# APPENDIX F: METAARCHIVE MICROSERVICES

# MetaArchive Microservices

NHPRC, MetaArchive Project

2012-03-09


**Summary**

The purpose of this document is to explain the PREMIS Event microservice as prototyped for the MetaArchive Cooperative, primarily from a developer's standpoint. The document will attempt to explain the purpose of the microservice, the underlying principles on which it operates, and a practical example of its usage. The MetaArchive has installed all of the components needed to deploy this microservice and looks forward to opportunities to experiment with practical implementations in future development projects.


**Purpose**

The purpose of this microservice is to provide a straightforward way to send PREMIS-formatted events to a central location to be stored and retrieved. In this fashion, it can serve as an event-logger for any number of services that happen to wish to use it. PREMIS is chosen as the underlying format for events, due to its widespread use in the digital libraries world.

**PREMIS Events**

A standard PREMIS event looks something like the following (vocabularies and event/agent types, listed are merely for the sake of this example and not yet developed):

```
<premis:event xmlns:premis="info:lc/xmlns/premis-v2">
        <premis:eventType>
                http://purl.org/net/meta/vocabularies/preservationEvents/#MigrateSuccess
        </premis:eventType>
        <premis:linkingAgentIdentifier>
                <premis:linkingAgentIdentifierValue>
                        http://metaarchive.org/agent/metaMigrateSuccess
                </premis:linkingAgentIdentifierValue>
                <premis:linkingAgentIdentifierType>
                        http://purl.org/net/meta/vocabularies/identifier-qualifiers/#URL
                </premis:linkingAgentIdentifierType>
        </premis:linkingAgentIdentifier>
        <premis:eventIdentifier>
                <premis:eventIdentifierType>
                        http://purl.org/net/meta/vocabularies/identifier-qualifiers/#UUID
                </premis:eventIdentifierType>
                <premis:eventIdentifierValue>
                        e8ee3b1a8c9e4a5daf0a1e0446383d90
                </premis:eventIdentifierValue>
```

```
        </premis:eventIdentifier>
        <premis:eventDetail>
                Verification of data at /data3/meta-r1-003_dropbox/meta106w
        </premis:eventDetail>
        <premis:eventOutcomeInformation>
                <premis:eventOutcomeDetail>
                        Checking content after cache server migration
                </premis:eventOutcomeDetail>
                <premis:eventOutcome>
                        http://purl.org/net/meta/vocabularies/eventOutcomes/#success
                </premis:eventOutcome>
        </premis:eventOutcomeInformation>
        <premis:eventDateTime>
                2011-01-25 16:39:49
        </premis:eventDateTime>
        <premis:linkingObjectIdentifier>
                <premis:linkingObjectIdentifierType>
                        http://purl.org/net/meta/vocabularies/identifier-qualifiers/#ARK
                </premis:linkingObjectIdentifierType>
                <premis:linkingObjectIdentifierValue>
                        ark:/67531/meta106w
                </premis:linkingObjectIdentifierValue>
        <premis:linkingObjectRole/>
        </premis:linkingObjectIdentifier>
</premis:event>
```

This is a lot at first glance, but the pieces are more or less logical. The relevant things that a given PREMIS event record keeps track of are the following:

- Event Identifier - This is a unique identifier assigned to every event when it is entered into the system. This is what is used to reference given event.
- Event Type - This is an arbitrary value to categorize the kind of event we're logging. Examples might include fixity checking, virus scanning or replication.
- Event Time - This is a timestamp for when the event itself occurred.
- Event Added - This is a timestamp for when the event was logged.
- Event Outcome - This is the simple description of the outcome. Usually something like "pass" or "fail".
- Outcome Details - A more detailed record of the outcome. Perhaps output from a secondary program might go here.
- Agent - This is the identifier for the agent that initiated the event. An agent can be anything, from a person, to an institution, to a program. The PREMIS event microservice will also allow you to track agent entries as well.
- Linked Objects - These are identifiers for relevant objects that the event is associated with. If your system uses object identifiers, you could put those identifiers here when an event pertains to them.

It is important to note that most of the values that you use in a given PREMIS event record are arbitrary. You decide on your own values and vocabularies, and use what makes sense to you. It doesn't enforce any sort of constraints as far as that goes.

The microservice is responsible for indexing all PREMIS events sent to it and providing retrieval for them. Basic retrieval is on a per-identifier basis, but it is plausible to assume that you may wish to request events based on date added, agent used, event type, event outcome, or a combination of these factors.

**PREMIS Agents**
The PREMIS metadata specification defines a separate spec for agents that looks like the following:

```
<?xml version="1.0"?>
<premis:agent xmlns:premis="info:lc/xmlns/premis-v2">
        <premis:agentIdentifier>
                <premis:agentIdentifierValue>
                        MigrateSuccess
                </premis:agentIdentifierValue>
                <premis:agentIdentifierType>
                        FDsys:agent
                </premis:agentIdentifierType>
        </premis:agentIdentifier>

        <premis:agentName>
                http://metaarchive.org/agent/metaMigrateSuccess
        </premis:agentName>
        <premis:agentType>
                softw
        </premis:agentType>
</premis:agent>
```

As you can see from the above example, the agent's identifier above corresponds with the agent in the event example. You are able to create and register agents through the administrative panel on the PREMIS microservice.

**AtomPub and REST**
The microservice exists as a web application, and uses REST to handle the message passing between client and server. To better provide a standard set of conventions for this, we have elected to follow the AtomPub protocol for POSTing and GETing events from the system. The base unit for AtomPub is the Atom "entry" tag, which is what gets sent back and forth. The actual PREMIS metadata is embedded in the entry's "content" tag.

There is a lot more to AtomPub than that, but for the purpose of this document, it is helpful to just view the Atom entry as an "envelope" for the PREMIS XML.

**Example Use Case**
Imagine that we have just completed a mass server-to-server data copy, and as part of that migrated data we have a directory called object_123/ which contains a collection of files that represents a migrated digital object. This digital object conveniently enough, has the local

identifier (for our system) of "object_123." We have a script "validate_object" that we can run on our objects to make certain that the files match a previously stored fixity digest and are intact after this migration. In this case, we wish to log an event of the validation in order to properly track our actions.

Our microservice lives at the URL http://metaarchive.org/tools/event.

To begin with, we run the validate_object script on our directory and wait for it to run. For the sake of argument, let's say that it runs and comes back with an error:

> Validation of object_123/ failed
> Details:
> Generated sum for object_123/data/pic_002.tif does not match stored value

Obviously, we have to deal with the problem at some point, but right now we just want to log an event that will accurately reflect the results of the script. So, we create a PREMIS event XML tree

```
<premis:event xmlns:premis="info:lc/xmlns/premis-v2">
        <premis:eventType>
                validateObject
        </premis:eventType>
        <premis:linkingAgentIdentifier>
                <premis:linkingAgentIdentifierValue>
                        validateObjectScript
                </premis:linkingAgentIdentifierValue>
                <premis:linkingAgentIdentifierType>
                        Program
                </premis:linkingAgentIdentifierType>
        </premis:linkingAgentIdentifier>
        <premis:eventIdentifier>
                <premis:eventIdentifierType>
                        TEMP
                </premis:eventIdentifierType>
                <premis:eventIdentifierValue>
                        TEMP
                </premis:eventIdentifierValue>
        </premis:eventIdentifier>
        <premis:eventDetail>
                Validation of object object_123
        </premis:eventDetail>
        <premis:eventOutcomeInformation>
                <premis:eventOutcomeDetail>
                        Generated sum for object_123/data/pic_002.tif does not match stored value
                </premis:eventOutcomeDetail>
                <premis:eventOutcome>
                        Failure
                </premis:eventOutcome>
        </premis:eventOutcomeInformation>
        <premis:eventDateTime>
                2011-01-27 16:39:49
```

```
        </premis:eventDateTime>
        <premis:linkingObjectIdentifier>
                <premis:linkingObjectIdentifierType>
                        Local Identifier
                </premis:linkingObjectIdentifierType>
                <premis:linkingObjectIdentifierValue>
                        object_123
                </premis:linkingObjectIdentifierValue>
        <premis:linkingObjectRole/>
        </premis:linkingObjectIdentifier>
</premis:event>
```

As you can see, the values chosen for the tags in the PREMIS event XML are arbitrary, and it is the responsibility of the user to select something that makes sense in the context of their organization. One thing to note is that the values for the eventIdentifierType and eventIdentifierValue will be overwritten, because the microservice manages the event identifiers, and assigns new ones upon ingest.

Now, in order to send the event to the microservice, it must be "folded" into an Atom entry, so the following Atom wrapper XML tree is created.

```
<entry xmlns="http://www.w3.org/2005/Atom">
  <title>PREMIS event entry for object_123</title>
  <id>PREMIS event entry for object_123</id>
  <updated>2011-01-27T16:40:30Z</updated>
  <author>
       <name>Object Verification Script</name>
  </author>
  <content type="application/xml">
       <premis:event xmlns:premis="http://www.loc.gov/standards/premis/v1">
              ...
       </premis:event>
  </content>
</entry>
```

With the already-generated PREMIS XML going inside of the "content" tag.

With the entry generated, it is finally ready for upload.

In order to transmit the event, we POST the XML of the Atom entry to the event URL, which is http://metaarchive.org/tools/event/

When the microservice receives the POST, it reads the content and parses the XML. If it finds a valid XML PREMIS event document, it will assign the event an identifier, index the values and save them, and then generate a return document, also wrapped in an Atom entry. It will look something like:

```
<entry xmlns="http://www.w3.org/2005/Atom">
```

```
 <title>bfa2cf2c2a4f11e089b3005056935974</title>
 <id>bfa2cf2c2a4f11e089b3-005056935974</id>
 <updated>2011-01-27T16:40:30Z</updated>
 <author>
        <name>Object Verification Script</name>
 </author>
 <content type="application/xml">
        <premis:event xmlns:premis="http://www.loc.gov/standards/premis/v1">
                <premis:eventType>
                        validateObject
                </premis:eventType>
                <premis:linkingAgentIdentifier>
                        <premis:linkingAgentIdentifierValue>
                                validateObjectScript
                        </premis:linkingAgentIdentifierValue>
                        <premis:linkingAgentIdentifierType>
                                Program
                        </premis:linkingAgentIdentifierType>
                </premis:linkingAgentIdentifier>
                <premis:eventIdentifier>
                        <premis:eventIdentifierType>
                                UUID
                        </premis:eventIdentifierType>
                        <premis:eventIdentifierValue>
                                bfa2cf2c2a4f11e089b3-005056935974
                        </premis:eventIdentifierValue>
                </premis:eventIdentifier>
                ...
        </premis:event>
 </content>
</entry>
```

As you can see, the identifier has been changed to a UUID, which, in this case, is "`bfa2cf2c2a4f11e089b3-005056935974`". This identifier is unique and will be what the microservice will use to refer to that individual event in the future.

If the POST is successful, the updated record will be returned, along with a status of "200". If the status is something else, there was an error, and the event cannot be considered to be reliably recorded.

Later, when we (or, perhaps, another script) wish to review the event to find out what went wrong with the file validation, we would access it by sending an HTTP 'GET' request to

http://metaarchive.org/tools/event/bfa2cf2c2a4f11e089b3-005056935974

This would return an Atom entry containing the final event record, which we could then analyze and use for whatever purposes desired.

**AtomPub & Human-Readable URLs**

The following is a brief overview of the URL structure for the PREMIS Event Microservice, with some use cases for how it might be utilized.

There are two basic subgroups for the URLs. There is a URL hierarchy that provides AtomPub functionality, and there is a URL hierarchy that provides a more "human friendly" view into the contained data.

I'll list the URLs in a relative mode, which means that they're based on whatever the URL that the service ends up living on decides to be. So if the relative URL is "/APP" and the service lives at http://metarchive.org/services/premis_event, the full URL is http://metarchive.org/services/premis_event/APP/

I'll start with the AtomPub URLs, since this is where that which is of most interest to developers happens.

/APP/
-- AtomPub service document
The service document is an XML file that explains, to an AtomPub aware client, what services and URLs exist at this site. It's an integral part of the AtomPub specification, and allows for things like auto-discovery and the like.

/APP/event/
-- AtomPub feed for event entries
---- Accepts parameters:

- start - This is the index of the first record that you want...it starts indexing at 1.
- count - This is the number of records that you want returned.
- start_date - This is a date (or partial date ) in ISO8601 format that indicates the earliest record that you want.
- end_date - This is a date that indicates the latest record that you want.
- type - This is a string identifying a type identifier (or partial identifier) that you want to filter events by
- outcome - This is a string identifying an outcome identifier (partial matching is supported)
- link_object_id - This is an identifier that specifies that we want events pertaining to a particular object
- orderdir - This defaults to 'ascending'. Specifying 'descending' will return the records in reverse order.
- orderby - This parameter specifies what field to order the records by. The valid fields are currently: event_date_time (default), event_identifier, event_type, event_outcome
-

--- For the human-viewable feeds, the parameters are the same, except, instead of using a 'start' parameter, it uses a 'page' parameter, because of the way it paginates the output (see below).
--- Also POST point for new entries
--Issuing a 'GET' to this URL will return an Atom feed of entries that represent PREMIS events. This is the basic form of aggregation that AtomPub uses. Built into the Atom feed are tags that

allow for easy pagination, so crawlers will be able to process received data in manageable chunks. Additionally, this URL will accept a number of GET arguments, in order to filter the results that are returned.
--This is also the endpoint for adding new events to the system, in which case a PREMIS Event is sent within an Atom entry in the form of an HTTP POST request.

/APP/event/<id>/
-- Permalink for Atom entry for a given event
--This is the authoritative link for a given PREMIS Event entry, based upon the unique identifier that each event is assigned when it is logged into the system. It returns the event record contained within an Atom entry.

/APP/agent/
-- AtomPub feed for agent entries
-- Issuing a 'GET' request here returns an AtomPub feed of PREMIS Agent records. Because there will be far less agents than events in a given system, it is not known that we'll build search logic into this URL.
-- According to the AtomPub spec, this would be where we'd allow adding new Agents via POST, but because there are likely so few times that we'd need to add Agents, we would just as well leave this to be done through the admin interface.

/APP/agent/<id>/
-- Permalink for Atom entry for a given agent
--The authoritative link for a given PREMIS Agent entry, based on the agent's unique id.

Next are the URLs designed for human consumption.

/
-- Front page for the microservice
-- This URL will yield a basic information page about the microservice. Perhaps a small graphic and an explanation of what the microservice is/does. There would be contact info (settable through the administrative interface), plus possibly basic statistics (number of events contained, number of unique linked objects, date of last event, etc).

/event/
-- Human readable (HTML) listing of events (paginated)
-- Much like the AtomPub analogue of this URL returns a feed, this would return a "Google-esque" view of results for stored events. It would accept the same search parameters as the AtomPub version. The results page would also include arrows and clickable links to easily navigate through the results. Clicking on a given result would take you to the detailed human view for a given event.

/event/<id>/
-- Human readable (HTML) representation of an event entry

---- Links to different formats, including Atom representation
-- This would be the detailed human-readable view of a given event. It would list all of the details for that event in a neat, tabular format. In addition, it would provide links to view the event in original XML format, and possibly others (JSON?).

/agent/
-- Human readable (HTML) listing of agents (paginated)
-- Like the human readable event listing, this would be a search-engine-like listing of the agents in the system, clickable to get a detailed view.

/agent/<id>/
-- Human readable (HTML) representation of an agent entry
---- Links to different formats, including Atom representation
-- This would be the detailed, human-readable view of a given agent, with links to other formats.

Finally, there's the administrative interface for the microservice:

/admin
-- Administrative interface
-- This would take you to a password protected login where you could view and change the internals of the microservice. It would provide an internal view for all of the saved events and agents, where they could be added, deleted or modified. Additionally, you'd be able to set values for the microservice, like contact info for the front page, and possibly other settings that might be useful.

**Use Case**

Let's imagine that I have a data migration process running. In this process, I have a number of objects that are copied and then checked against a master record for integrity. If they pass the test, an event is logged as passing, and they are assigned to their final destinations. If they fail the test, an event is created indicating which files failed, and they are moved to a holding area.

So, imagine that I have a utility that goes through the failed files and attempts to repair them by grabbing files that failed. Let's see how that might work. First it crawls the directory of failed objects, getting the object name of each one. Then, once it has that, it issues a call to the PREMIS event service, asking for all of the events related to that object. So it might send a GET to the url http://metaarchive.org/service_root/APP/event/?link_object_id=008-abc. In return, I'd get an Atom feed containing every event that was logged pertaining to that record.

However, if I wanted to streamline things, and knew a little more about the process, I could make the microservice do a little more work. For example, if I knew that the events pertaining to integrity check were all of the type "integrity_check", then I could send my request to http://metaarchive.org/service_root/ APP/event/?link_object_id=008-abc&event_type=integrity_check. And if I knew that I was only interested in the ones that didn't pass, I could further narrow it with

http://metaarchive.org/service_root/APP/event/?link_object_id=008-abc&event_type=integrity_check&event_outcome=failure. And, since I'm only interested in the most recent, I could issue http://metaarchive.org/service_root/APP/event/?link_object_id=008-abc&event_type=integrity_ check&event_outcome=failure&sort_by=event_date. That would put it in chronological order, so I could just use the first result in the feed.

Once I'd gotten the feed entry with the identifier that I needed, I'd grab the authoritative record (feed entries should not be considered authoritative, according to AtomPub specs) at http://metaarchive.org/service_root/APP/event/<particlar_event_identifier> and parse the XML and use the data therein to discover which files I needed to re-copy before attempting to re-verify (and possibly finish the storage of) the object. And, of course, I'd generate new PREMIS events for any new verification attempts that I did.

**Installing the PREMIS Events Microservice (PEM)**

The following instructions were detailed by the University of North Texas and used to test install the PEM package from a copy of software packages made available on the MetaArchive's development server. The test installation was successful with minor incompatibilities between Python version 2.4 and the operating system running on the development server. Future installations will aim to avoid these incompatibilities.

**Requirements:**

(It is assumed that these are installed before beginning)
Apache 2.x server
Mod_python module for Apache
Python 2.4 or newer
MySQL

**Required Python Modules:**

lxml - This is an XML parsing module that the PEM uses heavily. It is possible that you can install this via yum. If this doesn't work, you can install it from the source package here: http://lxml.de/index.html#download

MySQLdb - This is the standard Python module to interface with MySQL. It is included by default with some platforms, (i.e., Ubuntu), but on others, it is not standard. It needs to be installed /before/ the Django installation, so if you don't have it, you can install it with python's "easy-install.py" tool, or get the source package and install that way.

simplejson - The standard package for working with JSON files in Python. If not installed, you can install it with the easy-install.py tool.

httplib2 - This is a module for dealing with HTTP connections. It isn't part of the standard Python install, but easy-install.py should be able to install it. If not, the source package can be obtained from http://code.google.com/p/httplib2/

Django - This is the web framework that we use for the PEM. As of writing, the latest release is 1.3, which should work fine for our purposes. It is obtainable here: http://www.djangoproject.com/download/ (Follow option 1 for download and install)

**Installation Instructions:**

- Unpack the PEM source tarball: There should be two directories called "coda" and the other called "premis_event". Store these directories in your desired location.

- Put the directories on the Python path: You want Python to be able to access coda and premis_event as modules, so you'll need to make certain that they're on your path. To do this, look inside /usr/lib/python<version_number>/. Depending on your version of python there will either be a directory called "site-packages", or "dist-packages". Create symlinks into this directory for both the coda and premis_event directories.

- Create a MySQL database for your Django project: Use MySQL, and whatever administrative tool you prefer to create a new database with a user and password for that database. Make sure that the user has full rights to the newly created database.

- Create a new Django project: When you installed Django, it should have installed a command line utility, django-admin.py. Decide on where you want the project to live, and a name for it, and type: django-admin.py startproject <projectname>.

- Unpack the templates tarball. This should contain a directory that reads 'templates' and should be moved into the root of your newly created project directory.

- Get the agent_type_choices.json file and save it somewhere. Make certain that the file is readable by the UID for your web server.

- Create a directory to serve as a cache for unpacking Python eggs on the fly. Make certain that the path is readable and writeable by your web server UID.

- Change into the newly created project directory and edit the settings.py file. Find the 'DATABASES' section, and fill in the values with those that chose when you created the database for the project. The ENGINE value should be 'django.db.backends.mysql'.

In the bottom of the settings.py file, you'll find the section for INSTALLED_APPS. Uncomment the the line for the django.contrib.admin. Also, add in another line 'premis_event', which will add the premis_event app to the project.

In the section of the settings file for TEMPLATE_DIRS, add the path to which you've installed the template directory to the list. This needs to be properly formatted as a string within a Python list, so take care to do it properly.

At the end of the settings file, create a line that looks like:

AGENT_TYPE_CHOICE_FILE_PATH = "/path/to/directory/agent_type_choices.json"

Edit the directory to reflect the absolute path for where you saved the JSON file in the previous step.

Save changes to the settings file and close it.

Edit the urls.py file. Uncomment the line at the top that reads 'from django.contrib import admin', and uncomment the line that reads: (r'^admin/(.*)', admin.site.root),. Additionally, you need to add this line to the urlpatterns section: url(r'', include('premis_event.urls')). Save your changes and close the urls.py file.

- Initialize the MySQL database with the Django tables: Change into the project directory. You'll see a file called 'manage.py'. Type: 'python manage.py syncdb'. This will do the initial setup of the databases. It will also prompt you to create a superuser name and password, which you should do. You'll use this to login to the administration panel later.

- Configure Apache to host the Django App: This might vary depending on your setup, but you'll want to create a virtualhost entry for the port that you plan to be running the Django framework on. The important section that you'll put in the header looks like this:

SetHandler python-program
PythonHandler django.core.handlers.modpython
PythonDebug On
SetEnv DJANGO_SETTINGS_MODULE name_of_my_project.settings
SetEnv PYTHON_EGG_CACHE /path/to/your/egg_cache/directory
PythonPath "['/path/to/my/project'] + sys.path"

Not that name_of_my_project should correspond to the project that you created previously, and /path/to/my/project should be the absolute path to where the project resides.

/path/to/your/egg_cache/directory should correspond to the Python egg cache directory that you created previously.

If all goes well, you should be able to restart Apache and then point your web browser to the host and the port at which you configured your virtual host at.

**Notes on this document**

This document was drafted by Kurt Nordstrom (UNT) and Matt Schultz(MetaArchive) in 2011-09 and edited in 2012-03