LightweightNetworkInterface

Removed in DSpace 6.0 (and above)

As of DSpace 6.0, the Lightweight Network Interface (LNI) is no longer provided with DSpace out-of-the-box. However, the codebase is still available (in an unmaintained state) at https://github.com/DSpace/dspace-lni

If LNI is of importance to you or your institution, please get it touch, and we can work to provide you the "keys" to this old codebase.
NOTE: This page was copied from the old MoinMoin wiki and contains many markup artifacts and incorrect translations. Consult the the original page preserved here if you need precise answers about URI syntax, identifiers, etc.

Contents

A Lightweight Network Interface (for DSpace)

Revised 8 May 06, Larry Stone

This page describes a simple but comprehensive network interface that was developed as part of the CWSpace project. It was designed to be complete enough for most programmatic uses of the DSpace API, particularly submitting and extracting archived materials as packages or byte streams.

It is documented here, and the source code made available on Downloads and Clients in the hope that it will be generally useful to the DSpace community.

See Also

Downloads and Clients for a sample implementation (unstable) and a pre-built test client.

This page is more of a design specification than a tutorial manual. For more documentation, see the WebDAV page (WebDAV protocol and client libraries in Java and C that can be used with the LNI).

Requirements

Since it has been difficult to elicit a coherent set of requirements for this project, one of our primary goals is to make the interface easy to extend, in the hope of meeting many as-yet-unknown requirements.

Also, please help by responding to this page and adding your requirements and anticipated usage here.

Specific Requirements

- Platform - and language-neutral implementation, to serve as a bridge to non-Java applications.
- Provide a simple and straightforward interface that is easy for busy developers to learn and adopt.
- Follow mature standards wherever possible, to leverage existing tools and platform support.
- Expose a complete and comprehensive view of the DSpace public object API (the "business logic" layer), so a client of the network interface can *e.g.* implement its own "application" like a user interface.
- It must be extensible without any changes to the client's API, and with backward-compatibility for older clients.
- Response time to simple requests (such as dissemination) should be fast enough to implement a usable interactive user interface.
- Support reliable transfer of very large data streams as SIPs and DIPs over imperfect networks.

Non-goals

- No transformation or repackaging of data beyond what is necessary for an efficient network interface – let the underlying layers of the DSpace system handle e.g. SIP and DIP packaging. This ensures any useful transformation and packaging code is available to other DSpace applications too.
- Do not "stretch" this network interface to cover functions which are already available in other, existing interfaces, especially if they are implementations of standards:
    - Searching: Available now through SRW; add other specialized search protocols as needed (e.g. Z39.50).
    - Bulk metadata export and harvesting: Implemented now by OAI-PMH.

Network Interface Design

This network interface borrows the data model of the WebDAV protocol – the simplest case without locking or versioning. WebDAV is a proven and robust protocol that was designed for a similar purpose: accessing and modifying resources and their metadata over a network.

It is actually a full, though minimal, WebDAV server implementation, although it also supports SOAP RPC alternative calls for all the DAV methods which are not part of standard HTTP.

The SOAP alternatives are required by the CWSpace project, for which this interface was created. Since SOAP is not capable of the bulk data transfers required to ingest and disseminate content packages, some operations are still implemented on top of HTTP. The implementation is structured so it is not difficult to maintain both the hybrid SOAP/HTTP and pure HTTP-WebDAV interfaces.

Rationale

Adopting the WebDAV data model lets us borrow its XML schemas and leverage its proven architecture. The WebDAV interface lets many existing WebDAV clients applications interact with DSpace to some degree. There are many WebDAV clients available as software libraries on various platforms, which makes it easy to integrate other applications with DSpace through WebDAV.

The hybrid of SOAP RPC and HTTP satisfies a CWSpace requirement to use SOAP wherever it is practical. We employ HTTP GET and PUT for bulk data transfers to avoid the inefficiency and implementation limits of processing large bitstreams and content packages as base64-encoded components in SOAP message.

While the *SOAP attachment protocol* would appear be a more obvious choice for bulk data transfers than HTTP, it lacks maturity and widespread implementations. SOAP attachments also require cumbersome, inefficient and poorly-standardized MIME encoding.

DSpace Network Interface – Specification

Resource URIs

The LNI lets you refer to DSpace objects (such as Items, Collections, etc) as *resources*, in the HTTP sense. There is a simple and deterministic, though intentionally opaque, mapping of DSpace objects to resource URIs.

Do not conflate the WebDAV resource URIs with "URLs" to be used in an interactive Web browser. The LNI's resource URIs are not to be used interactively, and should *never* be archived or saved for later reference. Most of the URIs (with the notable exception of `lookup`) used with the LNI are obtained from the LNI itself, and are not guaranteed to be stable over the long term.

About the "prefix"

Prefix is the initial URL path at which the LNI service is "mounted" on its Web server. Since the LNI is intended to be run out of a Java Servlet container such as Tomcat, possibly the same one running other DSpace services (e.g. the Web UI), it will probably be mounted under a separate initial path to distinguish its requests from those bound for other servlets. For example, on a server host "`library.uni.edu`", if the LNI is mounted at `/dspace/dav`, the complete prefix is `http://library.uni.edu/dspace/dav`.

The absolute path of the URI for each type of content object is composed as follows:

DSpace Site	/[prefix]/
Lookup handle	/[prefix]/lookup/handle/[handle] , /[prefix]/lookup/bitstream-handle/[sequence]/[handle]_
Community, Collection, Item (DSpace Objects)	/[prefix]/dso_[persistent-identifier]
Bitstream	/[prefix]/dso_[persistent-identifier]/bitstream_[pid]
Workspace Items list	/[prefix]/workspace
Workspace Item	/[prefix]/workspace/wsi_db_[id]
Workflow Items list	/[prefix]/workflow_own/ , /[prefix]/workflow_pool/
Workflow Item	/[prefix]/workflow_own/wfi_db_[id]
EPerson list	/[prefix]/eperson/
EPerson	/[prefix]/eperson/ep_db_[id]

Key:

- [prefix] was covered above.
- [persistent-identifier] is a specially-encoded version of the DSpace object's persistent identifier (such as a Handle). (See discussion of `/lookup` below for details).
- [handle] is the Handle or other persistent identifier. It may only be used with the `lookup` service.
- [pid] in the bitstream URI is a *persistent bitstream identifier*.
- [id] is the numeric identifier of row in the database, used to identify e.g. an in-progress Item, such as a workflow or workspace item.

About Identifiers

DSpace assigns "persistent identifiers", such as Handles, to all of its first-class content objects. The LNI's resource URIs do not contain literal DSpace identifiers, however, because some identifier formats contain characters (e.g. '/') which cause problems with existing WebDAV software, *even when properly "escaped".* To work around this problem and to prevent similar trouble with any new persistent identifier schemes adopted in the future, the LNI transforms an identifier by its own proprietary process before putting it into a resource URI. The exact transformation is not published so it can be changed without affecting clients.

This is actually no limitation for the typical DSpace WebDAV client, since the client never has to compose its own resource URI for a DSpace object. Typically, the client obtains a resource URI in one of these ways:

1. Starting at a known URI, e.g. the DSpace Site, and navigating from there.
2. The URI is given to the client in the response from another WebDAV operation, such as `PROPFIND`.
3. The client starts with a DSpace persistent identifier (e.g. returned by a search service) , and makes a `lookup` URI out of it and does an HTTP GET on that to acquire the actual LNI resource URI of the object.

Note that the transformed identifier in a DAV URI is **NOT** necessarily a predictable encoding of the object's identifier; the client should refrain from trying to create this URI itself based on an identifier – always use the `lookup` resource for that. Treat these resource URIs as opaque objects.

Authorizations and Access to Resources

The LNI applies the same authorization model as the Web user interface, since it is mostly embedded in the DSpace "business logic" layer (or it should be). The properties of Communities and Collections are visible to everyone, since they are, effectively, metadata. The contents of a Community or Collection are only visible to principals with `READ` access, however. Item properties and contents are likewise only visible to principals with `READ` access.

The DSpace authorization model controls all modifications to resources representing DSpace objects. For example, you need `WRITE` access on an object to change or delete WebDAV properties on its corresponding resource.

The Workspace and Workflow (in-progress) objects mirror the access controls of their corresponding DSpace objects.

The EPerson resources are an exception, since EPerson objects are not exposed so explicitly in the Web UI. The logged-in EPerson can modify some properties on his/her own EPerson object, and can only read most properties on the others. Administrators can modify all EPersons.

Distinguishing DSpace Object Resources

Note that Items, Collections, and Communities are all "DSpace Objects" referenced by persistent identifiers (such as Handles), so they share the same URI format. The *type* of DSpace Object which a resource represents is available as a WebDAV property: `"dspace:type"`. Its value is an XML element in the DSpace namespace, one of:

- `<site/>` – the root of the site.
- `<collection/>` – Collection object.
- `<community/>` – Community object.
- `<item/>` – Item object.
- `<bitstream/>` – Bitstream object.
- `<eperson-collection/>` – Set of all EPersons in the site.
- `<eperson/>` – An EPerson.
- `<workspace/>` – Set of all your workspace items.
- `<workspace-item/>` – Workspace item.
- `<workflow/>` – Set of workflow items.
- `<workflow-item/>` – Workflow item.

The "`dso_`" path elements can be cascaded as you descend a hierarchy. The server only pays attention to the *last* element in a path of DSO's, e.g.

```
http://uni.edu/dspace/dav/dso_123456789$4/dso_123456789$13
```

is the same as

```
http://uni.edu/dspace/dav/dso_123456789$13
```

Since a Bitstream must be referenced in the context of an Item, it is identified by an extra pathname element following the URI of an Item.

The root of the URI hierarchy is a special "Site" collection object which contains top-level properties, and all top-level communities.

Some example URIs:

```
    Site          http://uni.edu/dspace/dav

    Lookup        http://uni.edu/dspace/dav/lookup/handle/1721.1/46
                  http://uni.edu/dspace/dav/lookup/handle/1721.1%2F46
                  http://uni.edu/dspace/dav/lookup/bitstream-handle/13/1721.1%2F46

    Community     http://uni.edu/dspace/dav/dso_1721.1$46

    Collection    http://uni.edu/dspace/dav/dso_1721.1$3549

    Item          http://uni.edu/dspace/dav/dso_1721.1$5543
                  http://uni.edu/dspace/dav/dso_1721.1$3549/dso_1721.1$5543

    Bitstream     http://uni.edu/dspace/dav/dso_1721.1$5543/bitstream_13
    Bitstream     http://uni.edu/dspace/dav/dso_1721.1$5543/bitstream_13.pdf
```

HTTP and WebDAV Methods (API)

The LNI repsonds to the following HTTP requests using WebDAV methods. It also supports a SOAP RPC API, which has the same operations and semantics as the WebDAV methods.

PROPFIND

`PROPFIND` is a core WebDAV method, an extension to HTTP. This description also applies to the `propfind` call in the SOAP API. `PROPFIND` gets the WebDAV *properties* bound to a resource, and optionally for the resource's descendents as well. It has two man uses:

1. Read the properties (administrative metadata) on a resource.
2. Discover the *hierarchy* of resources under a given URI, by asking for some set properties at a depth of 1 or `INFINITY`.

In the `PROPFIND` HTTP request, the URI identifies the resource to be queried.

The body of the request is an XML `propfind` element constraining the query. It may contain either a single `propname` element (meaning just return the names of all available properties), a `prop` element containing a sequence of `propname}}s for which to return values, or an {{allprop` element meaning "return all available properties and values."

Finally, the `Depth` header determines how deeply the query descends the hierarchy of resources. A depth of `0` means just return properties on the resource itself; `1` includes only the direct descendents (i.e. Items in a Collection or Bitstreams in an Item). The special constant `INFINITY` `(-1 in the SOAP call)` means return the chosen properties on *all* descendents.

Limiting Types of Objects Returned

There is one other twist in the LNI's implementation of `PROPFIND`: you can select which types of DSpace Objects to visit, in order to limit a traversal to the "upper layers" (i.e. Communities and Collections) of the object hierarchy.

Add the query argument `type` to the resource URI to invoke this feature. The value must be the name the type of DSpace Object to which you want the query restricted, one of: `COMMUNITY, COLLECTION, ITEM, BITSTREAM.` The name is case-insensitive. To choose more than one type, add another `type` query arg to the URL.

The request's URI is *always* visited, regardless of any `type` limitation.

For example, the first URI visits all of the collections and communities in the whole Site, while the second only shows Collections under a given Community (avoiding sub-Communities):

```
http://uni.edu/dspace/dav/?type=COMMUNITY&type=COLLECTION

http://uni.edu/dspace/dav/dso_1721.1$3549?type=collection
```

More on PROPFIND

See the propfind_xml description of the `propfind` XML element and the WebDAV protocol specification for complete details on the use of `propfind` and `multistatus`.

You can map the hierarchy of resources under a specific URI by calling `propfind` with an empty list of properties, so it just returns the structure of resources without any property data.

The properties supported by the DSpace LNI are described below. Each kind of DSpace Object (e.g. Collection, Item, etc) has its own set of supported properties, and there are a few common ones.

`PROPFIND` returns the results in a `multistatus` element that contains a `propstat` element for each property found. See the WebDAV protocol specification for more details.

PROPPATCH

The `PROPPATCH` method modifies and/or deletes properties on a single resource. In the HTTP request, the URI identifies the resource to be modified.

The body of the request is an XML `propertyupdate` element describing the properties to be changed or removed. It contains a series of `set` and `remove` elements, each has a `prop` element child describing the property to be set or deleted.

`PROPPATCH` reurns a `multistatus` element similar to the one returnd by `propfind`, with a separate status for each property that was changed.

The properties supported by the DSpace LNI are described below. Each kind of DSpace Object (e.g. Collection, Item, etc) has its own set of supported properties, and there are a few common ones. *Not all properties may be modified.* Some are inherently read-only, some are derived from e.g. the descriptive metadata which administrators may not want to be modified through the LNI.

See the propfind_xml description of the `proppatch` XML element and the WebDAV protocol specification for complete details on the use of `propfind` and `multistatus`.

COPY

The `COPY` method creates what appears to be a copy of a *resource* at a new location. It actually just establishes a *link*, or reference, to the resource at the destination location, but we use the `COPY` method because of all the core WebDAV methods, it comes closest to having the right semantics.

Although it would arguably be more correct to implement the WebDAV BIND protocol instead, that protocol is not yet mature and established enough to become part of the LNI. Also, not much client support is available.

`COPY` is in the LNI mainly so an Item can be installed into multiple Collections. This is sometimes required when ingesting content packages. The LNI does not (yet?) support the operations to modify the resource namespace in a general way.

Copy Semantics

Both source and destination URIs must be within the same DSpace Site.

The type of the destination resource must meet the limits imposed by the source:

Source is:	Destination must be:
Bitstream	*not allowed*
Item	Collection
Workspace Item	Collection
Workflow Item	Collection

Collection	Community *(not implemented yet)*
Community	Community, Site *(not implemented yet)*

Upon success, the source URI will be a member of the destination collective object.

HTTP GET

The `GET` method returns the contents of a resource, which must be either an Item or a Bitstream.

- `GET` on an Item goes through the PackagerPlugins to return the contents of the item as a Dissemination Information Package (DIP). This can actually be anything that a packager wants to generate, *i.e.* a Zip file stream, just the manifest, or an archive of only some of the content.
- `GET` on a Bitstream returns the contents of the bitstream, just as an HTTP server would return the contents of a file resource.

The general format of a GET URI is:

GET URI for Item:: ***/* prefix */* resource_path *?package=* package-format *&other-packager-parameters***
GET URI for Bitstream:: **/* prefix */* resource_path [.optional-extension]*

Where *prefix* and the URI path are as described in the section above on **Resource URIs**.

The `package` query argument and *Package-format* is required for Item resources; *package-format* selects the PackagerPlugins module that creates the package.

Any other query arguments are forwarded to the PackagerPlugins plugin, so the client has a direct channel of communicaton with the packager. These parameters typically control the kind of package created and what content is included.

When the resource is a bitstream, the package query arg is omitted, since the bitstream is always sent unpackaged. You can always add an extension to the bitstream URI, since this helps some HTTP clients handle the content correctly.

`GET` is not implemented for Communities and Collections. `PROPFIND` can fetch all of the significant information about those types of objects, namely their administrative metadata and their member objects.

Some examples of GET requests:

```
GET /dspace/dav/dso_1721.1$5543?package=METS&dmd=MODS

GET /dspace/dav/dso_1721.1$5543/bitstream_13

GET /dspace/dav/dso_1721.1$5543/bitstream_13.pdf
```

GET Headers

The GET request also responds to HTTP request headers to modify the request:

- Range:: Only send specified byte range of the resource. Syntax of the header value is described in RFC 2068.

If the client does not want to transfer a large object (e.g. multi-gigabyte video clip) in one transaction, it can use the `Range` header to break up the transfer into multiple requests. If one of those fails, it is faster (and perhaps safer) to retry it than to redo a large transfer.

(NOTE: Range is not implemented in the initial release.)

HTTP PUT

In WebDAV, the `PUT` method creates a new resource at the given URI (if there wasn't already anything there) or replaces the contents of an existing resource. In the DSpace LNI, `PUT` can only create a new resource *under* an existing "collection" (in the WebDAV sense of "collection") resource. It cannot create a new resource at a specified URI because DSpace must determine the persistent identifier, and thus the URI, *after* the object is created. The DSpace LNI `PUT` can also replace the contents of an existing resource, if this is allowed and implemented.

The LNI does not provide any means to install more than one Item in a single request, or to install an entire populated DSpace Collection.

PUT Semantics: Creating Items

The most obvious and common application of WebDAV is to create a new resource by supplying the exact URI at which it will reside.
This conflicts with the way DSpace names new objects, since a DSpace URI is based on a Handle (or other persistent identifier), and *that* is only available *after* the resource is created, – *not* at the time the client is sending its `PUT` request.

To work around this, we take advantage of the fact that a DSpace Collection is also a "collection" in the WebDAV sense of the word, that is, a grouping of resources. To create a *new* Item, just do `PUT` on a DSpace Collection resource, which creates a new Item within that Collection.

The HTTP response from such a `PUT` includes a `Location` header with the URI of the new Item, so the client can find out the URI (and thus the persistent identifier, if available) of the Item it created.

N.B. The URI returned by `Location` is a LNI resource URI. It will be one of these types of object:

1. Workflow Item, because the new item has entered workflow. It has no handle yet.
2. Item, because the collection has no workflow; the item will have a handle.

If you need a persistent identifier, use `PROPFIND` on the returned resource URI to retrieve its `dspace:type` and `dspace:handle` properties. (If a handle is not available, the type will help explain why.)

For example, to add a new item to the Collection at handle `1721.1/3549`, we observe the following request and response:

```
Request:    PUT /dspace/dav/dso_1721.1$3549?package=OCW-IMSCP
             ....package contents in body...

Response:   HTTP/1.1 201 OK
            Location: http://uni.edu/dspace/dav/dso_1721.1$F5549
             ....other headers....
```

PUT Semantics: Adding an Item to Multiple Collections

To add an item to more than one Collection, first create it under its parent Collection and then use the `COPY` method to map it into the other Collections.

PUT Semantics: Updating/Replacing an Item

Applying the `PUT` method to a DSpace Item resource replaces or updates that Item with the contents of the package supplied in the body of the `PUT`. The exact semantics depend on the individual packager plugin; some of them may not implement this operation so the result would be an exception.

`PUT` is not implemented for Bitstream resources (yet?).

PUT Query Arguments

In a `PUT`, the resource URI naming a Collection or Item *must* include a `package` query argument, which names the packager plugin to be called to import the data sent with the `PUT`.

Other query arguments are passed along to the packager plugin as parameters, allowing the client a direct channel of communication with the packager.

PUT Headers

The `PUT` method also responds to the following `HTTP` request headers:

- Content-Length:: The number of bytes in the content block following the request and entity headers.
- Content-MD5:: An MD5 checksum of the content. This is optional but encouraged. The checksum applies only to the bytes sent, in the case of a subset defined by a `Content-Range` header.
- Content-Range:: This request only includes part of the entity. In this network interface, ranges MUST be consecutive and MUST be sent in their natural order.

NOTE: The range header allows large objects to be transferred in segments, with checksum and/or length validation on each segment. This minimizes the risk of failure when sending extremely large packages over unreliable networks.

PROPFIND and PROPPATCH XML elements

The LNI makes most administrative and technical metadata of DSpace objects available through the WebDAV concept of "resource properties". WebDAV properties are an extensible metadata framework which works as well as anything we could custom-design for this purpose. Please see the WebDAV protocol specification for more details about the property mechanism.

The client makes inquiries about properties with the `PROPFIND` method, and the query itself is defined by the `propfind` XML element supplied in the request body. For example, this query gets the `displayname` WebDAV property, `dspace:type` (DSpace object type), and `getlastmodified` (the WebDAV last-modified time):

```
<propfind xmlns="DAV:">
  <prop xmlns:dspace="http://www.dspace.org/xmlns/dspace">
    <displayname/>
    <getlastmodified/>
    <dspace:type/>
  </prop>
</propfind>
```

The result of a `PROPFIND` is a `multistatus}}element, which contains the status (success/failure) of the query of each property on each resource, and the property values in `propstat` elements. This example returns successfully for the {{"displayname"` and `dspace:type` properties, and fails to access the `"getlastmodified"` property:

```
<multistatus xmlns="DAV:">
  <response>
    <href>/dspace/dav/dso_1721.1$5543</href>
    <propstat xmlns:dspace="http://www.dspace.org/xmlns/dspace">
      <prop>
        <displayname>Zen and the Art of Java Maintenance</displayname>
        <dspace:type>
          <dspace:item/>
        </dspace:type>
      </prop>
      <status>HTTP/1.1 200 OK</status>
    </propstat>
    <propstat>
      <prop>
        <getlastmodified/>
      </prop>
      <status>HTTP/1.1 403 Forbidden<status>
      <responsedescription>
      You do not have the right to read getlastmodified.
      </responsedescription>
    </propstat>
  </response>
</multistatus>
```

The client modifies and deletes properties with the `PROPPATCH` method, and the instructions on what to do are encoded in a `propertyupdate` element in the request body.

The `PROPPATCH` request returns a response with a `multistatus` element much like the response for `PROPFIND`.

Here is an example that changes (or adds) the `displayname` property and removes `dspace:license`:

```
<propertyupdate  xmlns="DAV:" xmlns:dspace="http://www.dspace.org/xmlns/dspace">
  <set>
    <prop>
      <displayname>Zen and the Art of Java Maintenance</displayname>
    </prop>
  </set>
  <remove>
    <prop>
      <dspace:license>
    </prop>
  </remove>
</propertyupdate>
```

DSpace-specific Properties

These WebDAV *property elements* are defined in the DSpace XML namespace (see XmlNamespaces). The namespace URI is `"http://www.dspace.org /xmlns/dspace"`, shown here as the prefix `dspace:`, but of course it could be anything.

DSpace resources also support the required WebDAV properties, and some optional ones. They are prefixed with `DAV:` to represent the namespace URI, which is also `DAV:`.

ALL DSpace Resources::

- `dspace:type` – DSpace Object type, the value is an element whose name is the type in lowercase, e.g. one of `<dspace:bitstream>`, `<dspace:item>`, `<dspace:collection>`, `<dspace:community>`, `<dspace:site>`, etc.
- `DAV:resourcetype` – WebDAV resource type, value is either `<DAV:collection/>` for a collection or empty for an "atom".
- `DAV:displayname` – Label for a human-readable display of this resource, typically title or filename.
- `DAV:current-user-privilege-set` – The permissions the current user has to access this resource, in the format specified by the WebDAV Access Control Protocol RFC3744. Note that we do not (yet) implement any more of RFC3744.

Site (root of URI hierarchy)::

- `dspace:news_top` – News (string) to be displayed in the top area of the DSpace home page.
- `dspace:news_side` – News (string) to be displayed in the sidebar of the DSpace home page.
- `dspace:default_license` – Site-wide default license text applying to new items. Read-only.

Community objects::

- `dspace:logo` – actual image content of the logo image, in a `<bitstream>` tag bitstream_xml (see below).
- `dspace:short_description` – short description.
- `dspace:introductory_text` – short introductory text.
- `dspace:side_bar_text` – sidebar text (news) to show on community page.
- `dspace:copyright_text` – copyright statement to show on community page.
- `dspace:handle` – the persistent identifier of this resource in URN format, e.g. `hdl:123.45/6789`.

Collection objects::

- `dspace:logo` – actual image content of the logo image, in a `<bitstream>` tag bitstream_xml (see below).
- `dspace:short_description` – short description.
- `dspace:introductory_text` – short introductory text.
- `dspace:side_bar_text` – sidebar text (news) to show on collection page.
- `dspace:copyright_text` – copyright statement to show on collection page.
- `dspace:default_license` – default license text applying to new items.
- `dspace:provenance_description` – the `"provenance_description"` metadata for collection, whatever that is.
- `dspace:handle` – the persistent identifier of this resource in URN format, e.g. `hdl:123.45/6789`.

Item objects::

- `dspace:submitter` – ePerson resource of original submitter of this item.
- `dspace:owning_collection` – persistent identifier (handle) of Collection that owns this item.
- `dspace:license` – contents of license bitstream for this item, in a `<bitstream>` tag bitstream_xml (see below).
- `dspace:cc_license_text` – text of Creative Commons license bitstream for this item, in a `<bitstream>` tag bitstream_xml (see below).
- `dspace:cc_license_rdf` – RDF of Creative Commons license bitstream for this item, in a `<bitstream>` tag bitstream_xml (see below).
- `dspace:cc_license_url` – URL of Creative Commons license for this item.
- `DAV:getlastmodified` – This DAV property is assigned the DSpace last-modified date.
- `dspace:handle` – the persistent identifier of this resource in URN format, e.g. `hdl:123.45/6789`.
- `dspace:withdrawn` – `true` or `false`, indicating whether the Item has been withdrawn. This is the only property most users will have permission to read on a withdrawn Item.

Bitstream objects::

- `DAV:getcontentlength` – value returned by bitstream's `getSize()`.
- `DAV:getcontenttype` – mimetype from bistream's format.
- *NOTE: the DSpace object model has no notion of `getcontentlength` for Bitstreams, so we cannot offer it.*
- `dspace:source` – bitstream's `source` attribute.
- `dspace:description` – bitstream's `description` attribute.
- `dspace:format` – bitstream's `getFormat().getID()`.
- `dspace:format_description` – bitstream's `getUserFormatDescription()`.
- `dspace:checksum` – bitstream's `getChecksum()`.
- `dspace:checksum_algorithm` – bitstream's `getChecksumAlgorithm()`.
- `dspace:sequence_id` – returned by bitstream's `getSequenceID()`.
- `dspace:bundle` – name of bundle in which bitstream was created.
- `dspace:handle` – the persistent identifier of this resource in URN format, e.g. `hdl:123.45/6789` (but usually not available for bitstreams (yet?))

In-progress Item::

- `dspace:submitter` – EPerson resource of original submitter of this item.
- `dspace:collection` – persistent identifier (handle) of Collection to which this item is being submitted.
- `dspace:has_multiple_files` – value of `hasMultipleFiles()`
- `dspace:has_multiple_titles` – value of `hasMultipleTitles()`
- `dspace:is_published_before` – value returned by `isPublishedBefore()`
- *Every in-progress item also has one child resource which is a DSpace Item.*

Workspace Item::

- *all properties of the* In-progress Item, *and:*
- `dspace:stage_reached` – Symbolic name of the last stage reached in the submission process.
- `dspace:state` – **Write-Only:** Change this to the keyword `start` or `start_without_notify` to enter a workspace item into its collection's workflow.

Workflow Item::

- *all properties of the* In-progress Item, *and:*
- `dspace:owner` – EPerson resource of owner of this workflow item.
- `dspace:state` – When read, returns the symbolic name of the current workflow step, e.g. `STEP1_POOL`. You can also **set** this property to one of the following keywords to manipulate its place in the workflow: (See API doc for `org.dspace.workflow.WorkflowManager` for details)
- `abort` – Aborts workflow, returning item to its submitter's workspace.
- `reject` – Reject an item at `STEP1`, send mail to submitter, otherwise the same as `abort`.
- `advance` – Move an item to the pool for the next step, or archive it if it was on the last step. You must have previously claimed the workflow item.
- `claim` – Take responsibility for a workflow item that was in the pool of unclaimed items.
- `unclaim` – Return an item you have claimed to the pool.

EPerson Collection::

- (None other than the default)

EPerson::

- `dspace:email` – email address.
- `dspace:first_name` – first, or given name.
- `dspace:last_name` – last, or family name.
- `dspace:require_certificate` – if true, this eperson can only login with an X.509 client certificate.
- `dspace:self_registered` – eperson record created by selfsame user.
- `dspace:can_login` – if true, user is allowed to login.
- `dspace:handle` – the persistent identifier of this resource in URN format, e.g. `hdl:123.45/6789` (but not available for epersons yet?))

Bitstream in XML Element

The `logo` attribute's value is a bitstream of an image. It is represented by the `dspace:bitstream` XML element, which gives the choice of encoding the bitstream as either a link (to an external resource) or inline base64-encoded data. The inline encoding should only be used for small (a few Kb) images. This example shows both alternatives:

```
<!-- link to external resource -->
<dspace:bitstream>
  <dspace:link href="http://dspace.myuniv.edu/dspace/dav/retrieve_54321" />
</dspace:bitstream>

<!-- inline encoding -->
<dspace:bitstream>
  <dspace:content contenttype="image/gif" contentlength="299" contentencoding="base64">
    ...text of base64-encoded data...
  </dspace:content>
</dspace:bitstream>
```

SOAP API Calls

This Java class outline shows the LNI's SOAP RPC interface, which is accessed through the `org.dspace.app.dav.client.LNISoapServlet` class. The utility class `org.dspace.app.dav.client.LNIClientUtils` also has some methods of use to client writers.

SOAP clients must still use HTTP `GET` and `PUT` to disseminate and submit packages and bitstream data, however, since standard SOAP is not effective for bulk data transfers.

NOTE: The SOAP servlet accepts WebDAV `GET` and `PUT` requests as well as SOAP messages, which simplifies the switching of modes in your client code. Typically, you will start with a SOAP endpoint URL that looks something like `http://uni.edu/dspace/lni/DSpaceLNI`. The final pathname element, `DSpaceLNI`, names a SOAP application within the servlet, so the servlet path is actually everything up to `/dspace/lni/`. To construct a WebDAV URL, simply use that servlet path as your WebDAV *prefix*, so e.g. a resource URL would become: `http://uni.edu/dspace/lni/dso_1721.1$5543`. There is a convenience method in the Java class `LNIClientUtils` to help manage this, but you will
have to implement it yourself for SOAP clients in other languages.

Every feature of the LNI is available through WebDAV (HTTP); this API is provided for callers who prefer to use SOAP RPCs where possible.

```
public class org.dspace.app.dav.client.LNISoapServlet {

  /**
  *** Returns Resource URI for the DSpace Object whose persistent
  *** identifier (i.e. Handle) is  "handle".  Optionally add Persistent ID
  *** (sequence ID) of a bitstream under the Item, if a URI to a bitstream
  *** is desired, otherwise bitstreamPID should be null.
  *** This does the same thing as the /lookup URI.
  ***/
  public String lookup(String handle, String bitstreamPID)
    throws java.rmi.RemoteException, org.dspace.app.dav.client.LNIRemoteException

  /**
  *** Same as PROPFIND WebDAV method.  "uri" may be relative to DSpace LNI
  *** prefix, or absolute; "propfind" is the propfind element, and depth is
  *** the content of the "Depth:" header.  Depth should be 0, 1, or the
  *** constant org.dspace.app.dav.client.LNIClientUtils.INFINITY (-1).
  *** Types is a comma-separated list of DSpace item types to which to
  *** restrict the query (see "type" option of PROPFIND method). May be null.
  *** Returns the multistatus document from the method's response.
  ***/
  public String propfind(String uri, Document propfind, int depth, String types)
    throws java.rmi.RemoteException, org.dspace.app.dav.client.LNIRemoteException

  /**
  *** Same as PROPPATCH WebDAV method.  "uri" may be relative to DSpace
```

```
*** LNI prefix, or absolute; "propertyupdate" is the propertyupdate
*** element.  Returns the multistatus document from the method's response.
***/
public String proppatch(String uri, String propertyupdate)
  throws java.rmi.RemoteException, org.dspace.app.dav.client.LNIRemoteException

/**
*** Executes COPY method; "uri" and "destination_uri" may be relative
*** to DSpace LNI prefix, or absolute.
*** The depth and 'keepProperties' parameters correspond to
*** parameters on the actual COPY WebDAV method, but DSpace ignores them
*** at this time.
*** The overwrite option will allow the copy to overwrite an existing
*** resource if necessary.
***
*** Returns the HTTP status code.
***/
public int copy(String uri, String destination_uri, int depth,
                boolean overwrite, boolean keepProperties)
  throws java.rmi.RemoteException, org.dspace.app.dav.client.LNIRemoteException
}

public class org.dspace.app.dav.client.LNIClientUtils {

/** Depth of infinity in SOAP propfind() */
public final static int INFINITY = -1;

/**
*** Make up a URL to access a WebDAV resource, given the SOAP "endpoint" URL
*** and a relative URI such as is returned by lookup().  Clients should
*** use this to obtain URLs to make HTTP GET and PUT requests.
*** Packager may be null for a resource such as a Bitstream that does
*** not require a packager.
***/
public static URL makeDAVURL(String endpoint, String davURI, String packager);
  throws MalformedURLException

/** alternate version that does not require packager. */
public static URL makeDAVURL(String endpoint, String davURI);
  throws MalformedURLException

/**
*** Translates a WebDAV URL, such as would be returned by the PUT
*** method, into a resource URI relative to the DAV root which can
*** be passed to the SOAP methods.  Inverse of makeDAVURL.
***/
public static String makeLNIURI(String endpoint, String davURL)
  throws MalformedURLException
}
```

Security Considerations

The LNI's approach to security has two layers: First, *authentication* and *authorization* , to let DSpace administrators control access to the resources in the archive. Second is *confidentiality* , which protects the contents of LNI transactions and the archive content itself from being copied as they are transmitted over the Internet.

Authentication and Authorization

The LNI relies on DSpace's "business logic" layer API for authentication and authorization. It calls on the StackableAuthenticationMethods to authenticate the user using the same authentication methods as are implemented for the Web UI. The only difference is that methods calling for a username and password get those data from the HTTP Basic Authentication protocol instead of an interactive Web page.

Both the SOAP and pure WebDAV modes of the LNI use the same authentication protocol.

By default, the LNI only allows access by authenticated users. There is no compelling reason *not* to allow anonymous access via the LNI, except that badly-behaved clients can put a misproportionate load on the server by issuing extensive recursive PROPFIND requests. It just seemed like a reasonably cautious default. To change it, see the #Configuration section; the property is `dav.access.anonymous`.

If you allow anonymous access, the anonymous users won't be allowed to do anything more than they would under the WebUI; i.e. mostly examine resources with PROPFIND and retrieve Items and Bitstreams with GET.

Authorization is implemented in the DSpace object model, so, as the LNI is at the application layer, it is subject to the same authorization constraints as other applications like the Web UI. All of the authorization policies imposed by DSpace administrators apply to LNI clients just as they do to other application clients.

Confidentiality

If you are letting users make authenticated LNI transactions on a server that is on the public Internet, I strongly recommend requiring all authenticated transactions to be encrypted (e.g. by SSL) to protect the user's credentials, as well as the material being transferred.

Both the SOAP and WebDAV modes of the LNI work perfectly well over HTTPS, which is HTTP over SSL. All you need to do is configure your web servlet container (e.g. Apache Jakarta Tomcat 5.0) for SSL connections.

You can also add a `security-constraint` tag to the `web.xml` configuration file in the DSpace web application, to require SSL connections on LNI transactions. There will be notes about this in the `dspace-web.xml` file in the source distribution.

Configuration

The LNI is configured by the following properties in the DSpace Configuration:

- dav.access.anonymous:: When true, allow client to make LNI requests without authenticating as a DSpace EPerson. Default is `false`.
- dav.propfind.limit:: Limit on the number of distinct resources returned in a response to a `PROPFIND` request. Default is 0, which means unlimited. Set this to prevent your server from being too heavily loaded by a malicious or pathological client that attempts to e.g. do a recursive `PROPFIND` listing every property of every object.

Scenarios

A. Simple SOAP client example

Here is an example of how a client would create a session and issue a `propfind` call with the LNI. It assumes the SOAP interface is built upon Apache's Axis SOAP implementation.

```
// DSpace credentials are either user/password in the URL, or X.509
// client cert supplied with https: connection.
String endpoint = "http://user:password@dspace.uni.edu/dspace/lni/DSpaceLNI";

// get Axis locator
LniSoapServletServiceLocator locator = new LniSoapServletServiceLocator();

// create client endpoint
LniSoapServle] lni = locator.getDSpaceLNI(new java.net.URL(endpoint));

// get resource URI for known handle:
String handle = "1234.56/789";
String resourceUri = lni.lookup(handle, null);

// get its properties..
String result = lni.propfind(resourceUri,
      "<propfind xmlns=\"DAV:\"><allprop /></propfind>", 1);

// ..now parse and display the XML in "result"..
```

B. Submit a new Item to multiple collections

Start with the item in an acceptable package format, for example, an IMSCP content package in a Zip file. The goal is to install it in two collections (handles `123456789/11` and `123456789/22`). Here are the HTTP WebDAV requests you would make:

1. Do a `GET` on `/dspace/dav/lookup/handle/123456789/11` to get the LNI resource URI for the first collection, which will be `/dspace/dav/dso_123456789$11`.
2. Submit the item with a `PUT` request to `/dspace/dav/dso_123456789$11?format=IMSCP`
3. The response to the `PUT` includes a `Location:` header giving the URI of the item it created, e.g. `Location: /dspace/dav/workflow/wfi_db_23`
4. To add it to the second collection, start with a `GET` on `/dspace/dav/lookup/handle/123456789/22` to get the LNI resource URI for the second collection, which will be `/dspace/dav/dso_123456789$22`.
5. Issue the request `COPY /dspace/dav/workflow/23` with the header, `Destination: /dspace/dav/dso_123456789$22` to add the workflow item to the second collection. When the Item gets through workflow and is archived, it will appear in both collections.

If there is no workflow and the item gets archived immediately, you would get back a regular Item URI in the `Location:` header of the response to the `PUT` and would copy that into the new collection.

C. Disseminating an Item

This example gets a dissemination of the item at handle `123456789/33` in the default format, which is (probably) a DSpace METS-based package very much like the internal AIP format, used e.g. for exchange between DSpace instances.

If the item is readable to the `Anonymous` group, we do not need to authenticate.

1. Do a `GET` on `/dspace/dav/lookup/handle/123456789/33` to get the LNI resource URI for that handle.
2. The client sends a `GET` request to the URI `/dspace/dav/dso_123456789$33?package=METS`
3. The response from the server is status 200 (success) followed by entity headers `Content-Type: application/zip`, any other relevant headers, and finally the Zip file data.

D. List All Collections I Can Submit Into

Get a list of all the collections to which the authenticated user can submit new Items. This would be a typical operation in a user interface for submitting Items.

Note that getting the list of *all* communities and collections on the DSpace Site may be slow if there are very many of them, so this should be narrowed down to a single Community (or repeated for several communities of interest) if it needs to be faster.

1. Call `propfind` on the URI `"/dspace/dav/?type=COMMUNITY&type=COLLECTION"`, depth=`INFINITY` and the `propfind` request document shown below, which lists the relevant authorization properties.
2. Examine the `multistatus` document returned by `propfind`, looking for resources whose `DAV:current_user_privilege_set` properties include the element `DAV:bind` (or `DAV:all`). These are the Collections into which you can submit.

Here is the document to send with the `propfind` request:

```
<propfind xmlns="DAV:">
  <prop xmlns:dspace="http://www.dspace.org/xmlns/dspace">
    <DAV:current_user_privilege_set />
  </prop>
</propfind>
```

Extending the Network Interface

This initial design is undoubtedly full of gaps and shortcomings. Since it was mainly driven by the needs of the CWSpace project and few other users were forthcoming about requirements, it was designed to be easily extensible.

There are several recommended ways to extend the LNI:

- Adding new types of resources to access more DSpace objects.
- Adding query argument options to resources to request different behavior (like the `type` option on URIs to `PROPFIND`).
- Implementing additional WebDAV methods (e.g. `MKCOL`, `DELETE`), and WebDAV extension protocols such as the fleshing out the ACL protocol.

Adding Resource Types

To add a new new kind of Resource URI, you have to modify the WebDAV server to respond to that URI. It must have a unique prefix so the dispatcher can tell it apart from, e.g. a request for an Item starting with `"dso_"`.

Next, you need to determine the mapping of URIs to DSpace objects. It's a good idea to reflect whatever hierarchical structure exists in the objects, but do not add any artificial hierarchy. As we will see in one example below, there is sometimes a reason to group resources into a collection so they can all be enumerated. If the target object is usually referenced by random access than by descending a hierarchy, perhaps it makes more sense to keep it at the top level (like DSpace Objects).

These are the points to consider when designing an LNI resource:

- How do URIs map to the underlying objects? Hierarchy, one collection, or completely flat space? What are the identifiers and how do they map to URIs?
- Does the resource have any children? What do the child URIs look like?
- What WebDAV properties does the resource support? It must respond to the required properties `DAV:displayname`, `DAV:resourcetype`, `DAV:current_user_privilege_set`, and `dspace:type`.
- Which of its properties can be set or deleted?
- What do the `GET` and `PUT` methods do (if anything)?
- How does the resource react to WebDAV methods like `COPY`, `DELETE`, etc.?

Resource Example: Bitstream Format Registry

Model the Bitstream Format registry as one resource of the "collection" type. That top-level resource is just a container for the format records, so they can all be listed (e.g. by `PROPFIND`). The formats themselves are child resources of the Registry, named by the string returned by `getShortDescription(` on the format.

- The Registry URI is _/prefix_`/format/`
- Each Bitstream Format's URI is _/prefix__`/format/`_short-description
- `GET` is not implemented on any of these resources.
- `PUT` of a new Bitstream Format URI creates it. The request body must be empty.
- Bitstream Format resources have the following properties. They are all settable unless marked read-only.
    - `DAV:displayname` – same as *description*
    - `DAV:resourcetype` – empty. (`/format/` is a collection).
    - `dspace:type` – `<bitstream-format/>`
    - **`description`** – Full description of the bitstream format.
    - **`extensions`** – Comma-separated list of filename extensions associated with this format.
    - **`ID`** – READ-ONLY; internal database ID of this format.
    - **`MIMEType`** – Internet format type, aka MIME type, associated with this format. May not be unique amongst formats.
    - **`support Level`** – Support level, one of `UNKNOWN`, `KNOWN`, `SUPPORTED`.
    - **`is Internal`** – Is this format useed to store internal system information, rather than content? One of `TRUE`, `FALSE`.

Since the format resources are named by the "short description", it is a simple matter to find a format if you know the short description – just compose the URI and make a `PROPFIND` request. Otherwise list all formats with a `PROPFIND` request of depth 1 on the parent collection, `/format/`.

It might be helpful to add query args to the `/format/` resource to search out formats, e.g. by filename extension. For example, the URI `/format?[findBy Extension]=pdf` would return a `Location:` header redirecting to the format URI that has `pdf` among its extensions.

Other good Candidates for New Resource Types

- Metadata Registry (formerly Dublin Core registry).
- EPerson Groups – collection (or hierarchy?) of e-person Groups.
- Browse – Render Browse object as a collection resource leading to Items (and author names).

Adding WebDAV Methods

The LNI can also be extended by adding support for all of the core WebDAV operations, according to these guidelines. The missing operations are:

- PUT Method:: Implement `PUT` on a Bitstream resource to replace its contents, and on an Item to replace it with the contents of a new package. This is waiting for policy decisions and possibly a versioning mechanism.
- DELETE Method:: DSpace objects are usually never deleted, although the object API supports it. Perhaps `DELETE` is worth implementing only for *some* types of resources.
- MKCOL Method:: Create a new collection, i.e. DSpace Collection or Community. Have it create an empty/blank object and then set properties with `proppatch`.
- MOVE Method:: Possibly useful for rearranging the namespace, i.e. moving children of Collections and Communities around.
- LOCK Method:: Not implemented, and not strictly required by WebDAV. Recommended that this be left out until DSpace has versioning.
- Versioning:: Not implemented, and not strictly required by WebDAV. Recommended that this be left out until DSpace has versioning.

See Also

LightweightNetworkInterface DownloadsAndClients for a sample implementation (unstable) and a pre-built test client.

Your Comments

Here's a (reciprocal) cross-reference link to a page about LNI over on the CWSpace project's wiki: – William Reilly 2005-12-07T15:07:01Z

- Lightweight Network Interface Some client test code, at CWSpace. (Note that this page at DSpace's wiki remains the place for full description, discussion.)

Posting (long after the fact) this visual aid to LNI submit process.

9 pp. PDF showing schematic description of LNI Submit from MIT OpenCourseWare to MIT DSpace.

- MIT-LNI-DSUG-Bergen2006-SUBMIT-DIAGRAM-9-slides.pdf

Complete PPT from which 9 pages above are extracted

- http://dsug2006.uib.no/archive/reilly.ppt
  "Technical Introduction To and Initial Use Of the Lightweight Network Interface (LNI) (to DSpace!)"
  DSpace User Group Meeting, Bergen, Norway, April 2006

Information accurate as of presentation date.

Wreilly 15:42, 12 June 2009 (EDT)