Binary Large Object Store – REST API Documentation

James Craig Lowery, 2016

THIS DOCUMENT AND THE ACCOMPANYING CODE IS A WORK IN PROGRESS.
IT IS FUNCTIONAL, YET INCOMPLETE, AND SUBJECT TO FUTURE CHANGES.

REVISION HISTORY

January 5, 2016	Initial draft
January 11, 2016	First revision
February 2, 2016	Second revision
February 18, 2016	Third revision

Overview

The Binary Large Object Store, simply called “the repository” in this document, is a storage service which accepts BLOBs – things such as video files, audio files, photos, etc. – and stores them for later retrieval. The repository adds value beyond that of a network shared volume in these ways:

- Access to the store is via REST/HTTP, the details of which are documented herein.

- An object is uniquely identified by a *handle*, which is simply a non-negative integer value that the repository assigns to the object upon creation. Once the handle is known, the metadata can be queried and manipulated, and the content retrieved or updated.

- BLOB’s are stored as version-controlled “objects” having descriptive meta-data: attributes and tags.
  - **Versions:** An object “version” is created whenever a new BLOB is associated with (uploaded to) the same object handle. When the first version of an object is uploaded, it becomes the initial version and the object is created and is assigned a handle. If subsequent BLOBs are uploaded to the same handle, then the older versions of the BLOB are kept but not considered current. Versioning is important because sometimes you may want to update an object’s BLOB replica with a newer version of the content (say, a better copy of a video) without having to replicate all of the object’s metadata (tags). If there is something wrong with the new version, you can “roll back” to the previous one quickly.

  - **Attributes:** Attributes are descriptive metadata about a specific version of the object, such as its title, BLOB size, date of import into the repository, and its health status (has the BLOB been corrupted since upload, for example). The names and types of attributes are set and do not change, although attribute values change as the system manages the BLOB over its lifecycle. The only attributes end-users may modify is the title. The content type will be inferred from the file name suffix at the time of import, and must be registered with the system to be recognized properly. (A future improvement is to support Content-Type headers directly.)

  - **Tags:** Tags are metadata that are descriptive about the object. Unlike attributes, tags are user-defined. You can create as many as you like to suit your needs. Conceptually, a tag is a name=value pair attached to the object.

    The repository tracks all the tag names in the system, and the different values each tag name has been paired with. To ensure that tagging doesn’t get sloppy, the repository insists that you tell it ahead of time what tag names and values you intend to use, before you try to tag an object. Without this constraint, you are likely to misspell names or values, or use them differently in ways that the repository will not be able to tell they are meant to be the same thing. For example, you may tag a video “Genre=SciFi” and another video “Genre=Science Fiction” which in your mind are the same, but are different in the logic used by the repository.

Not only can you define the tag names and values available to you, you can also tag objects with any number of name=value pairs, even if the same name is used twice. For example, if a video is both genre science fiction and drama, you can tag it twice with "Genre=Science Fiction" and "Genre=Drama". If you don't care to tag a video with the name of the director, you don't have to. The repository gives you ways to test set membership of objects in tag values.

Tags are not currently version controlled, although tag versioning will likely be introduced in future versions of this software. For example, tags like "Director" or "Genre" are unlikely to need versioning just because another, better copy of a video is uploaded for the object. However, a tag like "Quality" would be better if it tracked with the specific version of the object to which it was originally assigned.

- The repository provides automated health monitoring of the data that has been entrusted to it. Upon import, a checksum is computed for each BLOB version. The size of the BLOB and its checksum are periodically checked against what was recorded at import time. If there is a change, then you will be alerted to the corrupted data through information the repository records in the object's attribute set. The repository also keeps track of the last time it performed a health check and found things to be satisfactory. In this way, you can get a reasonable idea of when the corruption occurred, to help spend less time in recovering a healthy version of the BLOB from backup.

  The probability of a corruption is dependent upon how the service is implement (how this software is used). The actual BLOB storage mechanism is abstracted from the code, and could be something as simple as storage on a single hard disk, to using a highly reliable cloud storage service. On some systems, automatic recovery of the BLOB may be performed for you, and you never have to worry about health checks because it is all done in the background: You just see a highly reliable repository that automatically repairs corrupted objects as needed.

Error Resource Interface

In most cases when an error occurs during interface interactions defined within this document, the server will be able to return an XML element containing an `error` object that describes the error in detail. In the cases where the application server framework code detects the error and the interface code is never invoked, no `error` object can be returned (which is why the caveat “if possible” is mentioned).

The format of the error object is:

```
<error>
  <code>errorcode</code>
  <description>language-specific text</description>
  <detail>additional information</detail>
  <cause>unstructured text of what caused it</cause>
</error>
```

The *errorcode* is a unique non-negative integer value associated with the error. The description may vary for installations that have been localized to a language other than English. The detail element may have information such as an offending parameter value. The cause is highly variable with respect to the incident that created the error, and is typically a stack trace or Java exception message.

A list of error codes and descriptions can be retrieved using the following method:

```
  GET /errors/catalog
```

It returns an XML document of the format:

```
  <errors>
    <error>
      <code>errorcode<code>
      <description regioncode=”regioncode”>text</description>
    <error>
    ...
  </errors>
```

The `regioncode` attribute is one of the standard region codes from the ISO 639-1 standard, as described in RFC 5646. All descriptions in all available languages will be returned.

Object Resource Interface

The Object Resource Interface is used to retrieve, modify, and delete objects from the repository.

An object is an entity comprising one or more versions, and a set of tag assignments. Each version is a collection of attribute names and values, and a distinct BLOB (binary large object) which is the content that they describe. For example, the BLOB may be a video byte stream, and the attributes include its title, size, etc. Objects have at least one version, and always have a version designated as the "current" version (the most recent version created). The names and types of attributes, and which ones can be changed via this interface, can be retrieved through the `/object/schema` resource.

Objects and new object versions are created through the Upload Resource Interface. Objects are identified by a unique non-negative integer *handle* which is assigned automatically upon creation of the first version.

Objects may also be tagged with a tagging assignment, which is a set of *tagname*=*tagvalue* pairs. Tagging assignments are not versioned – there is only one set of tagging assignments for the object, and they are not recoverable through rollback, as are the attributes and BLOB contents. (This may change in future releases.) Tags differ from attributes in that they convey meta-information about what the object represents, rather than a particular version. Also, the number of tags and values is not predefined, meaning that new tag names and values can be created at any time, and an object can be tagged with multiple tags, and even the same tag name can be used multiple times with different values. This is useful when categorizing objects in multiple categories such as the tag "Genre" having values "Comedy" and "Science Fiction" on an object.

Here is a summary of the entry points in this interface:

Resource	**Method**	**Meaning**
/objects/schema	GET	Retrieve the object schema.
/objects	GET	Retrieve XML for all objects
/objects/*handle*	GET	Retrieve XML for a specific object
/objects/*handle*/download?versioncount=*int*	GET	Retrieve the URL for the content of this object.
/objects/*handle*	PUT	Modify the current version of an object
/objects/*handle*	DELETE	Retire an object and all its versions to the trash
/objects/*handle*/currentversion	DELETE	Roll back to the most recent old version of the object.

This is an outline schema of the XML returned and consumed by this interface:

```
<objects>
  <object handle="handle">
    <versions>
      <version current="true|false">
        <attributes>
          <attributename>value</attributename>
          ...
        </attributes>
        <references>
          <reference mode="mode">reference</reference>
          ...
        </references>
      </version>
        ...
    <versions>
    <tags>
      <tag name="tagname" value="tagvalue"/>
      ...
    <tags>
  <object/>
  ...
</objects>
```

The `<references>` container inside of each version provides a list of references to the object's content. The "download" mode is always available, the reference being the download URL through this interface. Other modes are installation-defined and could be things like "smb" or similar mode keys that client software can look for in order to find and consume the content of that version. For the reference implementation in which there is a SMB (Samba) server on the local area network which exposes the files backing the content library, the mode is "localsamba" and the reference is a Windows universal path, such as "\\JCLWHS\videos\Television\Comedy\I Dream of Jeannie\101 – My Master.avi".

GET /objects/schema

Returns a simple XML schema document (not XSD – see below) defining the names, type, and mutability of attributes of objects.

```
<schema type="object">
  <attribute name="name" type="type" readonly="yes|no"/> ...
</schema>
```

`type` can be one of:

- string
- integer
- double
- boolean
- timestamp

RETURN STATUS	RETURNED CONTENT
200 OK	XML document rooted by the `schema` element.
500 INTERNAL SERVER ERROR	If possible, an XML document rooted by an `error` element.

GET /objects

Returns a representation of all objects in the database in an XML `objects` element of `object` elements. This is effectively a “dump” of the repository’s meta data and can be quite large. It may only be accessible by users that are specifically granted access to this entry point since it can be resource intensive for the server and network.

RETURN STATUS	RETURNED CONTENT
200 OK	XML document rooted by the `objects` element.
500 INTERNAL SERVER ERROR	If possible, an XML document rooted by an `error` element.

GET /objects/*handle*

Returns metadata for a single object, as identified by its non-negative integer *handle*, in an XML `object` element.

RETURN STATUS	RETURNED CONTENT
200 OK	XML document rooted by the `object` element.
400 BAD REQUEST	If possible, an XML document rooted by an `error` element.
404 NOT FOUND	If possible, an XML document rooted by an `error` element.
500 INTERNAL SERVER ERROR	If possible, an XML document rooted by an `error` element.

GET /objects/*handle*/download?versioncount=*versionid*

Returns the content of an object as an application/octet-stream to be downloaded into a named file, the suggested name of which is sent via an HTTP Content-Disposition header. If `versioncount` is not specified, then the current version is retrieved.

RETURN STATUS	RETURNED CONTENT
200 OK	The byte-stream of the object's content (application/octet-stream).
400 BAD REQUEST	If possible, an XML document rooted by an `error` element.
404 NOT FOUND	If possible, an XML document rooted by an `error` element.
500 INTERNAL SERVER ERROR	If possible, an XML document rooted by an `error` element.

PUT /objects/*handle*

Updates the metadata for the object with the specified non-negative integer *handle*, using the `object` root element of the XML request payload as a source. The `object` element must have a handle attribute whose value matches the handle specified in the URL.

If a `versions` element is included as a subordinate to the `object` element, then only the following attributes of the first subordinate `version` element marked as `current="true"` will be updated into the current version of the object in the repository. All older or other versions, and all other attributes in the first version marked as the current version are ignored.

- `<title/>`

If a `tags` element is included, then the tag assignments for the object are replaced with those contained within the `tags` element. This provides a "wholesale" mechanism for updating tagging data, but the tag names and values must already be defined, otherwise the entire update will fail and no changes will be made. If a `tags` element is NOT included, the current tagging assignments are not modified.

RETURN STATUS	RETURNED CONTENT
200 OK	`<success/>`
400 BAD REQUEST	If possible, an XML document rooted by an `error` element.
404 NOT FOUND	If possible, an XML document rooted by an `error` element.
500 INTERNAL SERVER ERROR	If possible, an XML document rooted by an `error` element.

PUT /objects/*handle*/tags/tag/*name*/*value?*autocreate=*true|false*

Creates a tagging assignment for the object of *name*=*value*. The tag named *name* must exist in all cases. The *value* must exist unless autocreate is true, in which case the value will be created automatically if it does not already exist.

RETURN STATUS	**RETURNED CONTENT**
200 OK	`<success/>`
201 CREATED	`<success/>`
400 BAD REQUEST	If possible, an XML document rooted by an `error` element.
404 NOT FOUND	If possible, an XML document rooted by an `error` element.
500 INTERNAL SERVER ERROR	If possible, an XML document rooted by an `error` element.

DELETE /objects/*handle*/tags/tag/*name*/*value*

Deletes the tagging assignment *name*=*value* for the specified object, if it exists.

RETURN STATUS	**RETURNED CONTENT**
200 OK	`<success/>`
400 BAD REQUEST	If possible, an XML document rooted by an `error` element.
404 NOT FOUND	If possible, an XML document rooted by an `error` element.
500 INTERNAL SERVER ERROR	If possible, an XML document rooted by an `error` element.

DELETE /objects/*handle*/delete?force=*true|false*

Retires an object and all of its versions and tagging information to the trash, effectively deleting it from the repository. The content can be manually recovered from the trash by a system administrator.

Consistency of the object is first checked before deletion occurs. If there a problem, such as the version numbers are out of sequence with respect to timestamps, the current timestamp doesn't match the most recent timestamp, or there are no versions (there are other possibilities) then the deletion will fail unless `force` is `true`. The default value for `force` is `false`.

RETURN STATUS	**RETURNED CONTENT**
200 OK	`<success/>`
400 BAD REQUEST	If possible, an XML document rooted by an `error` element.
404 NOT FOUND	If possible, an XML document rooted by an `error` element.
500 INTERNAL SERVER ERROR	If possible, an XML document rooted by an `error` element.

DELETE /objects/*handle*/rollback

Provided there is an older version, the current version of the object is retired to the trash and the remaining most recent version becomes the current version. This effectively "rolls back" an object from any recently committed upload of a newer version. If there is only one version and none to which we can roll back, then nothing changes, and it is not considered an error. On success, a single XML element with the timestamp of the current version is returned.

RETURN STATUS	**RETURNED CONTENT**
200 OK	<success><imported>*currenttimestamp*</imported></success>
400 BAD REQUEST	If possible, an XML document rooted by an `error` element.
404 NOT FOUND	If possible, an XML document rooted by an `error` element.
500 INTERNAL SERVER ERROR	If possible, an XML document rooted by an `error` element.

Tagging Resource Interface

The Tagging Resource Interface is used to define tag names and values that may be used in tag assignments on objects. Tag names are strings very much like variable names used in programming languages. Although the underlying tagging system places very few restrictions on what can be used as a tag name, the Query Resource Interface has limitations on the tag names it can identify in queries, so it is best to use tag names that are simple and "identifier-like." In most cases, tag names are not case sensitive, but again this depends on the interface through which they are being manipulated, so case sensitivity is better assumed than not.

A tag name is represented by the XML element `<tag/>` which has three attributes:

- `name="`*name*`"` – the name of the tag
- `type="`*type*`"` – where type is one of `Category`, `Entity`, or `Sequence`. This is used by browsing client to interpret the type of information the tag is providing, whether it is categorical (like Science Fiction, Drama, Television), an entity (like Director, Studio, Actor), or if it is used `com.craiglowery.java.vlib.common.U_Exception: Unexpected at select: org.w3c.dom.DOMException: INVALID_CHARACTER_ERR: An invalid or illegal XM` to sequence material (like Episode, Series). This information means nothing to the repository, but browsing clients may depend upon it.
- `description="`*description*`"` – optional descriptive information about what the tag represents.

Each tag name has a defined set of string values that can be associated with that tag name when assigned to an object. This set must be non-empty – there must be at least one value from which to choose. A name value pairing is represented by the XML element `<value/>` that is subordinate to `<tag/>`.

Here is a summary of the entry points into this interface:

Resource	**Method**	**Meaning**
/tags?excludevalues=*true*|*false*	GET	Returns all tags and (optionally) their values
/tags/*name*?excludevalues=*true*|*false*	GET	Returns a single tag definition and (optionally) its values.
/tags/*name*/*value*	GET	Returns a single value for a tag if it exists.
/tags/*name*	PUT	Creates or updates a tag definition.
/tags/*name*	DELETE	Delete's a tag definition and values.
/tags/*name*/*value*	DELETE	Delete's a single name=value tag definition.

Here is an outline schema of the XML returned and consumed by this interface:

```
<tags>
  <tag name="name" type="tagtype" description="desc"
browsing_priority="priority">
    <value>value</value>
    ...
  </tag>
</tags>
```

Calls to the interface always may return any of the elements above as the root of the XML result. Calls do not consume XML payloads unless noted.

GET /tags?excludevalues=*true*|*false*

Returns all tagging definition information when `excludesvalues` is false (the default). If `excludevalues` is true, returns just the tag name definitions without values.

RETURN STATUS	RETURNED CONTENT
200 OK	The `<tags/>` element.
400 BAD REQUEST	If possible, an XML document rooted by an `error` element.
500 INTERNAL SERVER ERROR	If possible, an XML document rooted by an `error` element.

GET /tags/*name*?excludevalues=*true*|*false*

Returns the tagging definition and values for the name if it exists, and excludevalues is false. If excludevalues is true, only the tag definition is returned.

RETURN STATUS	RETURNED CONTENT
200 OK	A `<tag/>` element.
400 BAD REQUEST	If possible, an XML document rooted by an `error` element.
404 NOT FOUND	If possible, an XML document rooted by an `error` element.
500 INTERNAL SERVER ERROR	If possible, an XML document rooted by an `error` element.

GET /tags/*name*/*value*

Returns a single value element for the named tag, if such exists. Most callers will ignore the returned XML and simply look for the return code as a way to verify existence of the tag name=value pair.

RETURN STATUS	RETURNED CONTENT
200 OK	A `<value/>` element.
400 BAD REQUEST	If possible, an XML document rooted by an `error` element.
404 NOT FOUND	If possible, an XML document rooted by an `error` element.
500 INTERNAL SERVER ERROR	If possible, an XML document rooted by an `error` element.

PUT /tags/*name*

Creates a new tag name.

RETURN STATUS	RETURNED CONTENT
200 OK	`<success/>` The tag name already exists.
201 CREATED	`<success/>` The tag name was created.
400 BAD REQUEST	If possible, an XML document rooted by an `error` element.
500 INTERNAL SERVER ERROR	If possible, an XML document rooted by an `error` element.

PUT /tags/*name*/*value*

Creates a new value for a tag. The tag should have been previously created.

RETURN STATUS	RETURNED CONTENT
200 OK	`<success/>` The value already exists.
201 CREATED	`<success/>` The value was created.
400 BAD REQUEST	If possible, an XML document rooted by an `error` element.
404 NOT FOUND	If possible, an XML document rooted by an `error` element.
500 INTERNAL SERVER ERROR	If possible, an XML document rooted by an `error` element.

DELETE /tags/*name*

Deletes tagging information. All values and the tag name will be targeted for deletion. No changes are made if any of the values proposed for deletion are in use in a tagging assignment to an object. Either all deletions will be made as described above, or nothing will be changed.

RETURN STATUS	RETURNED CONTENT
200 OK	`<success/>`
400 BAD REQUEST	If possible, an XML document rooted by an `error` element.
404 NOT FOUND	If possible, an XML document rooted by an `error` element.
500 INTERNAL SERVER ERROR	If possible, an XML document rooted by an `error` element.

DELETE /tags/*name*/*value*

Deletes tagging information. Only the specified name=value pairing will be targeted for deletion. If this is the last value defined for the tag, the tag will still exist even after the value has been deleted. No changes are made if the value proposed for deletion is in use in a tagging assignment to an object.

RETURN STATUS	RETURNED CONTENT
200 OK	`<success/>` The deletion was successful.
400 BAD REQUEST	If possible, an XML document rooted by an `error` element.
404 NOT FOUND	If possible, an XML document rooted by an `error` element.
500 INTERNAL SERVER ERROR	If possible, an XML document rooted by an `error` element.

Upload Resource Interface

The Upload Resource Interface is used to create object versions and import them into the database. New objects are created by uploading a version with no associated object handle, in which case a new object is created and handle assigned and returned for future use. If a handle is provided with the upload, then the uploaded BLOB becomes the new "current" version of the object, and the old "current" version is retained in a LIFO stack, allowing for future rollback to previous versions provided they have not been otherwise deleted.

Since the interface is RESTful, there is no *de facto* session. However, as the BLOB payloads are usually larger than what most web and application servers will allow for incoming documents, they must be uploaded in parts over several consecutive HTTP PUT transactions. An "upload resource" (UR) is a unique, temporary resource associated with the user's access credentials. It functions as an initially zero-length byte-addressable file that grows automatically as bytes are written beyond the current end-of-file.

A UR's state can be obtained with a GET. A sha1sum can be computed on-demand by providing a query parameter, otherwise the sha1sum will not be provided because it cannot be "computed along the way" as segments are uploaded, in keeping with REST API's design principals. You are allowed to upload any content of any size to any (reasonable) offset within the UR at any time within the life span of the UR. It is up to you to ensure you have uploaded all the necessary bytes to the proper locations. Typically, one only computes this checksum after all parts are uploaded, to ensure the file was properly transmitted and reconstructed.

The typical algorithm for using the interface is:

1. Create a UR for a new or updated version of an object. This returns a new URL for the UR, which is simply the `/upload/key` path. The UR is temporary, and will exist for as long as the administrator has set the system to allow it to exist before being automatically removed.
2. Repeatedly upload segments of the BLOB as `application/octet stream` objects, providing an offset position to write the contents. The size of the blocks uploaded is up to you, but will be limited by the server. Usually, finding the largest size allowed yields the best performance, but this may not always be the case depending on many independent variables such as the server size, load, network reliability, number of hops, throughput, etc. A good rule of thumb for LANs is a 10MB maximum block size. You can usually query the server to determine the max size. This model makes it possible to create upload clients that can change these parameters as they learn the performance of the link, and even resume an interrupted upload later.
3. Get the UR state, forcing a checksum compute in the process.
4. If the computed checksum is correct, then finalize the upload by POSTing the UR's XML state with a suggested filename (the extension is important) and defined `title` element to the UR's URL.

Here is a summary of the entry points to this interface:

Resource	**Method**	**Meaning**
/upload	POST	Create UR for new object .
/upload?handle=*handle*	POST	Create UR for uploading a new version to object identified by *handle*.
/upload/*key?computechecksum=yes|no*	GET	Returns current UR state in XML. The sha1sum will only be computed if requested.
/upload/*key*/*offset*	PUT	Copy bytes to object's BLOB starting at *offset*
/upload/*key*	POST	Finalizes the upload and imports BLOB into repository. The UR is then deleted.
/upload/*key*	DELETE	Cancels upload (deletes the UR)

Here is an outline schema of the XML returned and consumed in this interface:

```
<upload>
  <key>key</key>
  <handle>handle</handle>
  <filename>filename</filename>
  <title>title</title>
  <initiated>timestamp</initiated>
  <lastactivity>timestamp</lastactivity>
  <size>size</size>
  <sha1sum>checksum</sha1sum>
</upload>
```

Values are all read-only unless otherwise specified below. The only time a writeable value should be specified is during the finalizing process, which is the only time a client should send the above XML document back to the server.

- **key –** Unique identifier for this upload, which is used to create the UR's URL as a suffixe, *e.g.*, /upload/*key.*
- **handle** – The object handle for which this upload is targeted. For new objects, the handle is not assigned until the upload is finalized.
- **Filename** – (writeable) The suggested name of the file. MUST be set in the finalizing POST operation to the UR.
- **title** – (writeable) This is the title to assign to this version of the object (to this BLOB). If empty or not provide at finalization, a default title will be derived from the filename.
- **initiated** – The time when the UR was created.
- **lastactivity** – The time when this UR was last accessed via the Upload Resource Interface. This value is typically used by automated system clean-up routines that clean up abandoned uploads.
- **size -** The current size of the UR's BLOB replica.
- **sha1sum** – The checksum of the current contents of the UR's BLOB replica. It is only computed upon request, since it can be time consuming to calculate, and cannot be computed reliably by watching PUTs. (The PUTs have to be idempotent, and sha1sum digest operations are not.)

POST /upload

Creates a new UR and returns a 303 SEE OTHER redirect to it. The new UR will have a unique key value, which is part of its URL in the suffix (/upload/*key*) to be used for subsequent operations. The handle will not be assigned until the upload is finalized, and the title will be empty.

RETURN STATUS	**RETURNED CONTENT**
303 SEE OTHER	Nothing. Client should use the redirect URL for future operations.
400 BAD REQUEST	If possible, an XML document rooted by an `error` element.
500 INTERNAL SERVER ERROR	If possible, an XML document rooted by an `error` element.

POST /upload?handle=*handle*

Creates a new UR and returns a 303 SEE OTHER redirect to it, provided there is an object with the handle specified. The new UR will have a unique key value, which is part of its URL in the suffix (/upload/*key*) to be used for subsequent operations. The upload will target the object specified by *handle,* and the title will be the title of the current version. (It can be changed when the upload is finalized.)

RETURN STATUS	**RETURNED CONTENT**
303 SEE OTHER	Nothing. Client should use the redirect URL for future operations.
400 BAD REQUEST	If possible, an XML document rooted by an `error` element.
404 NOT FOUND	If possible, an XML document rooted by an `error` element.
500 INTERNAL SERVER ERROR	If possible, an XML document rooted by an `error` element.

GET /upload/*key*?computechecksum=yes/no

Retrieves the XML document expressing the current state of the upload resource with key value *key*. If the `computechecksum` query parameter is provided it must be either `yes` or `no`. If not provided, the default is no. If the value of `computechecksum` is `yes`, then a sha1sum checksum is computed for the contents of the UR's BLOB replica and returned as part of the UR's state document. A checksum will also be returned when not specifically requested if the replica has not been modified since the last checksum was computed.

RETURN STATUS	**RETURNED CONTENT**
200 OK	XML document rooted by the `upload` element.
400 BAD REQUEST	If possible, an XML document rooted by an `error` element.
404 NOT FOUND	If possible, an XML document rooted by an `error` element.
500 INTERNAL SERVER ERROR	If possible, an XML document rooted by an `error` element.

PUT /upload/*key*/*offset*

Uploads binary data into the UR's BLOB replica starting at the specified *offset*. Initially, the replica is empty, and the first segment would be written at offset 0. However, there is no constraint except for an installation-dependent maximum filesize on the value of *offset*. If an *offset* is specified beyond the current end-of-file, then all bytes from the position immediately following the previous end-of-file location up to but not including *offset* will be 0, and the contents will be written at offset, expanding the file accordingly to a new end-of-file position.

The request payload should be of type "application/octet-stream" and the size will be determined by the client as part of creating the HTTP headers for the request.

RETURN STATUS	RETURNED CONTENT
200 OK	The current state of the UR (sans checksum) will be returned.
400 BAD REQUEST	If possible, an XML document rooted by an `error` element.
404 NOT FOUND	If possible, an XML document rooted by an `error` element.
500 INTERNAL SERVER ERROR	If possible, an XML document rooted by an `error` element.

POST /upload/*key*?duplicatecheck=yes|no

Finalizes the upload. The payload must be an XML documented rooted by an `upload` element. The element must contain a `filename` subordinate element. A `title` element is optional. If it is not provided, then the title from the previous version (if there is one) will be retained. Failing that, such as in the case of new object creation, a title will be derived from the filename. Any other content in the `upload` element will be ignored.

The `duplicatecheck` query parameter is used to control if the repository checks for a duplicate blob before performing the import. If `yes` (the default), then the operation will fail if there is already a blob in the repository with the same checksum and file length (together called a *fingerprint*).

Finalization proceeds as follows:

1. Ensures that a title has been set or retained.
2. Imports the BLOB into the repository, creating a version record for it, and either updating or creating a new object to own it.
3. Deletes the UR, returning an XML expression of it. The handle can be obtained from the XML.
4. Returns a 303 SEE OTHER status with a Location: header pointing to the new object's GET URL. The handle will be the last component in this path.

The returned XML can be examined to determine the object handle assigned, if the operation resulted in a new object. Failed finalize requests can be retried with different parameters. Failing does not invalidate the UR.

RETURN STATUS	RETURNED CONTENT
303 SEE OTHER	An XML document rooted by an `object` element.
400 BAD REQUEST	If possible, an XML document rooted by an `error` element.
404 NOT FOUND	If possible, an XML document rooted by an `error` element.
500 INTERNAL SERVER ERROR	If possible, an XML document rooted by an `error` element.

DELETE /upload/*key*

Deletes the UR, including its BLOB replica and contents. This effectively cancels the upload.

RETURN STATUS	**RETURNED CONTENT**
200 OK	`<success/>`
400 BAD REQUEST	If possible, an XML document rooted by an `error` element.
404 NOT FOUND	If possible, an XML document rooted by an `error` element.
500 INTERNAL SERVER ERROR	If possible, an XML document rooted by an `error` element.

Query Resource Interface

The Query Resource Interface provides a way to specify selection criteria (filters), sorting criteria, and projection criteria (attribute inclusion) in reporting information from the object meta-data and tagging information.

The interface is simple:

Resource	Method	Meaning
/query?select=*attributelist*&where=*filteexpression* &orderby=*attributelist*&includetags=yes|no	GET	Execute a query

It returns a `query` object:

```
<query
  select="attributelist"
  where="filterexpression"
  orderby="attributelist"
  includetags="yes|no">
  <objects>
    <object>
      <attributes>
        <attributename>value<attributename> ...
      </attributes>
      <tags>
        <tag name="name" value="value"/> ...
      <tags>
    </object>
  </objects>

</query>
```

The schema of the object (the attribute names and types) can be retrieved from `/object/schema`. Only attribute values from the current version are returned. The query parameters are all optional and have default values if unspecified:

- `attributelist` – A comma separated list of attribute names to be included in the result. The default is all attributes in the order they are listed in the schema.
- `filterexpression` – A textual filter expression used to select objects for inclusion in the result. The default is "TRUE" (select all objects). See below for the syntax of the expression.
- `orderby` – A comma separated list of attribute names by which to sort the result. The default is "`handle`".
- `includetags` – A flag indicating if tag assignments should be reported. The default is "`yes`".

Filter expressions

- Filter expressions must evaluate to a Boolean type.
- Tokens are not case sensitive.
- An *expression* is one of the following:
    - **A literal value.** These are constant values drawn from the six supported types:
        - String Literal: Strings can be delimited with single quotes are double quotes. An instance of the delimited quote character can be escaped in the string by a repeating sequence of two of them.
            - TimeStamp Literal: A string which can be parsed to get a TimeStamp value. The parser is very generous in its interpretation, so most reasonable formats can be parsed. There are two ways to write TimeStamp Literals:
                - String Literals which, when found in operational contexts adjacent to a known TimeStamp expression, such as an attribute of that type, are parsed to extract a TimeStamp value.
                - A TimeStamp Literal string can be explicitly marked as such by setting it inside of braces `{}` instead of double or single quotes like a String Literal.
        - Integer Literal: A sequence of characters that specify a decimal, octal, or hexadecimal encoded integer value. The actual underlying datatype is Java `Long`. Octal values start with a prefix of '0' and hexadecimal values start with '0x' or '0X'. An optional negative sign can precede the expression to indicate a negative value.
        - Double Literal: A floating point number, which is a sequence of decimal digits with exactly one embedded decimal point, or a decimal point at the beginning or the end.
        - Boolean Literal: The tokens TRUE or FALSE. Remember, they are not case sensitive.
        - Tag Literal: A tag is a set of strings. Tag literals are comma separated lists of String literals within a pair of brackets `[]`. For example, the tag value for Primary Colors could be denoted:

```
["red","yellow","blue"]
```

    - **An attribute name.** A non-delimited sequence of characters that constitute an attribute name as specified in the `/object/schema` resource. Attribute names are not tokens, and are case sensitive.

- **A binary operation.** All binary operations result in a Boolean type value. The operators are:

Name	**Syntax & Variations**	**Left Operand A**	**Right Operand B**	**Notes**
Equals	== =	Any type	Same type	True if A and B are the same value. For Tag types, this means they have the same set membership.
Not Equals	!= <>	Any type	Same type	True if A and B are different values. For Tag types, this means they differ in set membership.
Less Than	<	Any type except Tag	Same type	True if A precedes B in a natural ordering.
Less Than or Equal To	<=	Any type except Tag	Same type	True if A precedes B in a natural ordering, or is equal to B.
Greater Than	>	Any type except Tag	Same type	True if A follows B in a natural ordering.
Greater Than or Equal To	>=	Any type except tag	Same type	True if A follows B in a natural ordering, or is equal to B.
Is A Substring Of	$	String	String	True if A is a substring of B.
Case Insensitive String Operations	~== ~= ~!= ~<> ~< ~> ~<= ~>= ~$	String	String	Any comparison that can be performed with Strings can be made case insensitive by prepending the operator with a tilde (~).
Includes	== =	Tag	String	True if string B is in the Tag set A.
Does Not Include	!= <>	Tag	String	True if string B is not in the Tag set A.
Logical Conjunction	AND	Boolean	Boolean	True if both A and B are true.
Logical Disjunction	OR	Boolean	Boolean	True if either A, or B, or both A and B are true.

- **A unary operation.** There is only one unary operation, Logical Negation, represented by the token NOT. It is followed by a single Boolean operand, and returns the opposite of the operand.
- **A forced precedence.** Any expression can be set inside parentheses to guarantee a specific order of evaluation within a larger expression string.

- **EXAMPLES:**
    - `title=”The Doctor Dances”`
    - `title~=”the doctor dances”`
    - `imported>”Tuesday, May 1”`
    - `hm_lastseen<{Jan 2009}`
    - `Genre=”Science Fiction”`
    - `Genre=[“Science Fiction”,”Drama”]`
    - `Genre=”Science Fiction” and title$”doctor”`