BitstreamRelationships

Relationships Between Bitstreams

LarryStone, May 10, 2006

This page proposes a change to the DSpace object model to express relationships between the Bitstreams belonging to an item. It also adds calls to the "business logic" API to set and discover
these relationships.

Motivation

There are *already* relationships between Bitstreams (in DSpace 1.3 and 1.4) but they are not recorded in any formal or obvious way. For example, some of the MediaFilter classes add a derivative Bitstream for each of the "content" Bitstreams, such as a thumbnail image. To associate the thumbnail image with its full-size counterpart, the thumbnail Bitstream is given the same name as its master image with an extra image file extension tacked onto the end. So when the Web UI has the opportunity to display a thumbnail image of a Bitstream, it must know to look for a Bitstream with an extra ".jpg" appended to its name.
This *ad hoc* solution is not scalable and it violates modularity. The media filter plugin and the Web UI have to share a secret: the formula for naming thumbnail Bitstreams. For another application, e.g. a METS package disseminator, to associate thumbnail images with the original versions, it has to be let in on the secret.

It isn't scalable because every other media filter plugin with its own kind of derivative Bitstream has to invent its own bitstream-naming convention.

There is also the limitation that Bitstream names are actually metadata, and they need not be unique. It is perfectly legal for an Item to have two image Bitstreams named "illustration.gif", but the thumbnail naming convention breaks down.
However, my objection is not so much to the method (magic Bitstream names), as to the lack of a formal API to enforce the rules and make the Bitstream-association mechanism available to all other modules.

The generalization of this mechanism also helps several projects which need to associate metadata with an individual Bitstream, as opposed to the entire Item. The JHOVE integration will need a place to put the preservation metadata it generates for each content Bitstream. The CWSpace project has descriptive metadata at the Bitstream level for "learning objects" within Items. It goes on.

Defining the Problem

What are the reasons for associating one Bitstream with another? Here are the ones I've thought of; perhaps you can name others:

- Indicate its contents are *derived* from a *master* Bitstream for some special purpose, like full-text indexing or displaying thumbnails.
- Provide metadata (descriptive, technical, provenance, rights, and administrative) that is only relevant to an individual Bitstream.
- Declare that a Bitstream is an *alternate representation* of another Bitstream; the two are semantically equivalent (e.g. PostScript and PDF versions of the same document).

The first two kinds of relationships are not symmetric: the *"derived"_and _"is metadata about"* relationships have clear master/slave roles. The *alternate* relationship can be considered symmetric. Note: If one of the "alternate" formats was created by a preservation operation, that fact should be recorded in provenance metadata, it does not need to be in the object model as master/slave roles.

One Bitstream may be the target of multiple relationships – *e.g.* an image might have a thumbnail, "full text", and metadata associated with it. I do not forsee a need for one "subsidiary" Bitstream to express a relationship to more than one master, however.

Bundles take no part in expressing Bitstream relationships. Although a Bundle indicates the *purpose* of its Bitstreams like thumbnails, derived text, and metadata, that fact is orthogonal to the mapping of

a subsidiary Bitstream to its master. For example, an index builder looking for derived text files would enumerate the Bitstreams in the CONTENT (or ORIGINAL) Bundle, checking each one for a *derived* Bitstream in the TEXT Bundle.

Proposed Solution

Add the following API calls to the Bitstream object:

```
/** Constants describing the type of relationship: */

  // This Bitstream was derived from the contents of the related one.
  public static final int REL_TYPE_DERIVED     = 1;

  // This Bitstream is an alternate version of the related one.
  public static final int REL_TYPE_ALTERNATE   = 2;

  // This Bitstream contains descriptive metadata about the related one.
  public static final int REL_TYPE_MD_DESC     = 3;

  // This Bitstream contains administrative metadata about the related one.
  public static final int REL_TYPE_MD_ADM      = 4;

  // This Bitstream contains technical metadata about the related one.
  public static final int REL_TYPE_MD_TECH     = 5;

  // This Bitstream contains provenance metadata about the related one.
  public static final int REL_TYPE_MD_PROV     = 6;

  // This Bitstream contains rights metadata about the related one.
  public static final int REL_TYPE_MD_RIGHTS   = 7;

  /**
   * Establish a relationship with the target Bitstream.  The nature
   * of that relationship is described by setRelationshipType().
   * When relevant, the object executing this method is the "slave"
   * (e.g. the derived one) in the relationship.
   * @param related Bitstream object which is target (master) of relationship.
   */
  public void setRelatedBitstream(Bitstream related)

  /**
   * Get related Bitstream.
   * @return related (master) Bitstream, or null if none has been set.
   */
  public Bitstream getRelatedBitstream()

  /**
   * Set the type of relationship this Bitstream has with the
   * one set in setRelatedBitstream().  See the REL_TYPE_*
   * constants for details.
   * @param rel indicates type of relationship.
   */
  public void setRelationshipType(int rel)

  /**
   * @returns type of relationship, or 0 if none has been set.
   */
  public int getRelationshipType()

  /**
   * Get all the Bitstreams with relationships to this one.
   * @return Array related Bitstreams, or empty Array if there are none.
   */
  public Bitstream[] getBitstreamsRelatedToSelf()

  /**
   * Get all the Bitstreams with relationships to this one, and
   * a Bundle name matching the indicated one.
   * @param bundleName bundle name to match.
   * @return Array related Bitstreams, or empty Array if there are none.
   */
  public Bitstream[] getBitstreamsRelatedToSelf(String bundleName)
```

For example, if you have a "content" Bitstream and want to find out if it has a thumbnail image, you would call getBitstreamsRelatedToSelf("THUMBNAIL"), and check if the array returned has any elements.

Looking for LOM metadata is a more involved application: First you'd have to get any related Bitstreams in the METADATA bundle, and then search through those for one of the appropriate type (perhaps indicated by its BitstreamFormat).

Establishing a relationship is easy. In code that creates a Bitstream by deriving it from another Bitstream, like a media filter plugin, it's obvious: set the new Bitstream to be related from the one from
which it was derived, and setRelationshipType(REL_TYPE_DERIVED).

When ingesting a package (SIP) that includes an encoding of relationships between Bitstreams (e.g. METS), the ingester code should also make the appropriate calls to setRelatedBitstream()

and setRelationshipType() to mark metadata, derived, and alternate Bitstreams.

The implementation is straightforward. It can be done by adding two columns to the Bitstream table; one containing a bitstream_id key (or NULL) to the related Bitstream, and another with an integer for
the relationship type.

Comments

Can you think of any scenarios where this proposal is too limited?

Do you have any other ideas for using Bitstream relationships?

Jim+Downing, 2006-05-12

I think we need N-N relationships. I'm imagining a situation where you have an Item containing a single logical image with several equivalents in different formats and a single thumbnail. In a way, the thumbnail is derived from all of the equivalents. Similarly a text index record from several migrations of a word document.

This would also allow two bitstreams to have a relationship with multiple types - what if a single metadata bitstream contains descriptive, technical and provenance metadata for another?

Definitely keen on the way the API makes it the directionality of the relationships clear, and restricts responsibility for updating them to one end.

Can we replace the ints with a typesafe enum, please? Too many magic numbers already!

A not-particularly-small extension: how about extending this so that bitstreams can be related to the Item, so that a bitstream can assert "I am the metadata record for this item"?

Scott Phillips, 2006-05-12

I think we need set relationships analogous to the METS fileGrp or div elements. Right now DSpace has the concept of bundles, but those bundles are used to separate derived bitstreams from original/content bitstreams. This means the user has no way of selecting what bundle to place a bitstream in, nor can they create bundles. Everything the user adds to an item is placed in the original/content bundle.

If container relationships are added at the bitstream level there are several possible uses. The first use that comes to mind is relating scanned pages together into chapters, sections, or other groupings. Another possibility is relating together different types of pictures for one physical artifact. There are various use cases, and most depend on the type of artifact being stored.

This may be a bigger problem that what you wanted to tackle, however, I think that if we are going to change the DSpace information model we should not do it incrementally. The information model is fundamental to DSpace and if we ever want to support these types of rich relationships, then I advocate that we build in the basic support for them now. If we take the incremental approach, I believe that it will lead to fragmentation and poor design decisions.

Therefore, I propose in addition to Larry idea:

- Follow Jim's suggestion about N-N relationships
- Make these relationships interact with bundles; this would allow administrative and technical metadata to be applied to a bundle of bitstreams.
- Make bundles recursive, so that bundles can contain other bundles.
- Enable the user to create and modify bundles.

MacKenzie Smith, 2006-05-15

I had the same thought as Scott, that this proposal begins to put us on the slippery slope of dealing with arbitrary complex digital objects (e.g. scanned documents or multimedia publications or different views of a built artifact). DSpace has always studiously avoided handling these directly because there are an infinite number of such relationships and making sense of them in the submission, curation, or end-user interface is a huge job... take, for example, the Fedora solution of a "relationship ontology" (see http://www.fedora.info/download/2.1.1/userdocs/digitalobjects/introRelsExt.html) and the complexities that leads to in the applications that must make sense of them. Sometimes it's justified, but it comes at a very high price. The DSpace philosophy so far has been to put that complexity into the object encoding schemas like METS, MPEG21, XFDU, IMS-CP, and the like, and let the application and UI layers deal with those, rather than relying on DSpace's native bundle/bitstream data model.

So the tradeoff we should consider here is between simple (if incomplete and somewhat arbitrary) data modelling, as Larry is suggesting, and coming up with the Ultimate Solution, with all the implications that has for the other parts of the system. Do we really want to go there? If so, why bother with the bundle/bitstream model at all... why not go straight to RDF ontologies for complex objects and RDF-aware applications to process them?

I'm not opposed to figuring out how to handle very complex digital objects in DSpace... there's certainly demand to be able to support them. But *most* DSpace sites have relatively simple digital objects, so redesigning the system to deal with arbitrary complexity of digital objects natively would be a high price to pay for a minority of uses, in my opinion.

Graham Triggs, 2006-10-12

Well, the first thing that strikes me is to question the wisdom separating out the setting of a relationship, and it's type. I suppose you could argue the need to update a relationship later on, but in all cases setting a relationship is establishing it (even an 'update' would be removing the old relationship and establishing a new one), and should therefore be accompanied by the type.

But aside from that, if we want to track relationships for objects outside of their 'natural' hierarchy (which is what appears to be the case here), then it shouldn't directly be a property of the objects themselves. Those relationship(s) should be managed separately, and whilst objects can know which relationship(s) they are a part of, they shouldn't be directly aware of what and how they are related to. Otherwise, this would go down the road of having quite complicated structures within these objects to track the relationships, and lead to problems with traversal of the relationships.

Oh, and another minor point - if this is a bidirectional link, then a primary function of the 'set' methods would have to be to alter the state of the object passed in. By convention, set methods are mutators for the object on which they are called, and altering any object passed as a parameter should not be a primary outcome. It would be better to use a different naming - such as establishBitstreamRelationship().