AipPrototype

This page & all code described on it is now OBSOLETE. It has been replaced by the AIP Backup and Restore feature, which will first be released in DSpace 1.7.0

PLEDGE AIP Prototype

This page & all code described on it is now OBSOLETE. It has been replaced by the AipBackupRestore feature, which will first be released in DSpace 1.7.0

This page describes a prototype AIP implementation planned as part of the PLEDGE project. Since the PLEDGE project only needs AIPs to replicate them under the direction of a policy engine, it was not necessary to create an AIP-based asset store.

About This Implementation

The source changes and additions that implement AIPs have the following other benefits:

- PackageIngester class modified to ingest all kinds of archival objects - *i.e.* Communities and Collections – as well as Items. All package ingesters can be extended to take advantage of this change.
- METS packager framework extended and refactored to be more flexible and configurable. New metadata segments can be added to METS packages by configuration.
- Stream-oriented crosswalks added to support bulky and non-XML metadata.
- The addition of "internal AIPs" gives DSpace archives a badly-needed solution to data security: by storing internal AIPs in the asset store, the set of asset files contains all the information needed to restore a functional archive after catastrophic loss of the RDBMS. The archive *can* be salvaged if the RDBMS gets out of sync or permanently destroyed, which is not now the case.

Goals of AIP Prototype

- AIP is a package describing one archival object.
  - Archival object may be **Item**, **Collection**, or **Community**. Bitstreams are included in an Item's AIP.
  - Each AIP is logically self-contained, can be restored without rest of the archive.
  - AIP profile trades off favoring completeness and accuracy rather than presenting the semantics of an object in a standard format. It conforms to the quirks of DSpace's internal object model rather than attempting to produce a universally understandable representation of the object.
  - An AIP *can* serve as a DIP, especially when transferring custody of objects to another DSpace implementation, but it is not intended to be a general-purpose DIP (the DSpace METS SIP profile is better for that).
- The implementation is layered on top of DSpace 1.4 plus the EventSystemPrototype with minimal other changes to the source.
- Restoration of an archive from AIPs is not perfectly complete; it is intended to recover from catastropic loss of content and metadata, *not* restore the exact same archive as before. Some information (e.g. access controls) would be lost.
- This prototype does *NOT* attempt to redefine the asset store in terms of AIPs, as in the AssetStore proposals.

Makeup and Definition of AIPs

- There are two types of AIP:
  1. **Internal:** AIP is embedded in the DSpace archive's asset store, so the AIP "package" is just a METS manifest, which has references to other assets instead of wasteful copies of them.

2. **External:** AIP is an archive file (Zip) containing manifest and all content bitstreams.

- In contrast to SIP or DIP, the AIP should include all available DSpace structural and administrative metadata, and provenance such as history records.
- The archive only ever stores *one* version of an internal AIP, the latest (although it does not necessarily reflect the latest DB state).
- Archival Units are:
  - Item (including references to Bitstreams and Bundles)
  - Collection (including persistent IDs of members)
  - Community (including persistent IDs of members)
- Bitstreams and Bundles are second-class archival objects; they are recorded in the context of an Item.
- BitstreamFormats are not even second-class; they are described implicitly within Item technical metadata, and reconstructed from that during restoration.

Issues and Questions

- Although this prototype does not implement AIP-centric storage, it can be leveraged toward that goal.
- This design does not consider any of the proposals for versioning DSpace objects.
- See the ObjectUri proposal that establishes a pattern for generating URIs for pieces of the DSpace object model. This AIP implementation relies on it to name Bitstream asset-store files for internal AIPs.
- The impact on performance was a consideration:
  - Creating or rewriting an AIP manifest is too expensive to do synchronously as part of every data model change.
  - The solution is to let AIP updates lag behind real-time changes by some amount. Possible implementations include:
    1. Use asynchronous consumer (see *Event System Prototype*) to update AIPs at intervals and/or in a separate JVM.
    2. Periodically run the `AIPManager` command-line application (e.g. from cron) to update all "stale" AIPs.
- How best to record descriptive metadata? crosswalk to a standard like MODS, best for archival and external consumption? Save DIM directly for maximum completeness and accuracy of restoration in another DSpace? Include both?
- Some archival objects have elements that will probably be left out of the AIP prototype, e.g. template Items for Collections.
- Internal AIPs are susceptible to a race condition: Whenever any "member" elements of an object (e.g. the Bitstreams of an Item, or the Items of a Collection) are permanently deleted from the archive, the AIP for the parent object will contain unresolvable references until it is updated. If the internal AIP – which, remember, depends on the asset store for its components – is ingested at this point, the ingest will fail until the AIP is corrected.

AIP Details: METS Usage

- `mets` element
  - `@PROFILE` fixed value="http://www.dspace.org/schema/aip/1.0/mets.xsd" (this is how we identify an AIP manifest)
  - `@OBJID` URN-format persistent identifier (Handle) if available, or else a unique identifier.
  - `@LABEL` title if available
  - `@TYPE` DSpace object type, one of "DSpace ITEM", "DSpace COLLECTION", "DSpace COMMUNITY".
  - `@ID` is a globally unique identifier, such as `dspace670750919768620147179712097177493943 63`.
    - @IDs should be used wherever available, I'll put a note about forming IDs in the profile spec.
      [OK, but how unique does the ID have to be, just within the document or amongst all other AIP documentes? --lcs]
      I can't imagine a scenario where we would reference an ID within a METS document from without that document, except for perhaps the `mets@ID`. I would say the `mets@ID` should be unique amongst all AIP documents, but the other IDs should just be unique within the document.
- `mets/metsHdr` element
  - `@CREATEDATE` timestamp that AIP was created.
  - `@LASTMODDATE` last-modified date on Item, or nothing for other objects.
    - mets defines these attributes as describing the METS document itself, we use them to describe the AIP, which sometimes we think of as the METS document, but more often think of as the 'package' – i.e. the METS document and all the files. I don't have a problem with the use Larry put forth, but we need to mention it in a prolife. I wonder if these dates shouldn't rather be in a techMD section, or maybe both.
  - `agent` element:
    - `@ROLE` = "CUSTODIAN",
    - `@TYPE` = "OTHER",
    - `@OTHERTYPE` = "DSpace Archive",
    - `name` = *Site handle*.
- `mets/dmdSec` element
  - object's descriptive metadata crosswalked to MODS (or whatever the METS default is)
    - See link to RW's Comments Page below for notes on use of MODS
  - object's descriptive metadata in DSpace native DIM intermediate format, to serve as a complete and precise record for restoration or ingestion into another DSpace.
    - We should require mets/dmdSec@OTHERMDTYPE if @MDTYPE = "OTHER"
  - When the `mdWrap` `@TYPE` value is `OTHER`, the element *MUST* include a value for the `@OTHERTYPE` attribute which names the crosswalk that produced (or interprets) that metadata, e.g. `AIP-TECHMD`.
- `mets/amdSec` element - admin (technical, source, rights, and provenance) metadata for the entire archival object.
  - `rightsMD` elements of the following TYPEs:
    - `DSpaceDepositLicense` if the object has a deposit license, it is contained here.
    - `CreativeCommonsRDF` If the object is an Item with a Creative Commons license expressed in RDF, it is included here.
    - `CreativeCommonsText` If the object is an Item with a Creative Commons license in plain text, it is included here.
  - `sourceMD` elements - recorded twice, once in DSpace native format, once in PREMIS:
    NOTE: PREMIS is only implemented for Bitstreams at the moment, and for the forseeable future.
    - DSpace native format: MDTYPE="OTHER" OTHERMDTYPE="`AIP-TECHMD`" (see *Crosswalks* section below for details")
    - PREMIS expression of this technical metadata for archival object. (To be done later.)
      - RW:Comment [AIP Object (Item-Collection-Community)-specific Metadata in PREMIS] To see an example of the PREMIS version of this metadata, SEE link to RW Comments section page below
  - `digiprovMD`

        - When History data is available, includes a section of `TYPE="DSpaceHistory"` containing an RDF/XML rendition of the history data for the object. For internal AIPs, the history is stored in an external bitstream in the asset store; for self-contained packages it is a file in the package.
- `mets/amdSec` elements - technical metadata for each of an Items's Bitstreams, both in PREMIS and DIM formats
    - `techMD` element - PREMIS technical metadata, expanded from SIP, for each of an Item's Bitstreams.
    - `sourceMD` element, type is AIP-TECHMD.
        - Bitstream-specific metadata not all of which is explicitly encoded in PREMIS, i.e.
        - `name` (`dc.title`)
        - `description` (`dc.descripton`)
        - `userFormatDescription` (`dc.format`)
        - BitstreamFormat, including short name, MIME type, extension. (`dc.format.medium`)
            - RW:Comment – Bitstream Technical Metadata
              ***** Why are we recording the file format support status? That's a DSpace property, rather than an Item property. Do DSpace instances rely on objects to tell them their support status?
                - Format support and other properties of the BitstreamFormat are recorded here in case the Item is restored in an empty DSpace that doesn't have that format yet, and the relevant bits of the format entry have to be reconstructed from the AIP. --lcs
            - To see an example of the changes to the PREMIS version of this metadata, SEE link to RW Comments section page below
- `mets/fileSec` element
    - For archival objects of type ITEM:
        - Each distinct Bundle in an Item goes into a `fileGrp`.
            - Did the "ORIGINAL" bundle get renamed "CONTENT"?
              [Not in DSpace 1.4_ atUSE is set to the exact Bundle name in an AIP. --lcs]
        - Bitstreams in bundles become `file` elements under `fileGrp`.
        - `file/@SEQ` contains the Bitstream sequence ID
        - `file@CREATED` and `file@SIZE`
            - The DSpace SIP calls for the use of `@CREATED` for the file element, AIP examples do not use `@CREATED`, but do use `@SIZE`, which is not recommended by SIP.
              [Since Bitstreams don't have any dates (neither created nor last-modified) the at CREATED cannot be set on dissemination. --lcs]
    - `mets/fileSec/fileGrp/file` element
        - Set @SIZE to length of the bitstream. There is a redundant value in the techMD but it is more accessible here.
        - Set @MIMETYPE, @CHECKSUM, @CHECKSUMTYPE to corresponding bitstream values. There is redundant info in the techMD.
        - SET @SEQ to bitstream's SequenceID if it has one.
    - For archival objects of types COLLECTION and COMMUNITY:
        - *Only* if the object has a *logo bitstream*, there is a `fileSec` with one `fileGrp` child of `@TYPE="LOGO"`.
        - The `fileGrp` contains one `file` element, representing the logo Bitstream. It has the same file format, checksum, etc fields as the Item content bitstreams, but does not include metadata section references or a SequenceID.
        - See the main `structMap` for the reference to this file.
- `mets/structMap` - Primary structure map, `@LABEL="DSpace Object", @TYPE="LOGICAL"`
    - For COLLECTION objects: Top-level `div` has one child:
        1. `div` with `@TYPE="MEMBERS"`. For every Item in the Collection, it contains a `div` with an `mptr` linking to the Handle of that Item. Its `@LOCTYPE="HANDLE"`, and `@xlink:href` value is the raw Handle.

    - If Collection has a Logo bitstream, there is an `fptr` reference to it in the very first `div`.
    - For COMMUNITY objects: Top-level `div` has two children:
        1. `div` with `@TYPE="SUBCOMMUNITIES"`. For every Sub-Community in the Community it contains a `div` with an `mptr` linking to the Handle of that Community. Its `@LOCTYPE="HANDLE"`, and `@xlink:href` value is the raw Handle.
        2. `div` with `@TYPE="COLLECTIONS"`. For every child Collection, it contains a `div` with an `mptr` linking to the Handle of that Collection. Its `@LOCTYPE="HANDLE"`, and `@xlink:href` value is the raw Handle.

    - If Community has a Logo bitstream, there is an `fptr` reference to it in the very first `div`.
    - ITEM objects have the same kind of simple structure map as SIP/DIP: top level `div` with a `div` under it for each visible Bitstream.
        - If Item has primary bitstream, put it in first `structMap/div/fptr`.
- `mets/structMap` - Structure Map to indicate object's Parent
    - Contains one `div` element which has the unique attribute value `TYPE="AIP Parent Link"` to identify it as the older of the *parent pointer*.
        - It contains a `mptr` element whose `xlink:href` attribute value is the raw Handle of the parent object, e.g. `1721.1/4321`.
          <p>In order to restore a DSpace archive from internal AIPs in the asset store, the parent of each object must be available at the surface level of the METS document so the object can be *instantiated under its correct parent* before the metadata (which may also name the parent) is crosswalked.

Rob Wolfe's Comments on METS Usage

Crosswalks

DIM Descriptive Elements for Collection objects

Metadata Field	getMetadata() key
dc.description	introductory_text
dc.description.abstract	short_description
dc.description. tableofcontents	side_bar_text
dc.identifier.uri	getHandle();
dc.provenance	provenance_description
dc.rights	copyright_text
dc.rights_license	copyright_text
dc.title	name

DIM Descriptive Elements for Community objects

Metadata Field	getMetadata() key
dc.description	introductory_text
dc.description.abstract	short_description
dc.description. tableofcontents	side_bar_text
dc.identifier.uri	getHandle();
dc.rights	copyright_text
dc.title	name

AIP Technical Metadata for Item

Metadata Field	method and comments
dc.contributor	getSubmitter().getEmail()
dc.identifier.uri	getHandle()
dc.relation.isPartOf	getOwningCollection().getHandle() *as URN*
dc.relation. isReferencedBy	getCollections() *Handle URN of each non-owner*
dc.rights.accessRights	isWithdrawn() *"WITHDRAWN" if true*

AIP Technical Metadata for Bitstream

Metadata Field	method and comments
dc.title	getName()
dc.title.alternative	getSource()
dc.description	getDescription()
dc.format	getUserFormatDescription()
dc.format.medium	getFormat().getShortDescription()
dc.format.mimetype	getFormat().getMIMEType()
dc.format. supportlevel	getFormat().getSupportLevel()
dc.format.internal	getFormat().isInternal()

AIP Technical Metadata for Collection

Metadata Field	method and comments
dc.identifier.uri	getHandle()
dc.relation.isPartOf	getCommunities()[0]
dc.relation. isReferencedBy	getCommunities()[1]

AIP Technical Metadata for Community

Metadata Field	method and comments
dc.identifier.uri	getHande()
dc.relation. isPartOf	getParentCommunity()

Example AIP manifests

These are examples of *internal* AIPs for some representative DSpace objects:

- Item Example
- New larger Item example, in multiple collections
- Collection example
- Community example
- Community with Logo bitstream

Creating Internal AIPs for Later Restoration

Start with a DSpace archive that has the AIP Prototype patched into
its code base. Prepare internal AIPs for the first time with the command:

```
dsrun org.dspace.administer.AIPManager -u -a -v -e _admin-user_
```

Be sure that command completed successfully; check standard output
and the DSpace log for errors.

If it runs too long, you may wish to use the "-c" option to limit the number of AIPs it processes and repeat the process at off-hours for several days. Since the -u option *updates* internal AIPs, it will not re-create existing AIPs unless the underlying objects have changed.

Maintaining Internal AIPs

You should periodically run the command

```
dsrun org.dspace.administer.AIPManager -u -a -e _admin-user_
```

to update internal AIPs for objects that have been changed or added.
Once a day should be enough.

Note that it will *always* re-create AIPs for Collection and Community objects, since the DSpace object model does not have a last-modified date for them and there is no way to tell if the AIP is out of date or not. Since there are relatively few collections and communities in an archive (compared to Items) this is not seen as a serious problem.

Procedure to Restore an Archive from AIPs

The following steps have been tested for a very small archive
and successfully restored the RDBMS tables from internal AIPs in the
asset store. Note that this is a coarse overview and does not
consider error-handling.

Restoration

1. Run `/dspace/bin/cleanup` to clear out unused bitstreams from the asset store.
2. Shut down your servlet container, if necessary.
3. Remove the search indices: `rm /dspace/search/*`
4. If your archive is configured to use History, save the old History by renaming its directory, and create a new, empty History directory
   e.g. `mv history history.old ; mkdir history`
5. Start with an empty database. Either:
   a. Backup the current state of the RDBMS, and destroy it with
      e.g. `drop database dspace;`
   b. Simply change your DSpace configuration to point to a different database instance, if you have room for another database.
6. Create a new, empty database:
   `createdb -U dspace -E UNICODE dspace`
7. Run the scripts in your install directory to initialize the DB:
   `ant setup_database load_registries`
8. Back in the DSpace run directory, create an admin user:
   `/dspace/bin/create-administrator`
9. Initialize the search and browse indices:
   `/dspace/bin/index-all`
10. In your DSpace configuration, ensure that the AIP restoration application will run with History turned off:
    a. Set up a separate dispatcher for the AIPManager application:
       `aipManager.dispatcher = restore`
    b. Ensure that the `restore` Dispatcher does **NOT** call the History consumer, although it should call the search and browse consumers synchronously:
       `event.dispatcher.restore.class = org.dspace.event.BasicDispatcher<br>event.dispatcher.restore.consumers = search:sync, browse:sync`
11. Rebuild the Bitstream table:
    `/dspace/bin/dsrun org.dspace.administer.RebuildBitstreamTable -r`
12. Rebuild the InternalAIP table:
    `/dspace/bin/dsrun org.dspace.administer.AIPManager -c -a -f -v -e ''admin-user''`
13. Restore archive from the internal AIPs:
    `/dspace/bin/dsrun org.dspace.administer.AIPManager -r -a -v -e ''admin-user''`

At each stage, carefully monitor the output and the DSpace log for indications of errors. You can retry the restore of an internal AIP, or even the whole set of them, if necessary; it automatically skips any objects that already exist.

Creating and Ingesting External AIPs

Since external AIPs are really just another kind of package, you can manage them with the same package manipulation tools you use with, e.g., METS-based SIPs. The only difference is that you may need to
apply some packager parameters to the AIP ingester since its default
behavior assumes it is restoring an object to the exact same place in the archive, i.e. its former parent and Handle.

You can use external AIPs to migrate objects between archives or even as a backup strategy (similar to the use of internal AIPs).

Creating AIPs

To create an AIP in a file, use this command template:

```
/dspace/bin/dsrun org.dspace.app.packager.Packager -d -t AIP -e _eperson_ -i _handle_ _file-path_
```

for example:

```
/dspace/bin/dsrun org.dspace.app.packager.Packager -d -t AIP -e florey@mit.edu -i 1721.1/4567 aip4567.zip
```

The command needs to run under the identify of an EPerson with permission to read the specified object.

To create an *internal AIP*, just add the package parameter `internal=true` to the command.
The resulting "package" will be a METS manifest document, e.g.

```
 /dspace/bin/dsrun org.dspace.app.packager.Packager -d -t AIP -e florey@mit.edu -i 1721.1/4567 -o internal=true
mets.xml
```

Ingesting External AIPs

To ingest an AIP and create a new object under a parent of your choice, add the `ignoreParent` and `ignoreHandle` package parameters to the command:

```
 /dspace/bin/dsrun org.dspace.app.packager.Packager -s -t AIP -e _eperson_ -p _parent-handle_ -o
ignoreParent=true -o ignoreHandle=true _file-path_
```

If you leave out these package-parameter
options, the AIP package ingester will
attempt to install the AIP under the parent handle it had before,
and give it back its original Handle. After all, the point of
AIPs was to reproduce the exact object that was exported. When you are effectively using the AIP as a SIP, however, you may not want it back under the same parent or handle, so there is a way to override these features.

Restoring Objects from External AIPs

If your goal is to restore the original state of an object from
its external AIP, you can do this as well, by leaving out the
"ignore" parameters *and adding the -r option* to the Packager application. "-r" tells the Packager to ingest in "replace"
mode, applying the parent and Handle from the package. Note
that you still have to specify a parent with the "-p" option but
it is not used.

Here is an example of ingesting an AIP with the "-r" option:

```
/dspace/bin/dsrun org.dspace.app.packager.Packager -s -t AIP -e _eperson_ -p 123456789/0  -r  _file-path_
```

If you are restoring an entire archive, or a hierarchy of objects,
from external AIPs, then you'll have to ingest the "ancestors" first:
for example, ingest the top-level Communities, then the sub-Communities and Collections under them, and so on, and finally
the Items when all the Collections are ready. You'll have to
examine each package to determine its parent handle, and the
handle of the object it creates, to determine the order.

Internal AIPs

Although it is possible to create internal AIPs and even ingest them
with the Packager, this is not recommended (unless you are just
satisfying curiosity or testing the system). The AIPManager application was created specifically to maintain internal AIPs within
the asset store so there is no need to export them.

Downloads and Installation

IMPORTANT: The patches from EventSystemPrototype must be applied first before attempting to install the AIP Prototype.

First, download the new files and diffs:

1. source diffs part 1
2. source diffs part 2
3. source diffs part 3
4. new source files

Then apply the changes to your DSpace installation directory:

NOTE: The interface of `org.dspace.content.packager.PackageIngester` has been changed slightly. This will break any existing package ingesters, although the ones in the DSpace core have been fixed. Look at the changes to e.g. `org.dspace.content.packager.PDFPackager` for an example of how to update your code. The changes are quite minimal.

1. Unpack the new source Zip file in your install directory with `unzip`.
2. For each of the "diff" files, *in order*, go to your install directory and apply the diff with the command:
   `patch -p 0 -l < ''diff-file''`
3. Build and install the code: `ant install_code build_wars`
4. Ensure the configuration changes in `config/dspace.cfg` get propagated to your run-time config file.
5. Ensure the new files in `config/crosswalks` are installed in your run-time directory.
6. Apply the database change by running the SQL code in the file:
   `etc/database_schema_14-15.sql`
7. Be sure to install the new WAR file(s) in your servlet container.
8. Test by updating internal AIPs as shown above

Configuration for AIPs

The following configuration keys apply to the AIP packager and management infrastructure.
They may also require certain crosswalk plugins to be configured,
but that is a separate issue that is addressed in the sample DSpace
configuration supplied with the system source.

- `aipManager.dispatcher`
  name of the Event Dispatcher for the AIPManager application; when restoring an archive from AIPs, it is best to set this to a dispatcher that calls the search and browse consumers, but not History.
- `aip.packager`
  plugin name of the Packager used to ingest and disseminate AIPs; by default it is `AIP`.
- `mets.dspaceAIP.ingest.crosswalk.''mdSecType''`
  crosswalk plugin (either XML or Stream-oriented) to be called to interpret the given mdSec type. To ignore a section, set it to `NULLSTREAM` (for stream data) or `NIL` for XML.
- `aip.disseminate.''mdSecName''`
  Sets the type name and crosswalks associated with each metadata section under the METS `amdSec: sourceMD, techMD, rightsMD, digiprovMD`. Value is comma-separated list of *mdSecType*:*pluginName* specifiers. For example:
  `aip.disseminate.techMD = PREMIS`
- `aip.disseminate.dmd`
  Sets the crosswalks and type names of descriptive metadata sections to include; value format is the same as the admin MD sections.
- `aip.ingest.createEperson`
  When value is "true", AIP ingester will create an EPerson if needed so it can set the Submitter of a newly-created Item to the "correct" value. An EPerson created this way cannot login. Default is false.

Known Deficiencies

- None of the access-control information ("policy" records) is preserved in the AIP. When an archive is restored from the asset store, all of the access controls are lost.
- EPerson records are not explicitly preserved in AIPs, although references may appear in Item AIPs (for the Item Submitter).
- EPerson Groups are not preserved as AIPs at all.
- Some bitstream format information may get lost; it is not preserved explicitly.
- Collection AIPs do not preserve the template item, or workflow configuration.

See Also

- EventSystemPrototype prerequisite to applying the AIP patches.
- HistorySystemPrototype can be applied to add history data to AIPs