PackagerPlugins

Contents

Pluggable Package Importer and Exporter

This page proposes a new core component to import and export packaged content. It defines a plugin interface to let a DSpace instance read and write many kinds of packages. Administrators can add a new package handlers with a simple configuration change.

Renaming note

After conversations with Richard & Larry, various renaming has been done, for consistency and predictable behaviour. The names of the classes now reflect what they are. e.g. the old 'DisseminationPackage' is actually a class that disseminates packages, rather than representing a package itself.

Old Name	New Name
AbstractMetsDisseminatio n	AbstractMETSDisseminato r
AbstractMetsSubmission	AbstractMETSIngester
DisseminationPackage	PackageDisseminator
DSpaceMetsSipExport	DSpaceMETSDisseminator
DSpaceMetsSipImpor	DSpaceMETSIngester
PDFPackage	PDFIngester
SubmissionPackage	PackageIngester

What is a package?

For this discussion, a *package* is a representation of a DSpace Item in a single data stream. DSpace must be able to ingest a new Item from the package representation. The package does not necessarily *contain* every bit of data and metadata within itself; it may include references to external resources which must be fetched separately.

The essential components of a package are:

1. **Metadata:** Descriptive metadata about the item, at least, and optionally administrative and technical metadata (to assist with validation and preservation of the content).
2. **Content:** The digital material making up the contents of the Item.
3. **Identification:** A package does ***not*** necessarily come with a DSpace persistent identifier or location, i.e. Collection membership. The package is an *external* representation, so it might not have any relationship to DSpace. This does not preclude the package from having DSpace identifiers, but they should *not* be expected.

The format of a package can be virtually anything. The components may be physically present or referenced by links. Some examples:

- A self-contained document such as PDF, which includes descriptive metadata.
- An archive format such as Zip or Tar, containing multiple files. The "manifest" or control file, is in one file with a special "magic" name. That manifest lists the other files in the archive, identifying which are content and which are metadata. It may also include descriptive, administrative, and technical metadata about the Item and its constituent files.
- A manifest-type document by itself, containing only metadata and references to content.
- Just a UI referencing the manifest document in the preceding example.

As you see, a package can be *anything* that a *packager plugin* is capable of interpreting into a new DSpace Item.

Uses of a Package

Three types of packages are defined in the http://public.ccsds.org/publications/lueooks.aspx OAIS eference Model, distinguished by their purpose:

- The SIP, or *Submission Information Package*, contains an item to be ingested by the DSpace archive.
- The DIP, or *Dissemination Information Package*, contains an item as it is exported or *disseminated* from the DSpace archive.
- The AIP, or *Archive Information Package*, is the internal representation of an item within DSpace. AIPs are part of the DSpace+2.0 design (see AssetStore).

The packager plugin architecture concentrates on SIPs and DIPs, since we only consider submission (ingest) and dissemination (export) of Items. Note that a SIP is usually produced by some other application so it has no knowledge of DSpace and does not necessarily fit the DSpace data model. The SIP ingester is reponsible for interpreting and translating it. The advantage of defining a package as a single data stream is that it simplifies ingest and export procedures immensely. There is no need to string together the transfers of multiple streams and treat them as one transaction. The single stream transfer fits the HTTP `GET` and `PUT` model of a WebDAV interface like LightweightNetworkInterface. It could add a streamlined package-oriented submission interface to the Web UI as well. By accepting and disseminating packages, DSpace can also connect directly with other applications that handle packaged content, such as a Learning Management Systems (LMS) and Content Management Systems (CMS).

Packaging Standards and Formats

The package importer/exporter has a plugin architecture because there are many existing standards for packaging content, and new ones are constantly being developed and refined. Some sites may also need to support locally-developed or refined package format so it must be easy to add new packagers. Many package standards are very loose and extensible, so they allow a great range of legal expressions. They typically have *profiles* which define a narrower interpretation of the standard and mandate certain content. Each *profile* is effectively a different type of package and usually needs its own specialized Plugin. Some package types we anticipate supporting:

- Various profiles of the IMS Content Package, from the http://www.imsglobal.org/ IMS Global Learning Consortium. A *package* is a Zip archive with an XML manifest named `imsmanifest.xml` at the top level. The manifest contains descriptive, administrative, technical, and structural metadata.
- http://www.loc.gov/standards/mets/ METS is a powerful metadata framework, ideal for encoding a package manifest. The METS standard does not include a package format, just the metadata, so we have build a package around it by adding content files in a Zip archive, similar to the IMSCP package format.
- A variant of the METS "package" format which is simply a bare METS document. It references the content by external links to HTTP resources. DSpace 1.4 includes an package ingester that recognizes the http://cwspace.mit.edu/docs/xsd/METS/SIP/profilev0p9p1/ DSpace METS Submission Information Package profile.

The Packager Plugins

We define two interfaces, `SubmissionPackage` to import (ingest) SIPs and `DisseminationPackage` to export (disseminate) DIPs.

Configuration

Packager plugins are managed as *named plugins* by the `PluginManager`. See its wiki page for details about how these are configured. The name of a packager plugin is an arbitrary label that identifies the type of package it handles. If an importer and exporter for the same package type are available they should have the same plugin name.

Runtime Options

Each of the `ingest()` and `disseminate()` methods that processes a package also takes an "options" parameter which is a `PackageParameters` object. This object is a list of attribute-value pairs – an extension of the `java.util.Properties` list, which lets an attribute have multiple values so it can represent HTTP query arguments. The caller can use this flexible options list, along with some special knowledge of the packager, to fine-tune the request. For example a request for a METS DIP might use the options to list the types of descriptive metadata to be included, *e.g.* `"dmd=MODS", "dmd=LOM"`. An option might also change the operation of the packager, *e.g.* telling a dissemination package to include all content by reference only instead of the contents of bitstreams, or to render just the metadata sections of the package manifest. Some standard option names and semantics ought to be developed, which apply to *all* package plugins.

Interfaces

First, an Exception class to encapsulate other exceptions encountered while ingesting or disseminating a package. Perhaps it can be subclassed to give a finer-grained view of the type of problem:

```
/**
 * Failure when importing or exporting a package, e.g. unacceptable
 * package format or contents.
 */
public class PackageException extends Exception
\{
\}
```

The ingester plugin has methods to create a new item out of a SIP (`ingest()`), and to overwrite an existing Item with a SIP (`replace()`). Since the package is only expected to describe the Item itself and its contents, other necessary information must be passed as parameters – for example, the Collection in which to create a new Item, or the Item to replace, and the deposit license string to apply by default. The packager implementation is free to override these parameters with information from other sources, e.g. if the SIP contains the persistent identifier of the Item it is replacing, the packager's `replace()` method could ignore its parameter and use the data from the SIP. Since most types of SIPs are completely unrelated to DSpace and have no Collection or Item identifiers in them, the packager needs to have them specified by an external source.

```
/**
 * Plugin to ingest SIPs.
 */
public interface SubmissionPackage extends SelfNamedPlugin
\{
    /**
     * Create new Item out of the ingested package, in the indicated
     * collection.  It creates a workspace item, so the application
     * can choose to enter it in workflow or just install it.
     *
     * @param context - DSpace context.
     * @param collection - collection under which to create new item.
     * @param in - input stream containing package to ingest.
     * @param params Properties-style list of options specific to this packager
     * @param license - may be null, which takes default license.
     * @return workspace item created by ingest.
     * @throws PackageException if package is unacceptable or there is
     *  a fatal error turning it into an Item.
     */
    public WorkspaceItem ingest(Context context, Collection collection,
            InputStream in,  PackageParameters params, String license)
        throws PackageException;
```

```
    /**
     * Replace an existing Item with contents of the ingested package.
     * The packager MAY not implement replace; if it does, the exact
     * action is highly implementation-dependent.
     *
     * @param context - DSpace context.
     * @param item - existing item to be replaced
     * @param in - input stream containing package to ingest.
     * @param params Properties-style list of options specific to this packager
     * @return item re-created by ingest.
     * @throws PackageException if package is unacceptable or there is
     *  a fatal error turning it into an Item.
     * @throws UnsupportedOperationException if this packager does not
     *  implement 'replace'.
     */
    public Item replace(Context context, Item item, InputStream in, PackageParameters params)
        throws PackageException, UnsupportedOperationException;
\}
```

The disseminator plugin to create a DIP out of a DSpace Item.

```
/**
 * Plugin to export items as DIPs
 */
public interface DisseminationPackage extends SelfNamedPlugin
\{
    /**
     * Export the object (Item, Collection, or Community) to a
     * package file on the indicated OutputStream.
     * Gets an exception of the object cannot be packaged or there is
     * a failure creating the package.
     *
     * @param context - DSpace context.
     * @param object - DSpace object (item, collection, etc)
     * @param params Properties-style list of options specific to this packager
     * @param out - output stream on which to write package
     * @throws PackageException if package cannot be created or there is
     *  a fatal error in creating it.
     */
    public void disseminate(Context ctx, DSpaceObject obj,
            PackageParameters params, OutputStream out)
        throws PackageException;
```

```
    /**
     * Identifies the MIME-type of this package, e.g. "application/zip"
     * equired when sending the package via e.g. WebDAV GET, to
     * provide the HTTP  Content-Type header.
     *
     * @return the MIME type (content-type header) of the package to be returned
     */
    String getMIMEType();
\}
```

Packager Plugin Implementations

Typical Ingest Operation

A typical package importer follows an algorithm something like this:

1. Validate and check the integrity of the package. Is the manifest well-formed and complete? Is every one of the content files referenced in the manifest, and are all of the files mentioned in the manifest available in the content?
2. Extract the item metadata from the manifest and convert it to the Dublin Core item metadata set, perhaps using one of the `CrosswalkPlugins`. Validate the metadata, ensure it is correct and complete.
3. Create a `WorkflowItem` and set the metadata on its `Item` to the crosswalked metadata.
4. For an archive-type package, extract bitstreams from the package, check their integrity if possible (e.g. by comparing checksums) and install them as itstream objects. Set each bitstream source to its relative path in the package, the name to either the same path or a name dictated by the manifest, and set the content-type (format) to whatever value is guessed or extracted from the manifest.
5. For a package listing its content "by reference", follow UI pointers to fetch the content bitstreams and check them in the same way as for files in an archive.
6. Add the license to the item, either the string specified explicitly or the default license for the collection.
7. Possibly put the manifest into a bitstream, in the `METADATA` bundle, if it should be preserved.
   The ingest returns a `WorkspaceItem` to the caller. That lets the caller choose whether to submit to the normal workflow or bypass that and install it in the archive.

Typical Export Operation

The exporter is given a DSpace `Item` object to disseminate. The export process is typically:

1. Create (or re-use from original SIP) the manifest and item metadata.
   - If the DIP being generated is the same package type as the SIP that was originally imported, the exporter might take a shortcut and grab the original manifest and/or metadata.
   - The Packager must be able to generate a manifest and metadata from the the Item itself. Use metadata tools from `CrosswalkPlugins` to translate the item's metadata to the format required by the package.
2. Write the package to the output stream in the appropriate format.

Command-Line (Batch) Tool

The `Packager` application can be run from the command line to serve as a package-oriented batch import and export tool.

```
% dsrun org.dspace.app.packager.Packager -h
usage: Packager  options  package-file
-c,--collection    destination collection(s) Handle (repeatable)
-d,--disseminate   Disseminate package (output); default is to submit.
-e,--eperson       email address of eperson doing importing
-h,--help          help
-i,--item          Handle of item to disseminate.
-o,--option        Packager option to pass to plugin, "name=value"
(repeatable)
-t,--type          package type or MIMEtype
-w,--install       disable workflow; install immediately without going
through collection's workflow
```

This command installs a SIP as a new item in
the collection with handle 123.4/13

```
% dsrun org.dspace.app.packager.Packager -e lcs@mit.edu -c 123.4/13  -t pdf thesis.pdf
```

This command gets a DIP of Item 123.4/34 in the METS format:

```
% dsrun org.dspace.app.packager.Packager -d -e lcs@mit.edu -i 123.4/34 -i -t METS
```

Issues

Should an importer save the exact original package file in a bitstream in the item it creates? A symmetric exporter *should be able to to recreate an exactly equivalent package. *ANSWER: Resolve this case-by-case, but generally do NOT save the package because it might be huge.*

- The manifest file that comes with a package is recorded in a Bitstream like the content files. Since it is actually metadata, there ought to be a way to indicate that in the item. For DSpace 1.x we can put it in a "METADATA" bundle, but this breaks down in DSpace 2 if bundles go away. *ANSWER: Put it in the METADATA bundle -- the bundle name is still significant as a tag on the bitstream. Also set `BitstreamFormat` to identify the particular manifest type.*
- Is it worth implementing export of Collections and Communities, or should that be taken out of the interface? The implementation of a packager may choose not to support anything but Items.
- RobertTansley wants to see more separation between importer and exporter, since we don't always need both. A `Packager` doesn't have to implement both `import` and `export` methods. Is it worth adding explicit `PackageImporter` and `PackageExporter` interfaces as subclasses of `Packager` so each packager could implement whichever it wants (or both). That would let the code detect which ones are importers before getting an exception trying to import, so it could post a list of package importers (and the same for exporters, of course). --lcs *DONE. Separate interfaces.*