RobsAssetStorePrototype

<?xml version="1.0" encoding="utf-8"?>
<html>

ob's Asset Store API & Prototype

The prototype is focused on working out a good interface between a DSpace+2.0 asset store and the rest of a DSpace system. (See proposal document and presentation from DspaceTwo for some details about what the asset store is and what it's meant to do.) None of this is meant to be set in stone, it's just trying out a couple of approaches.
For the impatient the API & prototype is here:
attachment:robs-proto-2005-04-25.zip
This prototype also includes a reasonable comprehensive set of unit tests (built using the http://junit.org/ JUnit framework) which give the prototype a bit of a workout, particularly relating to the transactionality.
You might find is easier to http://www.mit.edu/~rtansley/dspace/assetstore-proto-javadoc/ browse the Javadoc-generated API docs. In this doc set is a http://www.mit.edu/~rtansley/dspace/assetstore-proto-javadoc/org/dspace/filesystemassetstore/AssetStoreImpl.html description of the implementation.
Also check out JimsAssetStorePrototype and ichardsAssetStoreApi
While the interface itself is reasonably straightforwards, there's a lot of reasoning behind it; hence this discussion is considerably longer than the API!
In addition, the API is only part of the problem – what metadata packaging standard(s) we adopt is another big issue (see AipModel).

unning the tests

You can run the tests in Eclipse or using the provided Ant build file.
With Ant:

- You'll need the optional JUnit ant task. You probably have it already, only worry if you get errors.
- Make sure that `junit.jar` is in the `CLASSPATH`. Get it from http://junit.org.
- un `ant compile test` to compile & run the tests.
  With Eclipse:
  (If there's a way to export this with the project, please let me know!)
- Import the unzipped proto code into Eclipse (File -> Import... -> Existing Project into Workspace)
- Select un -> un... -> Select JUnit -> Click 'New'
- Enter `org.dspace.api.assetstore.test.AssetStoreTest` as the Test class
- Click on 'Arguments', enter the following under 'VW arguments' (but with no newlines):
  {{
  Unknown macro: {project_loc}
  \store
  storeTemp}}
  }
- Click 'un'. In theory, all 16 tests should pass!

TO DO

Summary of stuff still to work on for this, some discussed below

- AIP (metadata) checksums
- Allow for multi-AIP ('full') transactionality?
- Authorisation aspects
- Should there be an AIP object?
- itstream IDs
- Design pattern is for `AIPLock` to be a simple object. Maybe write operations should be methods on the `AIPLock` object? (i.e. rather than `AssetStore.setMetadata(lock, mdStream)`, you'd have `AIPLock.setMetadata(mdStream)`.
- Complete `StorePull` and `StorePush` implementation
- Determine our AIP format (minor task) See AipModel
- Prototype integratation into DSpace 1.x codebase

Notes on the API

Design Notes

I've stuck with the notion that, from 'above' the asset store API, an AIP looks conceptually rather like a file system directory, consisting of a metadata file, and bitstreams in the same 'directory'. This does not preclude any other storage mechanism (e.g. Grid-based like S) being used underneath; it just means the responsibility for managing the complexities of a mechanism like that lie with the implementation and those complexities are hidden from callers. I think this file system paradigm is familiar and intuitive to people.
I also think it's important to have the difference between metadata and bitstreams clearly evident in the API, rather than being implicit. It makes code more readable, which is important in open source where many people are using the code, and should reduce the probability of mistakes being made.
At present, the API isn't especially object oriented. I haven't created an 'AIP' class; this would be another potential way forward.

itstream IDs

esponsibility for identifiers is a tricky question. In this prototype, I've decided that the asset store should use the persistent identifiers (e.g. Handles) assigned by the system to store and retrieve AIPs. There doesn't seem to be any drawback to this, and introducing a different identifier for this purpose for the asset store API would seem to introduce pointless further complication.
(The database primary keys vs. Handles for communities, collections and items have been a source of confusion in DSpace 1.x.)
itstream IDs are slightly trickier. Different sites may have different policies about assigning IDs. e.g for MIT the decision was made not to give individual bitstreams Handles. So there are possibilities and questions for bitstream IDs:

- Eternally persistent (as is the intention with AIP IDs)?
- Unique globally, with an asset store or just an AIP?
- Deterministic? (e.g. SHA1 hashes)

There is also another dimension to this:

- Who assigns (or may assign) a bitstream identifier?
- Does the asset store need to be aware of a bitstream identifier
- Is the asset store responsible for storing and retrieving things based on a bitstream identifier? This is possible at higher layers

So the question is how should the **asset store API** identify bitstreams, which may or may not be the same answer as to how bitstreams are identified externally, e.g. to end users.
In DSpace 1.x, a bitstream has three identifiers:

- A semi-persistent identifier, consisting of the Handle concatenated with a sequence number. In practice this is expressed as a UL with a filename suffix to inform browsers that ignore MIME types what the file is, tying the bitstream to a particular location, e.g.:
  {`sequence number-`~~`https://dspace.myu.edu/bitstream/1234.567/1234/44/image5.gif--handle-----  -filename`~~}
- A database primary key in the bitstream table (bitstream_id)
- A completely arbitrary and large 'internal' ID, used to locate the file on the file system

This has been a source of confusion for 1.x, so reducing the number of identifiers something has in the system would be a good thing.
For this prototype API, I'm allowing the bitstream ID to be set by something externally to asset store. This seems important if both the above simplification is to be implemented, and the platform is to support a variety of ID schemes.
In the API, the bitstream ID is treated as an opaque String. The ``getitstream`` method does take the AIP ID as a parameter as well as the bitstream ID. If bitstream IDs are asset store and/or globally unique, then an asset store with an index of those IDs would not also need an AIP ID to get to the relevant bitstream. So we *could* add a method:
`{InputStream is = getitstream(String bitstreamID)}`
However, this would mean that bitstream IDs *have* to be at least asset store unique. Maybe a good thing? It doesn't seem very symmetrical to the set methods however. Need to decide on the bitstream issue before choosing.

Transaction safety/locking etc.

As noted above, I think a reasonable balance between transaction safety and ease of implementation is to have transaction-safety at the individual AIP level.
To reflect this, the proposed API introduces the notion of an 'AIP lock', which must be obtained prior to modifying an AIP. This AIP lock is a little like a classic 'file handle'. A caller must obtain an AIP lock, and use this to perform write operations. Naturally, if an AIP is already locked by another caller, this caller would not be able to get the lock.
This of course means that either callers need to close locks carefully, or have some garbage collection/timeout facility. oth of these are possible, and hopefully a decent automated test framework would be able to manage this.
Creation of an AIP automatically returns a lock; the AIP isn't really 'created' until the ``commit()`` method is called. This is so there isn't a 'blank' AIP in the system.
It may be possible to use this to allow (but not enforce) further transactionality by having extra methods that commit/rollback *sets* of AIP locks, e.g.
`{public void commit(AIPLock lock) throws AssetStoreException;}`
(or it could be a List, Iterator etc)
Not done that though.
Another potential situation: An AIP that many DSpaces replicate/mirror is modified by two DSpaces between some synchronisation schedule. Complexities of that should be dealt with inside the asset store itself. Not going to worry about this just yet.

Keeping indices & caches up-to-date with asset store contents

Push and pull models both possible. Will always need a 'bootstrapping' method to reindex/build cache from scratch; this is in the core ``AssetStore`` interface, as ``getAll``.
For incremental updates, there are two basic choices – 'push' (event/listener, messaging etc.) and 'pull' (periodic polling, similar to OAI-PMH). Each has relative advantages and disadvantages.
In the prototype API, I've provided an API for each – ``StorePush`` and ``StorePull``, both very simple.
It should only be necessary for an asset store implementation to implement one of these. Given the availability of an implementation of one, it should be very easy to use a small standard component to use that to implement the other, as shown below:
attachment:PushPullAPIs.gif
The 'message catcher' can receive update messages/events and use them to maintain an index suitable for polling. The Poller/messenger can poll a ``StorePull`` interface periodically, and use the results of this to send out 'AIP updated' messages/events. In the latter case, these messages may come in spurts right after polls; however, a DSpace system will need to be robust to this situation in any case, as things like preservation migrations, batch imports and so forth will cause 'clumps' of closely-packed updates in any case.
So it's quite possible that we can just say an asset store implementation has to implement *either* ``StorePush`` or ``StorePull``, and provide the bridge components so that every DSpace instance has access to both interfaces.

Auditing

An important operation the asset store needs to support is auditing. It is generally accepted in the digital preservation community that regular auditing of one's digital content is an essential preservation strategy, as digital storage media fail 'silently'; you cannot tell it has failed until you try to read it.
The proposed API supports this by allowing a caller to iterate through everything in the store (both metadata and bitstreams) to do this.
An open question here is to do with **checksums**.
For the metadata or the AIP as a whole there are various possibilities:

- An 'AIP checksum' for the whole AIP. This would be a checksum of the bitstreams and the metadata (the XML (METS) document) concatenated together. You'd need a easily-reproducable, documented concatenation algorithm. However, this would be pretty efficient for auditing a large number of AIPs – it requires the minimum amount of calculation. (You have to read everything once in any audit).
- Checksum the METS metadata. Then, checking the metadata's checksum is also implicitly verifying that the bitstream's checksums of record are intact, and these can be used to verify the actual bitstreams. This *may* be slightly less efficient than the above approach, as you must checksum the metadata, and then checksum bitstreams individually; however, I've no empirical data to back this up.
  In either case, this AIP/metadata checksum is slightly trickier to handle (and the proposed API doesn't deal with it yet.) The asset store could store the checksum separately, or it could be stored in the metadata package itself using http://www.w3c.org/Signature/ XML Signature or something similar.
  However, the problem is really the chicken-and-egg situation that arises during both AIP creation/modification and audit.
  During creation or modification, who actually creates the checksum? The asset store itself? It strikes me that the asset store isn't the best place for this to happen – if something is corrupted/incorrect being sent to the asset store, then the asset store will checksum the corrupt/incorrect data, and thus the data will appear to be correct during later audits. Perhaps the asset store should be provided with a checksum, which it should verify to ensure it's received everything correctly.
  During auditing, you're trusting the same source to provide the checksum and the data to checksum. For this to work the checksum needs to have been originally created by an impartial observer, or the original source of the AIP (ingest process or whatever). So this would need to be part of the asset store API specification.
  In both cases, it feels like the asset store should be provided with a checksum for the AIPs as opposed to creating one itself. It could (and maybe should!) verify this checksum itself.
  A similar situation exists for provenance information. If provenance/history information is part of the AIP, you have the same chicken-and-egg situation.
  TODO: More – perhaps this discussion should move to another forum

Use Cases/Scenarios

Some walk-through examples of how the API would be used in a variety of simple situations:

- Creating a community
- Creating a collection, adding to community
- Create item, add to collection
- Modify an item
- Audit an item in the asset store

TODO: (Will add these as code)

Authorisation

Tricky one this. There seem to be three possible approaches:
*__1__ The asset store interface/implementation doesn't deal with authorisation at all. It's the responsibility of other modules to verify that the action being performed is allowed, presumably by checking with the implementation of the authorisation interface the DSpace instance is using.
I like this approach because the asset store implementation doesn't have to worry about the authorisation part at all. Also, given that the asset store talks in relatively course-grained terms (e.g. the whole AIP metadata blob), it might not have enough information to be able to decide. E.g., say that a user is allowed to read or edit metadata field X, but not Y. You can't just say they can or can't update the metadata blob; it depends what they're changing.
On the other hand, it does seem like a bit of a security risk that any call just gets performed...
*__2__ equire the asset store interface/implementation verifies that what the user is doing allowed. This does open the door on the whole issue who's the current user, what are they trying to do etc.
*__3__ Some sort of layered approach, e.g.:
`{Moduleh4. h3. <- asset store interface with authorisation knowledgeAuthorisation filterh4. <- plain asset store interfaceAsset store implementation}`
The 'authorisation' wouldn't be the only authorisation part of the system, as other modules would need to know authorisation decisions as described above. So in fact, this layered approach is just kind of a coarse-grained 'safety net' variation on **2**, so that any module makes an error in checking the relevant authorisation.
This may result in a performance drain as authorisation may be checked twice (though with a decent, efficient authorisation system this needn't be a big deal).
For the time being I'll concentrate on an asset store interface and implementation that doesn't have to worry about authorisation.

</html>