Google Summer of Code 2009 DSpace REST Webapp

DSpace REST API and Web Application

Details

Project Title: DSpace REST Webapp

Student: Bojan Suzic, University of Technology Graz

Mentor: Aaron Zeckoski

Contacting author: bojan AT trust - b . com using subject line [DSPACE:DSpace]

SCM Location for Project: http://scm.dspace.org/svn/repo/modules/rest

Project Summary

A RESTful service as DSpace addon is to be implemented, enabling guest and authorized users to browse and retrieve DSpace collections and related data.

The principles to follow:

- Stateless communication
- Separation of concerns: methods (GET/PUT/DELETE) are used according to their designation
- JSON and XML will both be supported as output formats
- Configuration interface for administrator to control aspects of functionality
- Logging of requests will be handled via the framework
- The API will be versioned, enabling easier upgrades in the future
- The (return) status codes should be handled according to the HTTP spec
- Resource retrieval (books...) should be possible (to decide later: binary encoding or forwarding approach)
- BasicAuth will be supported for authentication; X509 support for user logging would be a good idea

Endpoint (API methods) descriptions

Available endpoints are described here. Please note that this list is **not final or complete**. Suggestions and comments are welcomed.

The required parameters are these found in path of the request URL in most cases (except where noted). Optional parameters are found in the query part of the URL. No optional parameters are found in the URL path, except one defining format (see bellow).

Optional parameters should indicate the default value when shown in the API definitions below. For example, `?thing=true` indicates that if the thing param is not included it will default to true. For the parameters without predefined value explicitly mentioned, it is assumed that the value is not predefined at all. It means that it is not required, but using this parameter usually produces narrower results if such are requested.

The optional version parameter in query can be used when necessary like this: `?version={version}`. If no version is specified then the current version will be returned or used. Currently it is not supported.

Universal parameters

These parameters are valid for each call and as such are not explicitly mentioned in the specification tables.

For the **format**, by default it is determined using the ACCEPT header (e.g. `setRequestHeader` in JS) but optionally may be specified in the URL as a suffix like `.json|.xml` (e.g. `/thing/item.json`). JSON is used by default if there is no ACCEPT header parameter present and the format is not indicated. The ACCEPT header overrides the format suffix. If there is wrong (unsupported) accept-header set, then the status code `415 Unsupported Media Type` shall be returned.

Authentication is to be based on provided parameters, supporting cookies and basic auth. For the authentication, if parameters are omitted (and no cookie present), the guest (readonly/public) user is used, otherwise user is authenticated according to provided parameters (`?user={username}&pass={password}`) or cookie (in this order). Parameters can be included in header too, in this case header has precedence over other methods. Later the possibility to use X509 certificate could be implemented.

In all cases, if the requested resource is out of reach of the user, the errors `401 Unauthorized` (not logged in) OR `403 Forbidden` (logged in but not allowed) are used accordingly.

For the **searching/sorting** methods, we will follow OpenSearch guidelines and RoR conventions - where it is applicable. The following list with supported requests is maintained and updated when needed. These will be valid for each endpoint which uses GET unless otherwise noted in the API.

When searching for entities in a list, the following parameters are handled specially in the system (note that all the RoR conventions are followed for sorting /paging):

- `_start={number}`: the position of the first entity to return (0 is the first, default), e.g. `_start=5`
- `_page={number}`: the page of data to display (0 is first, default), e.g. `_page=2`
- `_perpage={number}`: the number of entities to return for the page (0 means all, default), e.g. `_perpage=20`
- `_limit={number}`: the maximum number of entities to return (0 means all, default), e.g. `_limit=50`
- `_order={string}`: the sort order to return entities in (default is ascending), should be a comma separated list of entity field names which optionally include a suffix to determine order, suffix can be `_reverse` or `_desc` for descending order OR '' (blank) or `_asc` for ascending order, e.g. `_order=name` OR `_order=name_reverse` OR `_order=name,email_desc,firstname_asc,lastname_reverse`
- `_sort={string}`: same as order

This part usually may generate two (error) status codes: `204: No content`, in the case there are no fields satisfying criteria, and `400: Bad request`, in the case the query is malformed or incompatible parameters are used.

The searching criteria is applied only on items returning full info. Items returning only ids (`idOnly=true`) are not sent to sorting/filtering procedures.

Information usually returned

In the most cases there are two types of returning information entities:

- first, defined with `idOnly=true`, which returns only ids of entities satisfying request
- second, used by default, returning all available info

In the second case included is info for related entities. For instance, when user browses collection, it also receives information about communities related to collection, items related to it and so on. This principle goes through several layers. For instance, Collection -> Item -> Bitstream. So, in one request all these information are present.

Exception is present in the cases where chaining is possible. After some extent, not all information about sub/related-entities are sent, but only their ids. Example: Collection -> Item -> Bundle -> Bitstream -> BundleId. As Bitstream and Bundle are mutually referenced and included, this would cause unlimited chaining. For this reason the mechanism is implemented which encapsulates only id of entities after some extent. For more details please take a look at the example and code.

Browsing methods

Name and description	Value and notes
Base URI:	
Description:	Returns a list of all communities on the system or return just top level communities.
HTTP method:	`GET`
Optional parameters:	
Sorting fields supported:	`id, name, countitems`: sorting by id, community name and item count
Response formats:	`json`, `xml`
Status codes	200: OK 204: no content 400: bad request 500: internal server error
Response details	

Name and description	Value and notes
Base URI:	
Description:	Returns a list of all parent communities of the `id` community.
HTTP method:	`GET`
Optional parameters:	
Sorting fields supported:	`id, name, countitems`: sorting by id, community name and item count
Response formats:	`json`, `xml`

Status codes	200: OK 204: no content 400: bad request 500: internal server error
Response details	

Name and description	Value and notes
Base URI:	
Description:	Returns a list of immediate sub-communities (children) of the `id` community.
HTTP method:	`GET`
Optional parameters:	
Sorting fields supported:	`id, name, countitems`: sorting by id, community name and item count
Response formats:	`json`, `xml`
Status codes	200: OK 204: no content 400: bad request 500: internal server error
Response details	

Name and description	Value and notes
Base URI:	
Description:	Returns a list of collections in the `id` community
HTTP method:	`GET`
Optional parameters:	
Sorting fields supported:	`id, name, countitems`: sorting by id, collection name and item count
Response formats:	`json`, `xml`
Status codes	200: OK 204: no content 400: bad request 500: internal server error
Response details	

Name and description	Value and notes
Base URI:	
Description:	Returns a list of recent submissions to a community
HTTP method:	`GET`
Optional parameters:	
Sorting fields supported:	`id, name, lastmodified, submitter`: sorting by id, name(title), last modified date and submitter(name) of item
Response formats:	`json`, `xml`
Status codes	200: OK 204: no content 400: bad request 500: internal server error
Response details	

Name and description	Value and notes
Base URI:	
Description:	Returns a list of all collections in the system
HTTP method:	`GET`
Optional parameters:	
Sorting fields supported:	`id, name, countitems`: sorting by id, collection name and item count
Response formats:	`json`, `xml`

Status codes	200: OK 204: no content 400: bad request 500: internal server error
Response details	

Name and description	Value and notes
Base URI:	
Description:	Returns a list of all communities a collection with `id` belongs to
HTTP method:	`GET`
Optional parameters:	
Sorting fields supported:	`id, name, countitems`: sorting by id, community name and item count
Response formats:	`json`, `xml`
Status codes	200: OK 204: no content 400: bad request 500: internal server error
Response details	

Name and description	Value and notes
Base URI:	
Description:	Returns a list of all items from the collection `id`
HTTP method:	`GET`
Optional parameters:	
Sorting fields supported:	`id, name, lastmodified, submitter`: sorting by id, name, lastmodified date and submitter of item
Response formats:	`json`, `xml`
Status codes	200: OK 204: no content 400: bad request 500: internal server error
Response details	

Content searching

Name and description	Value and notes
Base URI:	
Description:	Returns a list of all objects found by searching criteria
HTTP method:	`GET`
Optional parameters:	
Sorting fields supported:	`id, name, lastmodified, submitter`: sorting by id, name, last modifed date or submitter of item
Sorting/ordering modifiers:	`title, issueDate, author, subject, submitter`
Response formats:	`json`, `xml`
Status codes	200: OK 204: no content 400: bad request 500: internal server error
Response details	

Name and description	Value and notes
Base URI:	
Description:	Returns a list of all objects that have been created, modified or withdrawn within specified time range
HTTP method:	`GET`

Optional parameters:	
Sorting/ordering modifiers:	`id, name, lastmodified, submitter`: information on item returned
Response formats:	`json`, `xml`
Status codes	200: OK 204: no content 400: bad request 500: internal server error
Response details	

Item status/info and retrieval

Name and description	Value and notes
Base URI:	`/items/{id}`
Description:	Returns detailed information about an item
HTTP method:	`GET`
Required parameters:	{`id`}: item id
Sorting fields supported:	`id, name, lastmodified, submitter`: sorting by id, name, last modifed date or submitter of item
Response formats:	`json`, `xml`
Status codes	200: OK 204: no content 400: bad request 500: internal server error
Response details	Contains an information about an item including resource name, metadata, owning collection, collections stored in, communities stored in, bundle ids, last modified date, archival/withdrawn status and submitter of an item

Name and description	Value and notes
Base URI:	`/items/{id}/permissions`
Description:	Returns status of user permissions on this item
HTTP method:	`GET`
Required parameters:	{`id`}: item id
Response formats:	`json`, `xml`
Status codes	200: OK 400: bad request 500: internal server error
Response details	Boolean variable, stating can user edit the listed item

Name and description	Value and notes
Base URI:	
Description:	Returns communities this item is part of
HTTP method:	`GET`
Required parameters:	
Sorting fields supported:	`id, name, countitems`: community properties used for sorting
Response formats:	`json`, `xml`
Status codes	200: OK 400: bad request 500: internal server error
Response details	Communities listed

Name and description	Value and notes

Base URI:	
Description:	Returns collections this item is part of
HTTP method:	`GET`
Required parameters:	
Sorting fields supported:	`id, name, countitems`: collection parameters
Response formats:	`json`, `xml`
Status codes	200: OK 400: bad request 500: internal server error
Response details	Collections listed

Name and description	Value and notes
Base URI:	`/bitstream/{id}`
Description:	Returns bitstream object - usually the library item file
HTTP method:	`GET`
Required parameters:	{`id`}: bitstream item id
Response formats:	`json`, `xml` (not yet complete)
Status codes	200: OK 404: Not found 400: bad request 401: Unauthorized 403: Forbidden 500: internal server error
Response details	Includes all information about referenced bitstream, including file name, licence, corresponding ittem etc. It is possible only to get information for particular bitstreams. When the request is made without parameters/references, the blank list is presented (there is no list of all bitstreams in the system available).

Name and description	Value and notes
Base URI:	`/bitstream/{id}/receive`
Description:	Returns checksum of bitstream
HTTP method:	`GET`
Required parameters:	{`id`}: bitstreamitem id
Response formats:	`binary`
Status codes	200: OK 400: bad request 401: Unauthorized 403: Forbidden 500: internal server error
Response details	Receive full bitstream

User-oriented functions

Name and description	Value and notes
Base URI:	
Description:	Returns list containing id, name and email of persons (optionally matching a query)
HTTP method:	`GET`
Optional parameters:	

Sorting fields supported:	`id, name, lastname, fullname, language`: sorting properties of user(submitter) supported
Response formats:	`json`, `xml`
Status codes	200: OK 204: no content 400: bad request 500: internal server error
Response details	List with information on particular user. Additionaly only identifiers are sent if idOnly is true.

Statistical info

Name and description	Value and notes
Base URI:	`/stats`
Description:	Returns general statistics
HTTP method:	`GET`
Response formats:	`json`, `xml`
Status codes	200: OK 400: bad request (if there is no stats package available) 500: internal server error
Response details	Returns cummulative list of statistics data for the system currently available

Comments

Concerning DSpace Data Model exposure in REST Paths

I am concerned about the adoption of the 1.x dspace data model, which in 2.0, is not hardcoded in this manner,
entity resource "type" being part of the url path. We are trying to move away from this convention and for the content and represent a generic mechanism for traversing and manipulating the graph/hierarchy of the resources (entities) .

I think we should treat them as such and think about how such resource/entity graphs are traversed using rest

Rather than: /communities/{id}/parents?idOnly=false&immediateOnly=true

We have something more like

```
/resource/{id}/related?relation=ds:isPartOfCommunity&idOnly=false&immediateOnly=true
```

Rather than: /communities/{id}/children?idOnly=false&immediateOnly=tru

We have

```
/resource/{id}/related?relation=ds:hasCommunityPart&idOnly=false&immediateOnly=true
```

I think we need to make sure the REST interfaces clearly map to 2.0 Services and the actions that can be performed on them. So harvest, stats and users make sense to me. But, Community, Collection, Item and Bitstream do not and we should be consolidating these under some service path like "content/" or "resource/" or the like.

--Mark Diggory 16:04, 12 July 2009 (EDT)

See Fedora REST API for reference

Please see for reference:

Fedora REST
Fedora API-M
Fedora API-A

for some examples of methods appropriate for the entity relationship model we are considering for 2.0

addRelationship

Creates a new relationship in the object. Adds the specified relationship to the object's RELS-EXT datastream. If the Resource Index is enabled, the relationship will be added to the Resource Index.

The DSpace 2.0 proposed mapping to Fedora places RDF references for ds:hasCollection/ds:isPartOfCollection, ds:hasCommunity/ds:isPartOfCommunity directly into the RELS-EXT as relationships between Fedora representations of DSpace objects.

URL Syntax

```
/objects/{pid} ? [relationship] [object] [isLiteral] [datatype]
```

Parameters:

- pid: The PID of the object.
- relationship: The predicate.
- object: The object.
- isLiteral: A boolean value indicating whether the object is a literal.
- datatype: The datatype of the literal. Optional.

For DSpaceObjects:

(a) Creates either a new Top Level Community, SubCommunity, Collection, Item, Bundle or Bitstream as defined in the DSpace Data Model, the context of which is the current {pid} entity

Get next pid, /objects/nextPID ? [DSPACE:type]

```
 /objects/nextPID?type="http://purl.org/dspace/model/Bitstream"

 /objects/{bundlePid}?relationship="http://purl.org/dspace/model/hasBitstream"&object={bitstreamPid}

 /objects/{bitstreamPid} ? ... see http://www.fedora-commons.org/documentation/3.0/userdocs/server/webservices
/rest/index.html#addDatastream
```

(b) Creates metadata properties attached to any of the above DSpace Objects.

```
/objects/{pid} ? relationship=http://purl.org/elements/1.1/title&object="My Title"&isLiteral=true
```

addDatastream

URL Syntax

```
/objects/{pid}/datastreams/{dsID} ? [controlGroup] [dsLocation] [altIDs] [dsLabel] [versionable] [dsState]
[formatURI] [checksumType] [checksum] [logMessage]
```

--Mark Diggory 15:58, 12 July 2009 (EDT)

References

Microformats conventions

RFC2616 Method Definitions

RFC2616 Status Code Definitions

Fedora API-M