

Customized Namespace PIDs in Fez and Fedora

Keith E. Maull

Colorado Alliance of Research Libraries
3801 E. Florida Ave, Suite 515
Denver, CO 80210
keith@coalliance.org

January 2008

Abstract

Fedora provides flexible support for customized PID namespaces. While Fez does not provide *native* support for Fedora's PID namespace, this paper will describe the motivation and implementation for such support in Fez and subsequent impact on Fedora.

1 Motivation

Fedora allows for the use of multiple namespaces within a single repository. Within our consortial organization, such customization is important because it is much easier to manage and maintain the repository where each consortial member's assets are assigned from a member-specific namespace. So, for example, member *A* will be assigned PIDs namespace *A*, *B* for namespace *B*, and so on. Such support may facilitate searches over specific member's assets by PID alone, but more importantly this scheme allows for an implementation of a PID-based namespace algorithm at the file system level. This simple mechanism for storing files within PID "buckets" is shown in the diagram below. The Fedora implementation for such an implementation will be discussed in section ??.

%% Figure

Fedora's support for Persistent Identifier namespaces [?] allows for PIDs to be generated from a pool of user specified namespaces that go beyond the default namespace specified in the `fedora.fcfg pidNamespace` property. It is important to remember that any custom namespace you wish to use outside of the default **must** be included in the `retainPIDs` parameter.

Listing 1: Sample `fedora.fcfg` parameters for custom PIDs

```
1 <param name="pidNamespace" value="A"/>
2 <param name="retainPIDs" value="A B C fedora-bdef fedora-bmech tutorial"/>
```

2 Fez Support and Implementation

Fez allows for a high degree of customization at the source code level []. Implementing support for alternate PID namespaces is a straightforward implementation and will be discussed in this section.

2.1 Updating the *FezMD* datastream

Our implementation, first requires that *Fez communities* and *collections* retain information about the namespace. These are required so that when a collection is added to a community, the namespace is inherited from the parent community. Furthermore, when an object is added to a collection, the object PID will inherit from the collection. This is the most simple implementation, but other variants can be considered.

To pass along the namespace information from parent to child, we must store the namespace somewhere. The decision was made to store it in the *FezMD* datastream. This was done for two reasons : (1) The *FezMD* datastream already contains interesting administrative information relevant only to Fez, and (2) it did not require changing any other extant datastream like *DC* for which adding such information may not make sense.

Updating the XSD Display within the Fez system simply requires the *FezMD Community Display* and *FezMD Collection Display* be updated with this administrative element. Here at our organization, we chose the element `adr_id` to correspond to the Alliance namespace for the given community or collection.

2.2 Obtaining the Namespace

Once this administrative information is added to the XSD Displays, Fez will need some way of getting at it. For the sake of simplicity, we introduce a new class, `include/class.util.php`, which provides a single method `getNamespace` that takes a single *pid* argument.

Listing 2: The `getNamespace` method

```
1 function getNamespace( $pid )
2 {
3     $xdis_array = Fedora_API::callGetDatastreamContents($pid, 'FezMD');
4     $namespace = $xdis_array['adr_id'][0];
5     return $namespace;
6 }
```

Notice that we obtain our new XSD element `adr_id` in line 3.

2.3 Generating a PID in the New Namespace

To generate an object in the new namespace the class `class.record.php` must be updated. Two methods are changed to take the *namespace* parameter. `Record::insert` and `Record::fedoraInsertUpdate`.

Listing 3: The `Record::insert` method

```
1 function insert($namespace)
2 {
3     $record = new RecordObject();
4     return $record->fedoraInsertUpdate(null, null, null, $namespace);
5 }
```

Notice that `fedoraInsertUpdate` takes a fourth optional namespace parameter.

Listing 4: The `Record::fedoraInsertUpdate` method

```
1 function fedoraInsertUpdate(
2     $exclude_list=array(), $specify_list=array(),
```

```

3     $params = array(), $namespace = "")
4 {
5     ...
6     if (empty($this->pid)) {
7         $this->pid = Fedora_API::getNextPID($namespace);
8         $ingestObject = true;
9     }
10 }

```

To generate an object such that Fedora returns the appropriate PID within the namespace we specify, a final change must be made to the `class.fedora_api_2_2.php` class. If you are using Fedora 2.1.1 you will need to update the corresponding `class.fedora_api_2_1_1.php` as well.

The change requires that the empty argument `getNextPid` method take a single optional argument *namespace*. The Fedora API-M-Lite call to get the next PID from the repository will also be updated to reflect this new argument. Using the call `{fedora/management/getNextPID&namespace=[namespace]}` will return the next available PID from the supplied namespace. The new `getNextPid` is shown below.

Listing 5: The `getNextPID` method

```

1 function getNextPID( $namespace = " " ) {
2     $pid = false;
3     $getString = APP_BASE_FEDORA_APIM_DOMAIN .
4         "/management/getNextPID?xml=true&namespace=" . $namespace;
5     ...

```

When the argument to `getNextPID` is not specified, the generated PID will be from the default namespace, and the default behavior of all existing code is preserved.

Finally, to assure that the namespace parameter is being transmitted through workflows properly, we add an additional update to the `class.lister.php` class, that stores the namespace in the template variable `namespace`.

Listing 6: The `Lister::getList` method

```

171     } else {
172         $browse_mode = "list";
173     }
174     if ( $pid ) {
175         $namespace = Util::getNamespace($pid);
176         $tpl->assign("namespace", $namespace);
177     }

```

We're now done with the major code updates to get namespaces to work. The next steps described in the successive sections review the template changes necessary to make the workflow screens properly update.

2.4 Updating Core Workflows

Now that objects from alternate namespaces can be obtained and we have a way of obtaining the existing namespace from a collection or community, the core workflows required will require small updates to allow the *namespace* parameter to be passed down to the workflows that matter : edit and update.

Making the requisite updates for this functionality is relatively simple – we simply add the parameter to the appropriate template files : `enter_metadata.tpl.html`, `edit_metadata.tpl.html`. To save space, each updated file will not be listed separately, rather the single change necessary in all of them is given :

Listing 7: The namespace template parameter

```
<input type="hidden" name="namespace" value="{ $namespace }">
```

That's it! The final step is to add update namespace workflow. The implementation is rather simple but requires a new form `enter_namespace.tpl.html` and new workflow logic in the corresponding `enter_namespace.php`. The core of that implementation rolls up into the workflow state change. Upon moving from the entry of the namespace to the next workflow, we merely capture the hidden `namespace` parameter in the form and store it in the `wfstatus` object. Note that the workflow implies that a new community is being created and that the namespace cannot be updated or changed. Hence, we need only make a change to the `enter_metadata.php` file. The changes in the edit workflow, once an object has already been created, will be discussed next.

Listing 8: The POST logic in `enter_metadata.php`

```
// check for post action
if (@$_POST["cat"] == "report") {
    // we will need this for the next workflow state
    $wfstatus->namespace = @$_POST["namespace"];
}
```

The `enter_metadata.php` file nows needs only grab the `wfstatus->namespace` variable and store it in the hidden namespace variable on it's form. This is done by :

Listing 9: Setting the namespace variable method

```
$tpl->assign("namespace", $wfstatus->namespace);
$namespace = $wfstatus->namespace;
```

When the form is submitted, we must send the namespace to our new `insert` method :

Listing 10: Capturing the form POST from `enter_metadata.php`

```
if (@$_POST["cat"] == "report") {
    $res = Record::insert();
    $res = Record::insert($namespace);
    ...
}
```

The Edit workflow follows a similar pattern, except we will get the namespace explicitly :

Listing 11: Assigning the `namespace` form variable within the Edit workflow

```
$namespace = Util::getNamespace($pid);
$tpl->assign("namespace", $namespace);
```

3 Fedora Pid Namespace Implementation

Now that we have Fez creating objects within Fedora with alternate namespaces, the specific needs of the Alliance can be met : namely that the disk space allocated for each member be easily

managed at the *physical* level. That is to say, we wish every member to have their own storage “bucket”, that can then be managed more precisely by allocating appropriate hardware to each community as necessary. Each of these is directly mapped by the PIDs we’re now able to specify within Fez.

%picture goes here%

To override Fedora’s default treatment of object storage, the `path_algorithm` will need to be changed. For the Alliance, a simple modification to allow for the *PID* to specify the name of the root of the physical storage was necessary. On the filesystem, it was important to have a directory mapping such that each member in the Alliance had (1) their own namespace identifier and (2) their own directory which mapped to the physical storage they have been allocated.

Listing 12: Setting the `path_algorithm`

```
<param name="path\_algorithm"
      value="fedora.server.storage.lowlevel.PidTimestampPathAlgorithm">
```

The implementation of the algorithm requires a straightforward modification that also requires a subclass of the `PathAlgorithm`. Since the *PID* is passed in as a parameter to the superclass method `format`, we instantly know the namespace of the community. A further refinement makes use of the `_` separator between the pid and the rest of the name (it should be noted that namespaces with addition `_` are *not* allowed, but such a restriction could likely be overcome by modifying `fedora.server.management.PIDGenerator`). The crux of the basic implementation follows.

Listing 13: The Java implementation for PID based storage

```
public String format (String pid) throws LowlevelStorageException {
    GregorianCalendar calendar = new GregorianCalendar();
    ...
    String minute = leftPadded(calendar.get(Calendar.MINUTE),2);
    String pidBase = pid.substring(0, pid.indexOf('_')) ;

    return storeBase + SEP + pidBase + SEP + year +
           SEP + month + dayOfMonth + SEP + hourOfDay +
           SEP + minute /*+ sep + second*/ + SEP + pid;
}
```

After compiling Fedora, installing the webapp and restarting the server, a basic refinement to the standard `PidTimestampPathAlgorithm` is implemented. All new objects will now be placed in the proper sub-directory based on the PID. For example, if the PID for an object is `codu`, the low-level storage will look like `[datastream_store_base]/codu/2008/01/...` and `[object_store_base]/codu/2008/01/...` Now `[object_store_base]/codu` and `[datastream_store_base]/codu` can actually be managed at the disk-level much differently from objects with another PID, satisfying disk-level management concerns.

3.1 Files Affected

class.util.php	New utility file
class.fedora_api_2.2.php	updated <code>getNextPID</code>
class.lister.php	updated <code>getList</code>
class.record.php	updated <code>insert</code> and <code>fedoraInsertUpdate</code>

Table 1: Core class files added, changed or updated

4 Conclusion

Implementing a repository with multiple namespaces may be important for many different reasons. In this paper, we show implementations in Fez and Fedora that serve the specific goals of (1) managing disk-level allocations based on community and (2) implementing a low-level storage mechanism for tracking assets by community via Fedora namespaces. There are many variants on this theme, but it is hoped that this demonstrates the practical and conceptual concerns for such an implementation.