DSpace System Documentation

Authors: Robert Tansley, Mick Bass, Margret Branschofsky, Grace Carpenter, Greg McClellan, David Stuve

DSpace Version: 1.2.1 (04-Feb-2005)
Documentation Version: 1.2.1-2 (04-Feb-2005)

TABLE OF CONTENTS

Introduction

DSpace is an open source software platform that enables institutions to:

- capture and describe digital works using a submission workflow module
- distribute an institution's digital works over the web through a search and retrieval system
- preserve digital works over the long term

This system documentation includes a functional overview of the system, which is a good introduction to the capabilities of the system, and should be readable by non-technical folk. Everyone should read this section first because it introduces some terminology used throughout the rest of the documentation.

For people actually running a DSpace service, there is an installation guide, and sections on configuration and the directory structure. Note that as of DSpace 1.2, the administration user interface guide is now on-line help available from within the DSpace system.

Finally, for those interested in the details of how DSpace works, and those potentially interested in modifying the code for their own purposes, there is a detailed architecture and design section.

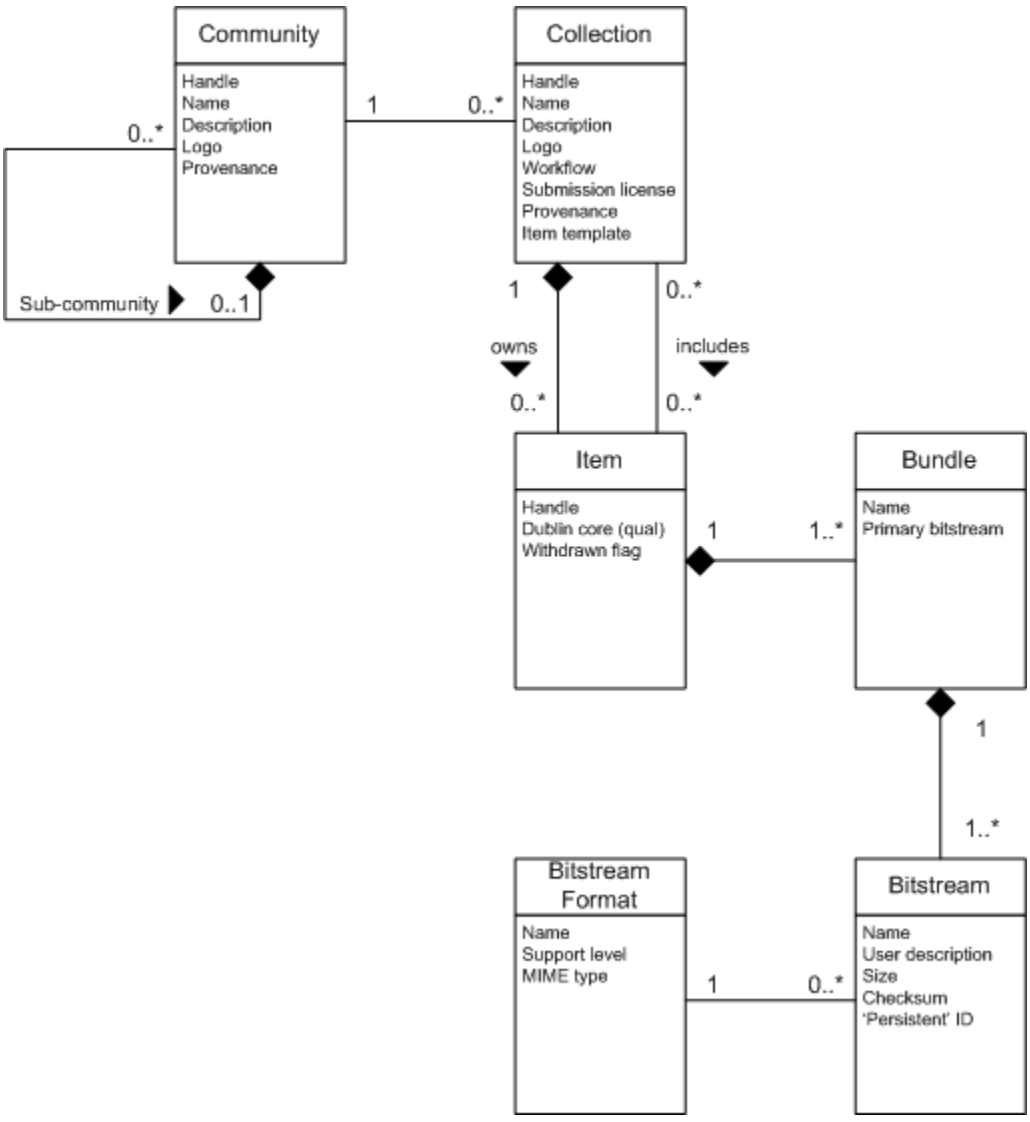
Other good sources of information are:

- The DSpace Public API Javadocs. Build these with the public_api Ant target.
- The DSpace Wiki contains stacks of useful information about the DSpace platform and the work people are doing with it. You are strongly encouraged to visit this site and add information about your own work.
- www.dspace.org has announcements and contains useful information about bringing up an instance of DSpace at your organization.
- The University of Tennessee's Jason Simms has written some additional installation notes.
- The dspace-tech e-mail list on SourceForge is the recommended place to ask questions, since a growing community of DSpace developers and users is on hand on that list to help with any questions you might have. The e-mail archive of that list is a useful resource. For example, the archive contains notes on running DSpace on Windows platform.

Functional Overview

The following sections describe the various functional aspects of the DSpace system.

Data Model



Data Model Diagram

The way data is organized in DSpace is intended to reflect the structure of the organization using the DSpace system. Each DSpace site is divided into communities; these typically correspond to a laboratory, research center or department. As of DSpace version 1.2, these communities can be organized into an hierarchy.

Communities contain collections, which are groupings of related content. A collection may appear in more than one community.

Each collection is composed of items, which are the basic archival elements of the archive. Each item is owned by one collection. Additionally, an item may appear in additional collections; however every item has one and only one owning collection.

Items are further subdivided into named bundles of bitstreams. Bitstreams are, as the name suggests, streams of bits, usually ordinary computer files. Bitstreams that are somehow closely related, for example HTML files and images that compose a single HTML document, are organised into bundles.

In practice, most items tend to have three named bundles:

- *ORIGINAL* -- the bundle with the original, deposited bitstreams
- *THUMBNAILS* -- thumbnails of any image bitstreams
- *TEXT* -- extracted full-text from bitstreams in ORIGINAL, for indexing

Each bitstream is associated with one Bitstream Format. Because preservation services may be an important aspect of the DSpace service, it is important to capture the specific formats of files that users submit. In DSpace, a bitstream format is a unique and consistent way to refer to a particular file format. An integral part of a bitstream format is an either implicit or explicit notion of how material in that format can be interpreted. For example, the interpretation for bitstreams encoded in the JPEG standard for still image compression is defined explicitly in the Standard ISO/IEC 10918-1. The interpretation of bitstreams in Microsoft Word 2000 format is defined implicitly, through reference to the Microsoft Word 2000 application. Bitstream formats can be more specific than MIME types or file suffixes. For example, `application/ms-word` and `.doc` span multiple versions of the Microsoft Word application, each of which produces bitstreams with presumably different characteristics.

Each bitstream format additionally has a support level, indicating how well the hosting institution is likely to be able to preserve content in the format in the future. There are three possible support levels that bitstream formats may be assigned by the hosting institution. The host institution should determine the exact meaning of each support level, after careful consideration of costs and requirements. MIT Libraries' interpretation is shown below:

MIT Libraries' Definitions of Bitstream Format Support Levels

Supported	The format is recognized, and the hosting institution is confident it can make bitstreams of this format useable in the future, using whatever combination of techniques (such as migration, emulation, etc.) is appropriate given the context of need.
Known	The format is recognized, and the hosting institution will promise to preserve the bitstream as-is, and allow it to be retrieved. The hosting institution will attempt to obtain enough information to enable the format to be upgraded to the 'supported' level.
Unsupported	The format is unrecognized, but the hosting institution will undertake to preserve the bitstream as-is and allow it to be retrieved.

Each item has one qualified Dublin Core metadata record. Other metadata might be stored in an item as a serialized bitstream, but we store Dublin Core for every item for interoperability and ease of discovery. The Dublin Core may be entered by end-users as they submit content, or it might be derived from other metadata as part of an ingest process.

Items can be removed from DSpace in one of two ways: They may be 'withdrawn', which means they remain in the archive but are completely hidden from view. In this case, if an end-user attempts to access the withdrawn item, they are presented with a 'tombstone,' that indicates the item has been removed. For whatever reason, an item may also be 'expunged' if necessary, in which case all traces of it are removed from the archive.

Objects in the DSpace Data Model

Object	Example
Community	Laboratory of Computer Science; Oceanographic Research Center
Collection	LCS Technical Reports; ORC Statistical Data Sets
Item	A technical report; a data set with accompanying description; a video recording of a lecture
Bundle	A group of HTML and image bitstreams making up an HTML document
Bitstream	A single HTML file; a single image file; a source code file
Bitstream Format	Microsoft Word version 6.0; JPEG encoded image format

Metadata

Broadly speaking, DSpace holds three sorts of metadata about archived content:

Descriptive Metadata

Each Item has one qualified Dublin Core metadata record. The set of elements and qualifiers used by MIT Libraries is the default configuration included in the DSpace source code. These are loosely based on the Library Application Profile set of elements and qualifiers, though there are some differences.

Other descriptive metadata about items may be held in serialized bitstreams. Communities and collections have some simple descriptive metadata (a name, and some descriptive prose), held in the DBMS.

Administrative Metadata

This includes preservation metadata, provenance and authorization policy data. Most of this is held within DSpace's relation DBMS schema. Provenance metadata (prose) is stored in Dublin Core records. Additionally, some other administrative metadata (for example, bitstream byte sizes and MIME types) is replicated in Dublin Core records so that it is easily accessible outside of DSpace.

Structural Metadata

This includes information about how to present an item, or bitstreams within an item, to an end-user, and the relationships between constituent parts of the item. As an example, consider a thesis consisting of a number of TIFF images, each depicting a single page of the thesis. Structural metadata would include the fact that each image is a single page, and the ordering of the TIFF images/pages. Structural metadata in DSpace is currently fairly basic; within an item, bitstreams can be arranged into separate bundles as described above. A bundle may also optionally have a primary bitstream. This is currently used by the HTML support to indicate which bitstream in the bundle is the first HTML file to send to a browser.

In addition to some basic technical metadata, bitstreams also have a 'sequence ID' that uniquely identifies it within an item. This is used to produce a 'persistent' bitstream identifier for each bitstream.

Additional structural metadata can be stored in serialized bitstreams, but DSpace does not currently understand this natively.

E-people

Many of DSpace's features such as document discovery and retrieval can be used anonymously, but users must be authenticated to perform functions such as submission, email notification ('subscriptions') or administration. Users are also grouped for easier administration. DSpace calls users e-people, to reflect that some users may be machines rather than actual people.

DSpace hold the following information about each e-person:

- E-mail address
- First and last names
- Whether the user is able to log in to the system via the Web UI, and whether they must use an X509 certificate to do so;
- A password (encrypted), if appropriate
- A list of collections for which the e-person wishes to be notified of new items
- Whether the e-person 'self-registered' with the system; that is, whether the system created the e-person record automatically as a result of the end-user independently registering with the system, as opposed to the e-person record being generated from the institution's personnel database, for example.

E-people authenticate with username/password pairs or X509 certificates. E-people can be members of 'groups' to make administrator's lives easier when manipulating authorization policies.

Authorization

DSpace's authorization system is based on associating actions with objects and the lists of EPeople who can perform them. The associations are called Resource Policies, and the lists of EPeople are called Groups. There are two special groups: 'administrators', who can do anything in a site, and 'anonymous', which is a list that contains all users. Assigning a policy

for an action on an object to anonymous means giving everyone permission to do that action. (For example, most objects in DSpace sites have a policy of 'anonymous' READ.) Permissions must be explicit - lack of an explicit permission results in the default policy of 'deny'. Permissions also do not 'commute'; for example, if an e-person has READ permission on an item, they might not necessarily have READ permission on the bundles and bitstreams in that item. Currently Collections, Communities and Items are discoverable in the browse and search systems regardless of READ authorization.

The following actions are possible:

Community

ADD/REMOVE	add or remove collections or sub-communities

Collection

ADD/REMOVE	add or remove items (ADD = permission to submit items)
DEFAULT_ITEM_READ	inherited as READ by all submitted items
DEFAULT_BITSTREAM_READ	inherited as READ by bitstreams of all submitted items
COLLECTION_ADMIN	collection admins can edit items in a collection, withdraw items, map other items into this collection.

Item

ADD/REMOVE	add or remove bundles
READ	can view item (item metadata is always viewable)
WRITE	can modify item

Bundle

ADD/REMOVE	add or remove bitstreams to a bundle

Bitstream

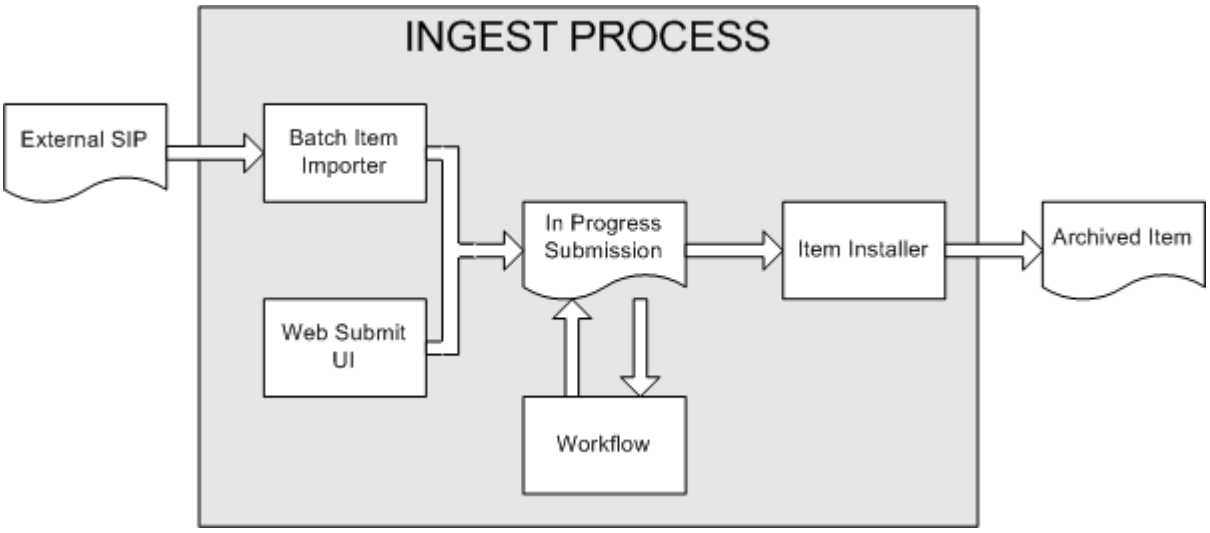
READ	view bitstream
WRITE	modify bitstream

Note that there is no 'DELETE' action. In order to 'delete' an object (e.g. an item) from the archive, one must have REMOVE permission on all objects (in this case, collection) that contain it. The 'orphaned' item is automatically deleted.

Policies can apply to individual e-people or groups of e-people.

Ingest Process and Workflow

Rather than being a single subsystem, ingesting is a process that spans several. Below is a simple illustration of the current ingesting process in DSpace.



DSpace Ingest Process

The batch item importer is an application, which turns an external SIP (an XML metadata document with some content files) into an "in progress submission" object. The Web submission UI is similarly used by an end-user to assemble an "in progress submission" object.

Depending on the policy of the collection to which the submission in targeted, a workflow process may be started. This typically allows one or more human reviewers or 'gatekeepers' to check over the submission and ensure it is suitable for inclusion in the collection.

When the Batch Ingester or Web Submit UI completes the InProgressSubmission object, and invokes the next stage of ingest (be that workflow or item installation), a provenance message is added to the Dublin Core which includes the filenames and checksums of the content of the submission. Likewise, each time a workflow changes state (e.g. a reviewer accepts the submission), a similar provenance statement is added. This allows us to track how the item has changed since a user submitted it. (The History system is also invoked, but provenance is easier for us to access at the moment.)

Once any workflow process is successfully and positively completed, the InProgressSubmission object is consumed by an "item installer", that converts the InProgressSubmission into a fully blown archived item in DSpace. The item installer:

- Assigns an accession date
- Adds a "date.available" value to the Dublin Core metadata record of the item
- Adds an issue date if none already present
- Adds a provenance message (including bitstream checksums)
- Assigns a Handle persistent identifier
- Adds the item to the target collection, and adds appropriate authorization policies
- Adds the new item to the search and browse indices

Workflow Steps

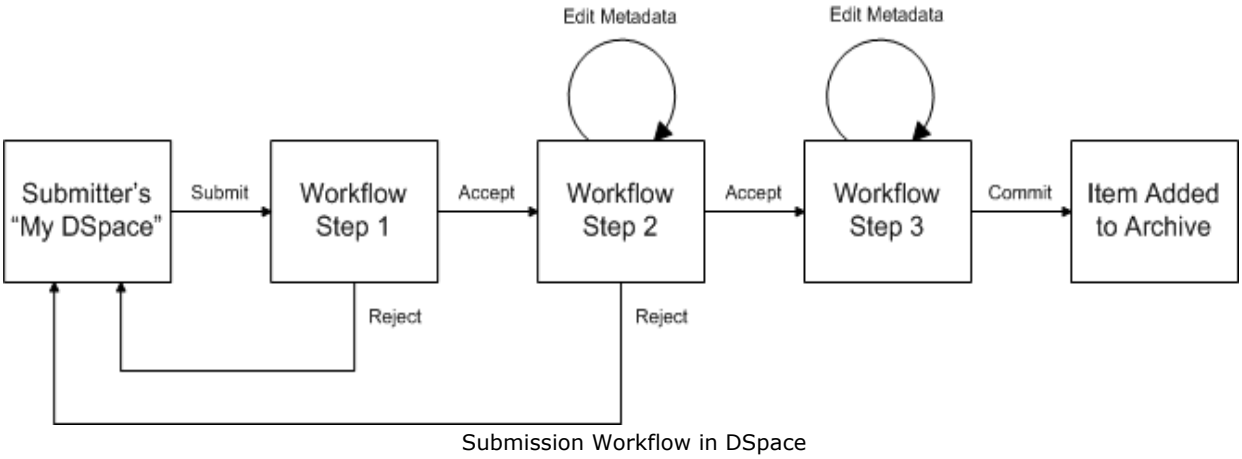
A collection's workflow can have up to three steps. Each collection may have an associated e-person group for performing each step; if no group is associated with a certain step, that step is skipped. If a collection has no e-person groups associated with any step, submissions to that collection are installed straight into the main archive.

In other words, the sequence is this: The collection receives a submission. If the collection has a group assigned for workflow step 1, that step is invoked, and the group is notified. Otherwise, workflow step 1 is skipped. Likewise, workflow steps 2 and 3 are performed if and only if the collection has a group assigned to those steps.

When a step is invoked, the task of performing that workflow step put in the 'task pool' of the associated group. One member of that group takes the task from the pool, and it is then removed from the task pool, to avoid the situation where several people in the group may be performing the same task without realizing it.

The member of the group who has taken the task from the pool may then perform one of three actions:

Workflow Step	Possible actions
1.	Can accept submission for inclusion, or reject submission.
2.	Can edit metadata provided by the user with the submission, but cannot change the submitted files. Can accept submission for inclusion, or reject submission.
3.	Can edit metadata provided by the user with the submission, but cannot change the submitted files. Must then commit to archive; may not reject submission.



Submission Workflow in DSpace

If a submission is rejected, the reason (entered by the workflow participant) is e-mailed to the submitter, and it is returned to the submitter's 'My DSpace' page. The submitter can then make any necessary modifications and re-submit, whereupon the process starts again.

If a submission is 'accepted', it is passed to the next step in the workflow. If there are no more workflow steps with associated groups, the submission is installed in the main archive.

One last possibility is that a workflow can be 'aborted' by a DSpace site administrator. This is accomplished using the administration UI.

The reason for this apparently arbitrary design is that is was the simplist case that covered the needs of the early adopter communities at MIT. The functionality of the workflow system will no doubt be extended in the future.

Handles

Researchers require a stable point of reference for their works. The simple evolution from sharing of citations to emailing of URLs broke when Web users learned that sites can disappear or be reconfigured without notice, and that their bookmark files containing critical links to research results couldn't be trusted long term. To help solve this problem, a core DSpace feature is the creation of persistent identifier for every item, collection and community stored in DSpace. To persist identifier, DSpace requires a storage- and location-independent mechanism for creating and maintaining identifiers. DSpace uses the CNRI Handle System for creating these identifiers. The rest of this section assumes a basic familiarity with the Handle system.

DSpace uses Handles primarily as a means of assigning globally unique identifiers to objects. Each site running DSpace needs to obtain a Handle 'prefix' from CNRI, so we know that if we create identifiers with that prefix, they won't clash with identifiers created elsewhere.

Presently, Handles are assigned to communities, collections, and items. Bundles and bitstreams are not assigned Handles, since over time, the way in which an item is encoded as bits may change, in order to allow access with future technologies and devices. Older versions may be moved to off-line storage as a new standard becomes de facto. Since it's usually the item that is being preserved, rather than the particular bit encoding, it only makes sense to persistently identify and allow access to the item, and allow users to access the appropriate bit encoding from there.

Of course, it may be that a particular bit encoding of a file is explicitly being preserved; in this case, the bitstream could be the only one in the item, and the item's Handle would then essentially refer just to that bitstream. The same bitstream can also be included in other items, and thus would be citable as part of a greater item, or individually.

The Handle system also features a global resolution infrastructure; that is, an end-user can enter a Handle into any service (e.g. Web page) that can resolve Handles, and the end-user will be directed to the object (in the case of DSpace, community, collection or item) identified by that Handle. In order to take advantage of this feature of the Handle system, a DSpace site must also run a 'Handle server' that can accept and resolve incoming resolution requests. All the code for this is included in the DSpace source code bundle.

Handles can be written in two forms:

```
hdl:1721.123/4567
http://hdl.handle.net/1721.123/4567
```

The above represent the same Handle. The first is possibly more convenient to use only as an identifier; however, by using the second form, any Web browser becomes capable of resolving Handles. An end-user need only access this form of the Handle as they would any other URL. It is possible to enable some browsers to resolve the first form of Handle as if they were standard URLs using CNRI's Handle Resolver plug-in, but since the first form can always be simply derived from the second, DSpace displays Handles in the second form, so that it is more useful for end-users.

It is important to note that DSpace uses the CNRI Handle infrastructure only at the 'site' level. For example, in the above example, the DSpace site has been assigned the prefix '1721.123'. It is still the responsibility of the DSpace site to maintain the association between a full Handle (including the '4567' local part) and the community, collection or item in question.

Bitstream 'Persistent' Identifiers

As of DSpace 1.2, bitstreams in DSpace also have more persistent identifiers. They are more volatile than Handles, since if the content is moved to a different server or organizaion, they will no longer work (hence the quotes around 'persistent'). However, they are more easily persisted than the simple URLs based on database primary key previously used. This means that external systems can more reliably refer to specific bitstreams stored in a DSpace instance.

Each bitstream has a sequence ID, unique within an item. This sequence ID is used to create a persistent ID, of the form:

dspace url/bitstream/handle/sequence ID/filename

For example:

https://dspace.myu.edu/bitstream/123.456/789/24/foo.html

The above refers to the bitstream with sequence ID 24 in the item with the Handle `hdl:123.456/789`. The `foo.html` is really just there as a hint to browsers: Although DSpace will provide the appropriate MIME type, some browsers only function correctly if the file has an expected extension.

Search and Browse

DSpace allows end-users to discover content in a number of ways, including:

- Via external reference, such as a Handle
- Searching for one or more keywords in metadata or extracted full-text
- Browsing though title, date and author indices, with optional image thumbnails

Search is an essential component of discovery in DSpace. Users' expectations from a search engine are quite high, so a goal for DSpace is to supply as many search features as possible. DSpace's indexing and search module has a very simple API which allows for indexing new content, regenerating the index, and performing searches on the entire corpus, a community, or collection. Behind the API is the Java freeware search engine Lucene. Lucene gives us fielded searching, stop word removal, stemming, and the ability to incrementally add new indexed content without regenerating the entire index.

Another important mechanism for discovery in DSpace is the browse. This is the process whereby the user views a particular index, such as the title index, and navigates around it in search of interesting items. The browse subsystem provides a simple API for achieving this by allowing a caller to specify an index, and a subsection of that index. The browse subsystem then discloses the portion of the index of interest. Indices that may be browsed are item title, item issue date and authors. Additionally, the browse can be limited to items within a particular collection or community.

HTML Support

For the most part, at present DSpace simply supports uploading and downloading of bitstreams as-is. This is fine for the majority of commonly-used file formats -- for example PDFs, Microsoft Word documents, spreadsheets and so forth. HTML documents (Web sites and Web pages) are far more complicated, and this has important ramifications when it comes to digital preservation:

- Web pages tend to consist of several files -- one or more HTML files that contain references to each other, and stylesheets and image files that are referenced by the HTML files.
- Web pages also link to or include content from other sites, often imperceptably to the end-user. Thus, in a few year's time, when someone views the preserved Web site, they will probably find that many links are now broken or refer to other sites than are now out of context.

  In fact, it may be unclear to an end-user when they are viewing content stored in DSpace and when they are seeing content included from another site, or have navigated to a page that is not stored in DSpace. This problem can manifest when a submitter uploads some HTML content. For example, the HTML document may include an image from an external Web site, or even their local hard drive. When the submitter views the HTML in DSpace, their browser is able to use the reference in the HTML to retrieve the appropriate image, and so to the submitter, the whole HTML document appears to have been deposited correctly. However, later on, when another user tries to view that HTML, their browser might not be able to retrieve the included image since it may have been removed from the external server. Hence the HTML will seem broken.

- Often Web pages are produced dynamically by software running on the Web server, and represent the state of a changing database underneath it.

Dealing with these issues is the topic of much active research. Currently, DSpace bites off a small, tractable chunk of this problem. DSpace can store and provide on-line browsing capability for self-contained, non-dynamic HTML documents. In practical terms, this means:

- No dynamic content (CGI scripts and so forth)
- All links to preserved content must be relative links, that do not refer to 'parents':
  - `diagram.gif` is OK
  - `image/foo.gif` is OK
  - `/stylesheet.css` is not OK
  - `http://somedomain.com/content.html` is not OK
- Any 'absolute links' (e.g. `http://somedomain.com/content.html`) are stored 'as is', and will continue to link to the external content (as opposed to relative links, which will link to the copy of the content stored in DSpace.) Thus, over time, the content refered to by the absolute link may change or disappear.

OAI Support

The Open Archives Initiative has developed a protocol for metadata harvesting. This allows sites to programmatically retrieve or 'harvest' the metadata from several sources, and offer services using that metadata, such as indexing or linking services. Such a service could allow users to access information from a large number of sites from one place.

DSpace exposes the Dublin Core metadata for items that are publicly (anonymously) accessible. Additionally, the collection structure is also exposed via the OAI protocol's 'sets' mechanism. OCLC's open source OAICat framework is used to provide this functionality.

DSpace's OAI service does support the exposing of deletion information for withdrawn items, but not for items that are 'expunged' (see above). DSpace also supports OAI-PMH resumption tokens.

OpenURL Support

DSpace supports the OpenURL protocol from SFX, in a rather simple fashion. If your institution has an SFX server, DSpace will display an OpenURL link on every item page, automatically using the Dublin Core metadata. Additionally, DSpace can respond to incoming OpenURLs. Presently it simply passes the information in the OpenURL to the search subsystem. A list of results is then displayed, which usually gives the relevant item (if it is in DSpace) at the top of the list.

Creative Commons Support

Dspace provides support for Creative Commons licenses to be attached to items in the repository. They represent an alternative to traditional copyright. To learn more about Creative Commons, visit their website. Support for the licenses is controlled by a site-wide configuration option, and since license selection involves redirection to the Creative Commons website, additional parameters may be configured to work with a proxy server. If the option is enabled, users may select a Creative Commons license during the submission process, or elect to skip Creative Commons licensing. If a selection is made a copy of the license text and RDF metadata is stored along with the item in the repository. There is also an indication - text and a Creative Commons icon - in the item display page of the web user interface when an item is licensed under Creative Commons.

Subscriptions

As noted above, end-users (e-people) may 'subscribe' to collections in order to be alerted when new items appear in those collections. Each day, end-users who are subscribed to one or more collections will receive an e-mail giving brief details of all new items that appeared in any of those collections the previous day. If no new items appeared in any of the subscribed collections, no e-mail is sent. Users can unsubscribe themselves at any time.

History

While provenance information in the form of prose is very useful, it is not easily programmatically manipulated. The History system captures a time-based record of significant changes in DSpace, in a manner suitable for later 'refactoring' or repurposing.

Currently, the History subsystem is explicitly invoked when significant events occur (e.g., DSpace accepts an item into the archive). The History subsystem then creates RDF data describing the current state of the object. The RDF data is modeled using Harmony/ABC, an ontology for describing temporal-based data, and stored in the file system. Some simple indices for unwinding the data are available.

Import and Export

DSpace also includes batch tools to import and export items in a simple directory structure, where the Dublin Core metadata is stored in an XML file. This may be used as the basis for moving content between DSpace and other systems.

There is also a METS-based export tool, which exports items as METS-based metadata with associated bitstreams referenced from the METS file.

Installation

Prerequisites

The list below describes the third-party components and tools you'll need to run a DSpace server. These are simply recommendations based on our setup at MIT; since DSpace is built on open source, standards-based tools, there are numerous other possibilities and setups.

Also, please note that the configuration and installation guidelines relating to a particular tool below are here for convenience. You should refer to the documentation for each individual component for complete and up-to-date details. Many of the tools are updated on a frequent basis, and the guidelines below may become out of date.

1. UNIX-like OS (Linux, HP/UX etc)
2. Java 1.4 or later (standard SDK is fine, you don't need J2EE)
3. Apache Ant 1.5 or later (Java make-like tool)
4. PostgreSQL 7.3 or later, an open source relational database.

Be sure to compile with the following options to the '`configure`' script:

--enable-multibyte --enable-unicode --with-java

Once installed, you need to enable TCP/IP connections (DSpace uses JDBC). Edit postgresql.conf (usually in /usr/local/pgsql/data or /var/lib/pgsql/data), and add this line:

```
tcpip_socket = true
```

Then tighten up security a bit by editing pg_hba.conf and adding this line:

```
host  dspace  dspace  127.0.0.1  255.255.255.255  md5
```

Then restart PostgreSQL.

5. Jakarta Tomcat 4.x/5.x or equivalent, such as Jetty or Caucho Resin.

   Note that DSpace will need to run as the same user as Tomcat, so you might want to install and run Tomcat as a user called '`dspace`'.

   Also you need to ensure that Tomcat has a) enough memory to run DSpace and b) uses UTF-8 as its default file encoding for international character support. So ensure in your startup scripts (etc) that the following environment variable is set:

   ```
   JAVA_OPTS="-Xmx512M -Xms64M -Dfile.encoding=UTF-8"
   ```

Quick Installation Steps

But First, a Word on Directories and Path Names

DSpace uses three separate directory trees. Although you don't need to know all the details of them in order to install DSpace, you do need to know they exist and also know how they're referred to in this document:

- the source directory, referred to as *`[dspace-source]`*
- the install directory, referred to as *`[dspace]`*
- the web deployment directory. If you're using Tomcat, this will be *`[tomcat]`*`/webapps/dspace` (with *`[tomcat]`* being wherever you installed Tomcat--also known as $CATALINA_HOME). This directory is generated by the web server when it unpacks dspace.war, and should never be edited.

For details on the contents of these separate directory trees, refer to directories.html. **Note that the source directory and install directory should always be separate!**

1. Create the DSpace user. This needs to be the same user that Tomcat (or Jetty etc) will run as. e.g. as root run:

   ```
   useradd -m dspace
   ```

2. Download the latest DSpace source code release and unpack it:

   ```
   gunzip -c dspace-source-1.x.tar.gz | tar -xf -
   ```

3. Copy the PostgreSQL JDBC driver (`.jar` file) into *`[dspace-source]`*`/lib`. If you compiled PostgreSQL yourself, it'll be in `postgresql-7.x.x/src/interfaces/jdbc/jars/postgresql.jar`. Alternatively you can download it directly from the PostgreSQL JDBC site. Make sure you get the driver for the version of PostgreSQL you're running and for JDBC2.

4. Create a `dspace` database, owned by the `dspace` PostgreSQL user:

   ```
   createuser -U postgres -d -A -P dspace ; createdb -U dspace -E UNICODE
   dspace
   ```

   Enter a password for the DSpace database. (This isn't the same as the `dspace` user's UNIX password.)

5. Edit *`[dspace-source]`*`/config/dspace.cfg`, in particular you'll need to set these properties:

   ```
   dspace.url
   dspace.hostname
   dspace.name
   db.password   (the password you entered in the previous step)
   mail.server
   mail.from.address
   feedback.recipient
   mail.admin
   alert.recipient (not essential but very useful!)
   ```

   Note that if you change `dspace.dir` you'll also need to change other properties with values that start with `/dspace`, e.g. `assetstore.dir`, `log.dir`...

6. Create the directory for the DSpace installation. As root, run:

   ```
   mkdir [dspace] ; chown dspace [dspace]
   ```

   (Assuming the `dspace` UNIX username.)

7. As the `dspace` UNIX user, compile and install DSpace:

   ```
   cd [dspace-source] ; ant fresh_install
   ```

   The most likely thing to go wrong here is the database connection. See the common problems section.

8. Copy the DSpace Web application archives (`.war` files) to the appropriate directory in your Tomcat/Jetty/Resin installation. For example:

   ```
   cp [dspace-source]/build/*.war [tomcat]/webapps
   ```

9. Create an initial administrator account:

```
[dspace]/bin/create-administrator
```

10. Now the moment of truth! Start up (or restart) Tomcat. Visit the base URL of your server, e.g. http://dspace.myu.edu:8080/dspace. You should see the DSpace home page. Congratulations!

In order to set up some communities and collections, you'll need to access the administration UI. To do this, append 'admin' to your server's URL, e.g. http://dspace.myu.edu:8080/dspace/dspace-admin.

Advanced Installation

The above installation steps are sufficient to set up a test server to play around with, but there are a few other steps and options you should probably consider before deploying a DSpace production site.

'cron' Jobs

A couple of DSpace features require that a script is run regularly -- the e-mail subscription feature that alerts users of new items being deposited, and the new 'media filter' tool, that generates thumbnails of images and extracts the full-text of documents for indexing.

To set these up, you just need to run the following command as the dspace UNIX user:

```
crontab -e
```

Then add the following lines:

```
# Send out subscription e-mails at 01:00 every day
0 1 * * * [dspace]/bin/sub-daily
# Run the media filter at 02:00 every day
0 2 * * * [dspace]/bin/filter-media
```

Naturally you should change the frequencies to suit your environment.

PostgreSQL also benefits from regular 'vacuuming', which optimizes the indices and clears out any deleted data. Become the `postgres` UNIX user, run `crontab -e` and add (for example):

```
# Clean up the database nightly at 2.40am
40 2 * * * vacuumdb --analyze dspace > /dev/null 2>&1
```

DSpace over HTTPS

Plain old HTTP is totally insecure, and if your DSpace uses username/password authentication or stores some restricted content, running it over HTTPS (HTTP over a Secure Socket Layer (SSL)) is advisable. There are two options for this: Using Apache HTTPD, or Tomcat/Jetty's in-built HTTPS support.

To use Apache HTTPD: The DSpace source bundle includes a partial Apache configuration `apache13.conf`, which contains most of the DSpace-specific configuration required. It assumes you're using mod_webapp, which is deprecated and tricky to compile but a lot easier to configure than `mod_jk2` which is the current recommendation from Tomcat. Use of this is optional, you might just want to use it as an example. To use it directly, in the main Apache `httpd.conf`, you should:

- Make sure `mod_ssl` and `mod_webapp` are configured and loaded
- Remove/comment out etc. any existing or default SSL virtual host
- Ensure Apache will run with the UNIX user and group DSpace will run as
- Include the DSpace part, e.g. with: `Include [dspace]/config/httpd.conf`. You can decide where the DSpace part will go in your file system--see the configuration section.

To use Tomcat or Jetty's HTTPS support consult the documentation for the relevant tool. Also, these alternative DSpace install docs briefly describe getting Tomcat running with SSL.

The Handle Server

First a few facts to clear up some common misconceptions:

- You don't **have** to use CNRI's Handle system. At the moment, you need to change the code a little to use something else (e.g PURLs) but that should change soon.
- You'll notice that while you've been playing around with a test server, DSpace has apparently been creating handles for you looking like `hdl:123456789/24` and so forth. These aren't really Handles, since the global Handle system doesn't actually know about them, and lots of other DSpace test installs will have created the same IDs.

  They're only really Handles once you've registered a prefix with CNRI (see below) and have correctly set up the Handle server included in the DSpace distribution. This Handle server communicates with the rest of the global Handle infrastructure so that anyone that understands Handles can find the Handles your DSpace has created.

If you want to use the Handle system, you'll need to set up a Handle server. This is included with DSpace. Note that this is not required in order to evaluate DSpace; you only need one if you are running a production service. You'll need to obtain a Handle prefix from the central CNRI Handle site.

A Handle server runs as a separate process that receives TCP requests from other Handle servers, and issues resolution requests to a global server or servers if a Handle entered locally does not correspond to some local content. The Handle protocol is based on TCP, so it will need to be installed on a server that can broadcast and receive TCP on port 2641.

The Handle server code is included with the DSpace code in `[dspace-source]/lib/handle.jar`. A script exists to create a simple Handle configuration - simply run `[dspace]/bin/make-handle-config` after you've set the appropriate parameters in `dspace.cfg`. You can also create a Handle configuration directly by following the installation instructions on handle.net, but with these changes:

- Instead of running:

```
java -cp /hs/bin/handle.jar net.handle.server.SimpleSetup /hs/svr_1
```

as directed in the Handle Server Administration Guide, you should run

```
[dspace]/bin/dsrun net.handle.server.SimpleSetup [dspace]/handle-server
```

ensuring that *[dspace]*/handle-server matches whatever you have in dspace.cfg for the handle.dir property.

- Edit the resulting *[dspace]*/handle-server/config.dct file to include the following lines in the "server_config" clause:

```
"storage_type" = "CUSTOM"
"storage_class" = "org.dspace.handle.HandlePlugin"
```

This tells the Handle server to get information about individual Handles from the DSpace code.

Whichever approach you take, start the Handle server with *[dspace]*/bin/start-handle-server, as the DSpace user. You will need to send the sitebndl.zip file to hdladmin@cnri.reston.va.us as described in the Handle server documentation.

Note that since the DSpace code manages individual Handles, administrative operations such as Handle creation and modification aren't supported by DSpace's Handle server.

Checking Your Installation

TODO

Known Bugs

In any software project of the scale of DSpace, there will be bugs. Sometimes, a stable version of DSpace includes known bugs. We do not always wait until every known bug is fixed before a release. If the software is sufficiently stable and an improvement on the previous release, and the bugs are minor and have known workarounds, we release it to enable the community to take advantage of those improvements.

The known bugs in a release are documented in the KNOWN_BUGS file in the source package.

Please see the DSpace bug tracker for further information on current bugs, and to find out if the bug has subsequently been fixed. This is also where you can report any further bugs you find.

Common Problems

In an ideal world everyone would follow the above steps and have a fully functioning DSpace. Of couse, in the real world it doesn't always seem to work out that way. This section lists

common problems that people encounter when installing DSpace, and likely causes and fixes. This is likely to grow over time as we learn about users' experiences.

Database errors occur when you run `ant fresh_install`

There are two common errors that occur. If your error looks like this—

```
[java] 2004-03-25 15:17:07,730 INFO org.dspace.storage.rdbms.InitializeDatabase @
Initializing Database
[java] 2004-03-25 15:17:08,816 FATAL org.dspace.storage.rdbms.InitializeDatabase @
Caught exception:
[java] org.postgresql.util.PSQLException: Connection refused. Check that the
hostname and port are correct and that the postmaster is accepting TCP/IP
connections.
[java] at
org.postgresql.jdbc1.AbstractJdbc1Connection.openConnection(AbstractJdbc1Connectio
n.java:204)
[java] at org.postgresql.Driver.connect(Driver.java:139)
```

it usually means you haven't yet added the relevant configuration parameter to your PostgreSQL configuration (see above), or perhaps you haven't restarted PostgreSQL after making the change. Also, make sure that the `db.username` and `db.password` properties are correctly set in *`[dspace-source]`*`/config/dspace.cfg`.

An easy way to check that your DB is working OK over TCP/IP is to try this on the command line:

```
psql -U dspace -W -h localhost
```

Enter the `dspace` database password, and you should be dropped into the psql tool with a `dspace=>` prompt.

Another common error looks like this:

```
[java] 2004-03-25 16:37:16,757 INFO  org.dspace.storage.rdbms.InitializeDatabase @
Initializing Database
[java] 2004-03-25 16:37:17,139 WARN   org.dspace.storage.rdbms.DatabaseManager @
Exception initializing DB pool
[java] java.lang.ClassNotFoundException: org.postgresql.Driver
[java]     at java.net.URLClassLoader$1.run(URLClassLoader.java:198)
[java]     at java.security.AccessController.doPrivileged(Native Method)
[java]     at java.net.URLClassLoader.findClass(URLClassLoader.java:186)
```

This means that the PostgreSQL JDBC driver is not present in *`[dspace-source]`*`/lib`. See above.

Tomcat doesn't shut down

If you're trying to tweak Tomcat's configuration but nothing seems to make a difference to the error you're seeing, you might find that Tomcat hasn't been shutting down properly, perhaps because it's waiting for a stale connection to close gracefully which won't happen. To see if this is the case, try:

```
ps -ef | grep java
```

and look for Tomcat's Java processes. If they stay arround after running Tomcat's `shutdown.sh` script, trying `kill`ing them (with `-9` if necessary), then starting Tomcat again.

Database connections don't work, or accessing DSpace takes forever

If you find that when you try to access a DSpace Web page and your browser sits there connecting, or if the database connections fail, you might find that a 'zombie' database connection is hanging around preventing normal operation. To see if this is the case, try:

```
ps -ef | grep postgres
```

You might see some processes like this

```
dspace 16325  1997  0  Feb 14  ?         0:00 postgres: dspace dspace
127.0.0.1 idle in transaction
```

This is normal--DSpace maintains a 'pool' of open database connections, which are re-used to avoid the overhead of constantly opening and closing connections. If they're 'idle' it's OK; they're waiting to be used. However sometimes, if something went wrong, they might be stuck in the middle of a query, which seems to prevent other connections from operating, e.g.:

```
dspace 16325  1997  0  Feb 14  ?         0:00 postgres: dspace dspace
127.0.0.1 SELECT
```

This means the connection is in the middle of a `SELECT` operation, and if you're not using DSpace right that instant, it's probably a 'zombie' connection. If this is the case, try `kill`ing the process, and stopping and restarting Tomcat.

You've made changes to the code or to the JSP's and rebuilt DSpace successfully, but when you run Tomcat you don't see any of your changes in DSpace.

After you've rebuilt DSpace and copied dspace.war from your *[dspace-source]*/build directory into your *[tomcat]*/webapps directory, you must also delete the existing *[tomcat]*/webapps/dspace directory before re-starting Tomcat. Otherwise Tomcat will continue to use the old code.

Updating a DSpace Installation

This section describes how to update a DSpace installation from one version to the next. Details of the differences between the functionality of each version are given in the Version History section.

Updating From 1.2 to 1.2.1

The changes in 1.2.1 are only code changes so the update should be fairly simple.

In the notes below *[dspace]* refers to the install directory for your existing DSpace installation, and *[dspace-1.2.1-source]* to the source directory for DSpace 1.2.1. Whenever you see these path references, be sure to replace them with the actual path names on your local system.

1. Get the new DSpace 1.2.1 source code from the DSpace page on SourceForge and unpack it somewhere. Do not unpack it on top of your existing installation!!

2. Copy the PostgreSQL driver JAR to the source tree. For example:

```
cd [dspace]/lib
cp postgresql.jar [dspace-1.2.1-source]/lib
```

3. Take down Tomcat (or whichever servlet container you're using).

4. Your 'localized' JSPs (those in jsp/local) now need to be maintained in the source directory. If you have locally modified JSPs in your *[dspace]*/jsp/local directory, you might like to merge the changes in the new 1.2.1 versions into your locally modified ones. You can use the `diff` command to compare the 1.2 and 1.2.1 versions to do this. Also see the version history for a list of modified JSPs.

5. You need to add a few new parameters to your *[dspace]*/dspace.cfg for browse/search and item thumbnails display, and for configurable DC metadata fields to be indexed.

```
# whether to display thumbnails on browse and search results pages
(1.2+)
webui.browse.thumbnail.show = false

# max dimensions of the browse/search thumbs. Must be <=
thumbnail.maxwidth
# and thumbnail.maxheight. Only need to be set if required to be smaller
than
# dimension of thumbnails generated by mediafilter (1.2+)
#webui.browse.thumbnail.maxheight = 80
#webui.browse.thumbnail.maxwidth = 80

# whether to display the thumb against each bitstream (1.2+)
webui.item.thumbnail.show = true

# where should clicking on a thumbnail from browse/search take the user
# Only values currently supported are "item" and "bitstream"
#webui.browse.thumbnail.linkbehaviour = item

##### Fields to Index for Search #####

# DC metadata elements.qualifiers to be indexed for search
# format: - search.index.[number] = [search field]:element.qualifier
#         - * used as wildcard
###      changing these will change your search results,     ###
###  but will NOT automatically change your search displays  ###

search.index.1 = author:contributor.*
search.index.2 = author:creator.*
search.index.3 = title:title.*
```

```
search.index.4 = keyword:subject.*
search.index.5 = abstract:description.abstract
search.index.6 = author:description.statementofresponsibility
search.index.7 = series:relation.ispartofseries
search.index.8 = abstract:description.tableofcontents
search.index.9 = mime:format.mimetype
search.index.10 = sponsor:description.sponsorship
search.index.11 = id:identifier.*
```

6. In *[dspace-1.2.1-source]* run:

```
ant -Dconfig=[dspace]/config/dspace.cfg update
```

7. Copy the `.war` Web application files in *[dspace-1.2.1-source]*`/build` to the `webapps` sub-directory of your servlet container (e.g. Tomcat). e.g.:

```
cp [dspace-1.2.1-source]/build/*.war [tomcat]/webapps
```

   If you're using Tomcat, you need to delete the directories corresponding to the old `.war` files. For example, if `dspace.war` is installed in *[tomcat]*`/webapps/dspace.war`, you should delete the *[tomcat]*`/webapps/dspace` directory. Otherwise, Tomcat will continue to use the old code in that directory.

8. Restart Tomcat.

Updating From 1.1 (or 1.1.1) to 1.2

The process for upgrading to 1.2 from either 1.1 or 1.1.1 is the same. If you are running DSpace 1.0 or 1.0.1, you need to follow the instructions for upgrading from 1.0.1 to 1.1 to before following these instructions.

Note also that if you've substantially modified DSpace, these instructions apply to an unmodified 1.1.1 DSpace instance, and you'll need to adapt the process to any modifications you've made.

This document refers to the install directory for your existing DSpace installation as *[dspace]*, and to the source directory for DSpace 1.2 as *[dspace-1.2-source]*. Whenever you see these path references below, be sure to replace them with the actual path names on your local system.

1. Step one is, of course, to back up all your data before proceeding!! Include all of the contents of *[dspace]* and the PostgreSQL database in your backup.

2. Get the new DSpace 1.2 source code from the DSpace page on SourceForge and unpack it somewhere. Do not unpack it on top of your existing installation!!

3. Copy the required Java libraries that we couldn't include in the bundle to the source tree. For example:

```
cd [dspace]/lib
```

```
cp activation.jar servlet.jar mail.jar [dspace-1.2-source]/lib
```

4. Stop Tomcat (or other servlet container.)

5. It's a good idea to upgrade all of the various third-party tools that DSpace uses to their latest versions:

   - Java (note that now version 1.4.0 or later is required)
   - Tomcat (Any version after 4.0 will work; symbolic links are no longer an issue)
   - PostgreSQL (don't forget to build/download an updated JDBC driver .jar file! **Also, back up the database first**.)
   - Ant

6. You need to add the following new parameters to your *[dspace]*/dspace.cfg:

   ```
   ##### Media Filter settings #####
   # maximum width and height of generated thumbnails
   thumbnail.maxwidth  80
   thumbnail.maxheight 80
   ```

   There are one or two other, optional extra parameters (for controlling the pool of database connections). See the version history for details. If you leave them out, defaults will be used.

   Also, to avoid future confusion, you might like to **remove** the following property, which is no longer required:

   ```
   config.template.oai-web.xml = [dspace]/oai/WEB-INF/web.xml
   ```

7. The layout of the installation directory (i.e. the structure of the contents of *[dspace]*) has changed somewhat since 1.1.1. First up, your 'localized' JSPs (those in jsp/local) now need to be maintained in the source directory. So make a copy of them now!

   Once you've done that, you can remove *[dspace]*/jsp and *[dspace]*/oai, these are no longer used. (.war Web application archive files are used instead).

   Also, if you're using the same version of Tomcat as before, you need **to remove the lines from Tomcat's conf/server.xml file that enable symbolic links for DSpace.** These are the <Context> elements you added to get DSpace 1.1.1 working, looking something like this:

   ```
   <Context path="/dspace" docBase="dspace" debug="0" reloadable="true"
   crossContext="true">
     <Resources className="org.apache.naming.resources.FileDirContext"
   allowLinking="true" />
   </Context>
   ```

   Be sure to remove the <Context> elements for both the Web UI and the OAI Web applications.

8. Build and install the updated DSpace 1.2 code. Go to the DSpace 1.2 source directory, and run:

```
ant -Dconfig=[dspace]/config/dspace.cfg update
```

9. Copy the new config files in `config` to your installation, e.g.:

   ```
   cp [dspace-1.2-source]/config/news-* [dspace-1.2-
   source]/config/mediafilter.cfg [dspace-1.2-source]/config/dc2mods.cfg
   [dspace]/config
   ```

10. You'll need to make some changes to the database schema in your PostgreSQL database. *`[dspace-1.2-source]`*`/etc/database_schema_11-12.sql` contains the SQL commands to achieve this. If you've modified the schema locally, you may need to check over this and make alterations.

    To apply the changes, go to the source directory, and run:

    ```
    psql -f etc/database_schema_11-12.sql [DSpace database name] -h
    localhost
    ```

11. A tool supplied with the DSpace 1.2 codebase will then update the actual data in the relational database. Run it using:

    ```
    [dspace]/bin/dsrun org.dspace.administer.Upgrade11To12
    ```

12. Then rebuild the search indices:

    ```
    [dspace]/bin/index-all
    ```

13. Delete the existing symlinks from your servlet container's (e.g. Tomcat's) `webapp` sub-directory.

    Copy the `.war` Web application files in *`[dspace-1.2-source]`*`/build` to the `webapps` sub-directory of your servlet container (e.g. Tomcat). e.g.:

    ```
    cp [dspace-1.2-source]/build/*.war [tomcat]/webapps
    ```

14. Restart Tomcat.

15. To get image thumbnails generated and full-text extracted for indexing automatically, you need to set up a 'cron' job, for example one like this:
    ```
    # Run the media filter at 02:00 every day
    0 2 * * * [dspace]/bin/filter-media
    ```

    You might also wish to run it now to generate thumbnails and index full text for the content already in your system.

16. **Note 1:** This update process has effectively 'touched' all of your items. Although the dates in the Dublin Core metadata won't have changed (accession date and so forth), the 'last modified' date in the database for each will have been changed.

    This means the e-mail subscription tool may be confused, thinking that all items in the archive have been deposited that day, and could thus send a rather long email to

lots of subscribers. So, it is recommended that you **turn off the e-mail subscription feature for the next day**, by commenting out the relevant line in DSpace's cron job, and then re-activating it the next day.

Say you performed the update on 08-June-2004 (UTC), and your e-mail subscription cron job runs at 4am (UTC). When the subscription tool runs at 4am on 09-June-2004, it will find that everything in the system has a modification date in 08-June-2004, and accordingly send out huge emails. So, immediately after the update, you would edit DSpace's 'crontab' and comment out the `/dspace/bin/subs-daily` line. Then, after 4am on 09-June-2004 you'd 'un-comment' it out, so that things proceed normally.

Of course this means, any real new deposits on 08-June-2004 won't get e-mailed, however if you're updating the system it's likely to be down for some time so this shouldn't be a big problem.

17. **Note 2:** After consulation with the OAI community, various OAI-PMH changes have occurred:
    - The OAI-PMH identifiers have changed (they're now of the form `oai:hostname:handle` as opposed to just Handles)
    - The set structure has changed, due to the new sub-communities feature.
    - The default base URL has changed
    - As noted in note 1, every item has been 'touched' and will need re-harvesting.

    The above means that, if already registered and harvested, you will need to re-register your repository, effectively as a 'new' OAI-PMH data provider. You should also consider posting an announcement to the OAI implementers e-mail list so that harvesters know to update their systems.

    Also note that your site may, over the next few days, take quite a big hit from OAI-PMH harvesters. The resumption token support should alleviate this a little, but you might want to temporarily whack up the database connection pool parameters in `[dspace]/config/dspace.cfg`. See the `dspace.cfg` distributed with the source code to see what these parameters are and how to use them. (You need to stop and restart Tomcat after changing them.)

    I realize this is not ideal; for discussion as to the reasons behind this please see relevant posts to the OAI community: post one, post two, as well as this post to the dspace-tech mailing list.

    If you really can't live with updating the base URL like this, you can fairly easily have thing proceed more-or-less as they are, by doing the following:

    - Change the value of `OAI_ID_PREFIX` at the top of the `org.dspace.app.oai.DSpaceOAICatalog` class to `hdl:`
    - Change the servlet mapping for the `OAIHandler` servlet back to `/` (from `/request`)
    - Rebuild and deploy `dspace-oai.war`

    However, note that in this case, all the records will be re-harvested by harvesters anyway, so you still need to brace for the associated DB activity; also note that the

set spec changes may not be picked up by some harvesters. It's recommended you read the above-linked mailing list posts to understand why the change was made.

Now, you should be finished!

Updating From 1.1 to 1.1.1

Fortunately the changes in 1.1.1 are only code changes so the update is fairly simple.

In the notes below *[dspace]* refers to the install directory for your existing DSpace installation, and *[dspace-1.1.1-source]* to the source directory for DSpace 1.1.1. Whenever you see these path references, be sure to replace them with the actual path names on your local system.

1. Take down Tomcat.

2. It would be a good idea to update any of the third-party tools used by DSpace at this point (e.g. PostgreSQL), following the instructions provided with the relevant tools.
3. In *[dspace-1.1.1-source]* run:

   ```
   ant -Dconfig=[dspace]/config/dspace.cfg update
   ```

4. If you have locally modified JSPs of the following JSPs in your *[dspace]*/jsp/local directory, you might like to merge the changes in the new 1.1.1 versions into your locally modified ones. You can use the diff command to compare the 1.1 and 1.1.1 versions to do this. The changes are quite minor.

   ```
   collection-home.jsp
   admin/authorize-collection-edit.jsp
   admin/authorize-community-edit.jsp
   admin/authorize-item-edit.jsp
   admin/eperson-edit.jsp
   ```

5. Restart Tomcat.

Updating From 1.0.1 to 1.1

To upgrade from DSpace 1.0.1 to 1.1, follow the steps below. Your `dspace.cfg` does not need to be changed. In the notes below *[dspace]* refers to the install directory for your existing DSpace installation, and *[dspace-1.1-source]* to the source directory for DSpace 1.1. Whenever you see these path references, be sure to replace them with the actual path names on your local system.

1. Take down Tomcat (or whichever servlet container you're using).
2. We recommend that you upgrage to the latest version of PostgreSQL (7.3.2). Included are some notes to help you do this. Note you will also have to upgrade Ant to version 1.5 if you do this.

3. Make the necessary changes to the DSpace database. These include a couple of minor schema changes, and some new indices which should improve performance. Also, the

names of a couple of database views have been changed since the old names were so long they were causing problems. First run `psql` to access your database (e.g. `psql -U dspace -W` and then enter the password), and enter these SQL commands:

```
ALTER TABLE bitstream ADD store_number INTEGER;
UPDATE bitstream SET store_number = 0;
ALTER TABLE item ADD last_modified TIMESTAMP;
CREATE INDEX last_modified_idx ON Item(last_modified);
CREATE INDEX eperson_email_idx ON EPerson(email);
CREATE INDEX item2bundle_item_idx on Item2Bundle(item_id);
REATE INDEX bundle2bitstream_bundle_idx ON Bundle2Bitstream(bundle_id);
CREATE INDEX dcvalue_item_idx on DCValue(item_id);
CREATE INDEX collection2item_collection_idx ON
Collection2Item(collection_id);
CREATE INDEX resourcepolicy_type_id_idx ON ResourcePolicy
(resource_type_id,resource_id);
CREATE INDEX epersongroup2eperson_group_idx on
EPersonGroup2EPerson(eperson_group_id);
CREATE INDEX handle_handle_idx ON Handle(handle);
CREATE INDEX sort_author_idx on ItemsByAuthor(sort_author);
CREATE INDEX sort_title_idx on ItemsByTitle(sort_title);
CREATE INDEX date_issued_idx on ItemsByDate(date_issued);
DROP VIEW CollectionItemsByDateAccessioned;
DROP VIEW CommunityItemsByDateAccessioned;
CREATE VIEW CommunityItemsByDateAccession as SELECT
Community2Item.community_id, ItemsByDateAccessioned.*  FROM
ItemsByDateAccessioned, Community2Item WHERE
ItemsByDateAccessioned.item_id = Community2Item.item_id;
CREATE VIEW CollectionItemsByDateAccession AS SELECT
collection2item.collection_id,
itemsbydateaccessioned.items_by_date_accessioned_id,
itemsbydateaccessioned.item_id, itemsbydateaccessioned.date_accessioned
FROM itemsbydateaccessioned, collection2item WHERE
(itemsbydateaccessioned.item_id = collection2item.item_id);
```

4. Fix your JSPs for Unicode. If you've modified the site 'skin' (jsp/local/layout/header-default.jsp) you'll need to add the Unicode header, i.e.:

   ```
   <meta http-equiv="Content-Type" content="text/html; charset=UTF-8">
   ```

   to the <HEAD> element. If you have any locally-edited JSPs, you need to add this page directive to the top of all of them:

   ```
   <%@ page contentType="text/html;charset=UTF-8" %>
   ```

   (If you haven't modified any JSPs, you don't have to do anything.)

5. Copy the required Java libraries that we couldn't include in the bundle to the source tree. For example:

   ```
   cd [dspace]/lib
   cp *.policy activation.jar servlet.jar mail.jar [dspace-1.1-source]/lib
   ```

6. Compile up the new DSpace code, replacing *[dspace]*/config/dspace.cfg with the path to your current, LIVE configuration. (The second line, touch `find .`, is a precaution, which ensures that the new code has a current datestamp and will overwrite the old code. Note that those are back quotes.)

   ```
   cd [dspace-1.1-source]
   touch `find .`
   ant
   ant -Dconfig=[dspace]/config/dspace.cfg update
   ```

7. Update the database tables using the upgrader tool, which sets up the new >last_modified date in the item table:

   ```
   Run [dspace]/bin/dsrun org.dspace.administer.Upgrade101To11
   ```

8. Run the collection default authorisation policy tool:

   ```
   [dspace]/bin/dsrun org.dspace.authorize.FixDefaultPolicies
   ```

9. Fix the OAICat properties file. Edit *[dspace]*/config/templates/oaicat.properties. Change the line that says

   ```
   Identify.deletedRecord=yes
   ```

   To:

   ```
   Identify.deletedRecord=persistent
   ```

   This is needed to fix the OAI-PMH 'Identity' verb response. Then run *[dspace]*/bin/install-configs.

10. Re-run the indexing to index abstracts and fill out the renamed database views:

    ```
    [dspace]/bin/index-all
    ```

11. Restart Tomcat. Tomcat should be run with the following environment variable set, to ensure that Unicode is handled properly. Also, the default JVM memory heap sizes are rather small. Adjust -Xmx512M (512Mb maximum heap size) and -Xms64M (64Mb Java thread stack size) to suit your hardware.

    ```
    JAVA_OPTS="-Xmx512M -Xms64M -Dfile.encoding=UTF-8"
    ```

Configuration and Customization

There are a number of ways in which DSpace can be configured and/or customized:

- Altering the configuration files in *[dspace]*/config

- Creating modified versions of the JSPs; these can be placed separately from and override the default installed JSPs, so that future updates of the code won't overwrite your changes
- Implementing a custom 'authenticator' class, so that user authentication in the Web UI can be adapted and integrated with any existing mechanisms your organization might use
- Editing the source code

Of these methods, only the last is likely to cause any headaches; if you update the DSpace source code directly, it may make applying future updates difficult. However, DSpace is open source, of course, and if you make any modifications that might be helpful to other institutions or organizations, feel free to send them to the DSpace team at MIT.

The `dspace.cfg` Configuration Properties File

The primary way of configuring DSpace is to edit the `dspace.cfg`. You'll definitely have to do this before you can operate DSpace properly. `dspace.cfg` contains basic information about a DSpace installation, including system path information, network host information, and other things like site name.

The default `dspace.cfg` is a good source of information, and contains comments for all properties. It's a basic Java properties file, where lines are either comments, starting with a '`#`', blank lines, or property/value pairs of the form:

```
property.name = property value
```

Due to time constraints, this document does not contain an exhaustive list of properties; they are all listed in the supplied `dspace.cfg`. Below are some particularly relevant properties with notes for their use:

`dspace.cfg` Main Properties (Not Complete)

Property	Example Values	Notes
`dspace.dir`	`/dspace`	Root directory of DSpace installation. Omit the trailing '/'. Note that if you change this, there are several other parameters you will probably want to change to match, e.g. `assetstore.dir`.
`dspace.url`	`http://dspace.myu.edu` `http://dspacetest.myu.edu:8080`	Main URL at which DSpace Web UI webapp is deployed. Include any port number, but do not include a trailing '/'
`dspace.hostname`	`dspace.myu.edu`	Fully qualified hostname; do not include port number
`dspace.name`	`DSpace at My University`	Short and sweet site name, used throughout Web UI, e-mails and elsewhere (such as OAI protocol)

config.template.foo	/opt/othertool/cfg/foo	When install-configs is run, the file [dspace]/config/templates/foo file will be filled out with values from dspace.cfg and copied to the value of this property, in this example /opt/othertool/cfg/foo. See here for more information.
webui.site.authenticator	edu.myu.MyAuthenticator	The Java class name of a class implementing the org.dspace.app.webui.SiteAuthenticator interface.
handle.prefix	1721.1234	The Handle prefix for your site, see the Handle section
assetstore.dir	/bigdisk/store	The location in the file system for asset (bitstream) store number zero. This should be a directory for the sole use of DSpace.
assetstore.dir.n	/anotherdisk/store1	The location in the file system of asset (bitstream) store number n. When adding additional stores, start with 1 (assetstore.dir.1 and count upwards. Always leave asset store zero (assetstore.dir). For more details, see the Bitstream Storage section.
assetstore.incoming	1	The asset store number to use for storing new bitstreams. For example, if assetstore.dir.1 is /anotherdisk/store1, and assetstore.incoming is 1, new bitstreams will be stored under /anotherdisk/store1. A value of 0 (zero) corresponds to assetstore.dir. For more details, see the Bitstream Storage section.
webui.submit.enable-cc	true	Enable the Creative Commons license step in the submission process. Submitters are given an opportunity to select a Creative Commons license to accompany the Item. Creative Commons licenses govern the use of the content. For more details, see the Creative Commons website.

Whenever you edit dspace.cfg, you should then run *[dspace]*/bin/install-configs so that any changes you may have made are reflected in the configuration files of other

applications, for example Apache. You may then need to restart those applications, depending on what you changed.

Wording of E-mail Messages

Sometimes DSpace automatically sends e-mail messages to users, for example to inform them of a new workflow task, or as a subscription e-mail alert. The wording of emails can be changed by editing the relevant file in *[dspace]*/config/emails. Each file is commented. Be careful to keep the right number 'placeholders' (e.g.{2}).

Local DSpace Administrator Contact Information

There are several places in DSpace in which the user will be shown contact information for the local DSpace Administrator: for instance, when an error occurs, or in the on-line help when the user is looking for more information. The contact information is displayed by *[dspace-source]*/jsp/components/contact-info.jsp. This JSP retrieves the help e-mail in dspace.cfg, but the phone number in the JSP is a dummy phone number that needs to be edited directly in the JSP. You should be sure to edit this file (adding any additional information you feel might be useful) so that users know who to contact for further information.

The Dublin Core and Bitstream Format Registries

The *[dspace]*/config/registries directory contains two XML files. These are used to load the initial contents of the Dublin Core type registry and Bitstream Format registry. After the initial loading (performed by ant fresh_install above), the registries reside in the database; the XML files are not updated.

Currently, the system requires that every item have a Dublin Core record. The exact Dublin Core elements and qualifiers that are used can be configured by editing the Dublin Core registry. This can either be done at install-time, by editing *[dspace]*/config/registries/dublin-core-types.xml, or at run-time using the administration Web UI. However, note that some elements and qualifiers must be present for DSpace to function correctly since they are used for various purposes by the code. Details are in the relevant .xml file.

Also note that altering the Dublin Core registry does not, at the current time, cause corresponding changes in the Web UI (e.g. the submission interface or search indices).

The bitstream formats recognized by the system and levels of support are similarly stored in the bitstream format registry. This can also be edited at install-time via *[dspace]*/config/registries/bitstream-formats.xml or by the administation Web UI. The contents of the bitstream format registry are entirely up to you, though the system requires that the following two formats are present:

- Unknown
- License

Configuration Files for Other Applications

To ease the hassle of keeping configuration files for other applications involved in running a DSpace site, for example Apache, in sync, the DSpace system can automatically update them for you when the main DSpace configuration is changed. This feature of the DSpace system is entirely optional, but we found it useful.

The way this is done is by placing the configuration files for those applications in *[dspace]*/config/templates, and inserting special values in the configuration file that will be filled out with appropriate DSpace configuration properties. Then, tell DSpace where to put filled-out, 'live' version of the configuration by adding an appropriate property to dspace.cfg, and run *[dspace]*/bin/install-configs.

Take the apache13.conf file as an example. This contains plenty of Apache-specific stuff, but where it uses a value that should be kept in sync across DSpace and associated applications, a 'placeholder' value is written. For example, the host name:

```
ServerName @@dspace.hostname@@
```

The text @@dspace.hostname@@ will be filled out with the value of the dspace.hostname property in dspace.cfg. Then we decide where we want the 'live' version, that is, the version actually read in by Apache when it starts up, will go.

Let's say we want the live version to be located at /opt/apache/conf/dspace-httpd.conf. To do this, we add the following property to dspace.cfg so DSpace knows where to put it:

```
config.template.apache13.conf = /opt/apache/conf/dspace-httpd.conf
```

Now, we run *[dspace]*/bin/install-configs. This reads in *[dspace]*/config/templates/apache13.conf, and places a copy at /opt/apache/conf/dspace-httpd.conf with the placeholders filled out.

So, in /opt/apache/conf/dspace-httpd.conf, there will be a line like:

```
ServerName dspace.myu.edu
```

The advantage of this approach is that if a property like the hostname changes, you can just change it in dspace.cfg and run install-configs, and all of your tools' configuration files will be updated.

However, take care to make all your edits to the versions in *[dspace]*/config/templates! It's a wise idea to put a big reminder at the top of each file, since someone might unwittingly edit a 'live' configuration file which would later be overwritten.

Customizing the Web User Interface

The Web UI is implemented using Java Servlets which handle the business logic, and JavaServer Pages (JSPs) which produce the HTML pages sent to an end-user. Since the JSPs are much closer to HTML than Java code, altering the look and feel of DSpace is relatively easy.

To make it even easier, DSpace allows you to 'override' the JSPs included in the source distribution with modified versions, that are stored in a separate place, so when it comes to updating your site with a new DSpace release, your modified versions will not be overwritten.

However, note that the data (attributes) passed from an underlying Servlet to the JSP may change between versions, so you may have to modify your customized Servlet to deal with the new data.

The JSPs are stored in *[dspace-source]*/jsp. Place your edited version of a JSP in the *[dspace-source]*/jsp/local directory, with the same path as the original. If they exist, these will be used in preference to the distributed versions in *[dspace-source]*/jsp. For example:

DSpace default	**Locally-modified version**
[dspace-source]/jsp/community-list.jsp	*[dspace-source]*/jsp/local/community-list.jsp
[dspace-source]/jsp/mydspace/main.jsp	*[dspace-source]*/jsp/local/mydspace/main.jsp

Heavy use is made of a style sheet, in *[dspace-source]*/jsp/styles.css.jsp. If you make edits, call the local version *[dspace-source]*/jsp/local/styles.css.jsp, and it will be used automatically in preference to the default, as described above.

Fonts and colors can be easily changed using the stylesheet. The stylesheet is a JSP so that the user's browser version can be detected and the stylesheet tweaked accordingly.

The 'layout' of each page, that is, the top and bottom banners and the navigation bar, are determined by the JSPs *[dspace-source]*/jsp/layout/header-*.jsp and *[dspace-source]*/jsp/layout/footer-*.jsp. You can provide modified versions of these (in *[dspace-source]*/jsp/local/layout, or define more styles and apply them to pages by using the "style" attribute of the dspace:layout tag.

After you've customized your JSPs**, you must rebuild the DSpace Web application**. If you haven't already built and installed it, follow the install directions. Otherwise, follow the steps below:

1. Rebuild the dspace.war file by running the following command from your *[dspace-source]* directory:

   ```
   ant -Dconfig=[dspace]/config/dspace.cfg build_wars
   ```

2. Shut down Tomcat, and delete the existing *[tomcat]*/webapps/dspace directory.

3. Copy the new .war file to the Tomcat webapps directory:

```
cp [dspace-source]/build/dspace.war [tomcat]/webapps
```

When you restart the web server you should see your customized JSPs.

Custom Authentication Code

Since many institutions and organizations have exisiting authentication systems, DSpace has been designed to allow these to be easily integrated. To do this, you can provide a custom class implementing the Java interface `org.dspace.app.webui.SiteAuthenticator`. These methods are invoked when various authentication-related events occur in the Web user interface.

The basic authentication procedure in the DSpace Web UI is this:

1. A request is received from an end-user's browser that, if fulfilled, would lead to an action requiring authorization taking place.

2. If the end-user is already authenticated:
    - If the end-user is allowed to perform the action, the action proceeds
    - If the end-user is NOT allowed to perform the action, an authorization error is displayed.
3. If the end-user is NOT authenticated, i.e. is accessing DSpace anonymously:
    - The parameters etc. of the request are stored
    - The `startAuthentication` method is invoked on the currently configured `SiteAuthenticator` implementation
    - That `startAuthentication` might instantly authenticate the user somehow, or forward the request to some sort of log-in page--the parameters of the original request are safely stored and will be accessible after the log-in process has completed
    - If authentication is successful, the original request is resumed from step 2. above.

Please see the `SiteAuthenticator.java` source file for information about each of the methods. The default implementation, `org.dspace.app.webui.SimpleAuthenticator`, is a simple implementation that implements these policies:

- Use of inbuilt e-mail address/password-based log-in. This is achieved by forwarding a request that is attempting an action requiring authorization to the password log-in servlet, `/password-login`. The password log-in servlet (`org.dspace.app.webui.servlet.PasswordServlet` contains code that will resume the original request if authentication is successful, as per step 3. described above.
- Users can register themselves (i.e. add themselves as e-people without needing approval from the administrators), and can set their own passwords when they do this
- Users are not members of any special (dynamic) e-person groups

Included in the source is the implementation of `SiteAuthenticator` used at MIT, `edu.mit.dspace.MITAuthenticator`. This implements a slightly more complex authentication mechanism:

- If an authentication user in an MIT user, they must log in using an X509 Web certificate. The `certificate-login` servlet, similar to the `password-login` servlet, authenticates users via these certificates, and if successful, resumes the original request just as the password log-in servlet would.
- MIT users are also automatically added to the special (dynamic) group called 'MIT Users' (which must be present in the system!). This allows us to create authorization policies for MIT users without having to manually maintain membership of the MIT users group.
- Anyone can register themselves, but MIT users doing this cannot set a password; they must use their X509 Web certificate to log in.

The X509 certificate login servlet has an extra feature: If the `webui.cert.autoregister` configuration property is `true`, it will automatically register the user with the system.

You could create a customized version of the password login servlet to perform a similar action. For example, if your organization uses Windows NT domain authentication, you could implement a version of PasswordServlet.java that validates against Windows NT authentication, and automatically adds an e-person record for new users. It is strongly recommended that you do not edit PasswordServlet but create a new servlet for this, so that future updates of the DSpace code do not overwrite your changes. You would also have to implement a customized `SiteAuthenticator` in which the `startAuthentication` method would forward requests to your new servlet.

Displaying Image Thumbnails

Browse and Search Results Page Thumbnails

Image thumbnails can be enabled on the Browse and Search Results pages by setting the appropriate configuration values. To enable the display of thumbnails the following items must be set in the `dspace.cfg` file:

```
webui.browse.thumbnail.show = true
```

If set to `false` or this configuration item is missing then thumbnails will not be shown.

The size of the browse/search thumbnails can also be configured to a smaller size than that generated by the mediafilter. To do this set the following configuration items:

```
webui.browse.thumbnail.maxheight = <maxheight in pixels>

webui.browse.thumbnail.maxwidth = <maxwidth in pixels>
```

If these configuration items are not set, `thumbnail.maxheight` and `thumbnail.maxwidth` are used. Setting these values greater than or equal to the size of the thumbnail generated by the mediafilter (i.e. `thumbnail.maxheight` and `thumbnail.maxwidth`) will have no effect.

Note:

- where the primary bitstream is HTML, no thumbnail is shown;
- where the primary bitstream has a thumbnail, its thumbnail is shown;
- where the primary bitstream is not set, the first thumbnail found by DSpace will be shown;
- where the user does not have read access to the thumbnail bitstream, no thumbnail is shown;
- currently, for a thumbnail to display, a JPEG thumbnail under the current implementation rules must exist (i.e. primary bitstream name with ".jpg" suffix).

Configuring Thumbnail Link Behaviour

The target of a thumbnail in the Browse and Search Results Page can be configured by setting the following configuration item:

```
webui.browse.thumbnail.linkbehaviour = <target page type>
```

Currently the values `item` and `bitstream` are allowed. If this configuration item is not set, or set incorrectly, the default is `item`.

Item Display Page Thumbnails

Thumbnails may also be enabled or disabled on the Item Display page by setting the following configuration item in `dspace.cfg`:

```
webui.item.thumbnail.show = true
```

If set to `false` or this configuration item is missing then thumbnails will not be shown.

On-line Help About File Formats

Because the file format support policy is determined by each individual institution, the on-line help on this subject is intentionally left blank. The help file will, however, retrieve a list of formats and the support levels associated with them in your database and display this information to the user. We highly recommend that you edit the "Format Support Policy" section of the file *`[dspace-source]`*`/jsp/help/formats.jsp`.

Directories and Files

Overview

A complete DSpace installation consists of three separate directory trees:

The source directory:

This is where (surprise!) the source code lives. Note that the config files here are used only during the initial install process. After the install, config files should be changed in the install directory. It is referred to in this document as *[dspace-source]*.

The install directory:

This directory is populated during the install process and also by DSpace as it runs. It contains config files, command-line tools (and the libraries necessary to run them), and usually--although not necessarily--the contents of the DSpace archive (depending on how DSpace is configured). After the initial build and install, changes to config files should be made in this directory. It is referred to in this document as *[dspace]*.

The web deployment directory:

This directory is generated by the web server the first time it finds a dspace.war file in its webapps directory. It contains the unpacked contents of dspace.war, i.e. the JSPs and java classes and libraries necessary to run DSpace. Files in this directory should never be edited directly; if you wish to modify your DSpace installation, you should edit files in the source directory and then rebuild. The contents of this directory aren't listed here since its creation is completely automatic. It is usually referred to in this document as *[tomcat]*/webapps/dspace.

Source Directory Layout

- *[dspace-source]*
  - bin/ - Some shell scripts for running DSpace command-line tasks
  - build.xml - build file for Ant
  - config/ - configuration files
    - dspace.cfg - main DSpace configuration file
    - dc2mods.cfg - Mappings from Dublin Core metadata to MODS for the METS export
    - default.license - default license that users must grant when submitting items
    - mediafilter.cfg - Media Filter configuration
    - news-side.html - Text of the front-page news in the sidebar
    - news-top.html - Text of the front-page news in the top box
    - emails/ - Texts of emails sent out by the system
    - registries/ - INITIAL contents of the bitstream format registry and Dublin Core element/qualifier registry. These are only used on initial system setup, after which they are maintained in the database.
    - templates/ - configuration files for libraries and external applications (e.g. Apache, Tomcat) are kept and edited here. They can refer to properties in the main DSpace configuration - have a look at a couple. When they're updated, a command line tool fills out these files with appropriate values from dspace.cfg, and copies them to their appropriate location (hence "templates".)
  - etc/ - miscelleanous stuff that isn't really to do with system configuration - e.g. the database schema, and a couple of configuration files that are used

during the build process but not by the live system. Also contains the deployment descriptors (`web.xml` files) for the Web UI and OAI-PMH support `.war` files.
    - `jsp/` - The Web UI JSPs. As much as possible, these are simply HTML with little bits of Java - the business code resides in the servlets
    - `lib/` - Library JARs used by the system
        - `README` - Lists the packages third-party libraries (JARs) and their use
        - `licenses` - Contains the licenses associated with the JARs
    - `src/` - DSpace system source code. For details on how this is laid out, see the overview page of the Javadoc.

Installed Directory Layout

Below is the basic layout of a DSpace installation using the default configuration. These paths can be configured if necessary.

- *[dspace]*
    - `assetstore/` - asset store files
    - `bin/` - shell scripts
    - `config/` - configuration
    - `handle-server/` - Handles server files
    - `history/` - history files
    - `lib/` - JARs, including dspace.jar, containing the DSpace classes
    - `log/` - Log files
    - `search/` - Lucene search index files

Log Files

The first source of potential confusion is the log files. Since DSpace uses a number of third-party tools, problems can occur in a variety of places. Below is a table listing the main log files used in a typical DSpace setup. The locations given are defaults, and might be different for your system depending on where you installed DSpace and the third-party tools. The ordering of the list is roughly the recommended order for searching them for the details about a particular problem or error.

DSpace Log File Locations

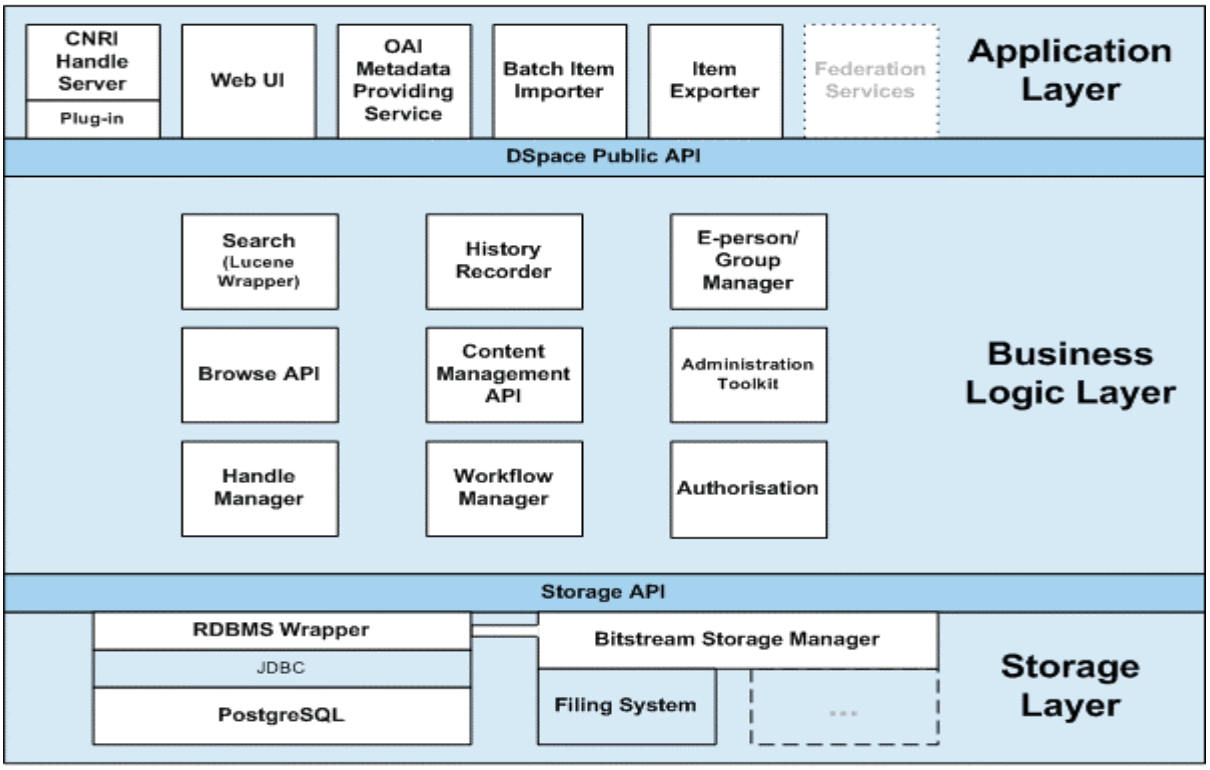
Log File	What's In It
[dspace]/log/dspace.log	Main DSpace log file. This is where the DSpace code writes a simple log of events and errors that occur within the DSpace code. You can control the verbosity of this by editing the *[dspace]*/config/templates/log4j.properties file and then running *[dspace]*/bin/install-configs.
[tomcat]/logs/catalina.out	This is where Tomcat's standard output is written. Many

	errors that occur within the Tomcat code are logged here. For example, if Tomcat can't find the DSpace code (`dspace.jar`), it would be logged in `catalina.out`.
[tomcat]/logs/hostname_log.yyyy-mm-dd.txt	If you're running Tomcat stand-alone (without Apache), it logs some information and errors for specific Web applications to this log file. `hostname` will be your host name (e.g. `dspace.myu.edu`) and `yyyy-mm-dd` will be the date.
[tomcat]/logs/apache_log.yyyy-mm-dd.txt	If you're using Apache, Tomcat logs information about Web applications running through Apache (`mod_webapp`) in this log file (`yyyy-mm-dd` being the date.)
[apache]/error_log	Apache logs to this file. If there is a problem with getting `mod_webapp` working, this is a good place to look for clues. Apache also writes to several other log files, though `error_log` tends to contain the most useful information for tracking down problems.
[dspace]/log/handle-plug.log	The Handle server runs as a separate process from the DSpace Web UI (which runs under Tomcat's JVM). Due to a limitation of log4j's 'rolling file appenders', the DSpace code running in the Handle server's JVM must use a separate log file. The DSpace code that is run as part of a Handle resolution request writes log information to this file. You can control the verbosity of this by editing *[dspace]*/config/templates/log4j-handle-plugin.properties.
[dspace]/log/handle-server.log	This is the log file for CNRI's Handle server code. If a problem occurs within the Handle server code, before DSpace's plug-in is invoked, this is where it may be logged.
[dspace]/handle-server/error.log	On the other hand, a problem with CNRI's Handle server code might be logged here.
PostgreSQL log	PostgreSQL also writes a log file. This one doesn't seem to have a default location, you probably had to specify it yourself at some point during installation. In general, this log file rarely contains pertinent information--PostgreSQL is pretty stable, you're more likely to encounter problems with connecting via JDBC, and these problems will be logged in `dspace.log`.

Architecture

Overview

The DSpace system is organized into three layers, each of which consists of a number of components.



DSpace System Architecture

The storage layer is responsible for physical storage of metadata and content. The business logic layer deals with managing the content of the archive, users of the archive (e-people), authorization, and workflow. The application layer contains components that communicate with the world outside of the individual DSpace installation, for example the Web user interface and the Open Archives Initiative protocol for metadata harvesting service.

Each layer only invokes the layer below it; the application layer may not used the storage layer directly, for example. Each component in the storage and business logic layers has a defined public API. The union of the APIs of those components are referred to as the Storage API (in the case of the storage layer) and the DSpace Public API (in the case of the business logic layer). These APIs are in-process Java classes, objects and methods.

It is important to note that each layer is trusted. Although the logic for authorising actions is in the business logic layer, the system relies on individual applications in the application layer to correctly and securely authenticate e-people. If a 'hostile' or insecure application were allowed to invoke the Public API directly, it could very easily perform actions as any e-person in the system.

The reason for this design choice is that authentication methods will vary widely between different applications, so it makes sense to leave the logic and responsibility for that in these applications.

The source code is organized to cohere very strictly to this three-layer architecture. Also, only methods in a component's public API are given the `public` access level. This means that the Java compiler helps ensure that the source code conforms to the architecture.

Source Code Packages

Packages within	**Correspond to components in**
`org.dspace.app`	Application layer
`org.dspace`	Business logic layer (except `storage` and `app`)
`org.dspace.storage`	Storage layer

The storage and business logic layer APIs are extensively documented with Javadoc-style comments. Generate the HTML version of these by entering the source directory and running:

```
ant public_api
```

The package-level documentation of each package usually contains an overview of the package and some example usage. This information is not repeated in this architecture document; this and the Javadoc APIs are intended to be used in parallel.

Each layer is described in a separate section:

Storage Layer

RDBMS

DSpace uses a relational database to store all information about the organization of content, metadata about the content, information about e-people and authorization, and the state of currently-running workflows. The DSpace system also uses the relational database in order to maintain indices that users can browse.

Most of the functionality that DSpace uses can be offered by any standard SQL database that supports transactions. Presently, the browse indices use some features specific to PostgreSQL, a mature open-source relational database, so some modification to the code would be needed before DSpace would function fully with an alternative database back-end.

The `org.dspace.storage.rdbms` package provides access to an SQL database in a somewhat simpler form than using JDBC directly. The main class is `DatabaseManager`, which executes SQL queries and returns `TableRow` or `TableRowIterator` objects. The `InitializeDatabase` class is used to load SQL into the database via JDBC, for example to set up the schema.

All calls to the `Database Manager` require a DSpace `Context` object. Example use of the database manager API is given in the `org.dspace.storage.rdbms` package Javadoc.

The database schema used by DSpace is stored in *`[dspace-source]`*`/etc/database_schema.sql` in the source distribution. It is stored in the form of SQL that can be fed straight into the DBMS to construct the database. The schema SQL file also directly creates two e-person groups in the database that are required for the system to function properly.

The DSpace database code uses an SQL function `getnextid` to assign primary keys to newly created rows. This SQL function must be safe to use if several JVMs are accessing the database at once; for example, the Web UI might be creating new rows in the database at the same time as the batch item importer. The PostgreSQL-specific implementation of the method uses `SEQUENCES` for each table in order to create new IDs. If an alternative database backend were to be used, the implementation of `getnextid` could be updated to operate with that specific DBMS.

The `etc` directory in the source distribution contains two further SQL files. `clean-database.sql` contains the SQL necessary to completely clean out the database, so use with caution! The Ant target `clean_database` can be used to execute this. `update-sequences.sql` contains SQL to reset the primary key generation sequences to appropriate values. You'd need to do this if, for example, you're restoring a backup database dump which creates rows with specific primary keys already defined. In such a case, the sequences would allocate primary keys that were already used.

Maintenance and Backup

When using PostgreSQL, it's a good idea to perform regular 'vacuuming' of the database to optimize performance. This is performed by the `vacuumdb` command which can be executed via a 'cron' job, for example by putting this in the system `crontab`:

```
# clean up the database nightly
40 2 * * * /usr/local/pgsql/bin/vacuumdb --analyze dspace > /dev/null 2>&1
```

The DSpace database can be backed up and restored using usual methods, for example with `pg_dump` and `psql`. However when restoring a database, you will need to perform these additional steps:

- The `fresh_install` target loads up the initial contents of the Dublin Core type and bitstream format registries, as well as two entries in the `epersongroup` table for the system anonymous and administrator groups. Before you restore a raw backup of your database you will need to remove these, since they will already exist in your backup, possibly having been modified. For example, use:

  ```
  DELETE FROM dctyperegistry;
  DELETE FROM bitstreamformatregistry;
  DELETE FROM epersongroup;
  ```

- After restoring a backup, you will need to reset the primary key generation sequences so that they do not produce already-used primary keys. Do this by executing the SQL in *`[dspace-source]`*`/etc/update-sequences.sql`, for example with:

  ```
  psql -U dspace -f [dspace-source]/etc/update-sequences.sql
  ```

Future updates of DSpace may involve minor changes to the database schema. Specific instructions on how to update the schema whilst keeping live data will be included. The current schema also contains a few currently unused database columns, to be used for extra functionality in future releases. These unused columns have been added in advance to minimize the effort required to upgrade.

Configuring the RDBMS Component

The database manager is configured with the following properties in `dspace.cfg`:

`db.url`	The JDBC URL to use for accessing the database. This should not point to a connection pool, since DSpace already implements a connection pool.
`db.driver`	JDBC driver class name. Since presently, DSpace uses PostgreSQL-specific features, this should be `org.postgresql.Driver`.
`db.username`	Username to use when accessing the database.
`db.password`	Corresponding password ot use when accessing the database.

Bitstream Store

DSpace currently simply stores content in the file system on the server. This is achieved using a simple, lightweight API.

The `BitstreamStorageManager` provides low-level access to bitstreams stored in the system. In general, it should not be used directly; instead, use the `Bitstream` object in the content management API since that encapsulated authorization and other metadata to do with a bitstream that are not maintained by the `BitstreamStorageManager`.

The bitstream storage manager provides three methods that store, retrieve and delete bitstreams. Bitstreams are referred to by their 'ID'; that is the primary key `bitstream_id` column of the corresponding row in the database.

As of DSpace version 1.1, there can be multiple bitstream stores. This means that the potential storage of a DSpace system is not bound by the maximum size of a single disk or file system.

Stores are numbered, starting with zero, then counting upwards. Each bitstream entry in the database has a store number, used to retrieve the bitstream when required.

At the moment, the store in which new bitstreams are placed is decided using a configuration parameter, and there is no provision for moving bitstreams between stores. Administrative tools for manipulating bitstreams and stores will be provided in future releases. Right now you can move a whole store (e.g. you could move store number 1 from `/localdisk/store` to `/fs/anotherdisk/store` but it would still have to be store number 1 and have the exact same contents.

Bitstreams also have an 38-digit internal ID, different from the primary key ID of the bitstream table row. This is not visible or used outside of the bitstream storage manager. It is used to determine the exact location (relative to the relevant store directory) that the bitstream is stored in the file system. The first three pairs of digits are the directory path that the bitstream is stored under. The bitstream is stored in a file with the internal ID as the filename.

For example, a bitstream with the internal ID `12345678901234567890123456789012345678` is stored in the directory:

```
(assetstore dir)/12/34/56/12345678901234567890123456789012345678
```

The reasons for storing files this way are:

- Using a randomly-generated 38-digit number means that the 'number space' is less cluttered than simply using the primary keys, which are allocated sequentially and are thus close together. This means that the bitstreams in the store are distributed around the directory structure, improving access efficiency.
- The internal ID is used as the filename partly to avoid requiring an extra lookup of the filename of the bitstream, and partly because bitstreams may be received from a variety of operating systems. The original name of a bitstream may be an illegal UNIX filename.

When storing a bitstream, the `BitstreamStorageManager` DOES set the following fields in the corresponding database table row:

- `bitstream_id`
- `size`
- `checksum`
- `checksum_algorithm`
- `internal_id`
- `deleted`
- `store_number`

The remaining fields are the responsibility of the `Bitstream` content management API class.

The bitstream storage manager is fully transaction-safe. In order to implement transaction-safety, the following algorithm is used to store bitstreams:

1. A database connection is created, separately from the currently active connection in the current DSpace context.

2. An unique internal identifier (separate from the database primary key) is generated.

3. The bitstream DB table row is created using this new connection, with the `deleted` column set to `true`.

4. The new connection is `commit`ted, so the 'deleted' bitstream row is written to the database.

5. The bitstream itself is stored in a file in the configured 'asset store directory', with a directory path and filename derived from the internal ID.

6. The `deleted` flag in the bitstream row is set to `false`. This will occur (or not) as part of the current DSpace `Context`.

This means that should anything go wrong before, during or after the bitstream storage, only one of the following can be true:

- No bitstream table row was created, and no file was stored
- A bitstream table row with `deleted=true` was created, no file was stored
- A bitstream table row with `deleted=true` was created, and a file was stored

None of these affect the integrity of the data in the database or bitstream store.

Similarly, when a bitstream is deleted for some reason, its `deleted` flag is set to true as part of the overall transaction, and the corresponding file in the filesystem is not deleted.

The above techniques mean that the bitstream storage manager is transaction-safe. Over time, the bitstream database table and file store may contain a number of 'deleted' bitstreams. The `cleanup` method of `BitstreamStorageManager` goes through these deleted rows, and actually deletes them along with any corresponding files left in the file system. It

only removes 'deleted' bitstreams that are more than one hour old, just in case cleanup is happening in the middle of a storage operation.

This cleanup can be invoked from the command line via the `Cleanup` class, which can in turn be easily executed from a shell on the server machine using `/dspace/bin/cleanup`. You might like to have this run regularly by `cron`, though since DSpace is read-lots, write-not-so-much it doesn't need to be run very often.

Backup

The bitstreams (files) in the bitstream store may be backed up very easily by simply 'tarring' or 'zipping' the `assetstore` directory (or whichever directory is configured in `dspace.cfg`. Restoring is as simple as extracting the backed-up compressed file in the appropriate location.

It is important to note that since the bitstream storage manager holds the bitstreams in the file system, and information about them in the database, that a database backup and a backup of the files in the bitstream store must be made at the same time; the bitstream data in the database must correspond to the stored files.

Of course, it isn't really ideal to 'freeze' the system while backing up to ensure that the database and files match up. Since DSpace uses the bitstream data in the database as the authoritative record, it's best to back up the database before the files. This is because it's better to have a bitstream in the file system but not the database (effectively non-existent to DSpace) than a bitstream record in the database but not the file system, since people would be able to find the bitstream but not actually get the contents.

Configuring the Bitstream Store

The bitstream stores (also called 'asset stores') can be configured in `dspace.cfg`. For example:

```
assetstore.dir = [dspace]/assetstore
```

(Remember that *[dspace]* is a placeholder for the actual name of your DSpace install directory).

The above example specifies a single asset store.

```
assetstore.dir = [dspace]/assetstore_0
assetstore.dir.1 = /mnt/other_filesystem/assetstore_1
```

The above example specifies two asset stores. assetstore.dir specifies the asset store number 0 (zero); after that use assetstore.dir.1, assetstore.dir.2 and so on. The particular asset store a bitstream is stored in is held in the database, so don't move bitstreams between asset stores, and don't renumber them.

By default, newly created bitstreams are put in asset store 0 (i.e. the one specified by the assetstore.dir property.) This allows backwards compatibility with pre-DSpace 1.1

configurations. To change this, for example when asset store 0 is getting full, add a line to `dspace.cfg` like:

```
assetstore.incoming = 1
```

Then restart DSpace (Tomcat). New bitstreams will be written to the asset store specified by `assetstore.dir.1`, which is `/mnt/other_filesystem/assetstore_1` in the above example.

Business Logic Layer

Core Classes

The `org.dspace.core` package provides some basic classes that are used throughout the DSpace code.

The Configuration Manager (`ConfigurationManager`)

The configuration manager is responsible for reading the main `dspace.cfg` properties file, managing the 'template' configuration files for other applications such as Apache, and for obtaining the text for e-mail messages.

The system is configured by editing the relevant files in `/dspace/config`, as described in the configuration section.

When editing configuration files for applications that DSpace uses, such as Apache, remember to edit the file in `/dspace/config/templates` and then run `/dspace/bin/install-configs` rather than editing the 'live' version directly!

The `ConfigurationManager` class can also be invoked as a command line tool, with two possible uses:

- `/dspace/bin/install-configs`

  This processes and installs configuration files for other applications, as described in the configuration section.

- `/dspace/bin/dsrun org.dspace.core.ConfigurationManager -property property.name`

  This writes the value of property.name from `dspace.cfg` to the standard output, so that shell scripts can access the DSpace configuration. For an example, see `/dspace/bin/start-handle-server`. If the property has no value, nothing is written.

Constants

This class contains constants that are used to represent types of object and actions in the database. For example, authorization policies can relate to objects of different types, so the

`resourcepolicy` table has columns `resource_id`, which is the internal ID of the object, and `resource_type_id`, which indicates whether the object is an item, collection, bitstream etc. The value of `resource_type_id` is taken from the `Constants` class, for example `Constants.ITEM`.

Context

The `Context` class is central to the DSpace operation. Any code that wishes to use the any API in the business logic layer must first create itself a `Context` object. This is akin to opening a connection to a database (which is in fact one of the things that happens.)

A context object is involved in most method calls and object constructors, so that the method or object has access to information about the current operation. When the context object is constructed, the following information is automatically initialized:

- A connection to the database. This is a transaction-safe connection. i.e. the 'auto-commit' flag is set to false.
- A cache of content management API objects. Each time a content object is created (for example `Item` or `Bitstream`) it is stored in the `Context` object. If the object is then requested again, the cached copy is used. Apart from reducing database use, this addresses the problem of having two copies of the same object in memory in different states.

The following information is also held in a context object, though it is the responsiblity of the application creating the context object to fill it out correctly:

- The current authenticated user, if any
- Any 'special groups' the user is a member of. For example, a user might automatically be part of a particular group based on the IP address they are accessing DSpace from, even though they don't have an e-person record. Such a group is called a 'special group'.
- Any extra information from the application layer that should be added to log messages that are written within this context. For example, the Web UI adds a session ID, so that when the logs are analysed the actions of a particular user in a particular session can be tracked.
- A flag indicating whether authorization should be circumvented. This should only be used in rare, specific circumstances. For example, when first installing the system, there are no authorized administrators who would be able to create an administrator account!

  As noted above, the public API is trusted, so it is up to applications in the application layer to use this flag responsibly.

Typical use of the context object will involve constructing one, and setting the current user if one is authenticated. Several operations may be performed using the context object. If all goes well, `complete` is called to commit the changes and free up any resources used by the context. If anything has gone wrong, `abort` is called to roll back any changes and free up the resources.

You should always `abort` a context if any error happens during its lifespan; otherwise the data in the system may be left in an inconsistent state. You can also `commit` a context, which means that any changes are written to the database, and the context is kept active for further use.

Email

Sending e-mails is pretty easy. Just use the configuration manager's `getEmail` method, set the arguments and recipients, and send.

The e-mail texts are stored in `/dspace/config/emails`. They are processed by the standard `java.text.MessageFormat`. At the top of each e-mail are listed the appropriate arguments that should be filled out by the sender. Example usage is shown in the `org.dspace.core.Email` Javadoc API documentation.

LogManager

The log manager consists of a method that creates a standard log header, and returns it as a string suitable for logging. Note that this class does not actually write anything to the logs; the log header returned should be logged directly by the sender using an appropriate Log4J call, so that information about where the logging is taking place is also stored.

The level of logging can be configured on a per-package or per-class basis by editing `/dspace/config/templates/log4j.properties` and then executing `/dspace/bin/install-configs`. You will need to stop and restart Tomcat for the changes to take effect.

A typical log entry looks like this:

```
2002-11-11 08:11:32,903 INFO org.dspace.app.webui.servlet.DSpaceServlet @
anonymous:session_id=BD84E7C194C2CF4BD0EC3A6CAD0142BB:view_item:handle=1721.1/
1686
```

This is breaks down like this:

Date and time, milliseconds	`2002-11-11 08:11:32,903`
Level (`FATAL`, `WARN`, `INFO` or `DEBUG`)	`INFO`
Java class	`org.dspace.app.webui.servlet.DSpaceServlet`
	`@`
User email or `anonymous`	`anonymous`
	`:`
Extra log info from context	`session_id=BD84E7C194C2CF4BD0EC3A6CAD0142BB`
	`:`
Action	`view_item`

:

Extra info handle=1721.1/1686

The above format allows the logs to be easily parsed and analysed. The `/dspace/bin/log-reporter` script is a simple tool for analysing logs. Try:

```
/dspace/bin/log-reporter --help
```

It's a good idea to 'nice' this log reporter to avoid an impact on server performance.

Utils

`Utils` comtains miscellaneous utility method that are required in a variety of places throughout the code, and thus have no particular 'home' in a subsystem.

Content Management API

The content management API package `org.dspace.content` contains Java classes for reading and manipulating content stored in the DSpace system. This is the API that components in the application layer will probably use most.

Classes corresponding to the main elements in the DSpace data model (`Community`, `Collection`, `Item`, `Bundle` and `Bitstream`) are sub-classes of the abstract class `DSpaceObject`. The `Item` object handles the Dublin Core metadata record.

Each class generally has one or more static `find` methods, which are used to instantiate content objects. Constructors do not have public access and are just used internally. The reasons for this are:

- "Constructing" an object may be misconstrued as the action of creating an object in the DSpace system, for example one might expect something like:

  ```
  Context dsContent = new Context();
  Item myItem = new Item(context, id)
  ```

  to construct a brand new item in the system, rather than simply instantiating an in-memory instance of an object in the system.

- `find` methods may often be called with invalid IDs, and return `null` in such a case. A constructor would have to throw an exception in this case. A `null` return value from a static method can in general be dealt with more simply in code.
- If an instantiation representing the same underlying archival entity already exists, the `find` method can simply return that same instantiation to avoid multiple copies and any inconsistencies which might result.

`Collection`, `Bundle` and `Bitstream` do not have `create` methods; rather, one has to create an object using the relevant method on the container. For example, to create a collection, one must invoke `createCollection` on the community that the collection is to appear in:

```
Context context = new Context();
Community existingCommunity = Community.find(context, 123);
Collection myNewCollection = existingCommunity.createCollection();
```

The primary reason for this is for determining authorization. In order to know whether an e-person may create an object, the system must know which container the object is to be added to. It makes no sense to create a collection outside of a community, and the authorization system does not have a policy for that.

`Item`s are first created in the form of an implementation of `InProgressSubmission`. An `InProgressSubmission` represents an item under construction; once it is complete, it is installed into the main archive and added to the relevant collection by the `InstallItem` class. The `org.dspace.content` package provides an implementation of `InProgressSubmission` called `WorkspaceItem`; this is a simple implementation that contains some fields used by the Web submission UI. The `org.dspace.workflow` also contains an implementation called `WorkflowItem` which represents a submission undergoing a workflow process.

In the previous chapter there is an overview of the item ingest process which should clarify the previous paragraph. Also see the section on the workflow system.

`Community` and `BitstreamFormat` do have static `create` methods; one must be a site administrator to have authorization to invoke these.

Other Classes

Classes whose name begins `DC` are for manipulating Dublin Core metadata, as explained below.

The `FormatIdentifier` class attempts to guess the bitstream format of a particular bitstream. Presently, it does this simply by looking at any file extension in the bitstream name and matching it up with the file extensions associated with bitstream formats. Hopefully this can be greatly improved in the future!

The `ItemIterator` class allows items to be retrieved from storage one at a time, and is returned by methods that may return a large number of items, more than would be desirable to have in memory at once.

The `ItemComparator` class is an implementation of the standard `java.util.Comparator` that can be used to compare and order items based on a particular Dublin Core metadata field.

Modifications

When creating, modifying or for whatever reason removing data with the content management API, it is important to know when changes happen in-memory, and when they occur in the physical DSpace storage.

Primarily, one should note that no change made using a particular `org.dspace.core.Context` object will actually be made in the underlying storage unless `complete` or `commit` is invoked on that `Context`. If anything should go wrong during an operation, the context should always be aborted by invoking `abort`, to ensure that no inconsistent state is written to the storage.

Additionally, some changes made to objects only happen in-memory. In these cases, invoking the `update` method lines up the in-memory changes to occur in storage when the `Context` is committed or completed. In general, methods that change any [meta]data field only make the change in-memory; methods that involve relationships with other objects in the system line up the changes to be committed with the context. See individual methods in the API Javadoc.

Some examples to illustrate this are shown below:

```
Context context = new Context();
Bitstream b =
Bitstream.find(context, 1234);
b.setName("newfile.txt");
b.update();
context.complete();
```

Will change storage

```
Context context = new Context();
Bitstream b =
Bitstream.find(context, 1234);
b.setName("newfile.txt");
b.update();
context.abort();
```

Will not change storage (context aborted)

```
Context context = new Context();
Bitstream b =
Bitstream.find(context, 1234);
b.setName("newfile.txt");
context.complete();
```

The new name **will not** be stored since `update` was not invoked

```
Context context = new Context();
Bitstream bs =
Bitstream.find(context, 1234);
Bundle bnd = Bundle.find(context,
5678);
bnd.add(bs);
context.complete();
```

The bitstream **will** be included in the bundle, since `update` doesn't need to be called

What's In Memory?

Instantiating some content objects also causes other content objects to be loaded into memory.

Instantiating a `Bitstream` object causes the appropriate `BitstreamFormat` object to be instantiated. Of course the `Bitstream` object does not load the underlying bits from the bitstream store into memory!

Instantiating a `Bundle` object causes the appropriate `Bitstream` objects (and hence `BitstreamFormat`s) to be instantiated.

Instantiating an `Item` object causes the appropriate `Bundle` objects (etc.) and hence `BitstreamFormat`s to be instantiated. All the Dublin Core metadata associated with that item are also loaded into memory.

The reasoning behind this is that for the vast majority of cases, anyone instantiating an item object is going to need information about the bundles and bitstreams within it, and this methodology allows that to be done in the most efficient way and is simple for the caller. For example, in the Web UI, the servlet (controller) needs to pass information about an item to the viewer (JSP), which needs to have all the information in-memory to display the item without further accesses to the database which may cause errors mid-display.

You do not need to worry about multiple in-memory instantiations of the same object, or any inconsistenties that may result; the `Context` object keeps a cache of the instantiated objects. The `find` methods of classes in `org.dspace.content` will use a cached object if one exists.

It may be that in enough cases this automatic instantiation of contained objects reduces performance in situations where it is important; if this proves to be true the API may be changed in the future to include a `loadContents` method or somesuch, or perhaps a Boolean parameter indicating what to do will be added to the `find` methods.

When a `Context` object is completed, aborted or garbage-collected, any objects instantiated using that context are invalidated and should not be used (in much the same way an AWT button is invalid if the window containing it is destroyed).

Dublin Core Metadata

The `DCValue` class is a simple container that represents a single Dublin Core element, optional qualifier, value and language. The other classes starting with `DC` are utility classes for handling types of data in Dublin Core, such as people's names and dates. As supplied, the DSpace registry of elements and qualifiers corresponds to the Library Application Profile for Dublin Core. It should be noted that these utility classes assume that the values will be in a certain syntax, which will be true for all data generated within the DSpace system, but since Dublin Core does not always define strict syntax, this may not be true for Dublin Core originating outside DSpace.

Below is the specific syntax that DSpace expects various fields to adhere to:

Element	Qualifier	Syntax	Helper Class

`date`	Any or unqualified	ISO 8601 in the UTC time zone, with either year, month, day, or second precision. Examples:
		
		`2000`
		`2002-10`
		`2002-08-14`
		`1999-01-01T14:35:23Z`
			`DCDate`

contributor	Any or unqualified	In general last name, then a comma, then first names, then any additional information like "Jr.". If the contributor is an organization, then simply the name. Examples: Doe, John Smith, John Jr. van Dyke, Dick Massachusetts Institute of Technology	DCPersonName
language	iso	A two letter code taken ISO 639, followed optionally by a two letter country code taken from ISO 3166. Examples: en fr en_US	DCLanguage
relation	ispartofseries	The series name, following by a semicolon followed by the number in that series. Alternatively, just free text. MIT-TR; 1234 My Report Series; ABC-1234 NS1234	DCSeriesNumber

Workflow System

The primary classes are:

org.dspace.content.WorkspaceItem	contains an Item before it enters a workflow
org.dspace.workflow.WorkflowItem	contains an Item while in a workflow
org.dspace.workflow.WorkflowManager	responds to events, manages the WorkflowItem states
org.dspace.content.Collection	contains List of defined workflow steps
org.dspace.eperson.Group	people who can perform workflow tasks are defined in EPerson Groups
org.dspace.core.Email	used to email messages to Group members and submitters

The workflow system models the states of an Item in a state machine with 5 states (SUBMIT, STEP_1, STEP_2, STEP_3, ARCHIVE.) These are the three optional steps where the item can be viewed and corrected by different groups of people. Actually, it's more like 8 states, with STEP_1_POOL, STEP_2_POOL, and STEP_3_POOL. These pooled states are when items are waiting to enter the primary states.

The WorkflowManager is invoked by events. While an Item is being submitted, it is held by a WorkspaceItem. Calling the start() method in the WorkflowManager converts a WorkspaceItem to a WorkflowItem, and begins processing the WorkflowItem's state. Since all three steps of the workflow are optional, if no steps are defined, then the Item is simply archived.

Workflows are set per Collection, and steps are defined by creating corresponding entries in the List named workflowGroup. If you wish the workflow to have a step 1, use the administration tools for Collections to create a workflow Group with members who you want to be able to view and approve the Item, and the workflowGroup[0] becomes set with the ID of that Group.

If a step is defined in a Collection's workflow, then the WorkflowItem's state is set to that step_POOL. This pooled state is the WorkflowItem waiting for an EPerson in that group to claim the step's task for that WorkflowItem. The WorkflowManager emails the members of that Group notifying them that there is a task to be performed (the text is defined in config/emails,) and when an EPerson goes to their 'My DSpace' page to claim the task, the WorkflowManager is invoked with a claim event, and the WorkflowItem's state advances from STEP_x_POOL to STEP_x (where x is the corresponding step.) The EPerson can also generate an 'unclaim' event, returning the WorkflowItem to the STEP_x_POOL.

Other events the WorkflowManager handles are advance(), which advances the WorkflowItem to the next state. If there are no further states, then the WorkflowItem is removed, and the Item is then archived. An EPerson performing one of the tasks can reject the Item, which stops the workflow, rebuilds the WorkspaceItem for it and sends a rejection note to the submitter. More drastically, an abort() event is generated by the admin tools to cancel a workflow outright.

Administration Toolkit

The `org.dspace.administer` package contains some classes for administering a DSpace system that are not generally needed by most applications.

The `CreateAdministrator` class is a simple command-line tool, executed via `/dspace/bin/create-administrator`, that creates an administrator e-person with information entered from standard input. This is generally used only once when a DSpace system is initially installed, to create an initial administrator who can then use the Web administration UI to further set up the system. This script does not check for authorization, since it is typically run before there are any e-people to authorize! Since it must be run as a command-line tool on the server machine, generally this shouldn't cause a problem. A possibility is to have the script only operate when there are no e-people in the system already, though in general, someone with access to command-line scripts on your server is probably in a position to do what they want anyway!

The `DCType` class is similar to the `org.dspace.content.BitstreamFormat` class. It represents an entry in the Dublin Core type registry, that is, a particular element and qualifier, or unqualified element. It is in the `administer` package because it is only generally required when manipulating the registry itself. Elements and qualifiers are specified as literals in `org.dspace.content.Item` methods and the `org.dspace.content.DCValue` class. Only administrators may modify the Dublin Core type registry.

The `org.dspace.administer.RegistryLoader` class contains methods for initialising the Dublin Core type registry and bitstream format registry with entries in an XML file. Typically this is executed via the command line during the build process (see `build.xml` in the source.) To see examples of the XML formats, see the files in `config/registries` in the source directory. There is no XML schema, they aren't validated strictly when loaded in.

E-person/Group Manager

DSpace keeps track of registered users with the `org.dspace.eperson.EPerson` class. The class has methods to create and manipulate an `EPerson` such as get and set methods for first and last names, email, and password. (Actually, there is no `getPassword()` method--an MD5 hash of the password is stored, and can only be verified with the `checkPassword()` method.) There are find methods to find an EPerson by email (which is assumed to be unique,) or to find all EPeople in the system.

The `EPerson` object should probably be reworked to allow for easy expansion; the current EPerson object tracks pretty much only what MIT was interested in tracking - first and last names, email, phone. The access methods are hardcoded and should probably be replaced with methods to access arbitrary name/value pairs for institutions that wish to customize what EPerson information is stored.

Groups are simply lists of `EPerson` objects. Other than membership, `Group` objects have only one other attribute: a name. Group names must be unique, so we have adopted naming conventions where the role of the group is its name, such as `COLLECTION_100_ADD`. Groups add and remove EPerson objects with `addMember()` and `removeMember()` methods. One important thing to know about groups is that they store their membership in memory until the `update()` method is called - so when modifying a group's membership don't forget to invoke `update()` or your changes will be lost! Since group membership is used heavily by the authorization system a fast `isMember()` method is also provided.

Another kind of Group is also implemented in DSpace--special Groups. The `Context` object for each session carries around a List of Group IDs that the user is also a member of--currently the MITUser Group ID is added to the list of a user's special groups if certain IP address or certificate criteria are met.

Authorization

The primary classes are:

`org.dspace.authorize.AuthorizeManager`	does all authorization, checking policies against Groups
`org.dspace.authorize.ResourcePolicy`	defines all allowable actions for an object
`org.dspace.eperson.Group`	all policies are defined in terms of EPerson Groups

The authorization system is based on the classic 'police state' model of security; no action is allowed unless it is expressed in a policy. The policies are attached to resources (hence the

name `ResourcePolicy`,) and detail who can perform that action. The resource can be any of the DSpace object types, listed in `org.dspace.core.Constants` (`BITSTREAM`, `ITEM`, `COLLECTION`, etc.) The 'who' is made up of EPerson groups. The actions are also in `Constants.java` (`READ`, `WRITE`, `ADD`, etc.) The only non-obvious actions are `ADD` and `REMOVE`, which are authorizations for container objects. To be able to create an Item, you must have `ADD` permission in a Collection, which contains Items. (Communities, Collections, Items, and Bundles are all container objects.)

Currently most of the read policy checking is done with items--communities and collections are assumed to be openly readable, but items and their bitstreams are checked. Separate policy checks for items and their bitstreams enables policies that allow publicly readable items, but parts of their content may be restricted to certain groups.

The `AuthorizeManager` class' `authorizeAction(Context, object, action)` is the primary source of all authorization in the system. It gets a list of all of the ResourcePolicies in the system that match the object and action. It then iterates through the policies, extracting the EPerson Group from each policy, and checks to see if the EPersonID from the Context is a member of any of those groups. If all of the policies are queried and no permission is found, then an `AuthorizeException` is thrown. An `authorizeAction()` method is also supplied that returns a boolean for applications that require higher performance.

ResourcePolicies are very simple, and there are quite a lot of them. Each can only list a single group, a single action, and a single object. So each object will likely have several policies, and if multiple groups share permissions for actions on an object, each group will get its own policy. (It's a good thing they're small.)

Special Groups

All users are assumed to be part of the public group (ID=0.) DSpace admins (ID=1) are automatically part of all groups, much like super-users in the Unix OS. The Context object also carries around a List of special groups, which are also first checked for membership. These special groups are used at MIT to indicate membership in the MIT community, something that is very difficult to enumerate in the database! When a user logs in with an MIT certificate or with an MIT IP address, the login code adds this MIT user group to the user's Context.

Miscellaneous Authorization Notes

Where do items get their read policies? From the their collection's read policy. There once was a separate item read default policy in each collection, and perhaps there will be again since it appears that administrators are notoriously bad at defining collection's read policies. There is also code in place to enable policies that are timed--have a start and end date. However, the admin tools to enable these sorts of policies have not been written.

Handle Manager/Handle Plugin

The `org.dspace.handle` package contains two classes; `HandleManager` is used to create and look up Handles, and `HandlePlugin` is used to expose and resolve DSpace Handles for the outside world via the CNRI Handle Server code.

Handles are stored internally in the `handle` database table in the form:

```
1721.123/4567
```

Typically when they are used outside of the system they are displayed in either URI or "URL proxy" forms:

```
hdl:1721.123/4567
http://hdl.handle.net/1721.123/4567
```

It is the responsibility of the caller to extract the basic form from whichever displayed form is used.

The `handle` table maps these Handles to resource type/resource ID pairs, where resource type is a value from `org.dspace.core.Constants` and resource ID is the internal identifier (database primary key) of the object. This allows Handles to be assigned to any type of object in the system, though as explained in the functional overview, only communities, collections and items are presently assigned Handles.

`HandleManager` contains static methods for:

- Creating a Handle
- Finding the Handle for a `DSpaceObject`, though this is usually only invoked by the object itself, since `DSpaceObject` has a `getHandle` method
- Retrieving the `DSpaceObject` identified by a particular Handle
- Obtaining displayable forms of the Handle (URI or "proxy URL").

`HandlePlugin` is a simple implementation of the Handle Server's `net.handle.hdllib.HandleStorage` interface. It only implements the basic Handle retrieval methods, which get information from the `handle` database table. The CNRI Handle Server is configured to use this plug-in via its `config.dct` file.

Note that since the Handle server runs as a separate JVM to the DSpace Web applications, it uses a separate 'Log4J' configuration, since Log4J does not support multiple JVMs using the same daily rolling logs. This alternative configuration is held as a template in `/dspace/config/templates/log4j-handle-plugin.properties`, written to `/dspace/config/log4j-handle-plugin.properties` by the `install-configs` script. The `/dspace/bin/start-handle-server` script passes in the appropriate command line parameters so that the Handle server uses this configuration.

Search

DSpace's search code is a simple API which currently wraps the Lucene search engine. The first half of the search task is indexing, and `org.dspace.search.DSIndexer` is the indexing class, which contains `indexContent()` which if passed an `Item`, `Community`, or `Collection`, will add that content's fields to the index. The methods `unIndexContent()` and `reIndexContent()` remove and update content's index information. The `DSIndexer` class also has a `main()` method which will rebuild the index completely. This is invoked by the `dspace/bin/index-all` script. The intent was for the `main()` method to be invoked on a

regular basis to avoid index corruption, but we have had no problem with that so far. Which fields are indexed by `DSIndexer`? These fields are currently hardcoded in `indexItemContent()` `indexCollectionContent()` and `indexCommunityContent()`/ methods.

The query class `DSQuery` contains the three flavors of `doQuery()` methods--one searches the DSpace site, and the other two restrict searches to Collections and Communities. The results from a query are returned as three lists of handles; each list represents a type of result. One list is a list of Items with matches, and the other two are Collections and Communities that match. This separation allows the UI to handle the types of results gracefully without resolving all of the handles first to see what kind of content the handle points to. The `DSQuery` class also has a `main()` method for debugging via command-line searches.

Our Lucene Implementation

Currently we have our own Analyzer and Tokenizer classes (`DSAnalyzer` and `DSTokenizer`) to customize our indexing. They invoke the stemming and stop word features within Lucene. We create an `IndexReader` for each query, which we now realize isn't the most efficient use of resources - we seem to run out of filehandles on really heavy loads. (A wildcard query can open many filehandles!) Since Lucene is thread-safe, a better future implementation would be to have a single Lucene IndexReader shared by all queries, and then is invalidated and re-opened when the index changes. Future API growth could include relevance scores (Lucene generates them, but we ignore them,) and abstractions for more advanced search concepts such as booleans.

Indexed Fields

The `DSIndexer` class shipped with DSpace indexes the Dublin Core metadata in the following way:

Search Field	Taken from Dublin Core Fields

Authors	`contributor.*`
	`creator.*`
	`description.statementofresponsibility`

Titles	`title.*`
Keywords	`subject.*`
Abstracts	`description.abstract` `description.tableofcontents`
Series	`relation.ispartofseries`
MIME types	`format.mimetype`
Sponsors	`description.sponsorship`
Identifiers	`identifier.*`

Harvesting API

The `org.dspace.search` package also provides a 'harvesting' API. This allows callers to extract information about items modified within a particular timeframe, and within a particular scope (all of DSpace, or a community or collection.) Currently this is used by the Open Archives Initiative metadata harvesting protocol application, and the e-mail subscription code.

The `Harvest.harvest` is invoked with the required scope and start and end dates. Either date can be omitted. The dates should be in the ISO8601, UTC time zone format used elsewhere in the DSpace system.

`HarvestedItemInfo` objects are returned. These objects are simple containers with basic information about the items falling within the given scope and date range. Depending on parameters passed to the `harvest` method, the `containers` and `item` fields may have been filled out with the IDs of communities and collections containing an item, and the corresponding `Item` object respectively. Electing not to have these fields filled out means the harvest operation executes considerable faster.

In case it is required, `Harvest` also offers a method for creating a single `HarvestedItemInfo` object, which might make things easier for the caller.

Browse API

The browse API maintains indices of dates, authors and titles, and allows callers to extract parts of these:

Title

Values of the Dublin Core lement **`title`** (unqualified) are indexed. These are sorted in a case-insensitive fashion, with any leading article removed. For example:

```
The DSpace System
```

Appears under 'D' rather than 'T'.

Author

Values of the **`contributor`** (any qualifier or unqualified) element are indexed. Since `contributor` values typically are in the form 'last name, first name', a simple case-insensitive alphanumeric sort is used which orders authors in last name order.

Note that this is an index of authors, and not items by author. If four items have the same author, that author will appear in the index only once. Hence, the index of authors may be greater or smaller than the index of titles; items often have more than one author, though the same author may have authored several items.

The author indexing in the browse API does have limitations:

- Ideally, a name that appears as an author for more than one item would appear in the author index only once. For example, 'Doe, John' may be the author of tens of items. However, in practice, author's names often appear in slightly differently forms, for example:

  ```
  Doe, John
  Doe, John Stewart
  Doe, John S.
  ```

  Currently, the above three names would all appear as separate entries in the author index even though they may refer to the same author. In order for an author of several papers to be correctly appear once in the index, each item must specify exactly the same form of their name, which doesn't always happen in practice.

- Another issue is that two authors may have the same name, even within a single institution. If this is the case they may appear as one author in the index.

These issues are typically resolved in libraries with authority control records, in which are kept a 'preferred' form of the author's name, with extra information (such as date of birth/death) in order to distinguish between authors of the same name. Maintaining such records is a huge task with many issues, particularly when metadata is received from faculty directly rather than trained library cataloguers. For these reasons, DSpace does not yet feature 'authority control' functionality.

Date of Issue

Items are indexed by date of issue. This may be different from the date that an item appeared in DSpace; many items may have been originally published elsewhere beforehand. The Dublin Core field used is **`date.issued`**. The ordering of this index may be reversed so 'earliest first' and 'most recent first' orderings are possible.

Note that the index is of items by date, as opposed to an index of dates. If 30 items have the same issue date (say 2002), then those 30 items all appear in the index adjacent to each other, as opposed to a single 2002 entry.

Since dates in DSpace Dublin Core are in ISO8601, all in the UTC time zone, a simple alphanumeric sort is sufficient to sort by date, including dealing with varying granularities of date reasonably. For example:

```
2001-12-10
2002
2002-04
2002-04-05
2002-04-09T15:34:12Z
2002-04-09T19:21:12Z
2002-04-10
```

Date Accessioned

In order to determine which items most recently appeared, rather than using the date of issue, an item's accession date is used. This is the Dublin Core field **`date.accessioned`**. In other aspects this index is identical to the date of issue index.

Items by a Particular Author

One last operation the browse API can perform is to extract items by a particular author. They do not have to be primary author of an item for that item to be extracted. You can specify a scope, too; that is, you can ask for items by author X in collection Y, for example.

This particular flavour of browse is slightly simpler than the others. You cannot presently specify a particular subset of results to be returned. The API call will simply return all of the items by a particular author within a certain scope.

Note that the author of the item must exactly match the author passed in to the API; see the explanation about the caveats of the author index browsing to see why this is the case.

The API is generally invoked by creating a `BrowseScope` object, and setting the parameters for which particular part of an index you want to extract. This is then passed to the relevent `Browse` method call, which returns a `BrowseInfo` object which contains the results of the operation. The parameters set in the `BrowseScope` object are:

- How many entries from the index you want
- Whether you only want entries from a particular community or collection, or from the whole of DSpace
- Which part of the index to start from (called the focus of the browse). If you don't specify this, the start of the index is used
- How many entries to include before the focus entry

To illustrate, here is an example:

- We want **7** entries in total
- We want entries from collection x
- We want the focus to be 'Really'
- We want **2** entries included before the focus.

The results of invoking `Browse.getItemsByTitle` with the above parameters might look like this:

```
         Rabble-Rousing Rabbis From Sardinia
         Reality TV: Love It or Hate It?
FOCUS>   The Really Exciting Research Video
         Recreational Housework Addicts: Please Visit My House
         Regional Television Variation Studies
         Revenue Streams
         Ridiculous Example Titles:  I'm Out of Ideas
```

Note that in the case of title and date browses, `Item` objects are returned as opposed to actual titles. In these cases, you can specify the 'focus' to be a specific item, or a partial or full literal value. In the case of a literal value, if no entry in the index matches exactly, the closest match is used as the focus. It's quite reasonable to specify a focus of a single letter, for example.

Being able to specify a specific item to start at is particularly important with dates, since many items may have the save issue date. Say 30 items in a collection have the issue date 2002. To be able to page through the index 20 items at a time, you need to be able to specify exactly which item's 2002 is the focus of the browse, otherwise each time you invoked the browse code, the results would start at the first item with the issue date 2002.

Author browses return `String` objects with the actual author names. You can only specify the focus as a full or partial literal `String`.

Another important point to note is that presently, the browse indices contain metadata for all items in the main archive, regardless of authorization policies. This means that all items in the archive will appear to all users when browsing. Of course, should the user attempt to access a non-public item, the usual authorization mechanism will apply. Whether this approach is ideal is under review; implementing the browse API such that the results retrieved reflect a user's level of authorization may be possible, but rather tricky.

Index Maintenance

The browse API contains calls to add and remove items from the index, and to regenerate the indices from scratch. In general the content management API invokes the necessary browse API calls to keep the browse indices in sync with what is in the archive, so most applications will not need to invoke those methods.

If the browse index becomes inconsistent for some reason, the `InitializeBrowse` class is a command line tool (generally invoked using the `/dspace/bin/index-all` shell script) that causes the indices to be regenerated from scratch.

Caveats

Presently, the browse API is not tremendously efficient. 'Indexing' takes the form of simply extracting the relevant Dublin Core value, normalising it (lower-casing and removing any leading article in the case of titles), and inserting that normalized value with the corresponding item ID in the appropriate browse database table. Database views of this table include collection and community IDs for browse operations with a limited scope. When a browse operation is performed, a simple `SELECT` query is performed, along the lines of:

```
SELECT item_id FROM ItemsByTitle ORDER BY sort_title OFFSET 40 LIMIT 20
```

There are two main drawbacks to this: Firstly, `LIMIT` and `OFFSET` are PostgreSQL-specific keywords. Secondly, the database is still actually performing dynamic sorting of the titles, so the browse code as it stands will not scale particularly well. The code does cache `BrowseInfo` objects, so that common browse operations are performed quickly, but this is not an ideal solution.

History Recorder

The purpose of the history subsystem is to capture a time-based record of significant changes in DSpace, in a manner suitable for later refactoring or repurposing. Note that the history data is not expected to provide current information about the archive; it simply records what has happened in the past.

The Harmony project describes a simple and powerful approach for modeling temporal data. The DSpace history framework adopts this model. The Harmony model is used by the serialization mechanism (and ultimately by agents who interpret the serializations); users of the History API need not be aware of it. The content management API handles invocations of the history system. Users of the DSpace public API do not generally need to use the history API.

When anything of archival interest occurs in DSpace, the `saveHistory` method of the `HistoryManager` is invoked. The parameters contains a reference to anything of archival interest. Upon reception of the object, it serializes the state of all archive objects referred to by it, and creates Harmony-style objects and associations to describe the relationships between the objects. (A simple example is given below). Note that each archive object must have a unique identifier to allow linkage between discrete events; this is discussed under "Unique IDs" below.

The serializations (including the Harmony objects and associations) are persisted as files in the `/dspace/history` (or other configured) directory. The `history` and `historystate` tables contain simple indicies into the serializations in the file system.

Archival Events

The following events are significant enough to warrant history records:

- Communities
  - create/modify/delete
  - add/remove Collection to/from Community
- Collections
  - create/modify/delete
  - add/remove Item to/from Collection
- Items
  - create/modify/delete
  - assign Handle to Item
  - modify Item contents (Bundles, Bitstreams, metadata fields, etc)
- EPerson
  - create/modify/delete
- Workflow
  - Workflow completed

Serializations

The serialization of an archival object consists of:

- Its instance fields (ie, non-static, non-transient fields)

- The serializations of associated objects (or references to these serializations).

Unique Ids

To be able to trace the history of an object, it is essential that the object have a unique identifier. Since not all objects in the system have Handles, the unique identifiers are only weakly tied to the Handle system. Instead, the identifier consists of:

- an identifer for the project
- a site id (using the handle prefix)
- an RDBMS-based id for objects

Storage

When an archive object is serialized, an object ID and MD5 checksum are recorded. When another object is serialized, the checksum for the serialization is matched against existing checksums for that object. If the checksum already exists, the object is not stored; a reference to the object is used instead. Note that since none of the serializations are deleted, reference counting is unnecessary.

The history data is not initially stored in a queryable form. Two simple RDBMS tables give basic indications of what is stored, and where. The `history` table is an index of serializations with checksums and dates. The `history_id` column corresponds to the file in which a serialization is stored. For example, if the history ID is 123456, it will be stored in the file:

```
/dspace/history/00/12/34/123456
```

The table also contains the date the serialization was written and the MD5 checksum of the serialization.

The `historystate` table is supposed to indicate the most recent serialization of any given object.

Example

An item is submitted to a collection via bulk upload. When (and if) the item is eventually added to the collection, the history method is called, with references to the item, its collection, the e-person who performed the bulk upload, and some indication of the fact that it was submitted via a bulk upload.

When called, the HistoryManager does the following: It creates the following new resources (all with unique ids):

- An event
- A state
- An action

It also generates the following relationships:

```
event  --atTime-->     time
event  --hasOutput-->  state
Item   --inState-->    state
state  --contains-->   Item
action --creates-->    Item
event  --hasAction-->  action
action --usesTool-->   DSpace Upload
action --hasAgent-->   User
```

The history component serializes the state of all archival objects involved (in this case, the item, the e-person, and the collection). It creates entries in the history database tables which associate the archival objects with the generated serializations.

Caveats

This history system is a largely untested experiment. It also needs further documentation. There have been no serious efforts to determine whether the information written by the history system, either to files or the database tables, is accurate. In particular, the `historystate` table does not seem to be correctly written.

Application Layer

Web User Interface

The DSpace Web UI is the largest and most-used component in the application layer. Built on Java Servlet and JavaServer Page technology, it allows end-users to access DSpace over the Web via their Web browsers.

It also features an administration section, consisting of pages intended for use by central administrators. Presently, this part of the Web UI is not particularly sophisticated; users of the administration section need to know what they are doing! Selected parts of this may also be used by collection [FIXME: administrators or editors?]

Web UI Files

The Web UI-related files are located in a variety of directories in the DSpace source tree. Note that as of DSpace version 1.2, the deployment mechanism has changed; the build process creates easy-to-deploy Web application archives (`.war` files).

Locations of Web UI Source Files

Location	Description
`org.dspace.app.webui`	Web UI source files
`org.dspace.app.webui.filter`	Servlet Filters (Servlet 2.3 spec)

org.dspace.app.webui.jsptag	Custom JSP tag class files
org.dspace.app.webui.servlet	Servlets for main Web UI (controllers)
org.dspace.app.webui.servlet.admin	Servlets that comprise the administration part of the Web UI
org.dspace.app.webui.util	Miscellaneous classes used by the servlets and filters
[dspace-source]/jsp	The JSP files
[dspace-source]/jsp/local	This is where you can place customized versions of JSPs -- see the configuration section
[dspace-source]/jsp/WEB-INF/dspace-tags.tld	Custom DSpace JSP tag descriptor
[dspace-source]/etc/dspace-web.xml	The Web application deployment descriptor. Before including in the .war file, the text @@dspace.dir@@ will be replaced with the DSpace installation directory (referred to as [dspace] elsewhere in this system documentation). This allows the Web application to pick up the DSpace configuration and environment.

The Build Process

The DSpace build process constructs a Web application archive, which is placed in [dspace-source]/build/dspace.war. The build_wars Ant target does the work. The process works as follows:

- All the DSpace source code is compiled.
- [dspace-source]/etc/dspace-web.xml is copied to [dspace-source]/build and the @@dspace.dir@@ token inside it replaced with the DSpace installation directory (dspace.dir property from dspace.cfg
- The JSPs are all copied to [dspace-source]/build/jsp
- Customized JSPs from [dspace-source]/jsp/local are copied on top of these, thus 'overriding' the default versions
- [dspace-source]/build/dspace.war is built

The contents of dspace.war are:

- (Top level) -- the JSPs (customized versions from [dspace-source]/jsp/local will have overwritten the defaults from the DSpace source distribution)
- WEB-INF/classes -- the compiled DSpace classes
- WEB-INF/lib -- the third party library JAR files from [dspace-source]/lib, minus servlet.jar which will be available as part of Tomcat (or other servlet engine)
- WEB-INF/web.xml -- web deployment descriptor, copied from [dspace-source]/build/dspace-web.xml
- WEB-INF/dspace-tags.tld -- tag descriptor

Note that this does mean there are multiple copies of the compiled DSpace code and third-party libraries in the system, so care must be taken to ensure that they are all in sync. (The

storage overhead is a few megabytes, totally insignificant these days.) In general, when you change any DSpace code or JSP, it's best to do a complete update of both the installation (`[dspace]`), and to rebuild and redeploy the Web UI and OAI `.war` files, by running this in `[dspace-source]`:

```
ant -D[dspace]/config/dspace.cfg update
```

and then following the instructions that command writes to the console.

Servlets and JSPs

The Web UI is loosely based around the MVC (model, view, controller) model. The content management API corresponds to the model, the Java Servlets are the controllers, and the JSPs are the views. Interactions take the following basic form:

1. An HTTP request is received from a browser
2. The appropriate servlet is invoked, and processes the request by invoking the DSpace business logic layer public API
3. Depending on the outcome of the processing, the servlet invokes the appropriate JSP
4. The JSP is processed and sent to the browser

The reasons for this approach are:

- All of the processing is done before the JSP is invoked, so any error or problem that occurs does not occur halfway through HTML rendering
- The JSPs contain as little code as possible, so they can be customized without having to delve into Java code too much

The `org.dspace.app.webui.servlet.LoadDSpaceConfig` servlet is always loaded first. This is a very simple servlet that checks the `dspace-config` context parameter from the DSpace deployment descriptor, and uses it to locate `dspace.cfg`. It also loads up the Log4j configuration. It's important that this servlet is loaded first, since if another servlet is loaded up, it will cause the system to try and load DSpace and Log4j configurations, neither of which would be found.

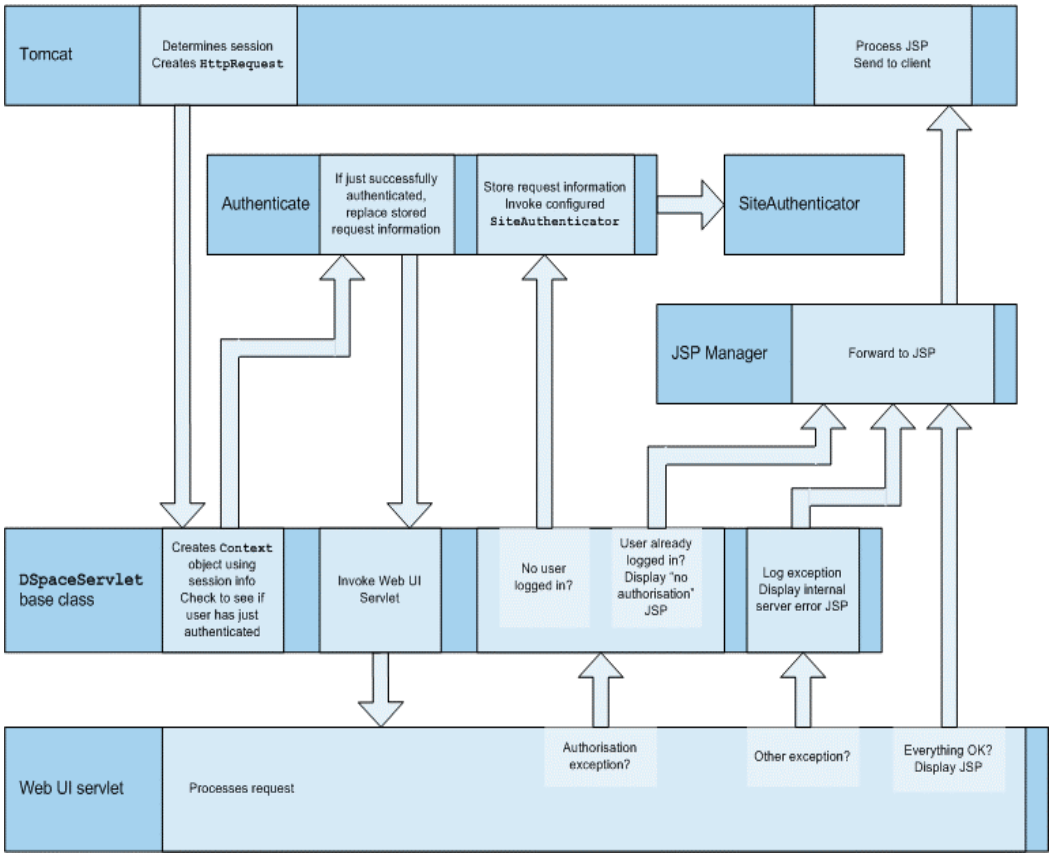
All DSpace servlets are subclasses of the `DSpaceServlet` class. The `DSpaceServlet` class handles some basic operations such as creating a DSpace `Context` object (opening a database connection etc.), authentication and error handling. Instead of overriding the `doGet` and `doPost` methods as one normally would for a servlet, DSpace servlets implement `doDSGet` or `doDSPost` which have an extra context parameter, and allow the servlet to throw various exceptions that can be handled in a standard way.

The DSpace servlet processes the contents of the HTTP request. This might involve retrieving the results of a search with a query term, accessing the current user's eperson record, or updating a submission in progress. According to the results of this processing, the servlet must decide which JSP should be displayed. The servlet then fills out the appropriate attributes in the `HttpRequest` object that represents the HTTP request being processed. This is done by invoking the `setAttribute` method of the `javax.servlet.http.HttpServletRequest` object that is passed into the servlet from

Tomcat. The servlet then forwards control of the request to the appropriate JSP using the `JSPManager.showJSP` method.

The `JSPManager.showJSP` method uses the standard Java servlet forwarding mechanism is then used to forward the HTTP request to the JSP. The JSP is processed by Tomcat and the results sent back to the user's browser.

There is an exception to this servlet/JSP style: `index.jsp`, the 'home page', receives the HTTP request directly from Tomcat without a servlet being invoked first. This is because in the servlet 2.3 specification, there is no way to map a servlet to handle only requests made to '`/`'; such a mapping results in every request being directed to that servlet. By default, Tomcat forwards requests to '`/`' to `index.jsp`. To try and make things as clean as possible, `index.jsp` contains some simple code that would normally go in a servlet, and then forwards to `home.jsp` using the `JSPManager.showJSP` method. This means localized versions of the 'home page' can be created by placing a customized `home.jsp` in `[dspace-source]/jsp/local`, in the same manner as other JSPs.



Flow of Control During HTTP Request Processing

`[dspace-source]/jsp/dspace-admin/index.jsp`, the administration UI index page, is invoked directly by Tomcat and not through a servlet for similar reasons.

At the top of each JSP file, right after the license and copyright header, is documented the appropriate attributes that a servlet must fill out prior to forwarding to that JSP. No

validation is performed; if the servlet does not fill out the necessary attributes, it is likely that an internal server error will occur.

Many JSPs containing forms will include hidden parameters that tell the servlets which form has been filled out. The submission UI servlet (`SubmitServlet` is a prime example of a servlet that deals with the input from many different JSPs. The `step` hidden parameter is used to inform the servlet which form has been filled out (which step of submission the user has just completed.)

Below is a detailed, scary diagram depicting the flow of control during the whole process of processing and responding to an HTTP request. More information about the authentication mechanism is mostly described in the configuration section.

Custom JSP Tags

The DSpace JSPs all use some custom tags defined in `/dspace/jsp/WEB-INF/dspace-tags.tld`, and the corresponding Java classes reside in `org.dspace.app.webui.jsptag`. The tags are listed below. The `dspace-tags.tld` file contains detailed comments about how to use the tags, so that information is not repeated here.

`layout`

Just about every JSP uses this tag. It produces the standard HTML header and `<BODY>`tag. Thus the content of each JSP is nested inside a `<dspace:layout>` tag. The (XML-style)attributes of this tag are slightly complicated--see `dspace-tags.tld`. The JSPs in the source code bundle also provide plenty of examples.

`sidebar`

Can only be used inside a `layout` tag, and can only be used once per JSP. The content between the start and end `sidebar` tags is rendered in a column on the right-hand side of the HTML page. The contents can contain further JSP tags and Java 'scriptlets'.

`date`

Displays the date represented by an `org.dspace.content.DCDate` object. Just the one representation of date is rendered currently, but this could use the user's browser preferences to display a localized date in the future.

`include`

Obsolete, simple tag, similar to `jsp:include`. In versions prior to DSpace 1.2, this tag would use the locally modified version of a JSP if one was installed in jsp/local. As of 1.2, the build process now performs this function, however this tag is left in for backwards compatibility.

`item`

Displays an item record, including Dublin Core metadata and links to the bitstreams within it. Note that the displaying of the bitstream links is simplistic, and does not take into

account any of the bundling structure. This is because DSpace does not have a fully-fledged dissemination architectural piece yet.

Displaying an item record is done by a tag rather than a JSP for two reasons: Firstly, it happens in several places (when verifying an item record during submission or workflow review, as well as during standard item accesses), and secondly, displaying the item turns out to be mostly code-work rather than HTML anyway. Of course, the disadvantage of doing it this way is that it is slightly harder to customize exactly what is displayed from an item record; it is necessary to edit the tag code (`org.dspace.app.webui.jsptag.ItemTag`). Hopefully a better solution can be found in the future.

`itemlist`, `collectionlist`, `communitylist`

These tags display ordered sequences of items, collections and communities, showing minimal information but including a link to the page containing full details. These need to be used in HTML tables.

`popup`

This tag is used to render a link to a pop-up page (typically a help page.) If Javascript is available, the link will either open or pop to the front any existing DSpace pop-up window. If Javascript is not available, a standard HTML link is displayed that renders the link destination in a window named '`dspace.popup`'. In graphical browsers, this usually opens a new window or re-uses an existing window of that name, but if a window is re-used it is not 'raised' which might confuse the user. In text browsers, following this link will simply replace the current page with the destination of the link. This obviously means that Javascript offers the best functionality, but other browsers are still supported.

`selecteperson`

A tag which produces a widget analogous to HTML `<SELECT>`, that allows a user to select one or multiple e-people from a pop-up list.

`sfxlink`

Using an item's Dublin Core metadata DSpace can display an SFX link, if an SFX server is available. This tag does so for a particular item if the `sfx.server.url` property is defined in `dspace.cfg`.

HTML Support

For the most part, the DSpace item display just gives a link that allows an end-user to download a bitstream. However, if a bundle has a primary bitstream whose format is of MIME type `text/html`, instead a link to the HTML servlet is given.

So if we had an HTML document like this:

```
contents.html
chapter1.html
chapter2.html
```

```
chapter3.html
figure1.gif
figure2.jpg
figure3.gif
figure4.jpg
figure5.gif
figure6.gif
```

The Bundle's primary bitstream field would point to the contents.html Bitstream, which we know is HTML (check the format MIME type) and so we know which to serve up first.

The HTML servlet employs a trick to serve up HTML documents without actually modifying the HTML or other files themselves. Say someone is looking at `contents.html` from the above example, the URL in their browser will look like this:

```
https://dspace.mit.edu/html/1721.1/12345/contents.html
```

If there's an image called `figure1.gif` in that HTML page, the browser will do HTTP GET on this URL:

```
https://dspace.mit.edu/html/1721.1/12345/figure1.gif
```

The HTML document servlet can work out which item the user is looking at, and then which Bitstream in it is called `figure1.gif`, and serve up that bitstream. Similar for following links to other HTML pages. Of course all the links and image references have to be relative and not absolute.

This can cope with relative links that refer to a deeper path, e.g.

```
<IMG SRC="images/figure1.gif">
```

Remember that in the Bitstream table in the database we have the 'name' field, which always contains the filename with no path (`figure1.gif`). We also have the `source` field, which may contain the full pathname of the file as it appeared on the submitter's hard drive, but this is browser- and OS-dependent, so we can't rely on it. All we can rely on is the filename.

We can still work out what images/figure1.gif is by making the HTML document servlet strip any path that comes in from the URL, e.g.

```
https://dspace.mit.edu/html/1721.1/12345/images/figure1.gif
                                          ^^^^^^^
                                        Strip this
```

BUT all the filenames (regardless of directory names) must be unique. For example, this wouldn't work:

```
contents.html
chapter1.html
chapter2.html
chapter1_images/figure.gif
chapter2_images/figure.gif
```

since the HTML document servlet wouldn't know which bitstream to serve up for:

```
https://dspace.mit.edu/html/1721.1/12345/chapter1_images/figure.gif
https://dspace.mit.edu/html/1721.1/12345/chapter2_images/figure.gif
```

since it would just have `figure.gif` in the Bitstream table. Thus, the limitations are:

- All links must be relative and not refer to parents (e.g. `../images/foo.gif` or `/images/foo.gif`)
- If links refer to deeper directory levels, all the filenames must be different (as explained above)

Thesis Blocking

The submission UI has an optional feature that came about as a result of MIT Libraries policy. If the `block.theses` parameter in `dspace.cfg` is `true`, an extra checkbox is included in the first page of the submission UI. This asks the user if the submission is a thesis. If the user checks this box, the submission is halted (deleted) and an error message displayed, explaining that DSpace should not be used to submit theses. This feature can be turned off and on, and the message displayed (`/dspace/jsp/submit/no-theses.jsp` can be localized as necessary.

OAI-PMH Data Provider

The DSpace platform supports the Open Archives Initiative Protocol for Metadata Harvesting (OAI-PMH) version 2.0 as a data provider. This is accomplished using the OAICat framework from OCLC.

The DSpace build process builds a Web application archive, `[dspace-source]/build/dspace-oai.war`), in much the same way as the Web UI build process described above. The only differences are that the JSPs are not included, and `[dspace-source]/etc/oai-web.xml` is used as the deployment descriptor. This 'webapp' is deployed to receive and respond to OAI-PMH requests via HTTP. Note that typically it should not be deployed on SSL (`https:` protocol). In a typical configuration, this is deployed at `dspace-oai`, for example:

```
http://dspace.myu.edu/dspace-oai/request?verb=Identify
```

The 'base URL' of this DSpace deployment would be:

```
http://dspace.myu.edu/dspace-oai/request
```

It is this URL that should be registered with www.openarchives.org. Note that you can easily change the '`request`' portion of the URL by editing `[dspace-source]/etc/oai-web.xml` and rebuilding and deploying `dspace-oai.war`.

DSpace provides implementations of the OAICat interfaces `AbstractCatalog`, `RecordFactory` and `Crosswalk` that interface with the DSpace content management API and harvesting API (in the search subsystem).

Only the basic `oai_dc` unqualified Dublin Core metadata set is exported at present; this is particularly easy since all items have qualified Dublin Core metadata. When this metadata is harvested, the qualifiers are simply stripped; for example, `description.abstract` is exposed as unqualified `description`. The `description.provenance` field is hidden, as this contains private information about the submitter and workflow reviewers of the item, including their e-mail addresses. Additionally, to keep in line with OAI community practices, values of `contributor.author` are exposed as `creator` values.

To add support for other metadata sets is simply a matter of creating another `Crosswalk` implementation, and adding it to the `oaicat.properties` file described below.

Note that the current simple DC implementation (`org.dspace.app.oai.OAIDCCrosswalk`) does not currently strip out any invalid XML characters that may be lying around in the data. If your database contains a DC value with, for example, some ASCII control codes (form feed etc.) this may cause OAI harvesters problems. This should rarely occur, however. XML entities (such as `>`) are encoded (e.g. to `>`)

In addition to the implementations of the OAICat interfaces, there are two configuration files relevant to OAI support:

`oaicat.properties`

This resides as a template in `[dspace]/config/templates`, and the live version is written to `[dspace]/config`. You probably won't need to edit this; the `install-configs` script fills out the relevant deployment-specific parameters. You might want to change the `earliestDatestamp` field to accurately reflect the oldest datestamp in the system. (Note that this is the value of the `last_modified` column in the `Item` database table.)

`oai-web.xml`

This standard Java Servlet 'deployment descriptor' is stored in the source as `[dspace-source]/etc/oai-web.xml`, and is written to `/dspace/oai/WEB-INF/web.xml`.

Sets

OAI-PMH allows repositories to expose an hierarchy of sets in which records may be placed. A record can be in zero or more sets.

DSpace exposes collections as sets. The organization of communities is likely to change over time, and is therefore a less stable basis for selective harvesting.

Each collection has a corresponding OAI set, discoverable by harvesters via the ListSets verb. The setSpec is the Handle of the collection, with the ':' and '/' converted to underscores so that the Handle is a legal setSpec, for example:

```
hdl_1721.1_1234
```

Naturally enough, the collection name is also the name of the corresponding set.

Unique Identifier

Every item in OAI-PMH data repository must have an unique identifier, which must conform to the URI syntax. As of DSpace 1.2, Handles are not used; this is because in OAI-PMH, the OAI identifier identifies the metadata record associated with the resource. The resource is the DSpace item, whose resource identifier is the Handle. In practical terms, using the Handle for the OAI identifier may cause problems in the future if DSpace instances share items with the same Handles; the OAI metadata record identifiers should be different as the different DSpace instances would need to be harvested separately and may have different metadata for the item.

The OAI identifiers that DSpace uses are of the form:

```
oai:host name:handle
```

For example:

```
oai:dspace.myu.edu:123456789/345
```

If you wish to use a different scheme, this can easily be changed by editing the value of `OAI_ID_PREFIX` at the top of the `org.dspace.app.oai.DSpaceOAICatalog` class. (You do not need to change the code if the above scheme works for you; the code picks up the host name and Handles automatically from the DSpace configuration.)

Access control

OAI provides no authentication/authorisation details, although these could be implemented using standard HTTP methods. It is assumed that all access will be anonymous for the time being.

A question is, "is all metadata public?" Presently the answer to this is yes; all metadata is exposed via OAI-PMH, even if the item has restricted access policies. The reasoning behind this is that people who do actually have permission to read a restricted item should still be able to use OAI-based services to discover the content.

If in the future, this 'expose all metadata' approach proves unsatisfactory for any reason, it should be possible to expose only publicly readable metadata. The authorisation system has separate permissions for READing and item and READing the content (bitstreams) within it. This means the system can differentiate between an item with public metadata and hidden content, and an item with hidden metadata as well as hidden content. In this case the OAI data repository should only expose items those with anonymous READ access, so it can hide the existence of records to the outside world completely. In this scenario, one should be wary of protected items that are made public after a time. When this happens, the items are "new" from the OAI-PMH perspective.

Modification Date (OAI Date Stamp)

OAI-PMH harvesters need to know when a record has been created, changed or deleted. DSpace keeps track of a 'last modified' date for each item in the system, and this date is

used for the OAI-PMH date stamp. This means that any changes to the metadata (e.g. admins correcting a field, or a withdrawal) will be exposed to harvesters.

'About' Information

As part of each record given out to a harvester, there is an optional, repeatable "about" section which can be filled out in any (XML-schema conformant) way. Common uses are for provenance and rights information, and there are schemas in use by OAI communities for this. Presently DSpace does not provide any of this information.

Deletions

DSpace keeps track of deletions (withdrawals). These are exposed via OAI, which has a specific mechansim for dealing with this. Since DSpace keeps a permanent record of withdrawn items, in the OAI-PMH sense DSpace supports deletions 'persistently'. This is as opposed to 'transient' deletion support, which would mean that deleted records are forgotten after a time.

Once an item has been withdrawn, OAI-PMH harvests of the date range in which the withdrawal occurred will find the 'deleted' record header. Harvests of a date range prior to the withdrawal will not find the record, despite the fact that the record did exist at that time.

As an example of this, consider an item that was created on 2002-05-02 and withdrawn on 2002-10-06. A request to harvest the month 2002-10 will yield the 'record deleted' header. However, a harvest of the month 2002-05 will not yield the original record.

Note that presently, the deletion of 'expunged' items is not exposed through OAI.

Flow Control (Resumption Tokens)

An OAI data provider can prevent any performance impact caused by harvesting by forcing a harvester to receive data in time-separated chunks. If the data provider receives a request for a lot of data, it can send part of the data with a resumption token. The harvester can then return later with the resumption token and continue.

DSpace supports resumption tokens for 'ListRecords' OAI-PMH requests. ListIdentifiers and ListSets requests do not produce a particularly high load on the system, so resumption tokens are not used for those requests.

Each OAI-PMH ListRecords request will return at most 100 records. This limit is set at the top of `org.dspace.app.oai.DSpaceOAICatalog.java` (`MAX_RECORDS`). A potential issue here is that if a harvest yields an exact multiple of `MAX_RECORDS`, the last operation will result in a harvest with no records in it. It is unclear from the OAI-PMH specification if this is acceptable.

When a resumption token is issued, the optional `completeListSize` and `cursor` attributes are not included. OAICat sets the `expirationDate` of the resumption token to one hour after it was issued, though in fact since DSpace resumption tokens contain all the information required to continue a request they do not actually expire.

Resumption tokens contain all the state information required to continue a request. The format is:

```
from/until/setSpec/offset
```

`from` and `until` are the ISO 8601 dates passed in as part of the original request, and `setSpec` is also taken from the original request. `offset` is the number of records that have already been sent to the harvester. For example:

```
2003-01-01//hdl_1721_1_1234/300
```

This means the harvest is 'from' `2003-01-01`, has no 'until' date, is for collection hdl:1721.1/1234, and 300 records have already been sent to the harvester. (Actually, if the original OAI-PMH request doesn't specify a 'from' or 'until, OAICat fills them out automatically to '0000-00-00T00:00:00Z' and '9999-12-31T23:59:59Z' respectively. This means DSpace resumption tokens will always have from and until dates in them.)

Item Importer and Exporter

DSpace has a set of command line tools for importing and exporting items in batches, using the DSpace simple archive format. The tools are not terribly robust, but are useful and are easily modified. They also give a good demonstration of how to implement your own item importer if desired.

Warning: templates may be applied

Due to a bug as of 1.2 beta 2, if you have an Item template in your Collection, then those default values may be added to Items that you import. Be sure to remove the template if this is unwanted behavior.

DSpace simple archive format

The basic concept behind the DSpace's simple archive format is to create an archive, which is directory full of items, with a subdirectory per item. Each item directory contains a file for the item's descriptive metadata, and the files that make up the item.

```
archive_directory/
    item_000/
        dublin_core.xml -- qualified Dublin Core metadata
        contents        -- text file containing one line per filename
        file_1.doc      -- files to be added as bitstreams to the item
        file_2.pdf
    item_001/
        dublin_core.xml
        contents
        file_1.png
        ...
```

The dublin_core.xml file has the following format, where each Dublin Core element has it's own entry within a <dcvalue> tagset. There are currently three tag elements available in the <dcvalue> tagset:

- <element> - the Dublin Core element
- <qualifier> - the element's qualifier
- <language> - (optional)ISO language code for element

```
<dublin_core>
    <dcvalue element="title" qualifier="none">A Tale of Two
Cities</dcvalue>
    <dcvalue element="date"
qualifier="issued">1990</dcvalue></dublin_core>
    <dcvalue element="title" qualifier="alternate" language="fr"
">J'aime les Printemps</dcvalue>
</dublin_core>

(Note the optional language tag which notifies the system that the
optional title is in French.)
```

Importing Items

Note: Before running the item importer over items previously exported from a DSpace instance, please first refer to Transferring Items Between DSpace Instances.

The item importer is in org.dspace.app.itemimport.ItemImport, and is run with the dsrun utility in the dspace/bin directory. Running it with -h gets the current command-line arguments. Another very important flag is the --test flag, which you can use with any command to simulate all of the actions it will perform without actually making any changes to your DSpace instance - very useful for validating your item directories before doing an import. In the importer's arguments you can use either the user's database ID or email address and the eperson ID, and the collection's database ID or handle as arguments. Currently with the importer you can add, remove, and replace items in a collection. If you specify more than one collection argument then the items will be imported to multiple collections, and the first collection specified becomes the "owning" collection. If there is an error and the import is aborted, there is a --resume flag that you can try to resume the import where you left off after you fix the error.

To add items to a collection with an EPerson as the submitter, type:

```
dsrun org.dspace.app.itemimport.ItemImport --add --eperson=joe@user.com
--collection=collectionID --source=items_dir --mapfile=mapfile
```

(or by using the short form)

```
dsrun org.dspace.app.itemimport.ItemImport -a -e joe@user.com  -c
collectionID -s items_dir -m mapfile
```

which would then cycle through the archive directory's items, import them, and then generate a map file which stores the mapping of item directories to item handles. Save this map file! Using the map file you can then 'unimport' with the command:

```
dsrun org.dspace.app.itemimport.ItemImport --delete --mapfile=mapfile
```

The imported items listed in the map file would then be deleted. If you wish to replace previously imported items, you can give the command:

```
dsrun org.dspace.app.itemimport.ItemImport --replace --
eperson=joe@user.com --collection=collectID --source=items_dir --
mapfile=mapfile
```

Replacing items uses the map file to replace the old items and still retain their handles.

The importer usually bypasses any workflow assigned to a collection, but adding the --workflow option will route the imported items through the workflow system.

The importer also has a --test flag that will simulate the entire import process without actually doing the import. This is extremely useful for verifying your import files before doing the import step.

Exporting Items

The item exporter can export a single item or a collection of items, and creates a DSpace simple archive for each item to be exported. To export a collection's items you type:

```
dsrun org.dspace.app.itemexport.ItemExport --type=COLLECTION --id=collID
--dest=dest_dir --number=seq_num
```

The keyword `COLLECTION` means that you intend to export an entire collection. The ID can either be the database ID or the handle. The exporter will begin numbering the simple archives with the sequence number that you supply. To export a single item use the keyword `ITEM` and give the item ID as an argument:

```
dsrun org.dspace.app.itemexport.ItemExport --type=ITEM --id=itemID --
dest=dest_dir --number=seq_num
```

Each exported item will have an additional file in its directory, named 'handle'. This will contain the handle that was assigned to the item, and this file will be read by the importer so that items exported and then imported to another machine will retain the item's original handle.

Transferring Items Between DSpace Instances

Where items are to be moved between DSpace instances (for example from a test DSpace into a production DSpace) the item exporter and item importer can be used in conjunction with a script to assist in this process.

After running the item exporter each `dublin_core.xml` file will contain metadata that was automatically added by DSpace. These fields are as follows:

- date.accessioned
- date.available

- date.issued
- description.provenance
- format.extent
- format.mimetype
- identifier.uri

In order to avoid duplication of this metadata, run

```
dspace_migrate <exported item directory>
```

prior to running the item importer. This will remove the above metadata items from the `dublin_core.xml` file and remove all `handle` files. It will then be safe to run the item exporter. Use

```
dspace_migrate --help
```

for instructions on use of the script.

METS Tools

The experimental (incomplete) METS export tool writes DSpace items to a filesystem with the metadata held in a more standard format based on METS.

The Export Tool

The METS export tool is invoked via the command line like this:

```
[dspace]/bin/dsrun org.dspace.app.mets.METSExport --help
```

The tool can export an individual item, the items within a given collection, or everything in the DSpace instance. To export an individual item, use:

```
[dspace]/bin/dsrun org.dspace.app.mets.METSExport --item [handle]
```

To export the items in collection `hdl:123.456/789`, use:

```
[dspace]/bin/dsrun org.dspace.app.mets.METSExport --collection
hdl:123.456/789
```

To export all the items DSpace, use:

```
[dspace]/bin/dsrun org.dspace.app.mets.METSExport --all
```

With any of the above forms, you can specify the base directory into which the items will be exported, using `--destination [directory]`. If this parameter is omitted, the current directory is used.

The AIP Format

Each exported item is written to a separate directory, created under the base directory specified in the command-line arguments, or in the current directory if `--destination` is omitted. The name of each directory is the Handle, URL-encoded so that the directory name is 'legal'.

Within each item directory is a `mets.xml` file which contains the METS-encoded metadata for the item. Bitstreams in the item are also stored in the directory. Their filenames are their MD5 checksums, firstly for easy integrity checking, and also to avoid any problems with 'special characters' in the filenames that were legal on the original filing system they came from but are illegal in the server filing system. The `mets.xml` file includes XLink pointers to these bitstream files.

An example AIP might look like this:

- `hdl%3A123456789%2F8/`
  - `mets.xml` -- METS metadata
  - `184BE84F293342` -- bitstream
  - `3F9AD0389CB821`
  - `135FB82113C32D`

The contents of the METS in the `mets.xml` file are as follows:

- A `dmdSec` (descriptive metadata section) containing the item's metadata in Metadata Object Description Schema (MODS) XML. The Dublin Core descriptive metadata is mapped to MODS since there is no official qualified Dublin Core XML schema in existence as of yet, and the Library Application Profile of DC that DSpace uses includes some qualifiers that are not part of the DCMI Metadata Terms.
- An `amdSec` (administrative metadata section), which contains the a rights metadata element, which in turn contains the base64-encoded deposit license (the license the submitter granted as part of the submission process).
- A `fileSec` containing a list of the bitstreams in the item. Each bundle constitutes a `fileGrp`. Each bitstream is represented by a `file` element, which contains an `FLocat` element with a simple XLink to the bitstream in the same directory as the `mets.xml` file. The `file` attributes consist of most of the basic technical metadata for the bitstream. Additionally, for those bitstreams that are thumbnails or text extracted from another bitstream in the item, those 'derived' bitstreams have the same `GROUPID` as the bitstream they were derived from, in order that clients understand that there is a relationship.

  The `OWNERID` of each `file` is the 'persistent' bitstream identifier assigned by the DSpace instance. The `ID` and `GROUPID` attributes consist of the item's Handle, together with the bitstream's sequence ID, which underscores used in place of dots and slashes. For example, a bitstream with sequence ID 24, in the item `hdl:123.456/789` will have the `ID` `123_456_789_24`. This is because `ID` and `GROUPID` attributes must be of type `xsd:id`.

Limitations

- No corresponding import tool yet
- No `structmap` section
- Some technical metadata not written, e.g. the primary bitstream in a bundle, original filenames or descriptions.
- Only the MIME type is stored, not the (finer grained) bitstream format.
- Dublin Core to MODS mapping is very simple, probably needs verification

MediaFilters: Transforming DSpace Content

DSpace can apply filters to content/bitstreams, creating new content. Filters are included that extract text for full-text searching, and create thumbnails for items that contain images. The media filters are controlled by the `MediaFilterManager` which traverses the asset store, invoking the `MediaFilter` subclasses on bitstreams. The file config/mediafilter.cfg contains a list of bitstream format types and the filters that operate on bitstreams of that type. The media filter system is intended to be run from the command line (or regularly as a cron task):

```
dspace/bin/filter-media
```

Traverse the asset store, applying media filters to bitstreams, skipping bitstreams that have already been filtered.

```
dspace/bin/filter-media -f
```

Apply filters to ALL bitstreams, even if they've already been filtered.

```
dspace/bin/filter-media -v
```

Verbose mode - print all extracted text and other filter details to STDOUT.

```
dspace/bin/filter-media -n
```

Suppress index creation - by default, a new search index is created for full-text searching. This option suppresses index creation if you intend to run `index-all` elsewhere.

Adding your own filters is done by creating a sub-class of the `MediaFilter` class. See the comments in the source file MediaFilter.java for more information. In theory filters could be implemented in any language (C, Perl, etc.) They only need to be invoked by the Java code in the `MediaFilter` class that you create.

Sub-Community Management

DSpace provides an administrative tool - 'CommunityFiliator' - for managing community sub-structure. Normally this structure seldom changes, but prior to the 1.2 release sub-communities were not supported, so this tool could be used to place existing pre-1.2 communities into a hierarchy. It has two operations, either establishing a community to sub-community relationship, or dis-establishing an existing relationship.

The familiar parent/child metaphor can be used to explain how it works. Every community in DSpace can be either a 'parent' community - meaning it has at least one sub-community, or a 'child' community - meaning it is a sub-community of another community, or both or neither. In these terms, an 'orphan' is a community that lacks a parent (although it can be a parent); 'orphans' are referred to as 'top-level' communities in the DSpace user-interface, since there is no parent community 'above' them. The first operation - establishing a parent/child relationship - can take place between any community and an orphan. The second operation - removing a parent/child relationship - will make the child an orphan.

Using the dsrun utility in the dspace/bin directory, the establish operation looks like this:

```
dsrun org.dspace.administer.CommunityFiliator --set --parent=parentID --
child=childID
```

(or using the short form)

```
dsrun org.dspace.administer.CommunityFiliator -s -p parentID -c childID
```

where '-s' or '--set' means establish a relationship whereby the community identified by the '-p' parameter becomes the parent of the community identified by the '-c' parameter. Both the 'parentID' and 'childID' values may be handles or database IDs.

The reverse operation looks like this:

```
dsrun org.dspace.administer.CommunityFiliator --remove --parent=parentID
--child=childID
```

(or using the short form)

```
dsrun org.dspace.administer.CommunityFiliator -r -p parentID -c childID
```

where '-r' or '--remove' means dis-establish the current relationship in which the community identified by 'parentID' is the parent of the community identified by 'childID'. The outcome will be that the 'childID' community will become an orphan, i.e. a top-level community.

If the required constraints of operation are violated, an error message will appear explaining the problem, and no change will be made. An example in a removal operation, where the stated child community does not have the stated parent community as its parent: "Error, child community not a child of parent community".

It is possible to effect arbitrary changes to the community hierarchy by chaining the basic operations together. For example, to move a child community from one parent to another, simply perform a 'remove' from its current parent (which will leave it an orphan), followed by a 'set' to its new parent.

It is important to understand that when any operation is performed, all the sub-structure of the child community follows it. Thus, if a child has itself children (sub-communities), or collections, they will all move with it to its new 'location' in the community tree.

Version History

Changes in DSpace 1.2.1

General Improvements

- Oracle support added
- Thumbnails in item view can now be switched off/on
- Browse and search thumbnail options
- Improved item importer
    - can now import to multiple collections
    - added --test flag to simulate an import, without actually making any changes
    - added --resume flag to try to resume the import in case the import is aborted
- Configurable fields for the search index
- Script for transferring items between DSpace instances
- Sun library JARs (JavaMail, Java Activation Framework and Servlet) now included in DSpace source code bundle

Bug fixes

- A logo to existing collection can now be added. Fixes SF bug #1065933
- The community logo can now be edited. Fixes SF bug #1035692
- MediaFilterManager doesn't 'touch' every item every time. Fixes SF bug #1015296
- Supported formats help page, set the format support level to "known" as default
- Fixed various database connection pool leaks

Changed JSPs

- `collection-home` changed
- `community-home` changed
- `display-item` changed
- `dspace-admin/confirm-delete-collection` moved to tools/ and changed
- `dspace-admin/confirm-delete-community` moved to tools/ and changed
- `dspace-admin/edit-collection` moved to tools/ and changed
- `dspace-admin/edit-community` moved to tools/ and changed
- `dspace-admin/index` changed
- `dspace-admin/upload-logo` changed
- `dspace-admin/wizard-basicinfo` changed
- `dspace-admin/wizard-default-item` changed
- `dspace-admin/wizard-permissions` changed
- `dspace-admin/wizard-questions` changed
- `help/formats.html` removed
- `help/formats` changed
- `index` changed
- `layout/navbar-admin` changed

Changes in DSpace 1.2

General Improvments

- Communities can now contain sub-communities
- Items may be included in more than one collection
- Full text extraction and searching for MS Word, PDF, HTML, text documents
- Thumbnails displayed in item view for items that contain images
- Configurable MediaFilter tool creates both extracted text and thumbnails
- Bitstream IDs are now persistent - generated from item's handle and a sequence number
- Creative Commons licenses can optionally be added to items during web submission process

Administration

- If you are logged in as administrator, you see admin buttons on item, collection, and community pages
- New collection administration wizard
- Can now administer collection's submitters from collection admin tool
- Delegated administration - new 'collection editor' role - edits item metadata, manages submitters list, edits collection metadata, links to items from other collections, and can withdraw items
- Admin UI moved from /admin to /dspace-admin to avoid conflict with Tomcat /admin JSPs
- New EPerson selector popup makes Group editing much easier
- 'News' section is now editable using admin UI (no more mucking with JSPs)

Import/Export/OAI

- New tool that exports DSpace content in AIPs that use METS XML for metadata (incomplete)
- OAI - sets are now collections, identified by Handles ('safe' with /, : converted to _)
- OAI - contributor.author now mapped to oai_dc:creator

Miscellaneous

- Build process streamlined with use of WAR files, symbolic links no longer used, friendlier to later versions of Tomcat
- MIT-specific aspects of UI removed to avoid confusion
- Item metadata now rendered to avoid interpreting as HTML (displays as entered)
- Forms now have no-cache directive to avoid trouble with browser 'back' button
- Bundles now have 'names' for more structure in item's content

JSP file changes between 1.1 and 1.2

This list generated with `cvs -Q rdiff -s -r dspace-1_1 dspace` and a sprinkling of perl.

- Changed: dspace/jsp/collection-home.jsp

- Changed: dspace/jsp/community-home.jsp
- Changed: dspace/jsp/community-list.jsp
- Changed: dspace/jsp/display-item.jsp
- Changed: dspace/jsp/index.jsp
- Changed: dspace/jsp/home.jsp
- Changed: dspace/jsp/styles.css.jsp
- Moved to dspace-admin and changed: dspace/jsp/admin/authorize-advanced.jsp
- Moved to dspace-admin and changed: dspace/jsp/admin/authorize-collection-edit.jsp
- Moved to dspace-admin and changed: dspace/jsp/admin/authorize-community-edit.jsp
- Moved to dspace-admin and changed: dspace/jsp/admin/authorize-item-edit.jsp
- Moved to dspace-admin and changed: dspace/jsp/admin/authorize-main.jsp
- Moved to dspace-admin and changed: dspace/jsp/admin/authorize-policy-edit.jsp
- Moved to dspace-admin: dspace/jsp/admin/collection-select.jsp
- Moved to dspace-admin: dspace/jsp/admin/community-select.jsp
- Moved to dspace-admin: dspace/jsp/admin/confirm-delete-collection.jsp
- Moved to dspace-admin: dspace/jsp/admin/confirm-delete-community.jsp
- Moved to dspace-admin: dspace/jsp/admin/confirm-delete-dctype.jsp
- Moved to dspace-admin: dspace/jsp/admin/confirm-delete-eperson.jsp
- Moved to dspace-admin: dspace/jsp/admin/confirm-delete-format.jsp
- Moved to dspace/jsp/tools: dspace/jsp/admin/confirm-delete-item.jsp
- Moved to dspace/jsp/tools: dspace/jsp/admin/confirm-withdraw-item.jsp
- Moved to dspace-admin and changed: dspace/jsp/admin/edit-collection.jsp
- Moved to dspace-admin and changed: dspace/jsp/admin/edit-community.jsp
- Moved to dspace/jsp/tools and changed: dspace/jsp/admin/edit-item-form.jsp
- Moved to dspace-admin and changed: dspace/jsp/admin/eperson-browse.jsp
- Moved to dspace-admin: dspace/jsp/admin/eperson-confirm-delete.jsp
- Moved to dspace-admin and changed: dspace/jsp/admin/eperson-edit.jsp
- Moved to dspace-admin and changed: dspace/jsp/admin/eperson-main.jsp
- Moved to dspace/jsp/tools and changed: dspace/jsp/admin/get-item-id.jsp
- Moved to dspace/jsp/tools and changed: dspace/jsp/admin/group-edit.jsp
- Moved to dspace-admin and changed: dspace/jsp/admin/group-eperson-select.jsp
- Moved to dspace/jsp/tools and changed: dspace/jsp/admin/group-list.jsp
- Moved to dspace-admin: dspace/jsp/admin/index.jsp
- Moved to dspace-admin and changed: dspace/jsp/admin/item-select.jsp
- Moved to dspace-admin and changed: dspace/jsp/admin/list-communities.jsp
- Moved to dspace-admin and changed: dspace/jsp/admin/list-dc-types.jsp
- Removed: dspace/jsp/admin/list-epeople.jsp
- Moved to dspace-admin and changed: dspace/jsp/admin/list-formats.jsp
- Moved to dspace/jsp/tools: dspace/jsp/admin/upload-bitstream.jsp
- Moved to dspace-admin and changed: dspace/jsp/admin/upload-logo.jsp
- Moved to dspace-admin: dspace/jsp/admin/workflow-abort-confirm.jsp
- Moved to dspace-admin and changed: dspace/jsp/admin/workflow-list.jsp
- Changed: dspace/jsp/browse/authors.jsp
- Changed: dspace/jsp/browse/items-by-author.jsp
- Changed: dspace/jsp/browse/items-by-date.jsp
- Changed: dspace/jsp/browse/no-results.jsp
- New: dspace-admin/eperson-deletion-error.jsp
- New: dspace/jsp/dspace-admin/news-edit.jsp
- New: dspace/jsp/dspace-admin/news-main.jsp
- New: dspace/jsp/dspace-admin/wizard-basicinfo.jsp
- New: dspace/jsp/dspace-admin/wizard-default-item.jsp
- New: dspace/jsp/dspace-admin/wizard-permissions.jsp

- New: dspace/jsp/dspace-admin/wizard-questions.jsp
- Changed: dspace/jsp/components/contact-info.jsp
- Changed: dspace/jsp/error/internal.jsp
- New: dspace/jsp/help/formats.jsp
- Changed: dspace/jsp/layout/footer-default.jsp
- Changed: dspace/jsp/layout/header-default.jsp
- Changed: dspace/jsp/layout/navbar-admin.jsp
- Changed: dspace/jsp/layout/navbar-default.jsp
- Changed: dspace/jsp/login/password.jsp
- Changed: dspace/jsp/mydspace/main.jsp
- Changed: dspace/jsp/mydspace/perform-task.jsp
- Changed: dspace/jsp/mydspace/preview-task.jsp
- Changed: dspace/jsp/mydspace/reject-reason.jsp
- Changed: dspace/jsp/mydspace/remove-item.jsp
- Changed: dspace/jsp/register/edit-profile.jsp
- Changed: dspace/jsp/register/inactive-account.jsp
- Changed: dspace/jsp/register/new-password.jsp
- Changed: dspace/jsp/register/registration-form.jsp
- Changed: dspace/jsp/search/advanced.jsp
- Changed: dspace/jsp/search/results.jsp
- Changed: dspace/jsp/submit/cancel.jsp
- New: dspace/jsp/submit/cc-license.jsp
- Changed: dspace/jsp/submit/choose-file.jsp
- New: dspace/jsp/submit/creative-commons.css
- New: dspace/jsp/submit/creative-commons.jsp
- Changed: dspace/jsp/submit/edit-metadata-1.jsp
- Changed: dspace/jsp/submit/edit-metadata-2.jsp
- Changed: dspace/jsp/submit/get-file-format.jsp
- Changed: dspace/jsp/submit/initial-questions.jsp
- Changed: dspace/jsp/submit/progressbar.jsp
- Changed: dspace/jsp/submit/review.jsp
- Changed: dspace/jsp/submit/select-collection.jsp
- Changed: dspace/jsp/submit/show-license.jsp
- Changed: dspace/jsp/submit/show-uploaded-file.jsp
- Changed: dspace/jsp/submit/upload-error.jsp
- Changed: dspace/jsp/submit/upload-file-list.jsp
- Changed: dspace/jsp/submit/verify-prune.jsp
- New: dspace/jsp/tools/edit-item-form.jsp
- New: dspace/jsp/tools/eperson-list.jsp
- New: dspace/jsp/tools/itemmap-browse.jsp
- New: dspace/jsp/tools/itemmap-info.jsp
- New: dspace/jsp/tools/itemmap-main.jsp

Java file changes between 1.1 and 1.2

This list generated with `cvs -Q rdiff -s -r dspace-1_1 dspace` and a sprinkling of perl.

- New: dspace/src/org/dspace/administer/CommunityFiliator.java
- Changed: dspace/src/org/dspace/administer/CreateAdministrator.java
- Changed: dspace/src/org/dspace/administer/DCType.java
- New: dspace/src/org/dspace/administer/Upgrade11To12.java
- Changed: dspace/src/org/dspace/app/itemexport/ItemExport.java

- Changed: dspace/src/org/dspace/app/itemimport/ItemImport.java
- New: dspace/src/org/dspace/app/mediafilter/HTMLFilter.java
- New: dspace/src/org/dspace/app/mediafilter/JPEGFilter.java
- New: dspace/src/org/dspace/app/mediafilter/MediaFilter.java
- New: dspace/src/org/dspace/app/mediafilter/MediaFilterManager.java
- New: dspace/src/org/dspace/app/mediafilter/PDFFilter.java
- New: dspace/src/org/dspace/app/mediafilter/WordFilter.java
- New: dspace/src/org/dspace/app/mets/METSExport.java
- Changed: dspace/src/org/dspace/app/oai/DSpaceOAICatalog.java
- Changed: dspace/src/org/dspace/app/oai/DSpaceRecordFactory.java
- Changed: dspace/src/org/dspace/app/oai/OAIDCCrosswalk.java
- Changed: dspace/src/org/dspace/app/webui/filter/AdminOnlyFilter.java
- Changed: dspace/src/org/dspace/app/webui/filter/RegisteredOnlyFilter.java
- Changed: dspace/src/org/dspace/app/webui/jsptag/IncludeTag.java
- Changed: dspace/src/org/dspace/app/webui/jsptag/ItemListTag.java
- Changed: dspace/src/org/dspace/app/webui/jsptag/ItemTag.java
- Changed: dspace/src/org/dspace/app/webui/jsptag/LayoutTag.java
- Changed: dspace/src/org/dspace/app/webui/jsptag/PopupTag.java
- New: dspace/src/org/dspace/app/webui/jsptag/SelectEPersonTag.java
- Changed: dspace/src/org/dspace/app/webui/servlet/AdvancedSearchServlet.java
- New: dspace/src/org/dspace/app/webui/servlet/BitstreamServlet.java
- Changed: dspace/src/org/dspace/app/webui/servlet/CommunityListServlet.java
- Changed: dspace/src/org/dspace/app/webui/servlet/HandleServlet.java
- New: dspace/src/org/dspace/app/webui/servlet/HTMLServlet.java
- New: dspace/src/org/dspace/app/webui/servlet/LoadDSpaceConfig.java
- Changed: dspace/src/org/dspace/app/webui/servlet/RegisterServlet.java
- Changed: dspace/src/org/dspace/app/webui/servlet/SimpleSearchServlet.java
- Changed: dspace/src/org/dspace/app/webui/servlet/SubmitServlet.java
- Changed: dspace/src/org/dspace/app/webui/servlet/admin/AuthorizeAdminServlet.java
- Changed: dspace/src/org/dspace/app/webui/servlet/admin/BitstreamFormatRegistry.java
- New: dspace/src/org/dspace/app/webui/servlet/admin/CollectionWizardServlet.java
- Changed: dspace/src/org/dspace/app/webui/servlet/admin/DCTypeRegistryServlet.java
- Changed: dspace/src/org/dspace/app/webui/servlet/admin/EditCommunitiesServlet.java
- Removed: dspace/src/org/dspace/app/webui/servlet/admin/EditEPersonServlet.java
- Changed: dspace/src/org/dspace/app/webui/servlet/admin/EditItemServlet.java
- Changed: dspace/src/org/dspace/app/webui/servlet/admin/EPersonAdminServlet.java
- New: dspace/src/org/dspace/app/webui/servlet/admin/EPersonListServlet.java
- Changed: dspace/src/org/dspace/app/webui/servlet/admin/GroupEditServlet.java
- New: dspace/src/org/dspace/app/webui/servlet/admin/ItemMapServlet.java
- New: dspace/src/org/dspace/app/webui/servlet/admin/NewsEditServlet.java
- Changed: dspace/src/org/dspace/app/webui/servlet/admin/WorkflowAbortServlet.java
- Changed: dspace/src/org/dspace/app/webui/util/Authenticate.java
- Changed: dspace/src/org/dspace/app/webui/util/FileUploadRequest.java
- Changed: dspace/src/org/dspace/app/webui/util/JSPManager.java
- Changed: dspace/src/org/dspace/app/webui/util/UIUtil.java
- Changed: dspace/src/org/dspace/authorize/AuthorizeManager.java
- Changed: dspace/src/org/dspace/authorize/FixDefaultPolicies.java
- Changed: dspace/src/org/dspace/authorize/PolicySet.java
- Changed: dspace/src/org/dspace/authorize/ResourcePolicy.java

- Changed: dspace/src/org/dspace/browse/Browse.java
- Changed: dspace/src/org/dspace/content/Bitstream.java
- Changed: dspace/src/org/dspace/content/Bundle.java
- Changed: dspace/src/org/dspace/content/Collection.java
- Changed: dspace/src/org/dspace/content/Community.java
- Changed: dspace/src/org/dspace/content/DSpaceObject.java
- Changed: dspace/src/org/dspace/content/InProgressSubmission.java
- Changed: dspace/src/org/dspace/content/InstallItem.java
- Changed: dspace/src/org/dspace/content/Item.java
- Changed: dspace/src/org/dspace/content/WorkspaceItem.java
- Changed: dspace/src/org/dspace/core/ConfigurationManager.java
- Changed: dspace/src/org/dspace/core/Constants.java
- Changed: dspace/src/org/dspace/core/Utils.java
- Changed: dspace/src/org/dspace/eperson/EPerson.java
- New: dspace/src/org/dspace/eperson/EPersonDeletionException.java
- Changed: dspace/src/org/dspace/eperson/Group.java
- New: dspace/src/org/dspace/license/CreativeCommons.java
- Changed: dspace/src/org/dspace/search/DSIndexer.java
- Changed: dspace/src/org/dspace/search/DSQuery.java
- Changed: dspace/src/org/dspace/search/Harvest.java
- Changed: dspace/src/org/dspace/search/HarvestedItemInfo.java
- Changed: dspace/src/org/dspace/search/QueryArgs.java
- Changed: dspace/src/org/dspace/search/QueryResults.java
- Changed: dspace/src/org/dspace/storage/rdbms/DatabaseManager.java
- Changed: dspace/src/org/dspace/storage/rdbms/TableRow.java
- Changed: dspace/src/org/dspace/workflow/WorkflowItem.java
- Changed: dspace/src/org/dspace/workflow/WorkflowManager.java

Changes in DSpace 1.1.1

Bug fixes

- non-administrators can now submit again
- installations now preserve file creation dates, eliminating confusion with upgrades
- authorization editing pages no longer create null entries in database, and no longer handles them poorly (no longer gives blank page instead of displaying policies.)
- registration page Invalid token error page now displayed when an invalid token is received (as opposed to internal server error.) Fixes SF bug #739999
- eperson admin 'recent submission' links fixed for DSpaces deployed somewhere other than at / (e.g. /dspace).
- help pages Link to help pages now includes servlet context (e.g. '/dspace'). Fixes SF bug #738399.

Improvements

- `bin/dspace-info.pl` now checks jsp and asset store files for zero-length files
- `make-release-package` now works with SourceForge CVS
- eperson editor now doesn't display the spurious text 'null'
- item exporter now uses Jakarta's cli command line arg parser (much cleaner)
- item importer improvements:
    - now uses Jakarta's cli command line arg parser (much cleaner)

    - imported items can now be routed through a workflow
    - more validation and error messages before import
    - can now use email addresses and handles instead of just database IDs
    - can import an item to a collection with the workflow suppressed

Changes in DSpace 1.1

- Fixed various OAI-related bugs; DSpace's OAI support should now be correct. Note that harvesting is now based on the new Item 'last modified' date (as opposed to the Dublin Core `date.available` date.)
- Fixed Handle support--DSpace now responds to naming authority requests correctly.
- Multiple bitstream stores can now be specified; this allows DSpace storage to span several disks, and so there is no longer a hard limit on storage.
- Search improvements:
    - New fielded searching UI
    - Search results are now paged
    - Abstracts are indexed
    - Better use of Lucene API; should stop the number of open file handles getting large
- Submission UI improvements:
    - now insists on a title being specified
    - fixed navigation on file upload page
    - citation & identifier fields for previously published submissions now fixed
- Many Unicode fixes to the database and Web user interface
- Collections can now be deleted
- Bitstream descriptions (if available) displayed on item display page
- Modified a couple of servlets to handle invalid parameters better (i.e. to report a suitable error message instead of an internal server error)
- Item templates now work
- Fixed registration token expiration problem (they no longer expire.)