blob-related sub `Context B` (e.g. `FileContext`). Additionally, high level functions like read/write transaction and object-triple mapping are also provided by `TupeloService` through `ThingSession` and `BeanSession` powered by `Tupelo`.

Based on `TupeloService`, `TupeloStorageService` should support `StorageService` and `BinaryStorageService`. All the functions related to `StorageService` will be dispatched to `Context A`, while those of `BinaryStorageService` can be delivered to `Context B`. It's quite flexible for the choices of `Context A` and `B`. No restrictions on the combination groups. For example, we can use `HashFileContext` or `DatabaseContext` for `Context A`, with `SesameContext`, `Sesame2Context` or `PersistenceJenaContext` as `Context B`. We can also use Spring configuration for `Context A` and `B` injections into the `UnionContext`. Currently, there's no `Mulgara` implementation of `Tupelo Context`. But I can develop a new `MulguraContext` in this GSoC project. The new `Context` will not affect the source code of `TupeloService` or `TupeloStorageService` at all. It can be easily plugin into the architecture through Spring configuration.

We separate `TupeloService` from `TupeloStorageService`, so that the API of `StorageService` is decoupled with different semantic triplestore implementations.



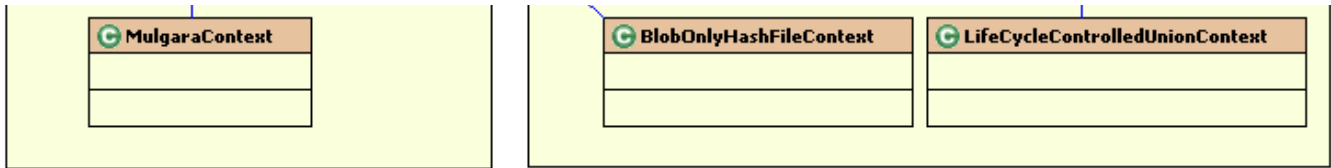


Figure 1. UML Diagram

## TupeloService

### TupeloService Functions

As is shown in the following table, TupeloService provides both triple and blob related functions in different levels. Low level functions are process-oriented: different Operators (e.g. TripleCounter, BlobFetcher) can be performed by TupeloService.perform() method. High level ones are object-oriented: users can create ThingSession or BeanSession to manipulate Thing (i.e. RDF Resource wrapper Class) objects, and make batch updates using sessions. Especially for BeanSession, user can create object-triple mapping and Tupelo will automatically manage the transformations from triples to objects/beans and vice versa.

	Low Level Functions	High Level Functions
	void TupeloService.perform(Operator operator)	ThingSession TupeloService.createThingSession(), BeanSession TupeloService.createBeanSession()
Triple Functions	TripleCounter, TripleMatcher, TripleFetcher, TripleWriter, TripleIterator, TripleReader, Unifier ...	Set<Thing> ThingSession.getThings(Resource predicate, Object value), void ThingSession.delete(Thing thing), void ThingSession.save(Thing thing) ...
Blob Functions	BlobFetcher, BlobIterator, BlobRemover, BlobWriter ...	ThingSession.removeBlob(Resource subject), ThingSession.fetchBlob(Resource subject), ThingSession.writeBlob(Resource subject, InputStream inputStream) ...

### TupeloService Access Policy

The policies are about how a TupeloService accesses an underlying Tupelo Context, when the Context may be existing or not. The policies are: open, access and renew.

Policy	Description
OPEN	Create an TupeloService instance that accesses an existing Tupelo Context. An Exception should be thrown, if the Tupelo Context does not exist or if it cannot be opened for any other reason.
ACCESS	Connect an TupeloService instance to a Tupelo Context. If the Context exists, connect to it; if it does not exist, create it.
RENEW	Create an TupeloService instance that accesses a Tupelo Context. If the Tupelo Context exists, it is deleted and replaced with a new empty Tupelo Context.

The policy can be configured by Spring as an argument of the constructor method of TupeloService:

```

<bean id="org.dspace.servivces.TupeloService" class="org.dspace.tupelo.TupeloServiceImpl">
    <constructor-arg ref="org.tupeloproject.kernel.Context" />
    <constructor-arg value="renew" />
</bean>

```

### Life Cycle Control of Tupelo Context

The life cycle methods of Tupelo Context are as follows:

Method	Description
void Context.initialize()	Some Context implementations require persistent resources (for instance storage space). For those that do, calling initialize will acquire those resources. When they are no longer needed, call destroy.

boolean Context.open()	Acquire resources necessary for performing operations (e.g., a database connection).
boolean Context.close()	dispose of any resources held by the context.
void destroy()	Release any persistent resources associated with this Context.

The access policy of TupeloService requests Tupelo Context to strictly control its life cycle. Actually Tupelo recommends doing in this way. But not all of the Context implementations follow this principle. For example, UnionContext lacks of life cycle control methods. Instead, we can use "org.dspace.tupelo.context.LifeCycleControlledUnionContext" to solve this problem. In short, all the Tupelo Contexts that used by TupeloService **MUST** control their life cycles appropriately, or unexpected behaviors will happen during startuping, accessing or shutdowning TupeloService.

## Discussions

### TripleStore RDF API Battles

Many triplestore implementations are "battling" to be the penultimate API that one would implement against with configuration of the others as underlying storage. The state-of-art of mainstream Java based triplestores are summarised as follows:

1. Tupelo defines its own RDF API and makes Jena, Sesame as its underlying storage in the form of Context.
2. AllegroGraph defines its own RDF API and provides Jena, Sesame wrapper classes for users to access AllegroGraph using Jena, Sesame API.
3. Mulgara use JRDF's RDF API and provides a bridge to Jena API.
4. Jena, Sesame defines their own stand alone RDF APIs.

In a word, their's no standard Java based RDF API. Neither is Tupelo. But we choose Tupelo as a facade/gateway for DSpace triplestore because:

1. Tupelo is designed naturally to be a pluggable RDF storage solution. Other triplestores can be easily developed as back-ends through extending Context, BaseContext, BasicLocalContext, etc.
2. Other triplestores can not store blobs, since they (e.g. binary files) can not be encoded into RDF triples. But Tupelo supports blob storage naturally and integrate it well with triple based metadata storage (using UnionContext).
3. Some triplestores support other RDF API through bridges or wrappers, but not underlying storages.

### Mappings Between DSpace and Tupelo

DSpace 2.0 contains StorageEntity, StorageProperty, StorageRelation and StroageBinary. How to map them into Tupelo RDF storage API?

- Map StorageEntity into Tupelo Resource.

The path(s) as metadata information triples. The entityId will be encoded into a Tupelo Resource URI.

DSpace	Tupelo
StorageEntity	Tupelo Resource
entityId	URI: "http://www.dspace.org/rdf/entity#" + escape(entityId)
entity path(s)	ds:hasPath (xsd:string)

- Map StorageProperty into Tupelo metadata information triple.

The datatype of the object of the triple is different according to different StorageProperty types. The StorageProperty name will be encoded into a Tupelo Resource URI.

DSpace	Tupelo
StorageProperty (different types)	Tupelo Triple
name	URI: "http://www.dspace.org/rdf/property#" + escape(name)
StorageEntity.class	Tupelo Resource
Boolean.class	xsd:boolean
Double.class	xsd:double

...	...
-----	-----

Map the property name to construct a RDF Resource. There are three cases:

(a) if the namespace can be recognized by NamespaceMapping, e.g.:  
dc:title -> "http://purl.org/dc/elements/1.1/title"

(b) if the namespace can not be recognized by NamespaceMapping, e.g.:  
eg:title -> "http://www.dspace.org/rdf/property#eg:title"

(c) if there's no namespace, e.g.:  
book title -> "http://www.dspace.org/rdf/property#book%20title"

We deal with (b) and (c) with the same policy.

- Map StorageBinary into Tupelo Blob Resource.

DSpace	Tupelo
StorageBinary	Tupelo Blob Resource (e.g. in file system)

## Project Plan

- Before mid evaluation (July 14th):
  - Design and develop TupeloService
  - Design and develop TupeloStorageService
  - Test TupeloStorageService with existing triplestore Context, e.g. SesameContext, JenaContext
- After mid evaluation
  - Develop MulgaraContext
  - Work with Andrius for back-porting DSpace 2.0 storage interfaces to 1.x.

## Future Work

The DSpace Storage API has recently been modified in another GSoC 2010 project by Andrius. In future, we should investigate how the changes will affect the TupleStorageService and make modifications if necessary.