APPENDIX G - All About Tuque

Please check the original GitHub Working With Fedora Objects Programmatically Via Tuque reference for updates on these notes.:

https://github.com/Islandora/islandora/wiki/Working-With-Fedora-Objects-Programmatically-Via-Tuque

Islandora introduces support for a Fedora repository to be connected to and manipulated using the Tuque PHP library. This library can be accessed using functions included with Islandora, available inside a properly-bootstrapped Drupal environment. It can also be accessed directly outside of an Islandora environment.

Tuque is an API, written and accessible via PHP, that connects with a Fedora repository and mirrors its functionality. Tuque can be used to work with objects inside a Fedora repository, accessing their properties, manipulating them, and working with datastreams.

This guide will highlight methods of working with Fedora and Fedora objects using Tuque both by itself and from a Drupal environment.

Tuque download

milestone 4 - Installing The "Tuque" library

Variables repeated often in this guide

From here on out, we're going to be repeating the use of a few specific PHP variables after the guide demonstrates how they are instantiated or constructed:

Variable	PHP Class	Description
`$repository`	`FedoraRepository`	A PHP object representation of the Fedora repository itself.
`$object`	`FedoraObject`	A generic Fedora object.
`$datastream`	`FedoraDatastream`	A generic Fedora object datastream.

Accessing the Fedora Repository

Connecting to Fedora

Tuque or Islandora

```
$connection = new RepositoryConnection($fedora_url, $username, $password)
```

Islandora Only (via module)

```
$connection = islandora_get_tuque_connection($user)
```

Accessing the repository

Tuque or Islandora

```
 /**
 * Assuming our $connection has been instantiated as a new RepositoryConnection object.
 */
$api = new FedoraApi($connection);
$repository = new FedoraRepository($api, new simpleCache(););
```

Islandora only, manually, using the Islandora Tuque wrapper:

```
 /**
 * Assuming our $connection has been instantiated as a new RepositoryConnection object.
 */
module_load_include('inc', 'islandora', 'includes/tuque');
module_load_include('inc', 'islandora', 'includes/tuque_wrapper');
$api = new IslandoraFedoraApi($connection);
$repository = new IslandoraFedoraRepository($api, new SimpleCache());
```

Islandora only, automatically, using the Islandora module:

```
 /**
 * Assuming $connection has been created via islandora_get_tuque_connection().
 */
$repository = $connection->repository;
```

Islandora only, using the IslandoraFedoraObject wrapper:

```
 /**
 * This method tends to be the most reliable when working with a single object,
 * since it builds on the success of the attempt to load that object.
 */
$pid = 'object:pid';
$object = islandora_object_load($pid);
if ($object) {
  $repository = $object->repository;
}
```

From here, all Fedora repository functionality supported by Tuque is available to you through `$repository`. This functionality is described in the rest of this document.

As of Islandora 7.x, there is a wrapper object, `IslandoraFedoraObject`, that handles some errors and fires some hooks in includes/tuque.inc. More error handling is available if one uses the wrapper functions in islandora.module.

Working with existing objects

Loading an object

Method	Code	On Success	On Fail
Tuque or Islandora, from a`FedoraRepository`	`$object = $this->repository->getObject($pid);`	Returns a`FedoraObject` loaded from the given `$pid`.	Throws a 'Not Found'`RepositoryException`.
Islandora only, from an`IslandoraFedoraRepository`	`$object = $this->repository->getObject($pid);`	Returns an`IslandoraFedoraObject`loaded from the given`$pid`.	Throws a 'Not Found'`RepositoryException`.
Islandora only, using the module itself	`$object = islandora_object_load($pid);`	Returns an`IslandoraFedoraObject`loaded from the given`$pid`.	Returns `FALSE`

Because the third method returns FALSE on failure, you can check if the object loaded correctly using `!$object`, e.g.:

```
$object = islandora_object_load($pid);
if (!$object) {
  /**
 * Logic for object load failure would go here.
 */
  return;
}
/**
 * Logic for object load success would continue through the rest of the method here.
 */
```

In the case of the other two methods, `try` to load the object and `catch` the load failure exception, e.g.:

```
try {
  $object = $this->repository->getObject($pid);
}
catch (Exception $e) {
  /**
 * Logic for object load failure would go here.
 */
}
/**
 * Logic for object load success would continue through the rest of the method here.
 */
```

Objects loaded via Tuque (either through Islandora or directly) have the following properties and can be manipulated using the following methods:

Properties

Name	Type	Description
`createdDate`	`FedoraDate`	The object's date of creation.
`forceUpdate`	`bool`	Whether or not Tuque should respect Fedora object locking on this object (`FALSE` to uphold locking). Defaults to `FALSE`.
`id`	`string`	The PID of the object. When constructing a new object, this can also be set to a namespace instead, to simply use the next available ID for that namespace.
`label`	`string`	The object's label.
`lastModifiedDate`	`FedoraDate`	When the object was last modified.
`logMessage`	`string`	The log message associated with the creation of the object in Fedora.
`models`	`array`	An array of content model PIDs (e.g. 'islandora:collectionCModel') applied to the object.
`owner`	`string`	The object's owner.
`relationships`	`FedoraRelsExt`	A `FedoraRelsExt` object allowing for working with the object's relationship metadata. This is described in another section below.
`repository`	`FedoraRepository`	The `FedoraRepository` object this particular object was loaded from. This functions precisely the same as the `$repository`created in the "Accessing the repository" section above.
`state`	`string`	The object's state (A/I/D).

Methods

Name	Description	Parameters	Return Value
`constructDatastream($id, $control_group)`	Constructs an empty datastream. Note that this does not ingest a datastream into the object, but merely instantiates one as an`AbstractDatastream`object. Ingesting is done via`ingestDatastream()`, described below.	`$id` - the PID of the object; `$control_group`- the Fedora control group the datastream will belong to, whether Inline (X)ML, (M)anaged Content, (R)edirect, or (E)xternal Referenced. Defaults to 'M'.	An empty`AbstractDatastream`object from the given information.

`count()`	The number of datastreams this object contains.	None	The number of datastreams, as an`int`.
`delete()`	Sets the object's state to 'D' (deleted).	None	None
`getDatastream($dsid)`	Gets a datastream from the object based on its DSID. `$object->getDatastream($dsid)`works effectively the same as`$object[$dsid]`.	`$dsid` - the datastream identifier for the datastream to be loaded.	An`AbstractDatastream` objeect representing the datastream that was gotten, or`FALSE` on failure.
`getParents()`	Gets the IDs of the object's parents using its`isMemberOfCollection`and `isMemberOf`relationships.	None	An array of PIDs of parent objects.
`ingestDatastream(&$abstract_datastream)`	Takes a constructed datastream, with the properties you've given it, and ingests it into the object. This should be the last thing you do when creating a new datastream.	Technically takes`$abstract_datastream`as a parameter, but this should be passed to it by reference after constructing a datastream with`constructDatastream()`.	A `FedoraDatastream`object representing the object that was just ingested.
`purgeDatastream($dsid)`	Purges the datastream identified by the given DSID.	`$dsid` - The datastream identifier of the object.	`TRUE` on success,`FALSE` on failure.
`refresh()`	Clears the object cache so that fresh information can be requested from Fedora.	None	None

Purging an object

A loaded object can be purged from the repository using:

```
$repository->purgeObject($object);
```

Working with datastreams

Datastreams can be accessed from a loaded object like so:

Tuque or Islandora

```
$datastream = $object['DSID'];
```

Islandora Only

```
$datastream = islandora_datastream_load($dsid, $object);
```

where `$dsid` is the datastream identifier as a `string`, and `$object` is either an object PID or a loaded Fedora object.

This loads the datastream as a `FedoraDatastream` object. From there, it can be manipulated using the following properties and methods:where `$dsid` is the datastream identifier as a `string`, and `$object` is a loaded Fedora object.

Properties

Name	Type	Description
`checksum`	`string`	The datastream's base64-encoded checksum.
`checksumType`	`string`	The type of checksum for this datastream, either DISABLED, MD5, SHA-1, SHA-256, SHA-384, SHA-512. Defaults to DISABLED.
`content`	`string`	The binary content of the datastream, as a string. Can be used to set the content directly if it is an (I)nternal or (M)anaged datastream.
`controlGroup`	`string`	The control group for this datastream , whether Inline (X)ML, (M)anaged Content, (R)edirect, or (E)xternal Referenced..

createdDate	FedoraDate	The date the datastream was created.
forceUpdate	bool	Whether or not Tuque should respect Fedora object locking on this datastream (FALSE to uphold locking). Defaults toFALSE.
format	string	The format URI of the datastream, if it has one. This is rarely used, but does apply to RELS-EXT.
id	string	The datastream identifier.
label	string	The datastream label.
location	string	A combination of the object ID, the DSID, and the DSID version ID.
logMessage	string	The log message associated with actions in the Fedora audit datastream.
mimetype	string	The datastream's mimetype.
parent	AbstractFedoraObject	The object that the datastream was loaded from.
relationships	FedoraRelsInt	The relationships that datastream holds internally within the object.
repository	FedoraRepository	The FedoraRepository object this particular datastream was loaded from. This functions precisely the same as the$repository created in the "Accessing the repository" section above.
size	int	The size of the datastream, in bytes. This is only available to ingested datastreams, not ones that have been constructed as objects but are yet to be ingested.
state	string	The state of the datastream (A/I/D).
url	string	The URL of the datastream, if it is a (R)edirected or (E)xternally-referrenced datastream.
versionable	bool	Whether or not the datastream is versionable.

Methods

Name	Description	Parameters	Return Value
count()	The number of revisions in the datastream's history.	None	An intrepresenting the number of revisions in the datastream history.
getContent()	Returns the binary content of the datastream.	None	A stringrepresenting the contents of the datastream.
refresh()	Clears the object cache so that fresh information can be requested from Fedora.	None	None
setContentFromFile($path)	Sets the content of a datastream from the contents of a local file.	$path - the path to the file to be used.	None
setContentFromString($string)	Sets the content of a datastream from a string.	$string - the string to set the content from.	None
setContentFromUrl($url)	Attempts to set the content of a datastream from content downloaded using a standatd HTTP request (NOT HTTPS).	$url - the URL to grab the data from.	None

Iterating over all of an object's datastreams

Since they exist on an object as an array, datastreams can be iterated over using standard array iteration methods, e.g.:

```
foreach ($object as $datastream) {
  strtoupper($datastream->id);
  $datastream->label = "new label";
  $datastream_content = $datastream->getContent();
}
```

Example of creating or updating a datastream

```
$dsid = 'DSID';
// Before we do anything, check if the datastream exists. If it does, load it; otherwise construct it.
// The easiest way to do this, as opposed to a string of cases or if/then/elses, is the ternary operator, e.g.
// $variable = isThisThingTrueOrFalse($thing) ? setToThisIfTrue() : setToThisIfFalse();
$datastream = isset($object[$dsid]) ? $object[$dsid] : $object->constructDatastream($dsid);
$datastream->label = 'Datastream Label';
$datastream->mimeType = 'datastream/mimetype';
$datastream->setContentFromFile('path/to/file');
// There's no harm in doing this if the datastream is already ingested or if the object is only constructed.
$object->ingestDatastream($datastream);
// If the object IS only constructed, ingesting it here also ingests the datastream.
$repository->ingestObject($object);
```

Creating new objects and datastreams

When using Tuque, Fedora objects and datastreams must first be constructed as PHP objects before being ingested into Fedora. Un-ingested, PHP-constructed Fedora objects and datastreams function nearly identically to their ingested counterparts, as far as Tuque is concerned, with only a few exceptions noted in the properties and methods tables below.

Constructing and ingesting an object

Constructing and ingesting an object

```
 $object = $repository->constructObject($pid); // $pid may also be a namespace.
/**
 * Here, you can manipulate the constructed object using the properties and methods described above.
 */
$repository->ingestObject($object);
```

Constructing and ingesting a datastream

```
 $datastream = $object->constructDatastream($dsid) // You may also set the $control_group.
/**
 * Here, you can manipulate the constructed datastream using the properties and methods described above.
 */
$object->ingestDatastream($dsid, $object);
```

Accessing an object's relationships

Once an object is loaded, its relationships can be accessed via the object's `relationships` property:

```
$relationships = $object->relationships;
```

From there, the object's relationships can be viewed and manipulated using the following properties and methods:

Properties

Name	Type	Description
`autoCommit`	`bool`	Whether or not changes to the RELS should be automatically committed. **WARNING**: Probably don't touch this if you're not absolutely sure what you're doing.
`datastream`	`AbstractFedoraDatastream`	The datastream that this relationship is manipulating, if any.

Methods

Name	Description	Parameters	Return Value
`add ($predicate_uri, $predicate, $object, $type)`	Adds a relationship to the object.	`$predicate_uri` - the namespace of the relationship predicate (if this is to be added via XML, use the `registerNamespace()`function described below first);`$predicate` - the predicate tag to be added; `$object` - the object to add the relationship to (not required if this is called using`$object->relationships->add()`); `$type` - the type of the attribute to add (defaults to`RELS_TYPE_URI`).	None
`changeObjectID($id)`	Changes the ID referenced in the`rdf:about`attribute.	`$id` - the new ID to use.	None
`commitRelationships ($set_auto_commit)`	Forces the committal of any relationships cached while the `autoCommit`property was set to`FALSE` (or for whatever other reason).	`$set_auto_commit` - determines the state of `autoCommit` after this method is run (defaults to `TRUE`).	None
`get ($predicate_uri, $predicate, $object, $type)`	Queries an object's relationships based on the parameters given. See below for an example of filtering relationships using parameters.	`$predicate_uri` - the URI to use as the namespace predicate, or `NULL`for any predicate (defaults to `NULL`);`$predicate` - the predicate tag to filter by, or 'NULL' for any tag (defaults to`NULL`); `$object` - the object to filter the relationship by (not required if this is called using`$object->relationships->get()`); `$type` - what type`RELS_TYPE_XXX`attribute the retrieved should be (defaults to`RELS_TYPE_URI`).	The relationships as an`array`. See the note below for an example.
`registerNamespace ($alias, $uri)`	Registers a namespace to be used by predicate URIs.	`$alias` - the namespace alias;`$uri` - the URI to associate with that alias.	None
`remove ($predicate_uri, $predicate, $object, $type)`	Removes a relationship from the object.	`$predicate_uri` - the namespace of the relationship predicate to be removed, or `NULL` to ignore (defaults to`NULL`); `$predicate` - the predicate tag to filter removed results by. or `NULL` to remove all (defaults to `NULL`); `$object` - the object to add the relationship to (not required if this is called using`$object->relationships->remove()`); `$type` - what type`RELS_TYPE_XXX`attribute the removed should be (defaults to`RELS_TYPE_URI`).	None

Example of retrieving a filtered relationship

```
$object_content_models = $object->relationships->get('info:fedora/fedora-system:def/model#', 'hasModel');
```

This would return an array containing only the object's `hasModel` relationships.

Example of setting an isMemberOfCollection relationship

Islandora provides the constant `FEDORA_RELS_EXT_URI` to make it easy to set the predicate as the first variable here:

Example of a retrieved relationship array

```
Array
(
    [0] => Array
        (
            [predicate] => Array
                (
                    [value] => isMemberOfCollection
                    [alias] => fedora
                    [namespace] => info:fedora/fedora-system:def/relations-external#
                )
            [object] => Array
                (
                    [literal] => FALSE
                    [value] => islandora:sp_basic_image_collection
                )
        )

    [1] => Array
        (
            [predicate] => Array
                (
                    [value] => hasModel
                    [alias] => fedora-model
                    [namespace] => info:fedora/fedora-system:def/model#
                )
            [object] => Array
                (
                    [literal] => FALSE
                    [value] => islandora:sp_basic_image
                )
        )
)
```

Using the Fedora A and M APIs

Tuque can work with the Fedora repository's "Access" and "Manage" API services in much the same way one would using standard Fedora API requests. This functionality is mimicked using an instantiated `$repository`'s `api` property.

Note that the methods above provide a much more PHP-friendly way of performing many of the tasks provided by API-A and API-M. They are nonetheless listed in full below for documentation purposes. When a method in this section and a method above share functionality, it is **always** recommended to use the method above, as not only is it nearly guaranteed to be easier to work with, but also we cannot predict the nature of the Fedora APIs in the future; if any Fedora functionality changes or is removed, your code may also lose functionality. For example:

```
/**
 * Adding a relationship to an object. The API method is clunky and requires information you wouldn't
 * need if you did things the tuque way, which is more Drupal-friendly as well.
 */
// API method.
$repository->api->m->addRelationship();
// Tuque method.
$object->relationships->add();

/**
 * Iterating through datastreams. The API method only gives you an associative array of DSIDs
 * containing the label and mimetype - you would have to load each datastream if you wanted to
 * work with it. Working through tuque is faster.
 */
// API method.
$array = $repository->api->a->listDatastreams($object->id);
foreach ($array as $dsid => $properties) {
 $datastream = islandora_datastream_load($dsid, $object);
 // Now you can do stuff with the datastream.
}
// Tuque method.
foreach ($object as $datastream) {
 // Do stuff with the datastream.
}
```

Documentation for the most current versions of each API can be found at:

- https://wiki.duraspace.org/display/FEDORA37/API-A
- https://wiki.duraspace.org/display/FEDORA37/API-M
- https://wiki.duraspace.org/display/FEDORA38/API-A
- https://wiki.duraspace.org/display/FEDORA38/API-M

Each API exists as a PHP object through Tuque, and can be created using:

```
$api_a = $repository->api->a; // For an Access API.
$api_m = $repository->api->m; // For a Management API.
```

From here, the functionality provided by each API mimics the functionality provided by the actual Fedora APIs, where the standard Fedora endpoints can be called as API object methods, e.g.:

```
$datastreams = $api_a->listDatastreams('islandora:1');
```

The following methods are available for each type of API:

FedoraApiA

All of these return results described in an array.

Method	Description
`describeRepository()`	Returns repository information.
`findObjects($type, $query, $max_results, $display_fields)`	Finds objects based on the input parameters.
`getDatastreamDissemination($pid, $dsid, $as_of_date_time, $file)`	Gets the content of a datastream.
`getDissemination($pid, $sdef_pid, $method, $method_parameters)`	Gets a dissemination based on the provided method.

`getObjectHistory($pid)`	Gets the history of the specified object.
`getObjectProfile($pid, $as_of_date_time)`	Gets the Fedora profile of an object.
`listDatastreams($pid, $as_of_date_time)`	Lists an object's datastreams.
`listMethods($pid, $sdef_pid, $as_of_date_time)`	Lists the methods that an object can use for dissemination.
`resumeFindObjects($session_token)`	Resumes a `findObjects()` call that returned a resumption token.
`userAttributes()`	Authenticates and provides information about a user's Fedora attributes.

FedoraApiM

All of these return results described in an array.

Method	Description
`addDatastream($pid, $dsid, $type, $file, $params)`	Adds a datastream to the object specified.
`addRelationship($pid, $relationship, $is_literal, $datatype)`	Adds a relationship to the object specified.
`export($pid, $params)`	Exports information about an object.
`getDatastream($pid, $dsid, $params)`	Returns information about the specified datastream.
`getDatastreamHistory($pid, $dsid)`	Returns the datastream's history information.
`getNextPid($namespace, $numpids)`	Gets a new, unused PID, incrementing Fedora's PID counter for that namespace.
`getObjectXml($pid)`	Returns the object's FOXML.
`getRelationships($pid, $relationship)`	Returns the object's relationships.
`ingest($params)`	Ingests an object.
`modifyDatastream($pid, $dsid, $params)`	Makes specified modifications to an object's datastream.
`modifyObject($pid, $params)`	Makes specified modifications to an object.
`purgeDatastream($pid, $dsid, $params)`	Purges the specified datastream.
`purgeObject($pid, $log_message)`	Purges the specified object.
`upload($file)`	Uploads a file to the server.
`validate($pid, $as_of_date_time)`	Validates an object.

Using the Resource Index

The resource index can be queried from the repository using:

```
$ri = $repository->ri;
```

From there, queries can be made to the resource index. It is generally best to use SPARQL queries for forwards compatibility:

```
$itql_query_results = $ri->itqlQuery($query, $limit); // For an iTQL query.
$sparql_query_results = $ri->sparqlQuery($query, $limit); // For a SPARQL query.
```

Methods

Method	Description	Parameters	Return Value
`itqlQuery ($query, $limit)`	Executes an iTQL query to the resource index.	`$query` - a `string`containing the query parameters; `$limit` - an `int` representing the number of hits to return (defaults to -1 for unlimited).	An array containing query results.
`sparqlQuery ($query, $limit)`	Executes a SparQL query to the resource index.	`$query` - a `string`containing the query parameters; `$limit` - an `int` representing the number of hits to return (defaults to -1 for unlimited).	An array containing query results.