HistorySystemPrototype

Contents

History System Protoype for DSpace 1.5

This page supercedes the HistorySystem proposal.

After examining the state of the current

```
HistoryManager
```

and the data it produces, I decided to discard
it and write a complete replacement, based on the EventSystemPrototype.
Since the PLEDGE project needs
a functional history system, we are motivated to rebuild it
in time for the next major release (1.5).

This page describes the prototype that was implemented in late 2006
and is to be submitted as a patch for DSpace 1.5.

Goals

- Preserve a fixed, unchangeable record of all significant changes to *archival* objects in the DSpace data model.
  - *Archival* means the objects which are to be preserved, such as Items, Bitstreams, and perhaps Communities and Collections. EPersons, for example, are not archival.
  - Record all information about events that is relevant to the provenance of an object, e.g. modifications, format migration, etc.
- Make history data accessible through an API:
  - Fetch history related to a specific object (with optional recursion to get history of its logical children).
  - Allow free-form queries on all history records.
- Carry the relevant history data with an Item when it is exported or ingested, e.g. add a History section to the AIP.
  - This implies the identifiers naming objects and the archive itself in the History records must be globally unique so they will be meaningful away from the archive that created them.
- Migrate whatever data can be salvaged from the old DSpace history system.

Non-Goals

- This is not a versioning system, so it does not usually attempt to record the substance of a change (e.g. contents of bitstreams).
- Do not record any events that do not result in changes to the archive, so the history will not have records of disseminations.
- Exclude any events and objects that we do not believe to have much archival value, e.g. authorization policies, EPersons, Groups.
- Only record history data for objects that are *in the archive*, which means history is not recorded for Workspace and Workflow objects.

Justification

If the purpose of saving history data is to establish the provenance of objects archived in DSpace as an aid to future preservation efforts, it makes sense to only save data about the objects which are to be preserved. This excludes transient objects (e.g.

```
WorkspaceItem
```

,

```
WorkflowItem
```

,
and objects that are only meaningful in the context of their home DSpace archive:

```
EPerson
```

,

```
Group
```

.

Since preservation often includes copying and/or transferring custody of Items to another
DSpace repository or even another type of archive, the history data has to be meaningful outside of the context of DSpace.

An RDF framework for History Records

The following sections describe a "framework" (since is not formal enough to be a schema) for the content of history records.

Namespaces

Namespaces in the History schema

prefix	Namespace URI	Description
`rdf:`	`http<b></b>://www.w3.org/1999/02/22-rdf-syntax-ns#`	RDF
`rdfs:`	`http<b></b>://www.w3.org/2000/01/rdf-schema#`	RDF Schema
`abc:`	`http<b></b>://metadata.net/harmony#`	ABC Harmony
`dc:`	`http<b></b>://purl.org/dc/elements/1.1/`	Dublin Core (unqualified)
`history:`	`http<b></b>://www.dspace.org/history#`	DSpace History
`dso:`	`http<b></b>://www.dspace.org/objectModel#`	DSpace Object Model

URIs of DSpace Objects

To write RDF statements about DSpace Objects, we need URIs for them.
These URIs have to meet the following requirements:

- Every *archival* DSpace Object is mapped to *one and only one* URI. This means, if we have the URI, we can look up the object, and vice versa.
- URIs are unique amongst all of the DSpace Archives in the world, so an object may be copied or transferred into a different archive and remain unique.
- The URI convention is not married to the Handle System, so that it can be adapted to other types of persistent identifiers.

We can assume all archival objects have persistent identifiers. Items, Collections, Communities, and Sites all have Handles. Every Bitstream within an Item has a persistent ID consisting of its parent Item's handle followed by a colon (*

```
:
```

*) separator, and the Bitstream's
unique-within-the-item Sequence ID number. The general pattern for making a URI out of a Handle is:

```
info:dspace/handle#
```

handle

```
:
```

subfragment

In the "info" URI, *dspace* establishes the class of the URI, followed by a separator and *handle* to indicate that the fragment contains a persistent identifier based on a Handle.

The trailing colon and *subfragment* are optional. For example, a URI of the Handle

```
1721.1/18582
```

would be

```
info:dspace/handle#1721.1/18582
```

The URI for the third Bitstream within that Item would be:

```
info:dspace/handle#1721.1/18582:3
```

See the Object URI discussion for
a more thorough discussion of this URI pattern and the reasoning behind it.

DSpace EPerson objects are the exception: they are identified by the
email address associated with the EPerson, in a

```
mailto:
```

URL.
This is the most unique identifying feature of an EPerson, and although it
is not archival it is at least globally unique.

DSpace History RDF "schema"

These are the classes and properties used to describe History events.

Action classes

The following are subclasses of

```
abc:Action
```

. Each class
corresponds directly with one of the *event types* from
the event prototype.

- `history:Action`

  - superclass for the other actions.
- `history:Add`

  - add a new member to the subject object.
- `history:Remove`

  - remove a member from the subject object.
- `history:Create`

  - create a new subject.
- `history:Delete`

  - destroy the subject.
- `history:Modify`

  - modify content of the subject.

- `history:ModifyMetadata`

  - modify metadata describing the subject.

DSpace Object classes

An object in the data model is typed by one of the following
classes, with names matching the corresponding DSpace constants.

These are all subclasses of

`abc:Manifestation`

- `dso:DSpaceObject`

  - superclass for all other DSO classes.
- `dso:Community`
- `dso:Collection`
- `dso:Item`
- `dso:Bundle`
- `dso:Bitstream`
- `dso:Site`

This is a subclass of

`abc:Agent`

- `dso:EPerson`

Properties of an Action

These properties have the domain

`abc:Action`

.

- `abc:creates`

  - range is a

  `dsh:DSpaceObject`

  , the "subject" of the event.

- `abc:destroys`

- range is a

`dsh:DSpaceObject`

, the "subject" of the event.

- `abc:hasPatient`

- range is a

`dsh:DSpaceObject`

, the "subject" of the event.

- `abc:atTime`

- range is a literal *ISO 8601 timestamp*

- `history:inArchive`

- range is a

`dso:Site`

- `abc:involves`

- range is a

`dsh:DSpaceObject`

, the "object" of the event.

- `abc:hasParticipant`

range is

`dso:EPerson`

- `history:usesTool`

range is literal, *ExtraLogInfo* from the event.

- `history:detail`

range is literal,

`"event.getDetail()"`

(if available).

- `history:transactionID`

range is literal,

```
"event.getTransactionID()"
```

(if available).

Properties of a DSpace Object

The following properties have the domain

```
dsh:DSpaceObject
```

:

- ```
  dc:title
  ```

  - range is a literal, the object's title or proper name.
- ```
  dc:type
  ```

  - range is a literal, the object's type or purpose. This only gets used on Bitstreams.

History Implementation

The prototype does essentially three things:

1. Record history of all relevant data model changes.
2. Fetch history statements covering the history of a given object.
3. Fetch history records in answer to a free-form query.

Transformation of Event into History Statements

When the History event consumer sees an event, it might apply a transformation before translating it to RDF:

1. If the subject is a
   ```
   Bundle
   ```
   (which has no Handle) *and* the event-type is either
   ```
   Add
   ```
   or
   ```
   Remove
   ```
   , replace the subject with the
   ```
   Item
   ```
   that owns that bundle, so that Bitstreams appear to get added directly to Items.
2. Otherwise, if the subject has no Handle (or other persistent identifier), skip this event.
3. If the *object* of an event does not have a Handle (persistent ID), ignore the event. This excludes events such as adding Bundles to an Item.

The event itself is represented by a unique, randomly named URI which is an

```
abc:Action
```

. It is assigned the following properties:

- `rdf:type`

  of

  `abc:Action`

- `rdf:type`

  of

  `history:Create`

  or whatever the *event type* is.
- *One of either* (

  `abc:creates | abc:destroys | abc:hasPatient`

  ), naming a

  `dsh:DSpaceObject`

  , the "subject" of the event.
- `abc:involves`

  naming a

  `dsh:DSpaceObject`

  which the "object" of the event, if there is one.
- `abc:atTime`

  with the timestamp from the event.
- `history:inArchive`

  naming the

  `dso:Site`

  (archive) where the event occurred.
- `abc:hasParticipant`

  naming the

  `dso:EPerson`

  responsible, if that is available.
- `history:usesTool`

  containing *ExtraLogInfo* from the event, if any.

- `history:detail`

  containing

  `"event.getDetail()"`

  , if any.
- `history:transactionID`

  with the

  `"event.getTransactionID()"`

  , if any.

Each

`dso:DSpaceObject`

mentioned in the statements above has
the properties:

- `rdf:type`

  of

  `abc:Manifestation`

  (or

  `abc:Agent`

  for an EPerson).
- `rdf:type`

  of

  `dso:Item`

  or whatever the type of the object is.
- `dc:title`

  with the name or title of the object, if available.
- `dc:type`

  with the name of the owning Bundle when the object is a a Bitstream. This can be helpful to preservationists since it indicates the purpose of the bitstream.

Lacunae: Events that Cannot Be Recorded

Due to the inherent conflict between the low-level style
architecture of the Event System,
and the requirement that History records identify all data model
objects by their *persistent* identifiers, some events simply
cannot be translated from the data in the event stream into History
records. The event stream identifies data model objects by "ephemeral" database keys (for speed, and since not all objects *have* persistent
identifiers like Handles), so the event consumer has to look up extras like the persistent identifier, and
any attributes of the Subject and Object of the event.
However, if any of those objects gets deleted in the transaction that generates that event, it is too late to look up
the persistent identifier (which is why it is packaged in the "detail"
field of some events). Here are the specific situations in which
an event cannot be recorded in the History:

- Delete events on Bitstream objects are lost if the Item containing the Bitstream is deleted in the same transaction, since the History consumer needs access to the owning Item and Bundle to construct a persistent identifier for a Bitstream.
- All events on Bundles are lost because Bundles do not have persistent identifiers. Add and Remove events on Bundles are treated as Add and Remove events on their owning Items, so in the History model, Bitstreams appear to belong directly to Items.
- Some Remove events on Communities, Collections, and Items are lost, when the Remove is part of a transaction that removes an entire hierarchy of objects. When the subject of a Remove is itself deleted in the same transaction, the Remove event cannot be recorded since there is no persistent identifier for the subject (although there *is* one available for the object of the Remove, in the detail field).

Although this leaves holes (*lacunae*) in the history record of an archive,
there is still enough information recorded to tell a preservationist
the fate of any objects missing from the archive. The Delete
events are recorded accurately for all archival objects. Since
all Bitstreams of archival significance are owned by Items (and
the Add events showing that are traceable in the History record),
their fate can be inferred from a Delete record for their Item. Some
Remove events are lost, but they can also be inferred from a
Delete event. The record is a bit messy and incomplete, but it
is still quite usable.

Retrieving History of an Object

NOTE: The new

```
RDFRepository
```

class, the superclass
of

```
HistoryRepository
```

, can fetch RDF statements related
to a "key" URI. The history system uses this feature to retrieve
all the statements about a given DSpace object in one simple operation, so RDF queries are not needed. See the javadoc
of

```
HistoryRepository
```

for more details.

To collect all the history records related to a DSpace Object, start
by creating the history system's URI for that object, e.g.

```
info:dspace/handle#1721.1/18582
```

If all of the history RDF triples are stored in a common repository,
then construct a query to fetch all of the resources that have
a property of either
(

```
abc:creates | abc:destroys | abc:hasPatient
```

) whose object is
target URI above. The *subject* of each these triples is an

```
abc:Action
```

in the history of the object, so collect all
triples of which it is the subject.

If you are collecting the history of a DSpace Item, you may wish to
collect the history of its Bitstreams as well. These resources will
be the objects of

```
abc:involves
```

properties of the actions.
Since the Item is the archival unit, but the Bitstreams have most of
the material of interest to preservationists,
you'll probably want to get the history
of all of an Item's Bitstreams as part of the history of the Item.

Finally, for each DSpace Object resource involved in the history,
collect all the statements of which it is the subject. This will
give you type and descriptive metadata about each object.
These resources are the RDF objects of properties

```
history:inArchive
```

, and

```
abc:hasParticipant
```

.

Examples

1. Item History report in N3
2. Item with life cycle ending in delete, in N3
3. Dead Link: Same Item with life cycle ending in delete, in RDF/XML

Installing and Operating

To install the prototype implementation, download the source
and follow instructions to install it:

Download and Install

Preparation

1. Start with DSpace 1.5 source checkout (ca. January 5, 2006)
2. Apply the EventSystemPrototype patch as directed on that page.
3. Apply the AipPrototype as directed on the page.

Downloads

1. Changes to dspace.cfg file
2. JAR files to add
3. Java Source files to add

Installation

Working in your DSpace installation directory:

1. Shut down your servlet container.
2. Apply the source change to the DSpace configuration with *patch*:

   ```
   patch -l config/dspace.cfg  < history-dspace.cfg.diff
   ```

   (or, manually apply the changes to your configuration file.)

3. Make sure the changes are propagated to the configuration file in your run-time directory.
4. Unpack the
   ```
   history-new-libs.zip
   ```
   file with
   ```
   unzip
   ```
   .
5. Unpack the
   ```
   history-new-source.zip
   ```
   file with
   ```
   unzip
   ```
   .
6. Rebuild all sources with
   ```
   ant clean install_code build_wars
   ```

Configuration

The History system requires the following configuration keys:

- Ignore History metadata in non-AIP METS packages:
  ```
  mets.default.ingest.crosswalk.DSpaceHistory = NULLSTREAM
  ```
- Streaming dissemination crosswalk, to be added to the plugins configured for
  ```
  StreamDisseminationCrosswalk
  ```
  :
  ```
  org.dspace.history.HistoryStreamDisseminationCrosswalk = HISTORY
  ```
- Streaming ingestion crosswalk, to be added to the plugins configured for
  ```
  StreamIngestionCrosswalk
  ```
  :
  ```
  org.dspace.history.HistoryStreamIngestionCrosswalk = HISTORY
  ```
- Add an event consumer named "history":
  ```
  event.consumer.browse.class = org.dspace.browse.BrowseConsumer<br>event.consumer.browse.filters =
  Item+Create|Modify|Modify_Metadata:Collection+Add|Remove
  ```
- Add the history consumer to the default dispatcher:
  ```
  event.dispatcher.default.consumers = history:sync ...
  ```
- To disseminate history records in AIPs, add:
  ```
  aip.disseminate.digiprovMD = DSpaceHistory:HISTORY
  ```
- To ingest history from AIPs, add:

```
mets.dspaceAIP.ingest.crosswalk.DSpaceHistory = HISTORY
```

Operation

Before starting a DSpace application or the servlet container for the first time, you may wish to move or clean out the contents of the

```
history
```

subdirectory in your run-time directory.
The new History System does not use any of the old data files. It will create some new files in the directory indicated by the configuration key

```
history.dir
```

so be sure it exists.

IMPORTANT: The files in the

```
history.dir
```

belong to an OpenRDF RDF "Native" repository. Do NOT modify or edit them, at risk of losing the history repository.

Start the Web UI or execute a command-line application and do something that changes the contents of the archive – ingest a new object, modify an Item or metadata, etc. Then, review the history to make sure that change was recorded, with the command:

```
/dspace/bin/dsrun org.dspace.history.HistoryRepository -x -f n3
```

This exports *all* History records, so you won't want to do this except at first when there are very few of them.

To view the history records related to one object, export the History related to its Handle with the command:

```
/dspace/bin/dsrun org.dspace.history.HistoryRepository -f n3 -d handle
```

You can export the RDF in RDF/XML format by specifying "xml" after the "-f" switch instead of "n3". Use the "-h" switch to see other options.

Future Work

Since this is a prototype, there are some things left undone:

1. Backup strategy. _NOTE: This has been solved, see -D and -R options of HistoryRepository command-line application._The History RDF data is stored in a "native" triple-store, which is an OpenRDF application-defined format. If it were ever corrupted, some or all of the history data would be lost. But don't worry about that just because this is based on an "alpha" release of OpenRDF 2.0...
    a. It is *not* really good enough to save just the RDF triples (as N3 or RDF/XML); OpenRDF actually records them as "quads", adding an extra resource called the "context". DSpace History uses that context to bind each triple to the URI of a DSpace Object, which makes it very efficient to retrieve all the History records about a particular object.
    b. You *could* just export the RDF in the triplestore with the "-x" option; it should be possible to sort out the mapping of records to objects again without the "context", it would be a lot of extra work and there is no code to do it yet. It's much better to simply save the state of the quads.
    c. See the -Q option of the
    ```
    org.dspace.history.HistoryRepository
    ```
    ; with a little tuning (notably dealing with data types and literals) this export could be used to restore the triplestore, although you'd have to write an ingester too.
2. Experiment with making the History consumer asynchronous.
3. Export History data to a SIMILE timeline

See Also

- EventSystemPrototype
- AipPrototype