

BuildCookbook

This is a collection of cookbook-style examples showing how to add certain types of modifications to a DSpace installation. It is based on experience with the DSpace 1.5.0 binary release, the first to use the new module layout and build system.

Please feel free to add your own examples of typical DSpace modifications.

Prerequisites

You must be familiar with [installing](#) and configuring DSpace 1.5. It may help to review [Building DSpace From Source](#).

See also the presentation [Customizing DSpace 1.5 with Basic Overlays](#) from [Open Repositories O8](#), by [Tim Donohue](#) and [Graham Triggs](#). It provides an excellent overview of the build process and how overlays work. It was the inspiration for this page.

This page concentrates on examples not covered in that presentation, such as the common case of just adding some plugin implementations.

Recipe 1: Adding Local Code as a Maven Project

Customizations to DSpace often require the addition of *local* code, i.e. classes in non-DSpace packages. For example, when you wish to provide your own AuthenticationMethod implementation. By putting this code into a separate Maven project, and modifying a few DSpace POMs, you can have your local classes added to both the webapps of interest *and* the DSpace command-line applications.

The trick is to manage dependencies correctly. Your local project will depend explicitly on the `dspace-api` project so it can import DSpace API interfaces and classes. Thus, the `dspace-api` project *cannot* depend on it. You have to add a dependency on your local project to every POM that generates an executable build product (i.e. the webapps, and the command-line apps). Those projects *also* depend on `dspace-api` so the API classes will be available.

Example of Local Code Project

Here is a live example of how the DSpace 1.5.2 POMs were modified to add a local package named `dash-api`. This project contains several plugin implementations, and it has its own dependencies on some other projects which support that code. All of the classes in `dash-api` are in packages under the `edu.harvard...` hierarchy, so they do not conflict with DSpace.

Note that the POMs are presented in a very simplified form, eliding the elements like *repositories* that depend on your local environment or can be easily extrapolated. We only show the elements pertinent to this discussion:

First, establish `dash-api` POM

Create a `dash-api` subdirectory at the same level as `dspace-api`, and add this POM:

```

<project>
  <groupId>edu.harvard.hul.ois.dash</groupId>
  <artifactId>dash-api</artifactId>
  <packaging>jar</packaging>

  <parent>
    <groupId>org.dspace</groupId>
    <artifactId>dspace-parent</artifactId>
    <version>1.5.2-SNAPSHOT</version>
  </parent>

  <dependencies>
    <dependency>
      <groupId>org.dspace</groupId>
      <artifactId>dspace-api</artifactId>
    </dependency>
    ...other dspace dependencies...
    <dependency>
      <groupId>edu.harvard.hul.ois</groupId>
      <artifactId>util</artifactId>
    </dependency>
    <dependency>
      <groupId>commons-httpclient</groupId>
      <artifactId>commons-httpclient</artifactId>
    </dependency>
    ....other external dependencies...
  </dependencies>
</project>

```

Modify the dspace POM

Add the following sections to the POM of the `dspace` project. This ensures the code gets built, and that it will be available to DSpace command-line utilities.

Add to profiles element

```

<profile>
  <id>dash-api</id>
  <activation>
    <file>
      <exists>../dash-api/pom.xml</exists>
    </file>
  </activation>
  <modules>
    <module>../dash-api</module>
  </modules>
</profile>

```

Add to dependencies element

```

<dependency>
  <groupId>edu.harvard.hul.ois.dash</groupId>
  <artifactId>dash-api</artifactId>
</dependency>

```

Modify POM for each relevant Webapp

For each webapp you plan to build, e.g. `xmlui`, add the following elements to the POM in its subdirectory under `dspace/modules/`. For example, add to `dspace/modules/xmlui/pom.xml` this element to its dependencies list:

```
<dependency>
  <groupId>edu.harvard.hul.ois.dash</groupId>
  <artifactId>dash-api</artifactId>
</dependency>
```

Modify the dspace-parent POM

Finally, you must add a version declaration for dash-api to the dspace-parent POM. Add this element to the dependencies list:

```
<dependency>
  <groupId>edu.harvard.hul.ois.dash</groupId>
  <artifactId>dash-api</artifactId>
  <version>${project.version}</version>
</dependency>
```

Other Dependencies

Don't forget to add version declarations for other new dependencies if they are not explicit in e.g. the dash-api POM.

Recipe 2: Adding Customizations to Webapps with Overlays

The excellent presentation [Customizing DSpace 1.5 with Basic Overlays](#) from [Open Repositories O8](#), by [Tim Donohue](#) and [Graham Triggs](#) shows how to manage your local UI customizations and keep your added code well-segregated from the DSpace codebase so upgrades are simplified. It also provides an overview of the build process and how maven overlays work.

Recipe 3: (Deprecated) Old Quick-and-Dirty Method of Adding Plugins

This is a slightly simpler but less powerful way to add code to a single webapp at a time. It has drawbacks over the local-API method described in the first section.

With the overlay mechanism, it is easy to add a [Crosswalk](#) or [PackagerPlugins](#) plugin to **one of the webapps**. This technique does *not* let you add code to the libraries accessed by the command-line utilities, such as [dspace/bin/packager](#) (more about that later).

The following notes assume a *binary* installation of DSpace 1.5.0, under the directories:

- **[source]** is the "source" directory where builds are done.
- **[dspace]** is the target runtime directory, e.g. /dspace

Procedure to Add a Plugin

Step 1: Install sources

Add the necessary Java source files to the *overlay directory* for each module that you want to have access to the plugin. This is `[source]/dspace/modules/{MODULE}/src/main/java` for additional Java sources.

Note that adding a plugin to multiple modules requires a separate copy of the source files for each module, which might complicate maintenance when you have to update the sources; use symbolic links to work around this if you are familiar with them.

For this example, I'll show an ingestion/dissemination crosswalk Plugin, and a package ingester Plugin that are both implemented in the following class files:

```
edu/mit/libraries/facade/PIMConstants.java
edu/mit/libraries/facade/PIMCrosswalk.java
edu/mit/libraries/facade/PIMMETSIngester.java
```

For the example, assume these files reside under a *development* directory, `{development}`.

To add these classes to the LNI module, we install the sources under `[source]/dspace/modules/lni/src/main/java` with the following commands:

```
mkdir -p [source]/dspace/modules/lni/src/main/java/edu/mit/libraries/facade
cp {development}/edu/mit/libraries/facade/*.java [source]/dspace/modules/lni/src/main/java/edu/mit/libraries/facade
```

Step 2: Update DSpace Configuration

If you maintain the DSpace configuration file in your source directory and use the build tools to copy it into the runtime hierarchy, then update the source copy of `dspace.cfg` now. (In my development environment, I just edit the runtime copy in `[dspace]/config/dspace.cfg`.)

Add entries for the crosswalks, e.g. like the **bold** line here (other entries elided for clarity):

```
# Crosswalk Plugins:
plugin.named.org.dspace.content.crosswalk.IngestionCrosswalk = \
    edu.mit.libraries.facade.PIMCrosswalk = PIM \
    org.dspace.content.crosswalk.PREMISCrosswalk = PREMIS \
    ...
```

Step 3: Modify the POM to Add Dependencies

If your code has any new external dependencies (i.e. it needs modules not already required by DSpace) then you need to add those to the POM for the overlay module. In this example, we add the dependency lines to the LNI module's POM at `[source]/dspace/modules/lni/pom.xml`

```
<project>
...
<dependencies>
...
<dependency>
  <groupId>org.openrdf</groupId>
  <artifactId>sesame</artifactId>
  <version>2.1</version>
</dependency>
</dependencies>
</project>
```

NOTE: Of course, this requires that all the libraries your code depends on are available to Maven. If not, you'll have to add them to the local Maven repository or convince someone to put them into a networked maven repository. This example creates an entry in the local repository:

```
mvn install:install-file \
  -Dfile=/opt/sesame/lib/openrdf-sesame-2.1-onejar.jar \
  -DgroupId=org.openrdf \
  -DartifactId=sesame \
  -Dversion=2.1 \
  -Dpackaging=jar \
  -DgeneratePom=true
```

Step 4: Build with Maven and Deploy

First, build the sources:

```
cd [source]/dspace
mvn package
```

Assuming that succeeds, run Ant to install the build products.

NOTE: This does NOT install the configuration files, because I don't work that way; perhaps someone who does could add an alternate command here?

```
{ shut down servlet container such as Tomcat }
cd [source]/dspace/target/dspace-1.5.0-build.dir
ant update
{ start up servlet container such as Tomcat }
```

Your DSpace instance should now be running with the new plugins in the LNI application.

Adding The Same Plugins to Other Applications

The procedure to add these same plugins to another DSpace application, for example the OAI-PMH server ("oai"), is identical.

If you are adding the plugins to both `lmi` and `oai`, you may wish to symbolically link the Java sources to one master copy someplace else, so that any changes will take effect in both applications.

In the case of OAI-PMH, you'll also need to modify the `oaicat.properties` configuration file to add the appropriate plugins to OAICAT.

Recipe 4: (Deprecated) Use Overlays to Segregate Local Modifications

You can also use the overlay mechanism to implement a local change or bug-fix to the DSpace codebase. The process is exactly the same as for adding plugin implementations, only you add the appropriate DSpace class files to the source directory instead. These will take precedence over the distributed code in the classloader.

Again, add the sources under `[source]/dspace/modules/{MODULE}/src/main/java`, only under the `org/dspace/...` hierarchy.

For example, to fix a bug in the `org.dspace.app.oai.DSpaceOAICatalog` class, you add *that* file in Step 1 instead of your own source:

```
mkdir -p [source]/dspace/modules/oai/src/main/java/org/dspace/app/oai
cp {development}/org/dspace/app/oai/DSpaceOAICatalog.java [source]/dspace/modules/oai/src/main/java/org/dspace/app/oai
```

The procedure thereafter is exactly the same as the last recipe (deprecated) for adding plugin implementations.

Recipe 5: (Deprecated) Old, Poor way of Adding Plugins to Command-Line Applications

The build system does not appear to have any way to accomplish this with a binary DSpace installation.
(Please correct this if I'm wrong.)

As a kludgy workaround, I've simply added a JAR file manually to the "lib" directory used by all command-line apps.

Using my crosswalk and packager example above, the command to add my code to the runtime directory is:

```
cd [source]/dspace
jar cvf [dspace]/lib/pim.jar \
  -C target/dspace-1.5.0-build.dir/webapps/oai/WEB-INF/classes/ edu
```

Note that the JAR output file `pim.jar` is simply what I chose to call it, use any unique name.
The classes are all in packages under `edu.mit` so the `jar` command picks up everything under `edu` in the overlay module's class directory.

Of course you *also* have to manually copy in whatever other libraries your code depends on, e.g.

```
cp /opt/sesame/lib/*.jar [dspace]/lib
```

Note that the build installation ("ant update") process wipes the runtime "lib" directory clean each time, so you'll have to repeat these commands after every new update.