Application Layer

The following explains how the application layer is built and used.

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. As of Dspace 1.3.2 the UI meets both XHTML 1.0 standards and Web Accessibility Initiative (WAI) level-2 standard.

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 administrators.

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 1.5, the deployment has changed. The build systems has moved to a maven-based system enabling the various projects (JSPUI, XMLUI, etc.) into separate projects. The system still uses the familar 'Ant' to deploy the webapps in later stages.

Location	Description
[dspace-source]/dspace-jspui/dspace-jspui-api/src/main/java/org/dspace/app/webui	Web UI source files
[dspace-source]/dspace-jspui/dspace-jspui-api/src/main/java/org/dspace/app/filters	Servlet Filters (Servlet 2.3 spec)
[dspace-source]/dspace-jspui/dspace-jspui-api/src/main/java/org/dspace/app/jsptag	Custom JSP tag class files
[dspace-source]/dspace-jspui/dspace-jspui-api/src/main/java/org/dspace/app/servlet	Servlets for main Web UI (controllers)
[dspace-source]/dspace-jspui/dspace-jspui-api/src/main/java/org/dspace/app/servlet/admin	Servlets that comprise the administration part of the Web UI
[dspace-source]/dspace-jspui/dspace-jspui-api/src/main/java/org/dspace/app/webui/util/	Miscellaneous classes used by the servlets and filters
[dspace-source]/dspace-jspui	The JSP files
[dspace-source]/dspace/modules/jspui/src/main/webapp	This is where you place customized versions of JSPs, see JSPUI Configuration and Customization
[dspace-source]/dspace/modules/xmlui/src/main/webapp	This is where you place customizations for the Manakin interface, see XMLUI Configuration and Customization
[dspace-source/dspace/modules/jspui/src/main/resources	This is where you can place you customize version of the *Messages.properties* file.
[dspace-source]/dspace-jspui/dspace-jspui-webapp/src/main/webapp/WEB-INF/dspace-tags.tld	Custom DSpace JSP tag descriptor

The Build Process

The DSpace Maven build process constructs a full DSpace installation template directory structure containing a series of web applications. The results are placed in *[dspace-source]/dspace/target/dspace-[version]-build.dir/*. The process works as follows:

- All the DSpace source code is compiled, and/or automatically downloaded from the Maven Central code/libraries repository.
- A full DSpace "installation template" folder is built in `[dspace-source]/dspace/target/dspace-[version]-build.dir/`
    - This DSpace "installation template" folder has a structure identical to the Installed Directory Layout

In order to then install & deploy DSpace from this "installation template" folder, you must run the following from `[dspace-source]/dspace/target/dspace-[version]-build.dir/` :

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

Please see the Installation instructions for more details about the Installation process.

Servlets and JSPs (JSPUI Only)

The JSPUI 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.

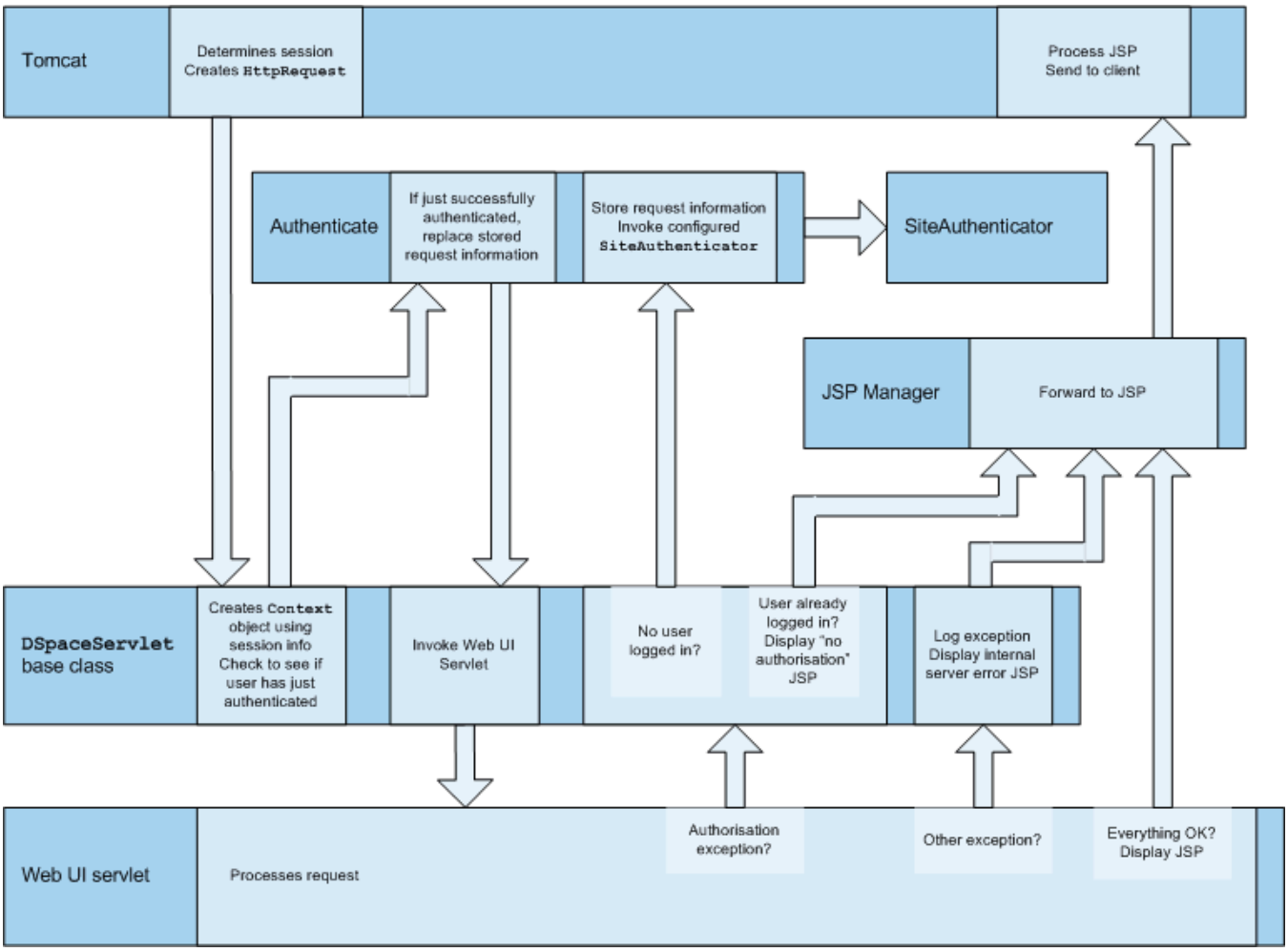
There is an exception to this servlet/JSP style: *index.jsp*, the 'home page', receives the HTTP request directly from Tomcat without a servlet being invoked first. This is because in the servlet 2.3 specification, there is no way to map a servlet to handle only requests made to '/'; such a mapping results 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.

[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 (*SubmissionController* is a prime example of a servlet that deals with the input from many different JSPs. The *step* and *page* hidden parameters (written out by the *SubmissionController.getSubmissionParameters()* method) are used to inform the servlet which page of which step has just been filled out (i.e. which page of the 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.



Flow of Control During HTTP Request Processing

Custom JSP Tags (JSPUI Only)

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*.

Internationalization (JSPUI Only)

XMLUI Internationalization

For information about XMLUI Internationalization please see: XMLUI Multilingual Support.

The Java Standard Tag Library v1.0 is used to specify messages in the JSPs like this:

OLD:

```
<H1>Search Results</H1>
```

NEW:

```
<H1><fmt:message key="jsp.search.results.title"/></H1>
```

This message can now be changed using the *config/language-packs/Messages.properties* file. (This must be done at build-time: *Messages.properties* is placed in the *dspace.war* Web application file.)

```
jsp.search.results.title = Search Results
```

Phrases may have parameters to be passed in, to make the job of translating easier, reduce the number of 'keys' and to allow translators to make the translated text flow more appropriately for the target language.

OLD:

```
<P>Results <%= r.getFirst() %> to <%= r.getLast() %> of <%=r.getTotal() %></P>
```

NEW:

```
<fmt:message key="jsp.search.results.text">
  <fmt:param><%= r.getFirst() %></fmt:param>
  <fmt:param><%= r.getLast() %></fmt:param>
  <fmt:param><%= r.getTotal() %></fmt:param>
</fmt:message>
```

(Note: JSTL 1.0 does not seem to allow JSP <%= %> expressions to be passed in as values of attribute in <fmt:param value=""/>)

The above would appear in the *Messages_xx.properties* file as:

```
jsp.search.results.text = Results {0}-{1} of {2}
```

Introducing number parameters that should be formatted according to the locale used makes no difference in the message key compared to string parameters:

```
jsp.submit.show-uploaded-file.size-in-bytes = {0} bytes
```

In the JSP using this key can be used in the way belov:

```
<fmt:message key="jsp.submit.show-uploaded-file.size-in-bytes">
  <fmt:param><fmt:formatNumber><%= bitstream.getSize()%></fmt:formatNumber></fmt:param>
</fmt:message>
```

(Note: JSTL offers a way to include numbers in the message keys as *jsp.foo.key = {0,number} bytes*. Setting the parameter as *<fmt:param value="${variable}" />* workes when *variable* is a single variable name and doesn't work when trying to use a method's return value instead: *bitstream.getSize()*. Passing the number as string (or using the <%= %> expression) also does not work.)

Multiple *Messages.properties* can be created for different languages. See ResourceBundle.getBundle. e.g. you can add German and Canadian French translations:

```
Messages_de.properties
Messages_fr_CA.properties
```

The end user's browser settings determine which language is used. The English language file *Messages.properties* (or the default server locale) will be used as a default if there's no language bundle for the end user's preferred language. (Note that the English file is not called *Messages_en.properties* – this is so it is always available as a default, regardless of server configuration.)

The *dspace:layout* tag has been updated to allow dictionary keys to be passed in for the titles. It now has two new parameters: *titlekey* and *parenttitlekey*. So where before you'd do:

```
<dspace:layout title="Here"
               parentlink="/mydspace"
               parenttitle="My DSpace">
```

You now do:

```
<dspace:layout titlekey="jsp.page.title"
               parentlink="/mydspace"
               parenttitlekey="jsp.mydspace">
```

And so the layout tag itself gets the relevant stuff out of the dictionary. *title* and *parenttitle* still work as before for backwards compatibility, and the odd spot where that's preferable.

Message Key Convention

When translating further pages, please follow the convention for naming message keys to avoid clashes.

For text in JSPs use the complete path + filename of the JSP, then a one-word name for the message. e.g. for the title of *jsp/mydspace/main.jsp* use:

```
jsp.mydspace.main.title
```

Some common words (e.g. "Help") can be brought out into keys starting *jsp.* for ease of translation, e.g.:

```
jsp.admin = Administer
```

Other common words/phrases are brought out into 'general' parameters if they relate to a set (directory) of JSPs, e.g.

```
jsp.tools.general.delete = Delete
```

Phrases that relate **strongly** to a topic (eg. MyDSpace) but used in many JSPs outside the particular directory are more convenient to be cross-referenced. For example one could use the key below in *jsp/submit/saved.jsp* to provide a link back to the user's *MyDSpace*:

*(Cross-referencing of keys **in general** is not a good idea as it may make maintenance more difficult. But in some cases it has more advantages as the meaning is obvious.)*

```
jsp.mydspace.general.goto-mydspace = Go to My DSpace
```

For text in servlet code, in custom JSP tags or wherever applicable use the fully qualified classname + a one-word name for the message. e.g.

```
org.dspace.app.webui.jsptag.ItemListTag.title = Title
```

Which Languages are currently supported?

To view translations currently being developed, please refer to the i18n page of the DSpace Wiki.

HTML Content in Items

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.

HTML documents must be "self-contained", as explained here. Provided that full path information is known by DSpace, any depth or complexity of HTML document can be served subject to those constraints. This is usually possible with some kind of batch import. If, however, the document has been uploaded one file at a time using the Web UI, the path information has been stripped. The system can cope with relative links that refer to a deeper path, e.g.

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

If the item has been uploaded via the Web submit UI, in the Bitstream table in the database we have the 'name' field, which will contain the filename with no path (*figure1.gif*). 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*

To prevent "infinite URL spaces" appearing (e.g. if a file *foo.html* linked to *bar/foo.html*, which would link to *bar/bar/foo.html*...) this behavior can be configured by setting the configuration property *webui.html.max-depth-guess*.

For example, if we receive a request for *foo/bar/index.html*, and we have a bitstream called just *index.html*, we will serve up that bitstream for the request if *webui.html.max-depth-guess* is 2 or greater. If *webui.html.max-depth-guess* is 1 or less, we would not serve that bitstream, as the depth of the file is greater. If *webui.html.max-depth-guess* is zero, the request filename and path must always exactly match the bitstream name. The default value (if that property is not present in *dspace.cfg*) is 3.

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/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 *oai*, for example:

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

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

```
http://dspace.myu.edu/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 *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 export is enabled by default; 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.

Other metadata formats are supported as well, using other *Crosswalk* implementations; consult the *oaicat.properties* file described below. To enable a format, simply uncomment the lines beginning with *Crosswalks.**. Multiple formats are allowed, and the current list includes, in addition to unqualified DC: MPEG DIDL, METS, MODS. There is also an incomplete, experimental qualified DC.

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 is one main configuration file relevant to OAI-PMH support:

- **oaicat.properties**: This file resides in `[dspace]/config`. You probably won't need to edit this, as it is pre-configured to meet most needs. You might want to change the `Identify.earliestDatestamp` field to more accurately reflect the oldest datestamp in your local DSpace system. (Note that this is the value of the `last_modified` column in the `Item` database table.)

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.)

DSpace Command Launcher

Introduced in Release 1.6, the DSpace Command Launcher brings together the various command and scripts into a standard-practice for running CLI runtime programs.

Older Versions

Prior to Release 1.6, there were various scripts written that masked a more manual approach to running CLI programs. The user had to issue *[dspace]/bin/dsrun* and then java class that ran that program. With release 1.5, scripts were written to mask the *[dspace]/bin/dsrun* command. We have left the java class in the System Administration section since it does have value for debugging purposes and for those who wish to learn about DSpace programming or wish to customize the code at any time.

Command Launcher Structure

There are two components to the command launcher: the dspace script and the launcher.xml. The DSpace command calls a java class which in turn refers to *launcher.xml* that is stored in the *[dspace]/config* directory

launcher.xml is made of several components:

- <*command*> begins the stanza for a command
- <*name*>*_name of command_*</*name*> the name of the command that you would use.
- <*description*>*_the description of the command_*</*description*>
- <*step*> </*step*> User arguments are parsed and tested.
- <*class*>*_<the java class that is being used to run the CLI program>_*</*class*>
  Prior to release 1.5 if one wanted to regenerate the browse index, one would have to issue the following commands manually:

  ```
  [dspace]/bin/dsrun org.dspace.browse.IndexBrowse -f -r
  [dspace]/bin/dsrun org.dspace.browse.ItemCounter
  [dspace]/bin/dsrun org.dspace.search.DSIndexer
  ```

  In release 1.5 a script was written and in release 1.6 the command *[dspace]/bin/dspace index-init* replaces the script. The stanza from *launcher.xml* show us how one can build more commands if needed:

  ```
  <command>
          <name>index-update</name>
          <description>Update the search and browse indexes</description>
          <step passuserargs="false">
              <class>org.dspace.browse.IndexBrowse</class>
              <argument>-i</argument>
          </step>
          <step passuserargs="false">
              <class>org.dspace.browse.ItemCounter</class>
          </step>
          <step passuserargs="false">
              <class>org.dspace.search.DSIndexer</class>
          </step>
  </command>
  ```